A Timestamp Database Structure of Efficiency for Big Data Generated from Sensors

Hyontai Sug

***Abstract*—** In the environment of Internet of Things (IoT) sensor data are generated and collected in very large scale. We should store the data in databases for later analysis or data mining. Sensor data consist of measured date and time as well as fractional seconds information. Because the interval of sensor data gathering is usually very narrow and a lot of sensor data are generated in a short period of time, it is highly possible that the data contain a lot of redundant measurement time or date information. As a result, the redundant data consume a large portion of storage space. In order to use the storage space more efficiently, we suggest a database structure consisting of key and time or date information to store the redundant part of time information separately. We also suggest related functions that will split time or date part from the original time format and merge the split data together for later retrieval. Because computing resource becomes cheaper and cheaper and in-memory database technology that can allow database operations cheaper becomes more popular, the expense for the additional database operations may be ignored. As a result, we may save significant amount of storage space by the expense of the additional database tables and related database operations.

***Keywords*—**Sensor databases, timestamp data, key structure, data storage.

I. Introduction

SENSOR data collection and storage is one of the major task in the tasks of Internet of Things (IoT). IoT technology has wide range of applications including healthcare [1], robotics [2], wireless sensor networks [3], and environment monitoring [4], etc. Wide applications of IoT technology generates and stores a lot data in storage so that we are confronting the era of big data. In big data environment storing large amount of data is one of major concern [5, 6, 7] so that saving storage as much as possible is important [8, 9, 10]. Sensor data has the information consisting of time or date and sensor data. For later analysis we should store the data in databases. Because sensors are used to detect some change as time elapses, time information is stored with the change. The following is an example of sensor data consisting of Unix time [11] and room temperature:

{1522122416, 341}
{1522122421, 342}
{1522122426, 343}
{1522122431, 342}

H. Sug is with the Division of Computer and Information Engineering, Dongseo University, Busan, 617-716 Korea (phone: +82-51-320-1733; fax: +82-51-327-8955; e-mail: sht@ gdsu.dongseo.ac.kr).

The standard Unix time data type that represents a point in time is a signed integer. Usually it occupies 32 bits. Using 32 bits to represent time covers a range of only about 136 years. The earliest time is 1901-12-13, and the latest time is Tuesday 2038-01-19. Because of its limitation in time range, some operating systems use 64 bits to represent time. Anyway, as you can see in the time data, there are many redundant numbers in representing the Unix time, because the temperatures were measured regularly and frequently.

On the other hand, many sensor systems store sensor data in conventional databases, and data in databases last long time. But, because date and time data type of databases use more bytes than sensor data to store the information, reducing the redundant information is important to save storage.

There are many database management systems that can store the sensor data. Among them Oracle and MySQL database management system (DBMS) can be two representatives in mostly used database management systems [12, 13, 14]. Both of Oracle and MySQL use their own internal format to store date and time information. Oracle DBMS uses seven bytes to store date and time information. MySQL DBMS uses five bytes to store the same information. But, additional bytes are needed in both DBMSs to store fractional seconds data which are essential for sensor data.

Date data in Oracle is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second. Because one byte can represents 256 different symbols, Oracle can represent up to 256 different centuries. In other words, Oracle can represent 25,600 years theoretically, and actual range of date information begins from January 1, 4712BC and ends December 31, 9999AD. On the other hand, MySQL can represent in the range from January 1, 1000AD to December 31, 9999AD only. Oracle also uses timestamp data type to store event time in nanosecond. In order to store timestamp data, Oracle needs additional five bytes. Similar format is used to store timestamp data n MySQL. Therefore, if we store event time using timestamp in Oracle, each time data needs twelve bytes. But, because sensor data can be generated very frequently and regularly, these bytes for time information can occupy a lot of storage redundantly, especially for date part. Therefore, we need some mechanism that may reduce the redundant date and time information in the databases. In section 2 backgrounds like the related data format of MySQL and Oracle and problem formation will be presented, and in section 3 suggested method will be presented, and finally

conclusion will be provided in section 4. This paper is the modified and extended version of previously presented paper at ACE'15[15].

II. BACKGROUD AND PROBLEM FORMATION

A. Time Format of MySQL

MySQL has five different data types to store date and time information; YEAR, DATE, TIME, DATETIME, and TIMESTAMP data type.

The YEAR type is used for values with a year part but no other part.

The DATE type is used for data values with a date part but no time part. MySQL displays DATE values in 'YYYY-MM-DD' format. The supported date range is '1000-01-01' to '9999-12-31'.

