Time Series Database (TSDB) Query Languages

Philipp Bende

January 26, 2017

Table of Contents

What is time series data?

A Time Series is:

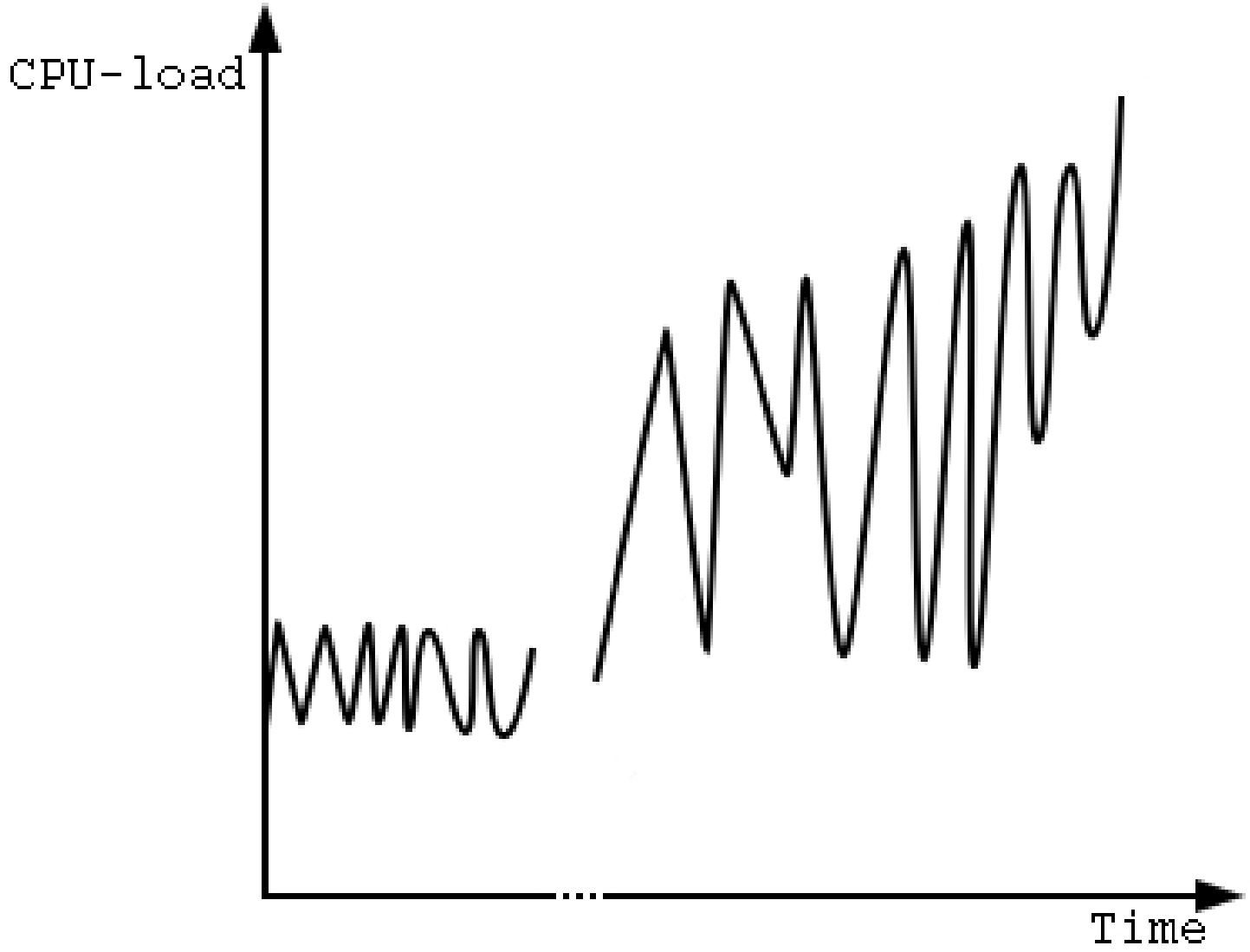
- collection of observations or data points obtained by repeated measure over time
- measurements happen in equal intervals
- measurement is well defined (who measures what)

Why are time series relevant?

Use cases:

- Industry 4.0
  - many sensors
  - continuous measures and evaluations
  - finding out when measurements deviate from the norm
- Monitoring data processing centers
  - observing processor / network load
  - predicting when storage capacity will not be sufficient
  - in fail cases: what lead to the failure?
- Finances
  - Observing trends of stock prices
  - predicting profits for the future

Why are time series relevant?



