Literature Study: Timeseries Databases

Niels de Waal

2022-12-16

Abstract

Time series data, i.e data indexed by time is common place in workloads such as IoT and DevOps. Timeseries databases are a type of database which is specialized to be able to deal with this type of information. In this work we investigate the current state of the literature regarding timeseries databases. We identify four core pillars, ingestion, storage, querying, and processing, which together build timeseries databases which in turn we investigate each. This investigation has enabled us to identify the problems and challenges associated with these pillars.

1 Introduction

As the amount of data generated by the human race keeps increasing, there is a large benefit to using specialized databases. Specialized databases can better keep pace with the velocity of the data generated [1]. Specialized databases utilize a specific focus on a type or feature of data they are storing [2]. Timeseries databases are an example of databases which focus on a specific type of data, and provides optimizations based on the inherent features that accompany this type of data. Timeseries databases are databases where data is stored in pairs of timestamp and value(s). This type of database provides optimizations for workloads where data needs to be indexed according to these timestamps. While storing this type of data in a traditional relational database is certainly possible [3], timeseries databases actively optimize for this workload.

The databases are mainly queried along their time axis. This leads to queries like: "what is the cpu usage for server X between 09:00 and 12:00", or "what was the maximum memory usage between 15:00 and 16:00 across all servers".

While there are many data generating workloads which can be handled by timeseries databases, in this paper we focus on two of them, as most literature falls within one of these two workloads. The first is DevOps. When deploying a large fleet of servers, system reliability engineers will want to closely monitor these servers, and not just the servers themselves, but also the applications they host. This means in practice that these engineers monitor a set of counters which indicate the health of the system. These counters are generated using a predefined set of queries, which often run on a scheduled basis. Timeseries databases are able to provide for such workloads, engineers can monitor recorded statistics, execute queries to investigate problems, or setup notifications to warn on of anomalous behavior. One can image that running 50 servers, which each generate and report a 100 metrics every 5 seconds, ends up being a lot of data.

The second workload is related to IoT data. Here we see that deploying a large number of sensors can lead to a large amount of data [4]. This follows a similar pattern as in the previous example, deploying say a 1000 sensors which each report 5 readings every 5 seconds leads to the same amount of data as in the previous example. The main differences here are how the end users look at the data. Companies might use this data for analytical workloads, such as analyzing the performance of a large scale factory. [5] shows a larger factory where machines report their operations and health to a control center where engineers monitor the facility. This data can also be used to find possible areas of optimizations.
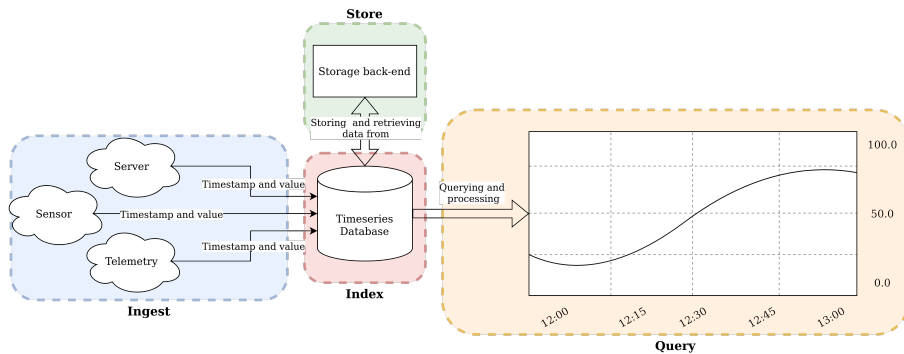

Figure 1: An overview of different systems interacting with timeseries databases.

Being able to efficiently store, index, query, and process this much data quickly becomes a non-trivial task when scaling the amount of data. The storage needs quickly become a challenge when not employing workload-

optimized compression to reduce size. When having to store 16 bytes per measurement, at 1000 measurements/second (as in the earlier example), for up to 6 months, uses ~ 250 GB of storage. If we scale this up to 150,000 measurements/second the storage requirements grow up to 23TB. While these figures are here for illustrative purposes, these ingestion rate figures are not unheard of, as demonstrated in [6]. Such is that even with a sufficient storage back-end, searching for a specific range of time with this much data becomes challenging to do in a reasonable amount of time, without some data structure which has indexed the data. Such a data structure would also enable faster queries. Finally, there is the problem of ingesting this at this data rate. An indexing data-structure also needs to allow for such high insertion rate.

Other scaling problems have to do with the processing of data. Users will likely want to do some form of processing [7], such as finding the maximum value or creating a histogram. In our earlier example we discussed system reliability engineers wishing to find abnormal behavior in systems, or an IoT company wishing to see how well an update performs, in both cases the engineers need to process data before analysis can take place.

There already exist a number of timeseries databases which attempt to provide a solution to these workloads [8] [9] [10] [11]. Most of these have variations which makes it hard to see which database is able to fulfill which role. Which is able to be a good fit for a system reliability engineer? Another problem is that it can be difficult to compare performance. Different databases have different limitations which do not show up easily unless we have a form of standardized benchmarking which can test different aspects of databases.

This report will serve to investigate the current state of timeseries databases by analyzing the existing literature and attempting to answer the research questions. We first look determine exactly how we wish to conduct our research, after which we investigate a couple of areas which we found to be the core pillars of this type of database.

2 Study design

This section will detail the goal of this study, and the method, such that it can be reproduced.

2.1 Research goal

This survey aims to get an overview of the existing timeseries databases, and how they solve different problems as well as how they can be integrated into different data analytical workloads.

In order to accomplish this goal we formulate the following main question for this survey: "What optimizations do timeseries databases use to provide for workloads which use data indexed by time?". From this main question we define the following sub-questions:

- **RQ1: What roles do timeseries databases serve?** While we have stated a couple of roles which are fulfilled by timeseries databases (i.e DevOps and IoT), the literature we find during this investigation might provide insight into more roles which a timeseries database can fulfill.

- **RQ2: What techniques are used to process data stored in timeseries databases?**

  We described in section 1 how it is not uncommon to store large quantities of data. We wish to find storage techniques which would help us deal the large amounts of data. Such as having statistical values ready when queries arrive which make use of these statistical values.

- **RQ3: What indexing techniques are used to index the stored timeseries data?** In section 1 we also mentioned how it could be a problem to index the large amount of data. This indexing is used to retrieve data more quickly during a query after storing.

- **RQ4: How does data get inserted into timeseries databases?** Data needs to get inserted into the database, is this done using traditional SQL, or are there different methods which might provide extra benefits.

- **RQ5: What storage back-ends are commonly supported by timeseries databases?** There might be different storage back-ends in use. Which can we find?

- **RQ6: What forms of compression are used on timeseries data?** With the amount which could possibly be stored, we should use compression to lessen the storage requirements. What forms and methods are in use in the literature?

- **RQ7: How do stream processing techniques interact with timeseries databases?** Data which is retrieved from a timeseries

database might need to be processed before the user wishes to view the result. Or data might need to be processed before it is inserted, this sub-question attempts to find different techniques for either stages.

- **RQ8: How does one properly benchmark and evaluate different timeseries databases?** What benchmarking techniques can we find in the literature? Different authors might design different benchmarks, what factors are common and which do we need to test the databases?

2.2 Scope

This work attempts to cover recent contributions, which we define as work published between 2010 and 2022. We will look at contributions towards storage techniques, data processing, and usability. We will have a focus on performance, for both ingestion and querying. Based on this we can setup the following criteria:

- I.1: This work has a focus on time indexed data storage or processing.

- I.2: This work is from between 2010 and 2022, papers from before 2010 are only included if they are fundamental work. Meaning that they are only relevant for historical context.

Papers which do not adhere to these requirements might still be included, however when done so, it will be explicitly mentioned as to why we've done so.

2.3 Methodology

This survey will follow the *snowball* method [12]. We start with a small collection of seed papers, from which we attempt to find more relevant papers by looking at the sources and the papers which have cited these seed papers. By looking at the sources we can go backwards in time and find papers which have served as the foundation for the seed paper, while by looking at the citations we can see which papers have been build from these seed papers. All papers in this survey have been discovered through Google Scholar.

