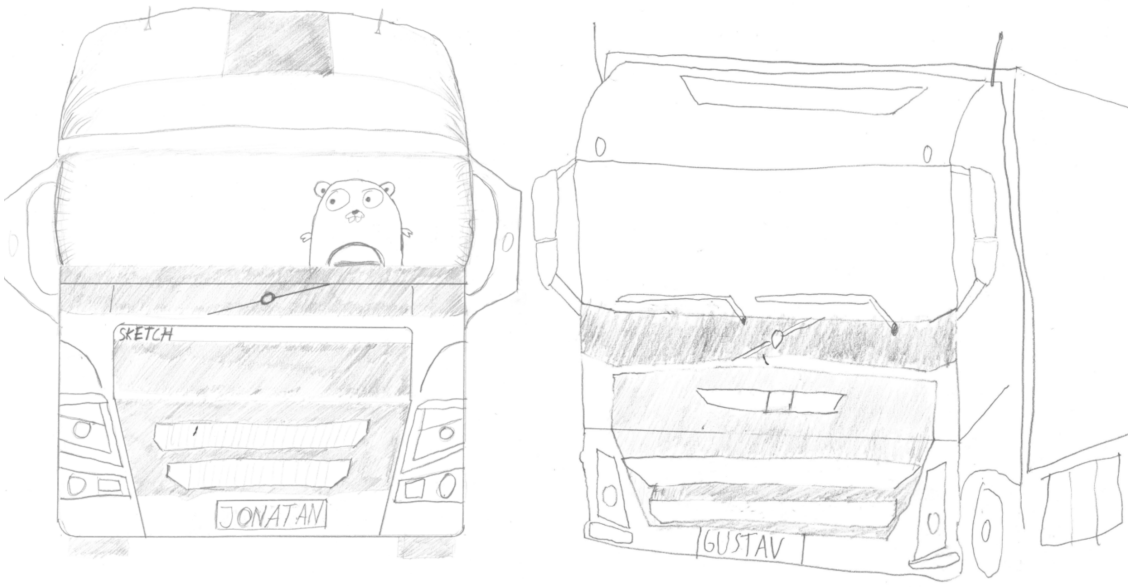
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Distributed Sketching Pipelines for Data Mining and Analytics

Master's thesis in Computer science and engineering

Gustav Bruhn

Jonatan Mentzer

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Distributed Sketching Pipelines for Data Mining and Analytics

Gustav Bruhn

Jonatan Mentzer

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Distributed Sketching Pipelines for Data Mining and Analytics
Gustav Bruhn, Jonatan Mentzer

Supervisor: Martin Hilgendorf
Advisor: Carl-Magnus Wall, Volvo Trucks
Examiner: Marina Papatriantafilou, Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Two "Sketched" trucks.

Typeset in LaTeX
Gothenburg, Sweden 2026

Distributed Sketching Pipelines for Data Mining and Analytics
Gustav Bruhn, Jonatan Mentzer
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The rapid growth of data generated by modern industrial systems at the edge poses significant challenges for efficient data management. Processing the data directly on the edge device offers benefits, including improved throughput, scalability, and reduced bandwidth usage. Another effective approach is summarizing the data using sketches. Sketches are stochastic data structures that can greatly compress data while preserving essential statistical properties. This thesis investigates the conditions under which it is beneficial to offload sketching computations to edge devices. The study evaluates the throughput and latency of two systems across multiple configurations, designed to reflect real world scenarios, comparing a federated architecture with a centralized architecture. The results indicate that, across all evaluated scenarios, executing computations at the edge increases the maximum throughput, especially when the number of edge devices increases. The thesis explores the trade-offs between scalability (in form of throughput with increasing set of vehicles) and data freshness (in form of latency due to the micro-batch sizes, i.e. the frequency of data summarization).

Keywords: Computer Science, Sketches, Distributed Systems, Big Data, Federated Computation, Federated Sketching, Federated Analytics.

Acknowledgements

We would like to thank our supervisor Martin Hilgendorf and examiner Marina Papatriantafilou for always making time for us, even through their busiest periods, and their continued support during our thesis. We would also like to thank Carl-Magnus Wall for his help from Volvo's perspective. Thanks to our thesis cohort who have been a great sounding board throughout our work. Especially our opponents, Erik Berg and Jakob Henriksson for their input on our thesis, and some much needed morale boosts at Volvo and on Teams. And finally, we would like to extend our gratitude to our families for their continued support.

Gustav Bruhn and Jonatan Mentzer, Gothenburg, 2026-02-10

Contents

List of Figures

List of Tables

1

Introduction

Modern industries produce Zettabytes of data yearly [1]. This data is mostly gathered on edge devices, such as vehicles and phones, and sent to a server where it is expected to be processed. The data is sent continuously or in large chunks from the edge devices, making it impractical for the server to process all the data in real time and potentially requiring large bandwidth [2].

Organizations collect data for a wide range of purposes, including to observe long-term trends or to spot anomalies that could be detrimental to their processes. Analytical objectives include estimating the number of unique visitors to a website or determining the most popular products [3]. This becomes particularly critical in contexts where stringent legal requirements mandate the collection of data on emissions and various other operational parameters. The process of finding useful information in the large amount of data collected is called *data mining* [2].

A solution to processing the high volumes of data is to use multiple machines, a *distributed system* [4]. In a distributed system, the machines must be able to communicate to facilitate data exchange and coordination of computational tasks. However, the communication inherent in distributed architectures often results in bandwidth becoming a critical bottleneck [5].

Federated computation is a specialized form of distributed systems, pushing computation to the data generating edge devices themselves without sharing raw data. This reduces the volume of data sent and helps preserve user privacy by keeping unprocessed data local [6]. But this approach hinges on the ability to effectively process data locally in a way that can be merged meaningfully at the server.

Sketches are data structures which summarize data and are used as a means of representing large amounts of data in a compact way [5]. A design philosophy of sketches is to trade some accuracy in exchange for a low memory footprint and operational performance, particularly fast updates and low-latency queries. By retaining only certain properties of the original data, they significantly reduce the amount of information that is stored. This enables efficient approximate analysis of large-scale data sets. Sketches can be used to estimate statistics such as cardinality, frequency, and quantiles, supporting tasks like anomaly detection, trend analysis, and resource optimization. Their ability to reduce data volume while retaining analytical utility makes them well-suited for environments where bandwidth, storage, or processing power is limited. In contrast to lossless data compression, which uses decompression

to recover all the data, sketches provide no method of recovering the original data. Therefore, a summarization algorithm tailored to the specific problem is essential for preserving relevant data. Several sketches are designed for a distributed environment and can therefore be merged, enabling partial summaries from the nodes to be efficiently combined at the server [7], [8].

Sketches and federated computation complement each other well, particularly in settings where data is naturally decentralized across numerous edge devices [6]. By summarizing data locally using sketches, devices can transmit compact representations instead of unprocessed data, reducing communication costs and central processing load. This enhances both performance and scalability. Furthermore, because only partial, non-invertible information is shared, and unprocessed data remains on-device, this approach supports data minimization, a principle aligned with privacy preservation [9].

1.1 Problem

Distributed pipelines for data mining face significant scalability challenges [10]. For instance, vehicular data is generated in large volumes by various sensors, transmitting and storing of all this data centrally is impractical and expensive [11]. Typically, edge devices collect sensor data streams and send them to a main server, which can congest the network and overwhelm the server. As a result, data is stored without adequate processing, which creates major problems for future analysis, as accessing and querying years of accumulated data becomes inefficient. Stream processing, either at the edge or at the server, can alleviate these issues by reducing storage overhead, improving query responsiveness, and enabling timely insights. This thesis focuses on the trade-offs between edge- and server-side stream processing, with sketching techniques as a lightweight yet effective approach to large-scale analysis and mining.

To evaluate the tradeoff between edge and server processing, this work examines how sketching at edge devices compares to sketching at a central server in terms of throughput and latency. To facilitate this evaluation, two proof-of-concept systems are developed. The first system represents a centralized architecture, in which streaming data is transmitted directly to a central server where sketching is performed. The second system implements a federated approach, where data is first sketched locally at the edge devices and the resulting sketches are subsequently merged at the central server.

The proposed system architecture is shown in Figure 1.1. Edge devices continuously ingest streaming data and are connected to a central server. The edge devices transmit data gathered from the data streams to a server, which is responsible for merging and processing the data for analysis. The processing should derive a reduced data representation, a summary of the original data. This architecture intends to enable data analysts to perform low-latency queries.

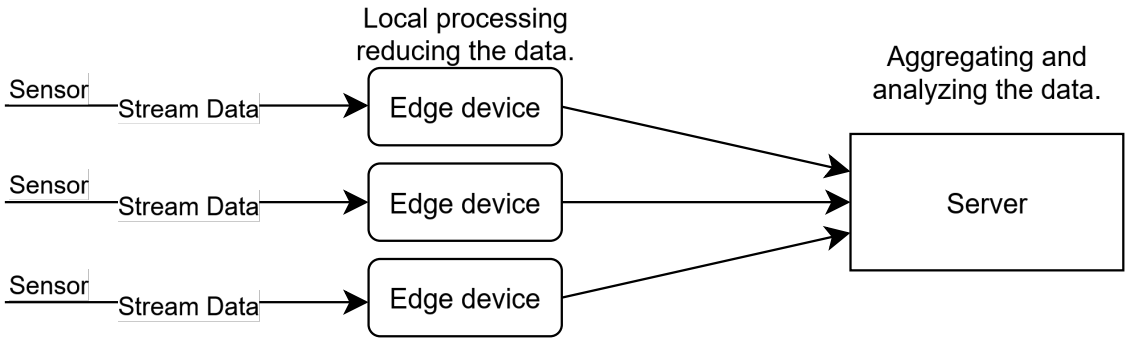


Figure 1.1: The proposed system architecture

1.2 Aim

The aim of this thesis is to research the throughput and latency differences of two proof-of-concept distributed data pipelines in order to answer the questions "When does a federated approach to sketching outperform a centralized approach for distributed systems?", "When is it feasible to offload computation to the edge?", and "What are the trade-offs between throughput, data safety, and data freshness at the server?". The data pipelines follow the model given in Figure 1.1 and handle common network errors. In one approach, edge devices send unprocessed data to a server for processing, whereas in the other approach, edge devices perform part of the computation and send the result to the server for further processing.

Use-cases for this are industries that produce large amounts of data at edge devices in a distributed fashion. The main use case for this thesis comes from Volvo Trucks, whose trucks collect data from a multitude of sensors, up to 10 terabytes of data per truck daily. Sketching the collected data and performing computations on the trucks can help reduce the bandwidth needed, reduce server and storage loads and in turn, streamline the workflow of data analysts by reducing query times. Other example use-cases are retailers and content delivery networks (CDNs). Sketched shelf stock levels could be used for large retail companies to coordinate large orders. CDNs distribute digital content to servers located closer to end-users to reduce latency and network load [12]. By using sketches on local servers, CDNs can efficiently estimate the popularity of different content items. This enables them to cache frequently accessed content near where it is consumed, reducing bandwidth usage and improving user experience. Aggregating these local popularity estimates at a central level can then reveal global trends, which in turn can guide creators and providers in deciding what content to produce or promote.

The system is evaluated on its *throughput* and *latency* using a dedicated testbed, described in subsection 4.1.1. These criteria are used to evaluate the data pipelines, the edge devices, and the servers. Throughput quantifies the amount of data processed per unit of time, whereas latency characterizes the delay incurred during data processing or transmission. Collectively, these metrics serve to evaluate the efficiency and computational demands of the system under defined workload conditions. The results are presented as a comparison between the two different data pipelines. Ad-

ditionally, the potential trade-off in the frequency at which edge devices transmit data to the server is examined.

1.3 Scope

This thesis explores a system architecture with a central server and multiple edge devices, shown in Figure 1.1. Although alternative architectures, such as a tiered server system, may offer additional benefits in terms of throughput and scalability, they are not evaluated in this thesis due to the hardware limitations of the testbed. Nonetheless, the insights derived from this architecture are expected to apply to other architectures, since distributing the work should have similar effects regardless of the architecture.

2

Background

This section provides context for relevant topics for this thesis, namely data streams, distributed systems, and sketches.

2.1 Data streams

A data stream is a continuous series of data, often exemplified by sensor outputs or system log files [13]. A single entry in a data stream can be seen as a set of data points. For instance, a typical data point from a truck equipped with multiple sensors might be represented as a tuple (x_1, x_2, x_3) where x_1 is the time of the measurement, x_2 is the forward acceleration from an accelerometer, and x_3 is the velocity from a speedometer. Usually, a stream has a fixed *stream rate* which dictates the frequency at which data is inserted into the stream. The stream rate varies for different tasks, where sensors may measure multiple times per second and logs may be reported every minute. Data mining is searching for knowledge or patterns in large data sets [14]. The practice of mining can also be applied to data streams.

2.2 Distributed systems

A distributed system is a network of nodes (computers) that collaborate to solve a problem. Each node has its own authority, but they collaborate by exchanging data and messages with each other [4]. The advantages of distributing work over multiple machines are improved performance, better scalability, and reduction of the risk of failure by redundancy. Distributed systems can be designed using various architectural paradigms, with the client-server architecture being a prevalent model. In this configuration, servers orchestrate interactions with clients (edge devices), as illustrated in Figure 1.1. Clients initiate communication by sending requests to the server, which subsequently responds with the requested data. This interaction pattern is commonly referred to as request-reply behavior. To mitigate the complexities associated with distributed system interactions, a widely adopted abstraction is the use of *remote procedure calls* (RPC) [15]. RPC mechanisms encapsulate remote invocations, presenting them as local function calls to enhance system transparency and simplicity.

Distributed systems enhance scalability and fault tolerance through redundancy, but they also introduce notable disadvantages. The incorporation of additional

computers increases the likelihood of hardware failures, and network communication introduces further sources of error. According to Van Steen and Tanenbaum, five primary issues can arise when utilizing RPC in distributed systems, contributing to the complexity and potential unreliability of such configurations [4].

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending the request.

The following can be done to mitigate these issues.

Client can not locate the server: A method to alleviate this issue is to raise an exception on the client [4]. However, this undermines the transparency that remote calls are equivalent to local ones since this exception needs to be handled.

Clients request is lost: Using a son the client side and resubmitting the request if no acknowledgment is received within a reasonable time frame can mitigate this issue [4]. The problems with this are that if too many messages are lost, the client may mistakenly assume the server is unavailable. It is also indistinguishable from some of the following cases.

Server crash: There are two types of server crashes that need to be considered [4]. Either the server completed the execution but crashed before sending a reply, or the server crashed before the execution could finish. They cannot be distinguished, and there are three schools of thought on how to handle this error according to Van Steen and Tanenbaum.

The first approach is to wait until the server has rebooted and then retry the request repeatedly until the client receives a reply. This method aims to satisfy the condition known as *at-least-once semantics*, which guarantees that the RPC call is carried out at least once, though possibly more times. However, this approach may not fully guarantee the condition in all scenarios due to potential failures.

The second approach is to immediately give up and report failure if no acknowledgment is received. This corresponds to *at-most-once semantics*, which guarantees that the computation is performed at most once, possibly not at all, thereby avoiding duplicate execution but risking no execution.

The third and simplest approach is to provide no delivery guarantees at all, prioritizing ease of implementation. While this method avoids the complexity of ensuring delivery, it does not address potential duplicate or missing executions. Ideally, the goal would be to achieve *exactly-once semantics*, which guarantees that each call is executed precisely one time. However, implementing exactly-once semantics is generally impossible in practice due to failure scenarios. For example, if the server crashes and later notifies the client that it has recovered, the client cannot determine

whether the request completed before the crash, resulting in uncertainty about the operations outcome.

Lost Replies: The problem is that the client cannot know why the reply was lost: if the server is unavailable or there is a network issue. An obvious approach to this is to use the same strategy as if the request did not reach the server [4]. If the request was idempotent, meaning it can be repeated multiple times without changing the final outcome, then resending it after recovery would not cause any issues. For example, a request to retrieve the first 1024 bytes of a file can be safely repeated without unintended side effects. However, if the request was nonidempotent, such as a banking transaction or warehouse stock updates, retransmitting it after a crash could lead to duplicate operations, like transferring money twice. To prevent this, the client could assign a unique sequence number to each request, allowing the server to detect duplicates and avoid re-executing nonidempotent operations.

Client crashes after a request: When a client crashes, *orphan computations*, ongoing remote operations that no longer have an active client to receive their results can continue consuming resources and potentially cause inconsistencies [4]. One mitigation is *orphan extermination*, where all RPCs are logged, and any orphaned computations are forcefully terminated after a client reboot. However, this approach is both expensive and complex to implement. Another method is *reincarnation*, which divides time into epochs; when a client restarts, all remote computations associated with the previous epoch are terminated. A more refined approach, *gentle reincarnation*, attempts to locate the owners of computations before deciding whether to terminate them. Alternatively, *expiration* limits the execution time of RPCs, requiring periodic renewal to ensure that orphaned computations eventually expire on their own. While these strategies focus on eliminating orphans, a more effective approach may be to restore the client to a state where it can properly handle orphaned responses instead of simply terminating them.

2.2.1 Centralized vs. federated computation

In a system employing a centralized approach, minimally processed data from edge devices is transmitted to a single central server, which performs the majority of the computation [6]. In this approach, data is moved to the computation, concentrating processing power at a central point.

Federated systems, by contrast, reverse this flow by moving the computation to the data. Each edge device processes the data it has collected locally [6]. Only these processed summaries are transmitted to a central server, for merging and further analysis. By transmitting summaries rather than raw datasets, federated systems significantly reduce bandwidth requirements and takes better advantage of available compute.

There are two major areas of interest for federated computation *federated learning* (FL) and *federated analytics* (FA) [6]. FL computes gradients for machine learning models at edge devices using local data. A server merges the gradients to evaluate and train the model. FA trains no models but computes statistics and merges

them. FA can be compared to the prominent distributed data processing method *MapReduce* where mappers perform work concurrently and the results are merged by reducers [16]. Edge devices can be seen as mappers, and reducers as merging servers.

