Caching and Storage Optimizations for Big Data Streaming Systems

Mikhail M. Rovnyagin[1], Valentin K. Kozlov[2], Roman A. Mitenkov[3], Alexey D. Gukov[4], Anton A. Yakovlev[5]
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)
Moscow, Russia
[1]mmrovnyagin@mephi.ru, [2]kozlov.valent@gmail.com, [3]r.mitenkov96@gmail.com, [4]alekseygukov.mk@gmail.com, [5]yakovlev9541@gmail.com

***Abstract*—Data processing is one of the most important processes in Big Data systems. In this paper, we propose a method and its performance model for data deduplication in distributed event driven software systems using Kafka streams and Apache Ignite cache, which reduces network and memory consumption. Also in this article the way of data storage systems optimization is considered by example of Apache Cassandra. The experiments showed that choosing of compression algorithms for different kinds of data with usage of neural network can help to find the balance between memory usage and read speed from the database.**

Keywords—Ignite, caching, message broker, Apache Kafka, compression

I. Introduction and Related Works

Modern industrial distributed systems often use messaging systems to combine components and NoSQL [1,2] systems for long-term storage of poorly structured data. In distributed messaging systems, one of the main problems is the presence of duplicates. Moreover, the generation of duplicates at the beginning of the interaction chain can lead to an increase in parasitic load at the end of the chain [3]. For protection against duplicates, some of the most common tricks are: idempotent data provider and deduplication cache. In the first case, each message sent to the message broker is assigned an identifier-counter, which is checked on the broker's nodes [4]. The deduplication cache contains a fixed in time or volume number of records with which each subsequent message passing through the system is checked. Accordingly, by accelerating the speed of the deduplicator and reducing the amount of memory consumed, the cache deduplication system can be improved.

Compression is often used to reduce the amount of disk space used to store persistent data. However, not all data is equally well compressible [5]. Therefore, if you use the same compression algorithm for different data in the storage system, this may turn out to be ineffective. In this article, we propose a method for automatically selecting a compression algorithm depending on the characteristics of the data entering the storage system.

II. Messaging deduplication

The problem of data duplication often arises when processing large amounts of information in real time. Duplicate data increases the load on the system and can cause errors in its operation. To avoid this problem, preliminary data deduplication is necessary. The proposed solution consists of the following components:

Publisher is a data source. It is an abstraction over a set of objects that forms an input data stream.

Apache Kafka [6] is a message queue through which data flows from one component to another.

Apache Ignite [7] - a distributed in-memory data warehouse

Message Deduplicator - a component of the system that deduplicates the input data stream, forming the output data stream.

Consumer - a subscriber who works with the output stream data stream.

The main component of this solution is Message Deduplicator. Its operation algorithm can consist of the following steps:

1. Receive an input message.

2. Calculate the hash code of the message.

3. Make a request to search for the received hash code in the distributed cache.

If the hash code of the message was not found, then add it to the distributed cache and send the original message to the output channel. If a hash code was found, then block the sending of the message.

Figure 1 shows the architecture of the system described above.

 978-1-7281-5761-0/20/$31.00

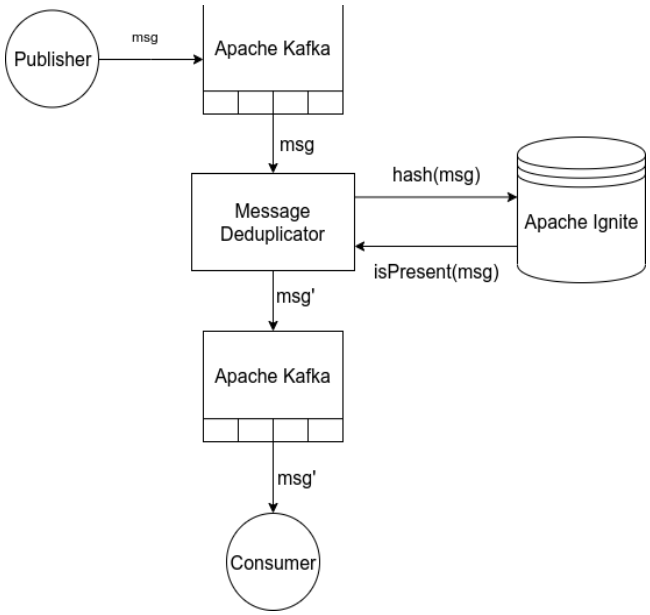

Fig. 1. The system architecture

To conduct stress testing of the developed system and collecting metrics, a stand consisting of three virtual machines was assembled. Using Docker Swarm [8], all the necessary components were deployed on it. The layout of the stand is shown in Figure 2.

