Fast Deduplication Data Transmission Scheme on a Big Data Real-Time Platform

Sheng-Tzong Cheng, Jian-Ting Chen and Yin-Chun Chen
Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan
{stevecheng1688, eytu0233, darkerduck}@gmail.com

Keywords: Big Data, Deduplication, In-Memory Computing, Spark.

Abstract: In this information era, it is difficult to exploit and compute high-amount data efficiently. Today, it is inadequate to use MapReduce to handle more data in less time let alone real time. Hence, In-memory Computing (IMC) was introduced to solve the problem of Hadoop MapReduce. IMC, as its literal meaning, exploits computing in memory to tackle the cost problem which Hadoop undue access data to disk caused and can be distributed to perform iterative operations. However, IMC distributed computing still cannot get rid of a bottleneck, that is, network bandwidth. It restricts the speed of receiving the information from the source and dispersing information to each node. According to observation, some data from sensor devices might be duplicate due to time or space dependence. Therefore, deduplication technology would be a good solution. The technique for eliminating duplicated data is capable of improving data utilization. This study presents a distributed real-time IMC platform -- "Spark Streaming" optimization. It uses deduplication technology to eliminate the possible duplicate blocks from source. It is expected to reduce redundant data transmission and improve the throughput of Spark Streaming.

1 INTRODUCTION

In recent years, with the development of Internet and prevalence of mobile devices, a very huge amount of data was generated daily. To be able to carry out some operations on larger and more complex data now, techniques for Big Data were presented. In 2004, Google released a programming model MapReduce (Dean, 2008) for processing and generating large data sets with a parallel, distributed algorithm. Packages have been developed and widely used nowadays. They can make big-data analysis more efficient. For instance, one of the mostly used packages is Hadoop (Shvachko, 2010). It provides an interface to implement MapReduce that allows people use it more easily.

Hadoop MapReduce adapts coarse-grained tasks to do its work. These tasks are very heavyweight for iterative algorithms. Another problem is that MapReduce has no awareness of the total pipeline of Map plus Reduce steps. Therefore, it cannot cache intermediate data in memory for faster performance. This is because it uses a small circular buffer (default 100MB) to cache intermediate data, and it flushes intermediate data to disk between each step and when 80% of the circular buffer space is occupied. Combined these overhead costs, it make some algorithms that require fast steps unacceptably slow. For example, many machine-learning algorithms were required to work iteratively. Algorithms like training a recommendation engine or neural networks and finding natural clusters in data are typically iterative algorithms. In addition, if you want to get a real-time result from the trained model or wish to monitor program logs to detect failures in seconds, you will need for computation streaming models that simplify MapReduce offline processing. Obviously, you want the steps in these kinds of algorithms to be as fast and lightweight as possible.

To implement iterative, interactive and streaming computing, a parallel in-memory computing platform, Spark (Zaharia, 2010), was presented. Spark is built on a powerful core of fine-grained, lightweight, and abstract operations by which the developers previously had to write themselves. Spark is lightweight and easy to build iterative algorithms with good performance as scale. The flexibility and support for iterations also allow Spark to handle event stream processing in a clever way. Originally, Spark was designed to become a batch mode tool, like MapReduce. However, its fine-grained nature makes possible that it can process very small batches of data.

Therefore, Spark developed a streaming model to handle data in short time windows and compute each of them as "mini-batch".

Network bandwidth is another bottleneck that we wish to resolve. Bandwidth shortage is not from its architecture but from the gateway between sensors and computing platform (Akyildiz, 2002). The bridge that collects data from sensors and transmits data to server is performed by one or more gateways. Their bandwidth is often low because of the wireless network environment. Our proposal is to utilize these transmitted data fully for low-latency processing applications. In order to maintain or even improve the throughput of computing platform, we adopt the real-time parallel computing platform based on data deduplication technology. It allows the efficient utilization of network resources to improve throughput.

Data deduplication is a specialized data compression technique for eliminating duplicated data. This technique is used to improve storage utilization and can also be applied to network data transmission to reduce the amount of bytes that must be sent. One of the most common forms of data deduplication implementation works by comparing chunks of data to detect duplicates. Block deduplication looks within a file and saves distant blocks. Each chunk of data is processed using a hash algorithm such as MD5 (Rivest, 1992) or SHA-1 (Eastlake, 2001). This process generates a unique number for each piece which is then stored in an index. If a file is updated, only the changed data is saved. For instance, Dropbox and Google Drive are also cloud file synchronization software. Both of them use data deduplication technique to reduce the cost of storage and transmission between client and server. However, unlike those cloud storages, there is no similar file between gateway and computing server. Hence, we propose a data structure to keep those duplicated part of data and reuse them. This is the part where our work is different from those cloud storages. In our work, the data stream from sensors can be regarded as an extension of a file. In other words, the data stream is also divided into blocks to identify which blocks are redundant. So data deduplication has quite potentials to resolve the problem of bandwidth inadequate.

