Design and Evaluation of Lifelog Mashup Platform with NoSQL Database

Kohei TAKAHASHI, 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
{koupe@ws.cs, shinsuke@cs, sachio@carp, masa-n@cs} .kobe-u.ac.jp

ABSTRACT

To support mashup of heterogeneous lifelog services, we have previously implemented the *lifelog common data model (LLCDM)*. The previous LLCDM was implemented with MySQL, where various types of application-specific data (e.g., numeric values, text, JSON or XML) were all stored in a `<content>` column in a schemaless text format. Any query with application-specific data had to be managed by individual applications.It had also a scalability issue as the data size grew.

To cope with the limitations, this paper re-engineers the LLCDM with MongoDB NoSQL database. We extensively use the document-oriented semi-strucuted data schema of MongoDB for representing the `<content>` column. We also re-implement Web-API for the LLCDM which allows queries with both application-specific and neutral attributes. We evaluate performance and complexity of the new system through application development with real sensor data.

Categories and Subject Descriptors

H.3.5 [**INFORMATION STORAGE AND RETRIEVAL**]: Online Information Services—*Web-based services*; D.2.12 [**SOFTWARE ENGINEERING**]: Interoperability—*Data mapping*; H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Performance evaluation*

General Terms

DESIGN, PERFORMANCE, STANDARDIZATION

Keywords

lifelog, data integration, mashup, api, web services, NoSQL

1. INTRODUCTION

Lifelog is a social act to record and share human life events in open and public form [9]. A variety of *lifelog services* currently appear in the Internet. Popular services include *Twitter* for delivering tweets, *foursquare* for sharing the "check-in" places.The great success of these services lies not only in information technologies, but also in the nature of human beings, loving to collect and store possessions, memories, experiences [10]. Integrating different lifelogs may create more values rather than using them separately. In this paper, we use a term *lifelog mashup* to refer to such integration of different lifelogs to create a value-added service.

To support efficient lifelog mashup, we have previously proposed *lifelog mashup platform* considering of the *lifelog common data model (LLCDM)* and the *lifelog mashup API (LLAPI)* [8]. The LLCDM prescribes a generic data schema for lifelog records, which does not rely on any specific lifelog service. The LLCDM is designed based on an interrogative analysis, deriving standard attributes from viewpoints of what, why, when, who(m), where and how.

The previous LLCDM was implemented with MySQL, and *application-neutral* attributes like `<date>, <time>, <user> and <location>` were managed within the platform. SQL over these attributes could retrieve data of heterogeneous lifelog services, which was published as the LLAPI. However, *application-specific* data was regarded as *unstructured data* named `<content>`. Various types of data (e.g., numeric values, text, JSON or XML) were all stored in a `<content>` column in a schemaless format. Any query with application-specific attributes could not be expressed by SQL, and had to be managed manually by individual applications. It had also a scalability issue as the data size grew.

To cope with the limitations, we re-engineer the LLCDM with MongoDB, a document-oriented NoSQL database, in this paper. Using semi-structured data schema of MongoDB extensively, we represent each `<content>` data as a document. Considering limitations of MongoDB, we carefully re-design the LLCDM so as to achieve both flexibility for various types of data and powerful queries over the documents. We also re-implement the LLAPI by which users can put and get various types of lifelogs using queries with both application-specific and neutral attributes. The LLAPI is deployed as Web service.

To show practical feasibility, we conduct an experimental evaluation. In the experiment, we develop an application finding summery days from real logs of a temperature sensor. We build the application using the new and old implementations of the LLAPI, and compare them from viewpoints of performance and development complexity. The results show that the new implementation achieves better performance and scalability with easier application development.

iiWAS2013, 2-4 December, 2013, Vienna, Austria

2. PRELIMINARIES

2.1 Lifelog Mashup

Mashup is a new application development approach that allows users to aggregate multiple services to create a service for a new purpose [3]. *Lifelog mashup* is a mashup that aggregates different lifelog services to create a new value-added service. For example, integrating Twitter and Flickr, a person may easily create a photo album with comments (as tweets). Integrating sensor log and activity log, one may recall the weather of a day(s) he went for a jogging.

