Time Series Database (TSDB) Query Languages

Philipp Bende
philipp.bende@student.uni-luebeck.de

***Abstract*—Time series data becomes became more and more relevant since the so called 4th industrial revolution. A large amount of sensors are continuously measuring data, so called time series data. This paper aims to explain what time series data is and why it is relevant. It goes into detail of storing large amounts of time series data and the problems with conventional relational SQL-style databases. Further, so called time series databases (TSDBs), which are databases specialized for handling large quantities of time series data, are introduced. Requirements for designing an efficient TSDB, as well as different TSDB designs are presented. Finally a choice of multiple TSDBs is introduced, emphasizing on their design and how queries for data are performed on the example of OpenTSDB.**

I. Time series data

In this section we give the reader an overview of what time series data is, where and why one might be interested in it and a few real world example use cases of where time series data is accumulated and used.

Time series data plays a more and more important role in many areas such as finances, statistics or monitoring. One could for example want to observe many temperature, pressure, humidity, etc. sensors in a factory environment to control a production line. An other very common use case is monitoring servers by gathering data from their processor or network load.

Data gathered from these use cases often comes in the form of timestamp-value pairs. Such timestamp-value pair data, that is measured from the same sensors continuously or multiple times within a certain time window is called time series data.

Time series data often is high precision data, which is stored for long times to be later evaluated to calculate trends or to figure out which sensors might be correlated with a later discovered error by checking all sensor data for irregularities.

One can define time series data by the following rules:

Time series data

- is a sequence of numbers representing the measurements of a variable at equal time intervals.
- can be identified by a source name or id and a metric name or id.
- consists of {timestamp , value} tuples, ordered by timestamp where the timestamp is a high precision Unix timestamp (or comparable) and the value is a float most of the times, but can be any datatype.

II. Difference between TSDB and conventional databases

After explaining what time series data is in the previous chapter, this chapter focuses on illustrating the difference between conventional databases (read relational SQL databases) and specialized time series databases and how a TSDB concept might be specified in order to be realizable as a efficient database for handling time series data.

Time series data as described in the previous chapter can be stored in a conventional database. One would for example have a SQL-based relational database table with 3 columns for the sensor ID, the timestamp and the value as illustrated below.

s_id	time	value
s01	00:00:00	3.14
s02	00:00:00	42.23
s01	00:00:10	4.14
	⋮	
s01	23:59:50	3.25

Table I
SQL Table example on how time series data can be stored

As one can see in Table I measuring something once every 10 seconds, which is low for many time series applications, one day worth of measurements created 8640 rows in the table for every sensor.

It is not hard to imagine a setting where hundreds or thousands of sensors are observed in intervals as low as a few milliseconds. A conventional SQL table with millions to billions rows would be required.

Performing operations like Joins or Merges on on tables of that magnitude can cause mayor performance problems, especially if these operations need to be repeated when new data enters a table.

Another approach is to store time series data as binary large objects (BLOBs) in a conventional relational database. This means not creating an entry for each and every value, but to combine all values from a certain time period into a BLOB which is then stored as a single entry in the database. Storing blobs instead of single values cuts the required size of the database by a factor depending on how many single values are combined into each BLOB.
Using BLOBs of course requires the database supporting the entry of BLOBs.

Database joins across multiple sets of time series data BLOBs are only possible, if the time frame represented by each entry spans the same time for each BLOB in the database.

This can lead to problems, since not every sensor is observed in the same time interval. To be able to perform joins, the less frequently observed sensor data must be up-sampled or padded with zeros, which in turn can lead to more problems e.g. calculating averages or finding the exact point when a sensor value reached a certain threshold.

Time series databases on the other hand impose a model for handling immense amounts of time series data efficiently.
A TSDB typically consists of a data store back-end and a front-end providing the query capabilities and graphical representation. The back-end usually is a NoSQL database configured to handle huge amounts of data, while the front-end is specific for each TSDB. The front-end typically provides options not only to query for data, but also to display multiple time series in a easy to read way, such as graphs and trends.

As such a system it can be defined as follows:

A TSDB is a software system that is optimized for handling arrays of numbers indexed by time, datetime or datetime range. Since within a TSDB are multiple time series, which in turn consist of arrays of indexed values, the above mentioned optimizations allow a TSDB to handle time series data efficiently.

To design and build such a specialized system, it is paramount to know what kind of workload is expected to occur and which features are required to handle time series data efficiently.

Characteristic workload patterns for TSDBs are as follows:

The TSDB will mostly encounter writing transactions. It is common, that 95% to 99% of all workload consists of write operations.

These writes are almost always sequential appends to the end of the current array.

