DEVELOPMENT OF A SCALABLE AND FLEXIBLE DATA LOGGING SYSTEM USING NOSQL DATABASES

M. Kago#, A. Yamashita, JASRI/Spring-8, Hyogo, Japan

Abstract

We have developed a scalable and flexible data logging system for SPring-8 control. The current data logging system, powered by a relational database management system (RDBMS), has been storing log data for sixteen years. With the experience, we recognized the lack of RDBMS flexibility with respect to data logging such as the lack of adaptability in data format and acquisition cycle, the complexity of the data management, and the lack of horizontal scalability. To solve this problem, we designed a new framework. Two NoSQL databases, Redis and Apache Cassandra, were adopted to store log data. ZeroMQ messages packed by MessagePack were employed for communication. The prototype of the new system showed high performance and reliability. In this paper, we discuss its requirements and structure, the results of performance evaluations and its current status.

INTRODUCTION

The SPring-8 control system adopts the MADOCA [1] framework developed in 1995. A relational database management system (RDBMS) [2] is one of the features of MADOCA. Device information, operation parameters, and log data are stored in a single RDBMS. This system has supported the stable operation and development of SPring-8.

However, in light of the new requirements resulting from the recent improvement and future plans of the accelerator, this system will not have the adequate performance, capacity, or extendibility. In accelerator beam diagnostics, handling large volume data such as wave forms and image data has become a common task. In addition, there is a need for data acquisition with a shorter sampling cycle (< 10 Hz). These changes require the data logging system to increase writing performance and to handle large-volume data, but it is not easy to extend the system for the following reasons.

- The RDBMS has no horizontal scalability. The advantages of the RDBMS, such as joining tables and securing ACID [3], become bottlenecks when scaled out. The general method for improving server performance is to change to a high-spec server, but hardware costs are continuously increasing.
- The data management is complex. To improve writing performance in MADOCA, a number of log data are placed on one row to reduce the number of SQL statements needed when they are stored in RDBMS. When new data are registered on the RDBMS, a new table has to be created.
- The data acquisition has no flexibility. The polling system of the server-client model uses tight coupled ONC-RPC and is highly interdependent. Therefore, the data acquisition is limited by the OS and language environments.

To solve these problems and contribute to the further development of SPring-8, we developed a new data logging system as a part of the MADOCA II project [4]. The design concepts of the new system are as follows:

- Database and data acquisition need to be scalable such that data volume can be increased and performance can be improved at a low cost.
- It should be easy to migrate from the current system. The MADOCA manages data with human readable names such as "sr_mag_ps_b/current_dac." The new system will adopt the same methods.
- The new system should be highly reliable without an SPOF (single point of failure).
- Users without any knowledge of databases should be able to easily start collecting data.
- The RDBMS will be used conventionally for static information such as device information and operation parameters.

DATABASE SELECTION

We designed a storage system that consists of a database for the perpetual archive and a database for the real time data cache. The permanent storage has to be able to handle large-volume data with high performance, to increase performance by scaling out, and to have no SPOF. In SPring-8, furthermore, low latency access of the latest values is required because there is a constant need for these values by many accelerator control GUIs. Thus, a cache server keeping only the most recent values was adopted so that a high performance can be achieved.

NoSQL (Not only SQL) databases that have been actively developed in the web service fields satisfy our requirements. Most of them provide the required mechanisms such as high performance, horizontal scaling, and fine control over availability. Among these databases, we focused on Apache Cassandra [5] and Redis [6].

Apache Cassandra

Apache Cassandra is an open-source distributed database of the Apache project. Its features include a column-oriented data structure, high write performance, fault tolerance, no SPOF, and so on. It is especially easy to increase total throughput by adding more nodes to the system (see Fig. 1). These features meet our requirements. Furthermore, Most of the log data that we handle is time-series data comprised of a time stamp and value, and so Cassandra with its column-type data structure is the most suitable database for the perpetual archive.

#kago@spring8.or.jp

ISBN 978-3-95450-139-7

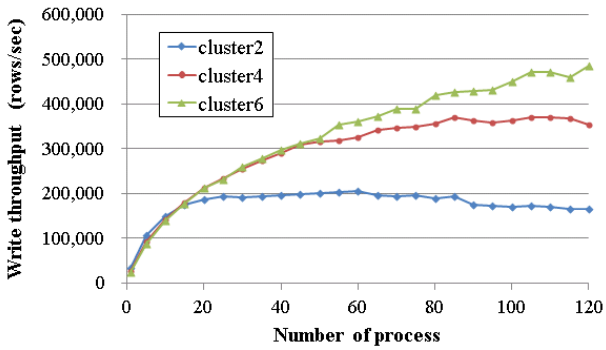

Figure 1: The write throughput when the access load is increased. Hardware specifications of a node are Xeon 2.93 GHz, 8 GB RAM, 64 bit Centos 6.2. Cassandra is version 1.0.5. Data size per row is 20 bytes, and a client inserts 1,000 rows at once.

