FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Distributed monitoring of web server metrics
Student:	Daniel Sedlák
Supervisor:	Ing. Tomáš Kvasnička
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The aim of this thesis is to design and implement an application to collect, aggregate, store and visualise various metrics from a large number of web servers. More broadly, the target is to build a distributed log/metrics pipeline helping with infrastructure performance validation, debugging and health monitoring.
In the thesis apply standard SE methods and consider the following requirements.

Collected data should include at least:
- timestamp and client IP for identification
- HTTP and cache (if used) statuses
- response time and size
- TCP RTT

Minimum supported aggregations are:
- sum
- avg
- percentile

Pay special attention to the following caveats:
- distributed nature for easy scalability and seamless failover
- reasonable space requirements
- time complexities of required aggregation operations
- network partitions

Optionally, the solution should be generic enough to support DNS/VPN/ and other servers and should allow for simple alerting capabilities.

Electronically approved by Ing. Michal Valenta, Ph.D. on 9 December 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Distributed monitoring of web server metrics

Daniel Sedlák

Department of Software Engineering
Supervisor: Ing. Tomáš Kvasnička

May 3, 2021

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on May 3, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Sedlák, Daniel. *Distributed monitoring of web server metrics.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce si klade za cíl navrhnout a implementovat aplikaci pro shromažďování, agregaci, ukládání a vizualizaci různých metrik z velkého počtu webových serverů. Naše sada aplikací, které budeme vyvíjet, je navíc dobře škálovatelná horizontálně i vertikálně, stejně jako replikovaná a odolná vůči chybám.

Klíčová slova distribuovaný, webový server, škálovatelný, monitorování, golang

Abstract

This thesis aims to design and implement an application to collect, aggregate, store, and visualize various metrics from a large number of web servers. Moreover, our application stack that we will develop is well scalable horizontally and vertically, as well as replicated and fault-tolerant.

Keywords distributed, web-server, scalable, monitoring, golang

Contents

List of Figures

List of Tables

List of Listings

Introduction

Firstly I would like to thank you for choosing my bachelor thesis as a source of information. I will do my best to explain to you all the details about metrics collecting and pipeline scalability.

This bachelor thesis aims to design and implement an application to collect, aggregate, store, and visualize various metrics from a large number of web servers. More broadly, the target is to build a distributed log or metrics pipeline helping with infrastructure performance validation, debugging, and health monitoring. I will guide you through all the steps of setting up distributed metrics monitoring. This thesis focuses only on web servers, but it can be applied in many areas, including DNS, streaming, and others. I pay special attention to scalability and distributed system properties.

We will start by explaining the basics of monitoring, metrics that can be monitored, and continue with caveats and clustering. These basics are prerequisites for later topics that we will discuss. They will be a building block for subsequent topics where we will start developing our knowledge about distributed monitoring of web-server metrics. Then we will go through useful tools that you can use for monitoring. Also, we will discuss the pros and cons of the tools. Then, we will talk a little bit about real-life stacks used by familiar tech giants. In the second half of this thesis, I will show you a practical implementation of the application stack that I have chosen, and we will discuss the reasons behind my decisions. Moreover, we will look at implementing the client in Golang, and we will send our data to the ClickHouse cluster. Furthermore, we will discuss possible errors that I encountered during implementation. Also, we will be monitoring the error log and processing it in the ELK cluster.

We are happy to say that a specific company running a worldwide CDN infrastructure adopted our solution. This provided us with real-life test scenarios as well as experience with production deployment and maintenance.

Chapter 1

About this thesis

In this chapter, we will discuss our goal during this thesis, and we will talk about the structure of this thesis. Moreover, this chapter is an extension of the introduction to ensure that we understand the situation and what will happen in the following chapters.

1.1 Expected results

The big picture end of this thesis is represented by implementing a monitoring pipeline that can be extended and scalable. This means that we are in the first place interested in distributed, scalable, and fault-tolerant, and if this research brings up something positive, we also want to try to dive deeper into the practical side of things. This means that we want to gain as much knowledge as we can about several topics like monitoring, distributed systems, and scalability. At the end of the research, we will want to look at open-source software tools that can be used for our purpose. These and many others represent required knowledge to at least know where to start with our analysis. After getting familiar with the necessary background, we want to focus on the implementation part. We want to use the acquired knowledge to implement a pipeline for our purpose.

1.2 Thesis structure

Here we will briefly describe the structure of this thesis. It will consist of three main parts: research of monitoring, research of monitoring tools, and implementation. Each one of these parts is dedicated to one entire chapter with several subsections. Also, each of these parts could be covered in a separate thesis; therefore, we sometimes reduce the provided information only to the essentials.

Chapter 2

Research of monitoring

Understanding the state of your company's infrastructure is essential for ensuring the reliability and stability of your services. Information about your web-server's health and performance helps your team react to issues or potential bottlenecks and gives them the soundness to make changes with confidence. One of the best ways to gain this insight is with a robust monitoring system that gathers metrics, visualizes data, and alerts administrators when things appear to be broken or just partially working[1].

Monitoring is the process of collecting, aggregating, and analyzing values gathered from the web-server to improve awareness of your component's characteristics and behavior. In the computer world, monitored subjects can be a web server, DNS server, database, and many more. One can use monitored data for infrastructure optimization, debugging particular errors within the infrastructure, or just for marketing purposes to present some results.

Monitored data or just metrics represent the raw measurement of your web-server that can be collected or observed. These values can be collected from the operating systems or directly from logs of your web-server. At first glance, it is not always straightforward what to monitor. You can watch the frequency of the type of syscalls or observe the frequency of active connections. Metrics can be divided into several categories like host-based metrics (CPU, memory, disk space processes), application metrics (error and success rate, service failures and restarts, performance and latency responses, resource usage), network and connectivity metrics (connectivity, error rates and packet loss, latency, bandwidth utilization)[1].

With every complex problem, various kinds of caveats occur. You need to have a proper design and consider subjects like hardware resources (e.g., CPU utilization, disk space, network connectivity, and so on), maintenance, and adequate software when thinking about data to process.

We will talk briefly about these problems in the following chapters.

2.1 What can be monitored

Monitoring is not as simple as it can be seen. There are several problems that we will need to take a look at. The monitored metrics will probably change as your infrastructure grows or evolves. Systems usually function hierarchically, with more complex layers building on top of the more basic infrastructure. Let's work from the bottom up and describe metrics that we can collect.

Host-based metrics are at the bottom of the monitoring hierarchy. These values are everything that is involved in evaluating the health or performance of individual hardware. These metrics consist of CPU utilization, memory utilization, disk space, monitoring active processes, syscall monitoring, etc. These values can give you a basic sense of factors that can impact a server's ability to remain stable or reliable.

Application metrics are involved with units of processing or work that depends on the host-level resources like software. In our web-server use-case, we are talking about the error and success rate of HTTP requests, service failures, and restarts, TCP performance, and latency responses, monitoring error logs, etc.

Network and connections metrics are another types of metrics that are worth exploring. It is mandatory to ensure a stable and reliable connection between the web-server and clients. Like other metrics that we have discussed so far, network monitoring is a must-have. We are specially talking about monitoring connectivity, error rates, packet loss, latency, and bandwidth utilization.

This thesis focuses only on collecting data from web servers so that we will concentrate only on essential metrics collected from web-servers, aka application metrics. In each metric type, we will discuss what it is about and why it is crucial, as well as units of that particular metric.

2.1.1 Sent bytes and request count

Sent bytes and request count is one of the most straightforward metrics to be monitored. Both are often used for not only performance analysis and debugging but can be used for billing your customers. If you want to charge your customers for transferred bytes, this is the first metric that you need to keep an eye on. Or your billing can be based on a number of requests. Counting requests can be useful, for example, for judging when to split the traffic between multiple machines, aka load balancing.

In-depth sent bytes states for the total amount of bytes that were transferred from the server to the client. Most web-servers do not count service data in sent bytes. Service data are, for example, TLS handshake, TCP handshaking, and so on, but it always depends on a web-server you use. On the other hand, a request just represents a single request for the given resource. The resource is a path after the domain, e.g. `/api/status`.

Request count is most frequently measured in requests per second. On the other hand, bytes can be in two; I would say formats, decimal, which is a part of SI, or binary, which is a part of IEC. The decimal format is based in the decimal base and binary in the binary base. Most people do not distinguish between these bases, and often all of these values are written in decimal base. For example, the RAM or even the SSD should be in binary bases, but sellers often write everything in decimal base[2].

2.1.2 HTTP status code

A server issues a status code in response to a client's request made to the server. There are five main categories for status code. The most interesting ones to monitor are the ones that are in format `4xx` or `5xx`. Firstly those which start with four means client error. On the other hand, those starting with five means server error. As of request count, the status codes are often monitored in status code per second. The meaning of the most frequent ones describes in the following table 2.1.

Keep in mind that this table does not cover all of the possibilities, but only the most frequent ones that you can encounter during your everyday web browsing. For example, it is a huge probability that you have experienced the `404` error. You, as a website owner, are interested in what your clients do on your website. Moreover, you are interested in how your clients browse your website. If your clients see the `404` message, they will probably be sad. So it is a good move to have been monitoring which URL resources return unwanted status codes since it can reflect user experience.

2.1.3 Round trip time

Roud trip time, aka RTT, is the amount of time it takes for a signal to be sent plus the amount of time it takes for an acknowledgment of that signal to be received. This time delay includes the propagation times for the paths between the two communication endpoints[4].

This metric is crucial for monitoring network performance. When the RTT is high, it means that clients are connecting to your server from a more considerable distance or connection quality is low. Moreover, this value tells you that you need to have a closer POP for your clients to achieve lower RTT.

It is better for those kinds of values that monitor some sort of a time to watch their percentile. Monitor percentile instead of average is better in many aspects. For example, the percentile is not that distorted as average, and average can give you wildly inaccurate results. Averages are ineffective because they are too simplistic and one-dimensional. Percentiles are a really great and easy way of understanding the real performance characteristics of your application. They also provide a great basis for automatic baselining, behavioral learning and optimizing your application with a proper focus[5].

