Facility Information Management on HBase: Large-Scale Storage for Time-Series Data

Hideya Ochiai
The University of Tokyo / NICT
jo2lxq@hongo.wide.ad.jp

Hiroyuki Ikegami
The University of Tokyo
ikegam@hongo.wide.ad.jp

Yuuichi Teranishi
NICT / Osaka University
teranisi@cmc.osaka-u.ac.jp

Hiroshi Esaki
The University of Tokyo
hiroshi@wide.ad.jp

***Abstract*—A very large number of sensors on facilities such as HVAC, light control systems and electric power meters, periodically submit their status information to Cloud platforms these days. As the amount of data can easily get petabyte scale, we must consider the use of distributed application layer storage for managing such facility information, which is often formatted on time-series data. This paper describes FIAPStoragePeta, petabyte scale storage for facility information access protocol (FIAP), proposing the architecture and the scheme of such data management on HBase. In this work, we have identified three requirements to the design of HBase row keys for implementing this storage using HBase. Though, we have not finished petabyte scale experiments, our preliminary evaluation results have shown good performance for managing large scale facility information. It has achieved scalable data retrieval on the data of 10 million sensors with properly balancing loads on distributed data storages.**

***Keywords*—Internet of Things, Time-Series Data, HBase, Hadoop**

I. Introduction

In the age of the Internet of Things, a very large number of sensor readings gather at Cloud data platforms. In facility management applications, the platform should collect from millions of sensors on thousands of buildings every minute. The amount of sensor readings collected in the platform can easily get one petabyte.

HBase[5] is known as distributed petabyte scale storage, that is widely-applicable, highly-available and highly-dynamic. System administrators can operate it from gigabyte scale storage, but by adding computers, they can extend it even to a petabyte scale. Here, the use of HBase and the design of HBase keys are open to the users. HBase itself just allows to persistently store values with lexicographic-ordered keys. Whether the user can take the advantages of HBase or not, is totally dependent on the manner of use for their applications.

In this paper, we focus on the use of HBase for facility information management, especially time-series data collected from sensors or facilities such as HVAC (Heating, Ventilation and Air-Conditioning), light control systems, environmental monitors and electric power meters. Targetting at managing thousands of buildings on a Cloud platform, we have launched a project of developing FIAPStoragePeta – intended ”petabyte scale storage for facility information access protocol (FIAP)”. This paper describes the architecture and design of FIAPStoragePeta, proposing a facility information management scheme on HBase.

HVAC Working Mode

Time	Value
2014-07-21 08:00:00	FAN
2014-07-21 08:30:00	FAN
2014-07-21 09:00:00	DRY
2014-07-21 09:30:00	DRY
2014-07-21 10:00:00	COOL
2014-07-21 10:30:00	COOL

Room Temperature

Time	Value
2014-07-21 08:00:00	25.6
2014-07-21 08:30:00	25.8
2014-07-21 09:00:00	26.2
2014-07-21 09:30:00	26.9
2014-07-21 10:00:00	25.5
2014-07-21 10:30:00	25.3

Fig. 1. Facilities generate sequences of ”time-value” pairs: i.e., time-series data. FIAP manages each sequence by a unique identifier called ”Point ID”.

FIAP[7] is a communication protocol for data-centric building automation systems. It provides data storage for managing time-series data collected from facilities, which looks like Figure 1. In FIAP, a sensor (or an element of a facility) provides a sequence of ”time-value” pairs. The value may be a string type, for example, if the sequence represents the change of HVAC working mode. It may be a decimal type if the sequence represents the change of temperature. Each sequence has a unique identifier, which we call ”Point ID”, corresponding to the sensor. Though the examples are monitoring values, FIAP applies this data structure even to actuator status and calculated values: e.g., periodically summarized values by hour or day. The time intervals between values are not always the same.

This paper proposes the scheme of putting such time-series data onto HBase. Here, we design HBase keys so that FIAPStoragePeta (1) distributes the values on HBase appropriately, (2) optimizes the performance of sequential-read and (3) gets scalability for the increase of managed Point IDs. In this work, we have carried out preliminary experiments with five rack mount servers, and evaluated the performance regarding to these aspects.

