HeteroTSDB: An Extensible Time Series Database for Automatically Tiering on Heterogeneous Key-Value Stores

Yuuki Tsubouchi
SAKURA Internet Inc.
Email: y-tsubouchi@sakura.ad.jp

Asato Wakisaka
Hatena Co., Ltd.
Email: astj@hatena.ne.jp

Ken Hamada
Hatena Co., Ltd.
Email: itchyny@hatena.ne.jp

Masayuki Matsuki
Hatena Co., Ltd.
Email: songmu@hatena.ne.jp

Hiroshi Abe
Lepidum Co. Ltd.
COCON Inc.
Japan Advanced Institute of Science and Technology (JAIST)
Email: h-abe@jaist.ac.jp

Ryosuke Matsumoto
SAKURA Internet Inc.
Email: r-matsumoto@sakura.ad.jp

***Abstract*—Monitoring service providers have arisen to meet increasing demands from system administrators. Providing a monitoring service requires high-resolution time series, long-term retention, and high write scalability to a time series database (TSDB). Nevertheless, no research to date has examined proposed TSDBs with consideration of the extensibility of data structures for function additions of a monitoring service. As described herein, we introduce a TSDB architecture for automatically tiering on heterogeneous key-value stores (KVS) that combines with an in-memory KVS and an on-disk KVS. Our proposed architecture, by unbundling the data structure on memory and disk as different KVSs, enables the TSDB extensibility to duplicate writes to new data structures optimized for each use case without reducing writing efficiency or data storage efficiency.**

1. Introduction

As internet services have become widely available, demand is increasing from users for availability and response speed of systems supporting these services. Therefore, it is necessary to monitor the systems for system administrators to identify a problem quickly when a failure occurs in the systems. However, system administrators bear burdens such as system setup work and continuous hardware and software update work because monitoring requires an additional stand-alone system. A monitoring service enables system administrators to alleviate these burdens.

A monitoring system periodically collects metrics for measuring the state of systems such as memory usage for the target system. Then it stores the time series data in a time series database (TSDB) [11] and displays metrics as graphs of the time series. It is necessary to record the metrics in a TSDB with high resolution to avoid missing the state change in a short time to analyze the unplanned events using the graphs. Additionally, it is necessary to retain past data for a long time to analyze causes after failure recovery and to analyze changes of past load situations for capacity planning.

However, writing high-resolution metrics to storage increases the number of writes to the disk. Furthermore, storing high-resolution metrics for long periods increases storage usage. Therefore, writing operations and data storage must be effective. Furthermore, a monitoring service has grown, for instance, the number of users has increased, the scale of system managed by the users has increased or the number of metrics collected per unit time has increased. Consequently, a monitoring service for the TSDB requires high write scalability.

Earlier methods [16], [10], use key-value store (KVS) [7], which is a database management system (DBMS) that has a basic element of data as a key-value pair. Actually, KVS makes it easy to scale write processing horizontally because it is easy to distribute data in element units that are not mutually dependent. Because earlier methods configure a DBMS as a single building block, we change the DBMS software itself to make extensions such as adding a new data structure.

However, a problem exists: it is difficult to extend the DBMS with tightly coupled functions because, in general, extending loosely coupled software is easier than extending tightly coupled software because loosely coupled software has fewer influential parts. Growth of monitoring services requires addition of new functions such as the analysis of long-term metrics other than merely displaying graphs. Therefore TSDB extensibility is required. The TSDB extension here means, for example, writing the same data to separately prepared new data structures such as *index* optimized for each use case by changing the write processing for each of the memory and the disk.

This research is intended to realize a TSDB architecture that achieves the extensibility of the data structure and write scalability without reducing either write efficiency or data storage efficiency. First, we adopt a loosely coupled architecture that adds different DBMSs matching the characteristics of the extension rather than changing the DBMS software itself to extend the data structure. The interface for writing and reading is integrated so that it resembles each TSDB user while using multiple different DBMSs. Next, our architecture uses KVS as a DBMS for writing scalability. Furthermore, our architecture uses in-memory KVS [20], which has all the data in the memory, instead of on-disk KVS, which has all the data on the disk, to reduce the number of writes to the disk and to improve writing processing efficiency.

