Bachelorarbeit

A Lightweight Persistence System for Network Traffic Measurements

Verfasser
Michael Mente

angestrebter akademischer Grad
Bachelor of Science (BSc)

Wien, 2021

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Allgemein
Betreuer:	Oliver Michel, BSc MSc PhD

Contents

Abstract

While network traffic analysis in data centers is already an area where there has been much research and solutions, most of the solutions focus exclusively on live monitoring of network traffic. However, no matter how sophisticated the analysis tool, it is possible that a suspected fault is not detected in time or not at all. Therefore, it is necessary to explore the possibilities of storing network packets for future, specific analysis, as recent research has found that generic database solutions are unsuitable for a data-center deployment scenario with regard to several aspects. In the course of this work, a prototype was developed as a proof of concept that meets the requirements of a data center deployment, as subsequent evaluation shows.

1 Introduction

As never before, the internet plays a central role in our society, driven on the one hand by technological advances and on the other by social conditions such as the increased use of home office due to the COVID-19 pandemic. All of these circumstances have led to services supported by cloud computing providers being more heavily utilized than ever before, be it by the increased usage of video conferencing tools or by online streaming providers. As a result network traffic is not only increasing in volume, but also in velocity. However, due to the growing amount of network traffic, it is becoming more and more challenging to analyze this data. Currently, traffic is mostly processed live by means of special applications to detect possible issues. Despite these sophisticated algorithms, it is still possible that a security or performance issue cannot be detected in real time. Nonetheless, there are only limited possibilities to analyze a disturbance after it has already occurred [15]. One possibility to analyze a fault at a later point in time is a database system that stores all network traffic. However, this approach is very storage intensive and existing solutions do not scale to meet these high traffic demands. In addition, these systems maintain - among other things - mostly complex indices that require a lot of computational effort due to which the insertion rate is comparatively low. Currently there are two existing approaches for permanent data storage predominant: On the one hand, classic SQL databases have the advantage of straightforward data analysis due to their natural structure. Already existing SQL databases optimized for time-series-data have a comparatively high computational overhead and are therefore not fast enough to meet the requirements of a data center deployment. One of the fastest relational databases for this type of data is Timescale DB. However, it has been shown that the maximum ingestion rate of about 400k packets per second is not sufficient to log the traffic of a top of rack (TOR) switch as traffic rates can be more than a magnitude higher [15]. On the other hand, NoSQL databases offer a potentially higher ingest rate which is one of the key aspects to meet the requirements. Nevertheless, even NoSQL systems are not nearly performant enough to meet the requirements.

2 Related Work

2.1 Network Monitoring & Analytics

While many already existing network monitoring systems address different challenges and implement various solutions, most of the share that traffic is only monitored in a live manner. In terms of functionality, many implementations perform similar tasks and differ mainly in the level of granularity of the telemetry data collection or specializations, be it pre-processing in the data plane or customizations for a specific purpose. Network telemetry can be done in two ways, either by flow-based sampling, where metadata is collected directly, or by packet-based sampling, which might provide the best possible analysis but may

```
packetStream ( W )
  .filter ( p => p. tcp . flags == 2 )
  .map ( p => ( p . dIP, 1 ) )
  .reduce ( keys =( dIP, ), f=sum )
  .filter ( ( dIP, count) => count > Th)
```

Figure 1: Sonata packet stream detecting new TCP connections [9]

```
DEFINE{query name tcpDest;}
     Merge tcpDest0.time : tcpDest 1.time
     From tcpDest0, tcpDest1
```

Figure 2: GSQL stream merge statement [4]

require specialized hardware such as programmable switches [9, 14] to mirror the network traffic to the monitoring devices.

Sonata [9] is an example of a network telemetry system with a wide range of functionality at the packet level, and a focus on scalability as well as a unified query interface. This means that the queries can be made without the network operator dealing with technical details about the network interface device or the way of execution. An example of the usage of the expressive query language is shown in figure 1, which detects newly created TCP Connections.

Other solutions [4] focus on similarity to existing database languages and have developed their own SQL derivative. In the case of Gigascope, GSQL was developed that implements known syntax with new functions like *stream-merge*:

As shown in figure 2 *stream-merge* can combine streams from two different devices into one single stream, with the intention of turning also simplex links into proper network traffic flows and thus creating more flexible and meaningful analysis.
In the context of this work, however, the approaches of analyzing only partial traffic as pursued in [9] are not implemented. This is due to the fact that the planned system collects and stores the complete traffic flow first and then queries can be performed afterwards, similar to the implementation of Gigascope [4].

2.2 Record Persistence & Retrospective Queries

Although the idea of retrospective network traffic analysis has existed for a considerable time, the requirements for such solutions have changed dramatically. In 2003 Gigascope – a database for network applications –, one of the first solutions for storing network traffic, was developed[4]. Apart from the pure database functionality Gigascope provides a whole suite of network monitoring tools for common traffic analysis as well as more advanced tools for specific analysis, e.g., router configuration analysis. Gigascope was installed on backbone links of the U.S. network operator AT&T, OC48 routers to be precise. A large part of this paper focuses on the development of a domain-specific query language similar to SQL. However, the challenges encountered relate to the nature of stream databases. Since the system is deployed on a router, Cranor, Chuck, et al. state [4] that it is necessary to query streams from multiple interfaces simultaneously in case the router uses optical directed simplex links. Another important point in their work was the performance insights. At that time (2003) the traffic rates were as high as about 650 megabits on a backbone link from AT&T, which corresponds to a throughput of 1.2 million packets per second resulting in 500 gigabyte of generated data sets per day. Nevertheless, it is remarkable that such performance values have been achieved with commodity hardware on a 2.4 GHz dual CPU server. The most important findings regarding the performance of such a system were that in general, the earlier the reduction of the data volume, the higher the performance. However, it should be noted that the focus of the performance of stream databases is not on the speed of query resolution but rather on the maximum ingestion rate. In order to achieve such values, it was necessary to build complex hardware setups to achieve the high write rates to the hard disks, in particular the authors used disk striping, i.e., a RAID-0 hard disk array. Limited by the technology of the time - SSD technology was still very new and, above all, too expensive for large-scale deployment - such a hard disk array tends to lead to an increased failure rate, as further sources of error are introduced.

Since the development of Gigascope, completely new possibilities for improving such a system have opened up, due to the constantly advancing development of hardware. On the one hand, the trend towards multi core CPUs gives the possibility to process a continuous data flow in parallel more efficiently, on the other hand, the storage possibilities have improved due to the continuous research on SSD storage technologies, be it in speed, durability, or price. As a result of these advances, research has been conducted to determine whether general purpose databases could permanently store network traffic [15]. Analogous to the technological progress, the requirements for persistence systems have also grown - such database solutions should be able to handle even hundreds of millions of packets [15]. Modern database solutions potentially offer sufficiently high ingestion rates, but since some structure is required for subsequent analysis of network traffic for high-expression queries, NoSQL are not suitable for these tasks. Therefore, Michel et al. [15] chose TimescaleDB as a promising candidate, since it provides an SQL interface and optimizations regarding to time scale data, to measure its suitability for storing network packets. In the course of the research, it was shown that the query performance is ideal for this application area of interactive data analysis. Despite the promising results in query performance, SQL database systems are not particularly well suited for write-intensive tasks such as network packet storage. Another factor that reduces the insertion rate is the amount of additional computation required to provide the indices for the SQL interface. The measured insertion rate was only about 400k packets per second, which means that if the link is fully utilized, only a fraction of the network traffic could be stored.

To avoid this problem there is the possibility to process the packets beforehand in order to reduce the insertion rate. For this the grouped packet vector format (GPV) was used which was already engineered in prior work [20]. GPV is a method to group packets in a way that a composition of a packet record and a flow record is created which has already been processed efficiently on hardware [20]. This has the additional advantage that the required storage volume is reduced since several packets are stored in one GPV file.