We recognize that OpenTSDB[9] allows the management of time-series data on HBase. However, the major application of OpenTSDB is the management of working metrics of computer systems (such as network gear, operating systems and applications), and the design of HBase keys are for these applications, and not optimized for FIAP’s data model. And, though it may just an implementation issue, OpenTSDB does not support string type, which is necessary in facility information management. This was another motivation that we started FIAPStoragePeta development with studying the

implementation of OpenTSDB.

This paper is organized as follows. The next section describes FIAP's time-series data model, which we target at in this paper. Section III provides the architecture and the design of FIAPStoragePeta. We provide our preliminary evaluation works in Section IV. Section V shows the related works and discussions. Section VI provides the conclusion of this paper.

II. FIAP Time-Series Data Model

Facility information access protocol (FIAP)[7] defines a model of time-series data called "Point". In the operation of FIAP, a Point is associated with a sensor or an actuator of a facility, which values change over time.

Let P be a set of Point IDs that data storage of FIAP manages. A Point $p \in P$ has a set of values, which we denote by $V(p)$ in this paper. Here, a value $v \in V(p)$ has two parameters: $time$ and $content$. In the following discussions, $v.time$ represents the timestamp of the value, which is unique in $V(p)$. $v.content$ represents the value associated to the timestamp. In the operation, Point IDs are provided by universal resource identifiers (URIs) often with path-based data structures. For example,

$$p = \text{http://gutp.jp/EngBldg2/10F/102B1/Humidity}$$

may represent a sequence of humidity values of room 102B1, 10th floor of Engineering Building 2 in the University of Tokyo (operated by gutp.jp).

In FIAP, we specify a range of datasets by a Query (we denote it by q in this paper). A Query q may contain multiple keys. Here, we denote the set of the keys associate to q as $K(q)$. Each key $k \in K(q)$ specifies a point ID, and time range. We denote the point ID specified by a key as $k.id$, and the time-range is specified with the combinations of $k.eq$, $k.lt$, $k.gt$, $k.lteq$, $k.gteq$, $k.neq$, $k.select$. Here, eq represents "equal to", lt represents "less than", neq represents "not equal to". Thus, $gteq$ represents "greater than or equal to". $select$ specifies one of $\{maximum, minimum\}$ to select only one value at the maximum time edge or the minimum time edge in the specified time range by the other parameters. In this way, we can specify and get values from FIAP's data storage (we call this data storage FIAPStorage) using queries. For example,

- Query $q_1(k.id = p)$ specifies all the values of point p.
- Query $q_2(k.id = p \wedge k.select = maximum)$ specifies the latest value of point p.
- Query $q_3(k.id = p \wedge k.gt =$2014-01-01$\wedge k.lt =$2014-02-01) specifies the values between 2014-01-01 and 2014-02-01 of point p.
- Query $q_4(k_0.id = p_0 \vee k_1.id = p_1)$ specifies all the values of points $\{p_0, p_1\}$.

The original FIAP[7] defines WRITE and FETCH procedures. However, to make the discussion simple in this paper, we define SET and GET methods instead of WRITE and FETCH. Then, a Gateway sends sensor data to a FIAPStorage by SET, and an UI-terminal retrieves stored data by GET (see Figure 2). The Gateway and UI-terminal are the terms defined in FIAP[7]. Formally, we can define SET and GET as follows.