In this study, we propose that the deduplication scheme reduces the requirement of bandwidth and improves throughput on real-time parallel computing platform. Interestingly, the data from sensors has quite duplicated part that can be eliminated. This is the tradeoff between processing speed and network bandwidth. We sacrifice some CPU efficacy of gateways and computing platform to exchange more efficient utilization of network bandwidth. In brief, we applied data deduplication technique completely to improve the data re-use rate on distributed computing system like Spark.

2 DATA DEDUPLICATION TRANSMISSION SCHEME

In this section, we elaborate on the details of our system design. We first clarify our problem in Section 2.1 and then the implementations and the parameter definition are listed in the following sections. In Section 2.2, we outline our system overview and provide a series steps explanation then formulate our bandwidth saving model. In Section 2.3, we describe how to choose block fingerprint and give a benchmark for hash functions to compare to select the option. In Section 2.4, we give some concept to guide users how to implement the data chunk preprocess model.

2.1 Problem Description

The main problem we want to resolve is to reduce the duplicated data delivery so that it can send more data in limited time. This problem can be divided into several sub-problems. The first one is that how to chunk data so that we can make the set of data blocks smaller. In other words, when the repetition rate of data blocks is higher, the bandwidth saving becomes more. However, if remote does not have similar data, these chunking methods would not effective.

The second problem is that how sender decides whether this data block has received or not. With Rsync algorithm (Tridgell, 1998), it uses a pair of weak and strong checksums for a data block to enable sender to check whether the blocks have not been modified or not. This gives a good inspiration to solve it. In order to find the same data block, Rsync uses strong checksum to achieve it. So, hash function is the solution that is able to digest block into a fingerprint. Block fingerprint can represent the contents of the block and utilize less space, this is we want. However, MD5 used in Rsync is not the best choice for our work. This will be analyzed in Section 2.3.

2.2 Scheme Overview

Before describing solutions of these sub-problems, we assemble these notions into a data block deduplication scheme. We believe this scheme helps

us to reduce bandwidth utilization between gateway and computing platform. Figure 1 shows the scheme overview that illustrates how we implement it.

Here we explain the meaning of control flow and data flow. In Figure 1, the two biggest dotted boxes represent a remote data source (i.e. Gateway) and a real-time parallel computing platform (i.e. Spark) respectively. The rectangles represent data handlers that compute these data blocks like Data Block Preprocessor and Block Fingerprint Generator and communication interfaces that deliver and receive control information and raw data. The cylinder represents a limited memory data structure to store data. In addition, all arrows are data flows that illustrate how these data or blocks flow in our scheme. The arrows around dotted box are related with metadata that is used to control data transmission. All steps in this scheme will be described as follow.

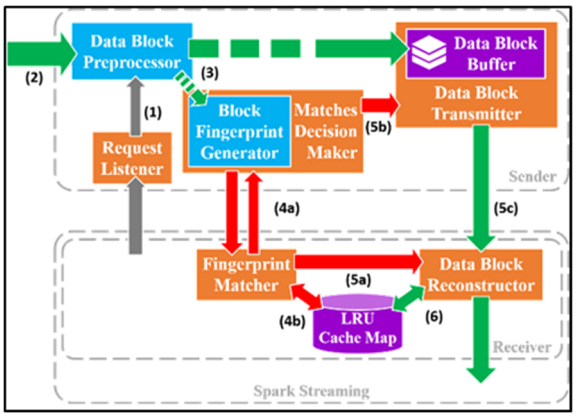

Figure 1: Scheme overview.

Step 1: Once the receiver triggers Request Listener, the listener accepts the connection and notifies Data Block Preprocessor to handle raw data stream.

Step 2: In Data Block Preprocessor, no matter if the source of raw data stream is from a reliable disk or sensor, it is spitted into data blocks. This preprocess for raw data is so important that it influences the whole data block deduplication scheme. The detailed explanation and implementation are presented in Section 2.4.

Step 3: These data blocks are pushed into Block Fingerprint Generator and Data Block Buffer. In this phase sender prepares the block fingerprints and data blocks that are ready to send. The Data Block Buffer has a memory space to cache these data blocks from preprocessor and records the sequences of blocks that will be used in data block transmitter. Its data structure is a first in, first out(FIFO) Queue. Besides, this process needs Block Fingerprint Generator to generate hash value for each block, the detail implementation is showed in Section 2.3.

Step 4: The Matches Decision Maker will exchange metadata with Fingerprint Matcher in arrow (4a). First, the Decision Maker sends the fingerprints that belong to blocks stored in buffer to Fingerprint Matcher. The matches that contain the information whether or not blocks have been sent are returned to Matches Decision Maker by Fingerprint Matcher. In arrow (4b), Fingerprint Matcher uses these fingerprints as key to ask the LRU cache map to find out if this block has received or not. It uses a Boolean array as matches, and the Boolean array retains the order information which Data Block Transmitter needs. Before returning matches, we need to do an additional checking for fingerprints. Because some duplicated data blocks are too close to each other, the results of matches from LRU Cache Map do not identify these duplicated data blocks. Before the data blocks are stored into LRU Cache Map in Step 6, these blocks are not in LRU Cache Map. This situation makes some blocks identified as unique. Hence, the additional checking is required.