Writes to the distant past or distant future are extremely rare due to the nature of time series data. The Same is true for updates to already existing entries.

Deleting operations usually occur in bulk, meaning that when a delete happens, it usually does not delete single entries, but many consecutive arrays of values at once.

Reading transactions on the other hand happen rarely. When a read does happen, the requested data is usually much larger then the memory. This results in little or no useful caching options for read data.

Multiple reading transactions are usually sequential ascending or descending and reads of multiple series as well as concurrent reads are fairly common.

As time series data by default consists of large quantities of values, the scaling of a TSDB is important. For that a TSDB should be a distributed system, as such systems have better scalability then monolithic systems. Further the concept of “sending the query to the data” should be implemented in a TSDB, because of the high network load that transmitting large amounts of time series data to the query processor.

Table II shows how a TSDB design with wide tables might look like. Contrary to the SQL table shown in Table I, the TSDB table does not need to have a row for each measurement of each sensor, but only one for

s_id	start_time	t+1	t+2	t+3	...
s01	00:00:00	3	1	4	...
s02	00:00:00	42	23	1337	...
s01	01:00:00	4	2	5	...
s01	02:00:00	...	...	...	...
	⋮				
s01	23:00:00	...	...	...	...

Table II
SCHEMA OF A TSDB DESIGN WITH WIDE TABLES

s_id	start_time	t+1	t+2	+t3	...	compressed
s01	00:00:00					{...}
s02	00:00:00					{...}
s01	01:00:00					{...}
	⋮					
s01	22:00:00	42	23	1337	...	
s01	23:00:00	3	1	4	...	

Table III
SCHEMA OF A TSDB DESIGN WITH HYBRID TABLES

each time period per sensor.
Each row stores not a single, but a whole range of values. These are ordered in columns labeled with an offset equal to the measuring interval from a starting time. The starting time intervals are much larger then the measuring intervals. For example a starting time interval might be an hour or a day, while the measuring interval is a few milliseconds to seconds. Thus a single row in the database table can store hundreds or thousands of values.

One might question what use a TSDB has, if all it does differently then a conventional database is storing fewer but much larger rows in the database. The reason this is an improvement is, that it is much cheaper in terms of time and processing power to query for one row of a table and then continue reading lots of data, then finding and starting to read many rows, thus reducing the retrieval overhead by large amounts. The same is true for other common operations, such as joins, merges and deletes.

This is due to TSDBs storing data not as highly redundant time-value-pairs, but rather as single values in many columns that takes just one entry in the database per time frame. In the example TSDB table Table II, this time frame is one hour. This approach only needs 24 entries in the Database per day and sensor, which is much less than the multiple thousand entries required in a conventional database. Further measuring and storing more values per time frame does not increase the number of rows, that need to be inserted into the database. The value array would simply get larger in that scenario.

A further refined design option is displayed in Table III.

The hybrid design consists of a the wide row design displayed in Table II with an additional column. This additional column is reserved for compressed data. After a starting time interval lies in the past and it is highly unlikely that values in that interval will need to change, all values from that row can be compressed into a single BLOB as explained at the beginning of this chapter.

This compression of data into a BLOB has two major advantages over the wide row TSDB design. Firstly storing data in a compressed way takes less disk space and secondly it makes retrieval of data even faster then the wide table design, since only 1 column needs to be retrieved instead of a whole row that might consist of thousands of columns.
The compression itself does of course take processing time, but that is more then equalized by the reduced retrieval times and since time series data, that lies in the past is normally never changed or updated, there is no need to recrompress the data ever again.

After retrieval the data usually will be decompressed, but retrieval firstly happens rarely, and when it happens, one will likely want to retrieve whole rows anyways since single points of time series data are rarely useful. Further one can choose a compression method for the TSDB that takes only a small fraction of the time used for compression to decompress the data, since the compression only ever happens once, while the decompression can happen multiple times.

A third design option is the so called “Direct Blob Insertion” where each row consists only of three columns for sensor id, starting timestamp and value blob respectively, see Table IV.

Similarly to the Hybrid design, the data values of each time period is compressed and stored as a binary large object in the table. The difference is, that the single values are not stored in the database, but accumulated in memory and only compressed inserted into the database after the allotted time frame has passed. This has the advantages of smaller and thus easier to handle database tables and the same fast data retrieval as the hybrid

s_id	start_time	values
s01	00:00:00	{...}
s02	00:00:00	{...}
s01	01:00:00	{...}
	⋮	
s01	22:00:00	{...}
s01	23:00:00	{...}

Table IV
SCHEMA OF A TSDB DESIGN DIRECT BLOB INSERTION

design. The disadvantage is, that the data must be accumulated in memory. This obviously requires large enough memory space, so that all values of one time frame fit in memory or small enough and thus more time frames so that fewer values belong to each time frame.