Some lifelog services are providing APIs for the purpose of lifelog mashups. However, there is no standard specification among such APIs or data formats of the lifelog. Figure 1(a) is a data record of Twitter, describing information of a tweet posted by a user "koupetiko" on 2013-07-05. Figure 1(b) shows a data record retrieved from SensorLoggingService [5], developed in our laboratory, representing a various sensor's values of user "koupe" on 2013-07-05. Although both are in the JSON (JavaScript Object Notation), the data schema of the two records are completely different, and there is no compatibility. Note also that these records were retrieved by different ways by using proprietary APIs and authentic methods. Thus, the conventional mashup applications have been developed as shown in Figure 2.

(a) A data record of Twitter

(b) A data record of Sensor Logging Service

Figure 1: Data of two different lifelog services

2.2 Lifelog Mashup Platform

To support efficient lifelog mashup, we have previously proposed a *lifelog mashup platform* [8]. The platform consists of the *lifelog common data model (LLCDM)* and the *lifelog mashup API (LLAPI)*, as shown in Figure 3.

Lifelog Common Data Model (LLCDM)

In the proposed platform, data stored in heterogeneous lifelog services are transformed and aggregated in an application-neutral form, called *lifelog common data model (LLCDM)*. Table 1 shows the data schema of the LLCDM. Data attributes were carefully determined from the viewpoints of when, who, where, how, what and why. Attributes from the when, who, where and how perspectives are *application-neutral* attributes, which are commonly interpreted among different lifelog services. Therefore, the data values for these attributes are normalized in the LLCDM.

On the other hand, data from the what perspective varies from a lifelog service to another. To accept a wide variety of (even unknown type of) lifelog data, the LLCDM stores the raw data in the `<content>` attribute, as a unstructured plain text. The LLCDM itself does not interpret the `<content>` data. Instead the LLCDM has a reference to an external data schema `<ref_schema>`, by which an application interprets the raw data. The previous LLCDM was implemented using MySQL.

Lifelog Mashup API (LLAPI)

The LifeLog Mashup API (*LLAPI*) searches and retrieves lifelog data conforming to the LLCDM. The following shows an API that gets lifelog data records matching a given query. Using `getLifeLog()`, heterogeneous lifelogs can be accessed uniformly without proprietary knowledge of lifelog services.

```
getLifeLog(s_date, e_date, s_time, e_time, user,
 party, object, location, application, device, select)
Parameters:
s_date s_date    : Query of <date> (start, end)
s_time e_time    : Query of <time> (start, end)
user,party,object: Query of <user>,<party>,<object>
location         : Query of <location>
application      : Query of <application>
device           : Query of <device>
select           : List of items to be selected
```

To allow invocations from various platforms, the LLAPI was published as Web service (REST, SOAP) wrapping an SQL statement to execute the query.

2.3 Limitations in Previous Implementaion

The previous implementation had two major limitations.

The first limitation is that we could not specify application-specific attributes (contained in the `<content>` column) for data query. As seen in the above, the content was stored in an unstructured plain text, over which SQL cannot describe a condition. Thus, all the parameters of the LLAPI are given over the application-neutral attributes.

For example, the previous LLAPI was able to support a query like "Q1: get logs on 2013-08-01 of all possible temperature sensors". However, we could not give a query like "Q2: get logs of a temperature sensor `t1`, where its temperature was greater than 25 degree". To process Q2, an application had to retrieve all logs of `t1` first, then manually parse the `<content>` column to evaluate the condition "temperature $>$ 25". The same thing is true for any application-specific