Although some of the challenges the authors of Gigascope experienced back then may have been solved simply by technological advances, such as faster memory access thanks to SSD technology, but the requirements have been increasing considerably as described in [15]. Many times the traffic rates observed on a backbone link in 2003 now occur on a TOR switch [4, 15] where such a system would be deployed. However, important preliminary work has already been done that benefits this work, both in terms of performance insights and the handling of stream databases, and that even state-of-the-art solutions are not adequate to meet today's needs. However, if a system is to be used for permanent storage of network traffic, such methods are not sufficient to utilize the available storage as efficiently as possible. The problem especially with network traffic is the high rate of entries that potentially allocates a lot of storage to the timestamp. This problem has also been encountered previously in other areas where high ingestion rates are required [6]. To counteract this problem modern databases such as TimescaleDB use sophisticated encoding mechanisms to reduce the size of individual entries.

2.3 Compression Techniques

It has been shown that even specialized databases that have been optimized for so-called timescale data sets still have a major impact on storage efficiency [19]. Such database types are optimized for repetitive data sets, for instance a timestamp, which accounts for a relatively large proportion of the storage volume, depending on the required accuracy of the data. However, there are also further possibilities for increasing storage efficiency through optimization for the domain specific application scenario. A common possibility for increasing the storage efficiency is the reduction of the length of the data types, by only storing the difference to the previous value, also called delta encoding. However, it is also possible to apply multi-stage delta compression processes in succession in order to achieve even higher memory efficiency at the expense of the computational effort [17, 21]. In general, prior knowledge of the data has the potential to save more storage space as opposed to data agnostic compression techniques. This means, for example in the context of network telemetry that one can reduce delta values of timestamps to a small 16-bit or even 8-bit data type if one knows that a certain value will not be exceeded due to a certain packet rate [12, 21]. Another technique is dictionary encoding which stores a few repetitive values using a dictionary at a different storage location and exchanges the value with the index in the individual entries. However, this requires that the storage space of the index is smaller than the actual value to achieve an advantage [12].

In the domain of network telemetry, it is important to note that mostly only metadata, i.e., packet headers, packet lengths and timestamps are analyzed [4, 15, 20]. Apart from the IP 5 tuple (source and destination IP address, source and destination port, protocol) the timestamp makes up a large part of the storage volume depending on the resolution. However, further advantages can be achieved by also compressing the values corresponding to each timestamp as shown by Shi, Xuanhua, et al [19].

3 Implementation

3.1 Architecture

Due to the analysis of previous work I decided to split the implementation into two modules. This has the advantage of reduced complexity during the development phase, but also offers the possibility of a realistic deployment since the query module could be recompiled in some cases to solve particularly complex queries without stopping the whole system and thus not recording a period of time.

3.2 Writer Module

The writer module has the task of reading network traffic, either from a live network interface or from a prepared Pcap file, then indexing and storing this traffic using special compression techniques.

Previous research has found that common, general-purpose databases, as well as append-only databases optimized for time-series data, are held back in performance by the maintenance of complex indexes [15]. Therefore, this implementation omits such an index to achieve improved performance. As with append-only databases does the writer module only provide the possibility of adding entries, after which they can no longer be changed or deleted. This also means that the atomicity, consistency, isolation and durability (ACID) properties [10] required for classic databases are not given. This is due to the fact that this implementation is both a prototype as a proof of concept and that such measures would potentially require more computing power and thus reduce the ingestion performance on which the greatest attention has been paid. For further improvement on the performance only the most common network protocol types in the IPv4 space are captured. Although only TCP, UDP and ICMP traffic is considered, a traffic coverage of 97.4% is achieved (table 2), as measured by WAN traces [1] from an equinix data center provided by the CAIDA [3] organization.

[1] Measured from *equinix-nyc.dirA.20180517-125910.UTC.anon.pcap*[3]

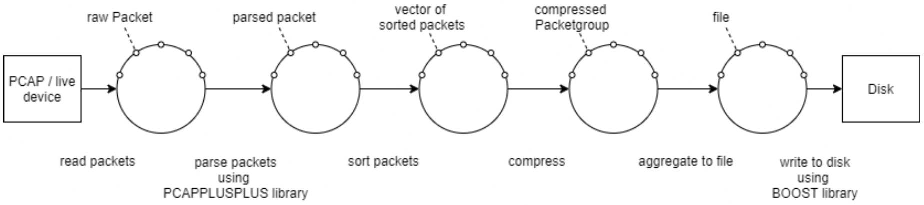

Figure 3: Data pipeline of writer module

In figure 3 the processing steps within the data pipeline of the writer module are depicted. The entire process is completely parallelized and the only kind of synchronization takes place with the help of a thread safe queue, as shown in the figure as circles.

3.2.1 Concurrent Queue

The performance of the queue is an integral part of the overall system and requires special care at the selection, therefore the decision was made to use a ready-made implementation rather than a custom development after it became apparent that the performance of custom implementations was not sufficient. It was an unexpected challenge to find a suitable implementation of such a queue as some existing solutions had limitations that made them unsuitable for this project. On the one hand these queues were limited to special data types like the open-source implementation from Intel's thread building blocks[2] , on the other hand the implementation of the commonly used Boost library[3] has the limitation that only simple destructors can be used.

The decision was finally made for an implementation[4] that is both significantly faster in terms of performance and overcomes the aforementioned shortcomings. The reason for the radically better performance, as shown in figure 4, is that data insertion is not synchronized since this implementation uses a separate queue for each producer and only upon dequeuing the data is read from the individual producer queues. This causes the disadvantage that the order of the dequeued elements can be different than they were inserted, but this poses no problem in this deployment scenario. Besides, in a multi producer multi consumer scenario the order is not deterministic anyway[5].

3.2.2 Packet Ingestion

The first step of the whole process is the ingestion of network traffic from either a live network interface or a specified Pcap file. This is done using the Pcapplusplus library[5] which provides a simplified interface to read traffic from

[2] `https://github.com/oneapi-src/oneTBB/blob/master/src/tbb/concurrent_bounded_queue.cpp` (Accessed on 13.07.2021)

[3] `https://github.com/boostorg/lockfree/blob/66f66c977038cb8a316bfc242de1843df9302613/include/boost/lockfree/queue.hpp` (Accessed on 13.07.2021)

[4] `https://github.com/cameron314/concurrentqueue` (Accessed on 13.07.2021)

[5] `https://pcapplusplus.github.io/` (Accessed on 13.07.2021)

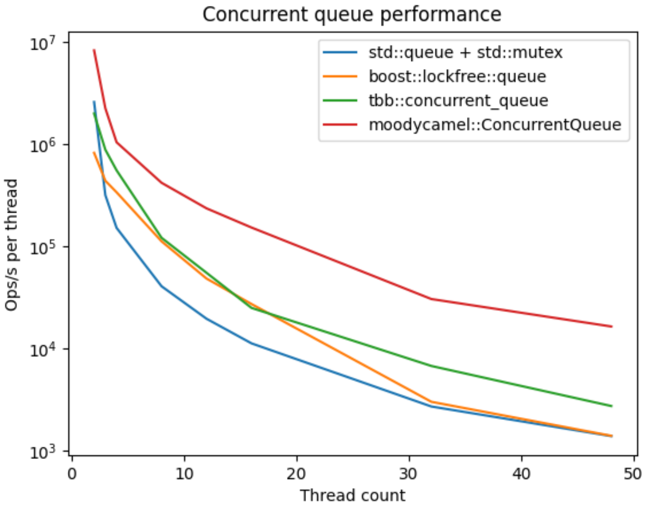

Figure 4: Concurrent Queue Performance as tested on AWS 32-core instance by creator of moodycamel::ConcurrentQueue[5]

network devices and then parse it without significant performance impact.

