Time Series Databases

© Dmitry Namiot
Lomonosov Moscow State University, Moscow

dnamiot@gmail.com

Abstract

Data persistence for time series is an old and in many cases traditional task for databases. In general, the time series is just a sequence of data elements. The typical use case is a set of measurements made over a time interval. Much of the data generated by sensors, in a machine to machine communication, in Internet of Things area could be collected as time series. Time series are used in statistics, mathematical and finance. In this paper, we provide a survey of data persistence solutions for time series data. The paper covers the traditional relational databases, as well as NoSQL-based solutions for time series data.

1 Introduction

According to the classical definition, a time series is simply a sequence of numbers collected at regular intervals over a period of time. More generally, a time series is a sequence of data points (not necessarily numbers). And typically, time series consisting of successive measurements made over a time interval.

So, time series exist in any domain of applied science and engineering which involves temporal measurements. By this reason, data persistence mechanisms for time series are among oldest tasks for databases.

Let us start from the relational databases. At the first hand, the table design looks simple. We can create a table with a timestamp as a key column. Each new measurement will simply add a new row. And columns will describe our measurements (attributes). For the different time series we can add series ID column too:

```
CREATE TABLE TS AS
(
  ts_time TIMESTAMP
          NOT NULL     PRIMARY KEY,
  ts_id INT,
  ts_value FLOAT
)
```

In the real machine to machine (M2M) or Internet of Things (IoT) application, we will have more than one sensor. So, more likely, our application should support many time series simultaneously. Of course, in any practical system the whole set of attributes is limited. But for the each individual measurement we could have (potentially) any subset from this limited list. It is the typical use case for Machine to Machine (M2M) or Internet of Things (IoT) applications. Devices in our system will provide data asynchronously (it is the most practical use case). So, each row in our table will have many empty (null-value) columns. This decision (one row per measurement) leads to the very inefficient use of disk space. Also, it complicates the future processing.

Now let us discuss the possible operations. For time series (for measurements) the main operation in data space is adding new data (INSERT statement in SQL). Updating or deleting data is completely uncommon for time series (measurements). And reading has got some special moments too. Obviously, the reading of data is closely related to processing methods. The challenge in a database of evolving time series is to provide efficient algorithms and access methods for query processing, taking into consideration the fact that the database changes continuously as new data become available [1].

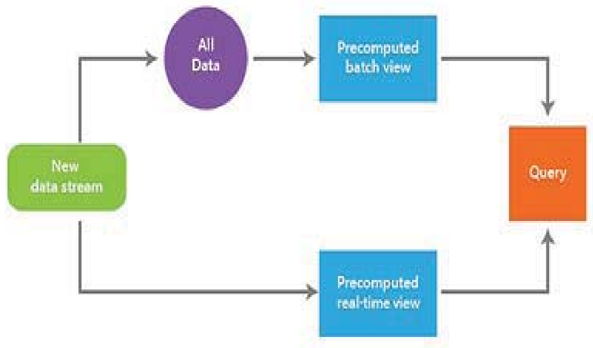
Many (most of) algorithms for time series data mining actually always work with only part of the data. And for the streamed data the sliding window is the most natural choice. Let us discuss some well-known techniques. Random Sampling lets us sampling the stream at periodic intervals. In this approach, we maintain a sample called the "reservoir," from which a random sample can be generated. As the data stream flows, every new element has a certain probability of replacing an old element in the reservoir [2]. Instead of sampling the data stream randomly, we can use the sliding window model to analyze stream data [3]. The basic idea is that rather than running computations on all of the data seen so far, or on some sample (as the above-mentioned random sampling), we can make decisions based only on recent data. So, any element for analysis, arrived at some time t will be declared as expired at time $t+w$, where w is the window "size". In many practical use cases, we can assume (e.g. sensing in IoT applications) that the only recent events may be important.

Proceedings of the XVII International Conference «Data Analytics and Management in Data Intensive Domains» (DAMDID/RCDL'2015), Obninsk, Russia, October 13 - 16, 2015

A histogram approach partitions the data into a set of contiguous buckets.

In a titled frame model, we use different granularities for time frames. The most recent time is registered at the finest granularity; the most distant time is registered at a coarser granularity. And so on.

