Reduction Scheme for Sensor-Data Transmission on a Big Data Streaming Platform

Yi-Wei Huang and Sheng-Tzong Cheng
Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan, R.O.C.
stcheng@mail.ncku.edu.tw

***Abstract*—**Recent advances in sensor technology has led to varieties of sensors. Problems facing now is how to exploit and compute these data efficiently. Map Reduce is a programming model to solve these problem. However, it may cause I/O bottleneck.

In-memory computing (IMC) comes up to solve the problems. However, network bandwidth remains a bottleneck. It restricts the speed of receiving the information from the source and dispersing information to each node. To our observation, some data from sensor devices might be similar due to temporal or spatial dependency. Therefore, compression technology could be an effective solution. It replaces data with smaller sizes of streams for improving data utilization.

This study presents an effective reduce transmission scheme on a distributed real-time IMC platform - Spark Streaming. We design and implement a system to provide a high compression ratio in a small batch data from the source. It is expected to reduce data transmission with little delay time in the soft real-time fashion.

Keywords—Big Data; In-memory computing; Spark Streaming; Data compression technique

I. INTRODUCTION

In 2004, Google published a programming model MapReduce [1] for processing and generating large data sets in a distributed fashion over a several machines. Some package has been developed and widely used nowadays that they analyze big data more efficient. For instance, one of the most known package is Apache Hadoop [2]; it designs an interface to implement MapReduce that make people easy to use. Moreover, Hadoop had been widely used in the distributed computing environment.

However, MapReduce uses coarse-grained tasks to do its work, which are too heavyweight for iterative algorithms. Another problem is that MapReduce has no awareness of the total pipeline of Map and Reduce steps, so it can't cache intermediate data in memory for faster performance. Instead, it flushes intermediate data to disk between each step. Overall, these sources of overhead make algorithms requiring many fast steps unacceptably slow.

To address Hadoop iterative I/O limitations, a parallel in-memory computing (IMC) platform [3] is presented. Spark [4] is built on a powerful core of fine-grained, lightweight, and enables applications to reliably store this data in memory. This is the key to Spark's performance, as it allows applications to avoid costly disk accesses. In addition, fine-grained nature means that it can process very small batches of data. Therefore, Spark develops a streaming model to handle data in short time windows and computes them as "mini-batches".

Spark improves disk I/O performance over Hadoop. However, the issue of network bandwidth between gateway and computing platform stills needed to addressed. Wireless Sensor Network [5,15] (WSN) is a wireless network consisting of spatially distributed autonomous devices using sensors to monitor physical or environmental conditions. Nevertheless, most of sensor nodes encounter resource limitation, such as CPU, power, bandwidth, etc. Therefore, collected information of sensor nodes has to be not only small size but also be transmitted fast and easily.

Compression is one of the most efficient method to reduce transmission. Compression technique involves encoding information using fewer bits to represent origin. Data compression is subject to a space–time complexity trade-off. Thus, various compression algorithms, such as Huffman [6], Run-length [7], and Lempel-Ziv-Welch (LZW) [8] are widely used.

To our observation, LZW compression has good effect on the unpredictable data. S-LZW [9] is a modified lossless compression algorithm of LZW. S-LZW splits the uncompressed input stream into fixed size blocks and then compresses each block individually. For each new block, the dictionary is re-initialized by using 256 codes which represent the standard character set. Nevertheless, it may not make good use of spatial or temporal dependency. As the unpredictable and similar pattern features of sensor' data, we adopt LZW to be our base algorithm in this work.

However, LZW does not analyze the incoming text. In some case, LZW may lead to low compression ratio. For instance, the length of dictionary's substring is not restrained, while longer substrings generally don't appear frequently. Especially for the low-repetition-ratio continuous numerical data, it increases the size of the dictionary and contributes little to the improvement of compression ratio. In this work a data-reduction scheme to improve the performance is developed.

II. SYSTEM DESIGN

The main problem we aim to solve is to reduce the transmission of continuous numbers, so that it can limit power

consumption and also send more data in time. This problem can be divided into multiple sub-problems. The first one is that the core algorithm LZW makes good use of duplicated text, but has poor effect on numerical characters.

