A Software Cache Mechanism for Reducing the OpenTSDB Query Time

Thada Wangthammang[1] and Pichaya Tandayya[2]
Department of Computer Engineering, Faculty of Engineering
Prince of Songkla University,
P.O. Box 2, Kho Hong, Hatyai, Songkhla, Thailand
wangth.thada@gmail.com[1] and pichaya@coe.psu.ac.th[2]

Abstract—As the volume of Internet of Things (IoT) data grows rapidly, a time series database is a major part of the system that helps in storing and retrieving the massive data for analysis and monitoring. According to the user behaviors, the data queries often are similar or overlapped to the previous queries, especially when the user adjusts the query parameters on the same time series data for displaying on a visualization tool. Caching the previous data will help reduce the query response time, in case that the current query range is overlapping with the previous query ranges. This paper proposes a cache mechanism for an open source time series database, OpenTSDB. Exploiting our cache mechanism, the query response time can be 3-16 times faster than the original OpenTSDB depending on the user query behaviors and related parameters.

Keywords—Time Series Database; Software Cache; Cache mechanism; Query time; OpenTSDB

I. Introduction

Internet of Thing (IoT) applications are widely applied in many fields such as recommender systems, smart home, IoT applications, monitoring systems and financial analyses [1]. Many researches and companies also emerge on this technology to improve their products and services. Moreover, the volume of data grows so rapidly and continuously and will surge in the future. It requires a lot of storage capacity to store the data. In addition, data retrieving can take a long time depending on the storage type. One of the popular data types for periodically storing and retrieving a massive amount of data from IoT devices is a time series. It records the observed data value and its timestamp which is sorted by the time. We can analyze the relationship between the observed data and the time effects such as the season, or the trend in time series.

Hence, the retrieval data speed is significant for analyzing time series on time. Time Series Data Server (TSDS) offers a fast database in many use cases [2]. Especially, in an emergency situation, it requires fast tools for analyzing and summarizing data in order to produce live solutions.

Many researches attempted to improve time series databases in several ways, depending on application purposes [3], [4]. One of the methods is to change the physical storage from hard disks to memory and use hard disks as backup storages in order to improve the overall query throughput such as Gorilla [5]. This method is suitable for real-time applications such as monitoring. Although this method is fast, it is expensive. Another solution to improve the query throughput is to apply a caching technique. Originally, the advantage of using cache is to reduce the budget spending on high memory capacity. The time series query data can also be cached similarly.

Time series databases are often applied in applications for monitoring and analyzing data. The main requirement for monitoring is to discover abnormal situations in real time. Most applications applied a memory layer before reaching permanent storage, usually lower speed, e.g. hard disks [4]. Data analysis applications also require a fast retrieval system but it does not need real-time retrieval. It also depends on the system requirements and the budget. A cache technique can improve the time series system for data analysis. Often, the time series cache mechanism was applied in a wide area hierarchy storage for storing sensor data [6]. There were many levels of sensor sites in the hierarchy. They designed the cache mechanism to deal with overlapping query time ranges, called partial match caching. Nevertheless, they did not elaborate detail about their caching technique and architecture. Also, they focused on a cache system for hierarchical storage in order to achieve fast retrieval from distributed sites.

OpenTSDB, an open source time series database, [7] enables a rapid graphical display of a large number of data points of time series. However, it still has query response time problems. Although the current query range is overlapped with previous query ranges, OpenTSDB still processes the query request for the whole requested time range. It means that OpenTSDB has no cache mechanism. If it can process only parts never been queried before, the throughput and response time will be improved, especially in case of many users on the client side. Therefore, this paper proposes a software cache mechanism for OpenTSDB on HBase. The objective is to reduce the response time of data retrieval with partial cache hit by using query times as the cutting points.

II. Backgrounds

A. Related Works

Many researches improved time series databases using different methods. The methods can be categorized into

two groups, depending on the system requirements: time series database improvements, and partial cache techniques.