Definition of time series data

Time series data can be defined as:

- a sequence of numbers representing the measurements of a variable at equal time intervals.
- identifiable a source name or id and a metric name or id.
- consisting 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.

Can time series data be stored in a conventional database?

Short answer: Yes

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

results in huge SQL-tables
(8640 rows per sensor per day in the above example)

Disadvantages of conventional databases for time series data

- lots of sensor
- small time intervals between data measurements
- millions of entries per second into the database are rather the norm then the exception with time series

$\Rightarrow$ results in database tables with billions or even more rows

- handling and accessing such huge databases is slow and error prone

$\Rightarrow$ specialized time series databases

Time Series Databases

A TSDB system is

- collection of multiple time series
- software system optimized for handling arrays of numbers indexed by time, datetime or datetime range
- specialized for handling time series data

Characteristic workload patterns of time series

Reads and writes of time series data follow characteristic patterns

$\Rightarrow$ allows for a TSDB to be specialized to handle these patterns efficiently

Characteristic writes

- write-mostly is the norm (95% to 99% of all workload)
- writes are almost always sequential appends
- writes to distant past or distant future are extremely rare
- updates are rare
- deletes happen in bulk

Characteristic reads

- happen rarely
- are usually much larger then the memory
  $\rightarrow$ caching doesn't work well
- multiple reads are usually sequential ascending or descending
- reads of multiple series and concurrent reads are common

TSDB designs

TSDBs need to handle huge amounts of data

- distributed database options allow for more scalability then monolithic solutions
- "sending the query to the data" concept saves network traffic compared to the conventional "sending the data to the query processor"

TSDB designs – wide tables

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	...	...	...	...

wide tables allow for storage of many values in a single row

TSDB designs – wide tables

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	...	...	...	...

+ less rows
+ continuing a read is less expensive then starting a new read
+ changing the measurement interval does not change the number of rows required
– larger rows

TSDB designs – hybrid 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	...	

hybrid tables allow for storage of multiple single values as well as a compressed data object in a single row

TSDB designs – hybrid 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	...	

+ same advantages as wide table design
+ smaller rows then wide tables
+ retrieval of compressed data faster, since only 1 column needs to be accessed
– additional processing time for compression / decompression needed

TSDB design – direct BLOB insertion

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	{...}

only storing binary large objects (BLOBs), the compressed form of all values of a row

