Survey and Comparison of Open Source Time Series Databases

Andreas Bader[1], Oliver Kopp[2], Michael Falkenthal[3]

Abstract: Time series data, i.e., data consisting of a series of timestamps and corresponding values, is a special type of data occurring in settings such as “Smart Grids”. Extended analysis techniques called for a new type of databases: Time Series Databases (TSDBs), which are specialized for storing and querying time series data. In this work, we aim for a complete list of all available TSDBs and a feature list of popular open source TSDBs. The systematic search resulted in 83 TSDBs. The twelve most prominent found open source TSDBs are compared. Therefore, 27 criteria in six groups are defined: (i) Distribution/Clusterability, (ii) Functions, (iii) Tags, Continuous Calculation, and Long-term Storage, (iv) Granularity, (v) Interfaces and Extensibility, (vi) Support and License.

Keywords: Time Series Databases, Survey, Comparison

1 Introduction and Background

The importance of sensors has been growing in the last years. Thereby, IoT technologies gained access to industrial environments to enable intensive metering of production steps, whole manufacturing processes, and further parameters. One key challenge of these endeavors is to efficiently store and analyze huge sets of metering data from many different sensors, which are typically present in the form of time series data. These principles are currently also applied to energy grids, where increasing amounts of dynamic and flexible power generation units, such as solar panels and thermal power stations, respectively, along with energy storages, such as batteries or pumped storage hydro power stations, require to intensively meter the parameters of the energy grid [Ko15]. At this, metered data is analyzed to smartly control the energy grid with respect to balancing volatile amounts of generated and consumed electricity, which also results in changes of the existing energy market [BF12]. To conceive the volume of time series data that may be collected in such scenarios, the equipping of a smaller city with smart meters can lead up to 200,000 million voltage measurements each month—which all have to be stored and processed [St15]. Real-world metering is also done in the context of our BMWi-funded project NEMAR (03ET4018), where Time Series Databases (TSDBs) play a crucial role to develop a Decentralized Market Agent (DMA) [Th15]. The goal of a DMA is to form a new role in the energy market to gain a global price and energy transfer optimum. This should be done efficiently, but also the TCO for the infrastructure should reasonable.

To handle such amounts of data, it is necessary to scale the Database Management System (DBMS) accordingly, which means distribution across several nodes with the possibility to

[1] University of Stuttgart, IPVS, Universitätsstr. 38, Stuttgart, Germany, andreas.bader@ipvs.uni-stuttgart.de
[2] University of Stuttgart, IPVS, Universitätsstr. 38, Stuttgart, Germany, oliver.kopp@ipvs.uni-stuttgart.de
[3] University of Stuttgart, IAAS, Universitätsstr. 38, Stuttgart, Germany, michael.falkenthal@iaas.uni-stuttgart.de

increase further when the city grows is inevitable. There are different views on the question if traditional Relational Database Management Systems (RDBMS) can handle such amounts of data. On the one hand, distributing writes in a relational model, whilst keeping full consistency and the same level of query latencies, is not easily possible [PA13]. On the other hand, VividCortex, a SaaS platform for database monitoring, uses MySQL Community Server [Or16a], a traditional RDBMS, in combination with InnoDB as storage subsystem to store 332,000 values per second while running on three Amazon Web Services (AWS) Elastic Compute Cloud (EC2) servers [Vi14]. Most of the used queries are Insertions [Sc14] and basic SQL queries like SUM queries [Sc15]. To ingest into MySQL with that ingest rate, several trade-offs are required [Sc15]: 1) Performing batch-wise ingestion into vectors[4] that are stored as delta values and grouped by a clustered primary key[5], 2) due to the clustered primary key, performing ad-hoc SQL queries is not possible, instead a time series service had to be used for querying, 3) while building the cluster, it must be decided and manually implemented how data is sharded, grouped, and how indexes are created. Scalable RDBMS which use small-scope operations and transactions are trying to achieve better scalability and performance than traditional RDBMS [Ca11]. This shows that, with enough effort and by considering the specific use case at hand, RDBMS can be used for storing time series data. However, it is not clear if this is still possible if the queries are more complex or algorithmic.

NoSQL DBMS provide a solution with more possibilities for distribution by weakening relations and consistencies [Gr13]. Knowing that sensor data has a timestamp attached to it, a specific type of NoSQL DBMS for that type of data arose: Time Series Databases (TSDBs). This new type provides the scalability to store huge amounts of time series data, ingested with an high ingest rate, and perform SCAN queries on it [Te14]. The boundaries between NoSQL DBMS and TSDBs are fluent, as there exists no precisely defined boundary between them.

However, since a vast amount of TSDB solutions have emerged in the last years, a clear overview is missing as of today. Especially the fact that TSDBs (i) are capable of diverse functionalities regarding operations on time series data, while (ii) building upon completely different technologies motivates to establish a set of criteria in order to make them objectively comparable. Such criteria can be used for architectural decisions, which may lead to other results than a plain performance comparison [Zi15].

Therefore, the contribution of this paper is threefold: (i) the main contribution is to present a survey and comparison on existing TSDB solutions, which (ii) builds upon a comprehensive list of found TSDBs, and (iii) a set of measurable comparison criteria. This survey presents a popularity ranking of all found TSDBs whereas the feature comparison regards the subset consisting of popular open source TSDBs and can be used as a basis for deciding which open source TSDBs to select for specific use cases at hand.

The remainder of this paper is structured as follows: Sect. 2 defines what a TSDB is and how TSDBs can be grouped. Sect. 3 presents related work. Sect. 4 defines the comparison criteria. Sect. 5 presents the search process and the found TSDBs. Sect. 6 introduces the

[4] Vectors help to hold multiple values per row, so that there is not one row for each second (using a distance of one second between values) [Sc15].

[5] A clustered primary key of VividCortex consists of the host name, the time series name, and a timestamp [Sc15].

chosen TSDBs more detailed. The main contribution of this paper, the feature comparison, is done in Sect. 7. Sect. 8 concludes and describes future work.

2 Definitions

For comparing TSDBs, it is necessary to define which databases belong to this group as the term "TSDB" is not consistently defined or used. Traditionally, 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" [Th01]. Most literature uses the term "DBMS" (like RDBMS) instead of "DBS". Therefore TSDB is mostly used in the meaning of "time series database management system". In this paper, DBMS, RDBMS, and TSDB will be used in this non-accurate way.

