An exploratory analysis of methods for real-time data deduplication in streaming processes

Esteves, João; Costa, Rosa; Zhou, Yongluan; Brito De Almeida, Ana Carolina

Published in:
Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems

DOI:
10.1145/3583678.3596898

Publication date:
2023

Document version
Peer reviewed version

Citation for published version (APA):
An exploratory analysis of methods for real-time data deduplication in streaming processes

João V. A. Esteves
joao.esteves@hellofresh.com
HelloFresh SE
Berlin, Germany

Yongluan Zhou
zhou@di.ku.dk
University of Copenhagen
Copenhagen, Denmark

Rosa M. M. Costa
rcosta@ime.uerj.br
State University of Rio de Janeiro
Rio de Janeiro, Brazil

Ana Carolina Almeida
aca@di.ku.dk
University of Copenhagen
Copenhagen, Denmark
State University of Rio de Janeiro
Rio de Janeiro, Brazil

ABSTRACT
Modern stream processing systems typically require ingesting and correlating data from multiple data sources. However, these sources are out of control and prone to software errors and unavailability, causing data anomalies that must be necessarily remedied before processing the data. In this context, anomaly, such as data duplication, appears as one of the most prominent challenges of stream processing. Data duplication can hinder real-time analysis of data for decision making. This paper investigates the challenges and performs an experimental analysis of operators and auxiliary tools to help with data deduplication. The results show that there is an increase in data delivery time when using external mechanisms. However, these mechanisms are essential for an ingestion process to guarantee that no data is lost and that no duplicates are persisted.

CCS CONCEPTS
• Information systems → Parallel and distributed DBMSs; Stream management.

ACM Reference Format:

1 INTRODUCTION
Due to the explosion of data produced by IoT devices and the widespread use of social networks and streaming services, companies are increasingly turning to real-time analysis of data to understand their customers better and respond more quickly to changes in their businesses. OLX Brasil\(^1\), a company grappling with the rapid growth of its data due to increased visibility, found that its existing batch-based processing infrastructure was no longer able to meet its data analysis requirements.

To address this issue, OLX Brasil opted to implement a streaming data infrastructure, which promised to enable real-time data processing and analysis. However, once this infrastructure was deployed, the data engineering team faced the challenge of managing duplicated data. RabbitMQ\(^2\) is used for exchanging data between event-based components and act as the data source for streaming analysis. However, RabbitMQ does not support exactly-once delivery, which means that message delivery errors caused by node failures or network delays can lead to producers sending messages multiple times to achieve at-least-once delivery, causing data duplication.

Data duplication is a common problem across many data sources in the company, exacerbated by the microservice architecture that requires data access across services to be done through messages. For instance, when accessing historical data for internal auditing processes, duplicate data could appear months later. Furthermore, while exactly-once delivery can help address the issue, it cannot prevent data duplication at the source caused by human errors, legacy systems, database errors, misconfiguration of the source, intermediary and aggregation systems, and so on. Therefore, OLX Brasil must address the challenge of managing duplicated data in its streaming data infrastructure to ensure accurate and reliable analysis of its data.

Conventionally, data deduplication is typically done after data ingestion via an auxiliary process or during data analysis by disregarding duplicates. However, these approaches do not meet the needs of real-time data analysis, which requires deduplication to occur at ingestion time. Given the highly heterogeneous causes of data duplications in OLX Brasil’s system, one cannot assume duplicate data only appears within a fixed time window. Therefore, data deduplication in a streaming fashion at ingestion time is highly challenging regarding memory management. Additionally, stream processing runs continuously with no specific time for completion and hence may experience unforeseen errors in the environment.

\(^{1}\)https://www.olx.com.br/

\(^{2}\)https://www.rabbitmq.com/
To work around these problems, this paper proposes persisting the known state in an external and autonomous environment to enable accurate results during crash recovery. In the presence of data duplication, this work aims to reduce the time from the ingestion of new data to its availability by applying deduplication strategies that are fault-tolerant, ensuring the consistent state of data and application in case of a restart. The paper explores various methods that can be used with the Apache Spark library to deal with data deduplication in real-time, evaluating their computational resources, delivery time, size of persisted data, fault tolerance, and time of analytical queries. The results are summarized in a table that serves as an aid for developers to make a faster decision on which tools to adopt for deduplicating data.

This paper extends the previous work published in [9] by detailing the architecture, performing a fault tolerance analysis, raising the size and number of files generated during processing, and evaluating the performance of queries executed on the evaluated test scenarios. The work is significant for OLX Brasil and other organizations that use streaming data infrastructure to help them manage duplicated data and improve their data analysis.

The rest of this paper is structured as follows. In Section 2, we present our motivational scenario, which includes the challenges we faced and the requirements raised by OLX Brasil. Section 3 gives an overview of the tools and techniques used for real-time data deduplication. Section 4 provides a detailed description of the infrastructure and architecture used for the tests, as well as the evaluated scenarios, experimental results, and a discussion of our findings. In Section 5, we discuss related works in the area. Finally, Section 6 concludes the paper.

## 2 MOTIVATION

### 2.1 Domain Description

OLX Brasil is a leading digital classifieds company in Brazil that records and persists every step of the ad life cycle on its platform. Initially, the ad data was stored in a relational database. However, due to the company’s rapid growth and the increasing volume of records produced daily, the database infrastructure could not keep up with the high number of write operations. As shown in Table 1, the average number of daily events registered before the pandemic was around 3 million in 2015, occupying an average of 1.5 GB of disk space daily. In subsequent years, these values increased significantly, with disk space usage reaching almost 8 GB per day (11 million events) in 2018.

As the company grew in popularity, it encountered difficulties with its existing batch process for extracting data from a relational database. As a result, the company migrated to a new streaming process that extracted data from messages. However, upon analyzing a subset of this data, specifically a 14-day period of ad announcements in November 2019 that amounted to 1.6 terabytes, it became apparent that there was an enormous amount of duplicate records. As shown in Figure 1, the percentage of duplicate data (blue colour) was significantly higher than that of unique data. This duplication issue was compounded by the fact that some of the events corresponded to ads with years of delay, which could not be discarded during ingestion due to the possibility that they did not exist in the final data repository.

Data duplication in the company can be explained by a few main factors: consumer failures, multiple data sources and historical data. When failures (e.g., network delay) occurred on the consumer side, this required RabbitMQ to send the same message multiple times until reaching at-least-once delivery, which could cause data duplication. Additionally, data was consumed from multiple data sources. Multiple sources may have the same data, causing the same data to be sent multiple times to the message broker. Thus, the message broker has no way of identifying the data as a duplicate. Finally, the company has historical data in several data sources that need to be consumed to provide inputs for the internal audit processes. The use of this historical data meant that the data duplication was only identified months (years) later. The event may be generated with a very high delay rate, its duplication being detected with up to a year of difference between its creation date and the date on which the event actually occurred (as seen in the data sample shown in Figure 1 with data from events that were created in 2019 but actually occurred in 2016-2019).