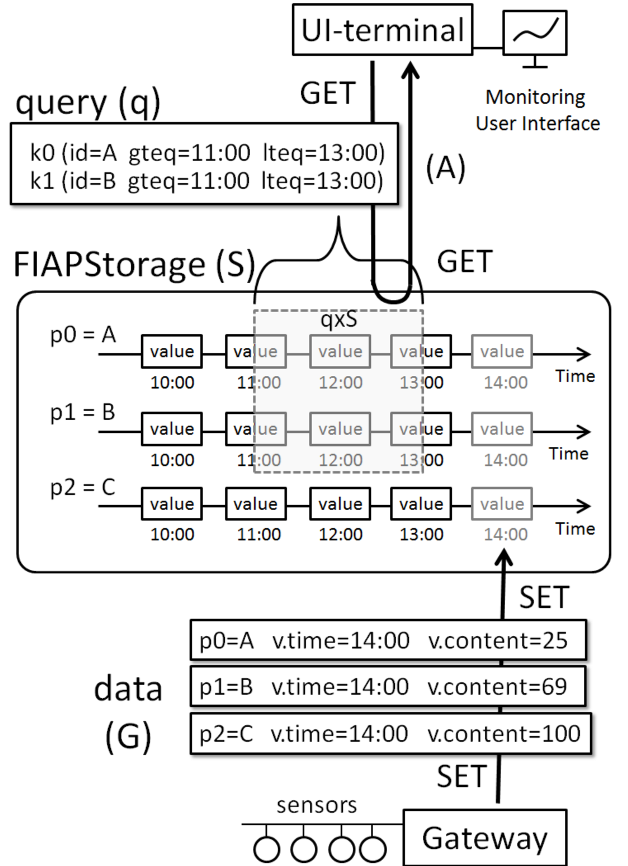

Fig. 2. FIAP time-series data model. FIAPStorage provides data storage with GET and SET methods. The Gateway sends the values to FIAPStorage by SET method, then it adds the values to the persistent memory. The UI-terminal can GET data with specifying the range of data by query. This paper redefine WRITE and FETCH procedures of FIAP as SET and GET for simplicity.

- **SET:** Let S be a set of points and associated values stored in FIAPStorage, and Let G be a set of points and associated values sending by a Gateway. The SET method is a process that adds G to S, that is:
$$S := S \cup G \tag{1}$$
- **GET:** The GET method is a process that takes points and values match to the query q, that is:
$$A := q \times S \tag{2}$$
Here, A is a set of points and associated values returned to the UI-terminal which has issued q. "$\times$" cuts out the dataset by the query.

III. FIAPStoragePeta

We have designed large-scale time-series data storage on HBase in the FIAP architecture. FIAPStoragePeta (intending peta-scale storage for facility information access protocol) is a code name of our project for developing such data storage.

A. Architecture

FIAPStoragePeta is organized with a FIAP protocol stack, driver for HBase, and a HBase platform (see, Figure 3).

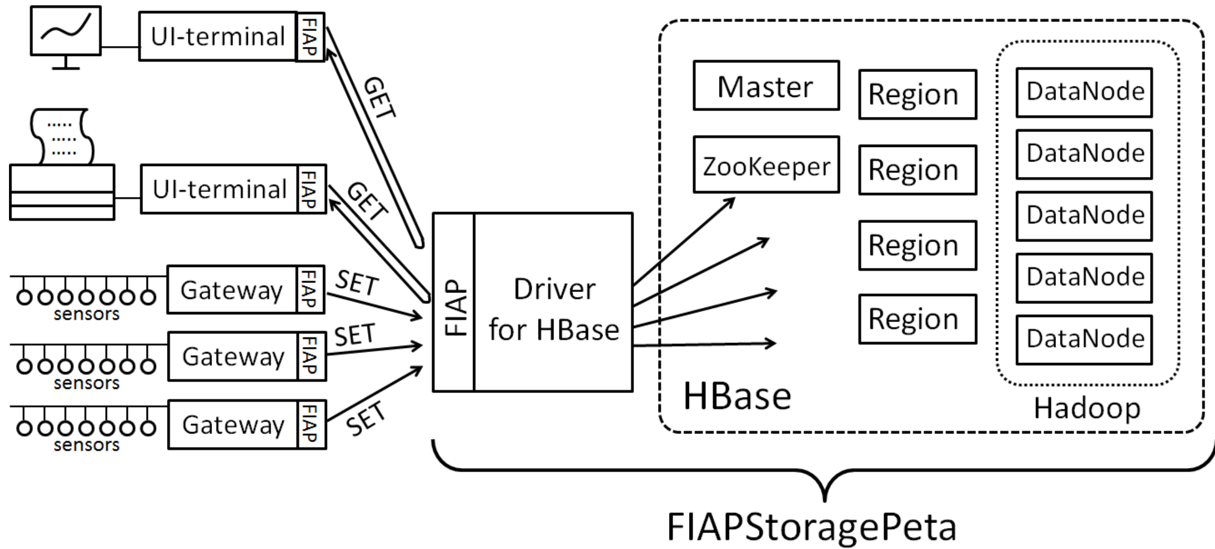

Fig. 3. Architecture of FIAPStoragePeta. The driver stores the sensor data SET from Gateways onto their HBase platform. UI-terminals GET data retrieved from the HBase by the driver.

