Vrije Universiteit Amsterdam Universiteit van Amsterdam

Master Thesis

FrogFishDB - A Timeseries Database for in-order timeseries using the TimeTree datastructure on flash SSDs

Author: Niels de Waal (2706170)

1st supervisor: dr. ir. Animesh Trivedi
2nd reader: prof dr. ir. Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 29, 2023

"Here, on the edge of what we know, in contact with the ocean of the unknown, shines the mystery and beauty of the world. And it's breathtaking."
from Seven Brief Lessons on Physics*, by Carlo Rovelli*

Abstract

With the increase in large-scale IT deployments, the amount of temporal data also grows, one instance of which is timestamped monitoring data. This data needs to be stored in real-time. Existing timeseries databases do not adequately utilize the performance that modern storage hardware can provide. This hardware is capable of multiple gigabytes per second of bandwidth. However, existing benchmarks have shown that existing databases only utilize a fraction of this bandwidth. Therefore we design and build FrogFishDB. A timeseries database build from the ground up to utilize the bandwidth capabilities of modern storage hardware.

We identify and provide a solution to two bottlenecks in timeseries databases. The first bottleneck is the ingestion protocol. Existing databases use encoding schemes that are human-readable but detrimental to performance. We solve this by using a binary protocol to encode data. Furthermore, we split the control and data plane. A timeseries must first be registered with the timeseries database, which responds with a unique token. The token is then used during ingestion to identify which timeseries the data belongs to. The second bottleneck is the complexity of indexing data structures. Understanding the requirements of timeseries data have allowed us to take a B+ tree and simplify it, creating an easy-to-implement data structure capable of a high insertion rate.

Fixing these two bottlenecks results in a significant performance improvement compared to InfluxDB, QuestDB, and Clickhouse. A single-threaded version of FrogFishDB outperforms existing databases in terms of ingestion bandwidth when using a small number of timeseries, with 40% more bandwidth using eight clients compared to QuestDB, which has the next highest ingestion bandwidth.

We also present various further optimization and feature avenues for FrogFishDB. The source code for FrogFishDB is available at `https://github.com/NielsdeWaal/Thesis` and `https://github.com/NielsdeWaal/TimeTree`.

Contents

List of Figures

List of Tables

LIST OF TABLES

1 Introduction

Our world is more and more interconnected. We generate vast amounts of data, not just by interacting with each other online through social media and messaging platforms, but also by measuring the world around us. The number of connected devices, otherwise known as the *Internet of Things* (IoT), is estimated to grow to 125 billion devices in 2030 (1), all of which generate data. Combined, it is estimated that the amount of data generated, processed, and stored by the human race will reach 200 Zettabytes (2), 30% of which is consumed in real-time. This growth has caused a large investment in our digital infrastructure. The recently published CompySys Manifesto for the Netherlands (3) has shown that the *Information and Communications Technology* (ICT) industry is responsible for 3.3 million jobs, and contributes to 60% of the total GDP. One class of the data generated by the IoT is called *Timeseries Data* (4), which are measurements taken repeatedly over time. Examples of timeseries data are stock prices, temperature readings, and brain signals.

An example that shows the scale of timeseries data generated and the importance of being able to process it in real-time is Formula 1. Formula 1 is a sport where teams attempt to build the fastest car possible and use it to compete on race tracks all over the world. These cars carry over 300 sensors (5), 30 times more than a home weather station. The sensors in the car are used to measure and monitor many aspects of the car. Three components are responsible for a tightly coupled chain of monitoring all systems with the *engine control unit* (ECU) at the center and responsible for processing and communication back to the team. Other components are the *power control module* (PCM) and the *master control unit* (MCU) (6).

As a Formula 1 car carries over 300 sensors, it generates roughly 1.1 million data points per second (5). With 20 cars on the circuit, this means 160TB of data per race, or in other words, 34 thousand DVDs. Sensors can be categorized into three distinct categories, instrumentation sensors which measure air pressure and fuel flow, monitoring sensors that keep track of the health of components such as tire degradation, and control sensors that measure driver inputs such as acceleration (6). As a result, these cars require unique databases to process and store this large amount of data. Figure 1.1 shows a Formula 1 car where the data is sent, in real-time, to a database and used by engineers who monitor the data.

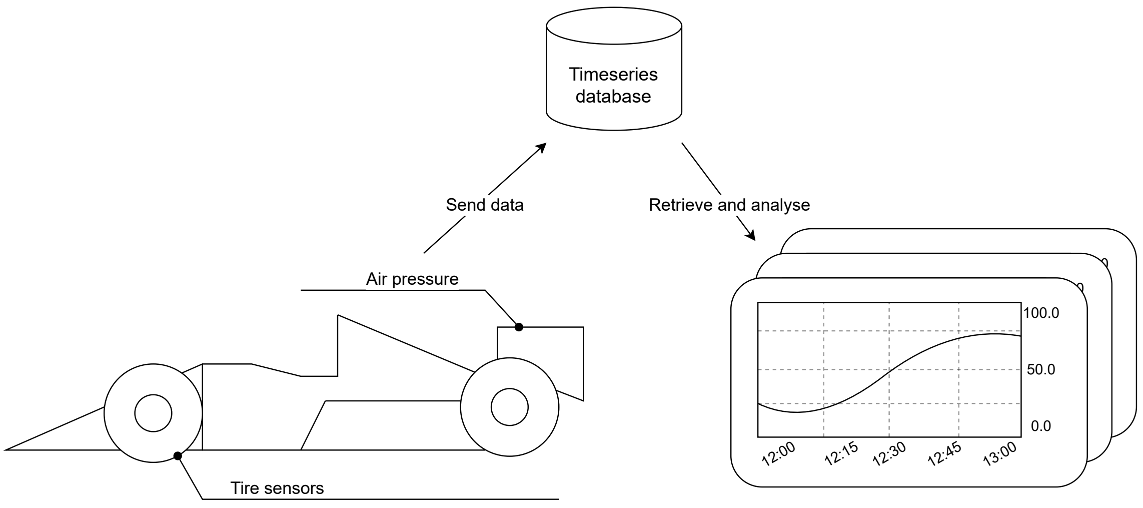


Figure 1.1: A Formula 1 car where sensor data is sent to engineers monitoring the data.

A new target has emerged in recent years, which generates timeseries data that we need to measure and monitor, namely the IT infrastructure. In 2020, Google detailed its database for monitoring its global IT infrastructure named Monarch (7). Monarch is responsible for detecting and alerting engineers when monitored services or machines fail, displaying health and status information to the engineers, and providing tools to investigate possible performance or resource usage issues. These three tasks must be handled in real-time. Otherwise, users will encounter disrupted services, such as the inability to read their email. In their evaluation, Google shows that in 2019 Monarch was responsible for storing 750TB of monitoring timeseries data and ingesting data at a rate of 2.5TB/s, which is equivalent to 50 blue-ray disks every second. Another example of an extensive IT infrastructure is AWS, which maintains an estimated 5 million servers (8). All of these must be monitored by maintenance personnel for Amazon to uphold its guarantees regarding uptime.

1.1 Timeseries databases

To ingest a large amount of timeseries data in real-time, we require a specialized database that can support the high volume of data being inserted. *Timeseries Databases* (TSDB) are a type of database that is specialized for storing timeseries data (9). Timeseries databases make use of three unique properties of timeseries data. The first property is that all data is tied to a timestamp and indexed by time. The second property is that timeseries data is write-heavy, meaning that data is more often written, than it is read. The data is also immutable, after it has been ingested it is not altered. The third and final property is that queries for timeseries data are biased towards `scan` operations, where the user is not searching for a single value but examining a range of time. These users are either interested in the raw values or in higher-level aggregated results.

While ingesting the data, databases need data structures that can be used to find data again after it has been stored. This *indexing structure* is one of the core components of a database, which means that it needs to be capable of ingesting data in real-time. The indexing structure can be seen as the index for a book. The book contains all the information, but if one wants to find some information without reading the entire book, one can use the index to find the page on which the information is located quickly.

After ingesting the data, timeseries databases provide users with tools to retrieve and analyze the data. Through specialized query programming languages, users cannot just view the data stored but can also analyze the data by applying mathematical operations. An example of such an operation would be that users can view the maximum value stored at specific time intervals.

Figure 1.2 is meant to provide an overview showing a timeseries database's core components. In blue, we present the ingestion process. The timeseries data is funneled through the index (highlighted in red) to physical storage (highlighted in green). Finally, the user can query the data and apply any optional processing (highlighted in yellow).

To better understand these different components, we fit them onto our earlier example of a Formula 1 car. The car generates timeseries data by taking measurements using the onboard sensors and sends it to the database that ingests it (highlighted in blue). The timeseries data is then stored (highlighted in green) and is monitored and analyzed by the engineers (highlighted in yellow)

To improve the performance of timeseries databases, both in terms of ingestion performance and in terms of query performance, we require a storage medium capable of handling sequential writes and random reads. In the next section we discuss flash based

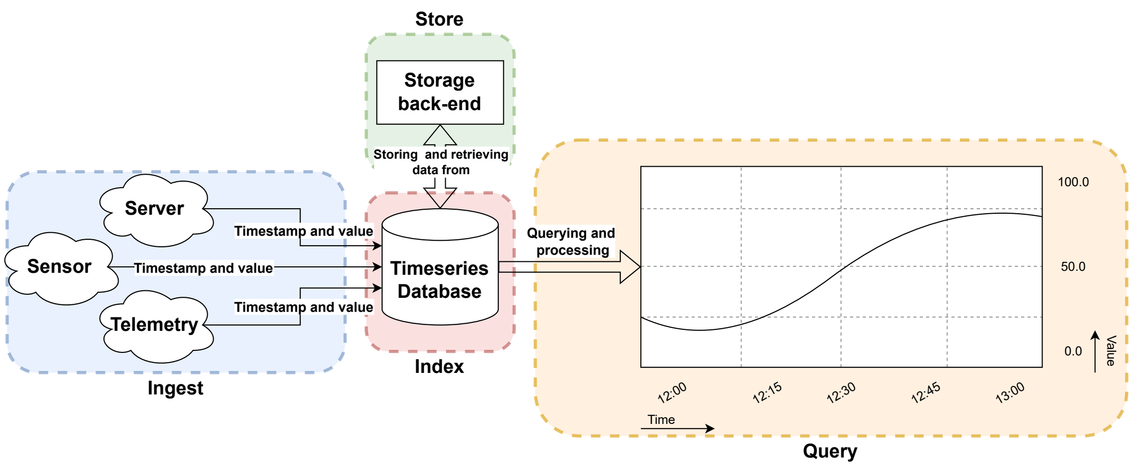


Figure 1.2: Overview showing the ingestion, indexing, storage, and querying stages of a timeseries database.

storage technologies. These storage devices are capable of high read and write bandwidth, over 350 times more than traditional storage media such as HDDs (10).

1.2 Flash storage

A storage technology that is gaining adoption in the data center is the *Solid State Drive* (SSD), projected to represent 30% of the created storage media from 2017-2025 (2). SSDs are an alternative storage medium in the data center compared to the *Hard Disk Drive* (HDD). HDDs have been widely adopted for their low cost and high storage capacity, making them an appealing storage medium for storing large amounts of data. However, due to physical limitations, the performance of the HDD is limited, relying on mechanical movement to read or write data. Contrary to an HDD, an SSD is capable of providing μ-second level access latency, enabling several gigabytes per second of read and write bandwidth, amounting to a 350 to 3,800$\times$ bandwidth performance improvement over the HDD (10).

Using SSDs does not come without challenges. The first challenge is the lack of in-place updates at the flash chip level, requiring an explicit erasure of the existing data before being able to overwrite it. Due to the physical properties of SSDs, overwriting data can incur a heavy latency penalty (11) and unpredictable performance (12, 13). The second challenge is that performance is asymmetrical, where reads are faster than writes. The third challenge is that the physical medium SSDs use suffers wear from repeated erasure.

This means that the whole section could become unreliable or unavailable if the workload is not even spread across the device.

These challenges are hidden or alleviated through special firmware running on the SSD. The *Flash Translation Layer* (FTL) (14) handles operations submitted by the host through placement policies, wear leveling, and space reclamation. Thus, SSDs show the same functional behavior as HDDs but have very different physical properties and different performance characteristics.

To unlock the potential of SSDs, people have started optimizing their workloads to match the characteristics of SSDs (15). Multiple works have shown significant performance gains (16, 17, 18, 19, 20, 21), such as increasing throughput by 41%, and decreasing latency 59% (22) compared to a database not optimized for SSDs.

1.3 Problem statement

We note a lack of research into timeseries databases, specifically a lack of research into optimizing for use with SSDs, and a lack of research into alternative ingestion methods. Existing performance evaluations have shown timeseries databases reaching a maximum of 30MB/s of ingestion bandwidth (23, 24, 25, 26), which is only 1% of the bandwidth SSDs have been shown to be capable of (10). To reach higher ingestion bandwidth, existing databases have turned towards scaling across multiple machines (27). Scaling through the use of multiple machines is an inefficient method of scaling, which we try to avoid (28). We note that the workload for timeseries databases is a good match for flash storage. The ingestion is append-only and the queries result in multiple random reads which can all be issued in parallel.

We specificaly identify and the following problems in this work:

1. Existing databases use “human-readable” protocols for ingesting timeseries data. While such protocols are easy to read for humans, they do suffer a performance penalty (29, 30).

2. The data structures used by existing timeseries databases to index stored data are not optimized for SSDs. For example, B-Trees, during updates, generate small writes, which are detrimental to the performance of SSDs, as discussed in section 1.2.

Tackling these problems enables us to design and implement a database that can better utilize a single machine’s performance and close the gap between the performance of SSDs and the measured performance of timeseries databases. Closing the gap will lead to a more sustainable future for storage systems and enables better scaling for future storage demands.

In an attempt to tackle these problems, we present FrogFishDB. This timeseries database is built from the ground up to demonstrate the effectiveness of using a different protocol for ingesting data and to demonstrate an indexing data structure designed to take advantage of the properties of SSDs.

1.4 Research questions

While there have been works in the past focused on timeseries databases, we consider this area not to be fully explored. In section 3 we examine existing work and note that there is a distinct lack of timeseries databases which are optimised for flash storage. Thus giving us the opportunity to provide new contributions. While we concentrate on ingestion performance, we consider query performance an important factor. Query performance dictates the practical usability of a timeseries database. Previous works have demonstrated the need for low-latency queries as a mechanism for monitoring system reliability (7, 27, 31, 32). So we have devised the following main research question: *How to build a timeseries database which optimizes for flash storage and provides high ingestion performance?*.

To answer this research question, we have designed the following sub-questions:

RQ1 : **How to design an ingestion protocol that trades readability away for performance?** Ingestion protocols such as the Influx line protocol (33) are user-friendly because they are human-readable. The downside of being human-readable is that it requires extra computational effort during ingestion to convert the human-readable data into machine-readable data, which can be stored and processed more efficiently. The overhead of the extra computational effort is becoming more noticeable due to the increase in storage performance while the CPU performance has stagnated (34, 35, 36). We investigate the overhead and present an alternative ingestion protocol, trading away human readability for performance.

RQ2 : **How to design a flash-friendly indexing structure that is specialized for storing and looking up time-indexed data?** Databases need an indexing structure to retrieve data after it has been written to storage. We design a simple datastructure allowing for fast sequential, write-once, in-order insertion and with random lookups. We optimize these patterns for flash SSDs.

RQ3 : **What is the quantitative impact of designing a new ingestion protocol and a specialized indexing structure?** This research question aims to identify the impact of the new ingestion method and indexing structure. There is limited academic literature on the performance of other timeseries databases uniquely for flash SSDs, meaning we have no data to fall back on for comparison.

1.5 Research Methodology

This thesis is a systems project in which we design and create a new timeseries database. We start by exploring a new ingestion method (**RQ1**), followed by the design of a new indexing structure (**RQ2**), and it is evaluated for performance (**RQ3**). We have used the following research methodologies to structure our research and implementation.

RM1 : For RQ1 and RQ2 we employ the **Design, Abstraction, and Prototyping** methodology (37). For this methodology, we construct a set of requirements to guide the design and implementation of the ingestion protocol and the indexing structure. Using these requirements, we use a back-and-forth approach where ideas are abstracted, designed, and then prototypes. The prototypes were evaluated on performance and, where necessary, were brought back to the abstraction phase, thus forming an iterative process.

RM2 : In order to answer RQ3 we need to utilize both micro and macro benchmarks. This is done through the **Experimental research, designing appropriate micro and workload-level benchmarks, quantifying a running system prototype** methodology (38, 39, 40). Using the benchmarks, we create a quantitative insight into the performance of existing timeseries databases and our design.

All provided research and benchmarking is open and available to the reader, ranging from the source code of the database to the benchmarking tools.

1.6 Contributions

This work contains the following contributions:

- **Database** In this work, we present the design of FrogFishDB, a new timeseries database which is optimized for flash storage. The timeseries database achieves higher ingestion bandwidth than existing state-of-the-practice timeseries databases.

- **Detailed evaluation** We compare the performance of FrogFishDB against existing state-of-the-practice timeseries databases, such as InfluxDB and QuestDB, using flash based storage.
- **Ingestion model** To achieve high ingestion bandwidth, we devise a new and different ingestion strategy compared to existing timeseries databases.
- **TimeTree** A new data structure that allows for range queries based on time.
- **Query model** This work provides a new approach to query languages.
- **Open source** All code and data for this work is open and available. Available at `https://github.com/NielsdeWaal/Thesis`.

1.7 Societal relevance

This thesis aims to design and build a database that can handle higher volumes of timeseries data compared to existing solutions. The goal is to create software that can better accommodate the growth of IT deployments. The CompSys Manifesto (3) shows the future challenge where we need to leverage existing storage systems better. The increasing size of IT deployments shows the fundamental role these deployments are playing in society. Systems running services such as electronic payments underpin a large section of the e-commerce economy. Reports show that electronics payments contributed an additional $296 billion to consumption between 2011 and 2015, which converts to an increase in the global GDP of 0.1% (41). These systems being down would thus result in significant economic damage.

Timeseries databases form a layer in the monitoring systems for these services. As systems underpinning services such as electronic payments keep growing. We require that the monitoring systems grow with them. Better utilizing existing hardware enables keeping up with the growth and reduces energy consumption as less hardware is required.

1.8 Thesis Structure

This thesis is split into multiple chapters. In Chapter 2, we establish the required background knowledge required for the remainder of the thesis. Next, in Chapter 3, we delve into the technical details of existing timeseries databases. We discuss the different techniques for reading and writing data and the possible shortcomings. We detail the design of FrogFishDB in Chapter 4. Here we first define the design requirements and discuss the different design decisions taken and the potential tradeoffs. We provide an extensive evaluation in Chapter 5. We evaluate existing databases, FrogFishDB, TimeTree, and the ingestion model. Chapters 6 and 7 are used to discuss our conclusion and future work.

1.9 Plagiarism Declaration

I confirm that all material present in this report, unless explicitly stated, is the result of my own efforts. No parts of this report are copied from other sources unless credited and properly cited. The work has also not been submitted elsewhere for assessment. I understand that plagiarism is a serious issue and should be dealt with if found.

2

Background

Before we can properly discuss the design and implementation of FrogFishDB, we first discuss the background on which this work is built. We start with an introduction to flash storage, we discuss the internal structure and how different access patterns influence performance. Next, we specify timeseries data and then discuss the internal workings of timeseries databases. After discussing timeseries databases, we discuss different interfaces provided by the operating system for enabling asynchronous IO, with a detailed examination of the `io_uring` interface. After discussing IO interfaces we discuss the internals of the B and B+ trees. Finally, we examine different aspects of the data model used in FrogFishDB. We start by giving an overview of the model, followed by a discussion of tags and out-of-order data. Finally, we discuss queries and introduce the inverted index structure.

2.1 Flash Storage

The first NAND flash was introduced by Toshiba in 1988 as a non-volatile storage technology (42). A flash cell stores information by holding an electrical charge. The NAND flash cells introduced by Toshiba could only store a single bit and are thus referred to as *Single-Level Cell* (SLC). Subsequently, different cell types have been brought to market, ranging from *Multi-Level cell* (MLC), meaning 2 bits per cell, to *Hepta-Level cell* (HLC), which stores 7 bits per cell (43). Data is written by *programming* and cleared by *erasing* the cell.

A NAND package can be created by combining two or more NAND flash memory chips (dies) into the same integrated circuit (IC) package. The dies can operate independently and accept different commands concurrently, e.g., one die can be erased while another is programmed. The dies are composed of different *planes*, each of which is composed of multiple *blocks*. A block is the smallest erasable unit in flash storage, while pages are the smallest programmable unit. Note here that we can write in smaller units than we can erase. Pages also contain extra space, called the *Out-of-Band* (OOB) area, which has error correction data and metadata. The manufacturer usually does not specify the size of a page or block. However, typical value ranges are 2-16 KB and 128-512 KB, respectively.

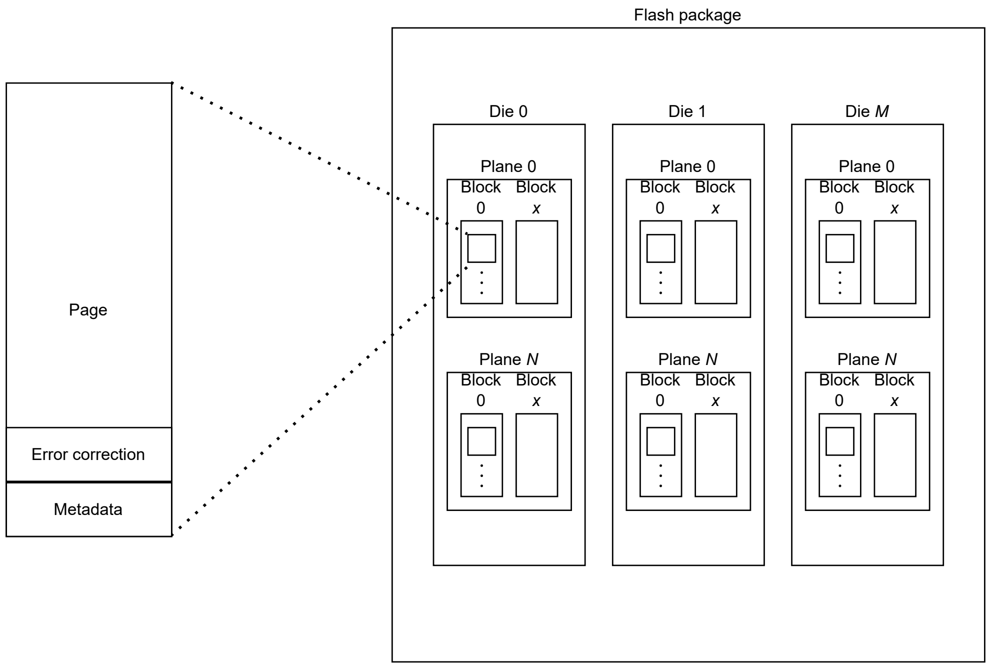


Figure 2.1: Overview of a IC package.

Continuously programming flash cells and then erasing them in so-called *program/erase* (PE) operations exerts wear on the flash cells (44, 45), causing them to wear out and gradually lose the capability to hold the electrical charge. The rate lifetime of a cell depends on the number of levels, where SLC flash has a higher lifespan than MLC, which in turn has a higher lifespan than HLC. The error correction space in the OOB area of a page is thus used to provide higher reliability.

With multiple flash packages, we can build a device that we can use to store data without any physically moving parts. Such a *Solid State Drive* is built from a few, up to tens of the IC packages. The packages are connected to a flash controller through a *channel*, a communication bus over which the controller issues commands to the packages. Increasing the number of channels increases the bandwidth available to issue commands to the packages as the number of commands which can be issued concurrently increases.

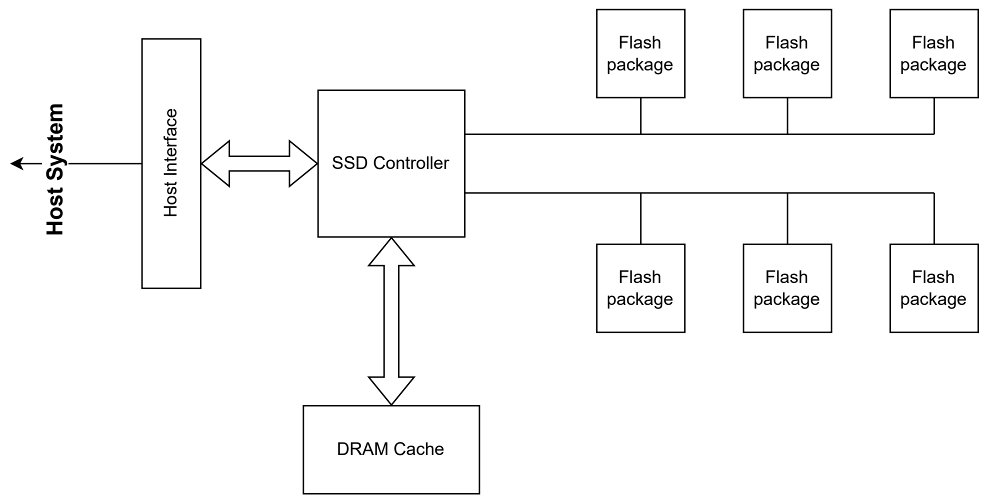


Figure 2.2: Internal architecture of an SSD.

In figure 2.2, we show an overview of the internals of an SSD (46). We show six flash packages connected to the controller using two channels at the bottom. In the middle, we show the controller connected to the host system using the host interface, together with a *DRAM* cache. The DRAM stores metadata such as physical-to-virtual address space mapping and temporary user data. Sometimes, the controller can use the DRAM cache to buffer consecutive writes to the same flash cell. This allows the controller to increase throughput by saving bandwidth and increase the flash cell's lifetime by only issuing a single PE cycle containing the result of the writes (47).

For the connection to the host, there exist multiple different connection interfaces, such as *Serial Advanced Technology Attachment* (SATA) and *PCI Express* (PCIe) bus. These interfaces carry protocols such as *Advanced Host Controller Interface* (AHCI) and *Non-Volatile Memory Express* (NVMe) (48). The NVMe standard has been created specifically for accommodating fast SSDs. The NVMe interface also uses an increased number of I/O queues, allowing for a higher degree of parallelism in terms of the amount of outstanding I/O requests. All these changes amount to an 350 to 3800$\times$ performance improvement compared to SATA (10).

In section 1.2, we discussed the existence of the *Flash Translation Layer* (FTL). The FTL is a part of the firmware running on the SSD, its main responsibilities are mapping virtual to physical addresses, bad block management, garbage collection, and wear leveling. The FTL has to manage writes that overwrite existing data due to the lack of in-place updates for flash storage. When data is supposed to be overwritten the FTL can bypass the required erasure by writing the new data to a different block and marking the old block invalid.

If no free blocks are available, the FTL has to initiate the process of *Garbage Collection* (GC).

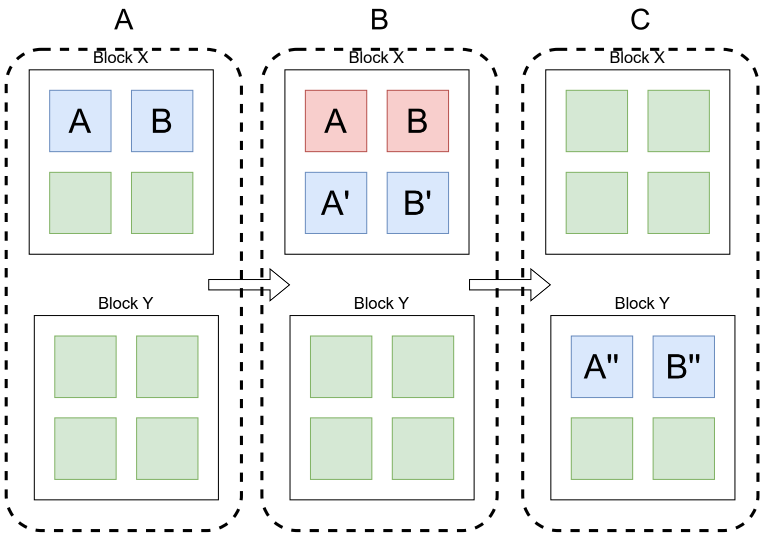


Figure 2.3: Simplified GC process. With the green blocks being free, the blue blocks containing valid data, and the red blocks having been overwritten.

Figure 2.3 shows a conceptual overview of garbage collection. 2 blocks are shown, each containing two pages. In situation *A* (on the left), two pages of block *X* have been written to, now containing `A` and `B`. Next, in *B*, these pages have been rewritten into `A'` and `B'`. The pages which used to contain `A` and `B` have been marked as invalid. Finally, in situation *C*, the garbage collection process has been run. Because the SSD can only erase a whole block and not a page, `A'` and `A'` have been copied to block *Y*. After copying the data, block *X* can be erased. Finally, the pages in block *X* can be reclaimed.

In summary, flash storage provides a fast storage medium. However, in order to make proper use of the capabilities of flash storage, we need to take some of the underlying properties into account. Inefficient access patterns can cause performance degradation. Rewriting a single page especially causes performance degradation due to the garbage collection process. Another performance improvement is aligning our access pattern with the underlying flash construction. This means that writes are aligned to the size of a page. Appending data to files in blocks that are sized equally to the size of a page would lessen the need for the garbage collection process to cause added latency and overhead.

2.2 Timeseries data

Before we can start describing the results of this thesis, we must first introduce the concept of timeseries data. Timeseries data can be defined through three key aspects (49, 50), the first is that timeseries data is indexed by its corresponding timestamps, the second is that data is only appended and not removed, and the third is that the recording workload is write-intensive.

Going over all three aspects. Timeseries data is always tied to an accompanying timestamp. Imagine a server recording its CPU temperature, and data is recorded every 10 seconds. Every time a new temperature value is measured, it is sent to the database with a timestamp indicating when this measurement was taken. This timestamp now indexes the recorded temperature. The second aspect means that the data is only ever growing in size. New data is appended to previously recorded data and should not overwrite existing data. The third and final aspect is that timeseries applications are write-intensive. The writes will dominate the issued operations, whereas reads will happen sporadically in comparison.

For this work, we distinguish between two different types of timeseries data. Namely, IoT and DevOps data. The differentiating factor here is that IoT data may arrive out-of-order, whereas DevOps data does not have this limitation. For DevOps data, we assume a datacenter setting with correctly functioning networking equipment. IoT data is the result of sensors deployed in the field. These might be located far away from a stable network, so much so that data might be out-of-order or missing entirely. The problem with timeseries data which arrives out-of-order is that it requires care to handle. If data always arrives in order, the database can take this into account, it can append data to storage. If the timeseries data can arrive out-of-order the possibility exists that timeseries data which has been stored, will have to be read and altered. Previous work has shown that handling out-of-order data can impose a significant performance penalty (31). There are two ways of dealing with out-of-order data. The first is to drop any timeseries data which arrives out-of-order. The second, is to employ a *read-modify-write* approach. In this approach, timeseries data which has been written is read again, merged with the newly arrived data, and then written back to storage. This approach imposes a large overhead since it potentially involves a large number of random reads and writes.

2.3 Timeseries databases

This section will delve deeper into the core components used to build a timeseries database. We start by examining the ingestion protocol and different possible implementations of it. Next, we will explore the concept of an indexing structure. Finally, we will discuss the methods used for scaling timeseries databases.

2.3.1 Ingestion protocols: Pull vs Push

To insert data into the timeseries database, an ingestion protocol is needed. Ingestion protocols can be categorized into two types, namely *pull-based* and *push-based* ingestion. The first type is where the database will pull data from processes for which it is configured to record data on. The second type is where processes send the data to the database, equivalent to an SQL insert for a traditional relational database.

Pull-based ingestion is used by databases such as *Prometheus* (51) and VictoriaMetrics (52). Prometheus consists of several distinct components, of which the component responsible for ingestion is called the *Scrape Manager*. This service retrieves timeseries data from other processes. This retrieval is done through `HTTP GET` requests to the other process, which exposes an HTTP endpoint, for example, at: `http://service:6900/metrics`. Upon receiving the request, the HTTP endpoint responds with a list containing key-value pairs of the metric names and values.

Push-based ingestion is used by databases such as InfluxDB or QuestDB (53). This protocol can be equated to an `SQL INSERT` command used by traditional database management systems such as SQLite (54). The protocol can be conceptualized as the application or process sending the data to the timeseries database. InfluxDB uses the *Influx Line* protocol to encode the information to be ingested (33). This format is supported by multiple timeseries databases (52, 53, 55), but is not standardized.

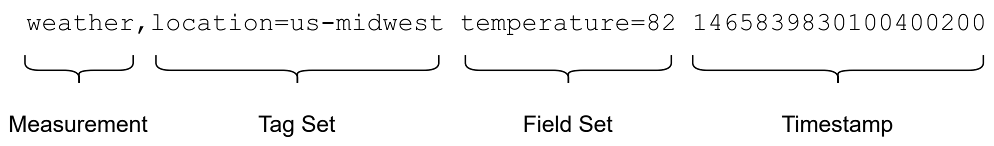

weather,location=us-midwest temperature=82 1465839830100400200
Measurement
Tag Set
Field Set
Timestamp

Figure 2.4: Influx line protocol example.

The Influx Line protocol consists of 4 parts. Figure 2.4 shows an example of an insertion. The `Measurement` field is the common name for a timeseries. This is the equivalence of the name of a table in a relational database. The `Tag Set` is a set of metadata that can be used during queries as a field to filter on. Next is the `Field Set`, which contains the recorded values. This set is the same as the columns of a relational database table. The final value in the line is the `Timestamp` at the measurement that was taken. The user sending the data using this protocol is free to choose the contents of all fields; tables do not have to be explicitly created.

It is important to note that while this protocol is human-readable and flexible, it also suffers from the same performance issues that other encoding protocols suffer from (29, 30). The problem is that the data has to be converted from its human-readable form to a binary format to be used by the database (56). For example, figure 2.4 shows the value 82 to be ingested. The value has to be converted by taking each digit of the value, transforming the ASCII code to the corresponding integer value, and scaling each to the correct order of magnitude. Inserting the value directly would be computationally cheaper, skipping the conversion step. The overhead is exacerbate by the difference in performance between HDDs and flash storage. In section 2.1 we discussed the difference in bandwidth between NVMe flash storage and HDDs. For example, imagine a timeseries database is able to process a human-readable at 100MB/s using a single thread. With HDDs, two thread would fully saturate the write bandwidth available. However, with flash storage it would require tens of threads to fully saturate the bandwidth of the storage device. If skipping the conversion step allows the database to process the ingestion protocol at 1GB/s, then again only a few threads would suffice to saturate the bandwidth of flash storage and thus require less CPU resources, increasing both efficiency and decreasing power requirements.

