DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

---

# Experimental evaluation of distributed similarity joins in stream processing environments

---

*Author:*
Tomás HERNÁNDEZ-QUINTANILLA

*Supervisor:*
Dr. Asterios KATSIFODIMOS
*Daily supervisor:*
Georgios SIACHAMIS

*A thesis submitted in fulfillment of the requirements for the degree of Master of Science*

*in the*

Web Information Systems Group
Computer Engineering

September 26, 2023

# Declaration of Authorship

I, Tomás HERNÁNDEZ-QUINTANILLA, declare that this thesis titled, “Experimental evaluation of distributed similarity joins in stream processing environments” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Date: September 26, 2023

*"I also don't know how to live, I'm improvising."*

Kase O

DELFT UNIVERSITY OF TECHNOLOGY

# *Abstract*

Electrical Engineering, Mathematics and Computer Science
Computer Engineering

Master of Science

**Experimental evaluation of distributed similarity joins in stream processing environments**

by Tomás HERNÁNDEZ-QUINTANILLA

Similarity joins are operations which involve identifying similar pairs of records within one or multiple datasets. These operations are typically time-sensitive, as timely identification of relations can lead to increased profitability. Therefore, it is advantageous to analyze them using a stream processing system, which offers real-time capabilities. Due to the computational complexity of comparing numerous records, similarity joins can be resource-intensive.

To address this challenge, employing a distributed setting for executing the operations proves to be the most effective approach for resource management. In this research, we evaluate four distinct distributed systems designed for similarity joins in stream processing environments. The primary objective is to assess their individual strengths and weaknesses, as well as their overall efficiency. Our investigation reveals that certain solutions exhibit superior scalability and resource utilization, while highlighting the potential for further advancements in this domain.

# *Acknowledgements*

Firstly, I express my gratitude to my supervisor, Asterios, for providing me with the opportunity to work on this project and for introducing me to the subject through the course Web-scale data management.

I would also like to extend my thanks to my daily supervisor, Georgios Siachamis, for his valuable guidance throughout the project and for always being available to assist me when needed.

I am grateful to the members of the committee for dedicating their time to evaluating my work.

Lastly, I would like to acknowledge the unwavering support of my friends and family during the course of this project. In particular, I would like to thank my parents, Rufino and Luisa, who have consistently supported my decisions and provided assistance in every possible way.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The exponential growth of data over the last few years has led to a surge of interest in data mining and analysis. One of the key operations to be performed on this data is to find similar pairs of records across different datasets, which are commonly known as similarity joins. These similarity joins are especially useful for applications such as fraud detection [9] ,trend detection [40] and spam detection [31].

As the amount of data to be processed has increased dramatically, the resources needed to analyze it by using traditional batch processing systems are too high, which has led to the surge of real-time processing known as stream processing. The concept of stream processing was studied a long time ago [5], but it was not until recently that stream processing frameworks were developed and implemented [32]. Stream processing systems can handle unbounded datasets by processing the data as it arrives and continuously updating the results, without the need to store the entire dataset in memory. It enables the processing of data as it arrives, allowing for real-time analysis and decision-making.

Processing data in a distributed manner rather than in a single-machine system can provide scalability, as the system can add more nodes as needed to handle an increase in demand; fault tolerance, as if one node fails, the system can continue to operate using the remaining nodes; high performance, especially when the workload can be divided into smaller tasks that can be executed in parallel across multiple nodes; and higher cost-effectiveness, as it allows the use of commodity hardware and enables scaling out instead of scaling up.

There are not many applications that can perform distributed similarity joins in stream processing environments, and there is no established method for testing and assessing these systems in order to compare them with one another. The major goal of this thesis is to put into practice the systems that have been suggested to accomplish such jobs and to develop a set of benchmarks and tests that will allow for a comparison of these systems to determine which one excels at particular tasks.

## 1.1 Research Questions

The research questions to be answered by this thesis are the following:

**How do the different systems compare in terms of efficiency, accuracy and scalability?**

The comparison among the systems will be conducted based on several key metrics, including the number of times a record needs to be replicated, the number of operations needed to process a record, and the average time taken to process a single record. Additionally, the assessment will encompass the system's capacity to handle records per second and the corresponding resource utilization during record processing.

**Which tasks are the most suitable for each system and how flexible are they?**

The systems under investigation employ various techniques to distribute the execution load and partition the data space, along with filtering methods to enhance data processing efficiency. These techniques demonstrate varying degrees of effectiveness based on the data's distinct characteristics, such as length or homogeneity. The present research endeavors to identify the most relevant data characteristics for optimal system performance.

**How can the performance of the systems be optimized for different use cases?**

To enhance the effectiveness and performance of the systems under investigation, a thorough exploration of potential system enhancements is warranted. By conducting a meticulous examination of the frameworks, it will be possible to ascertain whether their current design is flawless or if there are aspects that could be modified to yield improvements.

## 1.2 Contributions

In the same sequence that they were articulated in section 1.1, a list of the contributions that were made during the process of answering our research questions is provided in this section.

1. The different systems were effectively compared. Different experiments were performed, which are described in section 5 and the results maybe observed at section 6.

2. The experiments utilized in this study involved varying datasets consisting of diverse data types and objectives. The most suitable system for each scenario could be determined effectively by analyzing the information obtained during these experiments.

3. During the evaluation of the systems some inherent flaws of the different systems were found. Solutions for these flaws were proposed in the project.

## 1.3 Outline

The structure of the thesis is as follows: Chapter 2 presents the background and related work information on distributed systems to perform similarity joins in stream processing environments is presented. Chapter 3 provides an analysis of the relevant literature in the field, focusing on previous works and studies related to the topic. Chapter 4 presents the different solutions that were implemented and tested. Chapter 5 presents the implementation of the chosen solutions and the set up for the experiments to test them. Chapter 6 presents the results obtained in the experiments performed. Finally, chapter 7 provides a comprehensive summary of the work conducted and the results obtained and a discussion of potential future work that could be explored based on the outcomes of this thesis.

# Chapter 2

# Background

This section aims to elucidate the concepts of similarity joins, stream processing, and distributed environments, as well as providing a brief description of the technologies that were used to develop the project.

## 2.1 Similarity joins

Data mining and database management systems use similarity joins as a core procedure to find objects that are similar to one another across diverse datasets using a predetermined similarity metric. Similarity joins are advantageous because they may support a wide range of applications, including record linking, data integration, entity resolution, and information retrieval.

They were first proposed by the authors of [20] who introduced the idea of using hash-based techniques for approximate similarity joins in high-dimensional data sets. Since then several algorithms have been proposed to perform similarity joins such as [7, 8, 18, 22].

The similarity join problem involves comparing each record in one dataset with every record in the same or another dataset to identify pairs of records that are similar according to a given similarity function. The similarity function typically measures the distance or similarity between two objects based on some domain-specific criteria, such as the jaccard similarity between two strings or the cosine similarity between two vectors.

The usefulness of similarity joins lies in their ability to enable effective data integration and knowledge discovery by identifying similar records across different datasets. For example, similarity joins can be used to detect fraudulent activities [9], identify trends in social media platforms such as Twitter [40] or spam detection [31]. In the academic domain, similarity joins can be applied in various fields such as bioinformatics, text mining, and multimedia analysis to identify similar genes, documents, and images, respectively [35, 25, 41].

Similarity joins pose various challenges and problems that require careful attention to ensure their accurate and efficient execution while minimizing resource consumption. The main issues to be addressed will be exposed in the following subsections.

### 2.1.1 Scalability

Performing similarity joins on large datasets can be computationally expensive and time-consuming, especially when dealing with high-dimensional data. As the size of the dataset increases, the number of comparisons required grows exponentially, leading to increased processing time and resource requirements.

Pruning techniques such as prefix [13] or length [4] filtering may be applied in order to reduce the number of comparisons that are performed on a single record to find which ones are similar to it. The use of these and similar techniques will be present in all the frameworks implemented for this study.

### 2.1.2 Similarity metric

Depending on the similarity metric being used, the number of records found similar to a single record could increase or decrease. The chosen similarity metric could have a significant impact on the quality of the results. Different similarity metrics may be suitable for different types of data or applications, and selecting the wrong metric can lead to inaccurate results. Some of these metrics include the Jacquard similarity, cosine similarity, or angular distance between a pair of records. The chosen similarity metrics will be the same across all the frameworks studied in this project, with Jacquard similarity being the main used metric.

### 2.1.3 Threshold selection

Similarity joins require a threshold value to determine the level of similarity required between two records to be considered a match. Selecting an appropriate threshold can be challenging as it depends on the specific application and the characteristics of the data. A threshold that is too low may result in false positives, while a threshold that is too high may result in missed matches. As it can be observed in [36], the amount of matches found within a dataset varies dramatically when using different threshold values, so the appropriate threshold has to be chosen depending on the data that is being used.

### 2.1.4 Data preprocessing

Preprocessing the data can be necessary to improve the efficiency and accuracy of similarity joins. This may involve techniques such as data normalization, dimensionality reduction, or feature selection. However, preprocessing can also introduce its own challenges, such as selecting the right techniques and ensuring that the preprocessing steps do not alter the similarity between records. All the data that the different studied systems use is altered (e.g., sentences are divided into the words that compose them and then alphabetically sorted afterwards) in order to make the systems work more efficiently and smoothly.

### 2.1.5 Data skeweness

When employing a distributed system, data skewness can be a concern for similarity joins since it can result in uneven workloads throughout the cluster's nodes, causing some nodes to complete their duties considerably sooner than others. Systems must balance the workload across the various nodes that make up the system to overcome this issue. Several studies have already suggested methods for carrying out this work, like [33] and [28].

## 2.2 Stream processing

The amount of data generated by various sources such as social media, sensors, and IoT devices has increased exponentially, which has led to the creation of stream processing, which allows to quickly and efficiently process vast volumes of data right after they have been generated in real-time, which is crucial in some applications such as fraud and anomaly detection [29]. Unlike batch processing, stream processing can analyze data as it arrives, without the need to collect all the data beforehand. Stream processing is also more scalable than batch processing, as it can handle large volumes of data without running out of memory or computing resources while limiting the number of records to be processed at a time. On the other side, batch processing can be time-consuming and expensive when processing large datasets since it may need significant computational resources.

More over, stream processing models enable the analytical model to be continuously refined while taking into account the content of the incoming input, leading to more accurate results. In contrast, as batch processing is based on a static analysis paradigm and does not take into consideration how the data changes over time, it could produce less accurate results. Lastly, stream processing allows for real-time feedback and action capabilities, which are desirable in applications such as online advertising or recommendation systems [12].

Creating an efficient and fully functional stream processing system comes with several problems that need to be solved; the main ones will be explained in the following subsections.

### 2.2.1 Real-time processing

Stream processing systems need to process data in real-time as soon as it is generated or received. This means that the system needs to be designed to handle a continuous and high volume of data at a fast pace and, in many cases, with very low latency. In order to overcome this problem, systems need to have enough resources to run as well as allocate them efficiently.

### 2.2.2 Scalability

Similar to similarity joins, stream processing systems need to be scalable as the amount of data being processed could increase over time. Typically, they require a horizontally scalable architecture, which can be achieved through a distributed system capable of accommodating the addition of new nodes to its architecture.

### 2.2.3 Fault tolerance

Since stream processing systems operate continuously and process high volumes of data, they must be capable of handling failures and restarting their activities gracefully. To prevent data loss or significant downtime, the system should be designed to recover from errors such as node failures or network disruptions. This can be achieved by implementing the appropriate mechanisms and protocols, such as checkpoint creation and data replication.

### 2.2.4 Data consistency

Stream processing systems need to ensure that the data processed is consistent and accurate. This can be challenging because the data arrives in a continuous and unbounded stream, making it difficult to maintain consistency. Mechanisms to ensure that all the data generated has been processed will be needed, as will exactly-once processing guarantees, which are already provided by frameworks such as [11, 32].

## 2.3 Distributed systems

Performing tasks such as similarity joins in real-time using stream processing technologies requires the use of a large amount of resources, and single-machine systems are a very expensive option when implementing this sort of system. This is one of the reasons for using a distributed system. Distributed systems consist of multiple independent machines that work together to solve a complex problem. This type of system offers several advantages over single-machine systems.

Scalability is one of the prime reasons for employing distributed systems. As data volumes and computational requirements grow, it becomes increasingly difficult for a single machine to handle the workload. Distributed systems allow the workload to be distributed across multiple machines, enabling the system to scale in order to meet increasing demand. Fault tolerance is another benefit of distributed systems. Single-machine systems are susceptible to failures that could take the entire system down, such as hardware issues, software bugs, or power outages. On the other hand, distributed systems are created to use replication and redundancy to be resilient to failures.

Lastly, distributed systems can offer improved performance over single-machine systems by exploiting parallelism. In a distributed system, tasks can be divided into smaller sub-tasks that can be processed in parallel by multiple machines, allowing the system to complete tasks faster than a single machine could. For computationally demanding applications like machine learning, data mining, and scientific simulations, this is especially helpful.

On the other hand, implementing a system in a distributed manner implies a higher level of programming complexity and can potentially add additional communication and replication costs to a system. In the following subsections, the problems associated with using distributed systems will be addressed.

### 2.3.1 Consistency

Distributed systems need to maintain data consistency across multiple machines, which could have different characteristics and capabilities. This is a challenging task due to issues such as network latency and communication delays, as well as possible network errors. There are different levels and types of consistency, as well as mechanisms to ensure the consistency of the results while minimizing the resources used to maintain it.

### 2.3.2 Communication overhead

In a distributed system, the different machines need to communicate with each other over a network. This communication adds extra overhead and latency to the machines' computations. Therefore, system implementations strive to minimize the amount of communication between nodes in order to reduce these additional costs.

### 2.3.3 Complexity

Distributed systems are generally more complex than single-machine systems, as they involve multiple machines that must work together to produce accurate results. To minimize the amount of data replication and communication between nodes, coordination among the different nodes is necessary to perform tasks effectively.

## 2.4 Technologies used

In this sections the different technologies used to implement the systems and benchmark them will be described.

### 2.4.1 Apache Flink

All the systems have been adapted to work in Apache Flink [11], which is an open-source distributed data processing system that provides a powerful and flexible platform for stream processing. Flink is designed to handle high-throughput, low-latency data processing at scale. It provides a distributed architecture that enables parallel processing of streaming data, which allows for the efficient and timely processing of large volumes of data.

One of the key advantages of Apache Flink is its fault-tolerance mechanism. Flink uses a technique called "stream checkpointing" to ensure that data is processed reliably and without data loss in the presence of failures. Stream checkpointing periodically takes snapshots of the streaming data and stores them in a reliable storage system. In case of failure, Flink can recover the processing state from the latest checkpoint, ensuring that no data is lost. This feature will be used to implement the dynamic repartition protocols that the different chosen systems require in order to achieve their maximum performance.

Apache Flink provides a powerful and flexible platform for stream processing. Its fault-tolerance mechanism, unified programming model, and rich set of APIs and libraries make it a good choice for building complex and scalable data processing pipelines. This APIs include a REST API which can be queried within a Kubernetes cluster in order to make the different services running in the cluster interact with each other.

### 2.4.2 Kubernetes

Kubernetes [10] is an open-source container orchestration system that allows for the efficient deployment, scaling, and management of containerized applications. A high degree of automation, scalability, and reliability can be accomplished in software deployment processes by deploying a collection of services in Kubernetes pods.

One of the main advantages of Kubernetes is its ability to automate the deployment and scaling of containerized applications. Kubernetes enables the automatic deployment of containerized services to a cluster of nodes and provides automated scaling of these services based on resource utilization metrics such as CPU and memory usage. This can help organizations reduce the time and effort required for manual deployment and scaling of services.

High availability and resilience are two advantages of using Kubernetes to deploy a collection of services. Kubernetes has the ability to recognize node failures, recover from them, and automatically reassign failed services to healthy nodes. This can ensure that services continue to be responsive and accessible even in the event of hardware or software malfunctions.

By giving each of the various pods present in a cluster a distinct IP address and opening up ports inside each pod, Kubernetes also permits communication between the various services inside the cluster.

Moreover, Kubernetes offers a portable and adaptable platform for software distribution. Without making significant changes to the deployment configurations, services launched in Kubernetes can be quickly transferred across various environments, such as development, testing, and production. In order to design the product for subsequent deployment on stronger machines, this can be helpful.

## 2.5 Apache Kafka

Apache Kafka [23] is a messaging system that is distributed, fault-tolerant, and scalable, allowing for the efficient and reliable flow of data across computers. High performance, scalability, reliability, and flexibility in data processing can be attained by utilizing Kafka to ingest and extract data from a system.

One of Kafka's primary features is its ability to manage high-throughput, low-latency data streams. Kafka's distributed architecture allows it to manage large amounts of data while maintaining low latency, making it an ideal platform for real-time data processing applications. Kafka also has a scalable and fault-tolerant architecture that can help process data reliably and without data loss even when the system fails.

Furthermore, Kafka includes a rich collection of APIs and integrations that can aid in the creation and maintenance of complicated data processing pipelines. Kafka provides stream processing APIs that can be used to transform and aggregate streaming data in real time. Kafka also integrates with popular data processing frameworks like Apache Flink, allowing the creation of sophisticated data processing workflows.

## 2.6 Benchmarking

As there have not been a lot of systems that perform distributed similarity joins in stream processing environments, there are not any established benchmarks or experiments that can be used to compare a system with other developed systems. The authors of [38] describe a series of experiments that they have designed to compare the system that they developed with the other systems that employ different algorithms for both distributing the dataset's records to different partitions and to perform the similarity joins between the different records, however they do not provide the code for such experiments or the implementations of the other systems, so they can not be used to evaluate other systems.

In order to create fair benchmarks to compare the different systems, the experiments will use different types of data and datasets, although they will be mainly focused on analyzing text and sets of strings. The experiments will measure different metrics, taking into account both records that obtained a join result and those that did not. These metrics include:

- Throughput: Amount of records that the system is able to process per second.
- Number of joins: Number of different similarity joins performed during the execution of the program.

- Latency since ingestion: Time required to process a record since it was ingested by the system.
- Latency since creation: Time required to process a record since its creation by the producer.
- Operations: Number of operations that a record has needed in order to be process and find potential joins.
- Replication: Amount of times a record has been replicated by the system in use.

A comprehensive set of metrics will be measured to assess the system's performance. However, it should be noted that not all metrics are entirely representative of the system's overall performance. As a result, only select metrics will be utilized for the analysis of the system's performance.

Careful consideration will be given to choose the most relevant and meaningful metrics that provide valuable insights into the system's effectiveness and efficiency.

Which of these metrics will be used is described in section 5.4.

# Chapter 3

# Related work

This chapter presents a comprehensive overview of the current state-of-the-art concerning distributed similarity joins in stream processing environments. To collect relevant papers, diverse tools and keywords were utilized, while ensuring the exclusion of those that proved unsuitable for the project—specifically, papers that could not be adapted to function in distributed or streaming environments. To establish a benchmark that enables the evaluation of different systems under equitable conditions, our approach involved selecting systems that implemented distinct techniques for executing the joins.

The primary systems employed in this project include the Distributed Streaming Set Similarity Join (DSSSJ) [38] and the Cluster Join [15]. DSSSJ adopts a length-based partitioning approach and implements the "Bundle Join" algorithm to execute the joins efficiently. On the other hand, the Cluster Join utilizes a set of anchor records to distribute the data and perform the joins in a distributed manner. Both of these systems were proposed by the thesis supervisor and constitute integral components of the research undertaking.

In order to find other solutions and work related to the project, the search of papers mainly involved searching for papers that either cited the previously mentioned systems or were cited by them. The paper [19] presents an evaluation of different map-reduce systems that have been developed in order to perform similarity joins in a distributed manner. The systems described in the paper were not developed to work in a stream processing environment, but they can be adapted to such environments.

Taking into account the results exposed by the paper, it was decided to implement the Vernica Join [33], which distributes the records depending on the values of its prefix tokens. This system was qualified by [19] as the most efficient and also presents a partitioning system that is different from the others.

Other systems that were considered for the project would include [17], which was discarded as its implementation is very similar to Cluster join and according to [19] its performance is not outstanding, or [37], which was discarded as it required a lot of adaptation for working in a distributed environment as it would require sharing logs of information between the different nodes.

In total, four distinct systems were implemented for evaluation in this research. Additionally, a flexible framework was developed to test these systems, facilitating seamless adaptation of any new system for evaluation. As a result, the framework's extensibility allows for the potential assessment of additional systems in future research endeavors.

# Chapter 4

# Chosen solutions

As of today, the development of solutions for performing distributed similarity joins in stream processing environments remains limited. Notably, only one system, identified as [38], has been published with the explicit purpose of fully addressing this requirement. However, several systems have been proposed for similarity joins, although they do not necessarily adopt a distributed approach or utilize stream processing, such as [17] and [30]. Two frameworks, Cluster join [15] and Vernica join [33], were selected for adaptation to enable similarity joins within stream processing environments.

In addition to these frameworks, a naive solution was implemented to facilitate performance comparison with the other systems. This section provides a comprehensive overview of the adaptations made to the chosen frameworks and the corresponding adjustments applied.

## 4.1 Distributed Streaming Set Similarity Join

[38] proposes a distributed system for processing records in stream processing environments based on distributing the records to different machines depending on their length rather than distributing them randomly, depending on their prefix. [13, 6], or distributing them by applying partition-based signatures [17, 16]. The aim of using this distribution scheme is to have no data replication and low communication costs while achieving a good load balance that ensures a high system throughput.

Their proposal is to partition the length space of the records into several disjoint length intervals, which enables each node of the system to only take care of a length interval. However, instead of doing a uniform partition that could result in a critical load imbalance and thus low performance, they propose the use of a cost model to dynamically estimate the local workload of each node and efficiently partition the length space.The initial partitions are created based on historic data from previous executions of the system.

The highlighted distribution scheme offers a significant advantage over the prefix-based and partition-based models. In this scheme, each record requires indexing only once, eliminating the need for additional storage across all nodes. Figure 4.1 provides an illustration of all the distribution schemes. Notably, the distribution scheme proposed by the authors, which follows a length-based schema, stands out as it necessitates minimal data replication while ensuring low communication costs.

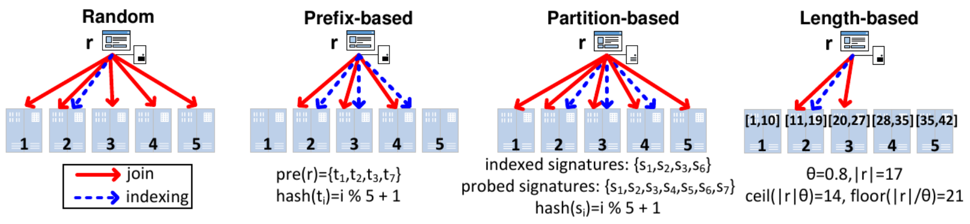

FIGURE 4.1: Example of different distribution schemes [38].

