DEVELOPMENT OF THE J-PARC TIME-SERIES DATA ARCHIVER USING A DISTRIBUTED DATABASE SYSTEM, II

N. Kikuzawa#, H. Ikeda, Y. Kato J-PARC, Tokai-mura, Naka-gun, Ibaraki, Japan
A. Yoshii, NS Solutions Corporation, Shinkawa, Chuo-ku, Tokyo, Japan

Abstract

J-PARC (Japan Proton Accelerator Research Complex) consists of Linac, 3 GeV rapid cycling synchrotron ring (RCS) and Main Ring (MR). In the Linac and the RCS, data of about 64,000 EPICS records have been acquired for control of these equipment. The data volume is about 2 TB every year, and the total data volume stored has reached over 12 TB. The data have been being stored by a Relational Database (RDB) system using PostgreSQL since 2006 in PostgreSQL, but it is becoming that PostgreSQL is not enough in availability, performance, and flexibility for our increasing data volume. In order to deal with these problems, we proposed a next-generation archive system using Apache Hadoop, a distributed processing framework, and Apache HBase, a distributed database.

In this paper we are reporting that we have re-designed and re-constructed the cluster with resolving some issues, including enhancing hardware of master nodes, creating scripts to automatically construct nodes, and introducing monitoring tools for nodes. Having adjusted the configurations of HBase/Hadoop and measured the performance of our new system, we are also reporting its results and considerations.

INTRODUCTION

J-PARC is controlled with a lot of equipment, and we have been archiving a time series of operation data provided from about 64,000 EPICS records for the Linac and the RCS since 2006 [1]. PostgreSQL has been used in the present data archiving system, but it has some problems of capacity, extensibility, and data migration. In order to deal with these problems, we proposed a next-generation archive system [2][3] using Apache Hadoop [4], a distributed processing framework, and Apache HBase [5], a distributed database.

In the previous paper we reported that we updated the versions of HBase/Hadoop composing our test system [6], to conquer a single point of failure by making redundant a master node, and we showed issues to fix our tools in the new system. Having adjusted the configurations of HBase/Hadoop and measured the performance of our new system, we are also reporting its results and considerations.

RECONSTRUCTION OF THE CLUSTER

Replacement Master Node

We have updated Hadoop to the version 2.2.0 now. Hadoop 2.x provides a hot standby NameNode, which can take over the state that the previous active NameNode has provided, with no downtime. At least three master nodes are needed for this function to deploy ZooKeeper [7] and JournalNode. ZooKeeper is a high-performance coordination service for distributed applications, and both Hadoop and HBase depend on. JournalNode is one of the components of Hadoop. ZooKeeper and JournalNode are based on a majority decision among nodes, and it is meaningful to deploy them on an odd number of machines. We replaced the master node 2 and 3. The layout of the system is illustrated in Figure 1 and the spec of our system is listed in Table 1.

When the master nodes has been replaced, processes of Hadoop has been relocated, and memory allocation has been reconsidered. The allocation of memory shows in Table 2.

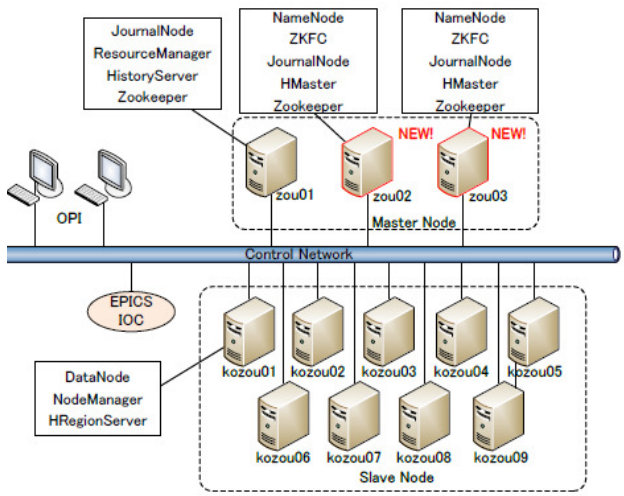

Figure 1: The archiving system configuration.

