Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3729

Comparison of Time Series Databases

Andreas Bader

Course of Study:	Informatik
Examiner:	Dr. Holger Schwarz
Supervisor:	Dipl.-Inf. Oliver Kopp, Michael Falkenthal, M.Sc.
Commenced:	2015-07-15
Completed:	2016-01-13
CR-Classification:	H.2.4, H.3.4, C.2.4, C.4

Abstract

Storing and analyzing large amounts of data are growing in importance since the fourth industrial revolution. As more devices are becoming “smart” and are equipped with sensors in today’s world, the amount of data that can be stored and analyzed grows. Insights from this data are important for several industries, e. g., energy companies for controlling smart grids.

Traditional Relational Database Management Systems (RDBMS) have reached their limits with such huge amounts of data, which resulted in a new database type, the NoSQL Database Management Systems (DBMS). NoSQL DBMS are specialized in handling huge amounts of data with the help of distribution and weaker consistency. Between these two a new type arose: Time Series Database (TSDB), which is specialized for storing and querying time series data.

The amount of existing TSDBs is big, whereby for this thesis 75 TSDBs have been found. 42 of them are open source, the remaining TSDBs are commercial. Many of the found open source TSDBs are under ongoing development. The challenge is the selection of one TSDB for a given scenario or problem. Benchmarks that have the ability to compare several TSDBs for a specific scenario or in general are hardly existing. This currently makes a choice based on performance only possible if TSDBs are manually compared in a test environment or with a self-written benchmark.

In this thesis, a feature comparison with 19 criteria in five groups between ten of these TSDB is presented and discussed. After presenting metrics, scenarios, and requirements for a benchmark, a benchmark for TSDB, TSDBBench, is presented. TSDBBench uses an Elastic Infrastructure (EI) and alterable workloads to measure the query latency and space consumption in different scenarios that include an alterable cluster setup. All benchmarking steps are automated, so that no user interaction is required after starting the benchmark. It also uses an adapted version of Yahoo Cloud Server Benchmark (YCSB) that is named Yahoo Cloud Server Benchmark for Time Series (YCSB-TS) for creating and measuring the queries of a workload, which is also presented in this thesis. For the performance part of the comparison, two scenarios are compared between the ten TSDBs with the use of TSDBBench. The results of the performance comparison are presented and discussed afterward. The thesis concludes with a discussion of the results from the feature and performance comparison.

Contents

List of Figures

List of Tables

1 Introduction

The importance of sensors has been growing in the last years as the fourth industrial revolution[1] began [BPV14; CBA+15; CON12; Han15]. Part of it are two concepts, among others [WCZ15], that demand for more senors: Internet of Things (IoT) and Cyber-Physical Systems (CPS). The first one describes the idea of smart objects that can collect data from their environment and exchange it via the internet. The latter one describes embedded systems that benefit from the connection between objects and manufacturing machines. As the borders between both concepts are fluent, most concepts take parts of both ideas and there exists no exact assignment.

An example for the intensified use of sensors are energy grids that need to be controlled smarter and on a much finer granularity as before. Increasing amounts of renewable energy, like solar panels, and smart meters are forcing a change in existing energy markets [BF12]. The traditional producer (e. g., power plant) and consumer structure changes because with solar panels there exist more consumers that are also small energy producers, for example. Additionally, higher amounts of renewable energy result in a more area and time dependent electricity generation (e. g., wind power is mostly used in advantageous areas and solar power can only produce electricity by day) [THK+15; XWWC10]. Smart meters help to control the energy flow on a finer granularity by reporting energy consumption of each household back to the energy provider. This can lead to a better prediction of energy consumption, which results in smaller costs and other advantages for energy providers [Ele11; Uni14]. In a future scenario, it could be possible to automatically control when a periodically running machine is started (e. g., washing machine) [EM10]. This would help lessen peaks in energy grids and distribute load over a day. Additionally, it can be forced by using different prices based on time (e. g., off-peak electricity, which was popular in Germany in the 70s), which in return could lessen energy costs for private end users.

The intensified use of sensors leads to an increased amount of sensor data that needs to be stored and processed. As an example, for a smaller city that is fully provided with smart meters, we can use 200,000 million voltage measurements each

[1] Also called “Industrie 4.0” [HR15].

month [SZGA15]. To handle that amount of data, it is necessary to scale the Database Management System (DBMS) accordingly, which means distribution across several nodes with the possibility to increase further when the city grows. There are different views on the question if traditional Relational Database Management Systems (RDBMS) can handle such amount of data. On one side, distributing writes in a relational model, whilst keeping full consistency and the same level of query latencies, is not easily possible [Aba12; PA13]. On the other hand, MySQL Community Server [Ora15f], a traditional RDBMS, has been used to ingest 332,000 values per second into it, running on three Amazon Web Services (AWS) Elastic Compute Cloud (EC2) servers [Viv15]. This shows that, with enough effort and considering the use case, RDBMS can be used for time series data.

NoSQL DBMS provide a solution with greater possibilities for distribution by weakening relations and consistencies [Cat11; GHTC13]. Knowing that sensor data has most of the time a timestamp attached to it and can therefore be used as an index, a trade-off between RDBMS and NoSQL DBMS arose: TSDBs. The boundaries between NoSQL DBMS and TSDBs are fluent, as there exists no precisely defined boundary between them.

The term “TSDBs” is not accurate, as it is used as an abbreviation for “time series database”, but is used in the meaning of “time series database management system”. Normally a Database System (DBS) consists of a Database (DB) and a DBMS. While “DB” describes a collection of data, “DBMS” describes the software that is used to define, process, administrate, and analyze the data in a “DB” [Här01; Mit13]. Most literature uses the term “DBMS” (like RDBMS) instead of “DBS”. In this thesis DBMS, RDBMS, and TSDB will be used in this non-accurate way.

A typical TSDB can store a combination of a timestamp and a value. Several timestamps are grouped together in a metric. Most TSDB add further possibilities like adding tags for finer relations and functions for aggregation (e. g., GROUP BY, CNT) and mathematical calculation (e. g., average). TSDBs can be used for making decisions (e. g., business intelligence [CLS09]), controlling complex systems (e. g., smart grids [BDM07; LLL+11; PA13; Sti14]), or creating benefits in some way by storing and analyzing time series data. As it seems not feasible to store continuously every data from any sensor everywhere forever, a trade-off is needed. Since each scenario has different requirements, it should be possible to decide which sensor data can be stored in an aggregated[2] way. An energy grid, for example, needs recent data on a very fine granular basis for energy flow control decisions. Data that is only needed for long term

[2] E. g., storing the data of a sensor for one day in the available resolution and older data as one average value per hour.

decisions, can be stored it on a lower granularity (e. g., by using averaging or only storing results) while deleting all fine granular data that is not needed anymore.

During this thesis, 42 open source and 34 commercial TSDBs have been found (see Section 3.1), which makes the decision for a specific TSDB not straight forward. Ongoing development and the launch of new TSDBs are also problematic. There are many factors that can be compared between each TSDB for choosing one of them. In this thesis 19 different criteria grouped in five groups (see Section 3.6) together with the benchmark results were used for comparison (see Chapter 5).

Some TSDBs are more like a “dump” storage, while others can do complex calculations besides storing data, which in itself changes the development of a project and its performance. As there is no standard set of features or even a standardized query language for TSDBs, it is needed to compare offered features of different TSDBs closely. There are “inner” and “outer” features that need to be considered. The “inner” ones decide which calculation can be done inside the TSDB and how (e. g., with which granularity) your data is stored. The “outer” features decide how a cluster of a TSDB is built and maintained, e. g., how to achieve High Availability (HA), how many nodes can form a cluster, or if clustering is even possible.

Another aspect is the performance of a TSDB, which is divided into two parts: queries and storage. The first part means how fast a TSDB can execute the queries passed to it by an application. The type of a query is important here, as there is a difference if an application uses simple READ queries, or INS queries (see below for explanation of queries), or uses more complex queries. Performance can also be affected by the amount of data already stored inside and being continuously written to the TSDB. Storing data can also be more or less effective, as it can be stored as much compressed as possible or in a very space consuming way, which makes the second part of the performance aspect: storage.

Not only companies might consider costs as an important factor, as they might want to know what the use of a specific TSDB costs in long term. Costs for acquiring a new software consist of two parts: Initial costs and periodical costs [Kon13]. Most of open source TSDBs are free to use, whereas most of the commercial alternatives are not. Besides the price for buying the TSDB itself, additional software and the amount and complexity of the development needed highers the initial costs. After the first setup and development, there are periodical costs that can vary depending on the existence of a support contract or if one or more employees are required to keep the TSDB in a running state.

Development is another point to consider when choosing a TSDB. On one side, there are TSDBs that are continuously developed, and on the other side, there are TSDBs

that have not been in development for years. For a commercial product one pays to get support and continuous development, but most open source alternatives do not offer any kind of paid commercial support. Some do not even release or support a stable or Long Term Support (LTS) version, which in result leads to the need of frequent non-stable[3] updates. Another point is code quality and documentation, the lesser both are, the higher can the costs for development and maintenance be, which should be carefully considered.

Scientific comparisons or benchmark-results of TSDBs are rarely found or, if existing, made for a specific purpose (e. g., showing that a new TSDB it better than a few others). There are two existing benchmarks for TSDBs (see Section 2.2), but both of them support only a subset of TSDBs. One uses a specific scenario (financial time series data) and the other one is closed source. This thesis presents a comparison between ten TSDBs, including a solution for an extensible benchmark for TSDBs with variable workloads. The benchmark is called TSDBBench (see Section 4.5) and uses an extended version of Yahoo Cloud Server Benchmark (YCSB) that is called Yahoo Cloud Server Benchmark for Time Series (YCSB-TS) (see Section 4.6).

In the remaining part of this thesis, nine different queries will be used for comparison and explanation: Insertion (INS), Updating (UPDATE), Reading (READ), Scanning (SCAN), Averaging (AVG), Summarization (SUM), Counting (CNT), Maximization (MAX), and Minimization (MIN). The resulting data inside a TSDB of one single INS, that was successfully executed, is called row (of data), or record. A TSDB can store several metrics, each metric consists of a name and several rows of data. It is used to group rows of data together on a higher level than tags [4]. A row of data consists of a timestamp[5], a value[6], and optional tag values. A tag can be imagined as a column of a table in which each row can place a tag value. A tag can be used to group rows together (e. g., each row consists of sensor values and a tag is named “room” and the tag values are room numbers). Both the tag values and the name of the tag are alphanumeric.

[3] A non-stable upgrade can require client side changes. Stable updates do not change Application Programming Interfaces (APIs) or interfaces (e. g., a REST interface), only when a security issue (or other important issues) require such changes.

[4] A metric is comparable to a table in RDBMS, e. g., storing voltage values from several sensors, grouped by tags, in a metric called “voltages”.

[5] A timestamp expresses a specific point in time. The finest granularity in a TSDB is used to describe this point exactly, usually milliseconds.

[6] Usually floating point with double or single precision.

The queries are described as follows: INS is a query that inserts one single row into a DBMS. UPDATE[7] updates one or more rows with a specific timestamp. READ reads one or more rows with a specific timestamp or in the smallest time range possible. SCAN reads one or more rows that are in a specific time range. AVG calculates the average over several values in a specific time range. SUM summarizes several values in a specific time range. CNT counts the amount of existing values in a specific time range. Deletion (DEL)[8] deletes one or more row(s) with a specific timestamp. MAX[9] and MIN[9] search for the maximum, respective minimum, value of several values in a specific time range.

A time range can also be a timestamp. A granularity defines the distance between two timestamps, and is usually used to describe a guaranteed granularity for storing rows of data ([guaranteed] storage granularity)[10] or a granularity with which rows of data are received[11]. Each query can have additional tags for grouping rows together (e. g., a tag is used to identify all values of a sensor). AVG, SUM, CNT, MAX and MIN are aggregating queries, which means that the results can be grouped together in time ranges (e. g., querying the maximum value of each day in a year). These descriptions are very general and the exact definition depends on the DBMS in context.

A set of queries can be grouped together to a workload, which is usually fitted to some scenario. A workload is called alterable, when the parameters or set of queries can be changed to fit a new scenario (e. g., increasing the amount of queries). This definition will be refined in Section 4.2.

For understanding this thesis, knowledge in the concepts of cloud computing are required, see [FLR+14] for further reading.

The structure of the remaining thesis is organized as follows: In Chapter 2, previous benchmark results, existing benchmarks for TSDBs, metrics, and other background for understanding this thesis are presented. Chapter 3 introduces the existing TSDBs and the search process used for finding them. Afterward, a subset of TSDBs is introduced

[7] This query is not used for comparing TSDBs in this thesis or with YCSB-TS, but is needed for Chapter 2.

[8] This query is not used for comparing TSDBs in this thesis or with YCSB-TS, but is needed for Appendix A.6.

[9] This query is not used for comparing TSDBs in this thesis, but is used in YCSB-TS as a replacement for missing functions. See Section 4.2.

[10] In Section 3.6 storage granularity is used to describe the granularity with which rows of data can be inserted, and guaranteed storage granularity describes the storage granularity that a TSDB guarantees at least.

[11] Which can be different from the granularity used to store it.

more detailed and compared using 19 criteria. In Chapter 4, TSDBBench and YCSB-TS are presented and explained in detail. This includes the benchmark architecture itself, the requirements used, the problems that needed to be solved, and which scenarios were used. In Chapter 5, ten TSDBs are measured in the scenarios from Chapter 4 and the results are shown and discussed. Chapter 6 combines the results from Chapter 5 with the comparison from Chapter 3 to present and discuss an overall result and give recommendations for different use cases. Chapter 7 completes the thesis with a conclusion and points out problems and improvement potentials for future work.

2 Background and Related Work

This chapter presents background and related work and is structured as follows: Section 2.1 summarizes the found scientific publications around TSDBs as background for this thesis. Section 2.2 presents found benchmarks for TSDBs and NoSQL DBMS. Section 2.3 presents time series datasets.

2.1 Preliminary Findings

Previous publications are motivated by releases of new TSDBs or the comparison of any kind of DBMS for processing or storing time series data. Before TSDBs were widely known, Pungilă, Fortiş, and Aritoni [PFA09] tried to find a fitting database for a set of households with smart meters. For reaching their goal, they compared three RDBMS, one TSDB, and three NoSQL DBMS in two scenarios. Both scenarios used a different amount of INS (10,000,000 and 4,000,000) and READ queries over time periods with sorted results. Each scenario was tested at different insertion rates. They came to the conclusion that there are three possible DBMS for their scenario, depending whether the focus lies on INS queries, READ queries, or both.

Bushik [Bus12] benchmarkes four NoSQL DBMS and one RDBMS (as a cluster and as a sharded version) with the help of YCSB [CST+10] using the six given default workloads of YCSB and one additional workload. Each workload runs on 1,000,000,000 records. Bushik comes to the conclusion that there is no NoSQL DBMS that fits all needs and that the choice depends on the queries used.

Deri, Mainardi, and Fusco [DMF12] present tsdb as a compressed TSDB that handles large time series better than three existing solutions. They discovered OpenTSDB [LSKD15a] as only available open source TSDB, but do not compare it to tsdb. The reason is the architecture of OpenTSDB, which is not suitable for their setup. Deri, Mainardi, and Fusco compare tsdb to MySQL[1] [Ora15f], RRDtool [Oet15] and Re-

[1] Deri, Mainardi, and Fusco do not specify which version of MySQL is used.

dis [San15] instead, with the result that the append/search performance of tsdb is best out of the four compared DBMS. A comparison to a TSDB that supports distribution, like OpenTSDB, would have been interesting because the only other TSDB in this comparison was RRDTool which has no distribution or scalability features (see Section 3.1).

Wlodarczyk [Wlo12] compares four solutions for storing and processing time series data. The results are that OpenTSDB is the best solution if advanced analysis is needed, and that TempoIQ (formerly TempoDB) [Tem15] can be a better choice if a hosted solution is desired. No benchmark results are provided, only a feature comparison is done.

Busemann et al. [BGG+13] propose a Service-oriented Architecture (SOA) for energy monitoring and prediction. Busemann et al. use PostgreSQL [PGD15a] as storage in their backend and evaluate its performance in two scenarios. PostgreSQL can execute 140 INS queries per second with no other queries running in parallel. After adding SELECT queries[2] with varying rate and keeping the INS queries constant at 60 queries per second, PostgreSQL can still handle the INS queries but more than 100 SELECT queries per second cannot be completed in parallel to the them. The response time is also varying dependent on the rate of queries, but both SELECT queries and INS queries lie below 100 milliseconds. When the SELECT queries reach their maximum at 100 queries per second, a doubled response time, with a few peaks with tenfold response time, are observable.

Nelubin and Engber [NE13] extend YCSB to measure a specific scenario consisting of a key-value database with “extremely high throughput” and “low latency” as requirements. The major idea is to provide an answer to one scenario instead of trying to measure a greater set of scenarios like Bushik; Cooper et al. [Bus12; CST+10] do. Nelubin and Engber also use Solid State Disks (SSDs) instead of Hard Disk Drives (HDDs). They measure Cassandra [ASF15a], HBase [ASF15d], MongoDB [Mon15f], and Aerospike [Aer15] with two different types of datasets, one with 50 million records in memory, the other one with 500 million records on SSDs. They also measure two workloads, one with balanced amounts of READ and UPDATE queries and one with 5% UPDATE queries. Their conclusion is, that Aerospike is the fastest competitor, despite the fact that it ensures ACID properties, but on the other hand Cassandra and MongoDB have more features. Other aspects like recoverability or performance on a Elastic Infrastructure (EI)[3] needs to be tested and considered in future tests.

[2] These queries are generated by HTTP requests and not described any further.

[3] An EI is a platform that uses virtualization to abstract RAM, CPU, storage, network and other hardware from the physical hardware to run multiple Virtual Machines (VMs) on it.

Prasad and Avinash [PA13] propose NoSQL DBMS and TSDBs as better long-term choices than RDBMS for the analytics of smart meter data. Prasad and Avinash present a solution consisting of OpenTSDB, HBase, and Hadoop [ASF15c], mentioning technical benefits and impacts on business value, when using their solution instead of a RDBMS. They conclude that there can be an advantage for service providers, when using open source NoSQL DBMS or TSDBs, but there is also skepticism for using them instead of a RDBMS. The reasons for the skepticism are due to their novelty, as learning and understanding these new technologies costs time and training. In addition, smart meter data can not be fed directly to OpenTSDB and needs a conversion beforehand. The tradeoffs when using a NoSQL DBMS like Cassandra or HBase (which OpenTSDB uses) in consistency compared to a RDBMS must also be considered.

Abramova, Bernardino, and Furtado [ABF14] compare, with the help of YCSB, how different cluster sizes affect the performance of Cassandra. After testing several workloads, each with LOAD phases of 1 million, 10 million and 100 million INS queries, Abramova, Bernardino, and Furtado come to the conclusion that adding more nodes does not always result in a performance increase. INS or READ queries, for example, benefit less than SCAN queries[4] from having more nodes available. The reason is the insensitivity of INS and READ queries to amount of data stored in the cluster. A higher amount of nodes results in fewer data per node, but also increased communication between nodes that equalizes possible benefits from the fewer data per node.

Goldschmidt et al. [GJK+14] benchmark OpenTSDB [LSKD15a], KairosDB [Kai15b], and Databus [Lin15]. The aim of the work is to proof the question if any of these TSDBs can scale linearly, handle industrial workloads, scale workload independent, tolerate the crash of two nodes, and has an independent read/write performance. After measuring two workloads on different cluster sizes, they conclude that only KairosDB fulfills all their hypotheses and would be able to handle 6 million smart meters with 24 nodes. The tests of OpenTSDB are problematic, as the results are not repeatable.

Huang et al. [HCC+14] present Operational Data Historian (ODH) as a combination of a TSDBs with complex queries in SQL. In this approach it is possible to query relational and time series data with the same interface, as time series data is represented in virtual tables. They also present IoT-X[5], an IoT benchmark with two different workloads, with which they show that ODH has a higher write performance as a commercial RDBMS

[4] Mainly when using a higher amounts of data, e. g., 100 million.

[5] Huang et al. plan to release IoT-X as an open source benchmark.

and MySQL Community Server [Ora15f] by an order of magnitude and has an equal read performance.

The End Point Corporation [End15] use a customized YCSB to compare Cassandra, Couchbase [COU15], HBase and MongoDB with five different workloads. Each workload run at a maximum of 9 million queries. Each node has 500 million records. Cassandra has the highest throughput and lowest latency over all workloads. The second position is not clear, as it depends on the workload, whether HBase or Couchbase performs second best. Although the results are obvious, the authors suggest doing a fitted benchmarking for each scenario and that there is not one database that suits every use case.

Strohbach et al. [SZGA15] describe Hadoop as a better choice as a RDBMS in a smart grid scenario. The main reason is the easier achievable scalability because less development is required. Strohbach et al. make their assumption based on the experience and the data from the Peer Energy Cloud (PEC) [PEC15], that uses a micro grid consisting of 100 households in Saarlouis. They extrapolate the data to a full rollout that would result in 200,000 million voltage measurements each month and 360,000 energy-related measurements every second[6]. Each household can produce 2 million energy-related measurements a day and 18 energy-related measurements every second[6].

Taowen [Tao15] searches for an agile TSDBs and compares several DBMS. An “agile” TSDB aggregates data in two steps: during ingestion and during querying. This two-step aggregation helps to get different two-dimensional views on multidimensional data in a faster way than with one-step or no aggregation. With two-step aggregation, there is no need to run each query on the unaggregated data that was originally ingested, but instead they are running on already aggregated data (that was aggregated after ingestion). All DBMS are grouped into four groups: 1) “Old school” (e. g., RRDtool, Graphite [Dav15]), 2) Fast key-value (e. g., OpenTSDB, KairosDB), 3) Aggregating (e. g., Elasticsearch [Ela15], Druid [YTL+14]), and 4) Future (e. g., VectorWise [Act15], Alenka [Ant15]). Not all of these DBMS are TSDBs, but can be adopted. Taowen concludes that RDBMS and group two (fast key-value) can be a good choice if the queries are well known beforehand and the data can fit into a “metric name centric” data model. As this is not fitting for his requirements, a mapping for Elasticsearch is build to use it as a TSDB.

As a summary it can be concluded that there is not one single database that suits every use case. Benchmarking needs to be done depending on the scenario with a fitted

[6]Including voltage measurements.

workload. Other features such as scalability need to be considered when choosing a solution for storing and processing time series data.

2.2 Benchmarks

To evaluate the performance of a TSDB, a software that measures performance indicating parameters in a controllable and defined environment, is needed. Such a software is called a “benchmark”, which usually consists of at least two parts: A initial set of records and a set of queries that are executed and measured on that initial set of records. The initial set can be empty.

From the related work (see Section 2.1) it can be concluded that a benchmark that can run and measure different scenarios, is required. It is needed to measure the performance depending on a scenario as it seems impossible to give a unique solution for every scenario. The main idea is that a workload, fitted for a scenario, can be run and gives performance insights that help to decide which TSDB to use for the given scenario. This leads to the need of alterable workloads. It is also needed to support different TSDBs, which leads to the need of an extensible client interface. Furthermore, the benchmark should be accessible, extensible and runnable by everyone that can provide the hardware to run it. It is not expected that results are comparable between different systems (meaning hardware). There are only few existing benchmarks especially for TSDBs or time series data. After extending the search for such benchmarks that are related to TSDBs, time series data, cloud databases, or NoSQL DBMS, it was possible to find more.

There are only two existing benchmarks for time series data: STAC-M3 Benchmark Suite [STA15] and FinTime [Kai15a]. The first one is a closed source benchmark that tries to measure performance of TSDB that can run analysis on time series data. There is no possibility to get the specification or other materials of the benchmark without being a council member. There seems to be support for only two TSDBs. The support of different workloads or development of clients for new TSDBs is not appreciable. As a result this benchmark is not fitting for the needs of this thesis because it supports two TSDBs and is not accessible without being a council member.

FinTime is an open source benchmark for financial time series data. It uses two different models and three metrics to determine the performance of a database in a realistic financial setup with time series data. Its models do not match modern TSDB

structures (Metric, Value, Tags)[7] and every query has to be run with ACID properties. Considering this, its age of 16 years and the fact that the source code is not even available for download anymore, it is not fitting for this thesis.

Taking benchmarks related to TSDBs into consideration, it was possible to find four more: YCSB [CST+10], Transaction Processing Performance Council (TPC) [Tra15a], Benchw [Kir15], and APB-1 [Cou98].

YCSB is a benchmark for NoSQL DBMS, consisting of a framework and a six standard workloads. The framework itself consists of three parts: A client that connects to a database and runs queries on it, a workload generator that generates the queries defined by the workload, and a core that does measuring, logging, as well as starting and stopping different phases of the benchmark. The main idea is that the clients, workload generator, and the workloads are alterable. The source code is open source and actively maintained. YCSB is sometimes even considered as a standard benchmark for NoSQL DBMS [Kam15; Rog14]. It seems that YCSB fits all criteria for this thesis, as it has alterable workloads for running different scenarios, an extensible client and is open source.

TPC has several benchmarks for measuring the performance of different business scenarios. TPC-C [Tra15b] uses an Online Transaction Processing (OLTP) workload that emulates a wholesale company. TPC-DI [Tra15c] benchmarks the integration of data from different sources into one database, sometimes referred to as Extract, Transform, Load (ETL) or Data Integration (DI). TPC-DS [Tra15d] emulates a decision support system for a retail product supplier. TPC-E [Tra15e] uses a brokerage company as a model and emulates OLTP transactions on its customer account database to reproduce a interaction between the company and its customers. TPC-H [Tra15f] emulates a decision support scenario, where the queries are more complex than OLTP queries and therefore are inclined to have a long execution time. TPC-VMS [Tra15g] tries to benchmark the performance of a database when running in a virtualized environment. It takes one benchmark of TPC-C, TPC-E, TPC-H, and TPC-DS and runs it three times parallel on three identical VMs. TPCx-HS [Tra15h] is a stress test for systems that use Hadoop, HDFS [Bor15] and MapReduce [DG08]. Although all TPC benchmarks are clearly specified and open source implementations are available for most of them, none of them is suiting for this thesis because all of them describe single scenarios in a business environment and therefore are not able to run alterable workloads.

[7] See Chapter 1.

Benchw uses queries that emulate a business warehouse to measure the performance of a database. Its intention is to be more easy runnable than the TPC-C, TPC-H and TPC-R (deprecated) and measuring query performance like TPC-H. It supports eight different RDBMS. Considering the same arguments used against the TPC benchmarks, its age of 11 years, and a stale development, Benchw is not a fitting benchmark for this thesis.

APB-1 was released by the OLAP council in 1998. It uses a business scenario to measure the Online Analytical Processing (OLAP) performance of a database. The scenario includes bulk loading, aggregation, time series analysis, complex queries, and ad hoc queries. The same arguments against TPC can also be used again APB-1: The lack of an alterable workload. It is also unclear if the OLAP council exists anymore, as its homepage and the last release of APB-1 are from 1998.

2.3 Data

For benchmarking a TSDB it is necessary to insert data into it, so that the INS queries themselves or some other queries, running on that data, can be measured afterward. There are two possibilities for data: using existing “real world” data or generating data.

For time series data, there were three sources for “real world” data found:

- The DEBS Grand Challenge 2014 [JZ14] and 2015 [JZ15],
- Awesome Public Datasets (a list of publicly available data sources) [Che15b], and
- UCR Time Series Classification Archive [CKH+15].

Some of the above data would be fitting for the presented benchmark (DEBS 2014 [JZ14] and some of the energy datasets from [Che15b]), while the rest is not fitting because of the structure of the data[8] (DEBS 2015 [JZ15]) or the small size (UCR Time Series[9] [CKH+15]). The main problem of using precaptured or precalculated data is

[8] DEBS 2015 uses reports of taxi trips as data (see Whong [Who14]) that containts 18 values (e. g., starting time, ending time, starting and ending longitude/latitude) per trip. The data is not usable for TSDB as the values are not based on a timestamp (e. g., a sensor value) and have therefore no time series character.

[9] Chen et al. offers several time series dataset, all of them below 10,000.

that it is only fitting for one specific scenario. Furthermore, not all these datasets are publicly available with a defined license (e. g., DEBS 2014, 2015).

The other idea is to use generated data for each benchmark run. The method for generating data must be the same for each benchmark run. A benchmark needs settings that determine exactly how data is generated, so that there are equal conditions[10] between two runs that use the same settings. Randomly generated data with several eligible distributions can be used in the generation method. The advantage is that this method can be easily distributed and adjusted to fit a specific scenario. It can be argued that there is a difference between each run when using random data because the data generated on each run is slightly different due to the randomization (e.g. every String is different). The structure of the data stays the same.

The presented benchmark in this thesis will use both methods, the main method will be generated data but it is possible to use predefined data sets. The reasons for this is that it should be possible to distribute the benchmark freely to everybody who wants to run it for its specific scenario with fitting generated data. If there is existing data for a scenario then there should be the possibility to use it to get possibly better results.

[10]For the generated data.

3 Time Series Databases

This chapter presents the method of searching for TSDBs, the selection and grouping of the compared TSDBs. It is structured as follows: Section 3.1 presents the method of searching for TSDBs and how the compared TSDBs are selected and grouped. Sections 3.2 to 3.5 present the four identified groups and representative of each group. Section 3.6 compares these representatives with 19 criteria in five groups.

3.1 Definition, Search, and Grouping

For comparing TSDBs, it is necessary to define which databases belong to this group as the term “TSDB” is not precisely defined or used. For this work, a DBMS that can (i) store a record that consists of timestamp, value, and optional tags, (ii) store multiple records grouped together[1], (iii) can query for records and (iv) can contain time ranges in a query is called TSDB. A further constraint is, that it must be possible to store more than one value per timestamp with different tags and to make queries that contain a time range (SCAN). This means that the “timeseries”-character of the DBMS must inherently exist, it must be possible to do queries by timestamps and time ranges without modeling it on another structure (as done for two RDBMS in this thesis). A RDBMS with a timestamp as primary key would be easy to think of as a TSDB, but it is necessary to have the possibility to store more than one entry per primary key[2]. Most TSDBs also provide some functions for querying like AVG, SUM, CNT and the possibility to use a granularity. The use of granularity helps to retrieve results with other time steps than originally stored: e. g., storing a sensor value every millisecond but retrieving an AVG value for each second.

The process of finding TSDBs was done as follows:

[1] E. g., in a metric.

[2] This problem was solved by using an additional auto-incrementing primary key in the JDBC (SQL) part of YCSB-TS, see Appendix A.2.

1. Search with Google for "`"time series database" OR "timeseries database" OR "tsdb"`" resulted in 284,000 results[3]. The first ten pages with together 100 results were considered and each result individually searched for TSDBs.
2. Search with "any field" and "matches any" operators for "`("time series database" "timeseries database" "tsdb")`" for papers in ACM Digital Library resulted in 44 results[4]. Each result was individually considered and searched for TSDBs.
3. Search with "`(("time series database") OR "timeseries database") OR "tsdb")`" in "Metadata Only" for papers in IEEE Xplore / Electronic Library Online (IEL) resulted in 54 results[5]. Each result was individually considered and searched for TSDBs.
4. Searching "Time series database" article on Wikipedia (English)[6].
5. Following further pointers found in step 1, 2, 3, and 4.

As a result, 75 TSDBs were found, 42 of them open source and the rest are proprietary. Discussion of every single TSDB extends the scope of this thesis, but while using different TSDBs it was found that some of them have similarities. These made it possible to group them into four groups and discuss a smaller number of each group in detail. The following groups were identified:

- Requirement on NoSQL DBMS
- No requirement on any DBMS
- RDBMS
- Proprietary

[3]The search URL was: https://www.google.com/search?q=%22time+series+database%22+OR+%22timeseries+database%22+OR+%22tsdb%22&pws=0&gws_rd=cr%2Cssl&gl=us&inurl=https&btnG=Google+Search&hl=en.

[4]The search URL was: http://dl.acm.org/results.cfm?query=(%22time%20series%20database%22%20%22timeseries%20database%22%20%22tsdb%22)&within=owners.owner=ACM&filtered=&dte=&bfr=.

[5]The search URL was: http://ieeexplore.ieee.org/search/searchresult.jsp?action=search&sortType=&rowsPerPage=&searchField=Search_All&matchBoolean=true&queryText=(((.QT.time%20series%20database.QT.)%20OR%20.QT.timeseries%20database.QT.)%20OR%20.QT.tsdb.QT.).

[6]URL: https://en.wikipedia.org/wiki/Time_series_database#Example_TSDB_Systems.

The first group exists of TSDBs that depend on an already existing NoSQL DBMS[7]. In the second group are TSDBs, which can run completely independent of other DBMS. The third group encloses RDBMS or RDBMS-based TSDBs. Some TSDBs can fit in more than one group, for example such that can optionally use a DBMS. In this case the decision in which group a TSDB is put depends on how the main part of it looks like. For example: Is it a cluster system in which some parts can use other DBMS (e. g., Druid) or a client written for an existing NoSQL DBMS (e. g., Rhombus). The three first groups contain open source TSDBs and DBMS only, as the main factor of the thesis is on them. As last group, Proprietary contains all commercially or freely available TSDBs that are not open source.

For each of the first three groups, there were TSDBs/RDBMS chosen based on popularity. Popularity was decided by the amount of results in Google search, see Appendix A.3 for further explanation and results. Based on the results in Table A.1, there should be at least three more TSDBs/RDBMS in this list: Prometheus [Pro15b], RRDtool [Oet15] and MySQL Cluster [Ora15e]. In a discussion with the supervisors [BKF15], it has been decided to additionally use Blueflood in this comparison.

Prometheus is a monitoring solution based on a pull-based model[8], hence the popularity. Prometheus mainly uses pull-based INS queries instead of push-based INS queries. It can do push-based INS queries, but needs a second component, called Prometheus Pushgateway [Pro15c], for push-based INS queries. The data that was pushed to the Pushgateway by one or more clients, is regularly pulled (“scraped”) by Promotheus. Promotheus normally replaces the timestamps of the pushed data with a new timestamp at the time of scraping. This can be avoided by setting an explicit timestamp field, which is considered not useful in most use cases (See [Pro15d]). In a discussion with the supervisors [BKF15], it has been decided against using Promotheus in this comparison.

RRDtool is one of the oldest tools for storing time series data. It uses a Round Robin Database (RRD) for storing data of a specific time range (e. g., a RRD that holds one year of data) and can aggregate data automatically (consolidation). As it has no client-server structure, lacks distribution and scalability, and has a problematic high Input/Output load factor when running updates [Car15], it is not considered for comparison.

[7] E. g., Cassandra, HBase, CouchDB.

[8] Pull-based means that Prometheus periodically asks for data.

MySQL Cluster is a MySQL variant[9] that has more features for a better scalability and distribution than MySQL Community Server. With MySQL Community Server there is already one of the most popular RDBMS (see Appendix A.3) used for comparison and since RDBMS are not the main aspect of the comparison, MySQL Cluster was not used in comparison.

Examining a TSDB was done in the following order[10]:

1. Architecture and dependencies
2. Initialization of the TSDB, metric and client
3. Functions (see Chapter 1) and Restrictions (e. g., no use of tags, cannot add metrics while ingestion)
4. User Interface (UI)[11], Graphical User Interface (GUI)[12] and clusterability aspects (e. g., HA, scalability)

The following sections introduce each group and two or more representatives of each group. In the following section, a feature comparison between them is done. A full list of all TSDBs found can be found in Appendix A.1.

3.2 Group 1: TSDBs with a Requirement on NoSQL DBMS

There are twelve existing TSDBs that belong to the first group. In this thesis five TSDBs of this group were taken into comparison. All of them depend on Cassandra [ASF15a] or HBase [ASF15d] for storing their time series data.

Blueflood is a “multi-tenant distributed metric processing system” [Rac15a] that uses Cassandra for storing its time series data. Blueflood consists of a server and a client component, both written in Java. Zookeeper [ASF15i] is optional used to coordinate locking on different shards while performing rollups. Elasticsearch [Ela15] is optionally used to search for metrics. Blueflood uses log4j [ASF15f] for logging and can optionally

[9] MySQL Cluster and MySQL Community Server are both products of the Oracle Corporation.

[10] It is not always adhered to the order as, for example, Druid requires more explanation than other TSDBs.

[11] Every non-graphical endpoint for user interaction with the TSDB is considered an UI, for example a REST interface via HTTP or a Command-Line Interface (CLI) tool. A library is not considered an UI.

[12] Every graphical endpoint for user interaction with the TSDB is considered a GUI, for example a webpage via HTTP.

use Coda Hale Metrics [Cod15] and JMX [Ora15c] for performance monitoring. It supports rollups every 5, 20, 60, 240, and 1440 minutes that calculate minimum, maximum, variance, and average data for further, more space efficient, storage. The use of tags is not possible in Blueflood, but is in development [Rac15b]. Metrics can be added "on the fly" while ingesting data. SUM is not implemented, but MAX, CNT, AVG, and MIN are. Granularities can be used in the same steps as rollups, with the insertion granularity as sixth alternative [Rac15c]. Blueflood does not provide any UI besides itself as a set of Java classes that can be executed via CLI. HA is possible by using more than one node per shard with coordination via Zookeeper. Scalability and load balancing do not exist beyond the existing abilities of Cassandra. An interface for JSON via HTTP, Kafka [ASF15e], Cloudfiles [Rac15d], StatsD [Ets15], Graphite [Dav15] (performance metrics only), and the possibility to ingest via UDP are provided.

KairosDB [Kai15b] uses H2 [Mül15] or Cassandra as a storage for time series data and describes itself as "Fast Time Series Database on Cassandra" [Kai15b]. H2 is considered as slow and only recommended for testing or development purposes (See [Mer15a]). HBase was also supported, but support was dropped due to missing features[13]. In the results in the rest of this chapter, Cassandra was used as the backend for KairosDB. Replication Factor (RF) and cluster information can to be configured in a configuration file. CNT, AVG, and SUM are supported[14]. Granularity can be chosen, but must be at least one millisecond. One cannot pick an exact point in time because a time range with at least one millisecond has to be provided, even for READ. No dependencies are required when using H2, otherwise Cassandra is required. An UI is provided in form of a HTTP interface. A Java library as well as REST, graphite, and telnet interfaces are provided.

NewTS [Eva15] is a "time-series data store based on Cassandra". It consists of a server and client, which both are written in Java. The given schema manager always creates keyspaces with a RF of one, which means that the RF must be changed afterward or the keyspace must be created by hand before the schema is applied. Queries can use AVG, MIN, and MAX as aggregating functions, but CNT and SUM are not available. Granularity can be chosen but must be at least two milliseconds for every aggregating function. It is not possible to query for tags and time range (or timestamp) at the same time. Filtering for tags is possible when not using aggregating functions. No other dependencies than Cassandra are needed. An UI is provided in form of a HTTP

[13]The KairosDB homepage still documents configuration options for HBase, but is says that support was dropped [Kai15c].

[14]There are more aggregating functions supported, this list is not extensive.

interface. SLF4J [Gül15] is used for logging. NewTS does not provide any approaches for HA, load balancing, or scalability other than already exist in Cassandra. A Java API as well as REST and graphite interfaces are provided.

OpenTSDB [LSKD15a] is a TSDB that uses HBase for storing its time series data. HBase uses Zookeeper for coordination between nodes. Before OpenTSDB can be started, HBase tables must be created. A metric can be created before ingestion but OpenTSDB can also be set to generated metrics automatically at first ingestion on a new metric. CNT[15], AVG, and SUM are supported[14]. Granularity can be one millisecond or second, but storage is only guaranteed on a granularity of one second. It is not possible to ingest data with millisecond-level granularity. Every query needs an aggregating function, which results in no possibility to perform a READ on a timestamp without any aggregating function, which means that values on the same timestamp and on the same metric must be distinguishable by tag values. HBase and Gnuplot are required as dependencies. HBase needs Zookeeper as dependency. An UI is provided in form of a HTTP interface that draws graphs with Gnuplot. A REST interface is provided. Compression for HBase (LZO) in productive use is advised, but must be compiled manually [LSKD15b]. HA and scalability can be achieved by using more than one OpenTSDB instance and the abilities of HBase. Load balancing is possible by using Domain Name System (DNS) Round Robin, HAProxy [Tar15], or Varnish Cache [Var15]. The latter one can load balance READ queries only.