Once a record has been dispatched to a node, the node has to perform two key operations with the data from the incoming record. It has to perform a join operation to find all the similar pairs in the record and index it for future joins. In order to perform these two operations, the authors develop a join bundle-based algorithm that they name BundleJoin.

The BundleJoin algorithm creates bundles with the join results returned from join operations. Each of these bundles contains highly similar records in terms of their prefix, length, bounds, and positional bounds. The belonging of a record to a bundle is verified in a batch processing operation by using the difference between the tokens of the record with the ones of the different records within the bundle.

Figure 4.2 illustrates an overview of the bundle algorithm. As can be observed, in the first step of the algorithm, the objective is to identify records that resemble the incoming record. If the record is the master of a bundle, then all associated records are retrieved for comparison. If the record is not the master of a bundle, only the found record is compared to the incoming record. If the comparison yields similarity and the record being compared is within a bundle, then the incoming record is added to the bundle. If there is no similarity found, a new bundle can be created with the incoming record as the master.

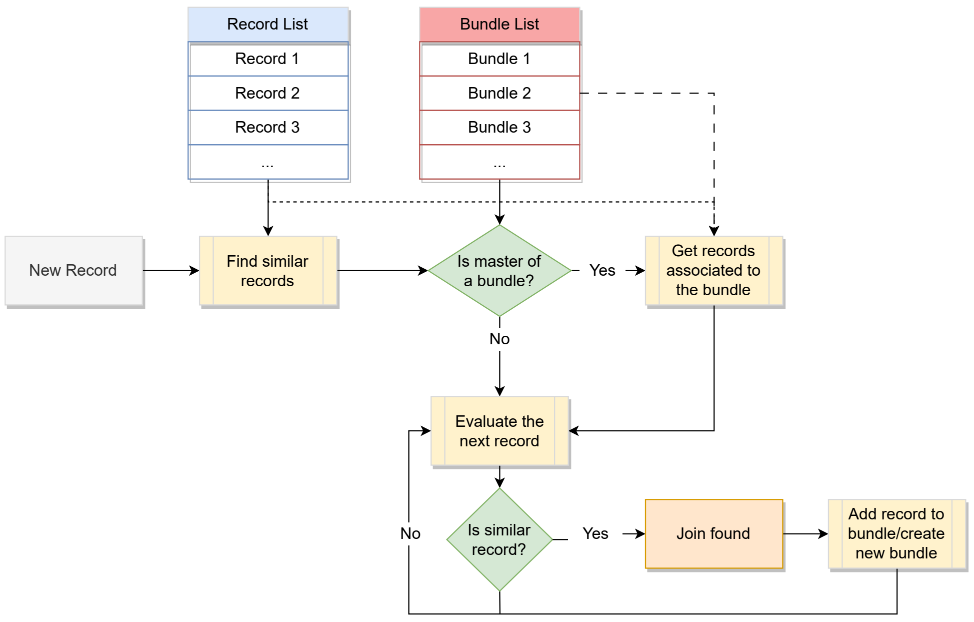

FIGURE 4.2: Overview of the BundleJoin algorithm.

The system utilizes an inverted list to link each processed record's token to the record itself. If a record does not belong to a bundle, it is labeled as a single record.

Additionally, there is a list of created bundles. The BundleJoin algorithm consists of two stages, the first being the join stage.

Figure 4.3 demonstrates the join portion of the algorithm. Initially, an empty set of candidates is created, and then the algorithm iterates through the tokens that make up the record. It retrieves records with the same token by querying the inverted list, which indicates which records contain the token. The set of candidates is then filtered using two criteria: similar length and passing the bundle positional filter. The latter verifies that the candidate record has at least a similar prefix. The similar records that meet these criteria are classified as similar records.

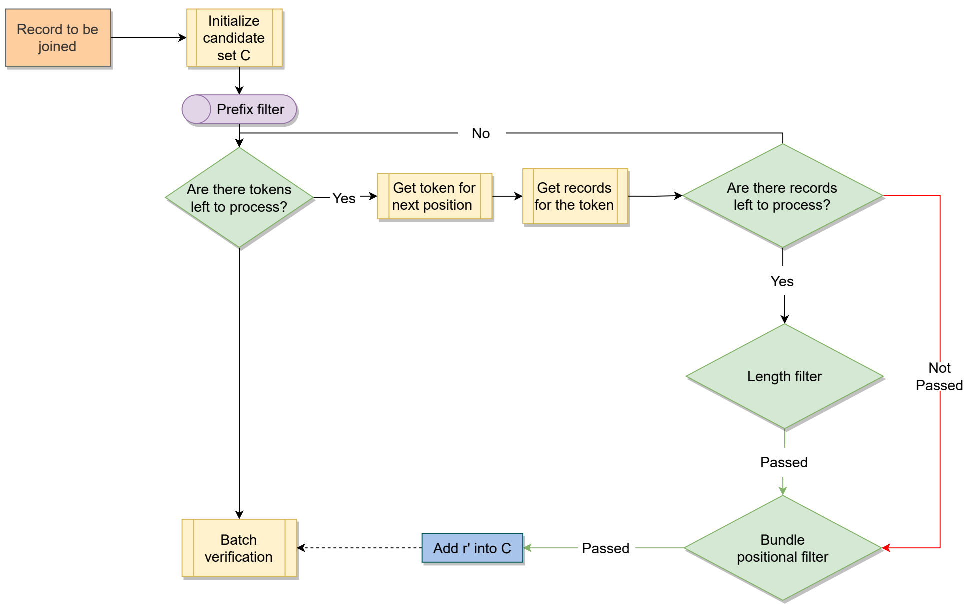
Record to be joined
Initialize candidate set C
Prefix filter
No
Are there tokens left to process?
Yes
Get token for next position
Get records for the token
Are there records left to process?
Yes
Length filter
Not Passed
Passed
Batch verification
Add r' into C
Passed
Bundle positional filter

FIGURE 4.3: Bundle join: Join algorithm.

After creating a set of candidate records, a batch verification process is used to generate a final set of truly similar records. The batch verification algorithm (as shown in Figure 4.4) compares each record not only with the set of similar records from the previous steps but also with all the records belonging to the same bundle as the master record of that bundle. Once the candidate set has been verified, the algorithm moves on to the next stage.

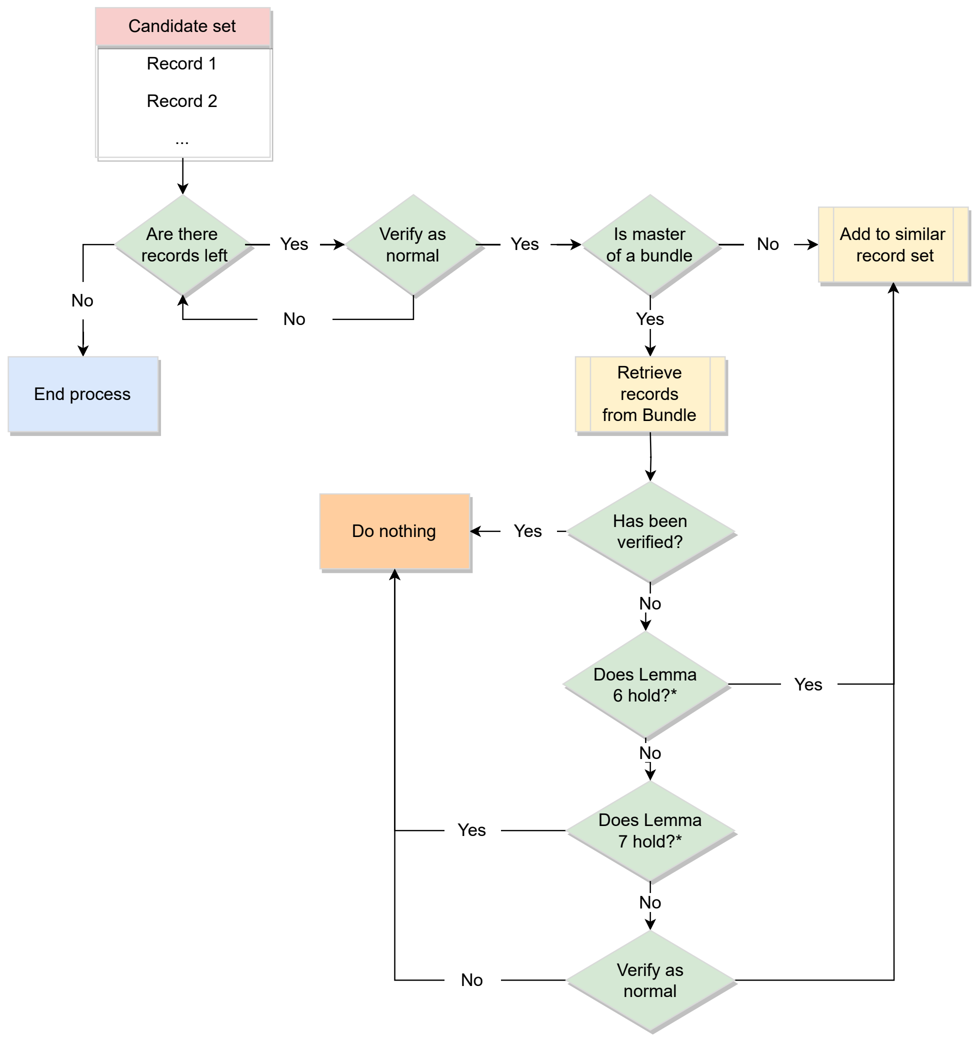

FIGURE 4.4: Bundle join: Batch verification algorithm.

The second stage of the bundle algorithm is indexing. Figure 4.5 illustrates the indexing part of the BundleJoin algorithm. Once a record is processed through the system, the algorithm tries to find the most adequate bundle for it. In the case that an adequate bundle has not been found, the record is marked as single and its tokens are inserted in the inverted list for future reference. If a suitable bundle has been found, the record's tokens that are not shared with the master entry of the bundle are indexed in the inverted list and the information about the bundle is updated.

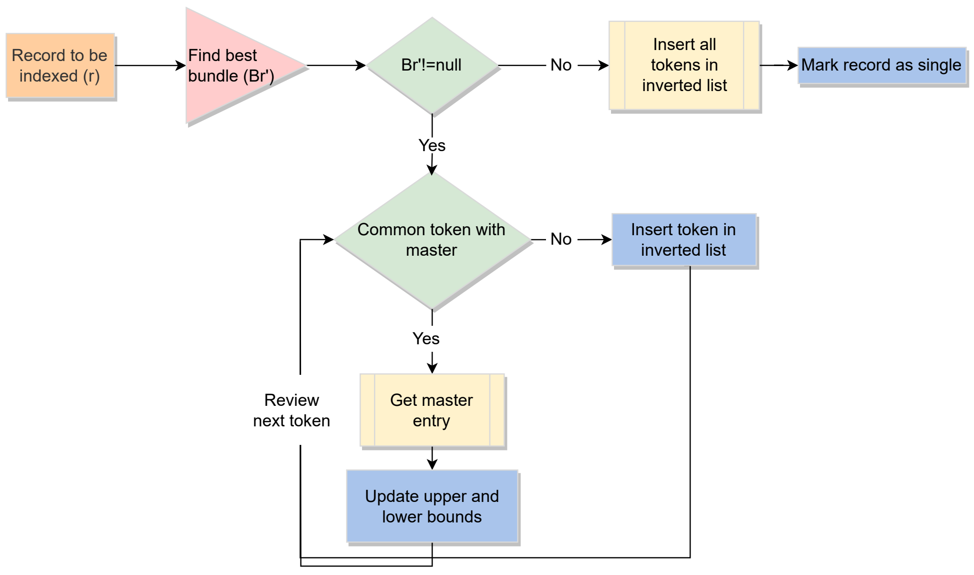

FIGURE 4.5: Bundle join: Indexing algorithm.

The list of bundles in the system has a master record associated with each bundle. The master record is similar to every other record in the bundle. If a record is similar to the master record of a bundle, it is added to that bundle. This reduces the number of comparisons required. All created bundles are registered in the bundle list.

## 4.2 Cluster Join

Cluster join [15], is a framework designed to perform similarity joins by leveraging the MapReduce method for comparing the records. They propose partitioning the space by designating a set of anchor records, sampled from the dataset that is going to be analyzed, that represent the centers around which the records can be clustered to form partitions.

Figure 4.6 illustrates an example in which a vectorial space has been divided by using the euclidean distance between its anchors. As can be observed, there are three distinct home partition spaces, one for each of the anchors, as well as an outer partition space for each of them. A record can be assigned a unique home partition, but may belong to multiple outer partitions. Similarity joins are executed between records located within the same partition. The use of partitions facilitates parallel processing of the joins, as they can be executed by different machines.

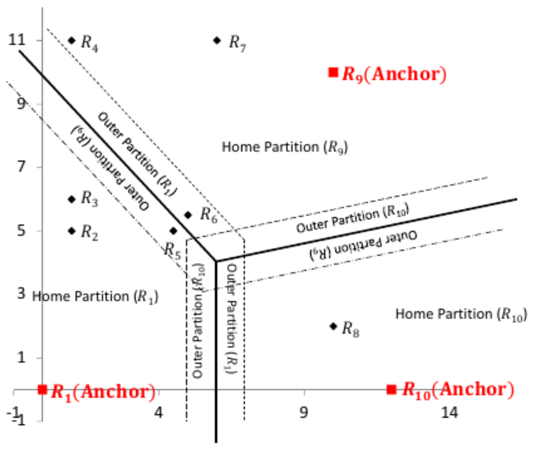

FIGURE 4.6: Example of space partitioning using euclidean distance [15].

A record that corresponds to the outer partition of an anchor will only be compared with the records in the home partition of said anchor; this is to prevent performing the same comparison twice. In the case of the home partition anchor, the record will be compared with all the records in both the outer and home partitions of the anchor. In a further effort to prevent the same comparison being made, the system implements a mechanism in which if a record corresponds to both a home partition and an outer partition at the same time, it will only be sent to the outer partition if it has a higher ID than the home partition, so no unnecessary comparisons are made.

If a partition receives more records than expected, the partition can be expanded by employing a 2-D hashing mechanism, so when the amount of records associated with one anchor exceeds a certain threshold, the anchor will be expanded and the load divided between the subpartitions made in the process. In their implementation, they perform this action by sampling the dataset they want to analyze again. Figure 4.7 exposes how this algorithm works. Once the anchor has too many query points assigned, it will expand to a hashmap whose side follows the formula $S = 2Q/T$, where Q is the amount of query points associated with the anchor, T is the threshold, and S is the side of the matrix.

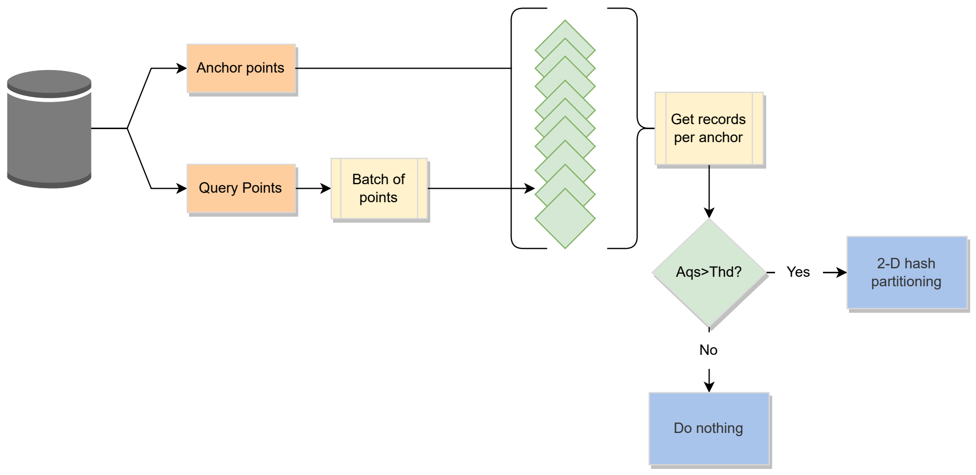

FIGURE 4.7: 2D-hashing expansion algorithm.

In order to ensure that a record will be compared with all candidate records for a similarity join, the authors propose performing the record dispatching in two different ways. When the records are from two different streams, each record will be assigned to either the columns or the rows of the hash map. The column or row to which it will be dispatched will depend on a hashing function, which will ensure that a record will be compared with every eligible candidate. In the case of a self-join, half of the matrix will be unnecessary, so the pairs with the same hash value will only be compared in diagonal cells. Figure 4.8 illustrates how the records are distributed when using the 2-D hashing mechanisms in both the multiple stream and single stream cases.

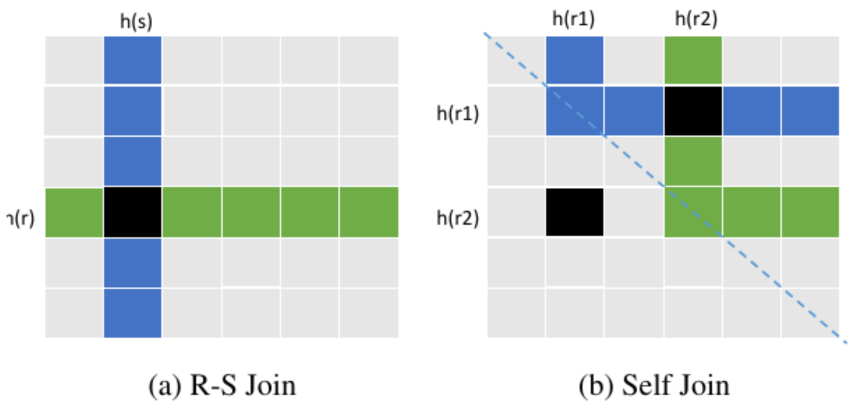

FIGURE 4.8: 2D hashing illustration [15].

In order to adapt this system to streaming environments, an historic file will be sampled to acquire the anchor points, and the anchors will be expanded dynamically, depending on the amount of records associated with the anchor during the execution. As it already partitions the record space, it does not need any adaptations to be correctly executed in a distributed environment.

## 4.3 Vernica Join

The authors of [33] propose a system that leverages the map-reduce programming model in order to perform the similarity joins in a distributed manner by distributing the records to different partitions depending on the tokens in their prefix in order to balance the workload.

Their proposal involves sorting the tokens within a record based on their frequency of occurrence in the dataset, prioritizing the placement of the least frequent tokens ahead of the others. Once the tokens have been ordered the first $n$ tokens (prefix tokens) are used to route the records to the processing nodes in which the similarity joins will be performed.

The paper proposes two ways to perform such routing which are using individual tokens, which involves replicating each record as many times as the number of tokens in its prefix and using grouped tokens, which involves mapping multiple tokens to the same key in order to reduce the amount of keys that are used. In this case, two records that share the same token group might not share any tokens. The groupId will be assigned to each record following a Round Robin approach.

In a stream processing environment, access to the complete dataset beforehand is not available. However, the whole set of words to be used during the experiments can be accessed. Historical data will be utilized to calculate the frequency of each token appearing in order to sort tokens from incoming records.

The partitioning algorithm proposed by [33] is illustrated in Figure 4.9. The first step involves calculating the frequency of each token to sort the record's tokens. Next, the prefix length is calculated, and a partitionId is assigned based on the tokens in the prefix. The partitionId can either be the token itself or a groupId based on the token's value.

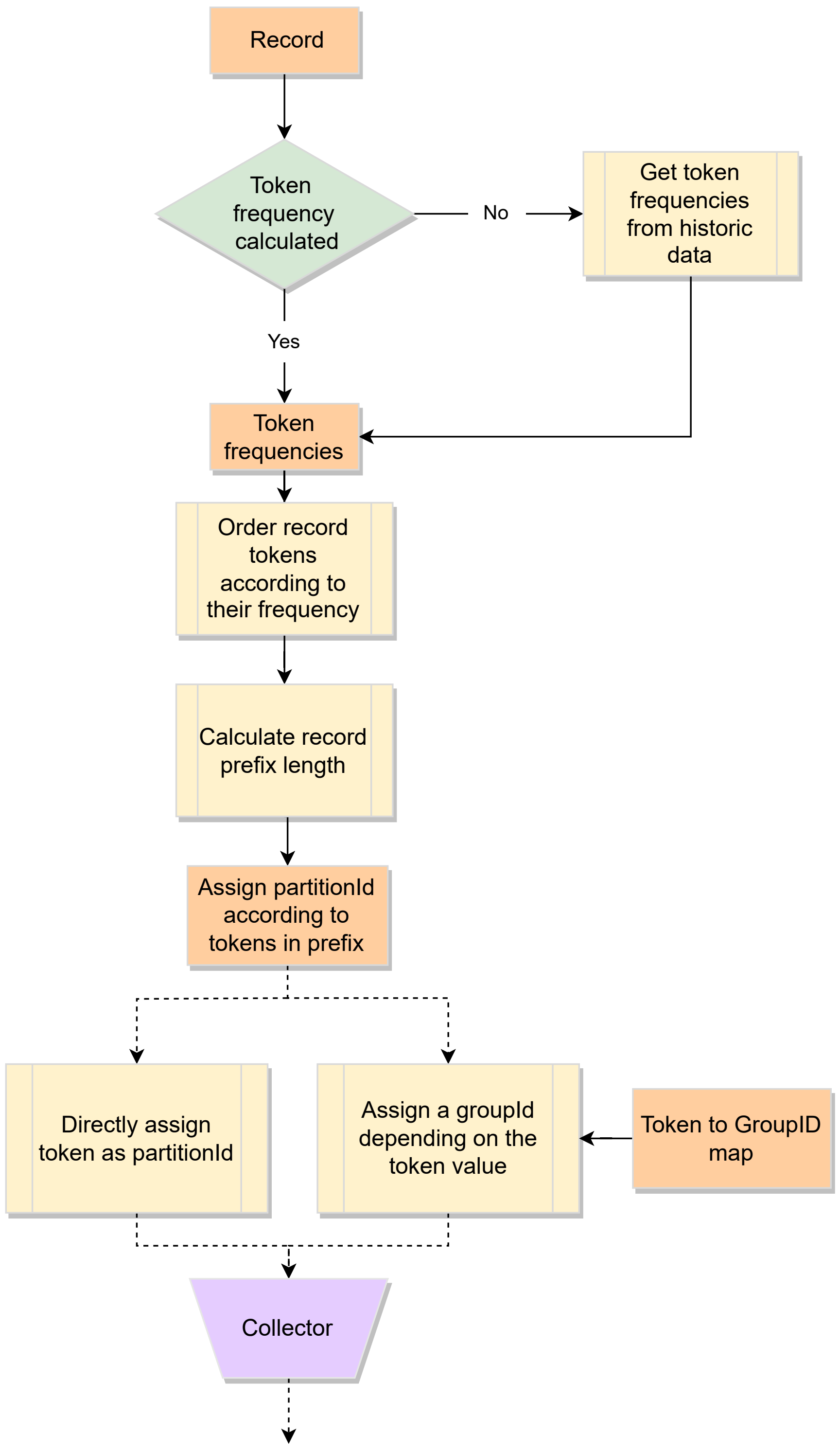
Record
Token frequency calculated
No
Get token frequencies from historic data
Yes
Token frequencies
Order record tokens according to their frequency
Calculate record prefix length
Assign partitionId according to tokens in prefix
Directly assign token as partitionId
Assign a groupId depending on the token value
Token to GroupID map
Collector

FIGURE 4.9: Vernica partitioning algorithm.