Table 1: Spec of Hadoop and HBase System

Master Node 1	DELL PowerEdge R610 CPU: Intel Xeon E5620 (4Core, 2.4GHz) MEM: 24GB HDD: 600GB x4 (RAID10)
Master Node 2, 3	DELL PowerEdge R320 CPU: Intel Xeon E5-1410v2 (4Core, 2.8GHz) MEM: 24GB HDD: 600GB x4 (RAID10)
Slave Node	DELL PowerEdge R410 CPU: Intel Xeon E5620 (4Core, 2.4GHz) MEM: 24GB HDD: 2TB x4
Software	OS: CentOS6.6 Hadoop: 2.2.0 HBase: 0.96..1.1 ZooKeeper: 3.4.5

#kikuzawa.nobuhiro@jaea.go.jp

ISBN 978-3-95450-148-9

Table 2: Memory Allocation (GB)

Master Node	1	2,3	Slave Node	
ZooKeeper	1	1		
Name Node		8	Data Node	1
ZKFC		1		
Journal Node	1	1		
Resource Manager	1		Node Manager	1
History Server	1		YARN container	6
HBase Master		4	Region Server	12
total	4	15	total	20

RAID and Mount Option

It is common sense to use non-RAID for slave nodes in Hadoop. The redundancy of data in Hadoop is made by holding the same data in several slave nodes, and RAID cannot be an alternative for the Hadoop's redundancy because RAID just makes HDD redundant, not for the whole node. In place of the hot-swap function we can use automatic scripts to construct from scratch. It is also said that, a Hadoop cluster constructed with non-RAID is faster than a one with RAID-0 because RAID-0 makes physical disks keep pace with the slowest one of them.

In the previous presentation, we constructed the slave nodes with RAID-5 with 4 physical disks. We have decided to follow the Hadoop common sense and have changed RAID-5 to non-RAID. To be exact, our RAID controller cannot handle non-RAID, and we have constructed RAID-0 for each one disk. Ironically, that gives us an advantage that we can still use battery backed-up memory in the RAID controller.

We use ext4 for the file system on the OS, which ext4 is a standard format in Linux these days. As to mount options, although Hadoop makes data redundant, we have chosen options to ensure persistence of data for a rainy day, for example, in the case that all nodes lose power supply all at once, with the exception that we invalidate I/O barrier because of battery backed-up memory in the RAID controller, as described above. We use *ordered* for the journal mode in order to shut out the possibility to see invalid data when it crashes, which setting is enough because Hadoop doesn't support random access and consequently it doesn't overwrite existing data. Our versions of ZooKeeper, Hadoop and HBase support Java6, which means *dirsync* is required because Java6 doesn't support fsync on a directory. To tell the truth, we overlooked that the slave nodes in Hadoop don't invoke fsync unless we set the Hadoop configuration property *dfs.datanode.synconclose* to be true, which property is hidden from the document. The performance test described later was done before we found it out.

Attribute of HBase

Data compression is a presupposition of HBase. There are two compression layers. The one is Data Block Encoding, which has been introduced since HBase 0.94, and it compresses sequential records by just storing the difference between records. The other is generally used compression algorithm like Snappy, which is applied after Data Block Encoding. Data Block Encoding is effective especially for the keys are relatively much larger than the corresponding values, and that compression matches our table design. We are showing what combination of Data Block Encoding and compression algorithm is best, in the performance test described later.

HBase can apply a bloom filter to search for data. HBase should search several files, and applying a bloom filter beforehand narrows down the files, at the cost of a little increasing the resource usage. This setting is a table attribute, and by default it is enabled for record keys. Bloom filters are not applied to extract records with a range condition while such a way is the only one we assume to search records, and we have decided to invalidate a bloom filter and avoid wasting its resource usage.

HBase prepares Block Cache for millisecond order latency. When Block Cache is enabled, data blocks once read from a disk are cached in memory. The cache itself is shared per region server, while it is per table whether Block Cache is used or not for the table, and this setting is a table attribute. As to our purpose, we assume to always get much data at once, and such a large data would be cached in client-side. The clients are not so many, and the data is rarely requested again. And moreover, by default, Block Cache can use 40% of the maximum heap and it might reduce the performance by consuming a lot of heap with frequently triggering GC. Because of these reasons we have decided to invalidate Block Cache.

Cluster Management

In order to monitoring servers we use Nagios, which is a mature tool and widely used in Linux servers. In Nagios anomaly detectors are introduced as plugins, and Nagios itself manages to schedule to trigger the detectors and notify users by e-mail if any.

Nagios is designed to use e-mail for notification, but our cluster is on the LAN which is basically separated from the outer network, and sending an e-mail to the mail server on the outer network is blocked off. For now we just explicitly login the node the Nagios is deployed on, and check by parsing the file that Nagios generates at fixed intervals for a web application, or run an automatic script to do the same things.