Cassandra values availability and partitioning tolerance in the CAP theorem [7]. This means that its consistency provides a few guarantees and is called eventual consistency. When the data are taken from a cluster with six nodes and a replication factor of three, we found that the time required for guaranteeing consistency is as much as 1 sec.

To prevent this inconsistency, the cache server is utilized. The data are inserted into both Cassandra and the cache server in parallel. The consistency is complemented by a way to obtain the latest value from the cache server.

Redis

Redis is adopted for the cache server. It is an in-memory key-value store, and provides high performance access to the data. We verified that the write latency of Redis is lower than that of Cassandra (see Fig. 2). This low latency complements Cassandra's consistency and provides the real time data to the control GUI.

Another advantage of Redis is its support of a wide variety of structures such as list, set, sorted set, and hash.

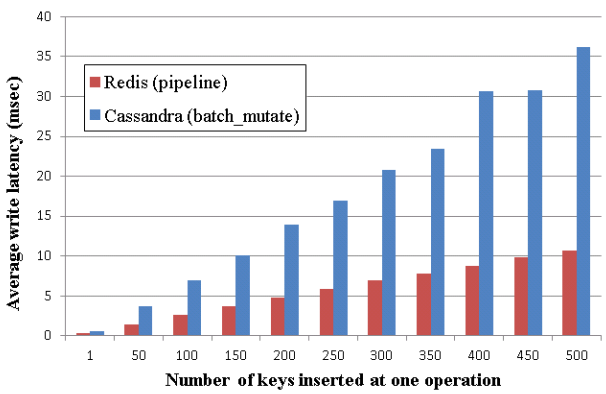

Figure 2: The write latency of Redis compared to Cassandra, where low latency is better. Testing conditions of Cassandra are the same as Fig. 1.

SYSTEM ARCHITECTURE

The new data logging system is designed on a three-layer model. Figure 3 shows the architecture of the new system. It is composed of clients who generate logs, a relay server, and the database engine. All devices are connected through a network. The log data is sent to several relay servers at the client's own timing. The relay server creates the database commands from the received message and inserts the log into the database.

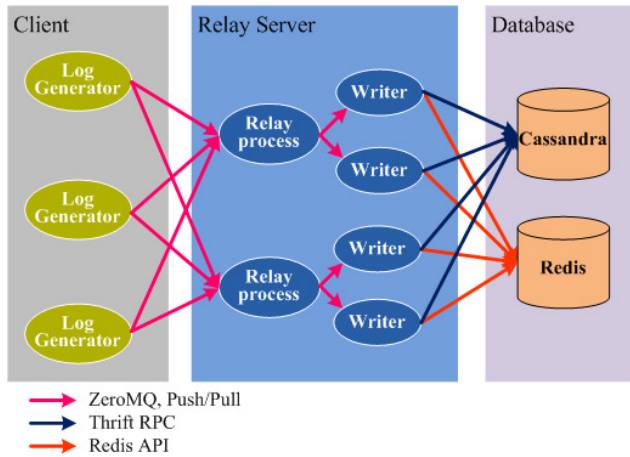

Figure 3: System architecture.

Client

Users who want to store the log into this system insert the prepared function into their data acquisition programs. The code packs the messages using MessagePack [8], and pushes the messages using the Push/Pull pattern of the ZeroMQ [9] communication library. The Push/Pull pattern sends messages from one sender to several receivers by the round robin algorithm and can easily realize load balancing. When the sender detects a problem with a receiver, the message is sent excluding this receiver.

Any platform or programming language that supports ZeroMQ and MessagePack can be a client of this system. Users can easily start or stop data acquisition because the client can be connected at any time to the pull socket of ZeroMQ.

Relay Server

The relay server works as a gateway between the client and database. Multiple processes run on the relay server: the relay process and writer processes.

The relay process manages the pull socket for receiving messages from the client and transfers the received message to a writer process. For communication between the relay and writer processes, the Push/Pull pattern of the ZeroMQ is adopted.