The second sub-problem is the optimization of the dictionary size due to the storage limit. This means that when dictionary keeps growing, the number of represented bits for codes will also be increased. Therefore, dictionary size is confined in the appropriate range. Obviously, dictionary needs to be re-built to adapt to contents appropriately.

The last sub-problem is about how to handle the balance between with compression ratio and delay time. Streaming data such as sensor data is hard to compress data in the real-time monitor system. Splitting streaming into small batch to compress so as to arrival on time. Nevertheless, small amount of data gets a bad effect on compression, because the dictionary is not full and flush in every batch. On the contrary, if the interval time is long for getting more data in the batch, it will be conflict to real-time requirement. To balance it, we will create mechanism to transmit in the suitable situation.

A. System Architecture

Generally, the structure of monitor real-time system is divided into two sides with few important components, which are shown in Fig. 1. Gray arrows with orders represent data flow; Purple rectangles mean the operations to process; Green rectangles represent the components of communication; Orange cylinders show a limited memory data structure to store data. They are briefly described below:

- Gateway: It represents a remote data source. It consists of preprocessor, mapper, encoder, and communication.
- Real-time parallel computing platform (Spark Streaming): It is almost symmetric to gateway. But it receives stream in every interval time.

B. Preprocess

Aiming at the data fluctuation in a certain range, value is decreased by calculating difference among the data of sample sequence in nearby. Suppose that O[i] is a sample sequence, Vmax is the value that sensor data can detect the highest value. Likewise, Vmin is the lowest value. b is the base value which can be calculated by the equation, $b = (Vmax + Vmin) / 2$. Assume that SubId[i] is the difference sequence which maintains the difference between two adjacent data, that is $SubId[i] = O[i] – O[i-1]$.

C. Mapper

Since all the data can be preprocessed numerically, the algorithm cannot satisfy numeric properly. Thus, numerical characters should be converted into text characters before compression.

During processing of converting, each difference value is replaced by a single character. Numbers from 0 to 25 are respectively replaced by a~z, while numbers from -1 to -25 by A~Y, and other data out of span are all replaced by Z. Z also can be involved in encoding. Out of bound are all represented by single character Z and saved into overflow[k] orderly.

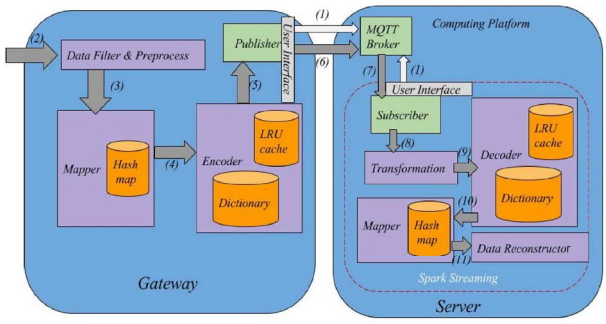

Fig. 1. System architecture.

We consider two different cases for analyzing results. The first case is depicted on the left side of Fig. 2. Without preprocessing, origin data may produce pattern like “A, C, A, C, A” by mapping. As our pre-process, it may get pattern like “c, B, c, B”. It can be seen that after transformation it does not make obvious difference in sequence repetition.

However, consider the situation on the right side of Fig. 2. Obviously, the sequence becomes irrelevant after mapping. They probably produce patterns like “G, F, D, C, A”. In other words, there is no effect to do mapping only. Therefore, we can see that data are necessary to be preprocessed in advance for having high repetition.

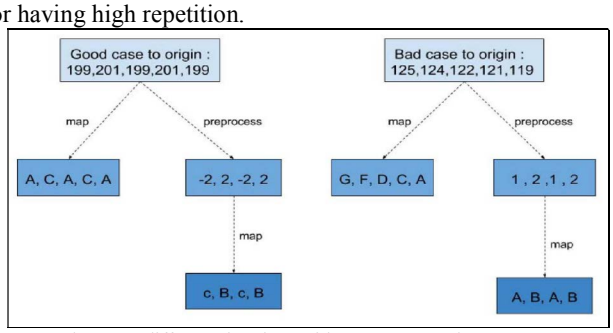

Fig. 2. Analyze two different situations with preprocess and non-preprocess.