Data duplication in the company can be explained by a few main factors: consumer failures, multiple data sources and historical data. When failures (e.g., network delay) occurred on the consumer side, this required RabbitMQ to send the same message multiple times until reaching at-least-once delivery, which could cause data duplication. Additionally, data was consumed from multiple data sources. Multiple sources may have the same data, causing the same data to be sent multiple times to the message broker. Thus, the message broker has no way of identifying the data as a duplicate. Finally, the company has historical data in several data sources that need to be consumed to provide inputs for the internal audit processes. The use of this historical data meant that the data duplication was only identified months (years) later. The event may be generated with a very high delay rate, its duplication being detected with up to a year of difference between its creation date and the date on which the event actually occurred (as seen in the data sample shown in Figure 1 with data from events that were created in 2019 but actually occurred in 2016-2019).

As the company grew in popularity, it encountered difficulties with its existing batch process for extracting data from a relational database. As a result, the company migrated to a new streaming process that extracted data from messages. However, upon analyzing a subset of this data, specifically a 14-day period of ad announcements in November 2019 that amounted to 1.6 terabytes, it became apparent that there was an enormous amount of duplicate records. As shown in Figure 1, the percentage of duplicate data (blue colour) was significantly higher than that of unique data. This duplication issue was compounded by the fact that some of the events corresponded to ads with years of delay, which could not be discarded during ingestion due to the possibility that they did not exist in the final data repository.

Data duplication in the company can be explained by a few main factors: consumer failures, multiple data sources and historical data. When failures (e.g., network delay) occurred on the consumer side, this required RabbitMQ to send the same message multiple times until reaching at-least-once delivery, which could cause data duplication. Additionally, data was consumed from multiple data sources. Multiple sources may have the same data, causing the same data to be sent multiple times to the message broker. Thus, the message broker has no way of identifying the data as a duplicate. Finally, the company has historical data in several data sources that need to be consumed to provide inputs for the internal audit processes. The use of this historical data meant that the data duplication was only identified months (years) later. The event may be generated with a very high delay rate, its duplication being detected with up to a year of difference between its creation date and the date on which the event actually occurred (as seen in the data sample shown in Figure 1 with data from events that were created in 2019 but actually occurred in 2016-2019).

The large volume of duplicate records poses a challenge for the company. As many processes and legacy applications depend on the uniqueness of this data at the source, it is impractical to rewrite queries to eliminate duplicates during query execution. Moreover, the use of the DISTINCT clause in queries submitted to the relational database degrades its performance. Ideally, data duplication should be eliminated before data ingestion, avoiding the need to rewrite legacy queries embedded in black box applications.

However, implementing a real-time deduplication mechanism poses challenges. The need to query historical records when deduplicating new records increases processing time, especially as historical volume grows. One can use a time window to constrain

### Table 1: Average size and number of daily events per year

<table>
<thead>
<tr>
<th>Year</th>
<th>Size (GB)</th>
<th>Size (in millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>1.545604</td>
<td>3.183938</td>
</tr>
<tr>
<td>2016</td>
<td>2.92892</td>
<td>4.93282</td>
</tr>
<tr>
<td>2017</td>
<td>6.532226</td>
<td>10.80746</td>
</tr>
<tr>
<td>2018</td>
<td>7.922492</td>
<td>11.09315</td>
</tr>
<tr>
<td>2019</td>
<td>4.756737</td>
<td>7.428618</td>
</tr>
<tr>
<td>2020</td>
<td>6.32158</td>
<td>9.056052</td>
</tr>
</tbody>
</table>
the number of records to be queried if the maximum time interval between the ingestions of duplicate records is known. But this approach faces several challenges:

1. In the scenario of OLX Brasil, the ingestion time of duplicate records can differ by months. Using such a large window size would lead to a prohibitively high memory consumption;
2. The time interval between duplicate occurrences may change during data ingestion. For instance, the business rule that affects duplicate occurrences may be changed, or the delay of a data source may increase. As a result, the window period will be compromised, possibly causing duplicates not being detected.
3. Upon a failure, the state of the window has to be reconstructed to recover the streaming deduplication process. Checkpointing the large window state to persistent storage services is challenging.

In addition to these challenges, OLX Brasil impose the following requirements to the data deduplication tool: (1) use only open-source tools; (2) reduce RAM usage for storing states between micro-batches to enable the use of more cost-effective AWS instances or leveraging a cluster to run more streaming applications in parallel; (3) reduce the execution time of micro-batches to prevent possible peaks from breaching the SLA time; (4) ensure the possibility to stop and restart the stream without having duplicates or loss of data; and (5) ensure that the query time for the processed data is equal to or less than that achieved by the current database in a batch process. Meeting these requirements will help ensure that the deduplication solution is cost-effective, fault-tolerant, and efficient in processing data.

3 Tools and Methods

This section presents an overview of the tools used and the methods evaluated in this paper considering the first requirement.

Apache Spark is a framework composed of distributed data processing libraries with several primitives for data analysis [4]. The Spark structured API allows ETLs to be implemented with primitives similar to those found in the ANSI SQL standard. This does not preclude the use of user-defined functions. In this way, it facilitates adoption by users already accustomed to the syntax and functionality of these primitives. In addition, Spark’s streaming API works with the micro-batch processing model, i.e., Spark’s streaming engine periodically checks the data source and executes a batch query on new data, which arrived since the last batch.

Many streaming processes use messages as a source to ingest data, due to the high performance of writing these messages and the better distribution of data (messages) among interested consumers. Apache Kafka® is a distributed event streaming platform that follows the PubSub model. A producer of a message persists it in topics, which one or more consumers can receive this message. Some platforms make use of queues instead of topics. As a consequence, a message in a queue is delivered only to one consumer and, if the latter does not signal receipt of the message, there will be some way of retry until message delivery is confirmed. Avoiding the retry policy, through the use of topics, requires the platform user to implement a policy for handling missing messages. While this effort exists, it allows for a considerable increase in message throughput, which makes this type of platform a better candidate for a large volume of data received per second.

Afterwards processing in Kafka, the message is persisted in a data store for further query and analysis. The storage system for this work chosen is the AWS S3® repository. This repository has integration with the tools used for the ETL and for subsequent data query and transparency of the infrastructure, not being necessary to manage it directly. Regarding fault tolerance, the AWS service itself guarantees that data will not be lost, transparently. There is no need to manage nodes and data replication as the storage automatically replicates data across multiple availability zones, with no delays or loss of information. Thus, the data repository meets the fourth requirement naturally, and prevents manual data duplication from interfering with ensuring the uniqueness of the data when accessing them.

So far, the main tools and the data repository that will be used have been described. Next, the deduplication methods that were analyzed and will be compared in this work are described. The first two methods are native operators of the tool, already described, Apache Spark. The other methods chosen are open-source and already available and approved by the company.

The simplest method for deduplicating data in Spark is the distinct operator [4]. This operator removes duplicate rows from a dataframe from the column-by-column comparison. Dataframe is the storage unit of data distributed collections in Spark, organized into rows and columns, similar to relational database tables. The dropDuplicates operator allows removing duplicates from a dataframe just by comparing a subset of dataframe columns. This subset being called dataframe keys [4].