Code	Description	Purpose
200	OK	Successful HTTP request.
206	Partial Content	The server is delivering only part of the resource due to a range header sent by the client.
301	Moved Permanently	This and all feature requests should be redirected.
302	Found	Tells the browser to look at another URL.
304	Not Modified	This tells the client, that a resource has not been modified so client does not have to download it again.
400	Bad Request	Server cannot process the request.
403	Forbidden	Server was able to process the request, but the server refuses to do it.
404	Not Found	Requested resource was not found.
410	Gone	The resource is no longer available.
500	Internal Server Error	Unexpected condition has happened on the server.
502	Bad Gateway	The server has received invalid response from the upstream.
504	Gateway Timeout	The server has not received a response from upstream.

Table 2.1: Common HTTP status codes

[3]

Type	Time
Local network	*20ms*
Data center	*200ms*
Continent	*300ms*
Transatlantic	*800ms*

Table 2.2: Common RTT values

To sum up, the best choice is to monitor RTT as percentile per second. Correct percentile values to calculate and monitor really depend on your use case. But often, you need to watch more than one percentile for RTT value. The best choice would be to monitor 50, 90, 95, 99 of RTT percentile[6]. Those four values can give you a broad insight into what is going on when you need them. Let's points out examples of RTT and its values.

2.1.4 Request time

Request time is a metric that tells us how long it took to process the request inside the web-server. Request time can be crucial, for example, when you are the developer of this particular web-server. Because when some requests take a significant amount of time to process, it is not suitable not only for your clients who will be unhappy but also for your infrastructure. Furthermore, request time depends on request size; when the request is large, it can affect your performance. It is useful for performance analysis.

The best measure unit is a percentile[5], as we described in the RTT. The same applies here because, in request time, we monitor some kind of time as well as in the RTT.

2.2 Caveats of monitoring

There are many possible caveats of monitoring that can occur. There are so many things that can go wrong, for example, hardware failure, network issues, software issues, data manipulation issues, etc. I will describe the problem of raw data, centralization or decentralization, distribution, and scalability.

2.2.1 Raw data

Raw data are data that we collected from the monitored subject. Moreover, there has not been any processing done on this data. There is a problem with these raw data since there has not been any processing on them, which means that we will have to store many pieces of information. This is a problem because these data can overgrow.

There are multiple options for how this problem can be solved. For example, you can drop old records after some time, or you can aggregate your data. Let's focus on data aggregation because it is a more complex and used solution in the wild. Data aggregation is the compiling of information from databases with the intent to prepare combined datasets for data processing. Aggregate data is high-level data that is acquired by combining individual-level data[7]. Let's bring an example for clarity because aggregating is the most critical aspect of monitoring.

In the previous chapter, we talked about sent bytes. Imagine that you have a client who is performing about 40000 HTTP requests per second to your web-server. And each request is 1 KB in size. Because you are doing billing at the end of each month, you would need to sum ALL transferred bytes for this client. That would be really time-consuming to sum this up from raw data because that is plenty of requests, and it would take plenty of time to iterate over each row in the DB. To optimize this, let's sum all transferred bytes for each minute and then insert it into the DB; this means that we would need some logic before inserting it into the DB that would

aggregate raw data. That saves up exactly 39999 inserts to the DB for each second. So now, when we perform select to sum all transferred bytes, it will be much faster because our data is already preprocessed.

2.2.2 Centralization vs. decentralization

Centralization and decentralization can be found in multiple fields, not only in computer science. But we will solely focus on centralization vs. decentralization in the monitoring domain area.

When you are building your monitoring data stack, you need to decide where to aggregate your data. You have two options; let's talk about it for a little bit.

The first option is to aggregate your data directly on your machine where you host your web-server. Moreover, each device that acts like a web server needs to have the spare computing power to aggregate and collect metrics. This method is decentralized because one machine can not affect another machine. This method of aggregating has its cons and pros. But it always depends on your situation. It does not have to be straightforward from the beginning whenever this method is the chosen one.

The second option is to aggregate your data on a dedicated machine whose only purpose is to aggregate and collect data. This method is purely centralized, and as the first one, it has its advantages and disadvantages. This method, for example, is inclined to higher sent bytes. Because you need to send raw data from your devices directly to your aggregating cluster, on the other hand, its most significant advantage is better control of your data. But centralizing data aggregation is not bad because the machine that aggregates data does not have to be a single machine but can be a cluster of machines.

The difference between centralization and decentralization is one of the hot topics these days. Some people think that centralization is better, while others are in favor of decentralization. In ancient times, people used to run their organizations in a centralized manner. The scenario has been changed entirely due to the rise in the competition where quick decision-making is required. Therefore many organizations opted for decentralization[8].

2.2.3 Distributed

The distributed system is a distributed collection of computing units that can make decisions locally[9]. Moreover, each node has its own goal to accomplish, and together all nodes crowdfund toward the final goal. These goals do or do not have to be the same.

In the monitoring, it can have multiple meanings. When collecting data from web servers, it means that if some web-server crashes or somewhat goes down, then it does not affect other web servers in the infrastructure. Furthermore, the traffic from the fallen web-server splits across healthy nodes.

Distribution is mandatory because your availability can be fatal to your pipeline when you have a single machine for handling aggregation. When your single instance goes down, you lost your data, or worse, you overload your entire infrastructure because traffic from the fallen node splits to other nodes, and it overloads them, and they fall as well. So often, there is a demand to have multiple machines in your cluster so one or more devices can go down without data loss.

2.2.4 Scalability

Scalability is the attribute of some kind of system to handle a growing amount of data to process by adding resources to the system. In the software aspect, this means that that you are not software nor hardware limited to some computation. Moreover, you are able to distribute the computation to more computation units. This can mean that there are multiple instances of running software on the same machine, or the instances are on separated hardware. An example is a database that should support an increasing number of connections. This property goes in hand with distribution.

Chapter 3

Research of monitoring tools

There are a lot of tools that can help us during monitoring. Most of them are open source and widely used by global companies like Cloudflare, Akamai, Google, Amazon, etc. In this chapter, we will take a look at some of the most commonly used tools for monitoring web servers. We can split the applications into several categories like collecting, processing, alerting, and visualizing.

The collecting category is responsible for getting data from the application. It can be simply from reading a log file, reading a socket, requesting API, requesting a DB, or some other interaction with the application. Then often transfer these collected data into applications that are responsible for the processing[6]. The processing category takes care of manipulating data. It transforms data into a better suitable format sot it can aggregate more easily, manipulate data, calculate better, etc. These data are then stored in a suitable database. Applications in the alerting class watch our converted or shaped data from the database and alert us when it sees unwanted anomalies. Applications in visualizing category are graphical tools that represent or plot our transformed data; visualizing category can handle alerting as well.

It is essential to stress out that these applications form a chain together. The chain is as strong as the weakest part in them. It means that you have to have a good design of your pipeline. When your collecting applications do not collect the essential data, your fancy graphing software would be useless.

There are many tools, so I will point out these that I found most widely deployed during my research.

3.1 Open source tools

Open source is source code that is made freely available for possible modification and redistribution. Open-source code is mostly managed by the community or company that uses that particular software. It is hard to believe that companies make public their code base for the community, but there can

be huge benefits from that move. For example, the community of people who like the software can help with maintaining it[10].

3.1.1 Grafana

Grafana is a multi-platform open source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources[11]. Grafana is commonly shipped and recommended to be used with InfluxDB[12]. Grafana uses one of the available data source adapters for visualizing your data. Keep in mind that Grafana is just a web app that you need to have to host, so it is an on-premise solution, so it requires dedicated hardware to be run on[13]. But there are companies[14][15][16] that will host it for you in a cloud or dedicated hardware.

Grafana offers time-series graphs, gauge, bar gauge, table, pie chart, and many more by default. And as a data source, it provides, for example, adapters for Prometheus, Graphite, Loki, Elasticsearch, Jaeger, MySQL, PostgreSQL, MongoDB, DataDog, and many more[11]. Grafana is easily extendable. You can write your own graph types and data source connectors, but the community already has developed almost everything, so it is unnecessary to reinvent the wheel since you can use something that is already coded. On the other hand, Grafana has its caveats. Let's point out the pros and cons[17].

You have a significant portfolio of available graphs for visualization. You have many publicly available data source connectors, so it should be easy to connect any widely accepted database as an input for your Grafana. On the other hand, I found out that Grafana lacks of input methods. This means it is useful for static graphs, where you do not have any select bar, where to specify some particular conditions. But when you want to have some custom input where you can select particular conditions for graphing, then it is not that good since it is not that straightforward. We will talk about this problem in subsection 4.1.3. Grafana is also resource-consuming for your web browser; when you have many items for graphing, it consumes a lot of RAM a CPU power, and it takes a lot of time simply to graph.

3.1.2 Prometheus

Prometheus is an open-source system monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this and clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes. Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped

samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana or other API consumers can be used to visualize the collected data. Google's monitoring system, Borgmon, served as inspiration for Prometheus[18].

Prometheus is primarily based on the pull model. This means that Prometheus node periodically scrapes the given API of each monitored subject at a specific polling frequency. Prometheus data from the API in a specific pre-defined format. Prometheus data is stored in the form of metrics, each metric has a name used for referencing and querying it. Prometheus stores data locally on disk, which helps for fast data storage and fast querying and store metrics in remote storage. Each Prometheus server is standalone, not depending on network storage or other remote services. This means that Prometheus is hard to scale since there are not any databases that could be used for atomic or transaction data access[19]. Prometheus is not designed to be scaled horizontally. Once you hit the limit of vertical scaling, you're done[20].

Prometheus provides its own query language, PromQL, that lets users select and aggregate data. PromQL is specifically adjusted to work in convention with a time-series DB and therefore provides time-related query functionalities[19].

