2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing

Implementing Materialized View of Large-Scale Power Consumption Log Using MapReduce

Yuki Ise, Shintaro Yamamoto, Shinsuke Matsumoto, Sachio Saiki and Masahide Nakamura
Graduate School of System Informatics, Kobe University
1-1 Rokkodai-cho, Nada-ku, Kobe, Hyogo 657-8501, Japan
Email: {ise, shintaro}@ws.cs.kobe-u.ac.jp, {shinsuke, masa-n}@cs.kobe-u.ac.jp, sachio@carp.kobe-u.ac.jp,

***Abstract*—Smart city provides various value-added services by collecting large-scale data from houses and infrastructures within a city. However, it takes a long time for individual applications to use and process the large-scale raw data directly. To reduce the response time, we use the concept of *materialized view* of database. For a given requirement of an application, the proposed method constructs a materialized view for caching the application-specific data. In this paper, we especially develop a method that uses MapReduce for large-scale power consumption data stored in HBase KVS. We conduct an experimental evaluation to compare the response time between cases with and without the materialized view. As a result, the proposed method with materialized view is effective especially when application repeatedly access the same data, or when the application-specific data is derived from a large set of raw data.**

***Keywords*-large-scale house log, materialized view, high-speed and efficient data access, MapReduce, KVS, HBase**

I. Introduction

Smart city [1][2] is a next-generation city planning which strives to improve the efficiency of the city with ICT technologies. The smart city provides various value-added services according to large-scale information of houses and infrastructures within a city. The information includes operation log of household appliances and equipments, environmental data such as temperature and humidity, traffic data from roads and railroads. These are gathered from sensors and system loggers within the city.

In general, the information gathered from smart city is huge and wide variety, which is so-called *Big data*. Our long-term goal is to construct a universal data platform that can store and manage such smart city data in an integrated fashion. In our previous research, we have proposed a logging platform, called *Scallop4SC* (Scalable Logging Platform for Smart City) [3] [4], for managing large-scale log from smart houses within a smart city. In Scallop4SC, the large amount of house logs (e.g. power consumption logs) from the smart city are stored in distributed KVS (Key Value Store) [5]. It provides applications for various purpose or services such as visualization of power consumption, discovery of waste and peak-cut, with the logs through a API.

In general, data range and schema required by individual applications vary among the applications. For example, a service visualizing power consumption of home appliances requires time series data of power consumption for each appliance. On the other hand, an electric peak shaving service requires the total amount of power consumption of the entire house. Therefore, each application generally has to search, process and format the raw data for individual usage. However, due to the nature of smart city data (i.e., large-scale and wide-variety), it takes a lot of time for each application to search and process the raw data every time it requires.

The study aims to propose a method that provides such large-scale smart city data for various applications effectively. The key idea is to employ the concept of *materialized view* [6] of database. The materialized view is a technology that caches results of various queries in an actual table in advance, in order to improve the response time. With this concept, the proposed method constructs the materialized view beforehand for a given requirement of an application. We extensively use *HBase distributed KVS* [7] to store both the raw data and the materialized view. The application accesses the materialized view instead of the raw data, which achieves fast data access. Note that the smart city data are basically log data that are not updated. Therefore, we can transform the obtained raw data into the materialized view immediately. The transformation from the raw data to the materialized view is performed with Hadoop/MapReduce[8]. Thus, even if the volume of data increases much, the scalability is easily assured by adding computing nodes

In this paper, we evaluate the validity of the proposed method by especially focusing on the power consumption log. Specifically, for power consumption logs from 32 appliances within an actual smart home environment, we construct three kinds of materialized views (minutely, hourly and daily) using MapReduce batch processing. We compare the proposed method with the materialized view and the previous method accessing the raw data directly. As a result, it is shown that the proposed method outperforms especially in cases where the applications repeatedly access the same data, or the application-specific data is derived from a large set of raw data.

