Streaming state management methods for real-time data deduplication

João V. A. Esteves[1], Sérgio Lifschitz[2], Rosa M. E. M. Costa[1], Ana Carolina Almeida[1]

[1] Department of Computer Science – State University of Rio de Janeiro (UERJ) - Rio de Janeiro – RJ - Brazil

[2]Department of Computer Science – Pontifical Catholic University of Rio de Janeiro (PUC-Rio) – Rio de Janeiro – RJ - Brazil

joao.esteves@acm.org, sergio@inf.puc-rio.br, {rcosta, ana.almeida}@ime.uerj.br,

Abstract. *Data duplication is a common problem on data streams processing applications that occurs due to software error or adoption of data loss prevention measures, jeopardizing real-time data analyses. This paper explores stream-based deduplication methods to identify challenges from these methods and proposes a decision method to choose the most appropriate strategy for a domain. This work investigates native solutions and auxiliary tools to provide data deduplication and fault tolerance. The experimental results show that it is necessary to use fast additional storage to persist the read keys, as long as they can appear, or to use the optimized storage, with a quick key search.*

1. Introduction

Batch processing is the processing of a large volume of data at once, which only allows decisions based on historical data, which takes hours or days to analyze the data. An advantage of this type of processing is the possibility of including semantic rules of the domain, that is, complex rules since decision-making does not need to be immediate.

However, some companies require real-time decision making which needs the use of data stream processing (DSP). According to Gedik et al. (2008), DSP is a method to process a time-ordered series of events on-the-fly. In this way, companies relax in their domain rules to guarantee agile decision making, with data analysis taking place in minutes or even seconds. In that case, domain rules are no longer a priority.

A crucial and common domain rule, which that may influence the performance of data processing, is the data uniqueness. Although in relational databases, this uniqueness is guaranteed by integrity restrictions, the same is not valid in DSP systems. In DSP systems, it is common to persist duplicate data due to implementations that prioritize fast persistence by removing domain rule checks as the volume of data increases.

Some DSP frameworks, such as Apache Spark [Zaharia et al. 2010], have real-time deduplication mechanisms. These mechanisms may consume a high amount of resources in a cluster, which can cause application failures and data loss while the application is unavailable. In this way, another relevant requirement to investigate is fault-tolerant. This work investigates Spark's native deduplication mechanisms and possible fault-tolerant solutions, to analyze the challenges from each solution, proposing a solution choice method based on use cases.

The remainder of this paper is organized as follows. In Section 2, we discuss the related works. In Section 3, we provide an overview of streaming state management methods for real-time data deduplication. Section 4 details our experimental results. Section 5 concludes this work.

2. Related works

There are many studies on fault-tolerant streaming state management [Kwon et al. 2008] [Fernandez et al. 2013][Wu and Tan 2015]. Most of them propose standalone solutions that are not interoperable with a general DSP framework like Spark, Flink[1], and Samza[2]. In addition, many works have analyzed how these DSP frameworks can be natively fault-tolerant to prevent state loss [Carbone et al. 2017][Noghabi et al. 2017]. These solutions often do not provide a native way to store state outside the cluster to prevent cluster node termination, standard on cloud environments, and lose this data.

There are some novel solutions to prevent state loss that can be used by these DSPs frameworks, like Megaphone [Hoffmann et al. 2019] and Rhino [Del Monte et al. 2020], but these solutions do not have their source code publicly available.

Data deduplication methods are often discussed as a subsequent step after the data is stored, with batch [Kaur et al. 2018][Xia et al. 2019] and real-time [Duan and Xiong 2015][Xia et al. 2016]. Our work evaluates how deduplication can be applied to data before its persistence (ingestion step). This approach enables less data to be stored and to be analyzed by future queries.

3. Methods for real-time data deduplication

Apache Spark is a data processing framework that originated as a batch solution with better performance than Hadoop MapReduce [Zaharia et al. 2010]. Stream processing has been added as an evolution of the batch model, in which at intervals of time, a subset of data, known as micro-batch, is extracted, transformed, and persisted.