D. Encoder

After transforming a sequence of numeric chars, we are able to get a string of characters as our encoding input. Several improving approaches with different purposes are given in this module: All single characters (ASCII) are not required to be put into the dictionary. Secondly, LRU cache is used to store pattern in every character so that dictionary can update contents. Thirdly, data is dynamically transmitted with encoding. Modified steps of encoding are described below:

1. At initial, we do not need so many characters to represent values. Hence, only single character $[a, z] \cup [A, Z]$ will be put in the dictionary.
2. Is the string P+C present in dictionary?
   a. If it is, put P+C to cache. Extend P with C
   b. If not, we will check that if the dictionary size is larger than max dictionary size 2^n
      i. If dictionary is full, we will look contents up in cache to update dictionary.

ii. If not, same as origin

1) Dictionary Tuning

Dictionary may have limit in specific range. In original LZW, dictionary maintains valid contents. As a result, data in different ranges may have bad performance. The reason is that dictionary contents are full of old sequence patterns. When a new data sequence comes out, the new pattern would not be able to be added into dictionary. Replacement rate also drops significantly. In this work, we use the LRU cache for dictionary adjustment. We expect that recent data will appear frequently in the near future, especially for sensor data. Therefore, either match or miss, it has to be put to cache. It means that every string includes sub-strings ever appeared. Thus, update of the cache is a must in every round.

2) Dynamic transmission

Generally, compression technology is rarely used in real-time systems, due to uncertain data. In fact, there are many compression tools such as ZIP, RAR, 7Z, etc. Traditionally, streams are split into small amounts of data and stored into file. If the above compression tools are used, several issues need to be raised. Firstly, it gets a worse compression result if a short time interval is used. Secondly, if file is accessed too frequently, it may cause an I/O bottleneck. Here, we propose a simple approach to solve these problems.

A compression process is introduced. Characters are combined to be checked whether it contains prefix in dictionary or not. LRU cache and patterns are saved in memory for the following combined characters. Only when a no match occurs during the following checks, a code number to the broker is output. As long as a code number is output by compression, it will be transmitted instantly.

E. Communication

This session describes communication in details between gateway and server as shown in Fig. 3. We use the MQTT [11] which is publish-subscribe-based "lightweight" messaging protocol to be the bridge. It runs over TCP/IP, which provides basic network connectivity. First, a subscriber is going to subscribe to a specific topic so that it can receive all the messages on that topic. After that, publisher sends code number to broker, while compression process meets the situation. The broker relays messages to those who subscribe to the topic. Finally, the messages are forwarded to a specific server.

During the compression process, char streams are checked by dictionary until a new pattern is not included. Afterward, code number is put into the queue. As soon as there is code number in queue, it is push to broker immediately.

F. Transformation

As we mention before, Spark Streaming discretizes the streaming data into tiny, sub-second micro-batches. In our cases, we split streaming data into batches in every 3 seconds. Spark Streaming receives code numbers in binary format. In order to do the decoding, it is necessary to split serial binary numbers in every n bits, where n is the length to represent code numbers. Next, discretized stream transforms the format from binary to decimal. Decoder inputs are produced by transformation. When new discretized streams are produced, Spark Streaming can make good use of these features to balance every worker's load.

G. Decoder & Re-Constructor

During decoding, one of the crucial steps is to re-create the original string from the dictionary by using code numbers.

To re-construct a string from an index, we need to traverse the dictionary strings backwards. We follow each successive prefix index until this prefix index is an empty index. We can build the reversed string into a temporary array and then output the byte values by traversing the array backwards. A pseudo code to describe the process of decoding is given below. Note that we set hashmap 2^n as dictionary size, where n is the representative bits of code number. Same as encoding dictionary. "LFI" is the least frequency index, which has less appearance pattern in the output.