Listed below are the seed papers used for this survey:

Paper name	Source	Publication year
Gorilla	USENIX	2015
BTrDB	ACM	2016
Monarch	ACM	2020
Druid	ACM	2014

For the seed papers we have limited the scope to the following conferences: USENIX, VLDB, and ACM SIGMOD. However, even though these works have served as good basis for this survey, they have not been able to serve as a basis for answers on all the questions. Thus we have also conducted a search through *Google Scholar* in order to find more. The following list of keywords have been used to find more relevant works for this survey:

- Timeseries database benchmarking

- Stream processing

for this survey, 40 papers were considered, 20 have been used, and 5 were discarded as they did not adhere to the criteria.

3   Background

This section will contain background information on what we define as timeseries databases and their usecases. We will go on to list and define some of the more well known existing implementations.

3.1   Timeseries databases

Timeseries databases are not a new type of database, they have been around for a long time. One of the earliest successful tools is [8]. These databases attempt to deliver an optimized environment for storing data which is indexed by timestamps. These databases are optimized to deal with the defining characteristics of timeseries data. One such characteristic is how the amount of writes vastly overshadows the amount of reads. Compared to what traditional relational database management systems (henceforth RDMS) such as MySQL are optimized for, timeseries databases can outperform those in terms of insertion rate [1]. Other characters are:

- The data is **append-only**. There are no random writes.

- Data is **immutable** after it has been inserted. There are no operations to support modifications.

Timeseries databases are a type of NoSQL database. NoSQL databases are one which do not have the tabular relation between data as found in RDMS's. While the concept is not new, the term is relatively recent [13].

When returning to our original examples this quickly becomes clear. If we consider the example of the 50 servers it is easy to imagine that the amount of data quickly grows beyond the amount required for simple queries. Such as returning a cpu usage metric for the last 6 hours, without applying any filters or statistical functions. The same holds for the IoT example, as the number of deployed sensors grows, so does the amount of data generated.

Databases in general have, over the years, come up with different methods of scaling. One example is horizontal scaling where a database instance is ran on multiple nodes at the same time. An example of how this can be accomplished is through sharding, meaning that a subset of data is directed towards a specific node and that all queries for that subset end up at that node. Timeseries sharding could for example be accomplished by taking the name of the series (e.g `server_a/cpu_usage`), hashing the name, and map the hash to one of the database instances.

The main goal of databases is to store data in a structure manner such that the data can be retrieved later on. With timeseries database this retrieval is going to involve a range of time, we call these **range queries**. We proposed a quick example in section 1 on what these queries look like. They query the database for a range of time.

3.2 Data storage

After being inserted into the database, the data needs to be stored somewhere. The data can be stored either in main memory (in-memory database) or on persistent media (on disk database). For the disk based storage there exits a multitude of different back-ends. Two examples of such back-ends are filesystems and block stores [14].

We define a filesystem as a system in which files are stored and referenced by name, they can be archived into folders, which also have unique names. Block stores are data stores in which data is stored in blocks of data. These blocks can then be referenced by a unique id, this id is generated for by the storage layer, and is unique to the data which the user has stored.

3.3 Data compression

When storing large amounts of data, it is beneficial to have compression techniques. These can help reduce the amount of data which needs to be

stored. There are two distinct types of compression, lossy and lossless. With lossless compression we do not lose any information after we have compressed it down. With lossy however, we do lose raw data, however some schemes are able to get away with this as they still retain a close enough approximation of the original data.

Timeseries data has a few unique properties which can be used in order to aid in compression. The first property is that a lot of data gets ingested at periodic intervals. An example of compression making use of this fact is *delta-compression* [15], where instead of storing the raw values, only the difference between those values is stored. For example, instead of storing 2, 8-byte timestamps, delta-compression would enable one to store 1, 8-byte timestamp, followed by a 2-byte delta, resulting in a $\sim 60\%$ reduction.

3.4 Data processing

Storing timeseries data is always with the intention of processing it at a later point in time. When site reliability engineers for example want to monitor the latency of a service they be more interested in the 99th percentile latency than just the raw latency figures. Thus it is very useful for engineers to be able to apply different transformations on the data they are reviewing.

These transformations are done through queries which can be written in a querying language such as SQL. Such a language provides built-in functions such as `min`, `max`, and `avg`. These languages fall under the category of *domain-specific language* (DSL). A DSL is used for specific tasks and cannot be used for general purpose computing. An example of such a DSL is SQL (structured query language). This language is used by databases in order to specify a query on the data that is stored in the database.

There might also be cases where data has to be processed before being inserted into the database. This processing is be done to, for example, summarize data or calculate the aforementioned statistics. This is what we define as **down-sampling**. It is the practice of summarizing data using a function to create a new timeseries.

Previous survey's have investigated areas such as stream processing [16] and data mining [17]. We define stream processing as the act of transforming data while it is in transit, while we define data mining as the processing of attempting to extract information from a pool of data. Both stream processing and data mining are an important part of how users interact with timeseries databases. Through these interactions is how users find information stored in the data beyond simple statistical information. Such as the example given in 1.

3.5 Benchmarking

In order to find differentiate different databases in terms of performance we need proper benchmarking. Differentiation is complicated by the fact that different databases might excel in different workloads or under different scenarios. As also stated in the conclusion of [18] where baderSurveyComparisonOpen2017 present an overview of existing timeseries database. They note that the lack of a standardized benchmarking framework as they could no long differentiate different databases which have feature parity, so they attempted to differentiate on the basis of performance.

[19] discuss two different benchmarking datasets which can be used to measure the performance of different relational databases under different scenarios and workloads.

For example the YCSB ([20] and [19]) benchmark includes generators which can simulate differently skewed workloads, such as update heavy or read only.

4 Ingestion

In this section we will discuss the different methods databases use for ingestion. This relates back to RQ4 (*How does data get inserted into timeseries databases?*), we wish to explore the different ingestion methods used.

We have found different ingestion methods, which can roughly be divided into two different categories. One where the database contains a subsystem which retrieves information from specified targets (**Pull based ingestion**), and one where data is streamed to the database by the applications (**Push based ingestion**).

Thus in this section we will look at two different databases which each use one of the two methods, namely *Prometheus* and *InfluxDB*. Both of these are heavily used by in the industry [21], and should such serve as good examples of the different ingestion methods.

2 shows the different methods of ingestion in a single and simplified overview. What is important to note is that sources, MQ, and databases do not have to represent singular entities, they could represent a distributed setup, however, for illustrative purposes these complexities are left out.

4.1 Pull based ingestion

Prometheus is a heavily used example of a pull based system [21]. This database consists of a couple of distinct components, one of which is called
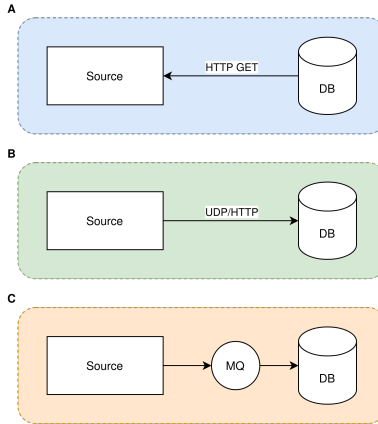
Figure 2: The different methods of ingestion. (A) shows pull based ingestion, (B) and (C) show push based ingestion. (C) ingestion is routed through a message queue (MQ), e.g Kafka.

the *Scrape Manager* [9]. This service retrieves statistics and other metric data from different third-party systems. The retrieval is done through HTTP `GET` requests to the third party system. The third-party system exposes an HTTP endpoint such as `http://service:6900/metrics` and replies to the `GET` request with a list containing pairs of metric names and values. These metrics get forwarded to the storage layer of *Prometheus*.

The scrape manager collects metrics from the targets which have been configured. The targets are discovered through the *Discovery Manager*, which identifies possible targets as they are defined in the configuration files.

With this pull based configuration, Prometheus is more suited towards acting as a metric based alert system. It can be very useful for system administrators to collect data and be notified in the case of some unexpected behavior. Due to the pull based ingestion design a timeseries database is able to detect a complete failure of a particular service or server due to the fact that during such a complete failure it is no longer able to retrieve metrics.