Step 5: In Step 4, the metadata has been exchanged between sender and receiver, and this said that sender knows which data blocks do not need to retransmit while the receiver knows how to reconstruct these blocks as well. For arrow (5a), the sequence of fingerprints and match information notifies Data Block Reconstructor about how to receive next data block. For example, the sequence is like [(f_1, F), (f_2, T) , (f_3, T), …] where f presents fingerprint, F is false, and T is true. At that time, arrow (5b) also indicates the result of matches as a sequence like [F, T, T, …] to Data Block Transmitter. After the metadata notifies the data communication interfaces, it begins to pass blocks of raw data sequentially. This is the reason why data block buffer is a first-in, first out (FIFO) queue. It is used to correspond matches sequence. Figure 2 illustrates the data flow of arrows (5a) and (5b) across network. This data flow completely shows how this scheme saves bandwidth. We can observe that some blocks are ignored to transmit on network, and this is reason why our scheme works well. In addition, we can also use some compression algorithm like gzip (Levine, 2012) to compress data and further reduce bandwidth utilization. Moreover, to prevent blocks from waiting for metadata, it is suggested to set a timer. When the timer expires, send must transmit data without control. This mechanism is to prevent receiver from waiting data too long.

Step 6: This is the last step for this scheme. The Data Block Reconstructor arranges received data blocks and matches and puts these received blocks to LRU cache map, which stores the pairs of fingerprint and block data with a limit size. This is the point that makes reduplicated data utilization more efficient. Because Spark requires much memory, the amount of memory for this scheme to utilize is limited. Hence, the LRU cache map is implemented with a least recently used least recently used (LRU) Java hash map data structure to reduce the influence of data reusing. To prevent from occupying excessive memory in receiver, we present the analysis about the parameter for the data structure in the next section. Finally, the duplicated data blocks could be ignored and not required to store again. After storing these blocks which are not received in LRU cache map, receiver uses *store* API to notify Spark how many blocks have received and need to compute with the sequence of pairs of a fingerprint and a match Boolean from matcher.

Suppose that sender sends a set of h-byte hashes as fingerprints to receiver, and that receiver uses these hashes to check for match of each data block. Suppose that the k-th block size is b_k bytes and the size of a match is 1 bit (equal to $1/8$ byte) as a constant symbol α. In addition, we also suppose the match of the k-th block is m_k. Finally, if there is n blocks handled in a time interval t, it will give a bandwidth-saving model, thus the bandwidth this scheme saves in terms of bytes is

$$\sum_{k=1}^{n} r_k\left(b_k - (h+\alpha)\right). \quad (1)$$

Note that r_k is $\begin{cases} 1, m_k \text{ is } true \\ -1, m_k \text{ is } false \end{cases}$. The symbol r_k means that if the block has been transmitted, this scheme will save bandwidth, or it will increase additional costs. Equation (1) shows that the reduction of network utilization by using this scheme is probably low because of the low repetitive rate, and the worse thing is probably a negative value. The repetition rate is a pivotal factor and it is expressed as

$$\frac{\sum_{k=1}^{n} c_k}{n}. \quad (2)$$

Note that c_k is $\begin{cases} 1, m_k \text{ is } true \\ 0, m_k \text{ is } false \end{cases}$. The repetition rate affects reduction of network utilization a lot, and it is a positive correlation between both of them. So, in order to gain the highest benefit for our work how to chunk raw data into most of identical blocks becomes the most crucial issue. In addition, size of data fingerprint and size of a data block are also factors. Thus, further analysis is required. We based these two formulas to experiment with various parameters in Section 3.

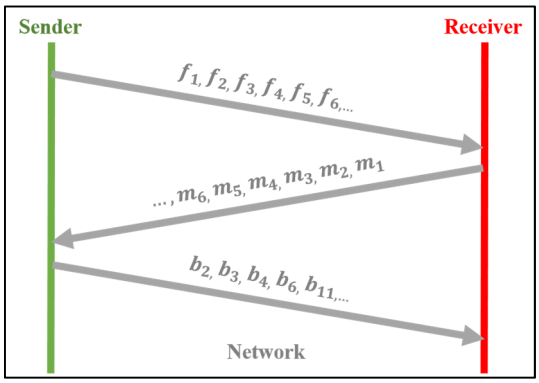

Figure 2: Data flow of Step 5.