For this paper, a DBMS that can (i) store a row of data that consists of timestamp, value, and optional tags, (ii) store multiple rows of time series data grouped together (e. g., in a time series), (iii) can query for rows of data, and (iv) can contain a timestamp or a time range in a query is called TSDB. This definition is more precise than defining a TSDB as a database that can contain historical data besides current data [DDL14]. This means that the "time series"-character of the DBMS must inherently exist, it must be possible to do queries by timestamps and time ranges. Storing time series data in a RDBMS can be done in two ways: timestamp-sorted or series-sorted. Depending on the indexes and queries used, this can impact performance of the queries [Sc15].

Queries of TSDBs Nine different queries will be used for comparison and explanation: Insertion (INS), Updating (UPD), Reading (READ), Scanning (SCAN), Averaging (AVG), Summarization (SUM), Counting (CNT), Deletion (DEL), Maximization (MAX), and Minimization (MIN). The resulting data inside a TSDB of one single INS, that was successfully executed, is called row (of time series data). A row of time series data consists of a timestamp, a value (usually floating point with double or single precision), and optional tag names and values. A TSDB can store several time series, whereby each time series consists of a name and several rows of time series data. A time series is used to group rows of time series data together on a higher level than tags, comparable to a table in traditional RDBMS (e. g., "voltage measurements" as name of the time series, whereby the tag names and values define from which sensor in which building the measurement came). A timestamp expresses a specific point in time, usually in milliseconds starting from 1970-01-01. A granularity defines the smallest possible distance between two timestamps that can be stored. The granularity used to store data and the granularity used for querying can be different. A time range describes a period of time between two timestamps and can also be zero. A tag consists of a tag name and a tag value (both usually alphanumeric). 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). Data can be aggregated using the aggregation queries AVG, SUM, CNT, MAX, and MIN. These queries can also be used to group the results together in time ranges, which are also called "buckets" (e. g., querying the maximum value of each day in a year).

The queries are defined as follows: INS is a query that inserts one single row into a DBMS. UPD 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. DEL deletes one or more rows with a specific timestamp or in a specific time range. MAX and MIN search for the maximum, respective minimum, value of several values in a specific time range.

Grouping of TSDBs Existing TSDBs can be subdivided into four groups. The first group exists of TSDBs that depend on an already existing DBMS (e. g., Cassandra, HBase, CouchDB, MySQL) to store the time series data. If another DBMS is only required to store meta data, the TSDB is not in this group. In the second group are TSDBs that can store time series data completely independent of other DBMS despite using DBMS for storing meta data or other additional information. RDBMS are not in this group, even if they do not require other DBMS. The third group encloses RDBMS that can store time series data. Independent from their requirements on other DBMS, RDBMS cannot be in group 1 or 2. As last group, Proprietary contains all commercially or freely available TSDBs that are not open source. Independent from their requirements on other DBMS and from their type, proprietary DBMS cannot be in group 1, 2, or 3.

3 Related Work

Previous publications are motivated by releases of new TSDBs or the search for a DBMS for processing or storing time series data for a specific scenario. Many of these publications include a performance comparison. The two most prominent performance comparison publication are described here, an overview on other related work is presented by Bader [Ba16]. Lately, an overview over 19 existing TSDBs was published on the internet [Ac16a], which is the latest and completest overview that could be found on existing TSDBs.

Wlodarczyk [Wl12] 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) can be a better choice if a hosted solution is desired. No benchmark results are provided, only a feature comparison is done.

Deri et al. [DMF12] present tsdb as a compressed TSDB that handles large time series better than three existing solutions. They discovered OpenTSDB 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. MySQL, RRDtool, and Redis are compared against tsdb with the result that the append/search performance of tsdb is best out of the four compared DBMS.

Pungilă et al. [PFA09] tried to find a fitting database for a set of households with smart meters. For reaching their goal, they compared three RDBMS (PostgreSQL, MySQL, SQLite3), one TSDB (IBM Informix – community edition with TimeSeries DataBlade module), and three NoSQL DBMS (Oracle BerkeleyDB, Hypertable, MonetDB) in two scenarios. The conclusion is, that if the data and therefore the queries are based on a key like client identifier, sensor identifier, or timestamp, then some DBMS result in increased performance. The performance is logarithmically decreased for these DBMS when using a combination of tags. Two of the compared DBMS are interesting for their scenario, depending whether the focus lies on INS queries, READ queries, or both. Hypertable is the best choice if the focus lies on the highest rate of executed INS queries in combination with a worse SCAN performance. BerkeleyDB, which has a lower rate of executed INS queries but executes more SCAN queries, is the second best choice.

Acreman [Ac16a] compares 19 TSDBs (DalmatinerDB, InfluxDB, Prometheus, Riak TS, OpenTSDB, KairosDB, Elasticsearch, Druid, Blueflood, Graphite (whisper), Atlas, Chronix Server, Hawkular, Warp 10 (distributed), Heroic, Akumuli, BtrDB, MetricTank, Tgres) including a performance comparison. For measuring performance, a two node setup generated about 2.4 to 3.7 million INS queries per second on average [Ac16b]. As a result, DalmatinerDB can execute two to three times as much INS queries queries than other TSDBs in this comparison, followed by InfluxDB and Prometheus. Although Steven Acreman is the Co-Founder of Dataloop, a company that is connected to DalmatinerDB, he tries to provide as much transparency as possible by releasing the benchmark [Ac16b].

As a summary it can be concluded that most of the existing publications focus on performance comparison rather than doing a feature comparison. Furthermore, the conclusion of the presented work is that there is not one single database that suits every use case. This paper closes this gap by presenting a systematic search for TSDBs resulting in 83 found TSDBs and a feature comparison of 12 found open source TSDBs that are chosen by popularity.

4 Comparison Criteria

The comparison criteria are grouped into six groups that will be explained in the rest of this section. The criteria are derived from requirements on the time series database in the context of the NEMAR project [Th15].

Criteria Group 1: Distribution/Clusterability *High Availability (HA)*, *scalability*, and *load balancing* features are compared in this group. *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. It is expected that the DBMS uses eventual consistency at least, but since some time series databases do not give any explicit consistency guarantee or depend on DBMS that have configurable consistency guarantees, consistency levels are not further considered. It is also expected that a client needs to know more than one entry point (e. g., IP address) 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 application the possibility to use two TSDBs in parallel for its queries would also result in scalability and a increased performance. *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 DBMS, it is also fulfilled if only the DBMS or the TSDB or both have these features.