Some standard Nagios plugins, including pinging to servers, are formally distributed as a set. For other plugins you should write a script or something by yourself, or download from trusted web sites. We use two such convenient external plugins, check_ipmi_sensor and check_openmanage. Both is for collecting remote hardware status, but check_ipmi_sensor does via IPMI using a tool freeipmi, which comes from the installation disks of Linux OS, and check_openmanage does via SNMP using Dell OpenManage. They are similar but each has its merits and demerits. In order to monitor modules in ZooKeeper, Hadoop and HBase, we have created scripts as Nagios plugins, using the command *jps* in JDK for checking existence of Java processes, and invoking a query via socket communication if any. They just emulate the

ISBN 978-3-95450-148-9

procedures to check the modules by hand, and we would easily grasp what happens.

In addition, we have installed Ganglia [8], which was also installed in the old cluster. Ganglia collects numerical data, which differs from Nagios, and Ganglia would be suitable to detect unusual CPU or resources usages, or to tune the performance. Ganglia has an attractive feature of using multicast so that a lot of monitoring targets don't inflict a heavy load on the network.

Automatic Scripts to Construct Nodes

We have created the kickstat file to install and configure OS and required software packages coming from the installation disks, and have created a set of installation files to install and configure the rest of the required packages, in line with the views we have described above.

It would be preferable to construct a node automatically from start to end, but we should manually set up SSH keys or security concerns, and set up configurations to get hardware status via IPMI or about local environment dependent concerns.

PERFORMANCE TESTS

We measured the performance of registration and acquisition of records based on the assumption of actual usage in J-PARC, which is not intended for the general situation. We measured both the old cluster and the reconstructed new cluster.

Conditions

For the measurement of registration of records, we assume that there would be 10,000 EPICS records and we should regularly get a record at one second intervals from each EPICS record. We put records for one day into our cluster without stops and we measure its consumption time, and we repeat the same things 5 times in a row. Because each record is very small, in actual usage in J-PARC we should use a buffer to send many records together. According to this assumption we invalidate auto-flush and explicitly invoke flush at the end, and we count the consumption time till the flush is complete. We create random names as EPICS records, which is 30 characters randomly selected from 52 uppercase and lowercase alphabetical letters. We generate data from random double precision values, selected from 0.0 (inclusive) to 1.0 (exclusive). Splitting regions in advance is generally preferable for HBase, although it is not trivial in our actual usage, and we expect that the balancer wakes up regularity and automatically splits and moves regions between nodes with distributing loads. As to the performance measurement, because we execute it in a short period and we should not expect the balancer, we pre-split regions and distribute loads from the beginning. To put it concretely, we assume that the randomly generated names are equally distributed in the name space, and equally divide the space by the first character of the names into regions whose number is same as that of our region servers.

For the measurement of acquisition of records, we make use of the environment after the measurement of registration. Assuming that there are 5 clients simultaneously connecting to the cluster, we use 5 threads and make each thread retrieve records for one random day for 10 random channels, and we measure its consumption time for each thread. HBase would not be designed for such acquisition; HBase gives priority to low latency, which is against to Hadoop which gives priority to throughput. Moreover, for data acquisition many clients' random accessing is suitable for HBase because it well distributes loads to multiple regions servers, improving its overall performance advantage. Despite these features, we are still interested in the performance of a few clients accessing large sequential data, along the assumption of actual usage in J-PARC.

As to buffers in client-side, we use the default size of the write buffer, which is 2MB, and we use 20,000 records for the read cache, which corresponds to 2MB if we count 100 bytes for one record.

As described before, we examine what combination of Data Encoding Block and compression algorithm is best for our table design. To put it concretely, we examine {none(non-use), *diff*, *fast_diff*} for Data Encoding Block, and examine {none(non-use), GZ, Snappy, LZ4} for the compression algorithm.

Results

The results of the performance measurement are shown in the following tables and figure. Table 3 shows the consumption time to register data for one day, averaging the times measured repeatedly 5 times in a row. Figure 2 shows its transition instead of averaging in the case of the compression combination none-none. Table 4 shows the consumption time to retrieve data, averaging the times for threads. Table 5 shows the total amounts of data written in the Hadoop distributed file system (HDFS), which are measured in the new cluster but theoretically they rather depend on the combination of Data Block Encoding and compression algorithm.