II. Preliminaries

A. Value-added Services in Smart City

The smart city provides various value-added services, named *smart city services*, according to the situation by big data within a city. Promising service fields include energy saving, traffic optimization, local economic trend analysis, entertainment, community-based health care, disaster control and agricultural support. Also for each field there are a wide range of services. For example, typical services in the energy-saving area are as follows:

1) **Power Consumption Visualization Service [9]:** The service collects data of power consumption from smart house and visualizes the use of energy from various viewpoints (e.g. houses, towns, devices, current power consumption, passage of past power consumption, etc.). This service is intended to raise user's awareness of energy saving by intuitively showing the current and past usage of energy.
2) **Wasteful Energy Detection Service [10]:** This service automatically detects wasteful electricity and notifies users using power consumption data and sensors data in a smart house. Furthermore, a user can review the detected waste occured in the past.
3) **Peak Shaving (Peak Cut) Service [11]:** This service reduces the maximum electric power usage in a house and a community by watching the usage of electricity. When the usage exceeds a pre-defined threshold, the service automatically stops or postpone energy-consuming operations.

The data for smart city services are gathered from various objects and systems within the city. Therefore, volume and variety of the data become quite large. Velocity (i.e., freshness) of the data is also imprtant to to reflect real-time or latest situations and contexts. Thus, the data for the smart city services is truly *big data*.

There exist services and applications that collect data from city and house. Most of the conventional applications store only necessary data with appropriate granularity, due to limitation of storage. However, such the limitation is recently relaxed significantly by cloud computing technologies. Thus, it is now possible to store various kinds of data as they are, and to reuse the raw data for various purposes. We are interested in constructing a data platform to manage the big data for smart city services.

B. Scallop4SC

In our previous research, we have developed a data platform, called Scallo4SC, which stores large-scale *house log* from the smart city [3][4]. Figure 1 shows the architecture of Scallop4SC. Various information within each house is collected from a logger deployed in the house. The data is then sent to Scallop4SC via a network, and is stored in HBase as the house log. The stored house logs are processed by Hadoop, distributed processing. Configuration information of the overall smart city is stored in MySQL relational database. The house logs are provided for external smart city services, via Scallop4SC API.

In the current prototype of Scallop4SC, we are gathering power consumption data from 32 appliances deployed in our actual smart home environment. The data is collected every three seconds. Therefore, 640 records per minute, 38,400 records per hour and 921,600 lines per hour are inserted to HBase. Thus, even the house log for the power consumption

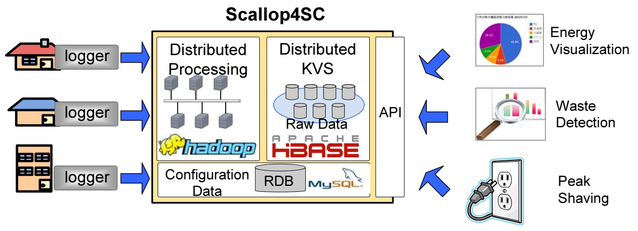

Fig. 1. Scallop4SC

only can be big data. Scapllop4SC are gathering other kinds of data, so the entire set becomes enormous.

C. Challange in Using Large-Scale Data

In Scallop4SC, the large-scale raw data of house log are accessed by various application. However, the necessary data range and schema vary from one application to another. The Scallop4SC API currently supports low level features for data acquisition only. Thus, each application has to process the obtained data by itself for desirable range and schema.

However, if the required data is derived from a large amount of raw data, then the application suffers from large processing time. Moreover, if the application repeatedly requires the same data, the application has to repeat the same calculation to the large-scale data, which is quite inefficient. To solve these problems, we need to develop an efficient method that can provide the required data for the individual applications.

III. Proposed Method

A. Architecture

To cope with the challenge, we here introduce a concept of *materialized view* of database systems. The range and schema of data required by individual applications can be considered as *view*, which looks up the raw data based on a certain condition. The ordinary view of database is represented as a query. Hence, the data is dynamically retrieved from the raw data when the view is accessed. On the other hand, the materialized view *statically caches* the query results in a table in advance. Thus, applications can access the cached data quickly.