An advantage of this evolution is the use of the batch model's transformation and persistence operations, and mainly of the table abstraction guaranteed by this model known as the data frame. Besides, new activities were included in the streaming model, which persists in memory parts of the data consumed among micro-batches, available through a time window configured in the system. This data set is the streaming state.

3.1 Apache Spark deduplication mechanisms

This section presents the two native methods (using distinct and dropDuplicates operators) available in Apache Spark to deduplicate the data.

The simplest deduplication method for a dataframe is the distinct operator, which removes duplicates by comparing rows, column by column. At the end of a micro-batch, the deduplicated dataframe is persisted at the destination. The state is updated with new rows for comparisons by subsequent micro-batches.

Another operator used to avoid the excessive use of memory and the delay of future micro-batches is the dropDuplicates. It performs the comparison between rows

[1] https://flink.apache.org/

[2] https://samza.apache.org/

using only their keys, avoiding unnecessary comparisons of all columns of a data frame, and persisting only these keys in memory.

By default, Apache Spark does not persist the streaming state on disk to avoid the loss of data necessary to guarantee the uniqueness if the application needs to be restarted.

3.2 Auxiliary tools for deduplication in Apache Spark

To avoid losing the streaming state, it is common to use auxiliary tools to persist it outside the application. It is updated at the end of each micro-batch and consulted when comparing the rows of a dataframe.

Apache Ignite

Apache Ignite is a non-relational database that persists data in memory, but that may be configured to copy all data to disk, keeping only the last data accessed in memory [Stan et al. 2019].

This database has SQL-like functions that allow us to perform critical comparisons on the database, ensuring that the Spark cluster does not need to persist a state between micro-batches, reducing the memory consumption by the application. Another benefit of Ignite is the replication of data stored in memory and on disk. This allows a new instance of the database to improve query performance if the number of comparisons increases and ensures that if one instance fails, the other instances will be able to make all data available without loss. To verify that a key did not exist in Ignite, a LEFT ANTI JOIN type join is performed between the data keys processed in the current micro-batch with the already processed keys persisted in Ignite.

Apache Hudi

Apache Hudi[3] is a file format which persist the keys of each data in a file in a Bloom filter contained in its metadata. This filter is consulted by all Spark writing operations to validate the existence of each data produced at the end of a micro-batch, to delete or update the already persisted data in each file. It is important to note that, unless configured, Hudi files do not delete or update the rows, but rather add a new version of this row so that it is possible to query historical data. If history is disabled, it is possible to make it impossible to keep duplicate keys in the destination.

4. Experiments

This section presents experimental environment and results.

4.1 Experimental Environment

An application that monitors the RAM usage and evaluates each micro-batch's execution time for each deduplication method was implemented with Scala. The application (i) consumes messages from an Apache Kafka topic, (ii) removes duplicate messages, and (iii) persists these messages in an AWS S3 bucket on Parquet format[4].

The cluster in which the application was executed uses the Amazon Web Services (AWS). There is an Elastic MapReduce (EMR) service, with six r5.2xlarge nodes, which

[3] https://hudi.apache.org/

[4] Except with the Apache Hudi file format test

has eight cores of the Intel Xeon Platinum 8175 processor and 64GB of memory RAM each, performing the slave function of the cluster, and one r5.xlarge master node with half the specifications of the slaves' nodes.

The Apache Kafka was configured in a 3-node m5.large cluster, with two cores of the Intel Xeon Platinum 8175 processor and 8GB of RAM each. The message topic was partitioned into 30 parts, using the keys of each message as partitioning criteria.

In the Apache Ignite test, this database was instantiated on Spark slave nodes to prevent data traffic out from the cluster during data comparison, avoiding the degradation of application performance.

In the Apache Hudi test, the application was configured not to persist historical data. Also, the default size limit of 128MB was not changed.

We have used a data set with a two-week sample from a Brazilian classifieds company, totaling about 540 million messages originating from Kafka, each with 276 fields, of which only 23% were unique messages, adding up to a total of 1.624 TB of data processed by Apache Spark.

The company had two important domain rules: a duplicate data could be produced even over a year after its first appearance, and the ingested data needs to be available within a ten minutes delay from creation at most. To enforce these rules, no time window was configured for any test (as it would be unfeasible to maintain its keys in memory for a year), and each micro-batch was triggered every tenth minute. The system was implemented to consume 10 million messages per micro-batch, totaling 54 micro-batches.