RocksDB is a non-relational key-value database, which grew out of LevelDB database design, commonly used to store metadata [10]. A key feature of RocksDB is the way data is written at database levels, where new data is written to an in-memory based table with an optional transaction log on persistent storage, the WAL (Write-Ahead Log). As this memory-based table fills up, data is moved to the next level in the database through a process called compression. When that level is filled, the data is migrated to the lower level again, and so on.

Apache Ignite is a non-relational key-value database, where computer memory is used as the main means for storing data [17]. To ensure fault tolerance, data in this database may be replicated between a subset of Ignite instances, increasing data availability and decreasing data traffic at query time. Note that this mechanism duplicates data, however, unlike the problem studied in this work, this does not influence the result of the queries, since Ignite processes a certain data replicated only in one of the instances. Another allowed strategy to make Ignite resilient is to move less accessed data in memory to disk, allowing memory consumption to be reduced. In addition, Apache Ignite can be used as a caching layer for other frameworks [15], such as Apache Spark and Apache Flink [3]. In this case, operations on Ignite data are mapped to operations directly on data in the database and, if a join between
The quick validation of the existence of data in a set of files allows control of data in a repository at the record level. This is possible due to the use of Bloom filters [2] on the metadata of a file, which makes it possible to probabilistically verify whether data belongs to a given file (with exact guarantees in case the filter returns that the data does not belong to the file). This quick validation of the existence of data in a set of files allows Hudi to provide primitives for updating and removing data in files based on versions. A new occurrence of known data (or its removal) will result in a new line in the current file, with an incremented revision number.

4 EVALUATION

This section describes the entire evaluation process for the methods used, including configuration, architecture, test composition and results.

4.1 Experimental Setup and Architecture

This section details the architecture of the pipeline used by the tests performed, evidencing the infrastructure and tools used in each layer. The tests were performed on the machines available by AWS. The specifications of these machines are in Table 2.

As can be seen in Figure 2, the layers of the architecture are: ingestion, processing, data storage and access to the data. Each of these layers is detailed below.

4.1.1 Data source and ingestion layer. Data ingestion starts by consuming streams of messages, originating from data source connectors. Connectors are responsible for extracting data change events from a data store (e.g. database system) and writing them to the stream consumed by the data pipeline. These events are sent via messages for each data change (insertion, update, or deletion).

The broker responsible for receiving messages from connectors and assigning them to a message queue, based on identifiers, is RabbitMQ. This broker is only used to reproduce the company’s current scenario. The company’s main messaging is Apache Kafka, which makes it our only source for events.

Subsequently, consumers, implemented in python and deployed on a Kubernetes’ cluster, consume the messages. The main objective of consumers is to pass the data records in batches to an Apache Kafka cluster provisioned for this pipeline.

The motivation to persist the records in a Kafka cluster before processing, instead of persisting them directly in the data repository or processing them directly is justified by the fact that Apache Kafka mitigates the following points: (i) Fault tolerance native to Apache Kafka in which messages are not deleted after being consumed for the first time, allowing for a re-reading of lost messages in case of failure in the processing layer (Requirement 4); (ii) Partitioning into topics makes it possible to control the parallelism of the application that will consume the messages, since the application can instantiate up to a maximum of n consumers, where n is the number of partitions of a topic; (iii) Message keys allow Kafka to send messages with the same key used for deduplication to the same consumer, which will avoid the duplicate data between consumers; (iv) Most of the tools for implementing streaming processes have native primitives for reading data in real time from Apache Kafka.

Three AWS instances of type m5.large were chosen for the Apache Kafka cluster, because their operations are IO-bound, that is, they have more data input and output load than processing (CPU-bound). For each instance, an SSD (Solid State Drive, i.e., solid state disk) with 2 TB of storage was allocated, and each of these disks had 81.2% (i.e., 1.624 TB) of its capacity used by the mass of test data from this work.

This high disk usage is explained by having configured a policy of message deletion of fourteen days, the same amount of days analyzed in the tests, and for each partition of the topic to have a replication factor equal to three. This replication allows all nodes of the cluster to have all the messages in Apache Kafka, avoiding the loss of large blocks of messages in case of loss of any node. All consumed messages refer to only one topic, partitioned into thirty pieces. This number of partitions was chosen for greater parallelism and to balance the number of consumers instantiated by streaming for each node in the processing layer cluster.

The reading of the topic partitions is centered on the main partition, and not in a replica, which allows Apache Kafka to also distribute the number of requests among the nodes of the cluster. For example, in this cluster, each machine had up to ten partitions being consumed simultaneously.

4.1.2 Processing layer. The base framework for this layer is the Apache Spark, which is widely used in the company’s data transformation processes. The processing layer is responsible for the execution of the operators streaming. During the tests, all the resources of the cluster are used exclusively for only one streaming application.

The ETL performed on this layer can be summarized in the following steps: (i) Consume the messages, in JSON format, from Apache Kafka (Ingestion Layer); (ii) Deduplicate messages based on their consumed keys and keys persisted in the datastore; (iii) Extract the message value (in JSON format) applying a schema; (iv) Persist the data in the data repository in a compressed and column-optimized format. The format used was Apache Parquet.

For each test performed, the deduplication logic defined in step (ii) is changed, varying the deduplication functions, tools and auxiliary infrastructure components.

To create the processing cluster instances, AWS EMR was used, which provides all the ETL tools that were used in the tests (with the exception of some auxiliary databases), removing the responsibility of configuring all the details of each tool and speeding up the start of each test.

---

Table 2: Specification of AWS machines used in tests

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th>Cores</th>
<th>Quantity</th>
<th>RAM Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>m5.large</td>
<td>Intel Xeon 8175</td>
<td>2</td>
<td>8 GB</td>
<td></td>
</tr>
<tr>
<td>m5.xlarge</td>
<td>Intel Xeon 8175</td>
<td>4</td>
<td>16 GB</td>
<td></td>
</tr>
<tr>
<td>r5.xlarge</td>
<td>Intel Xeon 8175</td>
<td>4</td>
<td>32 GB</td>
<td></td>
</tr>
<tr>
<td>r5.2xlarge</td>
<td>Intel Xeon 8175</td>
<td>8</td>
<td>64 GB</td>
<td></td>
</tr>
</tbody>
</table>

---

For this cluster, a machine of type r5.xlarge is used for master, with four vCPU’s and 32 GB of RAM memory. For the slave machines, six r5.2xlarge machines are used, doubling the amount of vCPU’s and RAM memory, totaling 48 vCPU’s and 384 GB of RAM memory. These machines were chosen due to the high consumption of RAM memory used during the execution of the ETL, especially when deduplication process need to persist the keys used for deduplication between moments of the stream.

Due to each test’s exclusive use of the cluster and the thirty topic partitions of the Apache Kafka, streaming applications were able to instantiate five consumers in each Cluster slave machine with a processing core and up to 15 GB dedicated to running the ETL on the consumed data.

As the results were persisted to an external repository, it was not necessary to allocate a high-capacity SSD as in the ingestion layer. However, persistence of the checkpoints, produced at the end of the processing of each micro-batch, was performed locally to avoid any latency in accessing this data. Also, when a new checkpoint was generated, a parallel process was also started to backup this data in an external repository.