Therefore, we propose a loosely coupled TSDB architecture with heterogeneous KVSs that improves data storage efficiency by on-disk KVS and improves the write efficiency by in-memory KVS. Specifically, it is possible to reduce memory usage on in-memory KVS by automatically tiering between KVSs, with acceptance of writing with in-memory KVS and moving old data on the memory to the on-disk KVS. In short, our architecture combines benefits of high write efficiency of in-memory KVS and data storage efficiency of on-disk KVS. Our architecture also unbundles the writing process to memory and disk from specific DBMS to extend the data structure merely by changing the part of the writing process.

We organize the paper as described below. Section 2 provides related works. Section 3 details our architecture. Section 4 details the implementation of our architecture. Section 5 presents experimental results of performance measurement. Section 6 describes the experience of deploying our architecture to the production environment of Hatena's Mackerel [13] monitoring service. In section 7, we summarize our contributions.

2. Related works

The TSDB for monitoring services requires write efficiency, data storage efficiency, and write scalability with extensibility of data structure. However, existing methods present difficulties related to extensibility of the data structure. Therefore, we describe the features and shortcomings of existing methods.

OpenTSDB stores data points in HBase [9], which is a KVS that places data points on disk, and which has write scalability by HBase's distribution mechanism. Similar to Bigtable [8] using Log Structured Merge Tree (LSM-Tree [15]), HBase's storage engine uses log precedence write, a memory structure called MemStore, and a disk called HStore. Data stored in the MemStore are written to the HStore collectively so that the number of times of disk writing per unit time is reduced compared with writing directly to the disk.

Gorilla [17] is an in-memory TSDB particularly addressing reading performance by holding all the recent data points on memory. It is storage called Operational Data Store (ODS) using HBase hold long-term data in the layer. Gorilla specifically examines reading data at high speed because data points are written simultaneously to both in-memory TSDB and ODS. However, it does not reduce the write processing to ODS.

InfluxDB [10], similar to HBase, implements Time Structured Merge Tree (TSM), which optimized LSM-Tree for time series data. Data points accumulated in the data structure in memory are written collectively to disk. Furthermore, InfluxDB achieves high writing efficiency and data storage efficiency because of the delta encoding method proposed by Gorilla.

From the viewpoint of data structure extensibility, OpenTSDB and InfluxDB are configured as a single DBMS in which various constituent elements are tightly coupled; it therefore can be said that their extensibility of the data structure is low. Although Gorilla probably extends the in-memory data structure because of requests that cannot be satisfied by HBase, there is no mention of extensions other than hastening reading processing.

3. Proposed Method

We propose HeteroTSDB, a TSDB architecture that unbundles memory and disk data structures into different KVSs, and which loosely couples them without reducing either write efficiency or data storage efficiency and which achieves write scalability by KVS, independent of the architecture of a specific DBMS to ensure the extensibility, which is a challenge of the existing method described in section 2.

3.1. HeteroTSDB Architecture

The HeteroTSDB architecture adds a different DBMS that matches the characteristics of the extension instead of changing the tightly coupled DBMS to increase the extensibility of the data structure. It also integrates each DBMS so that it can be regarded as one DBMS by adopting a combined architecture. HeteroTSDB uses KVS, which is easy to distribute, to achieve write scale-out. Furthermore, HeteroTSDB realizes automatically tiering by which the data points are written to the in-memory KVS. The old data on the in-memory KVS is moved to the on-disk KVS in units of time series to increase the write efficiency and the data storage efficiency to a practically acceptable level.