The last member of this group, *Rhombus* [Par15b], is a client for Cassandra, written in Java [Ora15b], and comes with a schema for writing "Keyspace Defintions". The "Keyspace Defintions" are a set of definitions that used by Rhombus to create a metric inside Cassandra. Each metric has a set of fields, every field has a type (e. g., varchar, int)[16]. For each metric indexes must be defined. Each index needs a sharding strategy and one or more fields. They describe which fields are possible to query in the later used queries. A query on one or more keys without an existing and fitting index is not possible. It is impossible to create metrics or add support for a higher number of tags "on the fly" while ingesting data. Rhombus supports CNT as function, AVG and SUM are not available. Setting a granularity or resolution for queries is not supported. Combining tags with Boolean algebra is poorly supported, as it is only possible to define an index on one or more fields for combining them. That can be used as a Boolean "AND". It has no other dependencies than the one to Cassandra. Since Rhombus is a client only and there is no server part, it is possible to bypass any

[15] Only supported in version 2.2.0, which is not yet stable (2015–11–29).

[16] The fields describe all components of a time series row: timestamp, value and tags. For the rest of this paragraph, the term "fields" is used with this definition.

limitations or develop missing features by directly using Cassandra Query Language (CQL). Rhombus does not provide any UI or approaches for HA, load balancing, or scalability other than already existing in Cassandra. As Rhombus is a Java client, there is no other possibility for connecting than using its Java API.

3.3 Group 2: TSDBs with no Requirement on any DBMS

The second group consists of 12 TSDBs that all follow different approaches, but have in common that they have optional dependencies on other DBMS for storing time series data only. Some of them need other DBMS for storing metadata. Three of them were taken into comparison in this thesis.

The first member of this group is *Druid* [YTL+14], which describes itself as a "fast column-oriented distributed data store". Druid uses a RDBMS like Derby, MySQL Community Server, or PostgreSQL as metadata storage and Zookeeper for coordination. It also needs a distributed storage like HDFS, S3 [AWS15], Cassandra, or Azure [Mic15a], but can be run with a local filesystem for testing purposes.

Druid configuration is more complex in comparsion to other regarded systems, since it has five different node types that need to be configured: Historical, Coordinator, Broker, Realtime, and Indexing Service. Each of them serves a specific purpose: *Historical* nodes load segments of (compressed) time series data from the distributed storage if a query requests it. New segments are assigned by the *Coordinator* node, which is responsible for assignment, distribution, load balancing, replication, and erasure of segments. Every segment is announced via Zookeeper. Segments are used by the *Broker* node to answer a query in a setup with more than one node (cluster). It uses the announcements of segments to route the queries to the fitting nodes and merges the results afterward.

Druid knows two different node types for data ingestion: *Realtime* nodes receive time series data and index it in realtime. This means that this data is immediately available for queries and is handed over to historical nodes in segments of a configured time span. Endpoints, called *Firehoses*, for different sources, like Kafka, S3, Azure, Spritzer, RabbitMQ, etc., are available. This type of ingestion is called "Realtime Ingestion".

The other approach, which is called "Batch Ingestion", uses an *Indexing Service* that consists of three node components: *Overlord*, *Middle Manager* and *Peon*. If a client wants to ingest data via the Indexing Service, it needs to contact the *Overlord* for creating a task on a *Middle Manager*. The *Overlord* is responsible for coordination (locking, distribution, etc.) of these tasks. A *Middle Manager* creates a locally running

Peon that creates an endpoint to which the client can continuously send its data. This *Peon* component segments the data and passes the segments to the *historical* node. *Middle Managers* and their *Peons* typically run on the same machine, while an *Overlord* can be on another node. An *Overlord* can also be used as *Middle Manager* in a small scenario, for example. There is also the possibility to replace the *Indexing Service* with a task called *HadoopDruidIndexer* that uses Hadoop to replace the *Indexing Service*. An *Indexing Service* can be used to ingest data from stream processors like Samza [ASF15g] or Storm [ASF15h] via the *Tranquility* API [Mer15b]. Despite the misleading names, "Batch Ingestion" is preferred over "Realtime Ingestion" if it is required to continuously ingest real time data into Druid. The reason behind this preference is that "Realtime Ingestion" was developed first, but Kafka, which is one of the most popular messaging systems, can hardly achieve HA [YTL+15a]. As a result, "batch ingestion" was developed.

Druid supports replication for backup and HA purposes. Improvement in performance by replication may exist, but was not the main reason for replication in Druid [Yan14]. CNT, AVG, and SUM are supported[14]. Self-written functions as aggregators in JavaScript are also possible. Granularity can be chosen, but must be at least one millisecond. Ingested data can be stored aggregated with a given granularity for saving space, which is called "rollup". UIs are provided in form of several HTTP interfaces. HA, scalability, and load balancing can be achieved by increasing the replication factor and the amount of existing nodes per type. For querying data, a REST interface, which uses JSON, is available. There are several firehoses for ingestion available, see the two paragraphs above.

InfluxDB [Inf15b] is a TSDB that does not depend on any other DBMS. Before data can be ingested, a database must be created. A database can then accept different metrics without the need of creating each metric. InfluxDB uses two protocols for data ingestion: a text based protocol called "Line Protocol" and a deprecated JSON protocol [Inf15e]. For every other query it uses a Structured Query Language (SQL)-like syntax called InfluxDB Query Language (InfluxQL). CNT, AVG, and SUM are supported[14]. Granularity can be chosen, but must be at least one millisecond and must be chosen cautious because the GROUP BY statement is only allowed to create 100,000 buckets at maximum [Bec15b]. Each bucket of a calculation contains the aggregated result of a certain part of the given time range. This leads to a problem if a function calculates over a time range exceeding 100,000 milliseconds and groups the result in milliseconds, for example. It is possible to run functions automatically at certain periods (named "continuous query" [Inf15d]) that can be used for "rollup" (see Section 3.3), for example. An UI is provided in form of a HTTP interface and a CLI client. Performance metrics are internally logged. Interfaces for querying via HTTP, UDP, OpenTSDB, Graphite, and Collectd [For15b] are provided. HA, load

balancing and scalability can be achieved by using more than one InfluxDB instance[17]. Using more than one instance as a cluster is an experimental feature [Inf15c]. A cluster can consist of several nodes, where the first started node together with the two first connected (to the first node) nodes are “raft nodes”, which means they are able to perform decision processes. All other nodes are called “data nodes” and cannot participate in decisions.

MonetDB is a “column store” [Mon15a] that uses SQL as query language. MonetDB does not depend on any other DBMS. Before data can be ingested, a database and a table must be created. While most TSDBs use metrics or any related concept, MonetDB uses databases and tables, which is typically used by most SQL-based RDBMS. Every SQL statement is internally translated to a MonetDB Assembly Language (MAL) statement. The differences between the SQL dialect of MonetDB and other RDBMS are described by Wikibooks [Wik15] and Mon2015a. There are MonetDB SQL bindings for different interfaces existing: Java Database Connectivity (JDBC), Open Database Connectivity (ODBC), PHP, Perl, Ruby, Node.js, Python, and Mapi (C binding). CNT, AVG and SUM are supported[14]. It is possible to ingest data with millisecond-level granularity[18]. No UI is provided besides a CLI client. Load balancing and scalability can be achieved by using “merged tables” together with more than one MonetDB instance. For doing so, existing tables must be manually grouped together to “merged tables”, which results in a horizontally split table. No UPDATE or INS can be run on a “merged table” and need to be run on one of the single tables instead. As a result, achieving load balance and scalability is not an automated process. Since 2015 it is also possible to span “merged tables” across different databases [Mon15b]. An approach for achieving HA by using “Transaction Replication” exists as an experimental feature [Mon15d].

3.4 Group 3: RDBMS

The third group consists of five RDBMS. There are many more RDBMS existing, this is a exemplary list. Two of them are used in this comparison. Both are compared to gain insight how “traditional” RDBMS perform against TSDBs in a time series environment.

[17]The Java client can only connect to one node, which may interfere with HA and load balancing, see [Maj15].

[18]Depending on the table schema.

For adopting RDBMS, a table structure that consists of an automatically incrementing primary key, a timestamp, a floating point value, and several tags are used. See Appendix A.2 for further details on the structure. At first, a table structure with the timestamp as primary key was used, but that led to a problem: A TSDB can contain more than one entry with the same timestamp (e. g., two sensors that are separated by tags) and the primary key needs to be unique. For solving this problem, a new column with an automatically incrementing primary key, called "id", was added. Another option is to use a combined primary key that consists of the value, all tag values and the timestamp. This option was not used due to two reasons: First, tag values are optional, which means that a row can have zero tag values. When two sensors insert the same value at the same time, both have the same primary key. When two rows have the same primary key, the unique criteria of a primary key is not fulfilled anymore. Secondly, because it requires more calculation steps than an automatically incrementing counter as for a new entry a combined key must be generated and checked if it is unique by the DBMS[19].

MySQL Community Server [Ora15f] is a popular RDBMS, which does not depend on any other DBMS. Before data can be ingested, a database and a table must be created. The schema described above was used in MySQL SQL dialect that can be seen in listing Listing A.2. Since MySQL Community Server is very popular, there are many bindings for different interfaces existing: ODBC, JDBC, J, Python, C++, C, and more. CNT, AVG, and SUM are supported[14]. It is possible to ingest data with millisecond-level granularity[20]. No UI is provided besides a CLI client[21]. Performance metrics are internally logged. Load balancing and HA for READ queries can be achieved by using several nodes with master to slave replication and DNS Round Robin or third-party load balancing software like Varnish Cache or HAProxy. Load balancing and HA for write queries and scalability for all queries cannot be achieved as MySQL Community Server is not able to perform master to master replication [Ora15g]. This does not lead to scalability like it would when adding nodes to a TSDB, as every node needs to store all tables completely. As a result, it is possible to gain performance for answering queries faster or more queries at the same time, but every write has to be replicated to all nodes, which may decrease performance. Additionally, the size of all databases can

[19] It must be considered that every primary key is checked if it is unique by the DBMS, even if it as an automatically incrementing primary key. It is assumed that this check requires more calculation steps for a combined key than for a automatically incrementing key, as several rows must be compared instead of one.

[20] Depending on the table schema.

[21] There exist many popular third-party UIs like phpMyAdmin [php15a], for example.

not increase, compared to running on a single node because it must still fit on every node. Possible solutions include the usage of MySQL Cluster [Ora15e].

PostgreSQL [PGD15a] is another popular RDBMS that does not depend on any other DBMS. Before data can be ingested, a database and a table must be created. The schema described above was used in PostgreSQL SQL dialect that can be seen in listing Listing A.3. Since PostgreSQL is very popular, there are many bindings for different interfaces existing: JDBC, Python, C++, C, and more[22]. CNT, AVG, and SUM are supported[14]. It is possible to ingest data with millisecond-level granularity[23]. No UI is provided besides a CLI client[24]. Performance metrics are internally logged. Load balancing, scalability, and HA can be achieved by using several nodes that run PostgreSQL. The problem is synchronization between the nodes that run PostgreSQL. The main ideas to solve this problem and the resulting disadvantages are the same as for MySQL Community Server (see above). PostgreSQL supports only master to slave replication and also offers dividing the data into two or more partitions that are put onto two or more nodes [PGD15b]. If an application needs data from one partition, it can only connect to the specific instance. If it needs data from both partitions, it needs to connect to both databases. This approach leads to the possibility to increase the storage by adding nodes. There are, however, several drawbacks as the “logic” is shifted from the database to the client: Each client must know every partition on any instance. If a new instance is added, it must be made public to every client. Every client also needs rules to choose the correct node for storing data for keeping a load balancing strategy. These rules also need to be updated when a instance is added. The last drawback is the resulting overhead when there are queries used, that span across nodes: For each query, a query to all needed nodes must be made and the results must be merged by the client. A solution to this is the use of master to slave replication of each node on each node, which results in a lack of scalability: Each node would need to store all data, which makes more free space by adding nodes impossible.

[22] Some of them are third-party software.

[23] Depending on the table schema.

[24] There exist many popular third-party UIs like phpPgAdmin [php15b] or pgAdmin III [pgA15], for example.

3.5 Group 4: Proprietary

This group consists of proprietary TSDBs and services that offer storage and/or analysis for time series data. Because this thesis focuses on open source DBMS, this group is not described any further than listing group members in Appendix A.1.

3.6 Feature Comparison of TSDBs

The TSDBs from the first three groups are compared in this section. The criteria are grouped into five groups:

1. Distribution/Clusterability
2. Functions and Long-term storage
3. Granularity
4. Interfaces and Extensibility
5. Support and License

The first group compares the following *Distribution* and *Clusterability* features: HA, scalability, and load balancing.

HA gives the possibility to compensate unexpected node failures and network partitioning. To compensate means that a query must be answered under the mentioned circumstances, but it is not expected that the given answer is always consistent[25]. It is expected that a client needs to know more than one entry point (e. g., IP) to compensate a node failure of an entry node.

Scalability is the ability to increase storage or performance by adding more nodes. The ability must exist in the server part of the TSDB, otherwise adding a second TSDB and giving the client side the possibility to use two TSDBs in parallel would also result in scalability and a increased performance[26].

Load balancing is the possibility to equally distribute queries across nodes in a TSDB, so that the workload of each node has just about the same level. If a TSDB uses another

[25] It is expected that the DBMS uses eventual consistency at least.

[26] Depending on the queries. If all queries only use the same node, then no performance increase is possible for these queries.

DBMS, it is checked if only the DBMS or the TSDB or both have these features. As seen in Table 3.1, Druid the only TSDB fulfilling all three criteria without restrictions. Both RDBMS do not fulfill any criteria, the remaining DBMS do with restrictions (e. g., fulfill them only for some parts).

TSDB	HA	Scalability	Load balancing
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Blueflood	available	available[28]	available[28]
KairosDB	available[28]	available[28]	available[28]
NewTS	available[28]	available[28]	available[28]
OpenTSDB	available	available	not available[27]
Rhombus	available[28]	available[28]	available[28]
Group 2: TSDBs with no Requirement on any DBMS			
Druid	available	available	available
InfluxDB	available[29,30]	available[30]	available[2930]
MonetDB	available (experimental)[31]	available with restrictions[32]	available with restrictions[32]
Group 3: RDBMS			
MySQL Community Server	not available[33]	not available	not available[33]
PostgreSQL	not available[34]	not available	not available[34]

Table 3.1: Comparison of criteria group 1: Distribution/Clusterability.

[27] Possible to achieve with DNS Round Robin or external tools like Varnish Cache (only for READ queries) or HAProxy.

[28] Only for the Cassandra part.

[29] The Java client can only connect to one node which may interfere with HA and load balancing, see [Maj15].

[30] Clustering is an experimental feature [Inf15c].

[31] "Transaction Replication" is an experimental feature [Mon15d]. See Section 3.3.

[32] Available with "merged tables" that cannot run INS or READ queries. See Section 3.3.

[33] Possible to achieve for READ queries with DNS Round Robin or external tools like Varnish Cache or HAProxy, see Section 3.4.

[34] Possible to achieve for READ queries with DNS Round Robin or external tools like Varnish Cache or HAProxy, see Section 3.4.

Continuous calculation, the availability of AVG, SUM, CNT functions, and long-term storage are compared in group 2: *Functions and Long-term storage*. Continuous calculation means that a TSDB, having this feature, can continuously calculate functions based on the input data and stores the results. An example is the calculation of an average per hour. As YCSB-TS uses AVG, SUM, and CNT as aggregating functions (see Section 4.2), it is checked if these three are available. See Appendix A.6 for a more detailed list of supported functions for each TSDB. It is also checked if tags are available as these are needed to differentiate different sources (e. g., different sensors). A solution for long-term storage is needed for huge amounts of data, as it is challenging to store every value in full resolution over a longer period, considering the city from Chapter 1 with 360,000 values per second (on a full rollout) as an example. Solutions could range from throwing away old data to storing aggregated values of old data (e. g., storing only an average over a minute instead of values for every millisecond). Solutions that are running outside the TSDB are not considered (e. g., a periodic process that runs queries to aggregate and destroy old data). As seen in Table 3.2, no DBMS fulfills any criteria without restrictions. Druid and InfluxDB can fulfill them with small restrictions on long-term storage and continuous calculation. Three DBMS do not fully support AVG, SUM, and CNT, one does not offer tags.

In group 3 "*Granularity*", downsampling, the smallest possible granularities that can be used for functions and storage, as well as the smallest guaranteed granularity for storage, are compared. When using queries with functions, most TSDBs have the ability to use downsampling for fitting the results when a result over a greater period of time is wanted (e. g., an average for every day of a month and not every millisecond). Downsampling does not mean that you can choose a period of time and get one value for that period. For downsampling two periods must be chosen

[35] See [Rac15b].

[36] Filtering for tag values can not used in combination with time ranges or aggregation functions, which results in a limited tag functionality, see Section 3.2.

[37] Versions above or equal 2.2.

[38] Boolean algebra is only poorly supported which results in a limited tag functionality, see Section 3.2.

[39] Druid uses an immediate "rollup" (see Section 3.3) to store ingested data at a given granularity which helps for long-term storage but there are no further "rollups" after the initial "roll-up". This can be used in combination with rules for data retention and kill tasks (see [YTL+15b]) to achieve long-term storage.

[40] See [Inf15d].

[41] Using coninuous queries for downsampling and retention policies [Bec15a; Inf15d].

[42] Via triggers or views.

TSDB	Continuous calculation	Tags	AVG	SUM	CNT	Long-term storage
Group 1: TSDBs with a Requirement on NoSQL DBMS						
Blueflood	n.a.	n.a.[35]	av.	n.a.	av.	av.
KairosDB	n.a.	av.	av.	av.	av.	n.a.
NewTS	av.	av.[36]	av.	n.a.	n.a.	n.a.
OpenTSDB	n.a.	av.	av.	av.	av.[37]	n.a.
Rhombus	n.a.	av.[38]	n.a.	n.a.	av.	n.a.
Group 2: TSDBs with no Requirement on any DBMS						
Druid	av.	av.	av.	av.	av.	av.[39]
InfluxDB	av.[40]	av.	av.	av.	av.	av.[41]
MonetDB	av.[42]	av.	av.	av.	av.	n.a.
Group 3: RDBMS						
MySQL Community Server [Ora15f]	av.[42]	av.	av.	av.	av.	na.
PostgreSQL	av.[42]	av.	av.	av.	av.	na.

Table 3.2: Comparison of criteria group 2: Functions and Long-term storage (av. = available, n.a. = not available).

and a value for each smaller period within the bigger period must be returned. The smaller period is called sample interval. Granularity describes the smallest possible distance between two timestamps, see Chapter 1. When inserting data into a TSDB, the granularity of the input data can be higher than the storage granularity that a TSDB guarantees to store safely. Some TSDB accept data in a smaller granularity than they can store under all circumstances, which leads to aggregated or dropped data. For TSDBs that use other DBMS for storing their data, some of the compared aspects can be implemented manually with direct queries to the DBMS. Such solutions are not considered. As seen in Table 3.3, eight DBMS support downsampling without restrictions, six support sample intervals, storage granularities and guaranteed storage granularities down to one millisecond. Two DBMS do not support downsampling, another does only guarantee to store in second-level granularity, and one cannot use time ranges smaller than two milliseconds.

[43] See [LSKD15c].

[44] Depends on the types used in keyspace definition.

TSDB	Down-sampling	Smallest sample interval	Smallest granularity for storage	Smallest guaranteed granularity for storage
Group 1: TSDBs with a Requirement on NoSQL DBMS				
Blueflood	not available	not available	1 ms	1 ms
KairosDB	available	1 ms	1 ms	1 ms
NewTS	available	2 ms	1 ms	1 ms
OpenTSDB	available	1 ms	1 ms	1000 ms[43]
Rhombus	not available	not available	1 ms[44]	1 ms[44]
Group 2: TSDBs with no Requirement on any DBMS				
Druid	available	1 ms	1 ms	1 ms
InfluxDB	available	1 ms[45]	1 ms	1 ms
MonetDB	available[46]	1 ms[46]	1 ms[47]	1 ms[47]
Group 3: RDBMS				
MySQL Community Server	available[46]	1 ms[46]	1 ms[47]	1 ms[47]
PostgreSQL	available[46]	1 ms[46]	1 ms[47]	1 ms[47]

Table 3.3: Comparison of criteria group 3: Granularity.

The fourth group, *Interfaces and Extensibility*, compares APIs, interfaces, client libraries, and plugins. In this group, available APIs, interfaces and client libraries, that are not third-party, are listed[48]. It was also checked if an interface for plugins exists. As seen in Table 3.4, six DBMS support different APIs and interfaces like CLI, HTTP, etc., three only support a CLI interface, and one does not offer support via any API or interface. Seven DBMS offer client libraries in Java, four in more languages, and three do not offer any client library. Plugins are supported by six DBMS.

[45]Amount of resulting intervals (bucket) is limited to 100,000, see [Bec15b].

[46]Implemented via SQL with WHERE ... AND ... BETWEEN query.

[47]Depends on the types used in table definition.

[48]Except for PostgreSQL, as it is not clear who developed the libraries.

[49]A set of executable Java classes.

[50]Experimental feature.

TSDB	APIs and interfaces	Client libraries	Plugins
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Blueflood	CLI[49], HTTP (JSON), Kafka, statsD[50], Graphite, UDP	not available	not available
KairosDB	CLI, HTTP(REST + JSON, GUI), graphite, telnet	Java	available
NewTS	HTTP(REST + JSON, GUI)	Java	not available
OpenTSDB	CLI, HTTP(REST + JSON, GUI), Kafka, S3, Azure, Spritzer, RabbitMQ,	not available[51]	available
Rhombus	not available	Java	not available
Group 2: TSDBs with no Requirement on any DBMS			
Druid	CLI, HTTP(REST + JSON, GUI), Storm[52], Samza[52], Spark[52]	Python, R, Java[52]	available
InfluxDB	CLI, HTTP(InfluxQL, GUI), UDP, OpenTSDB[53], Graphite[53], Collectd[53]	not available	available
MonetDB	CLI	Java (JDBC), ODBC, PHP, Perl, Ruby, Node.js, Python and Mapi (C binding)	not available
Group 3: RDBMS			
MySQL Community Server	CLI	Java (JDBC), ODBC, J, Python, C++, C, ...[54]	available
PostgreSQL	CLI	Java (JDBC), ODBC, Tcl, Python, C++, C, ...[5455]	available

Table 3.4: Comparison of criteria group 4: Interfaces and Extensibility.

Support and License is the fifth and last group and compares the availability of a stable version (LTS) and commercial support, as well as the license used. When using any TSDB it might be important to have a “stable” branch , that only receives security and bug fixing updates over a longer period, while there are newer versions (which are not “stable”) released. Updates normally contain security and bug fixes and new features, which can lead to changes that must be made to ones program. This requires development and testing for every update. A “stable” version guarantees that an update does not change anything that requires development or testing in ones program, which can be cost saving in the long term, as there is less development required[56]. If a issue in a TSDB is encountered, it needs to be solved by internal or external developers. In a situation where defined reaction times are required, that cannot be achieved internal, or when an internal development team is “stuck”, commercial support is very helpful. It might also save costs, when it is cheaper to outsource required TSDB development. Only commercial support from the manufacturer/developer(s) of a TSDB are considered. When developing or using an open source TSDB, it is important with which license the TSDB is released. A license regulates how and by whom a product can be used and what modifications are allowed to it. It also gives long term safety, as it is clear on which rules a TSDB can be used, which can be problematic if a TSDB does not have a license. As seen in Table 3.5, only the two RDBMS offer stable/LTS versions. Commercial support is available for three DBMS. All DBMS are released under licenses like GNU Lesser General Public License version 2.1 (LGPLv2.1)[57] [Fre99], GNU General Public License version 2 (GPLv2)[57] [Fre91], GNU General Public License version 3 (GPLv3)[57] [Fre07], Massachusetts Institute of Technology (MIT) [Mas88], or Apache 2.0 [ASF04], two use custom licenses. There are difference between those licensese. g., GPLv2 requires to provide modified and released source code of the complete system where the code is part of. MIT or Apache 2.0 do not have copyleft guidelines and products released under those licenses are better suited for this purpose [Spi03].

[51] There are many clients made by other users.

[52] Via Tranquility.

[53] Via plugin.

[54] This listing is not extensive.

[55] Some of them are third-party software.

[56] Depending on the issue(s), “stable” updates can also include API or interface changes that require development.

[57] A suffixed plus sign indicates that later versions of this license can be used to license this software.

[58] Their “normal” releases are called “stable releases”, but are not what is considered “stable” in this thesis as every release is called like that after it has passed several release candidate stages.

TSDB	LTS/stable version	Commercial support	License
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Blueflood	not available	not available	Apache 2.0
KairosDB	not available	not available	Apache 2.0
NewTS	not available	not available	Apache 2.0
OpenTSDB	not available	not available	LGPLv2.1+, GPLv3+
Rhombus	not available	not available	MIT
Group 2: TSDBs with no Requirement on any DBMS			
Druid	not available[58]	not available	Apache 2.0
InfluxDB	not available	available[59]	MIT
MonetDB	not available	available[60]	MonetDB Public License Version[61]
Group 3: RDBMS			
MySQL Community Server	available[62]	available[63]	GPLv2
PostgreSQL	available[64]	not available[65]	The PostgreSQL License[66]

Table 3.5: Comparison of criteria group 5: Support and License.

In conclusion of the feature comparison, Druid seems to be the best choice, as it fulfilled any of the criteria from the four groups besides having a stable/LTS version and commercial support. Druid is complexer than other TSDB and it requires more effort to get it running. A less complex DBMS like OpenTSDB or InfluxDB can be a

[59] See [Inf15a].

[60] See [Mon15e].

[61] "This License is a derivative of the Mozilla Public License (MPL) Version 1.1, where the difference is that all references to 'Mozilla' and 'Netscape' have been changed to 'MonetDB', and that the License has been made subject to the laws of The Netherlands" [Mon15c].

[62] See [Ora15d].

[63] See [Ora15h].

[64] See [PGD15e].

[65] Several commercial support companies are listed on the PostgreSQL homepage, see [PGD15d].

[66] See [PGD15c].

better choice. Both do not offer a client library or stable/LTS versions, OpenTSDB additionally guarantees only second based storage granularity and cannot perform load balancing. InfluxDB has clustering as an experimental feature only, but offers commercial support. RDBMS can be an alternative as they offer a stable/LTS version, but they lack the abilities for distribution and clustering and thus can be problematic when scalability is needed. They also have many client libraries due to their long history and wide popularity. MonetDB does also offer many client libraries and distribution as well as clustering features with restrictions. Six of seven DBMS that do not offer many client libraries, do offer many interfaces or APIs instead, which can be an alternative. When using a modified DBMS in a commercial product, a license that requires strong copyleft, should be avoided when it is not desired to release the source code of the product under the same license. In Chapter 5 a performance comparison using YCSB-TS between these ten DBMS is done. Four recommendations for different requirements based on this feature comparison:

1. If fulfillment of all criteria of groups 1 and 2, a granularity of 1 ms, but no commercial support, and no stable/LTS version is required, Druid is the best choice.
2. If all criteria, commercial support, no stable/LTS version, no existing client libraries are required, and experimental features are tolerated, InfluxDB is the best choice.
3. If not all criteria, but a stable/LTS version is required, then one of the two RDBMS are the best choice.
4. If not all criteria, but many client libraries and distribution as well as clustering features are required, MonetDB is the best choice if the restrictions in clustering and distribution can be tolerated.

These criteria excluded a performance comparison with the metrics of Section 4.1. If these metrics have to be taken into account, a benchmark is required. This is presented in the next chapter.

4 Concept of a Time Series Database Benchmark

This chapter presents the metrics and scenarios used for benchmarking TSDBs, as well es the requirements on the benchmark, and the benchmark used in this thesis which is named TSDBBench. It is structured as follows: Section 4.1 describes metrics for a performance comparison between TSDBs. Section 4.2 presents the scenarios and metrics used for benchmarking TSDBs. Section 4.3 describes the requirements for the TSDB benchmark used in this thesis. These requirements are used in Section 4.4 to introduce the components that are used to fulfill them. Section 4.5 presents the architecture of TSDBBench. Section 4.6 describes the benchmark used inside TSDBBench which is named YCSB-TS. Section 4.7 concludes this chapter with peculiarities for each TSDB (if existing) that were necessary to avoid.

4.1 Metrics

For benchmarking it is necessary to define what should be measured and how to evaluate the results for creating a ranking and distinguish between different results. For doing so, a performance metric must be specified. YCSB-TS uses two metrics: query latency and space consumption. Both are interesting if a TSDB needs to be chosen, as they give an answer for the question how fast different queries are executed and how space-efficient data is stored.

A single query is the smallest part in the interaction between a client and a database. It is measured how long a query is executed from start, which is the event of sending the query to the database, until a result is returned and received by the client. A query needs to be executed successfully and completely, which means that a correct result must be returned. A query that returns any kind of error or has not completed must be ignored because it cannot be used in the result for comparison[1]. Measuring only one

[1] It is unclear how the latency of a uncompleted or erroneous query compares to a completed query.

single query could be affected by random events created by network, by the operating system or by hardware (e. g., the linux kernel flushes pages or different network queue scheduling). It is necessary to repeatedly measure the same query type to average out such events.

With query latency as a metric, it is possible to divide the results into different query types so that it is possible to compare how fast each candidate ran each type [2]. Other benchmarks like YCSB and STAC-M3 Benchmark Suite also use query latency as a performance metric.

Query latency is measured for a query type in a set of queries with the following aggregating functions: Minimum, maximum, average, 95th percentage and 99th percentage values. All values are measured in microseconds. The values are defined as follows: Minimum latency is the latency of a query in a set of queries that required the smallest amount of time to complete. Maximum latency is the latency of a query in a set of queries that required the highest amount of time to complete. Average latency is the latency of all queries in a set of queries divided by the amount of queries in the set of queries. 95th percentage latency is the latency below or equal 95th percent of the latencies of all queries in a set of queries are. 99th percentage latency is the latency below or equal 99th percent of the latencies of all queries in a set of queries are. Amount of time to complete means the time elapsed between sending a query to the TSDB and receiving a successfully generated[3] result. For each definition, a set of queries consists of the same query type.

Space consumption is chosen as second metric as it is interesting if there is any difference in space efficiency between TSDBs when storing the same data. For measuring space consumption, the used space in a system is measured before and after a set of queries is run. The difference between both values gives the space consumption for the measured TSDB for this set of queries. Although the system is forced to write all data out of its buffers to hard drive before measurements are taken, it is not easy to precisely measure the space consumption due to two problems: Caching inside the TSDB and complexity of some TSDBs. Most TSDBs use some sort of internal caches, which are hard to measure and thus are not measured. Some TSDBs are complex systems that only transfer data after a given amount of time to another node for long-term storage. Exactly predicting this moment when this happens is not easy and therefore YCSB-TS ignores that. Knowing those inaccuracies, space consumption is considered less valuable in this thesis than query latency.

[2] Section 4.2 presents the query types used.

[3] See Section 4.1.

Space consumption is measured in bytes. The measurement is done with "df"[4]on all folders used by a TSDB. The results of all folders of a TSDB are summarized afterward.

After choosing those two metrics it is necessary to determine if they are a good performance metrics. There are six characteristics of a good performance metric: linearity, reliability, repeatability, easiness of measurement, consistency, and independence [Lil05]. In the following, the two metrics are evaluated whether they fulfill each characteristic.

A TSDB, that executes queries with half latency than another TSDB executes them, can run the double amount of queries in the same time as the other one. Query latency is therefore considered *linear*. Query latency is not *reliable*, as a result for a specific query type or set of queries cannot guarantee that this result holds for other query types or other types of circumstances. Query latency is *repeatable* because when running a query under the same circumstances twice (or more), the resulting latency stays the same[5]. Additionally, by running queries of the same type multiple times and therefore averaging out random events, query latency is more likely to be repeatable, as random events have less influence. How easy a metric is measurable depends on the point of view, but as query latency only requires two timestamps, it is considered as *easily* measurable. Results are not expected to be comparable between different systems or sets of workloads (queries), since query latency is expected to depend on the underlying hardware and the actual amount of data inside the TSDB[6], therefore query latency is not *consistent*. Because no manufacturer has influence on this metric, it is considered *independent*.

Space consumption is also *linear*, as a TSDB that stores the same data (without any loss) with half the space consumption than another one, stores it twice as efficient. Space consumption is not *reliable*, as it is not guaranteed that a result for a specific query or set of queries is true for all other types or other sets of queries. It is clear that space consumption is not precise in some circumstances (see above), but it is also considered *repeatable* as the same set of queries is expected to consume the same amount of space when executed again under the same circumstances. For this imprecise approach it is only necessary to execute commands for getting the actual used space and syncing buffers to hard disk before and after queries are executed. It is

[4]A Linux tool for measuring space consumption in a folder.

[5]Provided that the TSDB used behaves the same with unchanged circumstances.

[6]It is expected that a TSDB has a different performance with zero rows of data in comparison with e. g.,1,000,000,000 rows of data.

therefore considered to be *easily* measurable. It is not as easy when the measurement must be precise. As space consumption for the same data differs between filesystems, space consumption is not *consistent*. Using the same arguments as for query latency, it is considered as *independent*.

The given metrics do not satisfy all characteristics suggested by [Lil05]. As these suggestions are not requirements, most of the better known performance metrics (e. g., MIPS, SPEC) are not fulfilling all characteristics and still considered useful and are used for comparison. These two given metrics are considered as a good choice regardless because there are not many alternatives:

As an alternative, it is possible to not measure the latency and space consumption of an alterable set of queries, and measure it for an exactly defined workload instead. As a result, it is not possible to execute a workload suited for a specific scenario. It is also be possible to run a specific workload to determine how many queries a system can handle per second and to use that result to calculate a “unified” result, which would be free of any system-depending artifacts. Such a result would be comparable between different systems (hardware platforms). The issue is to measure how fast a system is and satisfying (most of) the six characteristics of a good performance metric (see [Lil05]). As it is not required for the presented benchmark, as its primary use is to determine a result for a specific scenario on one system, system-depending artefacts are acceptable as long as specific scenarios can be run. It is important that the system-depending artifacts equally affect all measured TSDBs of a comparison.

It is also possible to use a pseudometric (see [LL13]) that takes all measured values as well as some other values (that can also be not directly measurable) and creates a new resulting value. This way, for example, every result could be expressed in costs per query or costs per workload, which could be useful for determining the cheapest solution for a company. As a result, a pseudometric would also be more specific than considering all metrics individual, as the result clearly depends on how weights are put on different values. Since this thesis should give a base overview over different TSDBs, no pseudometric is used and the two metrics are considered individually and a feature comparison between the measured TSDB is done.

4.2 Scenarios

For comparison of the ten chosen TSDBs from Section 3.6, a performance evaluation is done[7]. In this section, the two chosen scenarios for this evaluation are presented. As mentioned in Chapter 2, when benchmarking TSDBs for finding a TSDB for a given use case (e. g., a set of sensor data with a set of repetitive queries) it should be done in a specific scenario. In this thesis exists no use case beyond the performance comparison, which leads to the problem that one or more scenarios are required for comparison, but all of them must be general or artificial as there is no given use case from which they can be derived.

The reason for the performance comparison is to get an overview over the performance of the compared TSDBs. Two simple scenarios have been chosen, which means that every function is measured on a reasonable sized data set, without fitting to any specific use case. Despite the missing use case, both scenarios have been designed with an energy related setup in mind that contains a large number of sensors that are grouped together in three categories (e. g., area, building and room). Furthermore, these two scenarios show the possibilities of TSDBBench, which is designed to compare different scenarios.

The comparison of the TSDBs is done using three different amounts of nodes in both scenarios: one, two and five nodes[8]. This means that each scenario run three times for each TSDB for examining any performance effects from distribution, which is also called *Elastic Speedup* [CST+10] or *Speedup* [DeW91].

For running any other queries than INS without an empty result, a set of data inside the used TSDB is required. For both scenarios, 1,000,000 rows of data are ingested into one metric. Each row consists of a timestamp between August 10, 2015 09:10:05:00 PM (UTC) and August 22, 2015 10:56:45:00 PM (UTC) (range is exactly 1,000,000,000 milliseconds). Each timestamp is chosen deterministic in ascending order from the starting point in time to the end point in time in steps of 1000 milliseconds, which results in a row of data every second. Each row of data has a floating-point[9] value that was chosen uniformly at random between 0 and 10,000.

Before the data is generated, a set of 1,000 predefined tag values are generated. A tag value is a alphanumeric string that has a fixed length of ten characters, consisting of

[7] See Chapter 5 for the results

[8] MySQL, MonetDB and PostgreSQL are only compared in one node setups, due to the restricted distribution and clusterability features, see Section 3.6.

[9] With dobule precision.

uniformly at random chosen alphanumeric characters. For each row of data, three tag values are chosen[10] uniformly at random from this set. See Table 4.1 for example data rows.

Timestamp	Value	TAG0	TAG1	TAG2
1439241005000[11]	6174.26568...	Wh64HSAlAU	dXannLxHhW	wbzkfQZt4Z
1439241006000	9382.65704...	NlIesvVgVw	8GcHATrQZS	nlnrV3cQQ5
1439241007000	4129.19188...	BMNGSOATxW	dWLbGdatQY	wTRxj0tNW9
...	...	...	...	...
1440241005000[12]	1595.94873...	TjKaU8X1yO	dWLbGdatQY	dgvOwrZ51n

Table 4.1: Example of the data that is used in both measurement scenarios.

The ingestion of the rows is done with INS queries, which give an overview how fast data can be ingested into a TSDB. Each row of data is inserted with one INS to simulate a group of sensors, with each of them sending data. This means that there were no block based insert methods used that pass a set of INS to the TSDB.

The set of data was chosen with energy sensors in mind. Most energy sensors, like voltage sensors, deliver floating point values with timestamps. Sensor values can be grouped together with tags and therefore every value has three tag values (e. g., considering a energy related setup: a sensor delivers a voltage measurement every second and is grouped into an area, a street and a building). 1,000,000 rows of data were chosen because the data requires to be loaded in ten TSDBs in two scenarios with querying afterward, which must be possible to load and execute in a reasonable time. In a realistic scenario, the amount of rows can be much higher (see Section 2.1 for an example).

SCAN, AVG, SUM, CNT, MIN, and MAX queries are the more complex queries found in most of the compared TSDBs, see Appendix A.6 for a comparison. The first four were chosen for YCSB-TS, MIN, and MAX queries are not used, only if one of the three used types of aggregating queries (AVG, SUM, and CNT queries) is not available.

One workload is used for each scenario. A workload consists of a *load phase* and a *run phase*. This initial group of INS queries is the *load phase* of both workloads, which

[10] One for each tag.

[11] Equal to August 10, 2015 09:10:05:00 PM (UTC).

[12] Equal to August 22, 2015 10:56:45:00 PM (UTC).

only consists of INS queries. The *run phase* consists of INS, READ, SCAN, SUM, CNT, and AVG queries, that are executed on the data created by the load phase. The two *run phases* of both workloads use two different groups of queries: group one consists of READ queries, group two consists of SCAN, AVG, SUM, and CNT queries.

The queries in group one emulate the simplest form of TSDB queries: Return every value at a specific timestamp. Timestamps are chosen uniformly at random between the start timestamp and the end timestamp. The READ queries do not use any filtering for specific tag values. This group is used to get an overview how fast values at a single timestamp can be read out of a TSDB. It is expected that some applications require to read data from a TSDB, which makes it a useable scenario[13]. READ queries are one way to read data from a TSDB, others are measured in the second group. As READ queries are the simplest form of queries for reading data, they are used to get an overview over the READ performance of the compared TSDBs. It is possible that some TSDBs perform worse in group one and better in group two, so it is necessary to measure all query types.

Group two uses SCAN, AVG, SUM, and CNT queries in combination with a period of time and a given set of tags. Each type should be used in equal proportions, not considering the initial INS queries. The idea behind these queries is, to emulate queries that return an average, a sum, the amount or all data from a set of sensors (e. g., an area) in a specific period of time. The period of time is chosen uniformly at random between 1,000 and 10,000 milliseconds. The starting point of time of a period is chosen uniformely at random between the start timestamp and the end timestamp subtracted by the chosen period of time. Without the subtraction, it would be possible that parts of the chosen period lie after the chosen end timestamp. Additionally, there are one to five tag values from each of the three predefined group of tags added, all chosen uniformly at random. This can result in an empty match, which means that there is no row in this period of time that has one tag of each group of tags. This is interesting because differences can exist in duration for such “non-matching” queries.