1) Time Series Database Improvements: there are two main purposes in exploiting time series databases: to monitor the time series data and alert the users, and to query for historical time series data. The time series monitoring and alerting systems require real-time alerting for abnormal situation detection. It uses a lot of memories for processing large data such as in an in-memory time series database called Gorilla [5] and data stream management for monitoring called Aurora [8]. They were designed for in-memory time series database, not a cache system. It is not suitable for a system with budget limitation. They do not deal with low memory capacity issues.

2) Partial Cache Techniques: The technique is mostly applied in sensor or time series database systems which have many small data fractions. When a user wants to query a group of data fractions, if all fractions need to be retrieved from a database, the process may be slow due to the disk speed. A solution is caching some data fractions and retrieving them from the cache when needed. If the data is not in the cache, the system will collect the rest of requested data fractions from the database and merge them together.

Deshpande and others proposed Cache-and-Query for wide area sensor databases [6] and their application, IrisNet [9]. It focused on querying wide area sensor databases which stored data spreading in a wide area over the distance of ten to thousands of miles. They proposed a cache technique, called partial match caching for storing partial data of a site. The proposed technique cached previous queries and other visited sites for reducing the query time. However, the cache technique was explained briefly. They did not use time ranges as cutting points of the cache objects but considered whether the whole of site sensor data was cached or not.

Druid [10] is a distributed time series database implemented by Yang et al. The system ingests time series events from log files, and then performs ingested data analytics for OLAP processing in real time. Druid also provides historical nodes for executing queries. In order to reduce query response time, they split the range of the query into the segment by time for caching the segments. Our work split queries by time which is similar to Druid for cache purpose. Nevertheless, they did not discuss the cache mechanism in detail and specially designed for Druid system.

B. OpenTSDB

OpenTSDB is a time series database that the users can manipulate and store time series. It is written on top of HBase which is a distributed data storage that enables scalability. OpenTSDB creates worker modules called Time Series Daemon (TSD) for handling OpenTSDB jobs. There may be one or more TSDs which are independent.

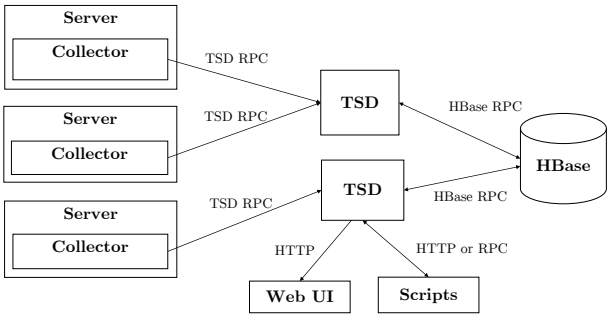

Fig. 1. OpenTSDB Architecture redrawn from [11]

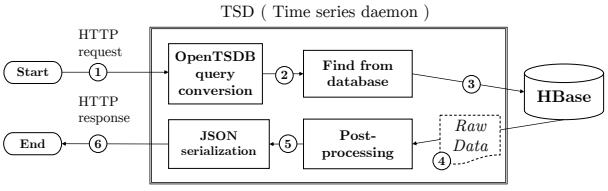

Fig. 2. OpenTSDB query procedure

Figure 1 shows how TSDs interact with the others and depicts the main data flow noticed by the arrow direction. A TSD conducts as an intermediary between writing and reading OpenTSDB jobs and an HBase storage. TSD allows for writing jobs via RPC and HTTP such as data collectors, Tcollector, and scripts. TSD also allows for reading jobs such as querying data from OpenTSDB, that is to be queried from HBase. Reading jobs can be done by using HTTP only. The Web User Interface is connected through HTTP.