Table 1: Common data schema of LLCDM

perspective	data items	Description	Instance
WHEN	<date>	Date when the log is created (in UTC)	2013-07-05
	<time>	Time when the log is created (in UTC)	12:46:57
	<epoch>	UNIX Time (sec) when the log is created (in UTC)	1373028417
WHO	<user>	Subjective user of the log	koupe
	<party>	Party involved in the log	shinsuke_mat
	<object>	Objective user of the log	masa-n
WHERE	<latitude>	Latitude where the log is created	34.72631
	<longitude>	Longitude where the log is created	135.23532
	<altitude>	Altitude where the log is created	141
	<address>	Street address where the log is created	1-1, Nada, Kobe
	<name>	place name where the log is created	Kobe University
HOW	<application>	Service/application by which the log is created	Flickr
	<device>	Device with which the log is created	Nikon D7000
WHAT	<content>	Contents of the log (whole original data)	<photo id=".." owner=".." title=".."
	<ref_schema>	URL references to external schema	http://www.flickr.com/services/api/
WHY	n/a	n/a	n/a

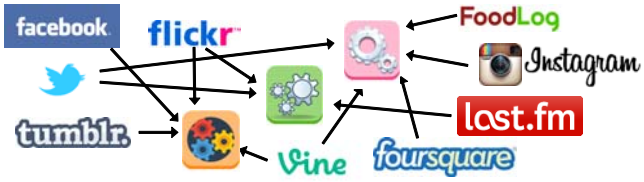


Figure 2: Conventional Approach of Lifelog Mashup

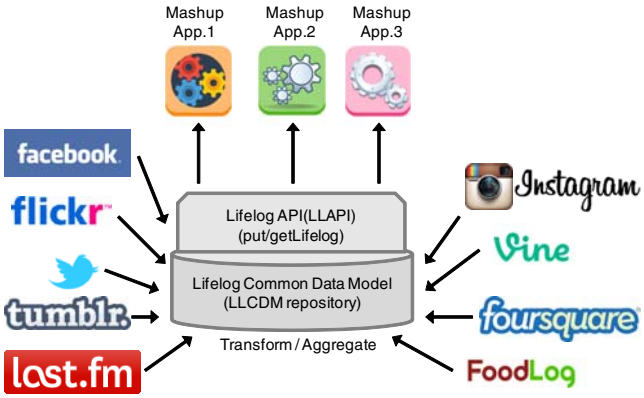


Figure 3: Proposed Lifelog Mashup Platform [8]

attributes like humidity, text of tweets, url of a picture, etc. Moreover, it is impossible to retrieve only temperature's value like the SQL statement's "SELECT".Any query with application-specific attributes had to be managed by individual mashup applications. This imposed large application overhead and expensive development cost.

The second limitation is scalability for the size of lifelog data as the previous implementation used MySQL RDB. In the future, more and more lifelog services will appear, and data from these new services will be exploded. Our platform should be scalable enough to the era of big data.

3. IMPLEMENTING LIFELOG MASHUP PLATFORM WITH NOSQL DATABASE

3.1 Goal and Approach

The goal of this paper is to overcome the above limitations. Specifically, we re-engineer the LLCDM and the LLAPI so as to meet the following goals:

- **G1:** Enable data queries with application-specific data attributes within the <content> column.
- **G2:** Address the scalability issue.

To achieve these goals, we replace MySQL with a document-oriented NoSQL Database *MongoDB* [6].

The essential problem of the previous implementation was that the rigorous data schema of RDB (i.e., MySQL) could not express well the structure of various (or even unknown) types of data stored in the <content> column. Our key idea is to employ a *flexible schema* of MongoDB, where data schema can be determined dynamically. With MongoDB, every data from heterogeneous lifelog services can be represented by a semi-structured *document*, consisting of nested key-values. MongoDB can execute flexible data queries for the semi-structured documents, which is promising to achieve goal G1. As for goal G2, NoSQL databases, including MongoDB, are easy to scale in general. The cost/gigabyte or transaction/second can be many times less that the cost for relational database management systems [7].

There are, of course, limitations of NoSQL compared to RDB. Therefore, we have to carefully re-engineer the LLCDM and LLAPI, considering the limitations of MongoDB within our problem domain.