The TIME type is used for data values with a time part possibly with fractional seconds part but no date part.

The DATETIME type is used for data values containing both date and time parts. MySQL displays DATETIME values in 'YYYY-MM-DD HH:MM:SS' format. The supported DATETIME range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

The TIMESTAMP data type is used for data values containing both date and time parts. TIMESTAMP has a range of '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC. So, the range is more restricted than that of DATETIME. MySQL DBMS converts TIMESTAMP values from the current time zone to UTC for storage, and back from UTC to the current time zone for retrieval. By default, the current time zone for each connection is the server's time. Note that other time types such as DATETIME are not in UTC time format.

UTC is the time standard across the world. The time scales are closely synchronized by international atomic time and coordinated by universal time (UT1) also known as astronomical time or solar time referring to the rotation of the earth. It is known that English speaking people wanted to use the name Coordinated Universal Time (CUT), while French speaking people wanted to use the name Temps Universel Coordonné(UTC) so that both people agreed to use the name UTC [16].

A TIME, DATETIME, and TIMESTAMP value can include a trailing fractional seconds part in up to 6 digits of precision so that the fractional seconds can be represented in microseconds. The fractional part is separated by a decimal point. With the fractional part included, the format for these values is 'YYYY-MM-DD HH:MM:SS.999999', the range for TIME values is '00:00:00.000000' to '23:59:59.999999', and the range for DATETIME values is '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999', and the range for TIMESTAMP values is '1970-01-01 00:00:01.000000' to '2038-01-19 03:14:07.999999'. Because of the short range of years in TIMESTAMP data type, DATETIME is often recommended.

MySQL attempts to save storage as much as possible by allowing the fractional part in different length if needed. Before MySQL 5.6.4 each time data type has fixed length. For example, YEAR type used 1 byte, DATE type used 3 bytes, TIME type used 3 bytes, DATETIME uses 8 bytes, and TIMESTAMP uses 4 bytes. But, as of MYSQL 5.6.4 fractional seconds part is included in different length as needed for TIME, DATETIME, and TIMESTAMP type as in table 1 and table 2. The fractional part requires from 0 to 3 bytes.

Table 1. Storage requirement in MySQL

Data type	Storage required
YEAR	1 byte
DATE	3 bytes
TIME	3 bytes + fractional seconds storage
DATETIME	5 bytes + fractional seconds storage
TIMESTAMP	4 bytes + fractional seconds storage

As you see in table 1 storage requirement for YEAR and DATE type is unchanged. But, the other three types like TIME, DATETIME, and TIMESTAMP uses different formats and DATETIME is packed more efficiently using 5 bytes for non-fractional part and 0 to 3 bytes for fractional part, depending on the precision of fractional seconds of stored values.

Table 2. Storage requirement for fractional seconds in MySQL

Precision of fractional seconds	Storage required
0	0 bytes
1 ~ 2	1 byte
3 ~ 4	2 bytes
5 ~ 6	3 bytes

The TIME, DATETIME, and TIMESTAMP types can have a fractional part. Storage structure for these types is big endian [17] with the non-fractional part followed by the fractional part.

Table 3 shows TIME data type encoding for non-fractional part. The negative sign bit is reserved for future.

Table 3. TIME data type encoding

Usage	Bits	Remarks
sign	1	1: nonnegative, 0: negative
-	1	Reserved for future
hour	10	0-838
minute	6	0-59
second	6	0-59
total	24	

Table 4 shows DATETIME data type encoding for non-fractional part. The negative sign bit is reserved for future.

Table 4. DATETIME data type encoding

Usage	Bits	Remarks
sign	1	1: nonnegative, 0: negative

Year & month	17	year: 0-9999 month: 0-12
day	5	0-31
hour	5	0-23
minute	6	0-59
second	6	0-59
total	40	

Table 5 shows TIMESTAMP data type encoding for non-fractional part. The negative sign bit is reserved for future.

Table 5. TIMESTAMP data type encoding

Usage	Bits	Remarks
sign	1	1: nonnegative, 0: negative
Year & month	11	year: 1970-2038 month: 0-12
day	5	0-31
hour	5	0-23
minute	6	0-59
second	6	0-59
total	32	

B. Time Format of Oracle

Oracle uses five different data types to store time data; DATE, TIME, TIMESTAMP, TT_DATE, TT_TIMESTAMP. TT_DATE and TT_TIMESTAMP are shorter and faster version of DATE and TIMESTAMP data type respectively.