3.1.3 ZooKeeper

ZooKeeper is a service for distributed synchronization, as well as providing group services. All of these kinds of services are used in some form or another by distributed applications. Because of the difficulty of implementing these kinds of services, applications initially sacrifice on them, making them fragile in the presence of change and challenging to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed[21]. More briefly, ZooKeeper acts like a distributed key-value store that supports transactions. It is used in many applications like ClickHouse and Kafka for its synchronization.

3.1.4 ClickHouse

ClickHouse is a fast open source OLAP database management system. It is column-oriented and allows to generate analytical reports using SQL queries in real-time. ClickHouse processes typical analytical queries two to three orders of magnitude faster than traditional row-oriented systems with the same available IO throughput and CPU capacity. Columnar storage format allows fitting more hot data in RAM, which leads to shorter typical response times. ClickHouse scales well both vertically and horizontally. ClickHouse can smoothly perform either on a cluster with hundreds or thousands of nodes or on a single server, or even on a tiny virtual machine. ClickHouse has concepts of shards and replicas. In a shard, you can have multiple replicas. Shards can

be geographically distributed to achieve better availability[22]. Furthermore, ClickHouse has a concept of engines. Your database or your table needs to have a specified engine while creating tables or databases. These tables or database engines define its behavior. Databases engines provide default behavior for tables created in that DB; they are set to Atomic[23] by default. On the other hand, table engines are more fascinating because you can tweak table behavior, and they have several categories[24]. Moreover, ClickHouse design is to be eventually consistent since it is defined by its shard policy[25]. We will talk about ClickHouse features more briefly since it was chosen for the practical part.

MergeTree family engines are the first and most used and universal and functional table engines for high-load tasks. The property shared by these engines is quick data insertion with subsequent background data processing. This subsequent background data processing is essential. That is why these engines are the most widely adopted. MergeTree family engines support data replication (engines with prefix Replicated), partitioning, secondary data-skipping indexes, and other features not supported in non-MergeTree engines like specified TTL. TTL is useful because it allows you to specify conditions for data deletion. Moreover, specify the duration for how long to keep data on the disk. Or you can use TTL to specify the conditions for moving data. Furthermore, when data becomes cold, you can move them from an SSD to HDD[24]. Keep in mind that MergeTree does not like frequent inserts. When you want to insert a lot of data into your table, you need to batch them. Furthermore, the batches should not be too large. It would be best if you kept a proper balance between large inserts and the frequency of inserts. Otherwise, the performance of your node will decrease significantly. There is an engine named Buffer that will help us to batch these values. We will talk about Buffer and MergeTree engine later in 4.1.2.

Next, there are Log family engines. They are Lightweight engines with minimum functionality. They are the most effective when you need to quickly write many small tables (up to approximately 1 million rows) and read them later as a whole. But sadly, they do not allow you to specify TTL[24].

The last family is integration engines. This family is somewhat experimental, and there are many bugs [26],[27],[28] and counting. It allows you to integrate your ClickHouse with the existing DB. For example, you can use your ClickHouse to access your data stored in MySQL or Kafka cluster[24].

There are more engines that do not belong to any family, but they have relatively a single purpose. I am explicitly aiming towards engines like Buffer, Distributed, MaterilizedView. These three engines I found the most crucial in my practical part of this thesis. MaterializedView is the glue that connects your raw data and your aggregated data. Materialized views in ClickHouse are implemented more like insert triggers. If there is some aggregation in the view query, it is applied only to the freshly inserted data batch. Any changes to the source table like an update, delete, drop a partition, etc., do not change

the materialized view. Furthermore, MaterializedView can be used to move data between solid tables or be used with AggregatingMergeTree to perform aggregations. We will talk about aggregations in 4.1.2. The distributed engine is more like a helper engine, helping you split queries across your cluster. Distributed engines do not store any data on their own but allow distributed query processing on multiple servers. Reading is automatically parallelized. During a read, the table indexes on remote servers are used, if there are any. Writing to the distributed table will distribute the inserted data across the servers themselves[29]. Lastly, let's focus on the buffer table. It buffers the data to write in RAM, periodically flushing it to another table. During the read operation, data is read from the buffer and the other table, which is being buffered simultaneously. Buffer table has rules when to insert the buffer data based on time, rows, and size. When you create the table with a buffer engine, you need to specify all of these three values, and they have to be balanced; otherwise, you may encounter some errors like inserting too much data[30].

Let's sum up the pros and cons of using ClickHouse. The ClickHouse is definitely suitable for scaling horizontally and vertically[22]. It is also very fast with its MergeTree engines, which are very popular in the ClickHouse community for their performance and distributed setup. ClickHouse is suitable for a high load of data and aggregating them. The most significant disadvantage is documentation with examples. There are not many resources to get knowledge about setting up ClickHouse. There are only a few of them, and they are mostly fundamental and not production-ready, as well as they do not mention many caveats that can occur. These pros and cons are based on my own personal experience with ClickHouse during this thesis.

3.1.5 MongoDB

MongoDB is a document database designed for ease of development and scaling. A record in MongoDB is a document, a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents. Using documents has many advantages, like dynamic schema support, embedded documents, and arrays to reduce the need for expensive joins, objects correspond to native data types in many programming languages. Furthermore, MongoDB proves high-performance data persistence. Since MongoDB is a NoSQL database, it means that it has its own query language, which looks more like JavaScript[31].

One of the most attractive features of MongoDB is its reliable scalability. It is precisely the opposite of traditional SQL databases, which scale more vertically than horizontally. MongoDB has concepts of shards for horizontal scale. As mentioned earlier, MongoDB is a very dynamic database, which means that document structure can be specified on the fly. MongoDB is also

swift because it stores most of the data in RAM, and query performance in MongoDB is much quicker[32].

On the other hand, MongoDB has poor data manipulation since it does not support transactions very well. The support for transactions looks experimental at the time of writing this thesis. This means that there can be data corruption when multiple resources want to modify the same document. Furthermore, MongoDB has not easy to use joins, so joining multiple collections is a resource-heavy task[33],[34],[35]. Furthermore, MongoDB is eventually consistent. It does not support ACID transactions or distributed transactions[36].

3.1.6 ELK

ELK or Elastic stack is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch.

Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. According to the DB-Engines ranking, Elasticsearch is the most popular enterprise search engine, followed by Apache Solr, also based on Lucene. Elastic search, due to its reverse indexes, is high-speed. It also supports parallel processing, which can speed it up by a lot, but it depends on many factors. Elastic search is also well scalable horizontally as well as vertically[37],[38].

Logstash is a lightweight, open-source, server-side data processing pipeline that allows you to collect data from various sources, transform it on the fly, and send it to your desired destination. It is most often used as a data pipeline for Elasticsearch, an open-source analytics and search engine. Because of its tight integration with Elasticsearch, powerful log processing capabilities, and over 200 pre-built open-source plugins that can help you easily index your data, Logstash is a popular choice for loading data into Elasticsearch[39]. Hoewer Logstash is also very slow and resource heavy[40],[41].

Kibana is an open-source data visualization and exploration tool used for log and time-series analytics, application monitoring, and operational intelligence use cases. It offers powerful and easy-to-use features such as histograms, line graphs, pie charts, heat maps, and built-in geospatial support. Also, it provides tight integration with Elasticsearch, a popular analytics and search engine, which makes Kibana the default choice for visualizing data stored in Elasticsearch[42].

Overall, the ELK stack became very popular. Most companies use it for their log pipeline because it is an open-source solution, so it is free. On the other hand, you need to have beefy machines where you can run your ELK

stack. Since it is recommended to use fast CPUs and at least SSDs, Logstash consumes many resources and Elasticsearch for fast storing data[43].

3.1.7 Kafka

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Kafka has a high throughput and delivers messages at network limited throughput using a cluster of machines with low latencies. Kafka also scales well. Furthermore, Kafka stretches clusters efficiently over availability zones or connect separate clusters across geographic regions and store streams of data safely in a distributed, durable, fault-tolerant cluster[44]. Kafka is primarily used to build real-time streaming data pipelines and applications that adapt to the data streams. It combines messaging, storage, and stream processing to allow storage and analysis of historical and real-time data[45].

When writing messages into Kafka, we write them into a topic. Topics are divided into several partitions. Partitions serve as our unit of ordering, replication, and parallelism. Topics are configured with a replication-factor, which determines the number of copies of each partition we have. All replicas of a partition exist on separate brokers (the nodes of the Kafka cluster). This means that we cannot have more replicas of a partition than we have nodes in the cluster. A replica is either the leader of its partition or a follower of the leader. A follower can either be synced with the leader or unsynced[46].

3.2 Stacks that global companies use

Many big companies have their blog, where they share their knowledge during working on exciting projects. These blogs were my starting point at the beginning of this thesis. Furthermore, these blogs were an inspiration for me, and I used them for resources and examples. These companies have a lot of useful knowledge that I used during my research, and their pipelines are already tested in production.

3.2.1 Cloudflare

Cloudflare is one of the biggest networks operating on the Internet. People use Cloudflare services for the purposes of increasing the security and performance of their websites and services[47]. More briefly, Cloudflare provides content delivery network services, DDoS mitigation, Internet security, and distributed domain name server services[48].

Cloudflare regularly publishes their knowledge on its blog. You can find security tips, achieved goals, language contributions, library contributions, interesting research, etc.

In Cloudflare, they use Clickhouse and Kafka in their log processing pipeline. They use it for aggregation statistics from the nginx, and they are processing six million requests per second which is a significant amount of requests to be processed[49].

3.2.2 Altinity

Altinity software and services help you deploy and operate innovative ClickHouse analytic applications for any use case in environments spanning the cloud to on-prem. Deploy and operate ClickHouse, a lightning-fast, an open-source SQL data warehouse for real-time analytics, time series, and log analysis[50].

Many developers from Altinity take care of ClickHouse, and they write a handy blog that is primarily focused on ClickHouse. In their blog posts, you can find good recommendations about how to set up your cluster features.

Chapter 4

Pipeline implementation

In this chapter, we will focus on practical implementation. We will more broadly discuss the implementation that I found the most interesting to research and most generic to use. We will use the theoretical background described in the previous chapters. The following methods are tested and developed only for the Linux environment.