2.3 Block Fingerprint

After chunking data blocks, it needs to further process these blocks. To identify the identical blocks, the fingerprints of their content are required. In Rsync, it uses two different types of checksum, weak checksum and strong checksum. The weak checksum used in Rsync is a modified blocks checker because of its fast process speed, and Rsync uses the rolling checksum based on Mark Alder's adler-32 checksum (Deutsch, 1996) as implementation. However, the weak checksum has no ability of determining which blocks are the same owing to its high hash collision probability, and therefore weak checksum is not our option. The another strong checksum used in Rsync is MD5. MD5 is a cryptographic hash function producing a 128-bit hash value equal to 16 bytes. Unlike rolling checksum, MD5 is able to identify the blocks of the same content, it might be a choice.

We can observe the factor that fingerprint influence is parameter h. This makes sense that once h is smaller, the benefit for this scheme is better. In other words, it can use less information to represent the data blocks. Hence, 128-bit hash value is not so ideal for our work. It needs to find a smaller size of hash function to substitute it with a premise, and the hash function can determine the same blocks as well.

The next property considered in Block Fingerprint Generator is fast process speed. Although the concept of this scheme is to utilize the compute resource of remote node and achieve a benefit for bandwidth saving, it could not lead to another bottleneck. So, the throughput of the hash function used in Fingerprint Generator must be as fast as possible. Obviously, MD5 has been ruled out in our implementation on account of its slow speed. It means it requires a more suitable hash function.

In summary, the implementation of Fingerprint Generator must have three properties, ability of identification, smaller size and fast process speed. The solution which we choose is xxHash (Collet, 2016). xxHash is an extremely fast non-cryptographic hash algorithm, working at speeds close to RAM limits. It is widely used by many software like ArangoDB, LZ4, TeamViewer, etc. Moreover, it successfully completes the SMHasher (Appleby, 2012) test suite which evaluates collision, dispersion and randomness qualities of hash functions.

Although xxHash is powerful and successfully completes the SMHasher test suite, its 32-bit version still has collision. Here we provide a simple test to verify 32-bit xxHash collision rate with a real-world data. The data is from a GPS trajectory dataset (Yuan, 2011) that contains one-week trajectories of 10357 taxis. The sum of points in this dataset is about 15 million and the total distance of the trajectories reaches 9 million kilometers. We use some data reprocessing to filter the raw data and gather them into a handled dataset. The file size of the handled dataset is about 410 MB. Figure 3 illustrates the repetition rates of the handled dataset with two hash functions SHA-1 and xxHash.

We can see from Figure 3 that the repetition rate of the first row is undoubted by using 160-bit SHA-1 function. We find that the repetition rate of xxHash32 is higher than SHA-1 about 0.1% in field Hash Map which does not have any restriction. This 0.1% difference means that the 32-bit xxHash occurs collision in this simple test. In contrast, xxHash64 has the same repetition rate with SHA-1. The collision rate of xxHash64 is lower than xxHash32, but xxHash64 also has higher cost because its longer hash value size for our scheme. Even the xxHash32 has the risk of collision, we still prone for it. There are two reasons that mitigate the influence of collision. The first one is about its probability; hence, we consider that 0.1% deviation could not affect the result a lot. On the other hand, this error can be handled in computing phase by some operations. Another one is the implementation of hash map is LRU hash map, so the limitation not only prevents to occupy excessive memory but also reduces the occurrence of collision with an extra cost of having the repetition rate a little lower. Because after discarding the least recently used data blocks, the occurrences of collision have high possibility to eliminate. In summary, we said the defect of xxHash32 used in this scheme is ignorable.

The memory size of LRU Cache Map is based on two factors, one is the size of hash value, and another one is its parameter. In Table 1, it shows that the standard hash map can store all fingerprints and data block, but it leads to out of memory. That is why we pick LRU hash map. The average size of records in the dataset is about 25 bytes. It shows xxHash32 has the smallest memory size for the LRU hash map.

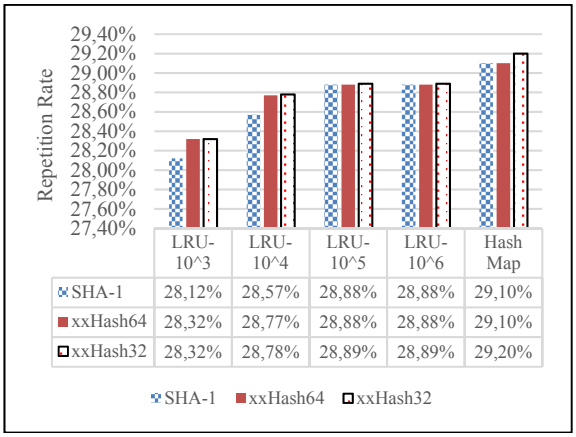

Figure 3: Repetition rate and LRU Cache Map analysis.

Table 1: Memory size of each data structure.