2.3 Sketches

Data sketching provides methods to make Big Data small [5]. A *sketch* is a probabilistic data structure that maps an input dataset or stream to a compact summary of fixed or sublinear size, allowing approximate answers to queries with provable accuracy and confidence guarantees. The accuracy is often expressed using two parameters: ε (the error guarantee) and δ (the probability of the error being larger than the guaranteed error ε). Sketches are space-efficient and often process data in a single pass making them suitable for streaming and large-scale analytics. Mergeability is a characteristic exhibited by certain sketches, which allows them to be computed independently at different locations and subsequently combined into a single sketch. Because they do not retain the original data, sketches can only answer queries about the properties they were designed to estimate, such as cardinality, frequency distributions, quantiles, set similarity, or heavy hitters. Most sketching algorithms achieve this efficiency through randomized methods such as hashing.

Sketches usually allow for the following operations.

Add(tuple) Add allows for reading a tuple form a data stream and adding it to the sketch.

Query(data) Query allows for fetching the data stored in the sketch to gain insights about the data set.

Merge(sketch1, sketch2) The merge operation is only supported by some sketches. Merge allows for two sketches to be combined into one, representing the union of their underlying data as one sketch. This enables clients to merge their local sketch to the server's global sketch.

Remove(tuple) The remove operation is only supported by some sketches. Remove allows for removal of a tuple from the sketch.

2.3.1 Bloom filter

In 1970, Bloom B. published the paper "Space/time trade-offs in hash coding with allowable errors," which introduced the first ever data sketch, later known as the *Bloom filter*. Bloom filters are used to determine if a particular element is part of a set or not [17]. The Bloom filter works by first allocating a bit field of length m and choosing f independent hash functions, f and m are parameters chosen when creating a Bloom filter. The hash functions map elements to the range $[0, m-1]$. The Bloom filter is initialized by setting all m bits in its bit field to 0.

To `add` an element e to a Bloom filter, hash it using all f hash functions and set the bits $j \equiv h_i(e) \pmod{m}$ to 1, see Listing 2.1.

Listing 2.1: The add function for the Bloom filter

```
func (BloomFilter) Add(element)
    for hash in BloomFilter.hashes
        hashResult = hash(element)
        BloomFilter.bits[hashResult] = 1
```

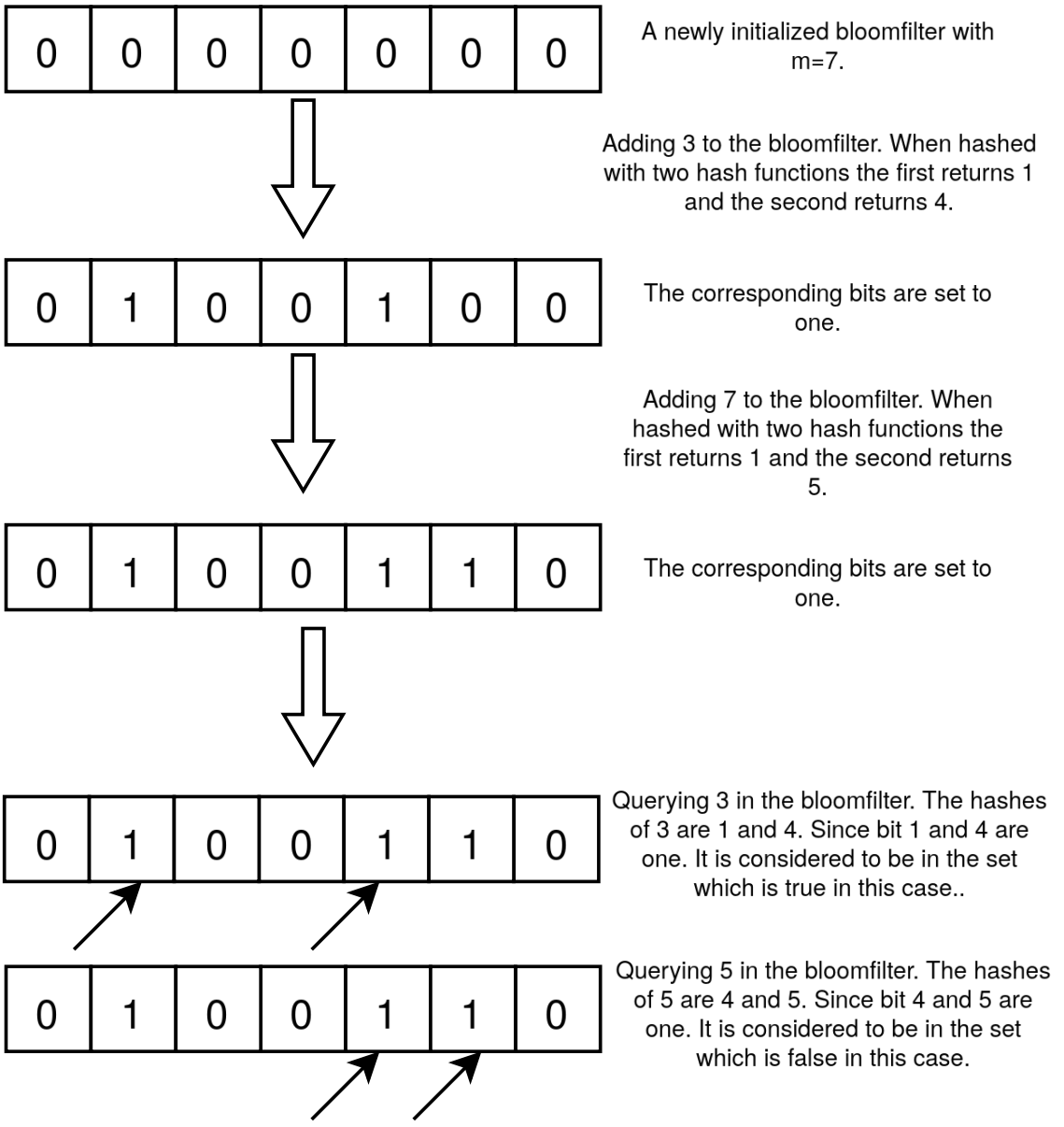


Figure 2.1: A diagram describing two additions and two queries of a Bloom filter with $m = 7$ bits and $f = 2$ hash functions

The `query` operation works by hashing the element, and if all the corresponding bits are 1, it is considered to be part of the set. It might be the case that all the bits happen to be 1 even if the element is not part of the set, a false positive (FP).

Note that a false negative is not possible. Assume n elements have been inserted into the Bloom filter. The probability of a bit being 0 is $(1 - \frac{1}{m})^{fn} \approx e^{\frac{-fn}{m}}$ which gives us Equation 2.1 describing the probability of a false positiv . A diagram of how additions and queries work in a Bloom filter can be seen in Figure 2.1.

$$\Pr[FP] = (1 - (1 - \frac{1}{m})^{fn})^f \approx (1 - e^{\frac{-fn}{m}})^f \tag{2.1}$$

When designing a Bloom filter it is important to pick parameters f and m to satisfy this equation to minimize the expected error. The approximated expression is minimized for:

$$f = \ln(2) \cdot \frac{m}{n} \tag{2.2}$$

If f is chosen using Equation 2.2, then half the bits are 1 and half are 0. To achieve a false positive rate of 1%, 10 bits should be allocated per element, such that $m = 10n$.

To `merge` two Bloom filters, assuming they have the same hash functions and length, a bitwise OR operation should be applied to their bits.

2.3.2 Count sketch

The *Count sketch* is a data structure for estimating the frequency of elements in *multisets*, sets that allow repeating elements [18]. It consists of a $f \times m$ matrix M of counters, f independent hash functions h_i, and a hash function h_s. Similarly to the Bloom filter, the hash functions h map to $[0, m-1]$, but each h_i maps into the ith row in the matrix. The hash function h_s maps elements to $\{1, -1\}$. The Count sketch is initialized by setting all counters to 0, as can be seen in Figure 2.2.

To `add` an element e to the sketch, hash it to get its corresponding counters. Then add $h_s(e)$ to the counters. The mathematical expression is shown in Equation 2.3 and an example can be seen in Figure 2.2.

$$\sum_{i=0}^{f} M[i][h_i(e)] \mathrel{+}= h_s(e) \tag{2.3}$$

To `query` the sketch for the frequency of an element q, the counters at positions given by $\{h_0(q), ..., h_{f-1}(q)\}$ are retrieved and multiplied by $h_s(q)$. The median of these values provides the estimated frequency of q; see Listing 2.2 and Figure 2.2. The hash function h_s ensures that the counters for q are equally likely to overestimate or underestimate its true frequency. Using the median helps mitigate the effect of collisions with high-frequency elements, providing an accurate estimate with low variance. In contrast, using the mean would make the estimate more sensitive to collisions with high-frequency elements, leading to greater variance and error.

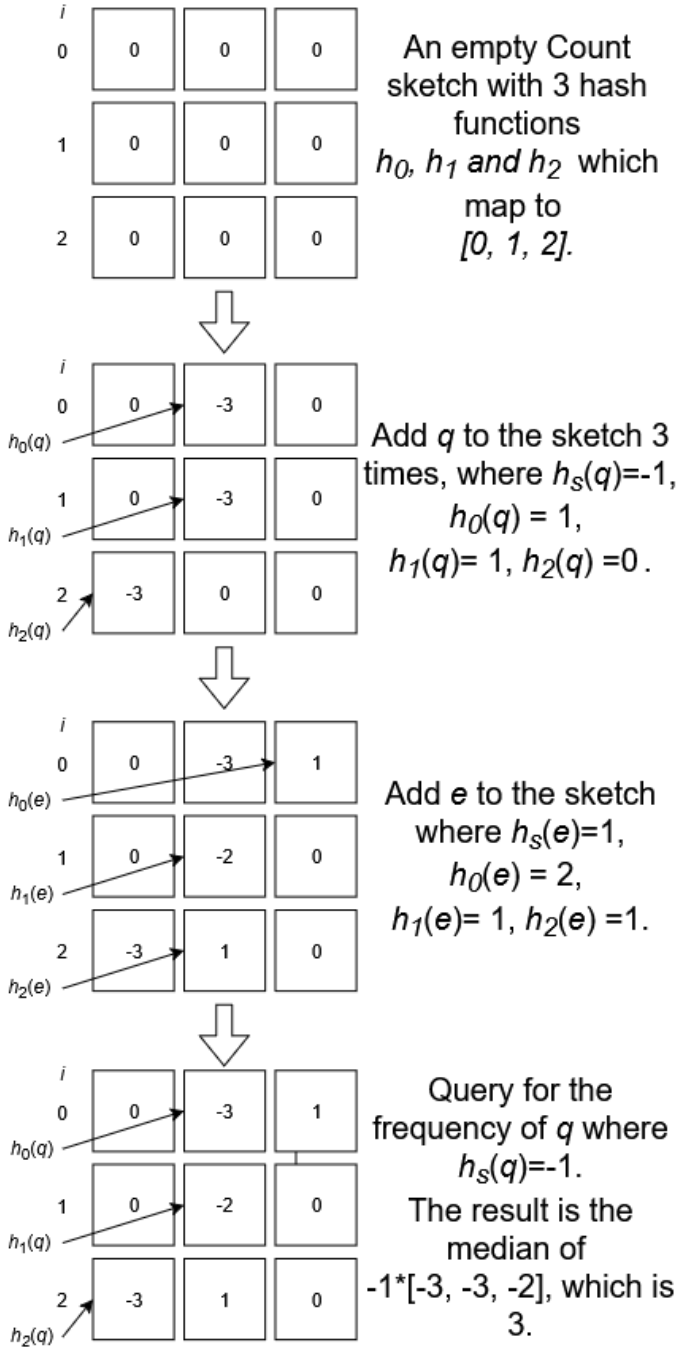


Figure 2.2: A figure showing how the Count sketch with f and m equal to 3 is initialized. Adding element q and e, and querying for q.

For a multiset V with $||V||_1$ total elements, the Count sketch, with parameters $f = \log \frac{1}{\delta}$ and $m = \frac{1}{\varepsilon}$, estimates the frequency of any element with an additive error of at most $\varepsilon||V||_1$, with probability at least $1-\delta$, using space $O\left(\frac{1}{\varepsilon} log \frac{1}{\delta}\right)$ [5]. Achieving smaller additive error or higher probability of correctness requires decreasing ε and δ, respectively, both of which result in increased space complexity.

Assuming that two count sketches have the same hash functions and size, the `merge`

Listing 2.2: The query function for the Count sketch

```
func (CountSketch) Query(element)
    sign = h_s(element)
    vals = []
    for row in CountSketch:
        h = row.hash(element)
        Append(vals, row[h] * sign)
    return Median(vals)
```

operation is done through a matrix addition of the two matrices.

An extension of the Count Sketch data structure can be used to approximate the top-K most frequent elements in a data set [18]. This variant maintains an auxiliary map of size K to track elements with the highest estimated frequencies. As new elements are processed, their frequencies are estimated using the Count Sketch. Each estimate is compared against those in the top-K map. If the estimated frequency of a new element exceeds that of any existing entry, it replaces the one with the lowest frequency. This strategy enables the data structure to maintain an ongoing approximation of the K most frequent elements in the set.

2.3.3 KLL sketch

The Karnin, Lang, Liberty sketch or the *KLL* sketch, is used to calculate an approximation of the *ϕ-quantile* of an ordered data set [8]. The *ϕ-quantile* of a data set refers to a value such that a fraction ϕ of the elements in the set are less than this value, and the remaining fraction $(1 - \phi)$ are greater than or equal to it. More formally, for a multiset V and $0 \leq \phi \leq 1$, the ϕ-quantile is an element e such that $\phi|S|$ elements in V are less than e, and $(1-\phi)|S|$ elements are greater than or equal to e [19].

The KLL sketch maintains a hierarchy of levels, each containing a list of elements with associated weights [8]. When a new element is inserted, it is first placed into the lowest level with weight 1. Each level has a capacity, determined by a user-defined parameter k that controls the accuracy of the sketch. A larger value of k allows the sketch to retain more elements, resulting in improved accuracy. Once a level exceeds its capacity, a compress operation is triggered: the elements in the full list are sorted, and randomly every odd or even-indexed element is discarded. The remaining elements are promoted to the next level with their weights doubled. This process continues if the next level also becomes full. The weight of an element reflects the number of original elements it represents due to this compaction. The sketch also maintains a counter n that tracks the total number of inserted elements. This hierarchical compaction scheme allows the KLL sketch to approximate any ϕ-quantile with an error bound that depends on k.

More specifically, to `add` an element to the sketch, append it to the lowest layer's list and increase n by 1. After each add, the compress function is called. The compress

function checks if any of the layers in the sketch are full, see Figure 2.3. The capacity of a layer can be constant or variable. An example of a variable capacity formula is Equation 2.4, where h is the current layer and H is the current total height.

$$\max\left(2, \lfloor k \cdot \left(\frac{2}{3}\right)^{H-h} \rceil\right) \tag{2.4}$$

In the case that a layer h's capacity is met, the layer is sorted, then it is randomly decided whether the elements at even or odd indices are added to the next layer $h+1$. If h is the highest layer, then layer $h+1$ becomes the new highest layer H. Then all elements from layer h are discarded, see Listing 2.3.

To `query` the sketch for the rank of element q, iterate through all the rows adding up the weight of all elements e, where $e < q$. Then return the rank R. The quantile is $\frac{R}{n}$. The mathematical expression is given in Equation 2.5.

$$R = \sum_{h=0}^{H} \sum_{e<q} 2^h \tag{2.5}$$

Listing 2.3: The compress function for the KLL sketch. H is the total height, h is the current layer and the parameter k impacts the capacity

```
func (KLLSketch) Compress()
    for h in range(KLLSketch.H):
        if CheckCapacity(KLLSketch, h):
            Sort(KLLSketch[h])
            coin = Random([heads, tails])
            if coin == heads:
                Append(KLLSketch[h+1], Even(KLLSketch[h]))
            else:
                Append(KLLSketch[h+1], Odd(KLLSketch[h]))
            KLLSketch[h] = []
    if notNull(KLLSketch[KLLSketch.H+1]) :
        KLLSketch.H++
```

The KLL sketch also allows for the `reverse query` operation, which instead of taking an element and returning its approximate rank, takes a rank and returns the approximate element corresponding to that rank. To determine the element corresponding to a quantile ϕ, iteratively remove the smallest element from the dataset and accumulate its weight into a sum. Continue this process until the sum exceeds $\phi \times n$, see Listing 2.4

To `merge` KLL sketch A with KLL sketch B, sum the n of the two sketches and concatenate each of the layers. Then, call the compress function on B, using B's k value and capacity formula. This means that two sketches with separate accuracies can be merged. The accuracy of the merged sketch is bounded by the lowest accuracy of the merged sketches.

Listing 2.4: The reverse query function for the KLL sketch

```
func (KLLSketch) QueryQuantile(phi)
    for row in KLLSketch.rows:
        Sort(row)
    sum = 0
    min_val = inf
    rank = KLLSketch.N * phi
    for sum < rank:
        min_index = -1
        for i in range(length(KLLSketch.rows)):
            if length(sketch[i]) == 0:
                continue
            if min_index == -1 or sketch[i][0] < min_val:
                min_val = sketch[i][0]
                min_index = i
        if min_index == -1:
            return min_val
        KLLSketch.rows[min_index] = KLLSketch.rows[min_index][1:]
        sum += 2^min_index

    return min_val
```

Using the capacity formula described in Equation 2.4, a KLL sketch can estimate quantiles with an additive error of at most ε with probability at least $1 - \varepsilon$, or alternatively, estimate all quantiles within an error of 2ε [5]. To achieve this, the parameter k must be set to $k = \frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon}}$. Consequently, the total space complexity of the KLL sketch is $O\left(\frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon}} + \log(\varepsilon|S|)\right)$, where the term $O\left(\frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon}}\right)$ corresponds to k, the space required at the highest level of the sketch, and $O\Big(\log(\varepsilon|S|)\Big)$ represents the height of the sketch, the space occupied by the lower levels. To improve accuracy by reducing ε, it is necessary to increase the value of k, which in turn increases the total space requirement.

