PAPER Special Section on Cloud and Services Computing

Design and Evaluation of Materialized View as a Service for Smart City Services with Large-Scale House Log

Shintaro YAMAMOTO†a), *Nonmember*, **Shinsuke MATSUMOTO**†b), **Sachio SAIKI**†c), *and* **Masahide NAKAMURA**†d), *Members*

SUMMARY Smart city services are implemented using various data collected from houses and infrastructure within a city. As the volume and variety of the smart city data becomes huge, individual services have suffered from expensive computation effort and large processing time. In order to reduce the effort and time, this paper proposes a concept of *Materialized View as a Service (MVaaS)*. Using the MVaaS, every application can easily and dynamically construct its own *materialized view*, in which the raw data is converted and stored in a convenient format with appropriate granularity. Thus, once the view is constructed, the application can quickly access necessary data. In this paper, we design a framework of MVaaS specifically for large-scale *house log*, managed in a smart-city data platform. In the framework, each application first specifies how the raw data should be filtered, grouped and aggregated. For a given data specification, MVaaS dynamically constructs a MapReduce batch program that converts the raw data into a desired view. The batch is then executed on Hadoop, and the resultant view is stored in HBase. We present case studies using house log in a real home network system. We also conduct an experimental evaluation to compare the response time between cases with and without MVaaS.

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

1. Introduction

Smart city refers to a next-generation city planning, encouraging to improve the efficiency of the city with ICT technologies [1], [2]. The smart city provides various value-added services. A significant characteristic of the services is to use various information of houses and infrastructures within the city. The information include energy consumptions of devices, operation log of household appliances and equipment, environmental data such as temperature and humidity, traffic data from roads and railroads. These are gathered from sensors and system loggers deployed in heterogeneous systems. In general, the information gathered from the smart city is *Big Data*, comprising large-scale and heterogeneous data items. Our long term goal is to provide a universal platform, on which applications and services can extensively use the smart city data for various purposes.

In our previous research [3], [4], we proposed a logging platform, called *Scallop4SC (Scalable Logging Platform for Smart City)*. Exploiting cloud technologies Hadoop and HBase, Scallop4SC processes and stores the large amount of *house logs* (e.g., power consumption logs) from *smart homes* within a smart city. Through API, Scallop4SC provides the smart city data for various applications, such as energy visualization, detection of wasteful use, and peak shaving.

In general, required data vary from one application to another. Hence, each application has to transform the raw data in Scallop4SC into its appropriate format and granularity. However, due to the size and variety of the raw data, the transformation poses expensive computation and long processing time for the application.

To cope with the problem, we introduced the concept of *materialized view* in Scallop4SC [5]. The materialized view is a database technology that *caches* results of queries in an actual table to improve the response time [6]. Our experiment showed that the materialized view dramatically improved the response time. However, each materialized view was statically created by a proprietary MapReduce program. Thus, application developers had to be familiar with complex knowledge of Hadoop/MapReduce and HBase, for implementing their own materialized views. It was also difficult to reuse the existing views for other applications.

This motivates us to encapsulate complex creation and management of the materialized views in an abstract cloud service. This is what we call *Materialized View as a Service (MVaaS)* in this paper. For a given *recipe* of required data (called *data specification*), MVaaS dynamically creates a materialized view for an individual application. Once the view is constructed, the application can quickly access necessary data through API of the view.

In this paper, we design a framework of MVaaS specifically for Scallop4SC. In the framework, an application developer of Scallop4SC creates a data specification prescribing how the raw data should be filtered, grouped and aggregated. Based on the data specification, MVaaS dynamically generates a MapReduce batch program that converts the raw data into a desired view. The batch is then executed on Hadoop, and the resultant view is stored in HBase. Finally the materialized view is accessed via MVaaS API. Thus, the developer can easily create and use own materialized view.

We discuss case studies of energy-related services of a smart city using MVaaS. We also conduct a performance evaluation using large-scale power consumption data, recorded in a real smart home environment for a year.

Manuscript received October 31, 2013.
Manuscript revised February 7, 2014.

†The authors are with the Kobe Graduate School of System Informatics, Kobe University, Kobe-shi, 657–0013 Japan.
a) E-mail: shintaro@ws.cs.kobe-u.ac.jp
b) E-mail: shinsuke@cs.kobe-u.ac.jp
c) E-mail: sachio@carp.kobe-u.ac.jp
d) E-mail: masa-n@cs.kobe-u.ac.jp

DOI: 10.1587/transinf.E97.D.1709

It is shown that the proposed MVaaS is especially effective in cases where the applications repeatedly access the same data, or the view is derived from a large set of raw data.

The digest version of this paper is published as a poster/demo paper in International Conference on Cloud Computing Technology and Science (CloudCom2013). Changes were made on this version most significantly in the addition of case study and discussion.

2. Preliminaries

2.1 Smart City and Services

The principle of the smart city is to gather data of the city first, and then to provide appropriate services based on the data. Thus, a variety of data are collected from sensors, devices, cars and people across the city. A 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 [7], traffic optimization [8], local economic trend analysis [9], entertainment [10], community-based health care [11], disaster control [12] and agricultural support [13].

The size and variety of gathered data become huge in general. Velocity (i.e., freshness) of the data is also important to reflect real-time or latest situations and contexts. Thus, the data for the smart city services is truly *big data*.

Due to the limitation of storage, the conventional applications were storing only necessary data with optimized granularity. Therefore, the gathered data was application-specific, and could not be shared with other applications.

The limitation of the storage is 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.