Therefore, it is possible to analyze the difference among micro-batches in an extreme case and force a high consumption of RAM. Since the domain rule imposed by the company in this application, there is a chance a duplicate message will be produced with up to 1 year away from the original, making it impossible to use temporal windows.

4.2 Experimental Results

The results presented here comprise the average of five repetitions of each test, which were performed in isolation on the cluster. To prevent any hidden RAM usage from being considered, the cluster was deleted at the end of the last micro-batch and provisioned anew for the next test.

The graphics in Figure 1 show the memory consumption of the four case studies, in which it is possible to highlight the high use of memory in the use of the distinct method (Figure 1-a), which failed during the execution of the 17th micro-batch on all repetitions due to lack of memory in the cluster nodes. By zooming it, figure 1-b shows that with the other methods, memory consumption remained below 50GB. DropDuplicates and Ignite methods grow linearly with each micro-batch, while Hudi is always oscillating below 5GB. It is worth mentioning that there is a considerably higher consumption in Ignite compared to dropDuplicates. This is because of the replication of keys among instances of Ignite. It is also necessary to emphasize that, in case of failure, Ignite would be able to recover data that would have been lost in a restart of the application only with dropDuplicates.

Figure 2 shows the duration (in seconds) of each micro-batch processing. There were peaks in the chart, especially during the 12th micro-batch. This is due to the comparison made by this method considering all columns if there is no difference. For

the dropDuplicates process, there is a significant improvement in time after the 13th micro-batch. As most messages are duplicated, the state of the application already has most of the possible keys in this data set. Hudi had the worst performance, as expected, due to the inevitable metadata persistency in S3, outside the cluster.

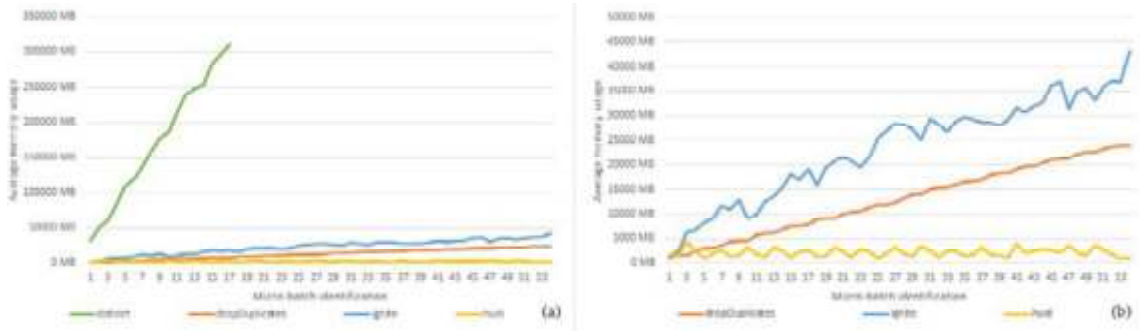

Figure 1. Memory consumption of micro-batches

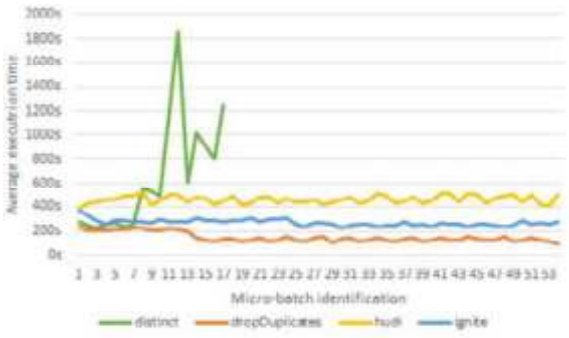

Figure 2. The execution time of micro-batches

Despite these results, it is essential to note that for Ignite and Hudi, duplicates were not removed in any other way than the specific solution itself. This means duplicates within the same micro-batch were not treated, only among micro-batches. An advantage of Hudi is that the file format itself displays only one version of a key, so deduplication occurs automatically during the query. Also, the format adds new data to the same file until it reaches a size limit. This feature prevents many small files from being in the data repository, impacting the execution time of a subsequent query to that data.