	Hash Map	LRU-10^3	LRU-10^4	LRU-10^5	LRU-10^6
SHA-1	OOM	50KB	500KB	5MB	50MB
xxHash64	OOM	35KB	350KB	3.5MB	35MB
xxHash32	OOM	30KB	300KB	3MB	30MB

2.4 Data Chunk Preprocess

In file synchronization systems, most of the time, the content difference between local node and remote node is slightly small. So, the methods of file synchronization are focus on how to find out the different parts between two files. Note that the data generated by sensors in a time interval comes in record by record. For instance, consider the GPS dataset. The average size of the record in the GPS dataset is about 25 bytes. On the contrary, the parameter s in Rsync is at least 300 bytes, let alone the average block size in LBFS is 8KB. Therefore, a fine-grained chunking method is essential for our work.

The data block in our scheme is like a record that sensor generates in a time interval. Spatial dependence leads to a neighbour cluster of sensors to detect similar values; time dependence leads to each record from the same sensor to measure smooth data. Therefore, we split raw data and obtain duplicated records as possible as it can be.

In sensors network, a cluster head collects the real-time data from many sensors. There is so much noise that causes low probability to distinguish the duplicated part. To identify the difference, we require

some measure to filter out noise. Take the GPS dataset as an example. Figure 4 shows the mapping from original trajectory dataset to the handled dataset. We observe that the original dataset has four fields which are separated by commas. These fields are taxi id, data time, longitude, and latitude. For the field of data time we call it the dynamic field, because it always changes so that it causes our scheme gain benefit with difficulty. After eliminating the dynamic field, the handled dataset exhibits several duplicated records set.

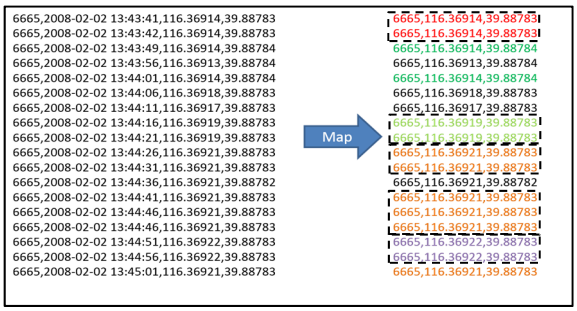

Figure 4: Filter out the dynamic field.

According to this case, the average size of records is obviously smaller than original. According to the parameter b_k in Equation (1), this causes some loss of bandwidth savings. So, how to keep original information of raw data with data preprocess is considered as a challenging problem. The balance between data integrity and data repetition rate depends on how users identify the dynamic fields.

Dynamic fields are often commonly found for data from sensors. The occurrences of dynamic fields are because the sensors are too sensitive or a lot of other factors. To face these situations, some methods from data mining can be used to preprocess data. One of methods is data generalization, for instance, there are three sensors in a room, and these sensors are able to sense someone entering this room with a distance by infrared ray. Suppose an application only cares when the person enters and leaves, the distance data is applied by concept hierarchy to map a value which shows if the person is in the room or not. It replaces the relatively dynamic distance value with a Boolean. Hence, the handled data from infrared ray sensors has higher probability to have duplicated part. Other methods also have similar idea that makes data general. Fuzzy sets (Zadeh, 1965) and fuzzy logic can also be used to process raw data. If we use fuzzy logic to classify continuous value, the data will be more general and generate duplicated part. Our another concern is complexity of the method. Because the gateway has limit processing resource, the complexity of the method must be low. In summary, we present a data deduplication scheme which eliminates the duplicated data that does not need to be retransmitted to improve the effectivity of data utilization in low bandwidth network environment.

3 IMPLEMENTATION

In this paper, we take Spark as the platform and introduce the implementation of the data deduplication scheme on the sender and the receiver sides. We also conduct several experiments with various parameters to show the significance of our scheme.

3.1 Experiment Environment and Setting

We use a peer-servicing cloud computing platform that contains eight homogeneous virtual machines. The software and hardware specifications of the receiver are detailed in Tables 2 and 3 respectively.

Table 2: Receiver environment.

Item	Content
OS	Ubuntu 15.10 Desktop 64bit
Spark	2.0.0
Java	1.7.0_101
Scala	2.11.8
Maven	3.3.9

Table 3: Hardware specification of receiver.

Item	Content
CPU	Intel(R) Xeon(R) E5620 @2.40GHz x 2
RAM	8 GB
Hard Drive	80GB
Network Bandwidth	1Gbps
Maven	3.3.9

Besides, to simulate the gateway used in the real world, we use raspberry pi 2 as the sender. The hardware and software specification for the raspberry pi is detailed in Tables 4 and 5 respectively.

Table 4: Sender environment.

Item	Content
OS	Raspbian-32bit
Java	1.8.0_65
Scala	2.9.2
Linux Kernel	4.1.19

Table 5: Hardware specification of sender.