The alerting system is also why pull based ingestion might be more useful in a DevOps/system reliability setting. In that case the database itself can act as the supervisor and push notification out to the responsible engineers. The obvious downside is that scaling becomes harder as now IO is a significant part of the ingestion pipeline. A database has to go out and gather metrics itself instead of waiting for them to arrive.

Another example of a pull based timeseries database is *VictoriaMetrics*.

During our survey, we have not been able to find more examples.

4.2 Push based ingestion

InfluxDB is another commonly used timeseries databases which can ingest data in a fairly straightforward way. This database exposes a UDP port which is then used by applications to push data into the database. The data is structured according to a wire-format which is parsed by InfluxDB and written to the specified database. The port which is exposed is used by all applications wishing to write data into the database, and can thus be seen as a point of congestion.

Writes to the database are collected into batches. These batches are then compressed and written to the write-ahead-log (WAL). Batches are also written to an in-memory cache, making them immediately available for querying, and the cache is periodically written to disk.

Other examples of push based ingestion are databases which are subscribed to a topic on a message queue ((C) in figure 2). This topic streams, for example, cpu usage statistics. A good example of such a database is Druid [22]. In this database the authors propose a design in which the different components are more separated, meaning that the different components are different processes, and could for instance run on different nodes. One of these components is a *Real-time node*. This node is responsible for ingesting the new data and forwarding it to the rest of the system. With the different components running as different processes a zookeeper instance is used to provide for communication between the different components. The main upside to this distributed architecture is that individual components can fail without rendering the entire system unavailable.

This method for ingestion is very useful when databases have to collect a larger amount of data. Using multiple *Real-time node* instances, ingestion can be spread across the nodes in order to spread the load. Another benefit of this setup is that the *Real-time nodes* can batch the incoming data, leading to higher network throughput, when, for example, recording IoT sensor data where the latency is not a primary concern.

4.3 Data insertion

Using the previous two sections we can answer RQ4 (*How does data get inserted into timeseries databases?*). We have defined three distinct methods for inserting data into timeseries databases. The first is pull based ingestion, this is where the database itself has a component which retrieves data from

pre-configured targets. The second is where data gets written or streamed to the database by the different applications wishing to write to the database. The third and final method is a combination of the previous two. Using a message queue data is streamed from the writers to the database through a queue, where the queue is read by the database. However, we have only encountered this third method in one database.

5 Storage

Table 1: Questions answered in this section

Question number	Research Question
RQ3	What indexing techniques are used to index the stored timeseries data?
RQ5	What storage back-ends are commonly supported by timeseries databases?
RQ6	What forms of compression are used on timeseries data?
RQ8	How does one properly benchmark and evaluate different timeseries databases?

In this section we aim to provide answers to RQ3, RQ5, RQ6, and RQ8, while the first three relate to either storing or indexing data, the last one relates to the performance of the first three sub-questions. We start with an investigation into in-memory and on-disk databases and follow this up with different indexing structures and compression methods. Finally we close this section with different benchmarking techniques found in the literature.

We define two different types of storage methods for timeseries data. The first one is **on disk** storage, and the other is **in-memory**. While most databases have an in-memory cache and staging area, in-memory databases exclusively store and retrieve data from memory and use disk storage for archival purposes.

5.1 In-memory

We define an in-memory database as one which primarily stores its data in memory. The databases discussed in this section, while being classified as in-memory databases, do use persistent storage for archival purposes.

In the *Gorilla* paper [23], Facebook introduces its in-memory timeseries databases. *Gorilla* is build such that it could store the system metric values

of servers, while also allowing for fast querying and high availability through replication. The reason they chose to use memory as storage is two-fold, the first is that they noticed that 85% of the queries that the previous system processed involved data collected which had been collected in the preceding 26 hours. This meant that realistically only recent (newer than 26 hours) data has to be ready to be queried. The second reason is the fact that the database could serve as a cache for the persistent disk-based storage, due to the database residing in-memory, meant that they could reach their targets for insertion speeds (700 million datapoints per minute or $\sim$ 12 million per second).

In *Monarch* [24] a workload was presented, similar to that of the one for which *Gorilla* was designed: a distributed timeseries database geared towards storing system metric values. The authors required strong consistency, however, directly writing to a store such as *Spanner* [25] was deemed to costly, however, the authors do not specify the cost figure. *Spanner* is Google's replicated and distributed database, however, writes to *Spanner* incur a very heavy latency cost as those have to go through the replication algorithm PAXOS [26]. This cost increases the mean-time-to-detection and mean-time-to-migration, meaning that it takes longer for issues to be found and mitigated. Combined with the requirement of a low number of dependencies, the choice was made for in-memory storage. The added bonus of data residing in memory was that alerts could be delivered promptly as there would be no IO cost for first having to write to either a WAL (*write-ahead-log*) or disk.

Both of *Gorilla* and *Monarch* are distributed databases. *Gorilla* is distributed by the use of sharding [27]. Timeseries are distributed across shards based on a key used to uniquely identify a timeseries (although it is unclear whether this is a name or some identifier like a UUID). In the case of outages, a *ShardManager* is used to reassign shards to different nodes. The *ShardManager* picks nodes such that it optimizes for resiliency. By maintaining two independent nodes in two different datacenters, *Gorilla* is protected against most forms of network partition and hardware failures.

The authors of *Monarch* take a more complex approach. Each datacenter has ingestion routers which receive data from clients across the datacenter. These routers find the correct *leaves*, which are the nodes which store the timeseries data. Distribution is done using the aforementioned *leaves*, *leaves* are always part of a much larger zone with multiple replicas. Ingestion performance is maintained by offloading the replication to the ingestion routers.

During this survey these are the only papers we have found which fit the requirements we defined earlier. For the papers we discussed in this section

we could argue that they technically are not in-memory databases as they are backed by persistent storage. *Gorilla* is backed by *GlusterFS*, as POSIX-compliant distributed filesystem, while *Monarch* is backed by *recovery log* files, however, the authors omit any details on what these are in order to save space on the paper. What can be determined from both of these papers is that in-memory databases can fulfill a large amount of requirements, ranging from high availability through the use of replication, to high insertion rates. We can offer a guess as to why memory is rarely used as primary storage, it is very cost prohibitive. [28] shows this too, while the cost per GB has come down, it is still orders of magnitude higher than that of traditional storage media, such as HDD's and SSD's.

5.2   On disk

The databases covered in this section use persistent storage. While these databases can still use memory structures for caching timeseries data, their primary storage method is disk based. The databases covered in this section use memory as either a staging ground in before flusing to disk or as a simple means of caching.

During our investigation we identified two methods databases employ to use persistent storage. The first is local storage, this could be local drives or some local filesystem, while the second is remote block storage focused. Stores such as S3 or MinIO are used as storage back-ends for these databases.

For the first method we find papers such as [29] and [30]. Both of these databases first cache data temporarily in memory before flushing to disk. For second method we find ones such as [31] and [22].

Whether one chooses local storage or remote block storage is a matter of preference, both have their strengths and weaknesses. One such example is that a block stores allow for easier access to the stored data. As chunks or groups of data are collected and stored in a block store, only a single address is required to retrieve this data again. This address can then be stored in some sort of indexing structure. It also makes it so the user can store relatively smaller chunks of data, instead of having to group into larger blocks, which is beneficial as in that this lessens the overhead from the IO stack of files having to be opened and read. A downside of block stores is that the latency is a lot higher due to the extra overhead of the network. [32] shows that SSD's can do 4KB random reads with a latency of $\sim 20\mu s$, while a latency optimized store such as IndigoStore [33] shows latency's two orders of magnitude higher ($\sim 2000\mu s$).

With this we can answer RQ5 (*What storage back-ends are commonly*

supported by timeseries databases?), we have discovered the common storage back-ends for timeseries databases. There are two types, the first of which is using memory as a storage back-end, the second is using either local storage such as a filesystem or a remote focused block storage. Using memory as the primary storage mechanism boats higher performance, but has a significantly higher cost associated, making it impractical for all but the largest commercial entities.