Our work involved with the query procedure in the OpenTSDB. When the query requests arrive at TSD, the TSD performs the requests for retrieving the time series data from HBase. OpenTSDB has many modules to manipulate the requests. We focus on only the part of raw data manipulation inside the OpenTSDB query process. Figure 2 shows the query procedure beginning from getting a time series query in the OpenTSDB format. Assuming that the query request has already been prepared for performing raw data, after the Find from database module retrieves data from HBase, the module will handle the HBase Raw data. When all data arrives, the data will be passed to the Post-processing module such as downsampling and data aggregation.

C. Memcached

Memcached [3] is an in-memory key-value object caching system. It has the least recently used (LRU) mechanism for cache replacement which will evict the expired cached objects. The system also supports distributed memory objects using many servers in order to share their in-memory storage with the other servers. Memcached requires a client-side library that manages data partitioning

over Memcached servers for performing the distributed memory. The client-side libraries are written in many languages such as Python, PHP, Java, C/C+, etc. Moreover, Memcached also supports UDP and TCP connections. Memcached provides three basic commands: get, store, and delete. In this work, Memcached is applied as a cache object management system. It provides simple functions to manipulate cache items, and is able to handle cache replacement when the memory is out of space.

III. Time Series Cache Design

Many user actions are performed by time series visualization tool, mostly connected with a time series database. When the user zooms, or moves the display window of the time series, it means a new query will be created, even though, the user requests for the time series data exactly the same to the previous one. When OpenTSDB handles similar or partially raw data from the database, but the post-processing results can be different from the user's parameters, even on the same time series data. Generally, according to the limited budget, the users mostly use hard disks as a storage, which is extremely slow when compared to memory. According to our preliminary work, it reveals that OpenTSDB has no cache mechanism in querying time series from the database even when the current query time range is overlapping with the past queries. Consequently, we have decided to design a cache mechanism for OpenTSDB. The design aims to reduce the latency in retrieving time for part of time series. Provided that the raw data can be cached, its query time should be reduced.

We handle the raw data before OpenTSDB performs the post-processing module. There are two approaches to manage cached objects: internal and external caches. The benefit of internal cache is no networking overhead, while external cache infers the overhead. In case of internal cache, the design and implementation of a cache management system are required, while external cache can instantly be applied to the existing system. Therefore, distributed memory storage likes Memcached is applied in our design to benefit from a high memory capacity pool. Nevertheless, we leave the issues of cache policy and eviction method to the chosen tool, Memcached. We only focus on how to cache time series data.

Generally, any item to be cached can incur responses in two statuses: cache hit and cache miss. However, in case of a big data time series database system, caching the whole time series can take much more memory space, and spend extra time for storing irrelevant time series ranges. Therefore, we have decided to store only part of time series ranges which the users have already queried, called partial hit.

The architecture of our caching system for OpenTSDB is depicted in Figure 3. We replace the Find from database module, which is a part of the OpenTSDB query procedure as shown in Figure 2, with a new algorithm in order to

