Université libre de Bruxelles

INFO-H-415: Advanced Databases
Winter Semester 2017 - 2018

Time series databases & OpenTSDB

Authors:

Keneth Ubeda, Matricule ULB: 000453317
Dagoberto Herrera Murillo, Matricule ULB: 000455058

Supervisor:
Esteban Zimányi

December 18, 2017

Contents

Introduction……………………………………………………………….3

Part 1: Theoretical Background

1. Time Series Data: Why Collect It?..4
2. Field of application……………………………………………….5
3. Storing and Processing Time Series……………………………..5
    a. Flat Files
    b. RDBMS: Why Relational Databases aren't enough?
    c. Time Series Databases: What's the difference?
4. Alternative designs for time series databases…………………..…8

Part 2: Hands-on OpenTSDB

1. Real-life application example: General Election United Kingdom 2017 on Twitter …………………………………………10
2. Installation and configuration……………………………………10
3. Naming schema……………………………………………………11
4. Writing Data……………………………………………………….12
5. Query components…………………………………………………13
6. Representative queries…………………………………………..…14
7. OpenTSDB vs. relational databases (MS SQL Server)………16
8. OpenTSDB vs. other time series databases …………………..…19

Conclusion……………………………………………………………..22
References……………………………………………………………23
Appendix…………………………………………………………………25

Introduction

The need to store time series data is not new. However, in recent years the availability of this type of data has become omnipresent thanks to the irruption of IoT, sensors and web transactions. For that reason, new methods have emerged for building time series databases that able to handle massive amounts of data.

This report examines available alternatives for storing and processing time series. It does not focus on methods for analyzing time series. Nor is it an extensive review of the topic of time series data storage. Instead, it explores some of the key issues associated with new types of time series databases and describes general scenarios of application of this type of data.

This document is structured in two sections. The first part deals with the theoretical aspects related to time series databases. The second part of the report describes the hands-on experience of OpenTSDB, an open source time series database which can store and serve massive amounts of single value time series data. The idea was to evaluate this product comparing its performance against a traditional relational database using temporal data from Twitter.

Part 1: Theoretical Background

1. Time Series Data: Why Collect It?

“A time series is a collection of observations made sequentially through time” (Chatfield, 2016). In terms of data is translated into repeated measures of some parameter, commonly a number, accompanied by the timestamps at which the measurements were made. Measurements are usually performed at regular intervals, although this is not a requirement.

According to Dunning et al. (2014), time series datasets are typically used in circumstances in which measurements, once made, are not revised or updated. Measurements accumulate as new data is added for each parameter at each new time point. It is important to mention these characteristics because they have a direct impact on the design of the technology for storing and processing time series.

The need to systematically accumulate data of this nature can be explained by the variety of questions that can be addressed by time series data. Here’s a list of representative cases (Dunning et al., 2014):

1. What are the short- and long-term trends for some measurements?
2. How do several measurements correlate over a period of time?
3. What can be expected about the future behavior of a measurement from its previous patterns?
4. What measurements might indicate the cause of some event?

2. Field of application

Kulkarni (2017) describes some of the use cases in which the effective manipulation of time series data is becoming a strategic asset:

a) Monitoring physical systems: connected devices, machinery, and equipment.
b) Financial trading systems: Classic securities and cryptocurrencies.
c) Monitoring software systems: applications, services, virtual machines, and containers.
d) Asset tracking applications: Vehicles, trucks and physical containers.
e) Eventing applications: user / customer interaction data.
f) Business intelligence tools: Tracking key performance indicators.

3. Storing and Processing Time Series

The idea of this section is to briefly review three alternatives for storing time series, starting from the simplest concept of flat file to databases optimized for time-stamped data.

a. Flat Files

Flat files are the simplest storage option for time series data. This mechanism can work effectively to the extent that the number of time series is relatively small. Such is the case of Parquet, a columnar file format that helps to compress the data and has an encoding scheme to handle complex data (Dunning et al., 2014).

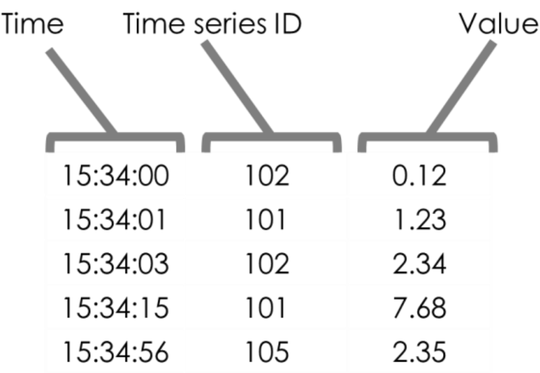
As the system grows, problems begin to appear, especially in those cases where it is necessary to increase the number of time series stored in the same file. In the previous scenario, the proportion of usable data for any query declines, because most of the data read belongs to other

time series. One way to solve this problem is to restrict the number of series per file, which produces many small files. A system composed of many files may cause stability problems and can be inefficient due to the increased seek time needed. It is concluded that flat files are not capable of handling large-scale time series data (Dunning et al., 2014).

b. RDBMS: Why Relational Databases aren't enough?

