Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems

Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang,
State Key Laboratory for Novel Software Technology, Nanjing University

https://www.usenix.org/conference/atc22/presentation/gu-rong

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems

Rong Gu   Han Yin   Weichang Zhong   Chunfeng Yuan   Yihua Huang
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Abstract

Stateful distributed stream processing engines (SPEs) usually call for dynamic rescaling due to varying workloads. However, existing state migration approaches suffer from latency spikes, or high resource usage, or major disruptions as they ignore the order of state migration during rescaling. This paper reveals the importance of state migration order to the latency performance in SPEs. Based on that, we propose Meces, an on-the-fly state migration mechanism which prioritizes the state migration of hot keys (those being processed or about to be processed by downstream operator tasks) to achieve smooth rescaling. Meces leverages a fetch-on-demand design which migrates operator states at record-granularity for state consistency. We further devise a hierarchical state data structure and gradual strategy for migration efficiency. Meces is implemented on Apache Flink and evaluated with diversified benchmarks and scenarios. Compared to state-of-the-art approaches, Meces improves stream processing performance in terms of latency and throughput during rescaling by orders of magnitude, with negligible overhead and no disruption to non-rescaling periods.

1 Introduction

In recent years, stateful Stream Processing Engines (SPEs) [16, 19, 33, 44, 53] have been widely adopted because of the increasing demands for complicated analytics over real-time data, e.g. fraud detection, log monitoring, sentiment analysis, and IoT applications [11, 13, 51, 60].

SPEs are expected to be long-running and always have low latency performance [21, 22]. It commonly requires SPEs to perform dynamic rescaling in the face of unpredictable circumstances (e.g. data rate fluctuations, machine failures, processing stragglers). However, as the processing operators in SPE are usually stateful and partitioned across workers, rescaling them calls for state migration, which means moving state data between workers, even across networks [26].

The problem of state migration in SPEs is fundamental and challenging. Prior major advances made in the last decades

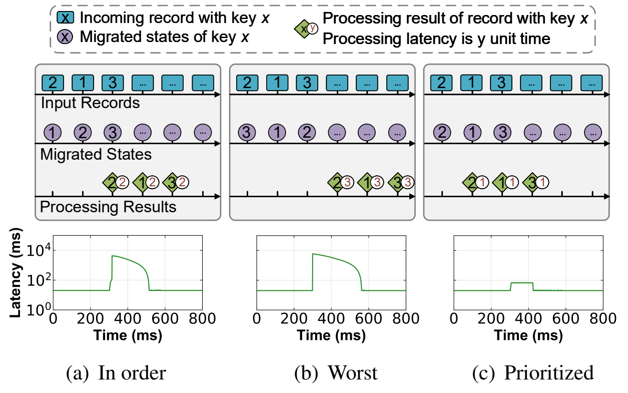


Figure 1: Impact of migration order on processing latency. The Y-axis indicates event-time latency, i.e., interval between a record's event-time and its emission-time from the output operator [30]. Point (X,Y) means the average processing latency of the records generated at Xms is Yms.

can be classified into four categories: (1) Full-Restart approach [16,33,53] pauses and resumes the whole task when redistributing states. (2) Partial-Pause approach [14, 52] restarts a subgraph instead of the whole job topology to reduce execution blocking. (3) Replicated-Dataflow approach [47, 61] executes a new dataflow in parallel with the old one until finishing the state migration. (4) Proactive approach [26, 43] adds extra behaviour to non-rescaling stream processing periods to relieve the pressure during state migration.

Unfortunately, prior approaches suffer from processing latency spikes, or high resource usage [26], or major disruption (see Section 5.6). Their common limitation lies in **order-unaware** state migration, i.e., not taking into account the order in which operator state migrates. Figure 1 illustrates how the migration order affects SPE latency performance during rescaling. We take a representative stateful stream processing job, key-count, as an example, where records with random keys come from upstream and are processed by the SPE in a FIFO manner. In this case, streaming operators store the count value of each key as their corresponding states. During rescaling, an affected operator needs to receive the migrated state (the global count values of keys) before it can process the corresponding incoming records.

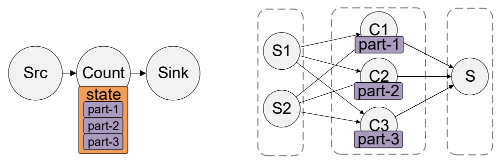
If the states are migrated in an order-unaware manner as in Figure 1(a), the first coming record may not be processed in time, because it needs to wait for the arrival of its corresponding state. This also blocks subsequent records in the queue due to FIFO processing, eventually accumulating the processing latency for all records over a period of time. In the worst case as shown in Figure 1(b), the state of the first record is the last one to be migrated, making all record processing blocked before the state migration is finished. Ideally, as shown in Figure 1(c), states are migrated in exactly the same order as the records arrive, therefore minimal time is spent in the waiting queue and the latency can be greatly reduced.

Based on this observation, we find that the state of hot keys (those being processed or about to be processed by downstream operator tasks) needs to be prioritized so that the stream processing proceeds without blocking. In this paper, we propose Meces, an on-the-fly rescaling mechanism which enables **prioritized state migration** for SPEs. In fact, it is challenging to dynamically adjust the state migration order according to the incoming records during SPE rescaling. To achieve this, Meces leverages a fetch-on-demand state accessing model, based on the fact that modern SPEs [16, 53] co-partition data records with stream operators to the same key space. During rescaling, the states can be actively *fetched* by the SPE operator instances when receiving a data record whose state is not local. In this way, the operators prioritize the transmission of those currently needed states to generate processing results with low latency.

Another challenge is to handle the state consistency during the prioritized state migration process. In Meces, the state consistency in the above process is maintained by a control messaging based coordination protocol, inspired by [7, 40, 43]. In addition, as the fetch-on-demand model used in prioritized state migration calls for light-weight state accessing, we devise a hierarchical data organization and adopt a gradual migration strategy for finer-grained state transfer like [26]. Meces is designed as a pluggable rescaling module without affecting non-rescaling periods. As far as we know, Meces is the first stateful SPE rescaling approach that enables prioritized state migration, which can reduce the latency spikes without high resource usage or incurring major disruption.