5.3 Indexing structures

When storing large amounts of data, it becomes more difficult to efficiently find data after it has been stored. Keeping track of possibly terabytes of timeseries data is a hard problem and one for which multiple solutions have been presented. What makes this particularly difficult is that keeping a direct mapping from timestamp to value in memory is impossible when the size of data grows beyond the amount of usable memory. Indexing structures are data-structures which are used to *index* the data and be able to find data in less than $\mathcal{O}(N)$ time complexity. Other than being able to find data, indexing structures also help in accelerating queries. An indexing data-structure might allow for fast range based queries, as in that it could point to the start of the requested range and allow the database to quickly scan for the relevant records. From this we aim to answer RQ3 (*What indexing techniques are used to index the stored timeseries data?*).

The authors of *Gorilla* present *Timeseries Map* (TSMap). While *Gorilla* uses memory as its primary storage, it still suffers from the problems of scaling to possibly petabytes of data. Finding the queried ranges of time in such a vast amount of data would lead to an unacceptable query latency. Scanning linearly, even at the maximum theoretical bandwidth [34], would take multiple seconds or minutes. TSMap is build from a vector containing C++ shared-pointers to timeseries data and a case-insensitive, case-preserving map from timeseries name to timeseries data. The vector allows the authors to quickly scan the data for the queried range. The authors claim to be able to scan in a few microseconds, this to make sure that the flow of incoming data is not impacted by users performing queries on the data.

The use of LSM (Log-structured merge) trees [35], or derivatives thereof, is common practice for the indexing structures of timeseries databases which use persistent storage. An LSM tree is a data structure which contains key-value pairs. The structure maintains multiple data structures which are each optimized for their respective underlying storage medium. oneilLogstruc-

turedMergetreeLSMtree1996 propose two structures, where one which resides in memory and the other on disk. In the version proposed by oneil-LogstructuredMergetreeLSMtree1996 the values stored on disk are sorted by their keys. This is very beneficial for timeseries data as this data could be sorted on timestamp, making for easy range queries.

Examples of timeseries databases using LSM trees include *InfluxDB*, *OpenTSDB [10]*, and *IoTDB* [36]. LSM trees are used for their high write performance and support for range based queries. An example of a database using LSM trees is [37]. This database stores timestamp-value pairs together with tags (tags are key-value pairs which can be assigned to timestamps and can be used during querying as a filter, for example `server_location=Dublin`). *InfluxDB* stores these LSM trees into files which are loaded into memory during queries in order to improve performance.

Timon [29] uses LSM Trees directly in order to index the SSTable files. SSTables [38] are ordered immutable maps which map keys to values, in the case of *Timon* timestamps to metric values. Because these files are ordered they allow for fast queries based on a range. When a range is queried, only the starting point has to be found, then the rest of the range follows until either the end of the range or the end of the file. The SSTable files contain multiple timeseries and are compacted at regular intervals. The files consist of a few different components, two of which form the *DataZone* and the Index section. The index section provides a fast lookup for different blocks in the *DataZone*. The *DataZone* is where the timeseries data is stored. cao-TimonTimestampedEvent2020 state that the amount data in the *DataZone* can be too large enough that a binary search is too expensive.
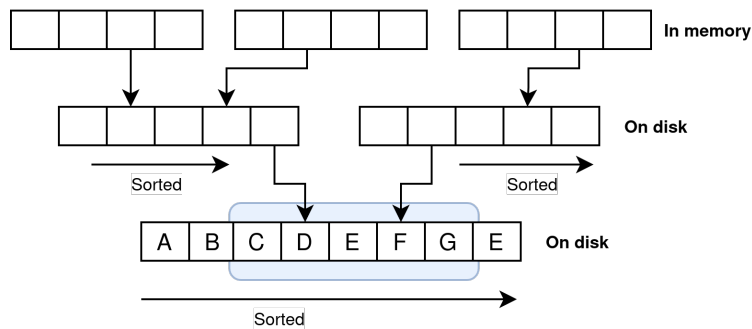

Figure 3: Simplified range query on LSM tree

Figure 3 shows a simplified LSM tree. Here we wish to view a range of C up to and including G. This means with an LSM tree that, when we find

C, we can simply continue to read from storage until we find G, as all keys are stored in a sorted order, i.e by timestamp.

In [31] andersenBTrDBOptimizingStorage2020 propose an indexing structure which allows them to ingest data from a smaller amount of sensors at a higher frequency, calling it BTrDB. They found that existing databases such as *Druid* did not allow them to record data at a high frequency with timestamps accurate to 100 nanoseconds. This forced them to create their own write optimized tree structure. A *time-partitioning copy-on-write version-annotated k-ary tree* is what they designed as a solution. The most notable difference between this tree and the LSM trees from *Timon* is that this tree does not split the tree according to the key space, but it splits according to the time space. What this means is that each child is a subset of the time range represented by its parent. In order to further improve ingestion performance, *BTrDB* uses batching. This means that a larger batch of values is first collected in memory before being written into the indexing structure.
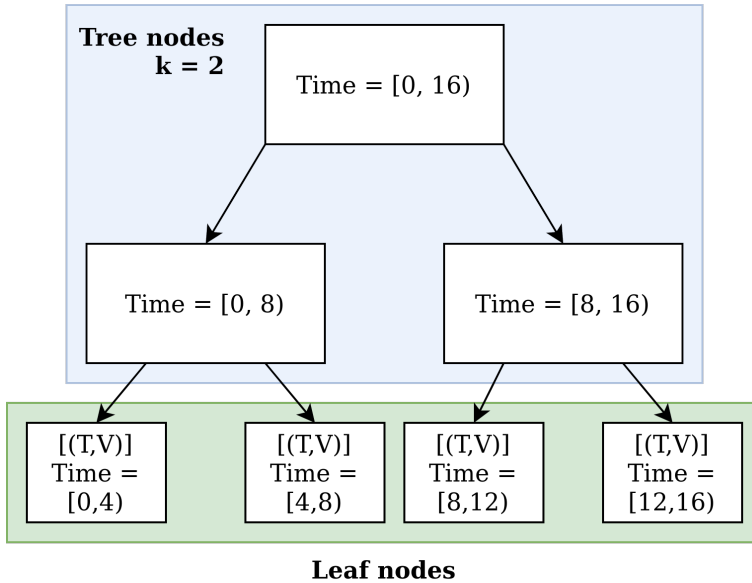

Figure 4: Simplified overview of the BTrDB indexing structure

Figure 4 shows a simplified overview of the tree data-structure used in BTrDB. It is important to note here that the leaf nodes contain the actual data points and that the tree nodes contain the intermediate figures such as `min`, `mean`, and `max`.

The authors of *Timon* present two different benchmarks for showing the

performance of both *Timon* and *BTrDB*. They show both ingestion rates as well as query latency. While both of these do not directly equate to the performance of the indexing structure (there are also IO costs to consider, like writing to disk and receiving values over the network), they give a good indication of the effectiveness as the performance both ingestion and querying is reliant on the indexing structure being fast. For ingestion this means that data needs to be inserted fast, and for querying this means that range lookup's need to be fast. Figure 15 and 19 of the *Timon* paper show the performance of ingestion and querying. From this we can see *Timon* has lower (800ms vs 2100ms) query latencies, while *BTrDB* has higher (36 million vs 20 million at a batch size of 50000) ingestion performance when increasing the batch size.

The databases discussed in this section use different methods for indexing timeseries data. We described a distinct difference in how in-memory databases index data compared to databases which use disk based storage. In-memory databases make use of a more direct mapping where timestamp-value pairs are stored directly in memory pages, while on-disk databases opt for the use of tree structures.

Considering all of this information, we can answer RQ3. All databases which we have found during this survey use on-disk storage use some form of a tree structure, while the in-memory database s use a more direct mapping, such as presented in *Gorilla* with TSMap.

5.4 Compression

Compression is very useful for timeseries databases as we previously pointed out in section 1. Luckily timeseries data provides some characteristics which can be exploited for the sake of compression. Such characteristics often relate to subsequent data being either the same of similar. The goal is to provide as high a ratio of bytes compressed to bytes stored as possible without compromising the ingestion performance of the database. Compression adds overhead to the ingestion performance as the compression operations require computational resources. With this section we aim to address RQ6 (*What forms of compression are used on timeseries data?*).