To avoid the problems associated with flat files, a natural step is to move to some form of a real database. The star schema is an alternative for storing time series in a relational database. In this model, the core data is stored in a fact table with references to other tables known as dimensions. A fundamental design assumption is that the dimension tables are relatively small and invariable. In a typical time series fact table, the only dimension referenced is the one that provides the details about the time series themselves, including the stored value (Dunning et al., 2014).

For instance, if our time series is coming from the air quality monitoring stations of the European Union (European Environment Agency, 2017), it might be expected that several values of air pollutants would be measured on each monitoring station such as mg/m^3 of carbon monoxide, ng/m^3 of arsenic, ng/m^3 of nickel, and so on. Each of these measurements for each station would constitute a separate time series, and each time series would have information such as the station name, country, type of area, longitude, latitude, and altitude stored in a dimension table. In such a database design, the core data is stored in a fact table that looks like what is shown in Figure 1.



Time	Time series ID	Value
15:34:00	102	0.12
15:34:01	101	1.23
15:34:03	102	2.34
15:34:15	101	7.68
15:34:56	105	2.35

Figure 1. A fact table for a time series in an RDB. The time, a time series ID and a value are stored. Detailed information about the series is stored in the dimension tables.

When compared to the flat file, a star schema solves the issue of having lots of different time series and can work effectively well up to levels of hundreds of millions of data points. However, this capacity is well below the mass time series generated today. For instance, British oil and gas company BP monitors thousands of oil wells with sensors. Each well has 20 to 30 sensors to measure pressure and temperature. This translates into a transmission rate of 500,000 data points every 15 seconds, which is equivalent to 1,051,200 million of data points per year (Winig, 2016).

As data scales continue to grow, a larger proportion of time series applications just don't fit very well into RDBMS. Moreover, storing the data is only part of the problem; retrieving it and processing it represent other challenges. Contemporary systems such as machine learning applications or status display systems may need to retrieve and process millions of data points in real time (Dunning et al., 2014).

Bader, Kopp, and Falkenthal (2017) compared the performance of two relational database systems (MySQL and PostgreSQL) against ten database systems specifically designed to manage time series. According to the authors' review, relational databases can compute all

the basic functions associated with time series, but they are ineffective in managing distribution/clusterability conditions. That means, they do not offer scalability, as do the time series databases.

c. Time Series Databases: What's the difference?

In the last years, a vast amount of Time Series Databases (TSDB) technologies have surged. They are design for storing and querying time series data. The differences between traditional databases and time series databases can be seen in two levels: scale and usability.

Time-series databases (based on relational databases or NoSQL) handle scale by introducing performance improvements: higher insertion rates and better data compression (Freedman, 2017), which is a vital feature when taking into account that time series data accumulates quickly.

The second factor is usability. This includes functions and operations that are common to time-series data analysis, including arithmetic expressions, string operations, aggregate functions, resampling into different time resolutions, ordering, ranking and limiting (Bader, Kopp, & Falkenthal, 2017). Expressions and operators may refer to different series, e.g. sum up all time series whose tags match a specific pattern.

TSDBs can be categorized into four groups. The first group is composed of TSDBs dependent on an existing DBMS (HBase, Cassandra) to store the time series data. The second group covers TSDBs using DBMS for storing meta data. The third group comprises RDBMS. The last group contains all TSDBs that are not open source (Bader, Kopp, & Falkenthal, 2017).

4. Alternative designs for time series databases

Time series databases generally have a design based on wide tables, blobs or hybrid designs that combine the two previous alternatives.

One of the disadvantages of the relational model is that it uses one row for each data point. A strategy to increase the speed at which data can be retrieved from a time series database is to store multiple values in each row, this design is known as a wide table. The speed of data recovery improves because the overhead associated with reading row by row is minimized. Some NoSQL databases (e.g. HBase) support an indefinite number of columns for a specific row (Dunning et al., 2014). The design of a wide table can be improved by compressing all of the data for a row into a data structure known as a blob.