2.3.4 Properties of the presented sketches

Different properties of the sketches are presented here to highlight them. The Bloom filter and Count sketch utilize hash functions and the KLL sketch randomization. The Bloom filter and Count sketch have a fixed size when initialized, but the KLL sketch grows dynamically. Bloom filters are idempotent, adding the same element multiple times has no effect, whereas Count sketches and KLL sketches are non-idempotent. The requirements for merging two Bloom filters or Count sketches are that they have the same size m and hash functions, whereas the KLL sketch has no such requirements.

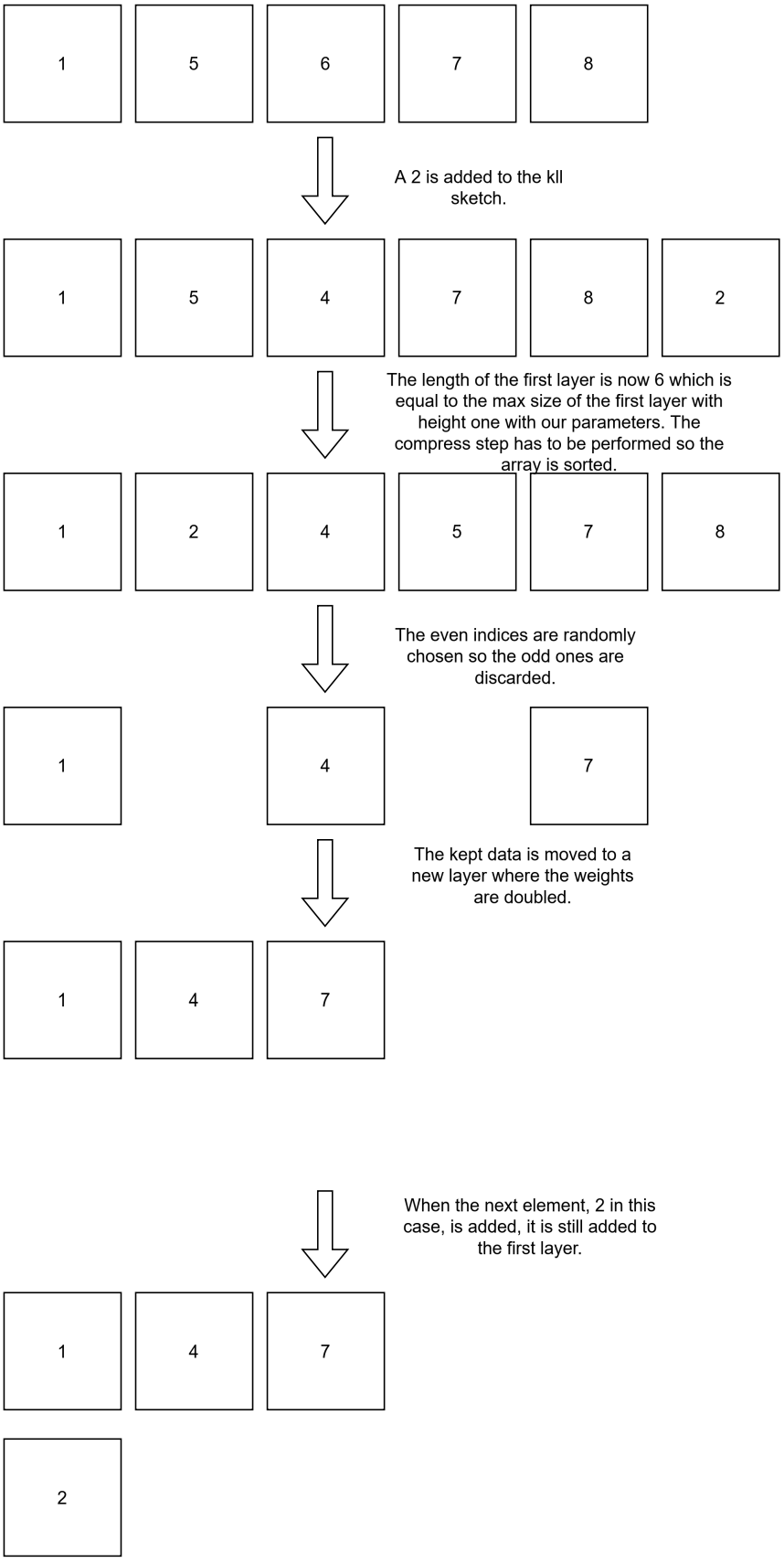


Figure 2.3: An element being added to a KLL sketch with a constant capacity 6.

3

Approach

This chapter details the selected methodology, alongside other evaluated options. It also provides specific implementation details.

3.1 Method

To address the challenges of scalable analytics of large data volumes generated on the edge, this work studies the system architecture illustrated in Figure 1.1. In this architecture, data streams are generated and processed locally on edge devices (clients), and are subsequently transmitted to a centralized server for further analysis. The primary data reduction technique employed is sketching, as detailed in section 2.3. Sketching is selected for its ability to significantly reduce data size while preserving critical statistical properties, including cardinality, frequency distributions, quantiles, and membership information.

A key architectural decision in this context is the placement of the sketching operation. Given that many sketching algorithms support mergeability, two primary design paradigms emerge:

Federated sketching: In this approach, each client independently computes a sketch of its local data stream. These sketches are then transmitted to a central server, where they are merged, taking advantage of the mergeability of the sketch. This design leverages the computational capabilities of the clients to distribute the processing workload and significantly reduces network bandwidth requirements by transmitting compact summaries instead of unprocessed data.

Centralized sketching: The centralized paradigm involves transmitting unprocessed or minimally preprocessed data from the clients to the server, where sketching is performed. This approach enables the server to access unprocessed or minimally processed data but results in increased computational load on the central server and higher data transmission demands from the client to the server, while underutilizing the computational resources available at the client.

To evaluate the trade-offs, relative throughput and latency between centralized and federated processing in stream data summarization, two proof-of-concept systems were designed and implemented. While both conform to the overarching architecture shown in Figure 1.1. The first system, referred to as the *federated system*, employs

the federated sketching approach, highlighting increased resource utilization and decreased bandwidth usage due to compressed data. In the second system, referred to as the *centralized system*, leverages the centralized sketching approach, providing server access to all unprocessed data. However, this design imposes a higher computational load on the central server and requires a higher bandwidth which may induce bottle necks.

To rigorously compare these systems, a comprehensive set of benchmarks was developed. These benchmarks measure throughput at both the system level and across individual components, offering a detailed view of throughput under varying conditions. The key configurable parameters include the *number of clients*, *batch size* (the number of tuples gathered by the client before transmitting its data to the server), the *stream rate* (the number of tuples generated per second), and the *sketch type*. The resulting measurements reveal the system configurations where federated sketching outperforms centralized sketching in terms of throughput, quantify the efficiency gains achievable through federated processing, and provide insights into the scalability of both approaches.

Ultimately, this comparison serves not only to benchmark throughput but also to reveal the conceptual trade-offs inherent in architectural decision-making for distributed data summarization. It demonstrates how the choice of computation placement, centralized versus federated, affects key system qualities such as resource utilization, scalability, data freshness and bandwidth efficiency. These findings inform the design of future data processing pipelines, where balancing local and global responsibilities is critical. A detailed explanation of the benchmarking methodology and experimental setup is provided in chapter 4.

3.1.1 Data safety

Within the scope of this thesis, *data safety* is defined as minimizing data loss prior to the data reaching the server. Once the data has reached the server it is considered safe. Data may be lost before reaching the server in two primary ways: (1) the client crashes, causing any data held in volatile memory to be lost before transmission, or (2) the client transmits the data, but it is lost due to a network failure.

A system with high data safety ensures that little or no data is lost in this process, while low data safety indicates that losses are more likely to occur before the server receives and stores the data. Unlike a probability measure, data safety is not restricted to values between 0 and 1, higher values simply indicate better protection against data loss, with an ideal system corresponding to infinitely high data safety.

3.1.2 Data freshness

Data freshness is defined as the recency of the data available on the server. It is inversely proportional to *data latency*, which is defined as the time from a tuple being ingested in the system to it being available to query from the server. Thus, reducing the maximum data latency enhances data freshness.

3.2 Representative sketches

This thesis explores two distinct sketching algorithms: the Count sketch and the KLL sketch. These were chosen for their ability to capture complementary statistical properties of data and for representing different sketching methodologies. Their methodological differences reflect important trade-offs in memory usage and scalability. These contrasting designs influence how each algorithm performs in distributed systems. The differences are explained in subsection 2.3.4.

The Count sketch is designed for frequency estimation. It uses a fixed-size structure and multiple hash functions to approximate item frequencies, making it well-suited for efficiently retrieving the top-K most frequent elements to detect heavy hitters, such as frequent IP addresses in network traffic monitoring.

In contrast, the KLL sketch is designed specifically for quantile approximation. It dynamically adjusts its size through random sampling and compaction, enabling it to estimate percentiles, such as the median or 99th percentile. The KLL sketch was chosen for its broad applicability to all ordered datasets, which extends its usefulness across a wide range of use cases. Furthermore, its superior speed and space efficiency compared to alternative quantile sketches within the same domain make it a compelling choice [5]. These characteristics make the KLL sketch particularly well-suited for real-time analytics applications.

In addition, the insights from these two sketches can extend to similar algorithms. For instance, while the Count sketch and Count-Min sketch differ in their statistical properties, such as the Count-Min sketch's biased estimates that are always greater than or equal to the true count [5], their underlying computational structures are similar. Both rely on hash-based projections and maintain a fixed-size two-dimensional array updated with each stream element. As a result, benchmarks obtained for the Count sketch may be indicative of the computational demands of the Count-Min sketch as well.

3.3 Applications of relevance

The selected sketches are applicable for a broad range of practical applications. The Count sketch with the top-K extension, for example, can be utilized to identify frequently occurring faults in vehicles or to monitor the most frequently accessed content in content delivery networks (CDNs). In contrast, the KLL sketch is suitable for estimating data distributions, such as analyzing battery charge levels in electric vehicles, where understanding the variability of measurements over time is essential. These sketches provide complementary analytical capabilities and cover a wide variety of use cases.

A Count sketch, with the top-K extension to track the most frequent elements, can be applied to monitor errors in trucks, enabling the identification of overrepresented errors. These disproportionately frequent errors may signal sensor malfunctions, intermittent faults, or systematic issues with specific vehicle components. For example,

if a particular sensor consistently triggers the same errors over time, it could indicate that the sensor is failing and requires replacement. At the fleet-wide level, merging Count sketches from multiple vehicles enables the identification of frequently occurring errors across the truck fleet. These merged summaries can reveal common points of failure, offering valuable insights to inform and enhance truck design improvements.

Another practical use of the Count sketch and top-K is in Content Delivery Networks (CDNs). By using Count sketch to estimate the frequency of content requests, it becomes possible to identify the most popular content accessed at local edge servers. This capability facilitates local trend detection, enabling frequently requested content to be cached closer to where it is consumed, which reduces latency. In addition to caching, Count sketch can support pre-caching, predictively storing trending or high-demand content before it is requested, based on observed patterns. This is especially useful in connected vehicles or remote areas with intermittent connectivity. When local usage trends are merged at a global scale, they provide valuable insights into broader content consumption patterns. Content creators and platform operators can leverage this information to guide future content development, tailoring offerings to align with both regional preferences and emerging global trends.

Maintaining optimal battery charge levels is critical for preserving long-term battery health in electric vehicles. Battery degradation accelerates when the state of charge consistently falls below 20% or exceeds 80% [20]. The KLL sketch can store a summary of the charge level over time. By querying the 20th and 80th percentiles from the KLL sketch, one can estimate the proportion of time the battery operates in these suboptimal charge ranges. Merging KLL sketches from multiple trucks enables fleet-level analysis of battery charge levels, identifying trends in suboptimal usage. If this proportion exceeds a predefined threshold, indicating frequent operation outside the optimal charge window, proactive measures should be taken to mitigate potential degradation across the whole fleet.

The KLL sketch algorithm provides a method for capturing the distribution of any ordered dataset. To illustrate the capabilities of the KLL sketch, a data set of vehicle speeds over time in discrete time intervals measured in m/s is presented without sketching in Figure 3.1, alongside the corresponding distribution as approximated by the KLL sketch in Figure 3.2.

3.4 Implementation

To implement the system described in Figure 1.1, multiple core components are necessary:

- **Data generation**: simulates a data stream for the clients.
- **Communication protocols**: handles data transmission between client and server.
- **Sketching library**: provides sketch algorithms implementations for the clients and server.

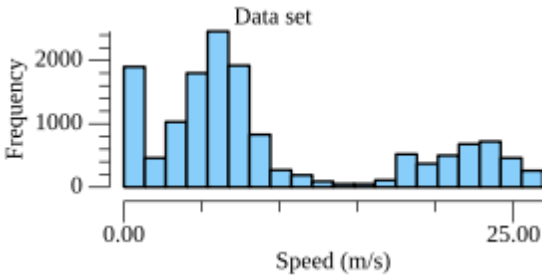


Figure 3.1: A histogram representation of the dataset. The histogram has 20 buckets.

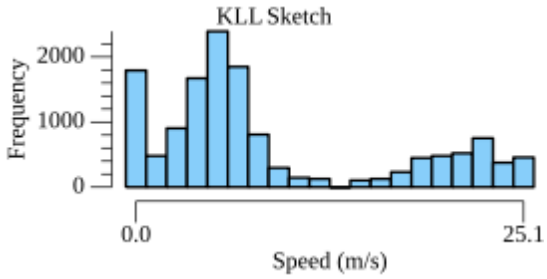


Figure 3.2: A histogram of the dataset stored in a KLL sketch. The histogram has 20 buckets.

- **Client**: reads a data stream, sketches the tuples from the data stream in the federated system or buffers the tuples from the data stream in the centralized system, and transmits data to the server.
- **Server**: receives data from clients, merges the sketch with the server's in the federated system or adds the tuples to the server's sketch in the centralized system.

To facilitate testing and debugging, a *query module* was implemented. The query module allows for executing queries on the server's internal state, enabling verification of the sketching functionality. The data flow of the system can be seen in Figure 3.3.

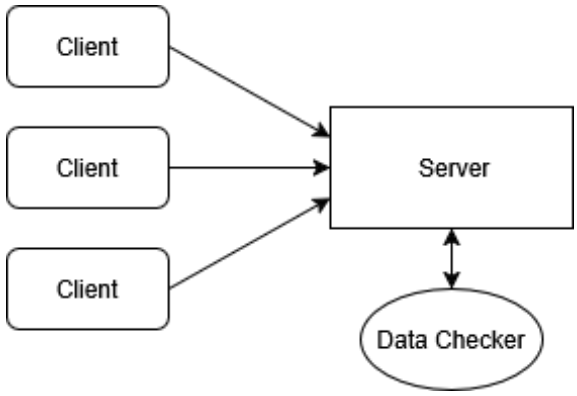


Figure 3.3: A diagram showing the system flow

3.4.1 The Go programming language

The distributed systems are developed in *Go* or *Golang* as it is sometimes referred to, since the name Go can be ambiguous [21]. Go is an open-source programming language supported by Google. The language was designed for developing distributed systems and networks. Concurrent execution is supported through *goroutines*. Go can be cross-compiled to both *x86* and *ARM* processor architectures, which is important since the benchmarks were performed on an *x86* laptop and *ARM*-driven Raspberry Pis as described in subsection 4.1.1.

3.4.2 Data Stream

The data stream generator is a concurrent process used by the client. A data stream is created from a .csv file. By specifying a data set's path and the header field name, the data set is accessed. The data set consists of multiple tuples containing one element. The data set is added, one tuple at a time, to a Go channel. Channels can be written to and read from while shared between concurrent processes.

The stream can be configured to mimic different stream rates. Depending on the stream rate r tuples per second configured, a tuple is added to the channel every $\frac{1}{r}$ seconds. The interval is measured using a busy wait timer to accurately simulate the time between adding tuples to the channel.

3.4.3 Communication Protocol

All communication within the system is done with Remote Procedure Calls, RPC. This approach was selected to take advantage of the extensibility offered by Googles gRPC framework [22], which facilitates integration with future systems. gRPC uses *Protocol buffers* to specify the shape of the transmitted data. Protocol buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data [23].

3.4.4 Architecture

This section describes the architecture of both the federated system and centralized system, as well as the implementation of the sketching library, client, server, and the query module.

Sketch library

To have sketches to run on the proof-of-concept system, a simple sketching library was implemented, featuring two different sketches.

The Count sketch implements the following methods where `type` is a float or integer:

- `NewCountSketch(int seed, uint size,  uint num_hashes) *CountSketch[type]`
- `NewCountFromData(int[][] arr, uint[] seeds) *Countsketch[type]`
- `Add(*Countsketch[type] cs, type item)`
- `Query(*Countsketch[type] cs, type item)`
- `Merge(*Countsketch[type] cs , *Countsketch[type] cs2)`
- `Print(*Countsketch[type] cs)`

The implementation utilizes the Murmur3 hash function due to its high computational efficiency and pairwise independence properties [24]. These characteristics ensure performant hashing, and a robust and uniform distribution of hash values,

which is critical for maintaining the accuracy guarantees of the Count sketch algorithm.

The KLL sketch implements the following methods where *type* is any ordered data type:

- `NewKLLSketch(int k) *KLLSketch[type]`
- `NewKLLFromData(type[][] arr, int n, int k) *KLLSketch[type]`
- `Add(*KLLSketch[type] kll , type item)`
- `Query(*KLLSketch[type] kll, type val) int`
- `QueryQuantile(*KLLSketch[type] kll, float phi) type`
- `Merge(*KLLSketch[type] kll, *KLLSketch[type] kll2)`
- `Print(*KLLSketch[type] kll)`

Client

The client consists of two distinct components. The first component is the data stream generator described in subsection 3.4.2 to be read by the second component of the client. The second part is responsible for processing the data. During the initialization phase, several parameters are configured, including the server IP address and port number, the type of sketch to be used, the path to a .csv file along with the specified header field name, the stream rate, and the batch size.

The client connects to the server using RPC as described in subsection 3.4.3. Communication is client-initiated; the client transmits data to the server and receives an ACK, but the server does not initiate any communication with the client. In the federated system, the client sends sketched data that is `Merged` with the server's sketch. In contrast, the client in the centralized system sends its buffer of tuples, which the server integrates to its sketch via the `Add` operation. The client transmits data at equal intervals, based on the configured batch size bs and stream rate r. After transmitting, the data is discarded.