However, an in-memory KVS entails the difficulty that data on the memory disappears. HeteroTSDB resolves that difficulty by write-ahead logging (WAL) [14], by which data points are written as logs on disks of servers independently from the in-memory KVS before written to the in-memory KVS. The WAL restores volatilized data points from logs to the in-memory KVS when the in-memory KVS breaks down and data points are volatilized. HeteroTSDB implements WAL by writing data points via message broker [12], which saves the received message and sends the message to the prepared subscribers.

Figure 1 shows the mechanism of HeteroTSDB architecture. First, the writing process is the following.

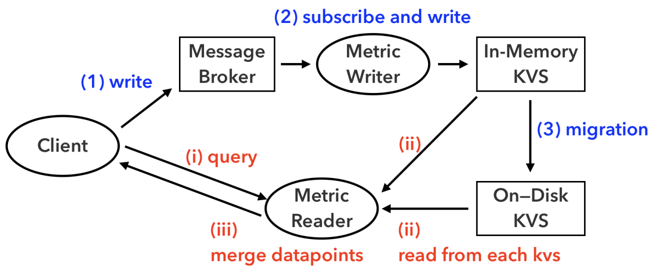


Figure 1: HeteroTSDB architecture.

(1) The client writes to the message broker. If the write to the message broker is completed, then a response is returned to the client.
(2) Asynchronously with the client writing, MetricWriter subscribing to the message broker writes to the in-memory KVS.
(3) Move data points with a timestamp older than a certain amount from the in-memory KVS to on-disk KVS in the background.

Next, the reading process is the following.

(i) The client sends a query including the metric name and timestamp range to MetricReader.
(ii) MetricReader reads the metric series from each KVS based on the condition of inquiry.
(iii) MetricReader reads from each KVS, combines the metric series and returns it to the client.

3.2. Time Series Data Structure

In the HeteroTSDB architecture, time series data must be stored in a key-value format to store time series data on KVS. Figure 2 shows the time series data structure in the key-value form combined with the timestamp alignment, division into time window, and the hash map. In Fig. 2, the *name* is the metric name, *timestamp* is a 64-bit integer type value representing the UNIX time of a data point, *value* is a numeric value of 64 bit floating point type representing the value of a data point, and wtimestamp is a UNIX time representing the start time of the time window.

Timestamp alignment: Align the timestamp to a multiple of the resolution before writing data points and specify the list of timestamps included in the range of the timestamp range statically to support range searching for KVS.

Division into time window: Store multiple data points in the same metric series in one key-value pair instead of storing one data point in one key-value pair to reduce the number of times of reference to KVS. The data points are contained within the limited size by dividing the metric series into fixed-width time windows.

Hash map: Use the hash map to prevent data loss when a temporary error occurs during the write process to KVS. The hash map preserves idempotent because the data points having the same timestamp are overwritten.

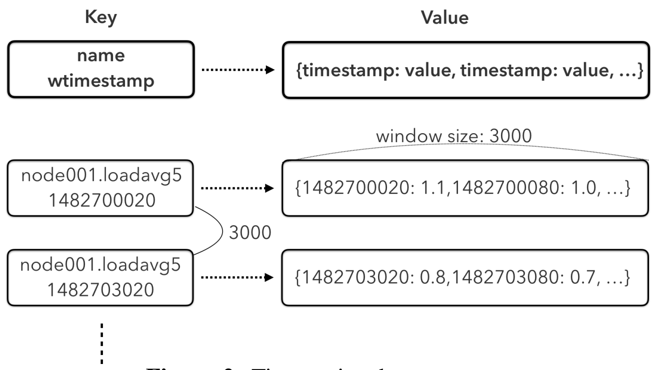


Figure 2: Time series data structure.

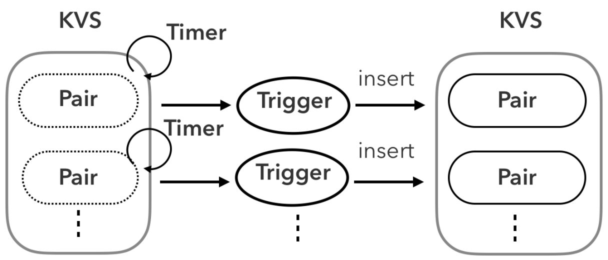


Figure 3: Data movement between KVSs: timer method.

