Adaptive Distributed Streaming Similarity Joins

Siachamis, G.; Psarakis, K.; Fragkoulis, M.; Papapetrou, Odysseas; van Deursen, A.; Katsifodimos, A

Publication date
2023

Document Version
Final published version

Published in
DEBS '23: Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems

Citation (APA)
Siachamis, G., Psarakis, K., Fragkoulis, M., Papapetrou, O., van Deursen, A., & Katsifodimos, A. (2023). Adaptive Distributed Streaming Similarity Joins. In M. Pasin (Ed.), *DEBS '23: Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems* (pp. 25-36)
http://10.1145/3583678.3596891

Adaptive Distributed Streaming Similarity Joins

George Siachamis[△] Kyriakos Psarakis[△] Marios Fragkoulis[°] Odysseas Papapetrou[□]
Arie van Deursen[△] Asterios Katsifodimos[△]
[△]Delft University of Technology [□]Eindhoven University of Technology [°]Delivery Hero SE
{initial.surname}@tudelft.nl,o.papapetrou@tue.nl,marios.fragkoulis@deliveryhero.com

ABSTRACT

How can we perform similarity joins of multi-dimensional streams in a distributed fashion, achieving low latency? Can we adaptively repartition those streams in order to retain high performance under concept drifts? Current approaches to similarity joins are either restricted to single-node deployments or focus on set-similarity joins, failing to cover the ubiquitous case of metric-space similarity joins. In this paper, we propose the first adaptive distributed streaming similarity join approach that gracefully scales with variable velocity and distribution of multi-dimensional data streams. Our approach can adaptively rebalance the load of nodes in the case of concept drifts, allowing for similarity computations in the general metric space. We implement our approach on top of Apache Flink and evaluate its data partitioning and load balancing schemes on a set of synthetic datasets in terms of latency, comparisons ratio, and data duplication ratio.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

data streams, similarity joins, data partitioning, load balancing, distributed computations

ACM Reference Format:
George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Odysseas Papapetrou, Arie van Deursen, Asterios Katsifodimos. 2023. Adaptive Distributed Streaming Similarity Joins. In *The 17th ACM International Conference on Distributed and Event-based Systems (DEBS '23), June 27–30, 2023, Neuchatel, Switzerland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3583678.3596891

1 INTRODUCTION

Similarity join is the task of identifying all pairs of similar records that reside in two or more datasets according to a similarity function. Similarity joins play an important role in *data integration, data cleaning, recommender systems* and many other domains. In fact, nowadays, with data in motion becoming more ubiquitous, many of the use cases requiring a similarity join have to be performed in a streaming fashion.

Performing similarity joins on data streams is challenging and computationally expensive. The brute force approach – even for the case of static datasets – has to compare all records of the first dataset against all records on the second, leading to quadratic time complexity, $O(n^2)$ where n is the number of records. As the number of records increases, brute-force solutions become infeasible. At the same time, the unbounded nature of data streams means that the complete set of records is not available in full prior to their processing. Moreover, since data streams are continuous, their statistical properties may change over time. Depending on how frequently those changes occur (also known as concept drift [13, 33]), they can have a significant impact in optimizing the similarity join operation. To recap, streaming similarity joins entail solutions that are *i*) efficient, *ii*) scalable, and *iii*) can adapt to concept drift.

Although equality joins on streams have been studied extensively, scant attention has been paid to streaming similarity joins. Existing works provide solutions specifically for set-similarity joins [38], optimizing a self-join operation in a single machine [8], and scaling out the cross-product comparisons in multiple machines [11]. However, none of these solutions target the general metric space or support efficient load balancing that can adapt the load distribution to the concept drift that frequently occurs (Table 1). At the same time, a significant research body targets batch similarity joins targeting the MapReduce [5, 7, 36] framework. Although these works address scalability and efficiency, their core techniques do not adhere to the properties of streaming data (unboundedness and concept drift). Thus, they cannot be applied on streams, and they do not provide load-balancing capabilities.

Streaming similarity joins can be performed in either exact or approximate fashion. Approximate similarity join algorithms, such as [17, 18] are interesting for use cases where applications can sacrifice completeness of results. In this work, we explicitly focus on exact algorithms, which are necessary for scenarios where complete answers are required. All available similarity join approaches share a common strategy: they group similar data to reduce the number of unnecessary comparisons. To this end, they either employ optimized indices [8] in a centralized setting or data partitioning schemes [7, 36, 38] in a distributed setting. We have already argued about the necessity of a distributed approach in order to handle the massive volumes of modern data streams. However, the distributed batch-based solutions require multiple passes over the data [7, 36], and they only provide means of splitting huge partitions into smaller ones rather than balancing the load. Additionally, the only distributed streaming similarity joins approach [38] targets only the specific sub-problem of set-similarity joins instead of providing an applicable solution for the general problem. The simple load-balancing scheme which the authors propose has limited scaling capabilities and can only provide balancing for scenarios where the input has varying lengths. All in all, existing solutions fail to provide efficient load balancing that can keep up with concept drift and harness the processing available processing power.

DEBS '23, June 27–30, 2023, Neuchatel, Switzerland

ACM ISBN 979-8-4007-0122-1/23/06.
https://doi.org/10.1145/3583678.3596891

Table 1: Related work comparison.

	Method Characteristics		*Applicability*		
Work/Feature	**Candidate Pruning**	**Load Balancing**	**Mode**	**Environment**	**Problem**
Morales et al.[8]	X		Streaming	Centralized	Similarity self-joins on a vector stream
Yang et al. [38]	X	X	Streaming	Distributed	Set similarity join on sets of various lengths
ClusterJoin [7]	X	X	Batch	Distributed	Similarity joins on general metric space
ElSeidy et al. [11]		X	Streaming	Distributed	Cross-product joins to multiple machines
Proposed Solution	X	X	Streaming	Distributed	Similarity joins on general metric space

In this paper, we propose S^{3}J, an approach for adaptive distributed streaming similarity joins[1] based on a load balancing scheme that leverages the underlying partitioning to redistribute the load across our group of workers. We pair our balancing scheme with a partitioning scheme that adheres to the properties required to facilitate it. The employed partitioning scheme is suitable for the general flavor of the similarity joins problem over a metric space and extends existing work by leveraging two levels of data partitioning to prune candidate pairs and scale out the similarity computations to multiple nodes.

Our contributions can be summarized as follows:

- The proposed solution is the first algorithm that solves the distributed streaming similarity join problem in the general metric space (see Table 1). Our algorithm is also the first to properly address the issue of load imbalance, thereby permitting better scalability and responsiveness.
- We propose a stream partitioning scheme for similarity joins in the general metric space, providing tight and fine-grained partitions, ensuring the completeness of results while reducing the number of computations. Our partitioning scheme has all required properties to effectively support our load-balancing scheme (Section 7).
- We show how to map the load balancing problem to the classic job rescheduling problem and propose a novel algorithm tailored to a partitioning/work imbalance measure (Section 8) .
- We propose a load balancing scheme to alleviate heavily loaded nodes while minimizing migration costs. In contrast to existing load-balancing solutions, we refrain from repartitioning the data, which is prohibitively expensive in streaming scenarios, and instead, we exploit the existing partitions to perform load balancing (Section 8).
- We conduct a detailed experimental evaluation of our solution using synthetic datasets in order to evaluate the efficiency of our method under various scenarios (Section 9).

2 PRELIMINARIES

This section provides a discussion of existing concepts and techniques on the foundations of similarity joins and our partitioning scheme.

The Inner-Outer Partitioning Paradigm. The state-of-the-art MapReduce solutions for the *general metric space* [7, 36] define the inner-outer partitioning paradigm. Specifically, for each worker, one centroid is randomly selected, and a pair P of inner and outer partitions is assigned to it. Inner partitions are disjoint, i.e., they have no common records, while *outer partitions can overlap*. We provide these definitions below.

Definition 1 (Inner Partition). The inner partition I_i of centroid c_i contains all records for which centroid c_i is the closest centroid among all available centroids, i.e., $I_i = \{r \mid \forall c_j \in C, dist(r, c_i) \leq dist(r, c_j)\}$, where $dist()$ is the employed distance metric.

To decide whether a record is included in an outer partition, ClusterJoin [7] proposes the following membership criterion:

Criterion 1 (Outer Partition Membership Criterion). *Let r be an incoming record and c_i be the closest centroid to r. Then r belongs to the outer partition of $c_j, \forall c_j \in C, c_j \neq c_i$, if and only if*

$$dist(r, c_j) \leq dist(r, c_i) + 2 \times t, \tag{1}$$

where dist() is the employed distance metric and t is the provided distance threshold.

Based on Criterion 1, Definition 2 describes an outer partition.

Definition 2 (Outer Partition). The outer partition O_i of centroid c_i contains all records that do not belong to the inner partition I_i and satisfy Criterion 1, i.e., $O_i = \{r \mid \forall r \notin I_i \wedge Criterion1(r, c_i)\}$.

Similarity Computations. A solution based on the inner-outer partitioning paradigm proceeds in performing the similarity comparisons based on the formed pairs of inner and outer partitions (Figure 2(a)). In short, the records of an inner partition are compared against the records of the same inner partition as well as the records of the corresponding outer partition, whereas records from an outer partition are compared only against records from the corresponding inner partition. All records with a similarity higher than a threshold are returned to the user. It can be easily shown that this algorithm retrieves all results [7, 36].

In this work, we extend the inner-outer partitioning paradigm to encapsulate fine-grained sets of data involved in computations as a group. This formulation enables us to perform load balancing of distributed streaming similarity joins with high adaptation to variable data velocity and distribution.

3 PROBLEM STATEMENT

A streaming similarity join operation identifies all pairs of records that belong to one or more data streams, arrive in the same time window, and have a similarity that exceeds a user-defined threshold. Consider a set of streams $\mathcal{S} = \{S_1, S_2, \ldots\}$, each containing records. Let $sim(r_i, r_j)$ denote the user-defined similarity function between two records r_i and r_j, which takes values within $[0, 1]$. [2]

[1] Code available in: https://github.com/delftdata/s3j-adaptive-similarity-joins

[2] To simplify exposition, hereafter, we expect that $sim(r_i, r_j)$ returns values between 0 (no similarity) and 1 (identical records). If this is not the case, we can define a metric-specific function $f(sim(r_i, r_j))$ that bounds the similarity metric within $[0, 1]$.

Definition 3 (Matching records within a time window). For a given similarity threshold θ_{sim}, two records are considered a match when their similarity exceeds θ_{sim}, and they both arrive within the same time window. Formally:

$$(r_i, r_j) \text{ is a match} \Leftrightarrow sim(r_i, r_j) \geq \theta_{sim} \; and \; t_i, t_j \in W_k, \quad (2)$$

where t_i, t_j are the ingestion timestamps of r_i, r_j and W_k the k_{th} window. [3]

The above definition can trivially be rewritten using distances, where $dist(r_i, r_j) = 1 - sim(r_i, r_j)$, and $\theta_{dist} = 1 - \theta_{sim}$.

Notice that similarity join (even over static data) is often a computationally expensive operation whose complexity increases with the increase of the dimensionality of the data. The streaming context further aggravates this issue due to the high velocity of incoming records in data streams and the need for quick answers. The only way to efficiently sustain the overall computational burden is by scaling out the processing to a distributed stream processing system. This practice, however, brings forward a new set of challenges, most importantly the partitioning of the data and the load balancing across the cluster.

Definition 4 (Partitioning for streaming similarity joins). Assume a set of streams $\mathcal{S} = \{S_1, S_2, \ldots\}$ that contain records consisting of a timestamp, an id, and a (potentially high-dimensional) value, i.e., $r = (timestamp, id, value)$. Consider a set of records $R_k = \{r \mid r \in S_i, S_i \in \mathcal{S}\}$, ingested within a time window W_k, a set of worker nodes N, and a given threshold θ. Partition the records in R_k in $|N|$ partitions, such that i) each pair of matched records based on the similarity threshold θ is contained in the same partition and ii) the computation load across the worker nodes is balanced.

Unfortunately, the partitioning of the records is not guaranteed to be stable in the lifetime of a streaming workload. Even if we could efficiently decide on an optimal partitioning of the data, this remains unknown since the data is not available when creating the partitions. Furthermore, statistical changes in the incoming streaming data, i.e., a possible concept drift, create skews in data partitions. Consequently, these aspects entail the consideration of computational load and accompanying challenges. We define the load of a worker node in a set of workers as follows.

Definition 5 (Load). Given a partitioning scheme PS, the load L_n^{ps} of a worker n in a set of workers N is equivalent to the number of similarity comparisons it needs to perform based on the partition of records assigned to it.

As new data arrive, the existing partitioning of the data in the cluster's workers may no longer provide a balanced workload across the cluster, thereby leading to the load balancing problem.

Definition 6 (Load balancing). Assume a set $R_k = \{r | r \in S_i, S_i \in \mathcal{S}, and \; r.timestamp \in W_k\}$ of streaming records ingested within a time window W_k, a set of the previously defined partitions PS that contain the streaming records R_k, and a set of workers N. Each worker n is assigned a partition P and has an estimated load L_n^{ps} based on partition P. Find a new optimal partitioning scheme $OPS(R, N)$ that minimizes how much the load of each worker differs from the desired average load, i.e.,

$$OPS(R, N) \text{ so that } min(\sum \mid L_n^{ops} - L_{avg}^{ops} \mid), \forall n \in N. \quad (3)$$

[3] Hereafter, without loss of generality, we refer to tumbling windows only to simplify the presentation of our approach with respect to time window semantics. Our work is applicable to any type of time window.

4 RELATED WORK

In this work, we discuss the works that are most relevant to the problem of adaptive distributed similarity joins.

Distributed Stream Equi-Joins. Equality joins have been investigated thoroughly in the literature. Najafi et al. [25] propose SplitJoin, a novel stream join architecture that achieves high scalability by dividing the join operation into independent storing and processing, and employing adjustable join output ordering guarantees. Dossinger et al. [10] introduce MultiStream, a novel multi-way stream join operator that leverages tuple routing and exploits a materialization vs. network cost to perform stream join optimization. Najafi et al. [26] propose a circular multi-way join operator that benefits from hardware and a parallel multi-way join operator that reduces computation time.

None of these works deal with similarity joins.

Single-node Similarity Joins on Streams. Contrary to equality joins, research on similarity joins on data streams is limited. Morales et al. [8] propose a solution to streaming similarity self-joins. However, this work does not discuss a parallel distributed solution, and their approach cannot be trivially scaled out. Similarly, [21] introduces an operator to tackle similarity joins on uncertain data streams, but their solution cannot be trivially scaled out. In addition, none of these works considers load (re-)balancing, which is necessary to retain high performance in the case of concept drift.

To the best of our knowledge, no existing work in stream processing can provide a general solution over the metric space, which can scale out to multiple machines and provide load balancing capabilities to tackle concept drift.

Distributed Similarity Joins on Streams. Closest to the spirit of our work is the work by Yang et al. [38]. The authors propose a distributed streaming similarity join framework that employs a simple length-based filter to distribute the data across the cluster. Such a filter cannot be used for metric-space similarity computations, which we consider in this work. Instead, [38] focuses on the *set similarity* problem.

In contrast to [38], in this paper, we aim for metric-space similarity computations, which are required in state-of-the-art approaches in similarity joins, like keyless joins [32].

k-Nearest Neighbours on Streams . In contrast to the exact similarity join problem on streams, streaming kNN queries attempt to identify and retrieve a specific number, k, of results. Koudas et al. [20] introduce an error-bounded variation of the problem and propose DISC that can answer error-bounded kNN queries over sliding windows. Sundaram et al. [31] propose PLSH, a fast distributed variation of LSH, that supports approximate nearest neighbours queries over high throughput streams. ADS-kNN [29] overlaps communication and computation stages and leverages an adaptive partitioning that keeps the load balanced in order to improve performance.

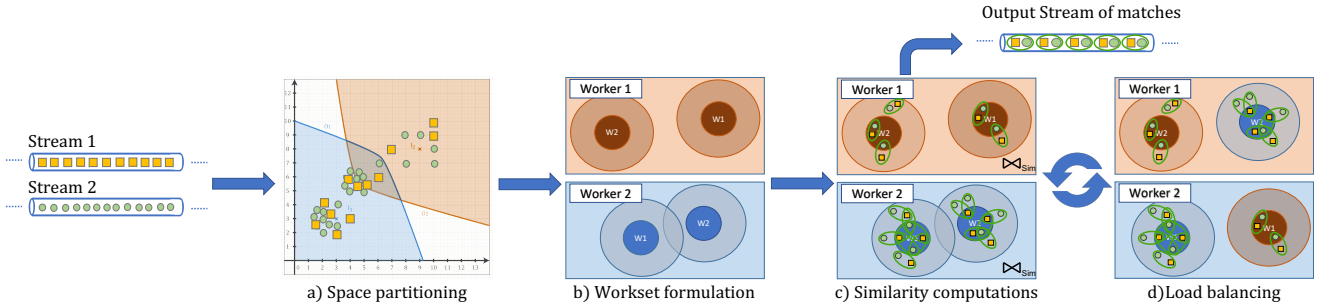

Figure 1: Overview of proposed solution's workflow

Streaming kNN employs similarity computations and requires efficient real-time results. However, it does not retrieve all existing pairs, and the results are not threshold bounded. Therefore, its solutions cannot be applied in our context.

Distributed, Batch Similarity Joins in MapReduce. There are two main approaches that MapReduce methods usually follow: Filter & Verification and General Metric Space [12]. The Filter & Verification methods [9, 24, 34] rely on prefixes and signatures, which they leverage to scale out the similarity computations and filter unnecessary comparisons. On the other hand, General Metric space methods [5, 7, 36, 37] divide the metric space into partitions to which similar objects are grouped.

None of the MapReduce solutions is applicable to streaming similarity joins, as they require multiple passes over the given dataset to gather statistics, as well as additional pre-processing steps.

Dynamic reconfiguration for stream processing. Load balancing is a native concern in distributed stream processing environments. Zhou et al. [40] formalize the problem of operator placement and propose heuristics that provide load balancing with minimum data movement across nodes that execute multiple queries. Pietzuch et al. [27] propose an intermediate layer between the physical network and the stream processing engine that provides load optimizations through operator placement. Madsen et al. [23] propose ALBIC, a stream processing optimizer, that unifies reconfiguration problems, such as load balancing and operation placement, and addresses it as a mixed integer linear program optimization. Finally, Cardellini et al. [3] propose a two-layered hierarchical architecture, EDF, that enhances a stream processing engine with autoscaling capabilities.

These works address reconfiguration problems over a cluster of nodes where multiple streaming queries run. However, they only focus on cluster level reconfigurations and do not deal with imbalanced parallel operators of a specific query due to skewed workloads.

Load-balancing for joining streams. A significant mass of work focuses specifically on join operations. Both [11] and [16] propose new dataflow multi-way join operators. Gu et al. [16] employs two routing algorithms that achieve load balancing without affecting the completeness of the results through data replication. On the other hand, ElSeidy et al. [11] focus on providing minimal state relocation costs while keeping at balance the trade-off between migration costs and the costs of not having optimal data distribution. Using a different architecture, [35] describes a ring model of multi-way window-based join operators that is based on time slicing and record propagation. Qiu et al. [28] introduces a streaming variation of the HyperCube algorithm [1] for static multi-way joins. BiStream [22] leverages a new model based on managing the computational cluster as a bipartite graph to scale out or down depending on the current workload.

In summary, these works do not optimize unnecessary comparisons and load balancing for the special case of similarity joins.

5 APPROACH OVERVIEW

In this paper, we propose S^{3}J, an adaptive method that enables efficient similarity joins over streams in a distributed share-nothing environment through a novel partition-aware load balancing paired with a stream partitioning scheme that can tackle the general metric space streaming similarity join problem. Figure 1 presents the workflow of our proposed approach, assuming two input streams. The *similarity join* pipeline employs a workflow of four key operators, executed in a loop (also depicted in Figure 1):

(a) *space partitioning* (Section 6), where the ingested data (the yellow squares and the green circles, corresponding, in this example, to two streams) is partitioned to two partitions (the highlighted blue and highlighted orange region), each assigned to one worker;
(b) *workset formulation* (Section 7), which divides the previously created local partitioning of the data at each worker into smaller sub-partitions (the worksets), to facilitate a more efficient load balancing;
(c) *similarity computation* (Section 7.7), where the workers leverage the formed sub-partitions to independently compute the results, without further coordination, and, finally;
(d) *load balancing* (Section 8), which relies on statistics collected from the previous steps to re-partition the data by reassigning some worksets to different workers, in order to reduce load imbalance at the workers.

The key statistics driving the load balancing policy include the join output and the side outputs from the workset formulation operators. Particularly, when load imbalance exceeds a threshold configured by the application, the load balancing scheme computes a new distribution of the existing worksets based on the existing distribution and a migration minimization policy and employs the new distribution to the available workers without the need to create new partitions or worksets. For example, in Figure 1, the distribution of the load is imbalanced between workers 1 and 2. Thus, the load balancing strategy exchanges workset W2 from worker 1 with

workset W1 from worker 2 to balance their load. The new distribution of the worksets is communicated to the workset formulation operators to ensure the correct routing of future records.

6 SPACE PARTITIONING

Our partitioning scheme aims to distribute the data among the available workers so that: (a) the computational load is evenly distributed across the workers, and (b) data duplication is reduced. In this work, we focus on providing adaptivity to streaming similarity joins through an efficient load-balancing scheme and a fine-grained partitioning scheme. Therefore, we adapt and extend the previously-proposed inner-outer partitioning paradigm, described in Section 2. In particular, we introduce two layers of partitioning: (a) partitioning the data into coarse partitions, and (b) breaking each partition into smaller worksets, which can be independently handled by each worker. In this section, we describe shortly our first partitioning layer, space partitioning.

6.1 Space partitions

To create our fine-grained worksets, we need to distribute our incoming records to our workers. However, we cannot divide the load arbitrarily. We need to ensure that possibly similar records are co-located to the same worker so that we will not have two workers creating similar worksets. In such a case, we would risk losing matches or duplicating every record belonging to these worksets. Therefore, we opt to divide the incoming records into inner and outer partitions, adapting the inner-outer paradigm, in order to create *space partitions*. Formally, the space partition SP_i of a centroid c_i is the pair of inner partition I_i and outer partition O_i of c_i.

6.2 Selecting Centroids

Selecting the right centroids for the space partitions is not a trivial task. A common approach employs a random sampling of the data under processing to select partition centroids [7, 36]. However, none of these techniques is applicable in the case of streams, as they all require an extra pass over the data. In addition, historical streaming data could serve as a source of candidate centroids, but those would be obsolete in the case of concept drift. In this paper, we opt for a simple approach: we randomly generate our partition centroids based on the expected space coverage. More specifically, after randomly selecting a set of partition centroids, we initialize our space partitioning instances by providing each one a copy of the available centroids. Each centroid is assigned to a downstream worker of our set of workers. For each incoming record, our partitioner instances calculate the distances to the centroids. Based on the provided Definitions 1 & 2, an incoming record is assigned to the inner partition of the closest centroid and to the outer partitions of the centroids that satisfy the Criterion 1.

6.3 Avoiding duplicate comparisons

According to the inner-outer partitioning paradigm, records that belong to neighbouring inner partitions can potentially be members of the corresponding outer partitions. Depending on how similarity comparisons are resolved, such a case would lead to the same candidate pair of records being evaluated twice, i.e., potentially by two different nodes. Figure 2(b) shows such an example. Record r_1 belongs to the inner partition I_1 and to the outer partition O_2, while record r_2

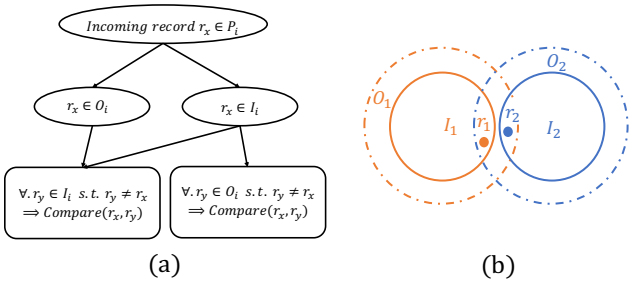

Figure 2: (a) Paradigm's similarity computations workflow. (b) Example with a duplicate evaluation of a candidate pair.

belongs to the inner partition I_2 and to the outer partition O_1. Therefore, the pair (r_1, r_2) is evaluated for both pair of partitions P_1 and P_2 ($P_1 : compare(r_1 \in I_1, r_2 \in O_1), P_2 : compare(r_1 \in O_2, r_2 \in I_2)$). To avoid these redundant comparisons and maintain duplicate-free results, we adopt the same routing criterion employed by ClusterJoin[7] and MR-MAPSS[36] to decide whether a record should be included in a neighbouring outer partition of a space partition or not.

7 WORKSET FORMULATION

After dividing the incoming records based on their position in the input space, the workset formulation operation takes place in each of the responsible workers. In this step, we attempt to create self-contained, minimal worksets on top of which we will perform all our similarity comparisons. We define a workset as follows:

Definition 7. The i^{th} workset $W_{j,i}$ of a space partition P_j has a centroid $c_{j,i}$ assigned to it, and it consists of an inner set $IS_{j,i}$, an outer set $OS_{j,i}$, and a set of outliers $Outliers_{j,i}$.

Similarly to the inner and outer partitions of the previous stage, inner sets are disjoint while outer sets can overlap with outer sets of other worksets, and they can contain records from inner sets or outliers' sets of multiple other worksets. The outliers' sets are also disjoint. In the following, we discuss the workflow of the workset formulation operator. We present how an incoming record is handled, and we provide all necessary definitions. We explain how we select new centroids and the concept of outliers.

7.1 Step 1: Deciding Inner vs. Outer Partition

For each incoming record received from workset formulation operator, we first compute the distances from all existing workset centroids in order to use them in the following steps. Then we need to specify if the record belongs to the inner or the outer partition of the space partition worker is responsible for. Records that belong to the outer partition can only participate in the outer sets of our worksets, while inner records must also be assigned to an inner set of a workset. If the received record is an outer record, we can move directly to step 5 (Section 7.5).

7.2 Step 2: Assign to an Inner Set

For each incoming record that belongs to the inner partition, we first need to identify the workset whose inner set will contain it. Each record is assigned to at most one inner set. We decide whether a record should be assigned to the inner set of a workset based on the following definition of inner sets.

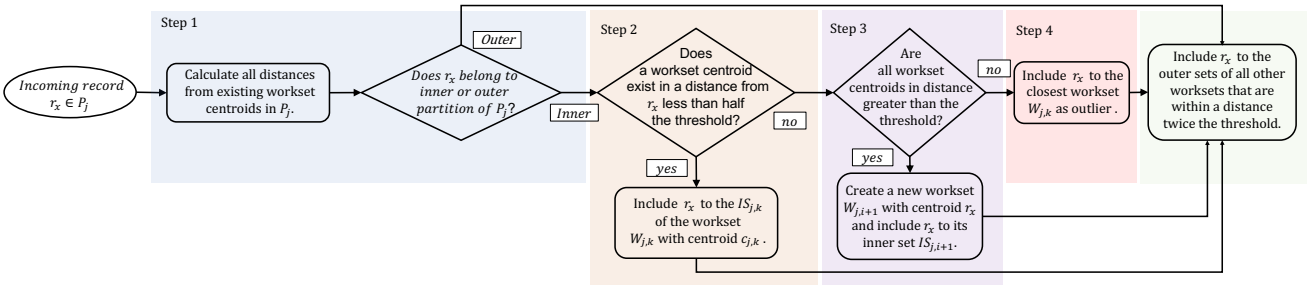

Figure 3: Workset Formulation Workflow

Definition 8 (Inner Set). The inner set $IS_{j,i}$ of the workset centroid $c_{j,i}$ contains all records which are in a distance less than half the provided threshold from $c_{j,i}$, i.e.,

$$IS_{j,i} = \{x | dist(x, c_{j,i}) \leq t/2\}, \tag{4}$$

where $dist()$ is the employed distance metric, and t is the provided threshold.

If we manage to assign the incoming record to an existing workset, we can move to step 5. Otherwise, if the incoming record cannot be assigned to the inner set of any of the existing worksets, we check in step 3 if we can create a new workset to assign it to.

7.3 Step 3: Creating New Worksets

In order to create a new workset, we first need to select an appropriate centroid for it.

Selecting workset centroids. To select the workset centroids, our strategy differs significantly from the random selection of a fixed number of centroids employed in the space partitioning stage. Without knowing the exact distribution of incoming records in the future, it is very difficult to cover all input space with a fixed number of centroids. Therefore, we opt to select centroids on the fly as we process the data. In this way, we can naturally adapt to occurring concept drifts.

In more details, only records that belong to the inner partition of each space partition can be selected as workset centroids. If an incoming record s cannot be assigned to the inner set of any of the existing worksets, we check if we can create a new workset with s as its centroid. Since the inner sets of all worksets must be disjoint, s must be at a greater distance than the provided threshold from any other centroid. Otherwise, there would be at least two worksets whose inner sets would have some overlap. Therefore, to select an incoming record s as a new centroid, it must satisfy the following criterion:

Criterion 2 (Workset Centroid Selection). *An incoming record s is selected as a workset centroid if and only if it is in distance greater than the provided threshold from all existing centroids, i.e., s is a centroid $\iff dist(s, c_i) > t, \forall c_i \in C$, where $dist()$ is the employed distance metric, t is the provided threshold, and C is the set of existing workset centroids.*

7.4 Step 4: Labeling Outliers

If an incoming record s neither belongs to an existing group nor can be selected as a centroid itself, it is labeled as an outlier.

Definition 9 (Outliers). We define as outliers *Outliers* the set of incoming records that are within greater than half the provided threshold but less than a threshold distance from all existing centroids, i.e.,

$$Outliers = \{o \,|\, t/2 < dist(o, c_i) \leq t, \forall c_i \in C\}, \tag{5}$$

where $dist()$ is the employed distance metric, t is the provided threshold, and C is the set of existing group centroids.

We assign each outlier to an existing workset based on a proximity criterion, i.e., each outlier is assigned to the outlier set of the workset whose centroid is the closest to it.

7.5 Step 5: Assign to Outer Sets

All incoming records are assigned to none, one or more outer sets of existing worksets based on the provided similarity threshold. More specifically, we decide if an existing record should be included to an outer set based on the following definition.

Definition 10 (Outer Set). The outer set $OS_{i,j}$ of centroid $c_{i,j}$ contains all records which are within a distance greater than half the provided threshold and less than twice the provided threshold, i.e.,

$$OS_{j,i} = \{x | t/2 < dist(x, c_{j,i}) \leq 2 \times t\} \tag{6}$$

where $dist()$ is the employed distance metric, and t is the provided threshold.

To ensure the completeness of our final output, all outer partition records are stored to be compared to new occurring centroids when a new workset is created.

Routing criterion. Similarly to the space partitioning stage, in workset formulation we also employ a routing criterion to decide on routing records to the outer sets of worksets. The reasoning behind this decision is again to avoid considering the same pairs of records in two different worksets. However, due to the dynamic way of creating our worksets we cannot employ the same criterion as in the space partitioning stage. That is because the aforementioned criterion considers all worksets known a priori. Therefore we opt for the simplest solution of routing records to the outer sets of a workset only if the id of the workset is smaller than the id of the workset whose inner set contains the record. Of course, as also discussed in [36], this routing scheme results in a skewed distribution of the computation load to the worksets with the smaller ids. We try to compensate for our decision through our load balancing scheme discussed later on.

Although the worksets are created within a space partition that is assigned to a specific worker, they can be distributed to any of the available workers for the downstream operation of the similarity computations. As a policy, we assign every newly created workset

to the same worker that it was created, in order to avoid shuffling overhead. However, the ability to later re-assign a workset to any available worker is crucial for our load balancing scheme discussed later.

7.6 Set Boundaries in Metric Space

Based on the metric space properties, the bounds for the inner and outer sets of a workset are carefully chosen to ensure the correctness of the final output and, at the same time, avoid as much as possible unnecessary similarity comparisons. For our inner sets, a bound of half the provided threshold is the maximum bound which can ensure that all records participating in an inner set are at a distance of at most the desired threshold. Thus it allows us to avoid performing the actual comparisons to determine those matches. On the other hand, the selected bounds for outer sets consider the existence of outliers and ensure the desired completeness of the final output while attempting to keep the number of computations as low as possible. Both bounds can be trivially proven using the metric space's triangle inequality property.

7.7 Similarity Computations

By creating worksets inside each space partition, we confine our comparisons and minimize the number of similarity computations performed. We can decide which similarity computations to perform by considering the type of the incoming records, e.g., an inner, outer, or outlier record.

When the incoming records belong to the inner set, we can immediately emit as matches all possible pairs of our incoming records and the other records of this inner set in our state. Yet, we still need to perform the comparisons with the records in the outer set of the workset as well as with the outliers assigned to workset. In the case of an incoming record that belongs to the outer set of a workset, this record needs to be compared against all records in the inner set of the workset and the outliers' set assigned to it. The case of an incoming outlier record is the most expensive since it needs to be compared against all other records of the workset.

8 ADAPTIVE WORKSET BALANCING

In the streaming context, it is important that the algorithm adapts to the incoming streams such that it continuously maintains good load-balancing properties. In this section, we discuss our approach to adaptively load-balance the worksets at runtime. Four main factors make the load balancing for similarity joins challenging: *i*) the quadratic complexity of the similarity join problem, *ii*) low latency requirements, *iii*) the zero knowledge of data distributions before execution, and *iv*) the volatile nature of streams that can lead to concept drift.

Similarity Joins are CPU-bound. Our early experiments showed that the similarity computation is the heaviest task of our pipeline, i.e., similarity joins are CPU-bound. In the rest of this section, we propose an approach that takes into account the existing partitioning scheme and reduces the load imbalance by reassigning worksets to similarity computation operators. More specifically, the goal of the balancing of worksets is to load balance the similarity computations across a set of workers.

8.1 Migrating Worksets W/O Repartitioning

The workset formulation algorithm (Section 7) aims at forming self-contained worksets in each partition, i.e., the worksets are the unit of computation, and any given workset is sufficient to produce the similar pairs of the records assigned to those worksets. We opted for creating self-contained worksets in order to be able to move them across workers without very complex state migration procedures: intuitively, a given workset that incurs very high computation cost can be moved to a worker that is less loaded.

As a result, the worksets can be easily redistributed to the available workers to reduce load imbalance without influencing the completeness and correctness of the join results. At the same time, balancing through reassigning only specific worksets ensures that we have to migrate only specific parts of an *operator's state*. In short, by minimizing load imbalance, we minimize state migration and network costs. This allows us to achieve low response latency while adapting to streaming load spikes and stream concept drifts. In what follows, we define the problem of load balancing by redistributing worksets on the fly across the available workers.

Definition 11 (Load balancing based on worksets). Assume the set $R_{t_1} = \{r | r \in A \ or \ r \in B, \ and \ r = (timestamp, record)\}$ of streaming records received until the timestamp t_1, a set of worksets WS_{t_1} that contain the streaming records R_{t_1} and a set of workers N. Each worker is assigned a subset of WS_{t_1}, e.g. the worker n is assigned the subset $WS^n_{t_1}$. Let $L^n_{t_1}(WS^n_{t_1})$ be the workload of worker n at timestamp t_1 based on its currently assigned subset of worksets $WS^n_{t_1}$. Let $DI_{t_1} = \sum \mid L^n_{t_1}(WS^n_{t_1}) - L^{avg}_{t_1} \mid$ be the degree of imbalance regarding the distribution of workload on our set of workers N. Find a new optimal distribution of worksets to workers that minimizes the degree of imbalance DI and the migration cost to reach this optimal distribution from the current distribution.

8.2 Workset-Balancing vs. Job-Scheduling

Recall that our worksets compose our most fine-grained data partitions and, at the same time, self-contained computational units. Therefore, it is possible to think of a workset as a computation job over a certain period of time. This observation allows us to link our load-balancing problem to the classic job-scheduling problem across multiple processors. However, the classic definition of job scheduling cannot be directly applied to a streaming setting.

There is a multitude of work on job scheduling for computational grids and multi-processor settings [4, 14, 15, 30, 39]. Specifically, the *job rescheduling* flavor of the problem [2, 6] could be adapted to our workset load balancing problem. This can be achieved as follows: each workset $W_{j,i}$ can be seen as a job $J_{j,i}$ with specific load $L_{W_{j,i}}$ and migration cost based on its size in bytes $M_{W_{j,i}}$, i.e., $W_{j,i} \equiv J_i(L_{W_{j,i}}, M_{W_{j,i}})$. Every similarity computation operator can be seen as a processor, with the primary objective becoming the *minimization of the degree of imbalance*. Note that the job rescheduling problem, and thus our load balancing problem as well, is NP-hard [2, 6]. Due to its NP-hardness, all existing algorithms for the job-rescheduling problem are approximations. Similarly, in the following, we devise a greedy workset balancing algorithm.

8.3 The Workset Balancing Algorithm

Our workset balancing algorithm (listed in Algorithm 1) optimizes for the desired load imbalance measure. The algorithm takes as

input a set of *overloaded* workers, a set of *underloaded* workers, and the *average load*, i.e., the target load across all workers. The algorithm starts by going over all overloaded workers. For each of them, if it contains one or more worksets with a load higher than the average worker load, we flag the workset with the highest load as *irremovable* (line 5). Since these worksets have a greater load than the average load of workers, moving them will not alleviate the load imbalance.

For all worksets in overloaded workers that are not flagged as irremovable, we calculate the benefit of removing the workset from the worker it currently resides in. If the benefit is positive, we add the workset to a priority queue, sorted descending on benefit. After processing all worksets, we pick the workset with the maximum benefit from the priority queue that is not yet included in the ignore list (a list initialized as empty and populated during the algorithm) and flag it as the best workset (Lines 8-17). The next step is to find an underloaded worker that can accept this workset. Therefore, for each underloaded worker, we calculate the benefit of adding the chosen workset to that worker and assign it to the worker that brings the maximum benefit. If there is no worker that has a positive benefit, we append the chosen workset in an ignore list (Lines 18-30). This procedure is repeated until there is no candidate workset left to be removed from the overloaded workers (Line 32).

Migration Costs. There have been efforts to formulate a concrete migration cost model [27, 40]. In the context of streaming similarity joins, several factors affect the migration cost: a) stopping and restarting the streaming job, b) calculating the new partitioning, c) updating the existing state, and d) moving the repartitioned data to the workers based on the new partitioning. In this work, we only consider c) and d) and leave a) and b) for future work. Since our workers share the same resources, only the size of the worksets we move affects the migration cost. The benefit function calculates the benefit as the difference in the degree of imbalance (DI), defined in Definition 11, between the current workset distribution and the one occurring after a workset move. In order to include migration cost in our algorithm's model, we subtract from the calculated benefit the workset size multiplied by a user-provided factor to negatively affect the calculated benefit that will be taken into account.

Gathering Statistics. Notice that our workset balancing algorithm requires as input a collection of statistics. By monitoring the main pipeline, we measure the load and the latency for each worker, and the size and the load of the worksets assigned to each worker. All statistics are collected over an application-specified monitoring window. The algorithm requires an extra step of categorizing the workers as underloaded or overloaded based on the desired average load of a worker, which, however, takes negligible time, even for networks involving tens of thousands of workers.

Algorithm 1 Workset Balancing Algorithm

Require: set of overloaded workers O, set of underloaded workers U, average load L_{avg}
Ensure: load balanced distribution D_{new} of worksets to workers

```
1:  ignore_list ← []
2:  best ← null
3:  over_benefits ← priorityQueue(priority : benefit)
4:  under_benefits ← priorityQueue(priority : benefit)
5:  irremovables ← find_irremovables(O, L_avg)
6:  repeat
7:    for o ∈ O do
8:      for workset w ∈ W_o do
9:        benefit ← calculate_removal_benefit(w, W_o)
10:       if benefit > 0 and w ∉ irremovables then
11:         over_benefits.put({benefit, w})
12:       end if
13:     end for
14:   end for
15:   repeat
16:     best ← over_benefits.pop()
17:   until best ∉ ignore_list or best is null
18:   if best is not null then
19:     for u ∈ U do
20:       benefit ← calculate_addition_benefit(w, W_u)
21:       under_benefits.put({benefit, u})
22:     end for
23:     repeat
24:       optimal ← under_benefits.pop()
25:     until compute_load(optimal, u) < L_avg
              or optimal is null
26:     if optimal is not null then
27:       assign_workset(best, optimal)
28:     else
29:       ignore_list.append(best)
30:     end if
31:   end if
32: until best is null
```

9 EXPERIMENTS

9.1 Performance Metrics

Previous works [8, 21, 38] in streaming similarity joins try to adapt existing metrics to stream processing use cases. However, to accurately evaluate the performance of streaming solutions, we need to employ performance metrics that adhere to the requirements of streaming workloads. For example, using the total runtime duration of a streaming similarity join operation as in [8, 38], is not suitable for a practical application involving unbounded data streams. While [21] provides a more interesting per-timestamp runtime measurement of the actual similarity join computation, this does not include an end-to-end performance measurement of the pipeline.

We argue that tuple latency is a more suitable metric since it is strongly connected to the natural requirement of stream processing for real-time results. We employ as tuple latency the processing-time latency in windowed join operators from [19]. The *processing-time latency* of a joined pair of tuples in a streaming similarity join is defined as the interval between the *maximum* ingestion time of the involved tuples and its emission time from the output sink.

A common metric in the existing literature is the *duplication ratio* of data partitioning. The *duplication ratio* is defined as the *average number of times* a tuple is duplicated across the available partitions. The duplication ratio shows the impact that the partitioning scheme has on the input size. Therefore it also provides valuable insights into the additional memory and storage resources

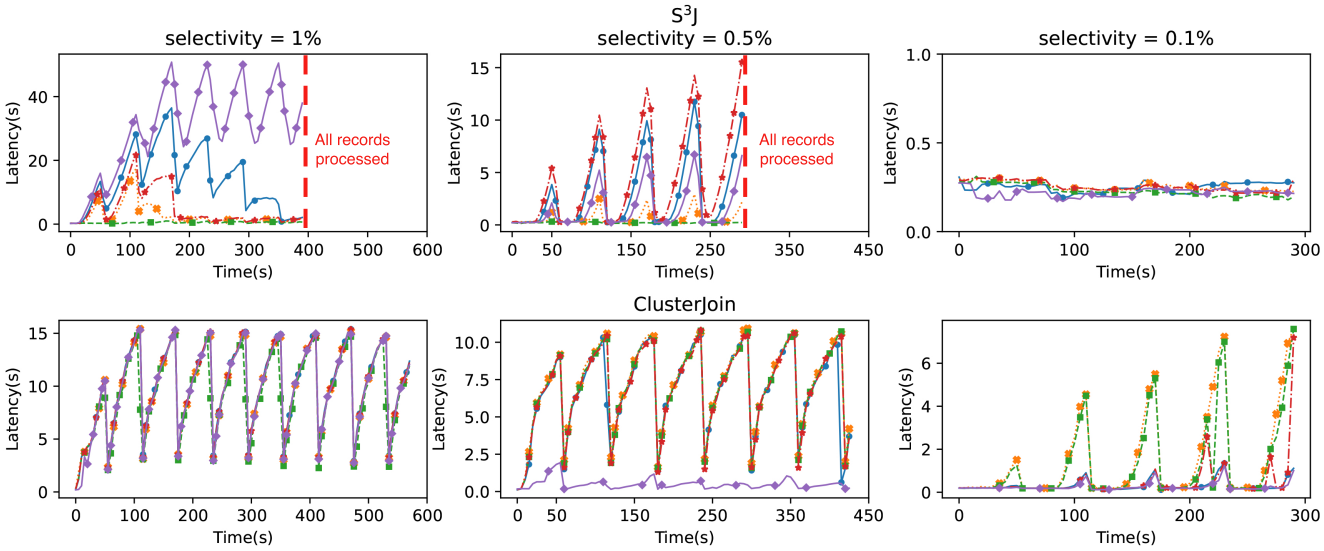

Figure 4: 99% latency percentile per worker for varying selectivities. Each line represents a single worker (in this case, 5 workers total). Incoming ratio 4000 records per sec, Parallelism: 5, Uniform distribution.

needed to apply our partitioning. The higher the duplication ratio is, the more redundant information is transmitted and stored.

Since the goal of this line of work is to reduce the number of computations performed, we also employ a *comparisons ratio*. We define the *comparisons ratio* as the *total number of performed similarity computations* over the *number of joined pairs*. This metric allows for efficiency comparisons between the solutions.

9.2 Datasets

In order to evaluate our proposed solution thoroughly and to understand its limitations, we perform an experimental evaluation over synthetic datasets of various configurations.

Synthetic stream generators. We employ synthetic data to investigate in depth the performance of both our partitioning and load balancing scheme under fully-controlled conditions. In particular, we implemented a set of configurable stream generators to provide streams of different velocities with records of varying dimensionality that follow different probability distributions.

9.3 Experimental Setup

The experiments are conducted on a 3-node Kubernetes cluster with AMD EPYC 7H12 2.60GHz CPUs. We configure an Apache Flink cluster with a single job manager and a dynamic set of task managers based on the parallelism of the running job. The job manager and the task managers are deployed with 2 CPUs and 16GB of memory each. Apache Kafka is used as the source and the sink of the Flink job. All generators feed the similarity join job through Kafka. Minio is used as a state backend for Flink and as file storage for complementary data. We employ vectors as input values and angular distance as our metric for all experiments. We evaluate our load balancing and partitioning scheme based on the aforementioned metrics. Since Flink does not provide any online mechanism for state migration, to perform our load balancing, we stop the job with a savepoint, alter the savepoint based on the new workset distribution using the state processor API, and restart the job with the new savepoint.

9.4 Baseline: ClusterJoin

There is no native stream processing solution that performs similarity joins. Therefore, we opt to adapt ClusterJoin to a streaming environment, and we include it in our experiments as a baseline. ClusterJoin follows the inner-outer-paradigms and resembles our space partitioning layer. However, it partitions the data into multiple virtual partitions, which it later assigns to workers, in contrast to our one space partition per worker design. For our experiments, we configure ClusterJoin to use 500 virtual partitions as suggested in the original work [7]. For these virtual partitions, we select centroids like we select centroids for our space partitioning layer.

9.5 Partitioning Performance

In what follows, we present S^{3}J's performance over synthetic data streams with various properties. We first investigate how our stream partitioning performs against join queries of different selectivities. Then we experiment with different levels of parallelism. For both experiments, we use synthetic data streams whose records follow a uniform distribution. In this set of experiments, we do not impose any load balancing.

9.5.1 Varying selectivities. In this first series of experiments, we focus on the correlation between the performance of our stream partitioning and the imposed query's selectivity. Although most of the existing literature targets various similarity thresholds, we opt for join selectivity since it better describes the properties of the join query.

We vary the selectivity through the similarity threshold while keeping the rest of the parameters the same. For the experiment depicted in Figure 4, we consider two streams of records of 2D values that follow the uniform distribution. Both streams have a rate of 2000 incoming records per second, which results in 4000 records per second total input rate, and a duration of five minutes. A tumbling window of 1 minute of processing time is employed, and the similarity job has a parallelism of 5. In this low parallelism setting, the partitioning scheme struggles to partition the data

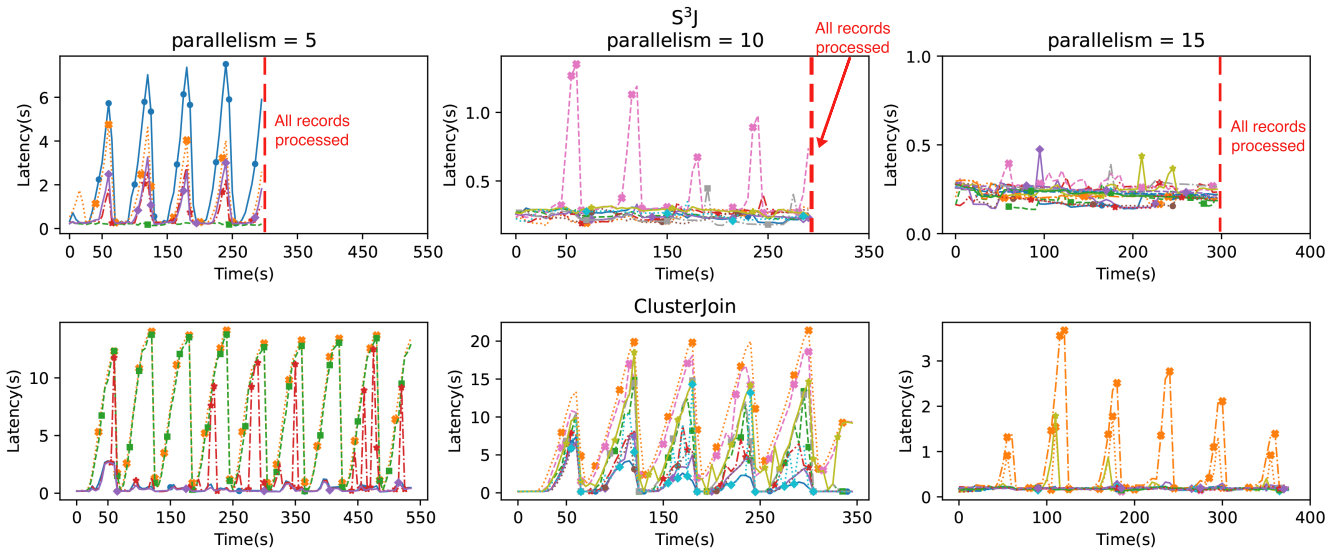

Figure 5: 99% latency percentile per worker for varying parallelism. Each line represents a single worker (in this case, 5, 10, and 15 workers accordingly). Incoming ratio 8000 records per sec, Selectivity: 0.1%, Uniform distribution.

evenly across the nodes for high-selectivity queries, while for low-selectivity queries, it provides real-time latency results.

In comparison to our baseline, ClusterJoin, we observe a significant performance improvement both in high and low selectivities. In the case of the 1% selectivity experiment, the input rate is not sustainable for either approach. However, S^{3}J outperforms ClusterJoin significantly. First of all, we manage to retain a higher processing throughput of 6000 records per second on average, while ClusterJoin is throttled to only 4000 records per second on average. As a result, we finish processing all records 200 seconds earlier than ClusterJoin. S^{3}J reaches max latency after more than 40 seconds while ClusterJoin maxes out at 15 seconds. Notice, however, that this is a side effect of using the ingestion time in order to measure latency and Flink's backpressure mechanism. Although the load is unsustainable, backpressure does not reach the source operators for S^{3}J, and the ingestion rate matches the input rate. As a result, records remain in S^{3}J's pipeline longer. On the other hand, the backpressure is much higher in ClusterJoin and reaches the source operators resulting in a drop in ingestion rate. The same also holds for the 0.5% selectivity experiment. In the experiment of low selectivity of 0.1%, in contrast to ClusterJoin, S^{3}J manages to retain a sub-second latency and provide real-time results.

For these experiments, we also measure the duplication ratio. Table 2 summarises our findings. S^{3}J manages to keep the duplication ratio around 2x for all experiments and selectivities. On the other hand, ClusterJoin's duplication ratio grows fast as the selectivity is increased. S^{3}J's fine-grained worksets manage to partition the data more efficiently than the random virtual partitions of the adapted ClusterJoin. The lower duplication ratio results in less network traffic and fewer comparisons to be performed.

The comparisons ratio showcases the efficiency of S^{3}J. As Table 2 shows, S^{3}J for high to medium selectivities manages to keep the numbers of performed similarity computations below 2x the number of joined pairs. Compared to ClusterJoin, for all selectivities S^{3}J has a better comparisons ratio by 4x to 5x.

Table 2: Effects of selectivity on duplication ratio and comparisons reduction.

	Duplication Ratio		*Comparisons ratio*	
Selectivity	**S^{3}J**	**ClusterJoin**	**S^{3}J**	**ClusterJoin**
10%	1,96	Timeout	1,58	Timeout
5%	1,96	Timeout	1,65	Timeout
1%	1,94	6,17	1,77	9,70
0.5%	1,97	3,53	1,95	11,16
0.1%	1,92	1,49	5,82	24,51

9.5.2 Varying parallelism. The parallelism of the job, i.e., the number of available workers, is another important parameter. We consider again a pair of 2D streams whose values follow the uniform distribution. However, we choose a low selectivity query and a higher incoming ratio of 4000 records per stream per second (8000 records per second in total). The streams have a duration of 5 minutes, and the processing happens in windows of 5 minutes. Figure 5 presents the effect of parallelism on the performance of our partitioning solution. The results show that our partitioning can leverage higher parallelism effectively and provide real-time latency results.

Compared to ClusterJoin, S^{3}J manages to scale much more efficiently. Even with the lowest parallelism, it manages to keep relatively low latencies and handle the entire load without a trailing lag. On the other hand, ClusterJoin needs almost double the time to process the entire load and has significantly worse latency performance. As we increase parallelism, in contrast to ClusterJoin, S^{3}J manages to harness the additional resources to achieve low latency near real-time results.

The measurement of the duplication ratio suggests that higher parallelism leads to higher record duplication for S^{3}J. Based on Table 3, the duplication ratio of S^{3}J grows slowly as we add more workers. This growth is mainly attributed to the increase in space partitions as more workers are added. Actually, our second layer of partitioning, i.e., the workset formulation, manages to even reduce

some of the duplicate records produced from the space partitioning layer. On the other hand, ClusterJoin's duplication ratio is unaffected by the increase in parallelism. This is due to the fact that ClusterJoin partitions the incoming records based on its virtual partitions and not the number of available workers.

The comparisons ratio showcases that S^{3}J also benefits from parallelism in terms of comparisons performed. As Table 3 shows, S^{3}J effectively reduces the number of performed comparisons as the parallelism increases. This behaviour is related to the number of the workset centroids assigned to each space partition. As parallelism increases, more space partitions are employed, resulting in fewer worksets per space partition and, thus, fewer unnecessary comparisons per incoming record.

Table 3: Effects of parallelism to duplication ratio and comparisons reduction.

	Duplication Ratio		*Comparisons ratio*	
Parallelism	**S^{3}J**	**ClusterJoin**	**S^{3}J**	**ClusterJoin**
5	1,96	1,49	4,48	20,36
10	2,12	1,49	3,16	20,36
15	2,37	1,49	2,93	20,36
20	2,62	1,49	2,65	20,36

9.6 Benefits of Load Balancing

Our experiments on the effectiveness of our partitioning scheme (Figures 4 & 5) showcased that the performance of S^{3}J can be improved by effectively balancing the load. In Section 8, we propose a novel approach that addresses the balancing problem as a workset balancing problem. In Figure 6, we show how this load balancing approach, on top of our partitioning scheme, can benefit the performance of the similarity join task. The experiment involves two streams following a uniform distribution with a total input rate of 8000 records per second. The selectivity is set to 0.1%, and for the simplicity of the presentation, a parallelism of 5 is selected. The processing is happening in windows of 1 minute. This configuration is similar to the experiment (top left) with a parallelism of 5 from Figure 5. We perform load balancing at the beginning of each window.

Our load balancing scheme positively affects the performance of the similarity join job. First of all, the load balancing scheme progressively manages to reduce the maximum latency within each window. Within 3 windows of processing, the load balancing scheme reduces the maximum latency from 3 seconds in the first window (0-60s) and 3.5 seconds in the second window (60-120s) to roughly 2 seconds in the last window (180-240s). At the same time, the load balancing scheme successfully involves all available workers in the processing. In Figure 5 (top left), where no load balancing is employed, there is a worker (green line) that roughly receives any load throughout the experiment. We can identify the same behavior during the first window (0-60s) of the load balancing as well. However, after the first balancing action, the worker receives worksets that have an evidently high load and participates actively in the processing. Of course, we can also identify some limitations of our approach. Although the load balancing scheme manages to bring 4 out of 5 workers to similar loads, a worker still has a higher load than the rest (blue line). Responsible for this behavior is a big,

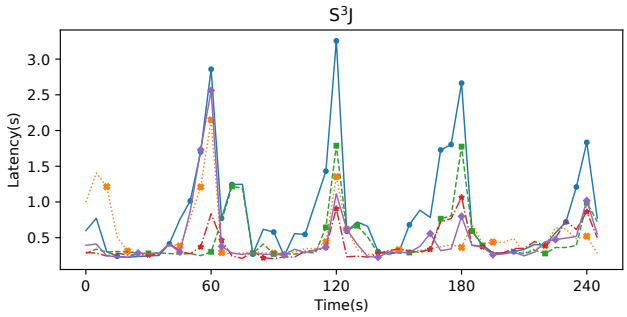

Figure 6: 99% latency percentile per worker. Each line represents a single worker (in this case, 5 workers in total). Uniform distribution, Incoming ratio 8000 records per sec, Selectivity: 0.1%, Parallelism: 5.

heavily loaded workset. As we describe in section 8, we flag big worksets with a load higher than the average load of the workers as irremovable, and we do not consider them for the load balancing. In future work, we plan to divide these big worksets into smaller ones which we can then consider for load balancing.

9.7 Summary of experiments

Our experiments show that the partitioning scheme of S^{3}J can retain sub-second latency for low selectivities even with low parallelism. S^{3}J's partitioning can handle high selectivities significantly more efficiently than the baseline and retain a higher processing throughput for unsustainable input rates. It can scale efficiently with increasing parallelism and leverages better than the baseline the available resources. In terms of duplication, S^{3}J retains an almost constant ratio of 2x as the selectivity increases, in contrast to the baseline. As parallelism increases, the duplication ratio of S^{3}J increases, but at a slower rate. As far as comparisons reduction is concerned, S^{3}J manages to drastically reduce the performed comparisons, primarily thanks to its workset concept (Section 7) that allows S^{3}J to emit pairs of records belonging to the same inner set without actually performing the similarity computation. The load balancing scheme manages to redistribute well the worksets and their load to the workers. This results in gradually reducing the maximum latency as well as equally involving all workers in the processing of the load, increasing their utilization.

10 CONCLUSIONS

Current approaches for distributed streaming similarity joins are tailored solutions to specific problems and are unable to adapt to concept drift or load imbalance. We presented S^{3}J, a generic approach for distributed streaming similarity joins that tackles the problem in the general metric space and applies load balancing to adapt to load imbalances while reducing the number of computations through smart partitioning that enables the load balancing technique. Our empirical evaluation suggests that S^{3}J can adapt efficiently to load imbalances, scales effectively as the parallelism increases without enforcing high duplication overhead, reduces the unnecessary similarity computations, and enables low latency similarity join results for low selectivity queries.

ACKNOWLEDGMENTS

This publication is part of the project number 19708, of the Vidi research programme partly financed by the Dutch Research Council (NWO). It is also partially funded by European Union's Horizon Europe research and innovation programme, under grant agreement No. 101070122, and by ICAI AI for Fintech Research Lab.

REFERENCES

[1] Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing Joins in a Map-Reduce Environment. In *Proceedings of the 13th International Conference on Extending Database Technology* (Lausanne, Switzerland) *(EDBT '10)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/1739041.1739056

[2] Gagan Aggarwal, Rajeev Motwani, and An Zhu. 2003. The Load Rebalancing Problem. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, California, USA) *(SPAA '03)*. Association for Computing Machinery, New York, NY, USA, 258–265. https://doi.org/10.1145/777412.777460

[3] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic Data Stream Processing. *Future Generation Computer Systems* 87 (2018), 171–185. https://doi.org/10.1016/j.future.2018.05.025

[4] Ruay-Shiung Chang, Jih-Sheng Chang, and Po-Sheng Lin. 2009. An ant algorithm for balanced job scheduling in grids. *Future Generation Computer Systems* 25, 1 (2009), 20–27.

[5] Gang Chen, Keyu Yang, Lu Chen, Yunjun Gao, Baihua Zheng, and Chun Chen. 2017. Metric Similarity Joins Using MapReduce. 29, 3 (March 2017), 656–669. https://doi.org/10.1109/TKDE.2016.2631599

[6] Lin Chen, Cho-Li Wang, and Francis. C.M. Lau. 2008. Process reassignment with reduced migration cost in grid load rebalancing. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–13. https://doi.org/10.1109/IPDPS.2008.4536164

[7] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. ClusterJoin: A Similarity Joins Framework Using Map-Reduce. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1059–1070. https://doi.org/10.14778/2732977.2732981

[8] Gianmarco De Francisci Morales and Aristides Gionis. 2016. Streaming Similarity Self-Join. *Proc. VLDB Endow.* 9, 10 (June 2016), 792–803. https://doi.org/10.14778/2977797.2977805

[9] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. 2014. MassJoin: A mapreduce-based method for scalable string similarity joins. In *2014 IEEE 30th International Conference on Data Engineering*. 340–351. https://doi.org/10.1109/ICDE.2014.6816663

[10] Manuel Dossinger and Sebastian Michel. 2019. Scaling Out Multi-Way Stream Joins using Optimized, Iterative Probing. In *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*. IEEE, 449–456. https://doi.org/10.1109/BigData47090.2019.9005973

[11] Mohammed ElSeidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and Adaptive Online Joins. (2014), 16. http://infoscience.epfl.ch/record/190035

[12] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on Mapreduce: An Experimental Survey. *Proc. VLDB Endow.* 11, 10 (June 2018), 1110–1122. https://doi.org/10.14778/3231751.3231760

[13] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 1–37.

[14] Yang Gao, Hongqiang Rong, and Joshua Zhexue Huang. 2005. Adaptive grid job scheduling with genetic algorithms. *Future Generation Computer Systems* 21, 1 (2005), 151–161. https://doi.org/10.1016/j.future.2004.09.033

[15] Shamsollah Ghanbari and Mohamed Othman. 2012. A priority based job scheduling algorithm in cloud computing. *Procedia Engineering* 50, 0 (2012), 778–785.

[16] X. Gu, P. S. Yu, and H. Wang. 2007. Adaptive Load Diffusion for Multiway Windowed Stream Joins. In *2007 IEEE 23rd International Conference on Data Engineering*. 146–155. https://doi.org/10.1109/ICDE.2007.367860

[17] Xiao Hu, Yufei Tao, and Ke Yi. 2017. Output-optimal Parallel Algorithms for Similarity Joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 79–90. https://doi.org/10.1145/3034786.3056110

[18] Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *ACM Trans. Database Syst.* 44, 2 (2019), 6:1–6:36. https://doi.org/10.1145/3311967

[19] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[20] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang. 2004. - Approximate NN Queries on Streams with Guaranteed Error/performance Bounds. In *Proceedings 2004 VLDB Conference*. Morgan Kaufmann, St Louis, 804–815. https://doi.org/10.1016/B978-012088469-8.50071-1

[21] X. Lian and L. Chen. 2011. Similarity Join Processing on Uncertain Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 23, 11 (2011), 1718–1734. https://doi.org/10.1109/TKDE.2010.208

[22] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 811–825. https://doi.org/10.1145/2723372.2746485

[23] Kasper Grud Skat Madsen, Yongluan Zhou, and Jianneng Cao. 2017. Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 227–230. https://doi.org/10.1109/ICDE.2017.81

[24] Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-Join: A Scalable Mapreduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proc. VLDB Endow.* 5, 8 (April 2012), 704–715. https://doi.org/10.14778/2212351.2212353

[25] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 493–505. https://www.usenix.org/conference/atc16/technical-sessions/presentation/najafi

[26] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2020. Scalable Multiway Stream Joins in Hardware. *IEEE Transactions on Knowledge and Data Engineering* 32, 12 (2020), 2438–2452. https://doi.org/10.1109/TKDE.2019.2916860

[27] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE'06)*. 49–49. https://doi.org/10.1109/ICDE.2006.105

[28] Yuan Qiu, Serafeim Papadias, and Ke Yi. 2019. Streaming HyperCube: A Massively Parallel Stream Join Algorithm. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, 642–645. https://doi.org/10.5441/002/edbt.2019.76

[29] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2021. Distributed Stream KNN Join. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1597–1609. https://doi.org/10.1145/3448016.3457269

[30] Mehdi Sheikhalishahi, Richard M Wallace, Lucio Grandinetti, José Luis Vazquez-Poletti, and Francesca Guerriero. 2016. A multi-dimensional job scheduling. *Future Generation Computer Systems* 54 (2016), 123–131.

[31] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over One Billion Tweets Using Parallel Locality-Sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1930–1941. https://doi.org/10.14778/2556549.2556574

[32] Sahaana Suri, Ihab F. Ilyas, Christopher Ré, and Theodoros Rekatsinas. 2021. Ember: No-Code Context Enrichment via Similarity-Based Keyless Joins. *Proc. VLDB Endow.* 15, 3 (nov 2021), 699–712. https://doi.org/10.14778/3494124.3494149

[33] Alexey Tsymbal. 2004. The problem of concept drift: definitions and related work. (2004).

[34] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient Parallel Set-Similarity Joins Using MapReduce *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 495–506. https://doi.org/10.1145/1807167.1807222

[35] Song Wang and Elke Rundensteiner. 2009. Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) *(EDBT '09)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/1516360.1516396

[36] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. 2013. Scalable All-Pairs Similarity Search in Metric Spaces *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 829–837. https://doi.org/10.1145/2487575.2487625

[37] J. Wu, Y. Zhang, J. Wang, C. Lin, Y. Fu, and C. Xing. 2019. Scalable Metric Similarity Join Using MapReduce. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1662–1665. https://doi.org/10.1109/ICDE.2019.00167

[38] Jianye Yang, Wenjie Zhang, Xiang Wang, Ying Zhang, and Xuemin Lin. 2020. Distributed Streaming Set Similarity Join. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 565–576.

[39] Matei Zaharia, Dhruba Borthakur, J Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2009. *Job scheduling for multi-user mapreduce clusters*. Technical Report. UCB/EECS-2009-55, EECS Department, University of California.

[40] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. 2006. Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*. Springer Berlin Heidelberg, Berlin, Heidelberg, 54–71.