4.1.3 Data storage and the access layer. The data used is from November 2019 (Figure 1). The data stored in S3 is persisted by the ETL in a partitioned way by the event’s creation date. This allows queries to this repository’s data not to need to access all persisted objects, in case the query only needs a subset of dates, which optimizes the data consumption and decreases query execution time (Requirement 5).

However, as much as the data persisted in this repository has the definition of its schema in the metadata of the objects, the query engine used (Apache Presto) cannot access the data in the fetch if it is not directed to the full path of the desired object. This factor makes it difficult to write queries that handle with more than one object. In addition, the user to be aware of how the data repository is organized. To solve this difficulty, and create an abstraction for the user that s/he is interacting with a traditional relational database, that is, with abstractions of schemas, tables and columns, Apache Hive is used. Apache Hive is used only as a metastore, allowing Apache Presto to query the data store. Apache Presto has necessary integrations with other relational and NoSQL data stores needed by the enterprise.

The queries carried out by Apache Presto are performed only via the command line or via an API call, which may make queries difficult for a user who does not know how to use either of these two access methods. For this reason, Redash was used as a graphical interface application so that the user has a more familiar access to an IDE (Integrated Development Environment), in addition to being able to create dashboards of metrics known as simple dashboards.

As for the infrastructure used by these two components: Apache Presto uses a cluster of ten r5.2xlarge machines (8 vCPU’s/64 GB) coordinated by an r5.xlarge (4 vCPU’s/64 GB); Redash uses only one m5.xlarge machine (4 vCPU’s/16 GB).

4.2 Evaluation Scenarios

This section specifies the six test scenarios evaluated.

T0: Baseline

Tests run with just the logic to read Kafka messages and persist to S3. The objective of this scenario is to understand how much the deduplication logic impacts the metrics of the other tests.

T1: Deduplication with Apache Spark’s distinct operator

The comparison with the distinct operator between two rows will stop at the first pair of different columns. However, for dataframes with multiple columns, as in the messages processed in the tests of this work, the comparisons may take a long time if the difference is found only in the last columns compared. This happens because the operator requires to be aware of all the columns of each row of the dataframe to remove duplicates. The entire row must be stored in the streaming state to possible duplicates of future micro-batches are not persisted in the data repository. It will result in a high memory usage by the application, impacting requirement 2.

T2: Deduplication with Apache Spark’s dropDuplicates operator

The dropDuplicates operator uses fewer columns when compared to the distinct operator. The runtime of dropDuplicates is

---

11When we carry out this work it was called Apache Presto, being rebranded later as Trino.
expected to be considerably less than distinct for dataframes with many columns and many rows. This implies less memory consumption, as only the dataframe keys need to be persisted between micro-batches. However, this operator can only be used if the application domain declares that these keys exist for all data processed by the application. Therefore, the use of this operator will not always be an alternative to the distinct operator.

**T3: Deduplication with Apache Spark dropDuplicates operator with persistence status in RocksDB**

A shortcoming of Apache Spark is the in-memory-only persistence of the state of the streaming, which implies the loss of this data in case of a failure to stop the execution of the application. Through RocksDB, Spark may persist the state data, at the end of each micro-batch, in the database instead of using the application’s memory space. This allows the application to recover the state of a previous run if the application is restarted. To prevent data from traveling over the network, RocksDB was instantiated on each slave machine, allowing the application to achieve superior performance by avoiding increased latency with an external connection to the cluster.

**T4: Deduplication with Apache Ignite**

By having native integration with Spark, Apache Ignite allows operations on the data of a dataframe, stored in Ignite, to take place in the memory space of the database. The advantage is evident when there is a need to join a Spark dataframe and an Ignite dataframe. In this case, only rows from Ignite that meet the join criteria are projected to Spark, thus decreasing data traffic between memory spaces.

In addition, the replication of data between nodes of an Ignite cluster prevents a set of rows of the state of a stream from being lost, in case one of the nodes of the cluster is unavailable, making the application tolerant to software and hardware faults. For this test, we configured the use of disk storage to contain all rows consumed by all micro-batches, and to keep the most recently accessed data in memory. To allow fast data access and avoid network traffic between nodes, Ignite instances were launched on Spark slave nodes. In addition, replication has also been configured to maintain a copy of all state rows across all Ignite instances.

As Ignite has the keys of micro-batches already processed, it was not necessary to use any Spark native operator to perform the deduplication. This operation being performed by an SQL query using Spark’s LEFT ANTI JOIN syntax, interoperable with Ignite. This allowed merging the micro-batch Spark dataframe with the keys persisted in Ignite, to retrieve new rows from the current micro-batch, without the need to use Spark memory to store the application state.

**T5: Deduplication with Apache Hudi**

The Apache Hudi file format allows the application that produces files in this format determine the number of versions of data that will be persisted. In this test, the application was configured to persist only one version, eliminating the possibility of duplicate data. As there are no updates or deletions in this test, data was persisted using the insert method, with duplicate validation, in files with a maximum size of 128MB (default Hudi configuration), reducing the number of files consulted.

### 4.3 Evaluation Results

Table 3 summarizes the evaluation results by comparing the criteria used and provides some comments that help in choosing the data deduplication approach. Details in the following subsection.

We run the experiment 5 times for each scenario and show the average of these values in tables and graphs. To avoid some memory accumulation between tests, a cluster of the processing was provisioned for each run of each test. At the end of streaming, the cluster is deleted.

#### 4.3.1 Memory Usage

Based on requirement 2, the use of RAM between the test scenarios was analyzed, based on the metrics available in the logs of each stream at the end of each micro-batch. This analysis considered total memory usage, including memory usage by Spark operators, memory usage by application state, and memory usage by databases used in tests T3 and T4.

The graph in Figure 3 shows the average memory usage of the five executions of each test performed, where the x-axis represents the execution order of the micro-batches and the y-axis represents the total amount of memory used by each micro-batches in the tests performed. In this graph, we can see a rapid growth in memory usage when using the distinct operator in the T1 test. This is due to the high number of columns in the processed data, which implies a greater amount of comparisons between data and columns to store in the application state between micro-batches. As there is a higher percentage of duplicate data (Figure 1), this implies that comparisons are performed on all columns, multiple times in most cases, without any interruptions for distinct values between two columns. Due to the excess of memory used in the tests carried out with this operator, an out-of-memory error occurred during the processing of the 18th micro-batch. This behavior was expected since its memory use after the first micro-batch was higher than the use of all the micro-batches among the other tests.

![Figure 3: Memory usage between micro-batches of tests performed.](image)

Removing the T1 test and adjusting the y-axis scale, we obtain the graph in Figure 4. Just by changing the use of the distinct operator, by the dropDuplicates operator in the test T2, there was a 92% reduction in RAM usage compared to the end of both tests. In addition, it was also possible to finish the execution of all micro-batches and keep growth in memory usage on a linear scale.

Using RocksDB as the state repository in the T3 test allowed dropDuplicates to reduce the amount of memory used between
micro-batches to store state. However, the application’s memory usage became unstable, as querying RocksDB data for processed data kept more comparison operations in memory.