Figure 4.10 displays how the groupId is assigned to each token according to the paper. As it can be observed, if the token has already been assigned an id, the system will just retrieve the id from the token and add it to the token to groupId map; if not, the token will be assigned a groupId following a round-robin system, in which the value will depend on the amount of nodes or partitions that are present in the system and the id assigned to the previous token.

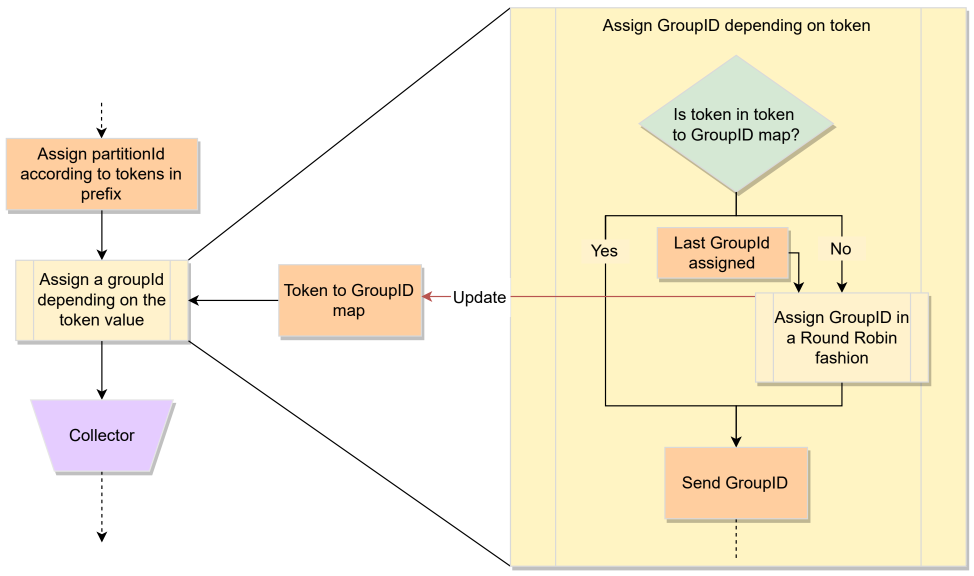
Assign partitionId according to tokens in prefix
Assign a groupId depending on the token value
Collector
Token to GroupID map
Update
Assign GroupID depending on token
Is token in token to GroupID map?
Yes
No
Last GroupId assigned
Assign GroupID in a Round Robin fashion
Send GroupID

FIGURE 4.10: Vernica groupId assignment algorithm.

After the record has been sent to its corresponding partitions, the paper indicates that the joins shall be done by using either a basic kernel (BK) algorithm to perform the join, which entails comparing all records within a partition with each other, or utilizing the PPJoin+ algorithm [34]. The latter approach applies a prefix and length filter to the records, which helps to decrease the number of comparisons that need to be executed.

## 4.4 Naive implementation

In addition to the previously described systems, a decision was made to develop a naive system for distributing records to various nodes and conducting join operations in a stream processing environment.

Under this approach, the records are dispatched to all nodes but indexed only in a single node. The assignment of the indexing node follows a round-robin system, ensuring a balanced workload and equitable distribution of join operations among the nodes. The distribution system is further illustrated in Figure 4.11.

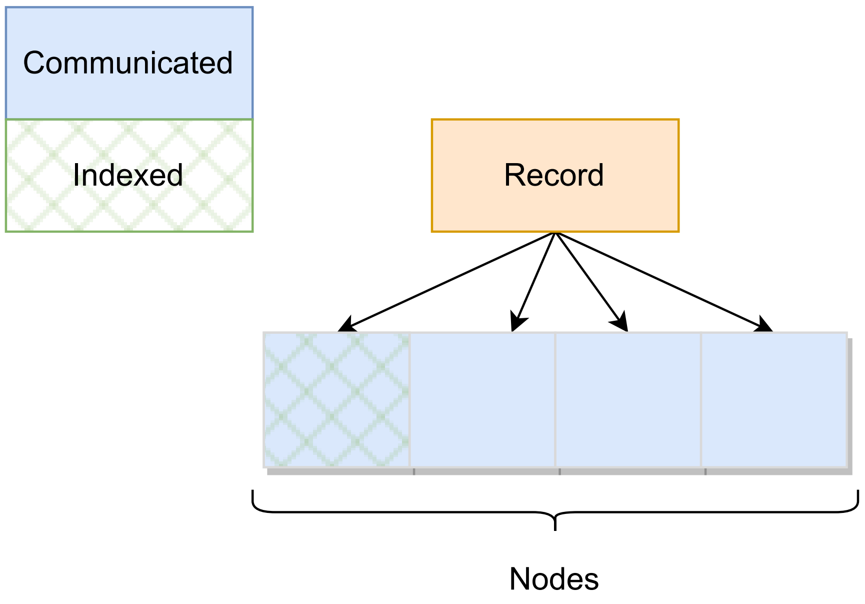

FIGURE 4.11: Naive system partitioning algorithm.

Once the record has been received by a node, the comparisons will be executed using the BK algorithm, as described in the Vernica join 4.3. Additionally, the system will have the capability to employ the PPJoin filtering technique, which was specifically developed for this project.

# Chapter 5

# Implementation

In this section it will be explained how the different solutions described at section 4 have been implemented, as well as the experiments to evaluate their performance.

## 5.1 Overview

All the systems have been deployed to Kubernetes in the same way. There are six different pods running different services, as illustrated in Figure 5.1.

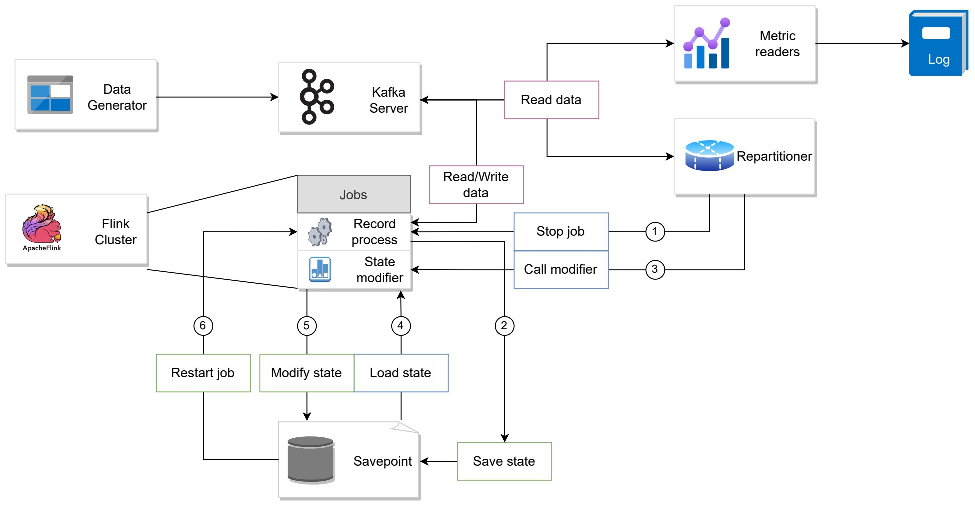

FIGURE 5.1: Overview of the system implementation.

Starting from the left, the first pod is in charge of writing the data to be processed to the second pod, which is the Kafka server[1], from which the main program will read the records as well as write the results of the execution so that the metrics calculators (two separate pods represented as one) and the repartitioner can read the information of the execution.

The main program is running in the Flink cluster, which is the central pod of the system. This cluster will be continuously run the system that is ingesting the data and performing the similarity joins.

The system will be in charge of writing the data about how the records are being distributed to Kafka, so the repartitioner can trigger the repartition for the system, whose sequential stages are delineated by numerical annotations within the image, wherein lower values correspond to initial steps, and higher values correspond to subsequent ones. It is also in charge of writing to Kafka information about how

[1]The two Kafka nodes showed in the diagram correspond to the same pod in reality.

the records are being processed (number of comparisons required, time required to process the record, etc.) and about the joins that have been performed.

This information is ingested by the consumers that write it into logs so that it can be later retrieved and the execution of the system can be evaluated.

### 5.1.1 Data producer

The initial component of the system incorporates a Kafka producer responsible for data generation from the datasets mentioned in subsection 5.3.2. For synthetic data, a list of 5,000 words is utilized for runtime generation. In contrast, text files were created to store at least one gigabyte of information for the other datasets, with the exception of the Tweets dataset, which comprises a dataset of 100 MB.

The synthetic data generation process consistently employs the same random seed, thereby enabling the evaluation of results across different systems under equivalent conditions. This ensures a fair and controlled comparison between the systems, as any variations in the results can be attributed to the differences in their underlying algorithms and processing techniques, rather than to variations in the generated data.

Following data generation, it is written to one or two Kafka topics, depending on the requirement to assess joins within the same stream or across different streams. Subsequently, the record processing job employs this data to identify similarity joins. In the experiments performed in this project two different streams of data are used for analyzing the performance of the systems.

To ensure uniform data creation by the Kafka writer, a special implementation was devised. Records slated for writing in a second are divided into several intervals, and a sleep operation is executed at each interval to prevent record bursts. The sleep duration corresponds to the following formula:

$$\frac{\text{Second-Record Rate} \times \text{TimePerRecord}}{N^{o} \text{ Intervals}}$$

In this formula, the variable *TimePerRecord* represents the time taken by the program to write a single record to Kafka. To determine this value, a million records were generated for each dataset, and the average time required to generate these records was calculated.

The utilization of the repartitioning algorithm presents a challenge in the system's performance as the data generator continues to operate uninterrupted during the repartitioning process.

Consequently, the system must handle all the data generated during this repartitioning period at once, which significantly influences the obtained results. However, implementing a mechanism to pause the data generator and accurately track its progress in reading the dataset proved to be excessively complex and time-consuming within the scope of the project.

Despite the complexities involved, the decision not to stop the data generator and simulate real-time conditions has merit. In practical scenarios, data generation typically continues without interruption even if a repartitioning event occurs.

In this study, the producer assumes the responsibility of marking the records that are to be sent to the record consumer. This ensures that one-third of the records are directed to the consumer, thereby guaranteeing the relevance of the observed latencies.

It is essential to note that, due to time constraints, the experiments conducted in this research exclusively employed Strings as input. Unfortunately, this limitation prevented the examination of other input types during the testing process.

### 5.1.2 Data adaptation

The various implementations used for this project utilize different types of record objects to store record information. However, a consistent adaptation approach is employed across all implementations to ensure fairness in system comparisons. This adaptation involves tokenizing the words within the input sentence and sorting them in alphabetical order.

Notably, the time required to adapt the input and generate the record object is deemed invariant when assessing system performance. This is because the adaptation duration is consistent across all implementations and does not provide any discriminatory information.

### 5.1.3 Repartitioner

The repartitioner is the node that reads the data that is written to the Kafka server by the record processing job about how partitions are being formed, and evaluates whether a recalculation of the system's partitions is needed or not.

Figure 5.2 illustrates how the repartitioner works. As it can be observed, if a repartition is required, the repartitioner stops the job, creating a savepoint in the process, and calls the state modifier (that will be executed in the main cluster) with the arguments required for its execution (jobName, jobId, etc.).

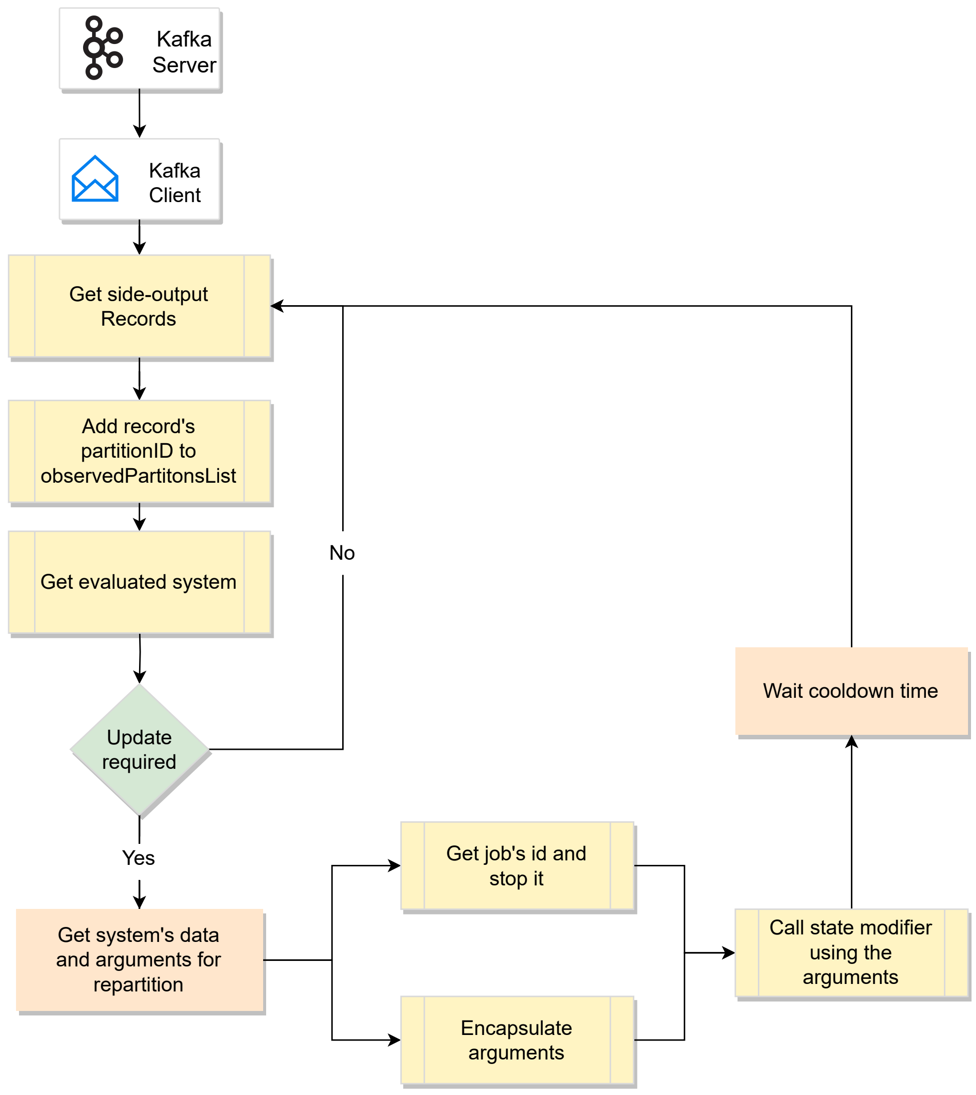

FIGURE 5.2: Repartition calculation algorithm.

The repartitioner makes sure that the program has been executed for at least 30 seconds before triggering a repartition, and once this repartition is triggered it checks again if 30 seconds have passed since the program resumed its execution to trigger the next one, in case the conditions for a repartition have been met again.

### 5.1.4 State modifier

The state modifier will load the savepoint information, modify it, and resume the record processing job with the modified state.

Figure 5.3 displays how the state modification algorithm works. Once the modifier decodes the arguments sent by the repartitioner, it retrieves the path where the savepoint information has been saved. It will then proceed to read the data, modify it, and store it in a new savepoint location. Once this process has finished, it will resume the job from the modified savepoint.

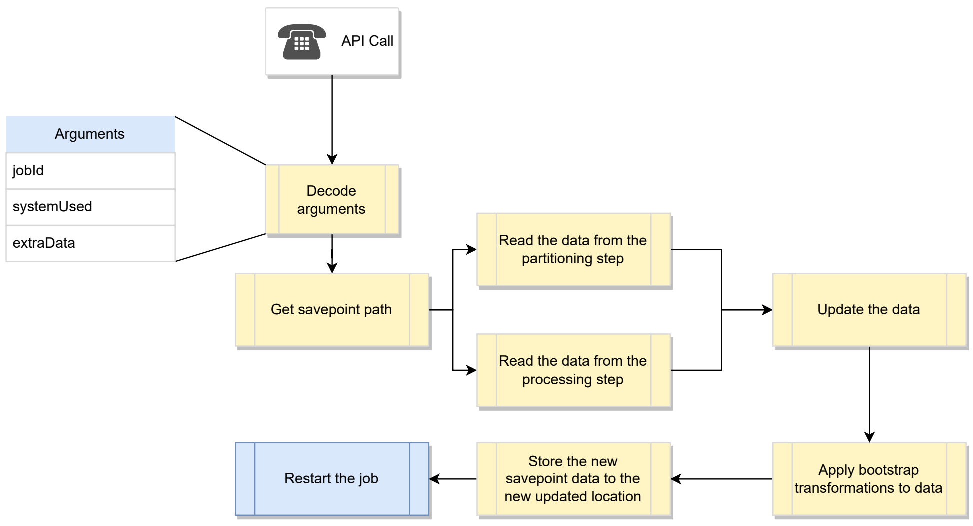
API Call
Arguments
jobId
systemUsed
extraData
Decode arguments
Get savepoint path
Read the data from the partitioning step
Read the data from the processing step
Update the data
Apply bootstrap transformations to data
Store the new savepoint data to the new updated location
Restart the job

FIGURE 5.3: Generic state modifier algorithm.

### 5.1.5 Metric readers

The last component of the system consists of two Kafka consumers tasked with reading the metrics data written by the system to the Kafka server, which pertains to system performance metrics such as processing latency, required operations, and more.

When a predefined quantity of data, as specified by the user, is received, it will be written to a log file. This log file will subsequently be utilized for evaluating system performance based on the metrics outlined in section 2.6. This systematic approach ensures a comprehensive and standardized assessment of the system's performance, facilitating a thorough evaluation of its capabilities.

## 5.2 Systems implementations

This section aims to provide an analysis of the various adaptations made to the systems detailed in Section 4, which enable them to execute similarity joins in a distributed manner within stream processing environments.

### 5.2.1 Distributed Streaming Set Similarity Join

The authors designed [38] to function in stream processing environments in a distributed manner, minimizing the need for significant modifications. The initial system was originally built utilizing the Apache Storm framework [32]. However, it has been adapted to work with the Apache Flink framework [11] in this project.

The repartition algorithm utilized by this system involves leveraging the processed record's length information to periodically evaluate the load balance across partitions. If any partition is deemed imbalanced, the algorithm recalculates the new partitions and dispatches them to the state modifier, which then executes the algorithm depicted in Figure 5.4.

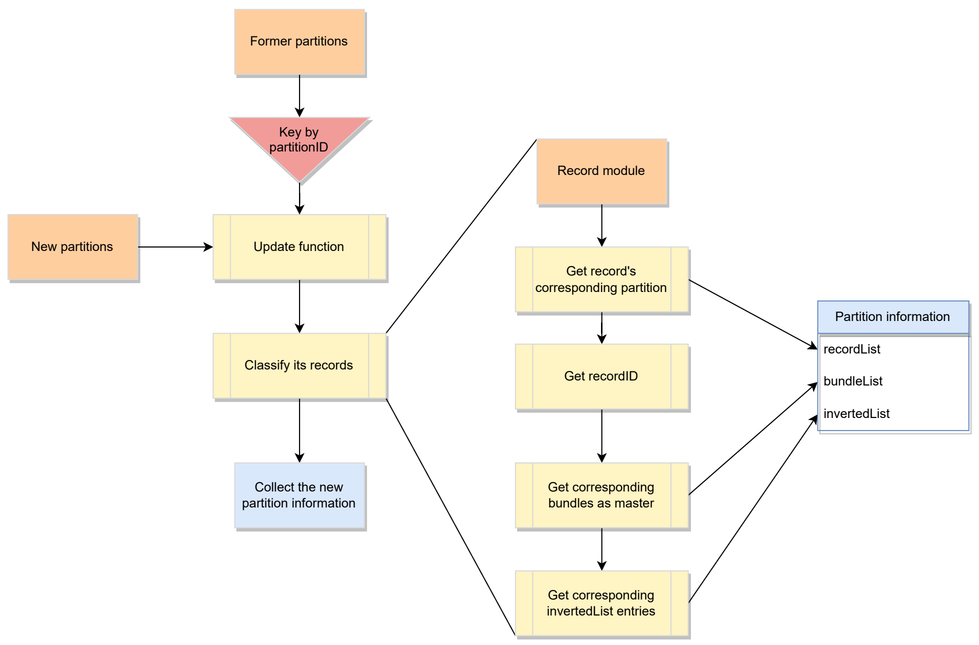

FIGURE 5.4: State modifier algorithm for the DSSSJ system.

As depicted in Figure 5.4, the state modifier is responsible for updating the partitioning operator's partitions, enabling new records to be dispatched to their designated partitions for processing.

Next, the modifier consolidates all the records from the processing step, facilitating their redistribution to their corresponding partitions. This step necessitates an update to all data structures in the processing operator, including the record, bundle, and inverted lists. Following this process, the updated information is saved as a savepoint, and the program execution resumes.

### 5.2.2 Cluster Join

In the case of the system defined in [15], the authors did not intend to run it in a stream processing environment, so some adaptations needed to be done. The first of these adaptations would concern the selection of the anchors.

It is important to ensure that the data used for creating the anchors is representative of the dataset being used in the experiment, and that every record can be sent to at least one anchor. As different datasets were used for the experiments, different sets of anchors were used for each one. The sets of anchors were generated through the utilization of the lists of unique words found in the datasets, as explained in Section 5.3.4.

A total of 500 anchors were selected, encompassing all the words from the specified word lists. Various experiments were conducted using different numbers of anchors to determine the optimal quantity for the system's implementation. The system's execution was evaluated with 250, 500, 1,000, and 2,000 anchors. The findings revealed that the number of anchors yielding the best results was 500. Despite the suggestion by the authors of [15] that a higher number of anchors may be ideal, their evaluation did not employ the Jacquard similarity metric, rendering their results unsuitable for direct comparison to this project's work. The use of the Jacquard similarity metric in the experiments ensures the relevance and applicability of the outcomes to this specific study.