Part 2: Hands-on OpenTSDB

OpenTSDB is a distributed and scalable time series database built on top of Hadoop and HBase. The data schema is highly optimized for fast aggregations of similar time series to minimize storage space.

1. Real-life application example: General Election United Kingdom 2017 on Twitter

OpenTSDB was used to model time series that correspond to the count of mentions on Twitter to terms of interest related to the elections in the United Kingdom that were held in 2017. The specific objective of the exercise was to monitor the mentions associated with the two main parties that contended in that election (Conservative & Labor) and the mentions to issues of national relevance during the process. The dataset used (Mak, 2017) contains 8.5 million tweets published between June 1 and June 8 -Election Day- 2017.

2. Installation and Configuration

To run OpenTSDB, itȷs necessary to meet the following requirements:

- ✓ A Linux system (or Windows with manual building)
- ✓ Java Runtime Environment 1.6 or later
- ✓ HBase 0.92 or later
- ✓ GnuPlot 4.2 or later

The first step of the installation is to setup HBase and Zookeeper. Once HBase is running, it is possible to choose an installation from a package (available in Github), from source using GIT or a source tarball. The last step of the installation is to create the necessary HBase tables. OpenTSDB can be configured via a file on the local system, via command line arguments or a combination of both (OpenTSDB User Guide, 2017).

OpenTSDB uses a design similar to a wide table, where rows containing data from a single time series are stored close to the disk. This structure implies that the need for large sequential disk operations is minimized when it is required to access data from a particular time series, contrary to what would happen if the rows were widely scattered. Naturally, the ability to capitalize on this benefit depends on whether the number of observations in each time series is large enough to cause a significant reduction in the number of rows retrieved (Dunning et al., 2014).

3. Naming schema

In OpenTSDB each time series has a generic ”metric” name, that can be shared by many unique time series. In the case study, the **geuk2017** metric identifies the number of times a term of interest is mentioned within the text of the tweets that were published in a second particular. What distinguishes a specific time series from another is the use of ”tags”, every time series must have at least one tag. In our example, a first tag called **topic** was created to identify the major thematic categories of interest (Labor, Conservative and Issues). Subsequently, a second tag called **keyword** was defined to identify the terms of interest according to the topic. Within the topics of the parties, it was decided to track three keywords: one for the name of the party, another for the name of the candidate and another for the campaign slogan. In the topic called Issues, it was decided to monitor three keywords (BREXIT, NHS and Terrorism). The above gives a total of 9 unit time series. Table 1 shows the hierarchical structure that relates the metric and tags (geuk2017, topic and keyword).

Table 1. Hierarchical structure between metric and tags

Metric˙uid	Topic	Keyword
geuk2017	Conservative	Conservative
		Theresa-May
		Strong-and-stable-leadership

	Labour	Labour
		Jeremy-Corbyn
		For-the-many-not-the-few
	Issues	BREXIT
		NHS
		Terrorism

The uniqueness of a specific time series comes from a combination of tag key / value pairs that allows for flexible queries with very fast aggregations. Following with the example where the metric was geuk2017. If it is necessary to know the data that corresponds to a specific keyword, the following query is enough: **sum: geuk2017 (topic = Labor, keyword = Jeremy˙Corbyn)**. If on the other hand, it is necessary to know the aggregate result of all the keywords, the query would be as simple as **sum: geuk2017**.

The underlying data schema will store all the geuk2017 time series next to each other so that aggregating the individual values is very fast and efficient. OpenTSDB was designed to make these aggregate queries as fast as possible since most users start at a high level, then drill down for detailed information.

4. Writing Data

There are three methods to write data into OpenTSDB: Telnet API, HTTP API and batch import from a file. A load line includes the name of the metric, the timestamp and the sequence of tags defined to identify the specific time series to which it belongs. An example is shown below:

```
Metric,_uid, timestamp, value, tag1, tag2, ... tagk
 geuk2017 1496275200 2 topic=Labour Keyword=Jeremy_Corbyn
```

5. Query Components

OpenTSDB provides a number of tools for query specifications: filtering, aggregation and downsampling. The components of a characteristic query can be seen in the following table.