2.3.2 Indexing structure

After writing data to storage, we need an index in order to be able to find the data again. A good analogy would be to consider the storage medium, a file or a flash based storage device, as a book and the indexing structure as the index at the back of the book. This problem is not just found in database systems but in many more storage applications; one example is the FTL discussed in section 2.1. For a database, the index allows us to efficiently retrieve records based on the column on which the indexing has been done.

One of the most time-consuming processes in databases is searching. Using a suitable search method over a bad one often leads to a substantial increase in speed (57). Databases have commonly used tree structures as an index. Tree-structured indexes have been used for a wide range of applications, such as data mining, financial analysis, and scientific computation (58).

While binary search is the theoretical optimal algorithm for searching a sorted array, it is not optimal in reality. The optimality proof assumes that the number of comparisons is the only metric by which to determine performance. In reality, the latency of random storage accesses overshadows the cost of comparing keys (59, 60). Indexing structures like the B+ tree reduce the number of expensive random accesses to find a key inside the index (61). B+ trees are used across different storage workloads, such as databases (54) and filesystems (62, 63). Section 2.5 will go into more detail about constructing a B+ tree.

2.3.3 Traditional relational databases

While timeseries databases attempt to provide high ingestion bandwidth, there have also been attempts to use traditional relational databases. One such attempt is where a MySQL (64) instance was able to provide an ingestion bandwidth of 332'000 values per second (65), or, assuming 16 bytes per value, 5MB/s. This has been accomplished by horizontally scaling the database across three AWS EC2 nodes. However, the setup required to achieve this came with several drawbacks (66). The first drawback is that data has to be stored in vectors, meaning that one row contains data for more than 1 point in time. The vectors are used to batch insertions, however, they limit flexibility during queries as each row now contains multiple datapoints. The second drawback is the use of clustered primary keys, which inhibit ad-hoc SQL queries. Thus a timeseries service has to be used for querying. The third and final drawback is the complexity and knowledge required for designing the cluster of nodes. Before building the cluster, it has to be manually implemented how data is going to be sharded (i.e., how data is going to be partitioned over the various nodes in the cluster), grouped, and how indexes are going to be created. This shows that, with effort, it is possible to scale to high ingest rates using traditional relational databases.

While this approach for scaling is viable in terms of the possible ingestion performance, there is also one significant disadvantage, namely that this is accomplished through horizontal scaling. Two different scaling techniques are found in database scaling, horizontal and vertical (67). Figure 2.5 shows a conceptual overview of the two techniques. Horizontal scaling means that a workload is spread across multiple machines, while vertical scaling means that as much of the hardware performance is used. We view horizontal scaling as a “brute force” method, where more hardware is thrown at the problem, even though an optimized application running on a single node can perform similarly, be it in terms of ingestion bandwidth or query performance. The costs of scaling either horizontally or vertically have been discussed before in works such as (28).

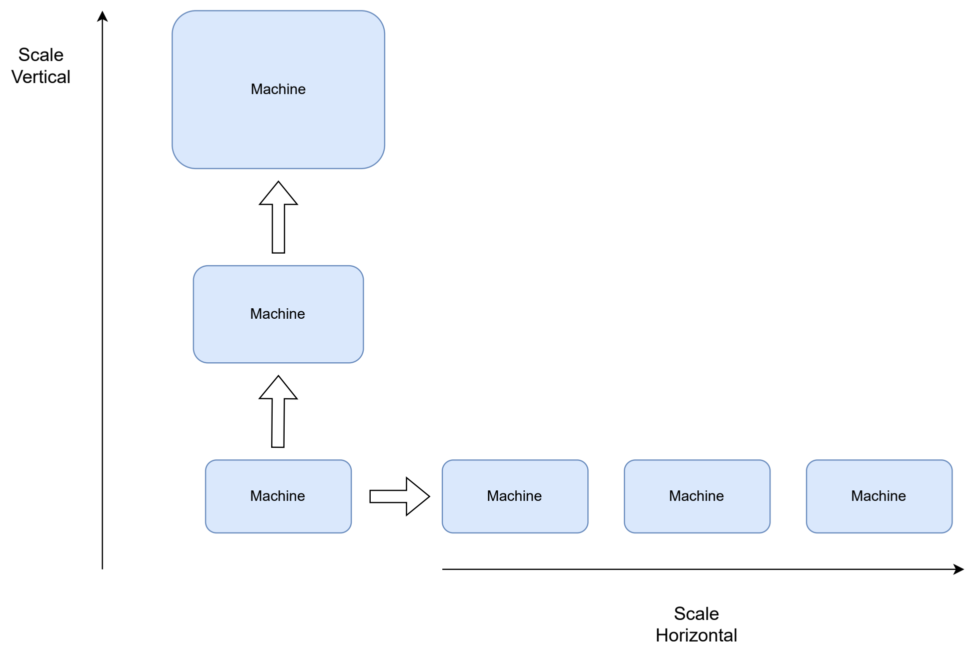


Figure 2.5: Conceptual overview of the two different scaling techniques.

Another major downside of horizontal scaling is the comparatively high energy, monetary resources, and space usage. Previous work has shown that the energy use of a data center can be lowered by optimizing a database for a single node (60). Other factors, such as the use of monetary resources and space, can be derived from the fact that scaling across multiple machines means that machines have to be bought and require space in server racks.

2.3.4 Summary

Timeseries databases differ from traditional relational databases in that they are optimized for a specific workload, namely that of timeseries data. TSDBs ingest data either through a pull or through a push configuration, whereby the timeseries data is either retrieved from applications by the database or is sent to the database by the application. The data is encoded to ensure that the database can read and parse the data. However, the protocols to do so are often human-readable and thus suffer from a large overhead as they have to be parsed.

In order to overcome the overhead from parsing, we turn towards different scaling techniques. We classify two variations of scaling, namely horizontal and vertical scaling. Horizontal scaling means that more resources are employed to improve performance, while vertical scalability means that an application is optimized to better utilize the resources available. For FrogFishDB, we aim to use vertical scaling to better utilize the available resources of a machine and thus improve both energy and monetary efficiency.

2.4 Asynchronous IO

In any application, one aspect of improving performance is ensuring that the code only spends time on meaningful computation, not on *Input/Output* (IO) operations. IO operations are operations where we interact with the "outside" world, such as writing data to disk or receiving data from a network socket. Linux has traditionally follows the POSIX interface (68), thus system calls for these IO operations are `read` and `write`. These operations are *blocking* IO operations. Which means that they wait for the operation to be complete before continueing. This in turn means that the latency of these operations is tied to the performnace of the underlying IO hardware and not just to the performance of the CPU. The consequence of this is that, for example, a write operation can have a latency which is orders of magnitude higher than commputing the sum of a list of integers. The difference in latency is a problem because the system call is waiting for completion and the application is not doing any meaningful computation.

One way of hiding the latency of IO is through the use of *asynchronous* IO operations. This is where the application submits a request to the operating system, goes off to do other work, and returns when the request has been completed. Any operation dependent on the request's result has to be put on hold until the request is complete. Figure 2.6 shows a conceptual overview of the difference between blocking and asynchronous IO.

Operating systems provide functions to the application which allow the application to submit asynchronous IO operations. We can classify two distinct types of asynchronous IO: *polled* and *interrupt driven* IO. The difference between the two is that with polled IO, the application has to periodically call into the operation system to check if the IO operation has finished, whereas, with the interrupt-driven IO, the operating system will call back into the application, letting it know that the IO operation has completed. There are several considerations for choosing to use either of the two types. The first is that polling can lead to wasted CPU cycles. This is the case when the application polls repeatedly without there being any work to do due to requests which have been completed. Interrupt-driven IO does not suffer from this issue. However, interrupt-driven IO suffers from higher latency due to the operating system having to call the application.

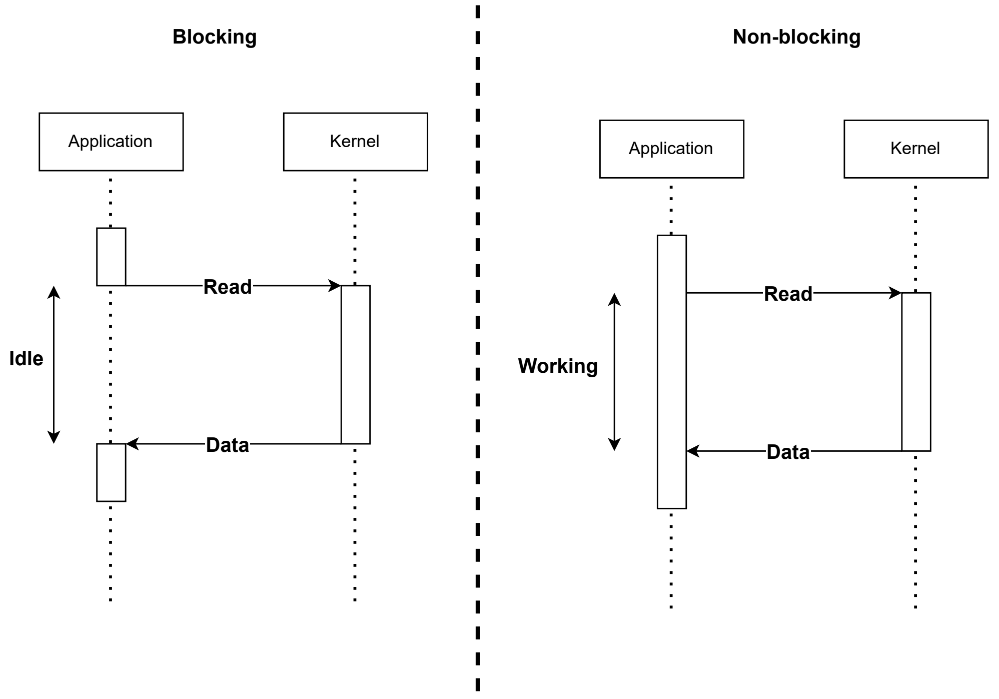


Figure 2.6: Visualisation of blocking vs. asynchronous IO operations.

2.4.1 Asynchronous IO in the Linux kernel

In the Linux kernel, there exist multiple interfaces which provide the application with asynchronous IO. System calls such as `select`, `poll`, and later `epoll` allow the application to poll a set of file descriptors to check if they are ready. The problem with the first two system calls is that they do not scale with large numbers of file descriptors. A comparison shows that in kernel version 2.6.25, when monitoring 1000 file descriptors (69, p. 1365) it takes 35 seconds to poll all 1000 using either `select` or `poll`. `epoll`, on the other hand, can poll 1000 file descriptors in 0.53 seconds. However, `epoll`'s readiness system only works on sockets and pipes, not files. Newer kernels have shown improvement however. In 2019, engineers at SUSE have shown epoll being able to poll 10,000 file descriptors in 5.5 milliseconds, where as `poll` and `select` require 80 milliseconds (70).

File IO before Linux kernel version 2.6 was handled using thread pools (71, p. 394). Threads were allocated to handle file read and write system calls, letting them block in the background. This model broke down when fast flash storage devices became more commonplace. We described in 2.1 how modern flash storage allows for IO latencies in the single-digit microsecond range. Latencies this low are on par with the latency of a context switch (72), making thread pools inefficient. On average polling provides 25% lower latency compared to interrupts (72, 73, 74).

Linux 2.6 introduced the *Asynchronous I/O* (AIO) interface. Allowing the application to submit IO requests using the `io_submit` system-call, and receive events that are ready using the `io_getevents` system-call. AIO allowed for fully asynchronous IO. However, in practice, the interface fell short of the expectations placed upon it. Linus Torvalds (the creator of the Linux kernel) famously lambasted the interface, callings its design one which was made for people "who seldom have any shred of taste" (75).

Outside of stylistic issues, there are technical ones as well. Three issues made it fall short of the expectations. The first issue is that asynchronous file IO only works with `DIRECT IO`. This is a file IO mode in which the page cache in the kernel is bypassed, and all requests go directly to the block device. The problem with `DIRECT IO` is that it requires buffers to be aligned to the block size of the underlying storage medium, thus if the storage medium uses a block size of 512 bytes and the users wishes to issue a write of only 16 bytes, then there is a 488 byte overhead. The second issue is that the asynchronous system-calls can behave in a blocking manner (76). For example, if metadata is required to perform an IO request, the submission system call will block waiting for the metadata. Another way the call can become blocking is if all the request slots are in use. There are a limited number of request slots for storage devices, and if none is available, the submission call will block until one is available. The third issue is that the API is not very efficient. Each submission requires 72 bytes to be copied, while completion events require 32 bytes for each IO request. The overhead of 104 bytes can be a significant factor for smaller-sized requests. Consider a write of 512 bytes to a file; with 104 bytes of overhead, there suddenly is an almost 20% overhead for an interface that promises to be zero-copy. These issues make it impossible for an application to assume that the libaio interface is asynchronous and forces the application to offload the requests again using a thread pool.

2.4.2 io_uring

`io_uring` (77) is a recent addition to the Linux kernel, introduced in kernel version 5.1, and provides a generic interface for issuing IO requests. The name `io_uring` hint at the fundamentals of the design. `io_uring` maps two ring structures into user space and shares them with the kernel. The first ring is the *submission* ring which is used by the application to submit requests to the kernel. The second ring is the *completion* ring, which contains the results of the previously submitted requests. The overhead of system calls when submitting requests is alleviated as the head and tail pointer of the ring structure can be atomically updated without system calls. Figure 2.7 shows a conceptual overview of the `io_uring`

ring structures. The left ring shows the submission queue, where the application submits IO requests. The ring on the right shows the completion ring, where the IO request results are posted.

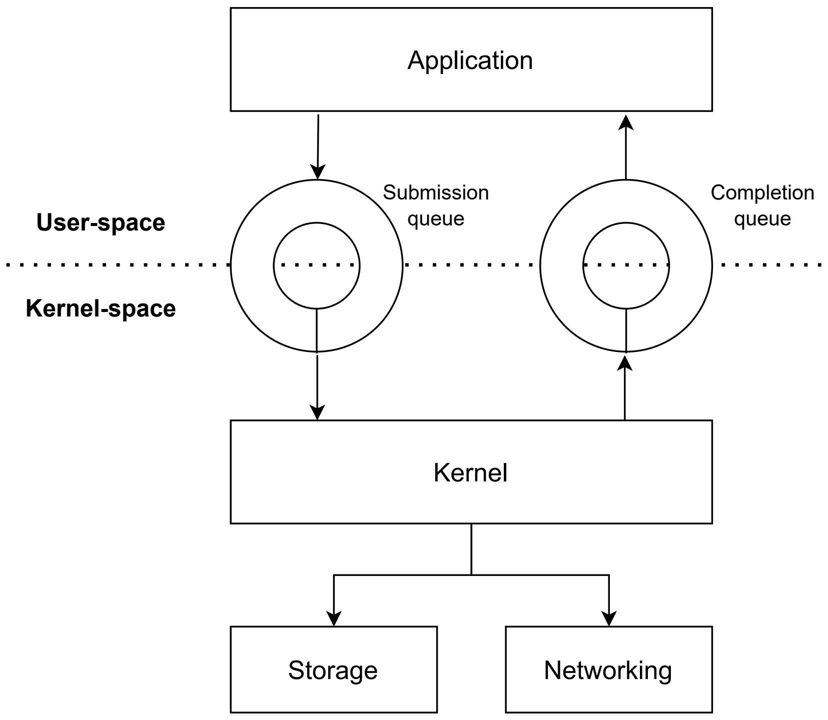


Figure 2.7: Conceptual overview of the `io_uring` ring structures.

Three key features of `io_uring` set it apart from the existing asynchronous interfaces. The first feature is that the interface is fully asynchronous by design. Submitting requests through the submission ring does not require a system call and will thus never block. The second feature is that `io_uring` supports any type of IO (77). Compared to AIO, which requires files to be opened in `O_DIRECT` mode, `io_uring` has no requirement for the kind or type of IO. The third feature is that the interface is extensible. It is written to provide an abstraction for any asynchronous IO, not just those required by database engineers.

We must examine the data structures used to understand why `io_uring` is more flexible than existing interfaces. The submission and completion queues contain a power of two number of `io_uring_sqe` and `io_uring_cqe` structures, respectively. The application retrieves a pointer to the next free `io_uring_sqe` structure when submitting an IO request. In this data structure, the application registers the type of operation, relevant flags, a pointer to user-supplied data, and the file descriptor on which the operation is supposed to be executed. While not limited to the fields mentioned earlier, these are the fundamental fields in the structure. At the time of writing, the submission request structure consists of 14 fields, of which 6 are union structures. We want to point out that the application has had to make no allocations for this request; the allocation for submitting the request is done upfront when constructing the ring structures.

The data structures used in the completion queue are comparatively simple. The `io_uring_cqe` consists of three fields. The first is the result value of the IO operation. The second is used to carry flag values. As of kernel version 6.0, the following flag values are defined (78).

- The `IORING_CQE_F_BUFFER` flag indicates that the upper 16 bits of the flag value contain the ID of the buffer chosen for this request. We will expand on the buffer system later in this work.
- The `IORING_CQE_F_MORE` flag indicates that the application should expect more completions from this request. Used in *multi-shot* requests, this flag indicates that the request is not fully fulfilled. Multi-shot requests have been introduced in kernel version 5.19, allowing an application to submit one request that can trigger multiple completion events. For example, the application can submit a multi-shot request which performs multiple `accept` calls without the application having to issue a new accept.
- The `IORING_CQE_F_SOCK_NONEMPTY` flag indicates that a receiving socket still contains more data. This flag is set after not all data from a receiving socket is read. In this case, a new completion event is generated with this flag set.
- The `IORING_CQE_F_NOTIF` flag is set for notification completion events, which are used for zero-copy networking send and receive support. The completion event's final field is the user-supplied data pointer. This field is the same as in the aforementioned IO request. This is a pointer to data that the user has constructed. The pointer value or the corresponding data is not altered by `io_uring` and is copied directly to the completion event. The data can, for example, be used by the application to determine which operation the completion belongs to or to notify other parts of the application of the completion.

When applications need the lowest possible latency, an application can opt to utilize one of the two polling modes provided by `io_uring` (77). By default, the kernel threads handling the IO requests coming from userspace rely on interrupts in order to be notified of requests which been completed before notifying the application through the completion ring. The first polling mode changes this behavior. Referred to as *completion polling*, the kernel threads are instructed to poll the IO devices for request completion. The second polling mode instructs the kernel to poll the submission ring for new submissions. By default, the application uses a system call to indicate to the kernel that new requests have been posted to the submission ring. Note that, by default, one system call invocation can be used to submit or receive multiple requests. In the second polling mode, referred to as *submission polling*. Consequently, when using submission polling, the system call invocations for both submitting requests and processing completion events can be eliminated. As the kernel polls the submission ring, it is no longer needed to inform the kernel of the newly submitted requests, while the application can poll the completion ring by simply checking the head of the queue. The overhead of system calls cannot be overstated. Previous works have shown that user-mode *instructions-per-cycle* degrades exponentially when increasing the frequency with which system-calls are used (79, 80).

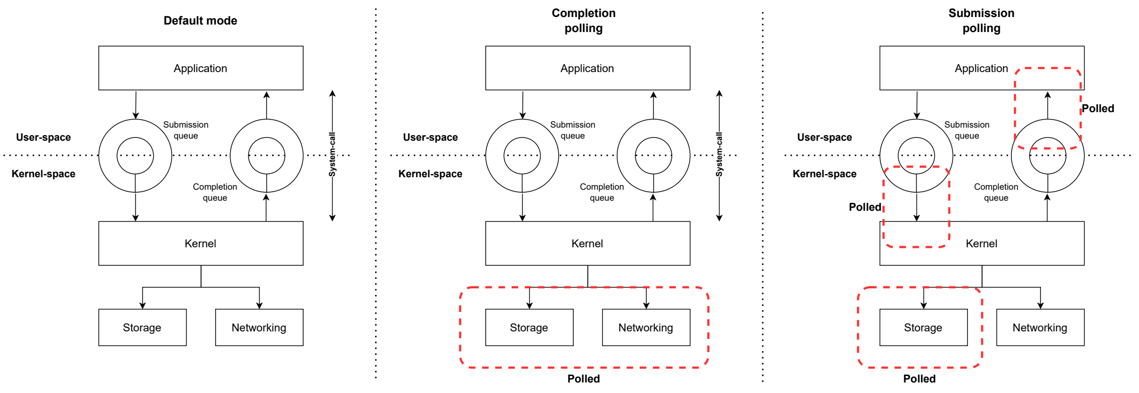


Figure 2.8: Conceptual overview of the `io_uring` polling modes.

Figure 2.8 shows a conceptual overview of the default and two polling modes. On the left, we show the default mode, where an application uses a system call to inform the kernel of newly submitted requests or to poll for completion events. In the middle, we show the use of completion polling. Note how the storage layer is now polled by the kernel, instead of relying on interrupts for when a request has been completed. On the right, we show the second polling mode, namely submission polling. In addition to the kernel polling the devices for completion, it now also polls the submission ring for newly submitted requests.

The final aspect of `io_uring` we need to discuss is the possibility of registering buffers. We alluded to earlier that a completion event can carry a flag that indicates the use of, and ID of, a buffer registered earlier by the application. Using the earlier `epoll` polling method, whenever a socket has received data, the file descriptor is considered to be in a *ready* state. This model has the advantage that a suitable buffer can be sought when the file descriptor is ready before reading the newly received data from the socket. This model breaks in a completion model. Using `io_uring`, the data is read from the socket by submitting a read request to the submission queue. The application can pick a buffer when this read request is submitted. However, this can lead to problems with scaling when handling hundreds of thousands of requests at a time.

In kernel version 5.7, the option was introduced to register a set of buffers with `io_uring`. Using these buffers, the kernel can pick a suitable buffer instead of one having to be provided by the application. The completion event will hold the necessary information about which buffer of the set has been chosen for the received event. Buffers are identified by a group ID and within that group by a buffer ID. Buffer groups allow the application to provide multiple sets of buffers. For example, multiple groups can be provided where each group contains buffers of a different size. When the application is done with a buffer, it hands control of the buffer back to the kernel such that it can be reused for a future receive event. In kernel version 5.17, the interface for providing buffers was replaced by so-called *ring mapped buffers*. While identical in functionality, ring-mapped buffers provide a more efficient method for providing buffers (81).

The performance of `io_uring` has been studied in the past. Research has shown that using `io_uring` provides the best performance for storage devices (both in terms of latency and in terms of throughput) out of the different interfaces provided (82, 83, 84, 85). The recorded performance demonstrated that `io_uring` should be used for performance-sensitive code dealing with IO in FrogFishDB.

2.4.3 FrogFishDB and io_uring

For FrogFishDB, we are interested in several of the aforementioned features and capabilities. The first is the overall efficient model for submitting and handling IO requests. The workload for timeseries databases is, in essence, dominated by the overhead of IO. Data is read from the network socket and written to storage. Ingesting timeseries data does not require transforming the data itself. If the overhead described in section 2.3.1 can be alleviated, then the workload is reduced to only the process of reading and storing data. The second is the option of several polling modes. Previous work has shown that using polling modes, the performance of `io_uring` can approach that of the underlying device (82). While in this thesis we were not able to properly investigate and test all of the possible features, such as ring-mapped buffers, we are still confident that these features allow for better performance of FrogFishDB, i.e. higher ingestion bandwidth and lower query latency.

We did manage to use `io_uring` for both networking and storage. For storage we use asychronous file IO and networking we use standard socket interface managed through `io_uring`. We also make use of the multishot accept feature, however, due to kernel incompatability, we do not use this feature for the performance evaluation in chapter 5.

2.5 B and B+ Trees

The B tree (86) is a self-balancing tree data structure that maintains sorted data. Operations such as searches, sequential accesses, insertions, and deletions happen in logarithmic time. The main feature distinguishing the B tree from a traditional binary tree is the fact that the B tree can store more than two children per node.

According to the formal description, the following properties will always hold:

- Every node has at most m children
- Every internal node has at least two children
- Every internal node has at least $\lceil m/2 \rceil$ children
- All leafs are on the same level
- An internal node with k children contains k - 1 keys

With internal nodes, we refer to nodes that are neither leaf nodes nor the root node. Such an internal node contains a maximum of m children and a minimum of L children, where $L = 2$. The rule is that m must be either *2L* or *2L-1*, or in other words, there is always 1 less element than the number of child pointers. The maximum number of children for a node is referred to as the *order* of the tree. Because of this number of children rule, internal nodes are always at least half full. A full node can be split into two nodes on the same level when there is room to push one element up to the parent level. The root node differs from internal nodes as, while the upper limit still holds, there is no minimum limit.

We provide a visual overview of a B tree to understand the data structure better. Figure 2.9 shows an example B tree of order 4. To search for the value 18, we only need to follow two pointers instead of the possible five if we had used a binary tree.

Now let us look at a B+ tree (61). The main difference is that all data is stored in the leafs, so the internal nodes only contain the first keys of the child nodes. The main

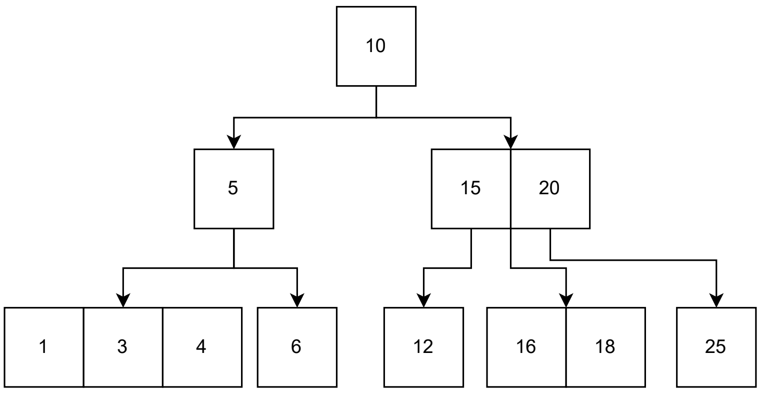


Figure 2.9: Example of a B tree.

advantage of this difference is that this enables us to reduce the memory usage of internal nodes and thus store more nodes per memory page for fast lookups. Another key difference is that the leaf nodes are linked, allowing us to do scan operations over the key space. An example where this is very useful is in timeseries databases. Consider a situation where a B+ tree holds keys that denote a range of time. If we then want to do range queries where we take a starting timestamp and an ending timestamp, we would only have to search for the start and can find the end by way of a linear scan using the links between leaf nodes.

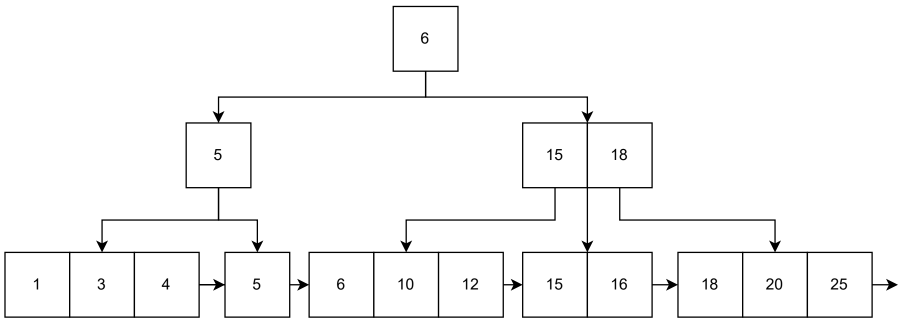


Figure 2.10: Example of a B+ tree with all keys being stored in the tree's leafs.

Figure 2.10 shows the difference between the B and B+ tree. Now if we wanted to do a range query between 5 and 15, we only need to traverse down to the leaf containing 5 and follow the links between leafs until we find the leaf containing the value 15.

As we alluded to earlier, both the B and B+ tree must rebalance during insertion when one of the nodes is full. Figure 2.11 shows the process of inserting values into a B+ tree. Here, **A** shows the tree before insertion, **B** shows the middle leaf node changing after inserting the value 6, and finally, in **C**, we show how the tree has to be rearranged to

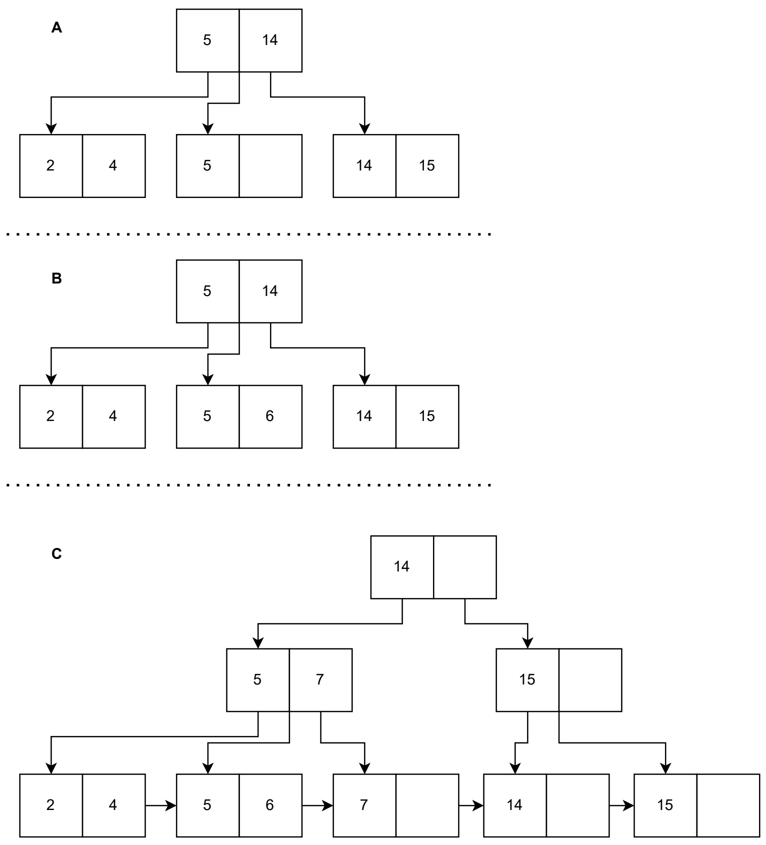


Figure 2.11: Example of an insertion into a B+ tree causing nodes to be split and added to the tree.

accommodate the value 7. An opposite operation happens when values are deleted from the tree. This splitting and merging makes implementing a B or B+ tree more complex than a binary tree.

The complexity of both the B and B+ tree differs from a binary tree in terms of the stability. For the B and B+ tree, the complexity of searching and insertion is $\mathcal{O}(\log n)$ in both the average as well as the worst case scenario. For a binary tree the worst case increases to $\mathcal{O}(n)$ for both searching and insertion. What this means is that the performance of a B and B+ tree is more predicatable than the performance a binary tree.

The number of children per node, otherwise known as the order, is configurable and often configured to be large, i.e., multiple hundred. Used in the context of block storage, the number is chosen such that a single node fills an entire block, which leads to high read efficiency. For example, consider a block size of 4KB, keys of 4 bytes, and 6 byte references. Then the number of children per node d, is chosen such that $4(d-1)+6d \leq 4096$. Solving the equation leads to a value d of 410.

In FrogFishDB we use the design for the B+ tree as the basis for a datastructure which is optimized for indexing time indexed data. In section 4.3 we discuss how we build TimeTree.

2.6 Control and data plane

When running large-scale networks, it becomes more and more time-consuming to manage and configure the network. Therefore, we require methods to configure networks that can be organized from a central position and are visible to the engineers. In the networking world, there exist two *planes* on which different parts of the network run. The *data plane* is the plane where all the data traffic lives, protocols such as TCP (87) live on this plane. The *control plane* is the plane where the routing configuration lives. Protocols that determine routes and are used to exchange reachability information, such as BGP (88) live on the control plane.

We can illustrate the difference between the planes with the following analogy. Imagine a road network with a lot of highways connecting various locations. The data plane can be viewed as this road network. The vehicles use the road network to travel from point A to point B and use the road network as their transport medium. The problem however is that the vehicles need to know which roads to use to move to their destination. In order to figure out the route, the vehicles might employ the use of satellite navigation systems. These give directions to follow the most optimal route. The control plane can be viewed as this navigation system. It not only gives routing directions, but it can also communicate and alter the route based on congestion.

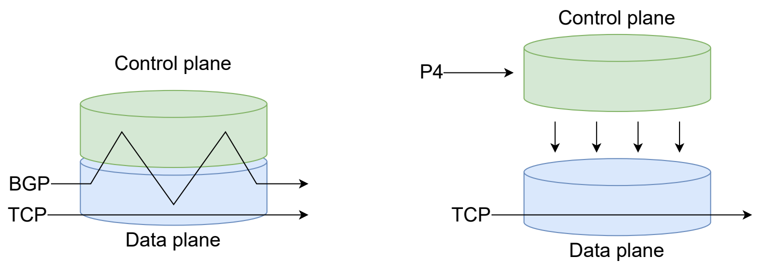


Figure 2.12: Conceptual overview of control and dataplane split.

Recently, there has been a shift in terms of how the data plane is configured. Protocols such as BGP, while classified as control protocols, still run on the data plane to exchange information. Creating a better split between control and data allows for better control and more consistency. *Software defined networking* (SDN) allows users to use a generalized control language such as P4 (89) to define networks. It allows for a physically separate control and data plane. Existing surveys show the large amount of research conducted in this area (90, 91, 92). A conceptual overview is given in figure 2.12. On the left, we show the traditional model in which control traffic is routed through the data plane. On the right, we show a separate setup. There, we show that the control traffic is routed through a separate channel.

While separating the data and control plane has been studied often in networking-related scenarios, we can also apply the logic in software development. For example, imagine an application that retrieves configuration from a remote location. In this application, the control has also been separated from the data logic, allowing for a more centralized control layer. The configuration which is used by the application can be stored and altered from a central position.

2.7 Data model

Before discussing the design and implementation of FrogFishDB, we first have to define our data model. The data model has taken inspiration from the InfluxDB Line Protocol, discussed in section 2.3.1. The data model is defined using the following concepts:

- **Timeseries**: a timeseries is the formal name for a set of values sorted and indexed by time.
- **Metric**: the name of a datapoint, for example, `temperature` or `query_rate`.
- **Metricset**: a collection of metrics. An example could be a database that records a multitude of different metrics such as `query_rate`, `query_latency`, and `buffer_size`.
- **Tag**: an extra piece of metadata attached to a metricset. This can be information such as `version` or `hostname`.
- **Tagset**: a grouping of tags. This is used in both the series name and the query model.
- **Series name**: a canonical name used internally to identify a timeseries uniquely. This name is created as a combination of the tagset and the metric name. For example: `hostname=host_01,version=4.4,temperature`.