Live Capture In the case of live capture, an instance of a capture device is created in a separate thread after a filter is applied to that device to receive only the selected protocols, in this instance TCP, UDP and ICMP. All packets passed by this filter are added to the queue already mentioned (section 3.2.1). In the meantime, the main thread waits for the program to finish by user input.

File Capture Alternatively, if needed for benchmark purposes, the network packets are read from a Pcap file, this is done in a similar way as the above mentioned Pcapplusplus library also allows to read these files. Thus first a filter is applied to the file-device as in live capture and then the filtered packets are enqueued.

If the packet ingestion is finished, either because the Pcap file is completely read or because the user has signaled this during a live capture, a flag is set to indicate that this step has already been completed. The purpose of this is to ensure that all subsequent steps simultaneously in parallel in other threads finish processing all remaining packets in the system before the program is terminated.

3.2.3 Packet Parsing

In this step, only the data required for further processing is extracted from the parsed network packets using the Pcapplusplus library. This step could

Field	Bytes	Description
v4Src	4	IPv4 source address
v4Dst	4	IPv4 destination address
portSrc	2	source port (only TCP & UDP)
portDst	2	destination port (only TCP & UDP)
protocol	1	protocol number as specified in RFC 3232
length	2	length of packet including IPv4 header
tv_sec	8	seconds since 00:00:00 UTC, 1st January 1970
tv_usec	8	nanoseconds since last second

Table 1: Fields of IP tuple and timestamp associated with a single packet stored

also be omitted, but the earliest possible reduction of the data volume leads to a significant increase in performance, as already noted by Cranor et al.[4]. The individual fields that are subsequently processed as well as the size of the individual data types are shown in table 1.

3.2.4 Packet Sorting

After the necessary data has been extracted, pre-processing for compression is performed in the sorting step. As the analysis[6] has shown, 84.7% of the relevant traffic is transmitted using the TCP protocol as shown in table 2. This results in a comparatively large overhead in terms of the number of packets due to the protocol (TCP handshake, TCP tear-down), especially for short network flows.

Therefore, in this step all packets (respectively the previously extracted necessary data) are matched by one of their IP addresses within a certain time interval. This is done with the help of a map data structure, which is *flushed* (values of the map are enqueued for the next step and map is cleared) every two seconds. When a new packet arrives, it is checked whether an entry with the source or destination IP address already exists. If this is the case, the new packet is assigned to the others, otherwise a new entry is created based on the source IP address as shown in algorithm 1.

The greater the number of packets that can be assigned to an IP address, the better the compression in the next step. Therefore it is important to note that in this step the first trade-off between compression and performance has to be made. This is due to the fact that, as mentioned above, each step is executed in parallel in multiple threads and each thread holds its own map instance. This means that in a certain period of time there are several mappings to the same IP address according to the number of threads. It is possible to counteract the negative effects of increased thread count to a certain degree by increasing the flush interval, but the interval should not exceed the average duration of a conversation, otherwise this will result in increased memory usage and therefore

[6] Measured from *equinix-nyc.dirA.20180517-125910.UTC.anon.pcap*[3]

Algorithm 1: Packet matching algorithm

```
queue outQueue;
map<ipAddress, list<packet>>map;
timeSinceFlush = now();
while not previousStepFinished or not inQueue.empty() do
    packet = inQueue.dequeue();
    if map.hasKey(packet.v4Src) then
        (map.at(v4Src)).insert(packet);
    else if map.hasKey(packet.v4Dst) then
        (map.at(v4Dst)).insert(packet);
    else
        list<packet>list;
        list.add(packet);
        map.add(packet.v4Src, list);
    end
    currentTime = now();
    if timeSinceFlush - currentTime > 2 seconds then
        outQueue.enqeue(map.values());
        map.clear();
        timeSinceFlush = currentTime;
end
```

reduce overall performance. It has been observed that four threads and a flushing interval of two seconds offer a good balance. Experiments with a singleton instance of a map accessed concurrently have revealed that the additional synchronization overhead has a rather negative impact on performance and does not scale well with increasing thread count.

Therefore it is necessary to use a map with excellent insertion performance to keep the thread count as low as possible for best possible packet throughput as well as compression. As shown in figure 5, the time required to insert a given number of entries is lowest for different map implementations using either linear robin hood hashing or hopscotch hashing. The choice was in favor of the robin hood hashing implementation[7] considering its ability to avoid collisions due to the reduced clustering and the fact that the better memory efficiency of hopscotch hashing was less important than the overall insertion performance[7].

3.2.5 Packet Compression

After the pre-processing step by sorting, the actual compression step takes place. Each time a group of packets that have been assigned to each other according to an IP address, they are calculated into a new compressed object that contains a multitude of packets that can no longer be accessed individually in their compressed form. As shown in algorithm 1 the values stored in the map of the

[7] `https://github.com/Tessil/robin-map` (Accessed on 13.07.2021)

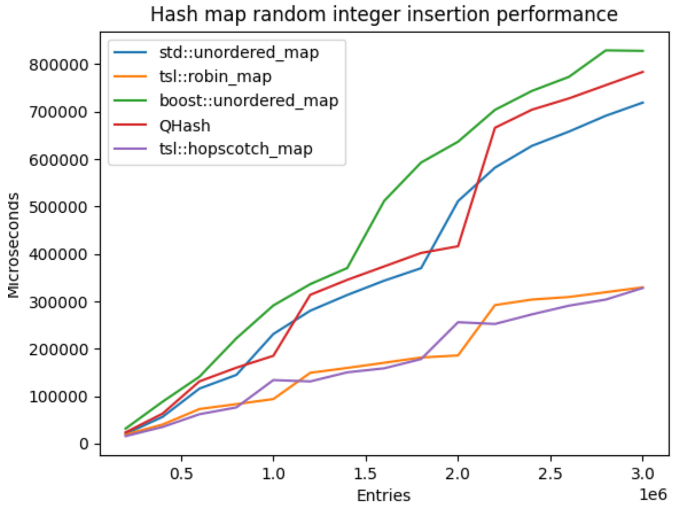

Figure 5: Hash map insertion performance comparison as tested of creator of tsl::robinmap[7].

previous step are enqueued and each list (an individual value-entry of the map) will correspond to one *compressed bucket*. However, it should be noted that if the packet rate drops, the compression rate also drops slightly, since, due to the flushing interval shown in algorithm 1, the maximum possible number of packets that can be assigned to an IP address ultimately impacts the amount of compressed buckets since each of them now holds a smaller number of packets compared to higher packet rates.

Dictionary Encoding Since it can be assumed for all packets in those lists that one IP address must always be the same, it is already possible to reduce the storage volume by replacing the repeating four byte long IP address, with a one byte Boolean flag, which indicates whether the other, different IP address is the source or destination address. This alone already brings a saving of 3 bytes per packet or 9.5% respectively. A further optimization offers dictionary encoding within the buckets for the remaining IP addresses. It was observed that in the majority of cases less than 256 different IP addresses occurred within one bucket. Consequently, the second IP address can be replaced with an eight bit index. However, if more than 256 IP addresses occur, a new compressed object must be created to store the remaining packets. The percentage improvement of this optimization depends strongly on the type of traffic but storage efficiency gains up to 14.0%[8] but at least 4.3%[9] have been observed.