We present this short survey for highlighting the fact that time series data mining algorithms actually almost always work with only part of the data. It is a common use case for time series processing. In terms of SQL, SELECT statement with some complex condition is uncommon for time series processing. What we need to read for most of the algorithms is some limited portion (window) of recent constantly updated (added) data. In the same time, we need the full log of data (e.g., for the verification, audit, billing, etc.). Probably, it is the perfect example of so-called lambda architecture [4]. It is illustrated in Figure 1.



Fig, 1. Lambda architecture [5]

On this picture we have data source with constantly updated data. Most of the data should be processed in the real time. It is especially true for Internet of Things and M2M systems, where time series processing is the main source behind control actions and conclusions (alerts). In the same time, we could still have some processing without the strong limitations for time-to-decision. And database could save processed data for queries from users and applications. It would be convenient to have such a processing as a part of a database system.

The rest of the paper is organized as follows. The section 2 is devoted to time series support in relational databases. In section 3 we describe NoSQL solutions for time series.

2 Time Series and relational databases

In this section, we would like to discuss time series persistence (and processing, of course) and "traditional" databases. As the first example, we have tested TokuDB as an engine for time series data [6]. This engine uses a Fractal Tree index (instead of more known B-tree). A Fractal Tree index is a tree data structure. Any node can have more than two sub-nodes (children) [7]. Like a B-tree, it that keeps data always sorted and allows fast searches and sequential access. But unlike a B-tree, a Fractal Tree index has buffers at each node. Buffers allow changes to be stored in intermediate locations. Buffers lets schedule disk writes so that each writing operations deals with a large block of data [8]. The simple database related analogue is a transactions monitor. This optimization lets perform fast data writing (INSERT operations we are mostly interested for time series). Also, these "local" buffers can be used during replications. Figure 2 illustrates the benchmark for INSERT operations (provided by TokuDB).

.

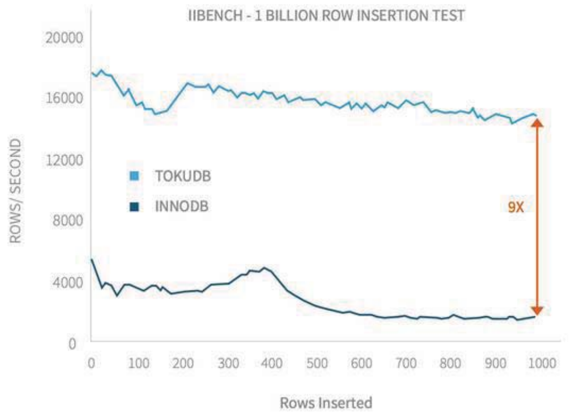


Fig. 2 TokuDB vs. InnoDB [9]

Of course, the "traditional" relation systems are easy to maintain, they could be cheaper to host and it could be easier (cheaper) to find developers. So, the maintenance is the biggest advantage.

As the second position in this section we would like to mention time series-related SQL extension in Vertica database [10]. Vertica provides so-called time series analytics as an extension for SQL. As we have mentioned above, input records for time series data usually appear at non-uniform intervals. It means they might have gaps (missed values). Vertica provides so-called gap-filling functionality. This option fills in missing data points using an interpolation scheme. Secondly, Vertica allows developers to use event-based windows to break time series data into windows that border on significant events within the data. SQL extension proposed by Vertica could be used as a useful example of language-based support for time series in SQL databases. Here is a typical example:

SELECT item, slice_time, ts_first_value(price, 'const') price FROM ts_test WHERE price_time BETWEEN timestamp '2015-04-14 09:00' AND timestamp '2015-04-14 09:25' TIMESERIES slice_time AS '1 minute' OVER (PARTITION BY item ORDER BY price_time) ORDER BY item, slice_time, price;

This request should fill missed data from 09:00 till 09:25 with 1 minute step in the returned snapshot.

Vertica provides additional support for time series analytics with the following SQL extensions:

The SELECT … TIMESERIES clause supports gap-filling and interpolation computation.

TS_FIRST_VALUE and TS_LAST_VALUE are time series aggregate functions that return the value at the start or end of a time slice, respectively, which is determined by the interpolation scheme.