Table 2. Query components

Parameter	Type	Required	Description	Example
Start Time	String or Integer	Required	Starting time for the query: absolute or relative time.	1496275200
End Time	String or Integer	Optional	An end time for the query. If not supplied, the current time on the TSD will be used.	1496357922
Metric	String	Required	The full name of a metric in the system (case sensitive).	geuk2017
Aggregation Function	String	Required	A mathematical function to use in combining multiple time series.	sum
Filter	String	Optional	Filters on tag values to reduce the number of time series picked up in a query or group and aggregate on various tags.	Topic=*, kewyword=BREXIT
Downsampler	String	Optional	An optional interval and function to reduce the number of data points returned across time.	1h

Functions	String	Optional	Data manipulation functions.	highestMax(...)
Expressions	String	Optional	Data manipulation functions across time series such as dividing one series by another.	(r1 / (r1 + r2)) * 100

Source: OpenTSDB User Guide, 2017

Relative (24h-ago) or absolute (1496275200) date and time formats are supported when querying for data. Relative time is specified in units (milliseconds, seconds, minutes, hours, days, weeks, months, years). Absolute time is represented with a Unix style timestamp. Queries using timestamps support millisecond precision appending three digits.

6. Representative queries

Query 1 - All Time Series for a Metric

This is the simplest query. OpenTSDB will scan all data points for the metric geuk2017 between the initial timestamp and final timestamp. The result will be a single dataset with all time series added together into one series.

```
m=sum:geuk2017
```

Query 2 - Filter on a Tag

This query filters time series that contain a tag combination (tags are put in curly brackets.

```
m=sum:geuk2017(topic=Labour)
```

This query will return an aggregate of time series with all the keywords related to the topic called Labour.

Query 3 - Specific Time Series

To retrieve the data of a specific time series, it is necessary to include all the tags that identify it.

```
m=sum:geuk2017(topic=Labour, kewyword=Jeremy_Corbyn)
```

To identify the number of times the term Jeremy Corbyn is mentioned, the query must contain the correct and complete combination of topic and keyword.

Query 4 - Grouping

The asterisk (*) is a grouping operator that will return a dataset for every time series that includes the given metric and the given tag name.

```
m=sum:geuk2017(Topic=*)
```

This query will return to an aggregate of all time series that correspond to any of the topics.

Query 5 - Group and Filter

The dataset generated by the grouping operator is filtered by specifying a subsequent tag.

```
m=sum:geuk2017(Topic=*,kewyword=Jeremy_Corbyn)
```

In this query, the time series of the topical labels are filtered by a second label called Jeremy Corbyn.

Query 6 - Grouping With OR

The pipe operator | allows to choose a few tag values.

```
m=geuk2017(topic=Labour|Conservative)
```

his query returns the sum of the time series where the topic is Labour or Conservative.

7. OpenTSDB vs. relational databases (MS SQL Server)

This section measures the performance of OpenTSDB with respect to a traditional relational database management system (MS SQL). The comparison parameter selected was the execution time of the queries. It is important to emphasize that in OpenTSDB the naming schema is determined by the simple definition of tags. In contrast, MS SQL has a star schema where the fact table contains a measure, called counts, that has references to dimensions corresponding to the tags. The performance measurements were executed in a system with the following characteristics:

-Linux system 16.04 LTS
-4 GB RAM
-4 processors 2.4 GHZ (Core i7-3630QM)
-64 bits

Subsequently, an SQL version of the 6 types of queries supported by OpenTSDB was generated to compare the execution times (the list of equivalent queries in SQL can be found in Appendix 1). In order to evaluate scalability, each query was executed in datasets of increasing size 10,000, 100,000 and 1,000,000 records respectively. The built-in SET STATISTICS TIME ON statement was used to measure the elapsed time for queries in MS SQL Server. It displays the number of milliseconds required to parse, compile, and execute each query. In the

case of OpenTSDB, the difference in milliseconds between the start and end time of the execution was computed with the queryStartTimestamp and queryCompletedTimestamp metrics (Figure 2).