[8] Measured from *equinix-nyc.dirA.20180517-125910.UTC.anon.pcap*[3]

[9] Measured from *equinix-nyc.dirB.20180517-134900.UTC.anon.pcap*[3]

Delta Encoding The greatest memory saving potential within a packet can be calculated by optimizing the timestamp, since it accounts for 51% in the packet's uncompressed form, i.e. 16 of a total of 31 bits as shown in table 1. An ideal method for such cases is the technique known as delta encoding[12], since only the difference between the timestamp and a known point in time is stored. In this implementation, each packet contains the Unix timestamp and the nanoseconds since the last elapsed second, as shown in table 1. However, before delta encoding is applied, it is possible to reduce the timestamp to a single 64 bit value without losing precision[10]. It is always the time difference to the first packet in the list that is measured. In a single threaded environment, there would be further opportunities for savings by storing the difference to the previous timestamp. However, it should be noted that due to the non-deterministic order of the packets, which is caused by the nature of multi-threading, no assumptions of this kind can be made for the list of packets which is processed sequentially. Therefore, it is also necessary that the delta value, i.e. the difference to the timestamp of the first packet in the list can be negative. Nevertheless, the resolution of the timestamp can be reduced to a quarter of the original size. The delta value is a signed 32 bit integer as the highest difference observed[11] was as high as 2^{30} nanoseconds and it is necessary that the highest possible interval is equal or greater as the flushing interval described in algorithm 1 used by the previous step. In this implementation, however, a distinction is made between the first packet and the remaining packets within a compressed bucket, since the first packet inserted is the starting point for the timestamp and also the IP address, which is repeated for all subsequent packets. Thus delta compression and dictionary encoding is applied instead starting from the second packet, but the timestamp is already calculated down to a single value.

3.2.6 Aggregation

During the aggregation phase, many compressed bucket objects are gathered and then written collectively into a single file. A maximum of one million of such compressed buckets are aggregated, but similar to algorithm 1, the lifetime of such a *metabucket*, i.e. the object containing the compressed buckets, is limited to ten seconds. The reason being, if the packet rate is significantly lower over a longer period of time, files are continuously generated which can then be excluded more granular in advance by the query module as explained in section 3.3.2. Since there is no complex index, the files corresponding to each metabucket object are named just by the time period they contain. Therefore, the earliest and latest timestamps are read from each compressed bucket to generate the filename. Since this step requires only a relatively small amount of computation, it is just performed by a single thread.

[10] This is only possible until the year 2038 since the first 32 bits are assigned as seconds and the last 32 bits as nanoseconds. After that the resolution could be reduced to 10 nanoseconds which is not a significant limitation in practice as most network interface cards cannot measure time to nanoseconds, the test data used in this project has a precision of 15 nanoseconds[3].

[11] Measured from *equinix-nyc.dirB.20180517-134900.UTC.anon.pcap*[3]

Experiments altering the amount of compressed buckets within a meta bucket have shown that the chosen value of one million compressed buckets represents a good balance between the smallest possible size for easy queryability and acceptable I/O overhead due to the operating system. This means that if the metabuckets were too small, there would be so many I/O requests to write that they would be queued and thus degrade overall performance in the next step (section 3.2.7). From the query module's point of view, it is desirable to keep the files as small as possible, as this allows a more precise selection to be made in advance and consequently reduces the decompression time, as described in section 3.3.2.

3.2.7 Writing

In the final step, the previously created files are serialized using the Boost libraries and written to the hard disk. Here, two threads work simultaneously to achieve optimal utilization of the write capacity. It has been shown that the use of more than two threads does not bring any advantages, since the files are not created quickly enough and the threads use a polling strategy that requires constant computing power that could benefit other steps otherwise.

3.3 Query Module

The query module has the task to read the binary files generated by the writer module, to make certain queries, to perform aggregation functions, but also in case the analysis with the query module alone is not sufficient, to restore them to a Pcap file for further analysis in other tools. As mentioned before, this system does not create an index comparable to current database solutions. Therefore, it is necessary to take other measures that may seem partially naive but are still remarkably effective as shown in section 4.3, however, the focus of such a system is not on the performance of the resolution of the queries but on the ingestion rate, as opposed to common general purpose databases[1, 4, 15, 17]. As shown in figure 6, the number of steps a packet passes through in the data pipeline is significantly lower than in the writer module. However, a large part of the steps carried out in advance is the parsing of the query and the subsequent selection of the individual files to be read.

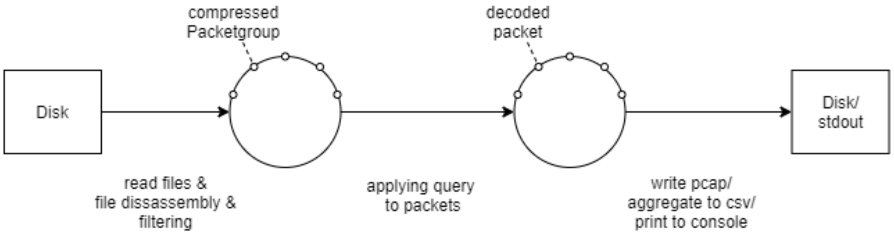
compressed Packetgroup
decoded packet
Disk
Disk/ stdout
read files & file dissassembly & filtering
applying query to packets
write pcap/ aggregate to csv/ print to console

Figure 6: Data pipeline of reader module

3.3.1 Query Language

During development, it was a priority to have the ability to dynamically parse different queries without having to recompile code. Thus the choice was made for a parametric interface whereby the query can be specified with a proprietary query language. However, unlike previous solutions, as mentioned in section 2.1, the query relates only to data that has already been stored and generated by the writer module, which is referred as long term query in existing literature [22]. This means that the query language is syntactically significantly different from the design of the Sonata implementation [9] shown in figure 1 or GSQL that still maintains a complex query (Figure 2). Compared to a classic SQL implementation, in this scenario there only exists one single table from which entries must be retrieved as efficiently as possible. Therefore, the correct term should rather be "filter language". Already prevailing solutions for such a scenario are employed in tools such as Wireshark [2, 11], where this query language is derived from.

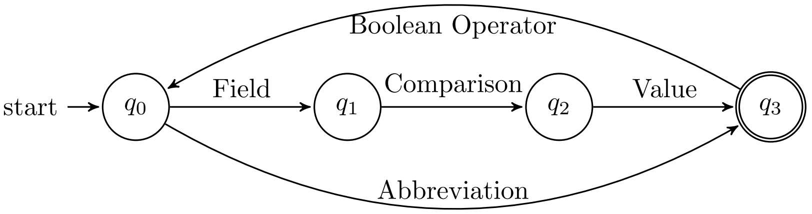

Figure 7: Non-deterministic finite state machine representation of query language

Syntax The automaton shown in figure 7 evaluates to the following regular expression which is left-associative:

$$(fcv|a)(b(fcv|a)^*$$

where

$$f \in \{frame.time, frame.len, proto, ip.src, ip.dst, ip.addr, port, port.src, port.dst\}$$

$$c \in \{==, !=, <, >, <=, >=\}$$

$$a \in \{tcp, udp, icmp\}$$

$$b \in \{\&\&, \|\}$$

and v is depending on the choice of f an user specified integer value, except if filtered by an IP address in which case an IP address in regular format, i.e. 10.0.0.1 can be passed as v. Furthermore if the filter is specified to be

frame.time the value must be formatted like the following {month day, year hours:minutes:seconds} e.g. Jul 18, 2021 16:00:00.
In addition, ready-made filters are available as abbreviations, in this case for the protocols similar to existing solutions. For instance, the filter *tcp* is equivalent to *proto == 6*.