To sum up, this paper makes the following contributions:

- We propose an on-the-fly rescaling mechanism, called Meces, for stateful distributed stream processing engines. It prioritizes the migration of hot states to achieve low-latency and resource-efficient rescaling.
- In addition, we adopt a control messaging based coordination protocol to maintain state consistency during prioritized state migration. We further devise a hierarchical state data organization and a gradual state migration strategy, which lowers data transmission granularity.
- Meces is implemented on the widely-used SPE Apache Flink, requiring minimal user code modification and no disruption to non-rescaling periods.
- We validate our design with various workloads. Compared with state-of-the-art approaches, Meces reduces nearly 95% of processing latency peak during rescaling.



(a) Streaming operators (b) Parallel instances

Figure 2: A key-count job example in stream processing

2 Background

In this section, we first introduce basic concepts and terminology of stateful stream processing in Section 2.1, and then review and analyze the design and shortcomings of existing rescaling approaches in Section 2.2.

2.1 Stateful Stream Processing

Stream Processing Topology: In scale-out distributed SPEs, the computation tasks are expressed as directed graphs (Figure 2(a)), where vertices represent stream processing operators. Each operator receives data from its upstream operators, processes data, and sends output to its downstream operators. Operators without upstream are called *source* operators, and those without downstream are called *sink* operators.

SPEs process input data in a data-parallel style by mapping the streaming *operators* to multiple parallel *instances*. The number of an operator's instances is called its degree of *parallelism*. In Figure 2(b), the vertices represent parallel instances and the directed edges represent data channels.

Stateful Operator: In practice, many analysis tasks require operators to compute their output based on both current and previously received data. Examples of such tasks include data aggregation, window computation, ML models, etc. To realize this, *stateful* operators maintain internal and mutable states. Operator states remember information about past input and can be used for processing of future input [18]. For example, a counting operator of a WordCount job stores the current occurrences of each word as its states.

A stateful operator receives input data as a stream of keyed *records*. Each record can be denoted as a pair (k, v) representing the key and the payload value. Correspondingly, the stateful operator also holds its state S as a set of key-value pairs and divides the set into several disjoint partitions. During processing, the partition S_I is assigned to an operator instance I. When I receives a record with key k, it processes the record by reading or updating the value v corresponding to k in S_I. In modern SPEs [16,53], states of the stateful operators are co-partitioned with the parallel instances. For example, in Figure 2(b) the state of the keys mapped to C_1 operator instance is only stored in *part-1*.

Checkpoints: SPEs usually achieve fault-tolerance via checkpoints. A checkpoint marks the persistent state for each operator at a specific time point. SPEs can resume from failure by restoring the operator states with a recent checkpoint and replaying the records since the generation of the checkpoint.

A common way to realize consistent state snapshots of distributed streaming operators is Chandy-Lamport algorithm [10] or its variants. The checkpoint algorithm in Flink [7] works by injecting special streaming records called barriers into the pipeline. Each operator instance will persist its state when receiving a barrier. This workflow makes a globally consistent operator state snapshot on the basis of reliable FIFO data channels.

2.2 Prior Rescaling Mechanisms

The critical difficulty in on-the-fly rescaling distributed stream operators is efficiently migrating states among instances while keeping the exactly-once semantics. Prior approaches can mainly be categorized into the following strategies [6,26].

Full-Restart: This approach halts the SPE execution, takes a state snapshot of all operators, redistributes the state among operator instances, and restarts the job execution until the state redistribution is complete. Since it is simple and naturally guarantees consistency using existing *checkpoint* mechanisms, it is adopted by many SPEs including Spark Streaming [53], Structured Streaming [5], Apache Flink [16], etc. However, this mechanism halts the whole pipeline and causes serious latency spikes during rescaling.

Partial-Pause: This approach only stalls the streaming operators that need state migration during rescaling. It mitigates the latency spikes by narrowing the interruption from the job-granularity to the operator-granularity. This mechanism is first introduced by Flux [52], and adopted by SEEP [14], FUGU [25], Chi [40], etc. However, if the affected operator is a critical component in the topology, this approach still pauses the entire pipeline during rescaling.

Replicated-Dataflow: This approach replicates the affected operators and executes the old and new configurations simultaneously until the migration completes. It minimizes processing latency and realizes on-the-fly state migration, but requires redundant computing resources for replication [26]. It also calls for additional de-duplication mechanisms as the concurrent execution generates duplicated data. This mechanism is adopted by ChronoStream [61], Gloss [47], etc.

Proactive: Proactive approach adds extra behaviour to non-rescaling periods to relieve the pressure when migrating states. Megaphone [26] works by splitting operators and embedding the migration flows into data flows. However, it introduces partitioning overhead inside the original processing operators during non-rescaling periods. Besides, in system design it calls for extra coordination and progress tracking mechanisms [26,43] which are not directly supported by many modern SPEs [16,53]. Rhino [43] periodically replicates operator states among all workers. It facilitates both fault-tolerance and state migration of extremely large states, but incurs extra network overhead to regular stream processing.

As far as we know, none of the existing approaches consider the state migration problem from the aspect of the state migration order. They rescale with latency spikes or high resource usage [26], or major disruption (see Section 5.6).

3 System Design

In this section, we first introduce the main idea of Meces in Section 3.1. Then, in Section 3.2 we describe the design in detail. Finally, we elaborate the optimization for finer-grained state transfer in Section 3.3 and Section 3.4.

3.1 Prioritized State Migration

If not introducing huge redundant resources [47,61] or disrupting regular processing [26,43], prior works [5,14,16,25,40,52,53] fail to achieve low latency when rescaling. They are mainly limited by "order-unaware" state migration. As Figure 1 in Section 1 shows, the migration for hot keys should be prioritized, so as to generate timely results and reduce queuing latency when rescaling.

In order to achieve prioritized state migration, we review the nature of states in streaming operators. We denote a record with key k as r_k, and a state value with key k as v_k. An important fact about recent scale-out stateful SPEs is that the operator states are partitioned across the operator instances in exactly the same way as data records according to their keys. That is to say, when an operator instance I processes a record r_k, the only state that it needs to access is v_k. It is also guaranteed that no common states need to be accessed when processing records with different data keys.