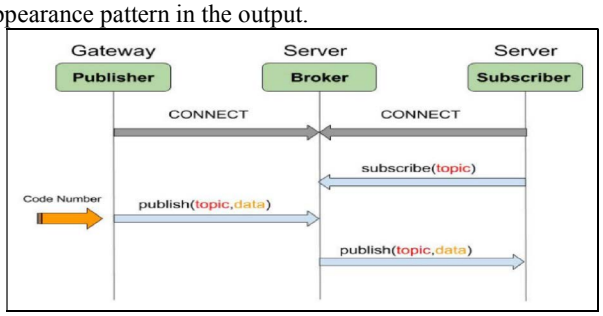

Fig. 3. A workflow between gateway and servers by the MQTT protocol.

Pseudo code:

```
1.  if ( Dictionary size > 2^n) {
2.    LRUcache put ( LFI , W + the first char of W)
3.    Dictionary replace ( LFI , translation of LFI ) with ( LFI , W +
       the first charcter of W)
4.    if ( code == LFI )
5.          output = translation of code
6.    else {
7.          Find LRU cache start with translation of code
8.    Put translation from short to full string in order in LRU cache
9.          output the translation of code
10.     } }
```

At beginning, we put what we have in the dictionary in advance. If dictionary is full, we need to replace the less appeared contents from the dictionary. Two cases are considered. At line 4, it is to check whether the least frequency code in LRU cache is equal to the code number or not. If yes, then the translation of code is output directly. Otherwise, LRU cache is adjusted in order from lines 7 to 9. We find the translation of code's sub-string, and then, put it into cache in order. Finally, append W to the result in every round.

III. EXPERIMENTAL RESULTS

In this session, we conduct several experiments by setting various parameters to show the performance of our scheme.

A. Experiment Environment and Setting

In the experiments, we use a peer-servicing cloud computing platform that contains four homogeneous virtual

machines. The software and hardware specifications of the receiver are detailed in Tables I and II respectively.

TABLE I. RECEIVER ENVIRONMENT

Item	Content
OS	Ubuntu 14.04 Desktop 64bit
Spark	2.0.0
Java	1.8.0_91
Scala	2.11.8

TABLE II. HARDWARE SPECIFICATION OF RECEIVER

Item	Content
CPU	Intel(R) Xeon(R) E5620 @2.40GHz x 2
RAM	8 GB
Hard Drive	80GB

Besides, to simulate the collecting devices in the real world, we use the BeagleBone Black as the hardware of the sender. The hardware specification of the device is given in Table III.

TABLE III. HARDWARE SPECIFICATION OF SENDER

Item	Content
CPU	AM335x 1GHz ARM® Cortex-A8
RAM	512MB
Hard Drive(SD card)	4GB

In addition, data source also has impact on the performance of the experiments. We obtain streaming data from two sources. One is from the PubNub [12] website, which is to simulate real-time data streaming. Data sample can be regarded as sensor data. Another data source is the Internet of Thing (IoT) benchmark for sensor data [13]. The IoT benchmark data sample is shown in Fig. 4. We measure the performance of the rebuild dictionary rebuild for compression. We average the results of 10 partitions of data samples to compute the performance values.

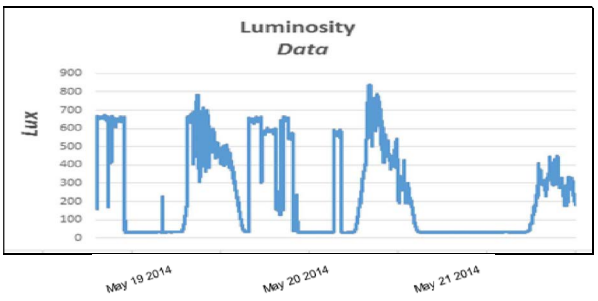

Fig. 4. A benchmark for sensor data: Weekly monitoring of the luminosity of a room in every second.