As a web-server, we chose Nginx because it is rapidly growing in popularity and is often used in large infastructures[51]. Nginx is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server[52]. Nginx is well configurable and can expose many valuable options with its access log and error log. We will set up our Nginx as a reverse proxy.

In particular, we will discuss two implementation pipelines for the monitoring. The first pipeline will take care of monitoring data from the access log. And the second pipeline will take care of monitoring errors from the error log which Nginx exposes. We will discuss these topics more briefly in the individual chapters.

4.1 Transfer metrics monitoring

This monitoring pipeline will take care of getting data from the Nginx and then store it in a database. Moreover, the database will aggregate these data, and we will visualize them with graphs. Furthermore, in each individual subchapter, we will solve each isolated part of the monitoring pipeline. In the first subchapter, we will focus on getting data out of the Nginx web-server. In the second chapter, we will focus on data aggregation, and lastly, we will take a look at data visualization.Let's describe the overall architecture and then well will dig deeper into each individual subchapter.

We will write a simple application in Golang, which will collect data from the Nginx, and this application will send data to the ClickHouse in larger batches. Batching will prevent overloading our ClickHouse cluster because ClickHouse does work well with small inserts[30]. Our application will be

based on a push model. This means that our Clickhouse cluster will not pull metrics from each Nginx web-server. Instead, we will insert data from our application written in Golang. I chose Golang because it has an extensive standard library, libraries for ClickHouse, community support, well supported on various operating systems, well readable. I took this push model approach because it is more scalable, in my opinion, since the aggregating endpoint does not need to know anything about the web-servers.

4.1.1 Data collecting

Let's start with the task of getting data out of the Nginx. Nginx has configuration directive `access_log`[53]. In this directive, you can specify multiple options; for now, the most useful options are format and path. Nginx will print a line to the access log for each new HTTP access, which ends with a new line character.

For the demonstrative purpose, we will collect only transferred bytes, RTT, and total request count. In our example, the format will look like `log_format '$bytes_sent $tcpinfo_rtt'`[53].

Then we need to discuss where to log our access log since Nginx supports multiple output options. Each of them has its pros and cons.

4.1.1.1 Logging to file

We can log our data into the file. This is very straightforward since most of the applications use this approach. This mechanism is to pool the log file. We are interested only in newly added records of the log file. So we need to implement something like Linux's `tail -f`[54]. This can be easily done in Golang in 4.1 since there is a library exactly for this purpose called simply a tail[55].

```
t, err := tail.TailFile("/var/log/nginx.log",
                        tail.Config{Follow: true})
for line := range t.Lines {
    fmt.Println(line.Text)
}
```

Listing 4.1: Golang tail a file

We can specify the following option, which will handle logrotating cases, but it will not handle reading duplicates when restarting the Golang tool. This library has even more options on how to detect logrotating. It can simply just pool for a new file with the same name. Or it can use Linux's event messaging for changes.

- The pooling technique periodically checks if a new file was made. When we open a file in a Linux environment, we get a number of file descriptor.

It does not have a concept of a name. If the file that we have opened is renamed by logrotate, we do not have a way how to check it unless we periodically compare inodes of the opened file and possibly a new file. If inodes do not match, then the file was logrotated.

- The inotify API provides a mechanism for monitoring filesystem events. Inotify can be used to monitor individual files or to monitor directories. When a directory is monitored, inotify will return events for the directory itself and files inside the directory[56].

There are other cons that go hand in hand together, and some of them are even language-specific or platform-specific.

Nginx does not support logrotating by default. This means that our access log will grow infinitely. This is not good since it can consume all disk space in a long-running web-server, leading to undefined behavior later. Many tools support logrotating, namely logrotate[57], which is widely adopted. When our file was logrotated, we need to finish reading our current file and open a newly created file and start processing it again. But both approaches do not scale well with growing requests. Inotify API gains a lot of resource overhead and stops working correctly, approximately around 20000 requests per second. The pooling technique starts to act undefine at approximately 50000 requests per second on the Nginx server. So let's drop this approach and try another one available.

4.1.1.2 Logging to UNIX socket

Since we are developing our tool for the Linux environment, we can take advantage of logging into the UNIX socket. A Unix domain socket or IPC socket is a data communications endpoint for exchanging data between processes executing on the same host operating system[58],[59]. Logging to UNIX socket handles all caveats which occur during logging to file. But it still is not a silver bullet. This approach has the following caveats.

- UNIX socket has a hardcoded limit of the inner buffer. This can be changed, but we would need to recompile the whole kernel a that is something that you do not want to do since it is really time-consuming and inefficient. In Linux kernel `4.x` it is hardcoded to 212992 bytes[60].

- Since our UNIX socket is not endless, we need to have some mechanism to slow down logging to the UNIX socket within Nginx. Because Nginx can quickly overwhelm this socket with data. Nginx has a feature that can buffer data in RAM before sending it to the UNIX socket. But this adds overhead since each Nginx worker needs to allocate its space in RAM. This mechanism helps us with full UNIX sockets, but it will only delay our problem. It keeps its data in userspace, and when the UNIX

buffer becomes free, it inserts data into it. This mechanism will work only as long as we have free space in userspace.

We can create the UNIX socket with the Golang code in 4.2.

```
// Remove and iniate a socket
sp := "/var/log/nginx/access.sock"
os.Remove(sp)
addr, _ := net.ResolveUnixAddr("unix", sp)
conn, err := net.ListenUnixgram("unixgram", addr)
if err != nil {
    log.Fatalf("Fatal error creating socket %s\n", err)
}
os.Chmod(config.SocketPath, 0777)

// Read data infinitely
for {
    // maximum line length should be around 2 KiB, but
    // sockets have some overhead
    buf := make([]byte, 4096)
    for {
        // this is blocking operation
        dn, _, _ := conn.ReadFromUnix(buf[:])
        println(dn)
    }
}
```

Listing 4.2: Golang listen on UNIX socket

I tested this approach on a machine with 48 CPU cores and 256 GiB of RAM. Nginx was configured to use 48 workers, and I was able to get up to 60000 requests per second, then Nginx started to throw out requests due to buffer limits and started logging to the error log that it could not write to the socket. It was unable to write to the socket because the socket was full. The application could not read the data faster because it was read from one thread, but Nginx was writing to the socket from all workers. We could try to read from more threads, but it would not do much better because the socket's max size is not enough.

4.1.1.3 Logging to UDP

The last option is to use UDP. So our Golang tool will acts as a UDP server and Nginx as a client. This mechanism handles all cavers that occurred during the previously described method. In UDP, its buffer limit can be specified on the fly without recompiling the whole kernel. Furthermore, we can take

advantage of socket options and distribute incoming datagrams between multiple sockets[61], in particular we are talking about `SO_REUSEPORT`, it can be demonstrated in 4.3. On the other hand, UDP adds some overhead because UDP datagrams have some headers that we do not use, but it is not significant because we can increase the buffer limit.

Overall this method performed exceptionally in the production. We were able to increase the buffer size to the maximum, so we did not need buffering techniques that Nginx implements. Moreover, we were able to write all logs to the sockets as they appeared. Furthermore, the Golang application could spawn more read threads because we took advantage of using socket options.

```
conn, err := ls.conf.ListenPacket(context.TODO(),
                                  "udp4",
                                  "127.0.0.1:8888")
if err != nil {
        log.Printf(err)
        return
}

buf := make([]byte, 4096) // maximum line length should be
                          // 2 KiB, but sockets have some overhead
for {
        conn.SetReadDeadline(time.Now().Add(5 * time.Second))
        // this is blocking operation
        n, addr, err := conn.ReadFrom(buf[:])
        if err != nil && !err.(*net.OpError).Timeout() {
                log.Printf("Error: Reading packet from %s: %s\n",
                addr,
                err,
                )
        }
        if n > 0 {
                println(string(buf[0:n]))
        }
        if ctx.Err() != nil {
                err = conn.Close()
                if err != nil {
                        log.Println(err)
                }
                return
        }
}
```

Listing 4.3: Golang listen on UDP socket

I tested this approach on a machine with 48 CPU cores and 256 GiB of RAM. Nginx was configured to use 48 workers, and it could handle at least 100000 requests per second. I was unable to generate more requests because of a hardware limitation. Moreover, this technique performed exceptionally, and there were no errors during testing.

4.1.2 Data aggregating

For data aggregation, we will use ClickHouse. It is easy to set up and use software with sufficient documentation. Furthermore, we will use it for calculating percentile and sum. We will demonstrate our example on a single node cluster, but the same applies to a cluster with more machines than one. The following approach will work with ClickHouse version `20.3.x`. ClickHouse needs to have a ZooKeeper running somewhere. It is recommended to have it on different hardware than ClickHouse. Furthermore, the ClickHouse cluster needs to be defined in its configuration file. We can find that in their documentation[29].