Server

The server is initialized by specifying the port number on which it will accept incoming connections. Once initialized, the server maintains the global state of each type of sketch and continuously waits for incoming requests to merge data.

Utilizing the sketching library described in subsection 3.4.4, the server applies the `Merge` operation in the federated system and the `Add` operation in the centralized system. These methods amend the data from the client to the server's sketch. Additionally, the server implements one or more query methods for the different sketches, depending on what information may be extracted.

Query module

The query module enables querying the server during runtime. During its initialization, the server's IP and port need to be configured. The query module connects to the server using RPC and remains idle until a command is entered by the user. When a command is entered, the query module initiates a remote procedure call to the server to retrieve the data associated with the specified query. The commands implemented are:

- `QueryCount(Any query)` Returns the estimated frequency of item `query` in a Count sketch.
- `QueryKll(Numeric query)` Returns the element of rank `query` in a KLL sketch.
- `ReverseQueryKll(Float quantile, String type)` Returns the item corresponding to quantile `quantile` in a KLL sketch of type `type`.
- `PlotKll(Int numBins, String type)` Generates and saves a histogram with `numBins` bins from a KLL sketch of the specified type `type`. The minimum and maximum values in the sketch are obtained via

$$\texttt{min} = \texttt{ReverseQueryKll}(0.0, \texttt{type}), \quad \texttt{max} = \texttt{ReverseQueryKll}(1.0, \texttt{type}), \tag{3.1}$$

and the range is computed as $range = \texttt{max} - \texttt{min}$. For each bin $x \in \{1, \ldots, \texttt{numBins}\}$, the bin boundaries correspond to

$$low_x = \texttt{min} + \frac{range \cdot (x-1)}{\texttt{numBins}}, \quad high_x = \texttt{min} + \frac{range \cdot x}{\texttt{numBins}}, \tag{3.2}$$

and the bin value is given by

$$\text{bin}_x = \texttt{QueryKll}(high_x, \texttt{type}) - \texttt{QueryKll}(low_x, \texttt{type})\,. \tag{3.3}$$

3.4.5 Fault Tolerance Measures

Two types of faults are anticipated in the system: **crashes** and **network errors**.

- **Crashes** comprise unexpected terminations of either the server or a client, which may occur at any point during execution, including before, during, or after a request.
- **Network errors** comprise connection loss, packet loss, lost replies, and situations in which the client cannot locate the server.

Section section 2.2 classifies these failure scenarios and discusses their implications for system consistency and recovery; it does not prescribe specific strategies. The methods described below are therefore presented as the handling strategies applied in the implemented system and are associated with the fault categories identified in section 2.2.

Crash handling

Crash handling seeks to restore failed nodes to a state in which they can resume normal operation while preserving system consistency. For the purposes of the implementation described here, the following behavior is employed:

- **Server.** The server is configured to restart after a crash and to retain the latest persisted version of the sketch state so that service can continue with minimal loss of state.
- **Client.** Clients are not provided with an automated recovery mechanism; a manual restart is required to return a crashed client to operation. If a client crashes after issuing a merge request but before the client-side operation completes, the server may still complete and apply the merge. Because merge operations do not produce orphaned state, no dedicated orphan-extermination procedure is necessary.

The implementation uses Go runtime and gRPC features to prevent uncontrolled server termination. In Go, a *panic* denotes an unanticipated runtime error that normally causes program termination, whereas ordinary errors are returned as values and handled explicitly. Server-side panics that occur during RPC handling are intercepted and translated into gRPC error responses using the Go `recover` facility together with a gRPC panic-recovery interceptor [25], [26]. This prevents the server from terminating unexpectedly and preserves the in-memory sketch state, but it does not attempt to resume the interrupted computation.

Network error handling

Network error handling is intended to preserve forward progress and consistency when client-server communication is disrupted. This addresses the scenarios identified in section 2.2: the client being unable to locate the server, client requests being lost, and replies being lost.

When a merge RPC fails due to connection loss, the client discards the data associated with that particular merge attempt and continues processing newly arriving data locally. The client periodically retries the merge operation; the number of retry attempts is configurable to avoid infinite retry loops if the server is unreachable. This approach enforces *at-most-once* semantics for merge transmission, since the client discards the data after each attempted merge regardless of the attempt's outcome. Upon successful reconnection, only the data collected since the last merge attempt is sent to the server.

4

Evaluation

The systems, including their clients and servers, are evaluated based on throughput and latency, to help answer the questions posted in section 1.2. The specific interpretation of these metrics for each component is detailed in their respective sections. Since the Count sketch and KLL sketch differ in data representation and the computational costs of `add` and `merge` operations, their impact on throughput varies. Consequently, the evaluation is conducted independently for both sketches.

The tools used for evaluation are `Benchmarks` from Go's `testing` package and shell scripts. The `testing` package's benchmarking feature runs the benchmarking code, which executes the tested code for a set number of iterations. The shell scripts start the benchmarks and save the results. The results from the benchmarks are used to evaluate the implementation and answer the research questions stated in section 1.2. Manual time measurements using Go's `time` package were also done in some cases. The unprocessed data and the code can be found in the GitHub repository [27].

4.1 Evaluation setup

In this section, the software and hardware used for all the benchmarks is explained, the data on which the benchmark was run as well as how the sketches were configured.

4.1.1 Hardware setup

The server tests were run on a framework 13 laptop with an AMD Ryzen 5 7640U processor that has a base frequency of 3.5 GHz and a boost clock speed of 4.9 GHz, and 32 GB of RAM. The system tests were conducted using a local area network connected through a passive 1 gigabit network switch, D-Link DGS 105GL. Connected to the network were three Raspberry Pis, referred to as testbed nodes, and the previously mentioned laptop. The Raspberry Pis used were Raspberry Pi 5 with a 64-bit quad-core Cortex-A76 (BCM2712) processor with a clock speed of 2.4 GHz and 8 GB LPDDR4X SDRAM. A picture of the testbed can be seen in Figure 4.1.

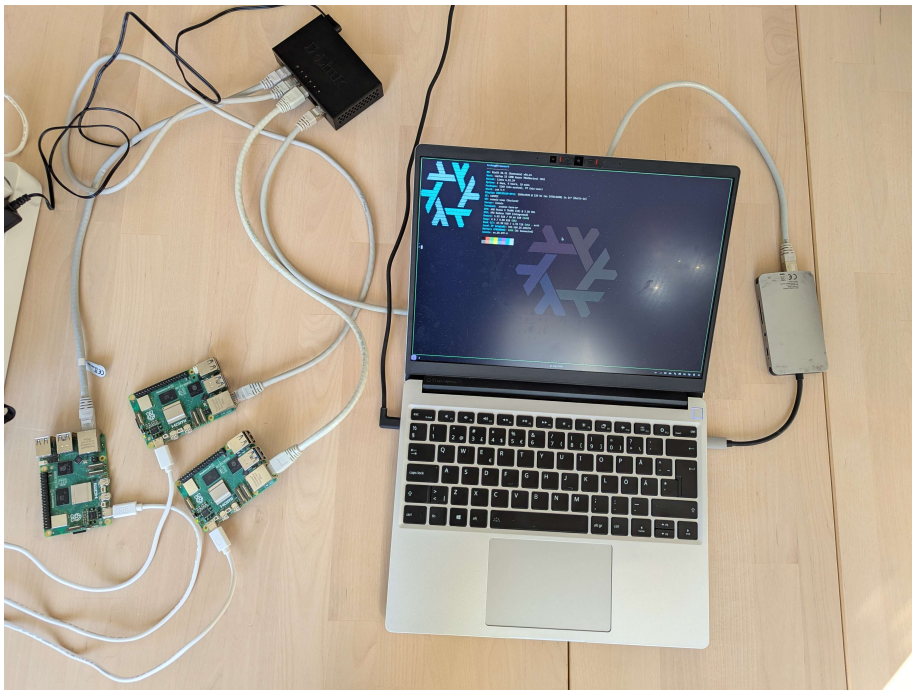


Figure 4.1: A picture of the testbed

4.1.2 Software setup

The computer used as the server during systems and server benchmarks runs NixOS 25.05. The package versions are specified in a nix flake in this thesis' GitHub repository [27]. The Raspberry Pis used as clients during system and client benchmarks were running Raspbian version 2024-11-19 Lite, which is based on Debian 12 (Bookworm). The clients were only accessed via ssh, so the Lite version of Raspbian was installed to have minimal overhead.

The program was compiled using Golang 1.24.1, and the imported modules used the following versions:

- github.com/spaolacci/murmur3 v1.1.0
- golang.org/x/exp v0.0.0-20250305212735-054e65f0b394
- google.golang.org/grpc v1.71.0
- google.golang.org/protobuf v1.36.5

4.1.3 Benchmark data

The dataset that was used comes from *Passive Vehicular Sensors Datasets* [28], which is a 46 GB collection of datasets, which contains video as well as vehicular sensor data including, speed, acceleration, and temperature. The dataset chosen to test the system is a time series of 1400 64-bit floating point values between 0.0 and 27.0 representing a vehicle's speed in meters per second over time and its frequency distribution is visualized in Figure 3.1. The dataset was the column called

"speed_meters_per_second" in the file "dataset_gps.csv" in the folder "PVS 1" in the dataset [28]. While this dataset has not been used in prior academic studies, it was selected due to its relevance to one of the primary use cases considered in this thesis: vehicle data processing. In benchmarks requiring more than 1400 tuples, the dataset is repeated within the stream. The longest simulated stream comprises 50 000 tuples, resulting in the dataset being repeated 36 times.

4.1.4 Sketch configuration

The sketches evaluated in this benchmark were configured to ensure a balance between accuracy and resource usage. The Count Sketch was configured with $f = 10$ hash functions and $m = 100$ counters per hash function. This configuration yields an additive error bound of $\varepsilon = 0.01$ and a low δ [5]. In practical terms, this means the sketch provides an estimate with 1% additive error with high confidence.

The KLL sketch was configured with $k = 100$ and employed the capacity function defined in Equation 2.4. Under this configuration, the KLL sketch achieves an expected additive rank error of approximately 3.18% [29].

4.2 Goals and measurements

To determine when a federated approach to sketching outperforms a centralized approach, three different benchmarks were conducted, each varying a set of independent variables. The evaluated parameters include the number of clients, the batch size, the stream rate, and the type of sketch used. The three benchmarks are designed to test different core parts of the system and to answer the three questions stated in section 1.2:

1. **When does a federated approach outperform a centralized one in terms of throughput?**
2. **When is it feasible to offload computation to the edge?**
3. **What are the trade-offs between throughput, data safety, and data freshness at the server?**

4.2.1 Data freshness

Data freshness, although not explicitly quantified by the benchmarks used in this thesis, is a critical parameter in certain applications, such as real-time analytics. This thesis approximates data freshness analytically focusing on data latency. Data latency refers to the delay between when a tuple is generated and when it has been merged to the server's sketch. Assuming the system is not saturated, the maximum data latency can be estimated as the sum of four components (Equation 4.1): the time required to accumulate a batch size worth of tuples, calculated as the batch size divided by the stream rate, the time for the client to process the last tuple, the network delay, and the processing time required to execute a merge operation on the server. This formulation arises because the tuple experiencing the greatest latency

is the first tuple generated and processed by the client following a merge event. It must wait for the accumulation and processing of a batch size worth of tuples before being transmitted, after which it incurs the additional delay of the server's merge processing time. In this way, the thesis helps illuminate the relationships between batch size and data freshness. The relation between batch size and system throughput is also explored in subsection 4.4.1, allowing for analysis of the trade-off inherent among these factors.

$$\begin{aligned} \textit{Maximum Data Latency} = \frac{bs}{r} + \textit{Client Processing Time} + \\ \textit{Network Latency} + \textit{Merge Processing Time} \end{aligned} \tag{4.1}$$

In Equation 4.1 r is the stream rate in tuples per second and bs is the batch size in tuples. The symbols used in this thesis can be found in Table 4.1.

Table 4.1: The symbols used throughout this thesis

Name	Symbol	Unit
Stream rate	r	Tuples per second
Set of stream rates	$\mathcal{R}_x$	Set
Batch size	bs	Tuples
Processing time	t	Seconds
Bandwidth	Bw	Bits per second
Tuple size	ts	Bits per tuple
Size of serialized message	M_{size}	Bits
Error bound	ε	None
Error rate	δ	None
Sketch Length	m	None
Element	e	None
Hash function	h_i	None
Number of hash functions	f	None
Number of elements in sketch	n	Tuples
Count sketch matrix	M	None
Count sketch sign hash function	h_s	None
Query	q	None
Multiset	V	None
Top K	K	None
Quantile	ϕ	None
KLL size parameter	k	None
KLL layer	h	None
KLL total height	H	Layers
KLL rank	R	None
Number of merge requests	y	Requests
Number of merge requests responses	y_{resp}	Responses
Number of clients per test bed node	N	Goroutines

Continued on next page

Name	Symbol	Unit
Number of nodes	ns	Nodes
Probability that a client crashes when processing a tuple	p_{cpt}	None
Probability that a client crashes during a batch	p_{crash}	None
Number of tuples lost due to a client crash per batch	d	Tuples
Number of tuples lost due to packet loss per batch	pl	Tuples
Probability that a packet is lost	$p_{\text{packetLoss}}$	None
Number of tuples lost per batch	D	Tuples
Amount of batches processed	nb	Batches

4.2.2 Data safety

Data safety is analytically evaluated by quantifying the expected number of tuples lost in a batch during data processing. This metric is subsequently employed to analyze the trade-offs among data safety, data freshness, and throughput. Assuming that each tuple has an equal probability of failure when being processed on the client, the data safety can be derived using the law of total expectation. Let D denote the random variable representing the number of tuples lost in a batch of size bs. Data loss may occur through two mutually exclusive mechanisms:

1. **Client-side failure during processing**, resulting in the loss of all tuples processed up until the crash. Let p_{cpt} be the probability of a crash per tuple. Then the number of tuples lost, d, is a *truncated geometric* random variable with maximum bs, representing the number of tuples processed before the crash [30]. Equation 4.2 details the expected number of tuples lost due to client-side failures, when $p_{cpt} \neq 0$.

$$E[d] = \sum_{k=1}^{bs} k(1-p_{cpt})^{(k-1)} p_{cpt} = \frac{1 - bs \cdot p_{cpt}(1-p_{cpt})^{bs} - (1-p_{cpt})^{bs}}{p_{cpt}} \quad (4.2)$$

   Examining Equation 4.2 using Matplotlib (Figure 4.2) it was determined that increasing bs, regardless of p_{cpt}, strictly increases the expected number of tuples lost per batch [31]. Although Equation 4.2 is strictly increasing in the batch size bs for any fixed p_{cpt}, its growth is asymptotically bounded. In particular, $E[d]$ is bounded above by $1/p_{cpt}$. When $bs \gg 1/p_{cpt}$, the probability of reading the entire batch without encountering a crash becomes vanishingly small. Consequently, the expected number of lost tuples approaches this asymptotic limit, and $E[d]$ exhibits a plateau for batch sizes much larger than $1/p_{cpt}$.

   The probability of a client-side failure during a batch is shown in Equation 4.3.

$$P_{\text{crash}} = 1 - (1-p_{cpt})^{bs}. \quad (4.3)$$

2. **Packet loss during transmission**, which causes the loss of the entire batch of bs tuples. Packet loss can only occur if the batch has not already been lost due to a client-side failure, since otherwise the request is never sent and thus no packet can be lost. Let $p_{\text{packetLoss}}$ be the probability of packet loss

during transmission. Equation 4.4 shows the expected number of tuples lost per batch due to packet loss.

$$E[pl] = (1 - P_{\text{crash}}) \cdot p_{\text{packetLoss}} \cdot bs. \tag{4.4}$$

Visual analysis of Equation 4.4 indicates that the expected number of tuples lost due to packet loss initially grows approximately linearly with bs for small batch sizes, but the factor $(1 - p_{cpt})^{bs}$ gradually limits this growth, causing $E[pl]$ to eventually decrease for larger batches.

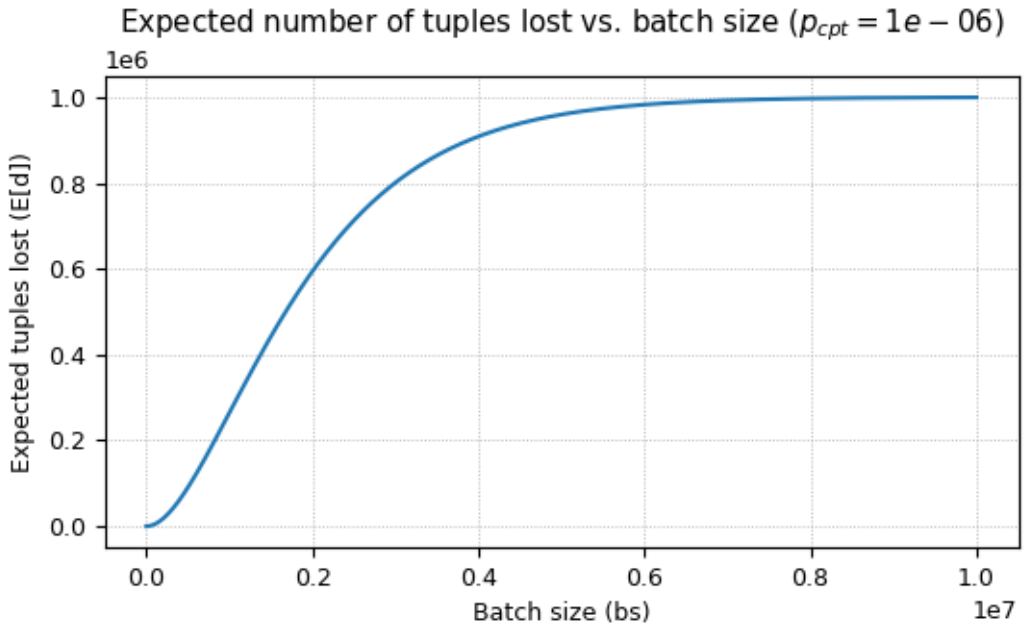


Figure 4.2: Visualization of the expected number of tuples lost per batch, $E[d]$, as a function of the batch size bs, with the parameter $p_{cpt} = 10^{-6}$.