TIME_SLICE is a (SQL extension) date/time function that aggregates data by different fixed-time intervals and returns a rounded-up input TIMESTAMP value to a value that corresponds to the start or end of the time slice interval.

The similar examples are so-called window-functions in PostgreSQL [11]. A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result. The typical example (a moving average for three rows) is illustrated below:

```
SELECT id_sensor, name_sensor, temperature,
        avg(temperature) OVER (ORDER BY
id_sensor  ROWS BETWEEN 1 PRECEDING AND 1
FOLLOWING)
    FROM temperature_table;
```

3 Time Series in NoSQL systems

NoSQL as one of the basic principles proclaimed rejection of a universal model of data. The data model must meet the required processing methods. The second basic principle is the lack of the dedicated programming access tools (layers). Data access API is a part of NoSQL systems, and it presents one of the important elements for the final selection of a data model.

By our opinion, NoSQL solutions for time series could be described as "best practices" in using NoSQL stores for time series. Let us see, for example, the architecture for OpenTSDB [12]. It is one of the popular NoSQL solutions for time series.

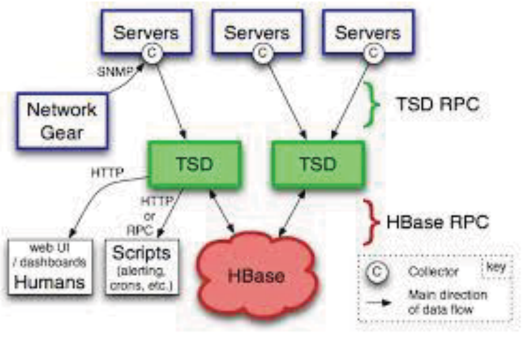


Fig. 3. OpenTSDB architecture [13]

OpenTSDB is a set of so-called Time Series Daemon (TSD) and command line utilities. Each TSD (and they are independent) is just a wrapper for access to HBase database [14]. Each TSD is independent. Each TSD uses the HBase to store and retrieve time series data. TSD itself supports a set of protocols for access to data.

The second dimension of this architecture solution is the optimized schema for data. In this case, the schema is highly optimized for fast aggregations of similar time series. This schema is actually almost a de-facto standard for presenting time-series in a so-called BigTable data model. In OpenTSDB, a time series data point consists of a metric name, a timestamp, a value and a set of tags (key-value pairs). So, for example, suppose we have a metric (name) data.test, a key name is host, a key value is host1. In this case, each row in master table looks so:

```
data.test Time Value host host1
```

here Time is a timestamp, Value is a measured value. And all APIs will use the similar format for data writing – there is no schema definition. Set of keys lets present so-called multivariate time series.

OpenTSDB handles things a bit differently by introducing the idea of 'tags'. Each time series still has a 'metric' name, but it's much more generic, something that can be shared by many unique time series. Instead, the uniqueness comes from a combination of tag key/value pairs that allows for flexible queries with very fast aggregations. Every time series in OpenTSDB must have at least one tag. The underlying data schema will store all of the tag's 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.

OpenTSDB follows to one of the commonly used patterns for time series data persistence in column-oriented databases like HBase. The basic data storage unit in HBase is a cell. Each cell is identified by the Row ID, column-family name, column name and the version. Each cell can have multiple versions of data. At the physical level, each column family is stored continuously on disk and the data are physically sorted by Row ID, column name and version [15]. The version dimension is used by HBase for time-to-live (TTL) calculations. Column families may be associated with a TTL value (length). So, HBase will automatically delete rows once the expiration time is reached. For time series data, this feature lets automatically delete old (obsolete) measurements, for example. The possible schemes for time series data are:

a) The row key is constructed as a combination of a timestamp and sensor ID. Each column is the offset of the time for the timestamp in the row. E.g., the timestamp is one hour, the column offset is two minutes. Each cell contains the values for all sensor's (defined by Sensor ID) measurements at the moment timestamp + offset. As the format for cell's data, we can use JSON or even commas separated values. As a variation, we can combine all data in a row into a binary object (blob).

b) The row key is constructed as a combination of a timestamp and sensor ID. Each column corresponds to one measurement (metric) and contains values for all time offsets.

KairosDB [16] is a rewrite of the original OpenTSDB and uses Cassandra as a data store.