Criteria Group 2: Functions The availability of *INS*, *UPD*, *READ*, *SCAN*, *AVG*, *SUM*, *CNT*, *DEL*, *MAX*, and *MIN* functions is compared in this group. Sect. 2 presents an explanation of these queries. All of these functions are specific to time series. There are more complex queries like changing the granularity, grouping by time spans, or performing autoregressive integrated moving average (ARIMA) time series analysis, but due to the experiences in the NEMAR project [Th15] and the fact that VividCortex (see Sect. 1) also uses mostly INS queries [Sc14] and other simpler query types (such as SUM queries) in their setup [Sc15], simple query types are considered sufficient for a comparison.

Criteria Group 3: Tags, Continuous Calculation, Long-term Storage, and Matrix Time Series *Continuous calculation*, *tags*, *long-term storage*, and the support of *matrix time series* are compared this group. *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. 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 Sect. 1 with 360,000 values per second (on a full rollout) [St15] 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). Most TSDBs support time series with one timestamp per value, so called vector time series. If, for example, forecasts (e. g., weather forecast) are stored in a TSDB, time series that support two or more timestamps per value (more than one time dimension) are needed, because a value then has two timestamps (e. g., a timestamp on which the forecast was created and a timestamp for which the forecast was made). These time series are called *matrix time series*. Solutions that can be implemented with other existing functions (e. g., tags) for storing matrix time series are not considered. The available time domain function for the first timestamp must be available for the second timestamp as well.

Criteria Group 4: Granularity *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 in this group. 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 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 (Sect. 2). 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.

Criteria Group 5: Interfaces and Extensibility *Application Programming Interfaces (APIs)*, *interfaces*, *client libraries* that are not third-party, are listed in this group. It is also compared if an interface for *plugins* exist. *APIs* and *interfaces* are used to connect to a TSDB to execute queries. Interfaces can be non-graphical (e. g., User Interface (UI)) or graphical (e. g., Graphical User Interface (GUI)). APIs (e. g., HTTP (using REST and JSON)) are used by programming languages to connect to a TSDB, execute queries, and retrieve query results. *Client libraries* encapsulate the connection management and the implementation of an API to provide easier access to a TSDB for a specific programming language. *Plugins* are used to extend the capabilities (e. g., to add new functions) of a TSDB. To develop plugins, a specific interface is required.

Criteria Group 6: Support and License The availability of a *stable version (Long Term Support (LTS))* and *commercial support*, as well as the used *license* are compared in this group. *"Stable" versions* are helpful to minimize maintenance time for updating a TSDB by only releasing updates that do not support the newest functionality but instead are considered free of bugs. 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 helpful. Only commercial support from the 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, which gives more long term safeness than without having a license.

5 Search for TSDBs

This section presents the procedure taken of searching for TSDBs. This was done by searching on Google, in ACM Digital Library, in IEEE Xplore / Electronic Library Online (IEL). Each result was individually considered and searched for TSDBs. The search terms and result numbers are presented in Tab. 1. The search terms need to include "database", otherwise many results from time series analysis or mining are included.

Search Engine	Search Term	Results
Google	"`"time series database" OR "timeseries database" OR "tsdb"`"	233,000[6]
ACM Digital Library	"`("time series database" "timeseries database" "tsdb")`"[7]	59
IEEE Xplore / Electronic Library Online (IEL)	"`((("time series database") OR "timeseries database") OR "tsdb")`"[8]	59

Tab. 1: Search terms and result numbers for the procedure of searching for TSDBs.

As a result, 83 TSDBs were found, 50 of them are open source solutions and thus 33 are proprietary. The most popular TSDBs are chosen from each of the introduced group. For ranking the found TSDBs by popularity, a Google search for each TSDB was performed. Since most TSDBs do not have any scientific papers, the amount of citations is not usable as a ranking method. Google's PageRank would also be usable for ranking, but then TSDBs

[6] The first 100 results were considered.
[7] "Any field" and "matches any" operators are used.
[8] Advanced search with "Metadata Only" operator is used.

that use more than one homepage (e. g., homepage and GitHub homepage) are ranked lower than others using only one homepage. Therefore, the amount of results found by Google was used, which also represents the amount of discussion (e. g., in news groups) about a specific TSDB in the internet. The search terms were adjusted 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 TSDB name in combination with `"time series"` was used as search string, e. g., “`"OpenTSDB" "time series"`”. For instance, when searching for `"Arctic"`, the word `"ice"` had to be excluded, because otherwise oceanic time series data is included in the search results. Further details are described by Bader [Ba16].

The American version of Google, `http://www.google.com`, was used for each search. The search was performed on September 12, 2016 between 08:11 and 09:08 AM UTC. Filtering and automatic completion was disabled, resulted in the URL `https://www.google.com/webhp?gws_rd=cr,ssl&pws=0&hl=en&gl=us&filter=0&complete=0`. The identified groups from Sect. 2 are used to rank the found TSDBs by popularity as shown in Tab. 2.

TSDB	Search Term	Results
TSDB Group 1: TSDBs with a Requirement on other DBMS		
OpenTSDB	`"OpenTSDB" "time series"`	12,900
Rhombus	`"Rhombus" "time series"`	11,700
Newts	`"Newts" "time series"`	6,610
KairosDB	`"KairosDB" "time series"`	3,130
BlueFlood	`"BlueFlood" "time series"`	2,010
Gorilla	`"Gorilla" "time series database" -"pound"`	1,520
Heroic	`"Heroic" "time series database"`	1,490
Arctic	`"Arctic" "time series database" -"ice"`	1,330
Hawkular	`"Hawkular" "time series"`	1,220
Apache Chukwa	`"Apache Chukwa" "time series"`	858
BtrDB	`"BtrDB" "time series"`	637
tsdb: A Compressed Database for Time Serie	`"tsdb: A Compressed Database for Time Series" "time series"`	634
Energy Databus	`"Energy Databus" "time series"`	605
Tgres	`"Tgres" "time series"`	445
SiteWhere	`"SiteWhere" "time series"`	436
Kairos	`"Kairos" "time series database" -"redis" -"agoragames"`	380
Cube	`"Cube" "time series" "Square, Inc."`	266
SkyDB	`"SkyDB" "time series"`	190
Chronix Server	`"Chronix Server" "time series"`	148
MetricTank	`"MetricTank" "time series"`	21
TSDB Group 2: TSDBs with no Requirement on any DBMS		
Elasticsearch	`"Elasticsearch" "time series" -"heroic"`	38,000
MonetDB	`"MonetDB" "time series"`	37,200
Prometheus	`"Prometheus" "time series" -"IMDB" -"Movie"`	33,700
Druid	`"Druid" "time series"`	28,900