Both groups used 1000 queries each, which results in 1000 measured READ, 250 SCAN, 250 AVG, 250 SUM, and 250 CNT queries per TSDB. Additionally, 1,000,000 INS queries were used to fill each TSDB. The latency, which is the duration between sending a query and retrieving a result, was measured for each query. From these measurements it was possible to calculate average and 99th percentage values, which in return help to minimize effects from discordant values due to random effects. The

[13] Otherwise, when no application requires to read data from a TSDB, data would only be stored without further usage.

execution time of each phase of a workload was measured, which is used to calculate how many queries are executed per second in each phase. See Section 4.1 for further details on 99th percentage and average values. Additionally, the space occupancy before, between and after both phases of a workload are measured to recognize differences in storage efficiency. See Section 4.1 for further details on measurement of space consumption.

These two scenarios are chosen for giving an overview over the performance of ten TSDBs and are only a subset of the scenarios that are measurable with TSDBBench. As explained in Section 2.1, alterable workloads are required for the benchmark presented in this thesis. To fit other scenarios, it is necessary that the first phase of a workload can have different amounts of INS queries than used in these two scenarios, and the second phase can have any combination of READ, SCAN, AVG, SUM, and CNT queries.

Not only the amount of data ingested in the first phase must be alterable, but also the data itself. The period of time, the range of the values and the amount and length of tags and tag values requires to be changeable. It must also be possible to add a random amount of tag values to each value instead of a constant amount. In the two scenarios above, randomization is also done uniformly, but can also be done in other distributions, e. g., with emphasize on the newest value. It is important to have flexible workloads to fit many scenarios, but instead of describing every required changeable aspect of a workload here, a description of the workloads used in YCSB-TS is given in Section 4.6.

4.3 Requirements on the Benchmark

In this section, requirements for a TSDB benchmark are presented. First of all, as presented in Section 3.1, there have been 71 TSDBs found. Despite the fact that this number is growing, most of those TSDBs are still in development and therefore are changing frequently. The benchmark used in this thesis should have the possibility to support most of existing and future TSDBs. As a result, there must be the possibility to add a client for unsupported TSDBs (R1).

Regarding the fast development of some TSDBs, the results can change frequently. Changes in other software (e. g., operating system, software that is a dependency for a TSDB) are also expected to affect results. As a result, the benchmark must be repeatable (R2), which means that it is possible to execute a workload again with

another version (e. g., newer) of a TSDB[14] and compare the results between both versions.

For comparing different TSDBs, it is required that the results are comparable if created on the same environment. The environment consists of the EI together with the hardware it is running on and the used operating system and non-TSDB-specific software. It is also required that each TSDB executes the workload with equal conditions, which means that the benchmark used does not give advantages or disadvantages to any TSDB (R3). It is not expected that results between different environments (e. g., measuring TSDB A on system A and TSDB B on system B) are comparable.

Section 4.2 presents a motivation for alterable workloads. This forms requirement R4.

Most of the TSDBs can run on more than one node. Depending on the scenario, it can be necessary to compare results between different amount of nodes, which helps to find the necessary amount of nodes required to handle the scenario. Additionally, for sending queries at least one second node is required, otherwise (when running on the same node with a TSDB) the results could be affected by the generation of queries and data[15]. If one node is not enough for saturating the TSDB, there are more nodes for sending queries required. Both arguments lead to the requirement of additional nodes, either physical or by virtualization (R5).

To compare several TSDBs in a user friendly way, automated deployment and provisioning of the nodes (VMs) is required (R6). Otherwise, every node must be provisioned by hand which is not very user friendly, as it is expected to compare a workload between several TSDBs (e. g., consider ten TSDBs with 5 nodes each, which would mean that 50 nodes must be deployed and provisioned by hand). Automatic deployment means that VMs are automatically created with a specific operating system running on them and automatic provisioning installs all necessary software and configurations for a specific TSDB. This two-step process, deployment and provisioning, makes it easy to derive each VM from a base VM or image (deployment) and install specific software and configuration for each TSDB afterward (provisioning). A base image contains an installation of an operation system prepared for further installation. Appendix A.5 presents the hardware specifications used for TSDB VMs and generator VM. In the rest of this chapter, it is only referred to “base image”, without naming base VM explicitly.

[14] API changes of the TSDB not considered.

[15] Which both requires a part of the available CPU cycles and RAM.

As last requirement, it is necessary to execute a benchmark, gather, prepare and visualize the results automatically to complete the automation (R7). Executing a benchmark means that a workload and the measurement for each TSDB is started automatically and the benchmark waits until the workload is completed. Afterward, it receives the results, and prepares tables and graphs for visualization of the results. This step is repeated for each compared TSDB and an overall result with graphs is generated afterward. With this amount of automation, a benchmark run requires to be initiated and the user can look at the prepared and visualized results later, without provisioning, deploying VMs, or starting/stopping a benchmark manually for every TSDB.

4.4 Components

From the requirements from the previous section at least four required components can be derived:

1. An application for measurement and execution of workloads.
2. A EI that runs multiple VMs.
3. An application to deploy and provision VMs on an EI[16].
4. An application, that starts a measurement[17], waits for its termination, and collects and processes the results afterward.

4.4.1 An Application for Measurement and Execution of Workloads

The first component is the “real” or “inner” benchmark[18], as this is the part of TSDBBench, that executes workloads and measures query latencies. First of all, it is necessary to define what this application should be able to do:

- Support of an alterable workload with help of a workload definition[19] (required for R4), that can change at least the following:

[16] The EI from item 2.

[17] This means starting the application from item 1.

[18] See Section 4.4.4 for an explanation on the use of “benchmark”.

[19] E. g., in form of a file.

  - Amount of queries used in each phase.
  - Combination of queries used in phase two (e. g., 25% of all query types).
  - If tag values are predefined or randomly on-the-fly created.
  - Amount of tags per row.
  - Range of time used.
  - Range of values used.
  - Increasing or randomized timestamps.
  - Distribution of randomization values (e. g., uniform, emphasis on the newest inserts).
  - Amount of rows per timestamp.
- Executing a workload based on a workload definition (required for R2 and R3), which includes the following steps:
  - Creating tags according to the workload definition.
  - Setup connection to a TSDB.
  - Start measurement of duration.
  - Start executing phase one of the workload, which means generating and execution of INS queries to create the initial data set. The latency of each query must to be measured. Every query must be checked if it has failed.
  - After all INS queries are finished, note the duration of phase one.
  - Start executing phase two of the workload, which means generating and execution of INS, READ, SCAN, AVG, SUM, and CNT queries. The latency of each query must be measured. Every query must be checked if it has failed[20].
  - After all queries of phase two are finished, note the duration of phase two and stop measurement of duration.
- Calculate averages and other arithmetic values for both phases (e. g., average, min, max) (required for R7).

[20] Under some circumstances, like no existing matching rows, failed queries are accepted if an empty result is considered erroneous by the TSDB or its client.

- Creating and closing the connection to different TSDBs (required for R7).
- Saving the results (required for R7).[21]
- Usage of the same conditions for each TSDB (required for R3).
- Clients for new TSDBs must be addable (required for R1).

Creating a completely new benchmark that fulfills all necessary requirements leaves the scope of this thesis, therefore an existing benchmark was adopted. In Section 2.2, two benchmarks for TSDBs, and four that are related to TSDBs, are presented. Only one benchmark has been found that is open-source, not "outdated"[22], and supports alterable workloads: YCSB.

The basic idea behind YCSB is to measure latencies of basic[23] queries, among others. Support of alterable workloads, as well as extensible database bindings, is build-in. The source code of YCSB is open source, which made it possible to extend YCSB to YCSB-TS for satisfying our requirements (from the list above) in measuring TSDBs, as YCSB cannot naturally benchmark TSDBs. See Section 4.6 for further details.

4.4.2 An EI Platform that runs multiple VMs

To run multiple VMs, an EI is considered as requirement (required for R5, R6 and R7). Otherwise every node must be setup manually on physical hardware and tests with different cluster sizes would not be possible in an automated way. There exist several virtualization solutions out of which we can choose. The only requirement is an existing vagrant provider or the possibility to create one by using an API. The first idea is the use of VirtualBox [Ora15a], as Vagrant has support for it built-in. VirtualBox, however, is mainly a single-host solution, it cannot provide the resources required for setting up a cluster of VMs without being a possible performance bottleneck. When choosing an existing commercial EI, AWS [AWS15] or Microsoft Azure [Mic15a] are preferable, due to the low price per performance to Google Cloud [Goo15] and IBM SoftLayer[24] [Sof15] [Hug15]. In Section 4.4.3, Vagrant is presented as solution for deploying VMs in a EI. Both preferable EIs have vagrant providers available and can be

[21] E. g., in a file

[22] If a benchmark has not received any development since more than five years, it was considered "outdated".

[23] YCSB names them Create, Read, Update, Delete (CRUD) queries.

[24] Tested in two versions: "bare metal" and "virtual".

used in combination with Vagrant [Has15b; Mic15c]. For this thesis access to two other popular EIs, one propriety and one open-source, was available: vSphere [VMw15b] and OpenStack [Ope15b]. It was chosen to support both of these, to reach a wider audience of possible users, instead of other commercial EIs. Vagrant has providers for both of these [Gia15; VMw15a]. Integrating another EI is possible, it is only necessary to install the provider in vagrant and setup the configuration accordingly, as the code in TSDBBench is not EI specific.

4.4.3 An Application to deploy and provision VMs on an EI

For deploying and provisioning VMs (required for R6 and R7) three possible approaches have been found: Vagrant [Has15a], Docker [Doc15] or direct usage of the API of an EI. In the preferred programming language for this thesis, Python, bindings to all three of them are available [DeL15; Fuh15; VMw15a].

An API of a specific EI requires extra code for each supported EI. This increases the effort for integrating new EIs compared to more abstracted and loosely coupled approaches, like Vagrant and Docker, that separates EI specific code from TSDBBench. It is therefore beneficial to use Vagrant or Docker to decrease the effort for integration of new EIs by making use of already existing EI providers.

Since knowledge in using Vagrant was already existing, it has been decided to use it for deploying and provisioning VMs. That leads to advantages when choosing an EI: Vagrant has different providers and the possibility to add a new one for an unsupported EI. In this case, supplying a fitting Vagrant provider for the chosen EI is necessary.

For provisioning there are also solutions such as Puppet (see [pup15]) or Chef (see [Che15a]), which supports recurring configuration tasks beyond the capabilities of Vagrant. Both provisioning tools are used to continuously manage (e. g., change configuration, install software) a greater amount (e. g., 500,000, see [Che15a]) of nodes. For our architecture the capabilities of Vagrant are sufficient for running install tasks and configuration tasks after the start of a VM, without the requirement of continuous management[25]. That leads to a decision against Puppet or Chef, which would only increase the complexity of the architecture of TSDBBench without real benefit.

[25] No changes are required after setup.

4.4.4 An Application that starts a Measurement, waits for its Termination, and collects and processes the Results afterward

After the possibility to deploy, provision and benchmark TSDBs on a EI, the last component must be some sort of controller, that must manage the other parts to work together (required for R2 and R7). It also fetches and processes the results after finishing a benchmark run. The following tasks must be executed by the controller:

- Deploying and provisioning a set of VMs with Vagrant
- Running a chosen workload with YCSB-TS
- Fetching and processing the results after YCSB-TS has finished
- Repeating those steps for several TSDBs[26] (optional)
- Creating a combined result from all results[26] (optional)
- Generating plots and tables for single and (optional) combined results[27]
- Making sure that there was not error in any of those steps

A tool that supports these tasks is described in Section 4.5. It is called TSDBBench. To be consistent to Section 4.4.1, *benchmark* always refers to a "hole" benchmark application, e. g., TSDBBench if not stated otherwise. This can be confusing because TSDBBench, which is a benchmark, uses an adopted version of YCSB, which itself is a benchmark as well.

Existing solutions where chosen for some of these tasks[28], see Table 4.2. Existing solutions where chosen for some of these tasks. A mapping is presented in Table 4.2. Pyhton-vagrant [DeL15] is a Python module that provides a Vagrant API. Bokeh [Ana15] is a Python module for generating plots. Wkhtml-topdf [KT15] is a tool for creating PDF files from HTML files. Fabric [For15a] is a Python module (also includes a command-line tool) for running remote and local tasks.

[26] If more than one TSDB was chosen to be measured.

[27] These plots and tables are written to HTML and PDF files.

[28] More Python modules and tools were used for smaller tasks overall.

Name of the task	Python-vagrant	Bokeh	Wkhtml-topdf	Fabric
Deploying and provisioning a set of VMs with Vagrant	x			x
Running a chosen workload with YCSB-TS				x
Fetching and processing the results after YCSB-TS has finished				x
Repeating those steps for several TSDBs[26] (optional)				
Creating a combined result from all results[26] (optional)				
Generating plots and tables for single and (optional) combined results		x	x	
Making sure that there was not error in any of those steps				

Table 4.2: A assignment of used Python modules and tools to the tasks described in Section 4.4.4.

4.5 Architecture

The architecture of TSDBBench consists of two main parts, a node that is called controller[29] and an EI[30] that contains all VMs as seen in Fig. 4.1. The controller can create and destroy VMs on the EI and can interact with every VM on it. Interacting means that the controller can send data, see the status and run processes on every VM.

[29]Can also run on a VM inside the EI.

[30]In this thesis, vSphere and OpenStack are used as EIs, see Section 4.4.2.

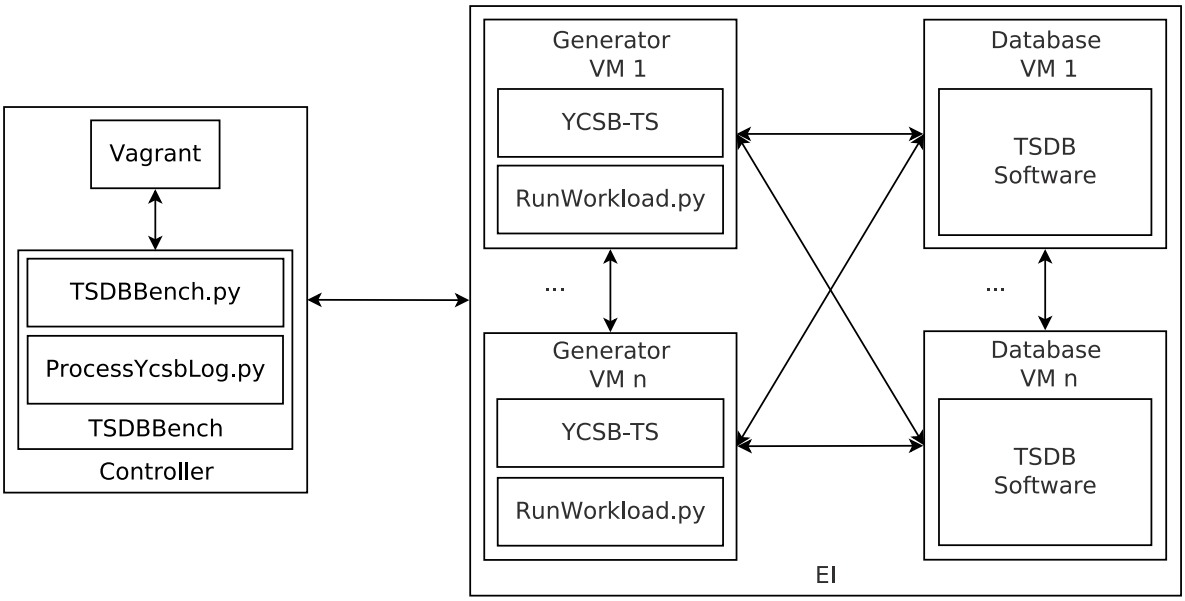

Figure 4.1: Overview of benchmark architecture

The controller consists of two main parts: TSDBBench and Vagrant. TSDBBench consists of three parts: TSDBBench.py, ProcessYcsbLog.py, and RunWorkload.py. It is used to create all VMs required for running a workload, starting a workload, waiting for it to finish and collect, as well as process, the results. For each supported TSDB there is a set of Vagrantfiles that describe the deployment and provision process of all required generator and TSDB VMs.

For every TSDB there must be VMs provisioned after deployment. There are three possible solutions:

1. Full installation including the installation of the operating system.
2. Deriving from a base image that contains the operating system and only installing TSDB-specific software afterward.
3. Creating an image for every TSDB and only derive VMs from the base images.

Option 1 consumes the most time during the test, as the operating system must be installed for every node for every TSDB. Option 2 consumes less time than Option 1, but it is necessary to distribute a base image or instructions how to create it. Option 3 is the fastest solutions, but requires the most space, as for every node type (e. g., master and slave) for every TSDB an image must be created. These images or instructions how to create them must also be distributed. Option 2 was used for TSDBBench as it was chosen to be the best tradeoff between the time necessary to setup a measurement and

the effort to distribute and use TSDBBench. Option 1 was not chosen because the time to install an operating system (at least 15 minutes) is much more than deriving a new VM from an image (approx. 2 minutes). Option 3 was not chosen because distributing approx. 15 images was considered using too much space (e. g., one gigabyte per image multiplied by 15) and bandwidth (when downloading) compared to Option 2. Even when only distributing instructions for generation in Option 3, the space and time required to create the images was considered too high. For comparison: Creating an image takes about 30–45 minutes, deriving a VM from an image approx. 2 minutes and installing TSDB specific software usually takes less than five minutes[31] and is done for each node in parallel. If Option 3 was used, then a new user must wait 7.5 hours until at least ten images are generated and a first measurement can be done. It would save five minutes per measurement, which seems unfavorable, considering long measurement durations (e. g., over an hour), future integration of more TSDBs and the effort to update the operating system (recreation of each image instead of one image).

For benchmarking a TSDB, two VMs types are required within the EI: generator VMs and TSDB VMs. Both types can be instantiated multiple times in TSDBBench, but for the scenarios described in Section 4.2, one generator VM and multiple TSDB VMs are required. Multiple generator VMs are usable for further measurements if the limits of one generator VM are exceeded (e. g., when saturation of the measured TSDB is required but not possible with one generator VM).

A generator VM contains an operating system, derived from a base image, and two components: RunWorkload.py and YCSB-TS. RunWorkload.py does further provisioning and starts YCSB-TS. Further provisioning is required because this is the first situation where all IP-addresses are known and can be given to each TSDB VM for further initialization and setup[32]. Since YCSB-TS cannot measure space consumption, RunWorkload.py measures it for every TSDB in its specific TSDB folder before and after the workload. It also measures the space consumption between the two phases of YCSB-TS. See Appendix A.5 for the hardware specifications used for generator VM.

The latter one can run a given workload against a given TSDB whilst measuring different values like query throughput, latency and duration (see Section 4.6 for a more detailed explanation of YCSB-TS).

[31] Depending on the TSDB and its required software.

[32] E. g., each TSDB VM requires at least one IP-addresses of another TSDB VM to create the connections between them.

TSDB VMs consist of everything that is required for that specific TSDB to run properly in the given scenario, which means an operation system, that is derived from a base image, and additional TSDB specific software. See Section 4.5 for further details on the creation process of a TSDB VM and Appendix A.5 for the hardware specifications used for TSDB VMs. After deployment and provisioning are finished, they are ready to receive queries.

The steps from the initial request to the final measuring results rendered in diagrams is presented using BPMN in Fig. 4.2. Each component is rendered as pool. The steps are rendered as tasks and the communication between the scripts is rendered as BPMN messages. Data exchange is done via files on the HDD and SCP commands.

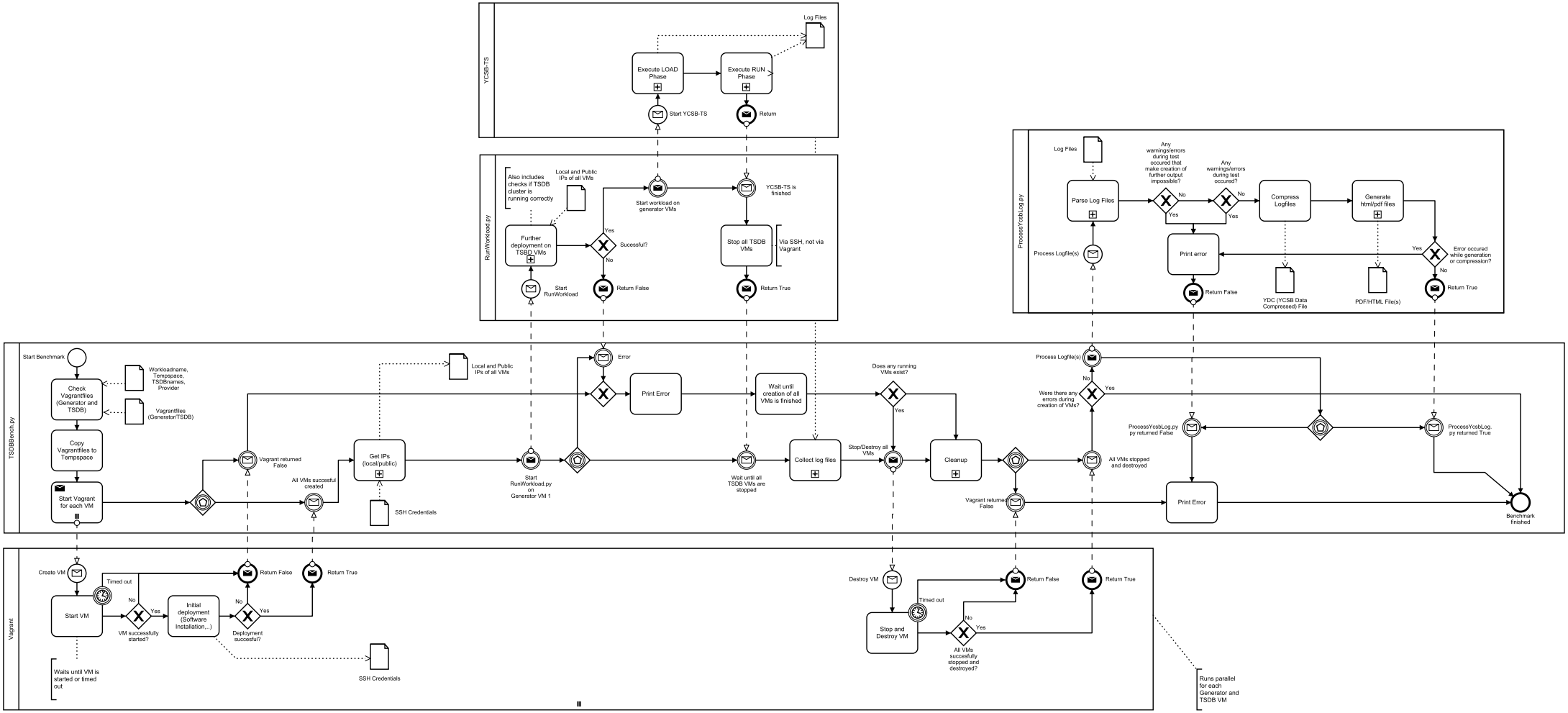

Figure 4.2: Overview of benchmark process as BPMN [Obj11] diagram. The exchange of data is not illustrated by data input and data output objects because bpmn.io [CZ15] does not support these objects.

The process of benchmarking a workload on a TSDB is executed in following steps (see Fig. 4.2):

1. TSDBBench.py is started with at least one given TSDB and a workload as parameters
2. Deploy and provision generator VM(s)
3. For each TSDB:
    1. Deploy and provision TSDB VM(s)
    2. Execute and measure workload
    3. Collect results after workload has finished
    4. After termination destroy TSDB VM(s)
4. Destroy generator VM(s)
5. Generate HTML and PDF files for the result of each TSDB
6. Generate HTML and PDF files for a combined result (optional)

Deploying and provisioning TSDB VM(s) is done as follows:

1. For every Vagrantfile for the given database do:
    1. Derive a VM from a given base image.
    2. Run every given instruction in the Vagrantfile for provisioning it (e. g., installing software, configuring, ..)
2. Start RunWorkload.py on the first generator VM which executes further provisioning[33]

Executing and measuring a workload for one TSDB is done as follows:

1. TSDBBench.py starts RunWorkload.py on the first generator VM and waits for the first TSDB VM to stop
2. RunWorkload.py measures space consumption on each TSDB VM
3. RunWorkload.py starts YCSB-TS in load phase with the given workload

[33] This step is required because this is the first situation where the IP-adresses of all TSDB VMs are known.

4. YCSB-TS creates, exutes, and measures (query latency) the queries of the load phase
5. RunWorkload.py measures space consumption on each TSDB VM
6. RunWorkload.py starts YCSB-TS in run phase with the given workload
7. YCSB-TS creates, exutes, and measures (query latency) the queries of the run phase
8. RunWorkload.py measures space consumption on each TSDB VM
9. RunWorkload.py stop the TSDB VM(s)
10. TSDBBench.py receives the results of the measurement

Generation of HTML and PDF files is done as follows:

1. ProcessYcsbLog.py is started by TSDBBench.py with the results from all measured TSDB
2. ProcessYcsbLog.py performs the following steps for each result:
    1. Parse collected log file to create a single compressed ydc file [34]
    2. Generate HTML file that contains results and plots
    3. Generate PDF file HTML file
3. Create HTML and PDF files with combined results (optional if more than one TSDB was measured)

4.6 Benchmark

YCSB [CST+10] 0.4.0 was extended to YCSB-TS to run and measure workloads on TSDBs. This section describes the necessary changes to YCSB.

YCSB uses CRUD queries[35] for benchmarking NoSQL databases. Those queries are not fitting for TSDBs due to the row schema and the missing support for delete and update queries in most TSDBs. Of the ten compared TSDBs six support delete queries and five

[34] A type of file that contains the same information as the log file but has different syntax and is compressed.

[35] INS is used for “Create”.

support update queries (see Appendix A.6). Two of them are RDBMS and therefore not counted as *real* TSDBs in this summation. As a result, *real* TSDBs support these query types, and therefore these two query types are not usable for comparing TSDBs. To support the typical queries of TSDBS (see Section 4.2), SCAN, AVG, SUM, and CNT queries have been added to YCSB-TS.

The original row schema in YCSB uses a primary key with additional fields, all of them alphanumeric[36]. As this does not fit for TSDBs due to the missing timestamp and floating point value, the schema was extended to consist of the following: a timestamp, a floating point value and additional alphanumeric tags. To save the timestamp, a floating point value with double precision (long), that describes the elapsed milliseconds since January 1, 1970 00:00:00:00 AM (UTC), was internally used. Not all TSDB clients directly use this format, some need conversions to a specific time format, which were performed accordingly.

Timestamps can be generated in ascending order or uniformly at random in a given range[37]. Values are generated uniformly at random in a given range. The amount of tag values can be constant or uniformly at random in a given range. The tag values are generated with constant or random[38] length, consisting of uniformly at random chosen alphanumeric characters. A set of tag values can be generated for each tag before the run phase. Tag values are then chosen uniformly at random from that set of tag values each time a tag value is required. This helps when tag values are used for querying, otherwise it is hardly possible to hit an existing row of data because it is unlikely that a uniformly at random generated tag value exists. To read predefined values and tag values, a threaded reader with variable buffer size for CSV files was added. See Section 4.6 for further details on workload options.

None of the provided clients in YCSB were suitable for any TSDB. Clients for Blueflood, Druid, InfluxDB, KairosDB, NewTS, OpenTSDB and Rhombus were added. The JDBC client in YCSB was adopted for testing SQL based databases[39] that were used as TSDBs. The non-suitable clients were dropped.

[36]The workload provided by YCSB, CoreWorkload, uses ten fields with 100 bytes length each as a default.

[37]It is possible to create an uniformely at random amount of rows for each timestamp in a given range, or pick a timestamp uniformely at random in a given range for each new row of data (see Section 4.6).

[38]With different distributions: uniform, zipfian, and histogram.

[39]In this thesis, JDBC was used for PostgreSQL, MySQL Community Server and MonetDB.

To determine the best possible result without any interaction with a TSDB, a *test* mode for every client was implemented. In this *test* mode, the workload is executed the same way as without *test* mode, but every TSDB interaction (e. g., connection, querying) is not performed. This *test* mode helps to see what negative performance effects are introduced due to use of YCSB-TS and what would be the lowest latency possible. It also helps to determine the impact on each TSDB because every TSDB should be provided with conditions as equal as possible. Exact equal conditions are not possible due to the different client used for each TSDB and their internal handling of the queries.

The provided default workload in YCSB (named CoreWorkload) required changes to fit the new row schema and to support different TSDB scenarios. Table 4.3 describes old and new options for query generation in Coreworkload. For the options that were dropped from YCSB whilst extending it to YCSB-TS, see Table 4.4.

Table 4.3: Options in the Coreworkload of YCSB-TS and their default values.

Name of the Option	Description	Default Values
insertstart	The first timestamp in the time range used for queries.	1439241005000
insertend	The last timestamp in the time range used for queries, must be before insertstart.	1439242005000
metric	The name of the metric used in every TSDB.	usermetric
tagcount	The amount of tags used for every row.	3
tagvaluelengthdistribution	The distribution used to choose the length of each tag value. Possible values are constant[40], uniform[41], zipfian[42] and histogram[43].	constant
tagvaluelength	The maximum or constant tag value length.	10
tagprefix	Prefix of the tag names. A increasing number, beginning from zero, is added to this prefix.	TAG
randomtagamount	If set to true, between zero to tagamount tag values are added to each row. Otherwise, a constant amount of tag values, given in tagamount, is used.	false
predefinedtagvalues	If set to true a set of predefined tag values for each tag is generated and tag values from these sets are picked. Otherwise, every tag value will be generated randomly.	true
predefinedtagvalueamount	This number describes how many predefined tag values will be generated for each tag.	100

[40]Same length for every tag value.

[41]Uniformely at random chosen length between 1 and tagvaluelength.

[42]Random with preference for short lengths between 1 and tagvaluelength.

[43]Read distribution from file defined in taglengthhistogram.

Added options in the Coreworkload of YCSB-TS and their default values. (continued)

Name of the Option	Description	Default Values
predefinedtag-storagefile	The name of the file in witch the predefined tag values are stored for passing them from load phase to run phase.	predefinedtags.txt
predefinedtag-storeused	Only store used tag value combinations[44], which means that a query can only miss by timestamp or range.	true
valuemax	Highest possible value.	10000
valuemin	Lowest possible value.	0
taglength-histogram	Name of the histogram file used when tagvaluelengthdistribution is set to histogram.	hist.txt
avgproportion	Proportion of AVG in run phase.	0
countproportion	Proportion of CNT in run phase.	0
sumproportion	Proportion of SUM in run phase.	0
timevalue	The granularity or bucket size passed to aggregating functions (e. g., aggegrate over each day). 0 is interpreted as the hugest possible bucket (e. g., return one aggregated value over the given range of time).	0
timeunit	Unit for timevalue.	ms
maxscanlength	Maximum time span of a SCAN in milliseconds.	1000
minscanlength	Minimum time span of a SCAN in milliseconds.	1000
scanlengthdistribution	Distribution with which the scan length is chosen. Possible are uniform[45] and zipfian[46].	uniform
maxrecordsperts	This value describes how many rows per timestamp are allowed.	10

[44] A tag value combination consists one chosen tag value for each tag.

[45] Uniformely at random between minscanlength and maxscanlength.

[46] Random between minscanlength and maxscanlength with preference on shorter values.

Added options in the Coreworkload of YCSB-TS and their default values. (continued)

Name of the Option	Description	Default Values
randomamount-perts	Desribes how multiple rows per timestamp are distributed. Can be constant[47], random[48] or tsrandom[49].	constant
randomfloor-filling	When randomamountperts is set to tsrandom, make sure there is at least one row per timestamp.	false
querymaxtag-values	This value describes how many tag values at maximum should be given for each tag when using filtering for tags. The amount is chosen uniformely at random between one and this value.	10
notagsforread	If true: Do not use filterfing for tags when performing READ.	true
readfromcsv	If true: Do not generate data, read it from CSV file defined in csvfile.	false
csvfile	Name of the CSV file when readfromcsv is set to true.	values.csv
csvbuffersize	This value describes the size of the buffer used to read a CSV file.	100000
description	Description of the workload.	Testworkload
timeresolution	The size of one time step in milliseconds between two rows (e. g., a value of 1000 means that each value is one second apart from each other.).	1000

[47] Constant amount of rows per timestamp.

[48] Between 1 and maxrecordsperts rows per timestamp, the amount(!) per timestamp is chosen uniformely at random.

[49] Chose timestamp for each row randomly, maxrecordsperts is ignored, lowest amount per timestamp is zero rows.

Name of the option	Reason
fieldcount	Replaced by tagcount.
fieldlength	Replaced by tagvaluelength.
readallfields	Not used for TSDBs.
writeallfields	Not used for TSDBs.
fieldlengthdistribution	Replaced by tagvaluelengthdistribution.
updateproportion	Not used forTSDBs, UPDATE is not implemented.
readmodifywriteproportion	Not used forTSDBs.
insertorder	Not used for TSDBs.
hotspotdatafraction	Not used for TSDBs.
hotspotopnfraction	Not used for TSDBs.
table	Replaced by metric.
columnfamily	Not used for TSDBs.

Table 4.4: Deleted options in the Coreworkload of YCSB whilst extending it to YCSB-TS.

4.7 Peculiarities

Some of the compared TSDBs needed special treatments or setups to be usable in this performance comparison. This section presents these peculiarities for each TSDB (if existing) and the solutions used. Each of the TSDBs is presented in Chapter 3.

4.7.1 Group 1: TSDBs with a Requirement on NoSQL DBMS

Blueflood does not support tags, SUM queries and freely chosen granularities. See Section 3.2 for more details. As a result, all queries are without tags, SUM is replaced by MIN and the given granularity steps are used. As a result, the results of Blueflood must be compared with caution, as it has advantages in the second scenario. If the queries of the second scenario (SCAN, CNT, AVG, and SUM) do not need to search for tag combinations and need to filter for timestamp only, a faster result is possible. It is also unknown how easier or harder MIN is executable compared to SUM. The queries of the second scenario use granularities, which they set to one value over the chosen timestamp for a query. Blueflood supports the granularity that was used when data was ingested or 5, 60, 240, or 1440 minutes. As a solution 1440 minutes was used to use as coarse granularity as possible.

KairosDB requires a time range of 1 millisecond for READ, which was used. It is unclear how this affects comparability. KairosDB is the only TSDB that is compared with two backend: Cassandra and H2. While KairosDB with Cassandra is compared with all cluster setups, KairosDB with H2 is only used in single node setups in Chapter 5.

NewTS has several limitations when filtering for tag values is used. It cannot filter for tag values and timestamp or time range together, which means that it is impossible to filter for values in a time range with one or more tag values. When filtering for tag values, there is no possibility to use Boolean logic, instead it is possible to pass a group of tag values to NewTS only. It can be chosen if all tag values must be fitting or only one of them. To chose a correlation which of the chosen tag values must fit on which tag is not possible. Filtering for tag values cannot be used in combination with aggregating functions. As a result, filtering for tag values is not used. NewTs does not support CNT and SUM, as replacements are MIN and MAX used. Granularity must be at least two milliseconds. It cannot be chosen to return one aggregated value over the whole time range (for receiving one aggregated value). As a result, granularity was chosen to be one year. The argumentation that comparability is not known is the same as for Blueflood.

OpenTSDB requires a time range of 1 millisecond for READ, which is used. MIN is used as aggregator for READ, due to OpenTSDB requiring an aggreating function for every query[50]. It is unclear how both affect comparability.

Rhombus does not support AVG and SUM. CNT is used as replacement. Rhombus does not support granularity. It is unclear if Rhombus benefits from not having to group values in a given granularity and only using CNT as aggregating function.

4.7.2 Group 2: TSDBs with no Requirement on any DBMS

Druid supports several endpoints for querying and ingestion. Tranquility (Finagle API) is used for ingestion in YCSB-TS. In the scenarios used for comparison, a specific time range is used for all TSDBs. It was not possible to use a time range in the past when using Druid, even when the window period is set accordingly[51]. The time range is shifted to the actual time as solution, while keeping the range at the same size. MySQL

[50] Other TSDBs in this comparison do not use an aggregator for READ.

[51] The window period is used by Druid to decide how far back values are accepted. Values outside that period are automatically dropped instead of ingested.

Community Server was used as metadata storage for Druid in the comparisons in Chapter 5.

InfluxDB has no peculiarities in the scenarios used. For ingesting data the “Line Protocol” is used. See Section 3.3 for more details.

MonetDB shares the same peculiarities as MySQL Community Server and PosteSQL. These are described in the next subsection.

4.7.3 Group 3: RDBMS

As *MySQL Community Server*, *PostgreSQL* and *MonetDB* share the same peculiarities, due to their SQL based languages and table structures, the following applies for all of them: For creating tables and queries similar the other TSDBs, a relational schema is used, see Appendix A.2 for the tables and queries. Due to their restrictions in clusterability and distribution features (see Section 3.6), all three are compared in single node setups on Chapter 5 only.

The benchmark presented in this chapter is used to create the performance comparison between the ten compared TSDBs in the two presented scenarios from Section 4.2. The results of the performance comparison are presented in the following chapter.

5 Measurements and Results

This chapter presents the result from the benchmarks of the two scenarios described in Section 4.2. It is structured as follows: Section 5.1 presents the results from Scenario 1 (1,000 READ queries). Section 5.2 presents the results from Scenario 2 (250 SCAN, AVG, SUM, and CNT queries).

Both sections are divided in four different setups, each presented in its own subsection. In the first subsection, each TSDB has one node[1] only. The three subsequent subsections present the usage of a cluster of five nodes for each TSDB, with changing Replication Factor (RF) (1, 2, and 5). Both sections present query latency results. Section 5.1 presents aditional space consumption results. Query latency results of INS queries and Space consumption results are omitted in Section 5.2 because both scenarios use the same data[2] and there are no differences in the results identifiable. Section 4.1 presents details on the metrics used. The lowest possible resolution for measuring query latency with YCSB-TS is 1 μs. For 99th percentile query latency[3] the lowest possible resolution is ms, see [Wak15]. Due to logarithmic scaling of the plots, bars can be missing, which means that the value is below one ms for 99th percentile values and one μs for average values[4]. 99th percentile values are named "99thPercentile" and average values are named "AVG". All given values in the text are rounded half way. Query latencies are shown in ms.

MonetDB, MySQL Community Server, PostgreSQL and KairosDB with H2[5] are only benchmarked in a one node setup, due to their limited distribution and clusterability features. Section 3.6 presents further details.

[1] "Node" is equally used with "VM" in this chapter.

[2] Values and tag values are randomly generated, which means that they are not exactly the same, see Section 2.3 for further details.

[3] And for 95 percent query latency, which is integrated into YCSB-TS, but not used for this thesis. Both values are reported in μs by YCSB-TS, but are calculated in ms internally.

[4] The latter one does not occur on any of the presented results.

[5] See Section 3.2.

In both scenarios, 99th percentile and average query latencies, denoted in ms, are used for query latency, and mebibytes (MiB)[6] are used for measuring space consumption. In all plots, each name of a TSDB has a suffix "_clN", where N is a number, and another suffix "_rfM", where M is a number. "_clN" stands for cluster size N, e. g., "_cl1" means a cluster consisting of one VM, "_cl5" means a cluster consisting of five VMs. "_rfN" stands for RF N, e. g., "_rf1" means a RF of one, "_rf5" means a RF of five.

5.1 Scenario 1: 1,000 READ Queries

5.1.1 One Node Cluster with a RF of One

5.1.1.1 Query Latency for INS Queries

Figure 5.1 presents the results for INS queries in a one node cluster with a RF of one. There are no 99th percentile values shown for Blueflood and OpenTSDB because both values lie below one ms.

The result of this measurement is that in a one node cluster with a RF of one, OpenTSDB has the best query latency for INS queries measured with a 99th percentile value below 1 ms and an average value of 0.210 ms. Blueflood follows with an average value of 0.532 ms. The next three following TSDBs are Druid, Rhombus, and PostgreSQL. Druid has a higher 99th percentile value than its average value, which means that is must have had a small amount (below one percent) of queries with a lower latency than the remaining queries. The worst result for INS queries have InfluxDB, preceded by MonetDB, and KairosDB with H2.

5.1.1.2 Query Latency for READ Queries

Figure 5.2 presents the results for READ queries in a one node cluster with a RF of one. The four TSDBs with the fastest query latencies are Blueflood, NewTS, Rhombus, and MonetDB. All four of them are very close to each other, with Rhombus having the fastest average value of 1.7 ms. MonetDB has with 2.3 ms the best 99th percentile value. The worst result for READ queries has KairosDB with H2 and with Cassandra, both are over an order of magnitude worse than the remaining TSDBs. This is an

[6] See [Int15] for further information.