Given this property, we propose the prioritized state migration mechanism, which enables the system rescaling and data processing of SPEs to efficiently run at the same time. Specifically, when the SPE triggers an online state repartition, the previously responsible instances **send** states in batches to the newly responsible instances. In our design, as for the newly responsible instances, instead of only passively waiting for the arrival of migrated state batches, they can also actively **fetch** states from previously responsible instances, to get the individual states corresponding to the data records for timely processing. For example, as no more records with key in $\{k_1,k_2,k_3\}$ will be sent to I_a due to rescaling, the SPE decides to migrate a set of state values $S_m = \{v_{k_1}, v_{k_2}, v_{k_3}\}$

from operator instance I_a to I_b. Then S_m is sent by I_a. Before I_b receives the entire state set, if I_b encounters a record r_{k_2}, it immediately performs a single-value fetch for v_{k_2}. This lightweight operation helps I_b to generate processing results in time, instead of getting blocked until receiving the entire S_m from I_a.

In this way, we can keep the stream processing operators working during SPE rescaling. The batched "send" aims at quick state migration, while the active "fetch" ensures in-time processing for the records that requires a remote state. The processing latency performance will be affected only when an active "fetch" is triggered. In that case, only the processing of that single record is delayed a bit because of one extra state fetch operation, but the queuing cost of subsequent records can be greatly reduced. In other words, the performance interference caused by the state migration comes down to the record-granularity, so that we can keep the stream processing performance as high as possible during the rescaling period.

To achieve this, two obstacles need to be carefully dealt with: First, how to ensure state consistency when transferring states among operator instances in a dynamic order (Section 3.2). Second, how to minimize the performance impact brought by fetch operations (Section 3.3 and 3.4).

3.2 Fetch-on-demand State Accessing Protocol

To support prioritized state migration with dynamic order during rescaling, Meces leverages a fetch-on-demand state accessing protocol. The state consistency of the fetch-on-demand model is based on a control messaging coordination protocol, inspired by [7, 40, 43]. In the following, we first briefly describe how the protocol works, then use an example to further explain its process.

Migration Process: We call the time period from the beginning to the end of a state redistribution as a *Migration* stage. The global controller of an SPE starts a *Migration* stage by injecting a special data record called *control message* into the source operators. The message then travels through the whole pipeline in the same way as a regular data record. Once an instance I receives a control message from its input data channels, it performs the following steps:

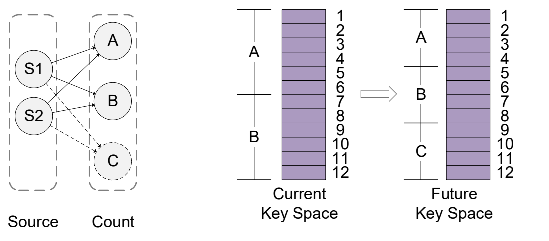
(1) I sends the control message downstream.

(2) If the downstream operator needs to migrate states, I updates its routing strategy.

(3) If I itself needs to migrate states, according to whether it has received messages from all input channels, it successively goes through two phases: *Aligning* and *Aligned*.

(4) I sends a confirming signal to the global controller. A *Migration* stage ends when the global controller receives confirming signals from all parallel instances.

Figure 3 and Figure 4 show the *Migration* stage during a scale-out operation of a streaming key-count job. The degree of parallelism of the count operator increases from 2 to 3, represented by A, B, C (see Figure 3(a)). The upstream source



(a) Instances rescaling (b) Key space redistribution

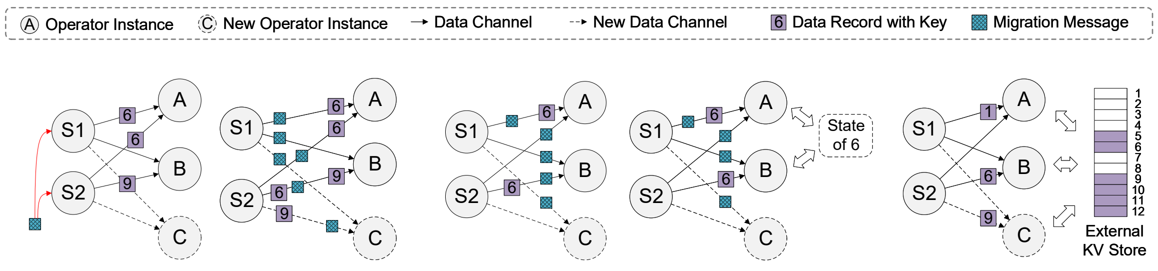
Figure 3: Example of a rescaling stream processing job

operator has two parallel instances $S1, S2$. For simplicity, we assume the keys are between 1 and 12. Therefore, based on the uniform distribution, the distribution of the key space before and after the rescaling is as shown in Figure 3(b).

When $S1$ or $S2$ receives the control messages, it updates the routing strategy and outputs the subsequent data records in accordance with the new topology. As its downstream operator's degree of parallelism will increase by 1, its new strategy divides the key space into three parts equally. As shown in Figure 4(a), the records mapped to "9" were previously sent to B, but will be shipped to the new instance C from now on. Meanwhile, the records with key "6", which were consumed by A, should then be in the charge of B.

As for the count operator, in general, the states are sent from the previously responsible instances, but the newly responsible instances also actively fetch states in response to the incoming data records. We denote an instance's current key space before migration as C_k, and its future key space after migration as F_k. Specifically, the count operator successively goes through two phases. Taking B as an example:

1. **Aligning (Figure 4(b)):** When B first receives a control message, such as from $S2$, it can foresee the arrival of keys that did not belong to it before. After that, when B encounters a record whose key is not in its C_k, such as key "6", instead of considering it as an error, B first checks its F_k. If the key is found, B "borrows" the corresponding state of this key from other count operator instances to complete the processing. Note that in this phase, the message from $S1$ has not reached B, which means A may still have to deal with records with the "6" key. Because of that, B should also be prepared that its state of "6" can still be borrowed back by others.
2. **Aligned (Figure 4(c)):** The *Migration* for B is aligned once it receives control messages from all of its input channels. In this situation, it is guaranteed that all future records with the "5, 6" key are shipped only to B, and B will no longer receive records with keys from 9 to 12. Therefore, B can start its state migration. It checks C_k and F_k, sends the states between 9 and 12 to other instances, and fetches all the states in 5 and 6 asynchronously. When finishing sending and fetching, B sends the completion signal of its *Migration* stage.