TSDB	Search Term	Results
InfluxDB	`"InfluxDB" "time series"`	28,900
RRDtool	`"RRDtool" "time series"`	22,600
Atlas	`"Atlas" "time series database"`	7,960
Gnocchi	`"Gnocchi" "time series"`	5,320
Whisper	`"Whisper" "graphite" "time series"`	5,210
SciDB	`"SciDB" "time series"`	4,140
BlinkDB	`"BlinkDB" "time series"`	2,250
TSDB	`"TSDB" "Time Series Database" "time series" -"OpenTSDB"`	1,640
Seriesly	`"Seriesly" "time series"`	1,330
TsTables	`"TsTables" "time series"`	1,100
Warp 10	`"Warp 10" "time series"`	1,020
Akumuli	`"Akumuli" "time series"`	741
DalmatinerDB	`"DalmatinerDB" "time series"`	527
TimeStore	`"TimeStore" "time series"`	443
BEMOSS	`"BEMOSS" "time series"`	391
YAWNDB	`"YAWNDB" "time series"`	385
Vaultaire	`"Vaultaire" "time series"`	339
Bolt	`"Bolt" "time series" "Data Management for Connected Homes"`	176
GridMW	`"GridMW" "time series"`	25
Node-tsdb	`"Node-tsdb" "time series"`	20
NilmDB	`"NilmDB" "time series"`	9
TSDB Group 3: RDBMS		
MySQL Community Edition	`"MySQL" "time series"`	309,000
PostgreSQL	`"PostgreSQL" "time series"`	131,000
MySQL Cluster	`"MySQL Cluster" "time series"`	23,800
TimeTravel	`"TimeTravel" "time series" "dbms"`	743
PostgreSQL TS	`"PostgreSQL TS" "time series"`	1
TSDB Group 4: Proprietary		
Microsoft SQL Server	`"Microsoft SQL Server" "time series"`	94,000
Oracle Database	`"Oracle Database" "time series"`	71,500
Splunk	`"Splunk" "time series"`	30,600
SAP HANA	`"SAP HANA" "time series"`	22,100
Treasure Data	`"Treasure Data" "time series"`	15,000
DataStax Enterprise	`"DataStax Enterprise" "time series"`	12,500
FoundationDB	`"FoundationDB" "time series"`	11,300
Riak TS	`"Riak TS" "time series"`	9,720
TempoIQ	`"TempoIQ" "time series"`	8,810
kdb+	`"kdb+" "time series"`	8,220
IBM Informix	`"IBM Informix" "time series"`	7,580
Cityzen Data	`"Cityzen Data" "time series"`	6,400
Sqrrl	`"Sqrrl" "time series"`	5,460

TSDB	Search Term	Results
Databus	`"Databus" "time series"`	5,100
Kerf	`"Kerf" "time series"`	3,850
Aerospike	`"Aerospike" "time series"`	3,740
OSIsoft PI	`"OSIsoft PI" "time series"`	3,200
Geras	`"Geras" "time series"`	3,030
Axibase Time Series Database	`"Axibase Time Series Database" "time series"`	2,420
eXtremeDB Financial Editio	`"eXtremeDB Financial Edition" "time series"`	1,660
Prognoz Platform	`"Prognoz Platform" "time series"`	1,440
Acunu	`"Acunu" "time series"`	1,360
SkySpark	`"SkySpark" "time series"`	1,240
ParStream	`"ParStream" "time series"`	1,140
Mesap	`"Mesap" "time series"`	741
ONETick Time-Series Tick Database	`"ONETick Time-Series Tick Database" "time series"`	503
TimeSeries.Guru	`"TimeSeries.Guru" "time series"`	464
New Relic Insights	`"New Relic Insights" "time series"`	233
Squwk	`"Squwk" "time series"`	191
Polyhedra IMDB	`"Polyhedra IMDB" "time series"`	149
TimeScape EDM+	`"TimeScape EDM+" "time series"`	43
PulsarTSDB	`"PulsarTSDB" "time series"`	4
Uniformance Process History Database (PHD)	`"Uniformance Process History Database (PHD)" "time series"`	4

Tab. 2: A list of `http://www.google.com` results for each TSDB, grouped by the groups defined in Sect. 2. The twelve compared TSDBs are marked.

6 Introduction of Compared Open Source TSDBs

This section provides an overview on the compared TSDBs and introduces two or more representatives of each group. The grouping of Sect. 2 is chosen as grouping for the presentation. In group 1 and 2, the first five most popular TSDBs are described. In group 3, the two most popular open source RDBMS are described. We exclude group 4 as we want to focus on open source TSDBs. A full list of all found TSDBs is provided in Sect. 2.

TSDB Group 1: TSDBs with a Requirement on other DBMS *Blueflood* [Ra16b] uses Cassandra for storing its time series data. Zookeeper is optionally used to coordinate locking on different shards while performing rollups. Elasticsearch is optionally used to search for time series. *KairosDB* [Ka16] uses H2 or Cassandra as a storage for time series data. H2 is considered as slow and only recommended for testing or development purposes [Me15]. *NewTS* [Ev16] uses Cassandra for storing its time series data. It consists of a server and client, which both are written in Java. In contrast to the compared TSDBs, it is not possible to query for tags and time range (or timestamp) at the same time. *OpenTSDB* [La16a] uses HBase for storing its time series data. HBase in turn uses Zookeeper for coordination between nodes. *Rhombus* [Pa16] is a Java client for Cassandra and ships a schema for writing “Keyspace Defintions”. “Keyspace Defintions” are a set of definitions that are used by Rhombus to create a time series inside Cassandra. In comparison to the other compared TSDBs, a query on one or more keys without an existing and fitting index in Cassandra is not possible. 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, which can be used as a Boolean “AND”.