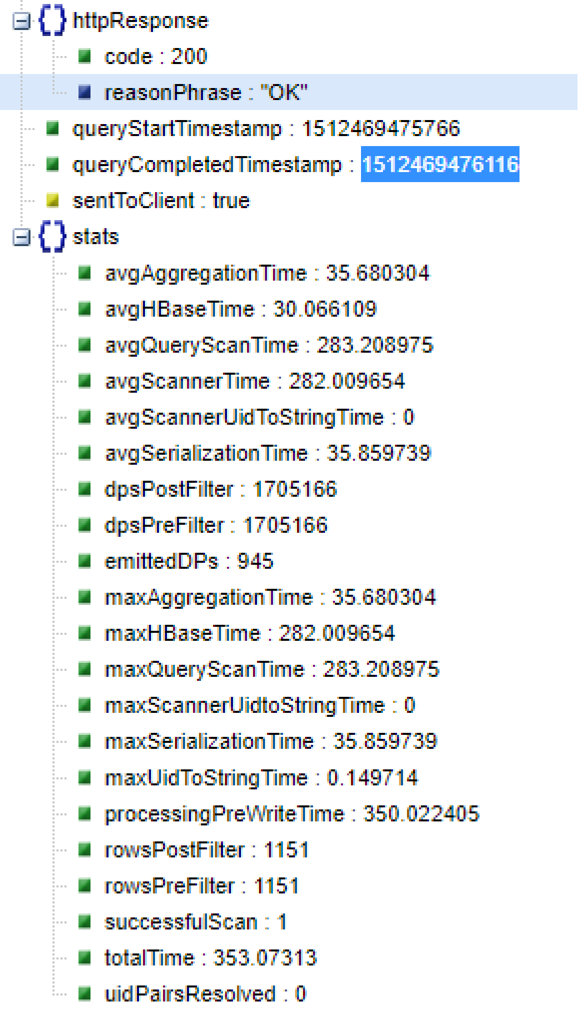


Figure 2. Query execution details OpenTSDB

To counteract the effect of the variation, every test was repeated 6 times, the result of the first run was discarded and the average of the five remaining runs was calculated. Tables 3-5 illustrate the results obtained. The first column represents the execution time for queries in OpenTSDB, the second column contains the execution time with SQL Server and the last column shows the ratio between both values.

Table 3. Elapsed time in ms, 10,000 records

	OPENTSDB	Microsoft SQL Server	Ratio
Q4: Grouping	18	289	16
Q5: Group and Filter	9	109	13
Q3: Specific Time Series	10	118	12
Q1: All Time Series for a Metric	13	128	10
Q2: Filter on a Tag	13	117	9
Q6: Grouping With OR	17	126	7

Table 4. Elapsed time in ms, 100,000 records

	OPENTSDB	Microsoft SQL Server	Ratio
Q4: Grouping	41	1,811	44
Q3: Specific Time Series	12	309	27
Q5: Group and Filter	11	224	21
Q2: Filter on a Tag	18	303	17
Q6: Grouping With OR	26	444	17
Q1: All Time Series for a Metric	30	409	13

Table 5. Elapsed time in ms, 1,000,000 records

	OpenTSDB	Microsoft SQL Server	Ratio
Q4: Grouping	452	16,029	35
Q3: Specific Time Series	104	2,346	23
Q5: Group and Filter	108	1,669	15
Q2: Filter on a Tag	214	2,376	11
Q6: Grouping With OR	355	3,872	11
Q1: All Time Series for a Metric	442	3,314	8

When comparing the execution times of the queries it can be concluded that OpenTSDB is faster in the three specified scenarios and for each one of the queries. The grouping operation was the one that made the biggest difference (35 times faster). The equivalent of this query in SQL must specify searches of all tags of the same type, which could explain its longer duration.

Figure 3 displays the ratios that result from dividing the execution time in OpenTSDB between the execution time in SQL Server. By moving from the load from 10,000 records to 100,000 records, performance improves. The same does not happen when moving from 100,000 records to 1,000,000 records, where performance remains relatively stable.

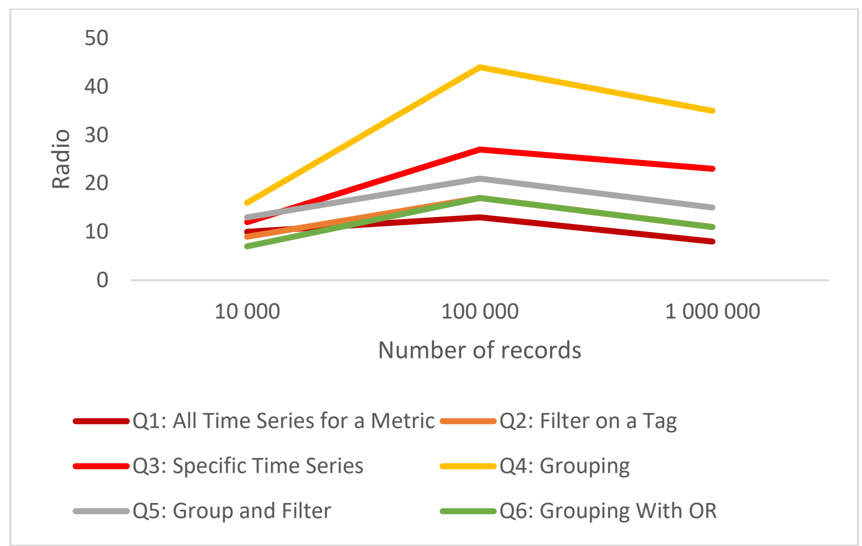


Figure 3. Ratio of execution OpenTSDB/SQL Server

To provide a comparison of compression capabilities, the space used by the relational model of the fact table versus OpenTSDB with 1,000,000 records was measured. OpenTSDB required just under a sixth of the space consumed by its counterpart in SQL Server. The detail can be seen in Figure 4.