Item	Content
CPU	Broadcom BCM2836 ARMv7 Quad Core Processor @900 MHz
RAM	1 GB
Hard Drive(SD card)	32GB
Network Bandwidth	1Gbps

3.2 Implementations

Both of the sender and receiver use Scala (Odersky, 2007) as the programing language. First we introduce the implementation of the sender. Sender accepts a TCP connection as Request Listener. Then it begins to read experimental data from SD card in raspberry pi, and pushes these data into Matches Decision Maker and Data Block Buffer. The Matches Decision Maker computes each fingerprint for each data block as Block Fingerprint Generator with xxHash32. These fingerprints are sent to receiver and then sender waits for the matches. The Data Block Buffer is implemented by a Java API, ArrayBlockingQueue class, which is thread safe and provides synchronous data access. Data Block Transmitter receives the responses of matches and decides which block needs to be transmitted to receiver.

On the receiver side, the implementation of receiver is based on the Spark platform. Nevertheless, our scheme can work well on other parallel computing platforms too. The original Spark only receives data from a reliable data storage such as storage, database, and HDFS. In order to receive data as stream, Spark Streaming lets user choose the interface of data source. Spark Streaming provides these interfaces like fileStream, socketStream, kafkaStream (Kreps, 2011), twitterStream, etc. Most importantly, Spark Streaming also provides an API to customize the data receiving interface. An API call *Receiver* is the place that allows us to implement our approach into Spark platform. Its native Receiver API implements simple operations. These operations include opening a socket, receiving each line from the socket, putting them into Spark to compute with *Store* API. Therefore, we augment the Spark with the data deduplication scheme to accept streaming data. Table 7 presents the parameters of the first experiment.

Before receiving data blocks, the scheme needs to exchange metadata between sender and receiver. Then, the data blocks that are required to receive must be determined. A TCP connection is used as a trigger to notify sender to start the whole process. The customized receiver gets fingerprints from the sender. Fingerprint Matcher uses these fingerprints to query LRU Cache Map whether the data block is received or not. The LRU Cache Map is implemented by LRU hash map described in Section 2.3. Another TCP connection returns the result of matches to sender. After metadata exchanging, the native line reading process is revised to Data Block Reconstructor. With metadata, Data Block Reconstructor rebuilds data from two sources: sender and LRU hash map. If the block was received before, it retrieves the data block from LRU hash map with the fingerprint of the data block; otherwise, it requests the sender a new data block by the third TCP connection and also puts the new data block into LRU hash map. Finally, Data Block Reconstructor reorganizes raw data as sequence and uses *Store* API to feed these data blocks to Spark Streaming to compute in parallel and batch.

4 EXPERIMENTAL RESULTS

The application of Word Count was tested to evaluate the performance of the scheme on Spark Streaming. It performs a sliding window count over 5 seconds. Table 6 presents the experimental configurations in Spark Streaming.

4.1 Empirical Result

In order to evaluate our proposed scheme, several evaluation scenarios are defined and conducted in this section. First, we explain the evaluation scenarios and assumptions. Equations (1) and (2) indicate the three parameters that have impact on the performance of our work. These parameters are the data block length, the length of fingerprint, and the repetition rate. Moreover, another environment parameter that is also an important factor for our scheme is bandwidth. So, the following experiments will be conducted to adjust one single parameter and fix the other three parameters. Consider a streaming application that computes data continuously. It generates a result in a specified time interval (5 seconds). We sample the result with 120 time intervals (600 seconds). We use a probability value to simulate the repetition rate of test data.

Table 6: Configuration in Spark Streaming.

The memory size of the driver	4GB
The memory size of each executor	1GB
The number of executors	8

4.1.1 Length of Data Block

The first experiment we study is the impact of the length of data block, namely b_k in Equation (1), on

the system throughput. The length of data block in this experiment is an average value in terms of bytes. Other parameters are shown in Table 7. The experimental results are given in Figure 5.

Table 7: Parameters of the first experiment.

Bandwidth	1Mbps
Repetition Rate	25%
Length of Fingerprint	32 bits
Limitation of LRU Cache Map	1,000,000

We see from Figure 5 that the throughput for the original scheme in Spark is almost the same for all lengths of data block. However, with our deduplication scheme, the throughput increases as the length of data block becomes bigger. These results conform to Equation (1). It means that if a fingerprint can present data blocks with a bigger size, it saves more bandwidth when a data block is repeated. The throughput improvement reaches the top when the length of data block is 30. When a data block with more than 30 bytes is used, the system throughput does not get higher.

4.1.2 Repetition Rate

The most crucial factor in our scheme is the repetition rate. In Section 2.2, the repetition rate is expressed in Equation (2). In this experiment, we focus on how the throughput goes with the changing of repetition rate. Parameters for this experiment are shown in Table 8. The experimental results are given in Figure 6.

We see from Figure 6 that the throughput for the original scheme in Spark is almost the same for all lengths of data block. With our deduplication scheme, the throughput increases as the repetition rate becomes bigger. Furthermore, we can see that the throughput only improves about 10% when repetition rate is 5%. Nevertheless, the improvement approaches dramatically to 60% when the repetition rate is 40%. We conclude that the proposed scheme can transmit more data in a limited bandwidth.