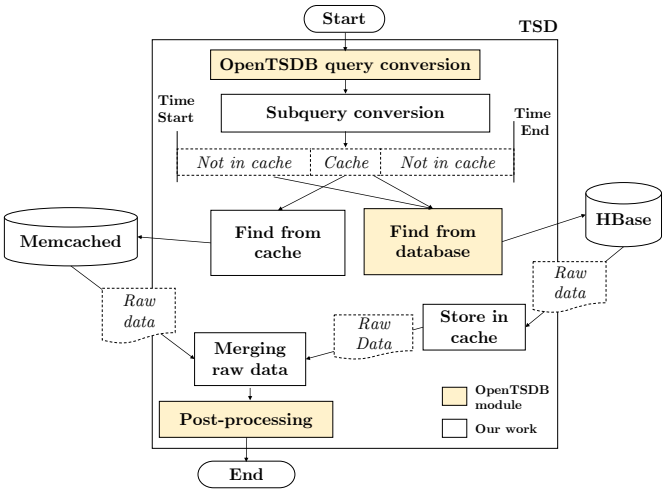

Fig. 3. The architecture of our cache system

reduce the query time by applying Memcached to store the uncached data and retrieve previous data as shown in Figure 3. This design aims to reduce the query time in case that some ranges of the incoming query and the previous queries are overlapping with each other. After the OpenTSDB query conversion module has completed its task, our cache mechanism will group the time series data into a fragment using a cache index as described in Section III-A. Section III-B shows the definition of the subquery range, called subquery. Each subquery decides whether part of the incoming time series data is in the cache or not. The Find from database module will retrieve raw data from HBase if the subquery is not in the cache, and then stores the raw data in the range of the subquery. The Find from cache module, described in SectionIII-E, will operate if the subquery has already been in the cache. After all the raw data from both paths has already arrived, it will merge the data into the same data structure as of the output of the original Find from database module and passed to the post-processing module.

A. Cache Indexing

A data point is around 22 bytes, calculated from the OpenTSDB HBase schema [12] and the massive amount of data points. The OpenTSDB HBase schema design groups data points into an HBase row. The recording time of data points is split by an hour, which is the default value of HBaseRowPeriod.

However, when the data is huge, the number of HBase rows even produces a large number of items to be cached resulting in more network overhead. Therefore, we have decided to group a number of HBase rows into a fragment, called chunk size (CS). In case that CS is small, the number of fragments can be high. On the other hand, the number of fragments is the number of items to be cached. If there are too many cached items, the query response time will be affected. The fragment can be represented by

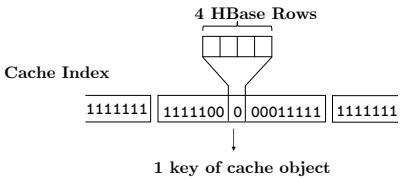

Fig. 4. Cache index time series for the OpenTSDB

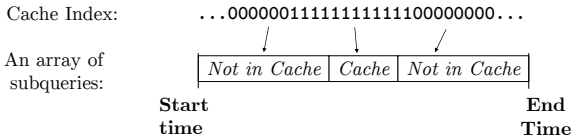

Fig. 5. Creating an array of subquerys from the cache index

fragment order (FO) calculated by Equation 1, whereas T is the timestamp and the time unit is the millisecond.

$$FO = \lfloor \frac{T}{CS \times HBaseRowPeriod} \rfloor \quad (1)$$

The chunk size of fragment affects the network overhead, according to the number of requests sent out to the external cache system. To reduce the number of requests of the external cache system, we propose a time series cache index to reduce the cache item retrieval miss rate as shown in Figure 4. We define the cache index as an array of 64-bit long integer, and leverages bit operations in order to reduce the computational overhead.

Figure 4 also gives an example of which chunk size is four. Each fragment is represented by one bit of the cache index and a key for looking up in the cache system, Memcached. In each bit, one indicates a fragment already stored in the cache. Otherwise, zero indicates that the fragment does not exist in the cache.

B. The Subquery Conversion Module

The aim of the Subquery conversion module is to generate an array of subqueries. The start and stop times of each subquery are reassigned. A subquery also has a status showing whether it is in cache or not. In order to store the fragments into the cache object, cutting the time series by the recording time is necessary for deciding which part is to be stored in the cache. The time cutting algorithm is also involved in cache indexing.

The time cutting algorithm inside the Subquery conversion module is divided into three procedures:

1) To calculate the start FO by the user's request start time using Equation 2 and the end FO by the user's request end time using Equation 1.

$$FO = \lceil \frac{T+1}{CS \times HBaseRowPeriod} \rceil \quad (2)$$

2) Firstly, to use the start FO and end FO as a range to look up in the cache index. Then, to convert the cache index in order to search for the adjacent bits which have the same value as the subquery as shown in Figure 5.
3) To convert the start FO and end FO into the start and end time in each subquery. This can be done by using Equations 3 and 4, respectively.

$$T = FO \times CS \times HBaseRowPeriod \quad (3)$$

$$T = (FO+1) \times CS \times HBaseRowPeriod - 1 \quad (4)$$

C. The Find from Database Module

The original module retrieves the raw data from the database according to the user's request. In our work, we modify the range of the time series request to match the new range provided by the subquery conversion module in order to retrieve only parts of the whole requested range.