3.3.2 Query Resolving

As explained in section 3.3.1, the query language can be represented as an automaton. This allows to build a parse tree from any possible query as shown in figure 8. However, it is important to note that the language does not support bracketing. Since the tree is evaluated internally from top to bottom, the example shown in figure 8 is interpreted as follows:

$$(\mathit{ip.addr}! = 0.0.0.0\ \&\&\ (frame.time < Jun\ 15,\ 2021\ 12:00:00\ ||\ \mathit{udp}))$$

This representation, which is possible as a tree (Figure 8), allows a relatively simple programmatic conversion into a class structure using the programming paradigm of the composite pattern[18]. This automatically yields simple optimizations such as that if filters are linked by the Boolean *or* operator, only partial evaluation need to be performed, which saves computational power if a branch is evaluated as true.

$$\mathit{ip.addr}! = 0.0.0.0\ \&\&\ frame.time < Jun\ 15,\ 2021\ 12:00:00\ ||\ \mathit{udp}$$

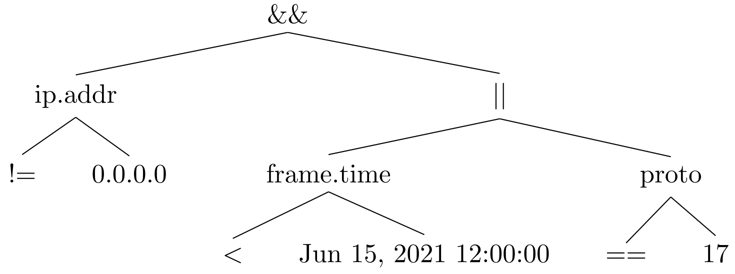

Figure 8: Parse tree representation of query

Pre-filtering So far, only the filters at the package level have been mentioned, but optimizations are made for better performance depending on how the query is formulated. On the one hand, if the query contains a time specification, i.e. it is filtered according to a time period or point in time, only the binary files that span this time period are selected in advance. This is especially important for such a system that potentially runs over a very long period of time, as it allows filtering to a relatively precise time interval, which is at least as precise as the lifetime of a single file, as described in section 3.2.6. For the reason just discussed, a query containing a timestamp is highly likely to be found in a subsequent analysis anyway. This pre-filtering also takes place at a second granularity level for each *compressed bucket* after extraction from the *metabuckets*. Although the additionally required storage of the contained period slightly reduces the efficiency of the storage usage, it provides a useful optimization since the lack of proper indexing options, and thus the query performance can be greatly increased since the decompression process is eliminated in many cases. On the other hand, if filtering by a certain IP address takes place, after the binaries have been determined before decompression takes place, a more refined selection can be made by analyzing the dictionaries of the individual compressed buckets. In addition, the time filters are also applied once more to the compressed buckets to reduce the number of packets as efficient as possible.

3.3.3 File Disassembly

After the binary files have been determined, they are read by several threads simultaneously, the number of which is dynamically scaled depending on the amount of files to be read. These threads reapply the pre-filters mentioned above to the compressed buckets and enqueue them to be processed in the next step if they pass all filters. Here the same queue data structure is used as in the writer module (section 3.2.1).

3.3.4 Query Application

Afterwards the decompression takes place in the next step so that the filters can be applied to the individual packets immediately, which cannot be accessed within the compressed bucket, as only the stored time period and the IP addresses, which have already been queried in the preceding step, can be accessed. Furthermore, this step takes place in several threads at the same time. Since there are no dependencies between threads, this step scales very well and the only thing to consider is to keep the number of accesses to the queue low. In order to minimize the interactions that could otherwise cause unnecessary waiting times on the queue, this processing phase, as well as others, makes use of batch enqueueing operations.

3.3.5 Post Processing

After the individual packages have been determined, the post-processing phase takes place. There are three ways in which the output can be presented.

Std Out If not specified otherwise via the parametric interface, a table of the data of the packets is presented on stdout showing all the fields as shown in table 1.

Pcap Another possibility is the ability to rebuild Pcap files that allow a network operator to include other tools for further analysis, such as Wireshark.

Aggregation One feature that sets this tools apart is the ability to apply aggregation operations to the packets. There are three parameters that can be specified. Firstly the interval specified in microseconds for which the aggregation operation should be performed, secondly which statistical operation should be applied to the packets and thirdly which field of the network packet shown in table 1 should be considered for the statistical operation.

The selected statistical operations were inspired by SQL aggregation operations and are: *sum*, *mean*, *min*, *max*, *count* and *count_dist* which is derived from the SQL keyword distinct and counts the number of unique objects. This means, for example, that one can measure the used bandwidth by aggregating the sum of the length of the packets in a certain time interval. By default, unless parameters for aggregation are specified, this is set for an interval of one second. The result of these queries, by means of aggregation, are saved as a CSV file and thus allow easy visualization and further machine processing.

4 Evaluation

As mentioned in the beginning, the performance of the implementation can be a very important aspect as current systems are not capable of storing network traffic permanently. Here one can distinguish several aspects of the performance. On the one hand there is the insertion rate of entries, and on the other hand there is the compression, which has to be done fast with low computational effort to still achieve a usable compression rate.

In order to achieve realistic, consistent and repeatable results, a predefined data set[12] was used for all measurements. This network trace does not originate from a data center scenario but is a WAN trace, which does not matter in this case of ingestion rate. This dataset is a one minute long capture from May 2018 from an Equinix data center on a WAN link between New York City and São Paulo[3]. It was chosen specifically due to its size and therefore the number of packets contained, which is just over 100 million. For this reason, the effective performance of the algorithms involved in the data pipeline can be better evaluated and the relation to the constant overhead needed to start the threads is minimized and does not have such a significant impact.

4.1 Record Ingestion

Since for the ingestion rate pure hardware performance and computing power are in the foreground, it is important to note that the experiments were performed within a virtual machine on a consumer grade personal computer with a 8c/16t@4.2 GHz CPU, out of which 12 threads were assigned to the virtual machine. The writer module uses all 12 threads at the same time, therefore, it is assumed that if such an implementation is tested on actual server hardware, the performance could be increased significantly due to the increased thread counts

[12] *equinix-nyc.dirB.20180517-134900.UTC.anon.pcap*[3]

of the hardware. To achieve valid results, each benchmark was conducted at least 100 times in an automated fashion.

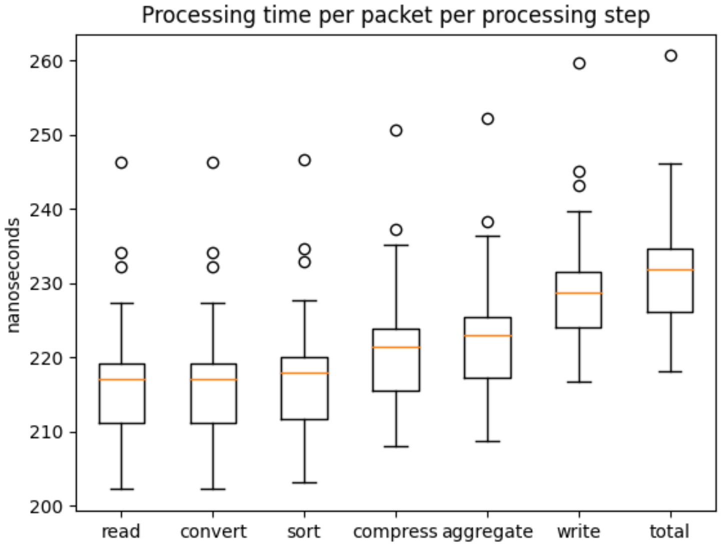

Figure 9: Cumulative packet handling times per processing step

As can be seen in figure 9, the compression step and the file creation process are the most time consuming. Here the diagram illustrates the complete time needed until the process is finished. Since each step can only be completed in dependence of the previous one, i.e. the next step cannot be faster than the previous one, it is easy to see where the most computing time within the data pipeline (Figure 3) is demanded.

However, since the compression step does not involve parallel dependencies as e.g. the sorting step (section 3.2.4), compression time can be reduced by increasing the number of compressing threads. Nonetheless, this was not advantageous in the test environment since only 12 threads were available that were already occupied for other tasks. Furthermore, it can be assumed that the write overhead is disproportionately high due to the execution in a virtual environment[16].