The writer process converts the received message into a database command and inserts the data into Cassandra and the Redis in parallel. As the API of both databases is synchronous, they are blocked during the write process.

ISBN 978-3-95450-139-7

To prevent waiting, several writer processes are activated to parallelize database access.

Cassandra

We designed the data structure to efficiently handle time-series data. The data structure is shown in Fig. 4. One row-key provides the information of one day's signal. The row-key name is formed from a signal name in addition to a date string and contains collections of columns. One column consists of a name formed from a timestamp, and a value. To provide a flexible data format such as an array, structure, or map, the logging data is serialized by MessagePack and inserted into the column value.

Cassandra distributes the data to each cluster node under the MD5 hash value of the row key. According to this mechanism, this data structure makes it possible to distribute data evenly into the Cassandra nodes. In addition, the reading performance will be improved when the data are accessed for a period of one day because a single node stores the data for one day of one signal.

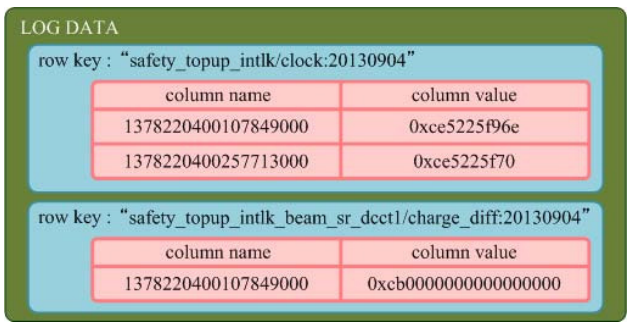

Figure 4: Cassandra's data structure.

Redis

Redis provides a wide variety of system types. In this system, a simple string type is employed. A key is formed from the signal name. The value is string data packed log data and its timestamp by MessagePack.

Redis works as a single process and does not support the cluster system. Furthermore, Redis has a simple master-slave replication function, but does not provide a failover mechanism. These features mean that Redis has no horizontal scalability and it has a single point of failure. To fix this problem, multiple Redis servers are parallelized by an access library that we developed. This library provides the functions for failover, load balancing, and horizontal scalability. These functions are realized by distributing the data to the target Redis server by the hash value of the key.

Data Access

Users can access the database using our prepared library. Since this library is compatible with the previous library, a user's code does not have to change. In addition, the programmers do not have to know in which database the data is stored. They can obtain data by just specifying the human-understandable signal name and time range.

TEST

The prototype of the new data logging system was evaluated long-term to check stability. Table 1 shows the test parameters. These data are actually generated in the SACLA [10]. The acquisition cycle is faster than the current one.

Table 1: Testing Parameters

Item	Specification
Hardware	Intel Xeon X3470 2.93 GHz 4 Core CentOS 6.2 64 bit
Redis	Version 2.6.10 Number of process: 4
Apache Cassandra	Version 1.1.5 6 nodes cluster (replica: 3) OracleJavaVM 1.6.0
Data Acquisition	Number of clients: 240 Number of relay processes: 4 Number of writer processes: 24
Log Data	Number: 47,397 Cycle: 1 Hz Average message size: 60 bytes

Write Test

The writing test was conducted for three months. The results show that no data went missing during the test and no impact was found on the writing performance even when the server was shut down.

The total write performance of the system was not measured because of the asynchronous writing characteristic, but the throughputs of each process were measured. It was seen that about 180,000 operations per second (ops/sec) can be transferred per relay process. The throughput of one writer process was about 5,000 ops/sec. From these results, it is estimated that the total throughput of the system is about 120,000 ops/sec or more.

Read Test

To simulate actual operation status as accurately as possible, the reading test was conducted along with the writing test. One reading client was activated and the latency was measured.

For Redis, the results of this measurement over 1,000,000 times showed that the average read latency was 0.26 ms and the standard deviation was 0.14 ms. Additionally, a test with a high load was provided. When 2,000 requests a second were made from 200 clients, the average read latency was 1 ms, with the worst latency being 6 ms.

For Cassandra, the read latency of the data for one day (86,400 points) was measured 10,000 times. The results showed that the average latency was 1.01 sec. and the

ISBN 978-3-95450-139-7

standard deviation was 0.18 sec. We found that the read latency could be shortened to 0.2 sec. by parallelizing the reading process.

CURRENT STATUS

We are now migrating from the previous MADOCA system. The new system has been installed in the actual environment. Figure 5 illustrates the structure and Table 2 shows the specifications of the servers.