The system was originally designed to handle spatial data, necessitating adaptations for text data integration. To maximize the detection of potential joins, an algorithm was devised. This algorithm involves sending the record to all anchors that shares at least a token with the record as outer partitions and sending it to the one with the highest similarity as the home partition anchor.

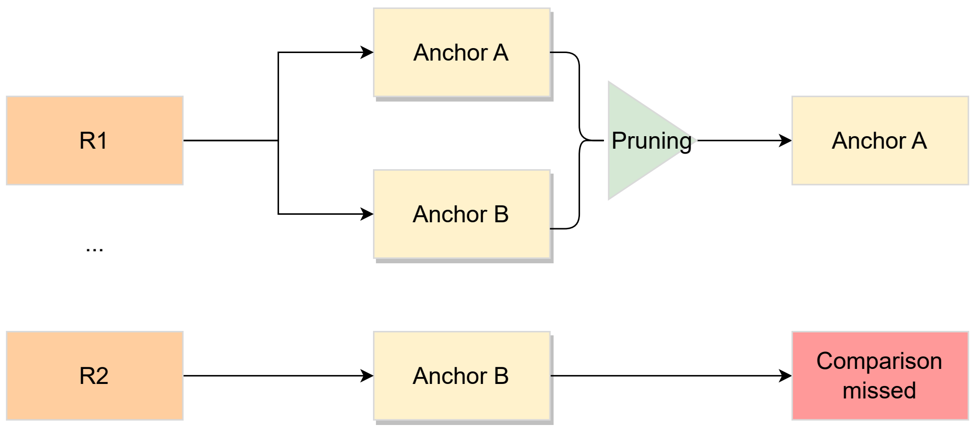

FIGURE 5.5: Limitations of the pruning mechanism proposed by [15].

As explained by Figure 5.5, it is not possible to apply the pruning technique that the authors proposed that it was based on comparing the ids of the records, as one outer anchor could have a lower id than the home partition anchor and be pruned while not sharing any token with the home partition anchor, meaning that potential comparisons could be skipped. As the priority is to reduce skipping matches to the minimum, no pruning could be executed, however as there are no repeated tokens in the anchors is not possible that a record is sent to two different anchors that share the same token with it.

Consequently, extensive data replication occurs, rendering the system akin to the Vernica join system. However, the authors did not propose any specific modifications for adapting their system to accommodate strings and this is the solution with more guarantees that could be implemented. Join replication will be measured as well as the number of unique joins observed.

The original system did not implement a repartition algorithm; instead, it expanded some of the selected anchors based on a sample of the whole static dataset. In the proposed implementation, the repartitioner will be in charge of measuring how many records have been assigned to each anchor and triggering an anchor expansion when the records assigned to an anchor have exceeded a certain threshold.

### 5.2.3 Vernica Join

In adapting the system described in [33] to a stream processing environment, certain modifications were necessary. It was decided to only implement the approach that makes use of GroupIds to group the different tokens as it is more efficient (requires less virtual partitions) and reduces the complexity of the system.

The initial system employed the extraction of a comprehensive token set comprising the constituent elements of a record's information from the static dataset subject to analysis. As previously expounded, word lists were meticulously generated, encompassing all tokens present in the datasets employed for the project's experimental purposes. These word lists, as elucidated in Section 5.3.4, serve as the token repositories for the Vernica join system.

The calculation of token frequencies in this study relies on the utilization of historical data associated with each dataset. When a token is absent from the available historical data, it is presumed to have occurred at least once due to its presence within the dataset used for the experiments.

The paper detailing the Vernica join system did not provide a repartitioning algorithm to balance the load between the various partitions. Consequently, an algorithm had to be developed to perform this task. This algorithm is triggered when the difference in the number of records assigned to one groupID and another exceeds a certain threshold. A high level description of the algorithm can be observed in Figure 5.6.

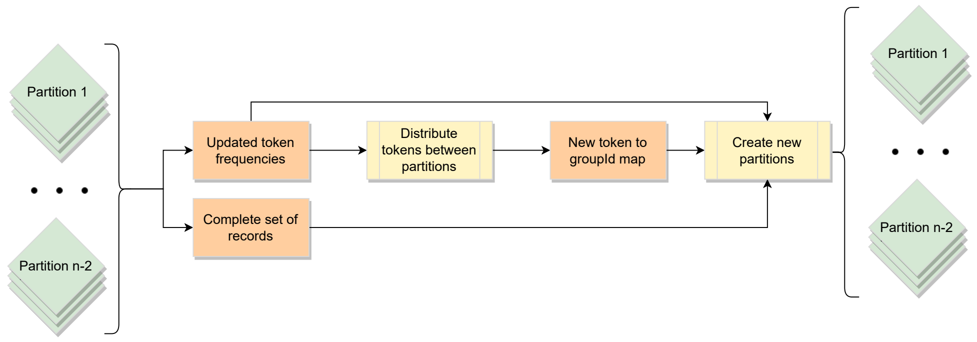

FIGURE 5.6: High level state modifier algorithm for the Vernica join system.

As it has been illustrated in Figure 5.6, after updating the token frequencies and creating set with all the records in the system, the tokens are distributed to create a new token to groupId map that is used to create new partitions.

The tokens and records associated with each groupID are redistributed in accordance with the algorithm described in Figure 5.7. Prior to redistribution, the token frequencies are sorted in descending order of the number of records associated with them.

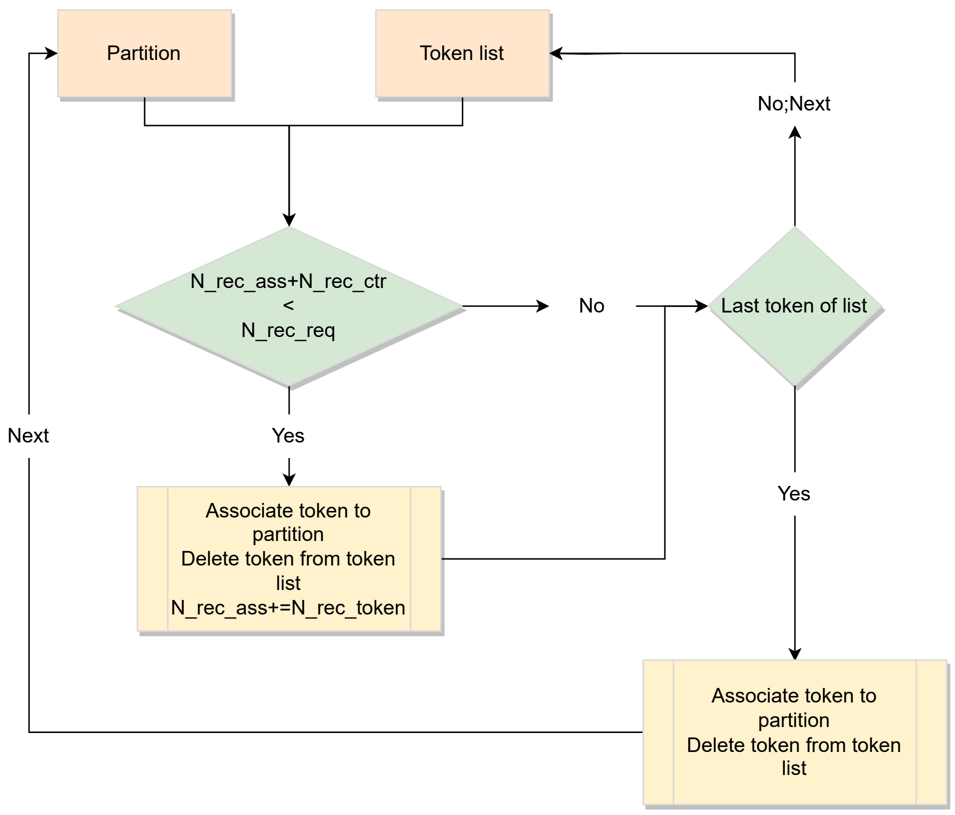
Partition
Token list
No;Next
N_rec_ass+N_rec_ctr
<
N_rec_req
No
Last token of list
Next
Yes
Associate token to partition
Delete token from token list
N_rec_ass+=N_rec_token
Yes
Associate token to partition
Delete token from token list

FIGURE 5.7: Greedy algorithm for the Vernica system state modification.

In Figure 5.7, N_rec_ass represents the number of records associated with the partition being processed, while N_rec_ctr represents the number of records associated with the token being processed. Additionally, N_rec_req represents the minimum number of records that each partition should have associated with it. This value corresponds to the total number of records divided by the number of partitions in the system.

As Figure 5.7 illustrates, the greedy algorithm implemented ensures that the records are fairly distributed across the partitions, although each partition will have more records than necessary assigned to it, except for the last partition. Consequently, load balance cannot be guaranteed entirely due to the heterogeneous nature of the number of records associated with each token.

### 5.2.4 Finding an incorrect number of joins

Following the execution of testing experiments, it was observed that despite the claims of perfection and absence of skipping joins by all the systems, there was a particular implementation that exhibited the potential to skip joins or detect false joins, particularly under conditions of low similarity thresholds. These implementations are identified as the DSSSJ and the Cluster join implementations.

#### Cluster join

In the case of the Cluster join implementation, a noteworthy scenario arises where a single record might possess multiple anchors that could serve as the home partition anchor or it should be indexed in less than optimal anchors to ensure complete comparison with any potential candidate.

Consequently, this situation may lead to two records, which are deemed to be potential matches due to the presence of shared tokens, failing to achieve a correct combination to facilitate a successful join. Moreover, the absence of any predetermined sorting order for the anchors introduces variability in the number of obtained joins across different executions.

Figure 5.8 visually depicts this situation, wherein two records that should ideally be compared (given their common tokens) are either compared or not, contingent on the selection of an anchor as the home or outer partition for them.

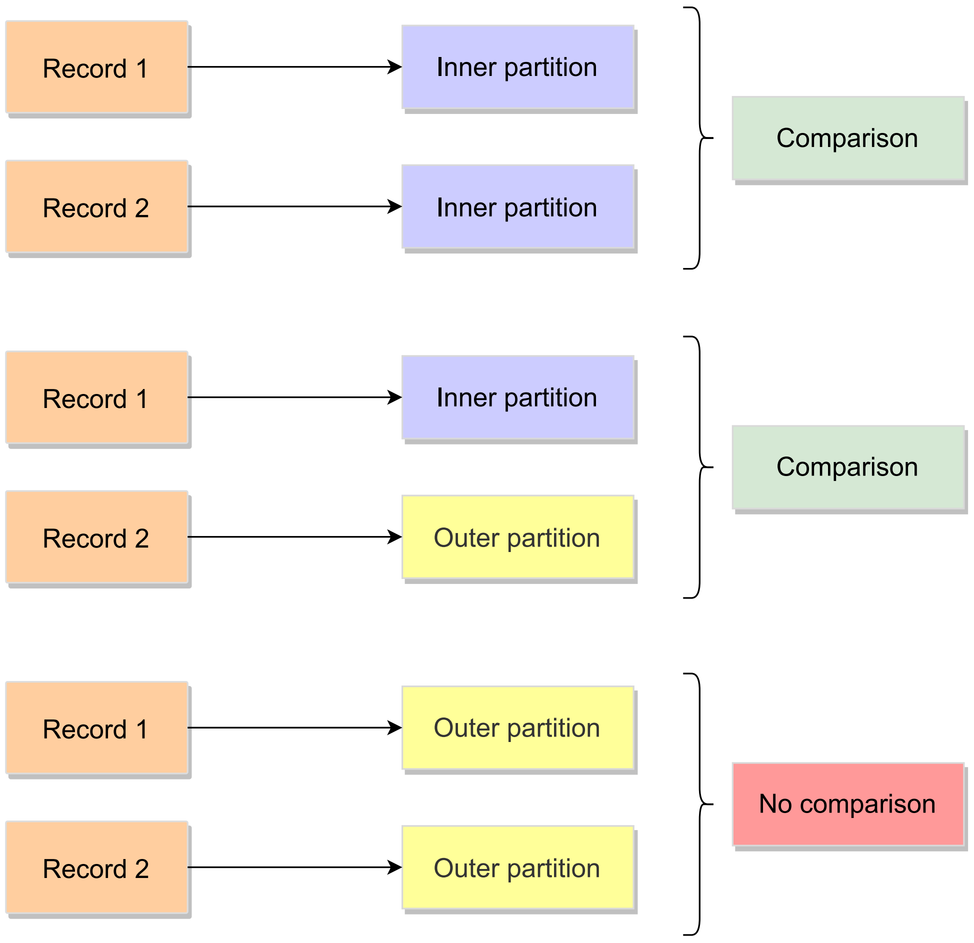

FIGURE 5.8: Example of the Cluster join system skipping a comparison.

The frequency of encountering such cases is higher than anticipated, especially when employing low similarity thresholds. This is attributed to the existence of a wide range of potential anchors that could be considered as the main one for the records, thereby significantly increasing the likelihood of inadvertently skipping a join. In some cases as much as 75% of the joins were skipped by this system.

Having numerous main anchors for the same record could address this, but doing so will likely degrade the system's already subpar performance (which is demonstrated in the results section 6). Therefore, it was chosen to leave the algorithm untouched because the system's performance is already poor, but acknowledge this limitation.

### DSSSJ

The case of DSSSJ presents a notable point of interest within the context under consideration. Upon an examination of Lemma 6 as presented in the scholarly work, depicted in Figure 5.9, it becomes evident that certain scenarios exist wherein the applicability of the mentioned lemma is limited.

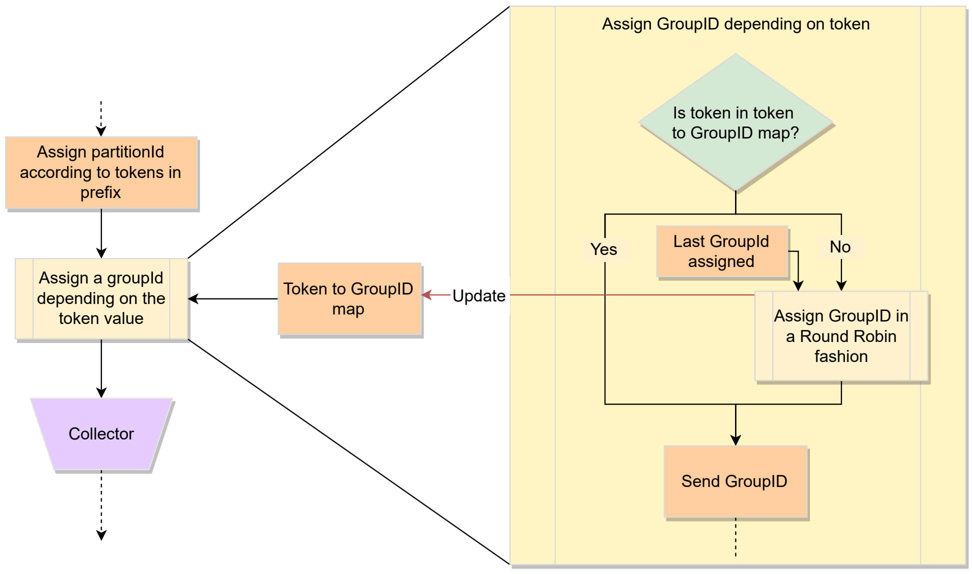
Assign partitionId according to tokens in prefix
Assign a groupId depending on the token value
Collector
Token to GroupID map
Update
Assign GroupID depending on token
Is token in token to GroupID map?
Yes
Last GroupId assigned
No
Assign GroupID in a Round Robin fashion
Send GroupID

FIGURE 5.9: Lemma 6 as presented in [38].

An instance exemplifying this scenario is delineated in Figure 5.10. In this context, should the similarity threshold measure below 0.143, Lemma 6 maintains its validity, potentially resulting in the identification of a join erroneously, as both records showed have a master of a bundle in common but do not share any tokens with each other.

It is noteworthy that this problem is found only in configurations characterized by low similarity thresholds. Thus, avoiding the usage of such configurations ensures the accurate program execution; however, this concern remains significant.

The paper states that it doesn't skip any true joins, and this is accurate. However, it's important to note that just because true joins are handled correctly, it doesn't eliminate the possibility of also encountering false joins.

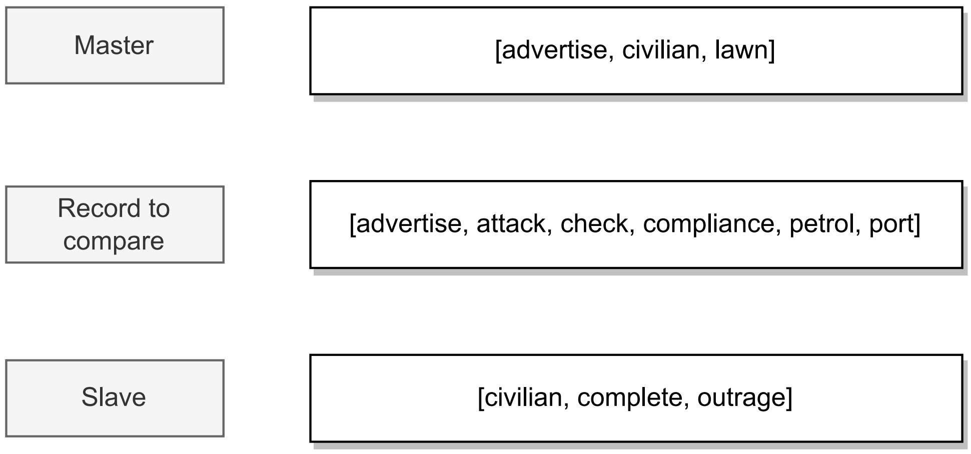
Master
[advertise, civilian, lawn]
Record to compare
[advertise, attack, check, compliance, petrol, port]
Slave
[civilian, complete, outrage]

FIGURE 5.10: Lemma 6 exception in DSSSJ.

This condition also manifest that comparisons that would not be made by just using a simple inverted list instead of the bundle join solutions are being made by the system, as in Figure 5.10 can be observed, the two records that are compared have no tokens in common and therefore that comparison could have been avoided in the first place.

### 5.2.5 Filtering

Both in the Vernica join [33] and Cluster join [15] papers is proposed the use of filtering in order to reduce the amount of comparisons per record. Both of them propose the use of the PPJoin algorithm, which is an extension of the PPJoin algorithm. The Naive solution can also benefit from the use of this algorithm., so it was incorporated as an extra to it as well.

The PPJoin algorithm incorporates filtering techniques to identify potential candidate pairs for similarity join based on token length and prefix token sharing.

PPJoin extends this algorithm by introducing positional filtering, utilizing an inverted list to track token positions in records. However, in dynamic settings such as stream processing environments, the use of an inverted list makes PPJoin+ more computationally expensive compared to the simple PPJoin algorithm.

In consideration of the cost-effectiveness of implementing PPJoin+ algorithm, it was determined that the benefits derived from conducting additional comparisons were outweighed. As a result, the decision was made to solely implement the PPJoin algorithm as a filtering mechanism for the tested systems. This filtering algorithm will be applied exclusively during the processing step and not during the partitioning step. This approach ensures that unnecessary comparisons are avoided while maintaining the efficiency of the overall process, so no joins will be skipped.

### 5.2.6 Metric readers

The system is composed of two metric readers that measure different statistics about the system's performance by reading the data written to the Kafka server by the program executor and analyzing it. Each metric reader reads from a separate topic.

Within the system, a specific metric reader is responsible for capturing data pertaining to the number of operations necessary for record processing, irrespective of the presence or absence of a join, along with the associated processing time. In this context, an operation denotes the comparison made between two records.

To mitigate the overhead associated with writing this record level information to Kafka, a design decision was made to collect data for only one third of the records. This sampling strategy was deemed sufficiently representative of the system's overall behavior while reducing the overall data collection burden.

The other metric reader is in charge of analyzing the number of joins found by the system. This metric reader collects statistics about the number of joins found and the latencies that the records that obtained a join experienced. All the obtained joins are written to Kafka. This metric reader removes the duplicate joins written to Kafka by the systems by maintaining a unique set of the pairs of joins that were observed. This is especially useful for the Vernica system, as it tends to produce a lot of duplicate joins.

Other metrics such as the amount of replication observed in the system or the amount of records that the system has processed are obtained by querying the REST API provided by the Apache Flink framework [3].

## 5.3 Experimental setup

This section aims to outline the essential details regarding the experimental setup, including datasets utilized, hardware specifications, software tools and versions and any relevant parameters or configurations.

### 5.3.1 Comparison metric

Given that the data used to evaluate the systems consists entirely of strings, the most appropriate comparison metric was selected, the Jacquard similarity. The decision to utilize this metric was particularly fitting due to its compatibility with string-based data and its widespread adoption in academic research.

By employing the Jacquard similarity metric, the framework developed for this research becomes highly applicable and accessible to future researchers seeking to evaluate their own systems, ensuring consistency and comparability across various studies.

### 5.3.2 Datasets employed

In order to facilitate the experiments, a combination of synthetic data, generated with parameters under controlled conditions, and real-world data from various text sources with varying specifications, such as the record length, was utilized.

As most of the systems employed in this study are meant to find similarity joins between records composed of strings and used Jacquard similarity as evaluation metric, the datasets used in the experiments are only text datasets from various sources.

#### Synthetic data

The experimental synthetic data utilized in the study comprises sentences of varying lengths. More specifically, it encompasses sentences with a range of 1 to 15 words (referred to as "Synthetic115" dataset) and sentences consisting of precisely 10 words (referred to as "Synthetic10" dataset).

The choice of the first range was motivated by its ease of handling for any system. This range ensures that some matches will be found and is similar to the length range typically found in tweets. Additionally, it provides a variety of lengths for the DSSSJ system to work with.

The second range was selected to analyze the behavior of the DSSSJ system when working with data of a single length. By focusing on sentences with exactly 10 words, the system's performance can be observed based on its ability to partition records according to their length.

To construct the sentences within the dataset, the list of 5,000 words provided by [27] was utilized. These words were read from the list, and each one could be used more than once to compose the sentences. The decision to employ a limited set of words was deliberate, as it allowed for the possibility of sentence similarity while maintaining a manageable dataset size.

#### News articles

A subset of news articles offered by [14] was utilized for this study. Specifically, articles corresponding to the timeframe of May 2022 were selected, capturing a period characterized by the onset of the Ukraine conflict but encompassing diverse news topics.