2.2 Scallop4SC (Scalable Logging Platform for Smart City)

We have been developing a data platform, called *Scallop4SC*, for smart city services [3], [4]. Scallop4SC is specifically designed to manage data from *houses*. The data from houses are essential for various smart city services, since a house is a primary construct of a city. In near future, technologies of smart homes and smart devices will enable to gather various types of house data.

Scallop4SC basically manages two types of house data: *house log* and *house configuration*. The house log is history of values of any dynamic data measured within a smart home. Typical house log includes power consumption, status of an appliance and room temperature. The house configuration is static meta-data explaining a house. Examples include house address, device ID, floor plan and inhabitant names.

Figure 1 shows the architecture of Scallop4SC. For each house in a smart city, a logger measures various data and records the data as house log. The house log is periodically sent to Scallop4SC via a network. Due to the number of houses and the variety of data, the house log generally forms big data. Thus, Scallop4SC stores the house log using *HBase* NoSQL-DB, deployed on top of *Hadoop* distributed processing. On the other hand, the house configuration is static but structural data. Hence, it is stored in *MySQL* RDB to allow complex queries over houses, devices and people.

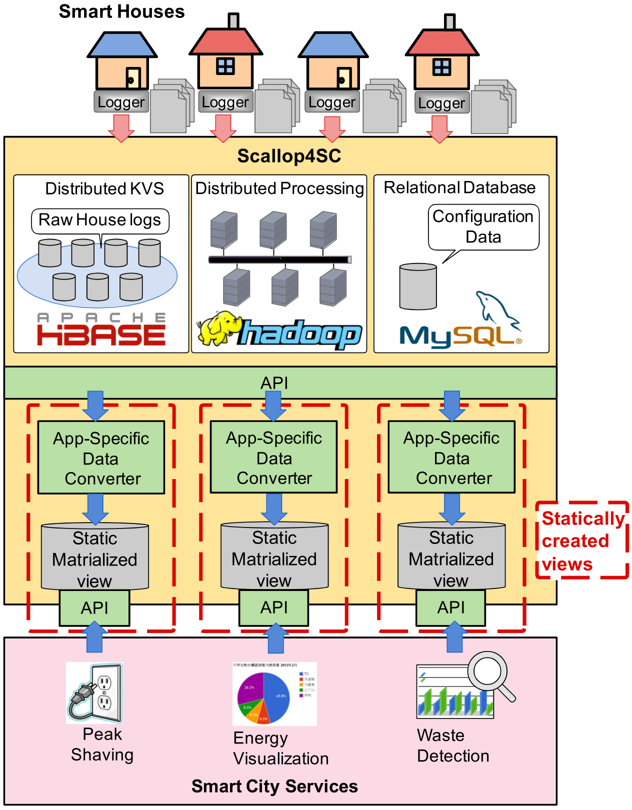


Fig. 1 Scallop4SC with static MVs.

Scallop4SC API (shown in the middle in Fig. 1) provides a basic access method to the stored data. Since Scallop4SC is an application-neutral platform, the API just allows basic queries (see [4]) to retrieve the *raw data*. Application-specific data interpretation and conversion are left for individual applications.

2.3 Introducing Materialized View in Scallop4SC

In general, individual applications use the smart city data in different ways. If an application-specific data is derived from much of raw data, the application suffers from expensive data processing and long processing time. This is because the application-specific data conversion is left to each application. 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 cope with this, we introduced *materialized view* in Scallop4SC, as shown in the lower part of Fig. 1 [5]. The application-specific data can be considered as a *view*, which looks up the raw data based on a certain query. The materialized view is constructed as a table, which *caches* results of the query in advance.

Note, however, that the raw data in Scallop4SC is very large, and that we cannot use SQL for HBase to construct the view. Therefore, in [5] we developed a Hadoop/MapReduce program for each application, which efficiently converts the raw data into application-specific data. The converted data is stored in an HBase table, which is used as a materialized view by the application. Our experiment showed that the use of materialized view significantly reduced computation cost of applications and improved the response time.

A major limitation of the previous research is that the MapReduce program was statically developed and deployed. This means that each application developer has to implement a proprietary data converter by himself. The implementation requires development effort as well as extensive knowledge of HBase and Hadoop/MapReduce. It is also difficult to reuse the existing materialized views for other applications. These are obstacle for rapid creation of new applications.

3. Materialized View as a Service for Large-Scale House Log

3.1 Materialized View as a Service (MVaaS)

To overcome the limitation, we propose a new concept of *Materialized View as a Service (MVaaS)*. MVaaS encapsulates the complex creation and management of the materialized views within an abstract cloud service. Although MVaaS can be a general concept for any data platform with big data, this paper concentrates the design and implementation of MVaaS for *house log* in Scallop4SC.

Figure 2 shows the new architecture of Scallop4SC with MVaaS. A developer of a smart city application first gives an order in terms of *data specification*, describing what data should be presented in which representation. MVaaS of Scallop4SC then dynamically creates a materialized view appropriate for the application, from large-scale house log of Scallop4SC. Thus, the application developer can easily create and use own materialized view without knowledge of underlying cloud technologies.

In the following subsections, we explain how MVaaS converts the raw data of house log into application-specific materialized view.

3.2 House Log Stored in Scallop4SC

First of all, we briefly review the data schema of the house log in Scallop4SC (originally proposed in [3]).

Table 1 shows an example of house logs obtained in our laboratory. To achieve both *scalability* for data size and *flexibility* for variety of data type, Scallop4SC stores the house log in the HBase key value store. Every house log is stored simply as a pair of key (**Row Key**) and value (**Data**). To store a variety of data, the data column does not have rigorous schema. Instead, each data is explained by a meta-data (**Info**), comprising *standard properties* for house log.