TSDB Group 2: TSDBs with no Requirement on any DBMS *Druid* [Ya14] 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, Cassandra, or Azure for storing its data, but it also can be run with a local filesystem for testing purposes. In contrast to the other compared TSDBs, Druid uses five different node types, each type for a specific task. In addition, self-written functions as aggregating functions in JavaScript are possible. *Elasticsearch* [El16a] is a search engine based on Apache Lucene. It stores data in JSON files and is not primarily designed to store time series data, but it can be adopted to store them [Ba15]. *InfluxDB* [In16b] is a TSDB that does not depend on any other DBMS and uses a SQL-like language called InfluxDB Query Language (InfluxQL). Using more than one instance as a cluster was an experimental feature, but is now only included in InfluxEnterprise [In16c]. HA and load balancing can be achieved using InfluxDB Relay [In16e]. *MonetDB* [Mo16a] is a column store that can use SQL as query language and does not depend on any other DBMS. In comparison to the other compared TSDBs, databases and tables are used instead of time series or any related concept. Every SQL statement is internally translated to a MonetDB Assembly Language (MAL) statement. *Prometheus* [Pr16a] is a monitoring solution based on a pull-based model, which means that Prometheus periodically asks for data. For querying Prometheus, a language called PromQL is used. This means that pull-based INS queries are mainly used. It can do push-based INS queries, but then it needs a second component, called Prometheus Pushgateway [Pr16f], from which data is regularly pulled (called “scraped”) by Prometheus. The timestamps of the pushed data are replaced with a new timestamp at the time of scraping, which can be unwanted. This can be avoided by setting an explicit timestamp field, which is considered not useful in most use cases [Pr16f].

TSDB Group 3: RDBMS *MySQL Community Server* [Or16a] is a popular RDBMS, which does not depend on any other DBMS. *PostgreSQL* [PGD16a] is another popular RDBMS that does not depend on any other DBMS. For both RDBMS, a database and a table as described in Sect. 2 must be created.

7 Feature Comparison of TSDBs

The TSDBs presented in Sect. 6 are compared in this section. The comparison results are divided into six tables, one for each criteria group from Sect. 4. The results of the feature comparison are organized regarding the criteria groups into Tab. 3 to 8. The results are described as follows: ✓ means available, (✓) means available with restrictions, and ✗ means not available. The contents of the table are obtained from the official documentation and basic usage of the described TSDBs if not otherwise noted.

From that comparison it can be concluded that there is are no features supported by all TSDBs in any of the criteria groups, besides being one millisecond the smallest storage granularity used by all TSDBs. Compared to traditional RDBMS (group 3), which provide a standardized query language (SQL) and stable versions, only one TSDB (group 1 and 2) provides a stable version. No TSDBs (group 1 or 2) provides a standardized query language or interface. In return, nine of them provide HA features, nine provide scalability features, and ten provide load balancing features. Three of them support all compared queries. It can be concluded that there is not only one specific TSDB that fits for all criteria and scenarios, due to the different features of the TSDBs. Druid the best choice if all criteria besides having a stable/LTS version and commercial support must be fulfilled. In contrast to other compared TSDBs, Druid also uses five different node types and supports self-written functions as aggregating functions in JavaScript. Other TSDBs such as InfluxDB, MonetDB, or one of the two RDBMS can be a better choice if stable/LTS versions or commercial support are required.

TSDB	**HA**	**Scalability**	**Load Balancing**
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Blueflood	✓	(✓)[9]	(✓)[9]
KairosDB	(✓)[9]	(✓)[9]	(✓)[9]
NewTS	(✓)[9]	(✓)[9]	(✓)[9]
OpenTSDB	(✓)[10]	(✓)[10]	(✓)[10]
Rhombus	(✓)[9]	(✓)[9]	(✓)[9]
Group 2: TSDBs with no Requirement on any DBMS			
Druid	✓	✓	✓
Elasticsearch	✓	✓	✓
InfluxDB	✓[11]	✗[12]	✓[11]
MonetDB	✓[13]	(✓)[14]	(✓)[14]
Prometheus	✗[15]	(✓)[16]	(✓)[17]
Group 3: RDBMS			
MySQL Community Server	✗[18]	✗	✗[18]
PostgreSQL	✗[18]	✗	✗[18]

Tab. 3: Comparison of Criteria Group 1: Distribution/Clusterability.

[9] Only for the Cassandra part.

[10] Using a multi node HBase setup, multiple TSDs and distribution of READ and INS queries using DNS Round Robin or external tools like Varnish Cache or HAProxy.

[11] Using InfluxDB Relay [In16e].

[12] Only available in InfluxEnterprise [In16c].

[13] "Transaction Replication" is an experimental feature [Mo16d].

[14] Available with "remote tables" [Mo16c] that use "merge tables" that cannot run INS or UPD queries [Mo16b].

[15] [Pr16d].

[16] Is not built in, but can be achieved by time series design or splitting time series [Kl16] or by using Federation [Pr16e].

[17] Prometheus Servers can be duplicated [Pr16d], but it is not clear if load can be distributed automatically without manual splitting of queries, using DNS Round Robin, or Federation [Pr16e].

[18] Possible to achieve for READ queries with master to slave replication, DNS Round Robin, or external tools like Varnish Cache or HAProxy.

TSDB	INS	READ	SCAN	AVG	SUM	CNT	MAX	MIN	UPD	DEL
Group 1: TSDBs with a requirement on NoSQL DBMS										
Blueflood	✓	✓[19]	✓	✓	✗	✓	✓	✓	✗	✗
KairosDB	✓	✓[19]	✓	✓	✓	✓	✓	✓	✗	✓
NewTS[21]	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
OpenTSDB[20]	✓	✓[19]	✓	✓	✓	✓	✓	✓	✗	✗
Rhombus[21]	✓	✓	✓	✗	✗	✓	✗	✗	✓	✓
Group 2: TSDBs with no requirement on any DBMS										
Druid[22]	✓	✓[19]	✓	✓[23]	✓	✓	✓	✓	✗	✗
Elasticsearch	✓	✓	✓[24]	✓	✓	✓	✓	✓	✓	✓
InfluxDB	✓	✓	✓	✓	✓	✓	✓	✓	✓[25]	✓
MonetDB	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Prometheus	✓	✓	✓[26]	✓	✓	✓	✓	✓	✗	✗
Group 3: RDBMS										
MySQL Community Server	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PostgreSQL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tab. 4: Comparison of Criteria Group 2: Functions.