The dataset employed followed the Web ARChive (WARC) format [24], which contained a lot of extra data that was not relevant for the study, so as much of this data as possible was removed, as the main focus was the body of the news articles. However, even in the main body of the news articles, additional website-specific information such as JavaScript or HTML code could be found that was not possible to remove. Despite these inherent limitations, similar records could be found within the dataset by using the appropriate thresholds.

However, after performing a set of experiments it became clear that no system was going to be able to process a reasonable amount of records from this dataset per second, being most of the systems unable to process more than 1 record with a 99th percentile latency of one second.

Due to the substantial number of tokens per record and the limited resources of the workers, utilizing this dataset to obtain meaningful results became unfeasible. The combination of large record sizes and resource constraints imposed significant challenges in processing and analyzing the dataset effectively.

As a result, it was necessary to exclude this dataset from the analysis to ensure accurate and manageable experimentation with the available resources.

#### Amazon reviews

A subset of the Amazon reviews, sourced from [1], was utilized in this study. These reviews were available in TSV (Tab-Separated Values) format. The primary focus was on extracting the textual content from each review, which was subsequently employed for comparative analysis.

The reviews within the dataset demonstrated diverse lengths, ranging from zero characters (in cases where only a rating was provided, and such sentences were removed) to a maximum limit of 5,000 characters. On average, the length of the reviews in the dataset amounts to approximately 30 words.

This variation in review length adds complexity to the analysis but also offers valuable insights into the nature of the data and the potential challenges associated with processing and evaluating it.

#### Yelp reviews

A subset of reviews from Yelp [39] was also used. These reviews are available in a JSON format, with a specific field labeled as "text" that encompasses the review content, which is utilized by the system for comparison purposes.

Yelp reviews share certain characteristics with Amazon reviews. They have a maximum length of 5,000 characters and can potentially consist of no text at all.

On average, the reviews in the dataset presented a length of 100 words. It is important to note that this average length indicates a relatively longer textual content compared to Amazon reviews, which may contribute to a relatively lower potential for similarity between Yelp reviews,which was demonstrated throughout the experiments, as well as the difficulty of processing such long sentences.

#### Tweets

This dataset that is used for the experiments comprises a subset of more than 1.6 million tweets [26]. These tweets are presented in a CSV format, and the conducted analysis focuses solely on the tweet content, which is located in the final field of each line.

Tweets are restricted to a maximum length of 280 characters, with an average length ranging from 75 to 120 characters. More specifically, the dataset used for the experiments in this project contained an average of 10 words per record.

Consequently, the data obtained from this dataset is notably smaller compared to the previously mentioned datasets. However, this reduced size presents a unique interesting opportunity for identifying similar records. Moreover, tweets often exhibit trending patterns, which can be effectively captured using the systems implemented in this project.

### 5.3.3 Data preprocessing

To optimize the data reading process from the various datasets, a decision was made to reduce their sizes to approximately one gigabyte of data each. This reduction was achieved by selecting only a subset of entries from each dataset and writing the corresponding text to plain text files.

Due to the Tweets dataset being initially smaller than the other datasets, only 100 megabytes of data were available for use in this case. By curating the datasets in this manner, the processing and analysis of the data became more manageable, enabling efficient experimentation and evaluation of the systems.

All sentences written to the text files used for ingestion are stripped of any non-alphabetical characters, as well as any embedded code (such as JavaScript and HTML) found within them. Furthermore, the sentences are transformed to lowercase to facilitate easier and standardized processing.

As the size of the entries in each of the datasets varies, the amount of entries for each of the data types is different. Table 5.1 displays the number of records that can be used for each dataset. The number of entries of each dataset is sufficient as the performed experiments do not last long enough or use a record rate high enough to use all of them.

| Dataset | Entries |
|---|---|
| Tweets | 1,431,768 |
| News | 137,017 |
| Amazon | 6,153,816 |
| Yelp | 1,771,197 |

TABLE 5.1: Available entries per dataset.

For the historical data that is required by the DSSSJ and Vernica systems the records that followed the ones used to create the text files were used, as they share similar characteristics to ones preceding them.

Each of the datasets created as historical data has a size of 100 megabytes (except the tweets dataset that only contains 10 megabytes of data), so the number of entries corresponding to each dataset is equal to 10% of the entries numbers exposed at Table 5.1. In the case of the synthetic data, an historical file of 100 megabytes was created for them, but with a seed different from the one used for the experiments to avoid complete resemblance.

### 5.3.4 Word lists

To ensure a comprehensive representation of the datasets' information and facilitate the assignment of records to suitable anchors, distinct word lists were created for each dataset. These word lists aimed to capture the entirety of the dataset's content.

In order to streamline the storage and utilization of relevant information, a filtering approach was implemented. Specifically, only words appearing in two or more records were retained, while those occurring in a single instance were discarded. This filtering criterion helped eliminate potentially insignificant or idiosyncratic terms that offered limited value for analysis purposes.

Furthermore, to further optimize the word lists, certain restrictions were imposed. Non-alphabetical characters were removed from the word set, ensuring that only meaningful words were included.

Moreover, specific preprocessing steps were applied to refine the dataset. For instance, any JavaScript and HTML code present in the news dataset was eliminated, aiming to eliminate extraneous programming-related artifacts. Additionally, a uniform lowercase format was adopted for all words during the creation of datasets intended for writing into Kafka. This standardization facilitated consistent and reliable downstream processing and analysis.

Overall, these measures ensured that the generated word lists encompassed relevant and meaningful terms, improving the accuracy and effectiveness of subsequent analytical tasks.

| Dataset | Unique words |
|---|---|
| Tweets | 176,984 |
| News | 624,112 |
| Amazon | 323,837 |
| Yelp | 51,825 |
| Synthetic | 4,893 |

TABLE 5.2: Unique words found per dataset.

Table 5.2 shows the amount of unique words found in each of the datasets that were used for the experiments. As it can be observed, even though Yelp is not the smallest dataset it contains the least unique words among the real world datasets, probably because of the reviews are quite similar. As expected, the News datataset contains the largest amount of unique words, as news articles tend to use a richer and broader vocabulary than other types of text.

Creating the anchors for the Cluster Join system involved dividing the unique words equally among the selected number of anchors, determined to be 500 as the most optimal quantity. However, this approach led to lengthy comparisons between the anchors and the records, especially in datasets with a substantial number of unique words. Since anchors typically contain more tokens, the execution of Cluster Join became slow and time-consuming. The slow execution can hinder the system's performance and impede its scalability, highlighting the importance of carefully considering the selection of anchors and the impact on system efficiency.

### 5.3.5 Parameters

To conduct the experiments, various parameters that influence the execution of the different join systems were systematically varied. The key parameters include:

1. Parallelism: Quantity of tasks that can be executed concurrently. In the context of the studied scenario, it determines the number of Flink task managers operating simultaneously.
2. Partitions: Number of partitions assigned to each task manager. In these experiments, its value will remain case always 1.
3. Record rate: Number of records per second that the data generator creates.
4. Similarity threshold: Minimum level of similarity required for two records to be considered a match. In this project only the jacquard similarity was considered. The chosen value for this parameter will be determined by the selectivity for each dataset.
5. Filters: Determines whether the implemented PPJoin filtering [34] is used or not.
6. Repartition: Determines whether the implemented repartition methods will be used or not.

The selection of values for each of the parameters in the experiments will be elaborated on in the following subsections.

**Parallelism**

The determination of the best parallelism for a given system, without extensive resource consumption, presents challenges. Therefore, commonly employed values for job parallelism 5, 20 and 30 have been selected.

The initial intention was to utilize a parallelism level of 50 however, it became evident that the cluster was unable to effectively handle such a high degree of parallelism, resulting in numerous errors. The same problem was faced when using a parallelism of 40. Consequently, a deliberate decision was made to reduce the parallelism to 30, with the primary objective being the attainment of meaningful and reliable results.

In the experiments, the performance of each system will be assessed under different parallelism settings to identify the optimal configuration for subsequent experiments.

Each of the task managers of the systems will be assigned a single CPU, and for the rest of the services (flink job manager, consumers, producer, Kafka and zookeeper server) 21 CPUS will be available for every parallelism setting, in order to make the comparison as fair as possible.

To maintain consistency and fairness across the experiments, an equal amount of RAM, specifically 300 gigabytes, will be allocated to each experimental setting. This approach guarantees that the system's performance is not hindered or biased by varying RAM availability and only on the system capability of distributing the task managers.

### 5.3.6 Number of partitions

As each of the task managers will only be assigned a single CPU, each one will be only responsible for one partition, so the number of partitions present in the system will be equal to the number of task manager pods running. This reduces the complexity of the system and secures an efficient resource utilization. Having a single partition helps the task manager to correctly balance the load between the available resources as well as making the system's state management easier and easier to scale.

### 5.3.7 Record rate

The record rate will determine the number of records that the data generator will produce per second. As it was mentioned before the entries of each dataset used in this experiment vary in size, meaning that in most experiments different records rates will be used for each dataset to ensure the correct operation of the system.

As it will be explained later, the main goal of the experiments conducted during this project is to find the maximum throughput that a system can process before experiencing latencies above 1 second for the 99th percentile of the records being processed. When using each of the selected datasets and different parallelisms, there will be no static value for the record rate; It will be dynamically calculated to achieve a latency of one second for processing a record.

To compute the average number of operations necessary for processing a record and the extent of record replication experienced by each system, a distinct set of experiments has been devised. These experiments are characterized by utilizing a constant rate of 50 records per second and generating a total of 5,000 records only. This deliberate restriction ensures equitable comparisons across the systems. Notably, the focus of these experiments is on the computation of average operations per record and the observed replication, with no consideration for latency experience. The experiments have been meticulously conducted for each system, encompassing all available datasets and varying parallelism configurations.

### 5.3.8 Similarity threshold

To determine the threshold of similarity to be utilized during the execution of an experiment, the selectivity of each dataset will be employed. This entails establishing the threshold based on the proportion of similarity joins achieved out of the maximum possible similarity joins that could be identified.

For the purpose of the experiments, selectivities of 0.01%, 0.05%, 0.1% and 0.5% will be employed. To ascertain the similarity thresholds corresponding to these selectivity levels, a gradient descent algorithm will be utilized, aiming to achieve a threshold accuracy of six decimal places.

Figure 5.11 illustrates the operation of the algorithm devised to determine the various selectivities. As depicted, the algorithm progresses from higher similarity values towards lower values in order to identify the appropriate selectivity for a given dataset. The experiments are executed using the Naive system outside the cluster, using a text file that contained the first 15,000 entries of the dataset being tested, except for the Yelp dataset, that due to the length of its entries was slower to process, so 7,500 entries were used instead.

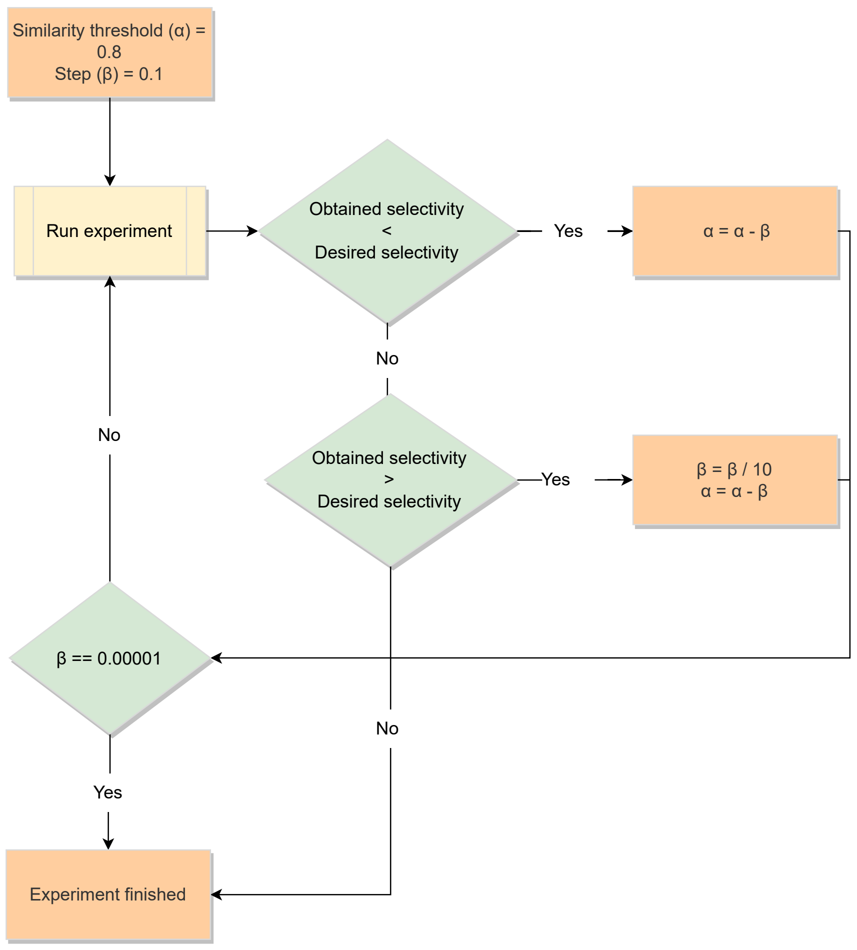
Similarity threshold (α) = 0.8
Step (β) = 0.1
Run experiment
Obtained selectivity < Desired selectivity
Yes
α = α - β
No
Obtained selectivity > Desired selectivity
Yes
β = β / 10
α = α - β
No
β == 0.00001
No
Yes
Experiment finished

FIGURE 5.11: Selectivity finder algorithm.

After conducting the experiments, the similarity thresholds associated with each of them were determined and presented in Table 5.3. The results reveal an inverse relationship between selectivity and similarity threshold. However, for the cases of 0.1% and 0.5% selectivities, the Synthetic10 dataset did not produce sufficient matches to establish a suitable similarity threshold.

| Selectivity (%) | Tweets | Amazon[2] | Yelp | Synthetic115 | Synthetic10 |
|---|---|---|---|---|---|
| 0.001 | 0.500001 | 1.000000* | 0.28355 | 0.250001 | 0.111111 |
| 0.01 | 0.285731 | 1.000000* | 0.256891 | 0.099992 | 0.052632 |
| 0.05 | 0.214282 | 0.499993 | 0.234082 | 0.090903 | 0.052623 |
| 0.1 | 0.190473 | 0.399994 | 0.224373 | 0.076924 | - |
| 0.5 | 0.142854 | 0.222215 | 0.201384 | 0.041665 | - |

TABLE 5.3: Similarity thresholds identified for each selectivity across datasets.

Within the framework, there exists the capability to control the selectivity of the synthetic data for a given similarity threshold. However, for the majority of experiments, a low selectivity of 0.001% was predominantly adopted to avoid the necessity of utilizing this option. This particular selectivity value represents the lowest

[2]*Dataset's lowest selectivity was 0.024% as there were several identical records in the used dataset.

observed during the investigation, thereby corresponding to the highest similarity thresholds.

Consequently, this choice of selectivity ensures the least occurrence of skipping or false joins, as elaborated in subsection 5.2.4. Throughout the experiments, consistency in the synthetic data was maintained by employing an identical random seed for sentence generation, thereby enabling the comparability of results across each execution.

## 5.4 Experimental design

In order to assess the proposed systems in this project, the sustainable throughput for each experimental dataset outlined in section 5.3.2 was obtained.

The experiments were conducted by joining two different streams of data. The records from one stream can only be joined with the records of the other stream and vice versa, however all the systems and the framework it self are adapted for having a single stream of data and performing self joins.

An investigation was carried out with the objective of ascertaining the approximate record rate at which systems are capable of sustaining a processing latency below one second for the 99th percentile of processed records. These experiments were denoted as "sustainable throughput experiments".

The decision to adopt the "processing" latency as the selected metric is driven by its capability to isolate the time required for record processing within the system. Unlike latency from record creation, which might be susceptible to network delays or external influences, the "processing" latency solely considers the system's internal processing time. This approach enables the acquisition of a more accurate and pertinent metric for the analysis. By concentrating on the processing time within the system, the evaluation yields a more reliable and meaningful result.

To perform these experiments, an algorithm was developed which is presented in Figure 5.12. It can be observed that if the latency transitions from below one second to above one second, or vice versa, between two executions, the step size is reduced by half. This process is repeated until either a step size of one is reached or the obtained latency precisely equal one second. Intermediate results are saved to facilitate analysis of the system's behavior.

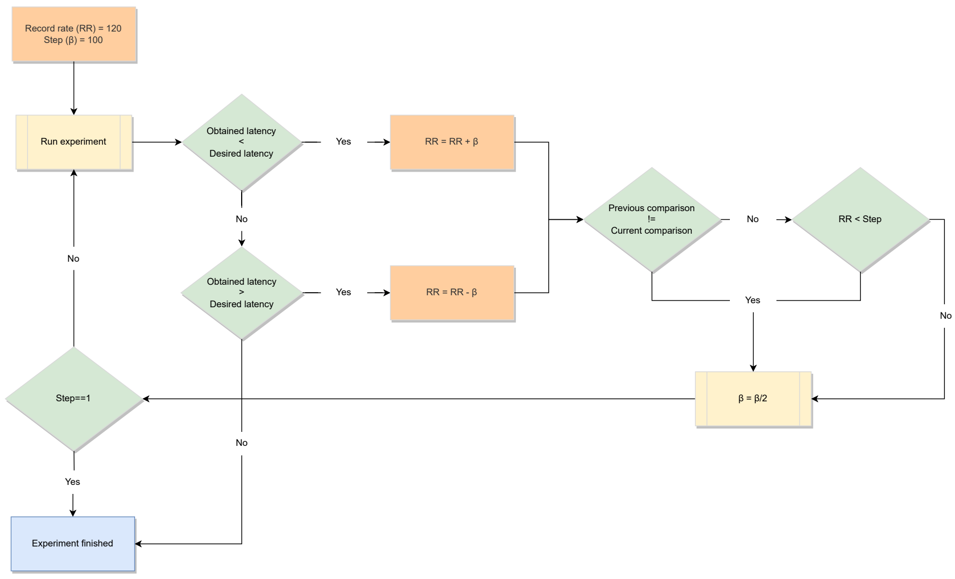

FIGURE 5.12: Sustainable throughput finder algorithm.

To expedite the experimental duration, a tolerance window of +/- 100 milliseconds was established to deem results as valid. This reduction significantly reduced the overall execution time of the experiments involving the four systems across the five datasets, decreasing it from approximately 96 hours to less than 24 hours per parallelism option.

Regarding the evaluation of metrics such as replication and operations per record, a fixed throughput was adopted for all systems, irrespective of their individual latency performance. This approach was chosen to ensure a fair comparison by ensuring that each system processes a comparable number of records. Bear in mind that the comparison between two records shall be regarded as the operative procedure throughout the entirety of the project.

The selected throughput for the experiments was set at 50 records per second, and to ensure a fair comparison between the systems, the producer was halted after precisely generating 5,000 records. Despite variations in the latency observed among the different systems, it is worth noting that the two metrics under investigation remain unaffected by these latency differences. A selectivity of 0.001% was also chosen to perform these experiments.

As all the systems are theoretically designed to not skip a single similarity join (as explained in the subsection 5.2.4, this does not hold up in the real implementations), there is little to no interest in comparing the selectivity of the systems, as for the same throughput they should all be the same, and differences among them could only be the product of how they were stopped or other events during their execution.

# Chapter 6

# Experimentals results

In this chapter, the results obtained from the various experiments that were performed will be displayed and analyzed in order to evaluate the different systems. As it has been described in section 4, each of the systems utilized in this project presented different algorithms for distributing the records and performing the joins, each of them presenting a series of advantages and disadvantages that will be explored in this chapter.

## 6.1 Results obtained with no repartition

This section presents an analysis of the outcomes derived from executing all systems without enabling the repartition capabilities. None of the systems employ the PPJoin filtering, with the exception of DSSSJ, which incorporates filters as part of its default implementation. The experiments were conducted across varying levels of parallelism, specifically 5, 20, and 30.

### 6.1.1 Sustainable throughput

Figures 6.1-6.3 show the sustainable throughput that was obtained for the different systems when using each of the before mentioned datasets when using parallelisms of 5, 20 and 30 respectively.

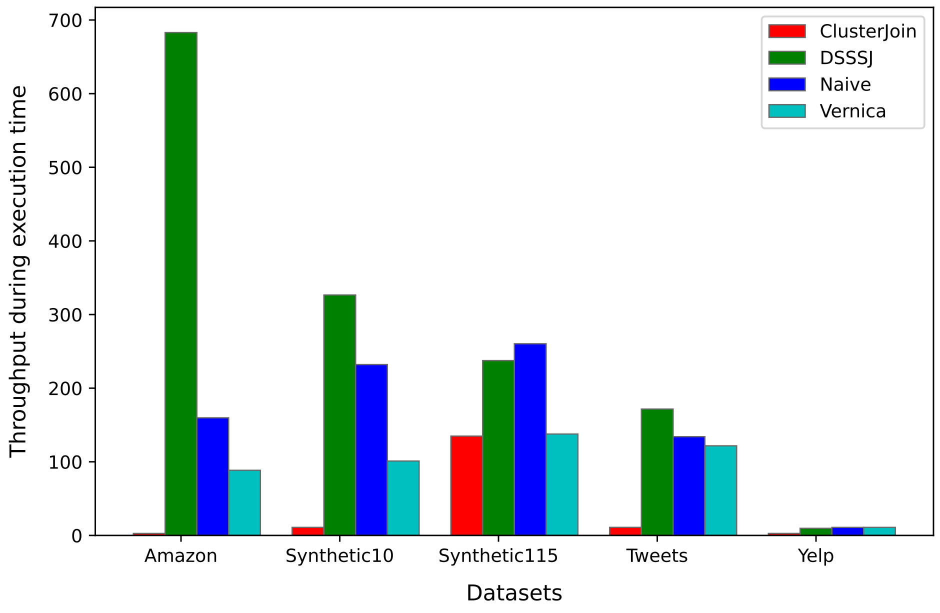

FIGURE 6.1: Sustainable throughput achieved with parallelism of 5 and repartitioning disabled.