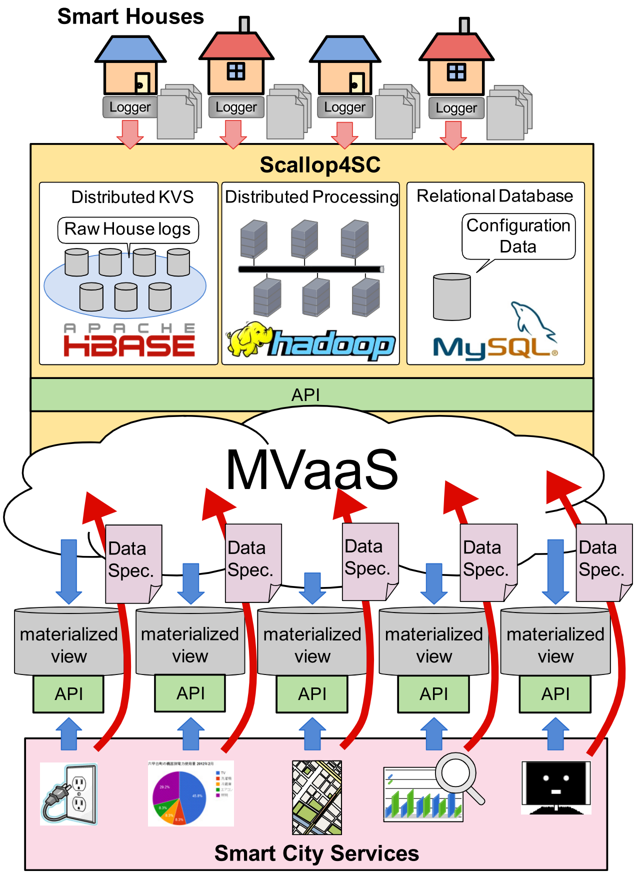


Fig. 2 Extended Scallop4SC.

The properties include *date* and *time* (when the log is recorded), *device* (from what the log is acquired), *house* (where in a smart city the log is obtained), *unit* (what unit should be used), *location* (where in the house the log is obtained) and *type* (for what the log is). Details of device, house and location are defined in an external database of house configuration in MySQL (see Fig. 2). A row key is constructed as a concatenation of date, time, type, home and device. An application can *get* a single data (i.e., row) by specifying a row key. An application can also *scan* the table to retrieve multiple rows by *prefix match* over row keys. Note that complex queries with SQL cannot be used to search data, since HBase is a NoSQL database.

For example, the first row in Table 1 shows that the log is taken at 12:34:56 on May 28, 2013, and that the house ID is cs27 and log type is Energy, and that the deviceID is tv01. The value of power consumption is 600 W. Similarly, the second row shows a status of tv01 where power is off.

We assume that these logs are used as *raw data* by various smart city services and applications.

3.3 Idea of Converting Raw Data into Materialized View

The primary task of MVaaS is to convert the raw data in Ta-

Table 1 Raw data: House log of Scallop4SC.

Row Key	Column Families							
	Data:	Info:						
(dateTtime.type.home.device)		date:	time:	device:	house:	unit:	location:	type:
2013-05-28T12:34:56.Energy.cs27.tv01	600	2013-05-28	12:34:56	tv01	cs27	W	living room	Energy
2013-05-28T12:34:56.Device.cs27.tv01	[power:off]	2013-05-28	12:34:56	tv01	cs27	status	living room	Device
2013-05-28T12:34:56.Environment.cs27.temp3	24.0	2013-05-28	12:34:56	temp3	cs27	Celsius	kitchen	Environment
2013-05-28T12:34:56.Environment.cs27.pcount3	3	2013-05-28	12:34:56	pcount3	cs27	people	living room	Environment
2013-05-28T12:35:00.Device.cs27.tv01	on()	2013-05-28	12:35:00	tv01	cs27	operation	living room	Device
:	:	:	:	:	:	:	:	:

ble 1 into an application-specific view, according to a given *data specification* (defined later).

For example, let us consider a statistical application that uses *daily total energy of each device*. If the house logs were stored in RDB, we would specify the following SQL:

```
CREATE VIEW dailyConsumptionPerDevice AS
SELECT date,device,SUM(Data) FROM houselog
WHERE type=Energy AND unit=W
GROUP BY date,device;
```

As we can see in the above SQL, a typical view creation consists of three steps:

- **Step 1 (Filter):** Filter necessary data records out of all raw data (corresponding to WHERE clause).
- **Step 2 (Group):** For the data records, group multiple data records based on one or multiple properties (corresponding to GROUP BY clause).
- **Step 3 (Aggregate):** For each group, aggregate data values using an aggregate function (e.g., SUM(Data)).

However, our problem is not that simple, because the table of house log is huge and stored in a NoSQL database.

Our key idea is to implement a framework that executes the above three steps with a MapReduce program. Figure 3 shows a scenario of creating a materialized view of `dailyConsumptionPerDevice` from raw data of house log. First, we filter the raw data to extract relevant rows with Energy type and W unit (Step 1). The rows are then grouped according to the same date and the same device (Step 2). For each group, the energy values are aggregated by SUM function to obtain the daily energy consumption of each device (Step 3). As will be explained later, in our MapReduce program, a map process, a shuffle process and a reduce process will execute Step 1, 2 and 3, respectively. Finally, the resultant view is stored in a new HBase table, which makes the view *materialized*.