The expected number of tuples lost in a batch is therefore:

$$E[D] = E[d] + E[pl]. \tag{4.5}$$

The *data safety* for a batch of size bs is defined by the inverse of the expected number of tuples lost per batch.

$$\text{DataSafety}(bs) = \frac{1}{E[D]}. \tag{4.6}$$

In the ideal case where $E[D] = 0$, no data is lost before reaching the server, and the data safety tends toward infinity, representing a perfectly safe system.

A visual analysis of Equation 4.6 indicates that increasing the batch size bs consistently leads to a higher expected number of tuples lost per batch, and thus lower

overall data safety. Although the expected packet loss, $E[pl]$, may appear to decrease for very large buffer sizes bs, this apparent reduction is misleading. The expression in Equation 4.4 does not account for tuples lost due to client-side crashes. In practice, the observed decrease occurs because larger buffer sizes increase the likelihood of such crashes. The program may terminate before packets are transmitted, resulting in zero reported packet-loss for those tuples, while the total number of tuples lost actually increases. Consequently, larger batch sizes are detrimental to data safety, making smaller batch sizes preferable.

4.2.3 System throughput

The system's throughput is measured as the number of tuples the system can process in a second. By measuring system throughput and comparing the federated and centralized systems, the throughput benefits of federating computation can be assessed. Specifically, the evaluation aims to identify which combinations of independent variables in Table 4.2 result in the federated system outperforming the centralized counterpart.

The system test is conducted on the hardware configuration detailed in subsection 4.1.1. The system benchmark is conducted systematically across all combinations of values listed in Table 4.2, which include the independent variables: number of clients, batch size, stream rate, stream length, sketch type, and compute location.

Prior to launching the system benchmark on the client devices, a server process is initialized on the laptop. Subsequently, the benchmark is simultaneously initiated on all three testbed nodes using parallel SSH (pssh) [32]. Upon receiving the SSH command, each testbed node spawns N goroutines, each running a client.

Running multiple client instances on a single testbed node proved infeasible, as it introduced significant performance degradation and distorted benchmark results. Therefore, an alternative approach was required to simulate systems involving more than one client per testbed node. To address this limitation, a precomputed sketch or buffer of tuples was generated for each simulated client before initiating the benchmark. The dataset used to precompute the sketches has 1400 tuples. Sketches of size 1000 and 10 000 are used in the benchmarks, as seen in Table 4.2. 1400 tuples is sufficient to produce a sketch with 1000 tuples. However, to produce a sketch with 10 000 tuples the dataset is repeated. This repetition does not affect the results since the focus is on processing efficiency rather than data novelty.

Each benchmark run simulates the processing of precomputed sketches or buffers under conditions that replicate real-time operation. The timing of merge requests is determined by the stream rate r (tuples per second) and the batch size bs (tuples per batch). A merge is triggered after every bs tuples, resulting in a merge request being issued at intervals of $\frac{bs}{r}$ seconds. This approach maintains the desired communication and timing characteristics while eliminating the computational overhead of generating sketches on the testbed nodes.

As soon as the clients are ready they start transmitting merge requests to the server at the fixed interval of $\frac{bs}{r}$ seconds. The delay between requests is enforced using

Go's `time.Sleep(`bs/r`)` function, ensuring consistent pacing throughout execution.

During the benchmark, each client process is configured to transmit merge requests for 3 seconds without measuring them, a warm-up period. After the warm-up, the client counts the number of successful merges performed in a 5 second interval. The parameters summarized in Table 4.2 correspond to test cases benchmarked. The stream rate r is varied according to the sequences specified in Definition 4.1 for the Count sketch and Definition 4.2 for the KLL sketch. These sequences comprise evenly spaced stream-rate values ranging from 1000 to the highest throughput sustained by the respective client, as determined by the client-side benchmark presented in subsection 4.3.2.

Definition 4.1.

$$\mathcal{R}_{Count} = \begin{Bmatrix} 1000, 45\,333, 89\,666, 133\,000, 176\,333, \\ 219\,666, 263\,000, 306\,333, 349\,666, 400\,000 \end{Bmatrix}$$

Definition 4.2.

$$\mathcal{R}_{KLL} = \begin{Bmatrix} 1000, 167\,556, 334\,112, 500\,667, 667\,223, \\ 833\,778, 1\,000\,334, 1\,166\,889, 1\,333\,445, 1\,500\,000 \end{Bmatrix}$$

The system benchmark yields three separate measurements, one from each testbed node. The sum of these measurements, denoted y_{resp}, is used as the total successful merges. The total tuples processed is given by the product of the number of successful merges and the batch size bs, the number of tuples. It is divided by the duration of the benchmark, t, in order to obtain the throughput in tuples per second. Therefore, the system throughput can be expressed as:

$$SystemThroughput = \frac{y_{resp} \cdot bs}{t} \text{tuples/second.} \tag{4.7}$$

Parameter	**Values**
Clients Per Testbed Node (Total)	{1 (3), 10 (30), 50 (150), 100 (300)}
Stream Rate (tuples/second)	{ Count: $\mathcal{R}_{\text{Count}}$ KLL: $\mathcal{R}_{\text{KLL}}$ }
Batch Size (tuples)	{1000, 10 000}
Sketch Type	{KLL Sketch($k = 100$), Count Sketch($f = 10$, $m = 100$)}
Compute Location	{Centralized, Federated}
Duration (seconds)	{5}

Table 4.2: Independent variables for the system benchmark and their tested values

Network limitations

The system benchmark operates over a bidirectional 1 Gbps per-port network switch, as described in subsection 4.1.1. This configuration introduces the possibility of a bandwidth-related bottleneck. To evaluate the impact of such constraints, the relationship between system throughput and available bandwidth is analyzed. The complete network topology, including maximum link capacities, is shown in Figure 4.3.

Network saturation occurs when the offered data rate approaches or exceeds the maximum link capacity. Initially, as the system load increases, throughput grows almost linearly. However, once the network capacity is reached, further increases in load do not yield higher throughput. At this point the system becomes bandwidth-limited, resulting in increased latency and throughput plateaus. This phenomenon is typically visualized by a knee curve, where the knee marks the inflection point at which throughput ceases to scale linearly with offered load [33].

Since communication occurs exclusively between the nodes and the server, and each node transmits an equal amount of data, the network can be abstracted to a single equivalent node with an aggregated bandwidth of 3 Gbps, as depicted in Figure 4.4.

From Figure 4.4, it is evident that link **D**, with a capacity of $1 Gbps$, is the limiting factor compared to link **A** at 3 Gbps. This constraint applies to both transmission directions. Consequently, for analytical purposes, the network can be further reduced to the minimal representation shown in Figure 4.5, consisting of a single link with a maximum bandwidth of 1 Gbps.

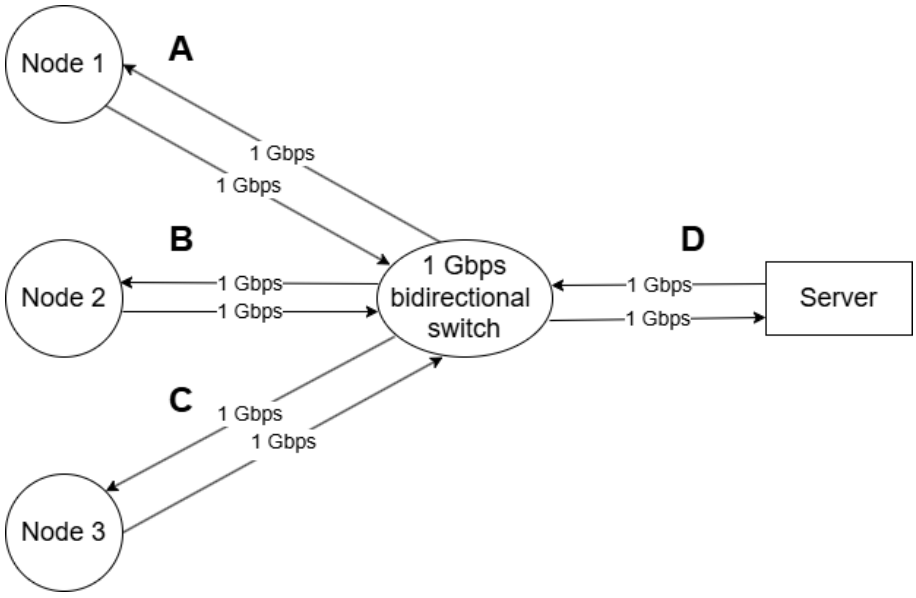


Figure 4.3: System network topology with corresponding maximum bandwidth in each direction. Links **A**, **B**, and **C** link the testbed nodes to the switch, while link **D** connects the server to the switch.

Figure 4.4: A simplified network model where link **A** represents the aggregated bandwidth between all nodes and the switch. Link **D** is between the server and the switch.

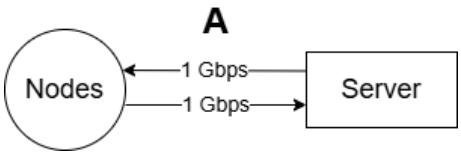


Figure 4.5: A minimal network model where link **A** represents the whole network, clarifying the 1 Gbps bottleneck imposed by the link to the server.

To identify if the network is a bottleneck, it is necessary to compare the throughput of the system to the bandwidth. As shown in Figure 4.5, the network supports a maximum bandwidth of $Bw = 1$ Gbps. Since client and server throughput are measured in tuples per second, the bandwidth must be expressed in the same unit to allow direct comparison. To determine whether the network constitutes a bottleneck, the system throughput must be compared to the maximum network bandwidth. System throughput is defined as the total number of tuples across all clients that are successfully merged into the server, divided by the total processing time from when clients begin processing until the last merge is finished. As illustrated in Figure 4.5, the network provides a maximum bandwidth of $Bw = 1$ Gbps. Because system throughput is measured in tuples per second, while network bandwidth is expressed in bits per second, the two quantities are not directly comparable. To enable a direct comparison, the bandwidth can be converted to tuples per second by dividing it by the size of a message in bits multiplied with the number of tuples in the message.

Before a message is transmitted in gRPC, the Protobuf structure is serialized, which reduces the data size [34]. The serialized data is then transmitted over the network. Let M_{size} denote the size of the serialized data in bits, and let bs denote the batch size. Each transmitted RPC message thus corresponds to bs tuples. Consequently, the maximum achievable throughput of the switch, expressed in tuples per second, can be computed as:

$$\text{MaximumSwitchThroughput}(bs, M_{size}) = \frac{Bw}{M_{size}} \cdot bs \text{ tuples/second} \tag{4.8}$$

This expression enables a direct comparison between measured throughput and the network's theoretical maximum, facilitating the identification of scenarios limited by bandwidth versus those constrained by client or server hardware. It should be noted

that Equation 4.8 assumes a continuous flow of data, whereas actual transmission occurs in discrete batches. Consequently, the equation represents an idealized upper bound; deviations from this bound could arise from factors such as batching effects and transient variations in network performance.

4.2.4 Client throughput

The client is required to process data streams in real time, ensuring that each element from the data stream is processed at the same or a faster rate than the stream rate r. This benchmark establishes the stream rate at which the client becomes saturated and determines the client's throughput.

To investigate the saturation point for the client, a benchmark is conducted using the following methodology. From the laptop, a benchmark is initiated on the Raspberry Pi via SSH. The hardware specifications for both the laptop and the Raspberry Pi are detailed in subsection 4.1.1. Prior to starting the timer, a data stream comprising $L = 1000$ tuples is generated. Once the stream is prepared, the processing time, t, is measured from the initiation of processing until the entire stream is fully processed by the client. No merges are performed in this benchmark. This benchmark is executed across a range of stream rates varying from 100 to 10^9 tuples per second. The throughput, expressed as tuples per second, is calculated using the formula $\text{Throughput} = \frac{L}{t}$. The benchmark is performed using all combinations of the variables in Table 4.3. The client benchmark's stream rate r, where $r \in \mathcal{R}_{\text{client}}$, is varied in increments of 2% (see Definition 4.3). This adjustment enables high-resolution exploration across a broad range of stream rates, facilitating precise identification of the saturation point as well as detailed comparison between different processing alternatives.

Definition 4.3.

$$\mathcal{R}_{client} = \left\{ \frac{10^9}{\lfloor 10^6 \cdot 0.98^x \rfloor} \;\middle|\; x \in \mathbb{N},\ \lfloor 10^6 \cdot 0.98^x \rfloor \geq 1 \right\} \textit{ tuples/s.}$$

Parameter	Values
Stream Rate (tuples/s)	$\mathcal{R}_{\text{client}}$
Processing Type	{KLL Sketch($k = 100$), Count Sketch($f = 10$, $m = 100$), Buffering}
Stream Length	1000

Table 4.3: Independent variables for the client benchmark and their tested values

4.2.5 Server throughput

The number of clients that the server can support is determined by its ability to process incoming merge requests. Increasing the batch size reduces the frequency of the requests, thereby reducing the communication overhead on the server. However, it can potentially increase the computational cost of each merge operation.

To measure the server's throughput in the centralized and federated system, enabling evaluation of the trade-off between performance, data safety, and data freshness, the following benchmark was implemented. A server is started on the laptop, described in subsection 4.1.1. Subsequently, a client is started on the same laptop, configured to send merge requests with a specified batch size bs. Upon receiving a request, the server records the current timestamp and subsequently compares it to the timestamp taken after completing the processing of the request using Go's `time` package. The difference between these two timestamps represents the processing latency, which is then used to calculate the throughput. This measurement approach isolates the server's internal processing time, more specifically step 5 in 4.6, thereby excluding any network transmission delays or client-side latency. The benchmark is repeated 1000 times per configuration to ensure statistical reliability. The throughput in tuples per second is calculated using the following formula $\frac{bs}{t}$, where t is the average processing time of the 1000 measurements. This benchmark is conducted with all combinations of the independent variables in Table 4.4.

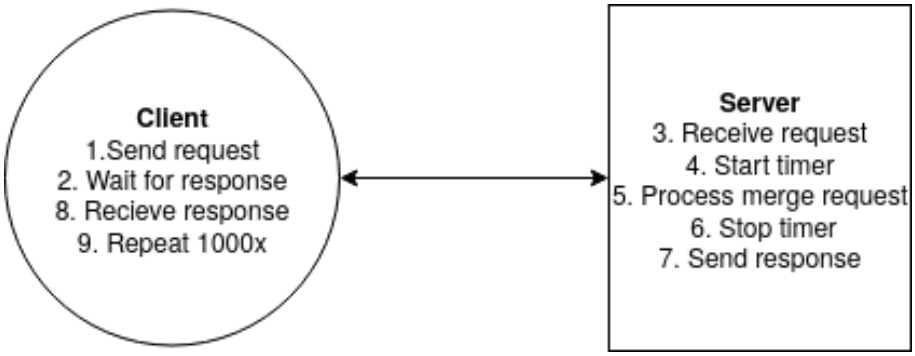


Figure 4.6: The server benchmark proceeds as follows. First, the client issues a merge request and awaits the corresponding response. Upon receiving the request, the server starts a timer and begins processing the merge request. Once the processing is complete, the server stops the timer and returns a response to the client. The client then receives the response and repeats this procedure 1000 times.

Parameter	**Values**
Batch Size (tuples)	{1000, 2000, 4000, 8000, 16 000, 32 000, 64 000, 128 000, 256 000, 512 000}
Stream length	$1000 \cdot$ Batch Size
Number of clients	1
Sketch Type	{KLL Sketch($k = 100$), Count Sketch($f = 10$, $m = 100$)}
Compute location	{Centralized, Federated}

Table 4.4: Independent variables for the server benchmark and their tested values

4.3 Results

In this section, the results of the benchmarks outlined in section 4.2 are presented. A discussion of the results with regards to the research questions can be found in

section 4.4

4.3.1 System throughput

In this section, the systems, consisting of multiple clients and a server, are evaluated on their throughput. The system throughput was benchmarked across the configurations described in Table 4.2 for different stream rates, with half of the configurations using the federated system and the other half using the centralized system. Each benchmark was run six times, with the average taken as the result. The stream rate is the combined stream rate of all clients.

A system is considered saturated when its throughput no longer increases with additional load; the corresponding *saturation point* is defined as the highest throughput observed.

In the benchmarks, two different batch sizes were tested, 1000 and 10 000 tuples. Using Equation 4.1, a batch size of 10 000 has a maximum data latency almost 10 times greater than 1000, significantly worsening data freshness. Similarly, Equation 4.6 shows that increasing the batch size typically degrades data safety.

Count sketch

Figure 4.7 presents the average throughput of systems employing the Count sketch under varying client counts, batch sizes, and stream rates. Although higher stream rates were also tested in this system benchmark, the client benchmark (see subsection 4.3.2) demonstrated that clients using the Count sketch could sustain operation up to a stream rate of 350 000 per client. For this reason, the presented results are restricted to stream rate within the feasible range established by the client benchmark. The observed saturation points are interpreted in relation to an estimated bandwidth limit.

With 3 clients, Figure 4.7a, the federated system exhibits superior performance, remaining unsaturated and achieving throughputs of approximately $1.1 \cdot 10^6$ T/s (tuples per second) with a batch size of 1000 and $1.2 \cdot 10^6$ T/s with a batch size of 10 000. In contrast, the centralized system demonstrates constrained throughput capacity. The configuration with a batch size of 1000 reaches saturation at approximately $0.9 \cdot 10^6$ T/s, while increasing the batch size to 10 000 yields a slight throughput improvement, marginally exceeding $0.9 \cdot 10^6$ T/s, without reaching saturation.

When scaling up to 30 clients, Figure 4.7b, the federated systems continue to operate below saturation, reaching $10 \cdot 10^6$ T/s with a batch size of 1000, and $12 \cdot 10^6$ T/s with a batch size of 10 000. Conversely, the centralized system encounters significant performance limitations, saturating at approximately $0.9 \cdot 10^6$ T/s with a batch size of 1000 and at a lower throughput of approximately $0.9 \cdot 10^6$ T/s with a batch size of 10 000.