The driver implements SET and GET methods, handling the process of data locating, storing and retrieving on HBase. Gateways send their data to FIAPStoratePeta by SET, and UI-terminals retrieve data by GET.

Here, HBase is operated by multiple servers providing key-value store. Though the key of HBase is three dimensional (i.e., row, column, time)[1], we decided to use only the row as the key of the FIAP's data values for the following reasons; (1) $v.content$ can be uniquely pointed by p and $v.time$; (2) the time of HBase is basically used for content version management (and time-range search seems not supported); (3) $v.time$ is not necessarily mapped to columns though OpenTSDB puts the second of $v.time$ for columns. Thus, in principle, we can use p and $v.time$ as the row key of HBase to uniquely coordinate $v.content$.

B. HBase Row Key Design for Time-Series Data

In the principle of HBase, the design of the row key structure is open to the users. Whether we can get full performance or not is totally depend on the design of it. We have identified the following three items as important features that FIAPStoragePeta should have to get full performance.

- **Do not make hotspots on the distributions of the row keys:** HBase divides the whole row-key space into tablets (some amount of range of row keys), and assigns those tablets to data nodes. If there are some hotspots in the distributions of the row keys, some specific data nodes get in charge of, and often suffer from, archiving larger amount of data and processing many requests. Thus, it is important to make sure that the distribution of the row keys does not have such a hotspot.
- **Put the values of each point locally, and keep them sorted by time on the row keys:** The row of HBase is a ordered key. For the following reasons, the time-series data of the same point should be put closely and sorted by time-order; (1) it allows the driver to directly specify the range of row keys corresponding to the time-range of the query, (2) it allows data read in block, (3) it allows the data to be kept sorted after the retrieval from HBase.
- **Get values without data filtering process even on a large number of points:** We generally apply $Hash$ to identifiers (i.e., Point IDs) to make near-uniform key distributions. But, as it is recommended to keep the length of row keys short to decrease the overhead, the bit-length of $Hash$ should be kept short. However, this leads to hash-conflict, especially **when the number of identifiers become large**, and we have to apply filtering process when retrieving data. If filtering process is necessary to remove such conflicts, it may result in the bottleneck of read performance.

Based on the discussions above, we have designed the structure of the row key as Figure 4.

To generate row keys, we have defined two functions: $Hash$ and Seq for point p. $Hash(p)$ takes top 16bit of the MD5 hash value of p, and $Seq(p)$ is the unique 24bit serial number corresponding to p. $v.time$ is given in second in Unix timestamp in UTC. From the most significant bit (MSB), the row key is organized with $Hash(p)$, $Seq(p)$ and $v.time$. The least significant bit (LSB) represents one second; i.e., the increase of the last bit means the increase of one second.

Under this definition, P should be distributed to the space of HBase row keys by $Hash$, meaning that in many cases we can avoid the hotspots discussed above. As $v.time$ is placed at LSB-side, the values of p (i.e., $V(p)$) should be put locally, and the time-order is kept on the HBase row's order. As $Seq(p)$ is a unique identifier corresponding to p, we can guarrantee that the pair of $Hash(p)$ and $Seq(p)$ (with 40 bits) should be corresponding to p. Though the maximum limit of points that this FIAPStoragePeta can manage is 2^{24}, if we have applied 40bit Hash instead of these pairs, we must worry about the

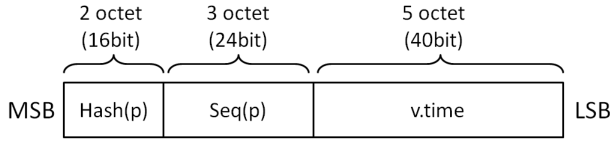

Fig. 4. Row key design for value v of point p. This row key locates $v.content$ with $Hash(p)$, $Seq(p)$ and $v.time$ on HBase. Under this definision, row keys should be distributed by $Hash$, but never conflict because of Seq, and $V(p)$ should be put locally.

hash conflicts because 40 bits are not large enough for 2^{24} elements.