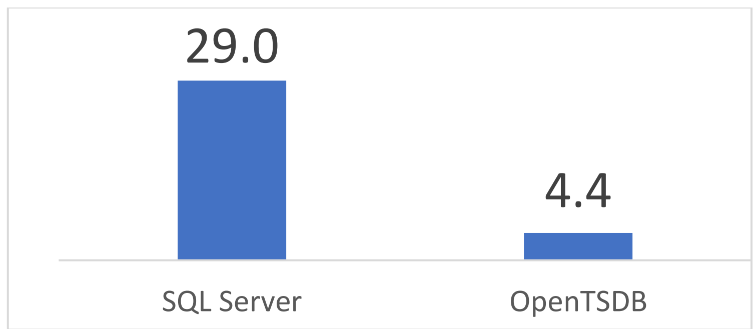


Figure 4. Data space 1,000,000 records (MB)

8. OpenTSDB vs. other time series databases

Bader, Kopp, and Falkenthal (2017) conducted a comparison of Open Source Time Series Databases. In this exercise, ten different alternatives divided into two groups were compared. The first group contains TSDBs with a requirement on other DBMS and the second TSDBs with no requirement on any DBMS. OpenTSDB belongs to the

first category because it uses HBase for storing its time series data. HBase, in turn, uses Zookeeper for coordination between nodes.

OpenTSDB, like most of its counterparts, offers high availability (compensation against unexpected node failures), scalability (increased storage or performance by adding more nodes) and load balancing (equally distribute queries across nodes). All basic query types are supported, except update and delete. As mentioned previously, OpenTSDB allows downsampling and offers tags to track the origin of a specific time series and matrix time series (more than one dimension). As in all compared systems, one millisecond is the smallest storage granularity implemented.

From that comparison, the authors concluded that there is are no features supported by all TSDBs. Druid is the best alternative if all criteria besides having a stable version and commercial support are required. In other cases, InfluxDB or MonetDB, can be a good choice when commercial support is needed. The detail of the comparisons can be reviewed in Appendix 2.

Acreman (2017) published a post on the blog of the technology company Outlyer to compare time series databases regarding writing performance in a single node. At the top of the list are technologies such as DalmatinerDB (3 million metrics / s) and Akumuli (2 million metrics / s). In the middle part, there are names like Prometheus (800k metrics / s), InfluxDB (470k metrics / s) and Graphite (220k metrics / s). And finally, there are OpenTSDB (32k metrics / s), ElasticSearch (30k metrics / s) and Druid (25k metrics / s).

Finally, the benchmark made by Persen (2016) to compare InfluxDB and OpenTSDB is included. The criteria used were on-disk compression and query performance. In the first criterion, InfluxDB outperformed OpenTSDB by delivering 16.5x better compression.

While in the second factor, InfluxDB outperformed OpenTSDB by delivering a minimum of 4.0x better query throughput.

Conclusion

This report analyzed OpenTSDB, a database for distributed and scalable time series that allows collecting, storing and serving millions of data points with no loss of precision.

Based on the conceptual definitions of the differences between relational databases and time series databases, an application case was developed with data from Twitter to evaluate the performance of both technologies. The execution times for similar queries were tested using OpenTSDB and a fact table in MS SQL Server, which retrieves the same information as the queries in our application. As a result, OpenTSDB exceeds MS SQL in all tested queries with increasing performance as the size of the processed data set grows. It was also possible to verify that OpenTSDB consumes a fraction of the space used by SQL Server to model the same scenario.

The literature review and industry benchmarks suggest that OpenTSDB is below other time series databases. Therefore, it is interesting to observe how a representative of the family of TSDBs, which potentially does not have a maximum performance, clearly shows its benefits against a conventional relational model when dealing with time series data.

The benefits of a time series database are not limited to performance and data compaction, but also to the availability of functions specially designed for the manipulation of time series. A characteristic case is downsampling, a function used to change the temporal resolution of the series, which is difficult to replicate in other platforms. We must also take into account the wide availability of visualization tools

complementary to the time series databases, as is the case of Grafana, which can be exploited for the design of dashboards and alert systems.