2.8 Tags

Tags are an essential feature of timeseries databases that warrants further introduction. Tags can fulfill many use cases; one example is identifying a timeseries through metadata. For example, a tag identifying which host the timeseries originates from.

Such a host could be a server in a data center using the tags to indicate information such as the hostname and IP address, but it might also be used to indicate a physical location, information which is very useful to engineers who need to respond to a failure, either physical or digital, and can thus immediately see which node has failed and where it is located.

2.9 Out-of-order data

One crucial network characteristic we need to discuss before continuing is the possibility of out-of-order arrivals. Data can arrive in an out-of-order pattern, for example, when there is no ordering guarantee from the transport protocol. This is a common problem for IoT data, where networks can be unreliable and unpredictable, and where the sensors make use of the UDP transport protocol (93). This means that one sensor data packet can overtake another one sent earlier. The main problem with supporting out-of-order insertions is that this requires an undetermined amount of buffering or an indexing structure that can support unordered insertions.

In this work, we assume that we are running in a data center network and have guaranteed ordering of data transmissions (94). The argument is that we are running in a data center network where the packet loss probability is negligible, i.e., $\leq 0.1\%$. Ordering can be guaranteed through the use of TCP as our transport layer.

2.10 Query types

In our data model, we discussed the existence of tags and tagsets. These are helpful tools for engineers to provide metadata along with the metrics in the timeseries. One example use case is to use them to identify a machine's physical location. Another example is to record the IP address of the machine.

Metadata tags containing information such as IP addresses can be valuable for engineers solving connection problems. This troubleshooting is aided by queries that support selecting based on tags and tag values. Imagine we have five machines where each records its IP address in the tagset. Then engineers wish to investigate abnormal latency behavior from

3 of those machines. They can do this by querying the timeseries database and using a selector option which allows them to query any timeseries as long as the value of the IP address tag is one from a list.

2.11 Inverted index

An inverted index is a data structure in software such as office document editors. For example, inverted indexes allow us to find which documents contain a specific word (95, 96). Thus when we query the index for a specific word, we get the documents we seek. Extending this operation, we can query which documents contain a set of words. By querying for each word and taking the intersection of all resulting sets, we can find which document contains all queried words.

In FrogFishDB, we use an inverted index to find all timeseries which carry a certain tag. For example, if the database stores 100 timeseries, of which five are tagged with `hostname=host_1`, the then inverted index enables the database to quickly find all five. This is useful when handling queries that search all timeseries which adhere to some predicate pertaining to their tagsets.

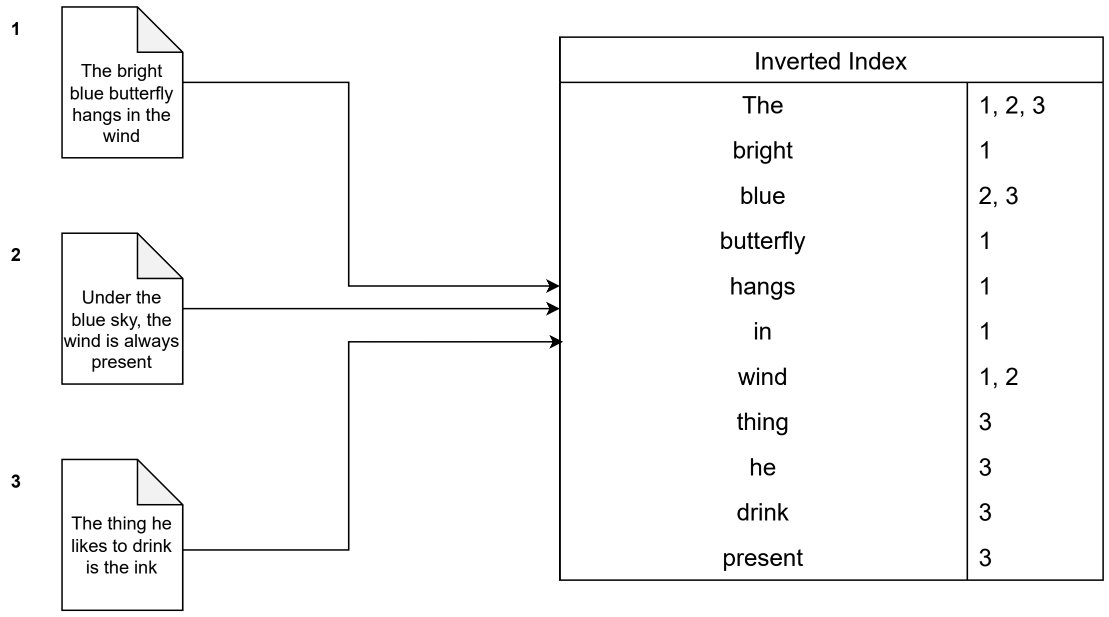

1
The bright blue butterfly hangs in the wind
2
Under the blue sky, the wind is always present
3
The thing he likes to drink is the ink
Inverted Index
The
1, 2, 3
bright
1
blue
2, 3
butterfly
1
hangs
1
in
1
wind
1, 2
thing
3
he
3
drink
3
present
3

Figure 2.13: Inverted index example.

To show how the inverted index works, we use an example. Figure 2.13 shows an example inverted index. Documents 1, 2, and 3 have been merged into one index, which we can query to find the documents containing a specific word. For example, when we wish to find documents containing the word “wind”, we search the index and find that documents 1 and 2 have this word.

A more complex example would be the query for the words “wind” and “blue”. We search for each of the words in the index and end up with 2 sets of indexes, namely $\{1,2\}$ and $\{2,3\}$. From these two sets, we take the intersection, which is document 2.

3

Related Work

The design of efficent timeseries databases are not a new concept and have been researched in the past. Several open-source and commercial examples exist. In this chapter, we will discuss several examples and compare key design decisions.

3.1 Performance of horizontal scaling

To accommodate the increasing bandwidth requirements for monitoring data, multiple efforts have been made to scale timeseries databases such that they are capable of high ingestion bandwidth.

Google designed Monarch (7) to serve as a *planet-scale in-memory timeseries database*, meaning that it is distributed and highly available. Data is ingested at a regional level to achieve higher scalability and reliability. Monarch is thus scaled through horizontal scaling, the workload is distributed globally. Queries are run through a federated layer that partitions queries across different geographical locations. Monarch is shown to be able to write over 2.5TB/s of timeseries data and complete over 6 million queries per second. The main contribution by Adams et al. (7) is the architecture with which timeseries data and queries are distributed across different nodes and data centers. Data is stored at a regional level to increase reliability and scalability. Data is replicated across multiple regional zones to attain reliability. A global query federation layer further indexes the data. This query layer can direct queries to the regions which contain the data. Furthermore, the query layer can direct queries to regions that might be further away from the issuer of the query, increasing latency but which is processing less traffic, meaning that the query can be resolved with less delay. In figure 3.1, we show an overview of the components working together to route both timeseries data and queries to different regions and regional zones. In this figure, we show three regional zones, namely zones 0, 1, and 2. These three zones each contain components for timeseries data storage, query processing, and timeseries indexing. The ingestion router at the top directs data to multiple regional zones to increase reliability. Finally, the query router distributes queries across the different regional zones in order to balance the load.

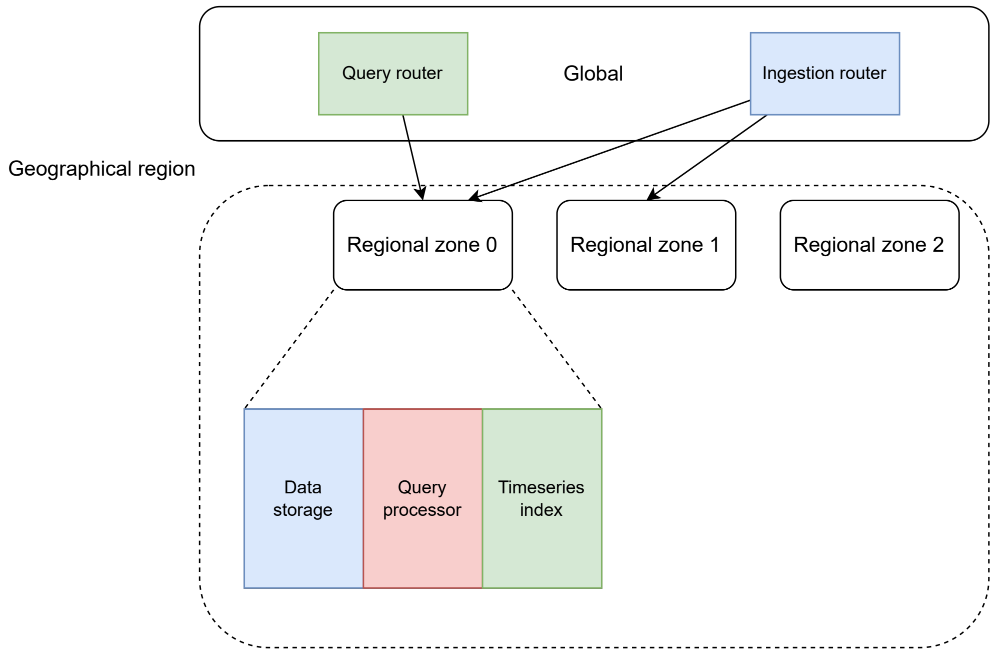


Figure 3.1: Simplified overview of different components making up the distributed architecture of Monarch.

Another contribution from Adams et al. (7) is the type-rich relational model. Using this model, they build a query language that can perform various operations and statistical analysis on timeseries data. One example is the *distribution* type. This type stores timeseries data as a binned histogram instead of as a collection of separate data points. The main upside of storing data in this format allows engineers to issue queries that analyze data in terms of percentiles without first calculating the distributions. For example, in Figure 4 of the paper, Adams et al. (7) show a heat map of the latency of an RPC server. Users can quickly identify outliers and tail latencies using such a heat map.

The next timeseries database we need to discuss is BTrDB (27). BTrDB was created to handle the data from a large number of microsynchophasors (97), which measure the load on the electrical grid. The requirement for this database was that data had to be ingested at a high frequency and using timestamps accurate to 100 nanoseconds. It has been shown that BTrDB can ingest $\approx$ 15 million timeseries data points per second per timeseries, or

240MB/s assuming 8 byte time and 8 byte value pairs. However, this is only achieved when processing a smaller number of connections, about 100. To accomplish this ingestion performance, Andersen et al. (27) introduced a *time-partitioned, version-annotated, copy-on-write tree data structure.* In the paper, Anderson et al. show that the database can scale to the maximum bandwidth of the Ceph (98) cluster used for storing data. However, they can only achieve this by scaling horizontally across four nodes.

3.2 Indexing structures

In section 2.3.2, we discussed the necessity of indexing structures for traditional databases and timeseries databases. Previous works have presented different solutions to this requirement.

One of the most deployed timeseries databases, according to the database engine ranking (99), is InfluxDB (55). InfluxDB is a timeseries database supporting integers, floating point values, and strings. Data is organized in *measurements*, comparable to tables in relational database management systems such as MySQL (64). In InfluxDB, data is stored as *points*, similar to rows in relational databases. Points are combined with *fields* and *tags*, similar to rows in relational systems. Tags are always indexed, making the number of tags used a performance factor. Measurements can be added on the fly and do not need to be explicitly created, as we discussed in section 2.3.1. Being able to add the measurements on the fly is different from traditional relational databases, where tables need to be explicitly created.

InfluxDB has based its storage engine on an LSM tree, at first InfluxDB used LevelDB (100) for its storage API, later a custom storage engine was implemented. Upon receiving batches of data, InfluxDB will first persist the data by writing it sequentially to a *write-ahead-log* (WAL). The authors of InfluxDB consider the WAL to be unsuitable for serving random queries. As such, they employ a cache in randomly accessible memory that holds the same content as the WAL but is used for serving queries. The WAL file compresses the data using the snappy compression algorithm (101), making reading very inefficient due to the overhead of decompression. Thus the need arises for a permanent storage format. WAL files thus cannot be used indefinitely for storing data, so InfluxDB periodically flushes the WAL files to TSM files. TSM files are read-only files that are memory-mapped. These files are structured similarly to *Sorted Strings Table* (SSTable) files found in LSM trees (102). SSTable files are files which contain immutable collections of key-value pairs, sorted by keys.

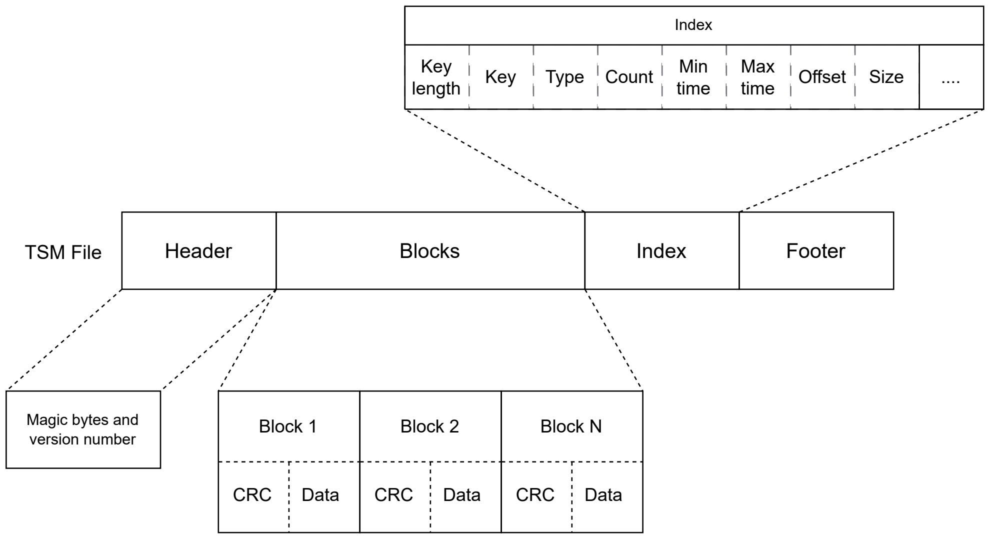


Figure 3.2: Overview of a TSM file.

In figure 3.2, we show an overview of the different sections of a TSM file. TSM files store the timeseries data in blocks. Each block contains a CRC32 checksum to verify data integrity and a variable amount of timestamp-value pairs. The size of the blocks is stored in the index section of the TSM files. This section is located at the end and serves to record information regarding the keys (the measurement name, the tagset, and the name of the field), the types of the values (e.g. bool, string, float, or integer), the size of the block, and the range of time the block represents. Each block is defined by the index section, meaning that the TSM files can contain mixed data.

A *Log-Structured Merge* (LSM) tree (102) is a tree data structure used by multiple time-series databases, such as OpenTSDB (103) and IoTDB (104). LSM trees are noted for their high write performance and support for range-based queries. The structure maintains multiple data structures, each optimized for its underlying storage medium. The original paper describes the use of two structures, one in memory and one on disk. Data is stored by their keys and in sorted memtables. Data is periodically grouped, sorted, and moved to a "lower" ranking storage medium. For example, in the example proposed by O'Neil et al. (102), data is moved from memory to disk, i.e., a medium that resides lower in the storage stack. All data is stored in sorted structures, so range queries are more straightforward, similar to how we handle range queries in TimeTree. A query only needs to find the start of the range by searching through the tree, and seeking the end of the range is a linear search.

Another timeseries database that uses LSM trees is Timon (31). It has to handle many insertions per second, developed and used by Alibaba for its cloud environment. The indexing structure is a tree structure which creates a link between time ranges and data sections stored in files. The key difference lies in how these files are constructed and the structure of the tree. Before writing to storage, data gets grouped into blocks which are written to SSTable files. These files are then indexed by a *Time-partitioning Tree Index*. This index has a similar approach regarding how data is addressed to the one used in FrogFishDB. The tree structure starts at the root node representing the entire range of time stored in the SSTable file, and this is then further partitioned using some fanout factor K, where each level represents a higher resolution but a smaller subset of time. The leaf nodes at the bottom of the tree contain pointers to blocks stored in the SSTable file, which contains the data.

The final indexing structure we need to discuss is the one used in BTrDB (27). Build for recording measurements using highly accurate timestamps ($\approx$ 100ns), the indexing structure needs to support highly accurate timestamps. Described as a *time-partitioning copy-on-write version-annotated k-ary tree*, the tree is built such that the root level describes a range of time, and the level below represents the same range of time but is partitioned across the fanout factor. When data is inserted, it is stored in the leaf. However, in contrast to how TimeTree is built, these leaf nodes are not all on the same level. Leaf nodes are split and moved to a lower level when a leaf is full. This means that when the tree uses, for example, a fanout factor of 10, if a leaf node is full, the tree will allocate ten new leaf nodes and reinsert the values initially stored in the leaf node. The tree used in BTrDB thus can be visualized as growing downwards instead of upwards.

Figure 3.3 shows a simplified overview of the BTrDB indexing structure. As mentioned earlier, the timeseries data is stored in the leaf nodes. Another point that warrants consideration is that the indexing structure for BTrDB allows for gaps, as each point in time is assigned to a location in the tree. Consider figure 3.3. Here the values for timestamps 9, 10, and 11 could be missing (they could have gotten lost due to packet loss). Missing these timestamp value pairs would not change the layout of the structure as described in figure 3.3, the leaf node assigned to the range $[8, 12)$ would still exist, the only difference being that it would only contain the data for $T = 8$ and $T = 12$.

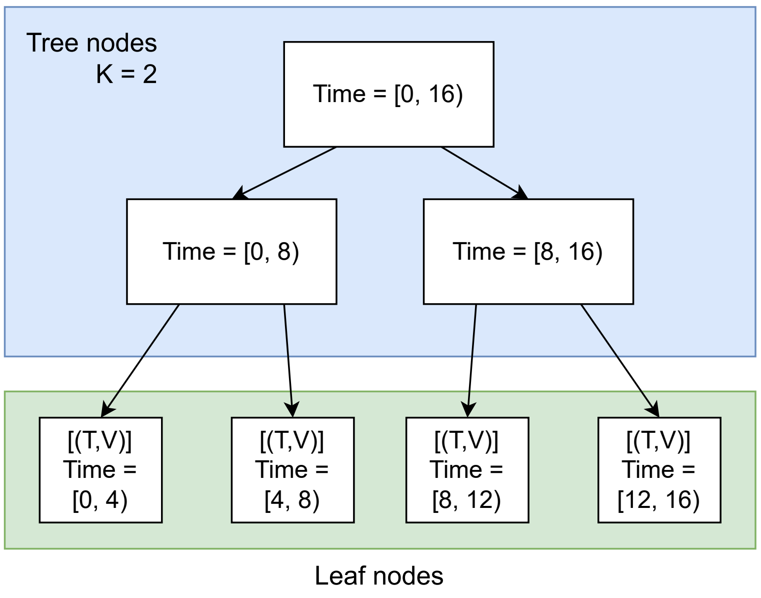


Figure 3.3: Simplified overview of the BTrDB indexing structure.

3.3 Flash optimizations

Section 2.1 discusses the inner workings of flash based storage. We discussed the importance of flash favorable access patterns and how the garbage collection process can induce unwanted overhead. We found little existing work optimizing timeseries databases for flash based storage.

One example we did find was Akumuli (105). However, while Akumuli advertises to be optimized for flash, the only optimization it employs is that block accesses are aligned to 4KB. The effectiveness of this optimization is not explored however, for example, section 5.2.1, we test the bandwidth of three different block sizes, namely 4KB, 2MB, and 4MB. We find that 2MB block accesses are more efficient than 4KB block accesses.

While not directly advertised as optimized for flash, QuestDB's access pattern is favorable to flash. QuestDB is a reasonably new work that aims to provide high ingestion performance through the use of memory-mapped (106) (`mmap`) files. The performance stems from the fact that all writes to the files are sequential and are thus friendly to the underlying storage. Each metric is stored in its own file. As the file are mapped into memory, and the database sees the file through a memory window, insertion can be done using a single `MOV` instruction. The memory window provided by `mmap` is moved once the database reaches the end. This gives the database a consistent memory profile during ingestion, as the window size is constant. While QuestDB advertises a higher ingestion bandwidth, the performance is also poorly understood, as `mmap` is considered not a good fit for database storage engines (107).

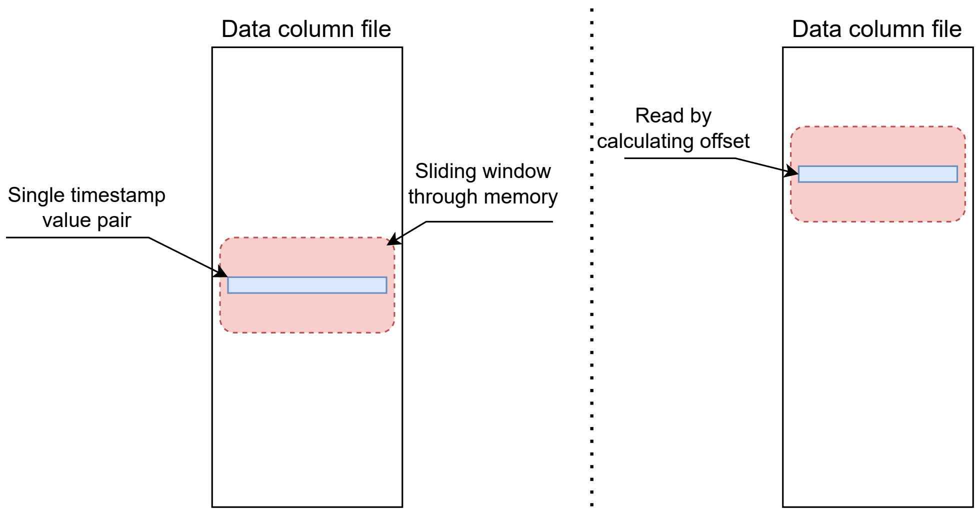

Data column file
Single timestamp value pair
Sliding window through memory
Read by calculating offset
Data column file

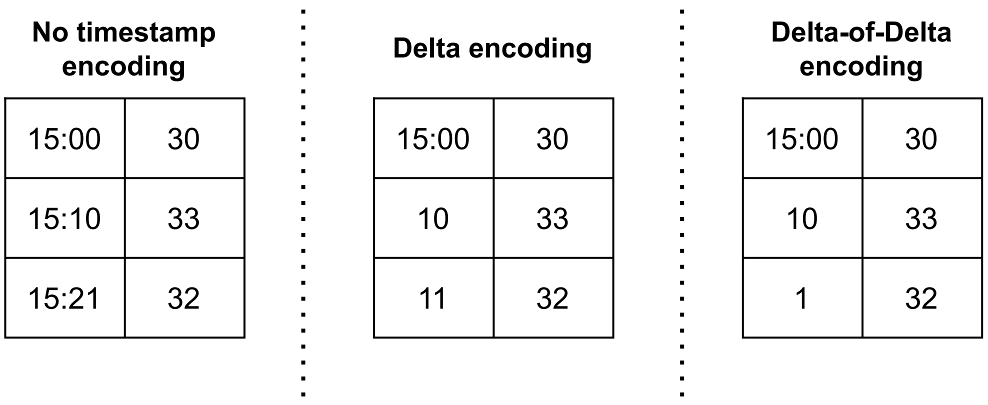
Figure 3.4: Overview of QuestDB writes and reads. We show a value inserted into the memory-mapped region on the left. On the right, after calculating the offset, we show a value being read from the memory-mapped region.

Figure 3.4 provides a simplified overview of how `mmap` offers windows into the data column files. On the left, we show how writes are entered into the memory window. The red box shows the memory window, while the blue line indicates the entry. We show the same on the right, but now we are reading the data. In this case, we only need to read the raw data through the window provided by `mmap`. The use of the sliding memory window means that writes are sequential but reads are still random. This access pattern is favorable to flash as it is sequential in terms of writes and allows for random reads.

Another optimization for flash storage is the use of compression (108). Compression of timeseries data allows a timeseries database to fit more timeseries data into the same amount of space compared to no compression. The use of compression allows a database to issue fewer IO operations to store or access the same amount of data while also incurring less wear as fewer blocks have to be written. However, there is a performance trade-off. With the increase in bandwidth provided by flash storage, if the (de)compression algorithm is too slow, then the overhead of issuing more IO operations is reduced in comparison. For example, if a compression algorithm would allow a database to half the storage requirements, it would also half the required storage bandwidth, however, if the compression algorithm also requires double the CPU resources, then the bandwidth savings might not be worth the computational effort.

For timeseries data, there exist specific compression techniques. An example of this is Gorilla (32). Gorilla is built by Meta to handle large amounts of operational data and is referred to as an *Operational Data Store* (ODS). Gorilla is designed as a fast in-memory database. The main contributions of Gorilla are the compression techniques for both timestamps and floating point data values. Pelkonen et al. (32) show that for Meta, 96% of timestamps can be compressed into a single bit, and 59% of the floating point values can be compressed into a single bit.

The timeseries compression works by taking the delta-of-delta of subsequent timestamps. The idea here is that timeseries often record at a fixed rate, for example, once every 10 seconds. If we only register the difference between two timestamps, we can reduce the data stored per timestamp. Pelkonen et al. (32) took this further and introduced delta-of-delta encoding. This is different from storing the delta between two timestamps in that it records *the difference of the difference*. This accomplishes that the only data recorded is the jitter of the interval at which the data is recorded. For example, imagine a machine recording its CPU temperature every 10 seconds. With delta encoding, we would store the value 10 together with every recording. However, if the jitter between these recordings is low, we can store this delta's difference.



No timestamp encoding

15:00	30
15:10	33
15:21	32

Delta encoding

15:00	30
10	33
11	32

Delta-of-Delta encoding

15:00	30
10	33
1	32

Figure 3.5: The difference in the amount of data needed for each timestamp.

Figure 3.5 shows the difference between the different timestamp encoding methods. Notice that for delta-of-delta encoding a lot less data needs to be stored for later timestamps. For FrogFishDB, we do not use any form of compression, however, this could be explored in the future.

3.4 Summary

In this chapter, we discussed various related works. First, we examined different implementations of timeseries databases that achieve high ingestion performance through horizontal scaling. Monarch and BTrDB advertise high ingestion bandwidth, however, they do so through the use of multiple nodes. For example, BTrDB required four nodes to reach the maximum bandwidth provided by the storage used.

Next, we discussed different indexing structures. We observed the trend of using tree-based structures to index timeseries data. We observed the use of LSM trees for managing stored timeseries data. Other techniques involved storing the timeseries data in the data structure itself.

Lastly, we discussed the optimizations applied to timeseries databases in order to better utilize the bandwidth provided by flash storage. We note a lack of work surrounding optimizations for flash storage. We found that one timeseries database advertises being flash optimized, but also noted the lack of a formal investigation into the effects of this optimization. We also observed the possible use of compression as a form of optimization but also make note of the required tradeoffs.

4

Design of FrogFishDB

In this chapter, we discuss the design of TimeTree and FrogFishDB. We start with a discussion of the various modules and how they interact with each other. While discussing the various modules, we also discuss the design requirements as per RM1. This is followed by an in-depth examination of the new ingestion model. After the ingestion model, we detail the design of our indexing structure. Finally, we discuss the query language we designed.

4.1 Overview and requirements

Before we discuss the design of FrogFishDB, we first discuss some of the design requirements we formulated before starting.

4.1.1 Requirements

As we use the *Design, Abstraction, and Prototyping* research methodology, we have to create a set of design requirements. We formulate the following requirements:

DR1 : *No out-of-order support.* As the database is designed to be run in a data center environment, we assume that no data arrives out of order.

DR2 : *Storage access patterns should be favorable to flash.* Like traditional spinning storage, Flash storage favors sequential accesses and block-aligned accesses for performance (15, 109).

4.1.2 FrogFishDB as a whole

Before we delve into the details of FrogFishDB, we start by giving an overview of the main modules.

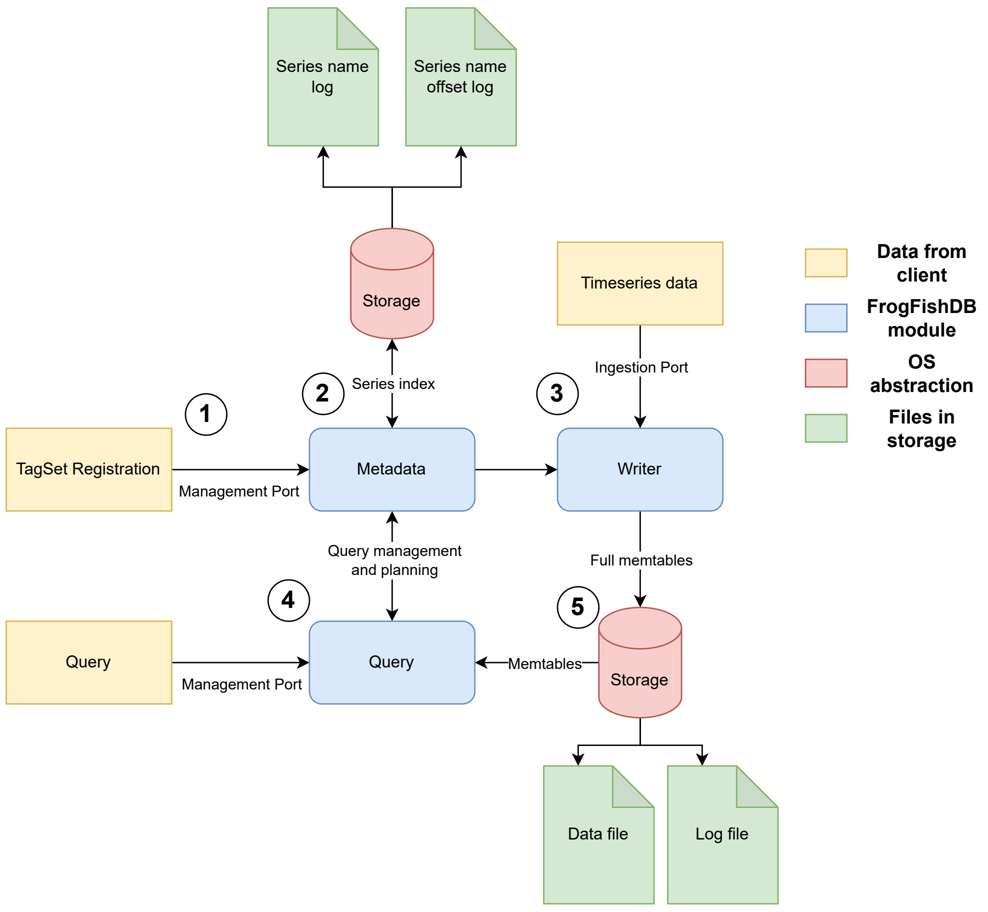


Figure 4.1: Overview of the different components and interaction of FrogFishDB.

Figure 4.1 shows an overview of different components interacting to build FrogFishDB. Three distinct components form FrogFishDB. The first is the metadata component, (2). Though the management port, this component is responsible for registering new timeseries (1) and providing a mapping between registered timeseries and internal index values, this will be further discussed in section 4.2.2. After registering a new timeseries, the metadata module sends the client a token. This token is used to identify the timeseries, this is also further discussed in section 4.2.2.

The second component is the writer (3). This component is responsible for ingesting data through the ingestion port, using the token, in conjunction with the metadata module, to find the relevant data structures for the timeseries data which is being ingested, writing the data to the per-timeseries memtable, and finally, for flushing full memtables to persistent storage ⑤. This component interacts with the storage layer through the `io_uring` kernel interface, as discussed in section 2.4.2. Compared to the writer module, the metadata module uses blocking IO during startup, as the metadata module has to be rebuilt from storage. Timeseries data is written sequentially in an attempt to provide the best possible access pattern for flash storage, as previously discussed in section 2.1, and demanded by DR2. The indexing structure is discussed in section 4.3.

The third and final module is the query module ④. This module receives queries through the management port. Queries are parsed and broken down into; which timeseries need to be read and which operations need to be performed on the data after it has been read. The query module is further expanded upon in section 4.4.

For performance, FrogFishDB is built using an eventloop. This eventloop will submitted requests through to `io_uring` and process completion events. Upon receiving a completion event, it will forward the completion to the component which initially submitted it.

4.2 Ingestion

To optimize the amount of data the database can ingest, we need to design the entire ingestion process such that it allows us to minimize overheads. This section will answer research question **RQ1** (*How to design an ingestion protocol that trades readability away for performance?*).

4.2.1 Push vs Pull

For FrogFishDB, we decided on **push-based** ingestion. With push-based ingestion, the writer module only has to process the incoming data the moment it is received and does not have to concern itself whether the application is reachable. A push-based model is also simpler for clients to implement as sending timeseries data to the database does not require configuring the database. Clients do not have to implement an endpoint from which the database can read the timeseries data, as is the case with Prometheus (51).

There are several options for connecting with a timeseries database. Examples such as InfluxDB use an HTTP endpoint. There, an *HTTP POST* request is send to the endpoint which contains the batch of data which is to be inserted. For FrogFishDB we use a socket based TCP server. We use a binary protocol encoding scheme to encode all data transfered between clients and the database. In order to simplify the implementation we choose *Cap'n proto* (110). This libary allows us to define a messages in a special definition language, from which we can generate implementation for multiple different programming languages, again making it easier for future clients to be written.

4.2.2 Protocol

In section 2.3.1 we discussed a problem with existing ingestion protocols. The problem we discussed was that the use of human-readable protocols introduced unnecessary overhead for the database. During the implementation of FrogFishDB, we find that the database spends around 80% of execution time parsing and converting the data to binary data.

The overhead is this large because, for every timeseries, we construct a separate memtable and indexing structure, and to find these structures, we converted the incoming tags and metric names into a series name. A series name is a canonical name that is unique for each timeseries. For example, in section 2.3.1, we showed an example of the Influx line protocol containing one distinct timeseries. Converting this to our canonical names would result in the following name: `location=us-midwest,temperature`. This name is created through string manipulation of the incoming protocol. Creating these canonical names and then matching them in a hashmap to the data structures (i.e., the indexing structure and memtable) is what ends up taking 80% of the execution time. To lessen the impact of the matching in the hashmap, we also test with different hashing algorithms and a trie. These alternatives did not improve the actual runtime performance, resulting in the same overhead as the hashmap.