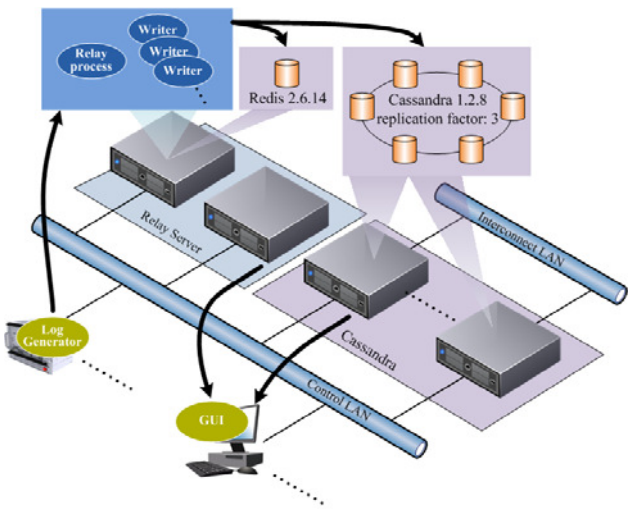

Figure 5: System structure. Arrows show the data flow.

Table 2: Specifications of Servers

Item	Relay Server	Cassandra Server
Processor	Intel Xeon E5-2430, 2.2 GHz, 12 core	
Memory	16 GB	
OS	CentOS 6.4 64 bit	
Storage 1	SAS 15 Kr/m 450 GB × 2 Raid 1	SAS 15 Kr/m 450 GB × 2 Non Raid
Storage 2	N/A	SATA 7,200 r/m 3 TB × 2 Non Raid
Network	1 Gb Ethernet × 2 ports	
Power Supply	Dual, Hot-plug, Redundant power supply, 550 W	Single, Hot-plug, 550 W

The system consists of eight 1U servers. Two out of the eight servers work as the relay server and the remaining six servers comprise the Cassandra cluster.

A relay process, twenty writer processes, and a Redis server run on one relay server. From the estimation of the write performance described previously, this structure can provide a throughput of about 100,000 ops/sec per relay server. This can handle the number of signal points that the current system handles. However, in consideration of availability and reliability, two relay servers were installed and the workload is balanced between them.

The Cassandra server mounts a 6 TB disk for data storage and a 450 GB disk for the commit log. The commit log disk that is constantly accessed for write durability, is separated to reduce I/O contention. Cassandra Versions 1.2 and later include the JBOD function that can distribute data onto several disks. The aggressive use of this function eliminates the need for RAID for data storage. In addition, it is expected that the write performance will improve by increasing the writing tasks.

Since Cassandra is a distributed system, it loads the network to handle its read/write requests and replication of data across nodes. Therefore, the network for communication between the clusters was constructed independently.

CONCLUSION AND FUTURE PLAN

We developed a data logging system using two NoSQL databases. Apache Cassandra was used as the permanent storage and Redis was used for the cache server. ZeroMQ was adopted for asynchronous communication. The complicated data structure was supported by MessagePack so that it could be stored into a database. The new system provides high performance, availability, and horizontal scalability. It has now been installed to replace the data logging system of the current MADOCA.

In the near future, data acquisition will begin small and the scale will gradually grow. Along with these future plans, we will construct an alarm system using this system and will pursue multi-platform support and web pages.

REFERENCES

[1] R. Tanaka et al., "The First Operation of Control System at the SPring-8 Storage Ring", Proceedings of ICALEPCS 1997, Beijing, China, (1997) p.1.
[2] A. Yamashita et al., "The database system for the SPring-8 storage ring control", Proceedings of ICALEPCS1997, Beijing, China, (1997).
[3] http://en.wikipedia.org/wiki/ACID
[4] T. Matsumoto et al., "Next-Generation MADOCA for The SPring-8 Control Framework", in these proceedings of ICALEPCS2013, San Francisco, California.
[5] http://cassandra.apache.org
[6] http://redis.io
[7] Nancy Lynch and Seth Gilbert, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", ACM SIGACT News, Volume 33 Issue 2 (2002), p. 51-59.
[8] http://www.msgpack.org
[9] http://www.zeromq.org
[10] T. Ishikawa et al., "A Compact X-ray Free-electron Laser Emitting in the Sub-angstrom Region", Nature Photonics 6, (2012) p. 540-544.

ISBN 978-3-95450-139-7