3.4 Describing Data Specification

To generalize the scenario in the previous section for other applications, we here define the *data specification*. Intuitively, the data specification is considered as an order from an application, specifying how the resultant view should be generated. An SQL equivalent to a template of the proposed data specification is expressed as follows:

```
CREATE VIEW $view_name AS
SELECT $prop1, $prop2,...,$prop_n,
       $aggregate_function($expression)
FROM houselog
WHERE $filtering_condition
GROUP BY $prop1, $prop2,...,$prop_n;
```

In the above, `$` represents a placeholder, which can be replaced by a concrete name or expression in accordance with individual context.

In the proposed MVaaS, the following three items should be specified in the data specification, to create an application-specific materialized view: (1) a filtering condition, (2) grouping properties, (3) an aggregation function.

Filtering Condition

A filtering condition is a condition that extracts relevant rows from all house logs. Equivalently, it corresponds to `$filtering_condition` in the above SQL. In our framework, an (atomic) filtering condition $filter$ is defined by $filter = [prop\ cmp\ val]$, where $prop$ represents a property name (i.e., column qualifier) of the house log table (see Table 1), cmp represents a comparison operator (`==, !=, >=, <=, > or <`), and val represents a value over the property. If $filter_1$ and $filter_2$ are both filtering conditions, then [$filter_1$ && $filter_2$] (logical AND), [$filter_1$ `||` $filter_2$] (logical OR), `!` [$filter_i$] (logical NOT) are also filtering conditions.

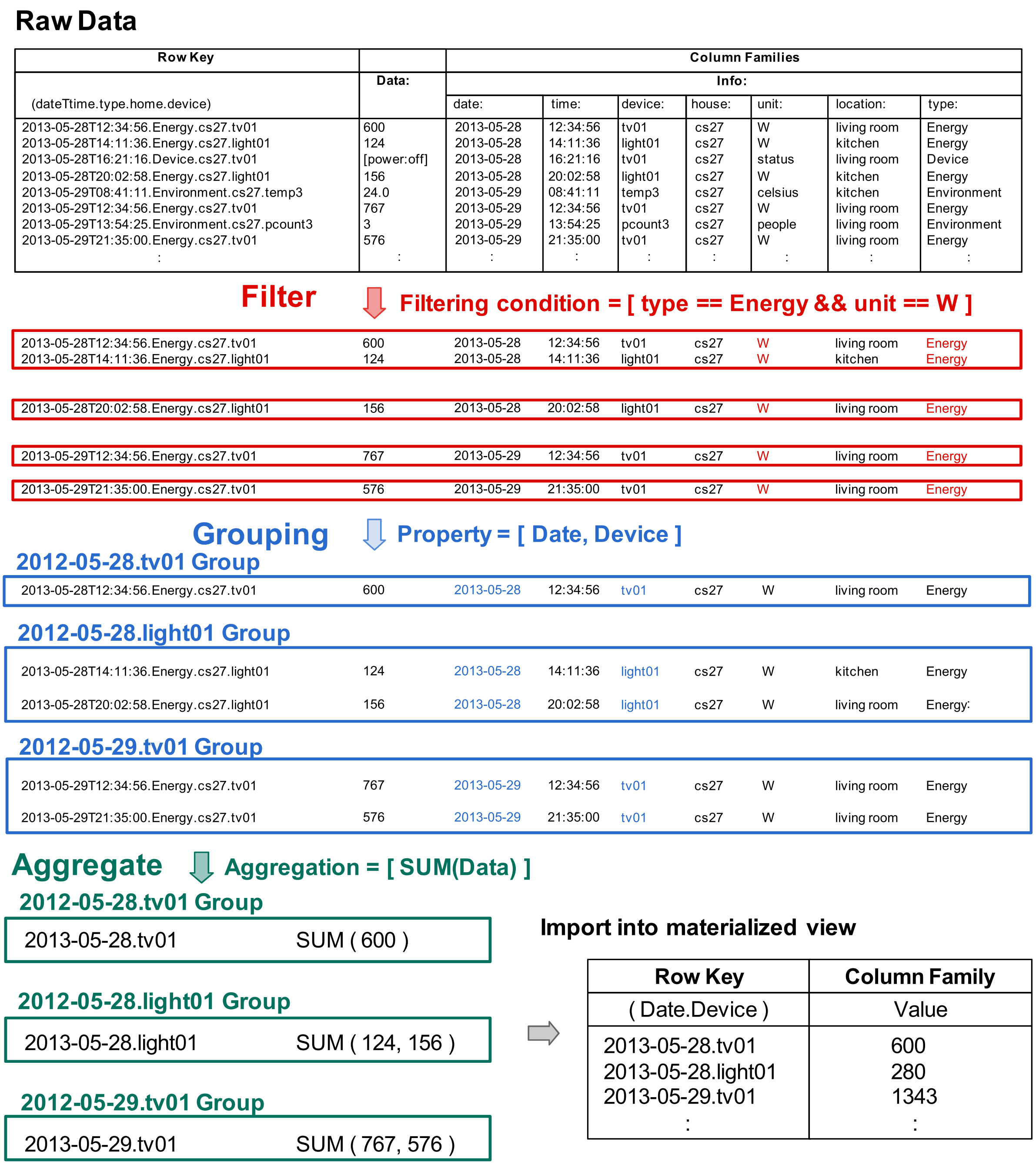
For example, a condition that "power consumption of tv01 is more than 600W" can be expressed by `[device == tv01 && type == Energy && unit == W && Data >= 600]`. A condition that "3 people is in living room" can be expressed by `[location == living_room && type == Environment && unit == people && Data == 3]`.