C. Tables for FIAPStoragePeta

FIAPStoragePeta has three major tables on HBase: point table (T_P), ID map table(T_{ID}) and value table (T_V). T_P has one key and two columns named id and $maxtime$. $T_P.key$ is a point p managed in this FIAPStoragePeta, $T_P.id$ is $Seq(p)$ corresponding to the point, and $T_P.maxtime$ is the maximum timestamp of the values associated to the point. T_{ID} is for looking up p from $Seq(p)$. This has one key and one column named pi. $T_{ID}.key$ is $Seq(p)$ and $T_{ID}.pi$ is the point p corresponding to $Seq(p)$. T_V contains all the values associated to all the points of T_P. It has one key and one column named $value$. $T_V.key$ is the row key defined in the previous sub section (i.e., $Hash(p), Seq(p), v.time$), and $T_V.value$ contains the value (i.e., $v.content$).

D. SET and GET Procedures

As HBase allows parallel execution of processes on data, we must carefully consider that SET and GET processes may be done in parallel. For example, a client may call a SET method adding new point ID – during another SET process adding another new point ID. We must guarrantee that $Seq(p)$ should be unique even when this kind of conflicts happen.

With carefully considering this issue, we have implemented the procedures of SET and GET as follows.

- **SET procedure:** For point p and associated value v, SET first tries to lookup the row key from T_P. If it succeeds, it generates a row key for T_V with $Hash(p), Seq(p) = T_P.id, v.time$ and put $v.content$ to $T_V.value$. Here, if $T_P.maxtime$ is smaller than $v.time$, it updates $T_P.maxtime$ with $v.time$. If it fails to lookup the row key from T_P, it gets $last_id$, puts $T_P.id = last_id + 1, T_P.maxtime = v.time$ where $T_P.key = p$ and increments $last_id$ with locked, and puts $T_{ID}.pi = p$ where $T_{ID}.key = last_id+1$. Here, $last_id$ is a globally synchronized integer.
- **GET procedure:** For query key $k \in K(q)$, GET specifies the range of row keys based on $k.eq$, $k.lt$, $k.gt$, $k.lteq$, $k.gteq$, $k.neq$ and $Hash(k.id)$ and $Seq(k.id)$, and scan the row keys on T_V. If the k specifies the latest value of p (e.g., with $select = maximum$ property only), GET specifies the time range after $T_P.maxtime$, and picks up the latest one after the retrieval. This is because $T_P.maxtime$ may not always be synchronized to the latest one in the distributed environment.

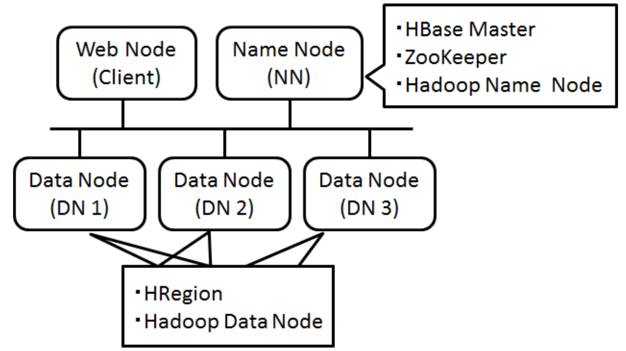

Fig. 5. Experiment configuration for FIAPStoragePeta preliminary evaluation. The name node operated HBase master, ZooKeeper and Hadoop name node. The data nodes operated HRegion and Hadoop data node. The web node operated the driver of FIAPStoragePeta (see, Figure 3). The performance of GET was measured in the web node.

IV. Preliminary Evaluation

This section provides our preliminary evaluation result regarding to (1) data distribution on HBase, (2) speed of sequential read and (3) scalability to point numbers. These three evaluation metrics corresponds to the discussion of Section III.B.