3.3. Data Movement Between KVSs

In the HeteroTSDB architecture, it is necessary to move only old data from in-memory KVS to on-disk KVS while receiving new data points. Therefore, we propose a *timer method* and a *counting method*.

Figure 3 portrays the timer method: a timer is set in units of key-value pairs of KVS; when the timer becomes 0, the key-value pair is delivered to a trigger registered in advance. Then the trigger executes the data movement processing of the key-value pair. The timer method enables concurrent processing easy in the trigger processing associated with each key-value pair because it is sufficient to process the data movement of one key-value pair and because each processing can be processed independently. The timer is implemented using time to live (TTL) set in units of key-value pairs.

We propose a counting method based on KVS that does not implement TTL, specifically examining that data points of metric series are written at regular intervals. The counting method counts data points in the time window when MetricWriter writes the data points to the in-memory KVS. MetricWriter reads the time window from the in-memory KVS if the number of data points is over a certain number. Then MetricWriter writes it to the on-disk KVS and deletes it from the in-memory KVS. However, a problem exists by which the data points remain on the in-memory KVS if the metric series are no longer written. They are not moved to the on-disk KVS. Therefore, independent from MetricWriter processing, the residual data points on the in-memory KVS move to the on-disk KVS.

3.4. Data Structure Extension

We present a method to extend the data structure: prepare different DBMSs with different data structures (hereinafter *additional DBMSs*) in addition to the in-memory KVS and the on-disk KVS having the data structure described in section 3.2, replicate the time-series data to the additional DBMS, and add processing to refer to the additional DBMS to MetricReader.

The methods for replication writing include the following two points. *Real-time writes*: Write the time-series data to additional DBMSs by process units such as MetricWriter that subscribe to the message broker. *Batch writes*: Write the same contents as the time-series data written in the on-disk KVS to the additional DBMSs after writing to the on-disk KVS in either the count method or timer method.

Real-time writes immediately write the time-series data written from the client to the message broker to the additional DBMS. They enable MetricReader to refer to the replicated data. Batch writes reduces the number of writes to the additional DBMS similarly to on-disk KVS, whereas the timing of writing to the additional DBMS is delayed by the latency to accumulate data points on the in-memory KVS. Furthermore, consistency of data between DBMSs is guaranteed in either method by retrying from the beginning process to an idempotent data structure on the additional DBMS, even if an error occurs during writing to the DBMS. Our approach can select either real-time writing or batch method according to the properties of the additional DBMS for the tradeoff between the number of writes to the additional DBMS and the delay time to replicate the data.

4. Implementation

We show an implementation of the HeteroTSDB architecture using Amazon Web Services (AWS). Our implementation constructs heterogeneous KVSs in a 3-tier structure, so that the data storage efficiency is superior to that of a 2-tier structure. In addition, operating heterogeneous KVSs and a message broker generate a greater burden than when operating a single DBMS to build the environment of OS and various applications running on the server, update for countermeasures against software bugs and vulnerabilities, and scale-out the server per an increase in load. Therefore, our implementation uses the serverless platform [19] to automate work easily by directly executing application programming interface (API) because the cloud service provider automatically processes these burdens.

Figure 4 presents the system configuration. Amazon Kinesis Data Streams [4] acts as a message broker, allowing users to pass received messages to the arguments of an AWS Lambda [6] function. MetricWriter is a Lambda function linked with Kinesis Data Streams. Amazon ElastiCache for Redis [3], [18] is an in-memory KVS. Amazon DynamoDB is an on-disk KVS built on SSD (solid state drive). Amazon S3 [5] is an object storage with a unit cost of about one-tenth of that of DynamoDB. MetricCleaner is a Lambda function that scans the data remaining on the in-memory KVS by batch processing and moves it to the on-disk KVS and launches the Lambda function periodically by Amazon CloudWatch Events [1]. MetricReader is a Web server on Amazon EC2 [2].