At 150 clients, Figure 4.7c, the federated system with a batch size of 1000 reaches saturation at approximately $22 \cdot 10^6$ T/s. However, the federated system with a batch size of 10 000 continues to perform without saturation, supporting a throughput of

up to $60 \cdot 10^6$ T/s. The centralized system remains severely limited, saturating at approximately $0.6 \cdot 10^6$ T/s with a batch size of 1000 and $0.6 \cdot 10^6$ T/s with a batch size of 10 000.

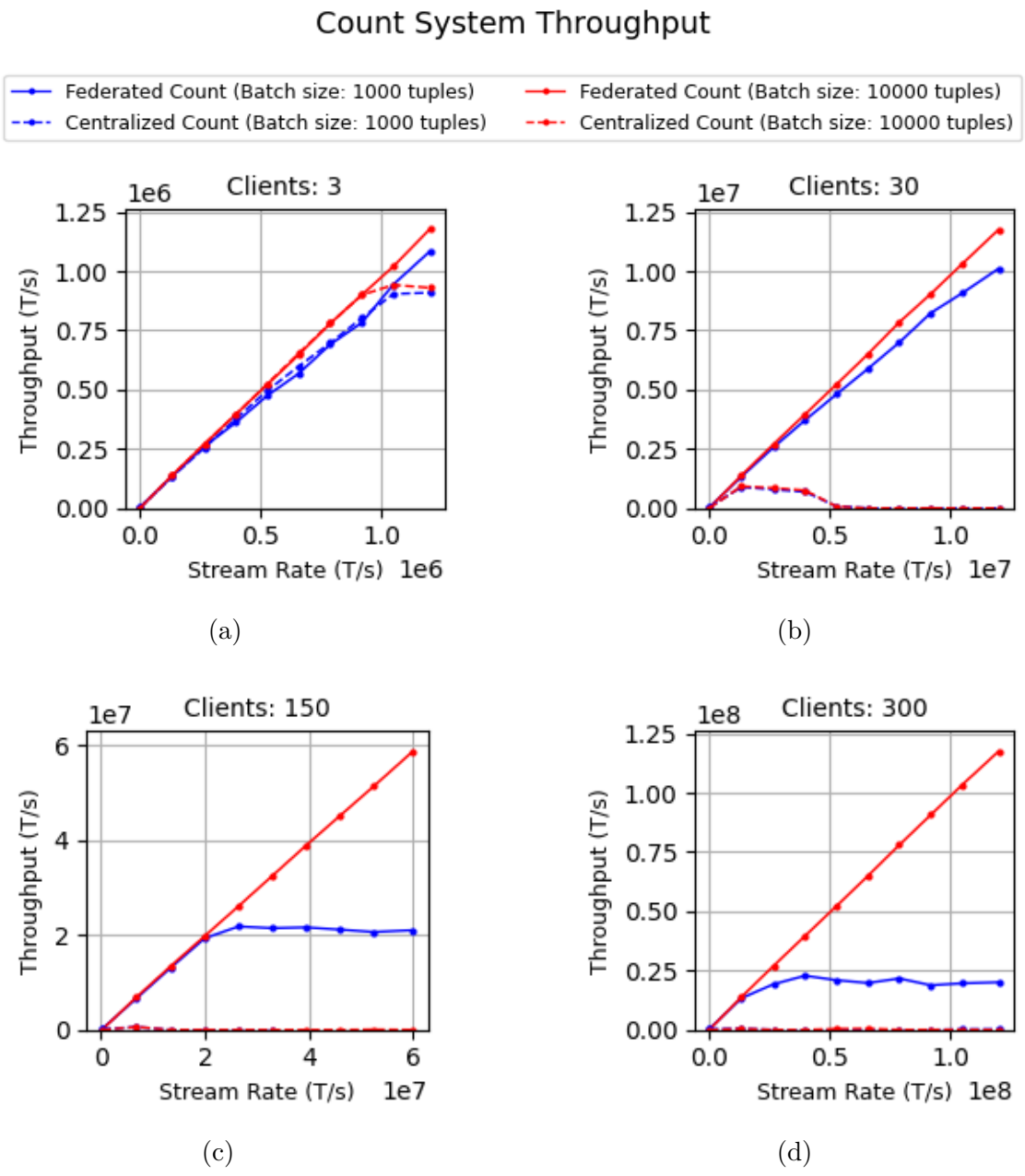


Figure 4.7: The average throughput of the system using the Count sketch with 3, 30, 150, and 300 clients and variable stream rate. The blue lines represent a batch size of 1000 tuples and the red a batch size of 10 000 tuples. The solid lines represent the federated system and the dashed the centralized system.

At the highest tested client load of 300, Figure 4.7d, the federated system sustains throughputs of approximately $23 \cdot 10^6$ T/s with a batch size of 1000 where it reaches its saturation point. The federated system with a batch size of 10 000 remains un-

saturated and achieves a throughput of $120 \cdot 10^6$ T/s. In comparison, the centralized system's performance constraints are pronounced, with the configuration with a batch size of 1000 achieving a throughput of approximately $0.5 \cdot 10^6$ T/s, and the configuration with a batch size of 10 000 managing only $0.5 \cdot 10^6$ T/s.

To investigate if the saturation points were bandwidth related, Equation 4.8 is used. The serialized requests transmitted by the centralized system were measured using Wireshark [35] to approximately 88 000 and 880 000 bits, for batch sizes 1000 and 10 000 respectively. Applying Equation 4.8, with these numbers and a 1 Gbps network link, the maximum throughput is estimated as:

$$MaximumNetworkThroughput^{(1000)}_{\text{Centralized}} = \frac{Bw}{M_{size}} \cdot bs = \frac{10^9}{88000} \cdot 1000 \approx 11.4 \cdot 10^6 \text{ T/s}. \tag{4.9}$$

$$MaximumNetworkThroughput^{(10000)}_{\text{Centralized}} = \frac{10^9}{880000} \cdot 10\,000 \approx 11.4 \cdot 10^6 \text{ T/s}. \tag{4.10}$$

As shown in Table 4.5, all tested centralized configurations reach saturation at throughput levels below the theoretical maximum. This suggests that factors other than network bandwidth are the primary system-level bottlenecks in the centralized architecture.

Measurements indicate that a merge request from the federated system utilizing the Count Sketch with a batch size of 1000 requires, on average, 42 000 bits, whereas a batch size of 10 000 requires approximately 47 000 bits. Applying Equation 4.8, the corresponding maximum throughput supported by a 1 Gbps link can be estimated as follows:

$$MaximumNetworkThroughput^{(1000)}_{\text{Count}} = \frac{10^9}{42\,000} \cdot 1000 \approx 23.8 \cdot 10^6 \text{ T/s} \tag{4.11}$$

for a batch size of 1000 tuples and

$$MaximumNetworkThroughput^{(10000)}_{\text{Count}} = \frac{10^9}{47000} \cdot 10\,000 \approx 212.8 \cdot 10^6 \text{ T/s} \tag{4.12}$$

for a batch size of 10 000. The federated system saturates at $23 \cdot 10^6$ T/s. This is about equal to the estimated maximum network throughput of $23.8 \cdot 10^6$ T/s which means that it is bounded by the available bandwidth.

In contrast, with a batch size of 10 000, the federated configuration remains unsaturated even at its highest observed throughput of $120 \cdot 10^6$ T/s, which is still below the estimated maximum bandwidth. This indicates that in this configuration, bandwidth is not the limiting factor.

KLL sketch

Figure 4.8 illustrates the average throughput of systems utilizing the KLL sketch under varying configurations of clients, batch sizes, and stream rates. The observed saturation points are contextualized against the bandwidth limit. The tested stream rates correspond to values determined by the client benchmark.

In the system with 3 clients, Figure 4.8a, the federated system with a batch size of 10 000 closely tracks the optimal throughput trajectory and remains unsaturated. In contrast, the federated system with a batch size of 1000 saturates at $2.3 \cdot 10^6$ T/s. In the centralized configuration, both systems with batch sizes of 1000 and 10 000 saturate at approximately $2.2 \cdot 10^6$ T/s and $2.7 \cdot 10^6$ respectively.

In the system with 30 clients, Figure 4.8b, the federated system demonstrates superior performance compared to the centralized system. The federated system with a batch size of 10 000 remains unsaturated across all tested stream rates, underscoring its scalability. The federated system with a batch size of 1000 also carries on unsaturated. In contrast, the centralized system experiences saturation: the configuration with a batch size of 1000 saturates at $2.5 \cdot 10^6$ T/s, while the configuration with a batch size of 10 000 saturates at $2.6 \cdot 10^6$ T/s, indicating a significant performance bottleneck.

At 150 clients, Figure 4.8c, the performance disparity between the federated and the centralized system widens. The federated system with a batch size of 10 000 continues to operate without saturation, demonstrating exceptional resilience to high client loads. The federated system with a batch size of 1000 saturates at a throughput of $39 \cdot 10^6$ T/s. The centralized system, however, saturates earlier: the configuration with a batch size of 1000 reaches saturation at approximately $2.2 \cdot 10^6$ T/s, and the configuration with a batch size of 10 000 saturates at approximately $2.4 \cdot 10^6$ T/s.

At the highest tested client count of 300, Figure 4.8d, the federated system maintains a significant performance advantage. The federated system with a batch size of 10 000 reaches a throughput of $415 \cdot 10^6$ T/s without reaching saturation, far surpassing other configurations. The federated system with a batch size of 1000 saturates at $47 \cdot 10^6$ T/s. In the centralized system, the configuration with a batch size of 1000 saturates at $0.6 \cdot 10^6$ T/s, consistent with its performance at 150 clients. The configuration with a batch size of 10 000 saturates at $0.8 \cdot 10^6$ T/s.

To determine if bandwidth limits the maximum throughput of the federated system, we employ the formulation given in Equation 4.8. The serialized message size of merge request sent using the KLL sketch is 10 000 and 12 000 bits for batch sizes 1000 and 10 000 respectively, measured using Wireshark [35]. Applying the equation with a batch size of 1000, a 1 Gbps give an estimate of:

$$MaximumNetworkThroughput_{\text{KLL}}^{(1000)} = \frac{10^9}{10\,000} \cdot 1000 \approx 100 \cdot 10^6 \text{ T/s} \tag{4.13}$$

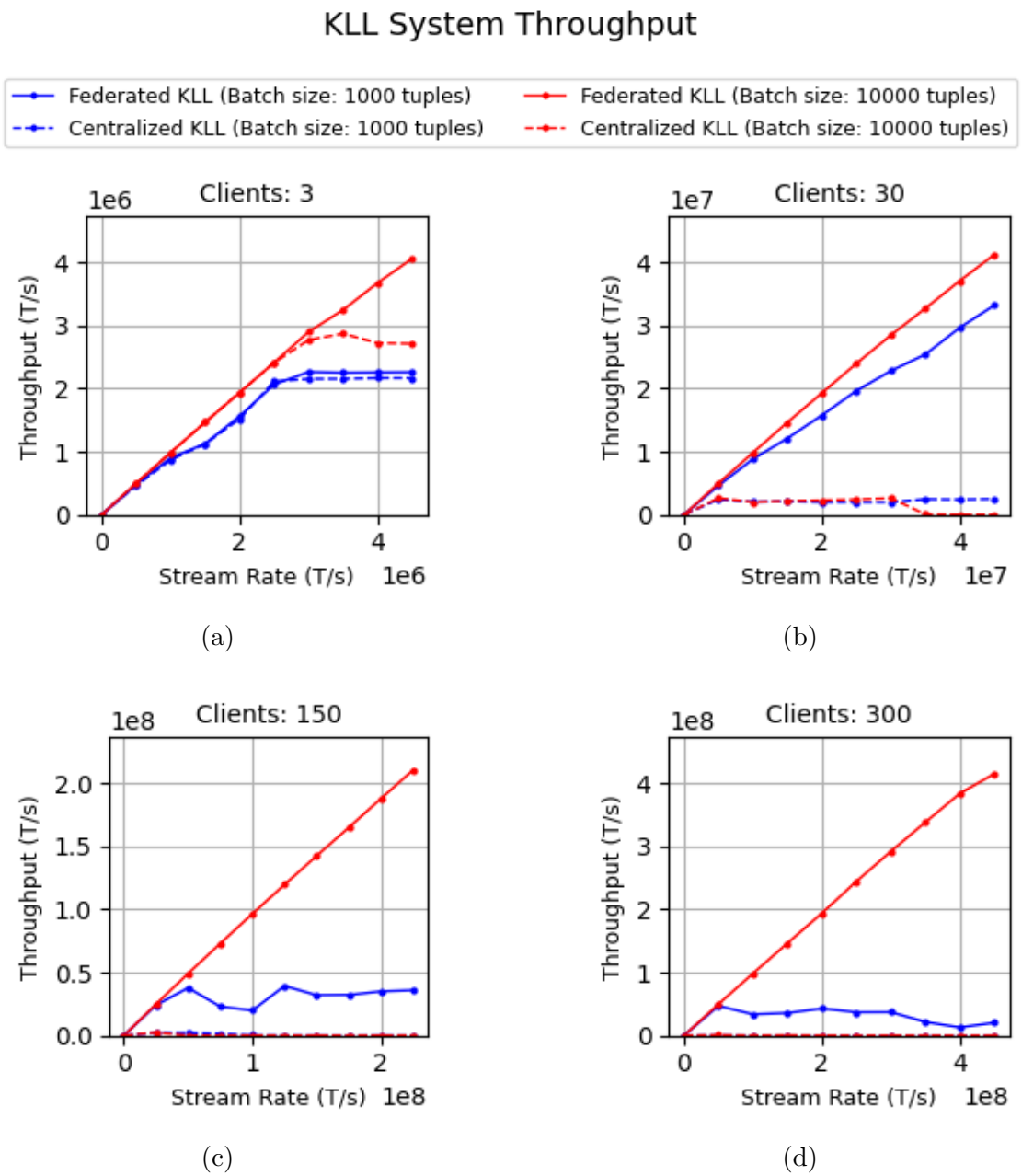


Figure 4.8: The average throughput of the system using the KLL sketch with 3, 30, 150, and 300 clients and variable stream rate. The blue lines represent a batch size of 1000 tuples and the red a batch size of 10 000 tuples. The solid lines represent the federated system and the dashed the centralized system.

For the configuration with a batch size of 10 000 the estimated maximum throughput is:

$$MaximumNetworkThroughput_{\text{KLL}}^{(10000)} = \frac{10^9}{12000} \cdot 10\,000 \approx 833 \cdot 10^6 \text{ T/s} \quad (4.14)$$

Examining the saturation point for the configuration with a batch size of 10 000 tuples, it is around $415 \cdot 10^6$ T/s which is below $781 \cdot 10^6$ T/s, which is why saturation

was not bandwidth related. Similarly, the saturation point for the configuration with a batch size of 1000 is observed at $47 \cdot 10^6$ T/s, which is lower than the estimated maximum bandwidth of $100 \cdot 10^6$ T/s. Therefore, the saturation is not attributed to bandwidth limitations.

The bandwidth of the centralized system is the same as for the Count sketch. Therefore, the result from equations 4.9 and 4.10 can be used, $11.4 \cdot 10^6$ T/s. The highest saturation point that was measured was $2.7 \cdot 10^6$ T/s which is below $11.4 \cdot 10^6$ T/s.

Table 4.5: All observed saturation points and estimated maximum network throughputs

Sketch	Batch size	Clients	Saturation point	Max estimated network throughput
Centralized Count	1000	3	$0.9 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	10 000	3	$0.9 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	1000	30	$0.9 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	10 000	30	$0.9 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	1000	150	$0.6 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	10 000	150	$0.6 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	1000	300	$0.5 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized Count	10 000	300	$0.5 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Federated Count	1000	3	N/A	$23.8 \cdot 10^6$ T/s
Federated Count	10 000	3	N/A	$212.8 \cdot 10^6$ T/s
Federated Count	1000	30	N/A	$23.8 \cdot 10^6$ T/s
Federated Count	10 000	30	N/A	$212.8 \cdot 10^6$ T/s
Federated Count	1000	150	$\mathbf{22 \cdot 10^6}$ T/s	$23.8 \cdot 10^6$ T/s
Federated Count	10 000	150	N/A	$212.8 \cdot 10^6$ T/s
Federated Count	1000	300	$\mathbf{23 \cdot 10^6}$ T/s	$23.8 \cdot 10^6$ T/s
Federated Count	10 000	300	N/A	$212.8 \cdot 10^6$ T/s
Centralized KLL	1000	3	$2.2 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	10 000	3	$2.7 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	1000	30	$2.5 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	10 000	30	$2.6 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	1000	150	$2.2 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	10 000	150	$2.4 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	1000	300	$0.6 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Centralized KLL	10 000	300	$0.8 \cdot 10^6$ T/s	$11.4 \cdot 10^6$ T/s
Federated KLL	1000	3	$2.3 \cdot 10^6$	$100 \cdot 10^6$ T/s
Federated KLL	10 000	3	N/A	$833 \cdot 10^6$ T/s
Federated KLL	1000	30	N/A	$100 \cdot 10^6$ T/s
Federated KLL	10 000	30	N/A	$833 \cdot 10^6$ T/s
Federated KLL	1000	150	$39 \cdot 10^6$ T/s	$100 \cdot 10^6$ T/s
Federated KLL	10 000	150	N/A	$833 \cdot 10^6$ T/s
Federated KLL	1000	300	$47 \cdot 10^6$ T/s	$100 \cdot 10^6$ T/s
Federated KLL	10 000	300	N/A	$833 \cdot 10^6$ T/s

4.3.2 Client throughput

In Figure 4.9 the throughput and saturation point can be seen for both the KLL sketch, Count sketch, and a client buffering the data as in the centralized system. The Count sketch is saturated at about $0.35 \cdot 10^6$ T/s, which is considerably slower than the KLL and the buffering client which saturates at around $30 \cdot 10^6$ T/s. This is because most of the time the KLL sketch is doing minimal work, appending to a list, and sometimes compressing which has a slightly increased computational complexity, while the Count sketch always has to perform multiple hashes, which is more expensive. The Count sketch is fast enough for most use-cases, but might be too slow in very high stream rate situations. There is a minimal difference between the KLL sketch and the centralized system, indicating that the KLL sketch is viable for all stream rates that the centralized system supports.