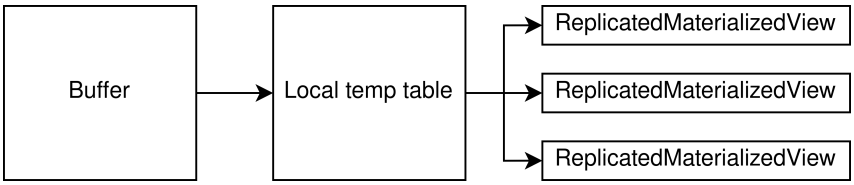

Figure 4.1: ClickHouse diagram

Our cluster will look like 4.4. If we would like to have more shard or replicas, we would need to specify it in its config on each machine.

```
<remote_servers>
    <logs>
        <shard>
            <replica>
                <priority>1</priority>
                <host>localhost</host>
                <port>9000</port>
            </replica>
        </shard>
    </logs>
</remote_servers>
```

Listing 4.4: ClickHouse config

We will get our data for aggregation from the Golang application. We will use the official Golang library[62] that is developed by the ClickHouse

team. This library implements all necessary features like reconnecting, load-balancing between multiple shards, etc. We will use the following code 4.5.

```
connect, err := sql.Open("clickhouse", "tcp://127.0.0.1:9000")
if err != nil {
        log.Fatal(err)
}
if err := connect.Ping(); err != nil {
        if exception, ok := err.(*clickhouse.Exception); ok {
                fmt.Printf("[%d] %s \n%s\n", exception.Code,
                                                exception.Message,
                                                exception.StackTrace)
        } else {
                fmt.Println(err)
        }
        return
}
var (
        tx, _   = connect.Begin()
        stmt, _ = tx.Prepare(`INSERT INTO example (metric_date,
                                                   rtt,
                                                   bytes_sent)
                                                   VALUES (?, ?, ?)`)
)
defer stmt.Close()
for _, item := parsed_data_from_socket{
        if _, err := stmt.Exec(
                time.Now(),
                item.RTT,
                item.BytesSent,
        ); err != nil {
                log.Fatal(err)
        }
}
if err := tx.Commit(); err != nil {
        log.Fatal(err)
}
```

Listing 4.5: Sending data to the ClickHouse

Now we need to set up our ClickHouse, so it can aggregate data. Our goal is to use AggregatingMergeTree, as discussed earlier, this engine acts as a trigger, so we need to store our inserted data into some temporary table. We will attach these triggers to this temporary table, and we will be able to create

aggregated data when we insert new data into the temporary table. For this temporary table, we will use a table with the MergeTree engine. But this is a problem. We cannot insert data fastly into tables with the MergeTree engine. If we do, then ClickHouse will reject our inserts. We need to know how to buffer data before inserting it into the temporary table with the MergeTree engine. We can use the Buffer engine, which is precisely for this purpose. We will use the Buffer table to batch our data from the Golang tool before inserting it into the actual table. Buffer table stores its data within RAM.

But Buffer table is still not a silver bullet. It has some flaws. If our machine goes down, we need to keep in mind that these data are lost since they are stored only in RAM, which does not survive reboot. We need to set rules for inserting data from the Buffer table into the MergeTree table. ClickHouse does not allow us to attach our aggregating views above tables stored in RAM, so that is the main reason we use this temp table. Data are stored on the disk only after we perform the insert into the MergeTree family engines.

After we insert data into the temporary table with the MergeTree engine, we can instantly drop them. Furthermore, we can set a TTL on that particular table. Since we need only to aggregate data, then after inserting, we do not need them. Our temp table will like like in 4.6, and then we can create the Buffer table like in 4.6.

```
-- buffer table
CREATE TABLE metrics_buffer AS metrics ENGINE = Buffer(
default,
metrics,
16,
3,
6,
500000,
1000000,
100000000,
1000000000);

-- temp table
CREATE TABLE metrics (
    metric_date DateTime,
    bytes_sent UInt64,
    RTT UInt32 )
ENGINE = MergeTree()
order by (toStartOfFiveMinute(metric_time))
TTL metric_time + INTERVAL 10 SECOND;
```

Listing 4.6: ClickHouse DDL script

Now we can focus on creating views that will aggregate our records. ClickHouse has its AggregatingMergeTree for this purpose, as we have discussed earlier. We will use the following SQL 4.7 that will create request counting for each hour, calculating median and summing transferred bytes for each hour. This example can be easily expanded to calculate more time variants, e.g., second, minute day, and more types of percentile.

```
CREATE MATERIALIZED VIEW traffic
ENGINE = ReplicatedAggregatingMergeTree
('/clickhouse/tables/1/traffic', '1')
PARTITION BY metric_date ORDER BY (metric_date, metric_time)
TTL metric_time + INTERVAL 1 HOUR
AS select
                                                metric_date,
  toStartOfInterval(metric_time, INTERVAL 1 HOUR) metric_time,
  sumState(bytes_sent)                          traffic
FROM metrics
GROUP BY metric_date, metric_time;

CREATE MATERIALIZED VIEW rtt_50p
ENGINE = ReplicatedAggregatingMergeTree
('/clickhouse/tables/1/rtt_50p', '1')
PARTITION BY metric_date ORDER BY (metric_date, metric_time)
TTL metric_time + INTERVAL 3 DAY
AS SELECT
                                                metric_date,
  toStartOfInterval(metric_time, INTERVAL 1 HOUR) metric_time,
  quantileTDigestMergeState(0.5)(rtt)           percentile
FROM metrics
GROUP BY metric_date, metric_time;

CREATE MATERIALIZED VIEW request_count
ENGINE = ReplicatedAggregatingMergeTree
('/clickhouse/tables/1/request_count', '1')
PARTITION BY metric_date ORDER BY (metric_date, metric_time)
TTL metric_time + INTERVAL 90 DAY
AS SELECT
                                                metric_date,
  toStartOfInterval(metric_time, INTERVAL 1 HOUR) metric_time,
  countState()                                  request_count
FROM metrics
GROUP BY metric_date, metric_time;
```

Listing 4.7: ClickHouse DDL script for views

Overall our table architecture can be visualized in the following diagram 4.1. We insert data into the Buffer table. Then ClickHouse will insert these buffer data into the temporary table with the MergeTree engine based on our rules. Then after ClickHouse transfers these data from the buffer table into the temporary table, ClickHouse will execute triggers that we attached to the temporary tables. These triggers will aggregate our data, and they will be stored in that trigger. We can select our data from this trigger with SQL.

4.1.3 Data visualization

Then we will use Grafana for data visualization since it is straightforward to set up and there is a freely available ClickHouse source adapter[63]. We can easily query our triggers for data.

For example, our case with RTT can be displayed with the following query. The same applies for the rest of our data but only with different merging combinators, here we use `quantileTDigestMerge`, a complete list can be found in the documentation[64]

```
SELECT
    t,
    groupArray((machine_name, c))
FROM
(
    SELECT
        $timeSeries as t,
        quantileTDigestMerge(0.5)(percentile) as c,
        'localhost' as machine_name
    FROM rtt_50p
    WHERE
        $timeFilter -- Grafana specific demand
    GROUP BY
        metric_date,
        metric_time,
        machine_name,
        t
)
GROUP BY t
ORDER BY t asc
```

Listing 4.8: Grafana ClickHouse SQL

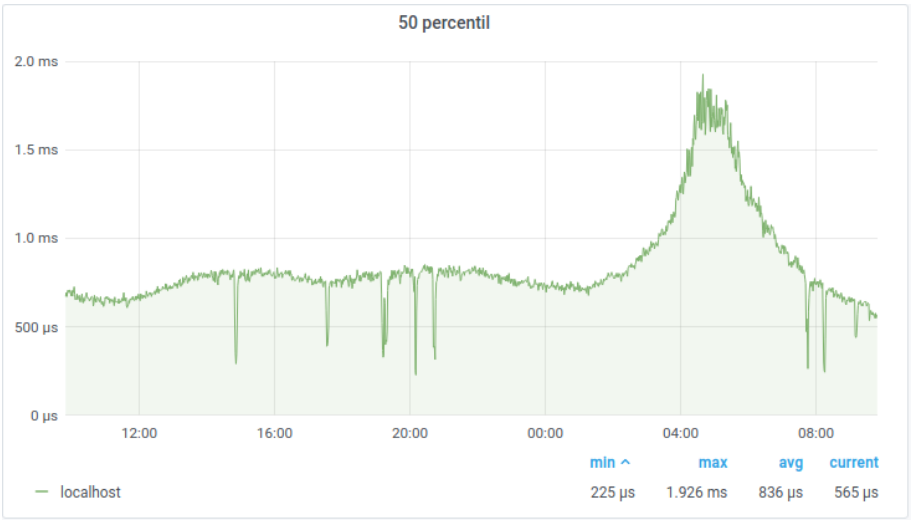

Figure 4.2: RTT median example

4.2 Error log monitoring

Error monitoring is crucial since you can spot unwanted behavior as well as weird behavior. We will use the ELK stack for this purpose, discussed in the previous chapter. ELK is a great tool for this purpose because it is widely adopted, and it scales well. Moreover, it has great documentation.

Nginx normally logs errors to error log via `error_log` directive. This directive has many options, but we will stick to the defaults. This approach will work for every Nginx, which has a version above `1.0.0+`. We will use a straightforward Nginx configuration, which will have two server directives and a few other optional params and `error_log` configured. Nginx will report unusual activity to the error log based on its verbosity level to the file, located on path `/var/log/nginx/error.log`. We can see example configuration in 4.9.

4.2.1 Filebeat

We need something that will read our error log file or files. This tool needs to care for many caveats like.

- It needs to reopen the file when Logrotate comes in place.
- It needs to batch data to prevent flooding a network with frequent requests.
- It needs to buffer data when a network is down or when the destination server is down.

Filebeat is a part of the Elastic stack. Filebeat can be easily installed from the Filebeat website. It has packages available for Debian, RPM, MacOS, Brew, Generic Linux, Windows. The following approach was tested for version `7.1.0`.

We need to provide a configuration file for the Filebeat. We will send our logs to Logstash because Filebeat has good support for sending data to Logstash. The configuration file is in YAML format. Filebeat will address the above caveats.

Filebeat has many available options, but we will talk only about these that we are using.

- We need to provide a path to the Filebeat. Filebeat will be listening on this path for any updated files. We can specify a whole directory with multiple files or a single file.
- We should enable the crawler in the configuration.
- We need to specify the output as the Logstash. We need to provide hostnames or IP addresses to our Logstash nodes.

So now Filebeat with this configuration will observe our error log located on path `/var/log/nginx/error.log`. Furthermore, it will send our data to the Logstash, where we will manipulate data next. When something goes wrong, it will handle all caveats, including buffering when the network is down, logrotating file, batching logs.

4.2.2 Logstash

Logstash will serve as middleware for preprocessing data before we send it to Elasticsearch. It can aggregate logs and events from various sources. It can be defined as collecting and processing data from these sources and sending them to other systems for storage and analysis. We will use the input as beats. Logstash supports several filter plugins that enable you to handle events. Furthermore, we can transform our data into another format suited for Elasticsearch.

Logstash is often run on the same server as Elasticsearch since it can reduce delay while inserting data. Logstash will automatically create indexes for Elasticsearch, so we do not care about creating indexes and their properties ourselves.