Gorilla [23] popularized two types of lossless compression. The first type being *delta-of-delta* compression. This is an alteration of *delta-compression*, where the periodic nature of timeseries data is exploited further by realizing that that most timestamp-value pairs are inserted at a specific rate. By taking the delta of the delta of the timestamps, the authors capture not the delta between timestamps but the jitter in delta between timestamps.

As most values are inserted at a set interval this value should be small, capturing, at most, small variations. See figure 5 for an example of both delta-compression and delta-of-delta compression.

Timestamp	Delta encoding	Delta of Delta
16:00:00	-	-
16:01:00	60	60
16:02:02	62	2
16:03:01	59	-3

Figure 5: Example comparing delta compression and delta-of-delta compression

Figure 5 shows an example where 4 timestamps are compressed using both *delta-encoding* and *delta-of-delta*. What is important to note is that using delta-of-delta encoding we can store the values using far fewer bits as the values themselves are smaller.

The second type the authors introduced is XOR compression. This type exploits the situation where a floating point value is close to or the same as the previous value. XOR compression exploits the IEEE-786 [39] representation, in that when subsequent values are very similar, then the XOR of the two shows which bits have changed and most likely fall within the same range, making for an encoding scheme where only the required bits are stored. This way the authors were able to identify three different cases. The first is where the values are equal and can thus be stored as a single bit, the second is where the values are similar enough that the value can be compressed into a couple of bits, preceded by two control bits, and the last case is where the values differ enough that more data has to be stored, also preceded by two control bits.

The authors point out in figure 5 of the paper that 51% of the values is reduced to a single bit (values are the same). 30% of the values fell into the

first case of the second option (values are very similar), and the remaining 19% of the values are compressed with the '11' control bits, meaning that the values differ greatly. The authors further go on to show that for a two-hour block of values, the average compression ratio came out to be 1.37 bytes per datapoint.

This scheme is further evaluated in [30] where the authors of *Heracles* note that a starting value can have a noticeable effect on the effectiveness of XOR compression. As such they chose to use 25th and 75th percentile. However, the authors also seem to have made no effort back up their claims of improved efficiency, either through a source or through measurements.

The authors of *Heracles* present their database as a fork of *Prometheus*. The main features of this paper are an encoding scheme and a memory management system. The encoding scheme is able to save extra bytes by grouping metrics with the same timestamps such that a single timestamp does not have to be repeated as seen in e.g *Gorilla* where each metric value is stored together with a timestamp. In *Heracles* metrics are grouped into tuples with the first item being the timestamp, this is more akin to what is found in classical RDBM's such as MySQL.

Other lossless options we find in different databases are one such as Snappy [40], ZigZag encoding [41] for converting singed integers to unsigned, and bitpacking. Snappy is used for string encoding, *InfluxDB* specifically mentions that they pack strings together and compress them as one block in order to achieve a higher compression ratio. Zigzag encoding is an encoding scheme which maps signed integers to unsigned integers, these can be more easily encoded using for example simple8b [42].

While the compression methods detailed here are fairly simple, none of the aforementioned papers, with the exception of *Heracles* go into detail into the computational costs associated. Neither in terms of what would happen to the ingestion performance when compression is not present or how much time in the hotpath is spend on compression.

wangHeraclesEfficientStorage2021 compare their implementation across different performance characteristics, such as throughput, cpu usage, query latency, and memory usage. In all metrics, *Heracles* is shown to either match or out perform *Prometheus*. *Prometheus* is tested using different storage back-ends, however, *Heracles* still outperformed *Prometheus*. For example, the p99.99 and maximum query latencies are shown to be 70% and 64% lower respectively. Data size reduction is shown to be 12% in-memory and 13% and 50% on average for on disk storage.

For lossy compression we can look at other papers. [29] is an example where the authors make the explicit choice to remove the accuracy of raw

data and retain only statistical values. The loss of accuracy was found to be bearable when compared to the gains found in efficiency due to being able to support *blind-writes*. Blind-writes are writes such that they require no strict ordering, allowing for out-of-order writes. Blind-writes hinge on a couple of mathematical assumptions about the operators they use, namely associativity and commutativity. This means in short that the order of operations does not change the outcome, e.g $a \times (b \times c) = (a \times b) \times c$. Another important benefit to supporting blind-writes is that out-of-order ingestion becomes a lot more cheaper, due to the commutative property. Here we define out-of-order as the scenario where metric values do not arrive in the same order in which they are recorded.

The operators which enable the blind-writes are ones as `sum`, `min`, and `max` plus a few others. In figure 13 caoTimonTimestampedEvent2020 show that out-of-order writes are orders of magnitude faster using blind-writes, namely 10x the performance of *InfluxDB* (23 million compared to 5 million data points per second). While this method is shown to deliver vastly improved performance statistics, even with out-of-order writes, as compared to other timeseries databases, the authors make no mention of how close these statistics come to the raw underlying values, and if they have since lamented on their decision of getting rid of the raw values.

Coming back to RQ6, we can define two forms of compression, lossy and lossless. However, the if this is a worthwhile trade-off can differ greatly depending on the workload of the database. If the workload requires the option to be able to query for individual datapoints lossy compression is not going to be sufficient, however, if the workload only requires users to be able to view statistical values based on the data which has been written to the database, then lossy compression can suffice.

5.5 Benchmarking

This section will focus on the different benchmarking systems found in the literature, in order to provide a partial answer to RQ8 (*How does one properly benchmark and evaluate different timeseries databases?*). For now we will focus on ingestion and storage performance, benchmarks for query performance are shown in 6.2. Most papers have concrete performance figures, but none appear to have standard benchmarking practices. The problem with non-standardized benchmarking is that it makes it challenging to compare across different paper. For example, the performance which *Gorilla* shows is very good, query latency in the order of single milliseconds and an insertion ratio of $\sim$ 12 million data points per second. However, the

authors provide neither the concrete dataset used nor the complete environment variables, such as the hardware specifications on which the benchmarks were conducted.

Papers such as *Gorilla* or *Monarch* only show the performance measured in production. Which, while showing good results, is neither transparent nor repeatable for the scientific community at large. The lack of transparency is most likely due to the commercial nature of the paper. Other examples of creating benchmarks from real-world performance are [43], [44], and [45].

In [22], [29], and [23] the authors present performance figures using data they recorded in production. While this style presents the real world performance of their work, it does not present comparable results. For example, pelkonenGorillaFastScalable2015 note the percentage of data were they found that a value was the same as the previous value, which could very well be false for different workloads.

The authors of [46] have addressed this problem by creating a framework to test timeseries databases with. This framework abstracts the interface of the databases and couples this with a data generation tool which can emulate different forms of data generation workloads, and follows up with different query workloads. We show a simplified overview of this in 6. This method tests both ends of the timeseries database workload, both ingestion and querying. However it provides no indication of storage efficiency or compression ratios. With the different queries the authors attempt to show queries which focus on, for example, data aggregation and down-sampling.
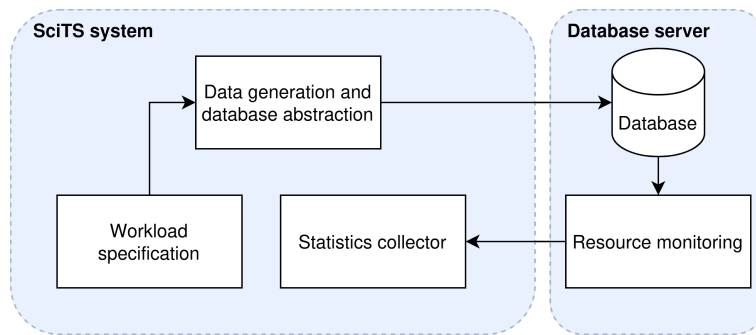

Figure 6: Simplified overview of the SciTS architecture

The ingestion and storage benchmarks of SciTS provide three different focuses. The first is testing the effects of batching. This is the process of grouping data points together before inserting, in order to improve performance. The second is testing the effects of concurrent writing. This can test

the effects of scaling the amount of clients writing to the database. The third benchmarks scaling the amount of data inserted into the database, testing the performance of the data when the size of the data keeps increasing.

mostafaSciTSBenchmarkTimeSeries2022 also make an example by comparing ClickhouseDB [47], *InfluxDB*, *TimescaleDB [48]*, and *PostgreSQL* [49]. Here they show that ClickhouseDB has the best scaling for ingestion, 1.2 million insertions per second for 48 clients. But also the lowest query latency for the different queries, being, on average for the 95th percentile, 50% faster than the database ranked second for that query.