Because of this considerable slowdown, we designed a different ingestion protocol. We observed that the tagsets and metricsets used by clients would very often be static and might only occasionally change. For example, an application could note its version number in the tagset, which is a value that is unlikely to change over the application's runtime. Using this insight, we take a similar approach to how RDMA handles connections (111). In RDMA, clients wishing to establish a connection must register with the other endpoint. When registered, they receive a token from the other endpoint, which the client initiating the connection uses to identify itself. In FrogFishDB, we have the client register a set of tags together with a group of metrics, upon which the database responds by sending back a token that the client uses to identify which timeseries a value belongs to. This can be seen as a separation of the control and data plane. Separating the data plane and the control plane is not a new idea. Apart from RDMA, in network configuration, splitting the control and data plane has been shown to be an essential factor in achieving large and scalable networking deployments (112, 113).

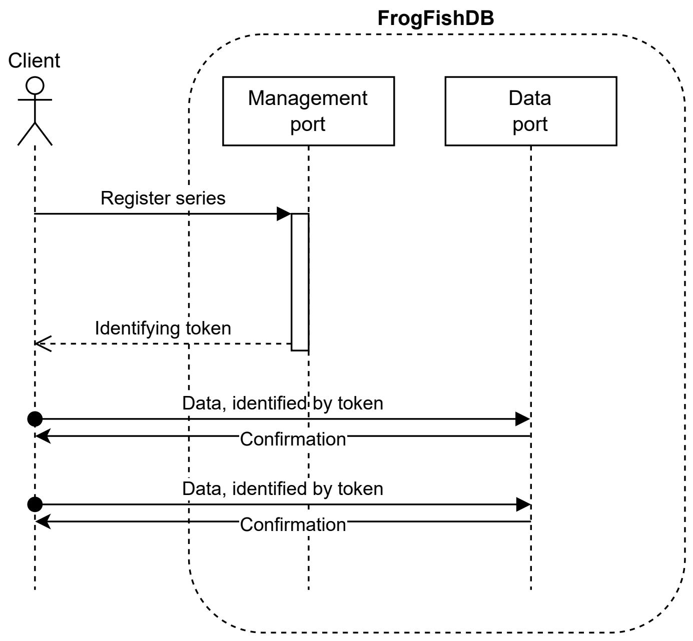


Figure 4.2: Simplified communication flow between client and data. A client registers a timeseries using the management port and receives a token used to identify the timeseries. After the registration, it can identify batches of data using the token.

Figure 4.2 shows a simplified overview of the interaction between a client and the database. The client registers a timeseries with the *management* port, which replies with a token containing the index value assigned to this timeseries. After receiving the token, the client sends timeseries data to the database over the *data* port.

When ingesting data, the database must find which memtable and indexing structure the data must be inserted into. Following what we discussed earlier in this section, we use the token to identify the timeseries and as a link to find the correct memtable and indexing structure. In other words, the index value in the token can be viewed as a canonical name. However, in this case, the name is no longer human-readable. We will discuss the memtable and indexing structure further in section 4.3.

For managing the tags we use an inverted index. This datastructure allows us to insert tags as key-value pairs and then enables us to find timeseries according to one or more tags. For example, in the aforementioned canonical name we saw two tags, namely `hostname` and `version`. Then, using the inverted index, we can find each of the timeseries where one of the tags matches in value. In this example, this allows us to search for each timeseries where `hostname` equals `host_1`.

The tokens are made persistent through two log files, as examplified in figure 4.3. In

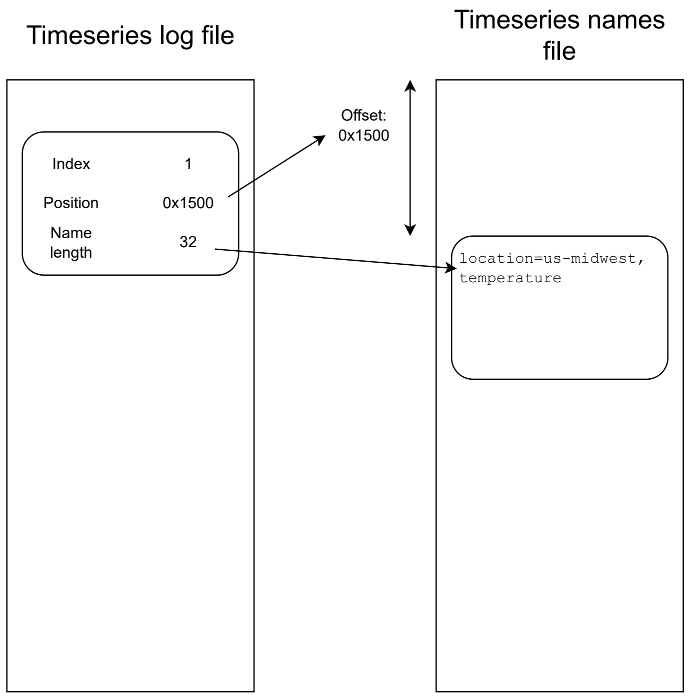


Figure 4.3: Example of the two log files used to record canonical timeseries names.

the *Timeseries log*, the tokens, together with the index values, are stored together with an offset value into the *Timeseries names* file. The timeseries name contains the canonical names of the timeseries which have been registered. The timeseries log, and timeseries name file are separate to allow for possibly lazy evaluation of the canonical names. The names are not required at startup and are only needed when doing specific lookups based on the tags.

Figure 4.4 shows the complete process of registering a timeseries in FrogFishDB. Step 0 is the client initiating the process by registering the timeseries for metric `cpu_usage` together with tag `hostname=host_1`. The management port receives the request and forwards it to the metadata module. A canonical name is created for the timeseries and checked against the index. If the index does not contain the canonical name, we start the process of registering the new timeseries, as seen in step 2. Otherwise, we immediately return a token with the index value. Steps 3a and 3b show how the timeseries is made to be persistent by registering the index value into the timeseries log, and the canonical name into the timeseries names log. Step 4 returns the index value to the metadata module. We register the tags into the inverted index in step 5. Finally, in steps 6 and 7, the metadata module packages the index value into the token, which is returned to the client.

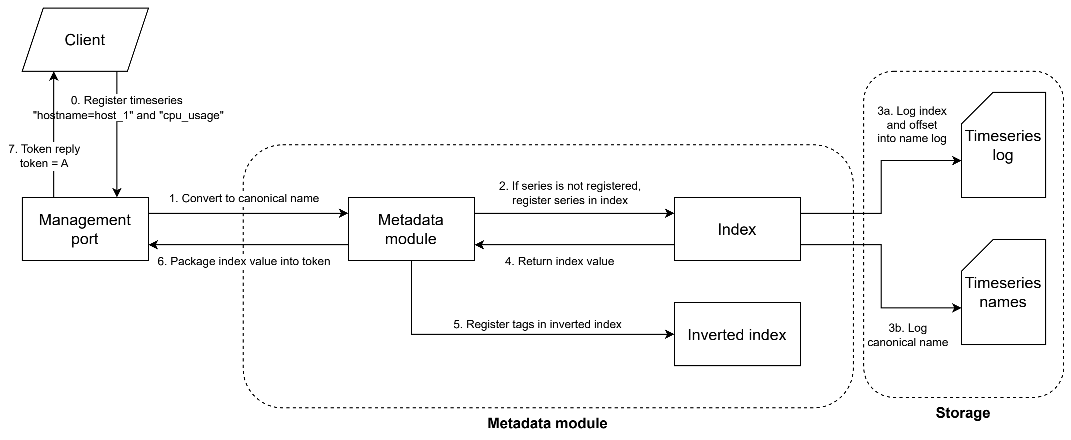

Client
0. Register timeseries
"hostname=host_1" and "cpu_usage"
7. Token reply
token = A
Management port
1. Convert to canonical name
6. Package index value into token
Metadata module
2. If series is not registered, register series in index
4. Return index value
Index
5. Register tags in inverted index
Inverted index
Metadata module
3a. Log index and offset into name log
Timeseries log
3b. Log canonical name
Timeseries names
Storage

Figure 4.4: Diagram showing registering a timeseries in FrogFishDB.

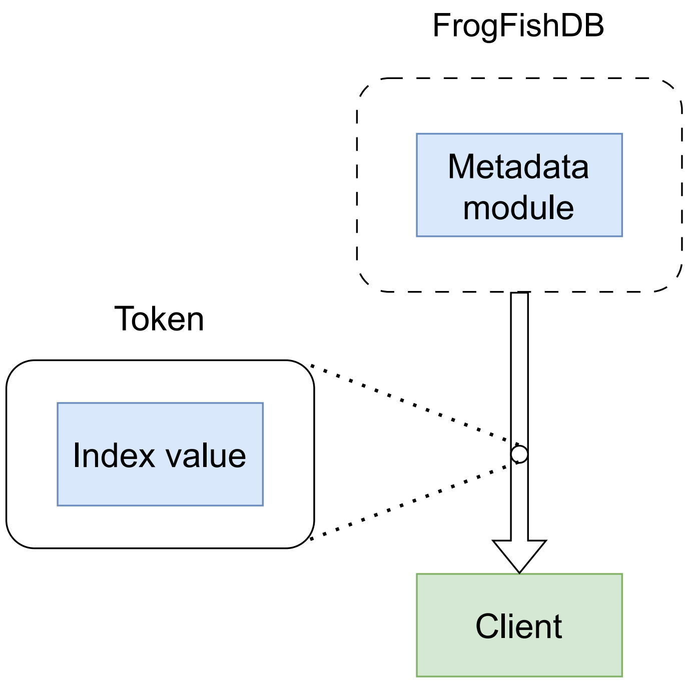
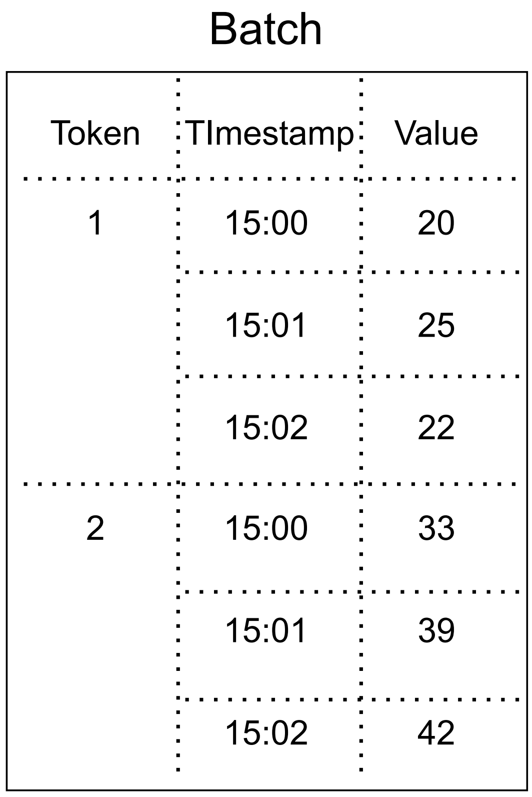


Figure 4.5: Database replies with token to the client.

To further illustrate steps 6 and 7 of figure 4.4, see figure 4.5. The database sends a token in reply to a request to register a new timeseries. The token contains an index value which is used by the database during subsequent ingestion process, as discussed in section 4.3.2.

The trade-off is that this new method requires an extra communication round-trip where the application will first have to register the tagset and metricset, however we find that the cost of this round-trip is fully amortized over the runtime of the application, especially considering the long run times of applications running in a data center.


Batch

Token	TImestamp	Value
1	15:00	20
	15:01	25
	15:02	22
2	15:00	33
	15:01	39
	15:02	42

Figure 4.6: Batch containing timeseries data to be inserted into the database.

Another method for increasing ingestion bandwidth is through the use of batch insertion. Here we insert a large batch of timeseries data in one transaction. The current implementation of FrogFishDB uses synchronous insertion. The client submits a batch of timeseries data and waits for a reply confirming that all timeseries data has been inserted. The batch size has previously been shown to impact insertion bandwidth (31). For increased efficiency, we designed the wire protocol such that a batch can contain multiple datapoints per token and that a batch can contain multiple tokens. Figure 4.6 shows how such a batch is layed out. This batch contains data for two timeseries, denoted by the tokens 1 and 2.

For each token, the batch contains three timestamp-value pairs.

4.2.3 Summary

In summary, we present the following design decisions for the ingestion protocol. The first decision is that we use push-based ingestion. Next, we decided that we shift part of the ingestion workload back to the client. A client is responsible for registering a set of tags for which it obtains a token to identify a batch of timeseries data. The client sends batches of timeseries data identified by the token to the database and waits for a response confirming all data has been inserted.

With this design, we can answer RQ1. We trade away readability for performance by registering sets of tags before sending the timeseries data. Data is further encoded using a binary protocol (only machine-readable) to avoid the need to decode human-readable text into machine-readable values. Future work would consist of making the protocol asynchronous. In that case, multiple batches can be sent in quick succession without the need to wait for confirmation.

4.3 Indexing structure

In this section, we will discuss the indexing structure built for FrogFishDB. We designed the data structure to take advantage of the fact that we assume all data arrives in order. Using the assumption, we can create a data structure with the search and insertion time complexity of a B+ tree but without the possible complexity of the splitting and merging operations, leading to more predictable performance. A B+ tree will have to split internal nodes in order to accomodate new entries, as we discussed in section 2.5. To retain balance the tree will rebalance internal nodes on the right side of the tree when we insert ordered data. We refer to our data structure as the TimeTree, a tree structure that fans out based on time. With the indexing structure we aim to provide an answer to RQ2.

One of the key problems with databases is finding the data again after committing it to storage. Different databases have shown different solutions, which we will discuss further in section 3.2. The key difference between standard relational database operations and those encountered by timeseries databases is that timeseries data is write heavy, but linear in the sense that timestamps are only ever increasing. Using this difference, we designed an indexing structure to exploit this fact. Indexing structures used in existing timeseries databases have used this same fact to different degrees of success (27, 31).

4.3.1 TimeTree

For the indexing structure of FrogFishDB, we take the B+ tree for its efficient search and insertion properties. We simplify its operations to better fit our assumptions and insights into how timeseries data will be processed by FrogFishDB. The first important factor is that we assume all timeseries data will arrive in order. This assumption makes it easy for us to reason about how the data can be laid out on flash SSD, as with in-order data, the writer module can write to a memtable in memory and flush this memtable directly to storage. The second is that timeseries data which has been written to storage is immutable. Once data is written to storage, it can only be read, but not altered.

From these two factors, we derive a B+ tree which is much simpler in terms of design and implementation. *TimeTree* is the data structure that we have developed. In a basic sense, the TimeTree can be seen as a B+ tree that grows upwards instead of downwards. The main idea is that all leafs contain pointers to memtables that have been written to storage. The internal nodes contain the range of time their children represent.

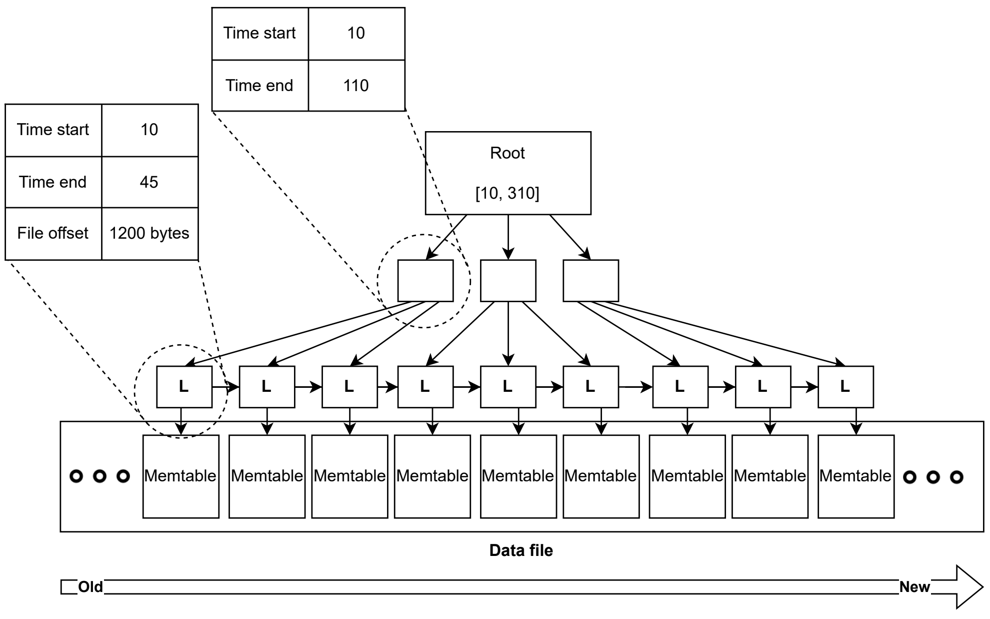


Figure 4.7: Overview of TimeTree. Full tree stores data between timestamps 10 and 310. Nodes with the L mark are the leaf nodes with the extra field pointing to the data file.

Figure 4.7 shows and overview of a TimeTree instance. This tree is configured to have a fanout degree of three. This means that each internal node in the tree has at most three children. The tree represents a range of time, starting at timestamp 10 and ending at timestamp 310. The left most internal node is shown to represent a range between 10 and 110. The range of this internal node is further partitioned in three children. These children are leaf nodes, where the first leaf node represents a range starting at 10 and ends 45. The leaf nodes differ from the internal nodes in that they contain a value which points to a memtable where the timeseries data is stored. The value represents the number of bytes from the start of the file where the memtable is located. The memtables are stored in memory and are written to by the writer module. When a memtable is full, it is flushed to storage by appending it to the data file. The offset, or number of bytes from the start of the file, is stored in the TimeTree. This is why data on the right side of the tree is newer than data on the left. As more memtables are flushed to storage, the number of leaf nodes also grows. In order to accomodate more leaf nodes, the tree needs to grow, thus the height of the tree is proportional to the fanout degree and the number of leaf nodes.

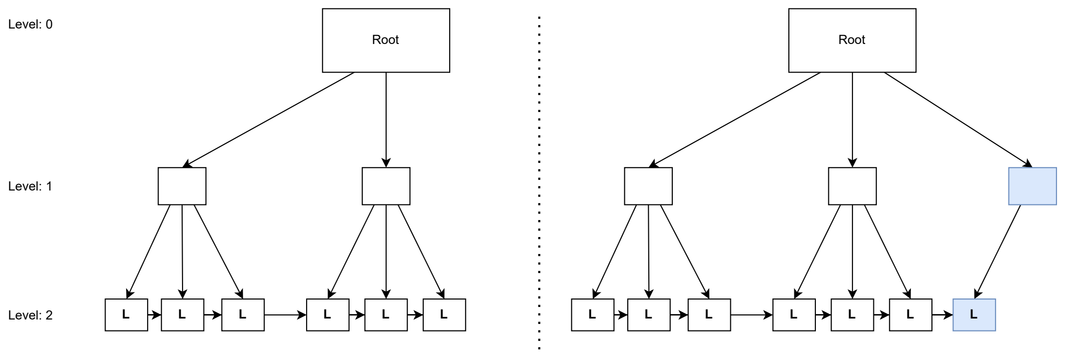


Figure 4.8: Insertion into the TimeTree structure example.

Figure 4.8 gives an overview of how the insertion works in the TimeTree structure. The blue nodes have been inserted into the tree. When inserting the blue leaf node, we had to create the blue internal node as there was no room left in the newest internal node at level 1. The cost of insertion thus depends on the space available in the internal nodes at levels above the leaf node. In the worst case, the number of internal nodes that need to be created is equal to the tree's height, meaning a new root and a new internal node on each of the existing levels. Also note that there are links between leaf nodes. These links aid with lookups where we need to scan over multiple leafs nodes.

We wish to make it very clear that in the context of FrogFishDB, we store do not store pointers but the offset into a file. However, this is not a hard requirement. If an application wishes to do so, the application can store a pointer to an object in memory or to an object stored in an object store such as S3. Another situation would be to store some aggregate value, for example, a median value. This median value can represent a median for a range of time. Reconstructing the tree from the log file would also recover these median values. We discuss this concept further in section 4.3.3.

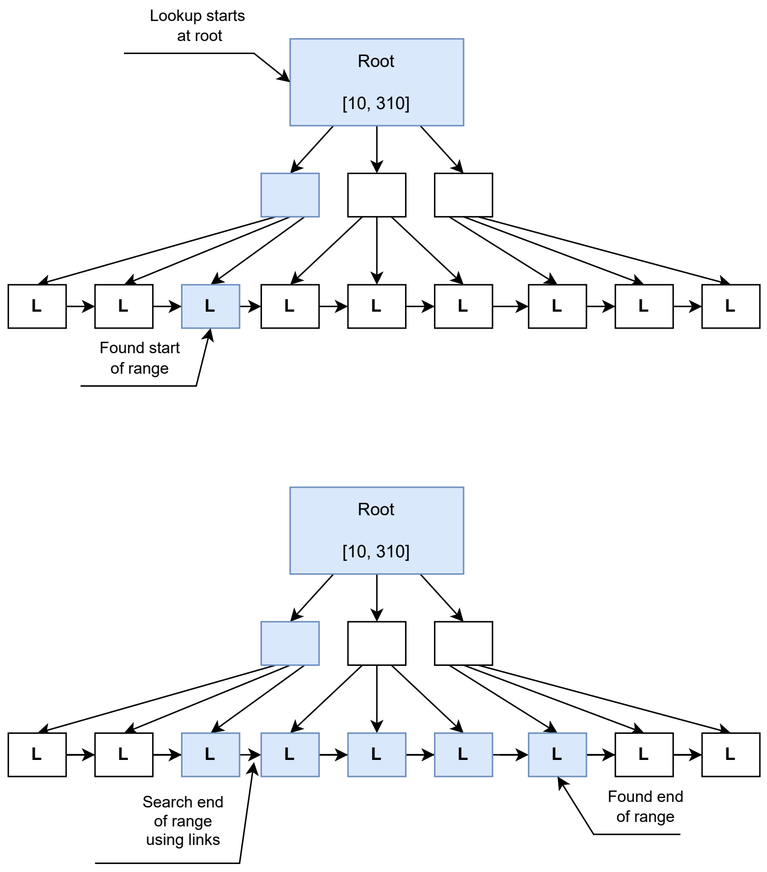


Figure 4.9: Lookup in the a TimeTree. The upper example shows the search for starting leaf node. The lower example shows the start having been found and the use of the links between leaf nodes to search for the end of the range.

Figure 4.9 shows a conceptual version of a lookup in the TimeTree. Doing a lookup is very similar to doing one with a B+ tree. The user supplies a start and end timestamp for which to search. We start by searching for the start of the range of time. We follow the tree nodes until we find the leaf node representing the range's start. After finding the start, we collect the offset values by walking the leafs using the links between leafs until we encounter the end timestamp.

4.3.2 Storage

The underlying storage mechanism is purposefully kept as simple as possible. This is done both to fulfill DR2 and to keep future implementations free of the burden of an existing storage solution. In the latter case, this allows future users to alter the storage layer and, for example, build an implementation that runs directly on flash through SPDK (114). We discuss this option further in section 6.2. Currently, we use `io_uring` for the storage IO. Together with the `DIRECT IO` mode for files, we route all storage related IO traffic through `io_uring`. While not investigated in this thesis, this would also enable us to use features such as different polling levels, as has been discussed in section 2.4.2.

Timeseries data is stored in memtables. These are blocks of memory that are configured to be a specific size. In section 5.2 we experiment with differently sized memtables. As these memtables are nothing more than just a block of memory, they allow us to potentially implement compression algorithms on top of them such as those discussed in chapter 3. However, that is outside the scope of this work.

Flushing a memtable to storage involves appending the memtable to the data file and recording the offset of where it was written. The offset is then stored in the TimeTree. The insertion into TimeTree is logged to the log file by the writer module when the flush of the memtable has completed. This log file is what ends up being used to recreate the indexing structures for all the timeseries.

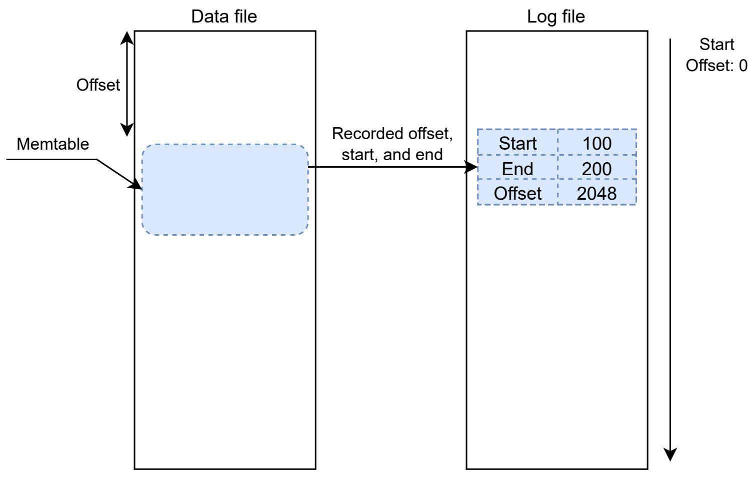


Figure 4.10: Memtable's offset and metadata are stored in the log file.

Figure 4.10 details the storage setup. This figure shows the two files, namely the data file and the log file. The blue box in the data file represents a memtable that has been flushed and is stored at offset 2048 bytes into the data file. After the write to the data file, the offset and metadata, such as start and end timestamps, are written to the log file. Upon restarting, the database reads the log file and can recreate the indexing structures in memory. The log file is also read from start to finish, making it so the invariant holds that all data arrives in order.

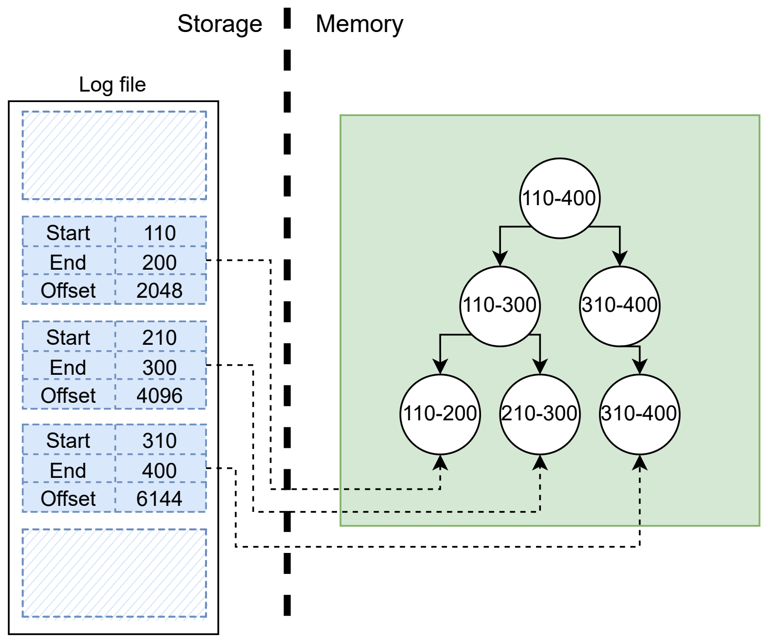


Figure 4.11: Reading the log file and recreating the indexing structure.

Recreating the indexing structure from storage is further visualized in figure 4.11. Here we attempt to make a clear distinction that the indexing structure lives in memory but is built by reading the log file. In this figure we show a tree which represents a range of time, spanning from timestamp 110 to 400. Each internal node has a fanout of two, where the left subtree (110-300) is complete, and the right subtree (310-400) is not complete. The three blue blocks in the log file on the left of the figure show the nodes after they have been made persistent. Note that the internal nodes of the tree are not in the log file, these are recreated during when the log file is replayed on startup. The log file is replayed in order to recreate the tree structure. As the internal nodes do not contain any actual data and are only a summary of the children of the node.

In the current implementation, we persist the TimeTree entries by writing them to a log file. However, this implementation is not the most efficient. The memory usage of TimeTree is a function of the degree of fanout that has been configured. Increasing the fanout degree will decrease the number of inner nodes as one node can represent more children. When we log insertions into the TimeTree, we use direct IO, meaning that each write is rounded up to 512 bytes. Assuming that each insertion can be represented using

24 bytes (16 for the two timestamps, and 8 for the value), we observe an overhead of 488 bytes (512 − 24). In section 6.3 we detail a possible solution to this problem.

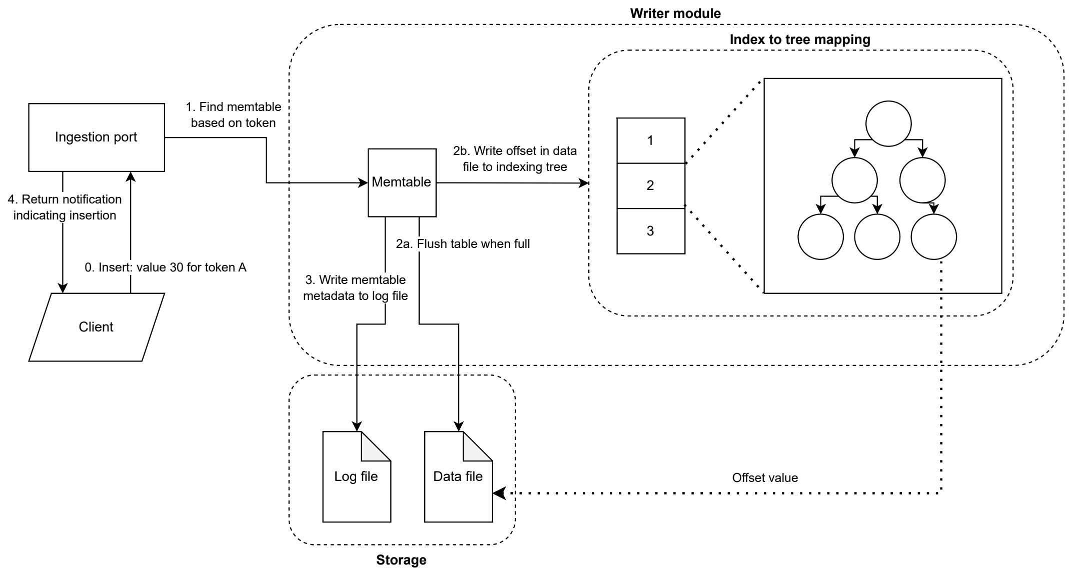

Writer module
Index to tree mapping
1. Find memtable based on token
Ingestion port
Memtable
2b. Write offset in data file to indexing tree
1
2
3
4. Return notification indicating insertion
2a. Flush table when full
0. Insert: value 30 for token A
3. Write memtable metadata to log file
Client
Log file
Data file
Offset value
Storage

Figure 4.12: Systems diagram showing the process of inserting a value into the database.

Figure 4.12 shows the insertion process. Step 0 is the client sending the value to the database. The values are received by the ingestion port and written to the memtable in step 1. The index value stored inside the token is used to find the memtable. Steps 2a and 2b show the memtable being written to storage if the memtable is full and its location in the indexing structure. If the memtable is not full the writer module skips to step 4. In step 3, the memtable's offset into the data file is stored together with its start and end timestamp in the log file. Finally in step 4, the ingestion port returns a confirmation to the client indicating that the write was successful.

In section 2.4.2, we discussed using registered buffers to improve performance. Due to time constraints, we were unable to experiment with this feature. Using registered buffers would have allowed us to forgo multiple data copying steps. For example, a multishot receive operation using `io_uring` would have allowed us to ingest batches of data more efficiently. The current version receives messages by reading the data from the socket file descriptor into a temporary buffer. This buffer is necessary to process messages bigger than the receive buffer. When batches contain more than 10,000 data points, the size is over 160KiB, far larger than what TCP can carry in a single message (i.e., 1,440 bytes).

In this thesis we do not consider fault tolerance. Timeseries data is only persistent after the memtable is flushed to storage and the TimeTree leaf node is logged to log file.

Both of the writes to storage use `DIRECT_IO`, meaning that when the completion for the writes is received through `io_uring`, we know that the write is persistent. Improving fault tolerance is considered to be future work. One option is through the use of a *Write-Ahead Log* (WAL) (115). All timeseries data we receive through the data port, we write to the WAL. Upon restarting the database, we can detect a fault by reading the log and checking if the latest values in the log are present in the indexing structures. However, using a WAL comes at a cost (116). For example, if we were to log each timestamp-value pair, this would result in many small writes to the log, reducing our total bandwidth.

4.3.3 Aggregation

Another advantage of assuming in-order insertion into the indexing structure is that it allows us to "fold" older data. When storing large amounts of timeseries data, employing a *retention* system can be beneficial. In this case, data older than some retention value, for example, one month, is deleted. The idea is that older timeseries data is less likely to be relevant in the future. Think of this in the context of monitoring system health. When the timeseries data is older than one month, it is unlikely to be relevant for monitoring tasks running in the present.

However, even though older timeseries data might not be relevant for current tasks, it can aid in identifying trends and investigating performance problems. There is a clear trade-off here. Older data *might* be relevant, but if it is not, it will still occupy storage space, and with high ingestion bandwidth, this storage space comes at a premium. When free storage space decreases, one will want to free up space by removing older data to be able to ingest the new data.

We present the *aggregation* process for this scenario in the TimeTree. Here we "fold" older nodes into nodes at a higher level. Figure 4.13 shows an example of this process. Here we offer a before and after visualization of the TimeTree. On the left we show a tree with seven leafs, and on the right, we show how the three oldest leaf nodes have been aggregated into the tree node one level higher.

We can only aggregate nodes into a higher internal node when the higher node is considered *complete*. This is only when there is no room left in the node, in other words, when all child slots have been filled. In the case of the example shown in figure 4.13, only the six oldest nodes are eligible for aggregation into their respective parent nodes, as the parent for the 7th node is not yet complete. Note that only the oldest three leaf nodes have been aggregated. This is a configurable setting, where the user which nodes are selected for aggregation. For example, consider our earlier example of a retention system. With

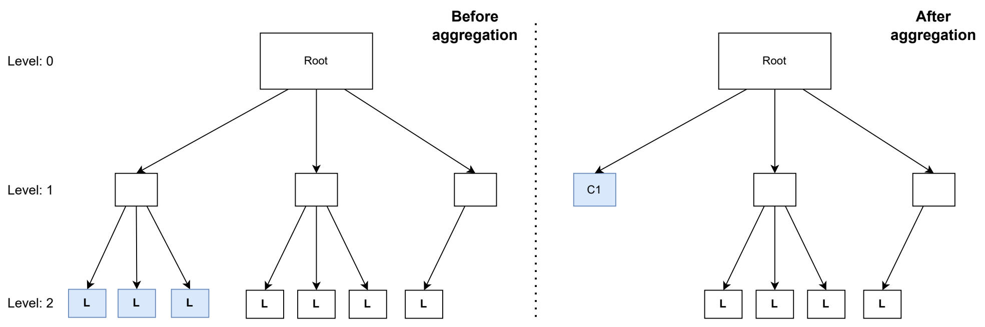


Figure 4.13: Example of three leaf nodes getting aggregated.

aggregation the user could opt to only aggregate nodes which are beyond a certain age. Another option would be to simply aggregate all complete nodes.