A. Experiment Setting

In our evaluation, we carried out experiments with five rack mount servers: using three servers as data nodes, one server as name node and the other server as web node. We installed CentOS 6.4 Linux 2.6.32-358 on those servers and software components as Figure 5. Here, each server was connected with Gigabit Ethernet.

Each server had a Xeon E5-1410 (2.8GHz 4Core) processor, 32GByte main memory and 2TByte hard drive for web and name nodes, and 8TByte hard drive for data nodes.

We have implemented SET and GET procedures in Java, carried out experiments and measured the execution time on the web node. In the experiment, we have changed some configuration parameters of HBase from the default settings as Dfs.replication=2 and hbase.snapshot.enable=true. IN_MEMORY option for HBase tables was false, and the block size of HadoopFS was 64KByte.

B. Data Distribution on HBase Platform

As we have designed to avoid hotspots in the distributions of row keys, the data should be distributed onto data nodes equally. To evaluate this, we have generated points and values as Table I, SET them to the FIAPStoragePeta, and checked the disk usage. Case 1 generated 1000 points and each point had 525600 values. This corresponds to one year data of 1000 points with every minute sampling. Case 2 generated 5000 points in addition to the dataset of Case 1, which means that it had 6000 points totally. In the same way, we generated dataset for Case 3 and Case 4.

Figure 6 shows the disk usage of the data nodes. Data were distributed to those nodes almost equally. The differences among data nodes were within 2.5%.

TABLE I. SETTING OF POINTS AND VALUES INTO FIAPSTORAGEPETA FOR DATA DISTRIBUTION EVALUATION: CASE 1 - 4

	Add to	Point ID Format	Generation Rule	Total Point IDs	Total Values
Case 1		http://a.hongo.wide.ad.jp/MPM/VA[x]	[x]=0000-0999	1000	525600000
Case 2	Case 1	http://fiap-gw.gutp.ic.i.u-tokyo.ac.jp/EngBldg2/10F/102A1[x]	[x]=0000-4999	6000	3153600000
Case 3	Case 2	http://fiap-gw.gutp.ic.i.u-tokyo.ac.jp/EngBldg2/10F/102A1[x]	[x]=5000-9999	11000	5781600000
Case 4	Case 3	http://www.hongo.wide.ad.jp/MPM/VA[x]	[x]=0000-0999	12000	6307200000

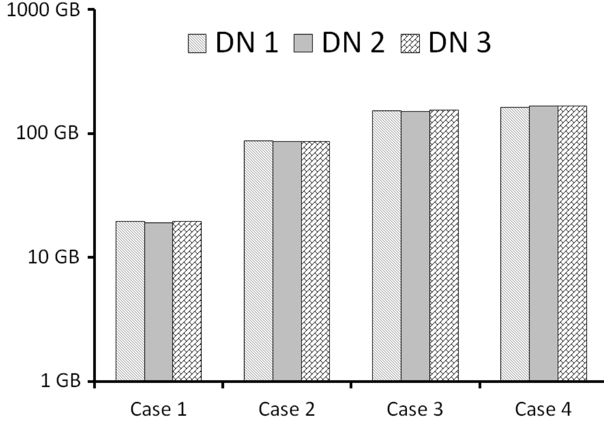

Fig. 6. The result of data distribution on data nodes for Case 1 - 4. The differences among data nodes were within 2.5% for all the cases.

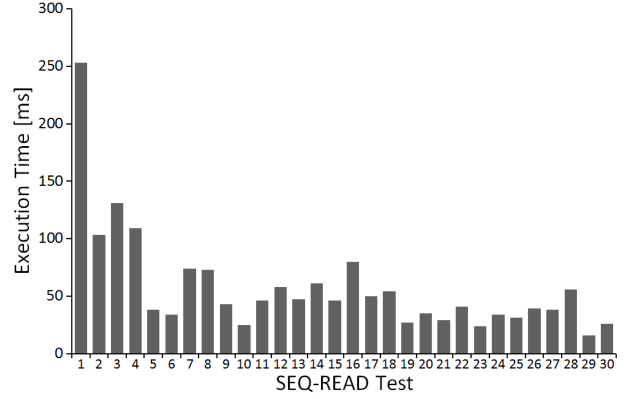

Fig. 7. Execution time of 10000 values sequential read. The average execution time was 48.8[ms], which was 3.01×10^3 times faster than random-read.