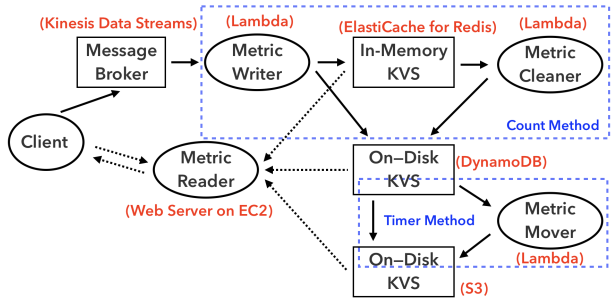


Figure 4: System configuration.

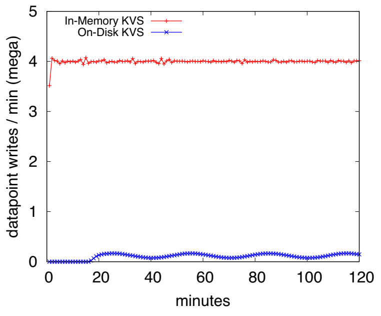
Our implementation uses a timer method that we can readily implement on the serverless platform in data movement from DynamoDB to S3 because, if TTL becomes 0, the Lambda function activated by DynamoDB Triggers deletes key-value pair. The processing of the Lambda function (MetricMover) here receives the deleted key-value pair and merely writes it to S3. However, because Redis has no function to link with the Lambda function such as DynamoDB in moving data from Redis to DynamoDB, our implementation uses a counting method to realize it on the serverless platform.

5. Performance Evaluation

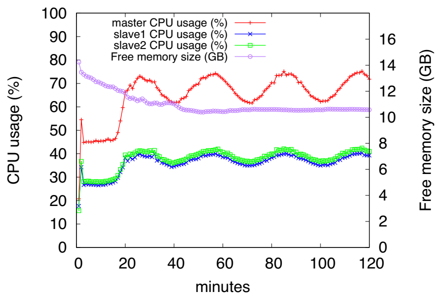
We evaluated write efficiency, data storage efficiency, and write scalability by part of the implementation shown in section 4 to confirm the effectiveness of the HeteroTSDB architecture. In the experiment, we built the experimental environment shown in Table 1 and implemented the benchmark to write the metrics to the message broker. We assume a situation in which an agent for sending the collected metrics to the monitoring system periodically runs on the host.

We implemented a benchmark program capable of specifying the number of metrics, the number of agents, and the agent's metric transmission interval and one transmission to imitate the production environment. For the following benchmark, the metric transmission interval was 1 min, the agent simultaneous transmission metric number was 100, and the number of data points in the same metric sequence was 15 on the in-memory KVS. Table 2 presents a combination of variable parameters for each benchmark, excluding these fixed parameter values. Write Capacity Units (WCU), defined by AWS, is the capacity for writing to a DynamoDB table. For example, one write per second of 1 KB or less for one item on the table becomes 1 WCU.

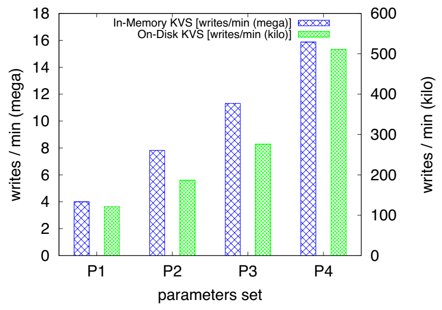
First, to evaluate the write efficiency, we confirmed whether the number of times of writing to the on-disk KVS decreases, or not, by receiving a write in the in-memory KVS, with comparison to the case of writing directly to the on-disk KVS.



(a) Number of minutes written.



(b) Resource usage of in-memory KVS.



(c) Write scale-out performance.

Figure 5: Experimental result.