The aggregate node (`C1`) shown on the right in figure 4.13 contains a summary of the information stored in the leaf nodes. The application can extend this summary information and could contain information such as the maximum, minimum, and average values. For this, we can also image applications storing more complex information, such as a distribution, similar to what Adams et al. (7) have shown in Monarch, which we discussed in chapter 3.

Due to time constraints, we have been unable to implement this feature in FrogFishDB. We do show the functionality working for TimeTree. However, this is only offered in memory. The aggregation process will also need to be recorded by storage. For this, we suggest the use of special log entries. These entries should indicate which nodes have been aggregated and the summary values. When the log file is replayed, this aggregation process is repeated, however, without the need to calculate the summary values.

4.3.4 Summary

In this section, we presented the TimeTree. A variant on the B+ tree with the aim to provide a simple indexing structure for storing timeseries data. This allows us to give an answer RQ2. A data structure for timeseries data can be designed through careful consideration of the domain in which it is employed. For timeseries data, this means that through the assumption that data is ingested read-only and arrives in-order, we designed a variation on the B+ tree which forgoes the process of splitting and merging nodes, forgoes the option to delete entries, and assumes add data arrives in-order. This variation allows for both fast insertion and efficient range queries.

4.4 Querying

This section will focus on the query aspect of FrogFishDB. In FrogFishDB, we provide a simple but specialized query language. Other works also present their own language implementations, as the authors found that the traditional SQL language did not provide the features to have effective and expressive queries on timeseries data.

4.4.1 Why the need for a new query language?

Traditional relational databases have traditionally employed the use of the *SQL* language. Through this language, one is able to describe a query to a database. It allows the user to, for example, specify the columns which need to be returned, in which table the query should be executed, and allows for specifying post-processing operations that may combine queries from multiple tables into a single result. While this language has worked for relational databases, its limitations become clear when used in different settings.

For example, consider a database with the following table. The table contains a log of a set of products, all of which are sold by five different vendors. One column specifies the price, the second column specifies the product to which the price belongs, another column specifies the retrieval date for the price, and the final column specifies from which vendor this price value has been retrieved. Now consider the following query, we which to find all the latest prices from all vendors for one specific product. To execute this query, we have to create two nested queries which use an inner join to get the result. The first query is a basic one that retrieves the basic information from the table, i.e., the price, the retrieval date, and the vendor to which this price belongs. The second query has to query the same table as the first. However, this time it has to retrieve the vendor and the maximum retrieval date, together with a filter that selects the product for which the prices must be retrieved. Finally, this query has to group the data based on the different vendors. Finally, after executing the second query, can the first query be completed through an inner join that matches the retrieval date and vendors with the results from the second query.

Note how we are only interested in the most recent price for all of the vendors. This workload is very similar to what is requested of timeseries databases in terms of workload. As a comparison consider a scenario where we query the database for the current temperature of multiple servers. In that case we are also only interested in the latest value. If we were to convert the log of product price data to InfluxDB, then we would ingest the prices together with the retrieval time as the timeseries data. The specific product and vendor would then be recorded as tags.

We compare SQL and Flux, the query language InfluxDB uses, to demonstrate the difference between query languages. Listing 4.14 shows the same query in two different query languages. On the left, we show the SQL equivalent of the query we described earlier. On the right, we show the equivalent query using InfluxDB.

```sql
SELECT price, product, vendor,
    retrieval_date
FROM price_table p
INNER JOIN (
    SELECT vendor,
        MAX(retrieval_date)
    AS ret_date
    FROM price_table
    WHERE product = product_1
    GROUP BY vendor ) pm
ON p.vendor = pm.vendor
AND p.retrieval_date =
    pm.retrieval_date
```

Listing 1: SQL example.

```
from(bucket:"prices")
    |> filter(fn: (r) =>
        r.product = product_1)
    |> last()
    |> yield()
```

Listing 2: InfluxDB's Flux language.

Figure 4.14: Comparison between two different query languages.

Listing 1 is difficult for the reader to process and understand compared to listing 2. First, the reader will have to process the fact that there are two queries at play here, and then they will have to realize that they pertain to the same table. Listing 2 shows a comparatively simple query that consists of only four statements. The first statement acts as the select statement from SQL, selecting the different columns from the "prices" table (otherwise referred to as bucket in InfluxDB). The second statement filters the queried data to only contain `product_1`. The third statement selects the latest data inserted into the table. The fourth and final statement denotes the end of the query and yields the processed data back to the issuer of the query. All statements are connected using the pipe operator (`|>`). This operator forwards the result from the previous statement to the next.

There exist multiple examples of timeseries databases in literature in which the authors opted to either design their own language or create extensions to SQL. Timon (31) uses a query language dubbed TQL. This language is intended as an extension of SQL with two key differences. The first is that queries and functions can be changed using a pipe operator (`|`), similar to how in Flux multiple statements can be chained. The second difference is the introduction of extensions to the SQL language, allowing users to match data according to tags used to identify timeseries. In Monarch (7), the query language is designed from the ground up. Described as a "pipeline of relational-algebra-like table operations", the language allows the user to chain queries similarly to TQL. Data is retrieved using a `fetch` operation, the results of which are piped to functions such as `filter` or `group_by`. When more than one query needs to be forwarded, they can be joined. Multiple queries are first described in a "block", denoted by a brace pair (`{}`). The main downside of these two languages (TQL and Monarch) is that both are used for proprietary systems and thus are only visible to us by what the authors have shared in their papers.

4.4.2 Language design

Due to time constraints, we opt to keep our language design as simple as possible. We could have opted to copy or otherwise imitate one of the languages mentioned earlier, however, we opted not to do this as this would have required the development of a significant parser and interpreter, while this work focuses on the ingestion pipeline. For the language, we identify the following requirements:

LR1 : Select timeseries based on tag

LR2 : Select timeseries based on index value

LR3 : Indicate the range of time from which to retrieve the specified timeseries

LR4 : Specify a `GROUP BY` operation which can operate on differently sized bins of time

These requirements were chosen such that they provide the minimum viable query language and provide all the operations necessary to execute the benchmarks discussed in section 5. Requirements LR1 and LR2 enable us to make a basic selection for a given timeseries. These queries are comparable to an `SELECT * FROM` statement in SQL. Requirement LR3 is used for narrowing the timeframe which the query selects. The usecase for such a time filter would be that a user wishes to view the last six hours of timeseries data and not all available. Requirement LR4 specifies a post-processing operation. This operation is used to create summary statistics of timeseries data. For example, a group-by operation can be used to the average values of timeseries data which has been binned in blocks of fifteen minutes.

To keep the implementation as simple as possible, we looked towards one of the most fundamental languages, namely LISP (117). While we show a language that can fulfill all the requirements we have set, it lacks any form of formal proof. Even though this work will not present a complete design document for a language, we can still delve into some of the underlying ideas of LISP which are relevant to a query language.

The syntax for LISP is characterized by the fact that all operations are written through symbolic expressions (otherwise referred to as s-expressions or sexpr's). Used initially for calculators in the 1970s, infix operators make for a stack-based model. In this stack-based model, operations are first put on the stack, followed by the arguments. The main advantage of the stack model for this work is that this makes writing a parser very easy.

The idea of using LISP inside another application is not new. One such example is *Game Oriented Object Lisp* (GOOL) (118). Created for the videogame *Crash Bandicoot*, the language provided several features usefull to game development. The core idea of the language was to enable developers to quickly iterate on code which was not performance sensitive enough that it required hand-written assembly. GOOL is implemented as an interpreted language and uses a *Read Eval Print Loop* (REPL) system for interacting with the language. This REPL enable developers to alter the code of an application at runtime and observe the effects without restarting the application. This means that the entire process of compiling and restarting the appliation is skipped.

Before we can discuss our query language, we first show a formal definition of the LISP lanaguage:

```
expression = atom   | list
atom       = number | symbol
number     = [+-]?['0'-'9']+
symbol     = ['A'-'Z']['A'-'Z''0'-'9'].*
list       = '(', expression*, ')'
```

Listing 3: EBNF form of the LISP language.

Listing 3 shows the formal definition of the LISP language in EBNF (119) form. From this we can see that expressions are made up of either an *atom* or a *list*. An atom is the smallest functional unit in LISP, it is either a symbol or a number. A number consists of one or more number characters in an unbroken string, optionaly prefixed by a symbol indicating wether or not the number is a negative. The same holds for a symbol. A symbol is an unbroken string of alphanumeric characters. A list is a matched pair of parentheses containing zero or more expressions.

As an example, the expression `(+ 1 2)` calculates the sum of 1 and 2. Note that due to the expression grammer, the mathematical notation follows the polish notation (120).

For FrogFishDB we also created an embedded LISP interpreter. Due to time constraints we were not able to create a full fledged interpreter, however, we were able to create a simple parser which is able to fullfill all of our previously defined requirements. What follows are a few examples queries to show the different features of the language.

```
(->>
    (metric "cpu_usage")
    (tag "hostname" '("host_0")))
```

Listing 4: Simple example collecting all measurements for given metric and tag.

Example 4 shows a straightforward example where we query the database for all values of `cpu_usage` where the tag `hostname` equals `host_0`, equaling the requirement from LR1. Going from top to bottom, the `->>` literal indicates the start of a new query, the metric we target is specified through the `metric` keyword, and the tag combination can be indicated with `tag`. The `tag` command has two overloads. The first is for selecting a single value, and the second is used to select multiple values, which is done by passing a list. Note that the `'` indicates a list literal, keeping in line with the common-lisp language. Otherwise, the tag value would have indicated a function.

```
(->>
    (index 50))
```

Listing 5: Simple example collecting all measurements for a given id.

Example 5 is a straightforward example showing LR2. If the user is aware of which index value corresponds to metric `cpu_usage` and `hostname=host_0`, then they can use this index value to issue a query.

```
(->>
    (metric "cpu_usage")
    (tag "hostname" '("host_0"))
    (range 1464710340000000000 1464720460000000000))
```

Listing 6: Simple example collecting from range of time.

Example 6 shows the same query as example 4, however this time, we indicate the window from which we wish to select the data, as required by LR3. The `range` keyword indicates that timestamps should be between the two arguments. The arguments shown here are two nanosecond precision UNIX timestamps.

```
(->>
    (metric "cpu_usage")
    (tag "hostname" '("host_0"))
    (groupby 1h max))
```

Listing 7: Simple example grouping data in bins of 1 hour.

Example 7 is again an alteration on example 4, however here we group the results into bins of 1 hour and apply the maximum operator on each of the bins, fulfilling LR4. While processing this query, we wait for the system to have loaded in all the data and then move to group them into the bins. During the move, we apply the operator specified as the second argument to the `groupby` operator. The supported operators for this work are: `min`, `max`, `avg`, and `count`.

```
(->>
    (metric "cpu_usage")
    (tag "hostname" '("host_0"))
    (where (and (< #TS 1451621760000000000) (> #V 95))))
```

Listing 8: Simple example using where clause.

The final example 8 shows the FrogFishDB equivalence of the traditional SQL `WHERE` clause. The `where` operator on the last line creates a filter where the timestamp has to be lower than 1451621760000000000 (Fri Jan 01 2016 04:16:00 GMT+0000) and the value has to be higher than 95. The two keywords shown here are `#TS` and `#V` for timestamp and value, respectively.

In summary, FrogFishDB supports the following operations:

Table 4.1 shows the different operations supported by FrogFishDB. Future work would be to support nested queries and to be able to use join operations between those queries. Join operations would enable users to, for example, do causality analysis on multiple machines. Imagine a user is investigating an issue where the 99th percentile latency of an RPC endpoint higher than expected. A cause for this could be that the machine is overloaded

Name	Code	Parameters
Query start	`->>`	Other expressions describing the query.
Metric selector	`metric`	Name of the metric which is to be selected.
Tag selector	`tag`	Tag name and list of values for a given tag.
Range selector	`range`	Start and end timestamp of the range of time which is to be selected using the query.
Group by operator	`groupby`	Size in time of the bins and an operation to execute on each of those bins.
Filter	`where`	Function with two parameters which acts as a filter on a per-datapoint basis.

Table 4.1: Supported query operations.

and is not able to respond to requests within an expected range of time. Being able to execute a query which combines the times at which the latency is very high, together with a metric detailing the load on a machine, could signal a causal relationship.

4.4.3 Query planner

The query planner is kept as simple as possible. When the user submits a query, the planner searches for the corresponding indexing structures. One limitation of the current implementation is that we only support a lookup for a single timeseries, meaning that no join operations are available. When the corresponding indexing structure is found, it is queried for the range of time specified by the user. This lookup in the indexing structure results in a list of offsets in the data file where the relevant memtables are stored.

4.4.4 Summary

In this section we discussed the query language we designed for FrogFishDB. We first discussed the problem with existing query languages and why timeseries databases have previously opted to create their own. We found that the traditional SQL query language did not suffice in providing the expressive tools necesarry for engineers to create timeseries queries. SQL often required nested queries in order to create filters which are suited towards a timeseries workload. An example we provided detailed how two queries are required in order to filter for a set of values where we wish to select the latest ones.

We followed this discussion by detailing our own query language. Created out of a necesity for something small and simple to implement, but with adequate expressive capabilities. We chose to base it of of the LISP language. We create queries by selecting a timeseries and follow this up by *filter* or *group by* statements.

Finally we detailed the query planner. This planner does not employ any further optimizations, and is kept as simple as possible. First the query is matched to the correct timeseries, followed by a lookup into the indexing structure, searching for which memtables have to be read. After reading the memtables to memory, we execute any post processing operations if they are defined.

4.5 Optimization opportunity: Multithreading

To extract more performance out of our machine, we turn towards multithreading. The key factor enabling multithreading is that timeseries on their own are isolated. Compared to traditional databases where multiple clients can issue writes to the same table or row, we consider timeseries data to come from a single source per timeseries. For example, when recording CPU temperatures, there is no reason for two distinct nodes to ingest data into the same timeseries.

To this end, we can draw inspiration from MICA (121). Here Lim et al. present a key-value database, where one of the contributions is using partitions across different cores. Each core hosts a separate partition of the key space, and applications that interact with the database determine the core with which to communicate by hashing the key value. Each core listens to a different port, each set to a base value + core number, i.e., assuming base port value 8000, core 0 would host port 8000, core 1 would host 8001, etc.

For FrogFishDB, we propose a similar setup. Using the tokens we defined in section 4.2.2, a client can determine the correct port offset by calculating the remainder between the index value stored in the token and the number of cores. The aim of this method is also to spread the load equally across the different cores. In section 2.4, we discussed how context switches are detrimental to performance when using flash storage, as the runtime costs for a context switch are similar to the latency of a read or write operation. To this end, the database would employ a *thread-per-core* model (122). ScyllaDB (123) is another example of a database using the shared-nothing-architecture of a thread-per-core threading model to achieve both better throughput (2x-5x) and 99th percentile latency (124).

Figure 4.15 shows four clients sending timeseries data to FrogFishDB, which runs on four cores. The index value in the token is mapped to the index value used internally for referencing the different timeseries structures (i.e., the indexing structure and memtable). As each core handles its subset of the different timeseries, they also need their own files.

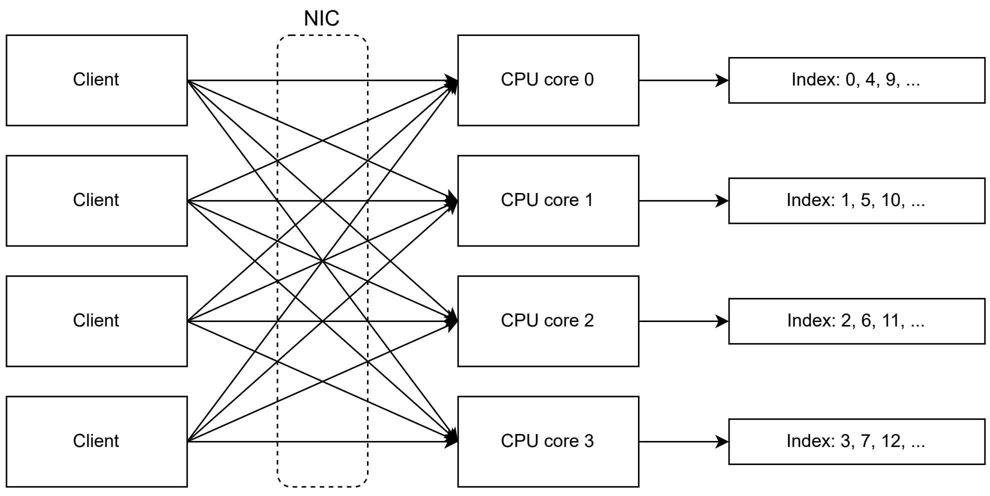


Figure 4.15: Ingestion in multithreaded context for FrogFishDB.

This allows cores to issue appends to the log files without having to use locks which can hamper performance.

The management port is hosted on only core 0. Upon receiving requests to register a tagset, it is checked against the local cache on core 0. If no matching tagset is found, a new index is created by passing a message to the next available core, and the index value is then wrapped into a token, which is sent back to the application. We again assume that there is little traffic on the management compared to the ingestion ports.

For queries, we need a different method for accessing the different timeseries. If a query combines multiple timeseries, for example, during a join operation, we cannot simply reference a single core that contains the data. In this case, we submit the query through the management port. Upon receiving the query, core 0 will issue reads to the other cores, which contain the timeseries associated with the tag values requests. When the other cores receive these read requests, they read the data into memory and reply to core 0 with a pointer which points to the data.

Figure 4.16 shows how a query is handled using a thread-per-core multithreading model. A query is submitted through the management port to core 0. Core 0 processes the query and determines which cores need to be issued a read request as they handle timeseries data relevant to the query. All cores that receive the read request start by reading the queried data to memory and, when finished, notify core 0 by sending a message containing a pointer to the data that has been read.

For this thesis, we have not been able to implement this. However, this model will be used in future work.

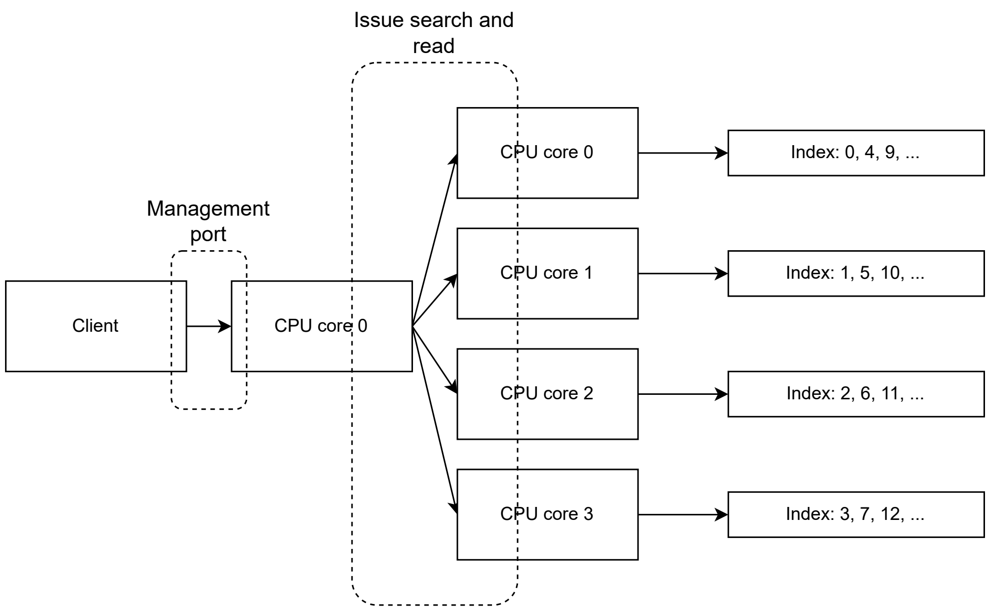


Figure 4.16: Queries in multithreaded context for FrogFishDB.

5

Evaluation of FrogFishDB

This section will evaluate FrogFishDB and compare it to existing timeseries databases. This is done to provide an answer to RQ3. To answer this question, we will evaluate FrogFishDB in two stages. First, we will examine the ingestion bandwidth. Secondly, we will analyze the query performance. We compare FrogFishDB against multiple state-of-the-practice timeseries databases. Before discussing the performance of FrogFishDB, we start with our evaluation plan, followed by a discussion of the benchmarking tools used.

5.1 Evaluation plan

This section will discuss the setup and background for the presented benchmarks. As we intend to test the various components of FrogFishDB, we require a detailed evaluation plan to isolate and test the various components. We split the evaluation into two distinct sections. First, we examine the ingestion bandwidth, which, as established earlier, is the main focus of this thesis. Second, we examine the query performance of FrogFishDB.

When picking timeseries databases to compare against, we attempted to select based on the state-of-the-practice and the state-of-the-art. The state-of-the-practice we can determine using the DB-Engine ranking (99). From this list, we selected InfluxDB as the state-of-the-practice. Not only does it carry the highest score by a significant margin, but it has also previously been cited across different academic works (31, 125, 126). For the state-of-the-art, we select QuestDB and ClickhouseDB. In chapter 3, we discussed QuestDB and how it achieves ingestion bandwidth by writing directly through an MMAP'ed file. Clickhouse is a relatively new timeseries database. However, previous work has shown significant ingestion bandwidth when compared to existing timeseries databases (23, 24, 26). We use the same timeseries databases for our query benchmarks.

5.1.1 Ingestion bandwidth

The ingestion bandwidth is the primary concern of this thesis. First, we have to establish that our indexing structure can no longer form a bottleneck for the ingestion bandwidth of FrogFishDB. We test this by examining the insertion performance of TimeTree under different fanout configurations. When pursuing vertical scalability, we need to ensure that we are bottlenecked not by software but by hardware. Testing TimeTree is the first step in determining this. As TimeTree contains offset values, each insertion represents a memtable being written to storage.

After determining if the indexing structure forms a bottleneck to the ingestion bandwidth of FrogFishDB, we need to determine the upper bound in terms of storage bandwidth. We need to determine the storage performance to understand what we can expect as an upper bound. We measure the performance of write operations issues through the `io_uring` kernel interface. All measurements are made using direct IO.

With the upper limits in terms of bandwidth in hand, we determine the difference between the performance of the InfluxDB and the ingestion protocol we described in section 4.2.2. Finally, we move to the ingestion performance of FrogFishDB. We use benchmarking tools discussed in section 5.1.4 to determine the overall ingestion performance of FrogFishDB and the three TSDBs mentioned earlier. We are interested in the total *points per second* ingestion performance.

We compare the bandwidth across two different setups. First, we examine the performance when using a single client. We compare the performance of using a single client against the aforementioned timeseries databases. After determining the ingestion performance in a broader context, we will zoom in on the performance of FrogFishDB. We examine the networking overhead by measuring a no-op scenario where we do not ingest any timeseries data but return immediately. After which, we determine the effects of using differently sized memtables. Increasing the memtable size should reduce the amount of write operations to storage. Finally, we examine the memory usage of FrogFishDB.

The second setup involves more than one client. We examine the ingestion bandwidth of multiple clients and attempt to uncover if increasing the number of clients also increases our performance. We start with two clients and work our way up to sixteen clients.

5.1.2 Query performance

As query performance is not this thesis's main focus, we will keep our evaluation brief. First, we evaluate the query performance of TimeTree to determine the impact of differently-sized range queries.

Secondly, We evaluate five different queries, each testing a different aspect of the query language. We start with a simple query that selects all points available for a given time-series. This query is meant to retrieve all data from a specific timeseries. The second query retrieves data for a range of time. The third query demonstrates a filter expression. The fourth query selects all data from a timeseries and executes a group by expression. The fifth and final query again executes a group by expression, but now on a range of time.

These five queries are meant to demonstrate the different features offered by the query language we created. The filter and group by expressions allow users to solve practical problems using the query language. The filter clause allows for finding specific entries, while the group by expression allows one to summarize data quickly.

5.1.3 Configuration

All FrogFishDB benchmarks reported in this thesis were conducted using the same physical machine.

- A 20-core 2.40GHz Intel Xeon Silver 4210R CPU (127). The physical machine provides two sockets, each socket is populated by a CPU of this type, and the sockets are connected in a NUMA mode.
- 256GB of DDR4 DRAM
- 960GB NVMe SSD, model Western Digital UltraStar DC SN540

All benchmarks are run on a virtual machine. This virtual machine is configured to only run in a single NUMA domain, making the benchmarks more reliable as we do not suffer from the overhead of crossing NUMA domains. The configuration is thus:

- 10-core 2.40BHz Intel Xeon Silver 4210R. One socket and thus one NUMA domain.
- 64GB of DDR4 DRAM
- 960GB NVMe SSD, formatted to the EXT4 filesystem (128)

The benchmarks are run using QEMU 6.0.0 (129) with KVM enabled. The image run is NixOS running kernel version 5.15.90. The image was run from the NVMe drive. The VM was configured to only run on a single NUMA node using `numactl`, a Linux tool for specifying specific NUMA scheduling or memory placement policies. These policies are subsequently inherited by child processes.

We pin each process to a core to provide the best performance for FrogFishDB. FrogFishDB and the clients are all single-threaded applications. This allows us to pin the database to core 2 and all clients on the upper cores. We keep cores 0 and 1 isolated for the kernel.

5.1.4 Benchmarking tools

To perform our benchmarks, we need tools that will allow us to measure and repeat experiments. The first tool we use is one with which we can benchmark small pieces of code. We use Nanobench (130) to benchmark the performance of our indexing structure. Nanobench allows us to write repeatable experiments for small pieces of code. This library comes with integrated reporting functionality, which gives us insight into micro-architectural level information such as the *instructions-per-cycle* (IPC).

The second tool is one which we use to measure the ingestion performance of the database. This tool is called *Time Series Benchmark Suite* (131), henceforth referred to as TSBS. Originally developed by the engineers of TimescaleDB (132), it supports a large number of existing databases and provides tools for both data generation as well as data ingestion. The data generation tool allows us to generate data using a set of parameters. These parameters include the following:

- The RNG seed for the generation function
- The number of simulated devices
- The start and end timestamps of the data
- The amount of time between each simulated reading

The data generator can operate in three different modes, *cpu-only*, *devops*, and *iot*. In the first mode, the generator generates a homogeneous list of metric values, namely the ten usage statistics of a simulated cpu such as `usage_user` and `usage_nice`. This data is generated together with ten tags which indicate from which simulated device the data originates. Ten statistical values per device mean that the number of timeseries is equal to 10 times the number of simulated devices. The second mode is used to generate a mixed group of data. Meaning that the number of tags and metrics per reading changes. Furthermore, the `cpu-only` dataset only contains integer values, while the `devops` dataset also contains floating point values. The third and final mode, `iot`, simulates a set of trucks from a fictional logistics company. The simulated data pertains to diagnostic data and metric data of the trucks. The main difference between the `iot` and `devops` is that the trucks are simulated to lose connection at random intervals. This means that data arrives possibly in batches or out-of-order.

For FrogFishDB, we assume that data arrives in order, and we only support integer values. This means that for benchmarking the ingestion performance, we can only use the `cpu-only` mode of the data generator. As our ingestion pipeline differs significantly from the existing model, such as the one presented by InfluxDB, we must prepare the data before starting our benchmarks. To prepare our data, we use a Python script to read the InfluxDB line protocol files generated by TSBS, register the timeseries, and create batches. Our testing clients then use these batches to benchmark the performance of FrogFishDB. We chose to use InfluxDB line protocol data as the base for our testing because those files can also be used by QuestDB and InfluxDB. Clickhouse uses a different file format, however, having to store the data in only three different formats (Influx, Clickhouse, and FrogFishDB), lessens the burden on the storage medium we use to store the raw data. Otherwise, we would have to store the testing data four times.

For this thesis, we create three differently-sized collections of timeseries data. All parameters are kept the same between the three configurations except for the scale parameter and the start and end timestamps. The scale parameter represents the number of simulated devices. We generate data with scales of 32, 320, and 3200, because these scales are divisible by 2, 4, 8, and 16, which is the number of different nodes we test and we need to make sure to balance the load across the different nodes. We want to observe the effects of increasing the number of timeseries. We change the start and end timestamp between the three different configurations to limit the amount of storage required. For the 32 scale version we generate 6 months of data, for the 320 scale version we generate 3 months of data, and finally for the 3200 scale version we generate 1 month of data. We decrease the number of months for which we generate data in order to decrease the storage requirements during benchmarking. As the scale parameter indicates the amount of nodes represented by the data and that recording represents 10 datapoints, we can calculate the size of each of the scales. The offset between recordings is kept at 10 seconds, meaning 6 measurements per minute, 360 per hour, 8640 measurements per node per day. Assuming a scale of 32, this means that we ingest 180 days worth of data, or 1.555.200 measurements which is equal to 15.552.000 data points as each measurement records 10 data points.

To measure memory usage, we use Valgrind (133). This tool will intercept any allocations and deallocations of memory in our code, allowing us to gain insight into the memory usage patterns. We test different scales of incoming timeseries. We hypothesize that increasing the number of timeseries will increase memory usage.

Storage is measured using `fio` (134). `fio` allows us to define different workloads to be tested against storage. Created as a swiss-army knife for storage benchmarks, `fio` generates IO workloads according to the configurations we provide. We require different workload definitions to determine the upper bound for differently sized memtables.

To increase the validity of our tests, we use the *Nix* package manager (135) to configure our environment. Nix is described as a *purely functional package manager*, meaning that packages are treated like variables in a functional programming language. Packages are built by functions that have no side effects and are immutable after creation. This allows us to reproduce exact versions of packages as we use the same language to describe our build environment and dependencies. Nix stores packages in a so-called *nix store*. Packages are identified using their hash values to avoid conflicts and allow for multiple versions. When a package is requested, a symbolic link is made between the store and the environment. Aside from package management, we can take Nix a step further. Nix allows us to define entire images of operating systems, meaning we can define an environment in which to conduct our experiments, which is then completely reproducible.

Name	Version	Notes
QEMU	6.0.0	-
fio	3.33	-
valgrind	3.20.0	-
Nix	2.13.2	-
TSBS	commit: *bcc00137d*	Appendix contains patch to fix clickhouse testing
InfluxDB	1.10	-
QuestDB	7.2.1	-
Clickhouse	23.3.5.9	-

Table 5.1: Versions of software used.

Table 5.1 lists all versions used of external software in this thesis.

5.2 Ingestion performance

We will start by examining the insertion performance of the indexing structure. We isolate the indexing structure to determine if it forms a bottleneck for the rest of FrogFishDB. We test it in isolation using micro benchmarks and we test it indirectly by measuring

FrogFishDB as a whole. After measuring TimeTree we investigate the upper bound performance of our storage through `fio`. Measuring FrogFishDB will be done in two stages. The first will measure the TimeTree and the overhead of processing the tokens. The second stage will be measuring FrogFishDB. We will compare the performance of FrogFishDB against existing databases and a no-op version of FrogFishDB.

5.2.1 TimeTree

In this section, we will examine the insertion performance of TimeTree. Starting with an examination of different fanout degrees. Starting a fanout of 2, we increment in steps of 20 up to a degree of 1022 and measure the number of insertions per second. We start at 2 a fanout of 2 as results in a TimeTree which funtions like a balanced binary search tree. We chose the upper limit of 1022 as this would make each internal node span several memory pages and we wish to observe the effects of crossing memory page boundaries.

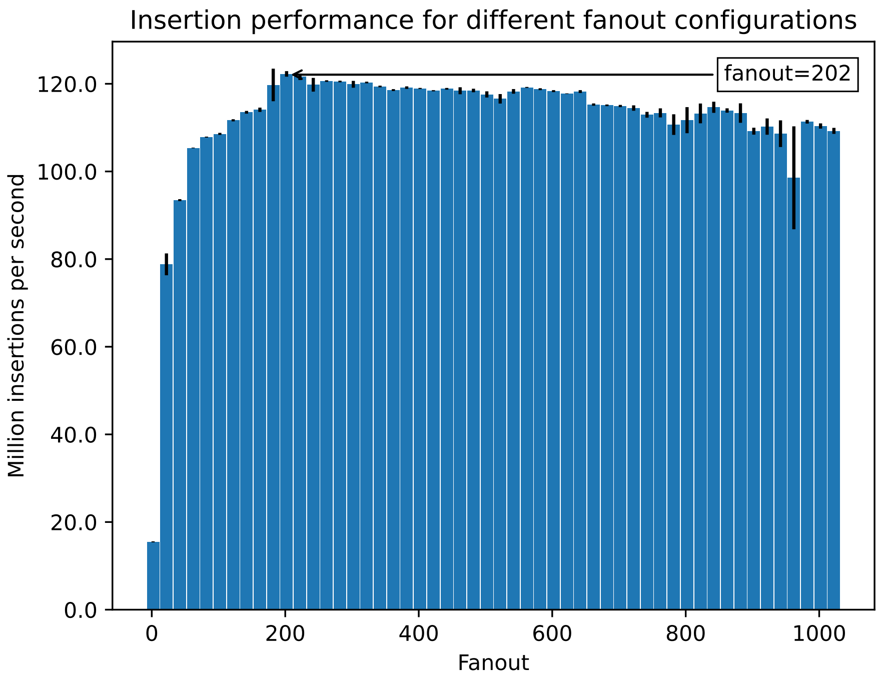


Figure 5.1: Insertions per second for the TimeTree using different fanout configurations.

Figure 5.1 shows the insertion performance for different fanout configurations of the TimeTree. Remember that each insertion represents the insertion of a start timestamp, end timestamp, and offset value, thus three 64bit integers. Here we show fanouts increasing in size on the x-axis and the number of insertions per second on the y-axis. What is interesting to note is that the performance increases rapidly towards a maximum and then slowly tapers off while also increasing in variance. The maximum we recorded we found using a fanout of 202. Using the size of the data points we store, i.e., the start time, end time, and offset value, we find that this is slightly less than the size of a single memory page (4096 bytes). This indicates that performance is bound to the performance of a linear search for the child pointer. We hypothesize that this performance tapers off as we have to do address translations when searching buffers that are larger than a single memory page.