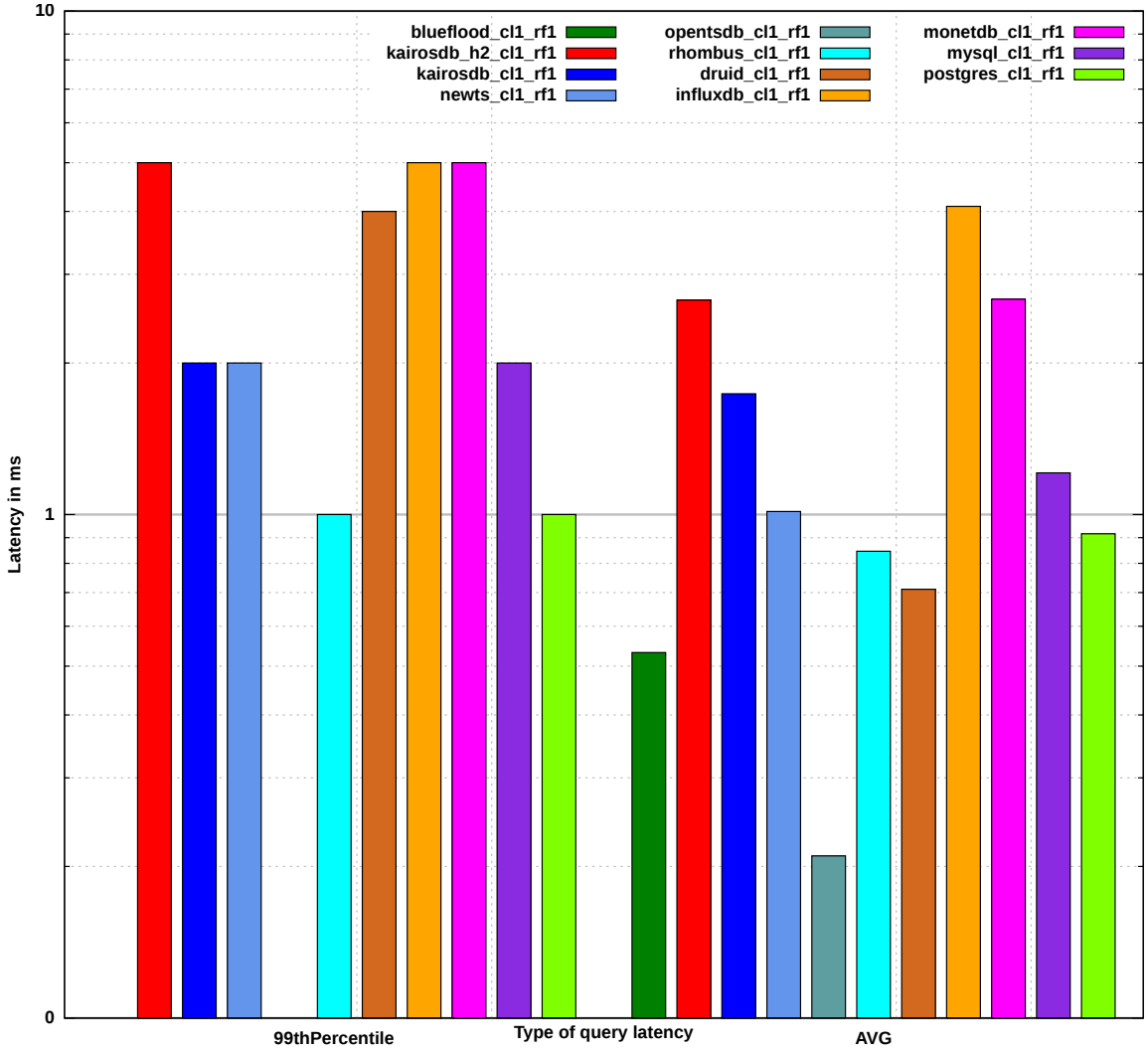

Figure 5.1: Query latency results for INS queries in a one node cluster with a RF of one. There are no 99th percentile values shown for Blueflood and OpenTSDB because both values lie below one ms.

interesting result for KairosDB with Cassandra because other TSDBs that use Cassandra have better results.

5.1.1.3 Space Consumption

Figure 5.3 presents the difference in space consumption in a one node cluster with a RF of one when 1,000,000 INS queries are executed. The space consumption is measured before and after the execution. Druid uses with 63.4 MiB the fewest amount of all

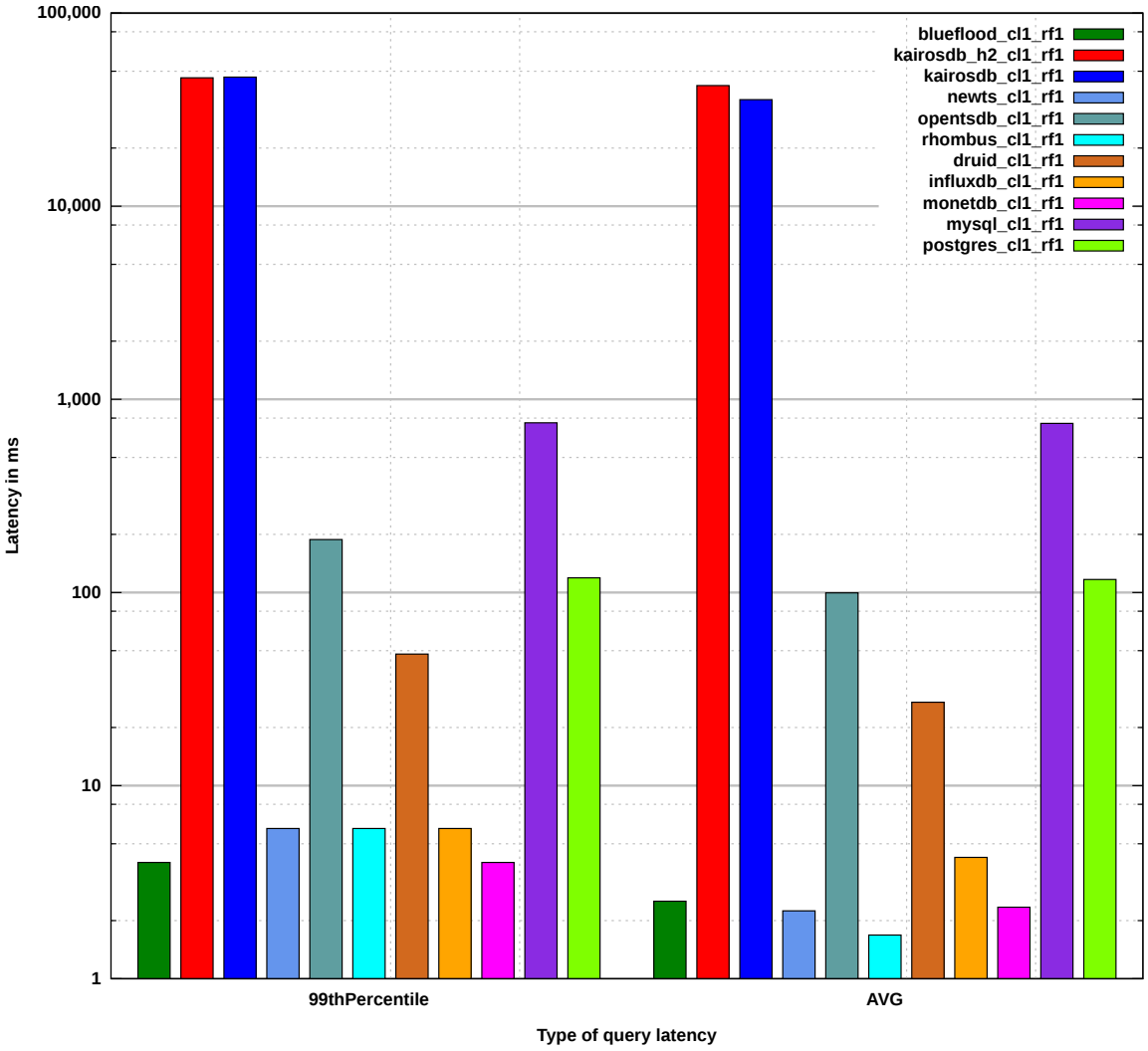

Figure 5.2: Query latency results for READ queries in a one node cluster with a RF of one.

compared TSDBs, followd by OpenTSDB with 74 MiB, and InfluxDB with 79.9 MiB. Rhombus uses with 2503.7 MiB the most space.

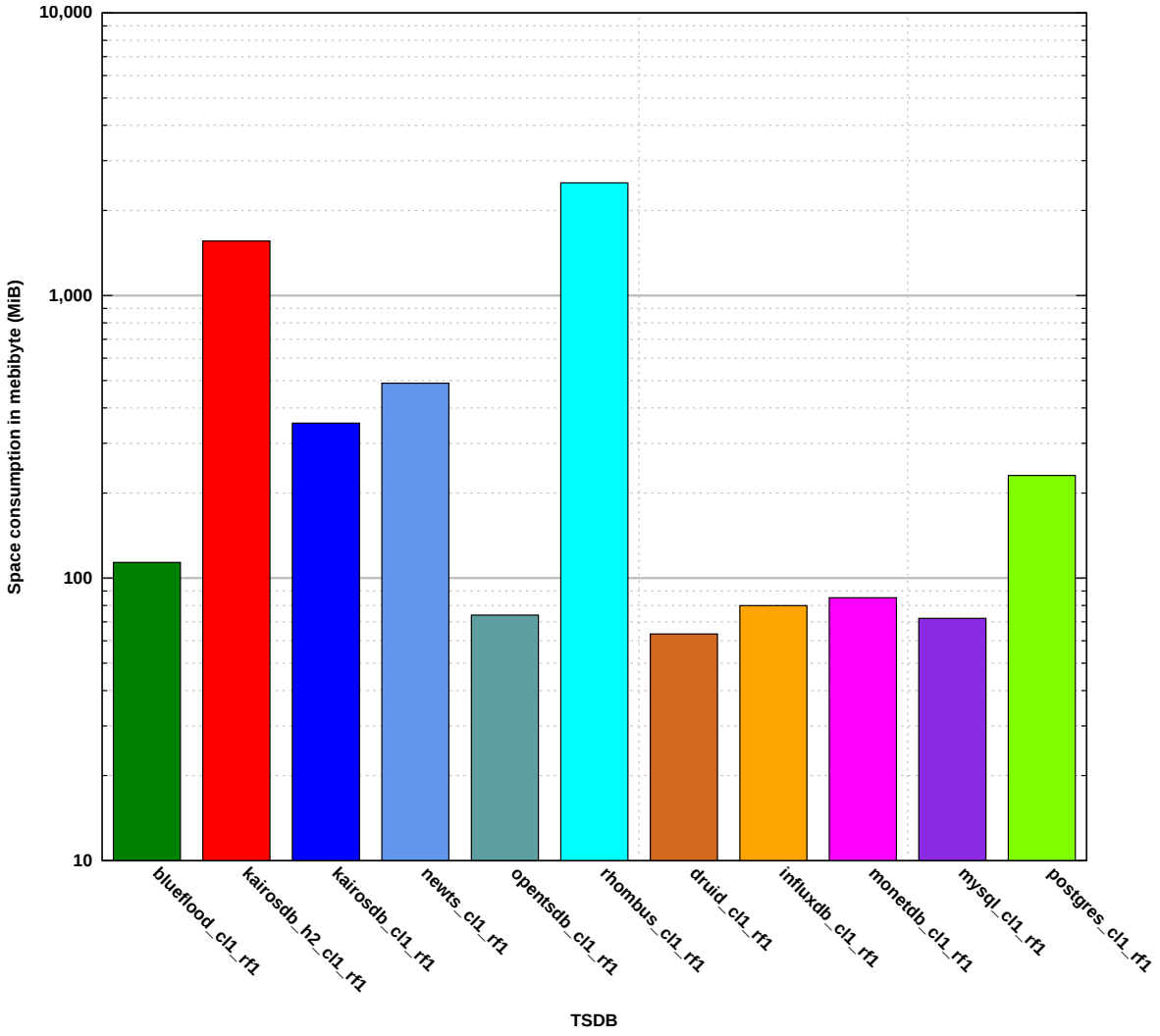

Figure 5.3: Space consumption results for a one node cluster with a RF of one.

5.1.2 Five Node Cluster with a RF of One

5.1.2.1 Query Latency for INS Queries

Figure 5.4 presents the results for INS queries in a five node cluster with a RF of one. There is no 99th percentile value shown for OpenTSDB because the value lies below one ms. One of the results from this measurement is that OpenTSDB has the lowest 99th percentile value and the lowest average value of 0.212 ms. The second fastest TSDB is not clear, Druid has the second best average value with 0.61 ms, and Rhombus has the second best 99th percentile value with one ms. Blueflood has the second smallest average query latency in Section 5.1.1.1 with 0.532 ms, which increased to 0.935 ms in this measurement. As a result, Blueflood is placed in the midfield. This is interesting because other TSDBs show a slightly decreased average query latency with five nodes compared to one node, e. g., OpenTSDB which increaed from 0.210 to 0.212 ms. The worst result achieved InfluxDB. Compared to the results in Section 5.1.1.1, none of the TSDBs could benefit significantly from having four more nodes.

5.1.2.2 Query Latency for READ Queries

Figure 5.5 presents the results for READ queries in a five node cluster with a RF of one. Rhombus has the best 99th percentile value with 2 ms and the best average value with 1.97 ms. The next best three TSDBs are NewTS, Blueflood, and InfluxDB, none of them is the best in both, the 99th percentile and the average, values. The worst result for READ queries has KairosDB (with Cassandra), they are around two orders of magnitude worse than the remaining TSDBs. Compared to the results in Section 5.1.1.2, none of the TSDBs could benefit significantly from having four more nodes.

5.1.2.3 Space Consumption

Figure 5.6 presents the difference in space consumption in a five node cluster with a RF of one when 1,000,000 INS queries are executed. The space consumption is measured before and after the execution. An increase in space consumption is not expected compared to Section 5.1.1.3, as with a RF of one still only one value is stored. A slight increase might occur due to overhead.

Druid uses with 63.4 MiB the fewest amount of all compared TSDBs, followd by OpenTSDB with 74.6 MiB, and InfluxDB with 80.7 MiB. Rhombus uses with

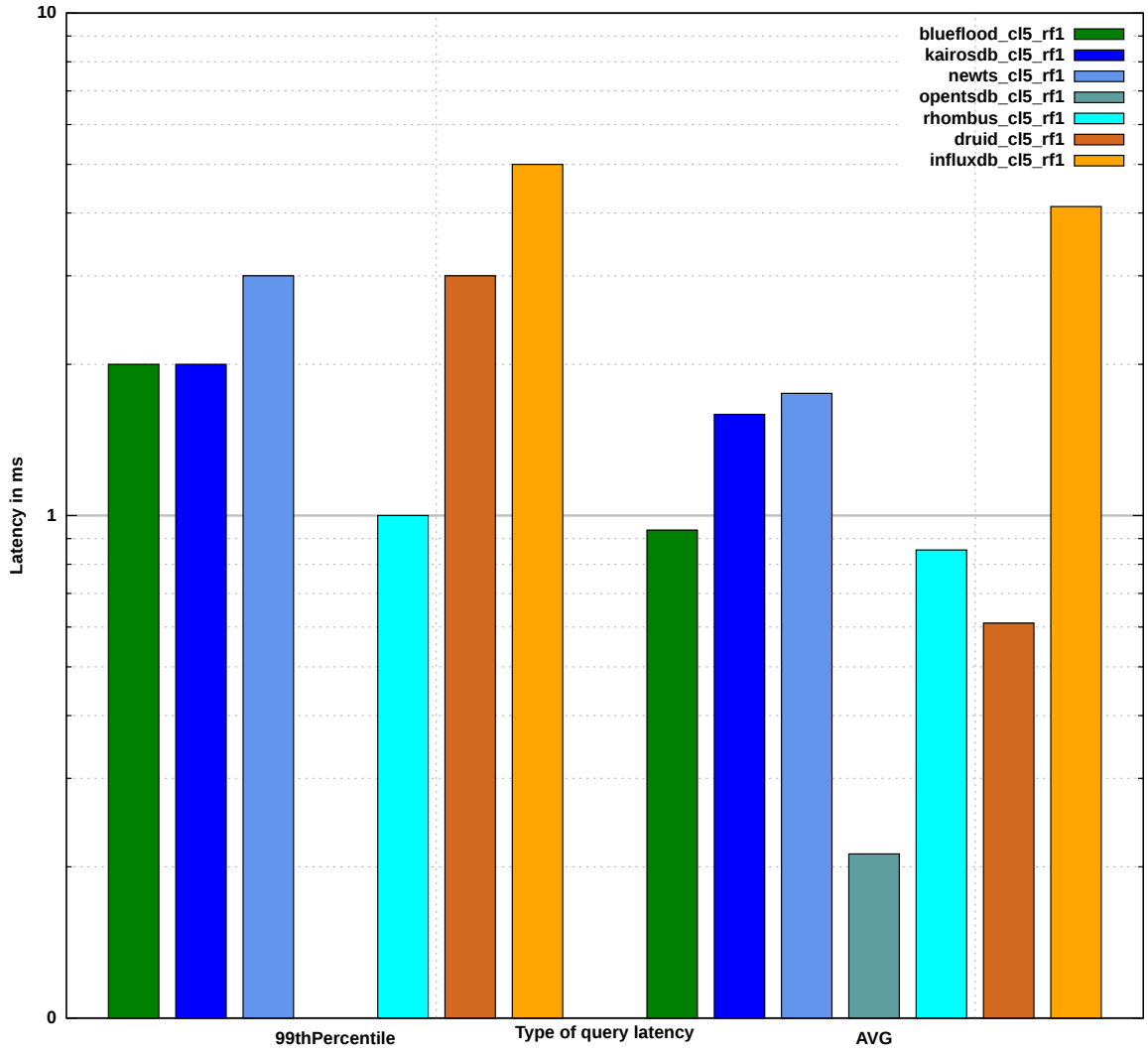

Figure 5.4: Query latency results for INS queries in a five node cluster with a RF of one. There is no 99th percentile value shown for OpenTSDB because the value lies below one ms.

2,287.6 MiB the most space. Compared to Section 5.1.1.3, the space consumption only increased slightly for most TSDBs. Rhombus, for example, uses slightly less space, but still the most.

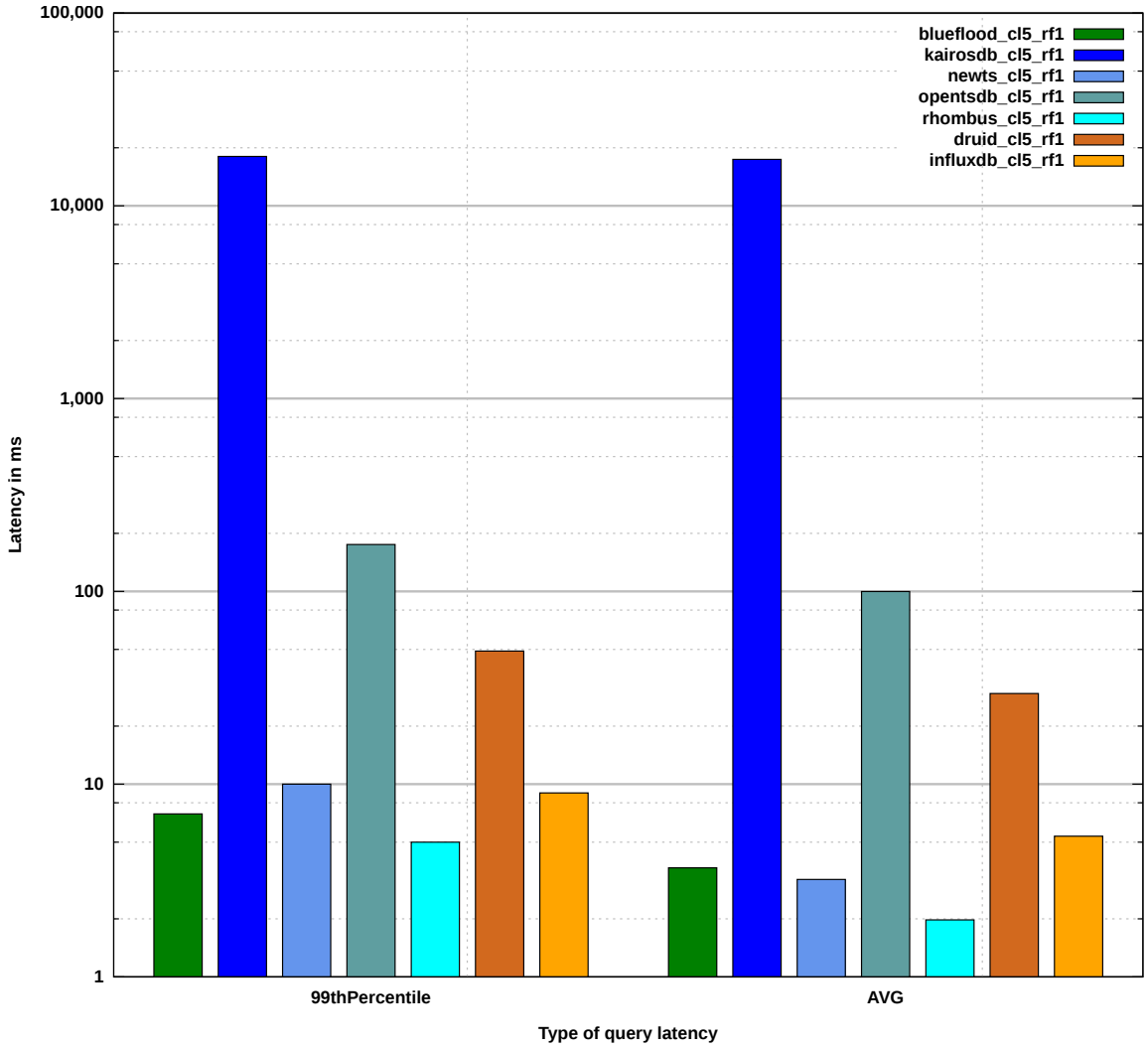

Figure 5.5: Query latency results for READ queries in a five node cluster with a RF of one.

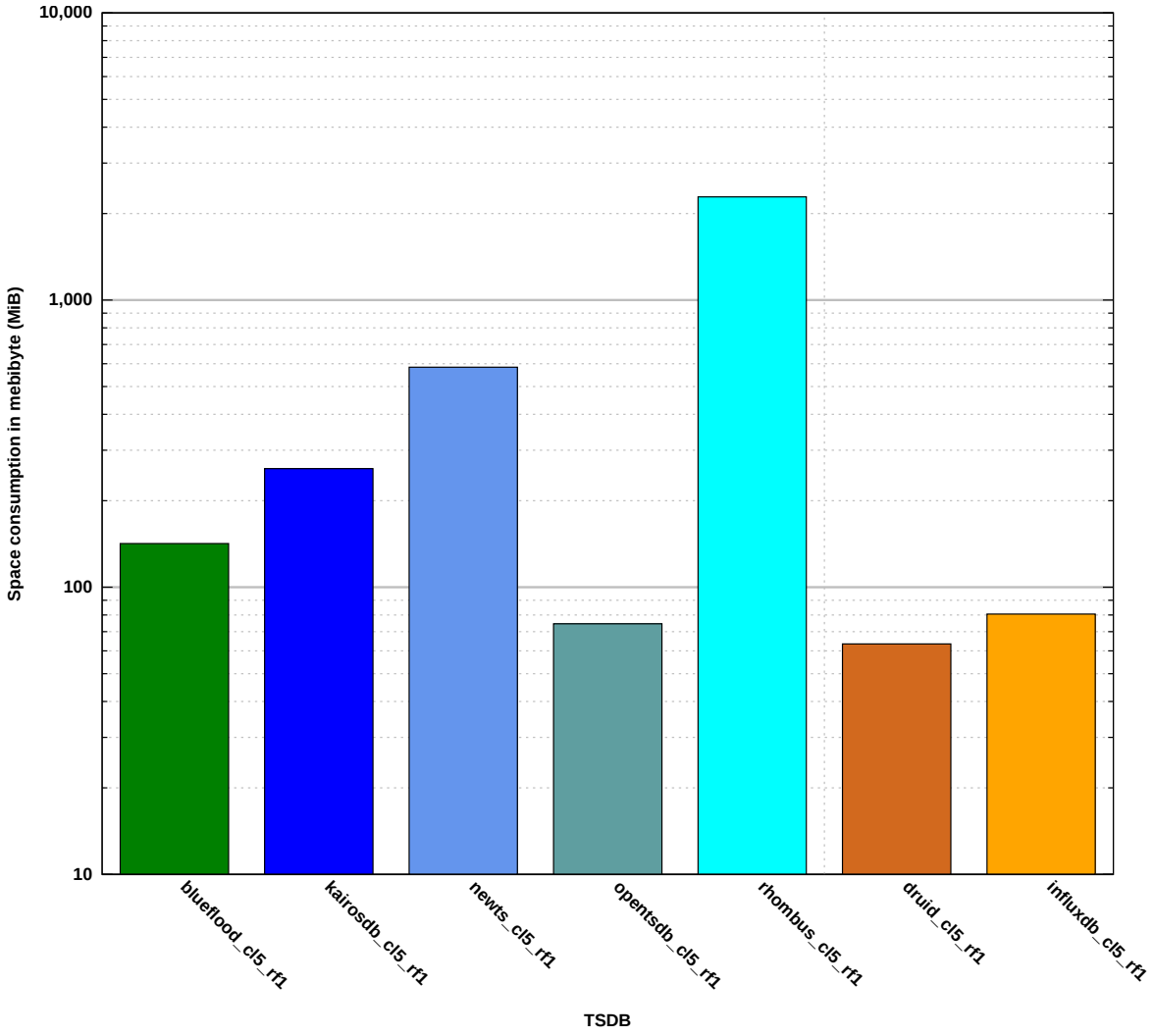

Figure 5.6: Space consumption results for a five node cluster with a RF of one.

5.1.3 Five Node Cluster with a RF of Two

5.1.3.1 Query Latency for INS Queries

Figure 5.7 presents the results for INS queries in a five node cluster with a RF of two, which means that one value is stored on two nodes. There is no 99th percentile value shown for OpenTSDB because the value lies below one ms. One of the results from this measurement is that OpenTSDB has the lowest 99th percentile value and the lowest average value of 0.223 ms. The second fastest TSDB is Druid, which has an average value of 0.438 ms and a 99th percentile value of one ms. Blueflood has the third best average query latency with 0.808 ms, which is faster than the value of 0.935 ms from Section 5.1.2.1. The worst 99th percentile value achieves NewTS, the worst average value achieves InfluxDB.

5.1.3.2 Query Latency for READ Queries

Figure 5.8 presents the results for READ queries in a five node cluster with a RF of two. Rhombus has the best 99th percentile value with 2 ms and the best average value with 1.94 ms. Compared to Section 5.1.2.2 with one replication for each value, Rhombus slightly decreased its average query latency when using the double amount of replications. NewTS has the second best result, with a 99th percentile value of 7 ms and an average value of 3.01 ms. NewTS has slightly decreased its latencies compared to Section 5.1.2.2. The next best two TSDBs are Blueflood and InfluxDB, none of them is the best in both, the 99th percentile and the average, values. The worst result for READ queries has KairosDB (with Cassandra), they are around two orders of magnitude worse than the remaining TSDBs.

5.1.3.3 Space Consumption

Figure 5.9 presents the difference in space consumption in a five node cluster with a RF of two when 1,000,000 INS queries are executed. The space consumption is measured before and after the execution. An increase in space consumption is expected compared to Section 5.1.2.3, as with a RF of two, two values must be stored, which uses more space than storing one value.

Druid uses with 58.8 MiB the fewest amount of all compared TSDBs, followd by OpenTSDB with 149.1 MiB, and InfluxDB with 160 MiB. Rhombus uses with 4,919 MiB the most space. Compared to Section 5.1.2.3, the space consumption approximately

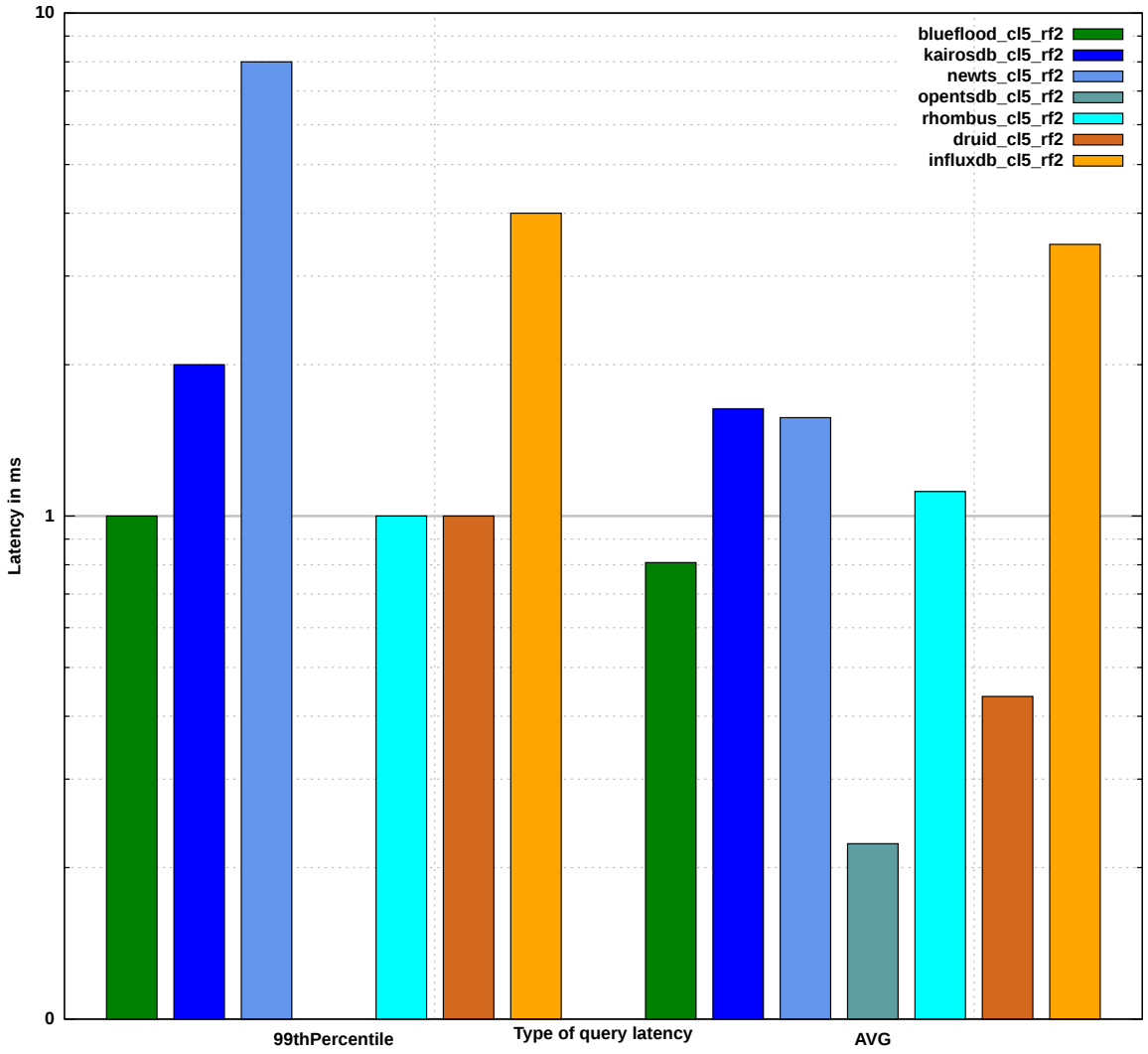

Figure 5.7: Query latency results for INS queries in a five node cluster with a RF of two. There is no 99th percentile value shown for OpenTSDB because the value lies below one ms.

doubles for all TSDBs, except Druid. Druid uses less space than in Section 5.1.2.3, which is unexpected. Three reasons are possible: First, due to the limitation to five nodes, which means that only one type of each node exists[7], Druid probably stores all copies on the same node or does not even produce a copy. Second, Druid has enough space allocated and did not need additional space for another segment of data. Third,

[7]Druid has five types of nodes, which are explained in Section 3.3.

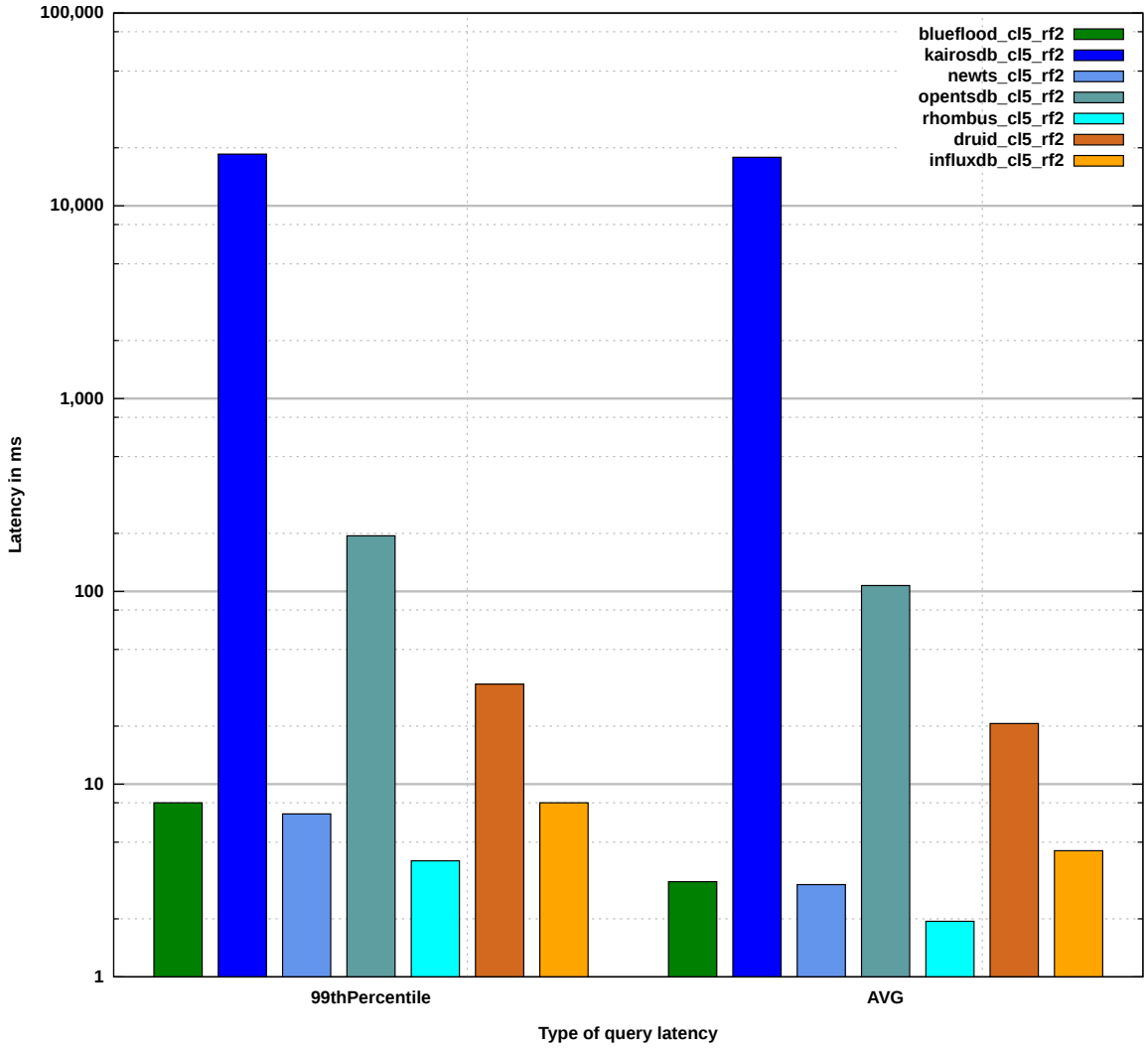

Figure 5.8: Query latency results for READ queries in a five node cluster with a RF of two.

Druid caches the data in RAM a long time and due to its complex architecture was not measured.

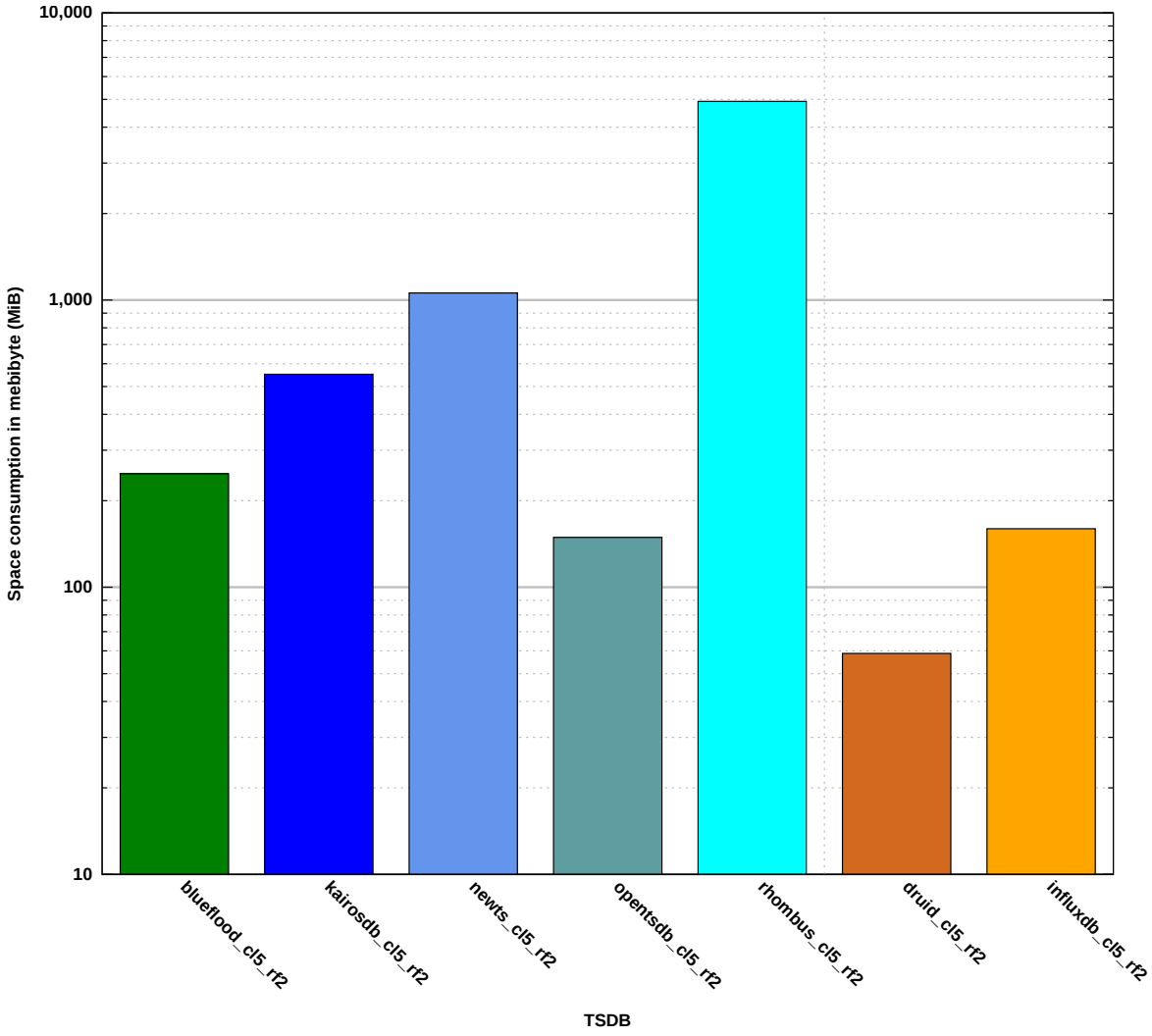

Figure 5.9: Space consumption results for a five node cluster with a RF of two.

5.1.4 Five Node Cluster with a RF of Five

Figure 5.10 presents the results for INS queries in a five node cluster with a RF of two, which means that one value is stored on five nodes. There is no 99th percentile value shown for OpenTSDB because the value lies below one ms. One of the results from this measurement is that OpenTSDB has the lowest 99th percentile value and the lowest average value of 0.290 ms. The second fastest TSDB is Druid, which has an average value of 0.542 ms and a 99th percentile value of one ms. Blueflood has the third best average query latency with 0.707 ms, which is faster than the value of 0.808 ms from Section 5.1.3.1. The worst 99th percentile value achieves NewTS, the worst average value achieves InfluxDB. The 99th percentile value seems to be increasing when a higher RF is used, compared to Section 5.1.3.1 and Section 5.1.2.1.