The DATE data type stores the century and year, the month, the day, the hours, the minutes, and the seconds. Oracle uses seven bytes to store dates in the Julian era, ranging from January 1, 4712 BC through December 31, 9999 AD. The fixed-length fields of seven bytes each corresponds to century, year, month, day, hour, minute, and second. There are no fractional seconds in DATE type. Oracle stores time in 24-hour format like HH:MI:SS. By default, the time in a date field is 00:00:00 A.M. if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month.

Julian dates allow continuous dating by the number of days from a reference. The reference is 01-01-4712 years BC, so Oracle uses integer values in seven digits to refer to a specific date. You can use TO_NUMBER function if you want to use Julian dates in calculations. You can also use the TO_DATE function to enter Julian dates with the seven digit numbers.

The TIME type is used for data values containing time only without date data so that eight bytes are enough. TIME values can be stored in 'HH:MM:SS' format ranging from 00:00:00 to 23:59:59. The fractional part of precision up to 9 digits can be stored in the TIME data type.

The TIMESTAMP data type is used for data values containing both date and time parts. TIMESTAMP has a range of '4712-01-01' BC to '9999-12-31' AD. The TIMESTAMP type needs 12 bytes of storage, and the fractional seconds precision range is up to 9.

The TT_DATE data type is used for data values containing date part only with reduced range of date range to use less amount of storage. It has a range of '1753-01-01' AD to '9999-12-31' AD. The TT_DATE type needs four bytes of storage.

The TT_TIMESTAMP data type is used for data values containing both date and time part with reduced range of date range to use less amount of storage. It has a range of '1753-01-01' AD to '9999-12-31' AD, and the fractional seconds precision range is up to 6. The TT_TIMESTAMP type needs eight bytes of storage.

As we understand the storage structure of MySQL and Oracle DBMS to represent time and fractional seconds, we can find that more attention is given to store the macro time information rather than the micro time information. This fact is originated from the fact that the traditional usage of databases is to store business data. Because most business data cover human business activities, the amount of 'micro' time information is relatively smaller than that of 'macro' time information.

C. Problem Formation

Lets' see how the date and time information are stored in the databases. The following data in table 6 and table 7 are some example of sensor data for acceleration measurements [18]. Table 6 shows the first half of the data.

Table 6. The first half of sensor data

63	209	2014-10-22 19:30:07.624000000
63	209	2014-10-22 19:30:07.644000000
63	209	2014-10-22 19:30:07.664000000
63	209	2014-10-22 19:30:07.684000000
63	209	2014-10-22 19:30:07.704000000

Table 7 shows the second half of the data.

Table 7. The second half of sensor data

-0.01928711	0.03955078	1.245605
-0.02612305	0.04418945	1.251465
-0.02172852	0.0378418	1.244629
-0.02954102	0.04052734	1.251953
-0.02392578	0.04003906	1.249023

If the time information in the table is stored in character format, it needs 27 bytes as we can see in the table. So, using 12 bytes to store timestamp information is economic choice as in Oracle. But, as we can see in the time data in table 6, there are many redundant time information of year, month, day, hour, minute, as well as second.

Let's see how we may save storage roughly, if we use some other structures to store the date or time information. In order to make the problem simple, assume that we want to store timestamp information in the database of Oracle so that we make a table to store century, year, month, day among the seven bytes that represent a timestamp. The (century, year, month,

day) information for a specific sensor data will be stored in this new table. The table has two attributes, {sequence_number, century_year_month_day}. Note that the following facts:

- Sequence number of 1 byte can store up to 256 different numbers.
- Sequence number of 9 bits can store up to 512 different numbers.
- One day consists of 3600 × 24 = 86,400 seconds.

So, if we use 9 bits for the sequence number in the table, we can store up to 512 different numbers. Therefore, instead of using the original timestamp format, if we use the new database structure and we measure the sensor data for up to 512 days, we can save 32 – 9 = 23 bits for each timestamp data. Let's see how much storage we can save by the above scheme.

As a simple example, let's assume that one sensor generates one data for each second and we want to gather the sensor data for 512 days, and also assume that we have sensor networks consisting of 10,000 sensors. So, for each day we can have 86,400 × 10,000 = 864,000,000 timestamp data. Conventional timestamp format requires 12 bytes × 864,000,000 ≈ 9.7 GB. But, in our new format, because we use the 9 bits and 8 bytes for (hour, minute, second, fractional seconds) data for each timestamp information, we need only about 75% storage excluding the additional table. Note that the size of the additional table is very small so that it's almost ignorable. Therefore, we can save storage about 25%.