Also, given how Spark stores state, its native methods do not have any protection against hardware or software failure. On this regard, using Ignite that is a NoSQL database that replicates its data across its nodes, or using Hudi that stores the necessary metadata to remove duplicates within the data itself, achieves the expected goal related to fault tolerance.

5. Conclusions

This paper compared four deduplication methods for data stream processing as well as the fault tolerance capability, by their RAM usage and impact on micro-batch duration.

Through the experimental results, it is possible to notice that in scenarios with a massive volume of processed data per minute, only the native operators' methods of Apache Spark are not enough to persist state between micro-batches of data of a high amount of processed data. Both Ignite and Hudi proved to be efficient in real-time data deduplication. However, it is possible to notice the advantage of Ignite in time processing

and Hudi in memory consumption. In this case, it is necessary to consider the priority of the analysis and how much resource is available when deciding which solution to choose.

Future work will address new comparisons with other data processing frameworks like Apache Flink, and different stateful scenarios can be considered, such as anomaly detection. Furthermore, the fault tolerance test of Hudi and Ignite may be deepened through simulated failures as losing a node or network bandwidth degradation.

References

Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S. and Tzoumas, K. (2017). "State management in Apache Flink®: consistent stateful distributed stream processing". In *VLDB Endowment*, v. 10, n. 12, pp. 1718-1729.

Del Monte, B., Zeuch, S., Rabl, T. and Markl, V. (2020). "Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines". In *ACM SIGMOD Int. Conf. on Management of Data*, pp. 2471-2486, Oregon, US.

Duan, L., and Xiong, Y. (2015). "Big data analytics and business analytics". In *Journal of Management Analytics*, v. 2, no. 1, pp. 1-21.

Fernandez, R. C., Migliavacca, M., Kalyvianaki, E. and Pietzuch, P. (2013). "Scalable and Fault-tolerant Stateful Stream Processing". Imperial Coll. Comp. Student Wksp, UK..

Gedik, B., Andrade, H., Wu, K. L., Yu, P. S., and Doo, M. (2008). "SPADE: the system s declarative stream processing engine". In *ACM SIGMOD Int. Conf. on Management of data*, pp. 1123-1134, Vancouver, CA.

Hoffmann, M., Lattuada, A. and McSherry, F. (2019). "Megaphone: Latency-conscious state migration for distributed streaming dataflows". In *VLDB Endowment*, v. 12, n. 9, pp. 1002-1015.

Kaur, R., Chana, I. and Bhattacharya, J. (2018). "Data deduplication techniques for efficient cloud storage management: a systematic review". In *Journal of Supercomputing*, v. 74, n. 5, pp. 2035-2085.

Kwon, Y., Balazinska, M., and Greenberg, A. (2008). "Fault-tolerant stream processing using a distributed, replicated file system". In *VLDB Endowment*, v. 1, n. 1, pp. 574-585, Auckland, NZ.

Noghabi, S. A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I. and Campbell, R. H. (2017). "Samza: stateful scalable stream processing at LinkedIn". In *VLDB Endowment*, v. 10, n. 12, pp. 1634-1645.

Stan, C. S., Pandelica, A. E., Zamfir, V. A., Stan, R. G., and Negru, C. (2019). "Apache Spark and Apache Ignite Performance Analysis". In *Int. Conf. on Control Systems and Computer Science (CSCS)*, pp. 726-733.

Wu, Y. and Tan, K. L. (2015). "ChronoStream: Elastic stateful stream computation in the cloud". In *IEEE 31st Int. Conf. on Data Engineering*, pp. 723-734, Seoul, KR.

Xia, W., Feng, D., Jiang, H., Zhang, Y., Chang, V. and Zou, X. (2019). "Accelerating content-defined-chunking based data deduplication by exploiting parallelism". In *Future Generation Computer Systems*, v. 98, pp. 406-418.

Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., and Zhou, Y. (2016). "A comprehensive study of the past, present, and future of data deduplication". In *IEEE*, v. 104, n. 9, pp. 1681-1710.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). "Spark: Cluster computing with working sets". In *USENIX conf. on Hot topics in cloud computing*, v. 10, pp. 10.