TSDB	Continuous Calculation	Tags	Long-Term Storage	Matrix Time Series
Group 1: TSDBs with a Requirement on NoSQL DBMS				
Blueflood	✗	✗[27]	✓	✗
KairosDB	✗	✓	✗	✗
NewTS	✓	(✓)[28]	✗	✗
OpenTSDB	✗	✓	✗	✗
Rhombus	✗	(✓)[29]	✗	✗
Group 2: TSDBs with no Requirement on any DBMS				
Druid	✓	✓	✓[30]	✗
Elasticsearch	✓[31]	✓	✗[32]	✓[33]
InfluxDB	✓[34]	✓	✓[35]	✗
MonetDB	✓[36]	✓	✗	✓[37]
Prometheus	✓[38]	✓[39]	✗[40]	✗
Group 3: RDBMS				
MySQL Community Server [Or16a]	✓[36]	✓	✗	✓[37]
PostgreSQL	✓[36]	✓	✗	✓[37]

Tab. 5: Comparison of criteria group 3: Tags, Continuous Calculation, Long-term Storage, and Matrix Time Series.

[19] Uses SCAN with the smallest possible range.
[20] DEL can be substituted with a command-line tool and HBase functions.
[21] Some unsupported functions can be substituted with own Cassandra Query Language (CQL) statements.
[22] Missing functions can be substituted with re-ingestion through an IngestSegmentFirehose.
[23] Uses SCAN and CNT.
[24] Using a Range Query [El16d].
[25] Possible to overwrite with INS when setting resolution to milliseconds, see [Sh14].
[26] Using range vector selectors in combination with an offset modifier [Pr16h] or by using the HTTP API's range queries [Pr16g].
[27] See [Ra15].

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	✗	✗	1 ms	1 ms
KairosDB	✓	1 ms	1 ms	1 ms
NewTS	✓	2 ms	1 ms	1 ms
OpenTSDB	✓	1 ms	1 ms	> 1 ms[41]
Rhombus	✗	✗	1 ms[42]	1 ms[42]
Group 2: TSDBs with no Requirement on any DBMS				
Druid	✓	1 ms	1 ms	1 ms
Elasticsearch	✗[43]	1 ms	1 ms	1 ms
InfluxDB	✓	1 ms	1 ms	1 ms
MonetDB	✓[44]	1 ms[44]	1 ms[45]	1 ms[45]
Prometheus	✓[46]	1 ms	1 ms	1 ms
Group 3: RDBMS				
MySQL Community Server	✓[44]	1 ms[44]	1 ms[45]	1 ms[45]
PostgreSQL	✓[44]	1 ms[44]	1 ms[45]	1 ms[45]

Tab. 6: Comparison of Criteria Group 4: Granularity.

[28] Filtering for tag values cannot be used in combination with time ranges or aggregation functions, which results in a limited tag functionality, see Sect. 6.

[29] Boolean algebra is only poorly supported, which results in a limited tag functionality, see Sect. 6.

[30] Druid uses an immediate "rollup" to store ingested data at a given granularity which helps for long-term storage but there are no further "rollups" after the initial "rollup". This can be used in combination with rules for data retention and kill tasks [Ya15] to achieve long-term storage.

[31] Elasticsearch's aggregation functions can be adopted to do this [El16b].

[32] Elasticsearch does optimize long-term storage while data is ingested, which can also be started manually by curator [Ba15]. There is no possibility to run individual long term storage functions (e. g., to reduce granularity).

[33] Assuming that "timestamp" can be used twice as properties with different names [Ba15].

[34] See [In16d].

[35] Using continuous queries for downsampling and retention policies [Be15; In16d].

[36] Via triggers or views.

[37] Using two timestamp columns.

[38] By using recording rules [Pr16c].

[39] Called "labels" in Prometheus [Pr16b].

[40] Long-term storage is not yet supported by Prometheus [Pr16i].

[41] OpenTSDB does not guarantee one millisecond or any other granularity to be stored safely [La16b]. Our measurements showed that OpenTSDB can lose values when using one millisecond as granularity and does not lose values when using 1 second as granularity.

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

[43] Using Elasticsearch's histogram aggregation function [El16c].

[44] Can be implemented in SQL with a WHERE ... AND ... BETWEEN query.

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

[46] Can only be done with HTTP API's range queries [Pr16g].

TSDB	**APIs and Interfaces**	**Client Libraries**	**Plugins**
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Blueflood	Command-Line Interface (CLI)[47], Graphite, HTTP(JSON), Kafka, statsD[48], UDP	✗	✗
KairosDB	CLI, Graphite, HTTP(REST + JSON, GUI), telnet	Java	✓
NewTS	HTTP(REST + JSON, GUI)	Java	✗
OpenTSDB	Azure, CLI, HTTP(REST + JSON, GUI), Kafka, RabbitMQ, S3, Spritzer	✗[49]	✓
Rhombus	✗	Java	✗
Group 2: TSDBs with no Requirement on any DBMS			
Druid	CLI, HTTP(REST + JSON, GUI), Samza[50], Spark[50], Storm[50]	Java[50], Python, R	✓
Elasticsearch	HTTP(REST+ JSON)	Groovy, Java, .NET, Perl, PHP, Python, Ruby	✓
InfluxDB	Collectd[51], CLI, Graphite[51], HTTP(InfluxQL, GUI), OpenTSDB[51], UDP	✗	✓
MonetDB	CLI	Java (JDBC), Mapi (C binding), Node.js, ODBC, Perl, PHP, Python, Ruby	✗
Prometheus	CLI[52], HTTP(JSON, GUI)	Go, Java, Python, Ruby	✗
Group 3: RDBMS			
MySQL Community Server	CLI	J, Java (JDBC), ODBC, Python, and more	✓
PostgreSQL	CLI	C, C++, Java (JDBC), ODBC, Python, Tcl, and more [53]	✓

Tab. 7: Comparison of Criteria Group 5: Interfaces and Extensibility.

[47] A set of executable Java classes.
[48] Experimental feature.
[49] There are many clients made by other users.
[50] Via Tranquility.
[51] Via plugin.
[52] Not for querying [Ra16a].
[53] Some of them are third-party software.