D. The Store in Cache Module

After the Find from database module has finished its task, the next procedure is the Store in cache module as shown in Figure 3. This module stores data in Memcached asynchronously following the OpenTSDB design. We have created a key for representing the fragment to be stored in Memcached. The key is the original HBase row key as it does not incur more overhead on the conversion. The fragment is needed to be serialized into a byte array which is required by Memcached. In order to prove the concept, we have implemented a simple serialization algorithm for converting the fragment.

E. The Find from Cache Module

The Find from database module is for retrieving raw data from HBase. We apply the Find from cache module to retrieve time series data which is already stored in Memcached with given keys, both modules have the same input and output data type. The time range of each subquery is passed into the Find from cache module as shown in Figure 3. This module holds the start fragment order of the subquery. Therefore, the key is to look upon Memcached using the start time which is converted from the start fragment order following Equation 3. Then, this module retrieves the cached data from Memcached by the key. This process also performs asynchronously. Lastly, our unserialization algorithm is applied to convert the byte array back to the fragment structure.

F. The Merging Raw Data Module

This module is performed when both raw HBase rows retrieved from the Find from database and the Find from cache modules have arrived as shown in Figure 3. The effect of the cutting algorithm is that it always has duplicated HBase rows due to the OpenTSDB design. For instance, if the data is partially found in an HBase row,

the scanner method of the Find from database will retrieve the whole HBase row. Therefore, all raw HBase rows are operated by using a set collection, which guarantees no duplicated rows.

IV. Experiment and Discussion

According to our preliminary work, OpenTSDB has no cache mechanism. All of the query requests need to retrieve the data from HBase. Therefore, the HBase query time plays a major role in time consuming as it involves reading from a hard disk. Our cache mechanism has been designed for reducing the latency of the data retrieval of OpenTSDB by applying Memcached.

The environment for testing the caching system on the OpenTSDB is setup on Docker [13], OpenTSDB 2.3.0, Memcached 1.5 and HBase 1.2.6 on the standalone mode are wrapped with Docker 17.12.1-ce. Both OpenTSDB and HBase run on Oracle JDK 8. In addition, the host machine uses Ubuntu server 16.04 with Intel Core i3-7100 2 cores, 4 threads, RAM 40GB, Hard disk WD WD20EZRX with speed SATA 6 Gb/s, and its capacity is 2 Terabytes. The tested data has been generated randomly for 10 million data points, the interval between the data points is 5 seconds. It means that an HBase row stores 720 data points. We set up our experiments on two systems. The first system is the original OpenTSDB with HBase, called OpenTSDB without cache. The second system is composed of OpenTSDB integrated with our caching system, Memcached and HBase, called OpenTSDB with cache.

The experimental setup reflects the graphical user's behavior that they select a time series range and move the window range from left to right or vice versa. On the other hand, the users often change the time range to view on the same time series. Each scenario sends query requests six times on both systems varying the percentage of two-contiguous query ranges. For instance, the whole time series range is 1-100. The first query range is 1-50. The second query range is 25-75. Thus, the percentage of both queries is 50%. We vary the percentage of overlapping query ranges on five scenarios: 100%, 75%, 50 The OpenTSDB with cache system is tested by applying five chunk sizes (1, 16, 256, 512 and 1024) in order to find optimal chunk size. In addition, the chunk size also affects the item size to be cached in Memcached. Each experiment has been tested repeatedly three times with the same parameters and scenarios.