Despite the limitations just mentioned, an average packet handling time of 242.25 nanoseconds as shown in figure 10 could be observed, which corresponds to a packet rate of slightly more than 4.1 million. However, this estimate is rather conservative since it is based on the total execution time, which means that the shutdown sequence of the program, in which actually no more packets are processed, was calculated as packet time. If the calculations are based on the required write time, - after all, the packet processing ends with it being written to the solid-state disk - the packet rate would actually be about 4.35

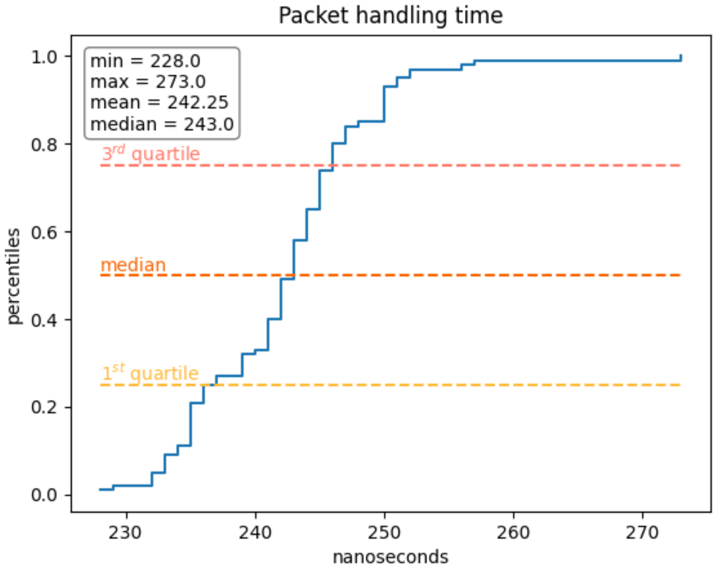

Figure 10: Empirical cumulative packet handling time

million on average.

With this, insertion rates of over 100x the speed of regular systems, such as PostgreSQL, have already been surpassed and even the insertion rate of specially optimized systems, for instance TimescaleDB, is over an order of magnitude lower[15] than the specialized implementation developed for this scenario.

It should be noted, however, that the course of the packet processing time drops considerably over the duration of the benchmark compared to the first run as shown in figure 11. This is due to the following reasons: on the one hand, the guest system may not have been provided with sufficient RAM by the host, which must then be made available first, and secondly, caches must be reserved and filled via the memory ballooning technique of Virtualbox on windows hosts[16]. This could be observed consistently over all benchmark executions.

4.2 Storage Requirements

Another important aspect to be evaluated in the context of this work is the efficiency of storage usage. Here it is possible to measure exactly how high the compression rate is compared to the original data.

It is important to note that the WAN trace mentioned above was used for the measurement, which is actually a worst-case scenario regarding the compression step and does not represent the traffic within a data center on which the algorithms perform best. This is due to the fact that the effectiveness of

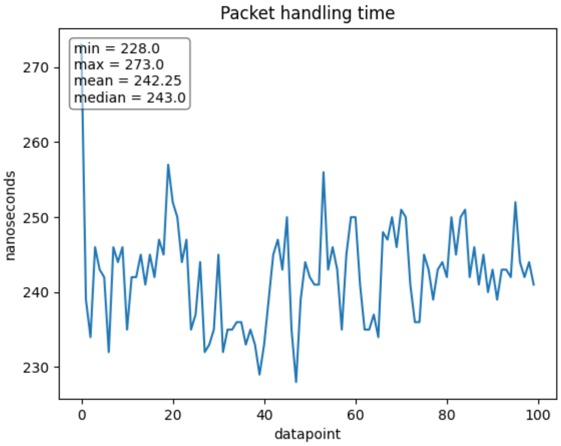

Figure 11: Empirical packet handling times during benchmark

Protocol	Packets Total	Packets Captured
Internet Protocol Version 4 TCP	82.5%	84.70%
Internet Protocol Version 4 UDP	14.4%	14.78%
Internet Protocol Version 4 ICMP	0.5%	0.52%
Internet Protocol Version 4 Other	1.1%	0%
Internet Protocol Version 6	1.5%	0%

Table 2: Traffic composition of WAN trace

the multilevel[13] dictionary compression process decreases massively the more different IP addresses occur in a certain period of time, as already illustrated in section 3.2.5.

Despite these difficult conditions, a remarkable compression rate can be achieved as visualized in figure 12, especially considering the comparatively low computational effort.

To determine the multiple stages of data reduction, the composition of the various protocols in the test file was first determined, as shown in table 2. The weighted average was calculated using the minimum possible header length and the frequency of occurrence of the respective protocol and is therefore 38.16 bytes long on average. Considering an average packet length of only 354.45 bytes per packet[14] an overall compression rate of about 20:1 can be observed as shown in figure 13. However, since trimming the payload cannot be considered

[13] It can be considered multilevel due to the fact that at first all IP addresses are sorted during the sorting step and then substituted within the sorted buckets, and then dictionary encoding is applied again, this time in a typical way during the compression step

[14] Measured from *equinix-nyc.dirB.20180517-134900.UTC.anon.pcap*[3]

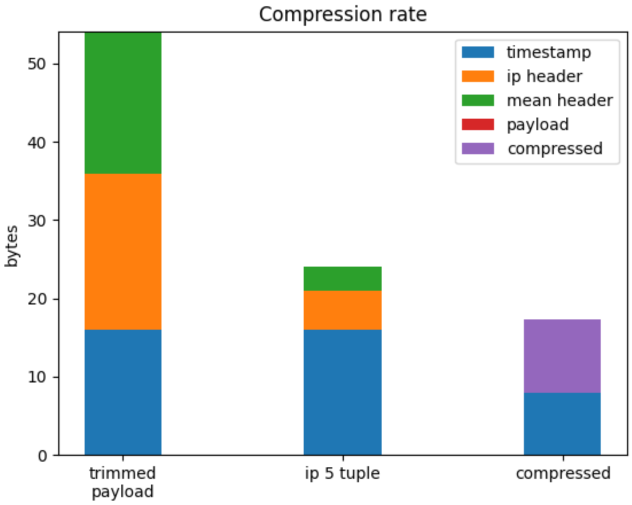

Figure 12: Compression rate and compressed packet composition

compression, the true compression ratio is actually 1.8 : 1 since after extracting the data to be stored as shown in table 1 it is reduced to only 17.25 bytes on average (Figure 12).

However, tests were also carried out with other data sets. Therefore, another data set was selected from the same data center, but at a time when the traffic rate was comparatively very low. Here, as already suspected, it was confirmed that even a lower packet rate, to a certain extent, can improve the overall compression rate significantly[15], in this case to 2.07:1, since the fewer IP addresses communicate in the same time period and can thus be better assigned in the sorting step.

In addition, in order to get as close as possible to a hypothetical data center scenario, the traffic from a single host was intercepted and then fed into the program as a Pcap file for simulation purposes, which resulted in a compression ratio of 2.92:1. As can be seen in figure 14, the compression rate can vary greatly depending on the type of traffic.

If such a system is connected to a data center network via a 100 Gbit/s link, a data rate of 1.375 GByte/s can occur when the link is fully utilized, exclusively in headers. This is true if the traffic has a similar composition as the measured file[16] and thus approximately 11% of the total traffic volume is accounted for by the headers. However, it can be assumed that the average packet size of 354

[15] This only holds true for a traffic composition similar to the tested WAN trace, where the traffic rate was lower due to less hosts communicating and not the average conversation length getting shorter

[16] *equinix-nyc.dirB.20180517-134900.UTC.anon.pcap*[3]

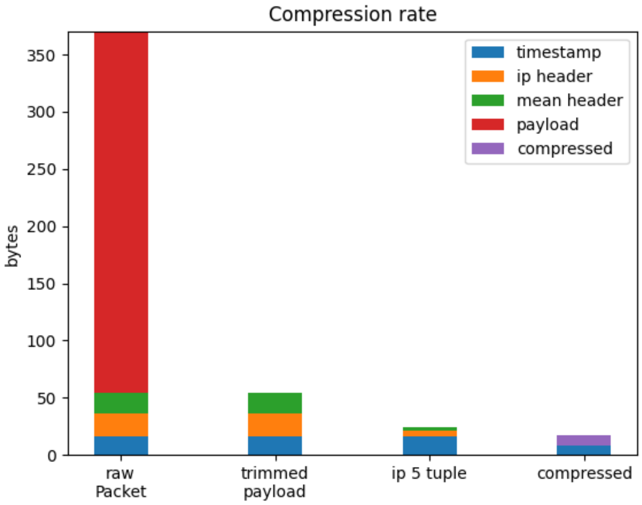

Figure 13: Compression rate and compressed packet composition including payload

bytes per packet, on which the calculated 11% is based, will not be below that in a data center scenario.