TSDB	**LTS/Stable Version**	**Commercial Support**	**License**
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Blueflood	✗	✗	Apache 2.0
KairosDB	✗	✗	Apache 2.0
NewTS	✗	✗	Apache 2.0
OpenTSDB	✗	✗	LGPLv2.1+, GPLv3+
Rhombus	✗	✗	MIT
Group 2: TSDBs with no Requirement on any DBMS			
Druid	✗[54]	✗	Apache 2.0
Elasticsearch	(✓)[55]	✓[56]	Apache 2.0
InfluxDB	✗	✓[57]	MIT
MonetDB	✗	✓[58]	MonetDB Public License Version
Prometheus	✗[59]	✗	Apache 2.0
Group 3: RDBMS			
MySQL Community Server	✓[60]	✓[61]	GPLv2
PostgreSQL	✓[62]	✗[63]	The PostgreSQL License

Tab. 8: Comparison of Criteria Group 6: Support and License.

8 Conclusion and Outlook

This paper presented a systematic search for TSDBs resulting in 83 found TSDBs, whereby 50 are open source TSDBs. These are grouped into three groups and representatives of each are chosen by popularity and presented more detailed. The twelve chosen representatives are compared in 27 criteria (grouped in six criteria groups). From that comparison (Sect. 7) it can be concluded that no TSDB supports all features from any of the criteria groups. However, three of them are most promising candidates for our setting. When features do not distinguish TSDBs, the performance might. Having enterprise-ready performance is a crucial step from marked readiness to enterprise readiness. As a consequence, our next step is a repeatable, extensible, and open source benchmarking framework for arbitrary TSDBs.

[54] The "normal" releases are called "stable releases", but are not what is considered "stable" in this paper as every release is called "stable" after it has passed several release candidate stages.
[55] After leaving development and testing phase, Elasticsearch releases are named "stable releases". Updates for older versions are still released for an undefined amount of time after a new "stable version" was released.
[56] See [El16e].
[57] See [In16a].
[58] See [Mo16e].
[59] The API is considered stable since version 1.0.0 [Pr16d].
[60] See [Or16c].
[61] See [Or16b].
[62] See [PGD16c].
[63] Several commercial support companies are listed on the PostgreSQL homepage, see [PGD16b].

References

[Ac16a] Acreman, S.: Top10 Time Series Databases, 2016, URL: https://blog.dataloop.io/top10-open-source-time-series-databases.

[Ac16b] Acreman, S.: Write Performance Benchmark (Github Gist), 2016, URL: https://gist.github.com/sacreman/b77eb561270e19ca973dd5055270fb28.

[Ba15] Barnsteiner, F.: Elasticsearch as a Time Series Data Store, 2015, URL: https://www.elastic.co/blog/elasticsearch-as-a-time-series-data-store.

[Ba16] Bader, A.: Comparison of Time Series Databases, Diploma Thesis, Institute of Parallel and Distributed Systems, University of Stuttgart, 2016.

[Be15] Beckett, S.: InfluxDB - archive / rollup / precision tuning feature, 2015, URL: https://github.com/influxdb/influxdb/issues/1884.

[BF12] Blumsack, S.; Fernandez, A.: Ready or not, here comes the smart grid! Energy 37/1, pp. 61–68, 2012.

[Ca11] Cattell, R.: Scalable SQL and NoSQL Data Stores. SIGMOD 39/4, pp. 12–27, May 2011.

[DDL14] Date, C. J.; Darwen, H.; Lorentzos, N.: Time and Relational Theory. Morgan Kaufmann, 2014.

[DMF12] Deri, L.; Mainardi, S.; Fusco, F.: tsdb: A Compressed Database for Time Series. In: Traffic Monitoring and Analysis. Springer, 2012.

[El16a] Elasticsearch BV: Elasticsearch, 2016, URL: https://www.elastic.co/de/products/elasticsearch.

[El16b] Elasticsearch BV: Elasticsearch Reference [2.4] - Aggregations, 2016, URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html.

[El16c] Elasticsearch BV: Elasticsearch Reference [2.4] - Histogram Aggregation, 2016, URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-histogram-aggregation.html.

[El16d] Elasticsearch BV: Elasticsearch Reference [2.4] - Range Query, 2016, URL: https://www.elastic.co/guide/en/elasticsearch/reference/2.3/query-dsl-range-query.html.

[El16e] Elasticsearch BV: Elasticsearch - Subscriptions, 2016, URL: https://www.elastic.co/subscriptions.

[Ev16] Evans, E.: Newts, 2016, URL: https://github.com/OpenNMS/newts; http://opennms.github.io/newts.

[Gr13] Grolinger, K.; Higashino, W.; Tiwari, A.; Capretz, M.: Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing: Advances, Systems and Applications 2/1, 2013.

[In16a] InfluxData: InfluxData - Services, 2016, URL: https://www.influxdata.com/services/technical-support; https://www.influxdata.com/services/consulting.

[In16b] InfluxDB: InfluxDB, 2016, URL: https://influxdb.com.

[In16c] InfluxDB: InfluxDB – Clustering, 2016, URL: https://docs.influxdata.com/influxdb/v1.0/high_availability/clusters.

[In16d] InfluxDB: InfluxDB – Continuous Queries, 2016, URL: https://docs.influxdata.com/influxdb/v1.0/query_language/continuous_queries.

[In16e] InfluxDB: InfluxDB Relay, 2016, URL: https://docs.influxdata.com/influxdb/v1.0/high_availability/relay.

[Ka16] KairosDB Team: KairosDB, 2016, URL: https://github.com/kairosdb/kairosdb; http://kairosdb.github.io.

[Kl16] Klavsen, K.: Explaining a HA + scalable setup? (Github), 2016, URL: https://github.com/prometheus/prometheus/issues/1500.

[Ko15] Kopp, O.; Falkenthal, M.; Hartmann, N.; Leymann, F.; Schwarz, H.; Thomsen, J.: Towards a Cloud-based Platform Architecture for a Decentralized Market Agent. In: Informatik. GI e.V., 2015.

[La16a] Larsen, C.; Sigoure, B.; Kiryanov, V.; Demir, B. D.: OpenTSDB, 2016, URL: http://www.opentsdb.net.