In the T4 test, where Apache Ignite was used, the highest memory usage occurred among the tests, excluding the T1 test. However, this test’s memory usage was largely utilized by Ignite, as the responsibilities of validating the preexistence of keys and storing previously processed keys became the responsibility of the database. Finally, the T5 test had the lowest memory usage among all tests, remaining below 5GB in all micro-batches, maintaining memory usage similar to baseline. This is because Hudi doesn’t need to persist any historical data on the stream state, just like the baseline.

4.3.2 Execution time of micro-batches. Meeting the third requirement, the logs were analyzed to evaluate the execution time. This analysis considered the total time of each micro-batch, from the moment it started to be processed, until the end of data persistence in the data repository.

The graph in Figure 5 shows the number of generated micro-batches (x axis) and the average running time of each micro-batch in minutes (y-axis).

Due to the greater number of comparisons, the distinct operator had the worst performance in terms of execution time, exceeding the 10-minute limit defined by the application domain. In the 12th micro-batch of the T1 test, the longest execution time occurred for the distinct operator. This occurred because the data processed in this micro-batch had the highest concentration of new data, causing more comparison between micro-batch and state elements.

In contrast, the dropDuplicates operator in the T2 test obtained the best performance. The average time was 2 minutes and 31 seconds, even shorter than the baseline (4 minutes and 22 seconds). This is because deduplication reduces the amount of data that will be persisted in the repository. In this way, there is a smaller amount of comparisons. In addition, the operator does not save the state in storage external to Spark, which also explains the lower average execution time.

An important aspect to be highlighted is that even the process without baseline deduplication persisting a larger volume of data compared to all other tests, the execution time of the micro-batch is not significantly impacted. This is due to the absence of deduplication logic. Whichever deduplication logic is implemented, a shuffle of data will occur between cluster nodes. Deduplication aims to ensure that data is unique, which considerably increases the execution time of the micro-batch.

T3 and T4 tests use external databases (RocksDB and Apache Ignite, respectively), obtaining results similar to the baseline, with average time of 4 minutes and 30 seconds. This time remained low, because even with disk queries, the databases were instantiated in the same cluster as Spark, decreasing external network traffic.

As Apache Hudi has to perform multiple queries to the datastore located outside the cluster, in S3, the T5 test had the second worst performance, with an average time of 11 minutes and 32 seconds, considerably over the SLA time limit. We do not consider the comparison between the RocksDB (local storage) and Hudi (S3) scenarios to be totally fair. However, using Hudi with local storage would not be scalable and would only guarantee the uniqueness of local data and not the entire dataset. Furthermore, the logic used by the library that integrates Hudi with Spark is treated as a black box. Thus, changing the behavior of this step would not be trivial.

4.3.3 Fault tolerance. In this discussion, fault tolerance is separated into two perspectives:

- Application failures: occurrence of an unhandled or expected error or exception in the scope of the application. For example, the need for a larger amount of memory or division by zero;

Table 3: Comparison between the studied solutions

<table>
<thead>
<tr>
<th>Solution</th>
<th>RAM Usage</th>
<th>Execution Time</th>
<th>Query Time</th>
<th>Fault Tolerance</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinct</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>None</td>
<td>Don’t use in streaming</td>
</tr>
<tr>
<td>dropDuplicates</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>None</td>
<td>Acceptable % of duplicates</td>
</tr>
<tr>
<td>RocksDB</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Application</td>
<td>More fault tolerance preferred</td>
</tr>
<tr>
<td>Ignite</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Application/Infrastructure</td>
<td>Data uniqueness</td>
</tr>
<tr>
<td>Hudi</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Application/Infrastructure</td>
<td>Data uniqueness and low query time</td>
</tr>
</tbody>
</table>

Figure 4: Memory usage between micro-batches of tests performed, except T1.

Figure 5: Execution time (in minutes) between micro-batches of tests performed.
• Infrastructure failures: failure or removal of a hardware component. For example, failure of a hard disk or memory module, removal of a Spot Instance from the cluster, among other failures.

We did not create artificial resource failure scenarios of the instantiated clusters for the test scenarios due to the unpredictability of infrastructure errors in a cloud environment. However, due to the characteristic behavior of each solution studied, we can discuss how each one is capable of dealing with an unexpected failure, should it occur.

As they are native Spark solutions that persist data only in the application’s (volatile) memory space, the distinct and dropDuplicates operators are not considered fault-tolerant, since if the application is terminated due to an unhandled error, all data in memory will be lost.

The RocksDB solution is considered to be application fault tolerant. Any unexpected closure of the application, the data will be persisted on disk. However, as there is no data replication, in the event of a cluster node being terminated, the database does not provide any mechanism for recovering data from this disk. One way to solve this deficiency of RocksDB, in cloud environments, is the use of disks attached to machines via network, which can be mounted to other instances, in case the original instance fails. Note that the process of mounting a volume to a new RocksDB instance is time consuming and non-trivial, causing slow recovery times.

Apache Ignite data replication allows any instance that owns the data can return it in a query. If a cluster node shuts down, this data is made available by other nodes while a new node is provisioned. As Ignite can operate in a cluster separate from Spark, the termination of Ignite or Spark cluster nodes will not generate inconsistency in the data used by the ETL, making the process more fault tolerant, ensuring the fulfillment of requirement 4.

Apache Hudi keeps the keys of all records persisted in the generated files itself, which guarantees that: if the entire Spark cluster is terminated, restarting the stream on a new cluster will consider all keys already known. This guarantees the uniqueness of the data to be persisted. In this way, Hudi can be classified as the most fault-tolerant solution (both application and infrastructure) tested in this work.

4.3.4 Size and number of generated files. To allow the execution time of queries in the access layer not to be harmed, this work prioritizes the generation of larger files in smaller quantities compared to generating a large number of small files, as presented in requirement 5.

Apache Spark generates as many files as there are dataframe partitions. This amount is related to the parallelism configured by the framework. There is no native Spark mechanism that allows the final data of these partitions to be added to existing files. As a result, ETL processes that process little data in a short period of time end up generating smaller files.

In addition to this problem, adding a partition agglomeration step to the ETL (coalesce) will significantly increase the ETL execution time, even though it avoids the shuffle. The comparison of the total amount of files generated by the average of five executions of each test, as well as the average of the minimum, maximum, average and total sizes, are presented in Table 4 and Table 5, respectively.

Note that the distinct operator was not able to process all the data due to lack of memory to store the state between microbatches, which is why the T1 test obtained great metric values in these tables. For that same reason, there were fewer files with varying sizes, getting better mins, maxes, and averages.

In the baseline scenario, a smaller amount of files is generated in the total (slightly more than double, compared to tests T2, T3 and T4). This is due to the way Kafka data is consumed, with a consumption process instantiated by Spark for each topic partition (in this case, thirty partitions). As there is no need for subsequent operations, in addition to persistence on the destination, Spark does not perform shuffle operations and everything that is consumed by a consumer is persisted in the same file.

Using RocksDB in the T3 test for state persistence with the dropDuplicates operator also did not improve the way Spark persisted the files, compared to the T2 test, obtaining values in the metrics that were very close, being considered equivalent.

Using Ignite in the T4 test, it was possible to reduce the number of files compared to the T2 test by 65,052 units (22.3%). This reduction also resulted in the total size of processed data decreasing by 18.1 GB (16.2%). This was due to the LEFT ANTI JOIN used in Ignite helping to compare partitions with the same key between Spark nodes. In this way, the most similar dataframe rows could be stored in the same partition, resulting in a better file compression rate when being persisted in the data repository.