Another initiative to get a standardized benchmark for timeseries databases is TSBS [50]. The goal is to emulate a couple of specific workloads, but at the time of writing only DevOps and IoT have been implemented. TSBS has implemented a lot more databases than SciTS has. TSBS contains a couple of different scripts which can test various aspects. Data generations scripts first generate data based on the aforementioned two workloads, using a seed value, allowing for repeatable measurements. Query generations scripts generate queries to be executed on the databases, these also stem from a seed value. The *load* scripts use the aforementioned generated data and queries to actually test and measure the databases.

For this section we consider the data generation tool meant to test the insertion and write performance. This tool generates a pre-configured amount of data in before using native clients to insert this data into the databases which the user is testing. The user is able to set the seed for the random number generator, this allows for repeatable experiments.

As we have covered only storage in this section we will only provide a partial answer for RQ8. We can see that there are multiple attempts are designing benchmarks which attempt to provide a standardized benchmark. Such a benchmark could be used by researchers to both test a timeseries database as well as provide performance figures which allow for comparisons between different timeseries databases. From the literature we have not been able to conclude that one of them succeeded in providing such a standardized benchmark, however *TSBS* does provide the highest number of integrations, it has the highest number of supported databases.

6 Querying

Querying is the process of retrieving data from a database. These queries can be enhanced with filters and different forms of processing. A filter makes it so only a subset of the data is returned, while the processing allows for

computing statistics in before returning them to the users.

In this section we attempt to complete our answer for RQ8 (*How does one properly benchmark and evaluate different timeseries databases?*)

6.1 Query DSL's

Databases providing their own query language instead of relying on something more traditional such as SQL is not uncommon, this is one of the defining features of NoSQL databases [2]. These have either their own special query language, or none at all and rely on simple operations such as `get` and `set` in the case of key-value databases [51].

During our research we found several timeseries database which provided their own query language. The common reasoning is that SQL is not ergonomic enough for the end users, such as reliability engineers. [24] mention some of the problems they encountered using SQL, namely the lack of a distribution (i.e histogram) datatype. Such a datatype would allow engineers to conduct more sophisticated statistical analysis. These new languages provide core constructs which were easier to use than SQL. These constructs for example, made it easier to execute range queries, or made it easier to gather statistics over a range of data such as p99 latency.

An example of such a DSL is Flux [52] for *InfluxDB*. The design language is such that it models a stream of data which can be processed, see the following example:

```
from(bucket:"telegraf/autogen")
  |> range(start:-1h)
  |> filter(fn:(r) =>
    r._measurement == "cpu" and
    r.cpu == "cpu-total"
  )
  |> aggregateWindow(every: 1m, fn: mean)
```

This example should be read top to bottom. It shows data being read from a bucket, with the start being 1 hour before the time of the execution of this query. This is then followed by a filter which filters on cpu statistics, and is finally combined into an aggregate window.

During our research we have not found any paper which investigated possible performance improvements using a custom DSL. We could image a case where a custom query planner could take advantage known the exact layout of the database.

The problem for benchmarking is that these languages are really hard to generalize. A general purpose benchmarking suite will have to adapt its query benchmarks to the different querying DSL's of different timeseries databases.

6.2 Benchmarking

Several papers have shown the performance of their database. In this section we will compare the different performance figures related to query latency.

We have mentioned both SciTS and TSBS before. Both of these systems also provide benchmarks for querying. For SciTS this is a collection of 5 predefined queries:

1. Fetch raw data

2. Query for the intervals where a sensor was acting abnormally

3. Aggregate a range of data into a single value

4. Down-sampling of data using a specific sampling function

5. Compare two down-sampled sensors using a comparison function

TSBS also provides querying benchmarks. Appendix 1 of [50] shows the different queries which can be executed. As we mentioned before, the random number generator can be seeded, this also effects the querying benchmarks. The queries are predetermined in terms of what the range of data is being queried, making the benchmarks fully reproducible.

With this we can complete the answer to RQ8. Earlier we covered benchmarks regarding storage and ingestion, however, now we can also draw conclusions regarding querying. SciTS provides the most complete model for testing query latency. The different queries test various aspects of the database, ranging from simple range-based queries to queries involving a lot of processing.

7 Stream processing and modeling

This final section will focus on modeling, stream processing, and use cases for timeseries databases. Stream processing is where a stream of data is processed by a series of operations which are performed on each element in the stream of data, before being forwarded to a different application, such as a database or user application. Some use it for managing and monitoring IoT

Table 2: Questions answered in this section

Question number	Research Question
RQ2	What techniques are used to process data stored in timeseries databases?
RQ7	How do stream processing techniques interact with timeseries databases?

data while others make use of this type of database for large scale monitoring of clusters and infrastructure. With this we hope to answer RQ2 and RQ7.

These can be seen as different as IoT can have a lower ingest frequency (only measuring every few seconds [4]), but a high degree of cardinality, and server monitoring can have a high ingest volume, but be fairly limited in cardinality.

There are several options to modeling using timeseries data [17]. These techniques and algorithms are agnostic to which database is used, so either, for example, MySQL or MongoDB. The main goal of stream processing is to extract some useful information from the data which is stored. In order to accelerate this finding some databases already store down-sampled data, such as averages. *BTrDB* [31] does this by storing *min*, *max*, and *mean* in each node in the indexing structure. This storing of intermediate values means that when a user queries the database, they can retrieve data at their desired granularity and use these statistics as an intermediary for the desired statistic.

[53] shows an example of using a combination of Apache Kafka [54] Streams and TimescaleDB. In this paper the authors show how meteorological data is collected into TimescaleDB, and is returned to the user through Kafka Streams. Kafka is used to transform the query result from a single large block of data to a stream which allows for the use of windowing (which Kafka Streams supports). However, as this paper focuses on TimescaleDB and not on Kafka, we are not able to see the added overhead of the Kafka Streams.

[22] shows the opposite approach. yangDruidRealtimeAnalytical2014 make the case of using stream processing for ingestion. This ingestion method has two main advantages. The first is fail-over, with a processing framework such as Kafka, there is always a queue. This queue holds messages and data which have not been processed yet. Using this queue means that the timeseries database could go down, without losing any data, as when the database is brought back up, then it could retrieve the unpro-

cessed data from the message queue. The second is that of scalability. When using a message queue, if the topics (i.e the name of the streams of data) can be sharded over the different databases instances, there is almost perfectly horizontal scaling.

This method of using a message queue is a shift in terms of responsibilities. The setup yangDruidRealtimeAnalytical2014 describe, of having separated processes, distributes the responsibilities across different nodes and processes. They make it clear that they needed a highly reliable system which could guarantee high query performance in an environment with a 1000+ users.

In *Heron* [55], kulkarniTwitterHeronStream2015 show the complexity which can result from the use of stream processing. During their workload they found that the existing solution was both inefficient and hard to debug. The existing solution was too reliant on a zookeeper instance to manage heartbeats from the worker nodes, which caused three major issues. The first is that the workers could interfere with each other in terms of performance, the scheduler managing workers did not support isolation and reservation. This lack of support meant that workers from different topologies running on the same machine could interfere with each other. This was mitigated by dedicating entire machines to a single topologies, which was identified as a waste of resources as the topology (a directed graph of inputs, i.e streams of data, and computations which are applied to the inputs) would not make use of all the resources available on that machine. The second was that the zookeeper would become a single point of failure. If the zookeeper instance would go down, then users would neither be able to submit any new topologies, or delete existing ones. Adding to this problem, when the zookeeper would go down, then a topology which undergoes a failure, would not get automatically detected and recovered. The third major issue was that the amount of processing done caused the zookeeper instance to become a bottleneck.