We can calculate the theoretical maximum ingestion bandwidth for a single timeseries. Consider a TimeTree with a fanout of 202 and a memtable of 2MB. In figure 5.1 we can observe that with a TimeTree with fanout 202 we can insert 120 million entries per second. Remeber that each insertion into the TimeTree represents flushing a memtable to storage. If we multiple the rate of 120 million entries per second with the size of the memtable we obtain the theoretical rate. $120 \times 10^6 \times 2\text{MB} = 240\text{TBs}^{-1}$. This rate holds for a single timeseries, however, this rate exceeds the memory bandwidth of the system (127), thus we will never observe this in practice.

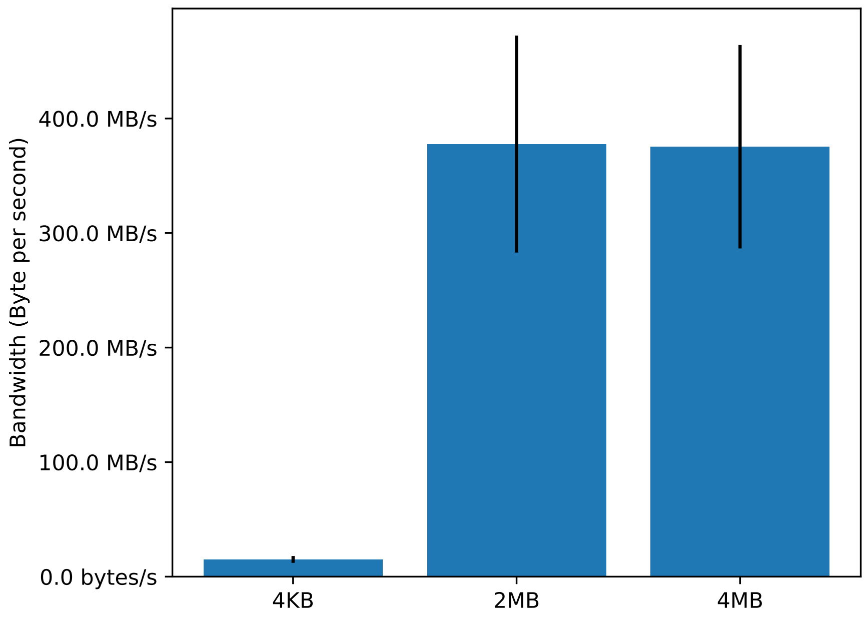


Figure 5.2: Maximum bandwidth for 4KB, 2MB, and 4MB sequential writes.

To determine the maximum achievable write bandwidth of the underlying storage, we used `fio`. Figure 5.2 shows the maximum bandwidth we can achieve. This performance was recorded using a queue depth of 1 and the `io_uring` kernel interface. We chose to investigate a queue depth of 1 in order to simulate the write bandwidth for a single timeseries when the rest of the queue is used by other timeseries. For example, imagine a queue with depth 64 and other timeseries take up 63 slots, then the bandwith remaining will be a worst case. As FrogFishDB uses direct IO, so does our test. We chose these three block sizes as they are the same experiment in section 5.2.2.4, where we test differently sized memtables. As each memtable counts as a single write, we test the various sizes. On the y-axis, we show the measured bandwidth in megabytes per second. While 2MB and 4MB size writes perform identically, 4KB writes are severely lacking in performance.

The previous two benchmarks have shown us the following: the indexing structure cannot form a realistic bottleneck for the timeseries database and a larger memtable will have better performance when flushed to storage. In the first benchmark we showed that TimeTree is capable of 120 million insertions per second, which is enough to fully saturate the memory bandwidth of the system if those insertions were accompanied by actual memtables being flushed to storage. In the second benchmark we showed that chosing a larger memtable improves performance when flushing to storage. As the memtable is written as one unit, we tested by writing blocks of different sizes to storage, thus a block of 4KB represents a memtable of 4KB.

5.2.2 FrogFishDB

In this section, we measure the performance of FrogFishDB. We start by quantifying the overhead of the InfluxDB line protocol compared to the ingestion protocol described in section 4.2.2. We then move to an evaluation of the ingestion performance of the database. We compare FrogFishDB's ingestion bandwidth with three other databases' ingestion bandwidth. We also evaluate the memory usage, different sizes for the memtable, and the upper bound performance using a no-op version of the database.

5.2.2.1 Token overhead

We compare the overhead between parsing the InfluxDB line protocol and mapping directly between the index value stored in a token. In this microbenchmark, we run two tests. In the first test, we generate an InfluxDB line protocol example, which we then parse and map to the pointer meant to point to the memtable and indexing data structure. In the second test, we forgo parsing the line protocol and map directly between the index value stored in a token and the pointer. This experiment is meant to isolate the overhead we avoid using the protocol described in section 4.2.2.

We found that when parsing the InfluxDB line protocol, we achieved a rate of 7,600 operations per second while parsing the index value from the token allowed for 5.6 million operations per second. This means that if we encoded a single timestamp-value pair per operation, we would be able to ingest 7,600 pairs per second using the InfluxDB line protocol, and 5.6 million pairs per second using our protocol. The bottleneck for parsing the Influx line protocol comes from parsing the string and converting the resulting data into the canonical name. The big difference between the two protocols, is that with our protocol design, we can directly use the index value stored in the token.

The only bottleneck remaining in our design is the performance of the hashmap. For this experiment we used the hashmap provided by the C++ language.

5.2.2.2 General ingestion performance

In this section, we will compare the ingestion performance of FrogFishDB with existing timeseries databases. Unless stated otherwise, all data collected for this section uses insertion batches of 10,000 values. In section 5.2.1, we showed that the best ingestion performance for the TimeTree occurs when using a fanout degree of 202. This is why, unless stated otherwise, we use 202 as the fanout degree for all subsequent FrogFishDB experiments. The size of the memtable is configured to be 2MB as a balance between memory usage and performance. The other databases are used with default configuration. In this section we are interested in the number of timeseries datapoints we can ingest per second. Thus, each datapoint refers to a timestamp-value pair.

Figure 5.3 compares FrogFishDB against three other timeseries databases: InfluxDB, QuestDB, and Clickhouse. On the y-axis, we show the points per second ingestion performance. For each database, we show three different scales of ingestion. This indicates the number of timeseries ingested. The results shown in figure 5.3 are all gathered using a single client. This figure shows a boxplot per database and per scale value described in section 5.1.4, where we show the number of points per second ingested on the y-axis, higher is better. From this figure, we notice two facts. The first fact is that FrogFishDB has a low variance for ingestion performance. The second fact is that Clickhouse shows the most stable performance when increasing the number scale.

To determine the efficiency of our ingestion method, we also examine a version of our database where we only measure the network performance. This no-op version shows the

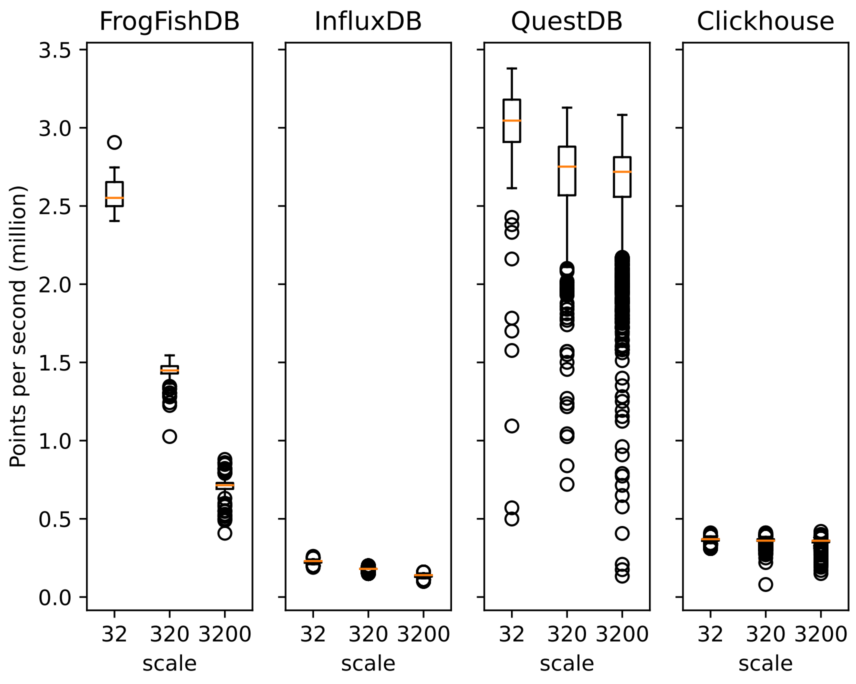


Figure 5.3: Ingestion performance of different timeseries databases. The y-axis represents the points per second ingested by the different databases.

theoretical upper bound of the ingestion performance. In this version, the database immediately replies with a confirmation of the ingestion, instead of writing to the memtable.

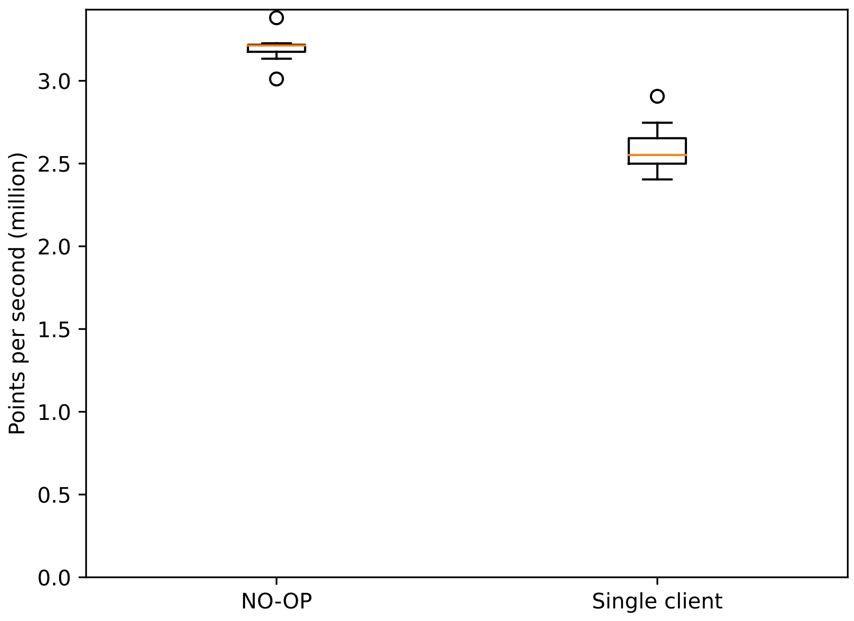


Figure 5.4: Ingestion performance compared between a no-op version and normal FrogFishDB.

Figure 5.4 shows a boxplot comparing the no-op version and the normal version of FrogFishDB. We compare the ingestion performance using 32 timeseries. On the y-axis, we again show the ingestion performance in points per second. From this figure, we can determine that we are not at the upper bound of performance. The single-client performance maintains an average ingestion bandwidth of 2.5 million points per second, while the no-op version achieves an average of 3.2 million. This means there is an overhead of 600 thousand points per second during the ingestion phase, or 25.5%.

5.2.2.3 Memory usage

In section 4.3.2, we covered the memory usage of the indexing structure. For ingestion, the database itself also allocates memory for things like the memtable. FrogFishDB allocates two memtables for each timeseries, one for ingestion and one as a buffer for flushing to storage. This means that our memory usage is directly proportional to the number of timeseries we are ingesting, i.e., $2 \times N \times B$, where N is the number of timeseries being ingested. B represents the size of the memtable in bytes. For this section, we measured the memory usage with memtables of size 2MB.

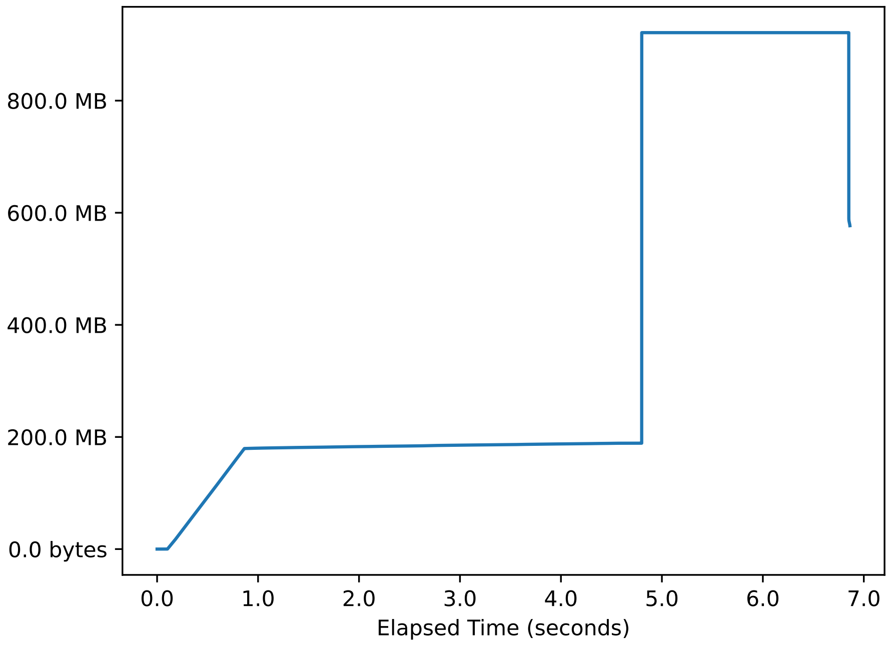


Figure 5.5: Memory usage of FrogFishDB when ingesting at a scale of 32.

Figure 5.5 shows the memory usage when running the ingestion benchmark at a scale of 32. On the x-axis, we show the runtime of the program, starting on the left and ending on the right. The y-axis denotes the memory usage. Note that this figure is the amount of allocated virtual memory and does not reflect the amount of bytes stored on the heap. From this figure, we observe two distinct sections, The startup phase, ranging from 0 to 4.8 seconds, and the ingestion phase, starting at 5 seconds and running until the end. The memory usage grows during the startup phase as all timeseries metadata is reloaded from the log files. During the ingestion phase, the allocated amount spikes as the data structures are created and used during ingestion, i.e., the indexing structure and memtable.

Figures 5.6 and 5.7 show the memory usage for the scales 320 and 3200, respectively. Note that the scale of the y-axis of figure 5.7 is higher than in figure 5.6. In these figures, we note the same memory pattern as we observed in figure 5.5. A startup phase where the metadata is reloaded from the log files and an ingestion phase where memory usage spikes.

From all of the memory usage graphs above, we note that memory usage is constant

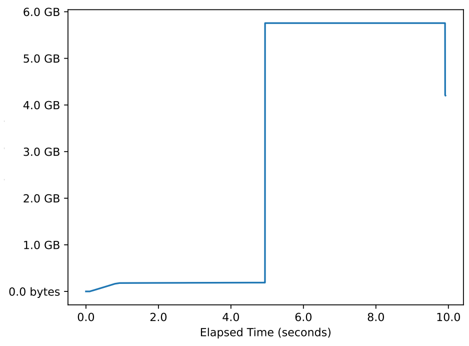


Figure 5.6: Memory usage for 320 timeseries.

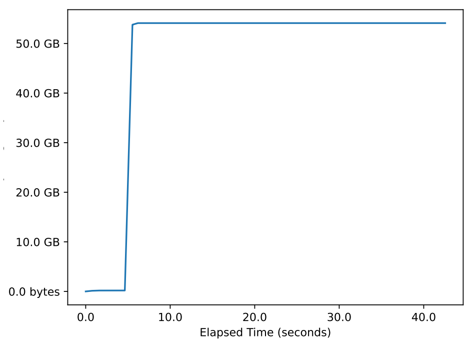


Figure 5.7: Memory usage for 3200 timeseries.

during ingestion. This is because we allocate a memtable for each of the timeseries when the first timeseries data arrives, after which we do not need to allocate any more. We do observe very high memory usage for the 3200 scale ingestion, namely 54GB.

5.2.2.4 Memtable size

Increasing the size of the memtable will decrease the frequency at which we need to flush data to storage. A larger memtable will be able to hold more data points. Thus we hypothesize that increasing the size of the memtables will increase the ingestion bandwidth.

Figure 5.8 shows three different configurations for the memtable size. We chose 4KB, 2MB, and 4MB as testing values. The first two represent the size of one normal memory page and one huge memory page. The third size represents multiple memory pages and is chosen to show the possible effect of pooling multiple pages. We again show a box plot where the y-axis represents the points per second ingested in millions. Figure 5.8 shows that 4KB memtables have decreased ingestion performance compared to 2MB and 4MB. However, the difference between 2MB and 4MB is negligible. From this we can conclude that using memtables of 2MB will provide a better tradeoff in terms of ingestion bandwidth and memory usage. The memtables will be flushed less frequently compared to memtables which are of size 4KB, as the fill up less quickly. Using memtables of size 2MB decreases memory usage compared to using memtables of size 4MB, as each timeseries allocates its own memtable, thus increasing the size of the memtable increases the overhead of each timeseries.

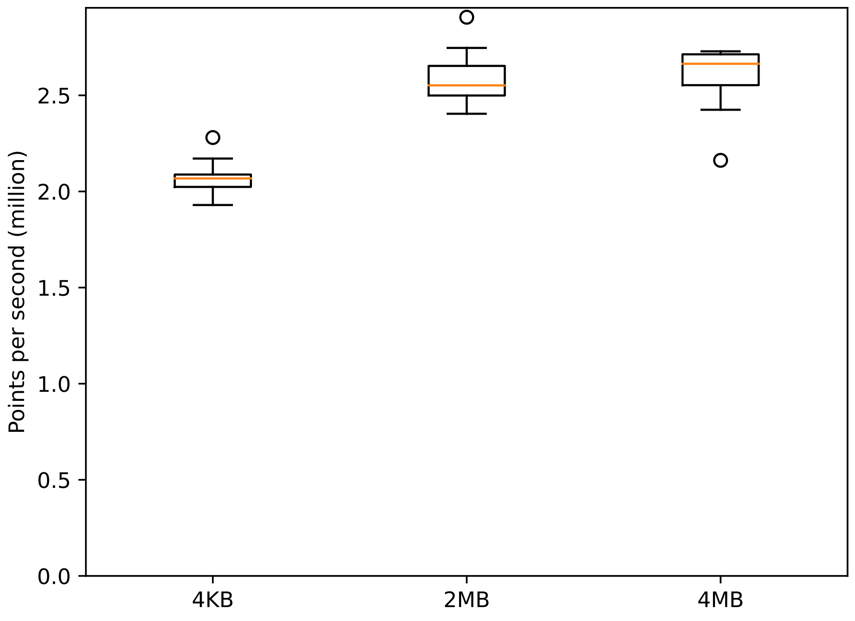


Figure 5.8: Ingestion performance for differently sized memtables.

5.2.2.5 Multiple clients

In this section, we will test the use of multiple clients. We hypothesize that utilizing multiple clients hamper performance as we only use a single thread to process each incoming batch. For each of the databases, we test the use of multiple clients. We start by examining two clients and work our way up to sixteen clients. We compare the performance against InfluxDB, QuestDB, and Clickhouse. We use the same dataset and configuration we used in section 5.2.2.2, for all databases tested in this section. First, we examine the performance of two clients. Two clients read the preprocessed data and send it to the database.

Figure 5.9 shows two clients ingesting data into FrogFishDB, InfluxDB, QuestDB, and Clickhouse. We observe a similar pattern compared to figure 5.3. FrogFishDB shows lower variance. However, performance degrades when increasing the number of timeseries. We show the number of data points ingested per second on the y-axis. FrogFishDB reaches an average of 5.8 million data points per second for a scale of 32 and 4.5 million points per second for scale 320, an increase of 123% and 200% respectively compared to a single client. When dealing with a scale of 3200, the ingestion bandwidth of QuestDB is 119% higher than that of FrogFishDB. However, we observe a high variance in rate compared to

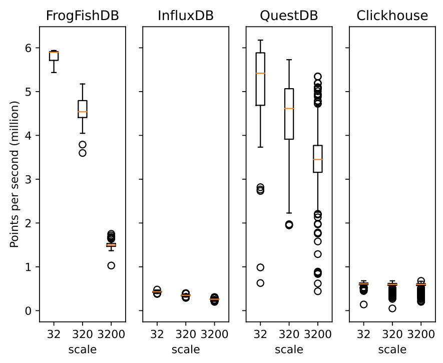


Figure 5.9: Ingestion performance of two clients ingesting data into a single database.

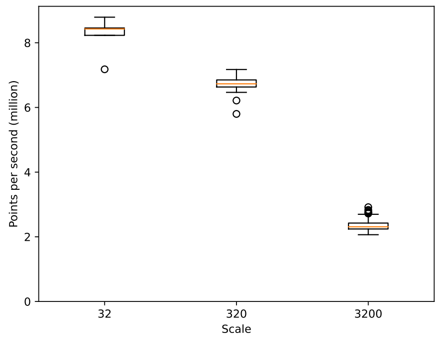


Figure 5.10: No-op performance of two clients.

FrogFishDB. In figure 5.10, we show the no-op performance of two clients. From figure 5.10, we can determine that there is a significant overhead involved when using two clients, as the performance observed from two clients using the 32 scale dataset differs by 45%.

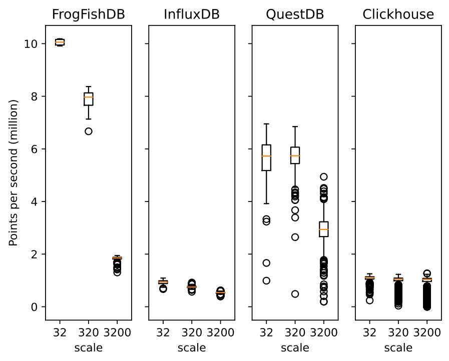


Figure 5.11: Ingestion performance of four clients ingesting data into a single database.

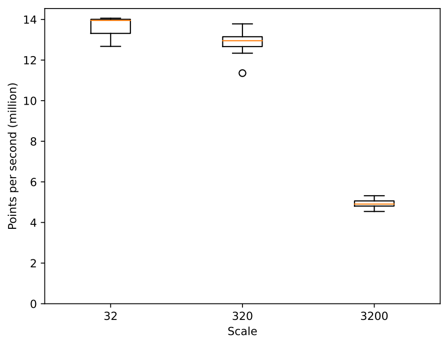


Figure 5.12: No-op performance of four clients.

Figure 5.11 shows the performance of using four clients. Compared to figure 5.9, the performance of scales 32 and 320 increase with 123% and 200% respectively, while the performance for scale 3200 only increases 113%. For FrogFishDB, we do observe that the variance remains consistent. In figure 5.12 we again observe a significant overhead, where the ingestion performance differs by 38% and 63% for scales 32 and 320, whereas the performance for scale 3200 differs by 150%. We can also compare the performance of QuestDB between figures 5.9 and 5.11. We observe that QuestDB slightly improves in performance, from an average of 5 million points per second to 6 million points per second, or 20%.

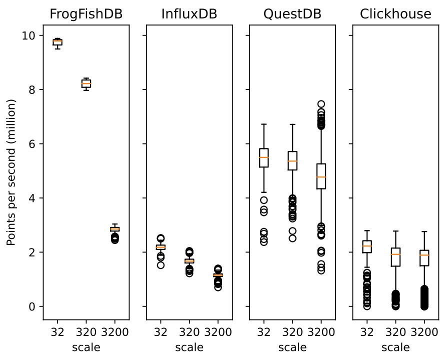


Figure 5.13: Ingestion performance of eight clients ingesting data into a single database.

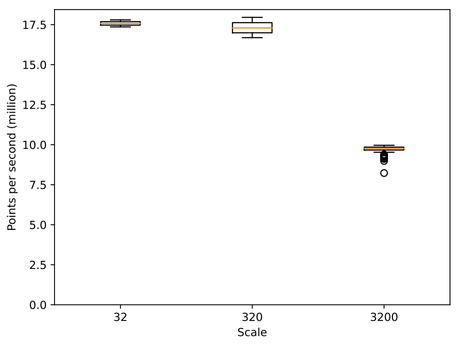


Figure 5.14: No-op performance of eight clients.

Figures 5.13 and 5.14 show the ingestion performance of eight clients. For FrogFishDB, ingestion performance is very similar to what we observed when using four clients for scales 32 and 320, differing -2% and 2%. Scale 3200 however, differs by 42%. This indicates that FrogFishDB requires more clients when ingesting a larger number of timeseries. The other databases only see a marginal improvement in bandwidth. However, the no-op performance shows an even greater disparity in performance, the differences increasing to 80%, 110%, and 257% for scales 32, 320, and 3200.

Finally, we observe the use of sixteen clients. Figures 5.15 and 5.16 show the performance of sixteen clients. From these figures, we observe a decrease in ingestion performance compared to figure 5.13 for FrogFishDB with regard to scales 32 (-18%) and 320 (-6%). However, this decrease is not observed for the 3200 scale benchmark, the bandwidth increases by 43%. The no-op performance demonstrated in figures 5.16 shows a similar pattern, where it is lower than in figure 5.14, but only for scales 32 (-5%) and 320 (-4%), scale 3200 differs by 40%. We hypothesize that the overhead of maintaining the network connections outweighs the performance benefits of employing multiple clients. Another hypothesis is that if FrogFishDB used multiple threads, this overhead could be alleviated, and we could observe greater scalability.

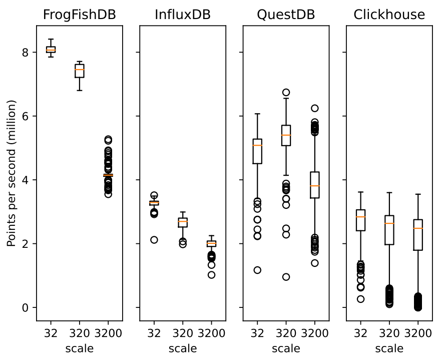


Figure 5.15: Ingestion performance of sixteen clients ingesting data into a single database.

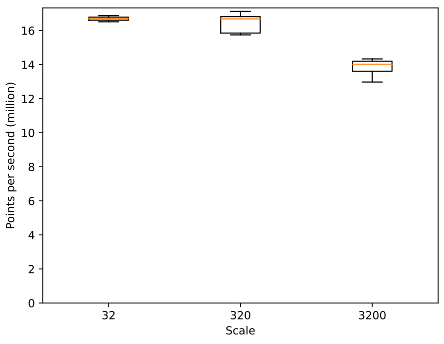


Figure 5.16: No-op performance of sixteen clients.

To better understand the load factor introduced by increasing the number of timeseries, we examine the latency of insertions. The clients record the latency between the moment they send the timeseries data to the database and the moment they receive a confirmation from the database about the insertion having been completed.

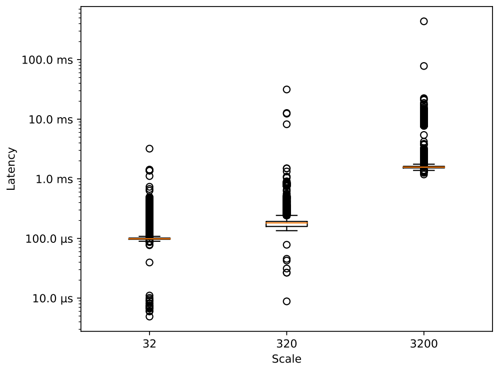


Figure 5.17: Insertion latency when using a single client.

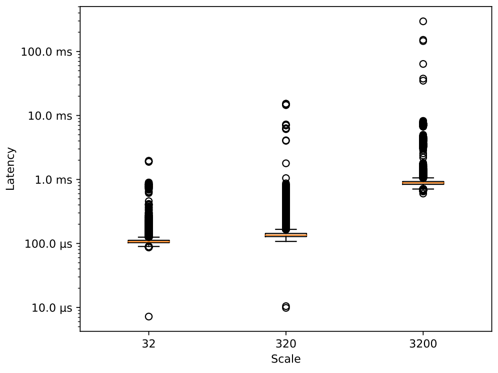


Figure 5.18: Insertion latency when using two clients.

Figures 5.17 and 5.18 show the insertion latency for one and two clients, respectively. We show boxplots for the three different scales. The y-axis denotes the latency in seconds and is displayed in log scale. These figures demonstrate that latency increases when increasing the number of timeseries, i.e., the scale factor. The latency remains constant when increasing the number of clients from one to two. We observe a more significant amount of outliers when going from a single client to two clients.

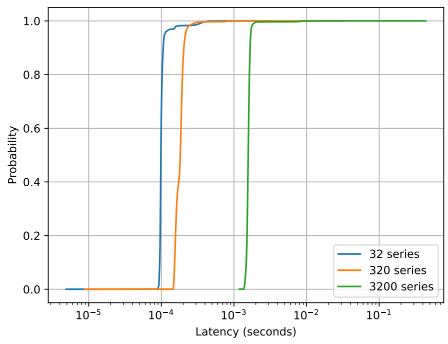


Figure 5.19: Cumulative distribution function of the insertion latency when using a single client.

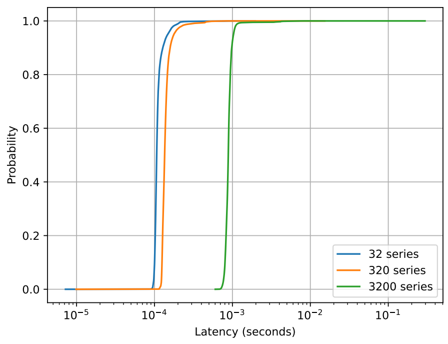


Figure 5.20: Cumulative distribution function of the insertion latency when using two clients.

To further demonstrate the insertion latency's tails, see figures 5.19 and 5.20. These are *cumulative distribution function* (CDF) graphs. These functions of x plot the probability of a latency figure occurring with a value lower than or equal to x. On the y-axis, we plot the probability, and on the x-axis, we plot the latency in seconds. Note that the x-axis is plotted in a logarithmic scale. From these figures, it is clear that increasing the number of timeseries will have a detrimental effect on the ingestion bandwidth of FrogFishDB. Between scales 32 and 3200 the latency increases by an order of magnitude. The average increases even though increasing the number of timeseries decreases the number of flushes to storage. Increasing the number of timeseries increases the number of allocated memtables and indexing structures, but as the amount of data points per batch remains stable, increasing the number of timeseries reduces the number of flushes.

Figures 5.21, 5.22, 5.23, and 5.24 show the latencies for four and eight clients. We observed similar performance to what we observed when using one or two clients. We observe that the latency increases when moving from 320 to 3200 is less when using eight clients. We hypothesize that this is due to the single-threaded nature of FrogFishDB. Each client is handled individually and sequentially. This means that the latency penalty of increasing the scale is hidden because a client might be later in line to be processed. We can also see that between four and eight clients, the variance in latency increases.

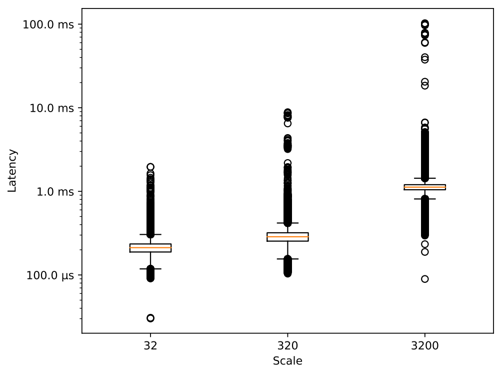


Figure 5.21: Insertion latency when using four clients.

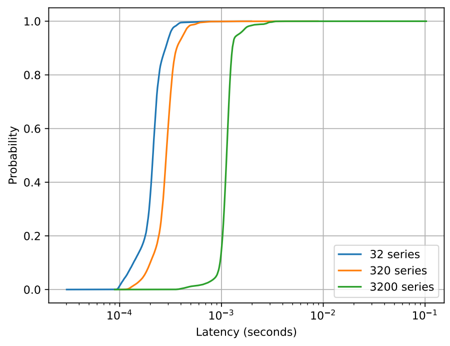


Figure 5.22: Cumulative distribution function of the insertion latency when using four clients.

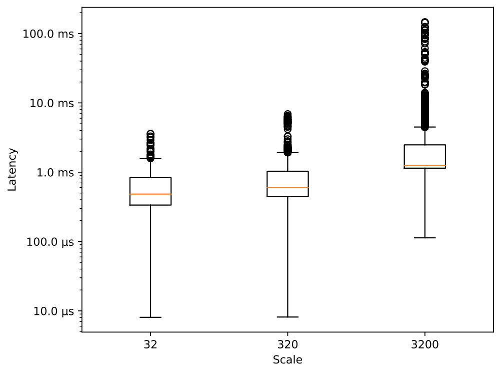


Figure 5.23: Insertion latency when using eight clients.

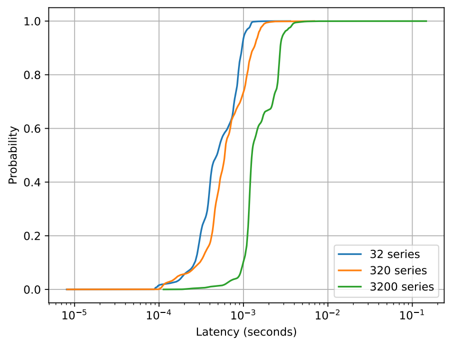


Figure 5.24: Cumulative distribution function of the insertion latency when using eight clients.

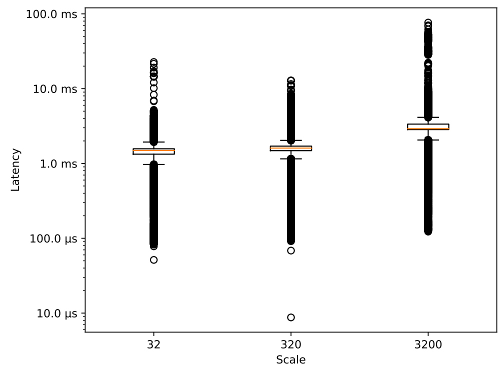


Figure 5.25: Insertion latency when using sixteen clients.

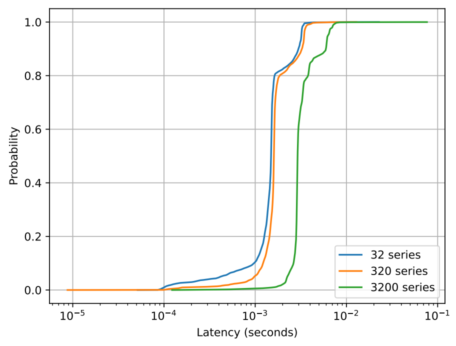


Figure 5.26: Cumulative distribution function of the insertion latency when using sixteen clients.

Finally, we observe the latency of sixteen clients in figures 5.25 and 5.26. We observe an increased variance in latencies. For scales 32 and 320, the variance spans 1.5 orders of magnitude.