Grouping Properties

Grouping properties define groups of the filtered data. If multiple rows take the same values with respect to the properties, these rows are grouped into the same group. The grouping properties are equivalent to `$prop1, $prop2,...,$prop_n` in the above SQL.

In the data specification, grouping properties $group$ are defined by $group = [p_1, p_2, \ldots, p_n]$, where each p_i is one of the followings: `Data, device, house, unit, location, type, TIME, CUSTOM`. Properties from `Data` to `type` are those in Table 1. `TIME` specifies temporal granularity of the grouping, which is defined by one from `year, month, date, hour, minute, second`. `CUSTOM` represents a user-defined grouping criteria, such as `am` gathering all logs taken in the morning.

For example, grouping properties that aggregates house



Raw Data

Row Key		Column Families						
	Data:	Info:						
(dateTtime.type.home.device)		date:	time:	device:	house:	unit:	location:	type:
2013-05-28T12:34:56.Energy.cs27.tv01	600	2013-05-28	12:34:56	tv01	cs27	W	living room	Energy
2013-05-28T14:11:36.Energy.cs27.light01	124	2013-05-28	14:11:36	light01	cs27	W	kitchen	Energy
2013-05-28T16:21:16.Device.cs27.tv01	[power:off]	2013-05-28	16:21:16	tv01	cs27	status	living room	Device
2013-05-28T20:02:58.Energy.cs27.light01	156	2013-05-28	20:02:58	light01	cs27	W	kitchen	Energy
2013-05-29T08:41:11.Environment.cs27.temp3	24.0	2013-05-29	08:41:11	temp3	cs27	celsius	kitchen	Environment
2013-05-29T12:34:56.Energy.cs27.tv01	767	2013-05-29	12:34:56	tv01	cs27	W	living room	Energy
2013-05-29T13:54:25.Environment.cs27.pcount3	3	2013-05-29	13:54:25	pcount3	cs27	people	living room	Environment
2013-05-29T21:35:00.Energy.cs27.tv01	576	2013-05-29	21:35:00	tv01	cs27	W	living room	Energy
:	:	:	:	:	:	:	:	:

Filter Filtering condition = [type == Energy && unit == W]

2013-05-28T12:34:56.Energy.cs27.tv01	600	2013-05-28	12:34:56	tv01	cs27	W	living room	Energy
2013-05-28T14:11:36.Energy.cs27.light01	124	2013-05-28	14:11:36	light01	cs27	W	kitchen	Energy
2013-05-28T20:02:58.Energy.cs27.light01	156	2013-05-28	20:02:58	light01	cs27	W	living room	Energy
2013-05-29T12:34:56.Energy.cs27.tv01	767	2013-05-29	12:34:56	tv01	cs27	W	living room	Energy
2013-05-29T21:35:00.Energy.cs27.tv01	576	2013-05-29	21:35:00	tv01	cs27	W	living room	Energy

Grouping Property = [Date, Device]

2012-05-28.tv01 Group

2013-05-28T12:34:56.Energy.cs27.tv01	600	2013-05-28	12:34:56	tv01	cs27	W	living room	Energy

2012-05-28.light01 Group

2013-05-28T14:11:36.Energy.cs27.light01	124	2013-05-28	14:11:36	light01	cs27	W	kitchen	Energy
2013-05-28T20:02:58.Energy.cs27.light01	156	2013-05-28	20:02:58	light01	cs27	W	living room	Energy:

2012-05-29.tv01 Group

2013-05-29T12:34:56.Energy.cs27.tv01	767	2013-05-29	12:34:56	tv01	cs27	W	living room	Energy
2013-05-29T21:35:00.Energy.cs27.tv01	576	2013-05-29	21:35:00	tv01	cs27	W	living room	Energy

Aggregate Aggregation = [SUM(Data)]

2012-05-28.tv01 Group

2013-05-28.tv01 SUM (600)

2012-05-28.light01 Group

2013-05-28.light01 SUM (124, 156)

2012-05-29.tv01 Group

2013-05-29.tv01 SUM (767, 576)

Import into materialized view

Row Key	Column Family
(Date.Device)	Value
2013-05-28.tv01 2013-05-28.light01 2013-05-29.tv01 :	600 280 1343 :

Fig. 3 Flowchart of converting raw data into materialized view.

logs for each device every day can be expressed by [date, device]. Those that gathers house logs for each house per year can be expressed by [house, year].

In the proposed method, concatenation of values of the grouping properties is used as every row key of the materialized view. Therefore, it is important to determine the order of p_i's, by considering priority among the properties. In the above example, [year, house] would be better if the application performs year-wise search more frequently than house-wise search.

Aggregate Function

An aggregation function defines how to aggregate the grouped house logs, which corresponds to $aggregate _function($expression) in the previous SQL.

In our framework, an aggregation function is defined by $aggregation = aggr(expression)$, where *aggr* is one of the following functions: SUM (total sum), MAX (maximum value), MIN (minimum value), AVG (average), COUNT (count of items), CONCAT (concatenation of items), ID (identity), CUSTOM (user-defined function). Also, *expression* is any expression defined with properties of the houselog table (see Table 1).

For example, an aggregation function that calculates total sum of data within the grouped data rows is expressed by `SUM(Data)`. Another function that concatenates all dates of the group is expressed by `CONCAT(date)`.