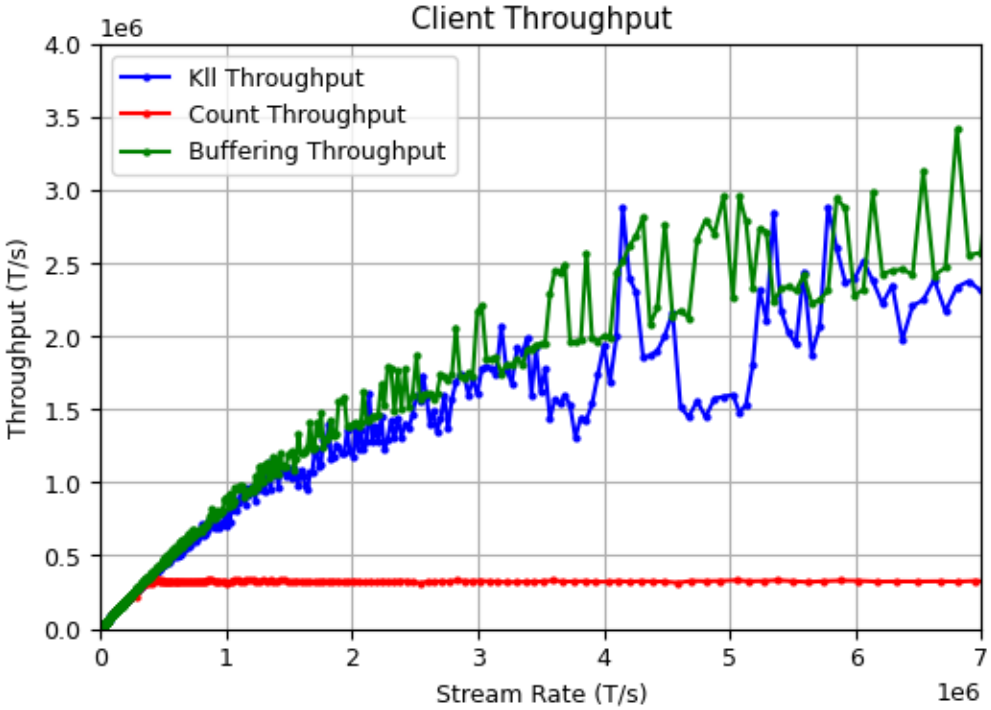


Figure 4.9: Throughput for the client running the KLL sketch, Count sketch and buffering for processing on the server with variable stream rate.

4.3.3 Server throughput

The server's throughput is shown in Figure 4.10 for the Count sketch and KLL sketch with both the federated and centralized systems. The federated Count sketch's throughput increases linearly with the batch size. The federated KLL sketch's throughput increases with the batch size, but at a slower rate than the federated Count sketch. Both centralized systems' throughput are constant when the batch size changes. When the batch size reaches 512 000 tuples, the resulting message in centralized systems becomes larger than the maximum size permitted by a single

RPC message. Since the current implementation does not support message fragmentation, such batches cannot be transmitted. Moreover, multiple messages would introduce additional overhead, further degrading performance.

Regarding saturation, the centralized system reaches a maximum throughput of $1 \cdot 10^6$ T/s for the Count sketch and $4 \cdot 10^6$ T/s for the KLL sketch, independent of batch size. In the federated system, the server supports a throughput of $50 \cdot 10^6$ T/s for the KLL sketch and $130 \cdot 10^6$ T/s for the Count sketch, both observed with a batch size of 1000 tuples. The Count sketch achieves a throughput of approximately $1400 \cdot 10^6$ T/s at a batch size of 10 000 and the KLL sketch reaches approximately $300 \cdot 10^6$ T/s under the same conditions. The server benchmark was conducted for batch sizes of 8000 and 16 000 rather than exactly 10 000, so the reported values for 10 000 are obtained via linear interpolation between these two measurements. For the Count sketch, server throughput grows approximately linearly with batch size, making this interpolation a reasonably accurate estimate. In contrast, the KLL sketch throughput increases in discrete steps due to the way the sketch compresses data. The relevant step could occur anywhere between 8000 and 16 000, which means the interpolated value for 10 000 could differ from the actual throughput, though the exact location of the step is not known.

The server benchmark demonstrates that throughput continues to scale in the federated configurations when tested with batch sizes larger than those used in the system benchmark described in subsection 4.2.3. This suggests that the system benchmark would likewise have continued to scale beyond the tested batch sizes. Moreover, increasing the batch size in the federated configuration raises the maximum estimated network throughput, as defined in Equation 4.8, indicating that the network also scales with batch size.

4.4 Discussion of results

This section provides an in-depth analysis of the benchmark results, examining the behavior and throughput of the implemented systems across different configurations. In addition to answering the research questions stated in section section 4.2, it provides a discussion of implementation choices and their impact on system throughput, reliability, and scalability.

4.4.1 The system benchmark

The throughput evaluation of Count and KLL sketches across federated and centralized system architectures reveals distinct differences in throughput and scalability, with implications for system design under varying system configurations. The federated system consistently outperforms the centralized system in terms of throughput across all tested configurations, as seen in Figure 4.7 and Figure 4.8. The difference in throughput becomes increasingly pronounced as the number of clients increases. This observation is consistent with the inherent advantages of federated architectures. The distributed nature of the federated system enables efficient workload distribution across multiple clients, thereby alleviating the computational bottleneck that

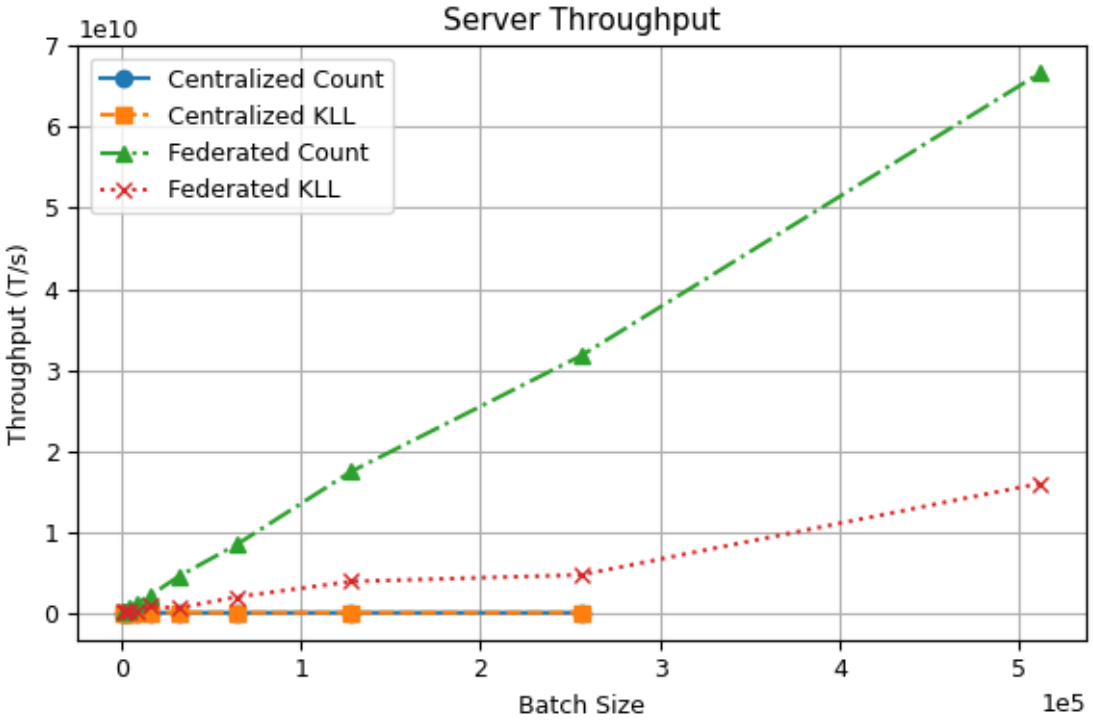


Figure 4.10: Throughput for the centralized and federated server for different batch sizes for the KLL and Count sketch

commonly limits centralized systems.

Analysis of the system benchmarks reveals distinct bottlenecks for each configuration. The centralized system utilizing the Count sketch highest system throughput of $0.9 \cdot 10^6$ tuples per seconds is probably related to the result of the server benchmark, where the Count sketch reached approximately 10^6 tuples per second. Therefore the server is likely the bottleneck. In the case of the centralized KLL sketch, the server benchmark produced a result of $3.5 \cdot 10^6$ tuples per second while the system benchmark reached $2.7 \cdot 10^6$ tuples per second. This probably means that it is limited by the server.

The federated KLL sketch with a batch size of 1000 matches the server benchmark, indicating that server capacity is the limiting factor for this configuration. In contrast, the federated Count sketch with a batch size of 1000 is constrained by network bandwidth rather than server processing. With a larger batch size of 10 000, the KLL sketch in the federated system slightly exceeds the server benchmark. This discrepancy may arise because the server benchmark was not measured for exactly 10 000, so the reported value is an interpolation between nearby measurements, which could introduce error. Meanwhile, the Count sketch at a batch size of 10 000 does not reach saturation, and does not exceed result of the standalone benchmark.

Research Question 1: When does a federated approach outperform a centralized one in terms of throughput? The results demonstrate that in systems with three or more clients the federated system always outperforms the

centralized system, and a batch size of 10 000 achieves a higher throughput than that of a batch size of 1000. Consequently, in applications where retaining unprocessed data on the server is not a requirement, the federated architecture with a large batch size is the recommended choice for maximizing system throughput.

Research Question 3: What are the trade-offs between throughput, data safety, and data freshness at the server? Further analysis of the impact of batch size on throughput reveals divergent trends between the two architectures. In the federated system, a larger batch size of 10 000 consistently achieves higher throughput than a smaller batch size of 1000. This performance advantage arises because, while the computational workload for clients remains equivalent in both configurations, the server's processing burden is significantly reduced with the larger batch size. In contrast, the centralized system exhibits better throughput with a smaller batch size of 1000, except in configurations with three clients.

A smaller batch size improves data freshness and enhances data safety. Equation 4.1 shows that freshness is inversely proportional to batch size, while data safety degrades as batch size grows as seen in subsection 4.2.2. However, empirical results in Figure 4.7 and Figure 4.8, most evident in Figure 4.7b and Figure 4.8b, reveal that smaller batch sizes impose a substantial throughput penalty in federated systems. In the centralized system, increasing the batch size does not improve throughput, which means that there is no trade-off between throughput, data freshness, and data safety. Smaller batch sizes maximize data freshness, data safety, and throughput, up to the point where processing overhead for small batches becomes too significant. Although overall throughput in the centralized system is severely limited compared to the federated systems, it can maintain high data freshness and safety as long as saturation does not occur. In federated configurations, throughput can improve further with larger batch sizes, but this comes at the cost of reduced freshness and safety. Thus, selecting an optimal batch size involves balancing these competing factors. The most effective strategy is to use the smallest batch size the system can sustain without saturating. Rather than prescribing a fixed value, Table 4.6 summarizes relative performance across configurations, providing guidance for selecting an appropriate batch size.

While much of the analysis focuses on the impact of batch size, it is also interesting to compare the centralized and federated architectures themselves. The federated system generally outperforms the centralized approach in throughput and scalability. However, in cases with lower client counts and lower stream rates, their performance is roughly equivalent. In these scenarios, the centralized system can be advantageous due to its simpler architecture, and the ability to retain unprocessed data on the server. Thus, even when performance is comparable, centralized designs may be preferred in settings where minimizing complexity and cost, or accessing raw server data, is more important than maximizing throughput.

In conclusion, the federated system demonstrates superior throughput and scalability, particularly in systems with a high number of clients, and performs better with larger batch sizes. In contrast, the centralized system, while effective in scenarios with fewer clients, is generally outperformed. In the federated system, maintaining a

minimal batch size is not always feasible; therefore, the smallest batch size that the system can sustain without compromising performance should be selected. These findings provide critical guidance for selecting and configuring system architectures based on specific operational requirements.

Table 4.6: Overview of trade-offs of different system details. The number of ↑ and ↓ indicates positive and negative comparative performance on a specific metric.

Compute Location	Sketch type	Batch Size	Metrics		
			Throughput	Data safety	Data freshness
Federated	KLL	Small batch	↑↑↑	↑	↑↑
		Large batch	↑↑↑↑	↓	↓
	Count	Small batch	↑	↑	↑↑
		Large batch	↑↑	↓	↓
Centralized	KLL	Small batch	↓	↑	↑
		Large batch	↓	↓	↓↓
	Count	Small batch	↓↓	↑	↑
		Large batch	↓↓	↓	↓↓

4.4.2 The client benchmark

The results presented in Figure 4.9 align with expectations based on the computational characteristics of the respective algorithms. The `Add` operation in the Count sketch incurs a higher computational cost due to its requirement to hash the input value 11 times and subsequently increment or decrement 10 corresponding values. In contrast, the `Add` operation in the KLL sketch involves appending the value to the end of a list, which is a relatively inexpensive operation, though it occasionally triggers the `Compress` function, which introduces additional computational overhead. The centralized system exhibits superior throughput, as its `Add` operation solely involves appending to a buffer, a process analogous to that of the KLL sketch but without the intermittent compression step. Consequently, it is reasonable that the KLL sketch performs slightly slower than the centralized system due to these operational similarities and the additional overhead of compression.

Research Question 2: When is it feasible to offload computation to the edge? The answer to research question 2 depends on the sketching algorithm and system constraints. For the KLL sketch, the results indicate that its throughput is comparable to that of the centralized system, rendering federated computation consistently feasible in almost all configurations. Therefore, federating the computation of the KLL sketch minimizes server overhead without compromising client throughput.

The Count Sketch, by contrast, incurs a higher per-tuple cost because its `Add` operation involves multiple memory accesses and arithmetic steps, making it more demanding than the KLL sketch. Consequently, client throughput saturates around $0.35 \cdot 10^6$ T/s. This means that federating the Count Sketch computation is practical up to that throughput. At higher stream rates, the relatively more costly `Add`

operation, compared to the other configurations, shifts the client into becoming the primary bottleneck. In such scenarios, centralization may be required to alleviate the client workload when employing a Raspberry Pi 5, as described in subsection 4.1.1.

In summary, federated computation is consistently feasible with the KLL sketch across all evaluated workloads. By contrast, the more costly `Add` operation of the Count sketch constrains its feasibility to moderate stream rates. Once this limit is exceeded, clients become the bottleneck, and centralization is required to prevent client overload.

4.4.3 The server benchmark

Since the `Merge` operation for a Count sketch consists of adding two matrices together, its execution time remains constant regardless of the batch size. This behavior is reflected in Figure 4.10. In contrast, the KLL sketch is expected to benefit from increased batch sizes, as its memory footprint scales sub-linearly with the number of elements it contains. Consequently, the number of elements incorporated into the server-side sketch grows more slowly than the total number of elements merged, which explains that the KLL sketch achieves a higher throughput at larger batch sizes in Figure 4.10. Additionally, the time required for a merge may vary depending on how often the `Compress` operation is required, which depends on the number of tuples added to the sketch and some variation depending on the randomness.

The throughput of the centralized systems, by contrast, is independent of the batch size, which aligns with expectations when communication overhead is excluded from the benchmark. In the centralized system, each element of the dataset is processed individually using the `Add` operation, regardless of how many elements are grouped together in a batch. This means that doubling or tripling the batch size simply increases the number of `Add` operations performed proportionally, but the time spent per element does not change. As a result, the throughput remains the same, independent of the batch size.

Research Question 3: What are the trade-offs between throughput, data safety, and data freshness at the server? The results suggest that the trade-offs depend on the particular sketching algorithm used and where the computation takes place.

In a centralized architecture, the analysis demonstrates that no trade-offs exist among the evaluated metrics. Specifically, system throughput is highest at smaller batch sizes and slightly decreases as the batch size increases. Thus, smaller batch sizes improve throughput, data safety, and data freshness up to a point, beyond which the overhead of sending many small messages begins to degrade performance. Batch sizes small enough to determine the point where the batch size is too small were not tested in this thesis. But within the ranges tested, frequent merging operations neither compromise throughput nor degrade system efficiency. Moreover, frequent merging enhances data safety by reducing the risk of data loss and improves data freshness by ensuring that the most recent data states are consistently reflected. Consequently, frequent merging is consistently associated with improved outcomes

across all evaluated dimensions in the centralized system.

In contrast, the federated system necessitates a more nuanced consideration of trade-offs, as both the KLL sketch and the Count sketch exhibit increased throughput for larger batch sizes. While fresher data and higher data safety are universally desirable, achieving these goals often requires processing smaller batch sizes, which in turn increases the computational demands on the server. Thus, the ideal configuration hinges on the server's excess computational capacity. A smaller batch size is generally preferable, as it maximizes data safety and freshness, with the primary drawback being increased server workload. Therefore, the system should be configured to adopt the smallest batch size that the server can sustainably support, balancing computational demands with the benefits of enhanced data safety and freshness.

5

Related work

A range of prior studies has explored distributed sketching and federated computation, highlighting the relevance and applicability of these techniques in various domains. This chapter reviews selected related works that address problems and methodologies closely aligned with the focus of this thesis, providing context and identifying gaps that motivate this thesis.

Pebbles: Leveraging Sketches for Processing Voluminous, High Velocity Data Streams Testing and benchmarking sketches were done in *Pebbles: Leveraging Sketches for Processing Voluminous, High Velocity Data Streams* [36]. The paper presents Pebble, a sketch that summarizes a data stream and retains the order of the observed data. The benchmarks performed in the paper were bandwidth usage and energy consumption by the edge device and throughput, among others, at the server. The independent variables were different data sets and different stream generators. While their work primarily focused on evaluating the performance of individual components and the development of a novel sketching algorithm, this thesis approach differs in several key respects. In this thesis, both individual components and the overall system throughput are benchmarked, with particular emphasis on scalability. Rather than proposing new sketching techniques, existing well-established sketches methods are utilized and concentrate on the surrounding system architecture. Specifically, this thesis focuses on the impact of computation placement within distributed environments.