3.2 Characteristics of MongoDB

MongoDB is a schema-less document oriented database developed by 10gen and an open source community [6]. In MongoDB, the terms "collection" and "document" are used instead of "table" and "row" in SQL databases. We here briefly review characteristics of MongoDB from the aspect of advantage (Pros) and disadvantage (Cons).

Pros1: Document-Oriented Storage
MongoDB stores documents as BSON (Binary JSON) objects, which are binary encoded JSON like objects. BSON supports nested object structures with embedded objects and arrays like JSON does [6].

Pros2: Full Index Support
MongoDB indices are explicitly defined using an en-

sureIndex call, and any existing indices are automatically used for query processing [1].

Pros3: High Availability, Easy Scalability
MongoDB supports automatic sharding, distributing documents over servers [1]. And it supports replication with automatic failover and recovery, too.

Pros4: Supports MapReduce
MongoDB supports MapReduce. MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks [2].

Cons1: No Transaction
MongoDB has no version concurrency control and no transaction management [4]. But, atomic operations are possible within the scope of a single document.

Cons2: No JOIN
MongoDB does not support joins. So, some data is denormalized, or stored with related data documents to remove the need for joins.

Cons3: Not Matured Technology
MongoDB has a relatively short history compared to RDBMSs such as mySQL, PostgreSQL and so on. It indicates frequently releases of new versions.

3.3 Validating Applicability of MongoDB

Based on the above characteristics, we examine the applicability of MongoDB to achieving goals G1 and G2.

P1 and P2 are primary characteristics to achieve G1. In the LLCDM repository, raw data from heterogeneous lifelog services are all stored in the `<content>` column. Each data item has a different set of attributes stored in a specific data structure. Therefore, the LLCDM cannot force the static data schema for `<content>` column. The BSON object of MongoDB is well suited to represent such dynamically-typed data. Columns except `<content>` must be defined in the standard schema of the LLCDM. The full index support is useful for queries over these columns. A drawback of the flexible schema is that the database may allow corrupted or invalid data to be inserted. Therefore, we have to implement a validator for the LLCDM to filter wrong data.

P3 and P4 contribute to G2. MongoDB can manage a huge amount of data, as the name "mongo" came from as sub-string of "hu**mongo**us". MapReduce provides powerful distributed processing for large-scale data, which should be useful for the large-scale data processing in the LLAPI.

Discussion from the Cons aspect is as follows. Once the lifelog data is put in DB, it is never updated since the lifelog is a log. Without the update, the lack of transaction is not a problem for managing the LLCDM. Thus, there is no special trouble with C1. C2 means that we need to break a way of the conventional data modeling with normalization. According to the practice of MongoDB, when an entity has a "contains" relationship, it is recommended to embed the child in the parent. When entities have one-to-many relationships, it is recommended to embed or reference. The embedding is recommended in case of one-to-many relationships where the many objects always appear in the parent. In the new implementation, we count the practice. We do not worry about C3, since MongoDB is a new technology.