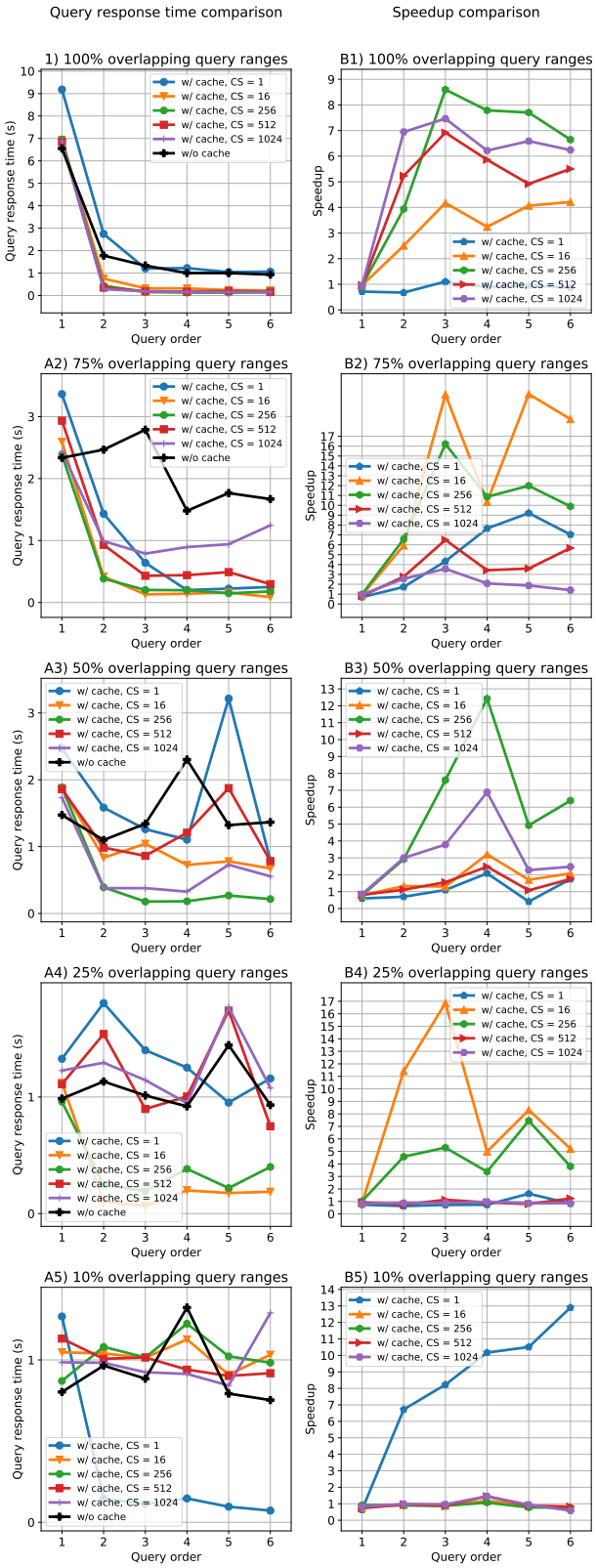

Fig. 6. Comparison between the OpenTSDB and OpenTSDB scanner function with cache varying the chunk size. A (Left side) shows average query response time comparison. B (Right side) shows average speedup over the traditional OpenTSDB. Each row of the results has varied percentages of overlapping two adjacent query ranges.

In the experiment, we have measured the response time of raw data retrieval excluding the post-processing module. Moreover, we always validated the results before the measurement and guarantee all data points returned from the OpenTSDB integrated with our cache mechanism is similar to the original OpenTSDB.