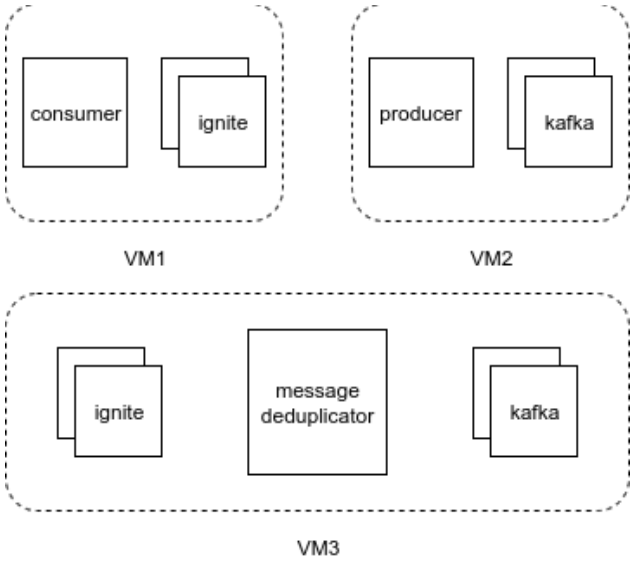

Fig. 1. Scheme of the stand

During the experiment, the search speed was compared in Apache Ignite in two versions. In the first variant (single), all messages from Kafka topics fell into one cache. In the second option (optimized), a separate cache was created for each Kafka topic, and the more duplicates in the topic, the greater the replication coefficient of the corresponding cache. Based on the results, a graph was constructed of the dependence of the message processing time (μs) on the number of messages in the cache, shown in Figure 3.

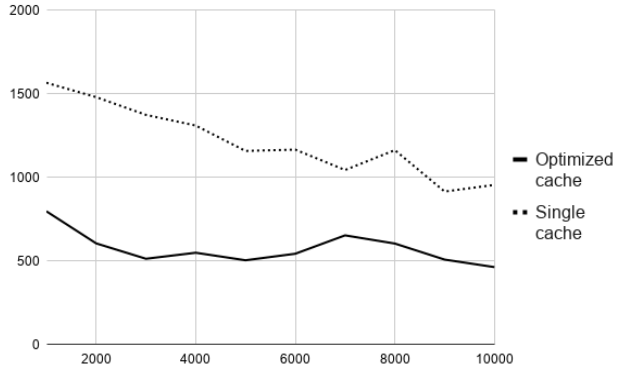

Fig. 3. Graph of the search time versus the number of messages in the distributed cache

The graph shows that in the case of an optimized cache, on average, it takes half as much time to search.

III. Storage Optimization

Another option for optimizing the work with big data, which is proposed in this article, is to find the most optimal ratio between the speed of reading from the data storage system and the degree of compression of this data. Both characteristics are of great importance, it would be nice to have a configurable mechanism that allows everyone to choose the appropriate ratio of these characteristics for themselves, for example, by slightly increasing the size of the memory occupied by the data, significantly increasing the reading speed, or vice versa, saving on the used space a little, losing a bit in the speed of receiving data .

The algorithm proposed in this article is based on a partial analysis of the internal structure of the data that you want to write to the storage system. The task is to evaluate the effectiveness of applying certain compression algorithms to them based on some configurable number of input data samples.

The main characteristic of data compressibility is the compression ratio, which is determined by the following formula:

$$k = \frac{s_0}{s_c}, \quad (1)$$

where k is the compression ratio, So is the volume of the source data, and Sc is the volume of compressed data. This coefficient depends both on the algorithm used and on the data itself. The value of this coefficient in the future will be understood as the concept of the efficiency of the compression algorithm. Typically, algorithms that compress data better are slower, and faster algorithms are often less efficient. Having estimated the compressibility of the input data in advance, we can choose the most suitable compression algorithm for them in accordance with our goals and priorities.

In this article, to analyze the compressibility of data based on their internal structure, a feedforward neural network is used, although in the general case there may be any other algorithm for predicting the compression coefficient, even explicit application of compression algorithms to data samples is allowed, but in practice this works somewhat slower than the

neural network described in this article, especially when it comes to more complex and, as a result, efficient compression algorithms. Apache Cassandra [9] was taken as a storage system. Compression algorithms used: LZ4 and Deflate [10]. The data sample size is 1 Kb.