5.2.2.6 Storage size

In the current implementation of FrogFishDB, we use a log to record insertions into the indexing structure. This log contains each entry inserted and is replayed on startup to recreate the indexing structures. This allows us to calculate the overhead of a single insertion into the indexing structure by taking the size of an entry in the log file. In the current implementation, we use log buffers of 512 bytes. The log statements consist of 3 values, the start timestamp, the end timestamp, and the offset into the data file. All three are 64 bits in size, or 8 bytes, totaling 192 bits, or 24 bytes. The log statement is 512 bytes, meaning that for each byte we write to the indexing structure, we write 22.3 bytes to storage.

5.2.3 Summary

In this section, we examined the ingestion performance of FrogFishDB. We first observed that the TimeTree can handle enough insertions to represent 240TB/s in write bandwidth. We found an optimal fanout configuration at a fanout of 202, for which TimeTree provided the best insertion performance. Next, we examined the bandwidth of the underlying storage layer in a worst-case scenario. We found that the flash SSD we used is capable of 380MB/s of bandwidth with sequential writes. This was followed by an examination of the ingestion bandwidth of FrogFishDB. We compared the performance of FrogFishDB against three other timeseries databases. We found that FrogFishDB does not offer the same ingestion bandwidth as QuestDB for a single client but provides better performance than InfluxDB and Clickhouse. When comparing against a no-op version of the database, we report a 25.5% overhead.

When ingesting data from multiple clients, we found that FrogFishDB was able to scale better in terms of points per second ingested. FrogFishDB increased in bandwidth with 123%, 200%, and 87% for scales 32, 320, and 3200 respectively when using two clients instead of one. With two clients, the performance was on par with that of QuestDB for scales 32 and 320. When ingesting from four clients, we observed 30% better ingestion bandwidth for 32 and 320 scale timeseries. A similar pattern was observed for eight and sixteen clients. We also observed that the no-op version demonstrated the existence of significant overhead for all of these multi-client tests.

Finally, we observed that a large number of timeseries entails a significant performance overhead. The latency figures show a large increase in latency when increasing the number of timeseries.

5.3 Query performance

We evaluate the performance using microbenchmarks to determine if the indexing data structure could become a bottleneck for scan operations. This is followed by an end-to-end examination where we investigate the performance of the database and attempt to quantify the impact of different query operations.

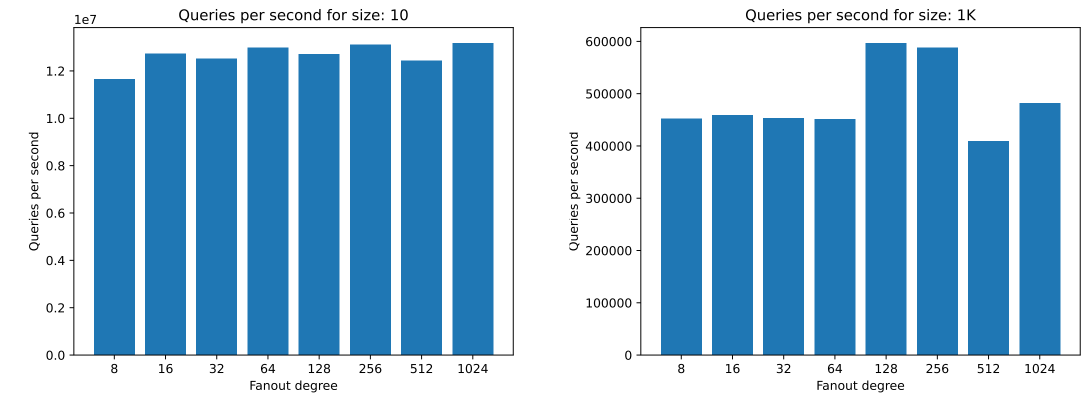
5.3.1 TimeTree

Before examining the full query performance, we first examine the lookup performance of the indexing structure. We take a variety of fanout configurations and determine the performance. The hypothesis is that increasing the fanout will improve our performance as we have to do less pointer chasing during lookups. For example, when we have a fanout of 64, we perform 64 comparisons before following a pointer to a lower level. In comparison, with a fanout of 8, we only do eight comparisons before following a pointer to a lower level. The hypothesis follows the idea that performing several sequential comparisons is more efficient than following a new pointer because of the effects of the branch predictor and the prefetching found in modern CPUs.

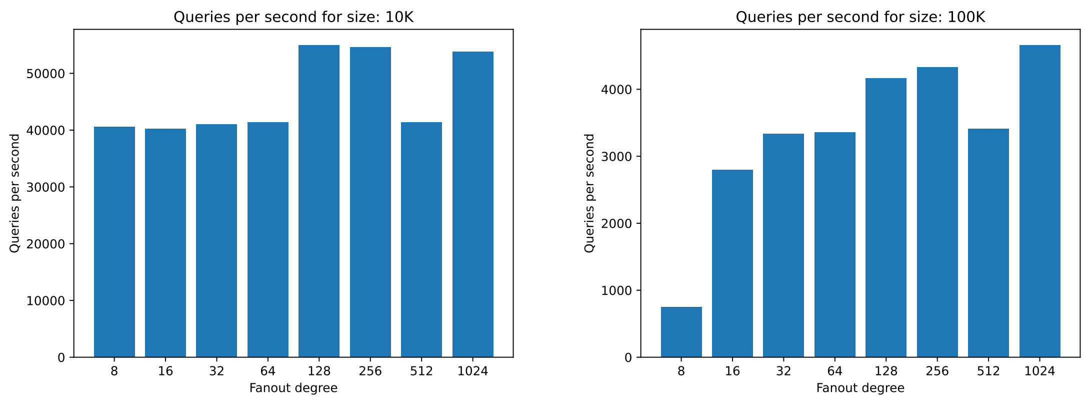
5.3.1.1 Query Performance

We start by examining the query performance of TimeTree. We measure five differently sized queries, ranging in size from 10 to 1 million. Note that this is meant to represent queries that range from 10 memtables to 1 million memtables. We test each query under different fanout configurations in an attempt to understand the performance characteristics and whether or not the indexing structure could form a bottleneck for query performance. To setup this experiment we first inserted 1 million entries, then we executed the queries where the start is at random points.

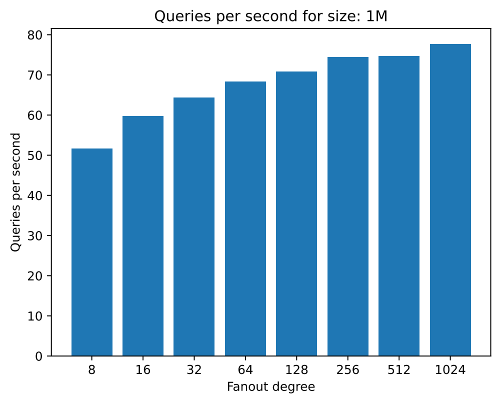
Figure 5.27 shows the query performance for different fanout configurations. On the x-axis, we show different fanout configurations, starting with a fanout of 8 and ending with a fanout of 1024. We test 5 different query sizes, increasing in size, per fanout configuration, starting at 10 and ending at 1 million. On the y-axis, we show the queries per second figure for each of the fanout degrees, more is better. We observe that choosing a higher fanout degree does not result in an immediate increase in performance for smaller queries. For queries of size 100k and higher, increasing the fanout increases performance. This can be explained by the fact that increasing the size of the query will increase the amount of execution time spend on collecting the values stored in the TimeTree. With lower fanout degrees, the trees also grow larger in height, leading to more pointer chasing.



(a)



(b)



(c)

Figure 5.27: Query performance for different fanout degrees.

5.3.2 FrogFishDB

In this section, we investigate the performance of queries submitted to FrogFishDB. The dataset we use for this experiment is again generated using TSBS. We generate data representing seven days, where each recording occurs at an interval of ten seconds, with a scale factor of 320. Each query is executed ten times to uncover any possible variance.

We test the following queries:

Q1 :

```
(->>
    (metric "usage_user")
    (tag "hostname" '("host_0")))
```

We select the `usage_user` timeseries where the hostname is *host_0*. This query will select all data available for this timeseries. This query is the most general of the five.

Q2 :

```
(->>
    (index 39747)
    (range 1452120960000000000 1452123510000000000))
```

In this query, we select timeseries data from a range of time where we know the index value. Remember that this index value is stored in the tokens returned by the management port upon registering a timeseries.

Q3 :

```
(->>
    (metric "usage_user")
    (tag "hostname" '("host_0"))
    (where (> #V 50)))
```

This query is similar to the first query. However, this time, we execute a filter. This filter will filter out any timeseries data which is not more than 50.

Q4 :

```
(->>
    (metric "usage_user")
    (tag "hostname" '("host_0"))
    (groupby 1h avg))
```

Similar to the first query. We create groups of data, each of which is a bin of data representing one hour. From each bin, we calculate the average value. This query is used to create a summary of the entire timeseries or for a time range.

Q5 :

```
(->>
    (index 39747)
    (range 1452120960000000000 1452123510000000000)
    (groupby 15m max))
```

The final query will group by fifteen minutes. This query combines the second and fourth queries by selecting from a known index value and filtering for a range of time.

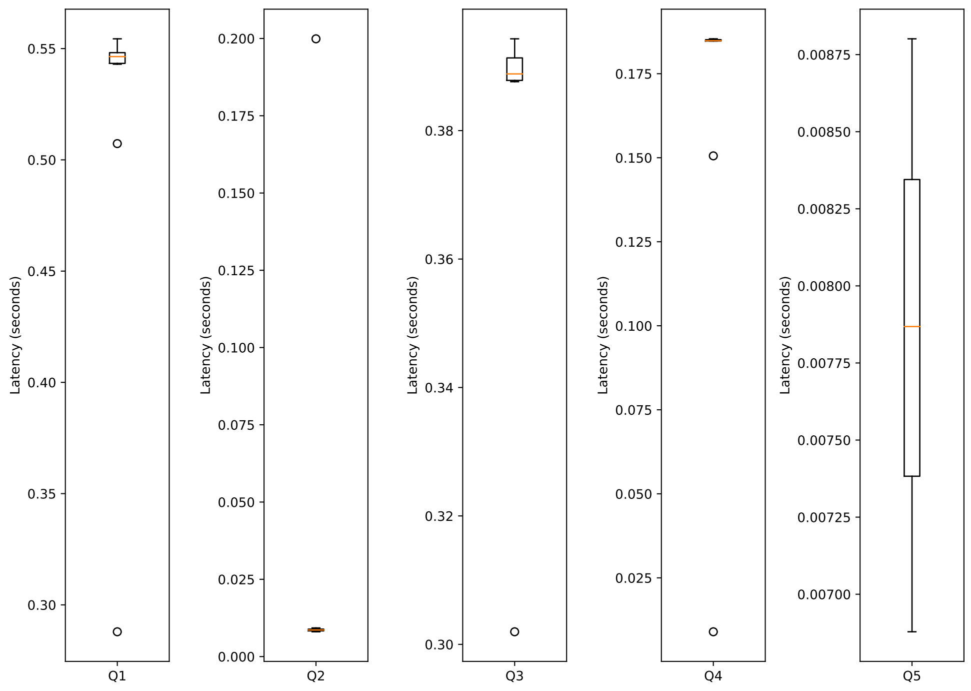


Figure 5.28: Query latency for the example five queries.

Figure 5.28 shows a boxplot of the latencies of the five queries. The boxplots show the latencies of the queries in milliseconds. This figure shows that the number of timeseries datapoints retrieved significantly impacts the query's latency. Queries two and five only retrieve a single memtable, while the other queries retrieve all timeseries data, which amounts to 120 memtables. We also observe that there is low variance for the first four queries. The variance is higher for the fifth, but the range is 1.75 milliseconds.

5.4 Summary

Now that we have evaluated the performance of FrogFishDB, we can provide an answer to RQ3. The impact of the ingestion protocol is that we went from being able to parse and ingest 7600 InfluxDB lines per second to 5.6 million when using a token. We found that this ingestion pipeline resulted in FrogFishDB being able to ingest 0.75 million to 2.5 million timeseries datapoints on average, depending on the number of timeseries ingested. When increasing the number of clients, this figure only grows until eight clients can push 3 to 10 million data points per second.

The impact of the indexing structure is more difficult to express quantitatively. When using microbenchmarks, we found that our indexing structure can handle insertions at a rate of 120 million entries per second, with a fanout of 202. When using a memtable of size 2MB, this converts to an ingestion bandwidth of 240 TB/s. The query performance of TimeTree is also dependent on the fanout configuration. In this case, increasing the fanout does improve performance. We measured five query sizes, ranging from small (ten items) to large (one million items). For FrogFishDB, we tested five example queries, each of which tests an aspect of our query language. Here we found that the size of a query significantly impacts performance, but the performance is stable and has low variance.

6

Future Work

6.1 Multi-threading

In section 4.5, we discussed a design for the threading model for FrogFishDB. We hypothesize that there is a significant performance increase to be gained by employing multiple threads. For example, we are currently not hitting the bandwidth limits of flash storage.

In section 2.1, we discuss the parallel nature of SSDs. On a hardware level, SSDs use multiple flash chips to which we can write concurrently.

6.2 Bypassing the filesystem for storage

Previous research has shown that the Linux storage stack imposes a significant overhead (82, 136). This research shows that bypassing the kernel and interacting with the raw block device can completely saturate the bandwidth of an SSD. In combination with section 6.1, we wish to use SPDK (114) to access the raw block device, bypass the kernel, and do so without the need to copy data.

We think that timeseries data is uniquely suited for flash storage due to its sequential nature. We discussed in section 2.1 that sequential writes benefit SSDs as this avoids triggering the GC process unnecessarily.

We can divide the SSDs into a collection of regions. Each region can be written sequentially, optimizing for the characteristics of flash storage. A few of the regions should be reserved for the log of the indexing structure. In section 4.3.2, we discussed using a log file to record insertions into the indexing structure. These log writes are also sequential, making it easy to use one or more regions for storing them.

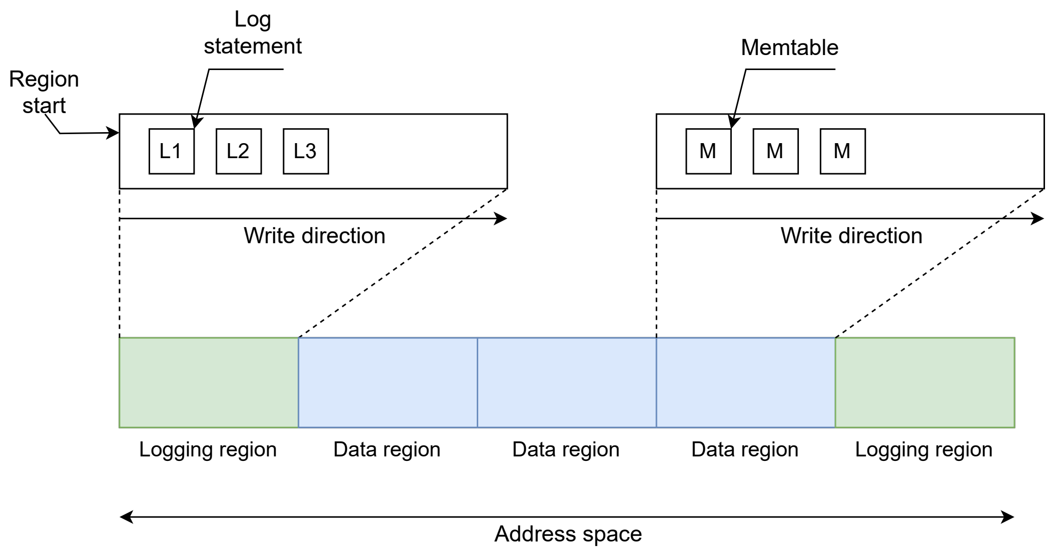


Figure 6.1: Different regions filling a flash devices.

Figure 6.1 shows a conceptual overview of such a system. We divide the address space of the block device into five regions, two of which are logging regions. In the logging regions, we store log entries, and in the data regions, we would store the memtables.

Combined with what we discussed in section 4.3.3, we can ensure we do not run out of storage space on the SSD. By aggregating old data, we can free up regions and make them available again for new data or log entries.

6.3 Log file compression

In section 4.3, we discussed the overhead of writing data to the indexing structure compared to the number of bytes written to storage for logging that insertion, i.e., the overhead for writing 1 byte to TimeTree equals writing 23 bytes to storage. One way to reduce this overhead is through the use of log compaction.

The log entries are currently rounded up to 512 bytes because we use `DIRECT_IO` for file access, meaning that all writes have to be 512 bytes. We envision a periodic process that will go through the log file and rewrite it by taking multiple log entries, combining them into 512-byte blocks, and writing those blocks to storage. Assuming 24 bytes (8 bytes for the start timestamp, 8 for the end timestamp, and 8 for the offset value) for each log entry, we can fit 21 log entries into a single 512-byte block. A process such as this, running in the background and periodically cleaning up log statements such as these, is akin to the one found in the LSM (102) tree.

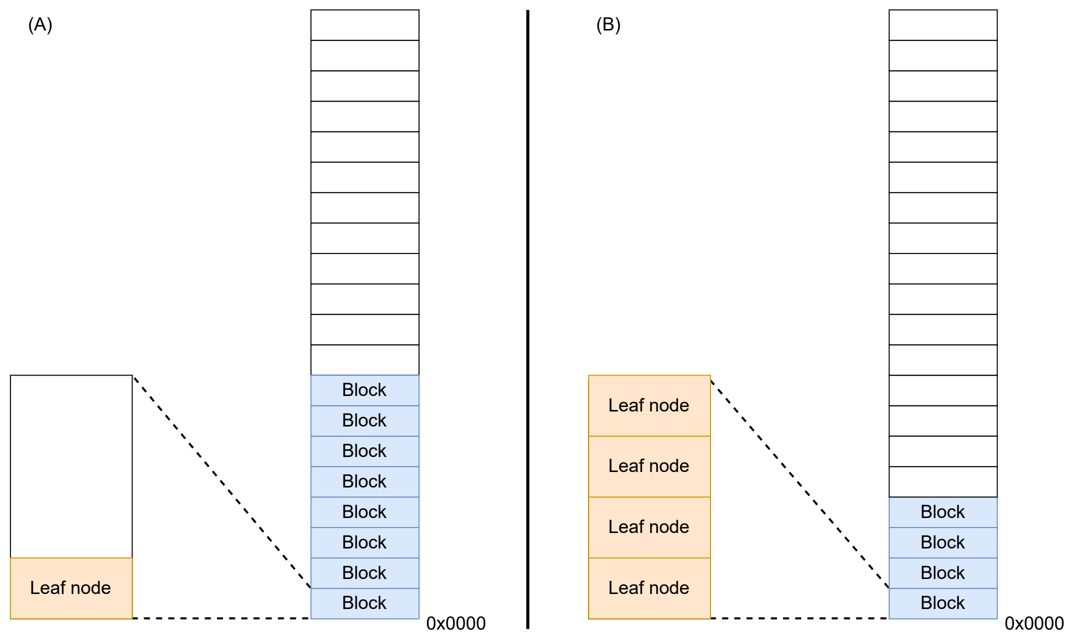


Figure 6.2: Log zone compression.

Figure 6.2 shows a conceptual overview of this log cleanup. On the left, we show the uncompressed state; on the right, we show the compressed form. The blue blocks represent the log data written to the log file. In (A), the uncompressed state shows a single leaf node stored in a block, while in (B), we show what happens when we compress multiple of those leaf nodes into a single block.

6.4 Usage of SIMD for the indexing structure

Searching in the TimeTree is not optimal. When executing a query, we first search for the leaf node containing the range's starting timestamp. Starting at the root and traversing downwards means that we have to do several comparison operations, the number of which is bounded by the fanout configured, e.g., for a tree with a fanout of 4, we need to do four comparisons per tree node. This process of comparing each node entry to the queried range's starting timestamp can be improved through the use of *Single Instruction Multiple Data* (SIMD) instructions. These instructions allow one to use a single instruction on multiple pieces of data. Modern processors, such as the Intel Core i9, contain registers that are 256 bits wide, allowing operations to execute four 64-bit data elements. For example, when using 256-bit SIMD instructions, we can simultaneously do a "less than" comparison on four entries. This idea is not new; it has been demonstrated in literature before (58, 137). Here the authors show a binary tree that has been optimized to work efficiently together with SIMD instructions.

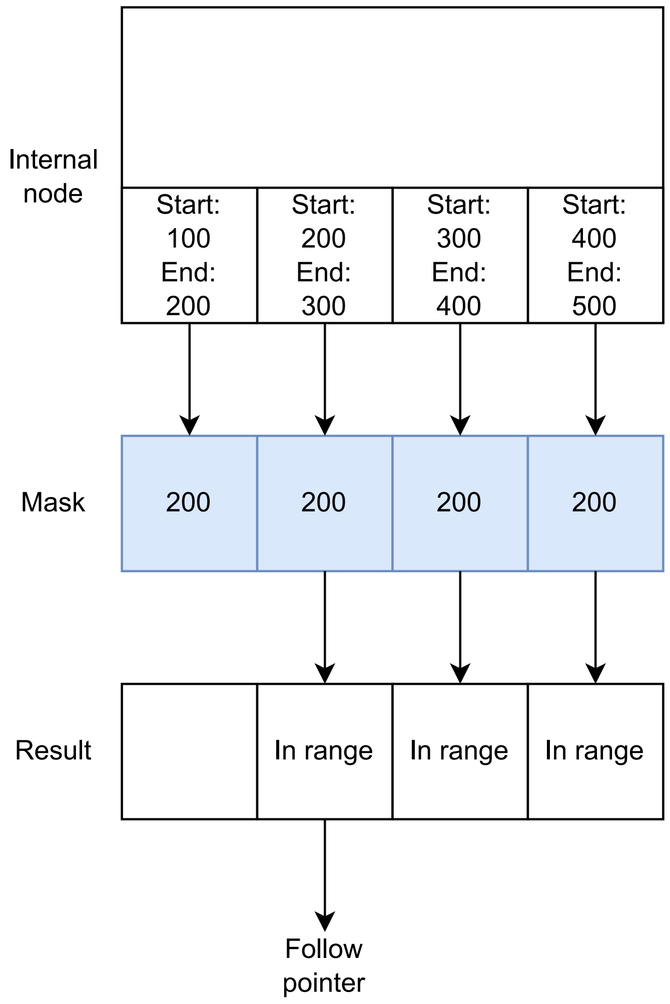


Figure 6.3: Lookup start of range in internal node.

Figure 6.3 illustrates comparing multiple entries in an internal node. This figure compares four starting timestamps using a single instruction, looking for a time range starting at timestamp 200. From this, we can determine that we should follow the second of the child links to the next internal node or leaf node. The mask is denoted in blue and contains the variable we are comparing the timestamps against. This mask is stored in a SIMD register and is overlayed on top of the starting timestamps stored in the internal node. Timestamps bigger than or equal to will be marked in the result register. Finally, we pick the index of the first matching entry and then follow the pointer to the node lower in the tree.

We can take the use of SIMD further than just looking for a specific timestamp; we can also use it to collect the stored values. As discussed in section 4.3.1, after finding the node which contains the starting timestamp, we collect all the stored values, which are then used to search for the data in storage. The collection of each of the stored values is done one by one. For this, we can also use SIMD instructions. We can use special gather-load instructions to collect multiple pieces of data stored in independent locations. The memory layout of the data in the leaf nodes would mean that we require the use of gather operations as the values are stored together with the start and end timestamp of the memtable. We would loop through the leaf nodes inside the queried range and collect all the stored values using these instructions. Another option would be to alter the memory layout of the leaf nodes. Such a layout would forgo storing the timestamps together with the values and keep the two separate. This would allow all non-timestamp values to be stored together and simplify the load instructions as we would resort to standard SIMD load instructions.

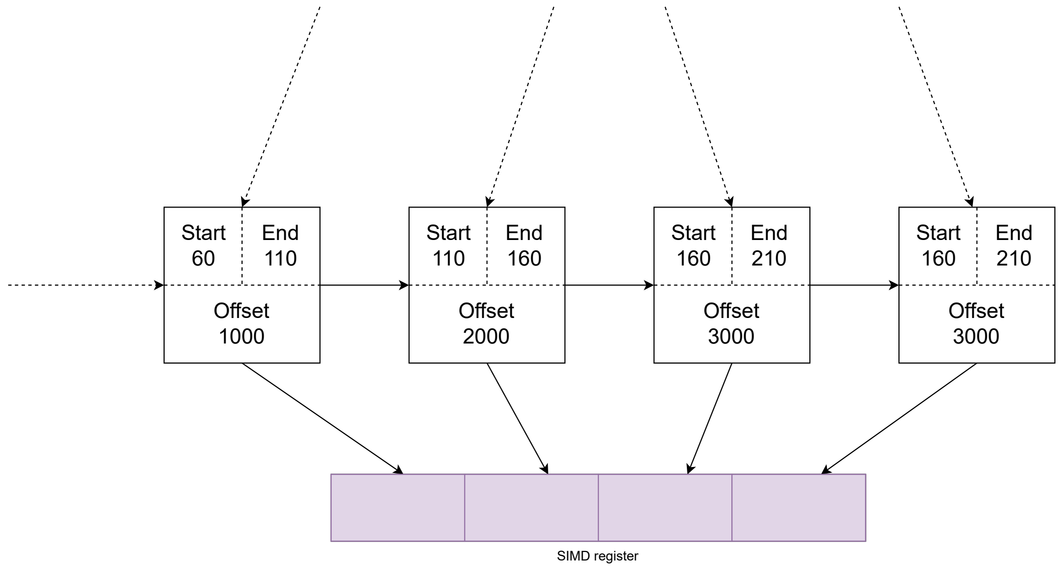


Figure 6.4: Collect values from multiple leafs in one load instruction.

Figure 6.4 shows an abstract view of the process of collecting values from the TimeTree using SIMD instructions. A register is loaded from four leaves. In the implementation used in this thesis, we would have to traverse the leaf nodes from left to right and collect each offset value individually.

Outside of TimeTree, we can use SIMD for ingestion as well. When ingesting a batch of data, we process each timeseries data point individually. Using SIMD instructions, we could ingest multiple data points simultaneously. The memtable is built from a single memory block containing pairs of timestamps and values. Using SIMD instructions, we could collect multiple data points into a single vector and copy the vector to the memtable.

6.5 Use the aggregation function of TimeTree in FrogFishDB

Section 4.3.3 discussed the possibility of aggregating older leaf nodes into non-leaf nodes to reduce storage costs. We mentioned that we have been unable to implement this due to time constraints, so we wish to examine this process in future work more thoroughly.

Depending on the size of the memtable, the storage savings could be significant. Imagine a TimeTree with a fanout of 4 and a memtable size of 2MB. Aggregating four nodes into a single node would reduce storage costs by 4× the memtable size. Using 3 summary operators (i.e. `min`, `max`, and `avg`), we would reduce 8MB to 24 bytes (assuming 8 bytes for each operator).

6.6 Examine the use of a Finite State Transducer

One of the more significant challenges during this work is converting the canonical name to an indexing structure. We have solved this through the use of the tokens. These are used to identify a timeseries and are a direct link to the memtable and indexing structure. This approach's main downside is that it makes it more difficult to find which tags are related to which structures. This downside has been partially mitigated through the use of an inverted index. However, it is still challenging to issue queries where we wish to do complex queries on multiple tags and metrics.

A possible solution could be the *Finite State Transducer* (FST) (138, 139, 140). An FST can be summarized as a prefix tree and a suffix tree. Before building an example, we assume the reader can read finite-state machine diagrams.

Figure 6.5 shows an example of an FST. We map three words to integer values. From top to bottom, we start with a single key-value pair, namely `jul`, mapped to the value 7. The middle FST shows that a new key-value pair has been added. The key is `mar`, which is mapped to value 3. Observe that there is no overlap in characters between `jul` and `mar`. The bottom example shows that the key `jun` has been added. Note that there have been some changes in how the values have been mapped. The first difference is that the value of the edge between nodes 0 and 4 has changed from 7 to 6. The second difference is the addition of an edge between nodes 5 and 3 and that the edge containing `l` now also maps a value.

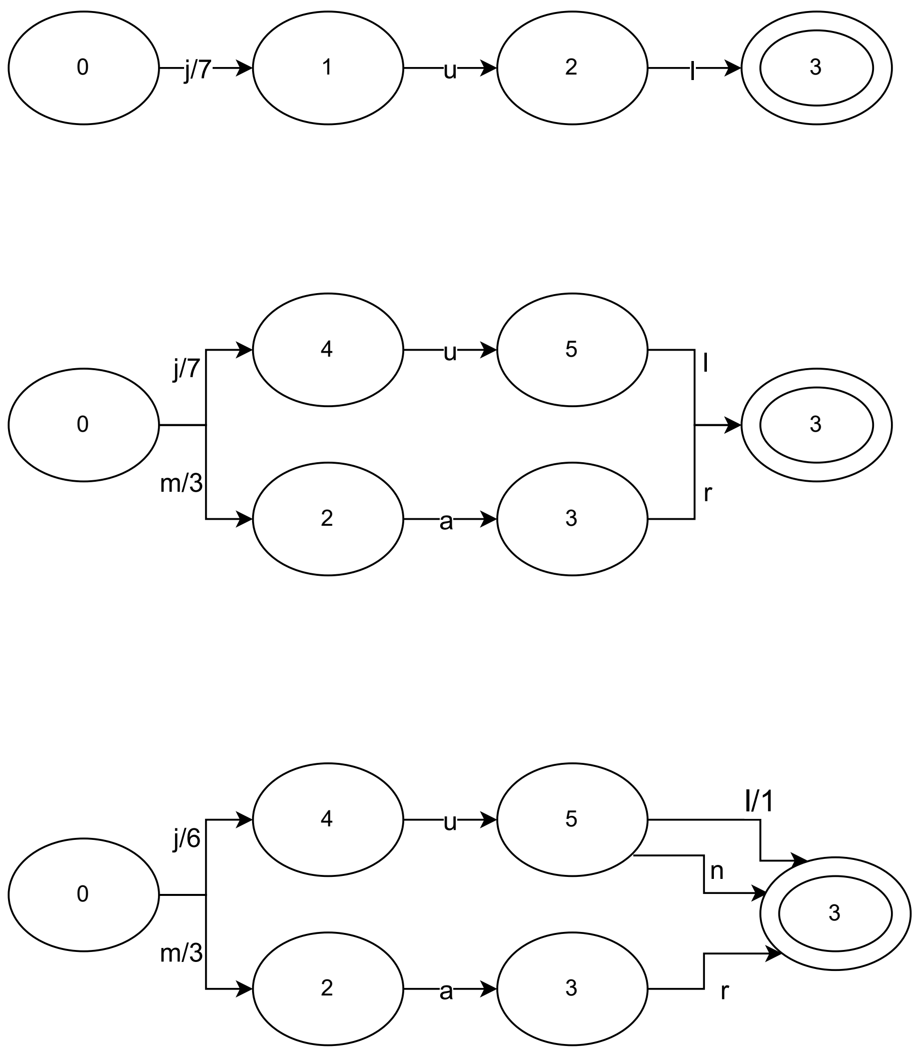


Figure 6.5: Example of an FST. The top example contains a single key-value pair, the middle contains two, and the bottom has three pairs with prefix overlap.

The FST should be read such that if a value is denoted in the edge, then this value is added. For example, the key `jul` is mapped to the value 7 in the top example. In the bottom example, this is still the case. However, the first value we encounter when reading the diagram is 6. If we then follow the final edge from 5 to 3, we add the value 1, resulting in again the value 7.

The main power comes from the overlap between `jun` and `jul`, i.e., the first two characters are removed, like a prefix tree. However, in contrast to a prefix tree, this effect does not only apply to prefixes; it also applies to postfixes. Consider what would happen when we map the key `kun` to the value 8; there is overlap in the final two characters, which can easily be encoded by creating another edge between nodes 0 and 4 containing the letter `k` combined with the value 8. This makes FSTs very space efficient.

One implementation (141) has shown that an FST, while not as space efficient as either gzip (142) or xz (143), the performance of both lookups and creating is better. Compared to gzip and xz, the time it takes to construct an FST is 2.04 seconds while taking 2.50 seconds and 14.66 seconds for gzip and xz, respectively.

The major downside of an FST is that it requires all data to be inserted in lexicographical order. This problem is that we cannot know the entire set of series names when starting the database. One solution to this problem is to collect series names into separate and smaller FSTs, and merge them later. What needs to be investigated is if an FST can be used without a significant performance regression and if there is a better way to handle the insertion into the FST.

7 Conclusion

In this thesis, we have presented a new timeseries database, FrogFishDB. This database is built from the ground up to address two perceived limitations that limit the ingestion bandwidth of existing timeseries databases. The first limitation addressed is the overhead of existing ingestion protocols. We designed a new pipeline where clients first register a timeseries and then receive a token. This token is then used during ingestion to identify which timeseries batches of timeseries data belong. The second issue is the complexity of indexing data structures. These data structures are used to find timeseries data after it has been written to storage. We have presented TimeTree. This data structure simplifies the B+ tree, where we assume data arrives in order. The assumption has allowed us to forgo the splitting and merging of internal nodes in the tree and has allowed us to execute queries based on a range of time.

The design of the ingestion pipeline and TimeTree have been shown in chapter 4. We evaluated the design in chapter 5. We found that TimeTree is theoretically capable of ingesting 240TB/s, far exceeding the bandwidth of modern NVMe flash storage. When benchmarking FrogFishDB for ingestion bandwidth, we found that it could ingest 0.75 to 2.5 million data points on average from a single client. When increasing the parallelism by increasing the number of clients, we found that the ingestion bandwidth increased to 3 to 10 million data points per second. Five example queries determined the query performance of FrogFishDB. We found that the performance depends on the number of data points retrieved. When querying for a week of data, we found that we could retrieve, filter, and process the data within 175 milliseconds. We found that the queries exhibited low variance.

This leads us to our main research question, *How to build a timeseries database which optimizes for flash storage and provides high ingestion performance?*. By redesigning the ingestion pipeline, we built a timeseries database that provides high ingestion performance.

This pipeline shifts part of the workload to the client, namely the part of the workload which is responsible for matching data to the timeseries. We can optimize the database for flash storage by taking advantage of access patterns that are favorable for flash storage. Flash storage favors sequential accesses. This is why we designed the storage for timeseries data and the log to write sequentially through appends.