B. Implementation

Both sender and receiver use Scala [10] and Java as the programming language. First, we introduce the implementation of the sender. Sender receives a streaming data from PubNub. Then it begins to handle these data. Filter the unnecessary parts of streaming data. Two different modes are used to measure performance. In this first mode, the streaming data is processed from PubNub directly. In the other mode, the streaming data is stored into file first.

We implement our system into the Spark Streaming platform. Nevertheless, our system can be integrated with other parallel computing platforms as well. In order to receive data as stream, Spark Streaming allow users to choose the interface of data source like kafkaStream [13]. Most importantly, Spark Streaming also provides an API to customize the data receiving interface. In our implementation, we utilize the API called *Receiver* to integrate our scheme into Spark platform.

C. Experimental Results

In order to evaluate our proposed scheme, several evaluation scenarios are considered and elaborated in this section. We explain the evaluation scenarios and assumptions here. There are four factors that have impact on compression. These parameters are data source length, data source pattern repetition rate, dictionary code length, and methods of rebuilding dictionary. In the following experiments, one factor will be studied while the other three factors remain unchanged. We store specific sizes of streaming data into files. And then, we use threads to simulate intervals of sensor data. Notably, we regard dictionary match rate as the repetition of the represented data source patterns.

Compression ratio, which is the most important indicator in our experiments, is depicted in (1). In general, we expect to get ratio that is bigger than one. In other words, data are going to be smaller to be transmitted. In addition, we use (2) as the second evaluation criteria. Literally, it means how much space we save in the sender.

$$\text{Compress Ratio} = O.D.S \,/\, C.D.S \quad (1)$$

$$\text{Space Saving Percentage} = (O.D.S - C.D.S)/O.D.S \times 100\% \quad (2)$$

* O.D.S = original data size
* C.D.S = data size after compression

1) Scheme Performance

In the first experiment, we evaluate performance between the original LZW and our scheme, PLZW, which stands for "LZW with preprocessing". As we observe in Fig. 5, it is easy to see that PLZW's compression ratio is higher than that of the LZW. For instance, consider the case of 10-bit code number. We can see that PLZW is higher significantly than LZW in a kilobytes of data size; likewise, for 4, 16, 64, and 256 kilobytes of data size, our scheme has better performance than the original LZW as well. For the cases of 11, 12, and 13 bits, the results are similar to the situations in the case of 10 bits. Thus, our scheme improved performance over LZW.

More details are illustrated in Fig. 6. At first, PLZW gets 34 percent of improvement. But, it drops while data size gets

larger. The reason for the drop is that dictionary gets full. PLZW is close to LZW while data size becomes larger. At least, it reaches 24 percent of improvement over all.

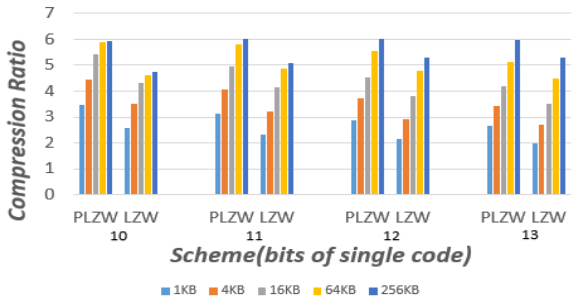

Fig. 5. Performance of the PLZW versus that of the LZW in different data sizes and various lengths of code numbers. .

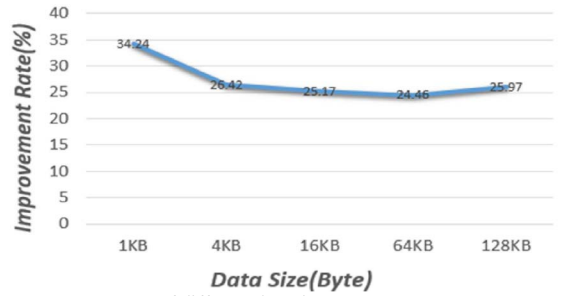

Fig. 6. Improvement rate of different data size.