Heron is build from a design which is similar to the previous stream processing framework (*Storm*), but the implementation is new. One method which helps eliviate the aforementioned problems is the use of *backpressure*. This is usefull for when one of the stages in a stream processing topology is slower than other stages in the topology. A *Stream Manager* helps to implement backpressure, this is a special control service which helps to route data between the different processing stages. A topology is managed through a *Topology Master*, this service is responsible for tasks such as allocating machine resources, and serves as a gateway for topology metrics through an endpoint. This solves the problems of having a zookeeper instance as a single

point of failure. Another way in which *Heron* provides better performance and a better use of resources is by having each processing stage as a separate process, called a *Heron Instance*. This instance is a small process consisting of 2 threads, one (the gateway thread) which handles communication with the rest of the system and other *Heron Instances*, the other is the *Task Execution* which handles the actual computation.

The performance problems show here could also apply to *Druid* as that database also relies heavily on a zookeeper instance. kulkarniTwitterHeronStream2015 opted to write a new processing system called *Heron*. What we are trying to show here is that stream processing can be used to achieve the same results as a query language can, as shown in 6.1. However, what a dedicated stream processing framework does allow for is the inclusion of more varied external sources and more complex processing, such as presented by kulkarniTwitterHeronStream2015. They presented the use of complex machine learning algorithms running on top of the incoming streams of data.

With this we can answer RQ2 and RQ7. For RQ2 (*What techniques are used to process data stored in timeseries databases?*) we can say that different databases have opted for different ways of processing stored data. For example BTrDB opts for storing intermediate values in the indexing structure. This allows for faster queries as the processing which is to be applied during the query is partially completed, e.g "find the max value" can be completed using the intermediate values. For RQ7 (*How do stream processing techniques interact with timeseries databases?*) we can see that there are many techniques, however, the research is rather unclear in how these interact timeseries databases. We gave an example where Kafka Streams were used to process data in before returning it to the user, but this research also did not go deep into the effects this has on matters such as latency.

8 Roles

Table 3: Questions answered in this section

Question number	Research Question
RQ1	What roles do timeseries databases serve?

We dedicate this section to answering a single sub-question, namely RQ1. What is difficult about RQ1 is that the same question can be asked about any database.

Over the course of this paper we have seen several practical uses of time-

series databases, such as [44], [45], [53], and [43]. We also discussed two more general roles in section 1, namely the IoT and DevOps roles.

One more option we have not yet discussed throughout this work is storing financial data [56]. This type of data is time indexed by nature, for example, stock market data is inherently tied to time.

From this we can conclude that the roles timeseries databases are best able to serve is when a user has the requirement to store a large amount of data for which queries need to be executed based on a range of time. For which specific workloads to use a timeseries database is no conclusive answer.

9   Possibilities for future work

Over the course of this literature study we have identified two areas in which we think there is room for a more through exploration.

The first is to better evaluate the queries and investigate possible performance optimizations. No paper has made mention of, or provided details for, a query planner [57]. Another option would be a *just-in-time* compiler for SQL, as shown in [58]. This is an embedded compiler which takes the querying code and produces machine instructions instead of an intermediate representation. questdbHowWeBuilt shows a large improvement in performance, especially using hot data.

QuestDB stores data in a column-based storage model [59]. Here each column is stored in a separate file, and new data is appended to this file. This is made extra efficient by mmap'ing data into memory, allowing for machine instructions to write to memory, which result in writes to the column files. This allows QuestDB to make use of the page-cache in the operating system, allowing for high insertion speeds. What limited query performance was that SQL was compiled into intermediary Java which did not allow for optimizations, and that this intermediary code could not make use of SIMD instructions. This meant that during a scan of a table, the table would be scanned row-by-row. Using SIMD instructions, because the data was mapped directly into memory, the authors could do operations on multiple rows at the same time.

Using JIT the SQL code is converted into raw machine instructions, which can operate directly on the data which is mapped into memory. What made this easier is that the exact layout of the data in memory is known as only a single column is stored per file, which means that only that single column mapped into a page. This property is knowing the exact data layout is something which we found to be unexplored in timeseries databases.

The second area is to use raw flash storage. The databases using persistent storage covered in this study all use either filesystems or block storage as back-ends. However, we have not found any investigation into the use of raw flash storage. Flash storage uses a translation layer which maps from filesystem logical blocks to physical addresses on the flash device. Implementing this layer in a database has been shown to provide performance improvement [60].

We see an opportunity for the use of *Zoned Namespace* (ZNS) SSD's [61]. This is because of the append only nature of timeseries databases. While some (such as Timon) offer out-of-order writes, the writes to disk are all sequential and immutable. What sets ZNS SSD's apart from the traditional SSD's is that ZNS devices are divided into a number of *zones*. These zones are append-only, this means that ZNS devices do not support random write operations, only random reads and sequential writes (otherwise known as appends).

The NB+ tree shown in [62] could be a good option for an indexing structure. This data-structure already makes use of fixed size leaf nodes in which the timeseries data is stored. This is something which needs to be taken into account as all the previously presented databases have variable sized chunks of data which are being written to storage. Apart from that, this data structure does not have the overhead of having to do operations such as compaction, which is something that LSM tree based databases need to do.

10   Conclusion

Over the course of this study we have discussed the core pillars of timeseries databases. We have discussed ingestion, storing, querying, and processing of data. Using this we have answered research questions, and finally we have presented two areas for which we think that more research is possible.

Now wish to present an answer for the main research question, "*How do timeseries databases optimize workloads which use data indexed by time?*". Timeseries databases optimize these workloads mainly through their indexing data structures, these allow the databases to accelerate both ingestion and queries. Data processing is also optimized through these data structures. Examples of this are the tree structure from BTrDB, where statistics are recalculated and are used during data processing.

To give more insight we also list all the answers for the individual subquestions:

- RQ1 (*What roles do timeseries databases serve?*): A role in which the user has the requirement of storing a large amount of data which is indexed by time, and upon which queries have to be executed which operate on a range of time.

- RQ2 (*What techniques are used to process data stored in timeseries databases?*): different databases have opted for different ways of processing stored data. For example BTrDB opts for storing intermediate values in the indexing structure. This allows for faster queries as the processing which is to be applied during the query is partially completed, e.g "find the max value" can be completed using the intermediate values.

- RQ3 (*What indexing techniques are used to index the stored timeseries data?*): All databases which we have found during this survey use on-disk storage use some form of a tree structure, while the in-memory databases use a more direct mapping, such as presented in *Gorilla* with TSMap.

- RQ4 (*How does data get inserted into the timeseries databases?*): We have defined three distinct methods for inserting data into timeseries databases. The first is pull based ingestion, this is where the database itself has a component which retrieves data from pre-configured targets. The second is where data gets written or streamed to the database by the different applications wishing to write to the database. The third and final method is a combination of the previous two. Using a message queue data is streamed from the writers to the database through a queue, where the queue is read by the database. However, we have only encountered this third method in one database.

- RQ5 (*What storage back-ends are commonly supported by timeseries databases?*): There are two types, the first of which is using memory as a storage back-end, the second is using either local storage such as a filesystem or a remote focused block storage. Using memory as the primary storage mechanism boats higher performance, but has a significantly higher cost associated, making it impractical for all but the largest commercial entities.

- RQ6 (*What forms of compression are used on timeseries data?*): Coming back to RQ6, we can define two forms of compression, lossy and lossless. However, the if this is a worthwhile trade-off can differ greatly depending on the workload of the database. If the workload requires

the option to be able to query for individual datapoints lossy compression is not going to be sufficient, however, if the workload only requires users to be able to view statistical values based on the data which has been written to the database, then lossy compression can suffice.

- RQ7 (*How do stream processing techniques interact with timeseries databases?*): we can see that there are many techniques, however, the research is rather unclear in how these interact timeseries databases. We gave an example where Kafka Streams were used to process data in before returning it to the user, but this research also did not go deep into the effects this has on matters such as latency.

- RQ8 (*How does one properly benchmark and evaluate different timeseries databases?*): There currently exists no standardized benchmarks for timeseries databases. There have been two attepts to create a general purpose testing framework for timeseries databases, namely *SciTS* and *TSBS*. Both benchmark the ingestion and query performance.