7.1 Research questions

RQ1: How to design an ingestion protocol which trades readability away for performance?
In section 4.2.2, we demonstrated how such a protocol can be built. We trade away readability through the use of a binary protocol and by using an identification token. We shift part of the ingestion workload to the client. The client is responsible for registering a timeseries. The database responds with a token which the client then uses to identify a timeseries. This increases performance because the finding datastructures for a timeseries is faster when the database does not have to do any string manipulation.
RQ2: How to design a flash-friendly indexing structure that is specialized for storing and looking up time-indexed data?
In section 4.3, we described the design of an indexing structure for indexing timeseries data. The design makes use of two observations. The first observation is that all timeseries data arrive in order. The second is that timeseries data is immutable after ingesting it. Considering these observations allows us to take a B+ tree and simplify its functionality to improve performance and ease implementation.
RQ3: What is the quantitative impact of designing a new ingestion protocol and a specialized indexing structure?
In chapter 5, we showed a quantitative evaluation of both TimeTree and FrogFishDB. Benchmarking FrogFishDB showed that when using a single client, it achieved an ingestion bandwidth of 0.75 to 2.5 million points per second. Increasing the number of clients to eight improves bandwidth to 3 to 10 million points per second. When using eight clients, the ingestion bandwidth of FrogFishDB is 1.8 to 5 times higher compared to existing timeseries databases.

7.2 Limitations

There are several limitations to the work presented in this thesis. Partly these are limitations in functionality, and partly these are limitations in what has been explored in this work. In chapter 6, we discuss ideas that will take FrogFishDB further than demonstrated in this work. In this section, we will reflect on what findings we do not clearly understand.

The first is the negative effect that increasing the number of timeseries has on the performance of FrogFishDB. In chapter 5, we observed that the ingestion bandwidth dropped significantly when increasing the scale from 32 to 320 and further to 3200. At a scale of 3200, we observed that the multiclient ingestion bandwidth dropped by more than 50%.

Another limitation is the lack of examination into the effects of changing the batch size when ingesting data. For all experiments in section 5, we used a batch size of 1,000. Cao et al. have shown in the Timon paper that changing the size of the batches can significantly alter the ingestion performance of a timeseries database. Figure 15 of the paper shows the effects when comparing Timon to BTrDB. BTrDB outperforms Timon regarding ingestion bandwidth when the batch size is increased beyond 50,000 points, assuming 16 bytes (8 for the timestamp and 8 for the value). This results in a batch size of 800KB.

The final limitation we have to discuss is the limited examination of the query performance. At the start of this work, we acknowledged that this work is focused on ingestion performance, not query performance. However, we also recognize that query performance is critical to the overall package. If we were to use FrogFishDB in the context of monitoring a large number of services and servers, then we would also want to have the ability to execute preconfigured queries that monitor the health of said services and servers. When queries have a latency of over half a second, as we observed in figure 5.28, this becomes unmaintainable. Imagine monitoring 300 servers with ten queries for each server, where each query is executed every second. Then this amounts to 3000 queries per second. While we have shown in figure 5.27 that TimeTree should be capable of this figure, a latency of half a second per query would require careful design to ensure there is no back pressure.

References

[1] Dave Wilson. **IoT Is Creating Massive Growth Opportunities**. https://blogs.cisco.com/internet-of-things/iot-is-creating-massive-growth-opportunities, October 2020.

[2] David Reinsel, John Gantz, and John Rydning. **The Digitization of the World from Edge to Core**. *Framingham: International Data Corporation*, **16**:1–28, 2018.

[3] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan S. Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**. *CoRR*, **abs/2206.03259**, 2022.

[4] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. **A Symbolic Representation of Time Series, with Implications for Streaming Algorithms**. In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 2–11, New York, NY, USA, June 2003. Association for Computing Machinery.

[5] Anshul Sharma. **How to Analyze F1 Data in Real Time with ADX and Grafana**. https://grafana.com/blog/2022/12/09/how-to-build-a-formula-1-real-time-analytics-stack-with-azure-data-explorer-and-grafana-cloud/, December 2022.

[6] Carey Wodehouse. **How Formula 1 Car Sensors Create Data at Every Turn**. https://blog.purestorage.com/perspectives/how-formula-1-car-sensors-create-data-at-every-turn/, December 2021.

[7] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Adam Tart, and Nick Taylor. **Monarch: Google's Planet-Scale in-Memory Time Series Database**. *Proceedings of the VLDB Endowment*, **13**(12):3181–3194, August 2020.

[8] Timothy Prickett Morgan. **A Rare Peek Into The Massive Scale of AWS**. https://www.enterpriseai.news/2014/11/14/rare-peek-massive-scale-aws/, November 2014.

[9] Abdullah Mueen, Eamonn Keogh, Qiang Zhu, Sydney Cash, and Brandon Westover. **Exact Discovery of Time Series Motifs**. In *Proceedings of the 2009 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 473–484. Society for Industrial and Applied Mathematics, April 2009.

[10] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. **Performance Analysis of NVMe SSDs and Their Implication on Real World Databases**. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 1–11, New York, NY, USA, May 2015. Association for Computing Machinery.

[11] Jeffrey Dean and Luiz André Barroso. **The Tail at Scale**. *Communications of the ACM*, **56**(2):74–80, February 2013.

[12] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. **Don't Stack Your Log on My Log**. *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.

[13] Jaeho Kim, Donghee Lee, and Sam H Noh. **Towards SLO Complying SSDs Through OPS Isolation**. *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[14] Se Jin Kwon, Arun Ranjitkar, Young-Bae Ko, and Tae-Sun Chung. **FTL Algorithms for NAND-type Flash Memories**. *Design Automation for Embedded Systems*, **15**(3):191–224, December 2011.

[15] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. **The Unwritten Contract of Solid State Drives**. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 127–144, Belgrade Serbia, April 2017. ACM.

[16] Jiulei Jiang, Jiajin Le, and Yan Wang. **A Column-Oriented Storage Query Optimization for Flash-Based Database**. In *2010 International Conference on Future Information Technology and Management Engineering*, **3**, pages 512–516, October 2010.

[17] Martin V. Jørgensen, René B. Rasmussen, Simonas Šaltenis, and Carsten Schjønning. **FB-tree: A B+-Tree for Flash-Based SSDs**. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, pages 34–42, New York, NY, USA, September 2011. Association for Computing Machinery.

[18] Yoshinori Matsunobu. **{InnoDB} to {MyRocks} Migration in Main {MySQL} Database at Facebook**. https://www.usenix.org/conference/srecon17asia/program/presentation/matsunobu, 2017.

[19] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. **NoFTL for Real: Databases on Real Native Flash Storage**, 2015.

[20] Keren Ouaknine, Oran Agra, and Zvika Guz. **Optimization of RocksDB for Redis on Flash**. In *Proceedings of the International Conference on Compute and Data Analysis*, pages 155–161, Lakeland FL USA, May 2017. ACM.

[21] Mohit Saxena and Michael M Swift. **Revisiting Database Storage Optimizations on Flash**. *University of Wisconsin-Madison Department of Computer Sciences*, 2010.

[22] Fei Yang, Kun Dou, Siyu Chen, Mengwei Hou, Jeong-Uk Kang, and Sangyeun Cho. **Optimizing NoSQL DB on Flash: A Case Study of RocksDB**. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1062–1069, August 2015.

[23] Jalal Mostafa, Sara Wehbi, Suren Chilingaryan, and Andreas Kopmann. **SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things**, June 2022.

[24] Muntazir Fadhel, Emil Sekerinski, and Shucai Yao. **A Comparison of Time Series Databases for Storing Water Quality Data**. In Michael E. Auer and Thrasyvoulos Tsiatsos, editors, *Mobile Technologies and Applications for the Internet of Things*, **909**, pages 302–313. Springer International Publishing, Cham, 2019.

[25] Matei-Eugen Vasile, Giuseppe Avolio, and Igor Soloviev. **Evaluating InfluxDB and ClickHouse Database Technologies for Improvements of the ATLAS Operational Monitoring Data Archiving**. https://iopscience.iop.org/article/10.1088/1742-6596/1525/1/012027/pdf, 2020.

[26] Adrian Göransson and Oskar Wändesjö. *Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry*. Master's thesis, Lund University, April 2022.

[27] Michael P Andersen and David E Culler. **BTrDB: Optimizing Storage System Design for Timeseries Processing**. *14th USENIX Conference on File and Storage Technologies (FAST 16)*, page 15, 2016.

[28] Frank McSherry, Michael Isard, and Derek G. Murray. **Scalability! But at What {COST}?** In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[29] Kazuaki Maeda. **Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats**. In *2012 Second International Conference on Digital Information and Communication Technology and It's Applications (DICTAP)*, pages 177–182, May 2012.

[30] Daniel Persson Proos and Niklas Carlsson. **Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV**. *2020 IFIP networking conference (networking)*, **2020 IFIP networking conference**:10/18, 2020.

[31] Wei Cao, Yusong Gao, Feifei Li, Sheng Wang, Bingchen Lin, Ke Xu, Xiaojie Feng, Yucong Wang, Zhenjun Liu, and Gejin Zhang. **Timon:**

A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 739–753, Portland OR USA, June 2020. ACM.

[32] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. **Gorilla: A Fast, Scalable, in-Memory Time Series Database**. *Proceedings of the VLDB Endowment*, **8**(12):1816–1827, August 2015.

[33] InfluxDB. **InfluxDB Line Protocol Reference | InfluxDB OSS 1.8 Documentation**. https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_reference/.

[34] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. **On The [Ir]Relevance of Network Performance for Data Processing**. *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, June 2014.

[35] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. **Non-Volatile Storage: Implications of the Datacenter's Shifting Center**. *Queue*, **13**(9):33–56, November 2015.

[36] Simon Peter, Jialin Li, Doug Woos, Irene Zhang, Dan R K Ports, Thomas Anderson, Arvind Krishnamurthy, and Mark Zbikowski. **Towards High-Performance Application-Level Storage Management**. *HotCloud'16: Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, June 2016.

[37] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *Journal of Management Information Systems*, **24**(3):45–77, December 2007.

[38] Richard W Hamming. *Art of Doing Science and Engineering.* CRC Press, 1997.

[39] Gernot Helser. **Gernot's List of Systems Benchmarking Crimes**. https://gernot-heiser.org/benchmarking-crimes.html.

[40] John Ousterhout. **Always Measure One Level Deeper**. *Communications of the ACM*, **61**(7):74–83, June 2018.

[41] Mark Zandi, Sophia Koropeckyj, Virendra Singh, and Paul Matsiras. **The Impact of Electronic Payments on Economic Growth**. Technical report, Moody's Analytics: Economic and Consumer Credit Analytics, February 2016.

[42] IEEE Spectrum. **Chip Hall of Fame: Toshiba NAND Flash Memory - IEEE Spectrum**. https://spectrum.ieee.org/chip-hall-of-fame-toshiba-nand-flash-memory.

[43] Hitomi Tanaka, Yuta Aiba, Takashi Maeda, Kensuke Ota, Yusuke Higashi, Keiichi Sawa, Fumie Kikushima, Masayuki Miura, and Tomoya Sanuki. **Toward 7 Bits per Cell: Synergistic Improvement of 3D Flash Memory by Combination of Single-crystal Channel and Cryogenic Operation**. In *2022 IEEE International Memory Workshop (IMW)*, pages 1–4, May 2022.

[44] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R Stan. **How I Learned to Stop Worrying and Love Flash Endurance**. *HotStorage*, 2010.

[45] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. **Flash Reliability in Production: The Expected and the Unexpected**. *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, 2016.

[46] Rino Micheloni, editor. *3D Flash Memories*. Springer Netherlands, Dordrecht, 2016.

[47] Rino Micheloni, editor. *Solid-State-Drives (SSDs) Modeling: Simulation Tools & Strategies*, **58** of *Springer Series in Advanced Microelectronics*. Springer International Publishing, Cham, 2017.

[48] Dave Landsman and Don Walker. **AHCI and NVMe as Interfaces for SATA Express™ Devices**. *SATA-IO*, 2013.

[49] PennState Eberly College of Science. **1.1 Overview of Time Series Characteristics | STAT 510**. https://online.stat.psu.edu/stat510/lesson/1/1.1.

[50] Robert H. Shumway and David S. Stoffer. **Characteristics of Time Series**. In Robert H. Shumway and David S. Stoffer, editors, *Time Series Analysis and Its Applications: With R Examples*, Springer Texts in Statistics, pages 1–44. Springer International Publishing, Cham, 2017.

[51] Prometheus. **Prometheus**. https://github.com/prometheus/prometheus/blob/8553a98267a56acf November 2022.

[52] VictoriaMetrics. **VictoriaMetrics**. https://github.com/VictoriaMetrics/VictoriaMetrics, August 2023.

[53] QuestDB. **QuestDB | Fast SQL for Time-Series**. https://questdb.io/.

[54] SQLite. **SQLite Home Page**. https://www.sqlite.org/index.html, May 2023.

[55] InfluxDB. **InfluxDB | Real-time Insights at Any Scale**. https://www.influxdata.com/home/, Sat, 15 Jan 2022 15:32:09 +0000.

[56] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. **Albis: High-Performance File Format for Big Data Systems**. *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[57] Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, April 1998.

[58] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. **FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs**. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 339–350, Indianapolis Indiana USA, June 2010. ACM.

[59] Agner Fog. **4. Instruction Tables**. https://www.agner.org/optimize/instruction_tables.pdf, April 22.

[60] Allan Omondi, Ismail Ateya, and Gregory Wanyembi. **Sustainable Energy Consumption in Data Centres**. *17th ICT Conference*, March 2019.

[61] Douglas Comer. **Ubiquitous B-Tree**. *ACM Computing Surveys*, **11**(2):121–137, June 1979.

[62] Edward Shishkin. **Reiser4 FS Wiki**. https://reiser4.wiki.kernel.org/index.php/Main_Page, August 2020.

[63] Silicon Graphics, Inc, Ryan Lerch, Eric Sandeen, Dave Chinner, and Darrick Wong. **XFS Algorithms & Data Structures**. https://mirror.math.princeton.edu/pub/kernel/linux/utils/fs/xfs/docs/xfs_filesystem_structure.pdf, 2006.

[64] MySQL. **MySQL :: Download MySQL Community Server**. https://dev.mysql.com/downloads/mysql/.

[65] Baron Schwartz. **How We Scale VividCortex's Backend Systems - High Scalability -**. http://highscalability.com/blog/2015/3/30/how-we-scale-vividcortexs-backend-systems.html, March 2015.

[66] Baron Schwartz. **Building A Time-Series Database on MySQL | SCALE 13x**, February 2015.

[67] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. John Wiley & Sons, April 2005.

[68] IEEE. **The Open Group Base Specifications Issue 7, 2018 Edition**. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[69] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, October 2010.

[70] Davidlohr Bueso and SUSE Labs. **Epoll Kernel Performance Improvements**. *Open Source Summit*, July 2019.

[71] Rajat P. Garg and Ilya Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall Professional Technical Reference, 2002.

[72] Damien Le Moal. **I/O Latency Optimization with Polling**. *Vault Linux Storage and Filesystems Conference*, March 2017.

[73] Gyusun Lee, Seokha Shin, and Jinkyu Jeong. **Efficient Hybrid Polling for Ultra-Low Latency Storage Devices**. *Journal of Systems Architecture*, **122**:102338, January 2022.

[74] Jens Axboe. **Io_uring and Optane2**, August 2020.

[75] Linus Torvalds. **Re: [PATCH 09/13] Aio: Add Support for Async Openat() [LWN.Net]**, Mon, 11 Jan 2016 16:22:28 -0800.

[76] Jonathan Corbet. **Toward Non-Blocking Asynchronous I/O [LWN.Net]**. https://lwn.net/Articles/724198/, May 2017.

[77] Jens Axboe. **Efficient IO with Io_uring**, 2019.

[78] Jens Axboe. **Io_uring(7) — Arch Manual Pages**. https://man.archlinux.org/man/io_uring.7, July 2020.

[79] Livio Soares and Michael Stumm. **{FlexSC}: Flexible System Call Scheduling with {Exception-Less} System Calls**. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[80] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. **Reaping the Performance of Fast {NVM} Storage with {uDepot}**. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[81] Jens Axboe. **Kernel/Git/Torvalds/Linux.Git - Linux Kernel Source Tree**. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c7fb19428d67dd0a May 2022.

[82] Zebin Ren and Animesh Trivedi. **Performance Characterization of Modern Storage Stacks: POSIX I/O, Libaio, SPDK, and Io_uring**. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pages 35–45, Rome Italy, May 2023. ACM.

[83] John Kariuki and Vishal Verma. **Improved Storage Performance Using the New Linux Kernel I/O Interface**. *SDC 2019*, 2019.

[84] Ruslan Savchenko. **Reading from External Memory**, February 2021.

[85] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. **Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring**. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, Haifa Israel, June 2022. ACM.

[86] Rudolf Bayer and Edward McCreight. **Organization and Maintenance of Large Ordered Indices**. *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1970.

[87] J. Postel. **Transmission Control Protocol**. Technical Report RFC0793, RFC Editor, September 1981.

[88] Yakov Rekhter, Susan Hares, and Tony Li. **A Border Gateway Protocol 4 (BGP-4)**. Request for Comments RFC 4271, Internet Engineering Task Force, January 2006.

[89] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. **P4: Programming Protocol-Independent Packet Processors**. *ACM SIGCOMM Computer Communication Review*, **44**(3):87–95, July 2014.

[90] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. **A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research**. *Journal of Network and Computer Applications*, **212**:103561, March 2023.

[91] Celio Trois, Marcos D. Del Fabro, Luis C. E. de Bona, and Magnos Martinello. **A Survey on SDN Programming Languages: Toward a Taxonomy**. *IEEE Communications Surveys & Tutorials*, **18**(4):2687–2712, 2016.

[92] Rahim Masoudi and Ali Ghaffari. **Software Defined Networks**. *Journal of Network and Computer Applications*, **67**(C):1–25, May 2016.

[93] J. Postel. **User Datagram Protocol**. Request for Comments RFC 768, Internet Engineering Task Force, August 1980.

[94] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. **Understanding Data Center Traffic Characteristics**. *ACM SIGCOMM Computer Communication Review*, **40**, 2010.

[95] Gerard Salton, Edward A. Fox, and Harry Wu. **Extended Boolean Information Retrieval**. *Communications of the ACM*, **26**(11):1022–1036, November 1983.

[96] Justin Zobel and Alistair Moffat. **Inverted Files for Text Search Engines**. *ACM Computing Surveys*, **38**(2):6–es, July 2006.

[97] Alex McEachern. **Micro-Synchrophasors for Distribution Grids: Instrumentation Lessons Learned (so Far!)**. *Power Standards Lab*, 2016.

[98] Ceph. **Ceph.Io — Home**. https://ceph.io/en/.

[99] DB-Engines. **DB-Engines Ranking**. https://db-engines.com/en/ranking/time+series+dbms, August 2023.

[100] Google. **LevelDB**. https://github.com/google/leveldb, August 2023.

[101] Google. **Snappy/Format_description.Txt at Main · Google/Snappy**. https://github.com/google/snappy/blob/main/format_description.txt, October 5.

[102] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. **The Log-Structured Merge-Tree (LSM-tree)**. *Acta Informatica*, **33**(4):351–385, June 1996.

[103] OpenTSDB. **OpenTSDB - A Distributed, Scalable Monitoring System**. http://opentsdb.net/, 2023.

[104] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. **Apache IoTDB: Time-Series Database for Internet of Things**. *Proceedings of the VLDB Endowment*, **13**(12):2901–2904, August 2020.

[105] Eugene Lazin. **Akumuli**. https://akumuli.org/, May 2020.

[106] Linux. **Mmap(2) - Linux Manual Page**. https://www.man7.org/linux/man-pages/man2/mmap.2.html.

[107] Andrew Crotty, Viktor Leis, and Andrew Pavlo. **Are You Sure You Want to Use MMAP in Your Database Management System?** *CIDR 2022, Conference on Innovative Data Systems Research. https://db.cs.cmu.edu/papers/2022/p13-crotty.pdf*, 2022.

[108] Bo Mao, Suzhen Wu, Hong Jiang, Yaodong Yang, and Zaifa Xi. **EDC: Improving the Performance and Space Efficiency of Flash-Based Storage Systems with Elastic Data Compression**. *IEEE Transactions on Parallel and Distributed Systems*, **29**(6):1261–1274, June 2018.

[109] Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. **uFLIP: Understanding Flash IO Patterns**. *arXiv preprint arXiv:0909.1780*, 2009.

[110] Kenton Varda. **Cap'n Proto**. https://github.com/capnproto/capnproto, August 2023.

[111] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. **Securing RDMA for High-Performance Datacenter Storage Systems**. *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[112] Anduo Wang. **Centralized Control — Separating Data- and Control-Planes**.

[113] Srinivas Narayana. **Control-Dataplane Separation**, October 2019.

[114] SPDK. **Storage Performance Development Kit**. https://spdk.io/.

[115] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. **ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging**. *ACM Transactions on Database Systems*, **17**(1):94–162, March 1992.

[116] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. **Scalability of Write-Ahead Logging on Multicore and Multisocket Hardware**. *The VLDB Journal*, **21**(2):239–263, April 2012.

[117] Richard L. Wexelblat, editor. *History of Programming Languages.* ACM Monograph Series. Academic Press, New York, 1981.

[118] Andy Gavin. **Making Crash Bandicoot – GOOL – Part 9**, March 2011.

[119] R S Scowen and Birchwood Grove. **Extended BNF — A Generic Base Standard**. *Proceedings 1993 Software Engineering Standards Symposium*, 1993.

[120] J. Borchardt. **Lampe, B./Jorke, G./Wengel, H., Algorithmen Der Mikrorechentechnik. Maschinenprogrammierung Und Interpretertechniken Des U880. Berlin, VEB Verlag Technik 1983. 364 S., 230 Abb., M 37,50. BN 5532246**. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, **64**(11):474–474, 1984.

[121] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. **MICA: A Holistic Approach to Fast In-Memory Key-Value Storage**. *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, page 17, 2014.

[122] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. **The Impact of Thread-Per-Core Architecture on Application Tail Latency**. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–8, Cambridge, United Kingdom, September 2019. IEEE.

[123] ScyllaDB. **Home**. https://www.scylladb.com/.

[124] Piotr Grabowski, Juliusz Stasiewicz, and Karol Baryla. **Apache Cassandra 4.0 Performance Benchmark**. https://www.scylladb.com/2021/08/24/apache-cassandra-4-0-vs-scylla-4-4-comparing-performance/, August 2021.

[125] Zhiqi Wang, Jin Xue, and Zili Shao. **Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries**. *Proceedings of the VLDB Endowment*, **14**(6):1080–1092, February 2021.

[126] Eugene Siow, Thanassis Tiropanis, Xin Wang, and Wendy Hall. **TritanDB: Time-series Rapid Internet of Things Analytics**. *arXiv:1801.07947 [cs]*, January 2018.

[127] Intel. **Intel® Xeon® Silver 4210R Processor (13.75M Cache, 2.40 GHz) Product Specifications**. https://www.intel.com/content/www/us/en/products/sku/197098/intel-xeon-silver-4210r-processor-13-75m-cache-2-40-ghz.html.

[128] Mingming Cao, Suparna Bhattacharya, and Ted Tso. **Ext4: The Next Generation of Ext2/3 Filesystem**. *LSF*, 2006.

[129] QEMU Team. **QEMU**. https://www.qemu.org/, April 2021.

[130] Martin Leitner-Ankerl. **Martinus/Nanobench: Simple, Fast, Accurate Single-Header Microbenchmarking Functionality for C++11/14/17/20**. https://github.com/martinus/nanobench/tree/master.

[131] TimescaleDB. **Time Series Benchmark Suite (TSBS)**. https://github.com/timescale/tsbs, November 2022.

[132] TimescaleDB. **Time-Series Data Simplified**. https://www.timescale.com.

[133] Nicholas Nethercote. **Valgrind Home**. https://valgrind.org/, 2023.

[134] Jens Axboe. **Fio**. https://fio.readthedocs.io/en/latest/fio_doc.html.

[135] Eelco Dolstra. **Nix Package Manager - NixOS Wiki**. https://nixos.wiki/wiki/Nix_package_manager, 2006.

[136] Gabriel Haas, Michael Haubenschild, and Viktor Leis. **Exploiting Directly-Attached NVMe Arrays in DBMS**. *CIDR*, 2020.

[137] Florian Gross. **Index Search Algorithms for Databases and Modern CPUs**, June 2017.

[138] Michael McCandless. **Using Finite State Transducers in Lucene**.

[139] Mehryar Mohri. **Weighted Finite-State Transducer Algorithms. An Overview**. In Janusz Kacprzyk, Carlos Martín-Vide, Victor Mitrana, and Gheorghe Păun, editors, *Formal Languages and Applications*, **148**, pages 551–563. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[140] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. **Incremental Construction of Minimal Acyclic Finite-State Automata**. *Computational Linguistics*, **26**(1):3–16, March 2000.

[141] Andrew Gallant. **Index 1,600,000,000 Keys with Automata and Rust**, November 2015.

[142] L. Peter Deutsch. **GZIP File Format Specification Version 4.3**. Request for Comments RFC 1952, Internet Engineering Task Force, May 1996.

[143] Apoorv Gupta, Aman Bansal, and Vidhi Khanduja. **Modern Lossless Compression Techniques: Review, Comparison and Analysis**. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–8, February 2017.

8

Appendix

8.1 Experiment reproduction

In this section, we will provide the necessary steps to reproduce the experiments shown in chapter 5. We start with a guide on how to setup the environment in which the experiments are to be conducted.

8.1.1 Setup

The main requirements for the setup are git and the Nix package manager.

```
$ git clone git@github.com:NielsdeWaal/Thesis.git
```

The `Thesis` repository contains the code for the database, client, virtual machines, and TimeTree. The code for TimeTree is located in a different repository. However, this repository does not need to be cloned explicitly, as it will be retrieved by the cmake code for the database.

8.1.1.1 VM generation

The images for the virtual machine are reproducible through the Nix package manager. The `VM-flakes` folder contains the nix flakes, which are used to generate the images.

The available images are:

- influx_machine
- clickhouse_machine
- questdb_machine

- zns_machine

As the VM's require SSH to be accessed, the authorized_key must be replaced in the `common/default.nix` file. When in the `VM-flakes` folder, an image can be generated using the following command:

```
$ nix build .#IMAGE_NAME
```

Where `IMAGE_NAME` has to be replaced with the desired image. Nix will download the required packages and configure the image. The resulting qcow2 image will be located in the `result` folder.

As the size of the benchmarking data is very large, increase the size of the image to accommodate the size during ingestion.

```
$ qemu-img resize result/nixos.qcow2 +200G
```

The `resize` command will increase the image size with 200GB.

8.1.1.2 Code compilation

To compile the code for FrogFishDB, navigate to the `DB` folder. To generate the build files and start the build, issue the following commands:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make -j $nprocs
```

The client is built in a similar manner, from the `client` folder in the thesis repository, execute the following:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make -j $nprocs
```

The TimeTree folder contains the TimeTree implementation and its performance tests. From the `TimeTree` folder, execute the following commands:

```
$ mkdir build
$ cd build
$ cmake -DTIME_TREE_TESTS=1 -DCMAKE_BUILD_TYPE=Release ..
$ make -j $nprocs
```

The TIME_TREE_TESTS directive will enable the testing and benchmarking code for the TimeTree.

Before going into the build folder, one can opt to use the Nix package manager to launch a shell which contains the exact compiler and library versions as those used in this thesis. The shell can be started with the following command:

```
$ nix develop
```

The Nix package manager will use the flake.lock and flake.nix files to populate the shell with the versions. The lock file dictates the version through the Nix store. The store is pinned through a collection of hashes stored in the lock file.

8.1.1.3 Generating data

For this step, the go programming language needs to be installed.

We use the TimeSeries Benchmark Suite for generating the timeseries data. Due to issues with the clickhouse integration, we need to apply a patch to fix the code.

```
$ git clone git@github.com:timescale/tsbs.git
$ cd tsbs
```

The patch is located in the scripts folder of the thesis repository. Apply the patch with:

```
$ git apply THESIS_FOLDER/scripts/clickhouse.patch
```

Replace THESIS_FOLDER with the location of the thesis repository.

Next, we generate the data. Be aware that this requires a lot of storage space. The 3200 scale data file will require 200GB of storage.

```
$ cd cmd/tsbs_generate_data
$ go build
$ ./tsbs_generate_data --use-case="cpu-only" --seed=420 --scale=32
 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-06-01T00:00:00Z"
 --log-interval="10s" --format="influx" > TIMESERIES_DATA_SCALE_32
$ ./tsbs_generate_data --use-case="cpu-only" --seed=420 --scale=320
 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-03-01T00:00:00Z"
 --log-interval="10s" --format="influx" > TIMESERIES_DATA_SCALE_320
$ ./tsbs_generate_data --use-case="cpu-only" --seed=420 --scale=3200
 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-02-01T00:00:00Z"
 --log-interval="10s" --format="influx" > TIMESERIES_DATA_SCALE_3200
```

```
$ ./tsbs_generate_data --use-case="cpu-only" --seed=420 --scale=32
 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-06-01T00:00:00Z"
 --log-interval="10s" --format="clickhouse" > TIMESERIES_DATA_SCALE_32
$ ./tsbs_generate_data --use-case="cpu-only" --seed=420 --scale=320
 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-03-01T00:00:00Z"
 --log-interval="10s" --format="clickhouse" > TIMESERIES_DATA_SCALE_320
$ ./tsbs_generate_data --use-case="cpu-only" --seed=420 --scale=3200
 --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-02-01T00:00:00Z"
 --log-interval="10s" --format="clickhouse" > TIMESERIES_DATA_SCALE_3200
```

Replace the end of the three `tsbs_generate_data` commands with the folder destination. Notice that we generate data for both InfluxDB and Clickhouse. We can reuse the data for InfluxDB for QuestDB and FrogFishDB. However, before we can use the data for FrogFishDB, we need to convert it to the proper format.o

In the scripts folder is a Python file which is used to prepare the InfluxDB data for ingestion into FrogFishDB.

```
$ cd Thesis/scripts
$ source bin/activate
$ python PrepareBenchmark.py TIMESERIES_DATA_FILE NR_CLIENTS SCALE BATCH_SIZE
```

Replace `TIMESERIES_DATA_FILE` with the location of the InfluxDB data file, replace `NR_CLIENTS` with the number of clients to be tested, replace `SCALE` with the scale factor (e.g. 32, 320, or 3200), and replace `BATCH_SIZE` with the size of each batch. Note that the batch size will have to be rounded to a multiple of the scale factor.

8.1.1.4 Ingestion experiments

The ingestion experiments need to run in each of the VM's seperately. For InfluxDB, QuestDB, and Clickhouse we use TSBS for the performance measurements. Copy the TSBS folder cloned earlier over to the virtual machine through a tool such as SCP.

Build the benchmark tools using these commands:

```
$ cd tsbs/cmd/load_influx
$ go build
```

Replace `influx` in the first command with the desired database, so either Clickhouse or QuestDB.

The TSBS benchmarking tools are invoked as follows:

```
$ cat TIMESERIES_DATA_SCALE | ./tsbs_load_influx --workers=NR_CLIENTS
 --reporting-period=1s --batch-size=BATCH_SIZE > result.csv
```

Running a load command will generate a csv file with the results. As the result file will be overriden and in preparation for the data processing step, move each file to the scripts folder and rename them into `database_{nr_clients}_{scale}_series.csv`.

To measure FrogFishDB there is a script in scripts folder. The `RunTests.sh` script will generate the required client configs, start the benchmark, and collect the results.

To prepare FrogFishDB there is a configuration file which is required to launch the database. The `DB` folder contains an example configuration file. `FrogFish.toml` contains all the settings which are configurable by the user.

Another setting, one which cannot be changed from the settings file is the size of the memtable. This setting can be altered in the `FrogFish.hpp` source file. In there is the `bufSize` variable which denotes the size in the number of bytes. Changing this variable does require the database to be recompiled.

All of the ingestion bandwidth tests are run using the `RunTests.sh` script. This script will run all of the tests and move the files to the scripts folder. These do have to be moved by hand in order to make sure they are not overwritten by running the script again after changing one of the settings.

To measure the performance of TimeTree, navigate to the TimeTree repository. In the build folder is an executable called `bench`. Running this executable will generate files containing the results which we can process using scripts located in the scripts folder of the Thesis repository.

8.1.1.5 Memory usage experiment

Testing for memory usage requires the same procedure as described in section 8.1.1.4. However, to generate the memory statistics, the FrogFish database has to be launched using the valgrind massif tool.

Using the tool, the database is launched using the following command:

```
$ valgrind --tool=massif ./source/FrogFish
```

Move the resulting massif files to the scripts folder and rename them to the `massif-SCALE`, where `SCALE` should be replaced with the scale which has been used during the test.

8.1.1.6 Query experiments

In the scripts folder, there is the `TestQuery.py` python script. This script will issue the queries to the database. The database will then record the latencies in `query_latencies.csv`. This file can be processed using the `ProcessQueryLatencies.py` python script to generate the query graphs.

8.1.2 Processing the data

After collecting all the data, we can generate the graphs used in this thesis.

Generating the ingestion bandwidth graphs is done using the ProcessMultieClientStats python script in the scripts folder of the Thesis repository. However, before we can process the statistics for InfluxDB, QuestDB, and Clickhouse we need to remove unneeded leftovers from the output of the benchmarking tool.

The following command removes the last couple of lines from the CSV files which do not contain any actual data but are statements about the different worker clients used during the experiment.

```
$ for DB in influx quest clickhouse;
  do for client in 1 2 4 8 16;
     do head -n ((-4 - ${client})) \
        ${DB}_${client}_32_series.csv > ${DB}_${client}_32_series_fixed.csv;
     done;
  done
```

To generate the graphs, we use the following Python scripts:

```
$ python ProcessMultiClientStats.py
```

To process the memory usage results, run the following script to generate the memory usage graphs:

```
$ python ProcessMemoryUsage.py
```

8.2 Clickhouse patch

The following code sample contains the diff required to fix the TSBS tool for the Clickhouse database. The patch must be applied to the `pkg/targets/clickhouse/creator.go` file.

```
146c146,148
<                           ) ENGINE = MergeTree(created_date, (tags_id, created_at), 8192)
---
>                           ) ENGINE = MergeTree
>                           PARTITION BY toYYYYMM(created_date)
>                           ORDER BY (tags_id, created_at)
183c185,188
<                           ") ENGINE = MergeTree(created_date, (%s), 8192)",
---
>                           //") ENGINE = MergeTree(created_date, (%s), 8192)",
>                           ") ENGINE = MergeTree\n"+
>                           "PARTITION BY toYYYYMM(created_date)\n"+
>                           "ORDER BY (%s, created_at)\n",
```