3.5 Generating MapReduce Converter

Once a data specification is given, MVaaS dynamically generates a MapReduce program that converts the raw data into the target view. The MapReduce program is generated so that a pair of *map* and *reduce* processes perform the following tasks.

- **Filter:** In the map process, only data rows satisfying the filtering condition ($filter$) are passed to the reduce process.
- **Group:** In the map phase, each filtered row is converted into such a pair of *key* and *value* that:
  - **key:** The key is constructed as a concatenation of values of the grouping properties. This makes all the rows in the same group have the same key.
  - **value:** The value is generated by evaluating *expression* in the aggregation function.
- **Aggregate:** In the reduce process, the values in the same key (i.e., the same group) are aggregated by the aggregation function (*aggr*).

A pseudo code of the MapRedeuce program is as follows:

```
class Mapper
  method map(rowkey, column)
    if(filter(column)==true)
      for (p in group)
        key = concat(key, column.p.getValue())
      value = eval(expression(column))
      EMIT(key, value)

class Reducer
  method reduce(key, [v1, v2, \ldots , vn])
    result = aggr(v1, v2,\ldots ,vn)
    EMIT(key, result)
```

3.5.1 Mapper

For each row of house logs, Mapper class takes a row key and column families. Mapper first check if data in the column satisfies filtering condition. If it does, it generates a key by concatenating the value of each grouping property. The value is obtained by evaluating the expression according to values of column. The pair of key and value is emitted to Reducer.

3.5.2 Reducer

After the map process, data values $v_1, v_2, \ldots, v_n$ with the same key are gathered and passed to Reducer. Reducer just aggregate values $v_1, v_2, \ldots, v_n$ using a designated aggregated function. The key and the result is emitted as a row of the materialized view.

The MapReduce program is then compiled and executed on Hadoop in Scallop4SC. The resultant key-values are stored in a new HBase table as a materialized view. The creation of the view generally takes time, depending on the data specification and the size of raw data chosen. However, once it is created, the application can access the data quickly.

3.6 Alternative Design by Pig

Although MVaaS itself is general concept for any type of data processing, there are several the design choices for implementation. In this paper, we considered data-specification as high-level language of native MapReduce program. As one of the other choices, Pig Latin [14] is useful for describing data processing on Hadoop environment. Pig is a high-level language for describing programs that analyze big data stored in Hadoop and HBase.

3.7 API of Created Materialized View

MVaaS also creates API for the created materialized view. Through API, applications can access the data of the view without knowing physical data model of HBase. Two methods are provided.

- `get(view_name, row_key)` returns a value corresponding a given key for a specified view.
- `scan(view_name, keyword)` returns all values matching a keyword (by prefix search) for a view.

Note that the response time of the API is quite short, once a materialized view is constructed. Thus, the application can quickly access the desired data. Note also that API provides an access to the existing views, which allows applications to share and reuse the materialized views.

4. Case Study

We demonstrate the effectiveness of MVaaS through two example services in this section.

4.1 Power Consumption Visualization Service

This service visualizes the use of energy at home from various viewpoints (e.g. houses, towns, devices, current power consumption, passage of past power consumption, etc.). It is intended to raise user's awareness of energy saving by intuitively showing the current and past usage of energy.

In this scenario, we assume that every smart home in a city sends the power consumption data of every device to Scallop4SC periodically every 3 seconds, and that the visualization service wants to see the energy consumption within a house `cs27`, for every device per hour per day.

As the size of power consumption logs is large, it is unrealistic for the service to use the raw data directly. Hence,

Table 2 Materialized view of consumption visualization service.

Row Key	Column Families
(hour.device)	**Data:**
2013-05-28T10.tv01	784415
2013-05-28T10.light01	1446
2013-05-28T10.light02	21321
2013-05-29T10.aircon01	54889
2013-05-28T11.tv01	51247
2013-05-28T11.light01	7742
2013-05-28T11.light02	3288
2013-05-28T11.aircon01	65774
:	:

the service developer submits the following data specification to MVaaS.

```
Filtering condition =
  [type==Energy && unit==W && house==cs27]
Property = [Hour, Device]
Aggregation = [SUM(Data)]
```

Based on the data specification, MVaaS constructs a materialized view as shown in Table 2. In the table, a row key is constructed as a concatenation of hour and device, value is a total sum of power consumption in the hour.

For example, the first row in Table 2 shows that the total sum of power consumption of tv01 between 10 and 11 AM on May 28, 2013 is 784415 W. The Application can visualize the use of energy using this table.

4.2 Wasteful Energy Detection Service

This service automatically detects and notifies wasteful use of electricity, using power consumption data and sensors data in a smart home.

In this scenario, we assume that every smart home periodically sends status of every devices and environment information every minute. The service wants to detect the wasteful electricity using two materialized views. The first view enumerates the time when each appliance is turned on for every location for every device. The second view enumerates the time for each location, where no user exists there. An intersection of the two views yields the time where any appliance is turned on while no one in the room. This is the wasteful use of energy.

To create the views, the developer submits the following data specifications to MVaaS:

```
Filtering condition =
  [type == Device && unit == status &&
  house == cs27 && Data == [power:on]]
Property = [Date, Location, Device]
Aggregation = [CONCAT(time)]
```

For the second view,

```
Filtering condition =
  [type == Environment && unit == people &&
  house == cs27 && Data == 0]
Property = [Date, Location]
Aggregation = [CONCAT(time)]
```

Table 3 Materialized view of diagnosis and improvement of lifestyle support service.