Because we may have a lot of sensors and each sensor may send sensing data frequently and regularly, we may store more information in the new table. For example, we may store (century, year, month, day, hour) information in the new table so that we can save database storage more. Depending on the measuring period, the size of the sequence number can be determined. Saving about more than 25% storage for date or time information will be significant.

III. Suggested Method

A. Oracle's Case

In order to split timestamp data in Oracle's format, we need a new function that can split a timestamp information into two parts; the first part that will have the left part of the timestamp information, for example, like (century, year, month, day), and the second part that will have right part of the timestamp information, for example, like (hour, minute, second, fractional seconds). The head of the function looks like the following:

split_timestamp(arg1, arg2, arg3, arg4)

- The first parameter, arg1, corresponds to the original timestamp information, and
- The second parameter, arg2, is an integer to indicate the location of split.
- The third parameter, arg3, will be used to return the first part of the timestamp information after split, and
- The fourth parameter, arg4, will be used to return the second part of the timestamp information after split.

Let's see an example on how the split_timestamp function can be used. Assume that arg2 is 4, then a timestamp information will be split into two parts, (century, year, month, day) and (hour, minute, second, fractional seconds). The arg3 that has (century, year, month, day) part of the timestamp information, and will be stored in a table, so called CYMD table as in Fig. 1, where the first part of the timestamp information is stored. The arg4 that has (hour, minute, second, fractional seconds) part of timestamp information and will be stored in SensorT table as in Fig. 2, where the measured sensor data and sensorID are also stored.

For our previous example CYMD table has the information of century, year, month, and day. SensorT table has the information of hour, minute, and second as well as sensor data including sensor ID and measured data. The primary key of SensorT table is composite key, {Sequence_number, hour_minute_second_fractional_seconds, sensorID}. Because the attribute century_year_month_day in CYMD table is defined as the primary key, we can check that a new input will need new sequence number or not right away. That is, if we cannot find a century_year_month_day in the CYMD table, we have to generate the next sequence number. In CYMD table and sensorT table underlined attributes are primary keys of each table. Note that depending on the frequency of measurement, we may use more parts in the timestamp format, and use more bits to represent the sequence number in the CYMD table. The corresponding change must be made in the sensorT table also.

Sequence_number	century_year_month_day
9 bits	4 bytes (PK)

Fig. 1. The structure of CYMD table

Sequence_number	hour_minute_second_fractional_seconds	sensorID	Attribute1 . .. AttributeN
9 bits	8 bytes	…	Sensor data . . .

Fig. 2. The structure of sensorT table

Because the size of CYMD table will be relatively small, we may take advantage of in memory database technology for efficiency [19].

Therefore, if we store century, year, month, day in a separate table, a possible SQL statement to recover the original timestamp data can be based on equijoin operation.

```
SELECT original_timestamp(century_year_month_day,
hour_minute_second_fractional_seconds), sensorID,
Attribute1,…,AttributeN
FROM CYMD, sensorT
WHERE CYMD. Sequence_number = SensorT.
Sequence_number;
```

The function original_timestamp(arg1, arg2) accepts two parameters, arg1 and arg2 that represents the separated timestamp information, and the function returns recovered timestamp information. Moreover, if we define the above query as a view, we can access sensor data without worrying about the physical structure of the table. The following is the corresponding view definition.

```
CREATE VIEW sensorD
AS
SELECT original_timestamp(century_year_month_day,
hour_minute_second_ fractional_seconds), sensorID,
Attribute1,…,AttributeN
FROM CYMD, sensorT
WHERE CYMD. Sequence_number = SensorT.
Sequence_number;
```

B. MySQL's Case

Let's see how we can apply our idea for Datetime data in MySQL's format. Datetime uses 5 bytes to store year, month, day, hour, minute, and second. Additional 2 bytes are used for the storage of fractional seconds if we want to store the precision of 4 digits below the decimal point. Note that the sensor data will have smaller precision than that of Oracle's. Table 8 shows an example of similar sensor data with the less precision.