5.1.4.1 Query Latency for INS Queries

5.1.4.2 Query Latency for READ Queries

Figure 5.11 presents the results for READ queries in a five node cluster with a RF of five. Rhombus has the best 99th percentile value with 2 ms and the best average value with 1.85 ms. Compared to Section 5.1.3.2 with two replications for each value, Rhombus slightly decreased its average query latency when using five replications. Blueflood has the second best result, with a 99th percentile value of 3 ms and an average value of 2.95 ms. NewTS has lost its second place that it has in Section 5.1.3.2 and has the third best result with a 99th percentile value of 5 ms and an average value of 3.8 ms. The worst result for READ queries has KairosDB (with Cassandra), they are around two orders of magnitude worse than the remaining TSDBs. Compared to Section 5.1.3.2 and Section 5.1.2.2, the latency of the READ queries of KairosDB (with Cassandra) did increase with a RF of five..

5.1.4.3 Space Consumption

Figure 5.12 presents the difference in space consumption in a five node cluster with a RF of five when 1,000,000 INS queries are executed. The space consumption is measured before and after the execution. An increase in space consumption is expected compared to Section 5.1.3.3, as with a RF of five, five values must be stored, which uses more space than storing one or two values.

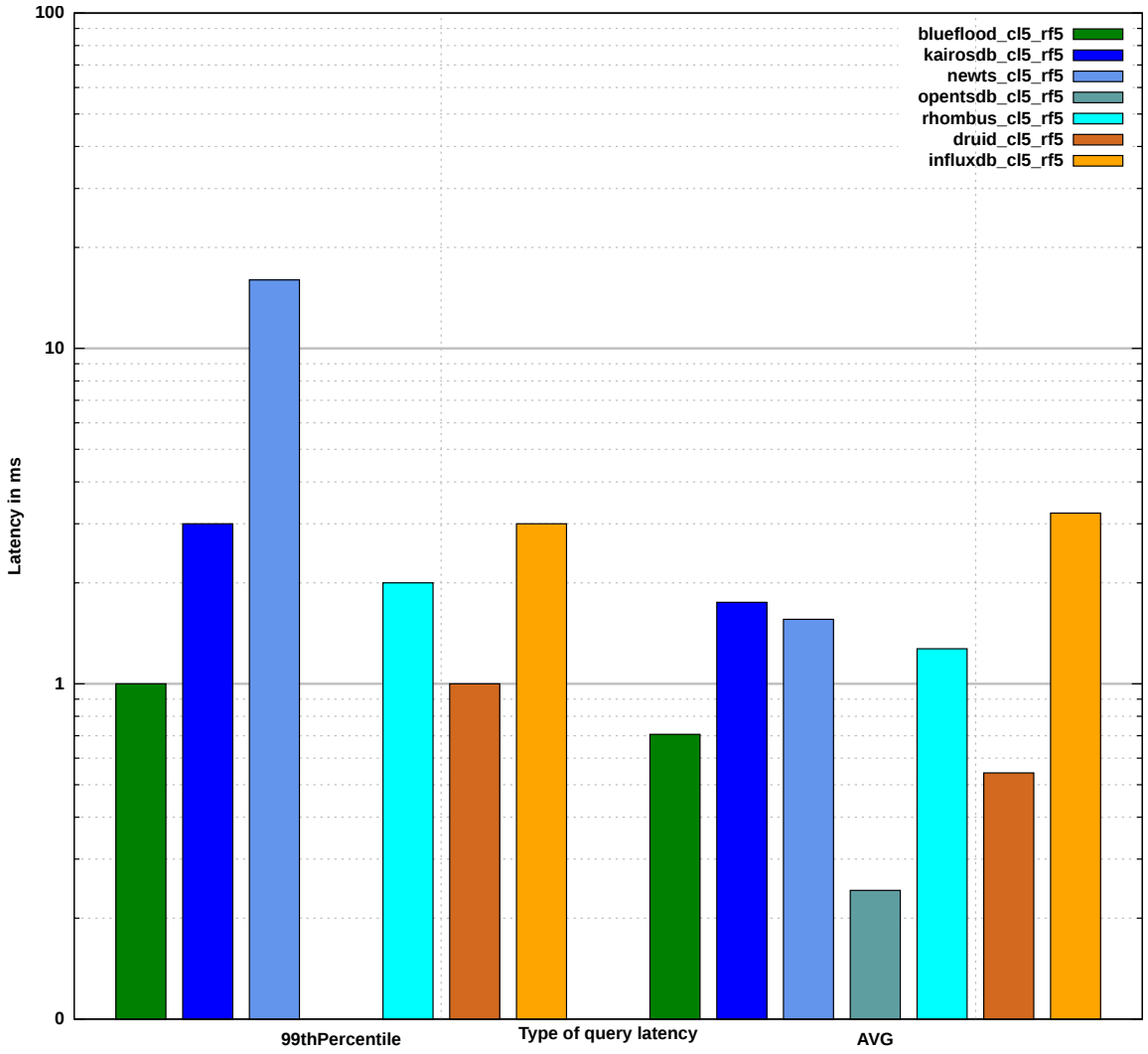

Figure 5.10: Query latency results for INS queries in a five node cluster with a RF of five. There is no 99th percentile value shown for OpenTSDB because the value lies below one ms.

Druid uses with 49.4 MiB the fewest amount of all compared TSDBs, followd by OpenTSDB with 373 MiB, and InfluxDB with 397.7 MiB. Rhombus uses with 12,854.9 MiB the most space. Compared to Section 5.1.3.3, the space consumption has increased by a factor of approximately 2.5 (e. g., OpenTSDB 2.50, Rhombus 2.61, and InfluxDB 2.49) for all TSDBs, except Druid. Druid uses less space than in Section 5.1.3.3, which is unexpected. In Section 5.1.3.3 the space consumption of Druid has been discussed. It was pointed out that druid possibly does not directly write data

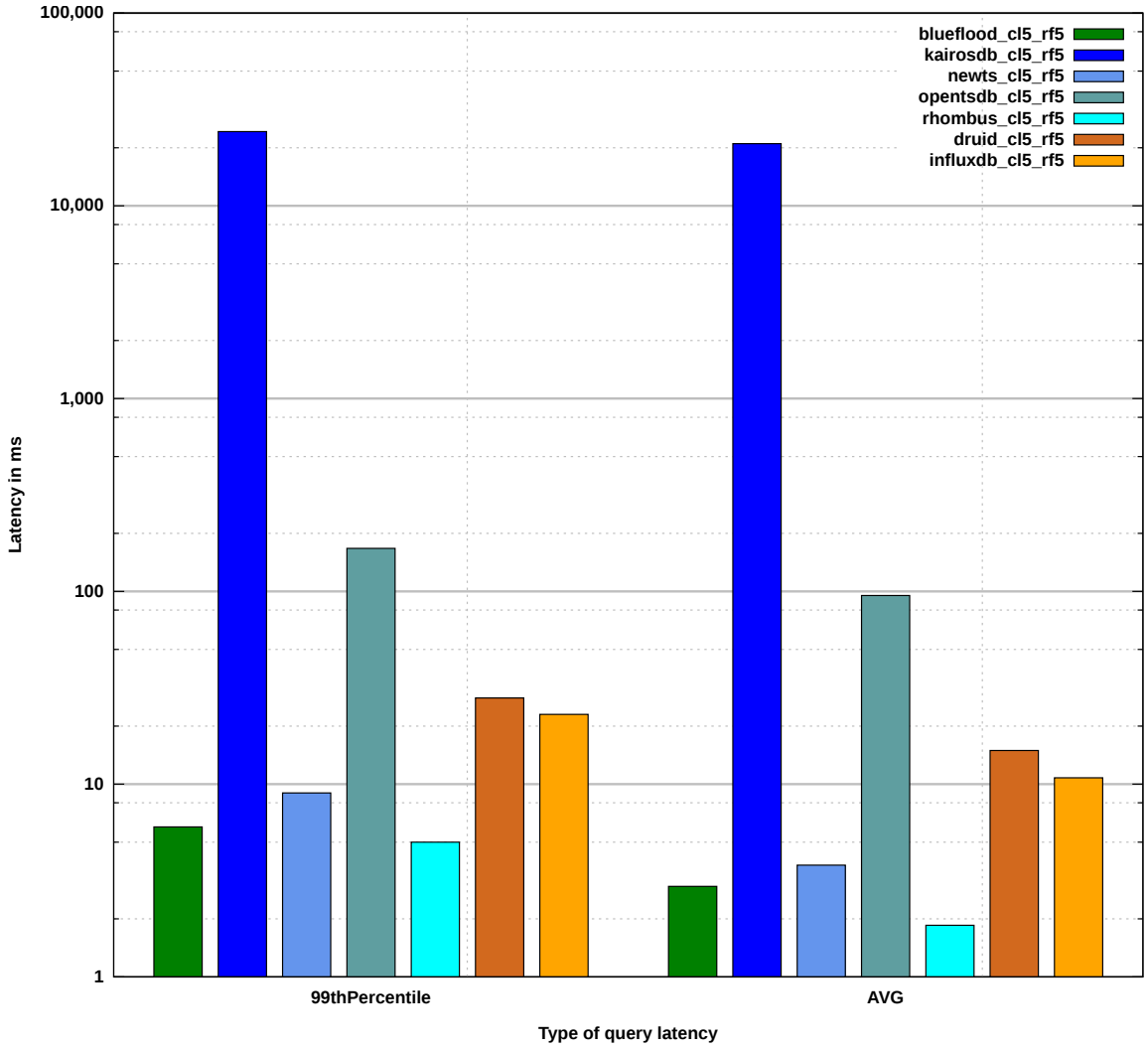

Figure 5.11: Query latency results for READ queries in a five node cluster with a RF of five.

or does not have enough nodes to replications. Therefore, the space consumption is lower here than expected.

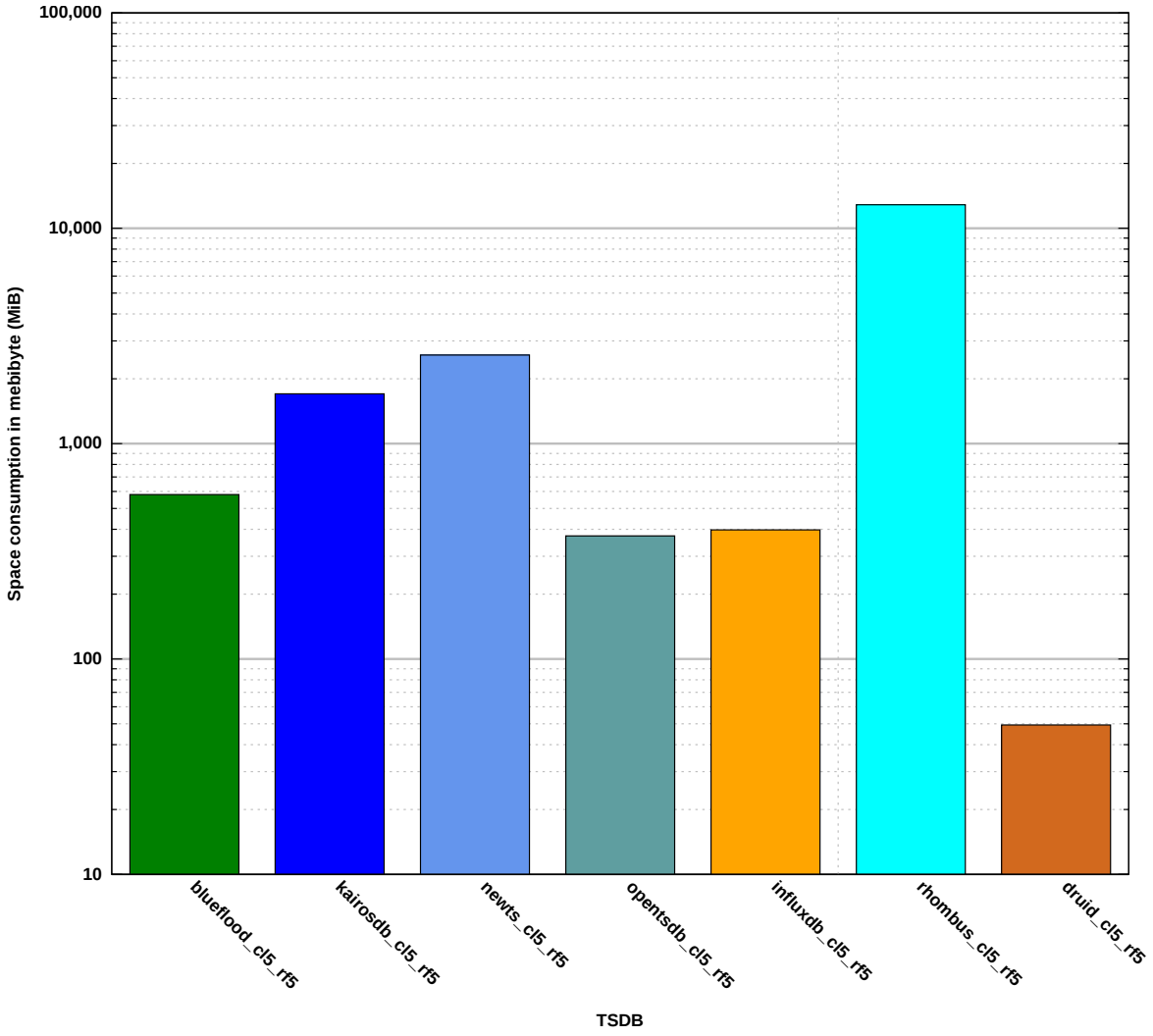

Figure 5.12: Space consumption results for a five node cluster with a RF of five.

5.2 Scenario 2: 250 SCAN, AVG, SUM, and CNT Queries

5.2.1 One Node Cluster with a RF of One

5.2.1.1 Query Latency for SCAN Queries

Figure 5.13 presents the results for SCAN queries in a one node cluster with a RF of one. The result of this measurement is that Rhombus has the best query latency for SCAN queries measured with a 99th percentile value of 8 ms and an average value of

1.55 ms. The second best result has MonetDB with a 99th percentile value of 20 ms and an average value of 9.98 ms. The average value is nearly an order of magnitude away from Rhombus. The third best results has Druid. KairosDB with H2 has the worst result, preceded by itself in combination with Cassandra.

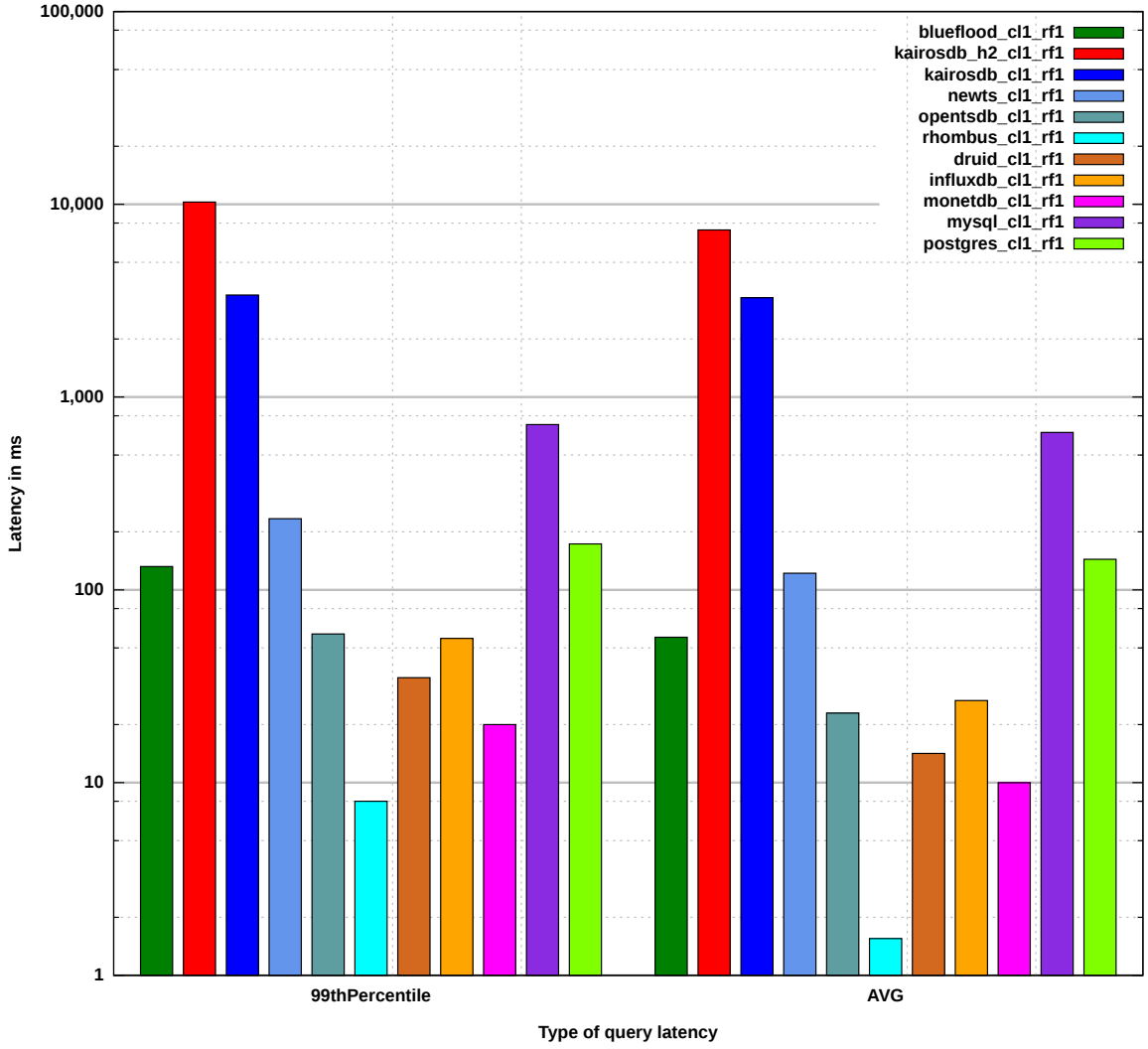

Figure 5.13: Query latency results for SCAN queries in a one node cluster with a RF of one.

5.2.1.2 Query Latency for AVG Queries

Figure 5.14 presents the results for AVG queries in a one node cluster with a RF of one. The results of Rhombus are omitted because AVG queries are not supported,

see Section 4.7 and Appendix A.6. The result of this measurement is that MonetDB has the best query latency for AVG queries measured with a 99th percentile value of 15 ms and an average value of 6.49 ms. The second best result has Druid with a 99th percentile value of 39 ms and an average value of 15.2 ms. The third best result has InfluxDB. KairosDB with H2 has the worst result, preceded by itself in combination with Cassandra.

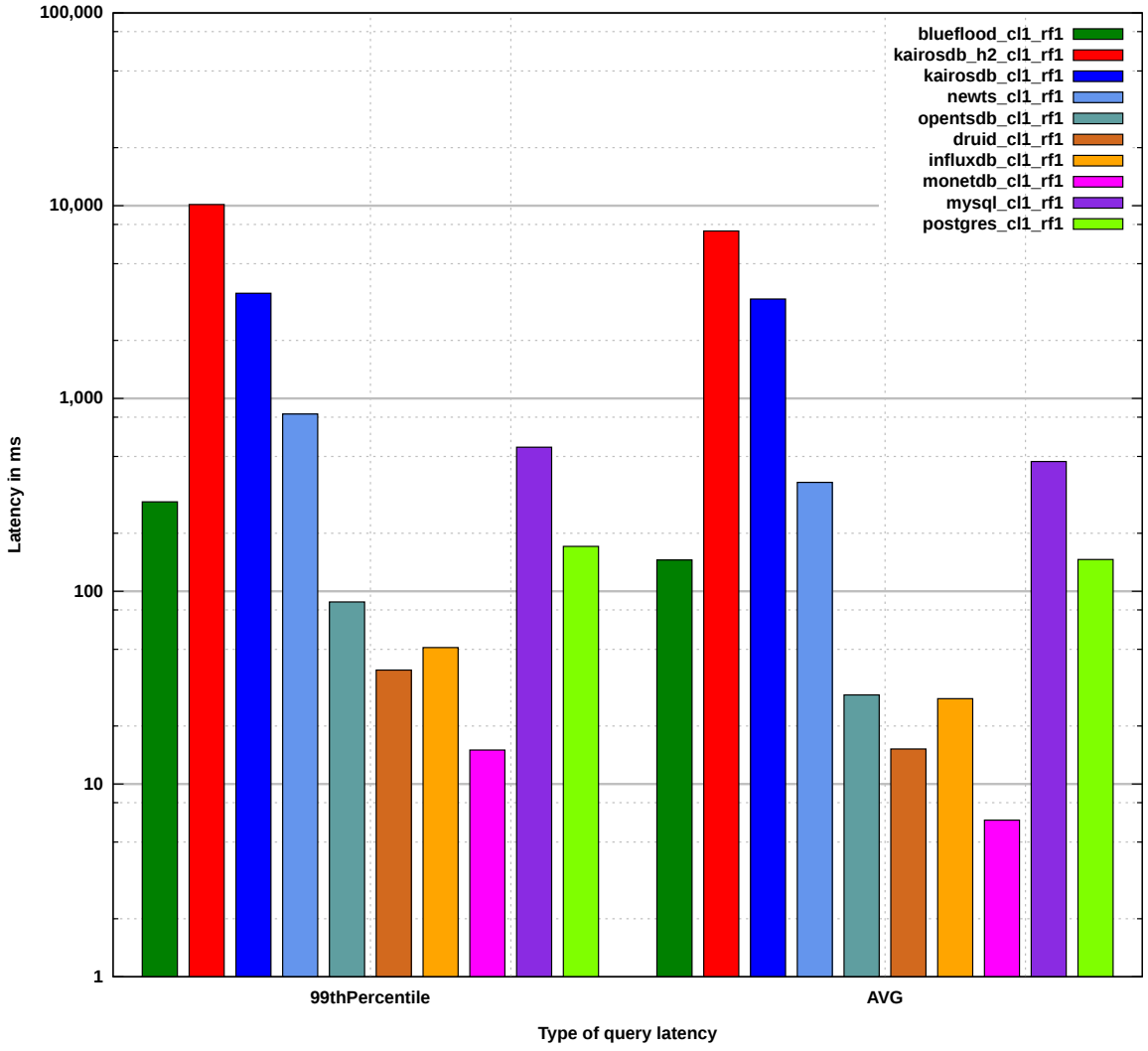

Figure 5.14: Query latency results for AVG queries in a one node cluster with a RF of one.

5.2.1.3 Query Latency for SUM Queries

Figure 5.15 presents the results for SUM queries in a one node cluster with a RF of one. The results of Blueflood, NewTS, and Rhombus are omitted because SUM queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that MonetDB has the best query latency for SUM queries measured with a 99^{th} percentile value of 14 ms and an average value of 6.28 ms. The second best result has Druid with a 99^{th} percentile value of 12.7 ms and an average value of 25 ms. The third best average value has OpenTSDB and the third best 99^{th} percentile value has InfluxDB. KairosDB with H2 has the worst result, preceded by itself in combination with Cassandra.

5.2.1.4 Query Latency for CNT Queries

Figure 5.16 presents the results for CNT queries in a one node cluster with a RF of one. The results of NewTS are omitted because CNT queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Rhombus has the best query latency for CNT queries measured with a 99^{th} percentile value of 1.41 ms and an average value of 6 ms. The second best result has MonetDB with a 99^{th} percentile value of 13 ms and an average value of 6.11 ms. The third best result has Druid. KairosDB with H2 has the worst result, preceded by itself in combination with Cassandra.

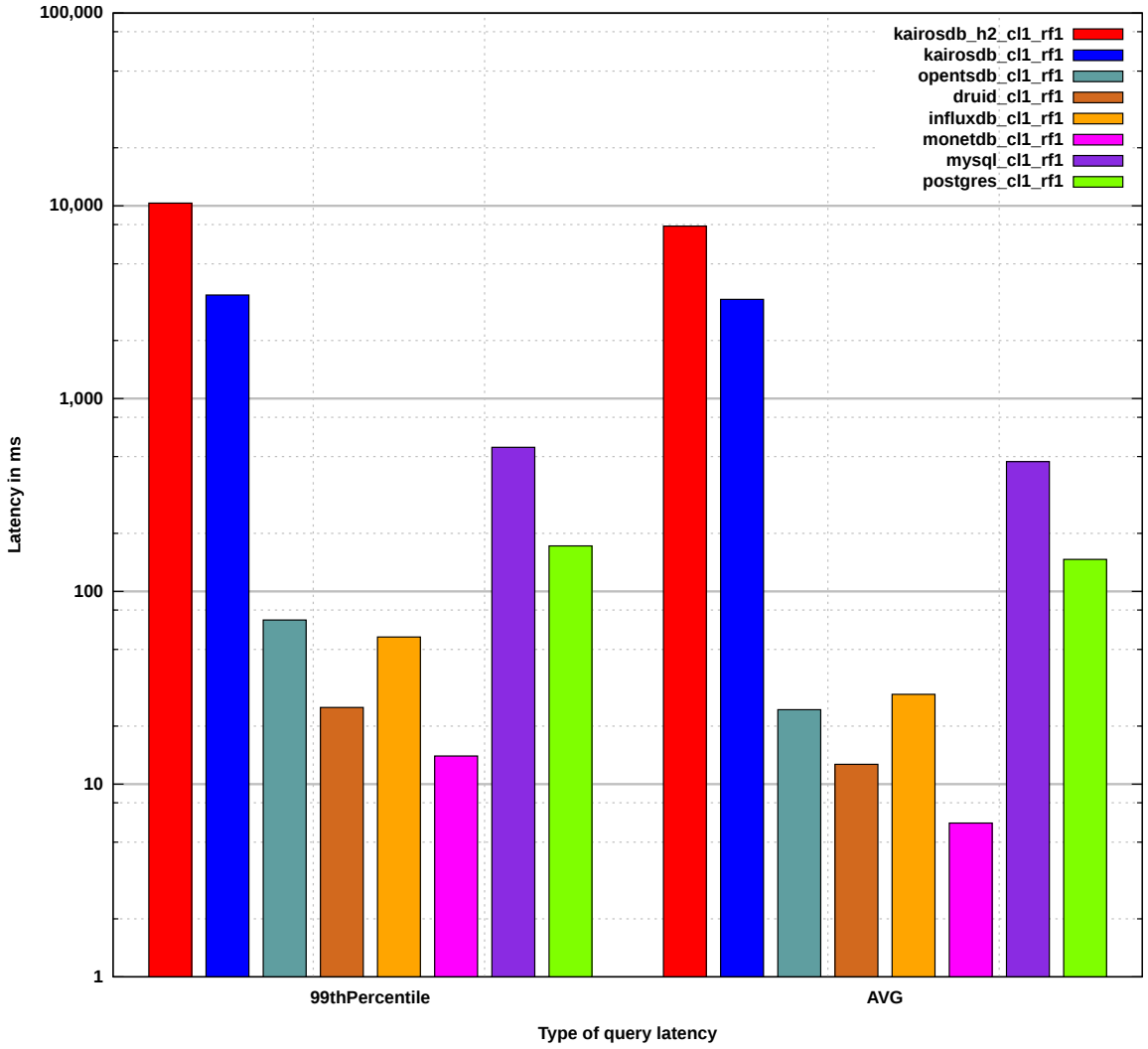

Figure 5.15: Query latency results for SUM queries in a one node cluster with a RF of one.

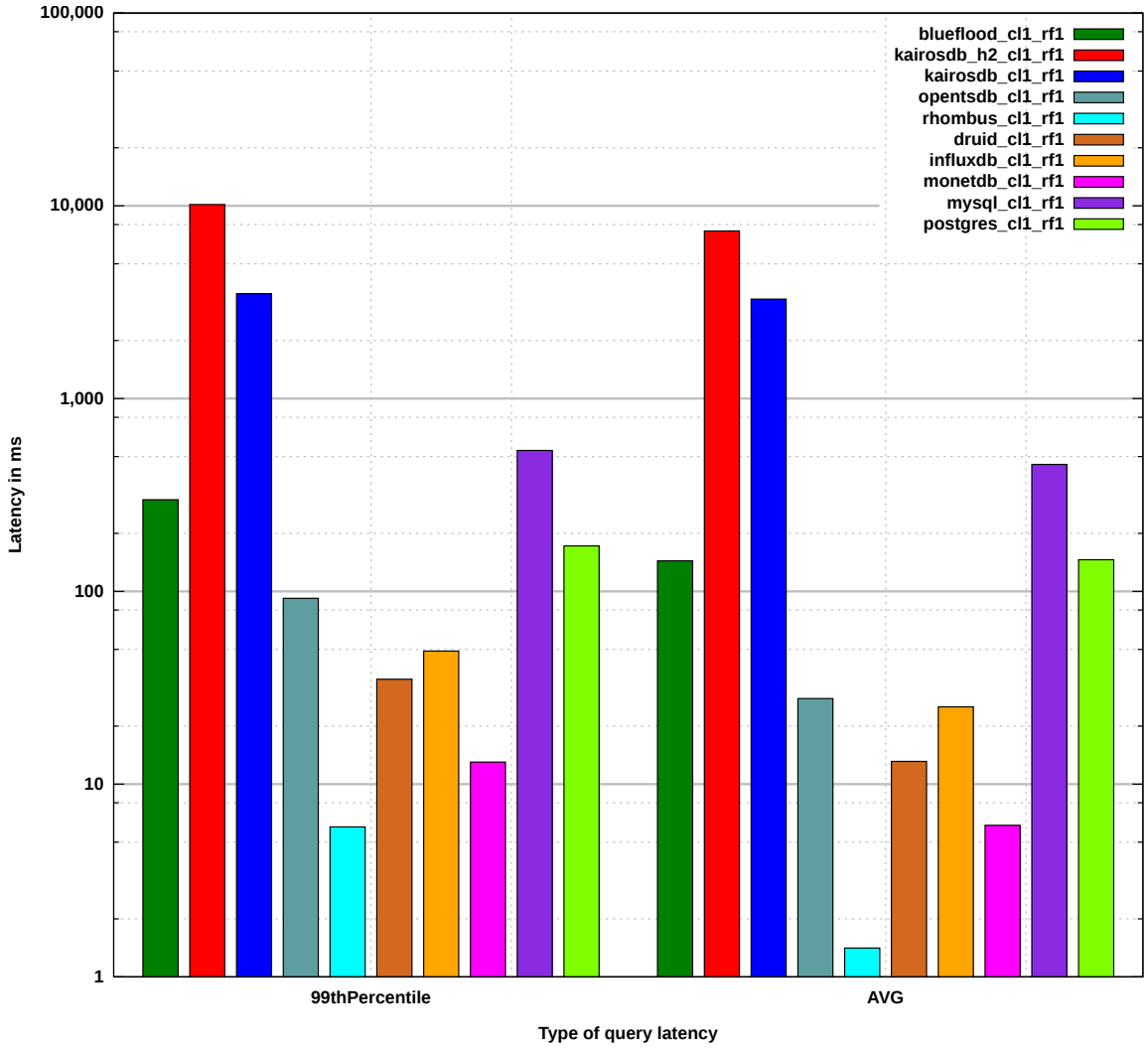

Figure 5.16: Query latency results for CNT queries in a one node cluster with a RF of one.

5.2.2 Five Node Cluster with a RF of One

5.2.2.1 Query Latency for SCAN Queries

Figure 5.17 presents the results for SCAN queries in a five node cluster with a RF of one. The result of this measurement is that Rhombus has the best query latency for SCAN queries measured with a 99th percentile value of 29 ms and an average value of 2.94 ms. The second best result has Druid with a 99th percentile value of 41 ms and an average value of 14.7 ms. The third best result has OpenTSDB. KairosDB (with Cassandra) has the worst result.

5.2.2.2 Query Latency for AVG Queries

Figure 5.18 presents the results for AVG queries in a five node cluster with a RF of one. The result of Rhombus is omitted because AVG queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Druid has the best query latency for AVG queries measured with a 99th percentile value of 36 ms and an average value of 16.5 ms. The second best 99th percentile value has InfluxDB with 14 ms and the second best average value has OpenTSDB with 25.4 ms. KairosDB (with Cassandra) has the worst result.

5.2.2.3 Query Latency for SUM Queries

Figure 5.19 presents the results for SUM queries in a five node cluster with a RF of one. The results of Blueflood, NewTS, and Rhombus are omitted because SUM queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Druid has the best query latency for SUM queries measured with a 99th percentile value of 28 ms and an average value of 13.6 ms. The second best 99th percentile value has InfluxDB with 58 ms and the second best average value has OpenTSDB with 26.7 ms. KairosDB (with Cassandra) has the worst result.

5.2.2.4 Query Latency for CNT Queries

Figure 5.20 presents the results for CNT queries in a five node cluster with a RF of one. The result of NewTS is omitted because CNT queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Rhombus has the best query latency for CNT queries measured with a 99th percentile value of 16 ms and an average

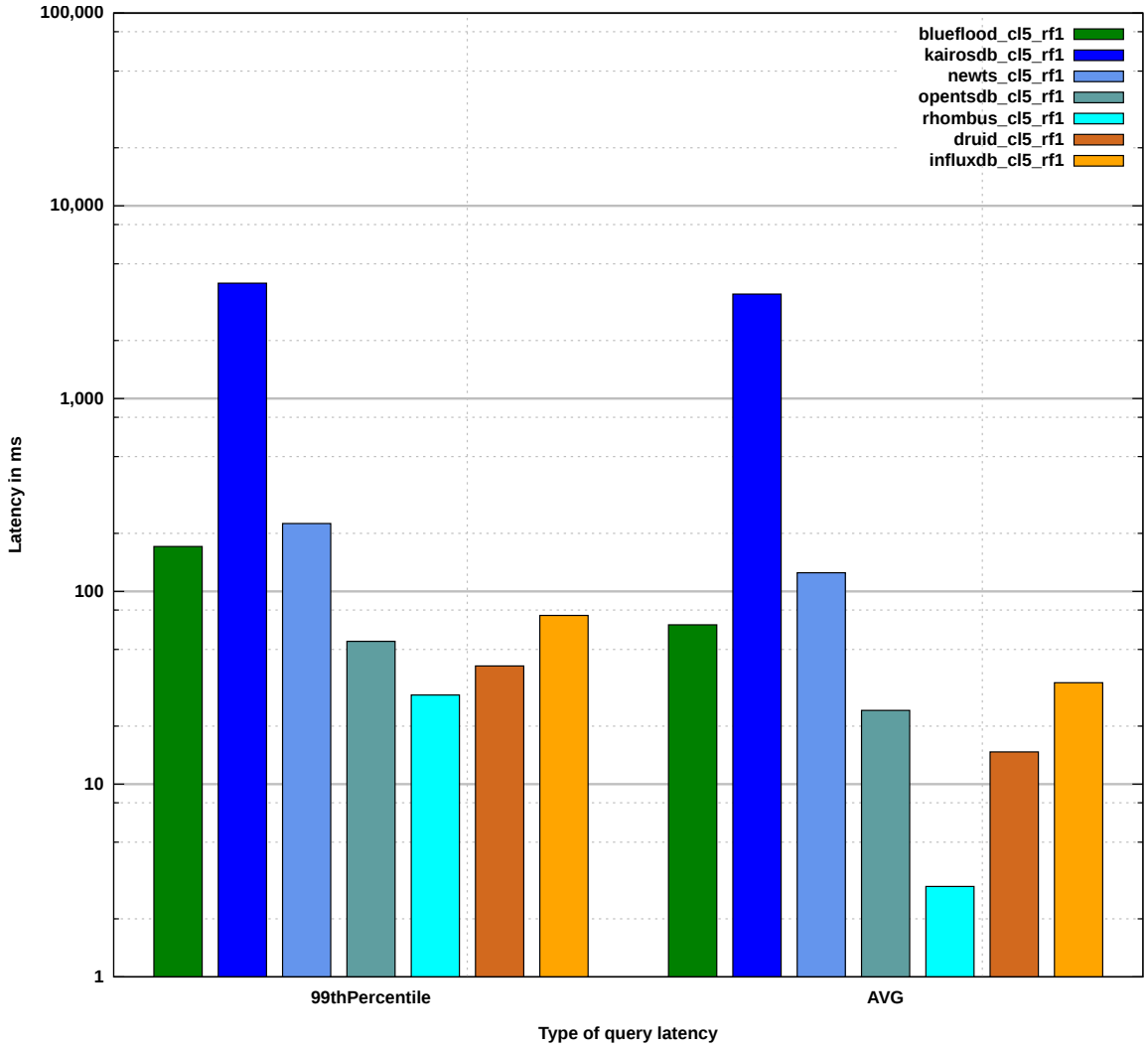

Figure 5.17: Query latency results for SCAN queries in a five node cluster with a RF of one.

value of 2.95 ms. The second best result has Druid with a 99th percentile value of 38 ms and an average value of 13.8 ms. The third best average value has OpenTSDB and the third best 99th percentile value has InfluxDB. KairosDB (with Cassandra) has the worst result.

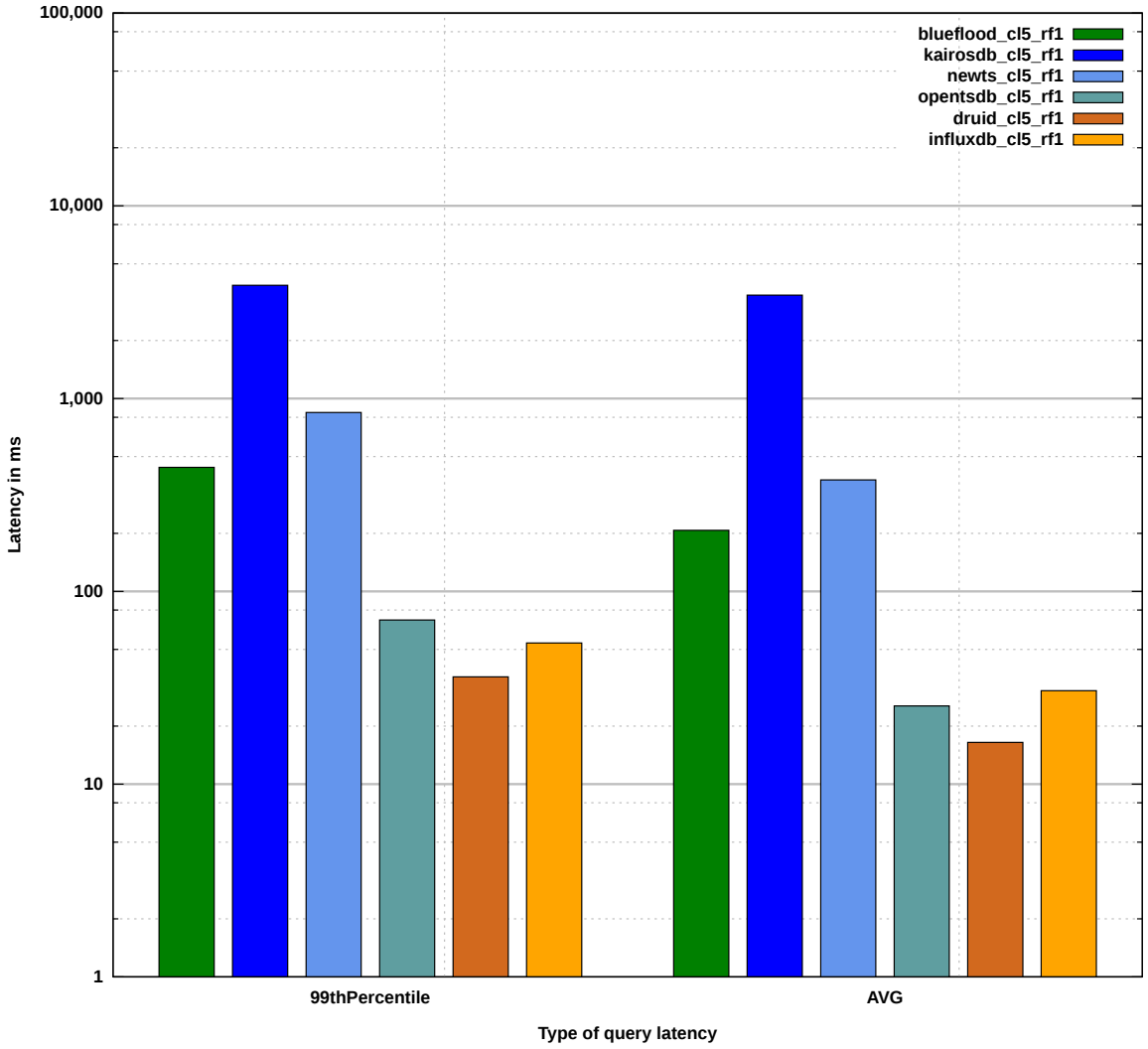

Figure 5.18: Query latency results for AVG queries in a five node cluster with a RF of one.