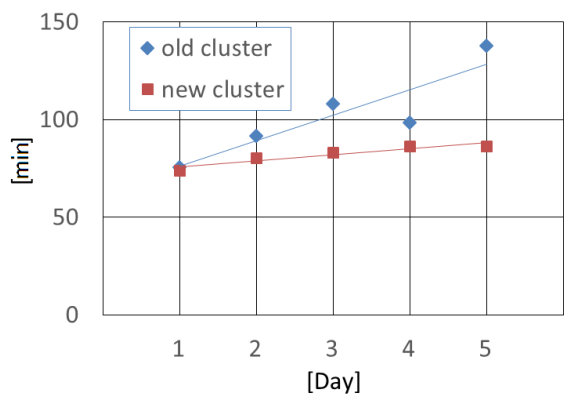

Figure 2: Write time in the none-none Case (min).

ISBN 978-3-95450-148-9

Table 3: Write Test (min)

	none	GZ	Snappy	LZ4
old cluster				
none	102.3	59.0	64.3	64.8
diff	57.4	54.0	58.9	59.4
fast_diff	56.7	55.9	58.5	58.8
new cluster				
none	81.9	77.6	74.9	80.8
diff	67.6	68.6	75.4	87.3
fast_diff	69.0	71.7	76.4	71.4

Table 4: Read Test (sec)

	none	GZ	Snappy	LZ4
old cluster				
none	7.3	8.3	8.8	7.1
diff	8.9	10.0	11.0	9.4
fast_diff	7.5	8.9	8.2	9.4
new cluster				
none	7.2	8.2	8.0	8.5
diff	9.8	8.5	10.0	11.3
fast_diff	8.6	8.6	9.0	8.3

Table 5: Sum of Stored Files (GB)

	none	GZ	Snappy	LZ4
none	286.2	64.1	80.5	76.7
Diff	53.1	47.3	52.9	53.1
fast_diff	53.8	48.7	53.8	53.9

Consideration

For the measurement of registration of records, the old cluster writes data 20-40% faster except the case of none-none, where the total amount of data written in HDFS is extremely larger than the other cases and the old cluster degrades its performance down according as written data in HDFS is increased. That indicates, RAID-5 supported by a RAID controller is superior on a lower load, and parallel accessing physical disks is superior on a higher load. We should have used much larger amount of data to point out specifically the advantage of the new cluster.

As to the combination of Data Block Encoding and compression algorithm, *diff* and *fast_diff* in Data Block Encoding have superior compression ratios and also have superior writing speed. Because we generate data from random values and the data almost always changes, it results in the compression ratio of diff being a little higher than that of *fast_diff*, although we expect *fast_diff* is more effective in the actual usage in J-PARC, where there would be many unchanged data to be recorded. Snappy and LZ4 prefer compression speed to compression ratios, but when applying diff or *fast_diff* we just find they have no effect to compress and waste time. On the other hand, GZ is said that it has the same compression speed as a fraction of that of Snappy and LZ4, but we don't find out such a disadvantage in our result. That would be because our hardware has much faster CPU in comparison with I/O.

In summary, our measurement result suggests that the new cluster has more scalability to the amount of data than the old cluster, but we should have used much larger data in order to point out specifically. Applying *diff* or *fast_diff* in Data Blocking Encoding is quite effective for our table design. Under applying either of them, only GZ is meaningful as compression algorithm.

CONCLUSION

We have archived to establish the procedure to construct a cluster of the practical use level. Now we are planning update the version of HBase, Hadoop and ZooKeeper. Because we had experience of being involved in troubles of their version compatibility, before everything we have just place the focus about making clear the procedure. Having achieved the goal, we are ready to update their versions, and adjust and fix.

We have made the data store redundant, but we also have to make the data capture tool redundant, otherwise it becomes a single point of failure from the viewpoint of data capture. It will be natural to use ZooKeeper currently working with Hadoop and HBase to select active/standby states of the redundant data capture tool.

REFERENCES

[1] S. Fukuta, et al., “Development Status of Database for J-PARC RCS Control System (1)”, Proceedings of the 4th Annual Meeting of Particle Accelerator Society of Japan, August 2007.
[2] A. Yoshii et al., “J-PARC operation data archiving using Hadoop and HBase” Proceedings of the 9th Annual Meeting of Particle Accelerator Society of Japan.
[3] N. Kikuzawa et al., “Development of J-PARC TimeSeries Data Archiver using Distributed Database System”, Proceedings of ICALEPCS2013.
[4] http://hadoop.apache.org/
[5] http://hbase.apache.org/
[6] N. Kikuzawa et al., “Status of Operation Data Archiving System using Hadoop/HBase for J-PARC”, Proceedings of PCaPAC2014.
[7] http://zookeeper.apache.org/
[8] http://ganglia.sourceforge.net/

ISBN 978-3-95450-148-9