The T5 test obtained the best result in this requirement with the least amount of files generated. These files averaging 114.3 MB, close to the limit of 128 MB (Apache Hudi default). This was because Hudi natively clusters data in existing files, creating files only when a configured threshold is reached, thus allowing less metadata to be created.

Observe that the minimums, maximums and averages between the size of the files of the tests T2, T3 and T4 are very close, affecting the execution time of the queries on these files in a similar way.

To analyze how much the number of files influences the amount of metadata generated, the result of the T2 test was compressed by a batch process using the repartition operator, with the objective of bringing the size of the files closer to 128 MB. Spark does not have any method of clustering partitions which is based on the final size of the compressed files. Therefore, this process was done with the best possible approximation.

Then, the same process was performed again, to approximate the size of the files to 64 MB. The results of these two processes, compared to the T2-T5 tests are shown in Table 6 and Table 7.

### Table 4: Total amount of files generated by each test

<table>
<thead>
<tr>
<th>Test</th>
<th>T0: Baseline</th>
<th>T1: distinct</th>
<th>T2: dropDuplicates</th>
<th>T3: RocksDB</th>
<th>T4: Ignite</th>
<th>T5: Hudi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>15,4KB</td>
<td>494,4KB</td>
<td>71,1KB</td>
<td>56,1KB</td>
<td>114,4KB</td>
<td>2.55MB</td>
</tr>
<tr>
<td>Medium</td>
<td>2.5MB</td>
<td>1.4MB</td>
<td>668,8KB</td>
<td>671,3KB</td>
<td>746,1KB</td>
<td>114,3MB</td>
</tr>
<tr>
<td>Maximum</td>
<td>21.7MB</td>
<td>3.1MB</td>
<td>2.2MB</td>
<td>2.2MB</td>
<td>2.4MB</td>
<td>122.5MB</td>
</tr>
<tr>
<td>Total</td>
<td>323.9GB</td>
<td>73.9GB</td>
<td>110.5GB</td>
<td>111.5GB</td>
<td>93.4GB</td>
<td>86.2GB</td>
</tr>
</tbody>
</table>

* Test not completed

### Table 5: Size of files generated by each test

<table>
<thead>
<tr>
<th>Size</th>
<th>T0: Baseline</th>
<th>T1: distinct</th>
<th>T2: dropDuplicates</th>
<th>T3: RocksDB</th>
<th>T4: Ignite</th>
<th>T5: Hudi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>15,4KB</td>
<td>494,4KB</td>
<td>71,1KB</td>
<td>56,1KB</td>
<td>114,4KB</td>
<td>2.55MB</td>
</tr>
<tr>
<td>Medium</td>
<td>2.5MB</td>
<td>1.4MB</td>
<td>668,8KB</td>
<td>671,3KB</td>
<td>746,1KB</td>
<td>114,3MB</td>
</tr>
<tr>
<td>Maximum</td>
<td>21.7MB</td>
<td>3.1MB</td>
<td>2.2MB</td>
<td>2.2MB</td>
<td>2.4MB</td>
<td>122.5MB</td>
</tr>
<tr>
<td>Total</td>
<td>323.9GB</td>
<td>73.9GB</td>
<td>110.5GB</td>
<td>111.5GB</td>
<td>93.4GB</td>
<td>86.2GB</td>
</tr>
</tbody>
</table>

* Test not completed
An exploratory analysis of methods for real-time data deduplication in streaming processes DEBS '23, June 27–30, 2023, Neuchatel, Switzerland

Through these two processes, the amount of metadata generated considerably reduces the total size occupied by the files persisted in the data repository. However, there is a limit to reduce the total size of the files due to the size of each file. Files closer to 64 MB occupy 4.5 GB (5.1%) less compared to files closer to 128 MB. These two processes show the benefit of consolidating the streaming files from the T2 test. Although consolidation is beneficial, as the streaming application is designed to never end, there is no ideal time to perform this consolidation. Likewise, consolidating the data at the end of each micro-batch would not be beneficial, since as the number of files grows, this step would increase the micro-batch execution time considerably by having to parse all the persisted data again. In addition, this strategy may also cause errors in queries that occur simultaneously with this consolidation. There is no mechanism to lock the data for reading while the files are consolidated. Without locking, the query can lose reference to the file being analyzed at the time the file is deleted and recreated by the consolidation process.

Apache Hudi performs considerably better in this regard, as the format will only add data to the end of the respective files, without removing them. In addition, access will only occur to necessary files and not to all data persisted. Even so, Hudi has a very high variance in the sizes of the generated files during the consolidation process. This is because Hudi runs a best effort policy to reduce file sizes. Compared with consolidating files to 128MB, Hudi achieved a smaller amount close to generated files in total, as well as a similar total size. Observe that the amount of metadata inserted by Hudi about the Parquet file format, which is used internally by Hudi, is minimal.

4.3.5 Query time. Using Apache Presto to access the data on S3, it was possible to analyze the time execution of queries on data persisted by each test performed, except test T1. In addition, tests were performed to verify the query time in the original batch ingestion process.

Five queries commonly performed by analysts of the classifieds company were used, obtaining the final execution time (hh:mm:ss) of each micro-batch would not be beneficial, since as the number of micro-batches increases, the amount of data generated by streaming.

The times obtained in queries carried out in tests T2, T3 and T4, made it impossible to use the respective methods as an alternative to the original process. The execution time of queries 2, 3 and 4 exceeded the maximum time that was configured in company's Presto cluster. This is due to the number of files to be accessed by Presto, highlighting the need to consolidate the number of files generated by streaming.

On the other hand, due to the reduced amount of files produced in the T5 test, the times obtained were lower than the original process, making it ideal for replacing the batch process and meet the requirement to reduce query time (requirement 5).

Regarding the baseline, the queries were modified to perform deduplication on top of the result obtained. Note that in query 5 this mass of data obtained a shorter time than tests T2, T3 and T4. This fact occurs due to the smaller amount of files generated in the baseline, as well as the type of the query itself. The query returns a count performed over a range of partitions that contain more compressed files. Already compared to query 5, the baseline had the worst performance due to this query demanding many comparisons with the OR-conditional operator, about a larger amount of data in total.

4.4 Result Analysis

Based on the results obtained, the choice of which method to use should be based on which requirement should be prioritized, with the exception of the distinct method, which is not suitable for streaming applications. The criteria are summarized in Table 3.