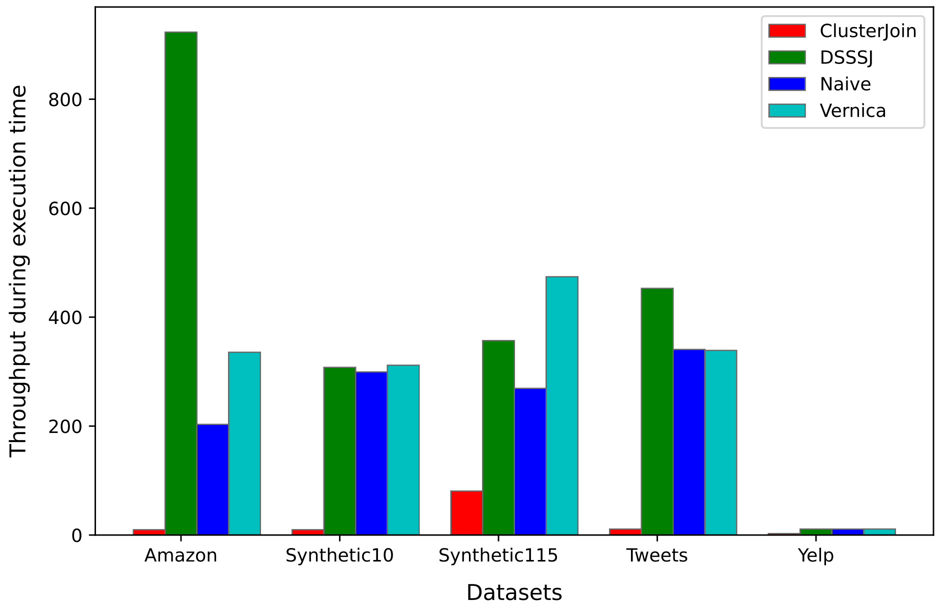

FIGURE 6.2: Sustainable throughput achieved with parallelism of 20 and repartitioning disabled.

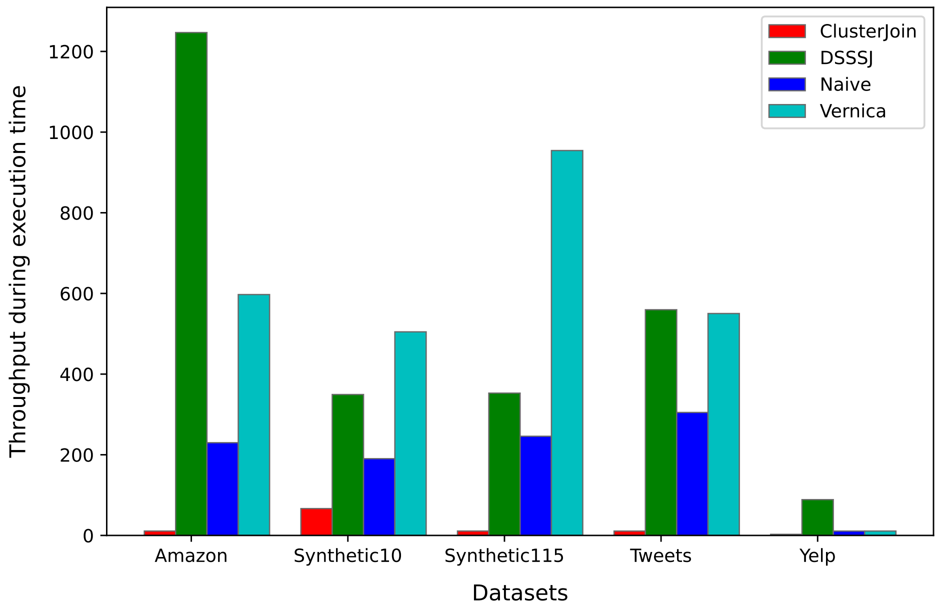

FIGURE 6.3: Sustainable throughput achieved with parallelism of 30 and repartitioning disabled.

Upon analyzing the three results figures, a significant observation emerges regarding the system's performance, which appears to be heavily influenced by the dataset under consideration. As anticipated, DSSSJ and the Vernica join demonstrate superior scalability, in line with expectations, given their specific design to process string records. In contrast, the Naive and Cluster join solutions exhibit a more generalized approach. This finding suggests that the choice of system should be carefully considered based on the nature of the dataset to be processed.

As observed, DSSSJ emerges as the most scalable system for all datasets, except for the synthetic dataset. The limitations in scalability for the Synthetic10 data arise from its uniform record length, resulting in an ineffective partitioning system. Consequently, all records are processed within a single node, thereby impeding the system's ability to scale. Similar issues are encountered with the Synthetic115 dataset, where the data cannot be distributed across more than 15 partitions. Consequently, this accounts for the nearly identical throughput obtained for parallelism values of 20 and 30.

Nevertheless, for datasets containing more varying record lengths, such as Amazon, Tweets, and Yelp datasets, the performance of DSSSJ surpasses that of other systems, exhibiting better scalability as well. Notably, in the case of the Tweets dataset, the difference in performance is less pronounced, mainly because the majority of records are of short length and are concentrated within the same partitions. Nonetheless, DSSSJ still demonstrates favorable scalability and outperforms other systems in these scenarios with datasets containing varying record lengths.

The Vernica join system also exhibits commendable scalability, particularly evident as the level of parallelism increases, facilitating a more even distribution of the workload across multiple nodes. Notably, this system demonstrates exceptional performance with synthetic datasets and, to a lesser extent, with Tweets dataset. This superiority can be attributed to the shorter length of records in these datasets, enabling faster processing.

However, the scalability of the Vernica join system encounters challenges with the Amazon and Yelp datasets, where longer records prevail. In such cases, a bottleneck is evident during the processing step of the system, primarily when comparing records with one another. The increased complexity of record comparisons for longer records hampers the system's ability to scale effectively in these specific scenarios.

The Naive system demonstrates commendable performance, emerging as the second-best performing system across all datasets when employing a parallelism of 5, with the exception of the Synthetic115 dataset, where it outperforms all other systems.

However, as anticipated, the Naive system faces limitations in scalability, as evident from the results obtained. The increased parallelism leads to a higher number of operations being skipped, but this advantage is outweighed by the higher replication experienced by the system's records. Notably, for parallelism value of 30, a notable degradation in performance is observed for the Synthetic and Tweets datasets, indicating that further scaling would not be feasible for these datasets using the Naive system.

Lastly, the results obtained from the Cluster join implementation are analyzed, which not only skips several joins but also exhibits significantly inferior performance compared to the other systems.

For a parallelism value of 5, the system's processing rate is less than 3 records per second for the Amazon and Yelp datasets, and only 11 records per second for the Tweets and Synthetic10 datasets. While the performance improves with the Synthetic115 dataset, processing almost 135 records per second, it still lags behind all other systems.

Moreover, the system demonstrates poor scalability, with only a slight performance increase observed for the Amazon dataset when transitioning from a parallelism of 5 to 20, but still considerably lower than other systems. Interestingly, for the Synthetic115 dataset, employing a higher parallelism results in processing even fewer records.

With a parallelism of 30, the system struggles to process more than 10 records per second for any dataset, except for the Synthetic10 dataset, where it manages 66 records per second, still falling well below other systems.

As previously mentioned, the Cluster join was originally designed for spatial data, and adapting it to handle strings led to a significant downgrade in its performance. Consequently, the system's poor performance is understandable, given the deviation from its intended use case.

It is worth noting that all systems, except for DSSSJ with a parallelism of 30, struggle to process more than 11 records per second when working with the Yelp dataset. This limitation can be attributed to the dataset's records, which demand more computationally expensive operations per record compared to other datasets, primarily due to their extended lengths.

This observation suggests that the systems would face considerable challenges when processing records from datasets similar to the News dataset, where the records exhibit exceptionally high lengths. The high computational cost associated with such lengthy records would likely lead to a significant reduction in the processing rate, making it challenging to achieve efficient data processing for datasets of this nature.

### 6.1.2 Record replication

As for the record replication experienced by the systems, the experiments were conducted as described in the subsection 5.4. Replication is only measured for this configuration as the use of filters and repartition should not influence the amount of replication observed by a system. The results obtained are reflected in Figures 6.4-6.6.

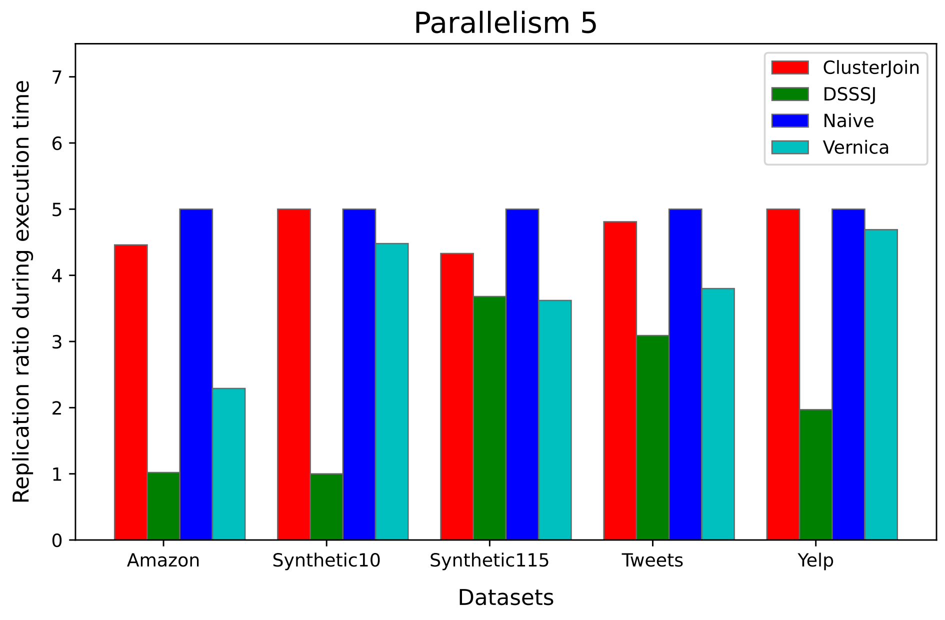

FIGURE 6.4: Observed record replication for a parallelism of 5.

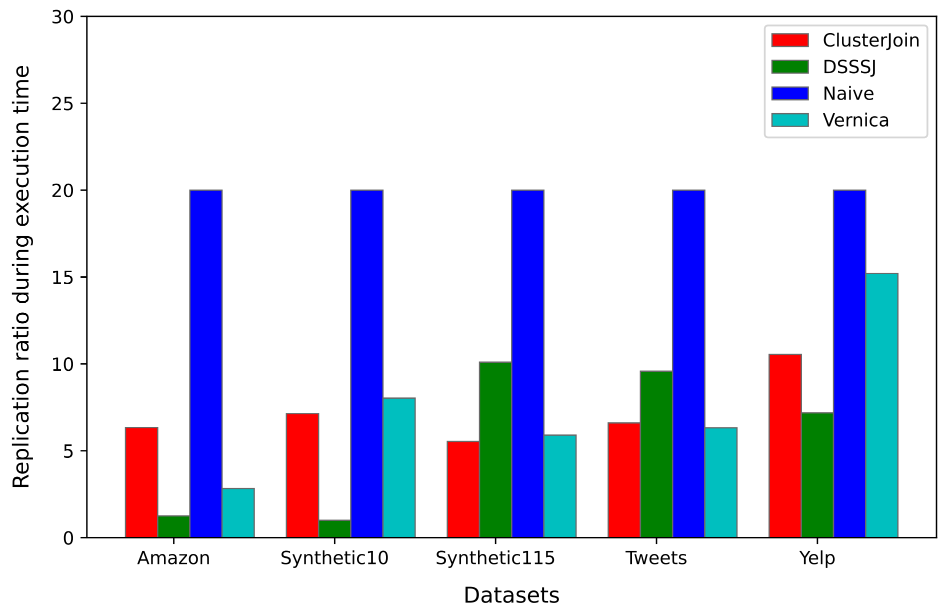

FIGURE 6.5: Observed record replication for a parallelism of 20.

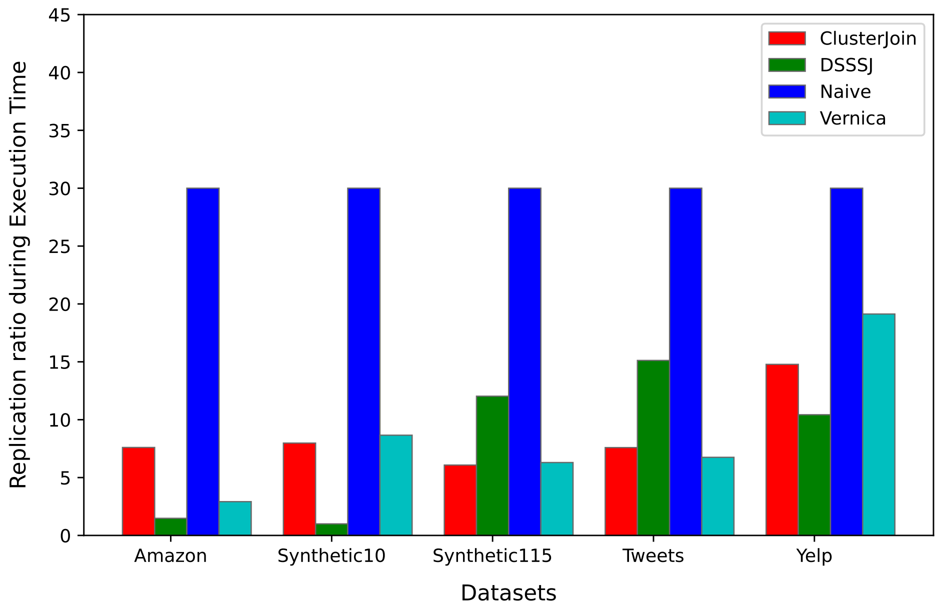

FIGURE 6.6: Observed record replication for a parallelism of 30.

The observations indicate that record replication in each system grows with higher parallelism, with the Naive implementation demonstrating the highest record replication as it aligns with the parallelism in all cases.

Cluster join demonstrates a consistently high replication ratio across all datasets when operating with parallelism 5. Notably, the replication ratio reaches its maximum possible value for the Synthetic10 and Yelp datasets. However, with an increase in parallelism, the replication ratio does not exhibit significant growth. This phenomenon explains why scaling up the parallelism does not lead to a substantial increase in throughput. One might expect that higher replication would result in better record distribution among nodes, reducing the load on each node and improving overall performance. Nevertheless, this is not observed in practice.

An interesting observation lies in the significantly higher replication ratio exhibited by Cluster join when used with the Amazon dataset, as compared to other systems (excluding the Naive solution). This disparity is likely attributed to the long length of the Amazon entries. While Vernica join only considers the prefix of records, Cluster join utilizes all tokens within a record for distribution. This characteristic accounts for the higher replication ratio observed in the case of the Amazon dataset.

Regarding the DSSSJ system, it consistently exhibits the lowest replication ratios across all datasets with a parallelism of 5. This suggests that limited partitioning occurs, leading to most records being allocated to the same nodes. Notably, this pattern is particularly intriguing in the context of the Amazon dataset, implying a concentration of data at specific lengths. Moreover, it is noteworthy that the DSSSJ system achieves the highest sustainable throughput in figure 6.1 using the Amazon dataset. This indicates that the filtering techniques employed by DSSSJ may play a crucial role in its success.

However, as the parallelism increases, replication ratios rise due to improved data space partitioning for most datasets, except for Synthetic10, which has only one associated length. Interestingly, despite Synthetic115 exhibiting a replication ratio similar to the Tweets dataset, DSSSJ achieves a sustainable throughput more than 100 records lower for Synthetic115. This observation suggests that other filters in the DSSSJ system may perform more effectively with the Tweets dataset.

The Vernica join system, like Cluster join, capitalizes on higher parallelism to increase the replication of records, thereby reducing the load on individual nodes within the system. When applied to the Yelp dataset, which comprises lengthy entries and a diverse array of tokens, the system exhibits a high replication ratio. Conversely, for the Amazon dataset, where records are not as lengthy and token diversity is lower, the replication ratio is not as high.

For the Synthetic10 dataset, the replication ratio approaches 10 for parallelisms of 20 and 30. This outcome aligns with the most efficient processing strategy for this dataset, where each token within a record is dispatched to a different node, optimizing record processing.

Another intriguing observation pertains to the sustainable throughput obtained for the Tweets dataset compared to the Synthetic115 dataset. Although the replication ratio for the Tweets dataset does not significantly differ from that of the Synthetic115 dataset, the sustainable throughput achieved for the former is lower. This discrepancy is likely attributed to the fact that records in the Tweets dataset contain more tokens. Consequently, the processing step of the system, involving record comparisons, becomes the limiting factor affecting system performance.

### 6.1.3 Average operations per record

This subsection presents the average number of operations performed to each record upon entering the processing stage of the system without repartitioning or filtering enabled. Each operation represents a comparison of a record with another record. The corresponding results are presented in Figures 6.7 to 6.9.

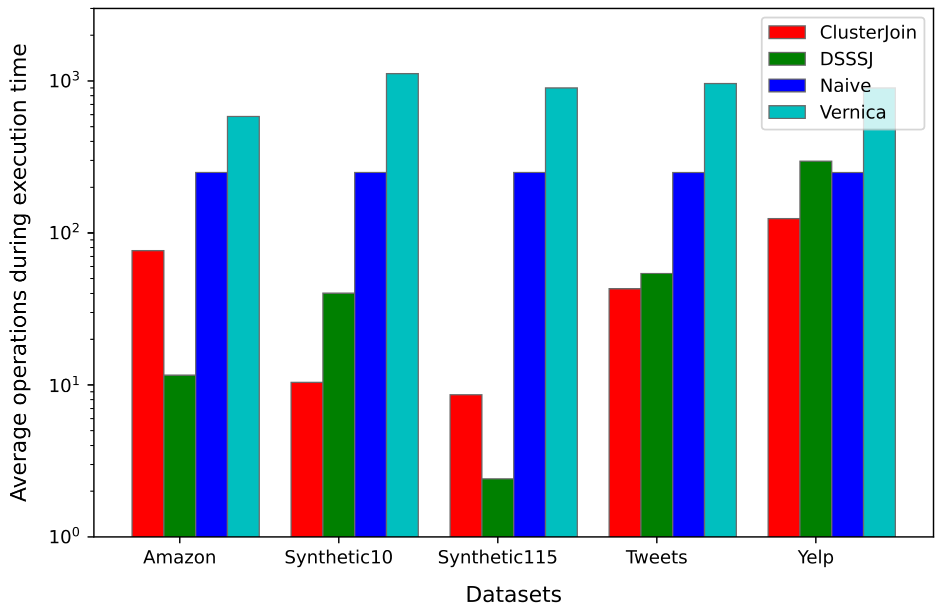

FIGURE 6.7: Observed average operations per record for a parallelism of 5.

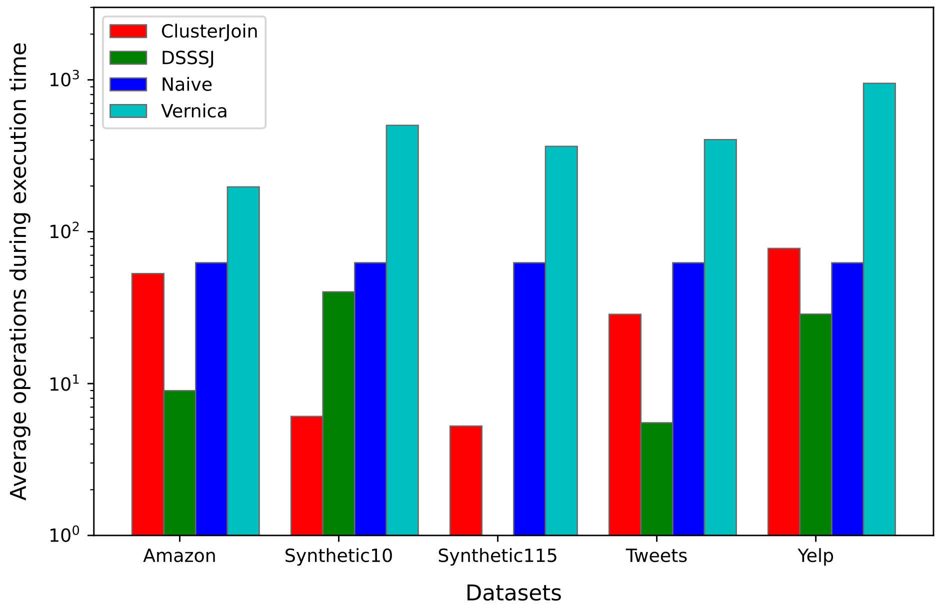

FIGURE 6.8: Observed average operations per record for a parallelism of 20.

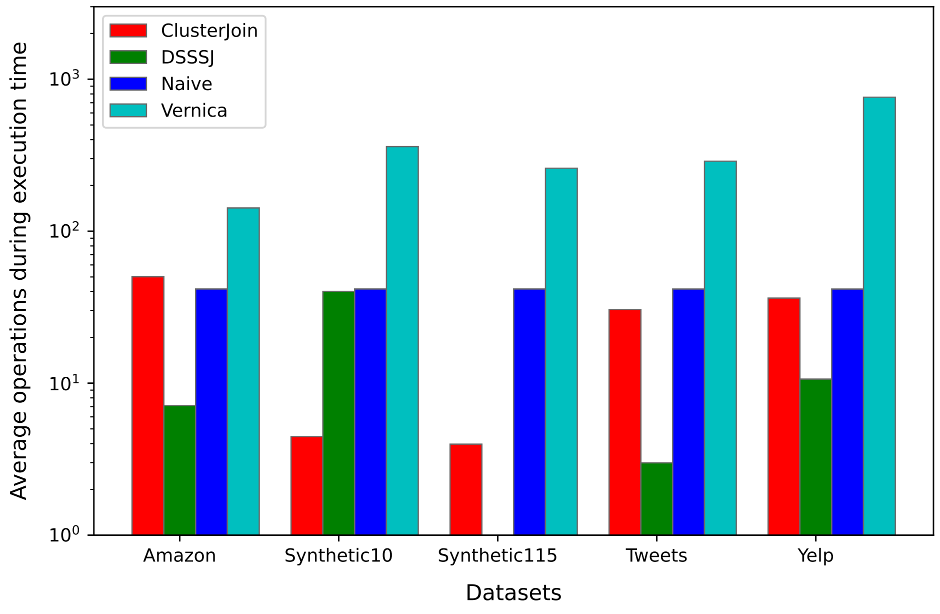

FIGURE 6.9: Observed average operations per record for a parallelism of 30.

The initial observation reveals that when parallelism is enhanced, all systems exhibit a reduction in the number of operations necessary for a join process. However, it is notable that certain systems demonstrate a greater propensity to capitalize on heightened parallelism compared to others.

As observed, Cluster join operates with minimal record comparisons, but we also have to take into account that it skips comparisons. The Synthetic datasets, characterized by low lengths and high replication ratios, particularly demonstrate the least number of operations required by the system. This observation aligns logically, as the increased spread of records with high replication reduces the number of necessary operations.