The experiment consisted of 3 stages: preparation, conduct and evaluation of the results.

In the preparatory phase, the following steps can be distinguished:

1. Preparation of datasets for training and testing the neural network, the basis of which was Silesia corpus, expanded by a number of archives, which are a good example of poorly compressed data, as well as video content. Datasets are sets of fragments of various files of a fixed length. The size of one fragment of the training dataset is equal to the size of the sample, i.e. 1 Kb The size of one fragment in the test dataset is equal to the default value of the chunk_length_kb tables parameter in Apache Cassandra (64 Kb).
2. Training a neural network, the task of which is to evaluate the difference in compression ratios using LZ4 and Deflate. If the difference between the coefficients is more than N percent, then a slower, but at the same time compressive Deflate, otherwise LZ4, is chosen. Learning takes place “with the teacher”, i.e. at the stage of preparing the dataset, all compression coefficients were experimentally obtained and stored in a special table in the database, the value of the parameter N was also predefined, which was set to 5% during the experiment. To work with a neural network, the Keras library was used, the structure of the neural network used is shown in Figure 4.

```
NETWORK_INPUT_SIZE = 1024
model = keras.Sequential()
model.add(keras.layers.Dense(NETWORK_INPUT_SIZE, \
                             input_shape=(NETWORK_INPUT_SIZE
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dense(2, activation='softmax'))
```

Fig. 4. Neural network structure

Deploy Apache Cassandra using Docker, create 4 experimental tables: LZ4PartTable, LZ4Table, DeflatePartTable, DeflateTable. The first 2 tables use the LZ4 compression algorithm, and the other Deflate.

The experiment:

1. Alternately taking M samples from each fragment of the test dataset (M is a configurable parameter, was set to 3 in the experiment), each of which is fed to a trained neural network.
2. The choice of compression algorithm for each fragment based on the results of the neural network.
3. Writing each test piece of data to a table with the compression algorithm selected in the previous step (DeflatePartTable or LZ4PartTable).
4. Writing all test data first to the DeflateTable table, and then to LZ4Table.

The experimental design is shown in Figure 5.

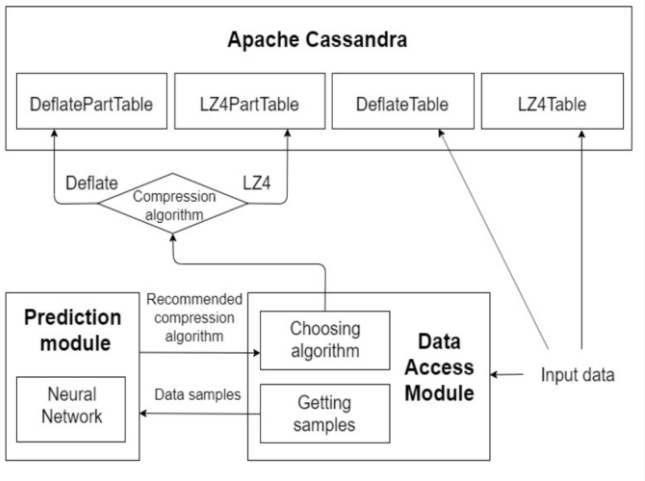

Fig. 6. The scheme of the experiment

Verification of the results:

Comparison of data compression ratios when using Deflate, LZ4, as well as the above option for choosing the appropriate algorithm based on the internal data structure.

Measurement and further comparison of sequential read time from

DeflatePartTable and LZ4PartTable;

DeflateTable;

LZ4Table.

Timing results of sequential reads from the above tables are presented in Figure 6.

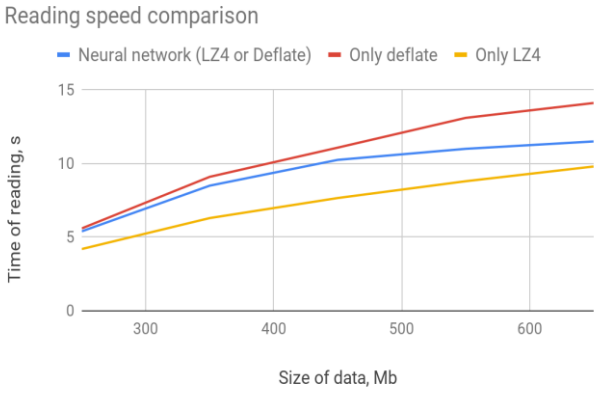

Fig. 6. Reading time comparison

The graph shows that the reading speed in the case of using 2 tables with a different compression algorithm is greater than in the case of using Deflate as a single algorithm. In this experiment, the overhead of working with 2 tables was compensated by an increase in read speed even at the size of