References

Acreman, S. (2017). Time-Series Database Benchmarks. Retrieved from https://blog.outlyer.com/time-series-database-benchmarks

Bader, A., Kopp, O., & Falkenthal, M. (2017). Survey and Comparison of Open Source Time Series Databases. BTW (Workshops) 2017: pp 249-268. Retrieved from http://btw2017.informatik.uni-stuttgart.de/slidesandpapers/E4-14-109/paper˙web.pdf

Chatfield, C. 2016. The Analysis of Time Series: An Introduction. Chapman and Hal. Retrieved from https://books.google.be/books?id=qKzyAbdaDFAC&printsec=frontcover&dq=time+series+pdf&hl=en&sa=X&ved=0ahUKEwiW9J˙AhvXWAhWGOhoKHQqWCDIQ6AEIJjAA#v=onepage&q=time%20series%20pdf&f=false

Dunning, T., Friedman, E., Loukides, M.K., Demarest, R. (2014). Time Series Databases: New Ways to Store and Access Data. O'Reilly Media, Sebastopol.

European Environment Agency. (2017). AirBase - The European air quality database [Data file]. Retrieved from https://www.eea.europa.eu/data-and-maps/data/airbase-the-european-air-quality-database-7#tab-figures-produced

Freedman, M. (2017). Time-series data: Why (and how) to use a relational database instead of NoSQL. Retrieved from: https://blog.timescale.com/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c

Kulkarni, A. (2017). Why do I need a time-series database)? Retrieved from: https://blog.timescale.com/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563

Mak, W. (2017). Tweets collected using the streaming api with terms related to the 2017 UK general election (jun-01 to jun-05). Retrieved from https://data.world/wwymak/uk-election-tweets-2017-june-1

Mak, W. (2017). Tweets collected using the streaming api with terms related to the 2017 UK general election. Retrieved from https://data.world/wwymak/uk-election-tweets-2017-june-2

OpenTSDB. (2017). User Guide. Retrieved from: http://opentsdb.net/docs/build/html/user˙guide/

Persen, T. (2016). InfluxDB Markedly Outperforms OpenTSDB in Time Series Data & Metrics Benchmark. Retrieved from https://www.influxdata.com/blog/influxdb-markedly-outperforms-opentsdb-in-time-series-data-metrics-benchmark/

Winig, L. (2016). GE'S big bet on data and analytics: seeking opportunities in the Internet of Things, GE expands into industrial analytics. MIT Sloan Management Review. Retrieved from: http://sloanreview.mit.edu/case-study/ge-big-bet-on-data-and-analytics/

Appendix 1

Queries of time series written in SQL

Query 1 - All Time Series for a Metric

```
SELECT t.METRIC, f.TIMESTMP, SUM(f.VALUE) as 'COUNT'
FROM Fact_TS_1M f, TS t
WHERE f.TS_ID = t.TS_ID
    AND t.METRIC = 'geuk2017'
GROUP BY t.METRIC,f.TIMESTMP
ORDER BY TIMESTMP;
```

Query 2 - Filter on a Tag

```
SELECT t.METRIC, t.TOPIC, f.TIMESTMP, SUM(f.VALUE) as 'COUNT'
FROM Fact_TS_1M f, TS t
WHERE f.TS_ID = t.TS_ID
    AND t.METRIC = 'geuk2017'
    AND t.TOPIC = 'Labour'
GROUP BY t.METRIC,t.TOPIC, f.TIMESTMP
ORDER BY TIMESTMP;
```

Query 3 - Specific Time Series

```
SELECT t.METRIC, t.KEYWORD, t.TOPIC, f.TIMESTMP, SUM(f.VALUE)
as 'COUNT'
FROM Fact_TS_1M f, TS t
WHERE f.TS_ID = t.TS_ID
    AND t.METRIC = 'geuk2017'
    AND t.TOPIC = 'Labour'
    AND t.KEYWORD = 'Labour'
GROUP BY t.METRIC,t.KEYWORD,t.TOPIC, f.TIMESTMP
ORDER BY TIMESTMP;
```

Query 4 – Grouping

```
SELECT t.METRIC, t.TOPIC, f.TIMESTMP, SUM(f.VALUE) as 'COUNT'
FROM Fact_TS_1M f, TS t
WHERE f.TS_ID = t.TS_ID
    AND t.METRIC = 'geuk2017'
    OR t.TOPIC = 'Labour'
    OR t.TOPIC = 'Coservative'
    OR t.TOPIC = 'Issues'
GROUP BY t.METRIC,t.TOPIC, f.TIMESTMP
ORDER BY f.TIMESTMP;
```