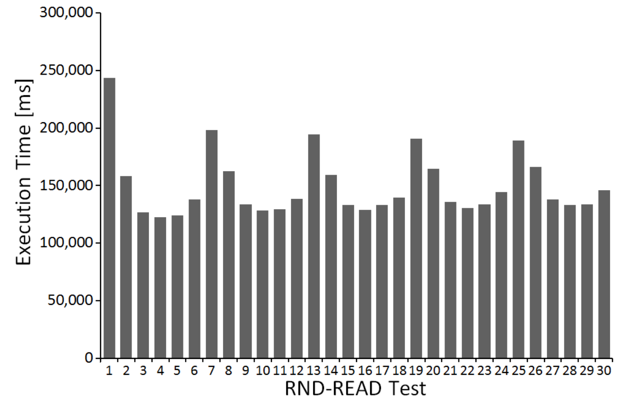

Fig. 8. Execution time of 10000 values random read. The averate execution time was 147[sec].

C. Execution Time of Sequential Read

As we have designed to put the same point's values locally and kept sorted, the sequential read should be done quickly. To evaluate this, we have carried out read experiments as follows.

With the data set of the Case 4 of the previous experiment, we have posted a time-range query that specifies 10000 values on a single point ID. We measured the time spent for the query execution. We carried out this measurement 30 times continuously on different point IDs; we call each measurement "SEQ-READ Test". To compare this sequential read process, we have also made random read experiment. In the random read, we have specified 10000 points with 10000 keys and one value for each point with eq property; we call each measurement "RND-READ Test".

Figure 7 and 8 shows the experiment results of sequential-read and random-read respectively. The average execution times were 48.8[ms] for sequential case and 147[sec] for random case. Here, in this calculation, we have excluded the execution time of the first tests, because the first tests have clearly taken longer time compared to the successive tests: probably because of loading necessary software on main memory at the first execution. The result shows that sequential cases performed 3.01×10^3 times faster execution than random cases, though in this preliminary experiment, we can say that the improvement for the random read is still required.

D. Scalability to the number of managed points

As we have designed HBase row key to allow data retrieval without filtering process even Hash conflict occurs for a large number of points, FIAPStoragePeta should have scalability especially in GETting values. To evaluate this, we have carried out an experiment as follows. We have cleared the HBase and set up points as "http://gutp.jp/FSP/[x]", where [x] = 0000000-9999999. In the experiment, we have generated first 10000 points and tested the performance of read, then generated additional 90000 points (totally 100000 points) and tested the read performance. In the same way, we have generated 10000000 (i.e., 10 million) points finally and tested the performance on that dataset. Here, each point had 60 values.

In the read process, we chose 1000 points at random and put them in a query. We did not specify any other parameters for the keys, meaning that the query specified all the values associated to the points. The total number of values to retrieve by one query was 60000.

Figure 9 shows the result of the average query execution time on the number of managed points in FIAPStoragePeta. The execution time was about 1.1[sec] when it was less than 1 million points. From about 1 million points, the execution time increased up to 1.9[sec] on 10 million points. This result indicates that FIAPStoragePeta has great scalability on the

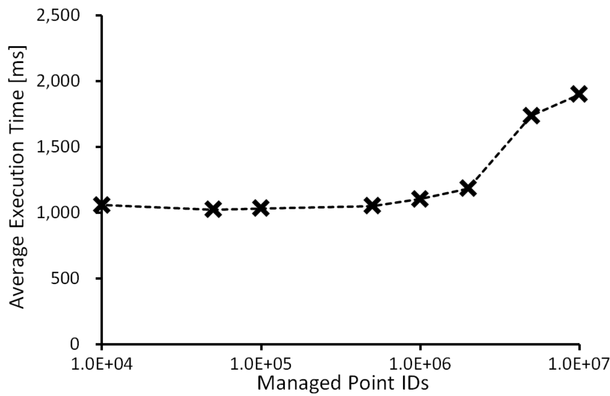

Fig. 9. The average execution time on a large number of managed points in FIAPStoragePeta.

increase of point IDs.