Figure 2 shows the new architecture of Scallop4SC. In the proposed method, a developer of an application gives the new Scallop4SC a specification of required data, which is referred as *Data Spec*. Next, a component *factory* of Scallop4SC interprets Data Spec. and creates a *MapReduce batch program*, which transforms the raw data into application-specific data. Then, Scallop4SC executes the batch program, and generate a materialized view with API. The application accesses the API to quickly retrieve necessary data. The batch program is periodically executed on Hadoop distributed processing system. Note that the house log is not updated basically. Therefore, the raw data can be transformed at any timing.

Our long-term goal is to implement the whole architecture of Figure 2. However, in this paper we mainly develop a portion surrounded by a dashed box. Specifically, we first develop

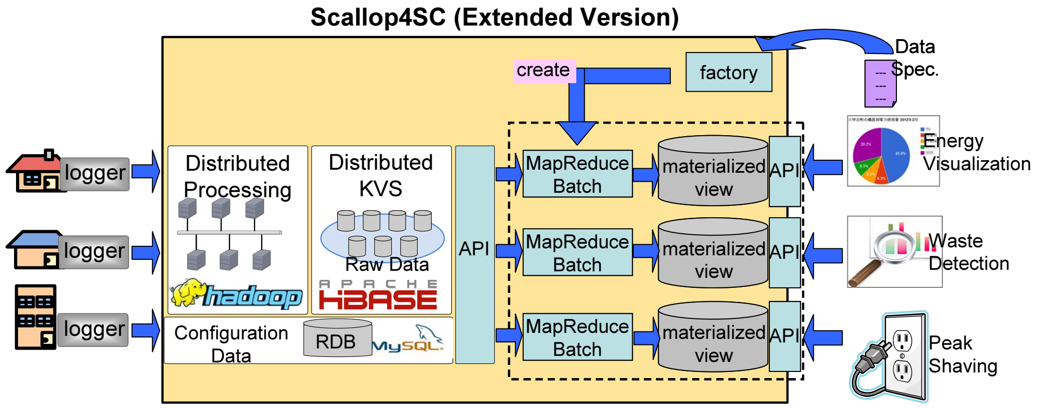

Fig. 2. Extended Scallop4SC

a design method of the materialized view. We then conduct preliminary evaluation of the proposed approach, by using a static MapReduce program for actual house logs. In this paper, we especially focus on the using power consumption log collected from our smart house environment.

B. Raw Data Stored in Scallop4SC

Before considering the design of the materialized view, we briefly explain the raw data stored in Scallop4SC. In order to store a variety of large-scale logs, Scallop4SC does not have rigorous data schema. Instead, it manages every data by a pair of key and value. All these pairs are stored in HBase.

Table I shows an example of power consumption logs obtained in our laboratory. A key of each row (called *row key*) is represented by a concatenation of "date and time (when the log is acquired)", "type (for what the log is)", "home (where the log is acquired)", and "device (from what the log is acquired)". The row key starts with date and time, since most batch processes are triggered by the date and time. Then, type, home and device are followed as they are arranged from coarse to fine granularity.

For each row, there are two column families: `data` representing the data value, and `info` representing meta-data explaining the data value. For example, the first row in Table I shows that the log is taken at 22:00:10 on January 18, 2013, and that the house ID is cs27 and log type is Energy, and that the deviceID is `pow001`. The value of power consumption is "140.3 W". Each attribute of `info` is stored in an independent column, and the application can refer to those values. A row also contains additional information such as `unit` and `location`.

C. Design of Materialized View for Power Consumption

Suppose now that there is an application that requires device-wise power consumption in a daily unit, from the raw data in Table I. Theoretically, the application should retrieve all the raw data matching given `date` and `device`, and calculate the total sum of `data`. However, if the number of devices is large or the interval of data sampling is short, the number of rows to be retrieved becomes huge. Accordingly, the overhead of the calculation becomes expensive. Furthermore, if the application requests the same data repeatedly, it is inefficient for the application to perform the same calculation many times.