Figure 6 shows the response time and speedup of our system. The first scenario starts with 100% overlapping query range, Figure 6 A1 shows the scanner time. Figure 6 B1 shows the speedup. The first query always is a compulsory miss, the time might not be reduced. However, in the next queries, it shows that our design cache can achieve the speedup about 4x-8x when the chunk size is 256. The scenario with 75%, 50% and 25% overlapping query ranges also achieve the speedup more than the original OpenTSDB as shown in Figures 6 B2, B3 and B4, respectively. The optimal chunk size also is 256 of which the result also shows that the 75%, 50% and 25% overlapping query ranges can achieve the speedup about 6x-16x, 3x-12x and 4x-7x, respectively. Lastly, that the scenario with 10% overlapping query range shows that the scanner time resembles with the original OpenTSDB as shown in Figure 6 B5. When chunk size is 1, the speedup is better because serialization is extremely fast when the data size is small as the fragment contains only one HBase row.

In conclusion, the percentage of overlapping query ranges is related to the number of cached items. Typically, when the number of cached items is high, it can help reduce the response time. In case that the chunk size is low, the number of fragments will be high. Thus, the number of Memcached requests also is high. The network overhead affects the query time. In case of big chunk size, the overhead is caused by the serialization that merges all HBase rows into a one-byte array. According to the experimental results, the optimal chunk size is 256. However, the optimal chunk size depends on the behaviors of the users and the data characteristics. In summary, our approach can provide faster response time than the original OpenTSDB for redundant data retrieval queries.

V. Conclusions

This paper proposes a cache mechanism to deal with time series data on OpenTSDB with HBase. The approach might be applied to other time series databases. It also reveals the methods to handle sequence data. However, this research focuses on non-real time data retrieval for the user's interested time series data. We compared the response time of the raw data query module on OpenTSDB integrating our cache mechanism with the original OpenTSDB by varying five scenarios. Our approach can retrieve the 10 million data points faster than the original OpenTSDB. Our approach can significantly reduce the query time as to reduce the response time after the data has already been stored in the cache. The response time can be reduced around 3x-16x when compared with HBase that stores the data in hard disks depending on the user query behaviors and the setting parameters like the chunk size. This work is especially beneficial as the user usually moves the window range of data and change the parameters of the query. It can also support many users. Moreover, both OpenTSDB and Memcached can be distributed. Therefore, the issue about how to apply cache in a distributed time series database will be interestingly challenging.

VI. ACKNOWLEDGMENTS

This work is funded by the excellent academic record scholarship, Graduated School, Prince of Songkla University.

References

[1] N. Agrawal and A. Vulimiri, "Low-latency analytics on colossal data streams with SummaryStore," in Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pp. 647–664, ACM, 2017.

[2] R. S. Weigel, D. M. Lindholm, A. Wilson, and J. Faden, "TSDS: high-performance merge, subset, and filter software for time series-like data," Earth Science Informatics, vol. 3, pp. 29–40, June 2010.

[3] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," vol. 27, no. 7, pp. 1920–1948, 2015.

[4] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," vol. 29, no. 11, pp. 2581–2600, 2017.

[5] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-memory Time Series Database," Proc. VLDB Endow., vol. 8, pp. 1816–1827, Aug. 2015.

[6] A. Deshpande, S. Nath, P. B. Gibbons, and S. Seshan, "Cache-and-query for Wide Area Sensor Databases," in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03, (New York, NY, USA), pp. 503–514, ACM, 2003.

[7] T. Wlodarczyk, "Overview of time series storage and processing in a cloud environment," in 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 625–628, 2012.

[8] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A New Model and Architecture for Data Stream Management," The VLDB Journal, vol. 12, pp. 120–139, Aug. 2003.

[9] S. Nath, A. Deshpande, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan, "IrisNet: An architecture for internet-scale sensing services," in Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '03, pp. 1137–1140, VLDB Endowment, 2003.

[10] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pp. 157–168, ACM.

[11] "Opentsdb overview." http://opentsdb.net/overview.html, Mar. 2018.

[12] "Hbase schema." http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html, Mar. 2018.

[13] "Docker - build, ship, and run any app, anywhere." https://www.docker.com/, Mar. 2018.