(a) Triggering controlling messages (b) Aligning phase (c) Aligned phase

Figure 4: *Migration* stage in a rescaling operation

In the above process, the stream processing operators keep functioning without explicitly blocking the input channels. The impact of state migration on the task is only perceived when an instance requires a remote state. The processing latency of this single record is then only increased by the fetching cost of a single state. For subsequent records, if the corresponding state has already been fetched, the processing of these records suffers from neither migration cost nor huge queuing cost. In this way, the influence of rescaling can be significantly reduced, therefore the system can avoid sudden and severe performance degradation in latency and throughput. For implementation details of the active state fetch process, please refer to Section 4.2.

State Consistency: In stream processing, the global state consistency usually refers to the exactly-once semantics. In Meces, at a *Migration* stage where the state k is migrated from I_1 to I_2, each incoming record affects the final results exactly once. Let t_1 be the timestamp when I_1 or I_2 receives the first control message of *Migration*, and t_2 be the timestamp when both of I_1 and I_2 have received control messages from all the input channels. Therefore, the *Aligning* phase begins at t_1. In this phase, the data records of k can be sent to both I_1 and I_2, but actually only one instance holds the state of k locally at a time. That is to say, only one instance can modify this state at a time. As the data record must be processed exactly once, and the state can only be flushed and "borrowed" after the processing of the current record is finished, the semantic is kept in $[t_1, t_2]$. After t_2, subsequent data records are sent to I_2 only. I_2 only needs to borrow at most once to transfer the state to the local and process each data record exactly once until its *Aligned* phase is complete.

3.3 Hierarchical State Data Organization

This subsection proposes an adaptive state data organization, which keeps regular stream running at a coarse granularity to avoid extra overhead, and performs prioritized state migration at a fine granularity to reduce the impact on latency performance of streaming operators.

Since the states in SPEs are key-value data and it is common to have billions of keys in a real-world streaming dataflow services, managing each key individually can be unrealistic. Many existing SPEs divide the states into key-groups [17], which are disjoint subsets of the entire key space.

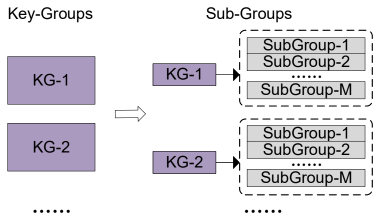


Figure 5: Split key-groups into sub-groups

However, the shortcoming of this flat index structure is that the state migration is also conducted on the key-group granularity. As the number of the operator states accumulates, the size of one single key-group could grow large. This can make the active "fetch" become time-consuming, bringing long delays to the record processing at the *Migration* stage. A naive solution is to increase the number of key-groups, but this is not practical in many SPEs. A vast number of key-groups will bring a lot of additional metadata management overhead and fragmented read/write of checkpoints, thereby reducing the performance of non-rescaling stream processing.

To address this issue, we introduce the nested layer of state data organization in Meces. Instead of using a single-layer map, we further divide each key-group into multiple sub-groups as illustrated in Figure 5. When encountering a record that requires a remote state, an instance tries to fetch the corresponding sub-group of the record key instead of the entire key-group. This reduces the time overhead used to obtain data that is not needed immediately.

Note that, key-groups are distributed among different operator instances and the metadata should be stored, indicating which instances are responsible for each key-group. When the number of key-groups increases, it incurs much more metadata management cost and record distribution cost. Differently, sub-groups of the same key-group must belong to the same operator instance when the system is not rescal-

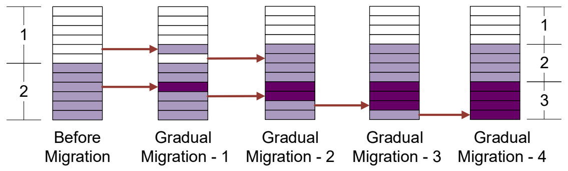


Figure 6: An example of gradual state migration

ing. Therefore, increasing the number of sub-groups does not bring significant extra overhead to a steady dataflow. Users can choose an appropriate number of sub-groups to achieve smooth state migration. The appropriate settings are usually based on the maximum size of states and the expected maximum latency during rescaling.

3.4 Gradual State Migration

During a rescaling operation, a large part of the overall states may need transferring, even if the degree of parallelism does not change much. For example, to divide the states evenly across all operator instances in Figure 3, changing the degree of the parallelism from 2 to 3 causes half of the keys to be redistributed. For example, if we have an operator with 128 key-groups and change the degree of parallelism from 25 to 30, a nearly full state migration (115 out of 128 key-groups) has to be triggered. Migrating all of these states in a single batch can dramatically slow down the overall performance, because most of the records processed by the task in the next period may be affected by the fetch operations. As the data streams continue to flood in, lots of minor processing delays may accumulate into large latency spikes.

To resolve this issue, inspired by [26], we also achieve finer-grained migration via a gradual migration strategy in Meces, which splits the update into several micro-batches of state migration as shown in Figure 6. The *Migration* stage in one rescaling operation is then composed of multiple *Gradual-Migration* steps. At each step, the global controller decides which states should be relocated based on the user-defined size of micro-batches ($batch\_size$). For example, in Figure 6, $batch\_size$ is set to 1, which means that an instance can only dispose at most one key-group of states at a time. This splits the single *Migration* stage into four *Gradual-Fetch* steps.

At *Gradual-Fetch* steps, the information of migrated keys is included in the control messages and each upstream instance creates a temporary routing table indicating which downstream instance it should send records to. In this way, we affect only a tiny portion of the whole states at each *Gradual-Fetch* step, while most of the records can be processed normally. By changing *batch_size*, users can trade-off the lower latency spikes against higher migration throughput.