Therefore, we consider a materialized view of this application, in which the device-wise power consumption is calculated beforehand by a batch process. We implement the view as a HBase table, whose row key is `device.date` (e.g., pow001.2013-01-18), and the value is the total of power consumption. Similarly, if another application requires hourly power consumption for each device, we create a HBase table whose row key is `device.dateThour` (e.g., pow001.2013-01-18T15). Generalizing the above idea, the proposed method constructs a materialized view as a HBase table as follows:

- **Row Key:** For each materialized view, a row key should be define so that the key clearly characterizes each row in the view. The key should be constructed by a concatenation of attributes of meta-data found in the row data. A row key specified in the material view is called *aggregation key*, since the key generally aggregates multiple raw data.
- **Value:** A value should correspond to a row key. The value is calculated from the raw data according to a condition of the aggregation.

The reason why using HBase for representing a materialized view is that the view should play a role of data cache, which quickly returns a value for a given key. It also reflects a requirement that we want to store application-specific and heterogeneous data without strict data schema.

For example, let us consider three applications which require device-wise power consumptions in minutely, hourly and daily basis, respectively. For these applications, we construct three kinds of materialized view: MinutelyView, HourlyView and DailyView. In the following, we use YYYY-MM-DD to represent year, month and day, respectively. Similarly, hh and mm denotes hour and minute, respectively.

a) **MinutelyView:** It represents power consumption per minute for every device. Hence, we construct a HBase table as shown in Table II(a).

TABLE I
RAWDATA

Row Key	Column Families							
	info:							data:
(dateTtime.type.home.device)	date	time	device	home	location	type	unit	
2013-01-18T22:00:10.Energy.cs27.pow001	2013-01-18	22:00:10	pow001	cs27	s101	Energy	W	140.3
2013-01-18T22:00:10.Energy.cs27.pow002	2013-01-18	22:00:10	pow002	cs27	s101	Energy	W	0
2013-01-19T12:30:00.Energy.cs27.pow001	2013-01-19	12:30:00	pow001	cs27	s101	Energy	W	120.7

TABLE II
MATERIALIZED VIEW(PER MINUTE,PER HOUR,PER DAY)

(a) MinutelyView

Aggregation Key	Column Families
(deviceID.YYYY-MM-DDThh:mm)	Minutely Consumption
pow001.2013-01-18T15:30	687.1
pow002.2013-01-18T15:30	0
pow001.2013-01-19T16:00	560.6

(b) HourlyView

Aggregation Key	Column Families
(deviceID.YYYY-MM-DDThh)	Hourly Consumption
pow001.2013-01-18T15	41363.7
pow002.2013-01-18T15	0
pow001.2013-01-19T16	38393.8

(c) DailyView

Aggregation Key	Column Families
(deviceID.YYYY-MM-DD)	Daily Consumption
pow001.2013-01-18	588072.6
pow002.2013-01-18	0
pow001.2013-01-19	633055.6

- Aggregation Key : [deviceID.YYYY-MM-DD T hh:mm]
- Value : power consumption of the device consumed within that minute.

b) **HourlyView:** It represents power consumption per hour for every device, Hence, we construct a HBase table as shown in Table II(b).

- Aggregation Key : [deviceID.YYYY-MM-DDThh]
- Value : power consumption of the device consumed within that hour.

c) **DailyView:** It represents power consumption per day for every device, Hence, we construct a HBase table as shown in Table II(c).

- Aggregation Key : [deviceID.YYYY-MM-DD]
- value : power consumption of the device consumed within that date.

D. Using MapReduce to Create Materialized View

We use MapReduce as an efficient method of creating a materialized view from the raw data. MapReduce is a framework of distributed processing for large-scale data set. It basically consists of a *map process* which creates the specified set of a key and a value from each input data, and a *reduce process* which aggregates the value having the same key.

For constructing the materialized view, we implement a batch program performing the following four phases.