There are several patterns for storing time series data in Cassandra. When writing data to Cassandra, data is sorted and written sequentially to disk. When retrieving data by row key and then by range, you get a fast and efficient access pattern due to minimal disk seeks.

The simplest model for storing time series data is creating a wide row of data for each measurement. E.g.:

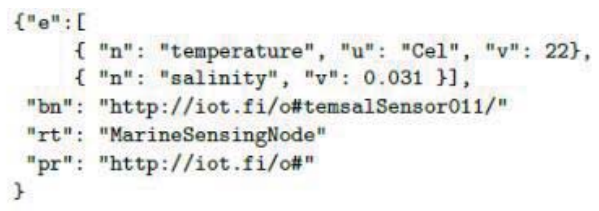
SensorID, {timestamp1, value1}, {timestamp2, value2} … {timestampN, valueN}

Cassandra can store up to 2 billion columns per row. For high frequency measured data, we can add a shard interval to a row key. The solution is to use a pattern called row partitioning by adding data to the row key to limit the amount of columns you get per device. E.g., instead of some generic name like smart_meter1 we can use smart_meter1_day (e.g. smart_meter_20150414).

Another common pattern for time series data is so-called rolling storage. Imagine we are using this data for a dashboard application, and we only want to show the last 10 temperature readings. Older data is no longer useful, so they can be purged eventually. With many other databases, we would have to setup a background job to clean out older data. With Cassandra, we can take advantage of a feature called expiring columns to have our data quietly disappear after a set amount of seconds [17].

What is really interesting and new for NoSQL solutions is the growing support for SenML [18]. SenML is defined by a data model for measurements and simple meta-data about measurements and devices. The data in SenML is structured as a single object with attributes. The object contains an array of entries (measurements). Each entry is an object that has attributes such as a unique identifier for the sensor, the time the measurement was made, and the current value (Figure 4). Serializations for this data model are defined for JSON and XML.

Geras DB [19] uses SenML as data format. Another important feature for any time series database is MQTT support. MQTT is a popular connectivity protocol for Machine-to-Machine and Internet of Things communications [20]. It was designed as an extremely lightweight publish/subscribe messaging transport. Sensors as data sources may use MQTT, so a time series database should be able to acquire data right from MQTT [21, 22].



```
{"e":[
     { "n": "temperature", "u": "Cel", "v": 22},
     { "n": "salinity", "v": 0.031 }],
  "bn": "http://iot.fi/o#temsalSensor011/"
  "rt": "MarineSensingNode"
  "pr": "http://iot.fi/o#"
}
```

Fig. 4 SenML example

Druid is an open-source analytics data store designed for OLAP queries on time series data (trillions of events, petabytes of data). Druid provides cost-effective and always-on real-time data ingestion, arbitrary data exploration, and fast data aggregation [23]. Druid is a system built to allow fast ("real-time") access to large sets of seldom-changing data. It provides:

the column-based storage format for partially nested data structures;

the hierarchical query distribution with intermediate pruning;

indexing for quick filtering;

realtime ingestion (ingested data is immediately available for querying);

the fault-tolerant distributed architecture that doesn't lose data.

Data is ingested by Druid directly through its real-time nodes, or batch-loaded into historical nodes from a deep storage facility. Real-time nodes accept JSON-formatted data from a streaming datasource. Batch-loaded data formats can be JSON, CSV, or TSV. Real-time nodes temporarily store and serve data in real time, but eventually push the data to the deep storage facility, from which it is loaded into historical nodes. Historical nodes hold the bulk of data in the cluster.

Real-time nodes chunk data into segments, and they are designed to frequently move these segments out to deep storage. To maintain cluster awareness of the location of data, these nodes must interact with Mysql to update metadata about the segments, and with Apache ZooKeeper to monitor their transfer.

Figure 5 illustrates Druid architecture.