2) Data Pattern Repetition Rate

We define the data pattern repetition rate based on the match rate in dictionary. The match rate conveys the data patterns appeared before. We describe how it affects compression effect in Fig. 7.

Obviously, there is a strong relationship between compression ratio and pattern repetition rate. For the case of 20.27 percent of data pattern repetition, the system reaches the lowest compression ratio. However, as pattern repetition rate increases, compression ratio is much better. Even though they cannot be generalized in a formula, we still are able to realize that the higher data pattern repetition rate, the better performance it obtains.

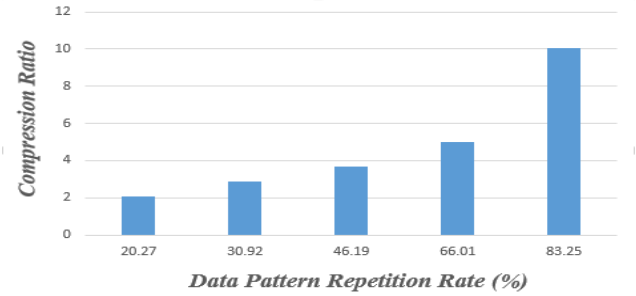

Fig. 7. Compression ratio of data pattern repetition rate.

3) Dictionary Code Length

One factor to influence compression ratio is dictionary code length. We need to balance the tradeoff between bit counts to present and space to save data pattern. It depends on different applications. Our experimental result is shown in Fig. 8. The case of 10-bit length has a better performance than that of 13-bit length for small amount of data. But, the case of 13-bit length has better performance when the data size is greater than 256 kilobytes. For the 10-bit case, dictionary has same contents as that in 13-bit dictionary until it is full. However, 13-bit code length needs more bits than 10-bit code length to represent the string. For the case of 13 bits, it needs to create 2^{13} indices for representing more strings.

In fact, we observe that compress ratio does not make significant differences in big data size. Compression ratio differences are in the range of $\pm 3\%$ to our test. In conclusion, we cannot ensure which code length is the best for compression ratio. It depends on data source. Mostly, we have a best effect in the case of 13-bit code length in our experiment.

4) Dictionary Rebuild

Dictionary rebuild is also one of factors to affect performance. In the data source, data samples frequently appear in different ranges. In this experiment, we have already preprocessed data samples in advance.

Flushing [14] is to clean up dictionary and then rebuild it. As the result of Table IV shows, we find that the flushing of LRU cache has a better performance. After tracing dictionary contents and checking with data distribution, the key point is that the dictionary is full with old pattern strings. As time goes by, old patterns are not reused. It justifies the flushing. Moreover, flushing is able to collect data in a period time and re-create dictionary in next cycle. Nevertheless, it cannot keep the high frequency pattern. On the other hand, LRU cache keeps some similar patterns between time intervals. These patterns will constantly accumulate in intervals. Therefore, LRU has better compression ratio than flushing.

Fig. 8. Compression ratio of different code lengths.

TABLE IV. PERFORMANCE OF DIFFERENT REBUILDING DICTIONARY APPROACH

	Compress Ratio	Space Saving
Original	5	0.8
Flushing	10.28	0.9
LRU cache	11.17	0.91

5) Output Time Delay

We are going to evaluate delay time of PLZW. It is an

important measure to study the trade-off between delay time and performance. We define output time delay as the time from collecting data, preprocessing, compressing, to actually outputting a single code. It is illustrated in (3).

$$\text{Average Sensor Data Delay Time: } \frac{\sum_{i=1}^{n} k*(Ti+T)}{n} \quad (3)$$

Parameters are defined as follows:
n: Counts of total sensor data
k: k^{th} match, k will be reset to 1 if dictionary is not match
Ti: i^{th} sensor data execution time (including preprocess, map, and compression operations)
T: Sensor data time interval = 0.5 s

Equation (3) has four parameters. The most changeable one is the parameter k, which is unpredictable. As we recall dynamic transmission, compressed data will only be output until the string does not match with dictionary. As long as data matches, we need to wait for the arrival of next data. It improves compression ratio but delays time slightly. The relationship between compression ratio and output time delay is shown in Fig. 9.