1) **Create phase:** Create a new HBase table $view$ for the materialized view.
2) **Scan phase:** Determine the range of the raw data based on a given Data Spec, and retrieves the data.
3) **Map phase:** For each record d of the retrieved data, make an aggregation key k by using meta data of d. Then, extract a necessary value v from d. Finally, output the key-value (k, v).
4) **Reduce phase:** For key-values $(k, v_1), (k, v_2)$ $, ..., (k, v_n)$ with the same key k, calculate a value $val = v_1 \otimes v_2 \otimes ... \otimes v_n$, where $\otimes$ represents an operation specified in Data Spec. Then, output the key-value (k, val).
5) **Put phase:** Put (k, val) in $view$.

For example, let us create DailyView on 2013-01-18 shown in Section III-C. First, the scan phase retrieves all data records starting with 2013-01-18. In the map phase an aggregation key is generated in the form of “deviceID.YYYY-MM-DD”. From each record, the values of deviceID and date are obtained from `info` meta-data, and the values are concatenate to generate a key. Also, the value obtained from `data` is specified as a value corresponding to the key. In the reduce phase, we apply add operation $(+)$ for the values with the same aggregation key, in order to calculate the total sum of device-wise daily consumption. Finally, in the put phase, each pair of the aggregation key and the total sum is inserted in DailyView.

Distributed processing of the MapReduce processing can be carried out using two or more computing nodes on Hadoop. Therefore, for large-scale raw data, efficient construction of materialized view is achieved by increasing computing node.

E. API for Accessing Materialized View

We consider API by which applications access the constructed materialized view. By design of the materialized view, the API can be implemented by a simple program performing the following tasks:

1) Make an aggregation key based on parameters given by an application.
2) Pass the key and obtain the corresponding value.
3) Return the value to the application.

For example, let us consider a case that an application requests a power consumption of the device “pow005” between 10am and 11am on January 18, 2013. The application specifies (pow005, 2013-01-18, 10) as parameters to the API. Then, the API constructs a key “pow005.2013-01-18T10”, and retrieves the data from HourlyView. Finally, the API returns the value to the application.

IV. Experimental Evaluation

A. Overview

We conduct an experimental evaluation using power consumption logs obtained from a real smart home to see practical feasibility of the proposed method. The objective here is to evaluate the following two metrics.

- **E1 (Response Time):** How fast will applications be able to access the data by using a materialized view?
- **E2 (Batch Processing Time) :** How much time is taken for creating a materialized view?

To evaluate E1, we compare the response time of two cases: (1) an application accesses the raw data directly as in the previous method, (2) an application accesses a materialized view as in the proposed method. To evaluate E2, we then measure the time taken for a batch program to create a materialized view from the raw data.

B. Environment of Experiment

The raw data used in experiment is power consumption logs gathered within an actual smart home environment CS27-HNS developed by our research group. The logs are taken every 3 seconds from 32 appliances, and are stored in HBase of Scallop4SC. From the raw data, we create three kind of materialized views: MinutelyView, HourlyView, and DailyView as exaplained in Section III-C.

The MapReduce batch program has been implemented a static Java program using MapReduce. The used libraries include `hadoop-core-1.0.3.jar` and `hbase-0.94.2.jar`. The program was executed on HBase installed in a Hadoop Linux cluster (Pentium4, 3.0GHz, 2GB $\times$ 8 nodes).

C. Experiments

To evaluate the metrics E1 and E2, we have conducted two kinds of experiments.

Experiment 1: Comparison of response time

In this experiment, we suppose that an application requests device-wise power consumption based on three granularity: minitely, hourly and daily. We also suppose that the request is received through the following API.

```
double getMinutelyConsumption(device, date, time);
double getHourlyConsumption(device, date, time);
double getDailyConsumption(device, date);
```

For each of the above API, we have implemented two versions of programs: one is with the conventional method which accesses the raw data directly, while another is with the proposed method which accesses the materialized view (i.e., MinutelyView, HourlyView and DailyView). We measured the execution time by providing the random parameters to each API. We executed each API 100 times and measured the average time.