The main reason for the "Direct Blob Insertion" design is, that in a wide row, or hybrid design, each insertion of a data point requires a row update operation on the database table. By having only one insertion operation per row, the data insertion rate can be increased up to thousand fold.

III. COMMONLY USED TSDBS

Quite a few ready-made TSDBs exist today. This paper focuses mostly on open-source variants. In this chapter a few choices for TSDBs are briefly introduced.

A. *OpenTSDB*

OpenTSDB is a open source Time series database which uses HBase as back-end data storage.

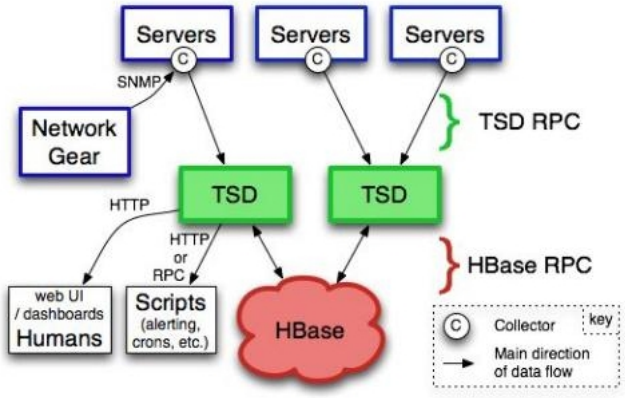

Figure 1. Visualization of the OpenTSDB architecture

Figure I shows a schematic representation of the OpenTSDB architecture.
The servers are being monitored by some statistics application. This data is send to a Time Series Daemon (TSD in Figure I). The time series data is send to the TSDs from the collectors (small Cs on the servers) where it is compressed and stored in the database, represented by the red HBase bubble, which acts solely as background storage. The TSDs are also responsible for accepting queries and retrieving data from the database.

OpenTSDB follows the design philosophy of "Direct Blob Insertion" as described in Chapter II, thus the back-end tables of HBase can be used and need to only very few columns as can be seen in the "Direct Blob Insertion" example in Table IV.

OpenTSDB offers the user a REST API and a Telnet interface for accessing the database. Both allow for the following actions in addition to the usual `GET`, `POST`, `PUT` and `DELETE`:

- **SELECT** by the sensor (called metric) name, time or values
- **GROUP BY** over multiple series by any selected property
- **DOWN-SAMPLING** it is common to have much higher precision data stored then it would be useful to visualize, thus one can retrieve a down sampled set of the time series data
- **AGGREGATE** functions like average, sum, min, max, etc
- **INTERPOLATE** the final results in desired intervals

Queries are usually including the following components:

- **Start Time** the earliest timestamp which is of interest
- **End Time** the latest timestamp which is of interest
- **Metric** the metric, or sensor name from which time series data is to be queried
- **Aggregation Function** possibly a function, what to do, or how to fetch the data
- **Tag** a tag that can further identify groups of relevant values
- **Downsampler** a mode to downsample the data if that is requested
- **Rate** the rate of which the values are supposed to be downsampled

Once a query reaches a TSD, the following steps are performed:

1) The query is parsed to check for syntax errors and that all metrics (sensor names), tag names and values exist.
2) The TSD sets up a scanner for the undelying storage.
3) If the query has no tags or tag values, then all rows fitting the requested metric and time stamp are fetched.
   If the query does have tags, only rows that match the tag in addition to the timestamp and metric are fetched.
4) The fetched data is organized into groups, if the **GROUP BY** fuction is requested.
5) Then the downsampling (if requested) of the data is performed.
6) Each group of data is aggregated by the requested aggregation function.
7) If a rate was set, the aggregates are adjusted to match the requested rate.
8) Finally the results are returned to the caller.

To insert data into OpenTSDB one would access the database via HTTP-API or Telnet as follows:

```
put <metric> <timestamp> <value>
<tag1=tagv1[tag2=tagv2 ...tagN=tagvN]>
```

For example:

```
put sys.cpu.user 123456 42.5
host=webserver01 cpu=0
```

To get from the database, one could send the following query in the format:

```
query START-DATE [END-DATE]
<aggregator> <metric> <tag1=tagv1[...]>
```

And as a concrete example:

```
query 24h-ago now
avg sys.cpu.user cpu=0
```

which would return the average value of cpu0 data of the last 24 hours.

Additionally to the REST API it is possible to access the database directly via the HBase API, but one should be mindful that the BLOB format of the data can make it more difficult to use conventional SQL-based tools.