Table 8. The first half of sensor data

63	209	2014-10-22 19:30:07.6240
63	209	2014-10-22 19:30:07.6440
63	209	2014-10-22 19:30:07.6640
63	209	2014-10-22 19:30:07.6840
63	209	2014-10-22 19:30:07.7040

Let's assume that sensor generates data for every 1/100 second, and we want to split a datetime data into two parts; the first part that will have the left part of the datetime information like (year, month, day, hour), and the second part that will have right part of the timestamp information like (minute, second, fractional seconds), because the sensor data are gathered more frequently. The head of the function looks like the following:

split_datetime(arg1, arg2, arg3, arg4)

- The first parameter, arg1, corresponds to the original datetime information, and
- The second parameter, arg2, is an integer to indicate the location of split.
- The third parameter, arg3, will be used to return the first part of the datetime information after split, and
- The fourth parameter, arg4, will be used to return the second part of the datetime information after split.

Let's see an example on how the split_datetime function can be used. Assume that arg2 is 4, then a datetime information will be split into two parts, (year, month, day, hour) and (minute, second, fractional seconds). The arg3 that has (year, month, day, hour) part of the datetime information, and will be stored in a table, so called YMDH table as in Fig. 3, where the first part of the datetime information is stored. The arg4 that has (minute, second, fractional seconds) part of datetime information will be stored in SensorTP table as in Fig. 4, where sensor data like sensorID and measured data are stored also.

For our example YMDH table has the information of year, month, day, and hour. SensorTP table has the information of minute, second, and fractional seconds as well as sensorId and measured sensor data. The primary key of SensorTP table is composite key, {Sequence_number, minute_second_fractional_seconds, sensorID}. Because the attribute year_month_day_hour in YMDH table is defined as the primary key, we can check that a new input will need new sequence number or not right away. That is, if we cannot find a year_month_day_hour in the YMDH table, we have to generate the next sequence number. In YMDH table and sensorTP table underlined attributes are primary keys of each table. Because our sensor generates signal every 1/100 second and we store hour data in YMDH table, we need more bits for the sequence number. Assume that we want to collect sensor data for 10 years. Because we store up to hours in YMDH table and sore up to 10 years, we need $10 \times 365 \times 24 = 87600$ different numbers to distinguish each hour for ten years. Therefore, 17 bits are enough for the sequence number.

Sequence_number	year_month_day_hour
17 bits	28 bits (PK)

Fig. 3. The structure of YMDH table

Sequence_number	Hour_minute_second_fractional_seconds	sensorID	Attribute1 .. . AttributeN
17 bits	28 bits	…	Sensor data . . .

Fig. 4. The structure of sensorTP table

If we store century, year, month, day, and hour in a separate table, a possible SQL statement to recover the original datetime data can be based on equijoin operation.

```
SELECT original_datetime(century_year_month_day_hour,
minute_second_fractional_seconds), sensorID,
Attribute1,…,AttributeN
FROM YMDH, sensorTP
WHERE YMDH. Sequence_number = SensorTP.
Sequence_number;
```

The function original_datetime(arg1, arg2) accepts two parameters, arg1 and arg2 that represents the separated datetime information, and the function returns recovered datetime information. Moreover, if we define the above query as a view, we can access sensor data without worrying about the physical structure of the table. The following is corresponding view definition.

```
CREATE VIEW sensorD
```

```
AS
SELECT original_datetime(century_year_month_day_hour,
minute_second_ fractional_seconds), sensorID,
Attribute1,…,AttributeN
FROM YMDH, sensorTP
WHERE YMDH. Sequence_number = SensorTP.
   Sequence_number;
```

By using the structure we can save storage about 20%. If you want to save some storage for sensor data, you may use TIMESTAMP type in MySQL, or TT_TIMESTAMP type in Oracle. The saving is 8 bits and 4 bytes for MySQL and Oracle respectively compared to the DATETIME and TIMESTAMP type. Anyway, similar procedures can be applied to these data types also to save storage.

Even though other date and time data type in different DBMSs uses some different storage formats, depending on the data redundancy and how much space we want to save, similar procedure and method as the previous examples can be applied to the other date and time data types also.

IV. Conclusion

Sensor data has the information consisting of time or date and measured data. For later analysis we should store the data in databases. Sensor data are gathered regularly and very frequently but not for very long period. Therefore, because the measurement interval is usually very short, it is highly possible that there is a lot of redundant information of measurement time or date, so that the information occupies a large portion of storage space in the databases extravagantly. In order to save the space, we suggest a key and time data structure to store the redundant part of time or date information in the database separately. Because of the separation two additional functions are suggested. The first function splits time or date information to store separately, and the second function merges the split time or date information together for later data retrieval. Rough counting about storage saving is about more than 20% for date or time information by the expense of additional table and database operations. The expense for the additional operation may be small compared to the waste of storage, because computing resource becomes cheaper and cheaper.