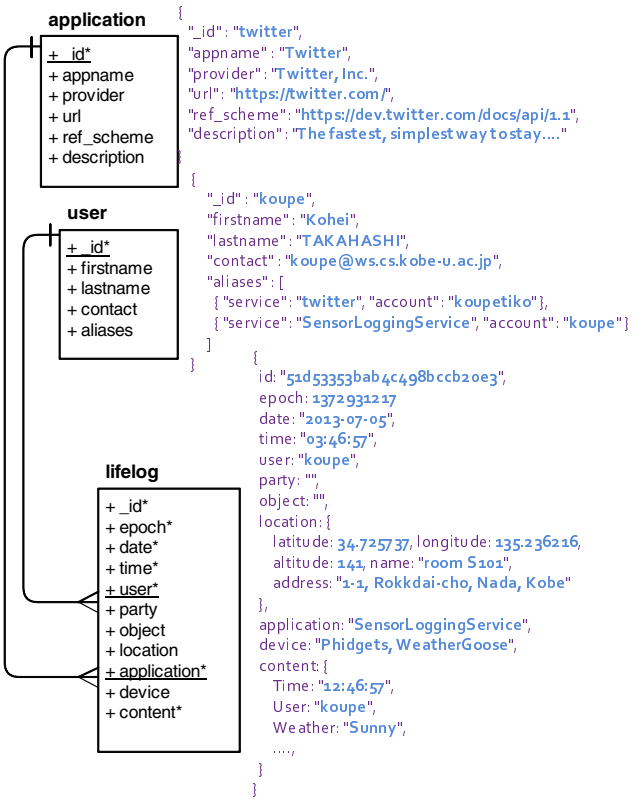


Figure 4: ER diagram for the LLCDM repository

3.4 Data Modeling of LLCDM with MongoDB

Considering the above issues, we re-design the data model of the LLCDM with MongoDB.

Figure 4 shows an ER diagram representing the new version of the LLCDM. A box represents an entity (i.e., collection in MongoDB), consisting of an entity name, a primary key, foreign key, and attributes. We enumerate instances beside each entity in JSON format to support understanding. A line represents a relationship between entities, where +—··· denotes a reference relationship. An underlined attribute of an entity represents a primary key. An attribute with brackets represents a foreign key, pointing to a primary key of another entity. An attribute with a asterisk represents a mandatory (non-null) attribute. The proposed model consists of the following three collections.

(a) lifelog: This is the main collection of the LLCDM. All lifelog data retrieved from individual lifelog services are managed in this entity. As seen in Section 2.2 and Table 1, the entity has attributes of the LLCDM. As for `content` attribute, the raw data of a lifelog service is stored in a BSON document. Detailed information of a user and an application are referred to external entities of user and application, respectively.

(b) application: In this collection, we manage the detailed information of individual lifelog services and applications. The application information is used to identify the access method to retrieve the data and the data schema of the retrieved data. Since we consider that these information are not always needed for every

lifelog record. we manage it in a separate collection.

(c) user: This collection manages the user information, consisting of ID, user name, contact information, alias names used in individual services. The reason why we provide this collection is that the user information is commonly attached in various lifelog data, and is one of the most frequently used information in the mashup. Also, the same user often uses different id in different lifelog services. Therefore, we newly define an aliases attribute to consolidate different names.

3.5 Designing LLAPI with MongoDB

On top of the new LLCDM, we develop a new LLAPI consisting of two methods: `putLifelog()` and `getLifelog()`. The `putLifelog()` method puts a given lifelog data into the LLCDM, while the `getLifelog()` method gets lifelog data from the LLCDM based on a given query.

As a drawback of the flexible schema, MongoDB does not have strict data checking mechanisms (e.g., type check, format check, key constraints check, etc.) which RDBMS usually has. Therefore, without any consideration, the LLCDM would accept invalid (or even corrupted) data. The `putLifeLog()` method plays a role of a *gatekeeper* of the LLCDM, which strictly validates the conformance of the given lifelog data. When an application calls `putLifelog()` method with parameters corresponding to the attributes of the lifelog entity, the parameters are validated based on the schema of the LLCDM. Once the validation is passed, a BSON document is created with the parameters and is inserted into the LLCDM. Thus, the lifelog data inserted via `putLifelog()` conforms to the LLCDM schema.

Once the lifelog data is stored in the LLCDM, we can retrieve the data using powerful queries language of MongoDB. Extending the capability of the previous LLAPI, we develop the `getLifelog()` method as follows.

```
getLifelog([s_date, e_date, s_time, e_time, s_term,
  e_term, user, party, object, s_alt, e_alt, s_lat,
  e_lat, s_long, e_long, loc_name, address, application,
  device, content, select, limit, order, offset])

Parameters:
  s_date, e_date   : Query of <date> (start, end)
  s_time, e_time   : Query of <time> (start, end)
  s_term, e_term   : Query of <epoch> (start, end)
  user,party,object: Query of <user>,<party>,<object>
  s_alt, e_alt     : Query of <location.altitude>
  s_lat, e_lat     : Query of <location.latitude>
  s_long, e_long   : Query of <location.longitude>
  loc_name         : Query of <location.name>
  address          : Query of <location.address>
  application      : Query of <application>
  device           : Query of <device>
  content          : Queries of <content>
  select           : List of items to be selected.
  limit            : Limit on # of data items.
  order            : Sorting order of items.
  offset           : Skip retrieved items.
  tz               : Timezone for date and time.
```