Table 6: Number of consolidated files - T2 to T5

<table>
<thead>
<tr>
<th>T2: dropDuplicates</th>
<th>T3: RocksDB</th>
<th>T4: Ignite</th>
<th>T5: Hudi</th>
</tr>
</thead>
<tbody>
<tr>
<td>291.633</td>
<td>1.237</td>
<td>4.280</td>
<td>1.451</td>
</tr>
</tbody>
</table>

Table 7: Size of consolidated files - T2 to T5

<table>
<thead>
<tr>
<th>T2: dropDuplicates</th>
<th>T3: RocksDB</th>
<th>T4: Ignite</th>
<th>T5: Hudi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>71.1KB</td>
<td>130.8MB</td>
<td>61.0MB</td>
</tr>
<tr>
<td>Medium</td>
<td>668.8KB</td>
<td>133.6MB</td>
<td>67.3MB</td>
</tr>
<tr>
<td>Maximum</td>
<td>2.3MB</td>
<td>135.9MB</td>
<td>64.1MB</td>
</tr>
<tr>
<td>Total</td>
<td>110.5GB</td>
<td>88.0GB</td>
<td>83.5GB</td>
</tr>
</tbody>
</table>

Table 8: Execution time of queries on the final test data

<table>
<thead>
<tr>
<th>Query</th>
<th>T0: Baseline</th>
<th>T1: Batch</th>
<th>T2: dropDuplicates</th>
<th>T3: RocksDB</th>
<th>T4: Ignite</th>
<th>T5: Hudi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00:20:48</td>
<td>00:05:20</td>
<td>00:56:29</td>
<td>00:56:01</td>
<td>00:49:58</td>
<td>00:01:32</td>
</tr>
<tr>
<td>2</td>
<td>+05:00:00</td>
<td>00:01:25</td>
<td>+05:00:00</td>
<td>+05:00:00</td>
<td>+05:00:00</td>
<td>00:01:11</td>
</tr>
<tr>
<td>3</td>
<td>+05:00:00</td>
<td>00:06:01</td>
<td>+05:00:00</td>
<td>+05:00:00</td>
<td>+05:00:00</td>
<td>00:03:06</td>
</tr>
<tr>
<td>4</td>
<td>+05:00:00</td>
<td>00:06:13</td>
<td>+05:00:00</td>
<td>+05:00:00</td>
<td>+05:00:00</td>
<td>00:03:12</td>
</tr>
<tr>
<td>5</td>
<td>01:15:50</td>
<td>00:02:38</td>
<td>00:59:39</td>
<td>00:58:42</td>
<td>00:50:47</td>
<td>00:01:29</td>
</tr>
</tbody>
</table>

(1) SELECT count(dtt)
FROM <table_name>
WHERE dt BETWEEN date ‘2019–11–01’ AND date ‘2019–11–30’
AND delete_reason = ‘user_account_deleted’

(2) SELECT ad_id, lifecycle_id, event_timestamp
FROM <table_name>
WHERE dt = date(‘2019–11–23’)

(3) SELECT ad_id, lifecycle_id, body, subject
FROM <table_name>
WHERE dt = date(‘2019–11–23’)
AND (regexp_like(body, ‘(?i)(((S|S))(A))[(d.*GB memory)][1,3]’)
OR regexp_like(subject, ‘(?i)(((S|S))(A))[(d.*GB memory)][1,3]’))

(4) SELECT date(timestamp) date, count(1)
FROM <table_name>
WHERE dt BETWEEN date(‘2019–11–22’)
AND date(‘2019–11–24’)
AND lifecycle_id IN (20, 52)
GROUP BY 1

(5) SELECT count(1)
FROM <table_name>
WHERE lower(subject) LIKE ‘$surgical_glove%’
OR lower(subject) LIKE ‘$mask%’
OR lower(subject) LIKE ‘$alcohol_gel%’
If there is no need to remove duplicates between micro-batches in a fault-tolerant way (for example, when an SLA term allows a low percentage of duplicates), the developer may use the dropDuplicates operator without the aid of any solution that persists the state of the application outside the scope of Spark, achieving maximum performance.

The use of an external database, such as Apache Ignite and RocksDB, for persistence of the keys used by the deduplication process did not significantly affect the streaming performance, which guarantees greater fault tolerance. If the database used can run on a cluster and has a data replication mechanism between nodes that can run in parallel, there is an even greater increase in the fault tolerance of the application. For this reason, Apache Ignite has an advantage compared to RocksDB.

In addition, if the SLA time is negotiable or the amount of data processed per micro-batch can be reduced, Apache Hudi is the most recommended for guaranteeing the uniqueness of the data internally in the metadata of the persisted files, and for reducing the number and total size of files.

As a lesson learned, developers always need to measure the trade-off between application performance by externalizing the state needed to maintain correct streaming and the required accuracy itself. If the accuracy is negotiable, an approach in which the lost state is not recovered (stateless streaming) becomes acceptable. Another alternative is to use probabilistic algorithms, such as the bloom filter, to maintain a smaller state and decrease execution time and memory usage.

5 RELATED WORK

Related work can be seen from two perspectives: data deduplication and state management of streaming applications.

5.1 Data deduplication

The reviewed works on data deduplication, for the most part, invest in the data deduplication when they are already persisted on disk. The objective is to reduce the volume occupied by stored files, as well as to reduce the amount of metadata associated with these files.

The work by Debnath et al. (2010) presents an analysis on how the use of disks based on flash memories can improve disk access during the inline deduplication process [5]. This occurs during the disk write operation, reducing the disk query time. The tests performed used an index proposed by Zhu et al. (2008) [21], based on BerkeleyDB and Bloom filters in environments using both traditional and flash-based hard disks. As a result, ChunkStash obtained a performance 36 times higher than that proposed by Zhu et al. (2008) running on a hard disk (HD) and 3.5 times higher when running on flash memory.

The work by Srinivasan et al. (2012) also analyzed the deduplication process inline, with the aim of proposing a solution that would allow deduplication to occur with the least possible impact on disk access operations performed by the data source application [16]. Through the use of two data properties, it was possible to reduce the amount of disk space and file fragmentation, optimizing future read operations of these files and reducing the amount of generated metadata.

The work by El-Shim et al. (2012) studied file deduplication processes on Windows Server systems from some companies to serve as a basis for architcting a new deduplication engine in Windows Server 2012 [7]. It was found that comparing by chunks resulted in the best disk space savings, obtaining results ranging from 2.3x to 15, 8x reduced footprint. However, the performance gain in reducing the size of the chunks starts to be canceled out by the loss of the data compression factor, since in smaller chunks there is a low compression rate obtained. For this reason, according to the authors, the result obtained using 4KB chunks is identical to 64KB chunks.

The work by Zhang et al. (2017) implemented a deduplication system for files stored in HDFS, using Hadoop MapReduce to implement the steps of aggregation, division and compression in file hashes, and Apache HBase as a database for indexing the hashes [20]. With larger files, the MapReduce application obtained a significant performance gain. A limitation of this practice is that due to the aggregation of smaller files not considering their context, accessing the data in the resulting files requires consulting the index created in HBase so that only the pieces of data needed by the user are extracted.

Recent studies [13][8][1] propose new approaches for data deduplication. Lin and colleagues (2023) address the problem of chunk fragmentation, generated by the deduplication process when removing duplicate data, which degrades the restore performance of storage systems [13]. Elouataoui et al. (2022) propose a deduplication framework based on a semi-supervised learning approach, in the context of big data, and with concern for the performance of the deduplication model during the serving [8]. Azeroual et. al. (2022) propose a record linkage-based data deduplication framework through the extension of an existing tool [1]. The authors of the last two works specify the dataset according to the number of records and not the size occupied on disk.