V. Related Work and Discussion

Time-series databases (TSDB) have been developed often in the context of network and system monitoring applications. One of the most traditional TSDBs would be the rrdtool[8], which is popular among network system administrators. With the rapid increase of time-series data with the advent of new IT systems, fine-grained real-time monitoring from thousands of hosts has become necessary. OpenTSDB[9] and tsdb[2] have been developed to satisfy these demands. Kacznmarski et al. [6] have studied the application of OpenTSDB to the real-time monitoring of content delivery networks. Groves et al. [3] have used OpenTSDB for distributed and push-based network monitoring at Yahoo!. OpenTSDB is built on top of HBase[5], which works like Google's Bigtable[1] on Hadoop[4].

FIAP[7] was originally developed for the delivery of time-series data over TCP/IP networks. However, FIAP defines time-series data for facility information management such as HVAC, light control systems, environmental monitors and power meters. Here, the context of time-series data is not the same as the network monitoring applications. Though Prasad et al.[10] have tested the applicability of smart meter data into OpenTSDB, we have needed another approach for storing FIAP time-series data on HBase. OpenTSDB manages data with "metrics" and "tags". But, FIAP manages them with "Point IDs". Instead of developing the mapping of Point IDs into metrics and tags of OpenTSDB, in this work, we have designed FIAPStoragePeta on HBase directly. We also needed to store string type values which were not supported by OpenTSDB.

According to our preliminary evaluation, FIAPStoragePeta used HBase platform quite efficiently though we still need further analysis on it. It appropriately balanced the load of time-series data onto the distributed data nodes, it achieved fast data retrieval in the sequential read cases, and it showed great scalability in retrieval time even on 10 million point IDs. In the future, we should evaluate it with larger number of data nodes and larger amount of values, and carry out measurement with many types of queries and data.

VI. Conclusion

In this paper, we presented FIAPStoragePeta, proposing a facility information management scheme on HBase. We described a model of time-series data used in facility management applications, and proposed the design of HBase row keys, table schemas and SET/GET procedures. To implement the following three requirements identified in this work; (1) no hotspots should be made in HBase row keys; (2) values of the same point should be put locally and sorted; and (3) allow to get values without filtering even on large number of point IDs, we used $Hash(p)$, $Seq(p)$ and $v.time$ in this order for the HBase row key of the value table T_V. According to our preliminary evaluation, (1) it appropriately balanced the loads of storing time-series data, (2) it achieved fast execution on the sequential read of the same point, (3) and it showed great scalability in retrieval time even on 10 million point IDs.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Seventh Symposium on Operating System Design and Implementation*, 2006.

[2] L. Deri, S. Mainardi, and F. Fusco. tsdb: A compressed database for time series. In *Traffic Monitoring and Analysis*, pages 143–156. Springer, 2012.

[3] T. Groves, D. Arnold, and Y. He. In-network, push-based network resource monitoring: scalable, responsive network management. In *ACM NDM 2013*, 2013.

[4] Apache Hadoop. http://hadoop.apache.org/.

[5] Apache HBase. http://hbase.apache.org/.

[6] K. Kaczmarski and M. Pilarski. Real-time content delivery networks monitoring. *Przeglad Teleinformatyczny*, 19(1):75–85, 2013.

[7] H. Ochiai, M. Ishiyama, T. Momose, N. Fujiwara, K. Ito, H. Inagaki, A. Nakagawa, and H. Esaki. FIAP: facility information access protocol for data-centric building automation systems. In *IEEE INFOCOM M2MCN workshop*, 2011.

[8] T. Oetiker. Rrdtool: Round robin database tool, 2013. http://oss.oetiker.ch/rrdtool/.

[9] OpenTSDB - a distributed, scalable monitoring system, 2014. http://opentsdb.net/.

[10] S. Prasad and S. Avinash. Smart meter data analytics using opentsdb and hadoop. In *Innovative Smart Grid Technologies-Asia (ISGT Asia), 2013 IEEE*, pages 1–6. IEEE, 2013.