Experiment 2: Measurement of batch processing time

In this experiment, we measure the execution time of the MapReduce batch programs that create MinutelyView, HourlyView,and DailyView from the raw data. For MinutelyView, we measured the time for calculating the total power consumption of one minute. Similarly, for HourlyView and DailyView, we measured the time for calculating consumption of one hour and one day, respectively.

TABLE III
EXPERIMENT1: COMARISON OF RESPONSE TIME (SEC.)

API	getMinutely()	getHourly()	getDaily()
direct access	0.320	16.943	408.277
view access	0.008	0.002	0.001

TABLE IV
EXPERIMENT2: TIME FOR CREATING MATERIALIZE VIEW (SEC.)

	MinutelyView	HourlyView	DailyView
CPU time	130.123	220.682	370.183
# of records processed	640	38,400	921,600

D. Result

Table III shows the result of Experiment1. In the table, the row “direct access” represents the response time of the API that accesses the raw data directly (i.e., conventional method). The row “view access” represents the one with the proposed materialized view. It can be seen in the table that all cases of “view access” significantly outperform those of the conventional “direct access”. It can be also shown that “view access” takes approximately constant response time, whereas “direct access” has longer response time as the size of aggregated data increases.

Next, Table IV shows the result of Experiment2. CPU time represents the execution time taken for the MapReduce program to construct a materialized view. The next row shows the number of records of raw data that are aggregated by the MapReduce program. It can be seen in the table that the execution time grows significantly as the number of the records processed.

V. Discussion

A. Scalability of Data Access

Using the experimental result in Table III, we here discuss the scalability with respect to the data access. In the conventional API of “direct access”, 640, 38,400 and 921,600 records are respectively aggregated during runtime in getMinutely(), getHourly() and getDaily(). The response time significantly increases in the direct proportion of the number of records. Thus, the conventional method is poor in scalability to the number of records to be aggragated.

On the other hand in the proposed API of “view access”, each API achieves very quick response time of around several milliseconds, since all the application data are prepared in advance. Thus, the proposed method is excellent in scalability for the data access. It is interesting to see that getMinutely() takes the longest response time. We consider that this is related to the number of rows in MinutelyView, which is the largest among the three view. If the granularity of the aggregation is small, then the number of aggregated groups becomes large. As a result, the number of rows in the materialized view becomes large, which yields a certain overhead to look up.

B. Cost of Batch Processing

Next, we discuss a cost of creating a materialized view, using Table IV. The time required for batch processing becomes larger according to the number of records processed. The reason why the execution time is not exactly in proportion to the number of the records is in the overhead of MapReduce. Even MinutelyView which processes quite a small number of records is suffer from a significant overhead. The result of Experiment 2 shows that more than one minute is taken to process the raw data of one minute. Therefore, for such small cases, it is not a good idea to use the expensive MapReduce.

The number of records processed by MapReduce generally depends on Data Spec. given by the application. We have to guarantee a certain scalability for Data Spec. that requires large-scale data aggregation. Fortunately, MapReduce is good to *scale out* for the large-scale dataset by increasing the number of computing nodes. By controlling execution timing and data range of each batch processing, it would be possible to create the materialized view more efficiently. Discussion of efficient batch operations is left to our future work.

C. Selection Criteria of Direct Access or View Access

In order to use the proposed method, it is necessary to create a corresponding materialized view beforehand. Therefore, if an application requires a small number of data, or the application does not frequently access the aggregated data, it would be better to choose the conventional method with the raw data.

We here discuss the *total performance* by integrating the cost of the view generation and the response time for data access. Specifically, from the results of Experiments 1 and 2, we discuss which of the conventional or proposed methods is faster. This can be a selection criteria of the proposed method.