Note that during rescaling, the total number of migrated keys can be reduced by dividing the states differently. For example, in Figure 6, if the middle four keys are assigned to instance 3 and the last four keys are assigned to instance 2,

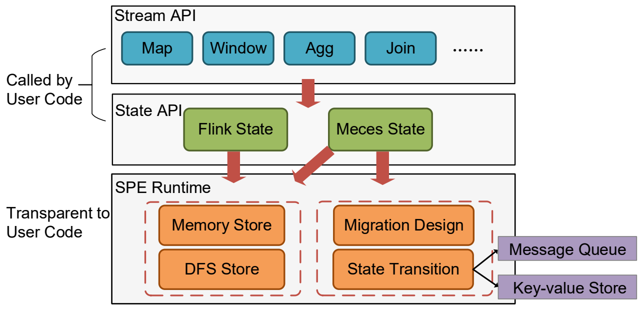


Figure 7: Meces Architecture

only 4 of 12 keys need relocating. Meces uses the uniform re-partition by default, but also supports custom partitioning, so that users can apply it to SPEs with sophisticated partitioning approaches [2, 29] such as consistent hashing. A brief evaluation of custom partitioning can be found in Section 5.3.

4 State Migration Implementation

This section describes the implementation of Meces. We demonstrate the overall system architecture of Meces in Section 4.1 and then introduce the details of state transition workflow among operator instances in Section 4.2. Finally, we discuss the fault tolerance mechanisms of Meces in Section 4.3.

4.1 System Architecture and Usage

The overall system architecture of Meces is shown in Figure 7, which consists of three layers:

1. Stream API: It provides basic operator functions for users to implement stream processing tasks.
2. State API: Meces provides Apache Flink-compatible APIs for operator functions to access their states. This enables the users to easily migrate the existing stream tasks from Flink to Meces.
3. SPE Runtime: This is where the system-level code is located. Meces basically reuses Flink's state backend module to store the key-value pairs in memory/DFS. Additionally, Meces implements the design in Section 3.

The underlying state management and migration in SPE Runtime are completely transparent to user codes. Therefore, it takes minimal effort to switch between Flink's and Meces's state implementations.

4.2 State Transition

The state transition among operator instances is of great importance to the performance of SPEs during rescaling. However, if the transfer of states happens directly between parallel instances, a mesh communication network has to be established among the worker machines. This would significantly increase the runtime overhead and make the maintenance of

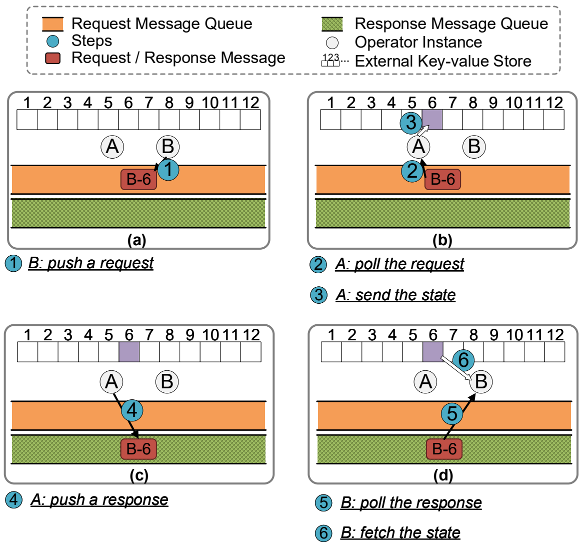


Figure 8: State transition process based on the pub-sub model

SPEs more complex. That is why SPEs like Flink [16] are designed to avoid direct inter-instance communication within an operator, i.e. adopting a shared-nothing architecture [17, 24] for low-cost, high-extensibility and high-availability.

To make it non-disruptive to SPEs, in implementation of Meces, we design a pub-sub model based approach to transfer states between parallel instances during record processing. The model leverages two message queues(*RequestMQ* and *ResponseMQ*), and an external key-value store (EKS) to provide a state transition service at *Migration* stage. At *Migration* stage, each instance performs light-weighted operations to read from these two message queues continuously.

Figure 8 demonstrates the state transfer process from operator instance A to B. When B needs to fetch the state with key "6", it simply pushes a message "B-6" to the *RequestMQ*, meaning "B is requesting 6". Later, A gets the message, finds "6" in its C_k, and then pushes the state value into the EKS and considers its local "6" state as borrowed. After that, A pushes a message into the *ResponseMQ*, indicating the completion of the request. Eventually, the message will be consumed by B, making B fetch the corresponding data from EKS. Thus far, the workflow of the state transition is complete.

In the above process, when the request message comes to A, if A happens to be processing a record with a key of 6, A will not start pushing until it finishes processing the current record. Meanwhile, if "6" has already been pushed into EKS and then A receives a record requiring this state, A should trigger a similar process to fetch the state, as "6" is now considered not held by A. These two situations do not bring much degradation to the rescaling performance, because they only happen in the *Aligning* phase, which only lasts for a short time if the system is not running under severe load imbalance.

For decoupling the components, many scalable message queue techniques [27, 46, 49] can be utilized to implement the pub-sub model. These are common components in real-world stream processing tasks, because they suit the stream processing paradigm as data sources and sinks. For EKS, a high-speed key-value store with fault-tolerance guarantees can be integrated to ensure real-time computation.

4.3 Fault Tolerance

Meces inherits fault-tolerance guarantees from the hosting SPEs, and relies on high-available message-queue/EKS service. Specifically, fault-tolerance is supported in both rescaling and non-rescaling scenarios.

During rescaling: If some of the message-queue/EKS nodes fail, the rescaling in Meces can keep going without data corruption. This is because message-queue/EKS service adopted by Meces is equipped with built-in fault-tolerance mechanisms [12,48], such as replication. If the entire message-queue/EKS or any Flink node fails to respond (simple/cascading failures), Meces considers this rescaling to have failed and invalidates the temporary data in message-queue/EKS. Then Flink's own failover mechanism is activated to recover the job from a checkpoint. Therefore, a failed rescaling does not break exactly-once semantics.

During non-rescaling: As Meces is designed to be non-disruptive when rescaling is not executed, it introduces no extra fault-tolerance issues. Any exception is handled by the hosting SPEs' failover mechanism.