Compared to the previous version (see Section 2.2), the new version can take more parameters, including queries of content, select columns of all fields, limit of records, order of sorting, timezone. Table 2 summarizes the comparison of features of the old and new versions of the LLAPI. In the Table 2, the triangle mark in the column of "select" means it is imperfection. Because the previous LLAPI can select the application neutral columns, but it can't select the columns of in `<content>` columns (e.g., `content.temperature`, `content.status.text`). And we consider this is the major difference.

For given parameters, `getLifelog()` first parses a query from each parameter. A null value is considered to be `true`. The method then adjusts `date`, `time`, `term` parameters to UTC time based on `tz` parameter. From `content` parameter, each query is stored in array `$contentq[]` (e.g., `[{content.temperature $ge 25}, {content.humidity $lt 40}]`, where `$ge` and `$lt` represent $\geq$ and $<$, respectively). From `select` parameter, each attribute is stored in array `selectq[]` (e.g., `[date, time, content.temperature]`. Finally, `getLifelog()` builds the following MongoDB query.

```
>db.lifelog.find({ date: {$ge: $s_date, $le: $e_date},
  time: {$ge: $s_time, $le: $e_time},
  epoch: {$ge: $s_term, $le: $e_term},
  user: $user, party: $party, object: $objectq,
  location.altitude: {$ge: $s_alt, $le: $e_alt},
  location.latitude: {$ge: $s_lat, $le: $e_lat},
  location.longitude: {$ge: $s_long, $le: $e_long},
  location.name: /$loc_name/,
  location.address: /$address/,
  application: $application, device: $device,
  $contentq[0], $contentq[1], ...
 }, { $selectq[0]: 1, $selectq[1]: 1, ...}
).limit($limitq).skip($offsetq).sort({$orderq})
```

3.6 Implementation of LLAPI

Based on the above design thought, we have implemented the LLAPI in Java. We used Morphia OR-mapper for marshaling tuples into objects. To validate the parameters, we have used Java Validation API (Hibernate Validator (JSR303) Reference Implementation for Bean Validation). In order to publish LLAPI as a Web service, we used JAX-RS (Jersey: JAX-RS (JSR 311) Reference Implementation for building RESTful Web services). The total lines of code is 1,436.

Now that the LLAPI can be executed by REST Web service protocol. Figure 5 shows a response of `getLifelog()`. In this example, a Twitter data record describing a tweet posted by a user "koupetiko" on 2013-05-01 is retrieved.

4. EXPERIMENTAL EVALUATION

4.1 Overview

We conduct an experimental evaluation to evaluate the proposed platform. In the experiment, we use environmental sensor log from `SensorLoggingService` deployed in our smart home. This service measures environment inside/outside of our laboratory, using various sensors including temperature, humidity, brightness, pressure, motion and the number of people. The sensor log has been recorded every minute for over three years, comprising 1,664,937 records. For the experiment, all the records are imported to both the new platform (with MongoDB) and the old platform (with MySQL).

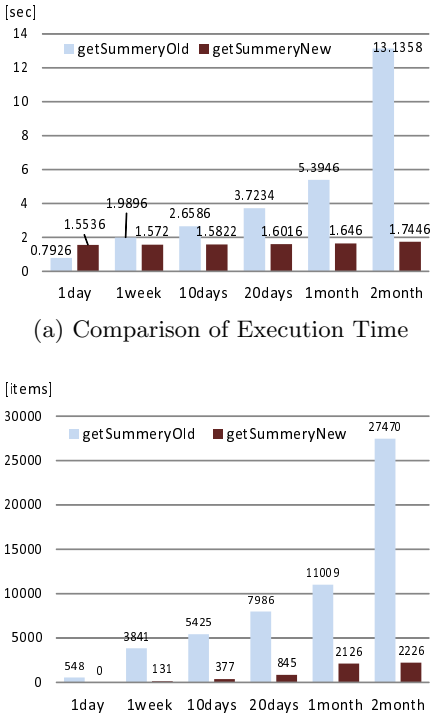
To measure the performance, we develop a client application that picks out *summery days* from the sensor log. A summery day is defined as a day, where the maximum temperature between 9 a.m and 6 p.m exceeds 25 degree. Using