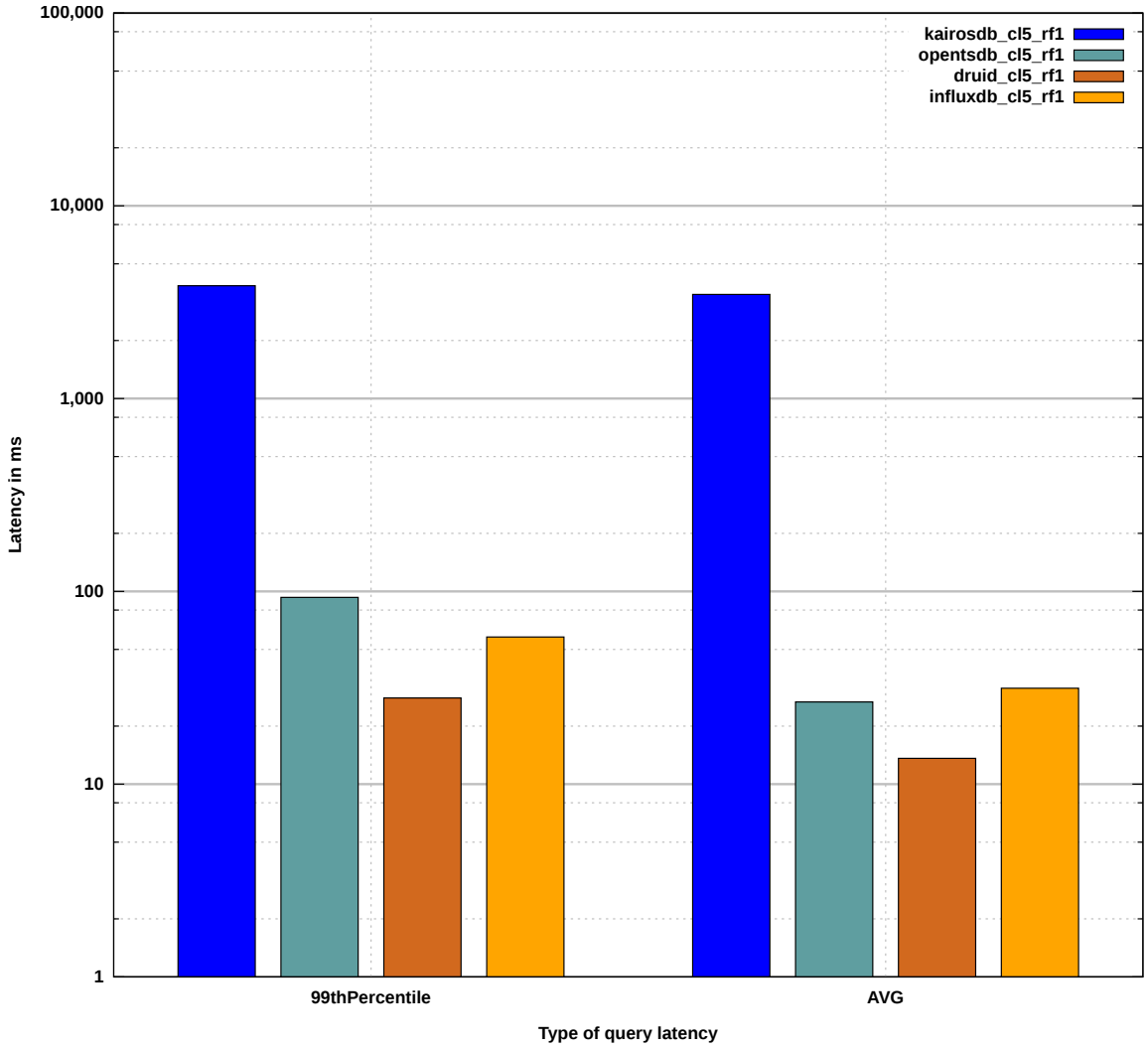

Figure 5.19: Query latency results for SUM queries in a five node cluster with a RF of one.

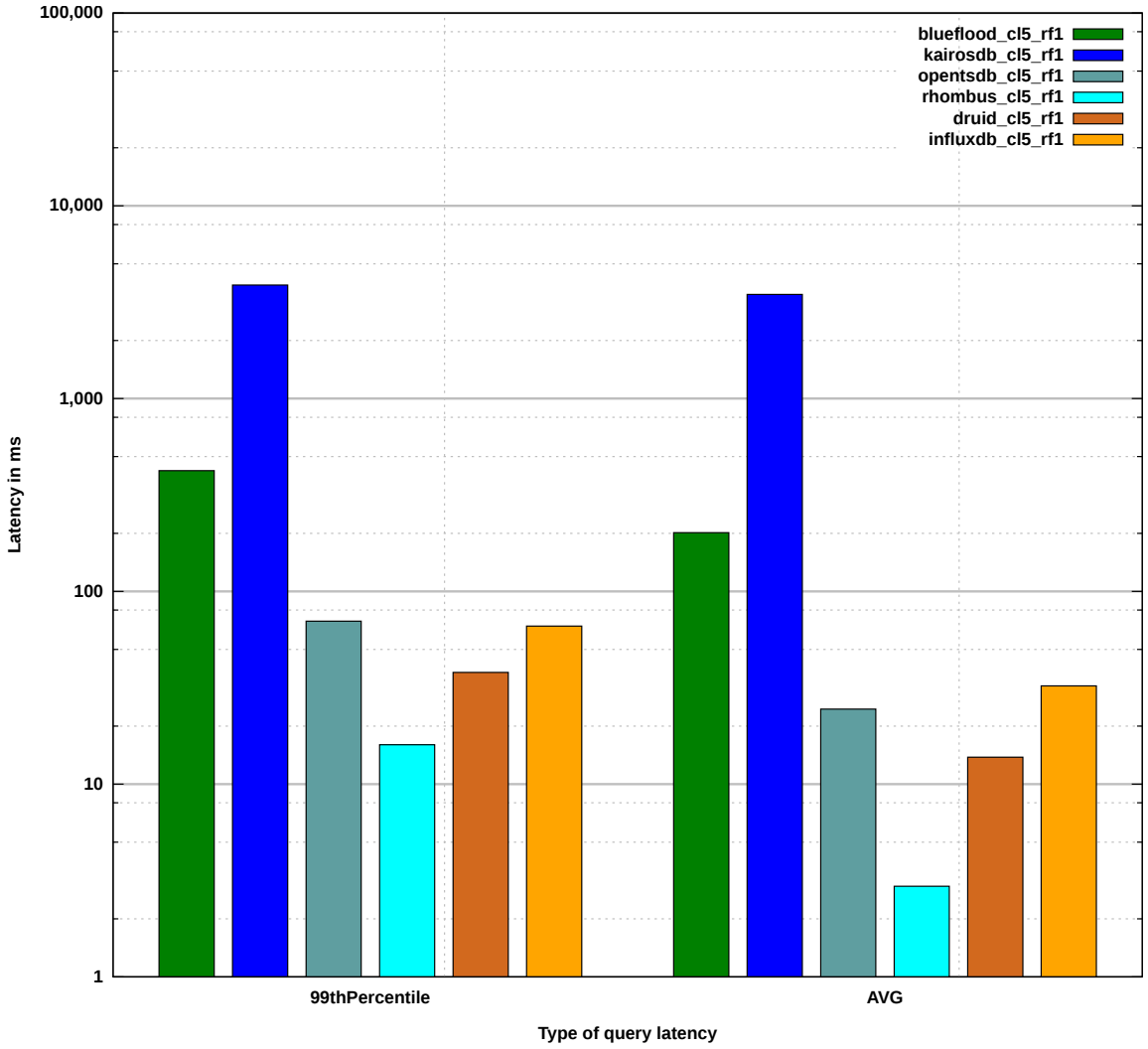

Figure 5.20: Query latency results for CNT queries in a five node cluster with a RF of one.

5.2.3 Five Node Cluster with a RF of Two

5.2.3.1 Query Latency for SCAN Queries

Figure 5.21 presents the results for SCAN queries in a five node cluster with a RF of two. The result of this measurement is that Rhombus has the best query latency for SCAN queries measured with a 99th percentile value of 31 ms and an average value of 2.34 ms. The second best result has Druid with a 99th percentile value of 49 ms and an average value of 15.6 ms. The third best average value has OpenTSDB and the third best 99th percentile value has InfluxDB. KairosDB (with Cassandra) has the worst result.

5.2.3.2 Query Latency for AVG Queries

Figure 5.22 presents the results for AVG queries in a five node cluster with a RF of two. The result of Rhombus is omitted because AVG queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Druid has the best query latency for AVG queries measured with a 99th percentile value of 35 ms and an average value of 13.3 ms. The second best 99th percentile value has InfluxDB with 56 ms and the second best average value has OpenTSDB with 26.4 ms. KairosDB (with Cassandra) has the worst result.

5.2.3.3 Query Latency for SUM Queries

Figure 5.23 presents the results for SUM queries in a five node cluster with a RF of two. The results of Blueflood, NewTS, and Rhombus are omitted because SUM queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Druid has the best query latency for SUM queries measured with a 99th percentile value of 26 ms and an average value of 12.1 ms. The second best result has InfluxDB with a 99th percentile value of 54 ms and an average value of 26.7 ms. The third best result has OpenTSDB. KairosDB (with Cassandra) has the worst result.

5.2.3.4 Query Latency for CNT Queries

Figure 5.24 presents the results for CNT queries in a five node cluster with a RF of two. The result of NewTS is omitted because CNT queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Rhombus has the

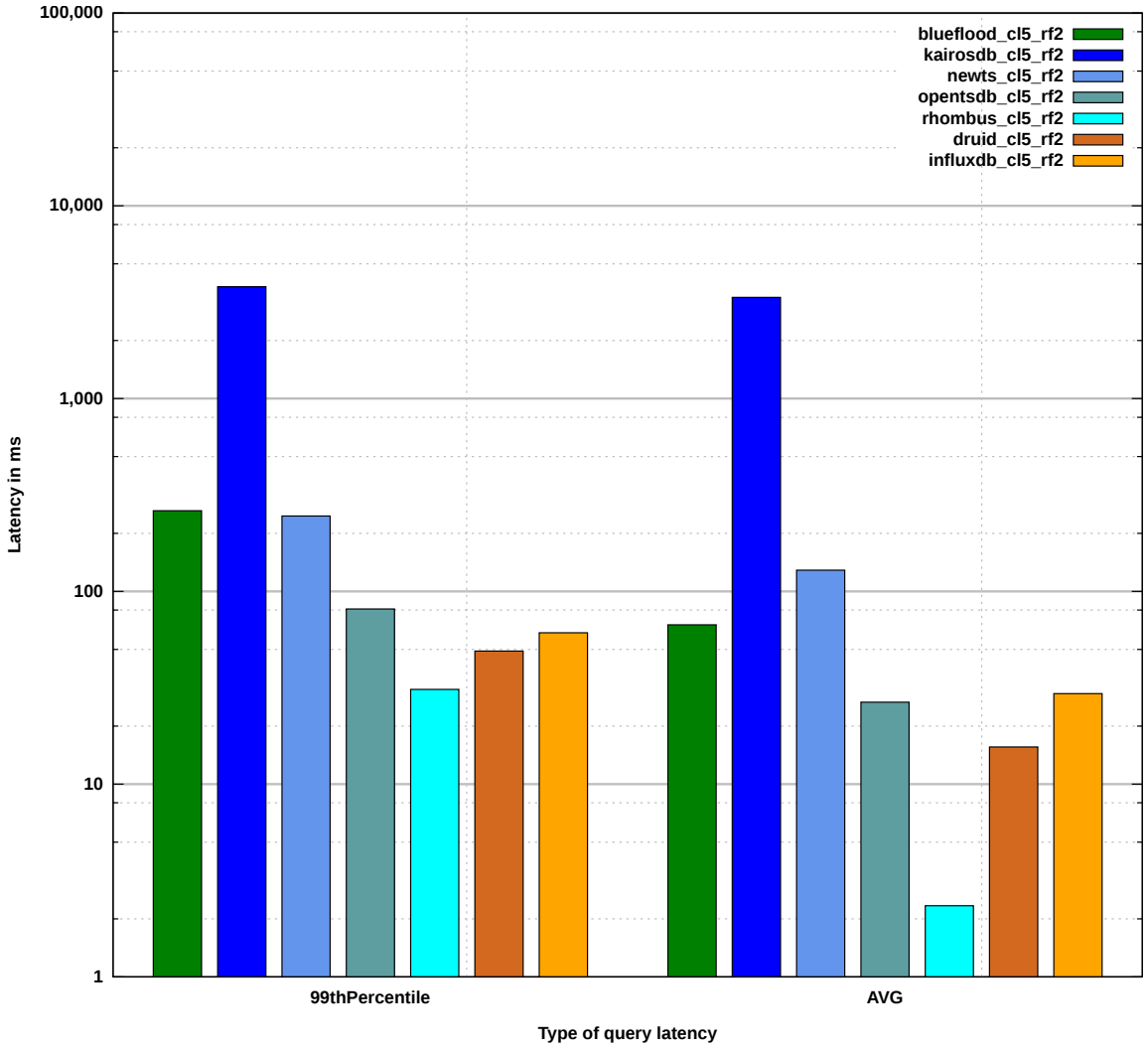

Figure 5.21: Query latency results for SCAN queries in a five node cluster with a RF of two.

best query latency for CNT queries measured with a 99th percentile value of 10 ms and an average value of 2.23 ms. The second best result has Druid with a 99th percentile value of 30 ms and an average value of 12.2 ms. The third best result has OpenTSDB. KairosDB (with Cassandra) has the worst result.

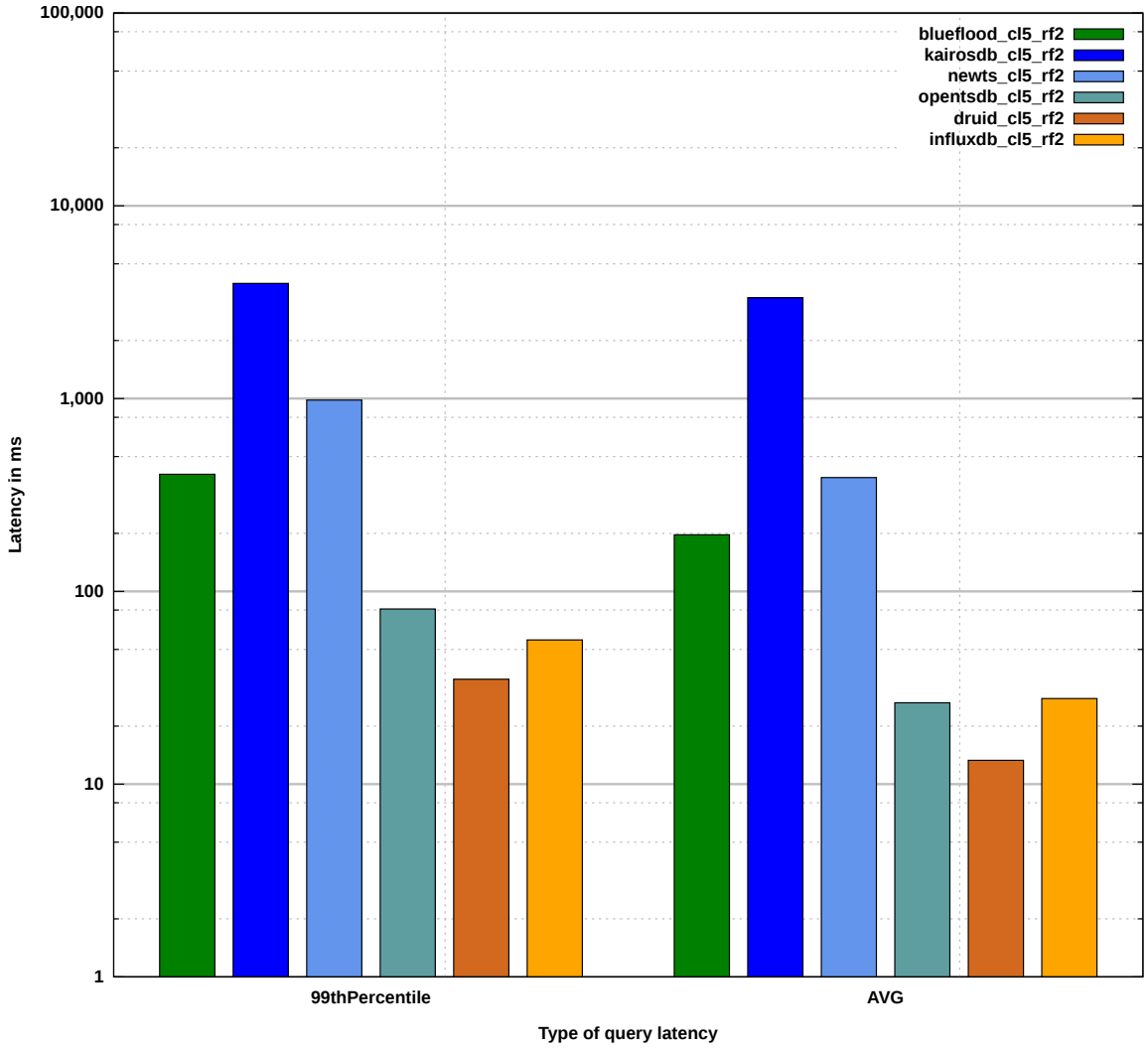

Figure 5.22: Query latency results for AVG queries in a five node cluster with a RF of two.

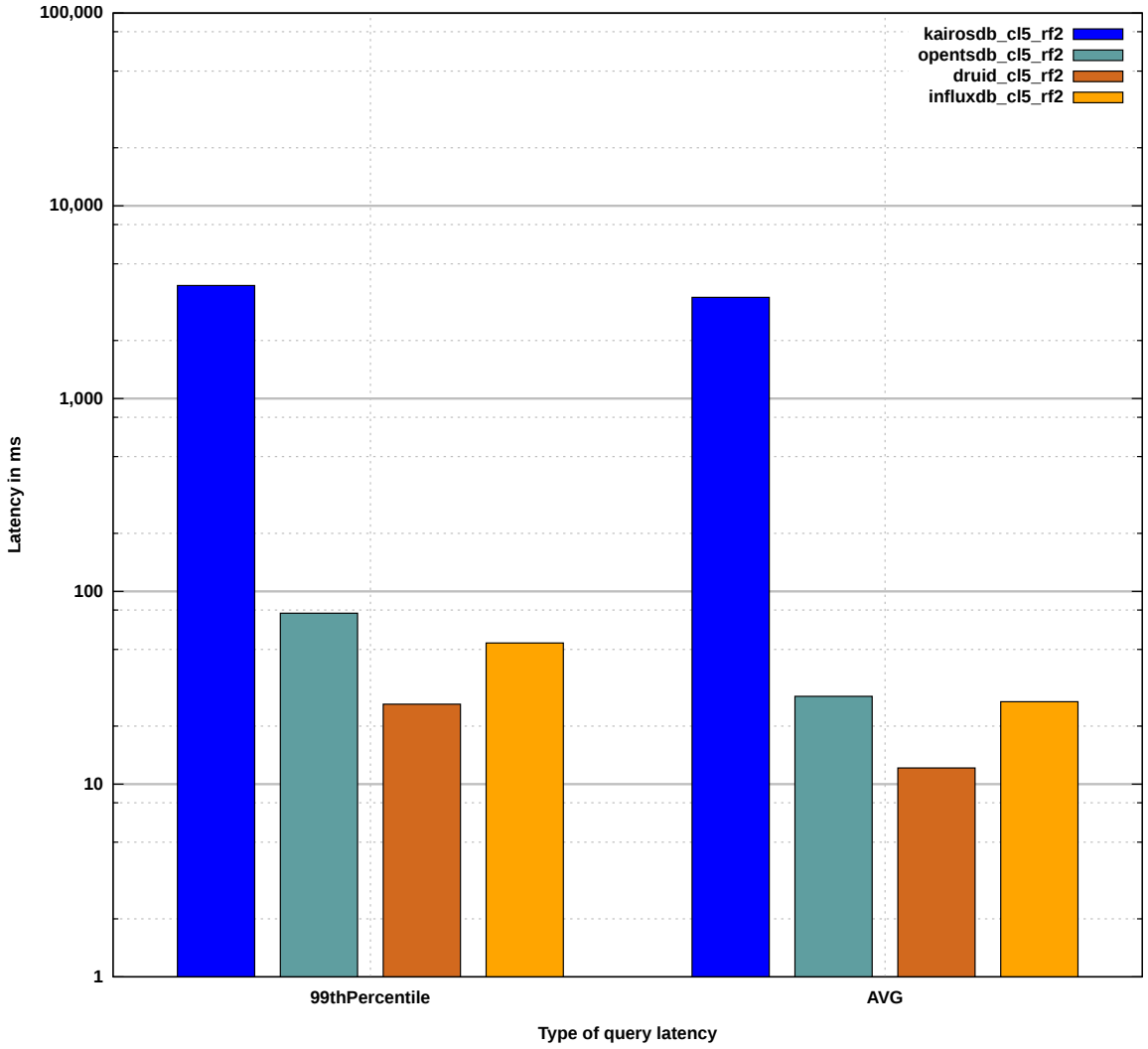

Figure 5.23: Query latency results for SUM queries in a five node cluster with a RF of two.

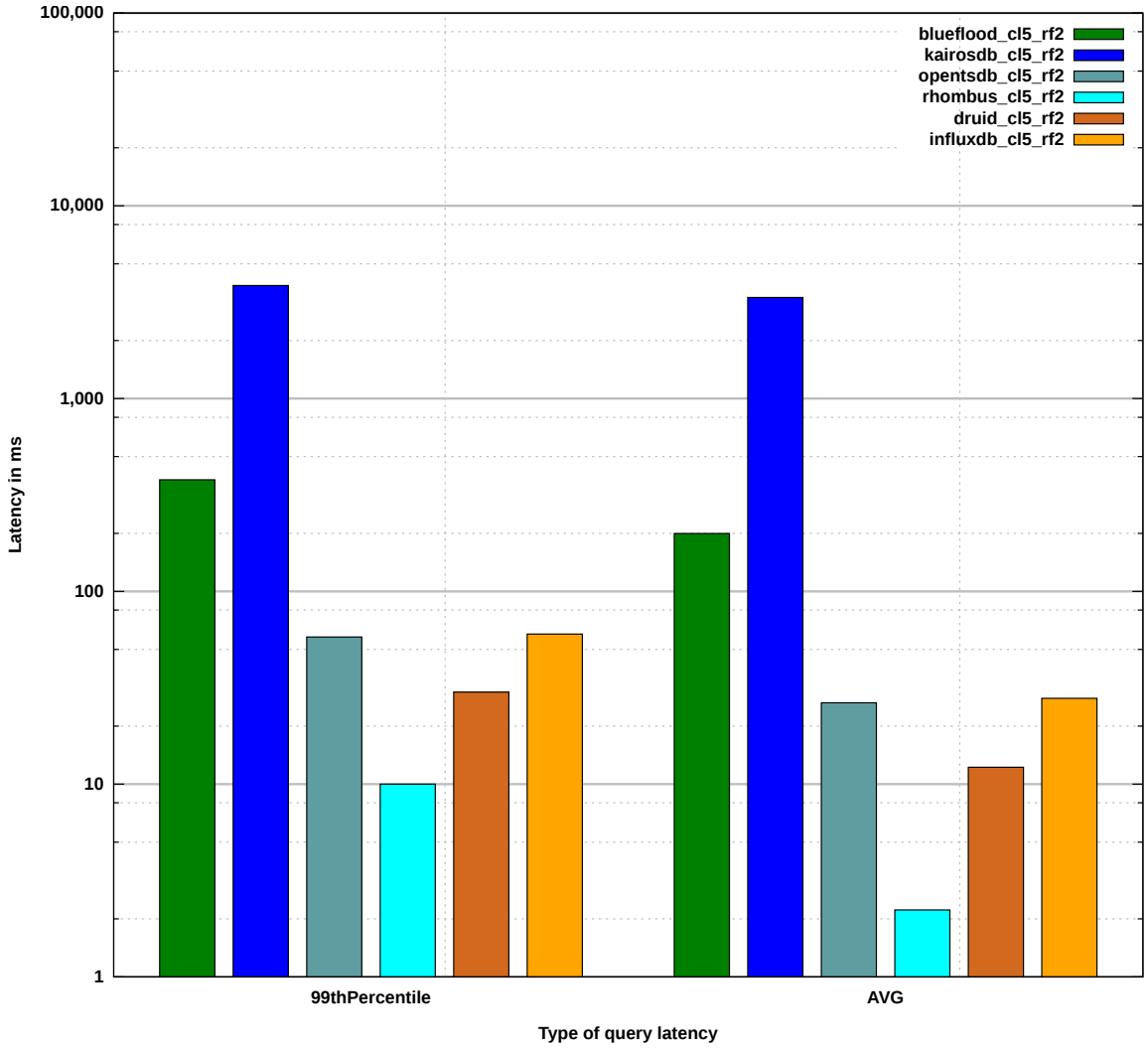

Figure 5.24: Query latency results for CNT queries in a five node cluster with a RF of two.

5.2.4 Five Node Cluster with a RF of Five

5.2.4.1 Query Latency for SCAN Queries

Figure 5.25 presents the results for SCAN queries in a five node cluster with a RF of five. The result of this measurement is that Rhombus has the best query latency for SCAN queries measured with a 99th percentile value of 20 ms and an average value of 2.2 ms. The second best result has Druid with a 99th percentile value of 44 ms and an average value of 11.8 ms. The third best result has OpenTSDB. KairosDB (with Cassandra) has the worst result.

5.2.4.2 Query Latency for AVG Queries

Figure 5.26 presents the results for AVG queries in a five node cluster with a RF of five. The result of Rhombus is omitted because AVG queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Druid has the best query latency for AVG queries measured with a 99th percentile value of 46 ms and an average value of 12.2 ms. The second best 99th percentile value has InfluxDB with 63 ms and the second best average value has OpenTSDB with 26.4 ms. KairosDB (with Cassandra) has the worst result.

5.2.4.3 Query Latency for SUM Queries

Figure 5.27 presents the results for SUM queries in a five node cluster with a RF of five. The results of Blueflood, NewTS, and Rhombus are omitted because SUM queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Druid has the best query latency for SUM queries measured with a 99th percentile value of 32 ms and an average value of 11.2 ms. The second best result has InfluxDB with a 99th percentile value of 57 ms and an average value of 29.1 ms[8]. The third best result has OpenTSDB. KairosDB (with Cassandra) has the worst result.

[8]OpenTSDB has a slightly higher average value of 30 ms, which is not obvious in Figure 5.27.

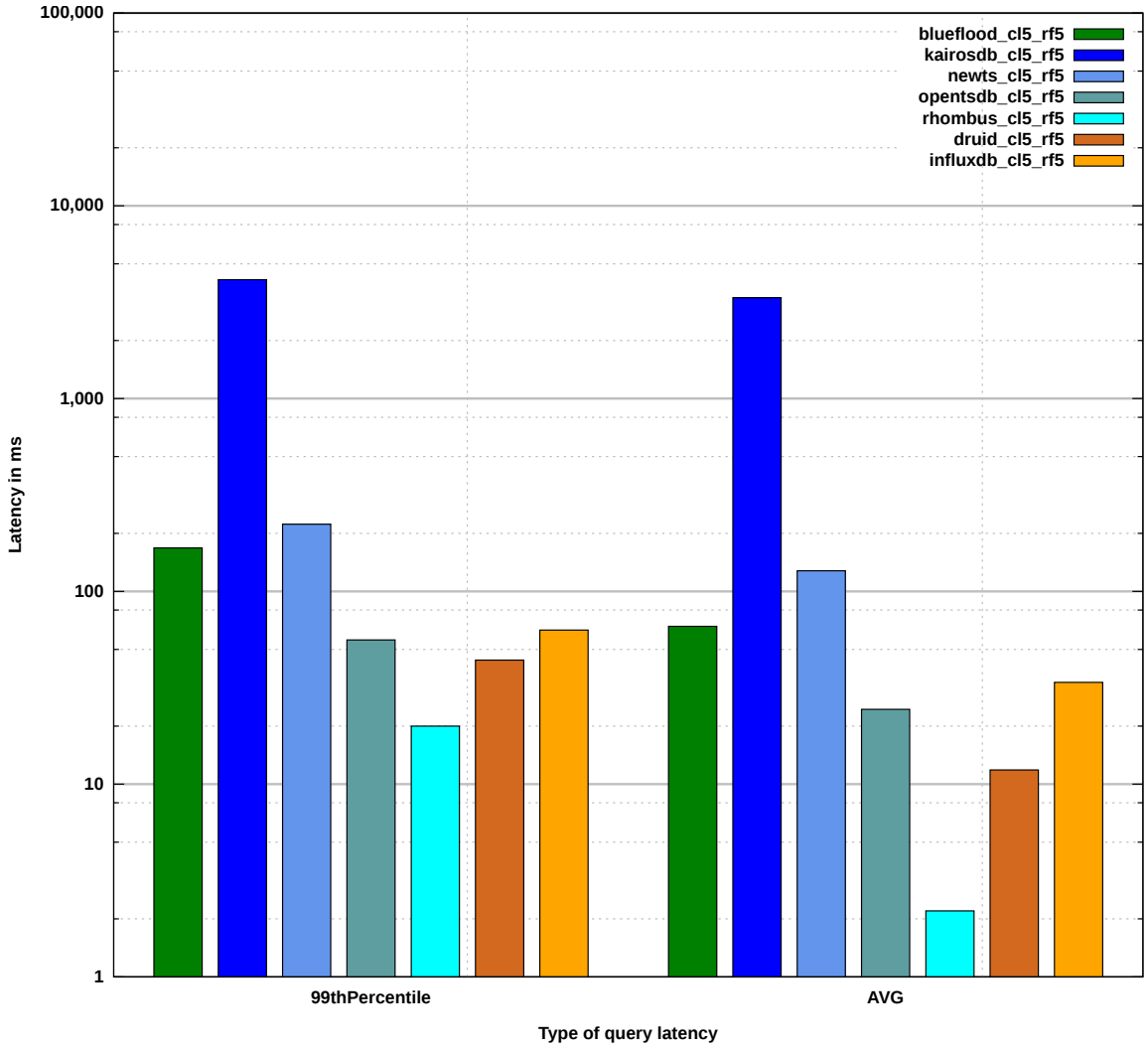

Figure 5.25: Query latency results for SCAN queries in a five node cluster with a RF of five.

5.2.4.4 Query Latency for CNT Queries

Figure 5.28 presents the results for CNT queries in a five node cluster with a RF of five. The result of NewTS is omitted because CNT queries are not supported, see Section 4.7 and Appendix A.6. The result of this measurement is that Rhombus has the best query latency for CNT queries measured with a 99th percentile value of 8 ms and an average value of 1.7 ms. The second best result has Druid with a 99th percentile value of 50 ms and an average value of 13 ms. The third best 99th percentile value has InfluxDB and

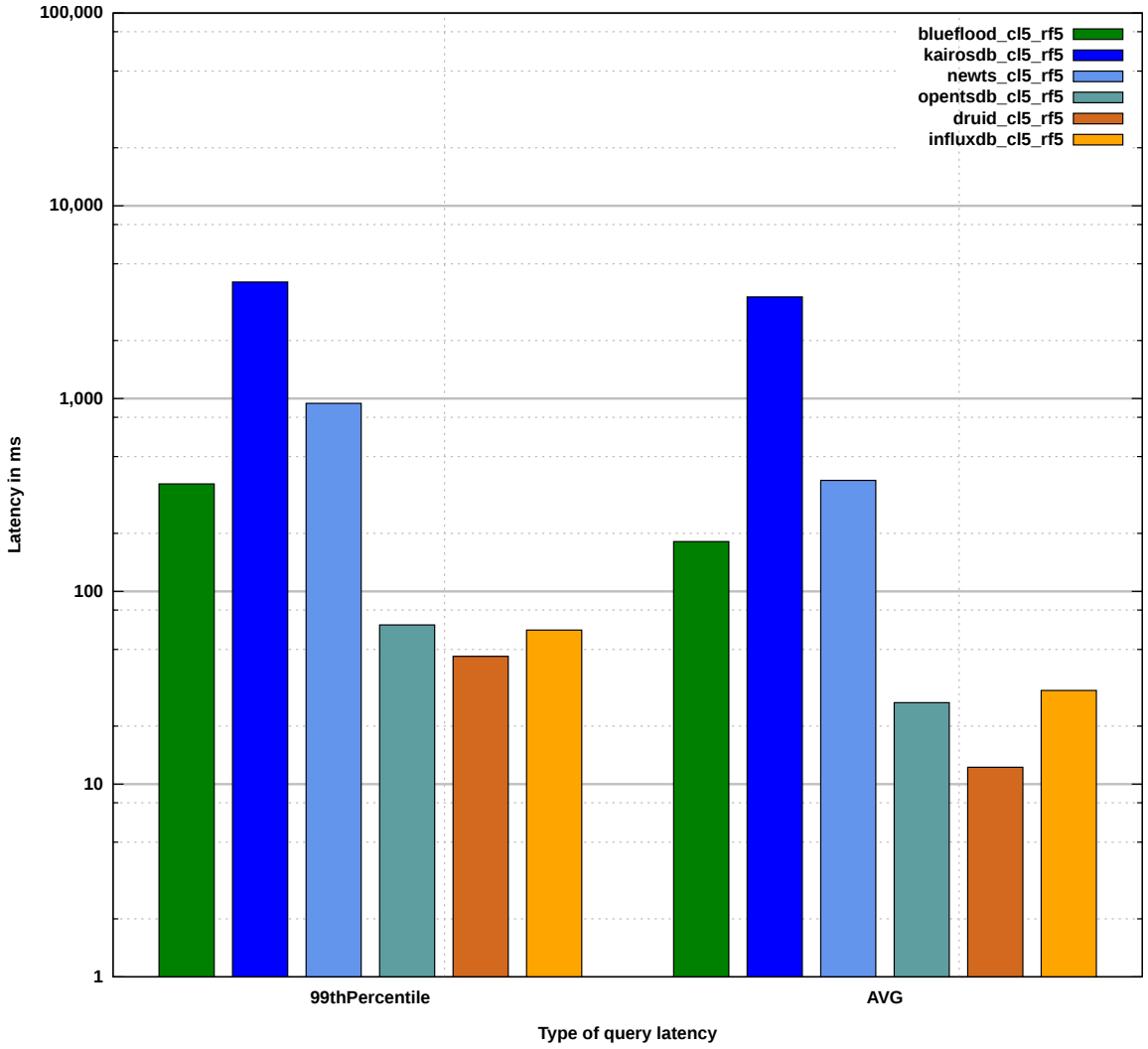

Figure 5.26: Query latency results for AVG queries in a five node cluster with a RF of five.

the second best average value has OpenTSDB. KairosDB (with Cassandra) has the worst result.

The performance comparison, presented in this chapter, is used in combination with the feature comparison from Section 3.6 to discuss the results between the ten compared TSDBs in the following chapter.

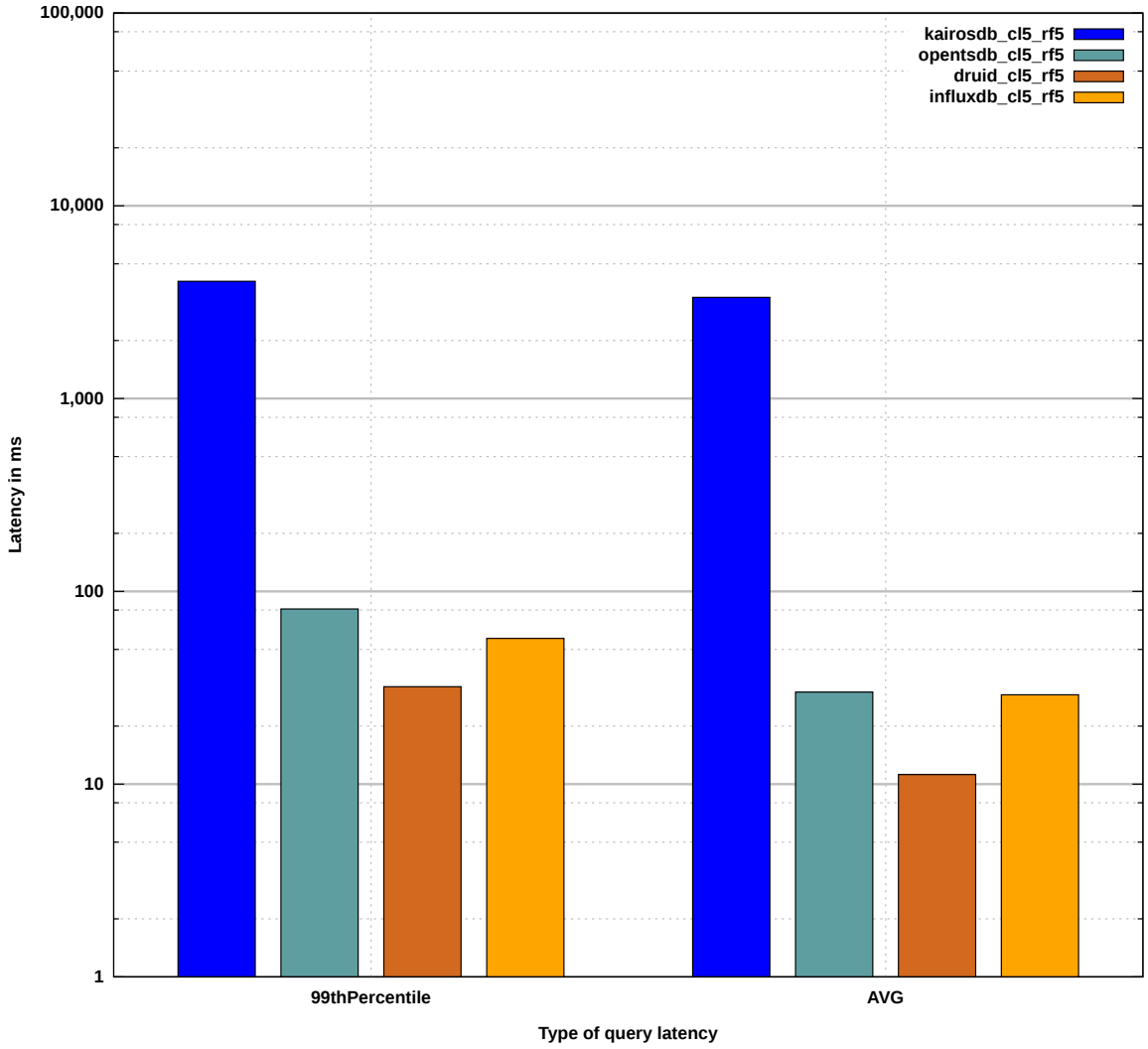

Figure 5.27: Query latency results for SUM queries in a five node cluster with a RF of five.