The differential of our work is that the deduplication process occurs at ingestion time. Related works perform deduplication after persisted data. Although data deduplication generates savings in storage space, the focus of our work is on the exploration of this process as a way to meet a data domain requirement, something that is not explored by other works. Related works aim to explore internal processes of the system operating system or a special tool. This work performs an exploratory analysis of several solutions that may help in the deduplication process, comparing the results obtained and identifying the use cases of each solution. Table 9 summarizes the comparison among the deduplication works.

5.2 State Management

This work analyzed the native mechanisms of Apache Spark for state management of a streaming application, as well as the use of other tools to help maintain this state in case of an unexpected termination of the deduplication process. These tools are not necessarily developed specifically for this purpose, however they were successful in ensuring that the keys needed for the deduplication process were not lost, without the main process suffering a major impact on micro-batch delivery time or usage of cluster resources.

In the literature, there are works that discuss the management mechanisms native to other data processing frameworks, such as
Table 9: Related Work - Data Deduplication

<table>
<thead>
<tr>
<th>Work</th>
<th>Data sample</th>
<th>Used Solutions</th>
<th>Deduplication Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debnath et al.(2010)</td>
<td>8GB, 32GB, 126GB</td>
<td>ChunkStash</td>
<td>Data already persisted</td>
</tr>
<tr>
<td>Srinivasan et al.(2012)</td>
<td>396GB read, 172 GB written</td>
<td>iDedup</td>
<td>Internal process of Windows Server 2012</td>
</tr>
<tr>
<td>El-Shimi et al.(2012)</td>
<td>6.8TB</td>
<td>Hadoop MapReduce</td>
<td>Data already persisted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hadoop HDFS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Apache HBase</td>
<td></td>
</tr>
<tr>
<td>Zhang et al.(2017)</td>
<td>600MB</td>
<td>Hadoop MapReduce</td>
<td>Data already persisted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F-greedy+ (Proposed improvement)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>F-greedy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capping</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMR</td>
<td></td>
</tr>
<tr>
<td>Lin et al.(2023)</td>
<td>105GB, 47GB, 120GB</td>
<td>proposed framework</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SDLER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DeepER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dedupe</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Magellan</td>
<td></td>
</tr>
<tr>
<td>Elouataoui et al.(2022)</td>
<td>864, 663000, 1001300</td>
<td>proposed framework</td>
<td>Data already persisted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DataCleaner</td>
<td></td>
</tr>
<tr>
<td>Azeroual et al.(2022)</td>
<td>5000+</td>
<td>Apache Kafka</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Apache Spark</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Apache Ignite</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Apache Hudi</td>
<td></td>
</tr>
<tr>
<td>Our work</td>
<td>1.6 TB</td>
<td>Data Ingestion</td>
<td></td>
</tr>
</tbody>
</table>

SAND [14] and Apache Flink [3], as well as works that propose managers with high availability and fault tolerance that are agnostic to the framework or programming language used, such as Megaphone [11] and Rhino [6]. In addition, there is an approach to mitigate the surge of memory demand from the processes running stateful streaming applications [12]. Table 10 summarizes the comparison among the state management works.

The SAND was developed with the objective of carrying out the data processing that travel over the network that allows handling a high volume of data and is fault tolerant. The resource limitation is given by the need to run the applications close to the origin of the network packets, which motivated the authors to use the C++ programming language to develop the data processor. As for fault tolerance, SAND removes state persistence from the scope of the application and transfers it to a Zookeeper cluster, which resumes the application to the last state in case of restart.

Apache Flink manages the state of a stream through internal snapshot creation routines of the entire application state that are executed at the end of each processing window, allowing these snapshots to be persisted both in a file system (of the cluster of processing or external), and in external databases.

Megaphone and Rhino are state management solutions agnostic to data processing frameworks, making it possible to use any of these solutions in an application that is implemented without the aid of any frameworks. Both solutions specialize in redistributing keys persisted into the application state when nodes are added or removed from a cluster. This approach is employed while the application is running. This improves fault tolerance, as if a node becomes unavailable, a new node can replace it.

Del Monte et al. (2020) compared the time of these three stages between Rhino, Megaphone and the native solution used by Flink with samples of 250GB, 500GB, 750GB and 1TB. In this comparison, Rhino obtained an average performance 50x greater than Flink and 15x greater than Megaphone, with the exception of the two largest loads, where Megaphone failed due to lack of memory.

Islam et al. (2022) proposed a multi-level caching architecture to deal with the memory consumption demands coming from stateful streaming applications. Although the work proposal is promising, this solution does not meet the problem raised in this work. The proposed cache layer could not be applied to Apache Spark because the dataframes data are outside the scope of the main application and require meticulous management of Spark itself, which even has its own in-memory cache system. In terms of technologies used in architecture and comparing them to our work, Islam et al. (2022) used Redis instead of Ignite and Storm instead of Spark.

Furthermore, the authors of the related work do not make it clear whether Redis was instantiated on the same nodes where the application runs, which could indicate poor performance. In our experiment, Ignite nodes were instantiated on the same Spark nodes. Additionally, Ignite was configured to keep a copy of all keys on all nodes, which ensured that no network calls were made at the time of deduplication at the cost of Ignite sending copies between nodes in the background.

Another advantage of our approach is the integration of Ignite with Spark through an API developed by the Ignite team itself. Using the native API allows the abstraction of Ignite dataframes interacting with Spark. In this way, the Spark optimizer (Catalyst) is able to generate the query execution plan considering the location of the data, including executing parts of the query in the context of Ignite itself. The integration of Ignite with Spark may justify the fact that our application did not have the same performance problems as those presented by Islam et al. (2022). Although it is outside the scope of the related work, the authors chose not to use the resources of a distributed environment, which could add complexity but improve performance. In addition, they used a
smaller mass of test data and a lower hardware configuration than that used in our experiments.

6 CONCLUSIONS

This paper presented the real-time data deduplication problem in the context of application with large volume of data. Additionally, it enumerated some challenges in streaming applications compared to batch applications. Finally, it compared some streaming deduplication methods using the Apache Spark framework and the scenarios in which each method is preferable. This comparison is summarized in a table to guide the choice of one of the evaluated methods.

Some limitations of this work are the failure scenarios analyzed and the storage of the consulted data set. Scenarios involving failure in communication between nodes via the network or the unavailability of a hardware component (e.g., HD without storage space, corrupted bit in memory) were not evaluated. Furthermore, analysis of the queries on the resulting data considered only the data saved in S3. This type of storage adds a significant time delay from being accessed over the network. If this data were stored locally (e.g., HDFS), some disk access optimization techniques could be applied, improving query execution time.

As a contribution, this work helps in decision making about the deduplication method to be used in a streaming application. In addition, it presents an analysis of the use of resources and processing time employed by different solutions streaming. As future work, there may be variations in the framework (e.g, Apache Storm, Apache Flink), in the auxiliary tools using other databases, in the size of the sample used, among others. One approach that could be verified is the use of Kafka Stream together with an intermediate infrastructure, not large volume data (state).

ACKNOWLEDGEMENT

Part of the PAPRICAS.org project supported by Independent Research Fund Denmark (DFF.dk).

REFERENCES

[10] Marcin Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John vuiLDwAAQBAJ