Table 2: Comparison of Functions

LLAPI	date	time	term	location	application	device	content	select	limit	order	offset	timzone
old (mySQL)	√	√		√	√	√		△				
new (Mongo)	√	√	√	√	√	√	√	√	√	√	√	√


```
{
  count: 19,
  result: [
    {
      id: "51d531d1bab4c498bccb1f63",
      date: "2013-05-01",
      time: "13:10:00",
      user: "koupetiko",
      party: "",
      object: "",
      application: "twitter",
      device: "<a href="krile...">Krile2</a>",
      content: {
        source: "<a href="http://krile...">Krile2</a>",
        created_at: "Wed May 01 13:10:00 +0000 2013",
        text: "What's a bad machine...",
        user: {
          profile_image_url_https: "https://..._normal.gif",
          screen_name: "koupetiko",
          ...
        },
        entities: {
          ...
        },
        ...
      }
    },
    {
      id: "51d531d1bab4c498bccb1f64",
      date: "2013-05-01",
      time: "13:08:05",
      ...
    },
    ...
  ]
}
```


Figure 5: a response of getLifelog



(a) Comparison of Execution Time

(b) Comparison of Retrieved Items for tha same purpose

Figure 6: Graphs of the Result of Picking out Summery days

the Perl language, we develop two versions of the client: one is with the new LLAPI and another with the old LLAPI. We execute both versions of the clients and measure the execution time. To see the scalability, we vary the term of the sensor log to be searched as follows: (1 day, 1 week, 10 days, 20 days, 1 month, 2 months).

4.2 Clients Finding Summery Days

We here see the difference of the two versions of the client.

(1) `getSummeryNew.pl`: client with the new LLAPI

The client with the new LLAPI requires the following 2 steps to identify the summery days:

Step1: Execute the new LLAPI as follows to obtain lifelog records that meets the condition of summery day:

```
getLifelog({s_date:start_date, e_date:end_date,
  s_time:09:00:00, e_time:18:00:00,
  content:[{content.temperature: {$gt: 25}}],
  application:SensorLoggingService, select:[date]})
```

Step2: Pick out only distinct date values.

(2) `getSummeryOld.pl`: client with the old LLAPI

Since the old LLAPI does not allow a query over the `content` column, the client application by itself has to evaluate the condition over the temperature.

Step1: Execute the old LLAPI as follows to obtain all lifelog records within the term:

```
getLifelog({s_date:start_date, e_date:end_date,
  s_time:09:00:00, e_time:18:00:00,
  application:SensorLoggingService})
```

Step2: By parsing the content column of every record, extract the value of temperature. If the value is greater than 25, put the date in the result set.

Step3: Pick out distinct date values from the result set.

4.3 Result of Experiment

According to the search term (1 day, 1 week, 10 days, 20 days, 1 month, 2 months), we set values of `start_date` and `end_date` in both clients. For each term, we execute the clients five times and measure the average execution time.

Figure 6 shows the result. Figure 6(a) shows the result of execution time. `getSummeryNew.pl` with the proposed

LLAPI outperforms `getSummeryOld.pl` with the old API. To evaluate the overhead within the clients, we also count the number of data records extracted from the database. Figure 6(b) shows the comparison of the number of data records. As shown in the figure, the client with the old LLAPI retrieves much more data records.

4.4 Discussion on Performance & Scalability

The execution time of `getSummeryOld.pl` is proportional to the number of retrieved items. This is because the temperature condition has to be evaluated by the client itself. Therefore, the performance drops as the search term becomes longer and the number of records becomes larger.