For configuration, we only need to define input as beats. Beats has a protocol of its own. We can send it cyphered, but we will send it plain without encrypting for the sake of simplicity. Then we can define the filter section, which parses lines of logs into some intermediate format, which then is translated into Elastic search data types. We need to define a regexp that will

handle all possible logline cases; otherwise, our logline will be flagged as unparsable. Our error logline has a straightforward format. In the end, we need to define our output which is Elasticsearch.

4.2.3 Kibana and Elasticsearch

Kibana is a frontend UI for Elasticsearch. By default, Elasticsearch exposes only REST API without UI. Kibana is a web interface for Elastic search with the ability to create plots and graphs. Logstash did all the heavy lifting for us and is filling Elasticsearch with data. Now we can, with drag and drop functionality, create multiple diagrams based on our requirements. Furthermore, Kibana allows us to change the properties of indexes created by Logstash. It is recommended to change lifecycle policies that will handle deleting old records for us. Since reverse indexes created by Elasticsearch can consume a lot of disk space, it is recommended to delete old data which are not needed anymore. An example of data visulization can be found in 4.3.

Time	log.level	kafka.log.component	message
Mar 21, 2021 @ 10:44:33.989	TRACE	Controller id=1	Leader imbalance ratio for broker 4 is 0.0
Mar 21, 2021 @ 10:44:33.989	DEBUG	Controller id=1	Topics not in preferred replica for broker 5 HashMap()
Mar 21, 2021 @ 10:44:33.989	TRACE	Controller id=1	Leader imbalance ratio for broker 5 is 0.0
Mar 21, 2021 @ 10:44:33.989	DEBUG	Controller id=1	Topics not in preferred replica for broker 1 HashMap(cdn_http_log-3 → List(1, 3, 4))
Mar 21, 2021 @ 10:44:33.989	TRACE	Controller id=1	Leader imbalance ratio for broker 1 is 0.0030864197530864196
Mar 21, 2021 @ 10:44:33.989	DEBUG	Controller id=1	Topics not in preferred replica for broker 2 HashMap()
Mar 21, 2021 @ 10:44:33.989	TRACE	Controller id=1	Leader imbalance ratio for broker 2 is 0.0
Mar 21, 2021 @ 10:44:33.989	DEBUG	Controller id=1	Topics not in preferred replica for broker 3 HashMap()

Figure 4.3: Kibana example

```
worker_processes  auto; # use all cores
error_log  /var/log/nginx/error.log;

events {
}

http {
  include    conf/mime.types;

  default_type application/octet-stream;
  log_format   main
    '$remote_addr - $remote_user [$time_local]  $status '
    '"$request" $body_bytes_sent "$http_referer" '
    '"$http_user_agent" "$http_x_forwarded_for"';
  access_log   logs/access.log  main;

 # reverse-proxy
  server {
    listen       80;
    server_name  domain2.com www.domain2.com;

    location / {
      proxy_pass      http://127.0.0.1:8080;
    }
  }

  # static content
  server {
    listen          80;
    server_name     domain.com www.domain.com;

    location / {
      return 200 "hello";
    }
  }
}
```

Listing 4.9: Nginx configuration for Filebeat

Chapter 5

Testing, improvements and debugging

In this chapter, we will talk about problems that occurred and the results that I have collected. I was able to test the results of this thesis in the production environment where a lot of traffic can be collected in a real production environment. The company where I tested this supplied me with the necessary hardware and technical help. I was able to stress test this setup in an environment where the edge servers handle up to 12 TiB of traffic. Overall I tested this on a ClickHouse cluster which consisted of 5 nodes, and the logging application was deployed on over 1000 web servers.

In the first part, we will focus on specific problems. In the second part, we will discuss possible improvements that can make this pipeline more robust, scalable, stable, and reliable. Lastly, we will take a look at the graphs that I have collected during my production stress test of this application.

5.1 What went wrong

During my implementation phase, so many things went wrong. In this chapter, we will talk about the most exciting errors that had occurred. We will talk about what made these errors as well as how to solve them correctly. Most of these errors appear only in the production, where it is under real life conditions.

5.1.1 Buffer tables

ClickHouse needs to insert data in batches, but these batches need to hold their tempo, as we have talked about earlier in 3.1.4. This is one of the problems that we can encounter during production since you do not often do stress tests in development, but you test it on a small portion of inserts. We can solve this issue with many approaches, but none of them is perfect.

5.1.1.1 Explained more briefly

Each of our servers performs inserts to the ClickHouse independently. We are using the MergeTree family engine for the temporary table. Furthermore, this MergeTree family engine does not support frequent inserts nor small batches. If we violate one of these rules, the ClickHouse can reject our insert, or the merging operations within the ClickHouse becomes inefficient.

When we tested our setup in the production where we were inserting directly to the temporary table, it became unstable. Sometimes it threw inserting errors, and sometimes it did not. I contacted the ClickHouse team about this issue, and they recommended us to use the Buffer table. So our solution was to use the Buffer engine table. We used it and encountered other problems.

- Data that are being buffered are not persistent unless our batch is inserted. This is a significant flaw since we can lose data due to machine upgrade or application segfault. Even when we restart the ClickHouse, we lost data.
- We need to specify buffering policy, which consists of max/min time interval, max/min size of the batch, max/min rows in the batch. Lost data are based on our buffering policy, but it can be many thousands of lost records. These policies cannot be configured dynamically. Furthermore, there are not any rules on how to configure these policies correctly. You need to try them out and test for each specific use case.

Another approach would be to use Kafka in front of ClickHouse. We will talk about this use case in possible upgrades.

5.1.1.2 Solution

We took this Buffer approach in the implementation part because it is the most straightforward and less experimental or heuristic than the others. Buffer table solves this issue, and it is performing well. The only disadvantage is that when you restart the ClickHouse, then you lose data. These lost data are not significant, and it is often less than 0.001Kafka would solve this losing data problem because it is designed as a log storage pipeline. We will talk more about how Kafka could be used in possible upgrades.

5.1.2 Corrupted Zookeeper

During upgrading ClickHouse, we had a problem where ClickHouse got corrupted. We accidentally upgraded one node in the cluster. By default, if you upgrade one node in a cluster, ClickHouse updates all nodes to keep consistent communication between nodes because the next version can have

different communication practices. But accidentally, by upgrading one, the whole cluster got corrupted.

5.1.2.1 Explained more briefly

In the production environment where I have tested my setup, we had deployed the ClickHouse cluster on five machines in total, and we had five Zookeeper nodes on another five machines. We were not using the LTS version recommended by the ClickHouse team, but we were using the last stable release. The problem was raised when we upgraded a single ClickHouse node to the latest version. It looked like that ClickHouse had changed the internal representation of data, and other ClickHouse nodes which were using older data format were confused, and data integrity collapsed. The newly upgraded ClickHouse node changed everything in the Zookeeper, and old nodes were confused and quickly began to spam error log with an unknown format and possible data corruption. Since ClickHouses talk to each other via Zookeeper, so cluster got unhealthy.

5.1.2.2 Solution

The only solution was only to upgrade the rest of the nodes to the same version. After this problem, we discussed this problem with the ClickHouse team and were told to jump only for LTS version of the ClickHouse. This issue can be prevented by keeping package manager up to date. The ClickHouse was deployed on Debian on our cause. Furthermore, not all of these packages managers were up to date, so ClickHoue could not download the new version of ClickHouse to the rest of the nodes.

5.1.3 MergeTree errors

We encountered a problem where ClickHouse was not able to merge parts quicker. This means that there were more inserts than merges. This is unwanted behavior because when you reach a specific limit in the configuration, ClickHouse starts to reject new inserts. The overall architecture of how ClickHouse's MergeeTree engine work we have discussed earlier in 3.1.4.

5.1.3.1 Explained more briefly

So let's dig deeper into the MergeTree engine. In ClickHouse terminology, they use the word tempo. Tempo needs to kept when inserting into the MergeTree table engine. This tempo consists of frequent (but not too frequent) inserts and inserting data in large batches (but not too large batches). If we keep this tempo then ClickHouse will be happy. Otherwise, it will complain that it is not able to merge fast enough. This tempo configuration depends on hardware and traffic. It often occurs when you do not insert anything at all

to the ClickHouse, and suddenly if you push everything you have, when you do this, you break the tempo.

When you insert data into ClickHouse, a temporary file is created. This temporary file is later then merged in a large portion of data with other temporary files. This merging is not deterministic. It is only based on ClickHouse specific rules. The consequence of inserting too many small batches is that more of these temporary files are created that are scheduled later for merging. But if you schedule too many of these parts into merging, then the merging part becomes very inefficient, and ClickHouse begins to slow down. If we insert huge batches, it becomes inefficient because ClickHouse spends most of its computing time merging this specific temporary file.

5.1.3.2 Solution

The only way to adjust this is via config file with directive `parts_to_delay_insert` and `parts_to_throw_insert`. These values are otherwise hardcoded. However, these settings depend on your hardware. If we have beefy hardware that we can multiple default values six times and more.

5.1.4 Quantile T-digest error

Calculate percentiles is somewhat a difficult task when we have millions of records. When we used a standard linear calculation where we remember all values, and all of them are sorted, the only bottleneck is hardware resources. When we use this standard calculation for quantile and issue a SQL where we, for example, query for median for 2 hours, then the ClickHouse tries to load all values into RAM. Since we have millions of records, loading this whole segment of data into RAM leads to an application crash. Fortunately, ClickHouse allows us to use T-digest data structure for calculating quantiles for a vast amount of data.

5.1.4.1 Explained more briefly

The T-digest is a probabilistic data structure for estimating the median from either distributed data or streaming data. Internally, the data structure is a sparse representation of the cumulative distribution function. ClickHouse team poorly implemented this data structure in ClickHouse, and it had a problem of uncontrolled growth. For some input data, the internal state of the data structure grew significantly. We reported this issue, and it was fixed in merge request `#16680`[65]. Furthermore, this issue only showed up on materialized views aggregating a longer time, like a day or more.