Installing OpenTSDB requires the installation of HBase since OpenTSDB is built upon HBase. To visualize the results the results of queries it is advised to install a time series display tool like Grafana which works for multiple TSDBs.

B. InfluxDB

InfluxDB is a TSDB with no external dependencies, meaning one can put a precompiled binary file on a server and run it without having to install or configure anything else. InfluxDB is developed by InfluxData, Inc. Tho advertised as open-source, only the monolithic design of InfluxDB source code can be publicly accessed. The distributed and thus highly scaling cluster version of InfluxDB is closed-source commercial software.

Unlike OpenTSDB InfluxDB used to utilize the Google developed LevelDB as storage back-end but switched to a custom LSM-tree based approach.

Just like OpenTSDB InfluxDB offers a HTTP REST API for queries. It's language is the very SQL-like "Influx Query Language" which is basically SQL with a few extra query options like **GroupBy** and **TopN**. In addition InfluxDB accepts many TSDB protocols like the Graphite- or OpenTSDB-Protocol. Further it has multiple client libraries in many popular languages including Python, Java, Javascript or Ruby.

C. Gorilla

Gorilla is the TSDB behind Facebook. It is advertised as a fast, scalable in-memory TSDB. Unlike other TSDBs Gorilla aims to not write time series data to a storage, but to keep it in memory for faster queries and evaluation of the data.

In order to be able to store huge amounts of time series data in memory, Gorilla's main aim is to compress the data as much as possible. Of course it is not possible to continuously store time series data in memory. For that, Gorilla has a HBase based long term storage, where time series data older then 26 hours is moved.

By being able to store 26 hours worth of time series data in memory, the authors of "Gorilla: A Fast, Scalable,

In-Memory Time Series Database" [4] claim to be able to reduce query latency by a factor of 73 and query throughput by a factor 14, compared to other HBase based TSDBs.

D. Graphite

Graphite is a non-distributed TSDB that stores time series data on a local disk in a Round Robin Database style called Whisper. The size of the database of Graphite is predetermined which does not allow to accumulate more and more data over an unlimited time span. It stores each time series in a separate file and overwrites old files after a certain amount of time. Further data is expected to be time-stamped in regular intervals thus does not support jittered time series data.

This makes Graphite less disk space consuming than for example OpenTSDB which stores time series ad infinitum, making Graphite a good solution for time series data, where only the recent past is relevant.

An other popular feature of Graphite is the availability of superb graphing tools, that allow for a fast and easy visualization of stored time series data. Grafana, primarily developed for Graphite, allows for graphing and analyzing of time series data.

IV. Conclusion

In this paper time series data was introduced pairs of timestamp-value. Such pairs are called time series if the timestamp continuously grows in set intervals.

Storing time series data in conventional relational databases can lead to problem in performance due to the high volume of values that would require large tables and high insertion rates.

Time series databases (TSDBs) are (usually) NoSQL databases specialized for handling large amounts of time series data by allowing very high insertion rates as well as storing the data in such a way, that it is easy to query and work with the time series data.

There are multiple solutions for TSDBs ready-made, both commercially sold and open-source versions. All of those specialize in different aspects of handling time series data, but are all able to handle very high (thousands to millions per second per machine) of write-transactions and the efficient storage of time series data.

Just as conventional relation databases are not optimized for time series data, query languages like SQL are not optimal for querying time series data.

Specialized query languages for TSDBs add new functionality to SQL or SQL like query languages, allowing the efficient handling and querying of time series data.

Aggregate functions like downsampling or top n% as well as various visualization tools make it possible for a user to work efficiently with clearly arranged representations of billions of data points.

References

[1] Minsam Kim and Jiho Park *Time-series Databases* http://www.cse.ust.hk/~dimitris/5311/P2-TS.pdf Dec.2016

[2] Andreas Bader *Comparison of Time Series Databases* University of Stuttgart 2016-01-13 ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3729/DIP-3729.pdf Dec.2016

[3] InfluxData, Inc *InfluxDB Version 1.1 Documentation* https://docs.influxdata.com/influxdb/v1.1/concepts/key_concepts/ Dec.2016

[4] Tuomas Pelkone et al. Facebook, Inc. *Gorilla: A Fast, Scalable, In-Memory Time Series Database* http://www.vldb.org/pvldb/vol8/p1816-teller.pdf Dec. 2016

[5] Netsil Inc. *A Comparison of Time Series Databases and Netsil's Use of Druid* https://blog.netsil.com/a-comparison-of-time-series-databases-and-netsils-use-of-druid-db805d471206 Dec. 2016

[6] Chris Davis *Graphite* http://www.aosabook.org/en/graphite.html Dec. 2016