The finding that the system requires relatively few operations reinforces the notion that the bottleneck of the system lies in the partitioning step. Since records need to be compared with all anchors during this phase, it likely contributes significantly to the low throughput achieved by the system. Identifying this bottleneck is crucial for optimizing the system's performance and increasing its overall efficiency.

Regarding the DSSSJ system, the number of operations required is directly proportional to the replication needed, and it decreases as replication increases. Notably, the Synthetic10 dataset requires the same number of operations for all parallelisms since all computations are performed by just one node.

For the other datasets, DSSSJ achieves a lower number of required operations compared to other systems when using parallelism higher than 5. This indicates that DSSSJ excels at load balancing among its nodes, even though its replication may not be as high as the one observed in other systems such as the Vernica join.

The efficiency of DSSSJ's filtering and partitioning techniques is particularly evident in the Synthetic115 dataset, where less than one comparison is performed per record, showcasing the remarkable efficiency of these techniques.

Furthermore, the amount of operations in the Amazon dataset does not reduce as drastically when increasing parallelism compared to the Tweets and Yelp datasets. This behavior is logical, given that the replication experienced by the Amazon dataset is lower than that of the other two datasets, resulting in less significant reductions in operations with increased parallelism. This observation reinforces the influence of replication levels on system performance.

As expected, the Naive implementation demonstrates a decrease in the number of operations per record as the parallelism level increases. With more nodes available in the system, the workload per node decreases, resulting in fewer operations required per node. This reduction in per-node operations explains why the achieved throughput notably increases when the parallelism is increased.

However, it is noteworthy that the difference in operations required by the system when transitioning from parallelism 20 to parallelism 30 is not as significant as the difference observed between parallelism 5 and parallelism 20. This is reasonable since the amount of records per node is not heavily reduced in the shift from parallelism 20 to 30.

This observation suggests that the Naive implementation might not be as scalable as other systems, such as Vernica join or DSSSJ, as its throughput growth is not as substantial when parallelism is increased. In contrast, the other systems show higher scalability and achieve more notable increases in throughput with increased parallelism.

Upon analysis, it becomes evident that the Vernica join system requires a higher number of operations compared to other systems for making joins. However, surprisingly, this does not result in lower throughput. The key factor contributing to this efficiency is the highly effective partitioning system employed by Vernica join. Despite the higher number of operations per node, the system compensates for it by maintaining a relatively low replication ratio, especially when compared to the Naive solution.

Moreover, it is interesting to observe that the number of operations required for each record decreases significantly with the increase in parallelism. This trend is understandable as higher parallelism leads to increased replication, reducing the number of records per node and consequently the number of operations per node.

This reduction in per-node operations leads to a notable increase in the achieved throughput.

In summary, the Vernica join system's remarkable efficiency in partitioning and balancing the load among nodes allows it to perform well even with a higher number of operations per record. The system's scalability is evident as the throughput increases notably with increased parallelism, showcasing its ability to handle larger workloads efficiently.

## 6.2 Results obtained with repartition

This section exclusively presents results for systems equipped with a repartition algorithm, specifically DSSSJ, Cluster join, and Vernica join. However, subsequent experiments revealed that the repartitioning algorithm employed by Cluster join incurred excessive execution time, rendering it incapable of completing the repartitioning process within a reasonable time frame. Consequently, this system was excluded from further experimentation.

The primary factor contributing to this limitation is the necessity to replicate all records across partitions associated with the expanded identifiers once an anchor is expanded. This particular process is exceedingly resource-intensive and time-consuming, making it infeasible to accomplish within the program's execution.

This outcome was met with substantial disappointment, considering the extensive work and effort invested in the development of the repartitioning algorithm. Nevertheless, considering the already sub optimal performance of Cluster join, it became increasingly doubtful whether further partitioning of the space would have yielded any improvements. Consequently, the evaluation in this section focuses solely on DSSSJ and Vernica join.

The use of repartitioning also supposed the use of too many resources for the Vernica join system when executing it with the Tweets and the Amazon dataset with parallelism 5. The system, as the system could not handle the correct creation of a checkpoint for later modification of the state.

In this particular section of the experiments, the performed analysis will focus solely on the sustainable throughput of the system. Comparing the replication and operations required for each record rendered to be pointless in this context, as none of the systems triggered a repartition when processing only 5,000 records .

The total processing time is actually lower when the repartitioning feature is enabled, as a portion of the overall execution time is allocated to executing the repartitioning operations. This was taken into account when analyzing the results, and both the results for only the time allocated to process records and the total execution time will be exposed.

### 6.2.1 Sustainable throughput

The sustainable throughput achieved by each dataset and system during their respective execution times, excluding the duration of the repartitioning process (referred to as ET), and the total runtime of the experiment (referred to as TT), is presented in Figures 6.10 to 6.12.

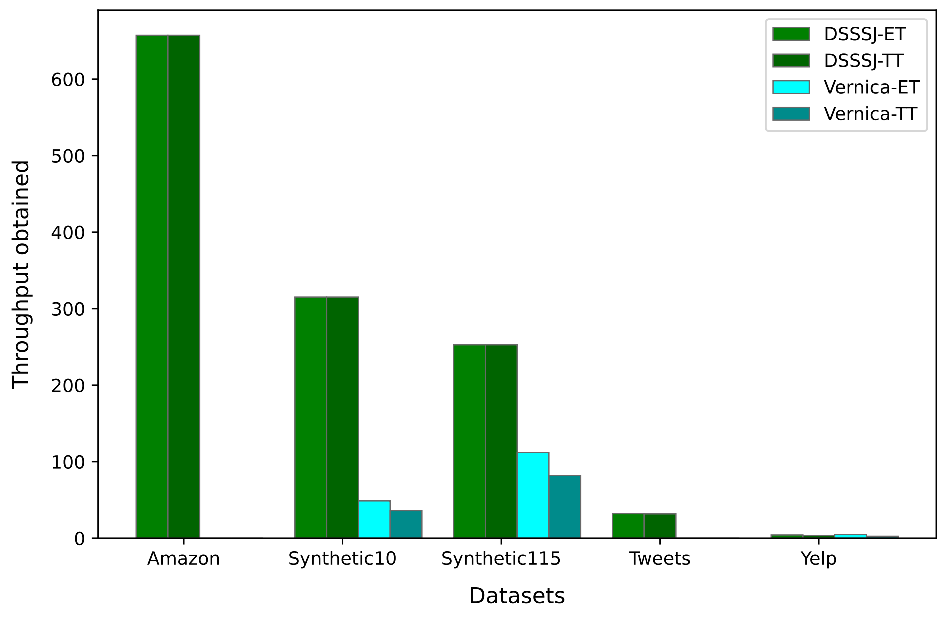

FIGURE 6.10: Sustainable throughput achieved with parallelism of 5 and repartition enabled.

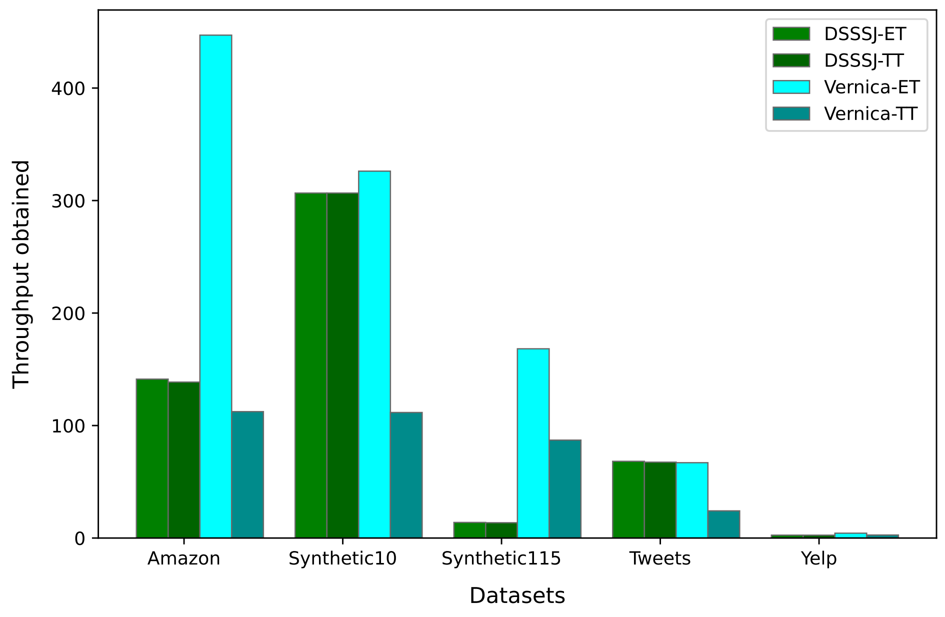

FIGURE 6.11: Sustainable throughput achieved with parallelism of 20 and repartition enabled.

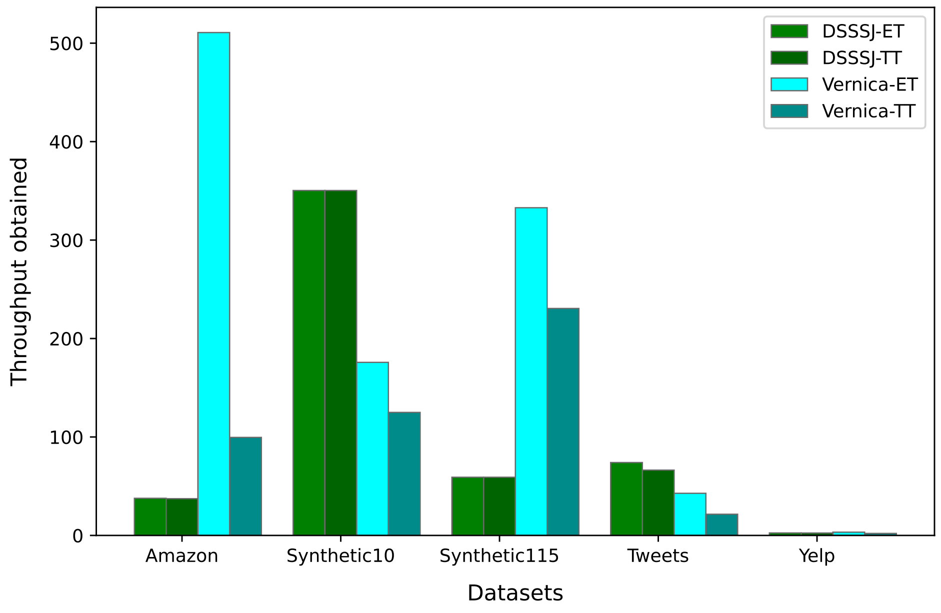

FIGURE 6.12: Sustainable throughput achieved with parallelism of 30 and repartition enabled.

The results demonstrate a lack of interest in employing a repartitioning algorithm for the systems under consideration. The obtained throughput is consistently lower when repartitioning was enabled compared to the cases where no repartitioning was utilized. Notably, in the instances where a relatively good throughput was observed (i.e., Amazon, Synthetic10, and Synthetic115 datasets for DSSSJ with parallelism 5), it is worth noting that the repartitioning algorithm was not actually triggered during the execution.

This observation further reinforces the notion that the implementation of the repartitioning algorithm did not yield favorable results in terms of sustainable throughput.

The bad results could also be attributed to how the experiments were implemented, as the total execution time is of only 180 seconds and a repartition can be triggered with a period of 30 seconds.

In the case of DSSSJ, although the repartitioning process was executed swiftly, the challenge emerged from dealing with all the records generated during the system's repartitioning, resulting in system saturation and diminished performance.

Notably, the Synthetic10 dataset exhibited the best performance among the datasets. This can be attributed to the fact that no repartitioning was triggered for this dataset, and thus, all records were directed to a single node. By avoiding repartitioning, the system was able to process the data more efficiently, leading to higher throughput.

As evident from the graphs, the throughput obtained during the time the system was running and for the entire execution duration displayed minimal difference. Consequently, the repartitioning mechanism seems beneficial primarily for resource distribution rather than achieving high throughputs. However, it is important to acknowledge that the throughput obtained with repartitioning enabled was significantly lower than when repartitioning was disabled.

In summary, the results indicate that while repartitioning might be useful for resource allocation, its application does not appear to be very effective in achieving high throughputs.

The case of the Vernica join did not show any significant improvement with the utilization of repartitioning. The repartitioning process demanded substantial time and resources, and as a result, while the throughput achieved during the system's execution was not inherently poor, it did not surpass the throughput obtained without repartitioning. Moreover, the overall throughput obtained throughout the entire experiment was considerably lower when repartitioning was enabled compared to the scenario where repartitioning was not employed.

The decision to use repartitioning in the systems proved to be an overall failure, as it did not lead to any noticeable performance enhancements. Instead, it necessitated an additional consumption of resources and introduced a significant level of complexity to the systems. The development and testing of the repartitioning component became the most time-consuming aspect of the project without yielding desirable outcomes.

## 6.3 Results obtained when applying filters

This section presents the results obtained from executing the Naive, Vernica join, and Cluster join implementations with activated PPJoin filtering. The experiments were conducted without enabling repartitioning to facilitate a comprehensive comparison of the results, taking into account both the achievable sustainable throughput and the number of operations required per record. This approach was chosen due to the previously observed inferior performance of Vernica join with enabled repartitioning, as demonstrated in the preceding subsections.

This approach allows for a comprehensive analysis of the systems performance while considering multiple performance metrics. As the filtering is only applied in the processing stage, replication will remain unaltered and therefore there is not point in analyzing this metric.

### 6.3.1 Sustainable throughput

Figures 6.13 to 6.15 present the sustainable throughput achieved by the various systems when employing parallelisms of 5, 20, and 30, respectively, with filtering enabled. These tables display the sustainable throughput for each of the mentioned dataset-system combinations.

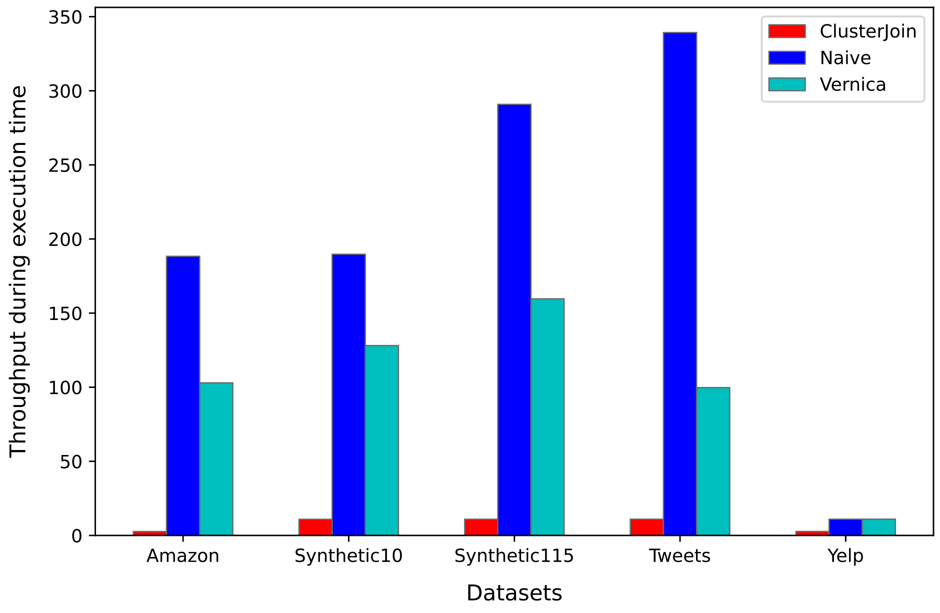

FIGURE 6.13: Sustainable throughput achieved with parallelism of 5 and filtering enabled.

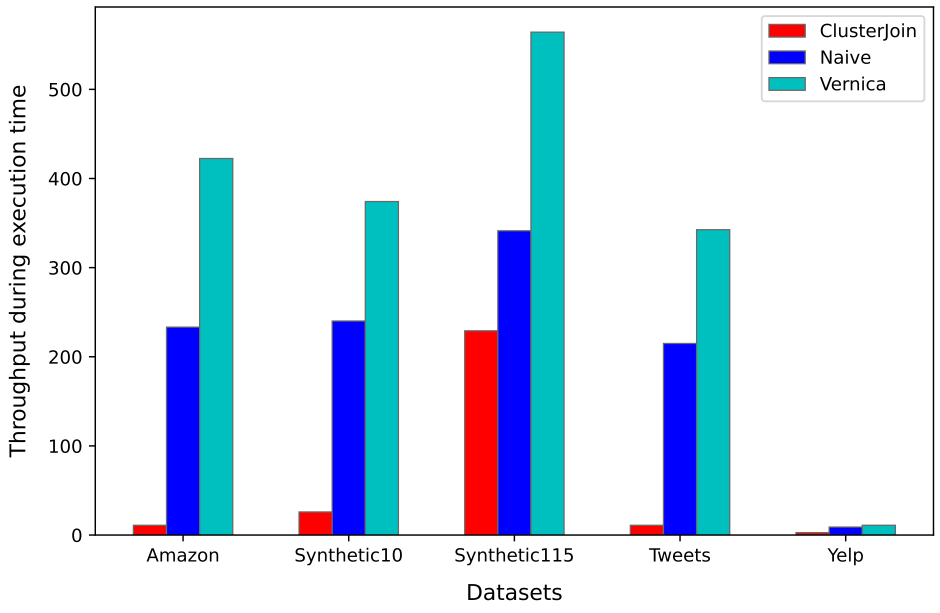

FIGURE 6.14: Sustainable throughput achieved with parallelism of 20 and filtering enabled.

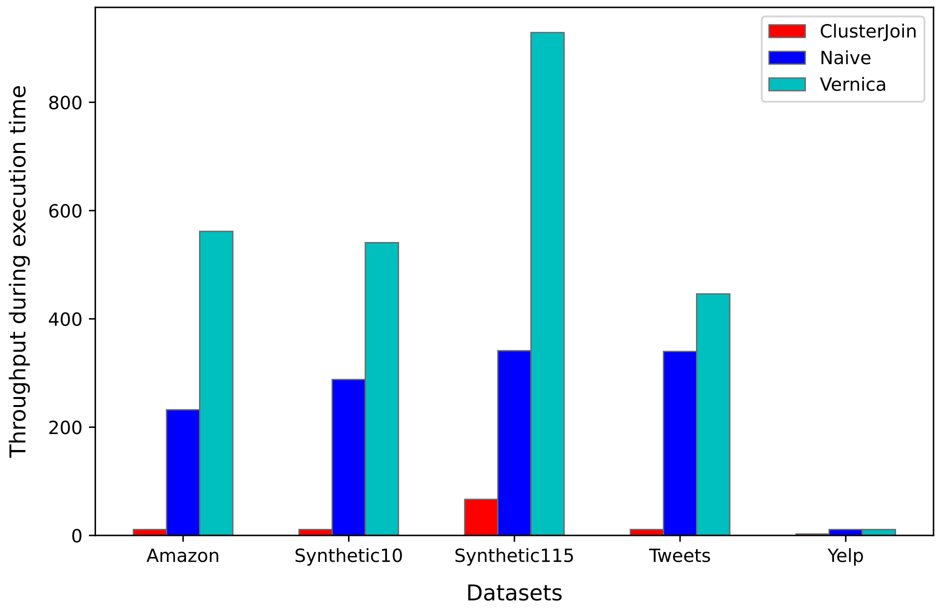

FIGURE 6.15: Sustainable throughput achieved with parallelism of 30 and filtering enabled.

As it is displayed, all the solutions experience some increase in their obtained throughput, even the Cluster join solution, when using the PPJoin filtering algorithm, however none of them experience drastic changes.

It can be observed, the performance of the Cluster join system remains quite poor when using filters for most datasets. However, there is a moderate improvement across all datasets, especially for those with shorter records such as the synthetic datasets and the Tweets dataset. The system's performance is significantly boosted when a parallelism of 20 is employed. Particularly, the Synthetic115 and Tweets datasets benefit from the length filtering, which reduces the number of comparisons needed and speeds up the processing of records.

Interestingly, the system's performance is lower when using a parallelism of 30 compared to a parallelism of 20 for all datasets. This can be attributed to the increased coordination required between nodes at higher parallelism levels. The higher costs associated with this coordination during the partitioning step likely contribute to reduced throughput, resulting in lower performance.

The performance of the Naive solution exhibits a notable increase in throughput for the Synthetic115 dataset, while other datasets do not experience significant improvements in their achieved throughput. This disparity can be attributed to the specific characteristics of the datasets, particularly the Synthetic115 dataset having more varied record lengths. The filtering process in this case effectively reduces the number of necessary comparisons, leading to improved performance.

On the other hand, for datasets with more homogeneous record lengths, the filtering might not substantially reduce the number of required comparisons. Consequently, the throughput improvement is not as significant. This observation is particularly interesting as the Naive solution requires a relatively high number of operations compared to the rest of the systems.

However, applying filters might require just as much computational resources as making full comparisons, especially when the load is evenly distributed among nodes, and communication becomes a major bottleneck for achieving high throughput in this system.

The communication overhead could potentially limit the system's ability to take full advantage of the filtering process and significantly boost throughput for datasets with more homogeneous record lengths.

When analyzing the Vernica join system's performance, it becomes evident that the achieved throughput is generally similar to the one achieved without using filtering techniques. The only dataset that experiences some increase in throughput is the Amazon dataset, but only when using a parallelism of 20. In fact, some datasets, such as the Tweets and Synthetic115 datasets, experience a decrease in achieved throughput when using a parallelism of 30. This observation indicates that the processing step is not the bottleneck of the system, and the use of filters does not significantly benefit the system's performance, specially when the similarity thresholds are not that high and the system cannot fully benefit from using the prefix filtering.

The reduced benefit of using filters can be attributed to the fact that, in some cases, the additional computations required by the filters do not compensate for the reduction in the number of comparisons that need to be made. This is especially true for datasets with short record lengths, such as the Synthetic115 and Tweets datasets, where the application of filters may not offer substantial advantages.

In conclusion, the use of filtering techniques is not a guarantee of increased throughput, and it should be considered selectively for datasets that contain records of varying lengths (to apply length filtering) or significantly different tokens in their prefixes (for prefix filtering). The efficiency of filtering largely depends on the dataset characteristics, and its potential benefits should be carefully evaluated before implementation.

### 6.3.2 Average operations per record

This subsection presents the average number of operations performed to each record upon entering the processing stage of the system without repartitioning or filtering enabled. Each operation represents a comparison of a record with another record. The corresponding results are presented in Figures 6.16 to 6.18.