[La16b] Larsen, C.; Sigoure, B.; Kiryanov, V.; Demir, B. D.: OpenTSDB - Writing Data - Timestamps, 2016, URL: http://opentsdb.net/docs/build/html/user_guide/writing.html#timestamps.

[Me15] Merdanović, E.: How to install KairosDB time series database?, Feb. 2015, URL: http://www.erol.si/2015/02/how-to-install-kairosdb-timeseries-database.

[Mo16a] MonetDB B.V.: MonetDB, 2016, URL: https://www.monetdb.org/Home.

[Mo16b] MonetDB B.V.: MonetDB – Data Partitioning, 2016, URL: https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/DataPartitioning.

[Mo16c] MonetDB B.V.: MonetDB - Distributed Query Processing, 2016, URL: https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/DistributedQuery-Processing.

[Mo16d] MonetDB B.V.: MonetDB - Transaction Replication, 2016, URL: https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/TransactionReplication.

[Mo16e] MonetDB Solutions: MonetDB Solutions Homepage, 2016, URL: https://www.monetdbsolutions.com.

[Or16a] Oracle Corporation: MySQL Community Server, 2016, URL: http://dev.mysql.com/downloads/mysql.

[Or16b] Oracle Corporation: MySQL Services, 2016, URL: http://www.mysql.com/services.

[Or16c] Oracle Corporation: MySQL – Which MySQL Version and Distribution to Install, 2016, URL: https://dev.mysql.com/doc/refman/5.7/en/which-version.html.

[PA13] Prasad, S.; Avinash, S.: Smart meter data analytics using OpenTSDB and Hadoop. In: Innovative Smart Grid Technologies-Asia (ISGT Asia). IEEE, 2013.

[Pa16] Pardot: Rhombus, 2016, URL: https://github.com/Pardot/Rhombus.

[PFA09] Pungilă, C.; Fortiş, T.-F.; Aritoni, O.: Benchmarking Database Systems for the Requirements of Sensor Readings. IETE Technical Review 26/5, pp. 342–349, 2009.

[PGD16a] The PostgreSQL Global Development Group: PostgreSQL, 2016, URL: https://www.postgresql.org.

[PGD16b] The PostgreSQL Global Development Group: PostgreSQL - Professional Services, 2016, URL: https://www.postgresql.org/support/professional_support.

[PGD16c] The PostgreSQL Global Development Group: PostgreSQL - Versioning Policy, 2016, URL: https://www.postgresql.org/support/versioning.

[Pr16a] Prometheus Authors: Prometheus, 2016, URL: http://prometheus.io.

[Pr16b] Prometheus Authors: Prometheus - Comparison to Alternatives, 2016, URL: https://prometheus.io/docs/introduction/comparison.

[Pr16c] Prometheus Authors: Prometheus – Defining Recording Rules, 2016, URL: https://prometheus.io/docs/querying/rules/#recording-rules.

[Pr16d] Prometheus Authors: Prometheus - FAQ, 2016, URL: https://prometheus.io/docs/introduction/faq.

[Pr16e] Prometheus Authors: Prometheus Federation, 2016, URL: https://prometheus.io/docs/operating/federation.

[Pr16f] Prometheus Authors: Prometheus Pushgateway, 2016, URL: https://github.com/prometheus/pushgateway.

[Pr16g] Prometheus Authors: Prometheus - Range Queries, 2016, URL: https://prometheus.io/docs/querying/api/#range-queries.

[Pr16h] Prometheus Authors: Prometheus – Range Vector Selectors, 2016, URL: https://prometheus.io/docs/querying/basics/#time-series-selectors.

[Pr16i] Prometheus Authors: Prometheus – Roadmap – Long-Term Storage, 2016, URL: https://prometheus.io/docs/introduction/roadmap/#long-term-storage.

[Ra15] Rackspace: Blueflood - FAQ, 2015, URL: https://github.com/rackerlabs/blueflood/wiki/FAQ.

[Ra16a] Rabenstein, B.: Promtool: Add querying functionality (Github), 2016, URL: https://github.com/prometheus/prometheus/issues/1605.

[Ra16b] Rackspace: Blueflood, 2016, URL: http://blueflood.io; https://github.com/rackerlabs/blueflood/wiki.

[Sc14] Schwartz, B.: Time-Series Database Requirements, 2014, URL: https://www.xaprb.com/blog/2014/06/08/time-series-database-requirements/.

[Sc15] Schwartz, B.: How We Scale VividCortex's Backend Systems, 2015, URL: http://highscalability.com/blog/2015/3/30/how-we-scale-vividcortexs-backend-systems.html.

[Sh14] Shahid, J.: Updating an existing point end up inserting, Second comment., Apr. 2014, URL: https://github.com/influxdb/influxdb/issues/391.

[St15] Strohbach, M. o.: Towards a Big Data Analytics Framework for IoT and Smart City Applications. In: Modeling and Processing for Next-Generation Big-Data Technologies. Springer, 2015.

[Te14] Ted Dunning Ellen, M. F.: Time Series Databases – New Ways to Store and Acces Data. O'Reilly Media, Inc, USA, 2014.

[Th01] Theo Härder, E. R.: Datenbanksysteme : Konzepte und Techniken der Implementierung ; mit 14 Tabellen. Springer-Verlag GmbH, 2001.

[Th15] Thomsen, J. et al.: Darstellung des Konzeptes – DMA Decentralised Market Agent – zur Bewältigung zukünftiger Herausforderungen in Verteilnetzen. In: INFORMATIK 2015. Vol. P-246. LNI, 2015.

[Vi14] VividCortex: Building a Time-Series Database in MySQL, 2014, URL: http://de.slideshare.net/vividcortex/vividcortex-building-a-timeseries-database-in-mysql.

[Wl12] Wlodarczyk, T.: Overview of Time Series Storage and Processing in a Cloud Environment. In: CloudCom. 2012.

[Ya14] Yang, F. et al.: Druid: A Real-time Analytical Data Store. In: SIGMOD. 2014.

[Ya15] Yang, F. et al.: Druid – Retaining or Automatically Dropping Data, 2015, URL: http://druid.io/docs/latest/operations/rule-configuration.html.

[Zi15] Zimmermann, O.; Wegmann, L.; Koziolek, H.; Goldschmidt, T.: Architectural Decision Guidance Across Projects – Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge. In: WICSA. IEEE, May 2015.

All links were last followed on 2016-12-01.