5.1.4.2 Solution

We reported this issue to the ClickHouse team, so the only solution was to upgrade to the latest ClickHouse that consisted of this patch.

5.1.5 Metrics backfilling

Our ClickHouse client that sends data into the ClickHouse has a significant flaw. When our ClickHouse is out of reach or when the ClickHouse cluster is down, our ClickHouse client cannot successfully insert data and drops them. This is a problem since the data we are collecting from the web server are snowballing. So we need to implement some intelligent mechanism of discarding old data when an insert is unsuccessful.

5.1.5.1 Explained more briefly

The connection between a client application and the ClickHouse cluster can be unstable or can go down; thus, we need to implement buffering technique while the connection can not be established. If we would not implement buffering technique, this means that we can lose data. Or when ClickHouse rejects our insert we would lose data too.

5.1.5.2 Solution

In production, we have implemented a simple ring buffer that is being filled when ClickHouse is unreachable. When the ClickHouse is unreachable for a more extended period, old data is overwritten with fresh data because of the ring buffer mechanism.

5.2 Possible upgrades

It is not harmful to take inspiration from other companies. In one of the previous chapters, we have mentioned the Cloudflare blog[49]. In this blog, they talked about their pipeline using ClickHose. They put a Kafka cluster in front of ClickHouse. This means that their edge servers do not directly push data into ClickHouse, but they first insert it into Kafka. And they have clients that transfer data between Kafka and ClickHouse. This can solve problems like buffer table problem and backfilling problem. Furthermore, this can help us to use these collected data in more places because of Kafka. It can be more generic and reusable. Moreover, we can use this Kafka pipeline for our ELK pipeline. With this step, we could aggregate errors in ClickHouse, and we would have the advantage of frequent errors.

Next, since we developed our application for Linux and we are using sockets, we can tweak our sockets' settings. Now all workers send their data into a single socket. With correct settings for sockets, we can spawn multiple sockets

and distribute the load. This can be helpful because we can take advantage of multiple cores and read data from multiple threads. Nginx supports sending data to multiple sockets.

We could tweak our settings in ClickHouse a little bit more and offload old data to HDD from SSD. ClickHouse has its TTL, as we talked about earlier. This TTL mechanism allows us to move data when data becomes cold. It can help us to save money on disks. Since cold data are rarely accessed, they do not have to be on fast disks like SSD or NVME.

We could even set up monitoring of ClickHouse itself. ClickHouse has its system tables where it inserts statistics. We can aggregate these statistics to have a better overview of what is going on. Furthermore, with this monitoring, we can monitor ClickHouse resources. With these pieces of information, we can know when to upgrade ClickHouse machines or when to expand the disk space.

5.3 Numbers in graph

This web-server monitoring stack with some improvements was tested in production in one of the top five CDN. Our Golang utility was deployed to all servers for collecting data, which are aggregated in ClickHouse.

This architecture can scale well as well as it is distributed. In the production, the Golang utility was deployed to at least 1000 servers, and we had a ClickHouse cluster of 5 nodes. Moreover, ClickHouse node servers were equipped with 40 cores CPU, 128 GiB RAM, and 12 TB SSD.

These nodes were able to aggregate over 3.5 million requests per second and traffic, up to 13 TiB/s. This traffic is collected from the Nginx server across all regions. Most of the aggregated traffic is from Europe. We can see these results in 5.1 and in 5.2. These graphs are an output of Grafana.

Overall while testing in production, there was not any major issue, and everything mainly went flawlessly. Still, some things can be better, as we talked about in the previous chapter.

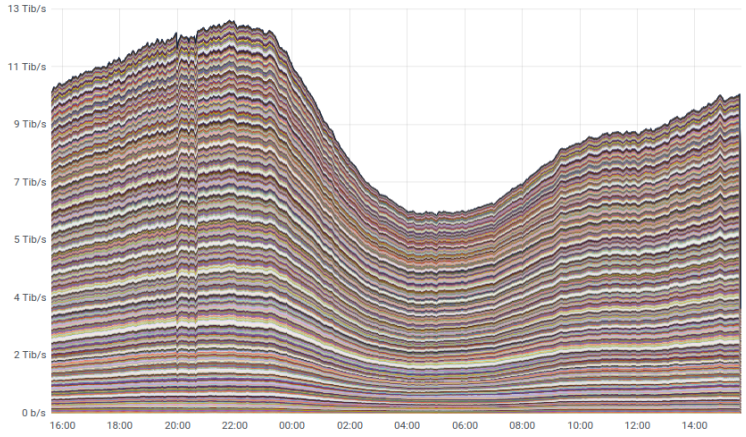

Figure 5.1: ClickHouse traffic

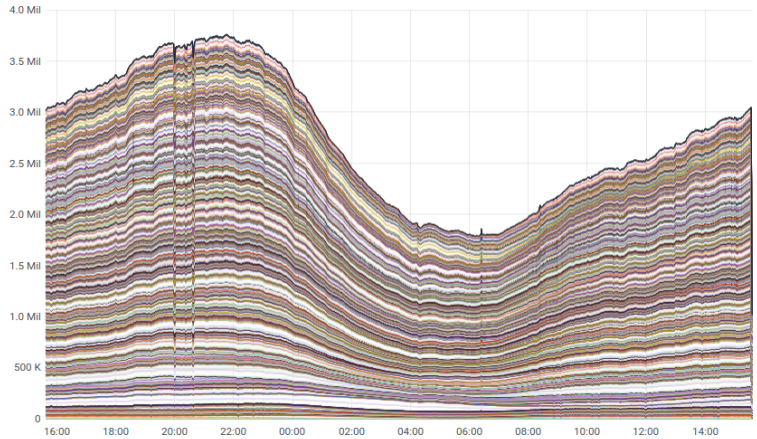

Figure 5.2: ClickHouse requests

Chapter 6

Outro

We have implemented a fully extensible, distributed, and scalable pipeline design. We have deployed this solution in the production environment consisting of thousands of web servers worldwide powering the network of one content caching network provider. At the time of writing this thesis, their servers handle more than 12 terabits of traffic per second, all of this processed using our monitoring pipeline. We have not encountered a single problem that we could not solve during the design and development of the pipeline.

6.1 Plans for the near future

Currently, we are developing a more stable pipeline that consists of Kafka in front of the ClickHouse. This feature adds stability to our solution. Furthermore, more applications can read raw logs as long as they can read data from Kafka. The idea behind this patch is inspired by Cloudflare blogs which we were describing earlier. We are now in the testing and debugging phase. As of today, Kafka is performing well and adds stability to our pipeline. We plan to increase stability and availability, and we are optimizing. Also, we plan to add better monitoring for ClickHouse itself and the Kafka and ZooKeeper.

Conclusion

This thesis had several objectives. The first was to introduce metrics logging and distributed systems. Then we gained knowledge of open-source tools that we can use. We finalized it by implementing a fully operational logging pipeline to help us with infrastructure performance, debugging, and health monitoring. Moreover, we used open-source tools, which we researched. Furthermore, we programmed a scalable Golang utility that collects data from the Nginx web-server, and it sends data into the ClickHouse.

In the first chapter, we explained some basics about monitoring. We dug deeper into what can be monitored and some introduction into basic metrics that can be monitored. We continued with some theories and caveats that go hand in hand with distributed setup.

In the second chapter, we talked about tools for monitoring.

The third chapter was dedicated to implementation, where we set up a primary monitoring pipeline that can be easily extended. We showed how to send data from Nginx into ClickHouse and continued with setting up ClickHouse. Then we talked about how to get visualized data in Grafana.

In the fourth and final chapter, we talked about caveats that occurred to me while testing in production as well, as we looked at results that were collected during testing in production.

We gained significant knowledge during the creation of this thesis. We improved our monitoring skills and built a fully functional logging pipeline. We discussed how our pipeline can be improved and how to fix common errors.

Bibliography

1. ELLINGWOOD, Justin. *An Introduction to Metrics, Monitoring, and Alerting* [online]. DigitalOcean, 2017 [visited on 2021]. Available from: `https://www.digitalocean.com/community/tutorials/an-introduction-to-metrics-monitoring-and-alerting`.
2. *Disk space* [online]. [N.d.] [visited on 2021]. Available from: `https://www.seagate.com/support/kb/why-does-my-hard-drive-report-less-capacity-than-indicated-on-the-drives-label-172191en/`.
3. *Hypertext Transfer Protocol – HTTP/1.1* [Internet Requests for Comments]. RFC Editor, 1999-06. RFC, 2616. RFC Editor. Available also from: `https://www.rfc-editor.org/rfc/rfc2616.txt`.
4. *round-trip delay time* [online]. 1996 [visited on 2021]. Available from: `https://www.its.bldrdoc.gov/fs-1037/dir-031/_4641.htm`.
5. KOPP, Michael. *Why Averages Suck and Percentiles are Great* [online]. 2012. Available also from: `https://www.dynatrace.com/news/blog/why-averages-suck-and-percentiles-are-great/`.
6. BEYER, Betsy. *The site reliability workbook: practical ways to implement SRE*. O'Reilly, 2018.
7. RAJAGOPALAN, Ramesh; VARSHNEY, Pramod K. Data aggregation techniques in sensor networks: A survey. 2006.
8. S, Surbhi. *Difference Between Centralization and Decentralization (with Comparison Chart)* [online]. 2017 [visited on 2021]. Available from: `https://keydifferences.com/difference-between-centralization-and-decentralization.html`.
9. GU, Zhaoquan; WANG, Yuexuan; HUA, Qiang-Sheng; LAU, Francis C. M. *Rendezvous in distributed systems: theory, algorithms and applications*. Springer, 2017.
10. PERENS, Bruce et al. The open source definition. *Open sources: voices from the open source revolution*. 1999, vol. 1, pp. 171–188.

11. *Grafana enterprise stack* [online]. [N.d.] [visited on 2021]. Available from: https://grafana.com/products/enterprise/.

12. *InfluxDB* [online]. [N.d.] [visited on 2021]. Available from: https://grafana.com/docs/grafana/latest/datasources/influxdb/.