(a) Status of devices

Row Key	Column Families
(date.location.device)	**Data:**
2013-05-28.living room.tv01	17:12:44, 17:13:43, 17:14:57,…
2013-05-28.bedroom.light01	23:34:56, 23:35:56, 23:36:57,…
2013-05-29.living room.tv01	17:22:16, 17:23:14, 17:24:17,…
2013-05-29.bedroom.light01	20:46:23, 20:47:24, 20:48:50,…
2013-05-30.living room.tv01	18:56:21, 18:57:21, 18:58:24,…
2013-05-30.bedroom.light01	21:57:21, 21:58:21, 21:59:20,…
:	:

(b) Period of time during absence from the room

Row Key	Column Families
(date.location)	**Data:**
2013-05-28.living room	10:12:44, 10:13:43, 10:14:57,…
2013-05-28.bedroom	7:34:56, 7:35:56, 7:36:57,…
2013-05-29.living room	11:22:16, 11:23:14, 11:24:17,…
2013-05-29.bedroom	8:36:23, 8:37:24, 8:38:50,…
2013-05-30.living room	9:56:21, 9:57:21, 9:58:24,…
2013-05-30.bedroom	7:57:23, 7:58:21, 7:59:20,…
:	:

Based on these data specifications, MVaaS constructs a materialized views as shown in Table 3 (a) and Table 3 (b).

5. Experimental Evaluation

5.1 Overview of Experiment

We conduct an experiment to evaluate performance of MVaaS. The experiment is performed on our Scallop4SC prototype, which is a Hadoop cluster comprising 9 nodes (Intel(R) Corei7-3770, 32GB, CentOS-x64). Libraries used for Hadoop/MapReduce are `hadoop-core-1.0.4.jar` and `hbase-0.94.7.jar`. The raw data used in experiment is *power consumption logs* gathered in our smart home environment CS27-HNS [15]. The consumption logs have been taken every 3 seconds from 32 devices, for over two years. For one day, the logs comprises 921,600 data rows (= 20 items x 60 minutes x 24 hours x 32 devices).

From the raw data, we create three kinds of materialized views with MVaaS: *DailyView*, *HourlyView* and *MinutelyView*. They store the total power consumption of each device for every day, hour and minute, respectively.

The purpose of experimental evaluation is to answer the following two questions.

- Q1: How much time is taken for creating a materialized view?
- Q2: How fast will applications be able to access the processed data?

To answer Q1, we measure and compare the time taken for creating a materialized view from the raw data by using proposed method and by using Pig Latin program. The proposed method means that a materialized view is created by MapReduce program converted from a data-specification. The Pig method means that we write a Pig script which has the same function as the proposed method.

To answer Q2, we measure the response time that a client application obtains single data from a materialized

Table 4 Result of experiments.

Materialized View	Monthly	Daily	Hourly	Minutely
# of rows aggregated	28,569,600	921,600	38,400	640
Time for MV creation (sec.)				
Proposed Method	3715.8	412.7	52.2	34.7
Pig Latin	4610.1	510.9	79.3	59.5
Response time (sec.)				
with MVaaS	0.002	0.002	0.002	0.002
without MVaaS	12,956.381	328.022	13.682	0.260

view through MVaaS API. We compare the response time with and without MVaaS. In the case of without MVaaS, the client application obtains raw data and processes the raw data into own format.

5.2 Result of Experiment

Table 4 shows the result. We can see that MVaaS took about 7 minutes to create a DailyView for converting 921,600 rows by MapReduce. Once the view is created, the application can access any data in the view very quickly (2 milliseconds). On the other hand, the conventional method without MVaaS took 5 minutes for *each* calculation of the daily consumption. Thus, we can see that the creation of a view is efficient especially when the required raw data is large and the application frequently accesses the aggregated data.

On the other hand, when the size of raw data to be aggregated is small, the overhead of MapReduce becomes dominant in the total execution time. As seen in the result of MinutelyView, the response time without MVaaS is just 0.2 second, which might be acceptable for some applications. In such cases, using the conventional method is one possible choice to avoid overhead of creating a materialized view.

Compared with Pig, the ratio of execution time of MVaaS to Pig is 50% to 80%. In other words, MVaaS is faster than Pig in any dataset size. The ratio comes close to 80% when the number of aggregated rows becomes larger. This result has the same tendency with the Pig experience reported by Gates et al [16]. They have reported in April 2009 that Pig took 1.5 times as much time as with native MapReduce program.

In summary, the answer to Q1 is about 7 minutes for 1 million rows and the processing time is in direct proportion to the number of rows. The answer to Q2 is 2 milliseconds and it is independent on the number of rows.

6. Discussion

6.1 Related Work

Hive [17] has powerful SQL-like feature for summarizing and aggregating big data stored in Hadoop. It seems to be useful for dynamic creation of materialized views in MVaaS. However, Hive basically requires to prepare tables with rigorous data schema (like those of RDB) on top of Hadoop. Hence, it is not directly applied to our design of Scallop4SC, where various (even unknown) types of house logs are stored in the same big table. Also, since the aggregated data is stored in Hadoop, random data access from applications would yield long response time.

Pig [14] provides a high-level language (Pig Latin) for describing programs that analyze big data stored in Hadoop and HBase. It might be adopted as the language for our data specification. However, since Pig Latin is designed for general-purpose analysis, it might be difficult for application developers to learn and use it. Also, the execution of a Pig program is generally slower than that of a native MapReduce program. Thus, we proposed a custom method to generate a MapReduce program from a data specification designated for MVaaS of Scallop4SC.

6.2 Validity of Design Choice

Although, we considered data-specification as high-level language of native MapReduce program, other general-purpose framework can be applied the implementation of our approach.