Finally, Meces temporarily disables checkpoint generation only at Migration stages, and suspends new rescaling during an ongoing rescaling to avoid interference. Therefore, data consistency is kept throughout the job lifecycle.

5 Evaluation

This section presents an empirical evaluation of our prototype Meces. We focus on the following three questions:

1. What is the performance of Meces during rescaling with state migration? (Section 5.2 to 5.4)
2. How much overhead does Meces incur? (Section 5.5)
3. How does Meces compare to other state-of-the-art on-the-fly state migration approaches? (Section 5.6)

5.1 Experimental Setup

Hardware and Software: All experiments presented in this paper are conducted on a cluster of six computing nodes, each with two Intel Xeon E5-2620 v2 @2.10 GHz CPUs and 64 GB memory, running CentOS 7.9.2009.

The Meces prototype is implemented based on Apache Flink 1.12.0. The same version of Apache Flink is used as a baseline in our evaluation, referred to as "Native Flink" in the following. Meces and Native Flink run with Oracle JVM

11.0.10 to enable the low-latency garbage collector ZGC [63]. We pair Apache Kafka 2.7.0 with Apache ZooKeeper 3.6.0 as reliable message brokers with 25 partitions. Redis 3.2.0 is deployed to provide the external key-value store service globally. For persistent checkpoint storage, we deploy Apache Hadoop 2.9.2 to provide HDFS.

For SPE configurations, we configure Native Flink and Meces to use at most 25 GB heap memory. Each computing node can host at most 5 parallel instances of an operator. The total number of key-groups is set to 128, as it is the default value in Flink and different settings of this value do not differ the performance much in our cases.

Workloads and Metrics: We choose the NEXMark benchmark suite [55] and the key-count job as workloads. Nexmark models an online auction system and provides real-world streaming queries. The key-count job takes a stream of randomly generated keys as input and accumulates the number of times each key has occurred.

To provide input data for stream processing jobs, we implement open-loop stream generators which continuously and concurrently push random records into Kafka topics. Unless otherwise specified, all these open-loop generators produce data at a steady rate of 800K records/s. This input rate is near the saturation point of processing and large enough to show the performance difference between the various approaches.

We focus on two metrics in measurement during rescaling:

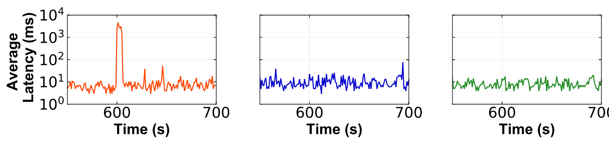
Latency: To evaluate the end-to-end latency of SPEs, we configure the stream generators to periodically insert marker events into Kafka. We denote latency as the time difference between these markers entering and leaving the SPE. For windowed operators, the markers simply bypass them to exclude the time spent in window buffers. The latency still grows when the system's processing rate cannot keep up with the production speed of upstream data, as the latency markers are queuing up. That is to say, the marker can still reflect the latency performance of the system with windowed operators.

Throughput: We define throughput of SPEs as the number of records output by the source operators per second. As the source operators are responsible for fetching records from Kafka, this reflects the capability of the system to read and process data from external data sources.

5.2 Latency Performance during Rescaling

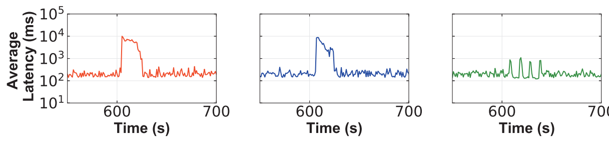
We first evaluate the latency performance of the SPEs during rescaling. The SPEs are initially configured with global parallelism of 25. Each job runs steadily for 600 seconds and rescales by increasing the parallelism of critical operators (e.g., join or window operators in NEXMark queries, counting operator in the key-count job) to 30. This causes 115 out of 128 key-groups to be relocated during rescaling.

We compare Meces with Native Flink (stopping the whole job when rescaling) and Order-Unaware (online block-based state migration without order prioritization). Figure 9~12 illustrate the end-to-end latency change of the representative queries in this process. For the performance evaluation results of all NEXMark queries, please refer to Appendix B.



(a) Native Flink (b) Order-Unaware (c) Meces

Figure 9: End-to-end latency of NEXMark Q1



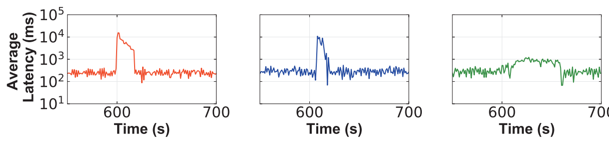
(a) Native Flink (b) Order-Unaware (c) Meces

Figure 10: End-to-end latency of NEXMark Q7

NEXMark Q1 performs currency conversion on a bid stream. This simple task maintains no states and the behavior is demonstrated as a basic case for our evaluation. As shown in Figure 9, both Meces and Order-Unaware reveal no latency peak but only system noise, because they incur no state migration cost during rescaling and operations can be done asynchronously. In contrast, Native Flink still needs to stop and restart the job even if there is no state to migrate.

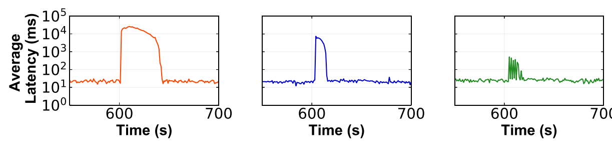
NEXMark Q7 and Q8 tests window operators. They perform tumbling window join of two streams, to find out the items with the highest price and the new users who just registered in the last period of time. They can maintain large states when the window size grows. We set the window sizes to be 10 seconds and 100 seconds. The performance comparison is illustrated in Figure 10 and Figure 11, where the latency peak of Meces during rescaling is an order of magnitude smaller than others. When the rescaling begins, Order-Unaware needs to block the currently processed records while migrating a considerable amount of states. As a result, Order-Unaware goes through a performance degradation near to Native Flink (Full-Restart), and reaches a latency peak of dozens of seconds. For comparison, Meces can give state migration priority to the hot keys being processed and smoothen latency.