Table 8: Parameters of the second experiment.

Bandwidth	1Mbps
Avg. Length of Data Block	25 bytes
Length of Fingerprint	32 bits
Limitation of LRU Cache Map	1,000,000

4.1.3 Length of Fingerprint

The factor studied in the third experiment is the length of fingerprint. In our work, a 32-bit version of xxHash is chosen as the implementation. In this experiment, we compare the performance of 32-bit version with the version of 64-bit xxHash. Parameters of this experiment are shown in Table 9. The experimental results are given in Figure 7.

Figure 7 shows that, for both 64-bit version and 32-bit version, the throughput improves when the repetition rate increases. However, it is noted that when the length of fingerprint becomes longer, the cost of metadata will be increased. Hence, the throughput for 64-bit version gets less improvement (compared with the 32-bit version) with longer length of fingerprint. We observe from the results that the performance of 64-bit version has only about half improvement over that of 32-bit version. It conforms that parameter h influences the saving of bandwidth in Equation (1). In other words, if the speed of hash functions is about the same, a shorter hash value will be a better choice for our scheme.

Table 9: Parameters of the third experiment.

Bandwidth	1Mbps
Repetition Rate	25%
Avg. Length of Data Block	25 bytes
Limitation of LRU Cache Map	1,000,000

4.1.4 Bandwidth

Bandwidth usage can be reduced by our proposed scheme. In this experiment, we investigate how the availability of network bandwidth impacts on the system throughput. Parameters of this experiment are shown in Table 10. The experimental results are given in Figure 8.

When the bandwidth gets higher, both the original scheme and the proposed scheme have bigger throughput. We also see that the throughput gap between these two schemes grows exponentially. The improvement ratio runs around 25% to 35%. It means that our scheme works better than the original scheme by at least one quarter of the system throughput.

Table 10: Parameters of the fourth experiment.

Repetition Rate	25%
Avg. Length of Data Block	25 bytes
Length of Fingerprint	32 bits
Limitation of LRU Cache Map	1,000,000

4.1.5 Physical World Taxi GPS Trajectory Dataset

In the last experiment we use the real-world taxi GPS trajectory dataset to evaluate the performance of our scheme. Table 11 presents the parameters setting in this experiment.

Table 11: Taxi GPS trajectory dataset.

Bandwidth	1Mbps
Avg. Length of Data Block	25 bytes
Length of Fingerprint	32 bits
Limitation of LRU Cache Map	1,000,000

Figure 9 illustrates the performance of our data deduplication scheme and that of the original scheme in Spark. Since the data in the dataset has different repetition rates over various time intervals, we present the repetition rate as time goes by in the low part of the figure. The axis for the repetition rate is shown at the right hand side of the *y* axis, while the throughput of the original scheme and our proposed scheme are indicated by diamond dots and circle dots respectively.

We can observe that the repetition rates of the real-world GPS trajectory data are not evenly distributed. Obviously, under different repetition rates, the improvement of throughput varies. Overall speaking, the maximum of improvement is about 57 %, while the minimum of improvement is only about 2.6%. The least improvement happens when the repetition rate is very low (about 9 %). Nevertheless, our proposed scheme performs better in all cases of the real dataset.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a fast deduplication data transmission scheme for parallel real-time computing platform like Spark Streaming. The proposed scheme does not need for the specialized data compression technique. Therefore, CPU resource will not be wasted to eliminate redundant chunk by compression and decompression.

According to these experiments, we draw a conclusion that our scheme works most effectively in the following situations:

- The average length of data block is long enough.
- The length of fingerprint is shorter.
- The repetition rate is higher.
- The bandwidth is required to be high.

In the last experiment, we use real-world taxi GPS trajectory dataset to prove that our method also works well on real-world data.

In the future work, we wish to further optimize our scheme. One of the possible directions is parallel transmission. In fact, Spark can execute several receivers to receive raw data. As we know, distributed messaging system like Apache Kafka exploits parallel transmission to send data in parallel to improve performance. It will be an interesting issue to follow. Additionally, the data preprocess can be improved further.

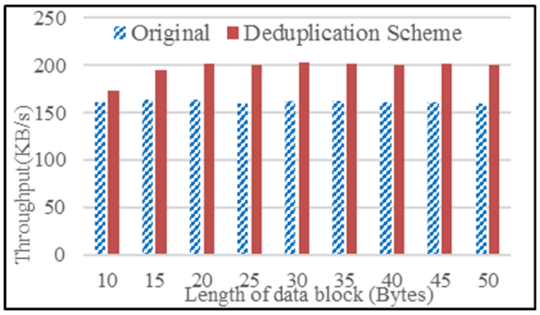

Figure 5: Length of data block versus throughput.