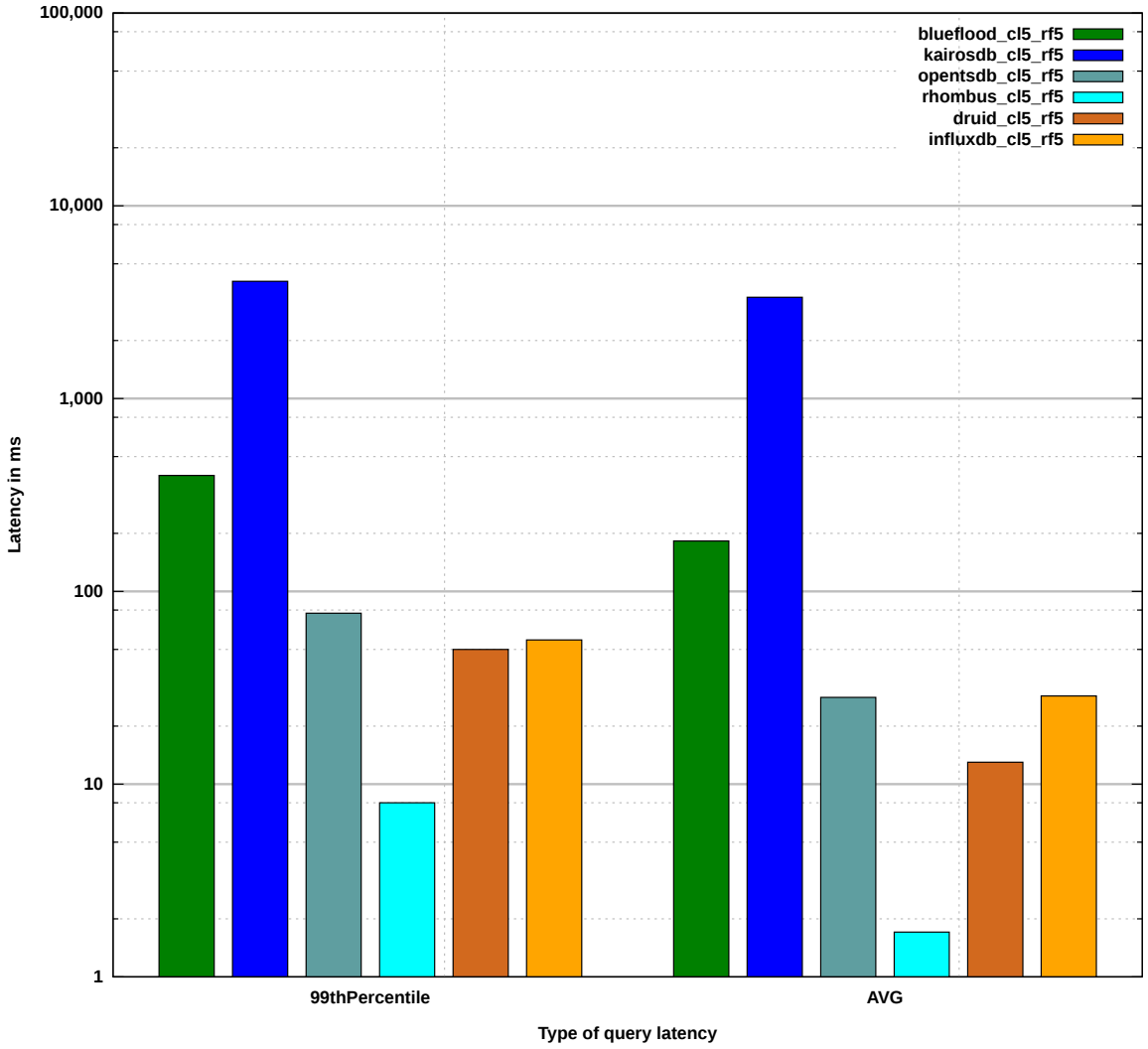

Figure 5.28: Query latency results for CNT queries in a five node cluster with a RF of five.

6 Discussion

In this chapter, the results of the feature comparison in Section 3.6 and the performance comparison in Chapter 5 are discussed. Afterward, TSDBs for different use cases are recommended.

For a better comparison of the performance results in Chapter 5, the best, second best, and third best results from each measurement type are listed in Table 6.1. If a placement is not conclusive, all qualified TSDBs are listed in the order they occur in the result description. From this table, a point based ranking is presented in Table 6.2. In this ranking, every best result from Table 6.1 receives three points, every second best result two points and every third best result one point. The remaining results of a measurement do not receive points. If a placement is not conclusive, all qualified TSDBs receive the amount of points for this placement. The points are summarized afterward for each TSDB. The ranking is grouped in four groups: 1) space consumption, 2) INS queries, 2) READ, 3) SCAN, AVG, SUM, and CNT queries, and 4) total points.

Group 1, space consumption, shows which TSDB is best in storage efficiency. Group 2, INS queries, helps in determining the TSDB that can handle INS queries with shortest latency. Group 3, READ, SCAN, AVG, SUM, and CNT queries, shows which TSDB handles non-writing queries with shortest latency. Group 4, total points, summarizes all points from the three previous groups and helps to determine an overall “best” TSDB.

Table 6.1: Overview over the three best results of each measurement in Chapter 5. CL stands for Clustersize (e. g., CL1 = Cluster with one node). If a placement is not conclusive, all qualified TSDBs are listed in the order they occur in the result description.

Measurement	TSDB with best result	TSDB with second best result	TSDB with third best result
Scenario 1: 1,000 READ Queries			
CL1, RF1, INS queries (Section 5.1.1.1)	OpenTSDB	Blueflood	Druid, Rhombus, and PostgreSQL
CL1, RF1, READ queries (Section 5.1.1.2)	Blueflood, NewTS, Rhombus, and MonetDB	InfluxDB	Druid
CL1, RF1, space consumption (Section 5.1.1.3)	Druid	OpenTSDB	InfluxDB
CL5, RF1, INS queries (Section 5.1.2.1)	OpenTSDB	Druid and Rhombus	Blueflood
CL5, RF1, READ queries (Section 5.1.2.2)	Rhombus	NewTS, Blueflood, and InfluxDB	Druid
CL5, RF1, space consumption (Section 5.1.2.3)	Druid	OpenTSDB	InfluxDB
CL5, RF2, INS queries (Section 5.1.3.1)	OpenTSDB	Druid	Blueflood
CL5, RF2, READ queries (Section 5.1.3.2)	Rhombus	NewTS	Blueflood and InfluxDB
CL5, RF2, space consumption (Section 5.1.3.3)	Druid	OpenTSDB	InfluxDB

Overview over the three best results of each measurement in Chapter 5. CL stands for Clustersize (e. g., CL1 = Cluster with one node). If a placement is not conclusive, all qualified TSDBs are listed in the order they occur in the result description. (continued)

Measurement	**TSDB with best result**	**TSDB with second best result**	**TSDB with third best result**
CL5, RF5, INS queries (Section 5.1.4)	OpenTSDB	Druid	Blueflood
CL5, RF5, READ queries (Section 5.1.4.2)	Rhombus	Blueflood	NewTS
CL5, RF5, space consumption (Section 5.1.4.3)	Druid	OpenTSDB	InfluxDB
Scenario 2: 250 SCAN, AVG, SUM, and CNT Queries			
CL1, RF1, SCAN queries (Section 5.2.1.1)	Rhombus	MonetDB	Druid
CL1, RF1, AVG queries (Section 5.2.1.2)	MonetDB	Druid	InfluxDB
CL1, RF1, SUM queries (Section 5.2.1.3)	MonetDB	Druid	OpenTSDB and InfluxDB
CL1, RF1, CNT queries (Section 5.2.1.4)	Rhombus	MonetDB	Druid
CL5, RF1, SCAN queries (Section 5.2.2.1)	Rhombus	Druid	OpenTSDB
CL5, RF1, AVG queries (Section 5.2.2.2)	Druid	InfluxDB and OpenTSDB	Blueflood

Overview over the three best results of each measurement in Chapter 5. CL stands for Clustersize (e. g., CL1 = Cluster with one node). If a placement is not conclusive, all qualified TSDBs are listed in the order they occur in the result description. (continued)

Measurement	**TSDB with best result**	**TSDB with second best result**	**TSDB with third best result**
CL5, RF1, SUM queries (Section 5.2.2.3)	Druid	InfluxDB and OpenTSDB	KairosDB[1]
CL5, RF1, CNT queries (Section 5.2.2.4)	Rhombus	Druid	OpenTSDB and InfluxDB
CL5, RF2, SCAN queries (Section 5.2.3.1)	Rhombus	Druid	OpenTSDB and InfluxDB
CL5, RF2, AVG queries (Section 5.2.3.2)	Druid	InfluxDB and OpenTSDB	Blueflood
CL5, RF2, SUM queries (Section 5.2.3.3)	Druid	InfluxDB	OpenTSDB
CL5, RF2, CNT queries (Section 5.2.3.4)	Rhombus	Druid	OpenTSDB
CL5, RF5, SCAN queries (Section 5.2.4.1)	Rhombus	Druid	OpenTSDB
CL5, RF5, AVG queries (Section 5.2.4.2)	Druid	InfluxDB and OpenTSDB	Blueflood
CL5, RF5, SUM queries (Section 5.2.4.3)	Druid	InfluxDB	OpenTSDB

[1] With Cassandra.

Overview over the three best results of each measurement in Chapter 5. CL stands for Clustersize (e. g., CL1 = Cluster with one node). If a placement is not conclusive, all qualified TSDBs are listed in the order they occur in the result description. (continued)

Measurement	**TSDB with best result**	**TSDB with second best result**	**TSDB with third best result**
CL5, RF5, CNT queries (Section 5.2.4.4)	Rhombus	Druid	InfluxDB and OpenTSDB

Table 6.2: Ranking of TSDBs with points based on Table 6.1. For each result, the TSDB with the best result receives three points, the TSDB with the second best two points, and the TSDB with the third best three points. If a placement is not conclusive, all qualified TSDBs receive the amount of points for this placement. The points are summarized afterward for each TSDB. TSDBs with zero points are omitted. Points are in descending order.

TSDB	**Points**
Space Consumption	
Druid	12
OpenTSDB	9
InfluxDB	6
INS Queries	
OpenTSDB	12
Druid	7
Blueflood	5
Rhombus	3
PostgreSQL	1
READ, SCAN, AVG, SUM, and CNT Queries	
Druid	38
Rhombus	36
InfluxDB	22
OpenTSDB	17
MonetDB	13
Blueflood	11
NewTS	8
KairosDB[2]	1
Total	
Druid	57
Rhombus	39
OpenTSDB	38
InfluxDB	28
Blueflood	16
MonetDB	13

[2]With Cassandra.

Ranking of TSDBs with points based on Table 6.1. For each result, the TSDB with the best result receives three points, the TSDB with the second best two points, and the TSDB with the third best three points. If a placement is not conclusive, all qualified TSDBs receive the amount of points for this placement. The points are summarized afterward for each TSDB. TSDBs with zero points are omitted. Points are in descending order. (continued)

TSDB	Points
NewTS	8
KairosDB[3]	1
PostgreSQL	1

From Table 6.2 it is now possible to give recommendations based on the performance comparison:

1. If the focus lies on lowest space consumption or non-writing queries like READ, SCAN, AVG, SUM, and CNT queries, Druid is the best choice, but it is only second best choice for INS queries.
2. If the focus lies on INS queries, OpenTSDB is the best choice, but is only second best in lowest space consumption and fourth best in non-writing queries like READ, SCAN, AVG, SUM, and CNT queries.
3. If there is no specific focus, Druid is the best choice, followed by Rhombus and OpenTSDB.

It must be considered that TSDBs that were only used in one node setups like MySQL, MonetDB, PostgreSQL and KairosDB with H2, are disadvantaged because they cannot get as many points as other TSDBs that can score in all measurements. This also means that they cannot get maximum possible points. In addition to that, TSDBs that miss some functions (e. g., Rhombus) also cannot achieve maximum points. When looking at the “total” category, it must also be known that TSDBs that perform well in non-writing queries like READ, SCAN, AVG, SUM, and CNT queries can achieve more points than TSDBs that perform well in INS queries or in space consumption.

Combining these results with the feature comparison of Section 3.6, the following recommendations can be given:

[3]With Cassandra.

1. If fulfillment of all criteria of groups 1 (see Section 3.2) and 2 (see Section 3.3), a granularity of 1 ms, but no commercial support, no stable/LTS version is required, and the performance focus lies on lowest space consumption, non-writing queries, or there is no focus at all, Druid is the best choice.
2. If all criteria, commercial support, no stable/LTS version, no existing client libraries from Section 3.6 are required, and experimental features are tolerated, and the performance criteria of Item 1 are required, InfluxDB is the best choice.
3. If not all criteria from Section 3.6 are required and the focus lies on the performance of INS queries, OpenTSDB is the best choice.
4. If not all criteria from Section 3.6, but a stable/LTS version is required, then one of the two RDBMS are the best choice.
5. If not all criteria, but many client libraries and distribution as well as clustering features are required, MonetDB is the best choice if the restrictions in clustering and distribution can be tolerated.

It can be concluded that for the comparison in this thesis, Druid is the best choice, if no commercial support, no stable/LTS version, not many client libraries, and not the best performance of INS queries are required. When one of these requirements must be fulfilled, other choices are better suited, as shown in the list above. It must be considered that this list only shows a few use cases, but a selection can depend on other factors that must be chosen from the comparisons in Section 3.6 and in Chapter 5.

After discussing the results of the feature and the performance comparison, the following chapter completes the thesis with a conclusion and points out problems and improvement potentials for future work.

7 Conclusion and Outlook

In Chapter 2, benchmark results and benchmarks for DBMS related to time series data, as well as possible data sets for benchmarks are presented. In the subsequent Chapter 3, after defining the search process that resulted in 75 found TSDBs, the 42 open source TSDBs are grouped into four groups and representatives of each are chosen by popularity and presented more detailed. The ten chosen representatives are compared in 17 criteria (grouped in five criteria groups). The feature comparison concluded with Druid being the best choice, if all criteria besides having a stable/LTS version and commercial support, must be fulfilled. Other TSDBs like OpenTSDB or MonetDB can be a better choice if stable/LTS versions or commercial support are required.

For the performance part of the comparison, a benchmark is required. Metrics and two scenarios for a performance comparison, as well as requirements on a benchmark are presented in Chapter 4. Both scenarios use 1,000,000 INS queries to create a initial set of data inside each TSDB. The first scenario is used to measure the query latency and space consumption of the initial 1,000,000 INS queries and 1,000 READ queries afterward. The second scenario measures the query latency and space consumption of 250 SCAN, AVG, SUM, and CNT queries. Afterward, an extensible benchmark for TSDBs with variable workloads, is presented. It is called TSDBBench and uses YCSB-TS, which is a extension of YCSB for running and measuring workloads on TSDBs.

In Chapter 5, the measurement results of the two scenarios defined in Chapter 4 are presented. There are four measurement setups used for each of the two scenarios used: a cluster consisting of one node with a RF of one, a cluster consisting of five nodes with RFs of one, three and five.

The conclusion in Chapter 6 from the measurement results are, that if there is focus on lowest space consumption or non-writing queries like READ, SCAN, AVG, SUM, and CNT queries, Druid is the best choice, but it is only second best choice for INS queries. If the focus lies on INS queries, OpenTSDB is the best choice with the shortest average INS query latency, which is approximately 2.5 times faster than the other TSDBs. It is is only second best in lowest space consumption and fourth best in non-writing

queries. The chapter concludes with a combined result, consisting of the feature comparison and the performance comparison. It suggests Druid as an overall solution if no commercial support, no stable/LTS version is required, and the performance focus lies on lowest space consumption, non-writing queries, or there is no focus at all. If one of the aspects is required, then other TSDBs might be better suited.

In this thesis, TSDBBench is presented and a performance comparison is done for getting an overview over the performance of the most popular TSDBs. As the performance comparison has only an "overview" character, there are several points that give possibilities for future work.

The two scenarios used are very general and unspecific. More fitted workloads are required if results for real-world scenarios must be given. The size of the data used in this thesis is not very big (1,000,000 rows of data) and therefore measurements with much more data should be done. More data could give results which show that some TSDBs are more suitable for "realworld" scenarios that use great amounts of data.

For the performance comparison in this thesis, no statistic methods other than integrated into YCSB are used. As Kalibera and Jones [KJ13] points out, statistic methods, like confidence intervals or ANOVA tests, should be used to create more qualified results. This however requires many repeating benchmark runs, which use a lot of time. Taking the two scenarios from this thesis, with a runtime of approximately two days for each[1], as a base and, for example, 1,000 repetitions, a runtime of approximately 11 years would be required to complete all repetitions for both scenarios. There are thinkable solutions for faster benchmark runs when using an EI with higher performance or multiple EIs. Parallel benchmarking on the same EI is possible, but it should be shown this has no influence on the results first.

Due to limited time, it was not possible to tune every TSDB to its maximum performance for the two scenarios. Obvious tips were used, but no fine tuning was done. With fine tuning other results may be possible. When TSDBBench is released to public, it is expected that the integrated TSDB settings are getting fine tuned over the time for better results, as developers and more experienced users can add their knowledge. In addition to that, fine tuning of the VM settings can also be done. For this thesis, reasonable settings, based in experience were used. As van Zanten [Zan12] and Lim et al. [LHY+13] show, even the correct amount of vCPUs can be problematic. On the other hand, no TSDBs should benefit more than others from a specific setting.

[1] Appendix A.5 lists the hardware that is used in this thesis.

For this thesis, ten TSDBs were added to TSDBBench. From the 75 TSDBs found in this thesis, 42 are open source and could be integrated for measuring a wider field of TSDBs. The remaining part consists of commercial solutions, which could also be integrated. Integrating TSDBs that do not run under linux or must run outside of an EI (e. g., a remote site) is not considered yet and could be developed. Support for additional EIs is also possible for reaching a wider field of users. The integration of provisioning tools like Chef or Puppet was not necessary yet but can be necessary for future TSDB if the provisioning features of vagrant are not sufficient.

YCSB-TS measures query latencies for determining the performance of a TSDB. As support for multiple generator VM is integrated in TSDBBench, it would be interesting to use multiple VMs to measure the saturation point of a TSDB. This means that the rate of parallel queries is increased until the TSDB cannot handle the queries anymore. Such a scenario could help to get more insights in the maximum performance of the benchmarked TSDBs.

Operational Data Historians (ODHs) is a new type of DBMS that combines TSDBs and RDBMS features for avoiding copies from TSDBs to RDBMS that are necessary when data must be analyzed. It could be possible to integrate ODHs into TSDBBench for comparisons between TSDBs and ODHs.

A Appendix

A.1 List of TSDBs

Section 3.1 presented the search process for TSDBs. In the following, the 75 identified TSDBs are listed.

A.1.1 Requirement on NoSQL DBMS

- Apache Chukwa [ASF15b]
- Arctic [Man15]
- BlueFlood [Rac15a]
- Cube [Squ15]
- Energy Databus [Buf15]
- Gorilla[1] [PFT+15]
- Kairos [Ago15]
- KairosDB [Kai15b]
- Newts [Eva15]
- OpenTSDB [LSKD15a]
- Rhombus [Par15b]
- SiteWhere [Sit15]
- Sky[2] [Joh15]
- tsdb: A Compressed Database for Time Series[3] [DMF12]

A.1.2 Requirement on NoSQL DBMS

- Akumuli [Laz15]

[1]No source code is available, but since a describing paper is available, it is considered open source.

[2]Development stopped, dependencies show that it uses Leveldb.

[3]Uses BerkleyDB, which is not a typical key-value database, but stores key-value data.

- Atlas [Net15]
- BEMOSS[4] [Rah15]
- BlinkDB [AIP+12; AMP+13]
- Bolt [Mic15d]
- DalmatinerDB [Pro15a]
- Druid[5] [YTL+14]
- Gnocchi [Ope15a]
- GridMW [YKP+11]
- InfluxDB [Inf15b]
- MonetDB [Mon15a]
- NilmDB [PDL14]
- Node-tsdb [Wan15]
- Prometheus [Pro15b]
- RRDtool [Oet15]
- SciDB [Par15a]
- Seriesly [Sal15a]
- TSDB: Time Series Database [Dug15]
- TimeStore [Sti15]
- TsTables[6] [Fie15b; Hil15]
- Vaultaire [Anc15]
- Whisper (Graphite) [GP15]
- YAWNDB [Gro15]

A.1.3 Proprietary

- Acunu[7] [Acu15]
- Aerospike [Aer15]
- Axibase Time Series Database [Axi15]
- Cityzen Data [Cit15]
- DataStax Enterprise [Dat15]
- Databus[8] [Lin15]

[4]Uses an internal simple Measurement and Actuation Profile (sMAP) [Daw15] server for storing time series data.

[5]Can use other databases for some parts of its nodes, but still considered a stand-alone solution.

[6]Formerly “Time Series Database” [Fie15a].

[7]Does not exist anymore because it was bought by Apple Inc. around March, 2015 [Cam15].

[8]Open source, but uses Oracle Database.

- eXtremeDB Financial Edition [McO15]
- FoundationDB[9] [Fou15]
- Geras [12415]
- IBM Informix [IBM15]
- kdb+ [Kx15]
- Kerf [Ker15]
- Mesap [Sev15]
- Microsoft SQL Server [Mic15b]
- New Relic Insights [New15]
- ONETick Time-Series Tick Database [One15]
- OSIsoft PI [OSI15]
- Oracle Database [Ora15i]
- ParStream[10] [Cis15]
- Polyhedra IMDB [Ene15]
- Prognoz Platform [JSC15]
- PulsarTSDB [DOT15]
- Riak TS [Bas15]
- SAP HANA [SAP15]
- SkySpark [Sky15]
- Splunk [BGSZ10; Spl15]
- Sqrrl [Sqr15]
- Squwk[11] [Ser15]
- TempoIQ[12] [Tem15]
- TimeScape EDM+[13] [Xen15]
- TimeSeries.Guru [WW15]
- Treasure Data [Tre15]
- Uniformance Process History Database (PHD) [Hon15]

A.1.4 RDBMS

- MySQL Cluster [Ora15e]
- MySQL Community Server [Ora15f]

[9] Does not exist anymore because it was bought by Apple Inc. around March, 2015 [Pan15].

[10] Bought by Cisco Systems, Inc. around October, 2015 [Sal15b].

[11] Squwk (http://www.squwk.com) does not exist anymore. Previously named Aspen.

[12] Formerly TempoDB.

[13] Contains TimeScape XDB for handling time series data.

- PostgreSQL TS[14] [Ber14]
- PostgreSQL [PGD15a]
- TimeTravel[15] [KFPL12]

We also tried to include *SensorDB*, but there is nothing other than a Python API to be found. The RDBMS group contains only the two actually used ones, there exist several more RDBMS.

A.2 Time series data in RDBMS

For storing time series data in RDBMS, a schema consisting of 13 fields[16] was used. The first field is an automatically incrementing ID that is used as primary key (see Section 3.4 for a more detailed explanation). Following the primary key, there are a timestamp, a floating point[17] value, and ten tags. Every metric is represented as one table. See listings Listing A.1, Listing A.2, Listing A.3, and Listing A.4 for the different implementations in SQL, MySQL Community Server, PostgreSQL and MonetDB.

Listing A.1 Used schema for time series data in SQL

```
CREATE TABLE usermetric(
  ID INT IDENTITY(1,1) PRIMARY KEY,
  YCSB_KEY TIMESTAMP,
  VALUE DOUBLE PRECISION,
  TAG0 VARCHAR, TAG1 VARCHAR,
  TAG2 VARCHAR, TAG3 VARCHAR,
  TAG4 VARCHAR, TAG5 VARCHAR,
  TAG6 VARCHAR, TAG7 VARCHAR,
  TAG8 VARCHAR, TAG9 VARCHAR);
```

when querying for data, the following queries are used:

For INS the following query is used:
`INSERT INTO usermetric (YCSB_KEY ,VALUE[,<Tags>]) VALUES (<Timestamp1>,<Value>[,<Tag Values>]);.`

[14] Default PostgreSQL with the timeseries approach of Robert Berry.

[15] No other sources than the paper available.

[16] The amount of tags is limited to ten values for this example. If a scenario requires more tags, additional fields must be defined.

[17] With double precision.

Listing A.2 Used schema for time series data in MySQL Community Server

```
CREATE TABLE test.usermetric(
  ID INT AUTO_INCREMENT PRIMARY KEY,
  YCSB_KEY TIMESTAMP,
  VALUE DOUBLE,
  TAG0 TEXT, TAG1 TEXT,
  TAG2 TEXT, TAG3 TEXT,
  TAG4 TEXT, TAG5 TEXT,
  TAG6 TEXT, TAG7 TEXT,
  TAG8 TEXT, TAG9 TEXT);
```

Listing A.3 Used schema for time series data in PostgreSQL

```
CREATE TABLE usermetric(
  ID SERIAL PRIMARY KEY,
  YCSB_KEY TIMESTAMP,
  VALUE DOUBLE PRECISION,
  TAG0 TEXT, TAG1 TEXT,
  TAG2 TEXT, TAG3 TEXT,
  TAG4 TEXT, TAG5 TEXT,
  TAG6 TEXT, TAG7 TEXT,
  TAG8 TEXT, TAG9 TEXT);
```

Listing A.4 Used schema for time series data in MonetDB

```
CREATE TABLE test.usermetric(
  ID INT AUTO_INCREMENT PRIMARY KEY,
  YCSB_KEY TIMESTAMP,
  VALUE DOUBLE,
  TAG0 TEXT, TAG1 TEXT,
  TAG2 TEXT, TAG3 TEXT,
  TAG4 TEXT, TAG5 TEXT,
  TAG6 TEXT, TAG7 TEXT,
  TAG8 TEXT, TAG9 TEXT);
```

For READ the following query is used:
`SELECT * FROM usermetric WHERE [<TagQuery>](YCSB_KEY = <Timestamp1>);.`

For SCAN the following query is used:
`SELECT * FROM usermetric WHERE [<TagQuery>] YCSB_KEY BETWEEN <Timestamp1> AND <Timestamp2>;.`

For AVG the following query is used:
`SELECT AVG(VALUE) as VALUE FROM usermetric WHERE [<TagQuery>] YCSB_KEY BETWEEN <Timestamp1> AND <Timestamp2> GROUP BY YCSB_KEY;.`

For CNT the following query is used:
`SELECT COUNT(*) as VALUE FROM usermetric WHERE [<TagQuery>] YCSB_KEY BETWEEN <Timestamp1> AND <Timestamp2> GROUP BY YCSB_KEY;.`

For SUM the following query is used:
`SELECT SUM(VALUE) as VALUE FROM usermetric WHERE [<TagQuery>] YCSB_KEY BETWEEN <Timestamp1> AND <Timestamp2> GROUP BY YCSB_KEY;.`

Parts of a query in `[]` are optional and parts in `<>` are variables that must be filled. `<Timestamp1>` and `<Timestamp2>` are different timestamps. `<Value>` is the actual value that is inserted. Some values are strings (e. g., tag values) and must be inserted with `""` which are omitted here.

When tags are inserted, `<Tags>` is defined as `<Tag1>, <Tag2>, <Tag3>, ...` and `<Tag Values>` as `<TagValue1>, <TagValue2>, <TagValue3>, ....` `<TagN>` is the name of a tag, `<TagValueN>` is the value of tag, `N` is a number.

When querying for tags, `<TagQuery>` is defined like this: `((<TagOrQuery1>) AND (<TagOrQuery2>) AND (<TagOrQuery3>) ...) AND. <TagOrQueryN>`, `N` is a number, is defined as follows: `<Tag1> = <Tag1Value1> OR <Tag1> = <Tag1Value2> OR <Tag1> = <Tag1Value3> ...` where `<TagMValueN>` (`M` and `N` are numbers) is a tag value.

A.3 Google search for popularity ranking of TSDBs

For ranking the found open source TSDBs from Appendix A.1 by popularity, a Google search for each open source TSDB was performed. The TSDB name in combination with "time series" was used as search string, e. g., "`"OpenTSDB" "time series"`". A TSDB with more results was considered more popular than one with lesser results. Some TSDBs names are very common terms like "`Cube`", "`Whisper`", "`Gorilla`", "`Arctic`" or "`TimeTravel`". Some of these terms needed "`time series database`" as additional term because "`time series`" as additional term found many results unrelated to TSDBs (e. g., "`Arctic`" in combination with "`time series`" finds many arctic ice time series). "`Kairos`" and "`KairosDB`" are also problematic, as the first one is often used as a short form for the latter one. PostgreSQL TS was not searched because it is no official name. It was tried to fit the results as close as possible to only match entries that are related to the TSDBs, but it is expected that the results are not exactly precise.

The American version of Google, http://www.google.com, was used for each search. The search was performed December 14, 2015 between 01:10 and 01:38 PM UTC+1. A typical search URL looked like this: https://www.google.com/search?gl=us&hl=en&btnG=Google+Search&pws=0&q=%22OpenTSDB%22+%22time+series%22&inurl=https&gws_rd=cr%2Cssl. See table Table A.1 for the results.

Table A.1: A list of http://www.google.com results for each TSDB. TSDBs used in YCSB-TS are highlighted.

TSDB	Search string	Results
GridMW	`"GridMW" "time series"`	7
NilmDB	`"NilmDB" "time series"`	9
Node-tsdb	`"node-tsdb" "time series"`	34
BEMOSS	`"BEMOSS" "time series"`	100
DalmatinerDB	`"DalmatinerDB" "time series"`	103
Bolt	`"Bolt" "time series" "Data Management for Connected Homes"`	169
Cube	`"Square, Inc." "Cube" "time series"`	191
Tsdb: A Compressed Database for Time Series	`"Tsdb" "A Compressed Database for Time Series" "time series"`	195

A list of http://www.google.com results for each TSDB. TSDBs used in YCSB-TS are highlighted. (continued)

TSDB	Search string	Results
Energy Databus	`"Energy Databus" "time series"`	239
SiteWhere	`"SiteWhere" "time series"`	274
SkyDB	`"SkyDB" "time series"`	288
Apache Chukwa	`"Apache Chukwa" "time series"`	336
Kairos	`"Kairos" "time series" -"KairosDB" "redis" "agoragames"`	361
TimeStore	`"TimeStore" "time series"`	547
Akumuli	`"Akumuli" "time series"`	558
YAWNDB	`"YAWNDB" "time series"`	563
TimeTravel	`"TimeTravel" "dbms" "time series"`	687
Arctic	`"Arctic" "time series database" -"ice"`	922
Vaultaire	`"Vaultaire" "time series"`	971
Gorilla	`"Gorilla" "time series database" -"pound"`	1,380
TSDB: Time Series Database	`"TSDB" "Time Series Database" "time series" -"OpenTSDB"`	1,430
TsTables	`"TsTables" "time series"`	1,480
BlueFlood[18]	`"BlueFlood" "time series"`	1,630
BlinkDB	`"BlinkDB" "time series"`	1,810
Seriesly	`"Seriesly" "time series"`	1,880
SciDB	`"SciDB" "time series"`	3,940
Gnocchi	`"Gnocchi" "time series"`	4,260
Whisper	`"Whisper" "graphite" "time series"`	5,060

[18]The reason for using this TSDB for comparison is explained in Section 3.1.

A list of http://www.google.com results for each TSDB. TSDBs used in YCSB-TS are highlighted. (continued)

TSDB	Search string	Results
Atlas	`"Atlas" "time series database"`	6,140
KairosDB	`"Kairos" "time series" -"redis" -"agoragames"`	6,310
Newts	`"Newts" "time series"`	7,330
MonetDB	`"MonetDB" "time series"`	8,270
Rhombus	`"Rhombus" "time series"`	11,500
OpenTSDB	`"OpenTSDB" "time series"`	12,800
MySQL Cluster[19]	`"MySQL Cluster" "time series"`	20,500
Druid	`"Druid" "time series"`	24,000
InfluxDB	`"InfluxDB" "time series"`	24,000
RRDtool[19]	`"RRDtool" "time series"`	26,500
Prometheus[19]	`"Prometheus" "time series" -"IMDB" -"Movie"`	32,000
PostgreSQL	`"PostgreSQL" "time series"`	113,000
MySQL Community Server	`"MySQL"`[20] `"time series"`	289,000

A.4 Python Modules and Tools

A list of used Java client libraries and TSDBs on Debian 8.1 x64 “Jessie”. All Java applications used OpenJDK Java 7 JRE with version 7u91-2.6.3-1d̃eb8u1.

A.4.1 TSDBs

- Blueflood 2.0.0-SNAPSHOT with Cassandra 2.1.12

[19] The reason for not using this TSDB for comparison is explained in Section 3.1.

[20] "MySQL Server" has 57,400 results. As the term "MySQL Community Server" is very unpopular und not used (only 923 results), `"MySQL"` was used.

- Druid 0.8.1 with MySQL Server 5.5.46-0+deb8u1
- InfluxDB 0.10.0-nightly-72c6a51
- KairosDB 1.0.0-1 with Cassandra 2.1.12
- KairosDB 1.0.0-1 with h2 1.3.170
- MonetDB Server 11.21.11
- MySQL Server 5.5.46-0+deb8u1
- NewTS 1.3.1 with Cassandra 2.1.12
- OpenTSDB 2.2.0RC1 with HBase 1.1.2 and Hadoop 2.7.1
- PostgreSQL 9.4.5-0+deb8u1

A.4.2 Java client libraries

- httpclient 4.5 and json 20140107 for Blueflood and OpenTSDB
- tranquility_2.10 0.5.1, httpclient 4.5, and json 20140107 for Druid
- influxdb-java 2.0-GIT for InfluxDB
- monetdb-jdbc 2.18 for MonetDB
- client 2.0 for KairosDB
- newts-cassandra 1.3.1 and newts-cassandra-search 1.3.1 for NewTS
- rhombus 2.0.12-SNAPSHOT for Rhombud
- libmysql-java 5.1.32-1 for MySQL
- libpostgresql-jdbc-java 9.2-1002-1 for PostgreSQL

A.5 Hardware and VM Settings

TSDBBench used vSphere as an EI, which has three Supermicro SuperServer SYS-6027R-TRF servers, of which each has the following hardware:

- Six Intel Xeon E5-2640 CPUs with six physical cores, each running with 2.5 GHz. Twelve threads are available with Hyperthreading.
- 192 gigabyte RAM, consisting of 24 8 Gigabyte ECC registered RAM modules (RAM-R31608ER)
- Two EMU-OCE11102-NX network adapters (copper)
- One 8 gigabyte SATA Embedded Module (DOM) with SLC Flash (ATP-AF8GSSEI)

VSphere used the following hardware as storage:

- Two Intel Xeon E5-2609 CPUs with four physical cores, each running with 2.4 GHz.

- 16 gigabyte RAM, consisting of eight 2 gigabyte ECC registered RAM modules (RAM-R31602ER)
- Open-E DSS V7 with Open-E Active-Active

The storage uses several storage modules, each of them uses the following hardware:

- 4-channel SAS 6 GBit/s Dual Core RAID-Controller (LSI-SAS9271-4I-SG)
- Four internal SATA+SAS ports
- LSI SAS2208 Dual Core ROC, 800 MHZ PowerPC
- 1 gigabyte DDR3 Cache
- LSI CVM01 Cachevault Kit FA1/4R for MEGARAID SAS 9266 series
- Two Intel 520 SSDs with 120 GB MLC flash each (SSD-INT-SC2CW120A310)
- 22 1 terabyte HDDs with 7,200 RPM and 64 megabyte cache each (HDD-25S21000-07)
- Two dual port 10 gigabit network adapters (EMU-OCE11102-NT)
- One 8-channel 6 SATA+SAS host bus adapter (LSI-SAS9207-8E-SGL)

The following settings are used for all TSDB VMs:

- Eight vCPUs with maximum frequency set to “unlimited”.
- 16,384 megabyte RAM
- 50 gigabyte HDD
- Debian 8.1 x64 “Jessie” as operating system

The following settings are used for all generator VMs:

- Four vCPUs with maximum frequency set to “unlimited”.
- 8,192 megabyte RAM
- 50 gigabyte HDD
- Debian 8.1 x64 “Jessie” as operating system

A.6 Supported functions by the compared TSDBs

Table A.2 presents a comparison of supported functions of the ten compared TSDBs in this thesis, see Section 4.2 for more details.

TSDB	INS	READ	SCAN	AVG	SUM	CNT	MAX	MIN	UP-DATE	DEL
Group 1: TSDBs with a requirement on NoSQL DBMS										
Blueflood	x	x[21]	x	x		x	x	x		
KairosDB	x	x[21]	x	x	x	x	x	x		x
NewTS[24]	x	x	x	x			x	x		
OpenTSDB[22]	x	x[21]	x	x	x	x[23]	x	x		
Rhombus[24]	x	x	x			x			x	x
Group 2: TSDBs with no requirement on any DBMS										
Druid[25]	x	x[21]	x	x[26]	x	x	x	x		
InfluxDB	x	x	x	x	x	x	x	x	x[27]	x
MonetDB	x	x	x	x	x	x	x	x	x	x
Group 3: RDBMS										
MySQL Community Server	x	x	x	x	x	x	x	x	x	x
PostgreSQL	x	x	x	x	x	x	x	x	x	x

Table A.2: Comparison of supported functions by the compared TSDBs in this thesis.

[21] Uses SCAN with the smallest possible range.

[22] DEL can be substituted with a command-line tool and HBase functions.

[23] Since OpenTSDB version 2.2.

[24] Some unsupported functions can be substituted with own CQL statements.

[25] Missing functions can be substituted with re-ingestion through a IngestSegmentFirehose.

[26] Uses SCAN and CNT.

[27] Possible to overwrite with INS when setting resolution to milliseconds, see [Sha14].

List of Abbreviations

Abbreviation	Meaning	First occurrence
UPDATE	Updating	14
API	Application Programming Interface	14
AVG	Averaging	14
AWS	Amazon Web Services	12
CLI	Command-Line Interface	28
CNT	Counting	14
CPS	Cyber-Physical Systems	11
CQL	Cassandra Query Language	31
CRUD	Create, Read, Update, Delete	56
DB	Database	12
DBMS	Database Management System	3
DBS	Database System	12
DEL	Deletion	15
DI	Data Integration	22
DNS	Domain Name System	30
EC2	Elastic Compute Cloud	12
EI	Elastic Infrastructure	3
ETL	Extract, Transform, Load	22
FCS	Frame Check Sequence	3
GPLv2	GNU General Public License version 2	42
GPLv3	GNU General Public License version 3	42
GUI	Graphical User Interface	28
HA	High Availability	13
HDD	Hard Disk Drive	18
InfluxQL	InfluxDB Query Language	32
INS	Insertion	14
IoT	Internet of Things	11
JDBC	Java Database Connectivity	33

(continued)

Abbreviation	Meaning	First occurrence
LGPLv2.1	GNU Lesser General Public License version 2.1	42
LTS	Long Term Support	14
MAL	MonetDB Assembly Language	33
MAX	Maximization	14
MIN	Minimization	14
MIT	Massachusetts Institute of Technology	42
MPL	Mozilla Public License	43
ODBC	Open Database Connectivity	33
ODH	Operational Data Historian	19
OLAP	Online Analytical Processing	23
OLTP	Online Transaction Processing	22
PEC	Peer Energy Cloud	20
RDBMS	Relational Database Management System	3
READ	Reading	14
RF	Replication Factor	29
RRD	Round Robin Database	27
SCAN	Scanning	14
sMAP	simple Measurement and Actuation Profile	126
SOA	Service-oriented Architecture	18
SQL	Structured Query Language	32
SSD	Solid State Disk	18
SUM	Summarization	14
TPC	Transaction Processing Performance Council	22
TSDB	Time Series Database	3
UI	User Interface	28
VM	Virtual Machine	18
YCSB	Yahoo Cloud Server Benchmark	3
YCSB-TS	Yahoo Cloud Server Benchmark for Time Series	3

Bibliography

[12415] 1248 Ltd. *Geras*. 2015. URL: http://1248.io/geras.php (cit. on p. 127).

[Aba12] D. Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story." In: *Computer* 45.2 (Feb. 2012), pp. 37–42 (cit. on p. 12).

[ABF14] V. Abramova, J. Bernardino, P. Furtado. "Testing Cloud Benchmark Scalability with Cassandra." In: *Services (SERVICES), 2014 IEEE World Congress on* (June 2014), pp. 434–441 (cit. on p. 19).

[Act15] Actian Corporation. *Vectorwise*. 2015. URL: www.actian.com/vectorwise (cit. on p. 20).

[Acu15] Acunu. *Acunu*. 2015. URL: http://www.acunu.com/ (cit. on p. 126).

[Aer15] Aerospike, Inc. *Aerospike*. 2015. URL: http://www.aerospike.com/ (cit. on pp. 18, 126).

[Ago15] Agora Games, LLC. *Kairos*. 2015. URL: https://github.com/agoragames/kairos (cit. on p. 125).

[AIP+12] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, I. Stoica. "Blink and It's Done: Interactive Queries on Very Large Data." In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1902–1905 (cit. on p. 126).

[AMP+13] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, I. Stoica. "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 29–42 (cit. on p. 126).

[Ana15] C. Analytics. *Bokeh*. 2015. URL: http://bokeh.pydata.org/ (cit. on p. 58).

[Anc15] Anchor Cloud Hosting. *Vaultaire*. 2015. URL: https://github.com/anchor/vaultaire; http://www.anchor.com.au/blog/2014/06/vaultaire-ceph-based-immutable-tsdb/ (cit. on p. 126).

[Ant15] Antonmks. *Alenka*. 2015. URL: https://github.com/antonmks/Alenka (cit. on p. 20).

[ASF04] The Apache Software Foundation. *Apache License, Version 2.0*. Jan. 2004. URL: http://www.apache.org/licenses/LICENSE-2.0 (cit. on p. 42).

[ASF15a] The Apache Software Foundation. *Apache Cassandra*. 2015. URL: http://cassandra.apache.org/ (cit. on pp. 18, 28).

[ASF15b] The Apache Software Foundation. *Apache Chukwa*. 2015. URL: https://chukwa.apache.org/ (cit. on p. 125).