13. *Grafana requirements* [online]. [N.d.] [visited on 2021]. Available from: https://grafana.com/docs/grafana/latest/installation/requirements/.

14. *Grafana Cloud VPS Hosting: SkySilk Cloud* [online]. 2021 [visited on 2021]. Available from: https://www.skysilk.com/grafana-vps-hosting/.

15. *Grafana hosting, Metrics and analytics dashboards as a Service (SaaS)* [online]. [N.d.] [visited on 2021]. Available from: https://www.stellarhosted.com/grafana/.

16. *Grafana Cloud* [online]. [N.d.] [visited on 2021]. Available from: https://grafana.com/products/cloud/.

17. *Why developers like Grafana* [online]. [N.d.] [visited on 2021]. Available from: https://stackshare.io/grafana.

18. RYCKBOSCH, Frederick. *Prometheus monitoring: Pros and cons* [online]. 2017 [visited on 2021]. Available from: https://jaxenter.com/prometheus-monitoring-pros-cons-136019.html.

19. KUMAR, Rajesh. *What is Prometheus and How it works?* [Online]. 2021 [visited on 2021]. Available from: http://www.devopsschool.com/blog/what-is-prometheus-and-how-it-works/.

20. ARILLA, Carlos. *Challenges using Prometheus at scale* [online]. 2020 [visited on 2021]. Available from: https://sysdig.com/blog/challenges-scale-prometheus/.

21. THE APACHE SOFTWARE FOUNDATION. *ZooKeeper* [online]. 2021 [visited on 2021]. Available from: https://zookeeper.apache.org/.

22. *ClickHouse DBMS* [online]. [N.d.] [visited on 2021]. Available from: https://clickhouse.tech/.

23. CLICKHOUSE TEAM. *Database engines* [online]. Yandex LLC, [n.d.] [visited on 2021]. Available from: https://clickhouse.tech/docs/en/engines/database-engines/.

24. CLICKHOUSE TEAM. *Table engines* [online]. Yandex LLC, [n.d.] [visited on 2021]. Available from: https://clickhouse.tech/docs/en/engines/table-engines/.

25. BLINKOV, Ivan. *ClickHouse consistency* [online]. 2019 [visited on 2021]. Available from: https://stackoverflow.com/questions/57312862/how-to-know-when-data-has-been-inserted-in-clickhouse.

26. NGSIOLEI. *Having consumer group member failover problem in ClickHouse Kafka engine* [online]. [N.d.] [visited on 2021]. Available from: https://github.com/ClickHouse/ClickHouse/issues/21118.

27. FILIMONOV. *Kafka: Exception during commit attempt: Local: No offset stored* [online]. [N.d.] [visited on 2021]. Available from: https://github.com/ClickHouse/ClickHouse/issues/18719.

28. YUUCH. *MaterializeMySQL do not drop rows, when I drop partition in MySQL* [online]. [N.d.] [visited on 2021]. Available from: https://github.com/ClickHouse/ClickHouse/issues/19707.

29. CLICKHOUSE TEAM. *Distributed* [online]. Yandex LLC, [n.d.] [visited on 2021]. Available from: https://clickhouse.tech/docs/en/engines/table-engines/special/distributed/.

30. CLICKHOUSE TEAM. *Buffer* [online]. Yandex LLC, [n.d.] [visited on 2021]. Available from: https://clickhouse.tech/docs/en/engines/table-engines/special/buffer/.

31. *Introduction to MongoDB* [online]. [N.d.] [visited on 2021]. Available from: https://docs.mongodb.com/manual/introduction/.

32. *MongoDB: A Scalable Database with Powerful Performance* [online]. [N.d.] [visited on 2021]. Available from: https://codecondo.com/mongodb-a-scalable-database-with-powerful-performance/.

33. ARJARAPU, Shyam. *Mastering MongoDB - Introducing multi-document transactions in v4.0* [online]. HackerNoon.com, 2019 [visited on 2021]. Available from: https://medium.com/hackernoon/mongodb-transactions-5654cdb8fd24.

34. *Limitations in MongoDB Transactions* [online]. 2018 [visited on 2021]. Available from: https://www.dbta.com/Columns/MongoDB-Matters/Limitations-in-MongoDB-Transactions-127057.aspx.

35. MONGODB TEAM. *MongoDB lookup* [online]. [N.d.] [visited on 2021]. Available from: https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/.

36. *MongoDB Consistency Features* [online]. 2015 [visited on 2021]. Available from: https://quabase.sei.cmu.edu/mediawiki/index.php/MongoDB_Consistency_Features.

37. *Elasticsearch: The Official Distributed Search and Analytics Engine* [online]. [N.d.] [visited on 2021]. Available from: https://www.elastic.co/elasticsearch/.

38. *Block Volume Elastic Search* [online]. [N.d.] [visited on 2021]. Available from: https://docs.oracle.com/en-us/iaas/Content/Block/Concepts/blockvolumeelasticperformance.htm.

39. *Elasticsearch* [online]. 2013 [visited on 2021]. Available from: https://aws.amazon.com/elasticsearch-service/the-elk-stack/logstash/.

40. MARQUARDT, Alexander [online]. 2019 [visited on 2021]. Available from: https://alexmarquardt.com/2019/06/15/improving-the-performance-of-logstash-persistent-queues/.

41. ELASTIC TEAM. *Logstash performance* [online]. [N.d.] [visited on 2021]. Available from: https://github.com/elastic/logstash/blob/master/docs/static/performance-checklist.asciidoc.

42. *Kibana* [online]. 2013 [visited on 2021]. Available from: https://aws.amazon.com/elasticsearch-service/the-elk-stack/kibana/.

43. SCOTT, Samuel. *These 15 Tech Companies Chose the ELK Stack Over Proprietary Logging Software* [online]. 2016 [visited on 2021]. Available from: https://logz.io/blog/15-tech-companies-chose-elk-stack/.

44. *Kafka* [online]. [N.d.] [visited on 2021]. Available from: https://kafka.apache.org/.

45. *What is Apache Kafka* [online]. [N.d.] [visited on 2021]. Available from: https://aws.amazon.com/msk/what-is-kafka/.

46. UTLEY, Jake. *Hiya's best practices around Kafka consistency and availability* [online]. 2019 [visited on 2021]. Available from: https://blog.hiya.com/hiyas-best-practices-around-kafka-consistency-and-availability/.

47. *Cloudflare* [online]. [N.d.] [visited on 2021]. Available from: https://www.cloudflare.com/learning/what-is-cloudflare/.

48. CLIFFORD, Tyler. *Cloudflare CEO: Dozens of U.S. states are using Athenian Project for election security* [online]. CNBC, 2020 [visited on 2021]. Available from: https://www.cnbc.com/2020/10/06/cloudflares-election-security-services-used-by-most-states-ceo-says.html.

49. BOCHAROV, Alex. *HTTP Analytics for 6M requests per second using ClickHouse* [online]. The Cloudflare Blog, 2018 [visited on 2021]. Available from: https://blog.cloudflare.com/http-analytics-for-6m-requests-per-second-using-clickhouse/.

50. *ClickHouse Software And Services* [online]. 2021. Available also from: https://altinity.com/.

51. *Usage statistics of nginx* [online]. [N.d.] [visited on 2021]. Available from: https://w3techs.com/technologies/details/ws-nginx.

52. *nginx* [online]. [N.d.] [visited on 2021]. Available from: https://nginx.org/en/.

53. *Module ngx_http_log_module* [online]. [N.d.] [visited on 2021]. Available from: https://nginx.org/en/docs/http/ngx_http_log_module.html.

54. GNU COREUTILS CONTRIBUTORS. *tail* [online]. 2021 [visited on 2021]. Available from: https://www.man7.org/linux/man-pages/man1/tail.1.html.

55. HPCLOUD. *hpcloud/tail* [online]. [N.d.] [visited on 2021]. Available from: https://github.com/hpcloud/tail.

56. *inotify* [online]. [N.d.] [visited on 2021]. Available from: https://www.man7.org/linux/man-pages/man7/inotify.7.html.

57. LOGROTATE CONTRIBUTORS. *Logrotate* [online]. 2021 [visited on 2021]. Available from: https://github.com/logrotate/logrotate.

58. *unix* [online]. [N.d.] [visited on 2021]. Available from: https://man7.org/linux/man-pages/man7/unix.7.html.

59. STEVENS, W. Richard. *UNIX network programming*. Prentice Hall PTR, 1999.

60. TORVALDS, Linus. *sock.c - net/core/sock.c - Linux source code (v4.5)* [online]. [N.d.] [visited on 2021]. Available from: https://elixir.bootlin.com/linux/v4.5/source/net/core/sock.c.

61. STEVENS, W. Richard; FENNER, Bill; RUDOFF, Andrew M. *UNIX network programming* [online]. Addison-Wesley, 2008 [visited on 2021].

62. CLICKHOUSE TEAM. *ClickHouse/clickhouse-go* [online]. [N.d.] [visited on 2021]. Available from: https://github.com/ClickHouse/clickhouse-go.

63. VERTAMEDIA. *ClickHouse* [online]. [N.d.] [visited on 2021]. Available from: https://grafana.com/grafana/plugins/vertamedia-clickhouse-datasource/.

64. CLICKHOUSE TEAM. *Combinators* [online]. Yandex LLC, [n.d.] [visited on 2021]. Available from: https://clickhouse.tech/docs/en/sql-reference/aggregate-functions/combinators/#aggregate_functions_combinators-mergestate.

65. HRISSAN. *prevent tdigest uncontrolled growth* [online]. 2020 [visited on 2021]. Available from: https://github.com/ClickHouse/ClickHouse/pull/16680.

Appendix A

Acronyms

CDN Content Delivery Network

DNS Domain Name Service

HTTP Hyper Text Transfer Protocol

POP Point of presence

TLS Transport Layer Security

IEC International Electrotechnical Commission

SSD Solid State Drive

LTS Long Time Support

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `images` ..... the screenshots of collected results
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis` ..... the thesis sources