TABLE 1: Experimental environment

Roll	Item	Specification
Benchmark Client	EC2 instance	
	type	c 5.4 xlarge
	OS, Kernel	Amazon Linux 2, 4.14
Message Broker	Kinesis Data Streams	
	number of shards	32
MetricWriter	Lambda	
	Memory Amount	1600 (MB)
	Runtime	Node.js 8.10
In-Memory KVS	ElastiCache for Redis	
	Type	cache.r4.large
	Version	3.2.10
On-Disk KVS	DynamoDB	

TABLE 2: Scale-out performance benchmark parameter

	Number of agents	Number of in-memory nodes	WCU
P1	40,000	3	6,000
P2	80,000	6	12,000
P3	120,000	9	18,000
P4	180,000	12	24,000

Next, to evaluate the data storage efficiency, we confirmed whether the data points can be transferred to the on-disk KVS, or not, without the free memory amount of the in-memory KVS becoming 0. By operating the benchmark program for 2 hr using the P1 parameter of Table 2, we observed and graphed changes in the number of minutes of writing to the in-memory KVS and the on-disk KVS, the amount of free memory in the in-memory KVS, and the change in the CPU utilization rate.

Figure 5(a) shows the transition of the number of write times. Figure 5(b) presents the transition of the amount of free memory in the in-memory KVS and the change in the CPU utilization rate.

Finally, we checked whether the number of writes scales-out to evaluate the write scalability when increasing the capacity of each KVS. For each combination of parameters of Table 2, we ran the benchmark program for 2 hr, found the average value of the number of written data points, and graphed it. Figure 5(c) shows the scale-out performance of the in-memory KVS and the on-disk KVS. However, we excluded the first 15 min of the benchmark from the calculation of the average value to prevent the number of times of writing to the on-disk KVS from unduly decreasing. This was true because the number of times of writing of the on-disk KVS became 0 because the data points were stored in the in-memory KVS for the first 15 min.

Figure 5(a) shows that the number of minutes of the in-memory KVS writing was constant at a value of about 4 M and the number of times of writing to the on-disk KVS was 0 in the first 15 min. Then it took a value between 70k and 170k. The value 4M matches the number of write data points for the minute specified by the benchmark program. The number of writing per minute to the on-disk KVS is about one-twentieth of the number of writes per minute of the in-memory KVS.

The number of writes per minute to the in-memory KVS is expected to be about the same as the ones to the on-disk KVS because, if we perform the same benchmark with the method of writing directly to the on-disk KVS, then the on-disk KVS will accept the number of write times accepted by the in-memory KVS instead of the in-memory KVS. Therefore, our architecture apparently reduces the number of writes to the on-disk KVS to about one-twentieth of those used with the method of writing directly to the on-disk KVS.

Figure 5(b) shows that the free memory usage of the in-memory KVS decreases gradually from the maximum capacity of 16 GB until the benchmark time increases to 50 min; then it transits while taking a constant value at around 10.5 GB. *Master* in figure 5(b) denotes the master node in the cluster, and *slave1* and *slave2*, to which MetricWriter is distributed as a part of the reading process, denote the two slave nodes. The CPU utilization of the master node was about 45% until 15 min pass but is constant. Then it increased to about 60% and to about 75% value between the waveform shapes. The shape of the CPU utilization graph of the slave nodes resembles that of the master node because, after 15 min, about 40% is the maximum value, about 35% is the minimum value.

The memory usage on the in-memory KVS is expected to increase monotonically with time because the benchmark client used for the experiment transmits data points having the same metric name and the same timestamp only once. The data points are only added for the in-memory KVS. Furthermore, it appears that our architecture remains constant with the memory usage of the in-memory KVS because the memory usage does not change with the lapse of time, by moving the data points to the on-disk KVS.

The parameter set for Figure 5(c) shows the combination of the parameters of Table 2. The nodes in the in-memory KVS increase by three because the in-memory KVS cluster is configured as one master node and two slave nodes as one shard. Figure 5(c) shows that our architecture most likely scales out because the number of write data points per minute increases linearly with the increase in the number of nodes and WCU.