We presented two areas in which we find that there is more research to be done. The first is in accelerating queries by analyzing them and running them through a query planner. The second is using ZNS SSD's. These could provide performance benefits due to timeseries data being append only and ZNS SSD's having fast appends.

11   References

References

[1] "TimescaleDB vs. PostgreSQL for time-series." https://www.timescale.com/blog/timescaledb-vs-6a696248104e/, Aug. 2017.

[2] C. Strauch, U.-L. S. Sites, and W. Kriha, "NoSQL databases," *Lecture Notes, Stuttgart Media University*, vol. 20, no. 24, p. 79, 2011.

[3] "8.5. Date/Time Types." https://www.postgresql.org/docs/15/datatype-datetime.html, Nov. 2022.

[4] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A Distributed Multi-resolution Time-Series Datastore," in *2013 IEEE 34th Real-Time Systems Symposium*, (Vancouver, BC, Canada), pp. 288–297, IEEE, Dec. 2013.

[5] "Industrial Internet of Things," *Industrial Internet of Things*, p. 9, 2017.

[6] VictoriaMetrics, "Case studies and talks · VictoriaMetrics." https://docs.victoriametrics.com/CaseStudies.html, Sept. 2018.

[7] D. Abadi, "SYSTEM IN COMMUNICATION WITH A DATA PROCESSING FRAMEWORK," p. 34.

[8] "RRDtool - The Time Series Database." http://www.rrdtool.org/.

[9] "Prometheus." Prometheus, Nov. 2022.

[10] "OpenTSDB - A Distributed, Scalable Monitoring System." http://opentsdb.net/.

[11] "Graphite." https://graphiteapp.org/.

[12] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, (London, England, United Kingdom), pp. 1–10, ACM Press, 2014.

[13] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?," *Computer*, vol. 43, pp. 12–14, Feb. 2010.

[14] R. Sears, C. van Ingen, and J. Gray, "To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?," p. 11.

[15] T. Suel and N. Memon, "Algorithms for Delta Compression and Remote File Synchronization," in *Lossless Compression Handbook*, pp. 269–289, Elsevier, 2003.

[16] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, pp. 491–541, July 1997.

[17] P. Esling and C. Agon, "Time-series data mining," *ACM Computing Surveys*, vol. 45, pp. 1–34, Nov. 2012.

[18] A. Bader, O. Kopp, and M. Falkenthal, "Survey and Comparison of Open Source Time Series Databases," p. 20, 2017.

[19] M. Barata, J. Bernardino, and P. Furtado, "YCSB and TPC-H: Big Data and Decision Support Benchmarks," in *2014 IEEE International Congress on Big Data*, (Anchorage, AK, USA), pp. 800–801, IEEE, June 2014.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing - SoCC '10*, (Indianapolis, Indiana, USA), p. 143, ACM Press, 2010.

[21] "DB-Engines Ranking." https://db-engines.com/en/ranking/time+series+dbms.

[22] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, (Snowbird Utah USA), pp. 157–168, ACM, June 2014.

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

[24] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia, N. Sakharov, G. Talbot, A. Tart, and N. Taylor, "Monarch: Google's planet-scale in-memory time series database," *Proceedings of the VLDB Endowment*, vol. 13, pp. 3181–3194, Aug. 2020.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Transactions on Computer Systems*, vol. 31, pp. 1–22, Aug. 2013.

[26] L. Lamport, "The part-time parliament | ACM Transactions on Computer Systems (TOCS)." https://dl.acm.org/doi/10.1145/279227.279229, May 1998.

[27] M. M. Patil, A. Hanni, C. H. Tejeshwar, and P. Patil, "A qualitative analysis of the performance of MongoDB vs MySQL database based on insertion and retriewal operations using a web/android application to explore load balancing — Sharding in MongoDB and its advantages," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, (Palladam, Tamilnadu, India), pp. 325–330, IEEE, Feb. 2017.

[28] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 92–105, Jan. 2010.

[29] W. Cao, Y. Gao, F. Li, S. Wang, B. Lin, K. Xu, X. Feng, Y. Wang, Z. Liu, and G. 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*, (Portland OR USA), pp. 739–753, ACM, June 2020.

[30] Z. Wang, J. Xue, and Z. Shao, "Heracles: An efficient storage model and data flushing for performance monitoring timeseries," *Proceedings of the VLDB Endowment*, vol. 14, pp. 1080–1092, Feb. 2021.

[31] M. P. Andersen and D. E. Culler, "BTrDB: Optimizing Storage System Design for Timeseries Processing," p. 15, 2020.

[32] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs," p. 15.

[33] Y. Li, L. Wang, F. Cheng, J. Wu, Y. Zhang, and Y. Zhang, "IndigoStore: Latency Optimized Distributed Storage Backend for Cloud-Scale Block Storage," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, (Beijing, China), pp. 883–890, IEEE, Dec. 2021.

[34] "Theoretical Maximum Memory Bandwidth for Intel® Core™...." https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html.

[35] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, pp. 351–385, June 1996.

[36] "Apache IoTDB." https://iotdb.apache.org/.

[37] "InfluxDB OSS 2.0 Documentation." https://docs.influxdata.com/influxdb/v2.0/.

[38] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, pp. 1–26, June 2008.

[39] "IEEE Standard for Floating-Point Arithmetic," tech. rep., IEEE.

[40] "Snappy." http://google.github.io/snappy/.

[41] "Encoding | Protocol Buffers." https://developers.google.com/protocol-buffers/docs/encoding.

[42] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Software: Practice and Experience*, pp. n/a–n/a, 2010.

[43] M. Fadhel, E. Sekerinski, and S. Yao, "A Comparison of Time Series Databases for Storing Water Quality Data," in *Mobile Technologies and Applications for the Internet of Things* (M. E. Auer and T. Tsiatsos, eds.), vol. 909, pp. 302–313, Cham: Springer International Publishing, 2019.

[44] A. Göransson and O. Wändesjö, "Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry," p. 86, Apr. 2022.

[45] M.-E. 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.

[46] J. Mostafa, S. Wehbi, S. Chilingaryan, and A. Kopmann, "SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things," June 2022.

[47] "Fast Open-Source OLAP DBMS." https://clickhouse.com/.

[48] "Time-series data simplified." https://www.timescale.com.

[49] P. G. D. Group, "PostgreSQL." https://www.postgresql.org/, Nov. 2022.

[50] "Time Series Benchmark Suite (TSBS)." Timescale, Nov. 2022.

[51] K. Doekemeijer and A. Trivedi, "Key-Value Stores on Flash Storage Devices: A Survey," May 2022.

[52] "Flux data scripting language | InfluxDB OSS 1.8 Documentation." https://docs.influxdata.com/influxdb/v1.8/flux/.

[53] L. Shen, Y. Lou, Y. Chen, M. Lu, and F. Ye, "Meteorological Sensor Data Storage Mechanism Based on TimescaleDB and Kafka," in *Data Science* (X. Cheng, W. Jing, X. Song, and Z. Lu, eds.), vol. 1058, pp. 137–147, Singapore: Springer Singapore, 2019.

[54] "Apache Kafka." https://kafka.apache.org/.

[55] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, (Melbourne Victoria Australia), pp. 239–250, ACM, May 2015.

[56] E. Sandberg, "High performance querying of time series market data," Feb. 2022.

[57] Y. E. Ioannidis, "Query optimization," p. 3.

[58] QuestDB, "How we built a SIMD JIT compiler for SQL in QuestDB | QuestDB: The database for time series." https://questdb.io/blog/2022/01/12/jit-sql-compiler/.

[59] QuestDB, "Storage model | QuestDB: The database for time series." https://questdb.io/docs/concept/storage-model/.

[60] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, (Amsterdam, The Netherlands), pp. 1–14, ACM Press, 2014.

[61] M. Bjørling, "From Open-Channel SSDs to Zoned Namespaces," p. 18, 2019.

[62] "Numeric B+tree reference." https://docs.google.com/document/d/1jFK8E3CZSqR5IPsMGojm2L