Key-count takes a stream of randomly generated keys as input and reports the cumulative counts of each key continu-



(a) Native Flink (b) Order-Unaware (c) Meces

Figure 11: End-to-end latency of NEXMark Q8



(a) Native Flink (b) Order-Unaware (c) Meces

Figure 12: End-to-end latency of key-count

ously. The state size of the counting operator can also grow large when the key range is big enough. Besides, this query requires reading and updating the states when processing every single record. We run the job with 10^8 unique keys. As reported in Figure 12, both Native Flink and Order-Unaware show a latency peak which is three orders of magnitude higher than usual. The latency decreases gradually after the restarting or migration completes. As for Meces, it keeps the latency under 600 ms during the prioritized state migration stages.

In conclusion, during rescaling, Native Flink and SPEs with Order-Unaware can suddenly become out-of-service when migrating states, while Meces significantly lowers the maximum latency. The impact of prioritized migration of Meces is further evaluated in Section 5.3. Note that in some cases with a rather large size of states, Meces also have increased latency. This is mainly caused by the Garbage Collection behaviour of JVM, which is further analyzed in Section 5.5.

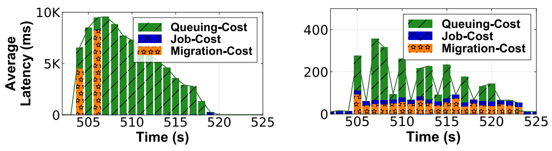
5.3 Impact of State Migration on Latency Performance during Rescaling

In this subsection, we evaluate how state migration affects the processing latency of SPEs, especially how the prioritized migration improves the performance.

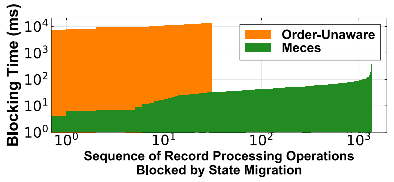
During a rescaling period involving state migration, the overall processing latency of SPEs can be generally divided into three parts: (1) Job-Cost: time to execute the processing logic. This is inevitable in both rescaling and non-rescaling periods; (2) Migration-Cost: when encountering a record whose state is not local, the task waits for the target states to arrive and then proceeds processing; (3) Queuing-Cost: If a record r is blocked due to migration cost, subsequent records may also be blocked in the queue as they cannot be processed until the processing of r is finished. Therefore the processing latency of these subsequent records is increased.

Figure 13 illustrates the various parts of the average processing latency of a certain operator instance per second during the rescaling of the key-count job. Note that in order to show the comparison between different parts, here we use linear axes instead of logarithmic axes and draw with different y-axis ranges in the upper two sub-figures.

As in Figure 13(a), Order-Unaware incurs high latency up to thousands of milliseconds, and the increased latency is composed of huge Migration-Cost and Queuing-Cost. Because Order-Unaware does not migrate the currently needed states



(a) Order-Unaware (b) Meces



(c) Distribution of Migration-Cost for all operator instances

Figure 13: Latency composition of during rescaling

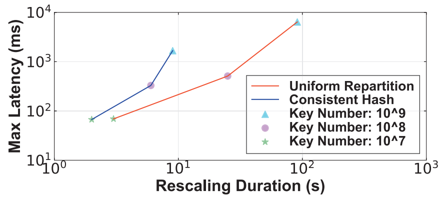
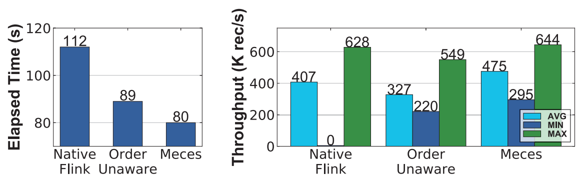


Figure 14: Rescaling performance of Meces using different repartition strategies

with priority, some of records have to wait for a long time before the arrival of their corresponding states. The waiting time can be up to several seconds as indicated by the Migration-Cost bar in the figure. As for the subsequent records, their states have been previously transferred and they need not wait for data migration. However, they suffer from huge Queuing-Cost due to the previous records being blocked. When no further migration is required, the Queuing-Cost decreases to 0 almost linearly.

Figure 13(b) reports the latency breakdown of Meces. Compared with Order-Unaware, the latency peak is greatly reduced to less than 400 ms. This is because when migrating states, Meces uses its fetch-on-demand model to give priority to the records which are currently being processed. When a record calls for a remote state, the operator instance immediately fetches the single state for the key of this record. Since the amount of data in a single state is very small, a fetch request can be quickly responded to. Consequently, as shown in Figure 13(c), the several long-duration execution blocks caused by Migration-Cost are converted into thousands of short-duration fetch operations. Although the sum of Migration-Cost does not differ much for two strategies, each record does not wait long time for its state to be migrated in the prioritized migration of Meces. More importantly, as the Migration-Cost of every single record is reduced, the Queuing-Cost for subsequent records is also significantly reduced. Eventually, the overall latency curve is flattened.

From another angle, Figure 14 compares the rescaling per-



(a) Recovery Time (b) Throughput

Figure 15: Rescaling performance under backpressure

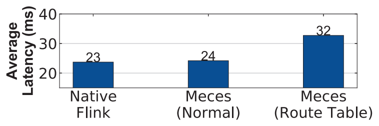


Figure 16: Latency comparison with Native Flink

formance of Meces using different state repartition strategies: the default uniform repartition and consistent hashing, which have different migration cost during rescaling. For a total of 128 key-groups, consistent hashing decreases the number of migrated key-groups from 115 to 15. Eventually up to 70% rescaling duration and 90% max latency are reduced. This indicates that we can equip Meces with existing re-balance technology to improve the rescaling performance. For a fair comparison, in all other experiments in this paper, we use the same repartition strategy in all SPEs including Meces.

Note that the above experiments are all conducted without any node failure or connection loss. As for the unusual scenarios including failing nodes with different roles, we observe that: (1) If any Flink node or Kafka/Redis leader fails, the latency curve is similar to Native Flink, because Meces restarts the job since the underlying service becomes unresponsive. (2) If Kafka/Redis loses some of the follower nodes but still provides timely service when job is rescaling, there is no observable fluctuation in the average latency of fetch operations, because of the relatively low traffic of messages sent by fetch operations. In both cases there is no data inconsistency in the stream processing results.