References

[1] I. Chiuchisan, O. Geman, "An approach of a decision support and home monitoring system for patients with neurological disorders using internet of things concents," *WSEAS Transactions on Systems*, vol. 13, 2014, pp. 460-469.

[2] C. Turcu, C. Turcu, V. Gaitan, "Merging the internet of things and robotics," *Proceedings of the 16th WSEAS International Conference on Systems*, 2012, pp. 499-504.

[3] Z. Bojkovic, B. Bakmaz, M. Bakmaz, "Some challenging issues for internet of things realizations," *Recent Advances in Telecommunications, Signals and Systems*, 2013, pp. 63-70.

[4] S. Shanmuganthan, A. Ghobakhilou, P. Sallis, "Sensor data acquisition for climate change modelling," *WSEAS Transactions on Circuits & Systems*, vol. 7, issue 7, 2008, pp. 942-952.

[5] S. Kaisler, F. Armour, J.A. Espinosa, W. Money, "Big Data: Issues and Challenges Moving Forward," *Proceedings of 2013 46th Hawaii International Conference on System Sciences*, pp. 995-1004, 2013.

[6] T. Plunkett, B. Macdonald, B. Nelson, M. Hornick, H. Sun, K. Mohiuddin, D. Harding, G. Mishra, R. Stackowiak, K. Laker, D. Segleau, *Oracle Big Data Handbook*, McGraw-Hill, Oracle press, 2013.

[7] Y. Li, L. Guo, Y. Guo, "An Efficient and Performance-Aware Big Data Storage System," Cloud Computing and Services Science, I.I. Ivanov, M. Sinderen, F. Leymann, T. Shan, eds., *Communications in Computer and Information Science*, Vol. 367, 2013, pp.102-116.

[8] J.S. Veen, B. Waaij, R.J. Meijer, "Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual," *Proceedings of 2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 431-438.

[9] J. Zhang, Y. Yang, L.J. Chen, M. Wang, T. Moscibroda, Z. Jhamg, "Impression Store: Compressive Sensing-based Storage for Big Data Analytics," *Proceeding of HotCloud 2014, USENIX – Advanced Computing Systems Association*, http://research.microsoft.com/apps/pubs/default.aspx?id=219982.

[10] G. Aceto, A. Botta, A. Pescape, C. Westphal, "Efficient Storage and Processing High-Volume Network Monitoring Data," *IEEE Transactions on Network and Service Management*, Vol. 10, Issue 2, 2013, pp. 162-175.

[11] Unixtimestamp.com, http://www.unixtimestamp.com/index.php

[12] As DBMS wars continue, PostgreSQL shows most momentum, http://www.zdnet.com/article/as-dbms-wars-continue-postgresql-shows-most-momentum/

[13] R. Greenwald, R. Stackowiak, *Oracle Essentials: Oracle Database 12c*, O'Reilly, 2013.

[14] P. DuBois, *MySQL Cookbook: Solutions for Database Developers and Administrators*, O'Reilly Media, 3rd ed., 2014.

[15] H. Sug, K. Cha, "An efficient database structure to store sensor data," *Proceedings of the 14th International Conference on Applications of Computer Engineering (ACE '15)*, 2015, pp. 116-119.

[16] NIST, Frequently Asked Questions, http://www.nist.gov/pml/div688/utcnist.cfm

[17] Big and Little Endian, https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html

[18] B.B. Graham, *Using an Accelerometer Sensor to Measure Human hand Motion*, BS and ME thesis, Massachusetts Institute of Technology, 2000.

[19] H. Zhang, G. Chen, B.C. Choi, K. Tan, M.Z. Zhang, "In-Memory Big Data Management and Processing: A Survey," *IEEE Transactions On Knowledge and Data Engineering*, Vol. 27, No. 7, 2015, pp. 1920-1948.

Hyontai Sug received BS degree in computer science and statistics from Busan national university, Korea, in 1983, and MS degree in computer science from Hankuk university of foreign studies, Korea, in 1986, and Ph.D. degree in computer and information science and engineering from university of Florida, USA in 1998. He was a researcher of Agency for Defense Development, Korea from 1986 to 1992, and a full time lecturer of Pusan university of foreign studies, Korea from 1999 to 2001. Currently, he is a professor of Dongseo university, Korea since 2001. His research interests include data mining and database applications.