The general-purpose framework has high flexibility and can be applied to any type of data processing. Of course, all example scripts written in this paper can be also achieved by Pig. Specifically, Pig can be used with small learning cost because it supports simple and abstract notations. However, a user of these general-purpose framework has to create program from a wide variety of script commands. In contrast, our approach specializes the existing framework to the smart city. This decreases flexibility of data processing but increases the ease of use. In our method, the sequence of data processing is fixed within the three steps; filter, group and aggregate. Thus, the end user can only focus on how to process raw data within the constraints, determined by our framework. These constraints provide not only simplification of data processing, but also simple and easy user interactions.

6.3 Issues for Practical Operation of MVaaS

In order to provide MVaaS efficiently for practical settings, we have to address some pragmatic issues, which are beyond this paper. One issue is *task scheduling* of MapReduce programs. A MapReduce task generally takes some time. If many applications request MVaaS to create views, MVaaS should consider an efficient schedule of the tasks, so that the waiting time meets a desired service level of each application. To handle many requests at a time, we should also consider *dynamic scaling* of MVaaS by adding more computing nodes.

Another issue is to find *good design* of materialized view. For every application, there are many design choices of materialized views. It would be difficult for the developer to find *optimal* view for the application. Developing best practices and design patterns for MVaaS is promising. We leave these practical issues for our future work.

7. Conclusion

In this paper, we proposed a materialized as a service (MVaaS) for large-scale house log in a smart city data platform Scallop4SC. Using MVaaS, a developer of a smart city application can easily construct an application-specific view, by which the application can access the necessary data efficiently. For a given data specification, MVaaS dynamically generates a MapReduce program that converts the raw data into the view. The program is then executed on Hadoop of Scallop4SC, and published as a service with API. We have evaluated the proposed method through case studies and performance evaluation. Our future works include the pragmatic issues discussed in Sect. 6.3 and development of practical applications using MVaaS.

Acknowledgments

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)].

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," International Conference on Cloud Computing Technology and Science (CloudCom2012), pp.141–148, Dec. 2012.

[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," International Workshop on Cloud Computing for Internet of Things (IoTCloud2012), pp.815–820, Dec. 2012.

[5] Y. Ise, S. Yamamoto, S. Matsumoto, S. Saiki, and M. Nakamura, International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2013), 2012 (to appear).

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

[7] S. Massoud Amin and B. Wollenberg, "Toward a smart grid: Power delivery for the 21st century," IEEE Power and Energy Magazine, vol.3, no.5, pp.34–41, 2005.

[8] E. Brockfeld, R. Barlovic, A. Schadschneider, and M. Schreckenberg, "Optimizing traffic lights in a cellular automaton model for city traffic," Phys. Rev. E, vol.64, 056132, 2001.

[9] IBM, "Apply new analytic tools to reveal new opportunities." http://wwiw.bm.com/smarterplanet/nz/en/business_analytics/article/it_business_intelligence.html.

[10] I. Celino, S. Contessa, M. Corubolo, D. Dell'Aglio, E.D. Valle, S. Fumeo, and T. Krüger, "Urbanmatch - linking and improving smart cities data," Linked Data on the Web (LDOW2012), 2012.

[11] IBM, "eHealth and collaboration - collaborative care and wellness." http://www.ibm.com/smarterplanet/nz/en/healthcare_solutions/nextsteps/solution/X056151Y25842W14.html.

[12] C. Hu and N. Chen, "Geospatial sensor web for smart disaster emergency processing," International Conference on GeoInformatics (Geoinformatics2011), pp.1–5, 2011.

[13] Y. Cho, J. Moon, and H. Yoe, "A context-aware service model based on workflows for u-agriculture," International Conference on Computational Science and Its Applications (ICCSA2010), pp.258–268, 2010.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," Proc. 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pp.1099–1110, New York, NY, USA, 2008.

[15] M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K. Matsumoto, "Constructing home network systems and integrated services using legacy home appliances and web services," Int. J. Web Services Research, vol.5, no.1, pp.82–98, 2008.

[16] A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: The pig experience," Proc. VLDB Endow., vol.2, no.2, pp.1414–1425, Aug. 2009.

[17] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "HIVE: A warehousing solution over a map-reduce framework," Proc. VLDB Endow., vol.2, no.2, pp.1626–1629, Aug. 2009.

Shitnaro Yamamoto recieved the B.E. degree in computer science and engineering from Kobe University, Japan, in 2012. He is currently a master course student at the Graduate School of System Informatics, Kobe University, Japan. His interests are cloud computing and mobile communications.

Shinsuke Matsumoto received M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology in 2008 and 2010, respectively. He is currently an assistant professor in the Graduate School of System Informatics at Kobe University. His research interests include software engineering, mining software repository and cloud computing.

Sachio Saiki received M.E. and Dr.Eng. degrees in Engineering course with information systems engineering minor from Kochi University of Technology, Japan, in 2006 and 2010, respectively. His research interests include adaptive signal processing and software engineering education, but also extend to practical applications related to these topics. He is currently an Assistant Professor at Kobe University, and is a Regular Member of IEICE and IEEE.

Masahide Nakamura received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at University of Ottawa, Canada. He joined Cybermedia Center at Osaka University from 2000 to 2002. From 2002 to 2007, he worked for the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. He is currently an associate professor in the Graduate School of System Informatics at Kobe University. His research interests include the service/cloud computing, smart home, smart city, and life log. He is a member of the IEEE, IEICE and IPSJ.