TSDB design – direct BLOB insertion

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	{...}

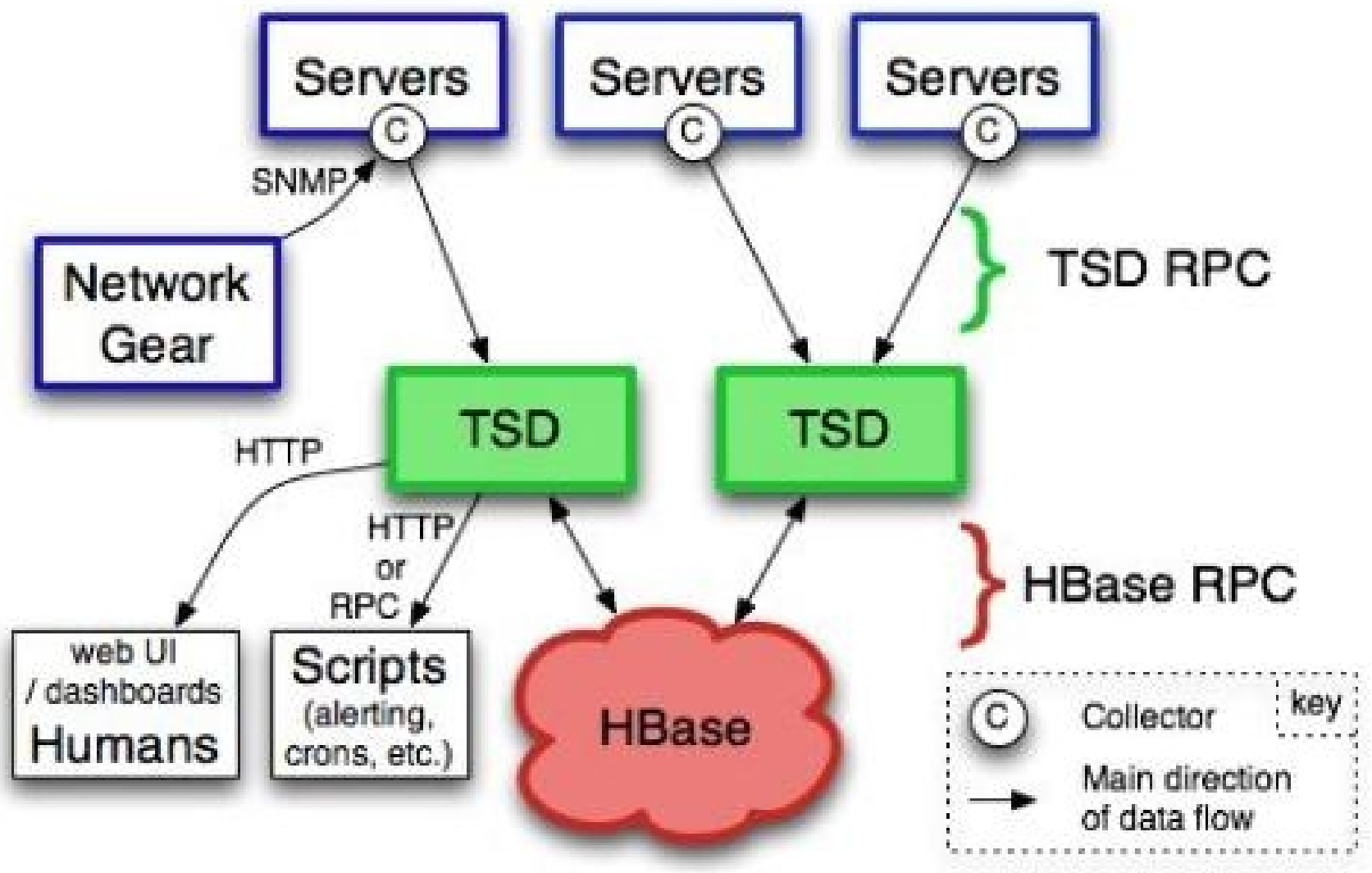
+ saves even more disk space then hybrid design
+ insertion and retrieval even faster, since only 1 entry needs to be accessed per row
– additional processing time for compression / decompression needed
– need to cache all data from time slot until it is complete before compression

Commonly used TSDBs

OpenTSDB

- open source TSDB
- HBase backend
- Design philosophy of direct blob insertion

OpenTSDB – schematic



OpenTSDB – queries

OpenTSDB offers access via

- REST API
- Telnet Interface
- HBase API (can be difficult due to the BLOB format)

with the usual REST methods GET, POST, PUT and DELETE

OpenTSDB – queries

Selection of a few methods aloowing querying and displaying of the results

- **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

OpenTSDB – queries

Queries usually include 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

OpenTSDB – example queries

Inserting values into the database:

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

For example:

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

OpenTSDB – example queries

Querying data from the database

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

For example:

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

Resulting in the output:

```
sys.cpu.user 123456 42.5
```

OpenTSDB – queries

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

1. parse query for syntax errors and existance of all metrics (sensor names), tag names and values
2. TSD sets up scanner for undelying
3. if query has tags → only rows that match the tag in addition to the timestamp and metric are fetched
4. fetched data is organized into groups, if the **GROUP BY** fuction is requested
5. downsampling (if requested) of the data is performed.
6. agregate each group of data by the requested aggregation function
7. if a rate was set → aggregates are adjusted to match the requested rate
8. reurn results to caller

InfluxDB

- partly open source TSDB
- no external dependencies
- monolithic version is open source
- highly scaling distributed version is commercial closed source
- was built with LevelDB as backend, but switched to a custom LSM-tree based solution

InfluxDB

- offers REST API similar to OpenTSDB
- queries via Influx Query Language (=basically SQL with a few additional features like **GROUP BY** or **TopN**)
- accepts many foreign TSDB protocols, like Graphite or OpenTSDB protocols

Gorilla

- TSDB behind Facebook
- aims to store relevant data in memory
- data older then 26 hours is moved to HBase based long term storage
- focuses on high compression rates
- in-memory storage of data allows for very fast queries
- factor 73 less query latency
- and factor 14 more throughput compared to OpenTSDB