6. Experience In Production

This section presents an example of deploying the HeteroTSDB architecture to the production environment of Mackerel, which is the monitoring service provided by Hatena Co., Ltd. Specifically, we describe the several unplanned events that occurred during the year from August 2017 through August 2018, which affected some portion of Mackerel's site availability. The system configuration in the production environment was equivalent to figure 4.

Two faults were found during the period above. In the first case, the write load concentrated on a specific node of the in-memory KVS. The memory consumption in the node reached the upper limit. Furthermore, as a result of the OS forcibly stopping the process, data of a plurality of specific metrics temporarily disappeared. For this reason, the data points that remained in Kinesis Data Streams were restored by running the Lambda function from the time before the time of loss, thereby recovering the lost data points. Apparently HeteroTSDB was able to resolve the shortcoming of data persistence at the time of the in-memory KVS failure by the message broker.

In the second case, more than the expected data points with the same metric name and the same timestamp are written in a short time. The write query size for the in-memory KVS exceeds the upper limit of implementation. Symptoms at that time were that errors occurred, the Lambda function continued to be retried, and processing of the entire MetricWriter was delayed. We modified the program of MetricWriter. Thereby, we eliminated duplication of data points having the same metric name and the same timestamp.

7. Conclusion

In this paper, we proposed HeteroTSDB, an extensible TSDB architecture for automatically tiering in-memory KVS and on-disk KVS, on the premise of heterogeneously DBMS configuration to enhance the data structure extensibility. We have implemented HeteroTSDB on an AWS serverless platform to reduce the burden of construction of multiple DBMSs and scale-out work. Our experiment confirmed that HeteroTSDB can withstand practical use while adopting a loosely coupled architecture for extensibility.

As future work, first, it is necessary to demonstrate that the HeteroTSDB is practical in terms of performance after comparison with other TSDBs. Next, we will implement extensions of the data structure and evaluate the extensibility of HeteroTSDB.

References

[1] *Amazon CloudWatch*, https://aws.amazon.com/documentation/cloudwatch/.

[2] *Amazon EC2*, https://aws.amazon.com/jp/ec2/.

[3] *Amazon ElastiCache for Redis*, https://aws.amazon.com/elasticache/redis/.

[4] *Amazon Kinesis Data Streams*, https://aws.amazon.com/jp/kinesis/data-streams/.

[5] *Amazon S3*, https://aws.amazon.com/jp/s3/.

[6] *AWS Lambda*, https://aws.amazon.com/jp/lambda/.

[7] Cattell R, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[8] Chang F *et al.*, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 4:1–4:26, 2008.

[9] George L, *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*, 1st. O'Reilly Media, Inc., 2011.

[10] InfluxData, *InfluxDB*, https://www.influxdata.com/time-series-platform/influxdb/.

[11] Jensen SK *et al.*, "Time Series Management Systems: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2581–2600, 2017.

[12] Kreps J *et al.*, "Kafka: a Distributed Messaging System for Log Processing," presented at the 6th International Workshop on Networking Meets Databases (NetDB), 2011, pp. 1–7.

[13] *Mackerel*, https://mackerel.io/.

[14] Mohan C and Levine F, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," presented at the ACM International Conference on Management of Data (SIGMOD), 1992, pp. 371–380.

[15] O'Neil P *et al.*, "The Log-Structured Merge-Tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[16] *OpenTSDB*, http://opentsdb.net.

[17] Pelkonen T *et al.*, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," *41st International Conference on Very Large Data Bases (VLDB)*, vol. 8, no. 12, pp. 1816–1827, 2015.

[18] Sanfilippo S and Noordhuis P, *Redis*, https://redis.io.

[19] *Serverless Computing and Applications*, https://aws.amazon.com/jp/serverless/.

[20] Zhang H *et al.*, "In-Memory Big Data Management and Processing: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.