Query 5 - Group and Filter

```
SELECT t.METRIC, t.TOPIC, t.KEYWORD, f.TIMESTMP, SUM(f.VALUE)
as 'COUNT'
```

```
FROM Fact_TS_1M f, TS t
WHERE f.TS_ID = t.TS_ID
    AND t.METRIC = 'geuk2017'
    AND t.KEYWORD = 'Jeremy_Corbyn'
    AND(
    t.TOPIC = 'Labour'
    OR t.TOPIC = 'Coservative'
    OR t.TOPIC = 'Issues'
)
GROUP BY t.METRIC,t.TOPIC, t.KEYWORD,f.TIMESTMP
ORDER BY f.TIMESTMP;
```

Query 6 - Grouping With OR

```
select C.TIMESTMP, c.count+L.count as 'total'
from (
select f.TIMESTMP, t.TOPIC, SUM(f.VALUE) as 'count'
        from Fact_TS_1M f, TS t
        where f.TS_ID = t.TS_ID
            and t.TOPIC = 'Conservative'
        GROUP BY f.TIMESTMP,t.TOPIC
)C,
        (
    select f1.TIMESTMP, t1.TOPIC, SUM(f1.VALUE) as 'count'
        from Fact_TS_1M f1, TS t1
        where f1.TS_ID = t1.TS_ID
            and t1.TOPIC = 'Labour'
        GROUP BY f1.TIMESTMP,t1.TOPIC
)L
    WHERE C.TIMESTMP = L.TIMESTMP
UNION
    SELECT f.TIMESTMP, SUM(f.VALUE) as 'total'
    FROM Fact_TS_1M f, TS t
    where f.TIMESTMP not in(
select DISTINCT(C.TIMESTMP)
        from (
    select f.TIMESTMP, t.TOPIC, SUM(f.VALUE) as 'count'
            from Fact_TS_1M f, TS t
            where f.TS_ID = t.TS_ID
                and t.TOPIC = 'Conservative'
            GROUP BY f.TIMESTMP,t.TOPIC
)C,
            (
    select f1.TIMESTMP, t1.TOPIC, SUM(f1.VALUE) as 'count'
            from Fact_TS_1M f1, TS t1
            where f1.TS_ID = t1.TS_ID
                and t1.TOPIC = 'Labour'
            GROUP BY f1.TIMESTMP,t1.TOPIC
)L
```

```
        WHERE C.TIMESTMP = L.TIMESTMP)
        AND f.TS_ID = t.TS_ID
        AND (t.TOPIC = 'Labour' or t.TOPIC = 'Conservative')
    GROUP BY f.TIMESTMP
```

Appendix 2

Comparative tables of databases for time series by Bader, Kopp and Falkenthal (2017)

Table A. Comparison of Criteria Group 1: Distribution/Clusterability

TSDB	High availability	Scalability	Load Balancing
Group 1: TSDBs with a Requirement on NoSQL DBMS			
Bluefood	✓	✓	✓
KairosDB	✓	✓	✓
NewTS	✓	✓	✓
OpenTSDB	✓	✓	✓
Rhombus	✓	✓	✓
Group 2: TSDBs with no Requirement on any DBMS			
Druid	✓	✓	✓
Elasticsearch	✓	✓	✓
InfluxDB	✓	✗	✓
MonetDB	✓	✓	✓
Prometheus	✗	✓	✓

Table B. Comparison of Criteria Group 2: Functions

TSDB	NS	READ	SCAN	AVG	SUM	CNT	MAX	MIN	UPD	DE
Group 1: TSDBs with a Requirement on NoSQL DBMS										
Bluefood	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
KairosDB	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
NewTS	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
OpenTSDB	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
Rhombus	✓	✓	✓	✗	✗	✓	✗	✗	✓	✓
Group 2: TSDBs with no Requirement on any DBMS										
Druid	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
Elasticsearch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
InfluxDB	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

MonetDB	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Prometheus	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

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

TSDB	Continuous Calculation	Tags	Long-Term Storage	Matrix Time Series
Group 1: TSDBs with a Requirement on NoSQL DBMS				
Bluefood	✗	✗	✓	✗
KairosDB	✗	✓	✗	✗
NewTS	✓	✓	✗	✗
OpenTSDB	✗	✓	✗	✗
Rhombus	✗	✓	✗	✗
Group 2: TSDBs with no Requirement on any DBMS				
Druid	✓	✓	✓	✗
Elasticsearch	✓	✓	✗	✓
InfluxDB	✓	✓	✓	✗
MonetDB	✓	✓	✗	✓
Prometheus	✓	✓	✗	✗