In general, the assumption was made that a write budget of one gigabyte per second is available to an instance of the persistence system. This is readily achievable with modern SSD technology and is already being deployed in data centers[23]. From this, it can be concluded that the compression ratio must not fall below 1.375:1, otherwise the write budget of 1 GByte/s would be exceeded. The observed values of the measurements have shown that this is extremely unlikely, as the lowest observed rate was 1.8:1 in a worst case scenario and is still significantly better than the lower bound. In an ideal scenario, a compression ratio of almost 3:1 was observed, which is the closest to a possible application in a data center. Therefore, it can be assumed that the compression ratio for such a case is greater than 2:1 and thus well suited for not encountering storage bottlenecks.

Furthermore, Michel et al.[15] did not mention any issues regarding the limits of the write performance of the disks in the experiments with PostgreSQL, which does not apply any compression techniques by default[13].

4.3 Query Capabilities

Another important aspect to be evaluated is the usability and expressiveness of the query system, since ultimately it is only by means of the query system that database and other persistence systems become fully effective.

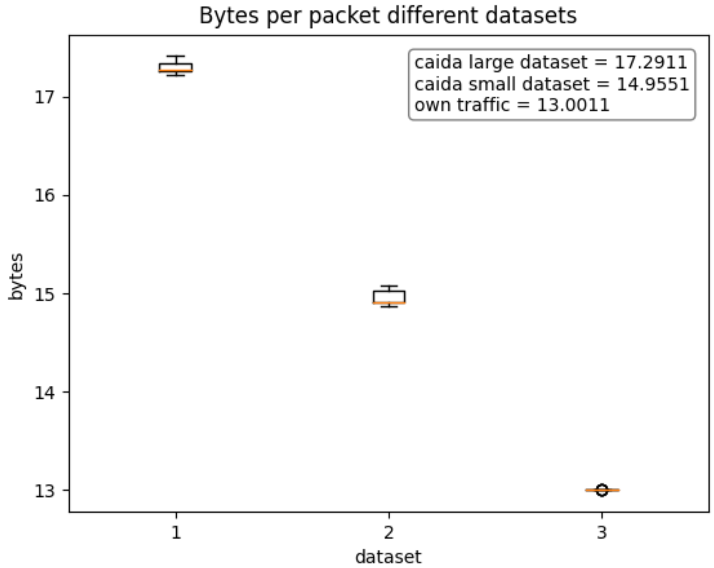

Figure 14: Compression rate on different data sets

Unparsable Query	Logically Equivalent Query
(udp ||tcp) && ip.addr != 10.0.0.1	ip.addr != 10.0.0.1 && udp ||tcp

Table 3: Exemplary query rearrangement for extended query usage

Query expressiveness As already explained in section 3.3.1 regarding the implementation, the query language does not support parentheses due to limitations in the parsing system and the language is therefore also evaluated from right to left. However, this obstacle can be overcome in many cases by clever application of De Morgan's rules[8], or rearranging the query to exploit the internal parsing logic as shown in table 3 as an exemplary case. This only becomes a potential issue if individual filters are concatenated by means of different Boolean operators. Nonetheless, if it should become necessary to use a very complex query that contains various Boolean operators, it is always an option to define the query in the source code, since all theoretically possible filter structures can actually be implemented due to the internal filter structure already mentioned. However, this requires a necessary recompilation of the source code.

A more straightforward approach to the same challenge is to split the query into several smaller queries, so that they do not conflict with the parsing system, since several queries are executed separately in time. This is most certainly not a significant drawback due to the high performance of the query module, and since the output is also machine-readable, if specified, it can be easily processed

Complexity level	Query
simple	frame.len > 0 && frame.time < Jun 15, 2021 12:00:00
complex	frame.len > 0 && frame.len <= 99999999 && frame.time < Jun 15, 2021 12:00:00 && frame.time > Jun 15, 2000 12:00:00
very complex	frame.len > 0 && frame.len <= 99999999 && frame.time < Jun 15, 2021 12:00:00 && frame.time > Jun 15, 2000 12:00:00 && port >= 0 && ip.addr != 0.0.0.0

Table 4: Query complexities used for benchmarks

subsequent to the query.

Query resolution performance On the one hand it is important to have a meaningful way to query the data, on the other hand the system must also have a high query resolution performance that does not interfere with a possible workflow of a network operator. Therefore, the total execution time of the individual processing steps within the data pipeline was measured in a similar way as was done for the writer module. However, the difference was that increasingly complex queries were used to observe the influence on the execution time. These queries, as shown in table 4, were selected as examples to approximate a realistic scenario. The filters were selected specifically so that the computational effort is as high as possible. The filter *frame.time* for example needs two values which are compared on the one hand the seconds and on the other hand the nanoseconds (table 1). Furthermore, as can be seen in the *very complex query*, a filter *port* was also used. This has a higher computational cost because both source and destination port have to be checked before a decision can be made. The same applies to the filter *ip.addr* (not to be confused with *ip.src* or *ip.dst*) as both values are checked here as well. Furthermore, the values of the individual filters were chosen in such a way that the number of packets from the WAN trace[17] is not reduced as much as possible.

Whenever the individual packet processing times are considered, it can be clearly seen in figure 15 that, if a query is used, a significant overhead is introduced for the general query application. However, even though the complexity of the various queries has been greatly increased, this is not reflected in the packet processing times; the average packet processing rate increases, but not in relation to the increase in complexity. Note that the average change from *no query*, where the packet processing time was just over 45 nanoseconds, to the *very complex query*, where the average processing time is about 47 nanoseconds, is only 2 nanoseconds. This is less than ten additional clock cycles per packet at 4.2 GHz on the tested system. These numbers can be justified by the fact that the filter application takes place multi-threaded and there are no

[17] *equinix-nyc.dirB.20180517-134900.UTC.anon.pcap*[3]

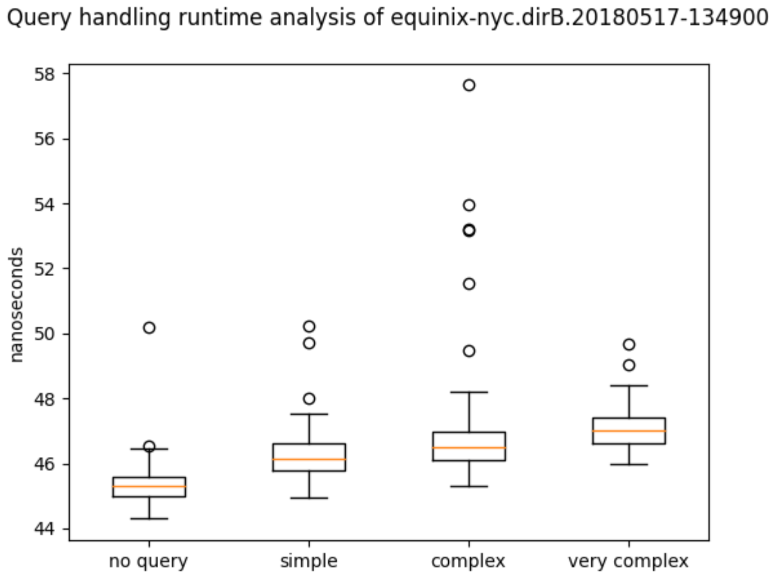

Figure 15: Packet handling time during query application

dependencies regarding parallelism as described in section 3.3.4 and thus can be scaled excellently.