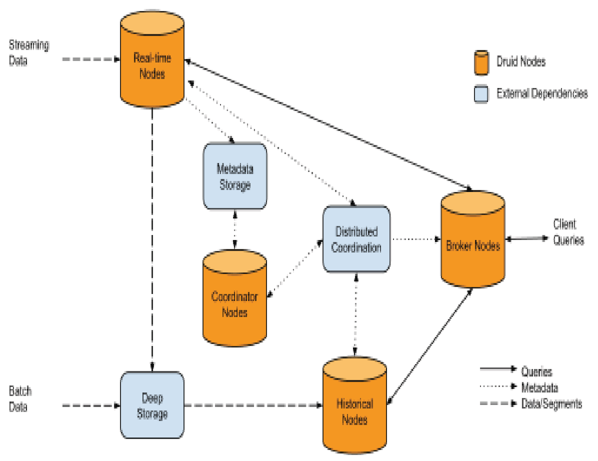


Fig.5 Druid Architecture [24]

SciDB's [25] native multi-dimensional array data model is designed for ordered, highly dimensional, multifaceted data. SciDB's data is never overwritten, allowing you to record and access data corrections and updates over time. SciDB is designed to efficiently handle both dense and sparse arrays providing dramatic storage efficiencies as the number of dimensions and attributes grows. Math operations run directly on the native data format. Partitioning data in each coordinate of an array facilitates fast joins and access along any dimension, thereby speeding up clustering, array operations and population selection.

BlinkDB [26] supports a slightly constrained set of SQL-style declarative queries and provides approximate results for standard SQL aggregate queries, specifically queries involving COUNT, AVG, SUM and PERCENTILE and is being extended to support any User-Defined Functions (UDFs). Queries involving these operations can be annotated with either an error bound or a time constraint, based on which the system selects an appropriate sample to operate on. For example:

SELECT avg(Temperature) from Table where SensorID=1 WITHIN 2 seconds

SAP HANA's [27] column-oriented in-memory structures have been extended to provide efficient processing of series data. SAP HANA provides:

series property aspect of tables;

built-in Special SQL functions for working with series data;

analytic functions: special SQL functions for analyzing series data;

storage support: advanced techniques for storing equidistant data using dictionary encoding

By adding Series Data descriptors to column tables, users can identify which columns contain series data, period information, hints on how to handle missing timestamps, and so on. By explicitly telling HANA about time series data, it can more efficiently store and manage this data to increase performance and decrease the memory footprint through improved compression.

TABLESAMPLE allows ad-hoc random samples over column tables so it is easy, for example, to calculate a result from a defined percentage of the data in a table.

There are examples for built-in functions:

SERIES_GENERATE – generate a complete series

SERIES_DISAGGREGATE – move from coarse units (e.g., day) to finer (e.g., hour)

SERIES_ROUND – convert a single value to a coarser resolution

SERIES_PERIOD_TO_ELEMENT – convert a timestamp in a series to its offset from the start

SERIES_ELEMENT_TO_PERIOD – convert an integer to the associated period .

Analytical functions are:

CORR – Pearson product-moment correlation coefficient

CORR_SPEARMAN - Spearman rank correlation

LINEAR_APPROX - Replace NULL values by interpolating adjacent non-NULL values

MEDIAN - Compute median value

InfluxDB [28] is open-source, distributed, time series database with no external dependencies. InfluxDB is targeted at use cases for DevOps, metrics, sensor data, and real-time analytics. The key moments behind InfluxDB are:

- SQL like query language.
- HTTP based API.
- Database managed retention policies for data.
- Built-in management interface.
- On the fly aggregation.

SQL-like query with aggregation by time looks so:

SELACT mean (value) FROM T GROUP BY time(5m).

The lack of external dependencies makes Influx very attractive from the practical point of view. The opposite approach is the above-mentioned Druid with almost full of Apache stack (Zookeeper, etc.)

From cloud-based solutions for time series data, we can mention Blueflood [29]. Blueflood uses Cassandra as the data store.

As per the classical definition, Big Data could be described via so-called 3V: Variety, Velocity and Volume. By our opinion, for time series databases the key factor is Velocity. NoSQL solution for time series should be selected in case of high frequency measurements. And in case of NoSQL solutions for time series, Cassandra is the preferred choice.

References

[1] Kontaki, M., Papadopoulos, A. N., & Manolopoulos, Y. (2007). Adaptive similarity search in streaming time series with sliding windows. Data & Knowledge Engineering, 63(2), 478-502.