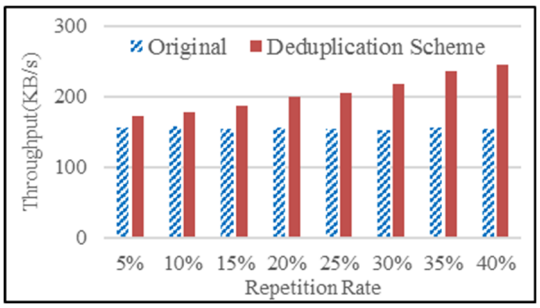

Figure 6: Throughput versus repetition rate.

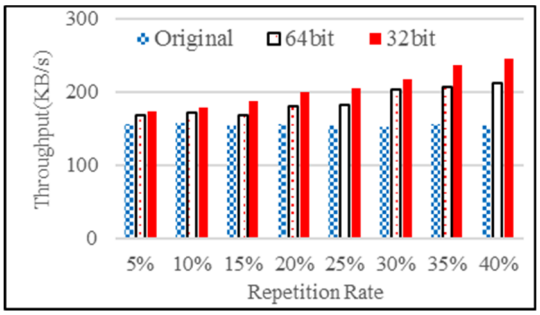

Figure 7: Fingerprint versus throughput (32-bit and 64-bit versions).

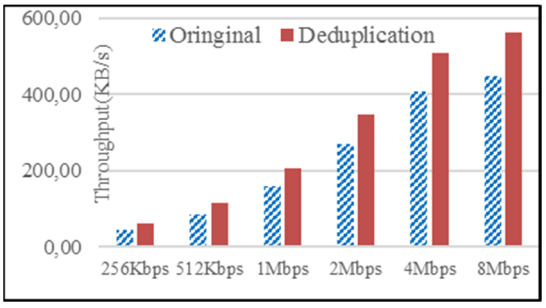

Figure 8: Bandwidth experiment.

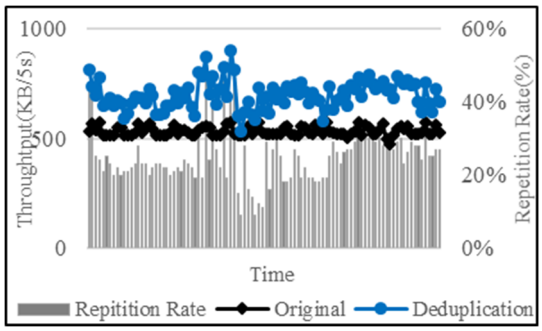

Figure 9: Throughput for taxi GPS trajectory dataset.

ACKNOWLEDGEMENTS

This work is supported by NCSIST project under the contract number of NCSIST-ABV-V101 (106).

REFERENCES

Dean, J., Ghemamat, S., 2008. MapReduce: Simplified Da ta Processing on Large Clusters. *Communications of th e ACM*, vol. 51, no. 1, pp. 07 - 13.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. T he Hadoop Distributed File System. In *MSST'10, IEEE Mass Storage Systems and Technologies.*

Zaharia, M., Chowdhury, M., Franklin M., Shenkr, S., Stoi ca. I., 2010. Spark: cluster computing with working set s. In *HotCloud'10*.

Akyildiz, I., Su, W., Sankarasubramaniam, W., Cyirci, E., 2002. A Survey on Sensor Networks. *IEEE Communic ations Magazine*, vol. 40, no. 8.

Rivest, R., 1992. The MD5 Message-Digest Algorithm. IE TF RFC 1320, April 1992; www.rfc-editor.org/rfc/rfc1 320.txt.

Eastlake, D., Jones, P., 2001. US secure hash algorithm 1 (SHA1), IETF RFC 3174; www.rfc-editor.org/rfc/rfc31 74.txt.

Tridgell, A., Mackerras, P., 1998. The Rsync Algorithm. T echnical Report TR-CS-96-05, Department of Comput er Science, The Australian National University, Canbe rra, Australia.

Odersky, M., Spoon, L., Venners, B., 2007. *Programming in Scala*. Artima Press. Mountain View, CA., 1st editio n.

Deutsch, P., Gailly, J., 1996. ZLIB compressed data forma t specification version 3.3. IETF RFC 1950.

Levine, J., 2012. The 'application/zlib' and 'application/gz ip' Media Types. IETF RFC 6713.

Collet, Y., 2016. xxhash. https://github.com/Cyan4973/xx Hash.

Appleby, A., 2012. SMHasher & MurmurHash. https://gith ub.com/aappleby/smhasher.

Yuan, J., Zheng, Y., Xie, X., Sun, G., 2011. Driving with k nowledge from the physical world," In *KDD'11, 17th I nternational conference on Knowledge Discovery and Data mining*.

Zadeh, L., 1965. Fuzzy sets. *Information and Control*, vol. 8.

Kreps, J., Narkhede, N., Rao, J., 2011. Kafka: A distribute d messaging system for log processing." In *6th Interna tional Workshop on Networking Meets Databases, Ath ens, Greece*.