However, considering the total execution time of the query module including the post-processing of the packets, which can take place in three ways as explained in section 3.3.5, the total execution time actually decreases. This is in consequence of the reason that the last step as shown in the data pipeline (Figure 6) cannot be executed by several threads at the same time. Considering the fact that with increasing query complexity the number of packets that have to be processed further tends to decrease, the total execution time decreases up to a certain degree as shown in figure 16. Finally, it should be noted that although the queries were selected in such a way that no data should actually be filtered out, the packet length was recorded as zero due to incorrect values in the test data set. However, this only affects a extremely small fraction of the total number of packets and does not take away any validity from the comparison of the query complexities. This, i.e. that the number of packets is not further reduced by the query complexity, can also be seen in figure 16 that, if a query is specified, the total execution time does not change significantly, since a large part of the packet processing time is due to the post-processing. From the observed packet processing times, a packet rate of approximately 9.5m respectively 9.4 million packets per second can be calculated, depending on whether a query has been defined or not. This means that this module is approximately twice as fast as the writer module. However, it should be noted that this packet rate only represents the number of entries actually filtered at packet level. By using the

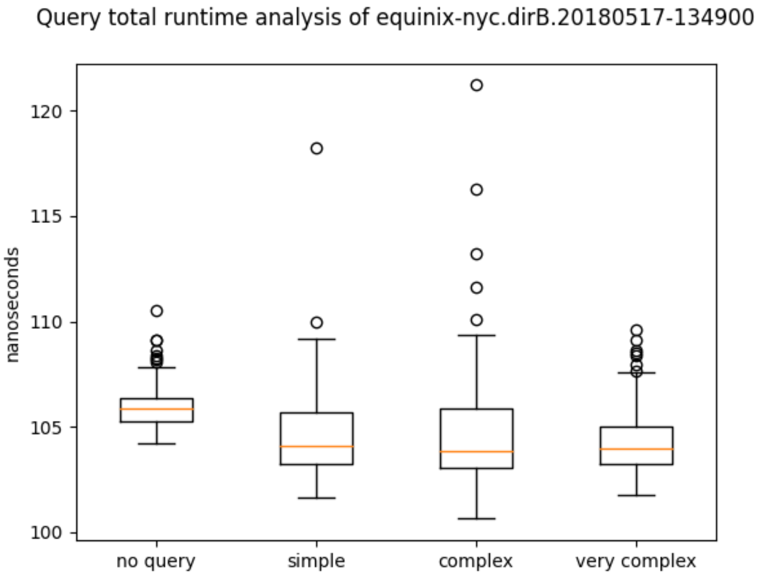

Figure 16: Total packet handling time

multi-stage filtering method described in section 3.3.2, the actual packet rate could be set much higher and these measurements are only meant to show the gross packet throughput and efficiency of the query application.

5 Conclusion

In this thesis the challenges of permanent storage of network packet entries were explored. Many aspects from different computer science disciplines were taken into account, on the one hand the constant attention to a high packet ingestion rate using parallel programming paradigms, on the other hand, more theoretical concepts were needed, for example the development a proprietary query language. This in turn required some knowledge about general software design and patterns in order to implement this efficiently.

As mentioned, there were multiple areas where trade-offs had to be made, such as the compromise between performance regarding insertion rate and efficiency in terms of compression rate.

Finally, it was possible to develop a functional prototype that tries to meet all requirements. The observed packet insertion rate is an order of magnitude higher than specially use-case optimized databases in a similar application scenario [15], while at the same time providing very high efficiency in storage usage, considering the low processing time per packet. Furthermore, it was necessary to analyze basic concepts used by classical databases in order to identify where optimizations can be made for such an application scenario. In this way, it is possible to continue to make fast queries even over very large data sets without developing a complicated indexing system. Currently, this system has been developed for a very specific application for which it has been optimized and also works well. For the future, however, there is further room for improvement in several aspects. Firstly, it has been shown that some steps within the data pipeline require very little additional computation and could possibly be combined into one processing step in order to reduce the access to synchronized structures such as the queue and thus further increase the overall throughput of the writer module. Secondly, there is still room for research on the impact on the query system if the writer module is configured in such a way that the compression rate is independent of the packet rate. Thirdly, more complex and expressive query languages can be developed and implemented to simplify the usability, as support is already realized internally.

Finally, it is important to mention that the need for such solutions has grown strongly in the recent past, as on the one hand essential areas of life are increasingly dependent on services that require cloud solutions, and on the other hand due to global phenomena such as the COVID-19 pandemic, which has also made society aware of the need for digital services. Such a development has the great potential to increase the reliability of these services as it offers new insights for the service providers.

List of Figures

List of Tables

References

[1] Bai, Y., Luo, C. R., Thakkar, H., and Zaniolo, C. Efficient support for time series queries in data stream management systems. In *Stream Data Management*. Springer, 2005, pp. 113–132.

[2] Baxter, J. H. *Wireshark essentials*. Packt Publishing Ltd, 2014.

[3] CAIDA Organization. Passive monitor: equinix-nyc. `https://www.caida.org/catalog/datasets/monitors/passive-equinix-nyc/`, Oct 2019. Accessed on 06.07.2021.

[4] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 647–651.

[5] Desrochers, C. Benchmark of major hash maps implementations. `https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.html`, Nov 2014. Accessed on 05.07.2021.

[6] Eichinger, F., Efros, P., Karnouskos, S., and Böhm, K. A time-series compression technique and its application to the smart grid. *The VLDB Journal 24*, 2 (2015), 193–218.

[7] Goetghebuer-Planchon, T. Benchmark of major hash maps implementations. `https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html`, Aug 2016. Accessed on 06.07.2021.

[8] Goodstein, R. L. *Boolean algebra*. Courier Corporation, 2007.

[9] Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., and Willinger, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication* (2018), pp. 357–371.

[10] Haerder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (Dec. 1983), 287–317.

[11] Lamping, U., and Warnicke, E. Wireshark user's guide version 3.5.0. *Interface 4*, 6 (2004), 1.

[12] Lockerman, J. Time-series compression algorithms, explained. `https://blog.timescale.com/blog/time-series-compression-algorithms-explained/`, Apr 2020. Accessed on 29.06.2021.

[13] Lockhart, T. Postgresql user's guide. *PostgreSQL Inc* (2000).

[14] Michel, O. *Packet-Level Network Telemetry and Analytics.* PhD thesis, University of Colorado at Boulder, 2019.

[15] Michel, O., Sonchack, J., Keller, E., and Smith, J. M. Piq: Persistent interactive queries for network security analytics. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization* (2019), pp. 17–22.

[16] Oracle, V. Virtualbox user manual. *Oracle Corporation.-2004.-C 357* (2020).

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

[18] Riehle, D. Composite design patterns. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1997), pp. 218–228.

[19] Shi, X., Feng, Z., Li, K., Zhou, Y., Jin, H., Jiang, Y., He, B., Ling, Z., and Li, X. Byteseries: an in-memory time series database for large-scale monitoring systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 60–73.

[20] Sonchack, J., Michel, O., Aviv, A. J., Keller, E., and Smith, J. M. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 823–835.

[21] Suel, T. *Delta Compression Techniques.* Springer International Publishing, Cham, 2018, pp. 1–8.

[22] Tan, L., Su, W., Zhang, W., Lv, J., Zhang, Z., Miao, J., Liu, X., and Li, N. In-band network telemetry: a survey. *Computer Networks 186* (2021), 107763.

[23] Viaud, A. Dominating the data center - the rise of ssd technology. `https://www.datacenterdynamics.com/en/opinions/dominating-data-center-rise-ssd-technology/`, Jul 2019. Accessed on 08.07.2021.