In contrast, the execution time of `getSummeryNew.pl` increases very slowly even if the search term becomes longer. The major reason is that the temperature condition is evaluated within the MongoDB management system, which is quite scalable. The number of retrieved records is suppressed and the overhead to the client is quite low.

Fitting the curve of Figure 6(a) onto a linear function, the slope of the execution time of `getSummeryOld.pl` is about 0.2 seconds per day, while the one of `getSummeryNew.pl` is about 0.003 seconds per day. Thus, the new LLAPI with MongoDB enables to create more scalable applications.

It is interesting to see in Figure 6(a) that `getSummeryOld.pl` is faster for 1 day. This shows that MySQL is a very fast database system, working efficiently with the relatively small number of records. Most of the execution time is taken for the Perl client to parse the content column. If MySQL allows a structured data field (although it is against the principle of RDB), then it might outperform MongoDB.

4.5 Discussion on Application Complexity

As discussed in Section 4.2, the old LLAPI does not accept query on `<content>` of the LLCDM. Therefore, if a client application wants to search lifelog data with application-specific attributes, then the client has to implement the following tasks:

- (T1: Get) Obtain the lifelog records based on a query with application-neutral data attributes.
- (T2: Parse) For every record obtained, parse the unstructured text data in `<content>` to extract necessary data values.
- (T3: Filter) Evaluate the data values for a certain condition.

The old platform takes care of task T1 only. Tasks T2 and T3 are left for application developers. Thus, individual developers have to write own code for implementing application-specific parser and filter. This increases the development complexity as well as the code complexity, which severely affects the reliability of the application. Our example of `getSummeryOld.pl` was relatively simple, since only a single attribute (temperature) was used. In general, however, more complex condition with more attributes would be used for lifelog mashup.

On the other hand, in an application with the proposed new LLAPI, we can execute a query with both application-specific and application-neutral attributes. Thus, all the above tasks T1, T2 and T3 are executed within the new platform. As a result, the structure of the application becomes much simpler, as shown in our example of `getSummeryNew.pl` in Section 4.2. The developer just concentrates on implementing how to cook and show the obtained lifelog data. The simple application structure improves the readability of the code and improves the reliability.

5. CONCLUSION

In this paper, we re-engineered the existing lifelog mashup platform [8] using a MongoDB NoSQL database, to cope with the limitation on data query and scalability.

We also evaluated performance and development complexity of the new platform through application development with real sensor data. The experimental results showed that the application with the new API achieves higher performance and scalability with lower application complexity, compared to the one with the previous API.

6. 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)] and Sekisui House, Ltd.

7. REFERENCES

[1] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[3] G. D. Lorenzo, H. Hacid, H. young Paik, and B. Benatallah. Data integration in mashups. volume 38, pages 59–66. ACM, 2009.

[4] A. Milanović and M. Mijajlović. A survey of post-relational data management and nosql movement.

[5] A. Okushi, S. Matsumoto, and M. Nakamura. Considering value-added services using environmental data collected by personal mobile sensing. In *IEICE Technical Report (Japanese Eds.)*, volume 112, pages 1–6, November 2012.

[6] R. P. Padhy, M. R. Patra, and S. C. Satapathy. Rdbms to nosql: Reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011.

[7] S. Pastore. Web-oriented data formats and their management in the mobile era. *Mobile Computing*, 2(2), 2013.

[8] A. Shimojo, S. Matsuimoto, and M. Nakamura. Implementing and evaluating life-log mashup platform using rdb and web services. In *The 13th International Conference on Information Integration and Web-based Applications & Services (iiWAS2011)*, pages 503–506, December 2011.

[9] K. Takata, J. Ma, B. O. Apduhan, R. Huang, and Q. Jin. Modeling and analyzing individual's daily activities using lifelog. In *Embedded Software and Systems, 2008. ICESS'08. International Conference on*, pages 503–510. IEEE, 2008.

[10] Trend Watching .com. Life caching – an emerging consumer trend and related new business ideas. `http://trendwatching.com/trends/LIFE_CACHING.htm`.