the initial data of 300 MB, with 600 MB there is already a significant performance increase compared to Deflate (about 20%). This is not surprising, because part of the data, namely 37%, was compressed by the faster LZ4 algorithm. Now let's check how much the average compression ratio has decreased. Figure 7 shows a comparison of compression ratios using Deflate, LZ4 algorithms, as well as a neural network that selects a more suitable algorithm based on samples.

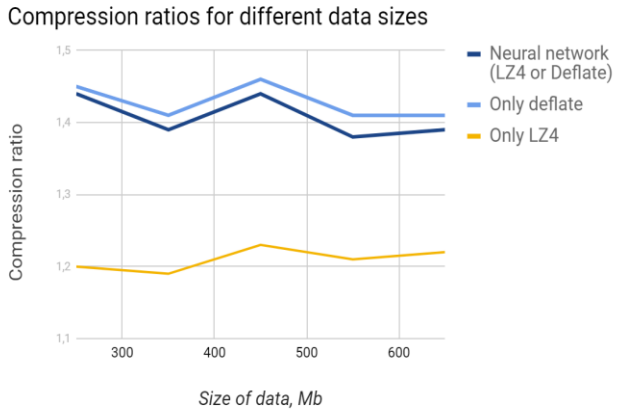

Fig. 7. Comparison of compression ratios

The compression ratio decreased on average by 2-3%, which confirms the success of the experiment. Thus, by analyzing the internal structure of the data during recording, it was possible to significantly increase the reading speed (about 20%) with a slight decrease in the average compression ratio (2-3%).

IV. Conclusion

In this work, we proposed ways to increase the productivity and efficiency of distributed computing systems by reducing the time it takes to determine duplicates in messages and reducing the amount of disk space. As the number of replicas in the duplicate cache increases, the time taken to check the message is reduced. It was also possible to use disk space more efficiently by using a lower-quality compression algorithm for poorly compressible data and a better algorithm for well-compressible data.

REFERENCES

[1] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly Media, Inc, 2017, ISBN: 978-1449373320

[2] A. Naskos, A. Gounaris and I. Konstantinou, "Elton: A Cloud Resource Scaling-Out Manager for NoSQL Databases," *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, Paris, 2018, pp. 1641-1644.
doi: 10.1109/ICDE.2018.00196

[3] M. M. Rovnyagin, S. S. Varykhanov, Y. V. Maslov, I. S. Riakhovskaia and O. V. Myltsyn, "NFV chaining technology in hybrid computing clouds," *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Moscow, 2018, pp. 109-113. DOI: 10.1109/EIConRus.2018.8317042.

[4] B. R. Hiraman, C. V. M and C. Karve Abhijeet, "A Study of Apache Kafka in Big Data Stream Processing," *2018 International Conference on Information , Communication, Engineering and Technology (ICICET)*, Pune, 2018, pp. 1-3.

[5] A. A. Kozlov, A. A. Aleshina, I. S. Kamenskikh, M. M. Rovnyagin, D. M. Sinelnikov and D. A. Shulga, "Increasing the functionality of the modern NoSQL-systems with GPGPU-technology," 2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), St. Petersburg, 2016, pp. 242-246. doi: 10.1109/EIConRusNW.2016.7448164

[6] T. Shapira, Kafka: the Definitive Guide. O'Reilly Media, Inc, 2017, ISBN: 978-1491936153.

[7] M. Zheludkov, S. Bhuiyan, The Apache Ignite Book. Lulu.com Publisher, 2019, ISBN: 978-0359439379

[8] N. Marathe, A. Gandhi and J. M. Shah, "Docker Swarm and Kubernetes in Cloud Computing Environment," 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 2019, pp. 179-184.

[9] J. Carpenter, E. Hewitt, Cassandra: The Definitive Guide: Distributed Data at Web Scale. O'Reilly Media; 2 edition, 2016, ISBN: 978-1491933664

[10] D. Harnik, E. Khaitzin, D. Sotnikov and S. Taharlev, "A Fast Implementation of Deflate," *2014 Data Compression Conference*, Snowbird, UT, 2014, pp. 223-232. doi: 10.1109/DCC.2014.66