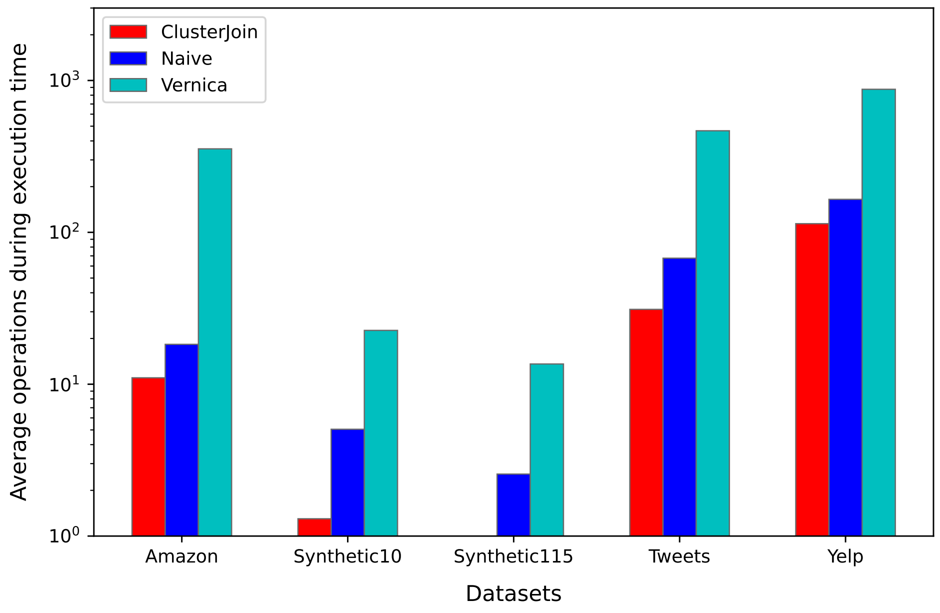

FIGURE 6.16: Observed average operations per record for a parallelism of 5 with filtering enabled.

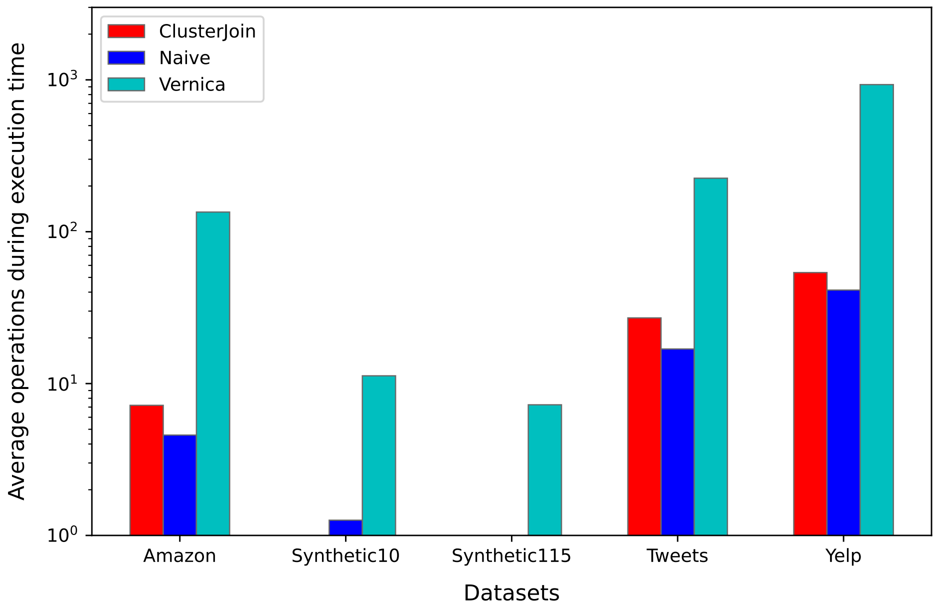

FIGURE 6.17: Observed average operations per record for a parallelism of 20 with filtering enabled.

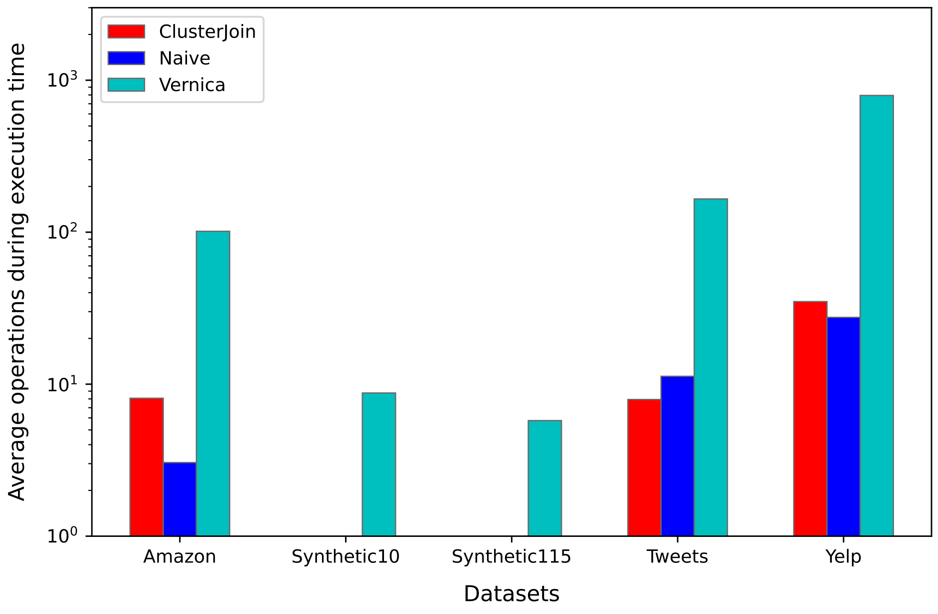

FIGURE 6.18: Observed average operations per record for a parallelism of 30 with filtering enabled.

As observed, there is a notable reduction in the number of comparisons between records required by each system for every dataset when using any of the parallelism settings during the experiments. This decrease in comparisons is expected as parallelism allows for workload distribution among nodes, resulting in fewer comparisons per node.

However, as mentioned earlier, this lower number of comparisons does not appear to fully compensate for the additional computations required by the use of filters. The benefits of filtering techniques might not be sufficient to overcome the computational overhead they introduce, especially for datasets with certain characteristics, such as short record lengths or similar tokens in prefixes.

When analyzing the results obtained by the Cluster join system, it becomes evident that the use of filtering techniques leads to a significant reduction in the number of comparisons required for specific datasets. For the Amazon and Synthetic115 datasets, the amount of comparisons is reduced by a factor of 10, while for the Synthetic10 dataset, it is reduced by a factor of 8.5 for every parallelism setting. This indicates that these datasets benefit the most from the application of filtering, which is due to the variety of lengths in the Synthetic115 dataset and the high similarity thresholds for the Amazon and synthetic datasets, which apply the prefix filtering and discard most of the records for comparisons.

Furthermore, there is also a reduction in the number of operations needed for the Tweets dataset, which explains the modest increase in the throughput obtained for this dataset. This observation suggests that the filtering techniques applied by the Cluster join system are particularly effective for these datasets, leading to improved performance and throughput. For the Yelp datases the amount of operations required does not differ as much from the case of not using filtering techniques and this explains why almost no increase in the achieved throughput is observed.

Upon analyzing the Naive solution, it is evident that there is a significant decrease in the number of operations required for every dataset, including the Yelp dataset. However, once again, the two datasets that benefit the most from the use of filtering are again the synthetic datasets and the Amazon dataset, for the same reasons as in Cluster join.

Among them, the Synthetic115 dataset experiences the most substantial reduction in the number of operations required, with less than one operation per join for parallelisms 20 and 30. This remarkable reduction explains why the biggest increase in throughput is observed for this dataset.

Furthermore, it is noteworthy that the amount of operations decreases significantly with the increase in parallelism. This reduction in operations is expected as higher parallelism allows for more efficient distribution of the workload among nodes, resulting in fewer operations per node.

The Vernica join system also shows a significant reduction in the number of operations needed for the synthetic datasets, and to a lesser extent, for the Amazon and Tweets datasets. However, the number of operations required by the Yelp dataset remains unaffected.

Among the datasets, the synthetic ones experience the most substantial reduction in operations. Interestingly, even though the synthetic datasets have the most operations reduced, the dataset that benefited the most was the Amazon dataset. Despite having a lower reduction in operations, the cost of a comparison between two records for the Amazon dataset is higher. This results in the observed increase in throughput for the Amazon dataset, showcasing the efficiency of filtering techniques for this dataset.

On the other hand, the Tweets dataset performed worse when using filters, as the reduction in operations was not enough to increase the achieved throughput. This observation highlights the importance of carefully analyzing and considering the characteristics of each dataset before deciding to use filtering techniques. Filtering benefits datasets with varying record lengths and diverse tokens in their prefixes, but its impact may vary based on dataset-specific factors such as the cost of comparisons and overall system setup.

# Chapter 7

# Conclusions

This section serves the purpose of presenting a comprehensive summary of the work carried out throughout the project. It aims to address the research questions formulated in chapter 1, analyze the findings, and draw relevant conclusions. Moreover, it explores potential future research directions that can be pursued based on the outcomes of this study, providing valuable insights for further advancements in the domain of distributed similarity joins in stream processing environments.

## 7.1 Summary

The research conducted focused on the evaluation of different systems to evaluate similarity joins. It was necessary to implement the systems using the Apache Flink framework [11] and evaluate them in a cluster with enough resources to guarantee their correct evaluation.

Answers to the research questions posed at the beginning of this project will now be used to explain the conclusions reached after conducting the experiments.

**How do the different systems compare in terms of efficiency, accuracy, and scalability?**

Based on the conducted experiments, it has been determined that DSSSJ exhibits superior performance across real-world datasets in terms of both achieved throughput and resource utilization. However, when applied to synthetic data with specific characteristics, DSSSJ demonstrated comparatively inferior results. Also, the system found false joins when using low similarity thresholds, an error that the Naive solution and the Vernica join solution do not suffer, so its accuracy is lower than that obtained by those two solutions.

The Vernica join, despite being developed over 12 years ago and not being designed to work in stream processing environments, remains remarkably efficient for data processing, yielding commendable throughput results. Nevertheless, it necessitates greater resource allocation compared to DSSSJ in most scenarios, although it does not miss any joins.

Furthermore, the Naive solution's scalability is severely limited, leading to suboptimal results relative to the considerable resources invested in its execution. This emphasizes the exigency for more sophisticated approaches to efficiently handle similarity joins; however, this approach should always be used as a baseline to compare the rest of the systems, both in terms of efficiency and accuracy.

Lastly, the Cluster join solution proves unsuitable for text datasets, underscoring the fact that not all solutions are universally applicable to all types of data, especially within stream processing environments; furthermore, the system is unable to correctly find all the similarity joins in the datasets used, rendering it a poor solution for processing text.

**Which tasks are the most suitable for each system, and how flexible are they?**

The answer to this question is similar to the one provided for the previous one. We have found that the type of data and its characteristics deeply affect the performance of the systems.

DSSSJ is ideal for high similarity thresholds, which guarantee lower data replication and better partitioning of the space, as well as a more efficient use of the Bundle Join algorithm. It also works better for data that contains a wide range of lengths, as it is focused mainly on distributing the records depending on their length.

Regarding the Vernica join system, it is also more convenient to have a high similarity threshold, as the prefix will be shorter and the records will be replicated fewer times. However, the system worked very well with the synthetic data, so it is apparent that the use of a dataset with a small vocabulary and sentences of shorter length is beneficial for this system.

In the context of the Naive solution, datasets do not exhibit any discernible advantage, and their performance is solely conditioned by the length of the records. Specifically, shorter records tend to yield superior outcomes for this approach, as the comparisons between records are less expensive.

In the case of Cluster join, acceptable performance was observed exclusively with synthetic datasets. This phenomenon can be attributed to the synthetic datasets possessing a smaller vocabulary, which facilitated the process of locating an anchor. Additionally, the shorter length of records in these synthetic datasets contributed to their improved performance compared to other datasets.

**How can the performance of the systems be optimized for different use cases?**

During the experiments, several ways to optimize the different systems for certain types of data manifested themselves.

In the context of DSSSJ, further testing of its algorithms should be done in order to find the reason why it misses some joins. This could be due to a misinterpretation of the algorithms on our part, but it seems that some of the mechanisms proposed by the authors are not defined well enough for another person to implement them correctly.

In the case of Vernica join, an optimal configuration involves utilizing a significant degree of parallelism. It is evident from the conducted experiments that its performance deteriorates when parallelism is set to a low value, such as 5. This outcome indicates that deploying a complex implementation like Vernica join is not advantageous when operating with limited parallelism. In such scenarios, it becomes impractical to employ this approach, and it may be more sensible to opt for simpler alternatives like the Naive solution.

The Naive solution can be optimized by adopting a different partitioning algorithm, such as employing a 2-D matrix to distribute the records. This method introduces minimal complexity to the system and could improve its performance. Although the Naive solution may not exhibit high scalability, it proves to be efficient and optimal in situations where parallelism is low. In such scenarios, its simplicity and straightforward implementation make it a viable choice for processing data effectively.

As for the Cluster join system, it is imperative to undertake additional efforts in its implementation to prevent missing joins, especially in cases where a record may have multiple home partitions. Addressing this limitation may slightly impact the system's performance. However, despite any optimizations, the experiments indicate that this approach may not be well-suited for handling text datasets effectively. Therefore, exploring alternative solutions becomes necessary to achieve better performance and results when dealing with text-based data.

## 7.2 Future work

In this section, some improvements to the designed framework are proposed, as are other potential experiments that could be performed using it.

### 7.2.1 Adapting the framework for numerical data

As it was explained in section 4.2, Cluster join is a system that is designed to work with vector data, as a vector space can be more easily partitioned into anchors and the amount of data replication is lower thanks to their distributing and pruning approaches. When using this type of data, Cluster join typically uses other similarity metrics, such as euclidean or cosine distances, in order to find which records are similar. Probably, when using this type of data, the results obtained by Cluster join are better than the ones that we obtained with our experiments.

However, due to time constraints, the designed framework has not been adapted for using vector data or similarity metrics other than Jacquard similarity. It would be interesting to implement functionalities that allow the use of vector data and other similarity functions to test the performance of cluster join and compare it with other approaches such as the Naive implementation or the solution proposed by [28].

### 7.2.2 Implementing other systems

The limitations in implementing all the desired systems and the potential of other existing systems, such as the Streaming hypercube [28] or Mass join [17] solutions, which present exciting avenues for future research and experimentation, Adapting other similarity join systems to be applicable in stream processing environments could provide valuable insights and comparisons with existing systems.

Furthermore, exploring the possibility of modifying certain solutions, like the Naive solution, to incorporate more efficient partitioning algorithms that could potentially yield superior results would also be a valuable avenue for future research.

It is crucial to recognize that not all systems are well-suited for stream processing environments. Stream processing introduces unique challenges, such as handling continuous data streams, real-time processing, and resource constraints. Therefore, careful consideration and adaptation of systems designed for batch processing or other paradigms are necessary to make them compatible with stream processing.

Exploring and adapting systems from various types of data processing approaches beyond map-reduce operations can lead to a richer understanding of the strengths and weaknesses of different technologies in the context of similarity joins in stream processing. Each system may have distinct trade-offs in terms of performance, scalability, and resource utilization in this dynamic environment.

### 7.2.3 Test the systems with different settings

Through the utilization of parallelisms set at 5, 20, and 30, a comprehensive evaluation of system scalability was conducted. Nevertheless, further insights can be gained by investigating system behavior at higher parallelism levels or by allocating additional resources to each task manager. Additionally, an interesting approach would involve allocating more CPUs for the task managers or adding more partitions to each of them. Such adjustments could potentially enhance the system's performance and resource utilization, providing valuable insights for optimizing the overall execution of the join operations.

While the current study presented results based on a specific selectivity, extending the investigation to incorporate multiple selectivity levels could provide a more comprehensive understanding of the systems' capabilities and limitations. It would enable researchers to assess how the systems respond to varying degrees of data similarity and how their performance scales accordingly.

Additionally, exploring the performance of the systems using alternative datasets would be of interest. For instance, assessing the behavior of the systems with the news dataset [14], given sufficient resources, could provide valuable insights. Furthermore, conducting tests using synthetic data that encompasses a larger number of tokens than the datasets employed in this project would yield valuable information for system evaluation.

### 7.2.4 Test the systems on the cloud

In the experiments conducted for this study, the systems were deployed on a cluster provided by Delft University of Technology, which offered ample resources for executing the designed benchmarks accurately. However, it would be interesting to explore the systems' performance on a cloud service capable of dynamically scaling resources on demand.

By utilizing cloud services such as Amazon Web Services [2] or Google Cloud [21], the execution of the systems could benefit from the ability to allocate additional resources as required. This approach would not only enhance the systems' efficiency, but also provide valuable insights into their adaptability to scaling in a real-world cloud environment. Testing under these conditions would offer a more comprehensive evaluation of the systems' capabilities in scenarios closer to practical, real-world usage.

# Bibliography

[1] Amazon. *Amazon Customer Reviews Dataset*. https://s3.amazonaws.com/amazon-reviews-pds/readme.html. Accessed on May 20, 2023. 2023.

[2] Amazon. *Amazon Web Services*. https://aws.amazon.com/. Accessed on July 31, 2023.

[3] Apache Flink. *Apache Flink REST API Documentation*. https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/ops/rest_api/. Accessed on May 22, 2023. 2023.

[4] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. "Efficient exact set-similarity joins". In: *Proceedings of the 32nd international conference on Very large data bases*. 2006, pp. 918–929.

[5] Brian Babcock et al. "Models and issues in data stream systems". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2002, pp. 1–16.

[6] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. "Scaling up all pairs similarity search". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 131–140.

[7] Christian Bohm and H-P Kriegel. "A cost model and index architecture for the similarity join". In: *Proceedings 17th International Conference on Data Engineering*. IEEE. 2001, pp. 411–420.

[8] Christian Böhm et al. "Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data". In: *ACM SIGMOD Record* 30.2 (2001), pp. 379–388.

[9] Brent Bryan, Frederick Eberhardt, and Christos Faloutsos. "Compact Similarity Joins". In: *2008 IEEE 24th International Conference on Data Engineering*. 2008, pp. 346–355. DOI: 10.1109/ICDE.2008.4497443.

[10] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444.

[11] Paris Carbone et al. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38 (Jan. 2015).

[12] Shiyu Chang et al. "Streaming recommender systems". In: *Proceedings of the 26th international conference on world wide web*. 2017, pp. 381–389.

[13] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. "A primitive operator for similarity joins in data cleaning". In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 5–5.

[14] Common Crawl. *Common Crawl News Dataset*. https://data.commoncrawl.org/crawl-data/CC-NEWS/index.html. Accessed on May 20, 2023. 2023.

[15] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. "Clusterjoin: A similarity joins framework using map-reduce". In: *Proceedings of the VLDB Endowment* 7.12 (2014), pp. 1059–1070.

[16] Dong Deng et al. "An efficient partition based method for exact set similarity joins". In: *Proceedings of the VLDB Endowment* 9.4 (2015), pp. 360–371.

[17] Dong Deng et al. "Massjoin: A mapreduce-based method for scalable string similarity joins". In: *2014 IEEE 30th International Conference on Data Engineering*. IEEE. 2014, pp. 340–351.

[18] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. "Similarity join in metric spaces using ed-index". In: *Database and Expert Systems Applications: 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003. Proceedings 14*. Springer. 2003, pp. 484–493.

[19] Fabian Fier et al. "Set similarity joins on mapreduce: An experimental survey". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1110–1122.

[20] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. "Similarity search in high dimensions via hashing". In: *Vldb*. Vol. 99. 6. 1999, pp. 518–529.

[21] Google. *Google Cloud*. https://cloud.google.com/. Accessed on July 31, 2023.

[22] Edwin H Jacox and Hanan Samet. "Metric space similarity joins". In: *ACM Transactions on Database Systems (TODS)* 33.2 (2008), pp. 1–38.

[23] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. 2011.

[24] Library of Congress. *Library of Congress Format Description for FDD000236*. https://www.loc.gov/preservation/digital/formats/fdd/fdd000236.shtml. Accessed on May 20, 2023. 2023.

[25] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. "A fast similarity join algorithm using graphics processing units". In: *2008 IEEE 24th international conference on data engineering*. IEEE. 2008, pp. 1111–1120.

[26] Marchetti, M. M. *Tweets Dataset*. https://www.kaggle.com/datasets/mmmarchetti/tweets-dataset. Accessed on May 20, 2023. 2023.

[27] *Oxford 3000-5000 Wordlist*. https://www.oxfordlearnersdictionaries.com/wordlists/oxford3000-5000. Accessed on May 20, 2023. 2023.

[28] Yuan Qiu, Serafeim Papadias, and Ke Yi. "Streaming HyperCube: A Massively Parallel Stream Join Algorithm." In: *EDBT*. 2019, pp. 642–645.

[29] Radhya Sahal, John G Breslin, and Muhammad Intizar Ali. "Big data and stream processing platforms for Industry 4.0 requirements mapping for a predictive maintenance use case". In: *Journal of manufacturing systems* 54 (2020), pp. 138–151.

[30] Zeyuan Shang et al. "K-join: Knowledge-aware similarity join". In: *IEEE Transactions on Knowledge and Data Engineering* 28.12 (2016), pp. 3293–3308.

[31] Liwen Sun et al. "On link-based similarity join". In: *Proceedings of the VLDB Endowment* (2011).

[32] Ankit Toshniwal et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 147–156.

[33] Rares Vernica, Michael J Carey, and Chen Li. "Efficient parallel set-similarity joins using mapreduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 495–506.

[34] C Xiao et al. "Efficient similarity joins for near duplicate detection. In WWW, pages 131-140". In: (2008).

[35] Chuan Xiao, Wei Wang, and Xuemin Lin. “Ed-join: an efficient algorithm for similarity joins with edit distance constraints”. In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 933–944.

[36] Chuan Xiao et al. “Top-k Set Similarity Joins”. In: *2009 IEEE 25th International Conference on Data Engineering*. 2009, pp. 916–927. DOI: 10.1109/ICDE.2009.111.

[37] Chengcheng Yang et al. “Dynamic Set Similarity Join: An Update Log based Approach”. In: *IEEE Transactions on Knowledge and Data Engineering* (2021).

[38] Jianye Yang et al. “Distributed streaming set similarity join”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 565–576.

[39] Yelp. *Yelp Dataset*. https://www.yelp.com/dataset. Accessed on May 20, 2023. 2023.

[40] Xiangmin Zhou and Lei Chen. “Event detection over twitter social media streams”. In: *The VLDB journal* 23.3 (2014), pp. 381–400.

[41] Lei Zhu et al. “SVS-JOIN: efficient spatial visual similarity join for geo-multimedia”. In: *IEEE Access* 7 (2019), pp. 158389–158408.