[ASF15c] The Apache Software Foundation. *Apache Hadoop*. 2015. URL: https://hadoop.apache.org/ (cit. on p. 19).

[ASF15d] The Apache Software Foundation. *Apache HBase*. 2015. URL: https://hbase.apache.org/ (cit. on pp. 18, 28).

[ASF15e] The Apache Software Foundation. *Apache Kafka*. 2015. URL: http://kafka.apache.org/ (cit. on p. 29).

[ASF15f] The Apache Software Foundation. *Apache Log4j 2*. 2015. URL: http://logging.apache.org/log4j/2.x/ (cit. on p. 28).

[ASF15g] The Apache Software Foundation. *Apache Samza*. 2015. URL: http://samza.apache.org/ (cit. on p. 32).

[ASF15h] The Apache Software Foundation. *Apache Storm*. 2015. URL: http://storm.apache.org/ (cit. on p. 32).

[ASF15i] The Apache Software Foundation. *Apache ZooKeeper*. 2015. URL: https://zookeeper.apache.org/ (cit. on p. 28).

[AWS15] Amazon Web Services, Inc. *Amazon Simple Storage Service (Amazon S3)*. 2015. URL: https://aws.amazon.com/de/s3/ (cit. on pp. 31, 56).

[Axi15] Axibase. *Axibase Time Series Database*. 2015. URL: https://axibase.com/products/axibase-time-series-database/ (cit. on p. 126).

[Bas15] Basho Technologies, Inc. *Riak TS*. 2015. URL: http://basho.com/products/riak-ts/ (cit. on p. 127).

[BDM07] V. Budhraja, J. Dyer, C. Morales. *Real-time performance monitoring and management system*. US Patent 7,233,843. June 2007 (cit. on p. 12).

[Bec15a] S. Beckett. *InfluxDB - archive / rollup / precision tuning feature*. 2015. URL: https://github.com/influxdb/influxdb/issues/1884 (cit. on p. 38).

[Bec15b] S. Beckett. *InfluxDB - Improvement to error message for GROUP BY with too many points*. 2015. URL: https://github.com/influxdb/influxdb/issues/2702 (cit. on pp. 32, 40).

[Ber14] R. Berry. *Querying Time Series in Postgresql*. May 2014. URL: http://no0p.github.io/postgresql/2014/05/08/timeseries-tips-pg.html (cit. on p. 128).

[BF12] S. Blumsack, A. Fernandez. "Ready or not, here comes the smart grid!" In: *Energy* 37.1 (2012). 7th Biennial International Workshop "Advances in Energy Studies", pp. 61–68 (cit. on p. 11).

[BGG+13] C. Busemann, V. Gazis, R. Gold, P. Kikiras, A. Leonardi, J. Mirkovic, M. Walther, H. Ziekow. "Integrating Sensor Networks for Energy Monitoring with Service-Oriented Architectures." In: *International Journal of Distributed Sensor Networks* 2013 (2013) (cit. on p. 18).

[BGSZ10] L. Bitincka, A. Ganapathi, S. Sorkin, S. Zhang. "Optimizing Data Analysis with a Semi-structured Time Series Database." In: *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*. SLAML'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 7–7 (cit. on p. 127).

[BKF15] A. Bader, O. Kopp, M. Falkenthal. *Meeting on 2015-11-12, 2015*. 2015 (cit. on p. 27).

[Bor15] D. Borthakur. *HDFS Architecture Guide*. 2015. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (cit. on p. 22).

[BPV14] H. Bauer, M. Patel, J. Veira. *The Internet of Things: Sizing up the opportunity*. McKinsey & Company. Dec. 2014. URL: http://www.mckinsey.com/insights/high_tech_telecoms_internet/the_internet_of_things_sizing_up_the_opportunity (cit. on p. 11).

[Buf15] Buffalo Software. *Energy Databus*. 2015. URL: https://github.com/deanhiller/databus (cit. on p. 125).

[Bus12] S. Bushik. "A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak." In: (Oct. 2012). Altoros (cit. on pp. 17, 18).

[Cam15] M. Campbell. *Apple acquires big data analytics firm Acunu*. 2015. URL: http://appleinsider.com/articles/15/03/25/apple-acquires-big-data-analytics-firm-acunu (cit. on p. 126).

[Car15] D. W. Carder. *RRDtool Scalability*. 2015. URL: http://net.doit.wisc.edu/~dwcarder/rrdcache/ (cit. on p. 27).

[Cat11] R. Cattell. "Scalable SQL and NoSQL Data Stores." In: *SIGMOD Rec.* 39.4 (May 2011), pp. 12–27 (cit. on p. 12).

[CBA+15] R. Chitkara, W. Ballhaus, O. Acker, D. B. Song, A. Sundaram, M. Popova. *The Internet of Things: The next growth engine for the semiconductor industry*. PricewaterhouseCoopers AG Wirtschaftsprüfungsgesellschaf. May 2015. URL: https://www.pwc.ie/media-centre/assets/publications/2015-pwc-iot-semicon-paper.pdf (cit. on p. 11).

[Che15a] Chef Software, Inc. *Chef*. 2015. URL: https://www.chef.io/chef/ (cit. on p. 57).

[Che15b] X. Chen. *Awesome Public Datasets*. 2015. URL: https://github.com/caesar0301/awesome-public-datasets (cit. on p. 23).

[Cis15] Cisco Systems, Inc. *ParStream*. 2015. URL: https://www.parstream.com/ (cit. on p. 127).

[Cit15] Cityzen Data. *Cityzen Data*. 2015. URL: http://www.cityzendata.com/ (cit. on p. 126).

[CKH+15] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista. *The UCR Time Series Classification Archive*. www.cs.ucr.edu/~eamonn/time_series_data/. July 2015 (cit. on p. 23).

[CLS09] H. Cheng, Y.-C. Lu, C. Sheu. "An ontology-based business intelligence application in a financial knowledge management system." In: *Expert Systems with Applications* 36.2, Part 2 (2009), pp. 3614–3622 (cit. on p. 12).

[Cod15] Coda Hale, Yammer Inc. *Metrics*. 2015. URL: http://metrics.dropwizard.io/3.1.0/ (cit. on p. 29).

[CON12] I. CONSULTING. *Sensors Markets 2016 – Press Release. Major Findings of the New Report onMarkets, Strategies, Technologies*. Mar. 2012 (cit. on p. 11).

[COU15] COUCHBASE. *Couchbase*. 2015. URL: http://www.couchbase.com/ (cit. on p. 20).

[Cou98] O. Council. *APB-1 OLAP Benchmark Release II*. Nov. 1998. URL: http://www.symcorp.com/downloads/OLAPBenchmarkReleaseII.pdf (cit. on p. 22).

[CST+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10 (2010), pp. 143–154 (cit. on pp. 17, 18, 22, 49, 65).

[CZ15] Camunda services GmbH, Zalando SE. *bpmn.io*. 2015. URL: http://bpmn.io/ (cit. on p. 63).

[Dat15] DataStax. *DataStax Enterprise*. 2015. URL: http://www.datastax.com/products/datastax-enterprise (cit. on p. 126).

[Dav15] C. Davis. *Graphite - Scalable Realtime Graphing*. 2015. URL: http://graphite.wikidot.com/ (cit. on pp. 20, 29).

[Daw15] S. Dawson-Haggerty. *sMAP: the Simple Measurement and Actuation Profile*. 2015. URL: http://www.cs.berkeley.edu/~stevedh/smap2/#smap-the-simple-measurement-and-actuation-profile (cit. on p. 126).

[DeL15] T. DeLuca. *python-vagrant*. 2015. URL: http://github.com/todddeluca/python-vagrant (cit. on pp. 57, 58).

[DeW91] D. J. DeWitt. "The Wisconsin Benchmark: Past, Present, and Future." In: *The Benchmark Handbook*. 1991, pp. 119–165 (cit. on p. 49).

[DG08] J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113 (cit. on p. 22).

[DMF12] L. Deri, S. Mainardi, F. Fusco. "tsdb: A Compressed Database for Time Series." In: *Traffic Monitoring and Analysis*. Ed. by A. Pescapè, L. Salgarelli, X. Dimitropoulos. Vol. 7189. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 143–156 (cit. on pp. 17, 125).

[Doc15] Docker, Inc. *Docker*. 2015. URL: http://www.docker.com (cit. on p. 57).

[DOT15] DOT Systems. *PulsarTSDB*. 2015. URL: http://www.dotsystems.pl/products/pulsartsdbsm.html (cit. on p. 127).

[Dug15] J. M. Dugan. *TSDB*. 2015. URL: https://code.google.com/p/tsdb/ (cit. on p. 126).

[Ela15] Elasticsearch BV. *Elasticsearch*. 2015. URL: https://www.elastic.co/products/elasticsearch (cit. on pp. 20, 28).

[Ele11] Electric Power Research Institute (EPRI). *Estimating the Costs and Benefits of the Smart Grid. A Preliminary Estimate of the Investment Requirements and the Resultant Benefits of a Fully Functioning Smart Grid*. Mar. 2011 (cit. on p. 11).

[EM10] M. Erol-Kantarci, H. Mouftah. "Using wireless sensor networks for energy-aware homes in smart grids." In: *Computers and Communications (ISCC), 2010 IEEE Symposium on*. June 2010, pp. 456–458 (cit. on p. 11).

[End15] End Point Corporation. "Benchmarking Top NoSQL Databases: Apache Cassandra, Couchbase, HBase, and MongoDB." In: (May 2015). Originally published in April 13, 2015. Revised in May 27, 2015. (cit. on p. 20).

[Ene15] Enea. *Polyhedra IMDB*. 2015. URL: http://www.enea.com/polyhedra/ (cit. on p. 127).

[Ets15] Etsy, Inc. *StatsD*. 2015. URL: https://github.com/etsy/statsd (cit. on p. 29).

[Eva15] E. Evans. *Newts*. 2015. URL: https://github.com/OpenNMS/newts; http://opennms.github.io/newts/ (cit. on pp. 29, 125).

[Fie15a] A. Fiedler. *Time Series Database*. 2015. URL: http://andyfiedler.com/projects/time-series-database/ (cit. on p. 126).

[Fie15b] A. Fiedler. *TsTables*. 2015. URL: https://github.com/afiedler/tstables (cit. on p. 126).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014 (cit. on p. 15).

[For15a] J. Forcier. *Fabric*. 2015. URL: http://www.fabfile.org/ (cit. on p. 58).

[For15b] F. Forster. *collectd – The system statistics collection daemon*. 2015. URL: https://collectd.org/ (cit. on p. 32).

[Fou15] FoundationDB. *FoundationDB*. 2015. URL: http://www.foundationdb.com/ (cit. on p. 127).

[Fre07] Free Software Foundation, Inc. *GNU General Public License (GPL), Version 3.0*. June 2007. URL: http://www.gnu.org/licenses/gpl-3.0.de.html (cit. on p. 42).

[Fre91] Free Software Foundation, Inc. *GNU General Public License (GPL), Version 2.0*. June 1991. URL: http://www.gnu.org/licenses/old-licenses/gpl-2.0.de.html (cit. on p. 42).

[Fre99] Free Software Foundation, Inc. *GNU Lesser General Public License (LGPL), Version 2.1*. Feb. 1999. URL: http://www.gnu.org/licenses/old-licenses/lgpl-2.1.de.html (cit. on p. 42).

[Fuh15] J. Fuhrer. *docker-py*. 2015. URL: http://github.com/docker/docker-py (cit. on p. 57).

[GHTC13] K. Grolinger, W. Higashino, A. Tiwari, M. Capretz. "Data management in cloud environments: NoSQL and NewSQL data stores." In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (2013), p. 22 (cit. on p. 12).

[Gia15] G. Giamarchi. *Vagrant OpenStack Cloud Provider*. 2015. URL: http://github.com/ggiamarchi/vagrant-openstack-provider (cit. on p. 57).

[GJK+14] T. Goldschmidt, A. Jansen, H. Koziolek, J. Doppelhamer, H. Breivold. "Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes." In: *IEEE 7th International Conference on Cloud Computing (CLOUD)* (June 2014), pp. 602–609 (cit. on p. 19).

[Goo15] Google Inc. *Google Cloud Platform*. 2015. URL: https://cloud.google.com/ (cit. on p. 56).

[GP15] The Graphite Project. *Whisper*. 2015. URL: https://github.com/graphite-project/whisper (cit. on p. 126).

[Gro15] D. Groshev. *YAWNDB*. 2015. URL: http://kukuruku.co/hub/erlang/yawndb-time-series-database; https://github.com/selectel/yawndb (cit. on p. 126).

[Gül15] C. Gülcü. *Simple Logging Facade for Java (SLF4J)*. 2015. URL: http://www.slf4j.org/ (cit. on p. 30).

[Han15] A. Hand. *How the Internet of Things Is Shaping the Sensor Market*. Aug. 2015. URL: http://www.automationworld.com/how-internet-things-shaping-sensor-market (cit. on p. 11).

[Här01] T. Härder. *Datenbanksysteme : Konzepte und Techniken der Implementierung ; mit 14 Tabellen*. Berlin Heidelberg New York Barcelona Hong Kong London Mailand Paris Tokio: Springer, 2001 (cit. on p. 12).

[Has15a] HashiCorp. *Vagrant*. 2015. URL: http://www.vagrantup.com (cit. on p. 57).

[Has15b] M. Hashimoto. *Vagrant AWS Provider*. 2015. URL: https://github.com/mitchellh/vagrant-aws (cit. on p. 57).

[HCC+14] S. Huang, Y. Chen, X. Chen, K. Liu, X. Xu, C. Wang, K. Brown, I. Halilovic. "The Next Generation Operational Data Historian for IoT Based on Informix." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14 (2014), pp. 169–176 (cit. on p. 19).

[Hil15] D. Y. J. Hilpisch. "TsTables – High Frequency Times Series Data with PyTables." In: (2015) (cit. on p. 126).

[Hon15] Honeywell. *Uniformance Process History Database (PHD)*. 2015. URL: https://www.honeywellprocess.com/en-US/explore/products/advanced-applications/uniformance/Pages/uniformance-phd.aspx/ (cit. on p. 127).

[HR15] M. Hankel, B. Rexroth. *Industrie 4.0: The Reference Architectural Model Industrie 4.0 (RAMI 4.0)*. Apr. 2015. URL: http://www.zvei.org/Downloads/Automation/ZVEI-Industrie-40-RAMI-40-English.pdf (cit. on p. 11).

[Hug15] J. Hugg. *Comparing Cloud Performance with YCSB*. Nov. 2015. URL: https://voltdb.com/blog/cloud-benchmark (cit. on p. 56).

[IBM15] IBM Corporation. *IBM Informix*. 2015. URL: https://www-01.ibm.com/software/data/informix/ (cit. on p. 127).

[Inf15a] InfluxData. *InfluxData - Services*. 2015. URL: https://influxdata.com/services/#technical-support; https://influxdata.com/services/#consulting (cit. on p. 43).

[Inf15b] InfluxDB. *InfluxDB*. 2015. URL: https://influxdb.com/ (cit. on pp. 32, 126).

[Inf15c] InfluxDB. *InfluxDB - Clustering*. 2015. URL: https://influxdb.com/docs/v0.9/guides/clustering.html (cit. on pp. 33, 37).

[Inf15d] InfluxDB. *InfluxDB - Continuous Queries*. 2015. URL: https://influxdb.com/docs/v0.9/query_language/continuous_queries.html (cit. on pp. 32, 38).

[Inf15e] InfluxDB. *InfluxDB – Write Protocols and Plugins – JSON Protocol*. 2015. URL: https://influxdb.com/docs/v0.9/write_protocols/json.html (cit. on p. 32).

[Int15] International Electrotechnical Commission (IEC). *Prefixes for binary multiples. Part 2: Telecommunications and electronics*. 2015. URL: http://www.iec.ch/si/binary.htm (cit. on p. 76).

[Joh15] B. Johnson. *Sky*. 2015. URL: https://github.com/skydb/sky (cit. on p. 125).

[JSC15] JSC PROGNOZ. *Prognoz Platform*. 2015. URL: http://www.prognoz.com/ (cit. on p. 127).

[JZ14] Z. Jerzak, H. Ziekow. "The DEBS 2014 Grand Challenge." In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14 (2014), pp. 266–269 (cit. on p. 23).

[JZ15] Z. Jerzak, H. Ziekow. "The DEBS 2015 Grand Challenge." In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15 (2015), pp. 266–268 (cit. on p. 23).

[Kai15a] D. S. Kaippallimalil J. Jacob. *FinTime - a financial time series benchmark*. 2015. URL: http://cs.nyu.edu/shasha/fintime.html (cit. on p. 21).

[Kai15b] KairosDB Team. *KairosDB*. 2015. URL: https://github.com/kairosdb/kairosdb; http://kairosdb.github.io/ (cit. on pp. 19, 29, 125).

[Kai15c] KairosDB Team. *KairosDB Getting Started - Changing Datastore - Configuring HBase*. 2015. URL: http://kairosdb.github.io/docs/build/html/GettingStarted.html#changing-datastore (cit. on p. 29).

[Kam15] G. Kamat. *YCSB, the Open Standard for NoSQL Benchmarking, Joins Cloudera Labs*. Aug. 2015. URL: http://blog.cloudera.com/blog/2015/08/ycsb-the-open-standard-for-nosql-benchmarking-joins-cloudera-labs/ (cit. on p. 22).

[Ker15] Kerf Software. *Kerf*. 2015. URL: http://www.kerfsoftware.com/ (cit. on p. 127).

[KFPL12] M. E. Khalefa, U. Fischer, T. B. Pedersen, W. Lehner. "Model-based Integration of Past & Future in TimeTravel." In: *PVLDB* 5.12 (2012), pp. 1974–1977 (cit. on p. 128).

[Kir15] M. Kirkwood. *Benchw*. Oct. 2015. URL: http://benchw.sourceforge.net/ (cit. on p. 22).

[KJ13] T. Kalibera, R. Jones. "Rigorous Benchmarking in Reasonable Time." In: *Proceedings of the 2013 International Symposium on Memory Management*. ISMM '13. Seattle, Washington, USA: ACM, 2013 (cit. on p. 122).

[Kon13] M. Kontschieder. *Open Source ERP-Software – Chance oder Risiko? Eine holistische Betrachtung von Open Source ERP-Software*. Bod Third Party Titles, 2013 (cit. on p. 13).

[KT15] A. Kulkarni, J. Truelsen. *wkhtmltopdf*. 2015. URL: http://wkhtmltopdf.org/ (cit. on p. 58).

[Kx15] Kx. *kdb+*. 2015. URL: http://kx.com/software.php (cit. on p. 127).

[Laz15] E. Lazin. *Akumuli*. 2015. URL: https://github.com/akumuli/Akumuli; http://akumuli.org/ (cit. on p. 125).

[LHY+13] S.-H. Lim, J. Horey, Y. Yao, E. Begoli, Q. Cao. "Performance Implications from Sizing a VM on Multi-core Systems: A Data Analytic Application's View." In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International* (May 2013), pp. 1001–1008 (cit. on p. 122).

[Lil05] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005 (cit. on pp. 47, 48).

[Lin15] LinkedIn Corp. *Databus*. 2015. URL: https://github.com/linkedin/databus (cit. on pp. 19, 126).

[LL13] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. 3., korr. Aufl. Heidelberg: dpunkt-Verl., 2013 (cit. on p. 48).

[LLL+11] J. Liu, X. Li, D. Liu, H. Liu, P. Mao. "Study on Data Management of Fundamental Model in Control Center for Smart Grid Operation." In: *Smart Grid, IEEE Transactions on* 2.4 (Dec. 2011), pp. 573–579 (cit. on p. 12).

[LSKD15a] C. Larsen, B. Sigoure, V. Kiryanov, B. D. Demir. *OpenTSDB*. 2015. URL: http://www.opentsdb.net/ (cit. on pp. 17, 19, 30, 125).

[LSKD15b] C. Larsen, B. Sigoure, V. Kiryanov, B. D. Demir. *OpenTSDB - Setup HBase - Using LZO*. 2015. URL: http://opentsdb.net/setup-hbase.html (cit. on p. 30).

[LSKD15c] C. Larsen, B. Sigoure, V. Kiryanov, B. D. Demir. *OpenTSDB - Writing Data - Timestamps*. 2015. URL: http://opentsdb.net/docs/build/html/user_guide/writing.html#timestamps (cit. on p. 39).

[Maj15] S. Majer. *InfluxDB – Built-in load balancing and failover*. 2015. URL: https://github.com/influxdb/influxdb-java/issues/12 (cit. on pp. 33, 37).

[Man15] Man AHL. *Arctic TimeSeries and Tick store*. 2015. URL: https://github.com/manahl/arctic (cit. on p. 125).

[Mas88] Massachusetts Institute of Technology. *The MIT License (MIT)*. 1988. URL: https://opensource.org/licenses/MIT (cit. on p. 42).

[McO15] McObject, LLC. *eXtremeDB Financial Edition*. 2015. URL: http://financial.mcobject.com/extremedb-financial-edition/platform/ (cit. on p. 127).

[Mer15a] E. Merdanović. *How to install KairosDB time series database?* Feb. 2015. URL: http://www.erol.si/2015/02/how-to-install-kairosdb-timeseries-database/ (cit. on p. 29).

[Mer15b] G. Merlino. *Tranquility*. 2015. URL: https://github.com/druid-io/tranquility (cit. on p. 32).

[Mic15a] Microsoft Corporation. *Microsoft Azure*. 2015. URL: https://azure.microsoft.com/de-de/ (cit. on pp. 31, 56).

[Mic15b] Microsoft Corporation. *SQL Server 2014*. 2015. URL: http://www.microsoft.com/de-de/server-cloud/products/sql-server/ (cit. on p. 127).

[Mic15c] Microsoft Open Technologies. *Vagrant Azure Provider*. 2015. URL: https://github.com/Azure/vagrant-azure (cit. on p. 57).

[Mic15d] Microsoft Research. *Bolt*. 2015. URL: http://labofthings.codeplex.com/ (cit. on p. 126).

[Mit13] B. Mitschang. *Datenbanken und Informationssysteme. Vorlesungsfolien*. 2013 (cit. on p. 12).

[Mon15a] MonetDB B.V. *MonetDB*. 2015. URL: https://www.monetdb.org/Home (cit. on pp. 33, 126).

[Mon15b] MonetDB B.V. *MonetDB - Distributed Query Processing*. 2015. URL: https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/DistributedQueryProcessing (cit. on p. 33).

[Mon15c] MonetDB B.V. *MonetDB - MonetDB License*. 2015. URL: https://www.monetdb.org/Legal/MonetDBLicense (cit. on p. 43).

[Mon15d] MonetDB B.V. *MonetDB - Transaction Replication*. 2015. URL: https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/TransactionReplication (cit. on pp. 33, 37).

[Mon15e] MonetDB Solutions. *MonetDB Solutions Homepage*. 2015. URL: https://www.monetdbsolutions.com/ (cit. on p. 43).

[Mon15f] MongoDB, Inc. *MongoDB*. 2015. URL: https://www.mongodb.org/ (cit. on p. 18).

[Mül15] T. Müller. *H2 Database*. 2015. URL: http://www.h2database.com/ (cit. on p. 29).

[NE13] D. Nelubin, B. Engber. "Ultra-High Performance NoSQL Benchmarking: Analyzing Durability and Performance Tradeoffs." In: (Jan. 2013). Thumbtack Technology (cit. on p. 18).

[Net15] Netflix, Inc. *Atlas*. 2015. URL: https://github.com/Netflix/atlas (cit. on p. 126).

[New15] New Relic, Inc. *New Relic Insights*. 2015. URL: http://newrelic.com/insights (cit. on p. 127).

[Obj11] Object Management Group. *Business Process Model and Notation (BPMN) Version 2.0*. OMG Document Number: formal/2011-01-03. 2011 (cit. on p. 63).

[Oet15] T. Oetiker. *RRDtool*. 2015. URL: http://oss.oetiker.ch/rrdtool/ (cit. on pp. 17, 27, 126).

[One15] OneMarketData. *ONETick Time-Series Tick Database*. 2015. URL: https://www.onetick.com/web/# (cit. on p. 127).

[Ope15a] OpenStack Foundation. *Gnocchi*. 2015. URL: http://docs.openstack.org/developer/gnocchi/; https://julien.danjou.info/blog/2015/openstack-gnocchi-first-release; https://github.com/openstack/gnocchi (cit. on p. 126).

[Ope15b] OpenStack Foundation. *OpenStack*. 2015. URL: http://www.openstack.org/ (cit. on p. 57).

[Ora15a] Oracle. *VirtualBox*. 2015. URL: http://www.virtualbox.org/ (cit. on p. 56).

[Ora15b] Oracle Corporation. *Java*. 2015. URL: https://www.java.com (cit. on p. 30).

[Ora15c] Oracle Corporation. *Java Management Extensions (JMX) Technology*. 2015. URL: http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html (cit. on p. 29).

[Ora15d] Oracle Corporation. *MySQL - Choosing Which Version of MySQL to Install*. 2015. URL: https://dev.mysql.com/doc/refman/5.0/en/choosing-version.html (cit. on p. 43).

[Ora15e] Oracle Corporation. *MySQL Cluster*. 2015. URL: https://dev.mysql.com/downloads/cluster/ (cit. on pp. 27, 35, 127).

[Ora15f] Oracle Corporation. *MySQL Community Server*. 2015. URL: http://dev.mysql.com/downloads/mysql/ (cit. on pp. 12, 17, 20, 34, 39, 127).

[Ora15g] Oracle Corporation. *MySQL HA/Scalability Guide - High Availability and Scalability*. 2015. URL: https://dev.mysql.com/doc/mysql-ha-scalability/en/ha-overview.html (cit. on p. 34).

[Ora15h] Oracle Corporation. *MySQL Services*. 2015. URL: http://www.mysql.com/services/ (cit. on p. 43).

[Ora15i] Oracle Corporation. *Oracle Database*. 2015. URL: https://www.oracle.com/database/index.html (cit. on p. 127).

[OSI15] OSIsoft, LLC. *OSIsoft PI*. 2015. URL: http://www.osisoft.com/ (cit. on p. 127).

[PA13] S. Prasad, S. Avinash. “Smart meter data analytics using OpenTSDB and Hadoop.” In: *Innovative Smart Grid Technologies - Asia (ISGT Asia), 2013 IEEE* (Nov. 2013), pp. 1–6 (cit. on pp. 12, 19).

[Pan15] M. Panzarino. *Apple Acquires Durable Database Company FoundationDB*. 2015. URL: http://techcrunch.com/2015/03/24/apple-acquires-durable-database-company-foundationdb/ (cit. on p. 127).

[Par15a] Paradigm4, Inc. *SciDB*. 2015. URL: http://scidb.org/ (cit. on p. 126).

[Par15b] Pardot. *Rhombus*. 2015. URL: https://github.com/Pardot/Rhombus (cit. on pp. 30, 125).

[PDL14] J. Paris, J. Donnal, S. Leeb. "NilmDB: The Non-Intrusive Load Monitor Database." In: *Smart Grid, IEEE Transactions on* 5.5 (Sept. 2014), pp. 2459–2467 (cit. on p. 126).

[PEC15] PEC: Peer Energy Cloud. *PEC: Peer Energy Cloud*. 2015. URL: http://www.peerenergycloud.de/ (cit. on p. 20).

[PFA09] C. Pungilă, T.-F. Fortiş, O. Aritoni. "Benchmarking Database Systems for the Requirements of Sensor Readings." In: *IETE TECHNICAL REVIEW* 26.5 (2009), pp. 342–349 (cit. on p. 17).

[PFT+15] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, K. Veeraraghavan. "Gorilla: A Fast, Scalable, In-memory Time Series Database." In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1816–1827 (cit. on p. 125).

[pgA15] pgAdmin Development Team. *pgAdmin*. 2015. URL: http://www.pgadmin.org/ (cit. on p. 35).

[PGD15a] The PostgreSQL Global Development Group. *PostgreSQL*. 2015. URL: http://www.postgresql.org/ (cit. on pp. 18, 35, 128).

[PGD15b] The PostgreSQL Global Development Group. *PostgreSQL - High Availability, Load Balancing, and Replication - 25.1. Comparison of Different Solutions*. 2015. URL: http://www.postgresql.org/docs/9.3/static/different-replication-solutions.html (cit. on p. 35).

[PGD15c] The PostgreSQL Global Development Group. *PostgreSQL - License*. 2015. URL: http://www.postgresql.org/about/licence/; http://opensource.org/licenses/postgresql (cit. on p. 43).

[PGD15d] The PostgreSQL Global Development Group. *PostgreSQL - Professional Services*. 2015. URL: http://www.postgresql.org/support/professional_support/ (cit. on p. 43).

[PGD15e] The PostgreSQL Global Development Group. *PostgreSQL - Versioning policy*. 2015. URL: http://www.postgresql.org/support/versioning/ (cit. on p. 43).

[php15a] phpMyAdmin Development Team. *phpMyAdmin*. 2015. URL: https://www.phpmyadmin.net/ (cit. on p. 34).

[php15b] phpPgAdmin Development Team. *phpPgAdmin*. 2015. URL: http://phppgadmin.sourceforge.net/doku.php (cit. on p. 35).

[Pro15a] Project FiFo. *DalmatinerDB*. 2015. URL: https://dalmatiner.io/ (cit. on p. 126).

[Pro15b] Prometheus Authors. *Prometheus*. 2015. URL: http://prometheus.io/ (cit. on pp. 27, 126).

[Pro15c] Prometheus Authors. *Prometheus Pushgateway*. 2015. URL: https://github.com/prometheus/pushgateway (cit. on p. 27).

[Pro15d] Prometheus Authors. *Prometheus Pushgateway Readme*. 2015. URL: https://github.com/prometheus/pushgateway/blob/master/README.md (cit. on p. 27).

[pup15] puppet labs. *Puppet*. 2015. URL: http://puppetlabs.com/ (cit. on p. 57).

[Rac15a] Rackspace. *Blueflood*. 2015. URL: http://blueflood.io/; %20https://github.com/rackerlabs/blueflood/wiki/ (cit. on pp. 28, 125).

[Rac15b] Rackspace. *Blueflood - FAQ*. 2015. URL: https://github.com/rackerlabs/blueflood/wiki/FAQ (cit. on pp. 29, 38).

[Rac15c] Rackspace. *Blueflood v2.0 endpoints*. 2015. URL: https://github.com/rackerlabs/blueflood/wiki/v2.0-endpoints (cit. on p. 29).

[Rac15d] Rackspace US, Inc. *Cloud Files*. 2015. URL: https://www.rackspace.com/de-de/cloud/files (cit. on p. 29).

[Rah15] S. Rahman. *BEMOSS*. 2015. URL: http://www.bemoss.org/ (cit. on p. 126).

[Rog14] A. Rogers. *VoltDB in-memory database achieves best-in-class results, running in the cloud, on the YCSB Benchmark*. May 2014. URL: https://voltdb.com/blog/voltdb-memory-database-achieves-best-class-results-running-cloud-ycsb-benchmark (cit. on p. 22).

[Sal15a] D. Sallings. *Seriesly*. 2015. URL: https://github.com/dustin/seriesly (cit. on p. 126).

[Sal15b] R. Salvagno. *Cisco Announces Intent to Acquire ParStream*. 2015. URL: http://blogs.cisco.com/news/cisco-announces-data-analytics-news (cit. on p. 127).

[San15] S. Sanfilippo. *Redis*. 2015. URL: http://redis.io/ (cit. on p. 18).

[SAP15] SAP SE. *SAP HANA*. 2015. URL: http://hana.sap.com/ (cit. on p. 127).

[Ser15] Serotonin Software. *Squwk*. 2015. URL: http://squwk.com/ (cit. on p. 127).

[Sev15] Seven2one Informationssysteme GmbH. *Mesap*. 2015. URL: http://www.seven2one.de/de/technologie/mesap.html (cit. on p. 127).

[Sha14] J. Shahid. *Updating an existing point end up inserting*. Second comment. Apr. 2014. URL: https://github.com/influxdb/influxdb/issues/391 (cit. on p. 136).

[Sit15] SiteWhere, LLC. *SiteWhere*. 2015. URL: http://www.sitewhere.org/ (cit. on p. 125).

[Sky15] SkyFoundry, LLC. *SkySpark*. 2015. URL: https://www.skyfoundry.com/skyspark/; https://www.skyfoundry.com/forum/topic/24 (cit. on p. 127).

[Sof15] SoftLayer Technologies. Inc. *IBM SoftLayer*. 2015. URL: http://www.softlayer.com/ (cit. on p. 56).

[Spi03] P. D. G. Spindler. "Rechtsfragen der Open Source Software." In: *Open Source* (2003) (cit. on p. 42).

[Spl15] Splunk Inc. *Splunk*. 2015. URL: http://www.splunk.com/ (cit. on p. 127).

[Sqr15] Sqrrl Enterprise. *Sqrrl*. 2015. URL: http://sqrrl.com/ (cit. on p. 127).

[Squ15] Square, Inc. *Cube*. 2015. URL: http://square.github.io/cube/ (cit. on p. 125).

[STA15] STAC Benchmark Council. *STAC-M3 Benchmark Suite*. 2015. URL: https://stacresearch.com/sites/default/files/d5root/STAC-M3_Antuco_Overview.pdf; https://kx.com/_papers/STAC-M3-KX_Antuco_KDB-130603.pdf; https://stacresearch.com/m3 (cit. on p. 21).

[Sti14] C. Stimmel. *Big Data Analytics Strategies for the Smart Grid*. Auerbach Book. Taylor & Francis, 2014 (cit. on p. 12).

[Sti15] M. Stirling. *TimeStore, a lightweight time-series database engine*. 2015. URL: http://www.livesense.co.uk/timestore; http://www.mike-stirling.com/redmine/projects/timestore; https://github.com/mikestir/timestore (cit. on p. 126).

[SZGA15] M. Strohbach, H. Ziekow, V. Gazis, N. Akiva. "Towards a Big Data Analytics Framework for IoT and Smart City Applications." In: *Modeling and Processing for Next-Generation Big-Data Technologies*. Ed. by F. Xhafa, L. Barolli, A. Barolli, P. Papajorgji. Vol. 4. Modeling and Optimization in Science and Technologies. Springer International Publishing, 2015, pp. 257–282 (cit. on pp. 12, 20).

[Tao15] Taowen. *In Search of Agile Time Series Database*. 2015. URL: https://www.gitbook.com/book/taowen/tsdb/details (cit. on p. 20).

[Tar15] W. Tarreau. *HAProxy*. 2015. URL: http://www.haproxy.org/ (cit. on p. 30).

[Tem15] TempoIQ. *TempoIQ*. 2015. URL: https://www.tempoiq.com/ (cit. on pp. 18, 127).

[THK+15] J. Thomsen, N. Hartmann, F. Klumpp, T. Erge, M. Falkenthal, O. Kopp, F. Leymann, S. Stando, N. Turek, C. Schlenzig, H. Schwarz. "Darstellung des Konzeptes – DMA Decentralised Market Agent – zur Bewältigung zukünftiger Herausforderungen in Verteilnetzen." In: *INFORMATIK 2015*. Ed. by D. Cunningham, P. Hofstedt, K. Meer, I. Schmitt. Vol. P-246. Lecture Notes in Informatics (LNI). 2015 (cit. on p. 11).

[Tra15a] Transaction Processing Performance Council (TPC). *Active TPC Benchmarks*. 2015. URL: http://www.tpc.org/information/benchmarks.asp (cit. on p. 22).

[Tra15b] Transaction Processing Performance Council (TPC). *TPC-C Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-C_V5-11.pdf (cit. on p. 22).

[Tra15c] Transaction Processing Performance Council (TPC). *TPC-DI Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPCDI-v1.1.0.pdf (cit. on p. 22).

[Tra15d] Transaction Processing Performance Council (TPC). *TPC-DS Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpcds_1.3.1.pdf (cit. on p. 22).

[Tra15e] Transaction Processing Performance Council (TPC). *TPC-E Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPCE-v1.14.0.pdf (cit. on p. 22).

[Tra15f] Transaction Processing Performance Council (TPC). *TPC-H Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpch2.17.1.pdf (cit. on p. 22).

[Tra15g] Transaction Processing Performance Council (TPC). *TPC-VMS Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-VMS-V1.2.0.pdf (cit. on p. 22).

[Tra15h] Transaction Processing Performance Council (TPC). *TPCx-HS Benchmark*. 2015. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpcx-hs_specification_1.3.0.pdf (cit. on p. 22).

[Tre15] Treasure Data. *Treasure Data*. 2015. URL: https://www.treasuredata.com/ (cit. on p. 127).

[Uni14] United States Department of Energy. *2014 Smart Grid System Report*. Aug. 2014 (cit. on p. 11).

[Var15] Varnish Software. *Varnish Cache*. 2015. URL: https://www.varnish-cache.org/ (cit. on p. 30).

[Viv15] VividCortex. *VividCortex: Building a Time-Series Database in MySQL*. 2015. URL: http://de.slideshare.net/vividcortex/vividcortex-building-a-timeseries-database-in-mysql (cit. on p. 12).

[VMw15a] VMware. *VMware vSphere API Python Bindings (pyVmomi)*. 2015. URL: http://github.com/vmware/pyvmomi (cit. on p. 57).

[VMw15b] VMware. *vSphere*. 2015. URL: http://www.vmware.com/de/products/vsphere (cit. on p. 57).

[Wak15] N. Wakart. *Consistently report latency in us(95/99 percentile were reporting ms)*. July 2015. URL: https://github.com/brianfrankcooper/YCSB/commit/c7a6680068e0d704ff9d1b220cbc40dec8e8dc16 (cit. on p. 75).

[Wan15] C. Wang. *node-tsdb*. 2015. URL: https://www.npmjs.com/package/tsdb; https://github.com/eleme/node-tsdb; https://github.com/hit9/tsdb.js (cit. on p. 126).

[WCZ15] J. Wan, H. Cai, K. Zhou. "Industrie 4.0: Enabling technologies." In: *Intelligent Computing and Internet of Things (ICIT), 2014 International Conference on*. Jan. 2015, pp. 135–140 (cit. on p. 11).

[Who14] C. Whong. *FOILing NYC's Taxi Trip Data*. 2014. URL: http://chriswhong.com/open-data/foil_nyc_taxi/ (cit. on p. 23).

[Wik15] Wikibooks. *SQL Dialects Reference*. 2015. URL: https://en.wikibooks.org/wiki/SQL_Dialects_Reference (cit. on p. 33).

[Wlo12] T. Wlodarczyk. "Overview of Time Series Storage and Processing in a Cloud Environment." In: *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on* (Dec. 2012), pp. 625–628 (cit. on p. 18).

[WW15] A. Wittig, M. Wittig. *TimeSeries.Guru*. 2015. URL: https://www.timeseries.guru/ (cit. on p. 127).

[Xen15] Xenomorph Software Ltd. *TimeScape EDM+*. 2015. URL: http://www.xenomorph.com/solutions/timescape-edm/ (cit. on p. 127).

[XWWC10] D. Xu, M. Wang, C. Wu, K. Chan. *Evolution of the smart grid in China*. McKinsey on Smart Grid. McKinsey & Company, 2010 (cit. on p. 11).

[Yan14] F. Yang. *Druid Development: Questions about pagination and replication*. 2014. URL: https://groups.google.com/d/msg/druid-development/0TBL5-3Z2PI/00lQvI9GLl0J (cit. on p. 32).

[YKP+11] J. Yin, A. Kulkarni, S. Purohit, I. Gorton, B. Akyol. “Scalable Real Time Data Management for Smart Grid.” In: *Proceedings of the Middleware 2011 Industry Track Workshop*. Middleware ’11. Lisbon, Portugal: ACM, 2011, 1:1–1:6 (cit. on p. 126).

[YTL+14] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, D. Ganguli. “Druid: A Real-time Analytical Data Store.” In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’14 (2014), pp. 157–168 (cit. on pp. 20, 31, 126).

[YTL+15a] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, D. Ganguli. *Druid Documentation: Loading Streaming Data*. 2015. URL: http://druid.io/docs/latest/tutorials/tutorial-loading-streaming-data.html (cit. on p. 32).

[YTL+15b] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, D. Ganguli. *Druid - Retaining or Automatically Dropping Data*. 2015. URL: http://druid.io/docs/latest/operations/rule-configuration.html (cit. on p. 38).

[Zan12] G. van Zanten. *How too many vCPUs can negatively affect performance*. May 2012. URL: http://www.gabesvirtualworld.com/how-too-many-vcpus-can-negatively-affect-your-performance/ (cit. on p. 122).

All links were last followed on December 11, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature