electronics

MDPI

Article

Dynamic Load Balancing in Stream Processing Pipelines Containing Stream-Static Joins

Josip Marić, Krešimir Pripužić *, Martina Antonić and Dejan Škvorc

Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, 10000 Zagreb, Croatia; josip.maric2@fer.hr (J.M.); martina.antonic@fer.hr (M.A.); dejan.skvorc@fer.hr (D.Š.)
* Correspondence: kresimir.pripuzic@fer.hr

Abstract: Data stream processing systems are used to continuously run mission-critical applications for real-time monitoring and alerting. These systems require high throughput and low latency to process incoming data streams in real time. However, changes in the distribution of incoming data streams over time can cause partition skew, which is defined as an unequal distribution of data partitions among workers, resulting in sub-optimal processing due to an unbalanced load. This paper presents the first solution designed specifically to address partition skew in the context of joining streaming and static data. Our solution uses state-of-the-art principles to monitor processing load, detect load imbalance, and dynamically redistribute partitions, to achieve optimal load balance. To accomplish this, our solution leverages the collocation of streaming and static data, while considering the processing load of the join and the subsequent stream processing operations. Finally, we present the results of an experimental evaluation, in which we compared the throughput and latency of four stream processing pipelines containing such a join. The results show that our solution achieved significantly higher throughput and lower latency than the competing approaches.

Keywords: data stream processing; adaptive load balancing; dynamic load balancing; partition skew

check for updates

Citation: Marić, J.; Pripužić, K.; Antonić, M.; Škvorc, D. Dynamic Load Balancing in Stream Processing Pipelines Containing Stream-Static Joins. *Electronics* **2023**, *12*, 1613. https://doi.org/10.3390/electronics12071613

Academic Editors: Marios Avgeris, Dimitrios Dechouniotis, Konstantinos Tsitseklis and Vitoropoulou Margarita

Received: 28 February 2023
Revised: 17 March 2023
Accepted: 28 March 2023
Published: 29 March 2023

1. Introduction

A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items, for which it is impossible to control the order in which items arrive, nor feasible to locally store a stream in its entirety [1]. Data streams come from various sources, including Internet of Things (IoT) devices, social media platforms, online gaming, media publishing, system logs, financial markets, mobile devices, sensors, security systems, smart energy systems, utility systems, etc. Data streams are either homogeneous, where all the items share similar structures and characteristics, or heterogeneous, with varying items. Stateless data stream processing handles each item independently, while stateful data stream processing shares the state between items. Data stream processing (DSP) is now essential for decision-making processes in data-driven organizations. Despite being an active research topic for over 20 years, recent successful initiatives by the research and open-source communities have led to its peak.

The most popular open-source streaming platforms are Apache Flink [2], Apache Spark [3], Apache Storm [4], and Apache Kafka Streams [5]. These platforms provide tools and libraries for developers to build streaming applications that can process data streams in real time; they are designed to handle large amounts of streaming data, often distributed across multiple worker nodes, and they provide features for fault tolerance, scalability, and data processing. While these platforms provide robust capabilities for building complex data stream processing systems (DSPSs), they lack dynamic adaptation mechanisms for handling skewed and fluctuating data streams: this can lead to partition skew, which negatively affects the overall processing performance of DSPSs, by increasing latency and decreasing throughput.

In this paper, we address the problem of load imbalance, due to the partition skew, in stream-static joins. This problem arises when streaming data has to be joined in real time to static data that is too large to be replicated on the distributed workers, but instead has to be partitioned among them. Moreover, the incoming streaming data also has to be partitioned among the workers, in order to perform the join in real time; therefore, a worker must have a part of the static data and the corresponding part of the streaming data, to perform a join between them, which is in fact just a subtask of the whole stream-static join operation. If a worker is missing the required part of the (static or streaming) data, then that worker would first need to fetch the necessary data from other workers, before being able to perform the join.

To achieve optimal performance for the entire stream-static join process, we present a solution that dynamically balances the processing load, and reduces the network load among the workers. The objective is to partition both streaming and static data in such a way that they match each other when needed in stream-static joins, and to dynamically assign them to worker nodes in a way that evenly distributes the load across the system. The solution can operate in different modes, depending on various factors:

- the location of the join—whether it is performed on workers with static or streaming data;
- the imbalance detection method—whether by counting streaming items or summing the processing load;
- whether the solution dynamically adapts to detected imbalances or not.

Overall, the solution is designed to be flexible and adaptable to different scenarios, in order to achieve the best possible performance for the stream-static join process.

Our approach is applicable to any streaming platform, making it a generic solution; however, for this paper, we experimentally evaluated its implementation on the Apache Spark platform. Specifically, we conducted experiments using four different stream processing pipelines that were described in our previous paper [6]; by doing so, we were able to provide more precise and concrete results for the performance of our solution in a real-world setting using a widely used streaming platform like Apache Spark.

The main contributions of the paper can be summarized as follows:

1. We propose a novel solution that dynamically balances the processing load, and reduces the network load in the stream-static join. Our solution operates in different modes, depending on the location where the join is performed, and the method used to detect imbalance.
2. To evaluate the effectiveness of our approach, we performed an experimental evaluation to measure throughput and latency, comparing our solution against competing approaches. By analyzing the results, we identify the best-performing modes of our approach, and we provide insights into the benefits of our solution.

The rest of the paper is structured as follows: Section 2 gives a brief overview of related work addressing load balancing in data stream processing systems; in Section 3, we present the architecture of our solution, and we explain how it monitors and detects the load imbalance and initiates the rebalance; we experimentally evaluate the throughput and latency of our approach and competing approaches in Section 4; finally, we discuss the results, and give directions for future work, in Section 5.

2. Related Work

The problem, that load balancing in data stream processing systems (DSPSs) can lead to data skew, has been extensively studied in the literature. Most papers address this issue in the context of MapReduce, a programming model used for distributed processing of large-scale datasets. Irandoost et al. [7] conducted a systematic review, to investigate current techniques for handling data skewness. The authors presented a new classification scheme for methods that address data skewness in MapReduce, and evaluated the advantages and disadvantages of algorithms within each category. Additionally, they identified key areas for further research aimed at improving data skewness handling techniques.

One of the early works in this area is presented in [8], where the authors introduced a static load-balancing algorithm that evenly distributed work among reducers in a MapReduce job, resulting in a significant reduction in elapsed time. The algorithm used a progressive objective-based cluster sampler to estimate the load associated with each reduce-key. If a key had a large load, it was split into sub-keys that could be assigned to different reducers. For keys with medium loads, the algorithm assigned them to reducers in a way that minimized the maximum reducer load. Keys with small loads were hashed, as they had minimal effect on the balance. The process was repeated until the balancing objective and confidence level specified by the user were met. Chen et al. [9] introduced LIBRA, a lightweight strategy designed to tackle the issue of data skew among reducers in MapReduce applications. LIBRA employs an innovative sampling technique that can accurately approximate the distribution of intermediate data by only sampling a small proportion of it during normal map processing; based on the assumption that it made a good representation of data, they distributed the partitions so that the workers would be optimally balanced in the reduce phase, i.e., they allowed the reduce tasks to begin copying as soon as the chosen sample map tasks had been completed. They also suggested cluster splitting, which is basically splitting larger partitions into smaller ones. We also redistribute the partitions; however, in our approach this is done periodically, and upon stream-static joins, which requires handling the collocation of both stream and static data. In [10], the authors expanded upon the capabilities of Chisel, a system that manages partitioning imbalances in reduce tasks by dynamically identifying skewed partitions, as maps generate their outputs. Chisel achieves this by dynamically introducing partial reducers and one global reducer that receives data from partial reducers. Chisel++, however, suggests removing a global reducer, by introducing a range partitioning technique, instead of the global reducer. The downside of both Chisel and Chisel++ is the necessity for data to go through multiple reducers, and splitting the reducer for one key: the former requires more network load being sent, and splitting the reducer can also be inefficient for complex reducers. This design pattern would be very costly, as it would require dynamic repartitioning of static data, which tends to be large.

Gao et al. [11] suggested a distributed algorithm that approximates the load balance problem in MapReduce, specifically for data quality detection. Their approach is applicable to independent tasks that are processed on MapReduce. In the first MapReduce round, the input data are sorted in descending order, based on the independent workload. In the second MapReduce round, each task in the input data is assigned to the reducer with the least workload in turn. Overall, this provides a general solution to load balancing in MapReduce. In [12], the authors introduced a novel method for addressing skew, referred to as multi-dimensional range partitioning (MDRP). This technique overcomes the limitations of conventional algorithms in two key ways: firstly, it considers the expected number of output records for each machine, which enhances its ability to handle join product skew; secondly, a small subset of input records is sampled prior to the join execution, to facilitate an efficient execution plan while taking data skew into account. The proposed algorithm can handle complex join operations like theta-joins and multi-way joins without any modifications to the original MapReduce environment.

Liroz-Gistau et al. [13] introduced a Hadoop-based system, FP-Hadoop, that enhances the parallelism of the reduce side in MapReduce by effectively addressing reduce data skew. FP-Hadoop incorporates a new phase, known as intermediate reduce (IR), where intermediate value blocks are processed in parallel by intermediate reduce workers. With this approach, the bulk of the reducing work can be performed in parallel, even when all intermediate values are associated with the same key, allowing for the full utilization of the available workers' computational power. This solution comes with the price of having an intermediate reducer(s), similar to Flux [14]. Zhao et al. [15] proposed a technique called kNN-DP for kNN-join using MapReduce, which aims to divide data objects into multiple partitions of equal size, which can be processed in parallel by mappers and reducers. The key component of kNN-DP is a data partitioning module, which strategically partitions data to optimize kNN-join performance by mitigating data skewness on Hadoop clusters: to achieve this, they first designed a sampling method to analyze the data distribution of a small sample dataset that represents large datasets; they then dynamically adjusted the partition boundaries, by analyzing the time complexity of each partition in the sample dataset; finally, they enhanced the accuracy of the parallel kNN-join, by adding a small amount of redundant data to each node's local data. They had a similar monitoring and redistribution mechanism to ours; however, being for a different use case, they only handled the join part of a MapReduce job, not dealing with streaming data, which we were interested in, in this paper. Contrary to our solution, their case did not require optimizing data collocation. Additionally, we also optimize the impact of the join on the subsequent stream processing operations. In [16], the authors suggested a novel technique called fine-grained partitioning for skew data (FGSD), which is capable of enhancing the load balancing of reduce tasks in the presence of skewed data. FGSD takes into account the characteristics of both input and output data, by employing a stream sampling algorithm. The technique introduces a new method for distributing input data, which facilitates efficient management of skew resulting from redistribution and join product. FGSD does not necessitate any alterations to the MapReduce environment, and can be utilized for complex join operations.

Although streaming platforms use a modified version of MapReduce to process data streams in real time, these papers proposed low-level enhancements that required significant changes to these platforms (i.e., their DSP operators), for practical use. However, we are interested in a different approach, which leverages the existing capabilities of a platform, to enhance partition skew handling. To be more precise, we handle partition skew by creating more partitions than there are available workers, and redistributing them among the workers, to balance the processing load, without repartitioning the data, which is a well-known technique for handling partition skew [17]. On the other hand, several research papers have proposed solutions based on the repartitioning of the data. Shah et al. [14] presented a state migration protocol that enabled the repartitioning mechanism of Flux, which was designed to handle Continuous Queries. Flux was introduced in 2003, when DSPSs were still being developed, and Continuous Queries can be considered an ancestor of modern DSPSs. The process of moving a partition from one instance to another involves three steps: first, pausing and buffering the input to the partition; second, transferring the partition; and finally, restarting the paused input stream. By pausing and buffering the input of the transferring partition, consistency is ensured while the remaining partitions remain active. Flux is basically an adaptive partitioning operator, placed between producing and consuming operators, which allows dynamic repartitioning policies to be executed, so that the consuming operator manages an optimally partitioned stream. Flux's state migration mechanism is a base for many techniques and research papers executing state migration. Analogous to migrating the states, our solution proposes redistributing the static data between workers in a similar way; however, join being a stateless operation, it does not require a pause in processing, as Flux's mechanism suggests. Zhang et al. [18] utilized the concept of *power of two choices*, from a previous study by Nasir et al. [19]. They proposed a method called Back Propagation Grouping (BPG), which involves key splitting, backpropagation, and calibration signal concepts, to achieve load balancing. By splitting the key into multiple operator instances, they were able to distribute a load of events with frequent keys. They monitored the real load centrally, and used this information to calibrate the load partitioning. They implemented a processing entity, called *tinker*, that collected load statistics from all workers at regular intervals, and sent a calibration signal to sources, allowing them to correct the load estimation bias accumulated in one cycle. In other words, they suggested a static load-balancing method for stateful processing, where one key was split between two workers, whereas we dynamically distribute the partitions, and do not require splitting partitions, which can be challenging. In [20], the authors proposed a key reassigning and splitting partition algorithm, to handle the partition skew. The proposed algorithm takes into account the partition balance of the intermediate data, as well as after the shuffle operator. Cardellini et al. [21] introduced stateful migration and autoscaling for Apache Storm. The approach involved extracting the state from the old instance, and replaying it to the new instance. The task needed to be paused, to ensure consistent migration. To successfully introduce this approach, they added multiple components. Distributed Data Store (DDS) enabled the decoupling of the transferring state from the related operator instance. Two classes were also implemented into Storm code, allowing for storing and retrieving the partition of the operator state in a user-transparent way.

Tang et al. [22] suggested a solution for handling the unequal distribution of data in bucket containers during the shuffle process of the Spark computing framework. Their proposed method, called Skew Intermediate Data (SCID), utilizes a sampling algorithm based on reservoir sampling, to identify the distribution of keys in intermediate data blocks. The main focus is on splitting and combining the output data from map tasks, to ensure that the data is allocated to the correct buckets, based on the estimated frequencies of the keys. The splitting is done when a data cluster exceeds the remaining space in the current bucket, and the remaining cluster is processed in the next iteration: this ensures that each bucket has an equal amount of data. The most similar approach to ours was [23], which introduced a novel SP-Partitioner to handle the partition skew much more efficiently than the built-in hash and range partitioning methods in Apache Spark. The SP-Partitioner, placed between the map and reduce stages, calculates the distribution of keys in the sample data, and uses this information to generate a new partitioning strategy that handles the partition skew much more efficiently. On the other hand, our approach can be used in combination with built-in partitioning methods, and does not require the use of a modified partitioning method.

Finally, it is very important to mention that achieving a balanced load is one of the oldest [24] and most significant issues [25] in the field of distributed systems: it has persisted throughout different types of distributed systems, including emerging ones [26]. For instance, federated file systems, which allow multiple computers to share files as if they are part of a single system, also require load balancing. Recently, a generic job and resource-aware data storage and placement algorithm (JRAP) was proposed by A. Khan et al. [27], to address load balancing in federated file systems. JRAP aims to compute the optimal edge server for job requests, while considering factors such as job type, available capacity, storage, computation, and network bandwidth. Similarly to our approach, this algorithm is executed by a centralized process, namely the JRAP manager, which is responsible for controlling and balancing the data flow in the federation.

3. Dynamic Load Balancing

In this section, we present our solution for dynamic load balancing in stream processing pipelines that contain stream-static joins.

Figure 1 shows the logical architecture of our solution. Each circle represents a worker who participates in the data stream processing. Squares that are transferred between workers represent streaming data. Flat cylinders represent partitioned static data that have to be joined to the streaming data. Each color (of squares and cylinders) represents a single partition of data. We see that streaming data is coming from workers on the left side, and has to be joined to static data by workers on the right side; therefore, a data stream operation that precedes the stream-static join is performed by the workers on the left side. The Dynamic Distributor, a process responsible for collecting load metrics from workers, and executing the rebalancing of the load when necessary, is shown at the bottom. The rectangles that are attached to the workers, and connected to the Dynamic Distributor with dotted lines, are simple components responsible for providing the load metrics, and for optionally executing the redistribution of partitions. The metrics that are collected provide information to the Dynamic Distributor, regarding the distribution of the load among the workers. For example, if the processing load of a single worker is noticeably higher than the average processing load, the Dynamic Distributor will initiate the redistribution of the partitions.

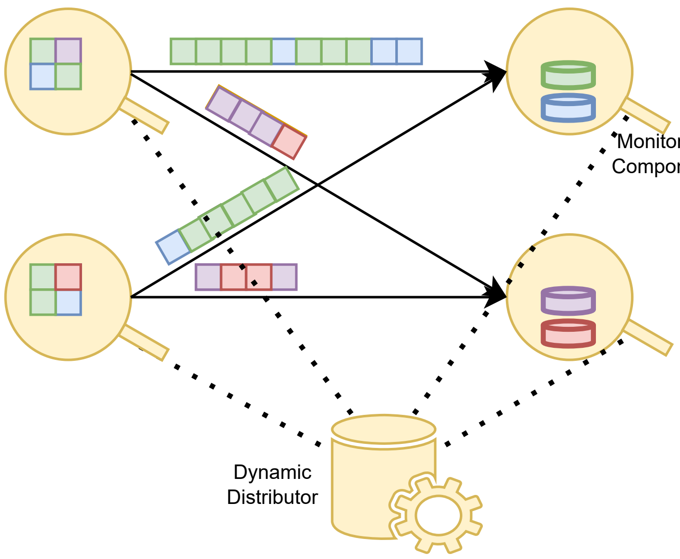


Figure 1. Logical architecture of dynamic load balancing method.

The core component of our solution is the Dynamic Distributor, because it dynamically collects load metrics from workers, to detect the load imbalance, and then initiates the rebalance by redistributing partitions of static data, depending on the selected mode. A streaming platform usually provides features that allow for the publishing and collecting of worker load metrics. After each rebalancing, a streaming platform is responsible for the routing of streaming data to new workers, to whom the corresponding partitions of static data have been redistributed. The Dynamic Distributor considers both the processing load of each partition from collected metrics, and the collocation of streaming and static data, when determining a new optimal distribution of static partitions. In more detail, the Dynamic Distributor needs the *location* of each partition (i.e., the worker responsible for processing the data belonging to the partition) and an *indicator* of the processing load for each partition, to recognize any imbalances in processing load among workers, and to initiate partition redistribution when needed.

In Algorithm 1, we present the pseudocode of the Dynamic Distributor. As we can see, the algorithm runs periodically in an infinite loop between lines 1 and 11, with a period defined in line 10. Running the algorithm periodically is necessary, because the collected load metrics may not change frequently, and to be constantly checking for updates when there is no change would result in unreasonable resource utilization. Inside the loop, the algorithm consists of the following three parts:

1. *monitoring load metrics* in lines 2 and 3;
2. *load imbalance detection* in lines 4 to 7;
3. *partition redistribution* in line 8.

Each part of the algorithm is explained in detail in Sections 3.1–3.3, respectively. Hereafter, we briefly present the algorithm.

The algorithm obtains the locations (i.e., the workers) and indicators of the processing load for each partition in lines 2 and 3. As we will see in Section 3.1, our algorithm is flexible in supporting different load metrics. In line 4, the algorithm calculates the processing load per location, from previously received data. In line 5, for each location, the algorithm calculates the *load distance per location*, which we define as the difference in the processing load of a location compared to the average load of all locations. Finally, the algorithm calculates the *maximum load distance* among load distances per location, which is used as a condition in line 7 to initiate the redistribution of partitions in line 8. Our algorithm is designed to react when the processing load of a single worker is much higher (i.e., above the *threshold*) than the average load, and this is achieved by using the maximum load distance as a condition. As we see in line 7, the maximum load distance is not the sole condition for initiating the partition redistribution. Our algorithm uses three *additional conditions* that need to be satisfied, to initiate the partition redistribution. We present and discuss these additional conditions in Section 3.2.

Algorithm 1 Algorithm of Dynamic Distributor

```
 1: while true do
 2:     $locations \leftarrow extractLocationPerPartition()$
 3:     $loads \leftarrow extractLoadPerPartition()$
 4:     $loadsPerLocation \leftarrow calculate(locations, loads)$
 5:     $loadDistancesPerLocation \leftarrow calculate(loadsPerLocation)$
 6:     $maxLoadDistance \leftarrow calculate(loadDistancesPerLocation)$
 7:     if $maxLoadDistance > threshold$ & additional conditions are satisfied then
 8:         initiate partition redistribution
 9:     end if
10:     $sleep(period)$
11: end while
```

3.1. Monitoring Load Metrics

As previously explained, our solution needs to monitor locations (i.e., workers) and processing loads for each partition; therefore, each worker who performs a stream-static join must provide these metrics to the Dynamic Distributor. While the monitoring of locations is quite easy to implement, this is not the case for the monitoring of processing load, as we discuss next.

The most simple processing load metric is to count the number of streaming items processed by each worker per (stream-static) join operation. Using this metric, our Dynamic Distributor is able to balance the number of streaming items processed by the workers. If the processing of each streaming item takes a similar amount of time, which means that the incoming stream is *homogeneous*, this would consequentially balance the processing load on the workers—an approach that was followed by the SP-Partitioner from [23]: however, this is not the case for a *heterogeneous* stream, where the processing of some streaming items takes much more time than for others. To balance the processing load on the workers in this case, we cannot use the simple metrics of counting items: instead, we need to sum the actual processing times of the streaming items. However, measuring only the processing time for the (stream-static) join operation is not enough, as each such operation is followed by additional operations in the stream processing pipeline, which we also must take into account when measuring the actual processing load. Therefore, by using the latter metric, Dynamic Distributor is able to balance the processing load on the workers for the whole stream processing pipeline, and not just for the join operation: this is true for both the homogeneous and the heterogeneous streams.

3.2. Load Imbalance Detection

The throughput and latency of a stream processing pipeline can be heavily degraded due to an imbalance in the processing load among the workers: in such cases, some workers are overloaded while others are mostly waiting idle. As operations in the processing pipeline are causally related, the idle workers cannot start working on their next tasks until the overloaded workers finish their previously assigned tasks. Additionally, the idle workers cannot help the overloaded workers without redistributing partitions, as they do not have the necessary data to perform these tasks, while it can be quite costly (in terms of network load and processing time) to transfer the missing data ad hoc from the overloaded workers to the idle workers.

Therefore, the processing load imbalance of a single worker will result in increased latency and reduced throughput of the whole stream processing pipeline: for this reason, we used the *maximum load distance coefficient* (MLDC) as a load imbalance metric, because it is directly affected by the imbalance of any single worker [28]:

$$MLDC = max(LDC_0, LDC_1 \ldots, LDC_n) \tag{1}$$

As we see from the above equation, the MLDC metric is defined as the maximum of the individual *load distance coefficients* (LDCs) of the workers. The load distance coefficient LDC_i of a worker i is defined as follows:

$$LDC_i = (L_i - \overline{L})/\overline{L}, \tag{2}$$

where L_i represents the *load* of a worker i, which can be any load metric, while $\overline{L}$ represents the average load of n workers:

$$\overline{L} = \sum_{i=1}^{n} L_i/n. \tag{3}$$

In our solution, we define the *load imbalance* as the value of MLDC being above a predefined *threshold*; however, in practice, it is too costly to initiate the redistribution of static data (i.e., rebalancing the processing load) each time the MLDC value is above the threshold, and thus we define *additional conditions* that have to be satisfied, as shown in Algorithm 1. The complete set of additional conditions that our solution uses in practice is as follows:

1. at least M items have been processed from the data stream;
2. the redistribution has not been initiated during the last N periods of duration D;
3. $MLDC_{current} - MLDC_{new} > miMLDC$.

The first additional condition protects against initiating the redistribution too early, when LDCs are probably not calculated precisely. The second additional condition prevents the redistribution being initiated more than once every $N \cdot D$ seconds. The third additional condition ensures that the redistribution is initiated only when there is a minimal improvement $miMLDC$ in the newly calculated MLDC value, $MLDC_{new}$, when compared to the current one, $MLDC_{current}$.

Our solution is flexible in supporting different stream processing use cases by changing the values of $threshold$, M, N, D, and $miMLDC$, which are basically the parameters of our Algorithm 1. As our solution is not self-adaptive, the values of these parameters need to be manually fine-tuned for each use case, to identify the optimal values. For example, in our experimental evaluation in Section 4, we identified the optimal values of these parameters, shown in Table 1, for the given stream processing pipelines.

Table 1. Default Parameter Values for the Dynamic Distributor.

Label	**Parameter**	**Value**
threshold	lower threshold of *MLDC*	0.5
M	number of processed items from the data stream	1000
N	number of periods to wait	3
D	period duration	10 s
miMLDC	minimal improvement of *MLDC*	0.3

3.3. Partition Redistribution

As we have already explained, our solution rebalances the processing load, by dynamically redistributing partitions of static data among workers, which also results in the redistribution of the streaming data.

Figure 2 shows an example of the redistribution of static data. As in Figure 1, each circle represents a worker who participates in the data stream processing, while the flat cylinders of different colors represent different partitions of static data. The number in each cylinder shows what percentage of the total processing load for a partition has been measured.

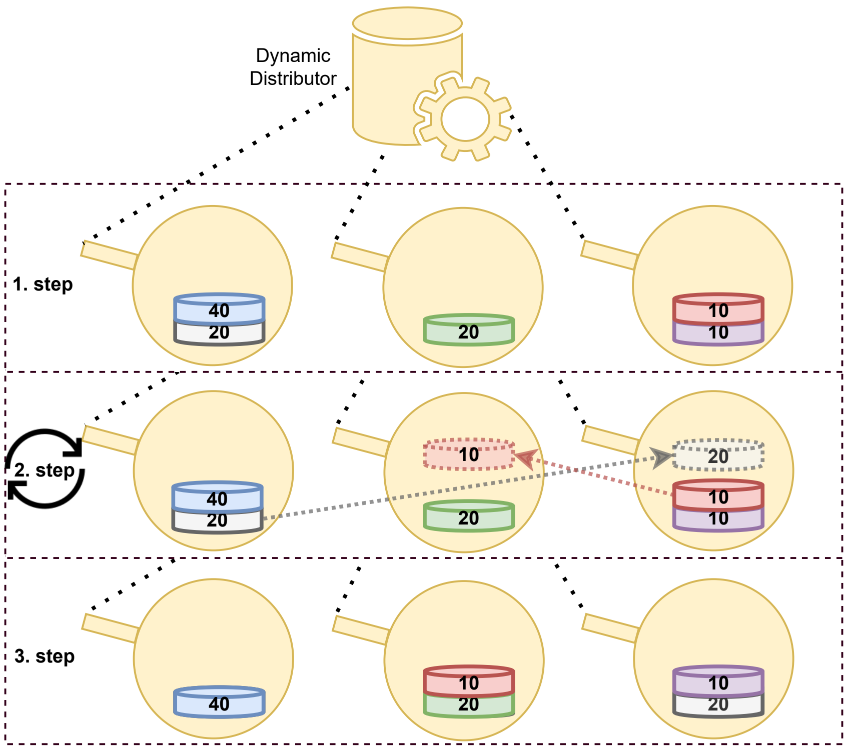


Figure 2. Redistribution of partitioned static data.

In the first step, the left worker is overloaded three times more than the other two workers. When all necessary conditions are satisfied, as explained previously in Section 3.2, the redistribution is initiated. Our solution implements a greedy partitioning algorithm, to find a new distribution of static data partitions that optimally balances the processing load, with minimal migrations of partitions between workers; however, any partitioning algorithm can be employed instead, as the redistribution is not initiated often. In the second step, the Dynamic Distributor finds a new optimal distribution of partitions, such that (1) the lower partition (20) on the left worker has to be migrated to the right worker, and (2) the upper partition (10) on the right worker has to be migrated to the middle worker. In the third step, we see the final distribution of partitions after finishing the redistribution, when streaming data for migrated partitions are also redirected to new workers.

4. Experimental Evaluation

In this section, we experimentally evaluate our solution presented in Section 3, by comparing the throughput and latency of four different stream processing pipelines that contain a stream-static join. The source code of our solution, implemented using the Apache Spark platform, is available in the following GitHub repository: https://gitlab.com/jmaric/dynamic-join.git, accessed on 10 March 2023. The datasets and Bash scripts utilized in the experimental evaluation presented in this section are also available in the same repository.

For the experimental evaluation, we decided to use stream processing pipelines from our previous paper [6], in which we first proposed a distributed geospatial publish/subscribe (GeoPS) system based on the Apache Spark platform, and then defined and compared four different subscription partitioning strategies for efficient processing of incoming publications. In the case of these strategies, the subscriptions were static, which allowed for their partitioning and replication among workers in the cluster. We saw these strategies as an ideal use case for our solution, because their stream processing jobs were based on a stream-static join between streaming publications and stored subscriptions.

Moreover, such a join is the most demanding part of these jobs, and is also highly dependent on load balancing, which we demonstrate in the experimental evaluation in this section.

To evaluate the performance of different elements of our solution, we ran each pipeline in six different modes, where each of these modes either turned on different elements of our solution or turned them off. In our experimental evaluation, we compared the throughput and latency of the pipelines in different modes, as an indication of the performance improvement achieved by our solution.

4.1. Apache Spark

Apache Spark [29] is an analytics engine, designed for processing Big Data, that includes pre-built modules for machine learning, data streaming, SQL, and graph processing. Apache Spark can operate on clusters in standalone mode, using its own cluster manager, or via other cluster managers, such as Apache Hadoop YARN [30]. In a Spark application, a single driver process launches a set of executor processes that are distributed across worker nodes. Executors have multiple task slots, and can execute many tasks concurrently throughout their lifetimes. The driver process first converts the application into one or more Spark jobs, and each job is then transformed into a logical execution plan represented by a directed acyclic graph (DAG). Once the DAG is constructed, the driver process divides it into stages that are subsequently divided into smaller tasks for execution by the executors. The stages need to be executed in topological order, as they are dependent on one another, while tasks within a stage can be executed concurrently.

4.2. Evaluated Stream Processing Pipelines

In Figure 3, we can see the most relevant part of the stream processing pipeline of our subscription partitioning strategies from [6], in which the stream-static join appears. For incoming publications from the data stream, we initiate such a join, to find candidate subscriptions for each publication (i.e., subscriptions that are probably interested in the publication). Then, using the $mapToPair$ method, we match each publication with its candidate subscriptions, to find subscriptions it satisfies (i.e., subscriptions that are certainly interested in the publication). Finally, using the $filter$ method, we filter out publications that do not satisfy at least one subscription, as no subscriber is interested in these publications. Hereafter, we present these four strategies briefly, as they are presented in detail in [6].

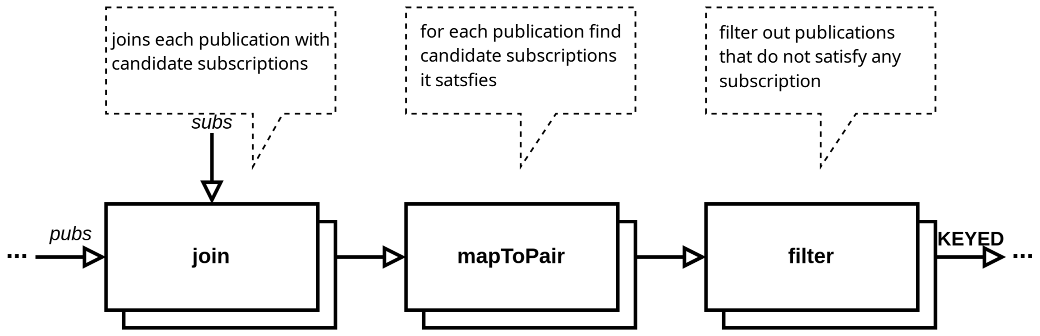

joins each publication with candidate subscriptions
for each publication find candidate subscriptions it satsfies
filter out publications that do not satisfy any subscription
subs
pubs
join
mapToPair
filter
KEYED
...
...

Figure 3. Relevant processing steps of the geospatial publish/subscribe system.

The first strategy is Spatially Partitioned Subscriptions (sPS), which employs a spatial partitioning method to distribute subscriptions among workers. Unlike the other strategies, it is a pure partitioning strategy that does not use spatial indexing for subscriptions. When a publication arrives, sPS identifies the spatial partitions to which it belongs, and forwards it to the workers responsible for those partitions; the workers then match the publication to the subscriptions belonging to the same partition.

The second strategy is Spatially Partitioned Index and Subscriptions (sPIS), which is similar to sPS but more time-efficient. Instead of checking all subscriptions within each partition, sPIS queries a spatial index of subscriptions for each partition, to efficiently identify candidate subscriptions.

The third strategy is Replicated Index Partitioned Subscriptions (RIPS), which utilizes a subscription partitioning method and a replicated spatial index that stores pairs of "partition ID–subscription ID". Upon the arrival of a publication, RIPS identifies candidate subscriptions by querying the spatial index, and forwards the publication and corresponding subscription IDs to the workers responsible for those partitions, who match the publication to subscriptions corresponding to those IDs. As the subscription partitioning method is used only to partition subscriptions, and not for identifying partitions for publications, RIPS can use both the hash and spatial partitioning methods for subscriptions. Consequently, there are two versions of RIPS, namely Replicated Index Hash Partitioned Subscriptions (RIhPS) and Replicated Index Spatially Partitioned Subscriptions (RIsPS).

4.3. Datasets

In our experimental evaluation, we used the datasets that are publicly available in repositories [31,32]. As we needed a geospatial dataset for our experiments, we selected a real-world dataset from [31], which provides information on car collisions in the UK. This dataset includes the exact location of each accident, the number of vehicles involved, the time of occurrence, and contextual details, such as road type, speed limit, and junction type; however, we only used the location information for our experiments. Although this dataset was useful, evaluating the selected stream processing pipelines based solely on location was not realistic. In order to conduct a more realistic evaluation, we required a dataset with more complex geospatial objects, such as polygons: we therefore combined the car collision dataset with polygons of UK postcode sectors, districts, and areas from [32]. By randomly selecting locations from the car collision dataset, and identifying the corresponding polygon from [32], we were able to generate spatial polygons in an approach similar to [33]. We believe this approach was suitable for our experimental study, as the generated polygons maintained the spatial distribution of the original geographic locations from [31]. It is important to note that the resulting data stream of publications was heterogeneous, as polygons require more processing time than points.

Our GitHub repository did not include generated subscriptions and publications. Instead, it contained the original datasets from [31,32], and the source code for the publication and subscription generators. This was because we conducted each experiment three times, to obtain average values, which removed any random effects; therefore, each time we generated a unique set of publications and subscriptions.

4.4. Experimental Setup

Our experimental evaluation was performed using Java 8 (OpenJDK) on a cluster that comprised 16 worker nodes. Each of these nodes contained 64 GB of RAM memory, and was equipped with an Intel Core i7-9700K CPU @ 3.60 GHz processor with eight physical cores, and no Hyper-Threading. For the Spark driver node, we employed a node with an Intel Core i7-4790 CPU @ 3.60 GHz processor containing 32 GB of RAM memory and four physical cores with HyperThreading. Additionally, our cluster comprised 7 more nodes, including 3 service nodes and 4 identical Kafka brokers, each equipped with 16 GB of RAM memory and Intel Core i7-2600 CPU @ 3.40 GHz processors. Our worker nodes were used solely for processing, and served as HDFS DataNodes, Spark Gateways, and YARN NodeManagers. We used Cloudera CDH 6.2.1 Express as a software distribution for the cluster, and enabled the HDFS, YARN, Spark, Zookeeper, and Kafka services. The Spark applications implemented consisted of subscription partitioning strategies (i.e., stream processing pipelines), having the following default parameters: 100,000 publications; 10,000 subscriptions; 16 Spark executors with 48 GB of memory and four cores per executor; four concurrent Spark jobs; 32 Kafka partitions; and a Spark micro-batch interval of 500 ms.

In Table 1, we can see the values of the Dynamic Distributor parameters that we used in our experiments: these parameters defined the conditions under which the redistribution was initiated, as shown in Algorithm 1, and as explained in Section 3.1.

We performed the evaluation of each subscription partitioning strategy (i.e., the stream processing pipeline), using the same method as proposed in [6]: "We generate subscriptions and publications, and then publish publications to a Kafka topic. After that, we start the Spark application of each subscription partitioning strategy that we want to test". During the experiments, we measured the throughput and latency as objective indicators of performance improvement.

4.5. Evaluated Modes

To evaluate the performance of different elements of our solution, we ran each stream processing pipeline in the following six different modes, which either turned on different elements of our solution or turned them off:

- *Static STRL*—the baseline mode from [6];
- *Static STAL;*
- *Dynamic LB STAL;*
- *Dynamic DB STAL;*
- *Dynamic LB STRL;*
- *Dynamic DB STRL*—the mode analogue to SP-Partitioner from [23].

The *Static* keyword indicated that both the streaming and static data were statically partitioned, while on the other hand, the *Dynamic* keyword indicated that they were dynamically partitioned. The *STRL* keyword stood for STReam-Local join, indicating that the join was performed on nodes containing streaming data. On the other hand, the *STAL* keyword stood for STAtic-Local join, indicating that the join was performed on nodes containing static data. For dynamic modes, we differentiated two types of load balancing: data-based (*DB*), which balanced the number of streaming items processed by each worker, and load-based (*LB*), which balanced the processing load (measured by the processing time) per worker.

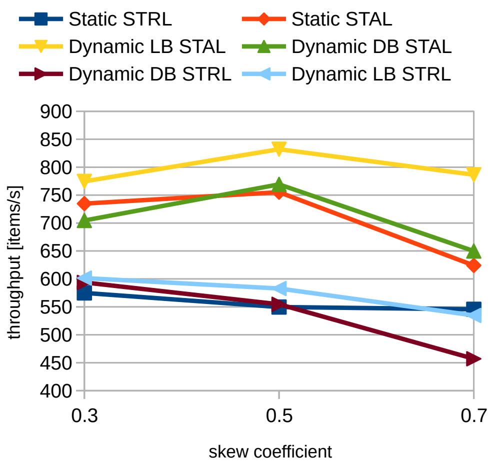
The *Static STRL* mode served as the baseline for comparison, as it did not incorporate any of the load-balancing mechanisms from our solution presented in Section 3. This mode was identical to our Apache Spark implementation from [6], and did not have any load-balancing mechanisms. The *Dynamic DB STRL* mode dynamically balanced the number of streaming items processed by each worker, being a load-balancing mechanism introduced by SP-Partitioner in [23].

4.6. Experimental Results

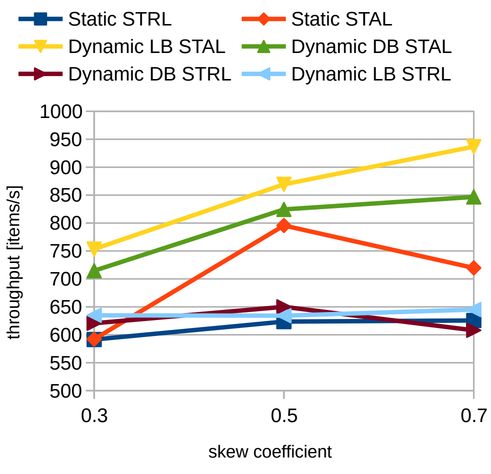
In the experiment, we evaluated the throughput and latency of different stream processing pipelines for different modes, when increasing the skew coefficient. The *skew coefficient* was defined as a portion of streaming data that belonged to partitions for which only 3 workers were responsible (out of 16 available). For example, half of the streaming data would be processed by only 3 workers when the skew coefficient was 0.5. A skew coefficient of $3/16 = 0.1875$ would represent completely balanced partitions.

In Figure 4, we see the average throughput in streaming items per second for different values of the skew coefficient. During the experiment, we increased the skew coefficient from well-balanced (0.3) to very skewed partitions (0.7). For the RIhPS pipeline, we only had values 0.3 and 0.4 of the skew coefficient, as we could not increase it further, due to the hash partitioning method that the RIhPS strategy used.

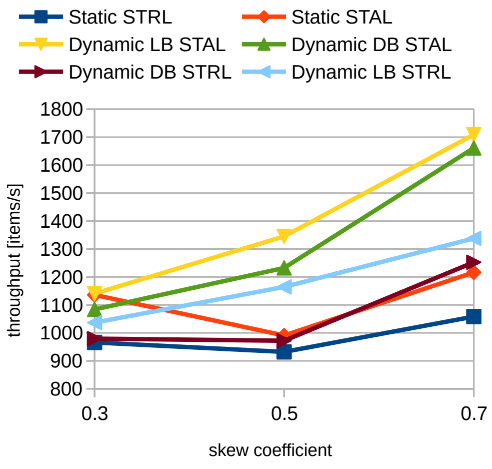
As we can see, the *Dynamic LB STAL* mode achieved the highest throughput for all the stream processing pipelines, as it turned on all elements of our solution, namely dynamic partitioning of subscriptions and publications (*Dynamic*), performed the stream-static join on nodes containing static data (*STAL*), and balanced the load on workers (*LB*). Moreover, we can see that the difference in throughput of the *Dynamic LB STAL* mode and baseline *Static STAL* mode increased with an increasing skew coefficient: this was expected, as dynamic partitioning, in the case of a high-skew coefficient, eventually leads to better load balancing, and thus to a more reduced skew coefficient, which results in higher throughput.



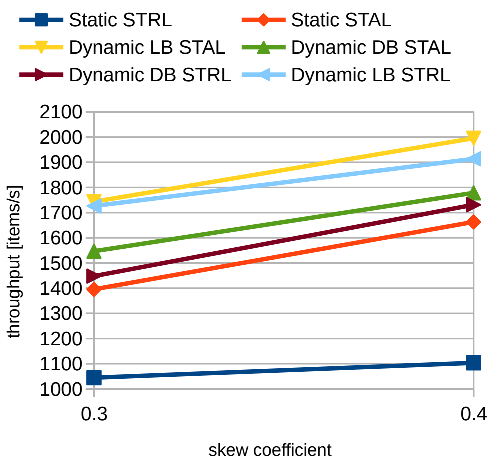
(**a**) sPS pipeline



(**b**) sPIS pipeline



(**c**) RIsPS pipeline



(**d**) RIhPS pipeline

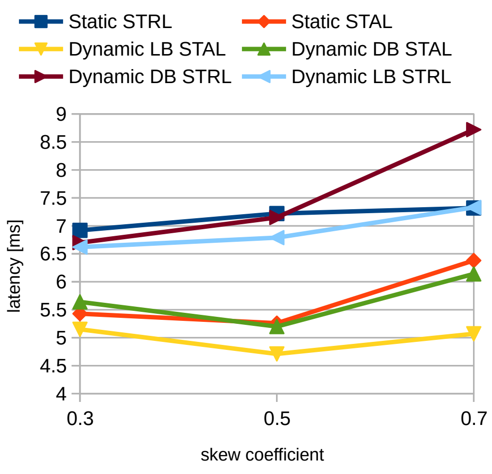
Figure 4. Average throughput against skew coefficient for different modes and pipelines.

We can also see that the *LB* modes (i.e., *Dynamic LB STAL* and *Dynamic LB STRL*) achieved higher throughput than the comparable *DB* modes (i.e., *Dynamic DB STAL* and *Dynamic DB STRL*) for all the evaluated stream processing pipelines. This was expected, because the incoming data stream of publications was heterogeneous, and thus we could achieve higher throughput by balancing the processing load per worker (*LB*), instead of balancing the number of streaming items processed by each worker (*DB*): for example, in the case of two workers, it would take much more time for one to process 10 complex items (i.e., publications that were polygons) than for the other to process 10 simple items (i.e., publications that were points); therefore, *LB* balancing was more efficient than *DB* balancing, because it would divide the load between workers, such that both would process 5 complex and 5 simple items.

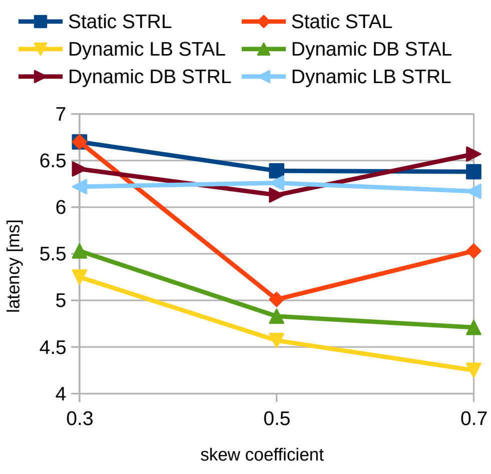
Similarly, we can see that the *STAL* modes (i.e., *Dynamic LB STAL*, *Dynamic DB STAL*, and *Static STAL*) achieved higher throughput than the comparable *STRL* modes (i.e., the *Dynamic LB STRL*, the *Dynamic DB STRL*, and the *Static STRL*) for all the evaluated stream processing pipelines. This was also expected, because it was more expensive to continuously migrate larger static data between the workers than smaller streaming data.

In Figure 5, we see the average latency for the same experiment. As expected, the *Dynamic LB STAL* mode was again the best-performing, and had the lowest latency, when compared to the other modes. The *Dynamic LB STAL* mode had a slightly higher latency than the *Static STAL* mode for the RIsPS pipeline, in the case of a skew coefficient of 0.3, due to an unnecessary rebalancing, which took time, and thus increased the latency.

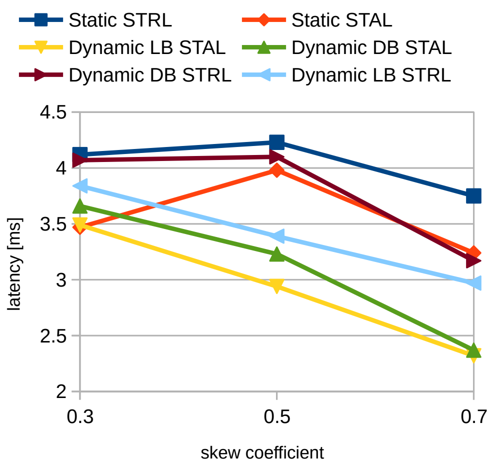
Additionally, for all the evaluated stream processing pipelines, the *LB* modes (i.e., the *Dynamic LB STAL* and the *Dynamic LB STRL*) generally achieved lower latency than the comparable *DB* modes (i.e., the *Dynamic DB STAL* and the *Dynamic DB STRL*), while the *STAL* modes (i.e., the *Dynamic LB STAL*, the *Dynamic DB STAL*, and the *Static STAL*) achieved lower latency than the comparable *STRL* modes (i.e., the *Dynamic LB STRL*, the *Dynamic DB STRL*, and the *Static STRL*).



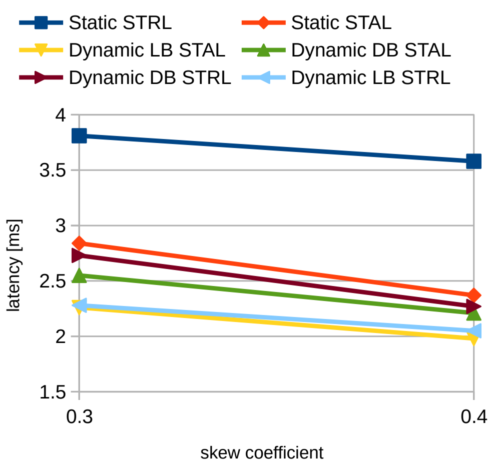
(**a**) sPS pipeline



(**b**) sPIS pipeline



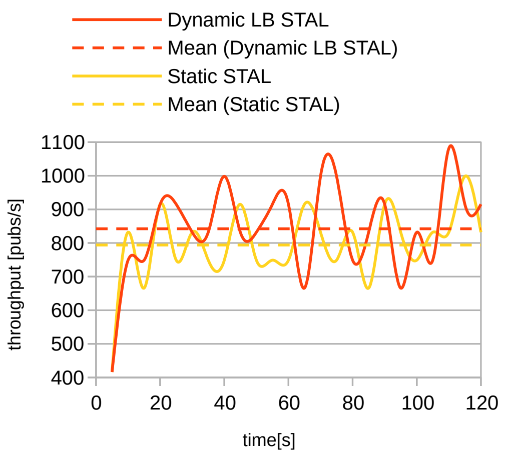
(**c**) RIsPS pipeline



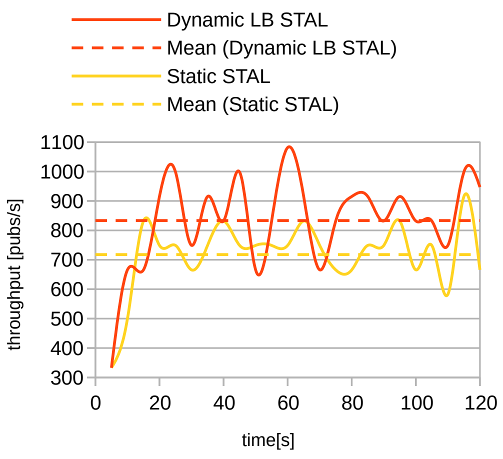
(**d**) RIhPS pipeline

Figure 5. Average latency against skew coefficient for different modes and pipelines.

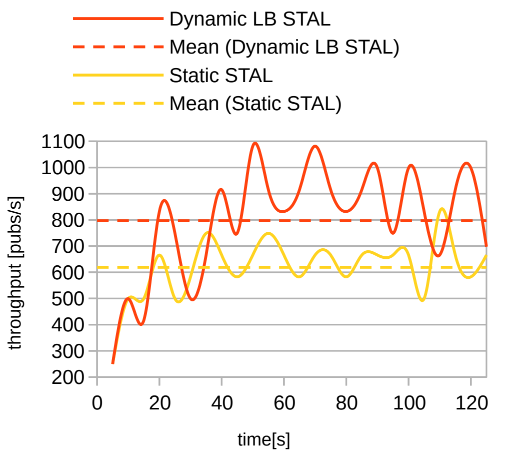
While Figure 4 shows the average throughput as aggregated characteristics, Figure 6 shows how the throughput varied over time. The current throughput was shown by the solid line, while the average throughput was shown by the dashed line. We show the behavior of the sPS pipeline only because other pipelines and other modes followed a quite similar pattern. As expected, the throughput oscillated, during the experiment, around the average value. Again, we see that the difference in throughput increased with the increasing skew coefficient.



(**a**) 0.3 skew coefficient



(**b**) 0.5 skew coefficient



(**c**) 0.7 skew coefficient

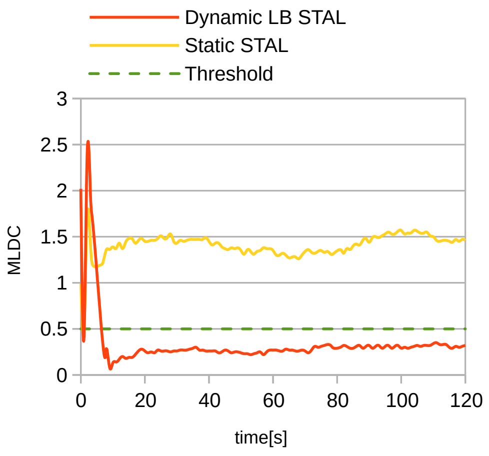
Figure 6. Throughput against time for different modes of the sPS pipeline.

Figure 7 shows how the Maximum Load Distance Coefficient (MLDC) value changed over time, during the experiment, for the *Dynamic LB STAL* and baseline *Static STAL* modes in the sPS pipeline. The MLDC values are represented by the solid line in the graph, while the dashed line indicates the threshold, which was a condition that had to be satisfied to initiate partition redistribution, as specified in Algorithm 1. As in Figure 6, we do not show analog figures for other pipelines and other modes, as they were quite similar.

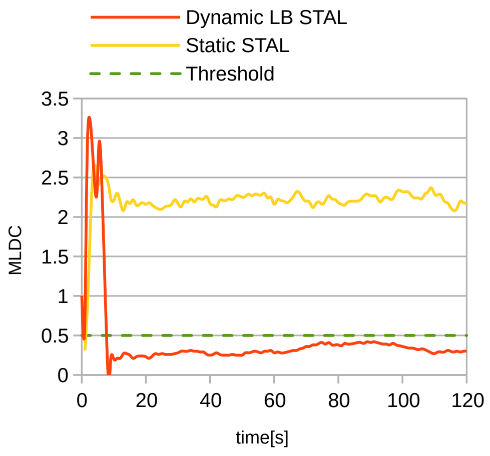
We can see that the MLDC value for the *Static STAL* mode remained stable over time. This was expected, as this mode did not dynamically rebalance the load, unlike the *Dynamic LB STAL* mode. We can see that the MLDC value increased as the skew coefficient value increased, because the higher partition skew led to a larger MLDC value. In contrast to the *Static STAL* mode, the *Dynamic LB STAL* mode reacted promptly to the identified load imbalance, which occurred when the MLDC value exceeded the threshold (and when other conditions were satisfied, as explained in Section 3.2), and then initiated the partition redistribution, to achieve a more balanced load distribution, which led to a reduced MLDC value.

We can also see that the MLDC value for the *Dynamic LB STAL* mode stabilized below the threshold for the rest of the experiment, suggesting that only one partition redistribution was required during the entire experiment. The spike at the beginning was due to the initial calculation of the MLDC value, and thus occurred in both modes. Moreover, because the actual distribution of data stream objects during the experiment varied slightly from the average distribution, the MLDC value for both modes was not completely stabilized, but rather oscillated, to some extent. We conclude that the load-balancing mechanism implemented by the *Dynamic LB STAL* mode effectively manages load imbalances.

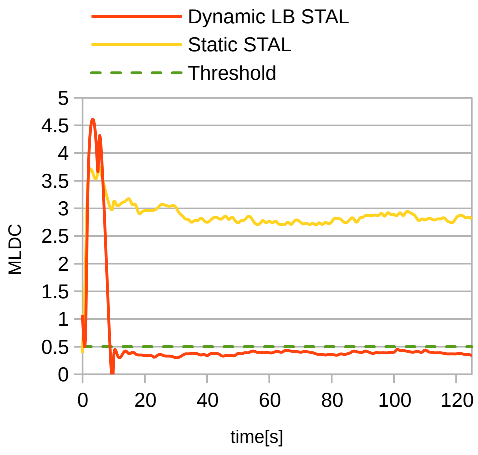
Finally, in Figure 8, we show how long it took for the *Dynamic LB STAL* mode to finish the redistribution of static data (i.e., subscriptions), depending on their size, for the sPS pipeline and skew coefficient of 0.7. As previously explained, we forced the rebalancing, by the redistribution of static data. As expected, we found that the redistribution time increased with the size of the static data; however, from the previous figures, we know that the gain in higher throughput and lower latency outweighed the loss. Additionally, we can see that the increase in the number of concurrent jobs, which is a Spark parameter for better utilization of worker resources, did not have an influence on the redistribution time. We conclude that it is beneficial to perform the redistribution of static data for long-running jobs, such as stream processing pipelines, but that the impact of the overhead should be taken into account in the case of huge quantities of static data.



(**a**) 0.3 skew coefficient



(**b**) 0.5 skew coefficient



(**c**) 0.7 skew coefficient

Figure 7. Maximum Load Distance Coefficient (MLDC) against time for different modes of the sPS pipeline.

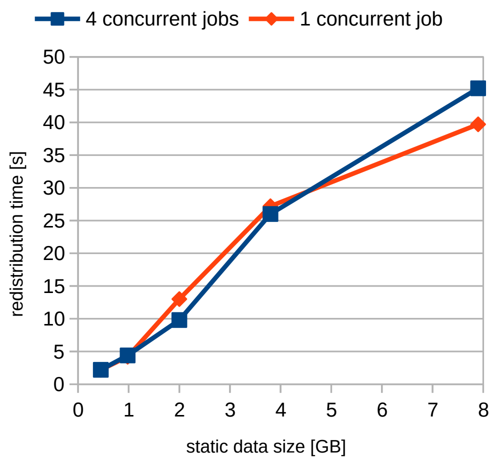


Figure 8. Redistribution time against static data size for the sPS pipeline.

5. Conclusions and Future Work

This paper addresses the problem of load imbalance in stream-static joins, due to partition skew, where streaming data must be joined in real time with static data partitioned among distributed workers. To address this, the paper proposes a solution that dynamically balances the processing load, and reduces network traffic by ensuring the optimal locality of static and streaming data, considering the load distribution during the join and following operations. This solution offers different modes, depending on (1) where the join is performed, (2) how imbalance is detected, and (3) whether the solution adapts to detected imbalances. Our approach is designed to be flexible and applicable to any streaming platform. As detailed in this paper, we evaluated the solution, using four different stream processing pipelines in Apache Spark, and compared it to the baseline approach from [6] and a similar approach in [23], which used a modified partitioning method, whereas the proposed solution works with built-in methods. The contributions of the paper include studying the problem of load balancing in stream-static joins, proposing a dynamic load balancing solution, and experimentally evaluating it in Apache Spark.

Our experimental evaluation showed that the *Dynamic LB STAL* mode effectively managed load imbalances, and performed best for all the evaluated stream processing pipelines, by using dynamic partitioning (*Dynamic*), performing stream-static join on nodes with static data (*STAL*), and balancing the load on the workers (*LB*). The *LB* modes outperformed the DB modes for all the pipelines, because balancing the processing load per worker was a more effective approach than balancing the number of streaming items, as was done in [23]. The *STAL* modes performed better than the *STRL* modes, which performed stream-static join on nodes with streaming data, for all the pipelines, due to the cost of continuously migrating larger static data. The *Dynamic LB STAL* mode had the lowest latency, except for the RIsPS pipeline, in the case of a small skew coefficient where unnecessary rebalancing increased the latency. The *LB* modes generally had lower latency than the *DB* modes, and the *STAL* modes generally had lower latency than the *STRL* modes for all the pipelines.

In future work, we plan to extend the applicability of our approach, to design a universal load balancing solution for data stream processing systems. Furthermore, we intend to investigate the potential of our approach for stateful data stream processing, which is a more challenging use case, in which items share states among themselves.

Author Contributions: Conceptualization, J.M., K.P., M.A. and D.Š.; methodology, J.M., K.P. and M.A.; software, J.M.; validation, J.M., K.P., M.A. and D.Š.; formal analysis, J.M., K.P., M.A. and D.Š.; investigation, J.M.; resources, K.P.; data curation, K.P.; writing—original draft preparation, J.M. and K.P.; writing—review and editing, J.M., K.P., M.A. and D.Š.; visualization, J.M.; supervision,

K.P.; project administration, K.P.; funding acquisition, K.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by the European Regional Development Fund under grant KK.01.1.1.01.0009 (DATACROSS), which included the salary of a PhD student, and reimbursement for attending scientific conferences. This work was also supported in part by the Croatian Science Foundation, under project UIP-2017-05-9066, which included the salary of a PhD student, the cost of the equipment on which our experiments were performed, and reimbursement for attending scientific conferences.

Data Availability Statement: The datasets used in our experimental evaluation are available in public repositories [31,32].

Acknowledgments: The authors would like to thank Ivan Livaja for developing subscription partitioning strategies from [6] during his research studies, as we used stream processing pipelines from these strategies in the experimental evaluation of our solution.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Golab, L.; Özsu, M.T. Issues in data stream management. *ACM Sigmod Rec.* **2003**, *32*, 5–14. [CrossRef]
2. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.* **2015**, *36*, 28–38.
3. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
4. Iqbal, M.H.; Soomro, T.R. Big data analysis: Apache storm perspective. *Int. J. Comput. Trends Technol.* **2015**, *19*, 9–14. [CrossRef]
5. Isah, H.; Abughofa, T.; Mahfuz, S.; Ajerla, D.; Zulkernine, F.; Khan, S. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access* **2019**, *7*, 154300–154316. [CrossRef]
6. Livaja, I.; Pripužić, K.; Sovilj, S.; Vuković, M. A distributed geospatial publish/subscribe system on Apache Spark. *Future Gener. Comput. Syst.* **2022**, *132*, 282–298. [CrossRef]
7. Irandoost, M.A.; Rahmani, A.M.; Setayeshi, S. MapReduce data skewness handling: A systematic literature review. *Int. J. Parallel Program.* **2019**, *47*, 907–950. [CrossRef]
8. Ramakrishnan, S.R.; Swart, G.; Urmanov, A. Balancing reducer skew in MapReduce workloads using progressive sampling. In Proceedings of the Third ACM Symposium on Cloud Computing, San Jose, CA, USA, 14–17 October 2012; pp. 1–14.
9. Chen, Q.; Yao, J.; Xiao, Z. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *26*, 2520–2533. [CrossRef]
10. Dhawalia, P.; Kailasam, S.; Janakiram, D. Chisel++ handling partitioning skew in MapReduce framework using efficient range partitioning technique. In Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing, Vancouver, BC, Canada, 23–27 June 2014; pp. 21–28.
11. Gao, Y.; Zhang, Y.; Wang, H.; Li, J.; Gao, H. A distributed load balance algorithm of MapReduce for data quality detection. In *Database Systems for Advanced Applications—DASFAA 2016 International Workshops: BDMS, BDQM, MoI, and SeCoP*; Springer: Cham, Switzerland, 2016; pp. 294–306.
12. Myung, J.; Shim, J.; Yeon, J.; Lee, S.G. Handling data skew in join algorithms using MapReduce. *Expert Syst. Appl.* **2016**, *51*, 286–299. [CrossRef]
13. Liroz-Gistau, M.; Akbarinia, R.; Agrawal, D.; Valduriez, P. FP-Hadoop: Efficient processing of skewed MapReduce jobs. *Inf. Syst.* **2016**, *60*, 69–84. [CrossRef]
14. Shah, M.A.; Hellerstein, J.M.; Chandrasekaran, S.; Franklin, M.J. Flux: An adaptive partitioning operator for continuous query systems. In Proceedings of the 19th International Conference on Data Engineering (Cat. No. 03CH37405), Bangalore, India, 5–8 March 2003; pp. 25–36.
15. Zhao, X.; Zhang, J.; Qin, X. *k* NN-DP: Handling Data Skewness in *kNN* Joins Using MapReduce. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *29*, 600–613. [CrossRef]
16. Gavagsaz, E.; Rezaee, A.; Haj Seyyed Javadi, H. Load balancing in join algorithms for skewed data in MapReduce systems. *J. Supercomput.* **2019**, *75*, 228–254. [CrossRef]
17. DeWitt, D.J.; Naughton, J.F.; Schneider, D.A.; Seshadri, S. Practical Skew Handling in Parallel Joins. In Proceedings of the 18th International Conference on Very Large Data Bases Madison, Vancouver, BC, Canada, 23–27 August 1992; pp. 27–40.
18. Zhang, X.; Chen, H.; Hu, F. Back Propagation Grouping: Load Balancing at Global Scale When Sources Are Skewed. In Proceedings of the 2017 IEEE International Conference on Services Computing (SCC), Honolulu, HI, USA, 25–30 June 2017; pp. 426–433.
19. Nasir, M.A.U.; Morales, G.D.F.; Garcia-Soriano, D.; Kourtellis, N.; Serafini, M. The power of both choices: Practical load balancing for distributed stream processing engines. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Republic of Korea, 13–17 April 2015; pp. 137–148.

20. Lv, W.; Tang, Z.; Li, K.; Li, K. An Adaptive Partition Method for Handling Skew in Spark Applications. In Proceedings of the 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), Guangzhou, China, 8–12 October 2018; pp. 1063–1070.
21. Cardellini, V.; Nardelli, M.; Luzi, D. Elastic stateful stream processing in storm. In Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS), Innsbruck, Austria, 18–22 July 2016; pp. 583–590.
22. Tang, Z.; Zhang, X.; Li, K.; Li, K. An intermediate data placement algorithm for load balancing in spark computing environment. *Future Gener. Comput. Syst.* **2018**, *78*, 287–301. [CrossRef]
23. Liu, G.; Zhu, X.; Wang, J.; Guo, D.; Bao, W.; Guo, H. SP-Partitioner: A novel partition method to handle intermediate data skew in spark streaming. *Future Gener. Comput. Syst.* **2018**, *86*, 1054–1063. [CrossRef]
24. Chou, T.; Abraham, J. Load Balancing in Distributed Systems. *IEEE Trans. Softw. Eng.* **1982**, *SE-8*, 401–412. [CrossRef]
25. Jiang, Y.C.; Jiang, J. A multi-agent coordination model for the variation of underlying network topology. *Expert Syst. Appl.* **2005**, *29*, 372–382. [CrossRef]
26. Jiang, Y. A survey of task allocation and load balancing in distributed systems. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *27*, 585–599. [CrossRef]
27. Khan, A.; Attique, M.; Kim, Y. iStore: Towards the optimization of federation file systems. *IEEE Access* **2019**, *7*, 65652–65666. [CrossRef]
28. Madsen, K.G.S.; Zhou, Y.; Cao, J. Integrative dynamic reconfiguration in a parallel stream processing engine. In Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE), San Diego, CA, USA, 19–22 April 2017; pp. 227–230.
29. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. *HotCloud* **2010**, *10*, 95.
30. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache Hadoop YARN: Yet another resource negotiator. In Proceedings of the SoCC: ACM Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16.
31. UK Car Accidents 2005–2015. 2021. Available online: https://www.kaggle.com/silicon99/dft-accident-data/ (accessed on 25 February 2021).
32. Pope, A. GB Postcode Area, Sector, District, [Dataset]. University of Edinburgh. 2017. Available online: https://doi.org/10.7488/ds/1947 (accessed on 20 February 2021).
33. Kassab, A.; Liang, S.; Gao, Y. Real-time notification and improved situational awareness in fire emergencies using geospatial-based publish/subscribe. *Int. J. Appl. Earth. Obs. Geoinf.* **2010**, *12*, 431–438. [CrossRef]