Let v denote the response time of the proposed "view access", and d denote the response time of the conventional "direct access". Also, let b denote the execution time of the MapReduce batch program, Suppose now that an application accesses the data n times. Then, the total execution time can be expressed as follows.

$$\begin{aligned}\text{previous method} &= n * d \quad \text{(time)}\\ \text{proposed method} &= b + n * v \quad \text{(time)}\end{aligned}$$

As for b, v and d, we use empirical values obtained in the experiment. For each of getDaily(), getHourly() and getMinutely(), we calculate n such that the total time of the proposed method is shorter than the previous method. As a result, we obtain the following conditions: [getDaily(): $n \geq 1$], [getHourly(): $n \geq 14$], [getMinutely(): $n \geq 418$]. For example, if the application uses getHourly() more than 14 times, it is faster to use the proposed method than the previous method. Thus, the proposed method with the materialized view is especially efficient when the application repeatedly access the same API. The same thing is true when the application-specific data is derived from a large set of the raw data. On the other hand, when an application does not use the API frequently, or a materialized view requires small data set, it would be better to use the conventional method without expensive MapReduce.

VI. Conclusion

In this paper, we proposed a method that allows various applications to efficiently use large-scale data. The method is specifically applied to a data platform, Scallop4SC, for the large-scale smart city data. In the proposed method, effective data access is achieved by using the materialized view, constructed based on data specifications of individual applications.

We also implemented the proposed method using HBase with Hadoop/MapReduce. Using power consumption data gathered from an actual smart home, we evaluated the response time of data access and the execution time of view generation. As a result, it is shown that the proposed method with the materialized view is efficient especially when the application repeatedly accesses the same data, or when the application-specific data is derived from a large set of the raw data. Our future work include the automatic generation of MapReduce batch program from Data Spec., as well as an efficient operation method of the batch.

Acknowledgment

This research was partially supported by the Japan Ministry of Education, Science, Sports, and Culture [Grant-in-Aid for Scientific Research (C) (No.24500079), Scientific Research (B) (No.23300009)], and Kansai Research Foundation for technology promotion. The authors thank to Prof. Kenji Hatano, Doshisha University, and Prof. Jun Miyazaki for fruitful discussion.

References

[1] R. G. Hollands, "Will the real smart city please stand up?" *City: analysis of urban trends, culture, theory, policy, action*, vol. 12, no. 3, pp. 303–320, 2008.

[2] A. Mahizhnan, "Smart cities: The singapore case," *Cities*, vol. 16, pp. 13–18, 1999.

[3] S. Yamamoto, S. Matumoto, and M. Nakamura, "Using cloud technologies for large-scale house data in smart city," in *In International Conference on Cloud Computing Technology and Science (CloudCom2012)*, December 2012, pp. 141–148.

[4] K. Takahashi, S. Yamamoto, A. Okushi, S. Matsumoto, and M. Nakamura, "Design and implementation of service api for large-scale house log in smart city cloud," in *In International Workshop on Cloud Computing for Internet of Things (IoTCloud2012)*, December 2012, pp. 815–820.

[5] S. M., "Key-value stores: A practical overview," in *Computer Science and Media. Ultra-Large-Sites*, vol. SS09, 2009, pp. 1–21.

[6] I. S. Mumick, "The rejuvenation of materialized views," in *International Conference on Information Systems and Management of Data (CISMOD 95)*, vol. 1006, 1995, pp. 258–264.

[7] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems," 2006.

[8] D. Jeffrey and G. Sanjay, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[9] City of Yokohama, "Yokohama smart city project," http://www.city.yokohama.lg.jp/ondan/english/.

[10] K. Kitaoka, H. Seto, S. Matsumoto, and M. Nakamura, "On identifying energy wasting behaviors from device status logs in home network system," in *IEICE*, vol. 110, no. 450, 2011, pp. 37–42.

[11] Y.-X. Lai, J. J. P. C. Rodrigues, Y.-M. Huang, Hong-GangWang, and C.-F. Lai, "An intercommunication home energy management system with appliance recognition in home network," in *Mobile Networks and Applications*, vol. 17 Issue 1, 2012, pp. 132–142.