Benchmarking Distributed Stream Data Processing Systems The article *Benchmarking Distributed Stream Data Processing Systems* [37] also benchmarks distributed streaming. They introduce a benchmarking framework for evaluating distributed stream processing systems, addressing the need for scalable and efficient stream analysis. The framework focuses on measuring throughput and latency, particularly for windowed operations, which are essential in real-time analytics. Their workloads are designed based on real-world industrial use cases. Similarly to this thesis, they generate streams on-device with a configurable generation rate. The study measures the saturation point of a system, which aligns with this thesis approach. They apply their framework to Apache Storm, Apache Spark, and Apache Flink. In contrast, this work introduces a new implementation designed specifically for the purposes of the study and focuses on sketching.

DRIVEN: A framework for efficient Data Retrieval and clustering in Vehicular Networks The *DRIVEN* framework, presented in [38], is designed for efficient data collection and clustering of vehicular data. DRIVEN employs lossy compression to reduce data volume on edge devices. In contrast to the approach proposed in this thesis, which utilizes sketches for data analysis, DRIVEN uses piecewise linear approximation (PLA) compression. PLA approximates consecutive data points as line segments, starting a new segment when a line cannot approximate a series of points within the specified maximum error. Although PLA is a form of lossy compression, DRIVEN supports decompression to recover data, which is not feasible with sketches. DRIVEN is tailored specifically for data clustering, whereas the implementation presented in this supports a broader range of data analysis tasks, depending on the sketch used.

MAD-C: Multi-stage Approximate Distributed Cl-uster-combining for obstacle detection and localization The research paper *MAD-C: Multi-stage Approximate Distributed Cluster-combining for obstacle detection and localization* describes *MAD-C* a method for approximate data summarization for object detection through clustering [39]. MAD-C uses *LIDAR* a sensor that measures distances and creates models of the surrounding environment using lasers. The models are clustered summarizations of LIDAR point clouds computed locally at the LIDAR sensor. Multiple clustering algorithms can be used in MAD-C, and the data is clustered on each sensor in a federated manner. The resulting clusters are combined concurrently, and clusters which overlap or are close together are merged to represent larger objects. Their findings indicate that MAD-C efficiently reduces both bandwidth and processing complexity while remaining accurate. Furthermore, they demonstrate that the system scales linearly with the number of detected objects. Their study highlights the benefits of federated computation in improving scalability through clustering. Similarly, this thesis demonstrates that federated computation can yield scalability benefits, but in a sketching context.

Communication-Efficient Federated Learning with Sketching The article *Communication-Efficient Federated Learning with Sketching* (*FedtchSGD*) [40] investigates the use of sketches to compress client-to-server gradient updates in federated learning. The study benchmarks per-round communication, number of rounds to convergence, and the effect of compression parameters, such as sketch size, on learning performance. While the primary focus of FetchSGD is on the trade-off of minimizing communication cost and optimizing convergence and validating the accuracy of compressed updates, this thesis differs in several key aspects. Rather than developing new sketching techniques or targeting convergence, system throughput is benchmarked. Established, mergeable sketches are adopted to study the impact of computation placement, comparing centralized versus federated update aggregation. Furthermore, this thesis investigates how batch size and compute location influence data safety, data freshness, and throughput, analyzing the trade-offs between these factors. In short, whereas FetchSGD evaluates sketching as a communication-efficient tool for federated learning, this thesis applies sketching techniques to a systems-level analysis of throughput, safety, and freshness in distributed environ-

ments.

6

Conclusion

This chapter summarizes the key findings of the thesis and outlines potential directions for future work.

6.1 Key findings

The findings of the thesis demonstrate that federated systems consistently achieve higher throughput and better scalability as the number of clients increases. This advantage is primarily due to the decentralized nature of the federated approach, which distributes the computational load and utilizes the available computational resources better.

The system benchmark indicates that the performance of federated system improves with larger batch sizes, which reduce server side processing and enables scalability. However, this comes at the cost of reduced data freshness and data safety. In contrast, the centralized system does not benefit from larger batch sizes, enhancing data freshness and safety without impacting throughput within the tested ranges. Thus, the centralized system remains a viable choice in small scale or resource constrained deployments, federated architectures are generally preferred for high-throughput, scalable applications.

Client-side performance evaluations demonstrate that federated computation is consistently viable for the KLL sketch, exhibiting throughput comparable to that of buffering the data. Conversely, the Count sketch supports federated computation in high stream rate environments but it fails to achieve equivalent throughput to the KLL sketch and buffering the data. The reason for this is that on average the Count sketch requires more operations per `Add` relative to the KLL sketch and buffering, as hashing is more expensive than appending to a list.

Server benchmarks further highlight the benefits of larger batch sizes in the federated system. While both the Count sketch and the KLL sketch exhibit improved throughput scaling with increasing batch size, the Count sketch scales more efficiently, achieving near-perfect linearity, while the KLL sketch shows step-wise increments. Meanwhile, the throughput of centralized processing remains constant across all tested batch sizes.

In conclusion, federated sketching is recommended for achieving high throughput and scalability in distributed data sketching environments where data is generated

at the edge. To appropriately balance throughput, data safety, and freshness, it is advised to select the smallest sustainable batch size.

6.2 Future work

This section outlines potential directions for extending and building upon the results of this thesis. This includes, leveraging parallelism to improve throughput and scalability, exploring different network topologies, analyzing performance on real networks, and benchmarking existing frameworks.

6.2.1 Parallelism

Neither the Count or KLL sketches' merge operation is inherently *atomic* under concurrent execution. This thesis' implementation relies on a mutual exclusion lock to prevent race conditions when multiple merge requests are received by the server simultaneously. While a divide-and-conquer strategy could improve concurrency by grouping incoming merge requests into pairs and merging them in parallel, this approach was not adopted due to its implementation complexity. Specifically, it would require the server to track incoming requests and determine execution timing, adding significant complexity to system design.

In contrast, *parallel sketches*, which inherently support concurrent merging operations, eliminate the limitations imposed by sequential execution. Methods for increasing concurrency in sketching exist [41]–[44]. Although, they are less well-documented than the Count and KLL sketch, which would have added significant implementation difficulty. Nevertheless, evaluating these concurrent techniques could yield valuable insights and represents a promising direction for future work.

In this thesis, each tuple contained only a single element, but in general, a tuple can have multiple elements. Suppose a tuple has several elements, and each element has its own sketch. If the `Add` or `Merge` operations are applied to each element one after another, the total time for processing the tuple is simply the sum of the times for each individual element. However, because the sketches are independent, these operations could also be performed concurrently, potentially speeding up processing by taking advantage of multi-threading.

6.2.2 Network topologies and deployment

One promising avenue for future work involves exploring alternative system topologies, such as tiered or mesh networks. For example, a tiered network could consist of multiple intermediate servers, each responsible for merging data from a subset of clients. These intermediate servers would merge their clients' data locally and subsequently forward the merged sketches to a higher-level server for further merging. Such a structure could reduce the computational and communication burden on the central server.

Alternatively, a mesh network would enable direct communication between clients,

facilitating decentralized solutions. In the context of the truck use case, for instance, a truck with available computational resources could act as a temporary server, receiving unprocessed data from nearby trucks and performing merges on their behalf. Investigating these alternative topologies could yield more nuanced insights into the optimal placement of computation and communication resources under different operational scenarios.

Moreover, testing on a real network with more variability would be more similar to what a deployment of a similar system would experience; for example, for Volvo, the trucks would be scattered over large distances and would have to wirelessly transmit the data. Testing such a setup would also give interesting results, regarding how a well a deployment would work in real world conditions.

6.2.3 Different sketches

Evaluating a broader range of sketching algorithms would contribute to a more comprehensive understanding of their applicability across diverse conditions. This thesis covered static and dynamic sized sketches, utilizing both hashing and randomness. One could, for example, evaluate similar sketches that provide different accuracy and error guarantees, such as comparing the Count sketch versus the Count-Min sketch, or the KLL sketch versus the GK sketch (GreenwaldKhanna) (deterministic quantile summary with worst-case guarantees) [5].

Furthermore, there are types of summaries and underlying data structures not explored in our work that could extend the study to broader applications. These include geometric sketches (ε-kernels) for spatial data and graph sketches for connectivity estimation [5]. Investigating these could provide practical insights into performance, accuracy, and applicability across diverse data domains.

6.3 Closing Remarks

This thesis has examined the trade-offs between performing sketching at the edge and at the server, with particular attention to throughput, data freshness, and data safety. The findings demonstrate that sketching at the edge can substantially increase throughput, particularly with larger batch sizes, while smaller batches are preferable when data freshness or data safety is prioritized. The recommended strategy is to select the smallest batch size that avoids saturating the system.

By articulating these trade-offs, this work highlights that the effectiveness of sketching depends not only on the algorithm itself but also on how and where it is deployed. While this thesis intentionally used plain implementations to showcase fundamental trade-offs and basic functionality, it is worth noting that a variety of ready-to-use frameworks and platforms, such as Apache DataSketches [45], Apache Kafka [46], Apache Flink [47], and Redis [48], already exist to simplify deployment. These platforms allow practitioners to build robust streaming and sketching pipelines without starting from scratch, bridging the gap between experimental analysis and production-ready systems.

In this way, the results not only provide guidance on algorithmic choices and batch sizing but also underscore the broader principle that system-aware design and practical tooling go hand in hand in achieving scalable, reliable data processing at the edge.

Bibliography

[1] *Data growth worldwide 2010-2028*, en, https://www.statista.com/statistics/871513/worldwide-data-created/, Accessed: 2024-11-26.

[2] M.-S. Chen, J. Han, and P. S. Yu, "Data mining: An overview from a database perspective," *IEEE Transactions on Knowledge and data Engineering*, vol. 8, no. 6, pp. 866–883, 2002. DOI: 10.1109/69.553155.

[3] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, and M. Data, "Practical machine learning tools and techniques," in *Data mining*, Elsevier Amsterdam, The Netherlands, vol. 2, 2005, pp. 403–413.

[4] M. van Steen and A. Tanenbaum, *Distributed Systems*. Maarten Van Steen, 2023, ISBN: 9789081540636. [Online]. Available: https://www.distributed-systems.net/index.php/books/ds4/.

[5] G. Cormode and K. Yi, *Small summaries for big data*, en. Cambridge, England: Cambridge University Press, Nov. 2020. DOI: 10.1017/9781108769938.

[6] A. Bharadwaj and G. Cormode, "An introduction to federated computation," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2448–2451.

[7] G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 13–24. DOI: 10.5555/1083592.1083598.

[8] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, 2016, pp. 71–78. DOI: 10.1109/FOCS.2016.17.

[9] *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*, https://eur-lex.europa.eu/eli/reg/2016/679/oj, Official Journal of the European Union, L 119, 4 May 2016, 2016.

[10] G. Cormode, "Data sketching: The approximate approach is often faster and more efficient.," *Queue*, vol. 15, no. 2, pp. 49–67, Apr. 2017, ISSN: 1542-7730. DOI: 10.1145/3084693.3104030.

[11] W. Xu, H. Zhou, N. Cheng, F. Lyu, W. Shi, J. Chen, and X. Shen, "Internet of vehicles in big data era," *IEEE/CAA Journal of Automatica Sinica*, vol. 5, no. 1, pp. 19–35, 2017. DOI: 10.1109/JAS.2017.7510736.

[12] S. Susnjara and I. Smalley. "What is a content delivery network (cdn)?" Accessed: 2025-08-11. (2024), [Online]. Available: https://www.ibm.com/think/topics/content-delivery-networks.

[13] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data streams: A review," *ACM Sigmod Record*, vol. 34, no. 2, pp. 18–26, 2005. DOI: 10.1145/1083784.1083789.

[14] D. J. Hand, "Principles of data mining," *Drug safety*, vol. 30, no. 7, pp. 621–622, 2007. DOI: 10.2165/00002018-200730070-00010.

[15] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," 1, vol. 2, New York, NY, USA: Association for Computing Machinery, Feb. 1984, pp. 39–59. DOI: 10.1145/2080.357392.

[16] D. K. Sharma, B. Tokas, and L. Adlakha, "Chapter 2 - deep learning in big data and data mining," in *Trends in Deep Learning Methodologies*, ser. Hybrid Computational Intelligence for Pattern Analysis, V. Piuri, S. Raj, A. Genovese, and R. Srivastava, Eds., Academic Press, 2021, pp. 37–61, ISBN: 978-0-12-822226-3. DOI: 10.1016/B978-0-12-822226-3.00002-7.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," en, *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. DOI: 10.1145/362686.362692.

[18] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, Jan. 2004, ISSN: 0304-3975. DOI: 10.1016/s0304-3975(03)00400-6.

[19] C. Buragohain and S. Suri, "Quantiles on streams," in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 2235–2240, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_290.

[20] E. D. Kostopoulos, G. C. Spyropoulos, and J. K. Kaldellis, "Real-world study for the optimal charging of electric vehicles," en, *Energy Rep.*, vol. 6, pp. 418–426, Nov. 2020. DOI: 10.1016/j.egyr.2019.12.008.

[21] The Go Authors, *Documentation - The Go Programming Language — go.dev*, https://go.dev/doc/, [Accessed 27-03-2025].

[22] gRPC Authors, *gRPC: A high-performance, open-source universal RPC framework*, https://grpc.io, Accessed: 2025-05-06, 2025.

[23] *Protocol Buffers — protobuf.dev*, https://protobuf.dev/, [Accessed 03-03-2025].

[24] *GitHub - spaolacci/murmur3: Native MurmurHash3 Go implementation — github.com*, https://github.com/spaolacci/murmur3, [Accessed 29-05-2025].

[25] *Handling panics*, https://go.dev/ref/spec#Handling_panics, [Accessed 17-03-2025].

[26] *Interceptors*, https://grpc.io/docs/guides/interceptors/, [Accessed 17-03-2025].

[27] G. Bruhn and J. Mentzer, *GitHub - bruhng/distributed-sketching: Our master thesis where we implement and research distributed sketching! — github.com*, https://github.com/bruhng/distributed-sketching, [Accessed 24-04-2025], 2025.

[28] *PVS - Passive Vehicular Sensors Datasets — kaggle.com*, https://www.kaggle.com/datasets/jefmenegazzo/pvs-passive-vehicular-sensors-datasets, [Accessed 18-02-2025].

[29] Apache DataSketches, *DataSketches / — datasketches.apache.org*, https://datasketches.apache.org/docs/KLL/KLLAccuracyAndSize.html, [Accessed 03-07-2025].

[30] D. P. Bertsekas and J. N. Tsitsiklis, *Introduction to Probability*. Athena Scientific, 2008, vol. 1, ISBN: 978-1-886529-23-6.

[31] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[32] B. N. Chun and A. McNabb, *Google Code Archive - Long-term storage for Google Code Project Hosting. — code.google.com*, https://code.google.com/archive/p/parallel-ssh/, [Accessed 04-04-2025].

[33] D. M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," in *Proceedings of the 1989 ACM SIGCOMM Conference on Communications Architectures and Protocols*, ACM, 1989, pp. 1–14. DOI: 10.1145/75246.75248.

[34] *Encoding — protobuf.dev*, https://protobuf.dev/programming-guides/encoding/, [Accessed 18-08-2025].

[35] W. Foundation, *Wireshark*, https://www.wireshark.org/, [Accessed 23-09-2025].

[36] T. Buddhika, S. L. Pallickara, and S. Pallickara, "Pebbles: Leveraging sketches for processing voluminous, high velocity data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2005–2020, 2021. DOI: 10.1109/TPDS.2021.3055265.

[37] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1507–1518. DOI: 10.1109/ICDE.2018.00169.

[38] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, M. Papatriantafilou, and A. C. Koppisetty, "Driven: A framework for efficient data retrieval and clustering in vehicular networks," *Future Generation Computer Systems*, vol. 107, pp. 1–17, 2020. DOI: 10.1016/j.future.2020.01.050.

[39] A. Keramatian, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Mad-c: Multi-stage approximate distributed cluster-combining for obstacle detection and localization," *Journal of Parallel and Distributed Computing*, vol. 147, pp. 248–267, 2021. DOI: 10.1016/j.jpdc.2020.08.013.

[40] D. Rothchild, A. Panda, E. Ullah, N. Ivkin, I. Stoica, V. Braverman, J. Gonzalez, and R. Arora, "FetchSGD: Communication-efficient federated learning with sketching," in *Proceedings of the 37th International Conference on Machine Learning*, H. D. III and A. Singh, Eds., ser. Proceedings of Machine Learning Research, vol. 119, PMLR, 2020, pp. 8253–8265. [Online]. Available: https://proceedings.mlr.press/v119/rothchild20a.html.

[41] A. Rinberg, A. Spiegelman, E. Bortnikov, E. Hillel, I. Keidar, L. Rhodes, and H. Serviansky, "Fast concurrent data sketches," *ACM Trans. Parallel Comput.*, vol. 9, no. 2, Apr. 2022, ISSN: 2329-4949. DOI: 10.1145/3512758.

[42] C. Stylianopoulos, I. Walulya, M. Almgren, O. Landsiedel, and M. Papatriantafilou, "Delegation sketch: A parallel design with support for fast and accurate concurrent operations," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10.1145/3342195.3387542.

[43] V. Jarlow, C. Stylianopoulos, and M. Papatriantafilou, "QPOPSS: Query and parallelism optimized Space-Saving for finding frequent stream elements," en, *J. Parallel Distrib. Comput.*, vol. 204, no. 105134, p. 105 134, Oct. 2025. DOI: 10.1016/j.jpdc.2025.105134.

[44] S. Elias Zada, A. Rinberg, and I. Keidar, "Quancurrent: A concurrent quantiles sketch," in *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA 23, ACM, Jun. 2023, pp. 15–25. DOI: 10.1145/3558481.3591074.

[45] Apache DataSketches, *DataSketches | — datasketches.apache.org*, https://datasketches.apache.org/, [Accessed 27-05-2025].

[46] Apache Kafka, *Kafka | — kafka.apache.org*, https://kafka.apache.org/, [Accessed 27-05-2025].

[47] Apache FLink, *Flink | — flink.apache.org*, https://flink.apache.org/, [Accessed 27-05-2025].

[48] Redis, *Redis - The Real-time Data Platform — redis.io*, https://redis.io, [Accessed 21-08-2025].