We are able to observe that output time delay grows up slowly when the input data size is greater than 16 kilobytes. While dictionary is full, bit counts of string repetition are similar as before. As a result, we obtain high compression ratio but only close to 0.96 seconds of output time delay.

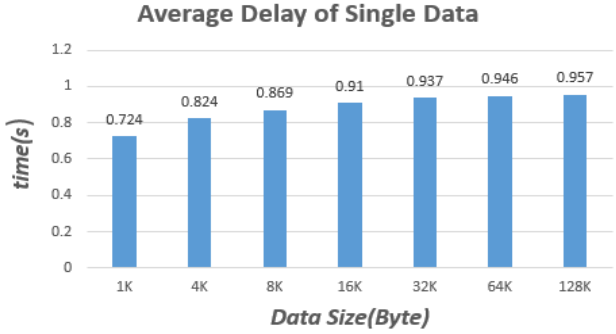

Fig. 9. Trade-off between compression effect and delay of sending data.

IV. Conclusion

In this paper, we proposed a reduction data transmission scheme on parallel streaming computing platform – Spark Streaming to promote data utilization. LZW algorithm inspires us to implement a compression technique for reducing data transmission. There are many classic compression tools, such as ZIP, RAR, 7z, etc. However, these tools were not suitable for streaming data in real-time. We implemented our scheme PLZW on Spark Streaming to solve the problem.

We showed how to apply this scheme between sender and receiver. We were able to observe that PLZW gets better compression ratio than original LZW and also has high compression effect with little delay time. According to the experiments, we can summarize that our scheme works well in the following cases:

- Data is in numerical format,
- Data source length is long, and
- More similar sequences appear periodically.

We used two different data sources to evaluate our scheme. Experimental results showed that it improved the compression effect in both situations.

References

[1] "Google MapReduce," 2011, http://research.google.com/archive/mapreduce.html [Jun. 30, 2017].

[2] "Hadoop," 2014, http://hadoop.apache.org/ [Jun. 30, 2017].

[3] Jiang, Tao, et al. "Understanding the behavior of in-memory computing workloads." Workload Characterization (IISWC), 2014 IEEE International Symposium on. IEEE, 2014.

[4] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenkr, and I. Stoica. "Spark: cluster computing with working sets," *HotCloud*, 2010.

[5] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cyirci, "A Survey on Sensor Networks," *IEEE Communications Magazine*, vol. 40, no. 8, Aug. 2002, pp.102 -114.

[6] Knuth, Donald E. "Dynamic huffman coding." *Journal of algorithms* 6.2 (1985): 163-180.

[7] Hauck, Edward L. "Data compression using run length encoding and statistical encoding." U.S. Patent No. 4,626,829. 2 Dec. 1986.

[8] Nelson, Mark R. "LZW data compression." *Dr. Dobb's Journal* 14.10 (1989): 29-36.

[9] Sadler, Christopher M., and Margaret Martonosi. "Data compression algorithms for energy-constrained devices in delay tolerant networks." *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006.

[10] "The Scala programming language," 2016, http://www.scala-lang.org [Jun. 30, 2017].

[11] Hunkeler, Urs, Hong Linh Truong, and Andy Stanford-Clark. "MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks." *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008.

[12] "Pubnub Sensor Network," 2010, https://www.pubnub.com/developers/realtime-data-streams/sensor-network/ [July. 05, 2017].

[13] "Benchmark IoT sensor data models," 2014, https://github.com/assaad/BenchmarkIoT/tree/master/DataSets [July. 05, 2017]. "

[14] Raghuwanshi, B.S., Jain, S. Chawda, D. and Varma, B. 2009. "New dynamic approach for LZW data compression". IJCNS Vol. 1, No. 1 (October), 22-2.

[15] Srisooksai T., Keamarungsi, K., Lamsrichan P., and Araki, K. 2012. "Practical data compression in wireless sensor netowkrs: A survey,". JNCA 35: 37-59.