[2] Chatfield C. The analysis of time series: an introduction. – CRC press, 2013.

[3] Han J., Kamber M., Pei J. Data mining: Concepts and techniques. – Morgan Kaufmann, 2006.

[4] Fan, Wei, and Albert Bifet. "Mining big data: current status, and forecast to the future." ACM SIGKDD Explorations Newsletter 14.2 (2013): 1-5.

[5] Lambda Architecture: Design Simpler, Resilient, Maintainable and Scalable Big Data Solutions http://www.infoq.com/articles/lambda-architecture-scalable-big-data-solutions

[6] Bartholomew, D. (2014). MariaDB Cookbook. Packt Publishing Ltd.

[7] Chen, S., Gibbons, P. B., Mowry, T. C., & Valentin, G. (2002, June). Fractal prefetching B+-trees: Optimizing both cache and disk performance. In Proceeding of the 2002 ACM SIGMOD international conference on Management of data (pp. 157-168). ACM.

[8] Bender, M. A.; Farach-Colton, M.; Fineman, J.; Fogel, Y.; Kuszmaul, B.; Nelson, J. (June 2007). "Cache-Oblivoius streaming B-trees". Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (CA: ACM Press): 81–92.

[9] TOKUDB® VS. INNODB FLASH MEMORY http://www.tokutek.com/tokudb-for-mysql/benchmarks-vs-innodb-flash/

[10] Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., & Bear, C. (2012). The vertica analytic database: C-store 7 years later. Proceedings of the VLDB Endowment, 5(12), 1790-1801.

[11] Obe R., Hsu L. S. PostgreSQL: up and running. – " O'Reilly Media, Inc.", 2012.

[12] Wlodarczyk, T. W. (2012, December). Overview of time series storage and processing in a cloud environment. In Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 625-628). IEEE Computer Society.

[13] OpenTSDB http://opentsdb.net

[14] George, L. (2011). HBase: the definitive guide. " O'Reilly Media, Inc.".

[15] Han, D., & Stroulia, E. (2012, September). A three-dimensional data model in hbase for large time-series dataset analysis. In Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the (pp. 47-56). IEEE.

[16] Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., & Breivold, H. P. (2014, June). Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. In Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on (pp. 602-609). IEEE.

[17] Planet Cassandra http://planetcassandra.org/getting-started-with-time-series-data-modeling/

[18] Jennings, Cullen, Jari Arkko, and Zach Shelby. "Media types for sensor markup language (SENML)." (2012).

[19] Geras DB http://1248.io/geras.php Retrieved: Feb, 2015

[20] Hunkeler, U., Truong, H. L., & Stanford-Clark, A. (2008, January). MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. In Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on (pp. 791-798). IEEE.

[21] Namiot D., Sneps-Sneppe M. On IoT Programming //International Journal of Open Information Technologies. – 2014. – T. 2. – №. 10. – p. 25-28.

[22] Sneps-Sneppe, M., & Namiot, D. (2012, April). About M2M standards and their possible extensions. In Future Internet Communications (BCFIC), 2012 2nd Baltic Congress on (pp. 187-193). IEEE.

[23] Druid http://druid.io

[24] Druid Whitepaper http://static.druid.io/docs/druid.pdf

[25] Stonebraker, M., Brown, P., Poliakov, A., & Raman, S. (2011, January). The architecture of SciDB. In Scientific and Statistical Database Management (pp. 1-16). Springer Berlin Heidelberg.

[26] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., & Stoica, I. (2013, April). BlinkDB: queries with bounded errors and bounded response times on very large data. In Proceedings of the 8th ACM European Conference on Computer Systems (pp. 29-42). ACM.

[27] Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., & Dees, J. (2012). The SAP HANA Database--An Architecture Overview. IEEE Data Eng. Bull., 35(1), 28-33.

[28] Leighton, B., Cox, S. J., Car, N. J., Stenson, M. P., Vleeshouwer, J., & Hodge, J. (2015). A Best of Both Worlds Approach to Complex, Efficient, Time Series Data Delivery. In Environmental Software Systems. Infrastructures, Services and Applications (pp. 371-379). Springer International Publishing.

[29] Blueflood: A new Open Source Tool for Time Series Data at Scale https://developer.rackspace.com/blog/blueflood-announcement/