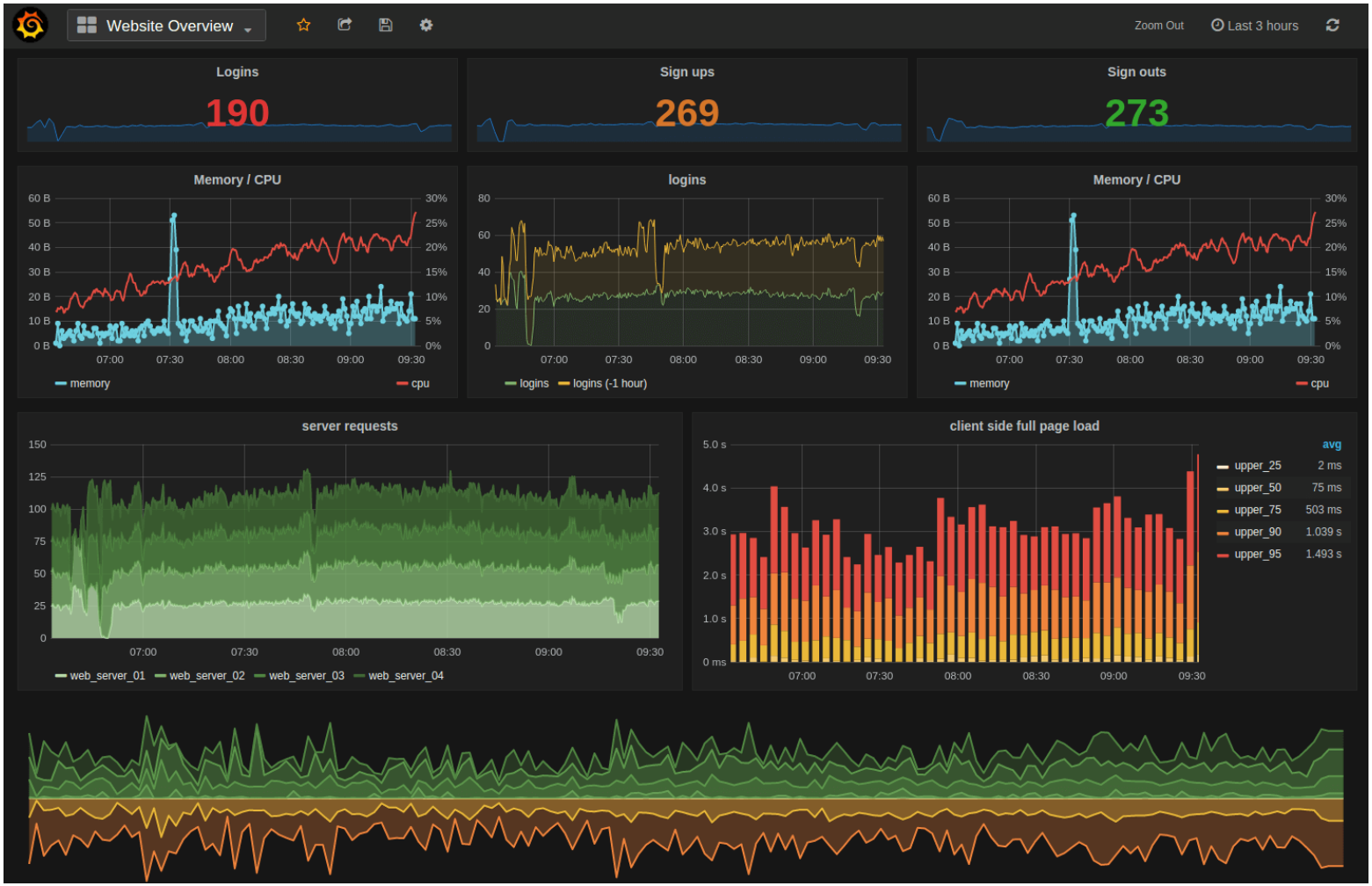
Graphite

- non-distributed open source TSDB
- stored data on local disk in Round Robin Database style called Whisper
- database size is predetermined
- stores each time series in a separate file and overwrites old files

$\Rightarrow$ less disk space consuming then OpenTSDB

Graphite – Grafana

most popular time series graphing tool Grafana was developed for Graphite (tho also compatible with other TSDBs)



Questions?

Thanks for your attention!

Sources:

 Minsam Kim and Jiho Park *Time-series Databases*

 Andreas Bader *Comparison of Time Series Databases* University of Stuttgart 2016-01-13

 InfluxData, Inc *InfluxDB Version 1.1 Documentation*

 Tuomas Pelkone et al. Facebook, Inc. *Gorilla: A Fast, Scalable, In-Memory Time Series Database*

 Netsil Inc. *A Comparison of Time Series Databases and Netsil's Use of Druid*

 Chris Davis *Graphite*