5.4 Performance under Backpressure

In practice, the rescaling of an SPE is usually triggered when it sends a backpressure signal, indicating that it cannot process data fast enough to keep up with the data generation rate. This happens when there is a sudden surge in data traffic or when the system is not configured properly with enough parallel instances. To validate our design in this situation, we evaluate the performance of Meces under backpressure scenarios.

A costly version of key-count query is chosen as the workload. We first configure the counter operator parallelism to 15 and run input generators at a speed of 300K records/s for 600 seconds. After that the input rate is increased to 600K records/s for 150 seconds, and then gets back to 300K records/s. This simulates a temporary surge in data traffic. The rescaling operation takes place at the 620th second, which increases the counter operator to 30 parallel instances.

Figure 15(a) shows how quickly the SPE recovers its real-time performance from backpressure. As soon as the data traffic surge comes, the latency increases suddenly, because the input is beyond the capability of the system and the following records are queuing up. Then, after the scale-out operation is completed and all the queuing records have been fully consumed, the processing latency can go back to a normal low level. Here we record the time interval from the arrival of the data surge until the system latency drops below 100 ms. As can be seen, Meces is the first one to get back to the previous processing rate, while it takes Native Flink and Order-Unaware much more time to consume the queuing data and recover because of the block of operator execution.

We then compare the system throughput within 2 minutes after triggering the rescaling operation, as shown in Figure 15(b). As Native Flink first triggers a global state snapshot and then restarts the job, its throughput immediately decreases to 0 and gradually increases after it restarts. As for Meces and Order-Unaware, they both keep the throughput at a non-zero level, but as Order-Unaware incurs long-duration blocking periods, it first goes through a decrease in throughput. Meanwhile, as Meces brings little degradation to the performance during the state migration, its throughput reaches the maximum much faster and higher than the others. This validates that the fetch-on-demand state migration mechanism of Meces outperforms the other approaches under backpressure.

5.5 Overhead Analysis

This subsection discusses the overhead introduced by Meces.

Latency Overhead: The latency performance is illustrated in Figure 16. On one hand, Meces does not incur extra latency when not migrating states, as reported in the "Meces (Normal)" bar. Under normal circumstances, the processing logic of Meces and Native Flink is substantially the same. The only difference is that Native Flink uses a normal HashMap to manage operator states, while Meces uses a nested HashMap with little overhead. Consequently, the difference in latency between the two SPEs is determined by system noise. This conclusion can also be drawn from Figure 9~12.

On the other hand, the processing latency increases when Meces is performing a rescaling operation. In this experiment, each route table contains 128 keys and increases the latency by 35%. During these periods, when deciding which downstream operator to send records to, the simple modulo operation is replaced by a more costly map query operation. In Meces, this mechanism is combined with the nested data structure, to reduce the migration granularity to an acceptable level without the need for an extremely high number of key groups. That means the route tables are kept in a reasonably small size, thus reducing the latency peak with negligible overhead.

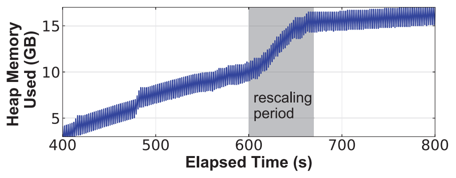


Figure 17: Memory consumption of Meces

Memory Overhead: Figure 17 demonstrates the memory consumption of one single machine. To clearly show the changes in memory usage, we configure the JVM to utilize g1gc as the garbage collector and do not trigger GC behavior for objects in old generation. The memory usage fluctuates up and down because JVM periodically reclaims the temporary objects in the continuously arriving data stream. In the meantime, the total memory usage shows an upward trend due to the gradual increase in the operator states.

When the state migration starts at 600 s, the curve rises rapidly, because the operator starts fetching states from others, allocating lots of new objects quickly. This can potentially decrease the processing performance if the system is implemented in programming languages like Java, as the garbage collector works under heavy pressure and can block the execution of user functions. To ensure the quick response of the system, we recommend a low-latency collector like ZGC [63], or a pre-allocated object pool to be utilized to reduce the overhead of Garbage Collection behavior.

5.6 Comparison with Other State Migration Approaches

In this subsection, we compare Meces with two representative state-of-the-art work Rhino [43] and Megaphone [26]. We choose the key-count job due to its simplicity so that we can minimize interference from associated computation and highlight the differences between rescaling approaches.

5.6.1 Comparison with Rhino

Rhino [43] proposes a state management approach which handles large states very well by periodically replicating operator states among all worker machines. Because the source code of Rhino from the original authors is not publicly available, we implement the mechanism of Rhino based on Flink for a fair comparison. During a stateful rescaling, Rhino generally follows the Partial-Pause approach, but only reads/writes the incremental parts of states since the last global replication. The replication interval is set to 60 seconds.

As in Figure 18, Rhino on Flink shows a similar latency peak to Order-Unaware, whose peak is near 10,000 ms. As a comparison, the per-record latency of Meces never reaches 1000 ms during the whole process. Rhino's replication fails to improve the system performance for two reasons:

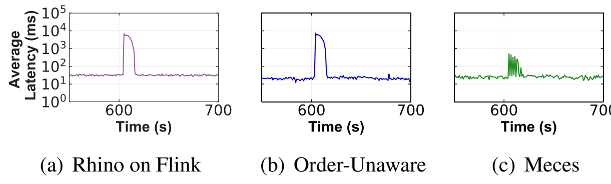


Figure 18: Latency comparison with Rhino during rescaling

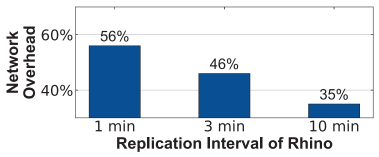


Figure 19: Network overhead of Rhino

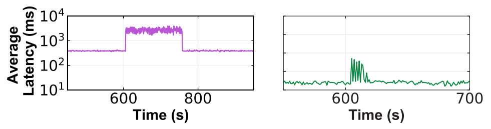
1. In such a scale-out scenario where new workers join the job, a global state migration is still necessary for these new workers because they lack the previous states.
2. Such read-modify-write jobs update the operator states very frequently. As lots of states are modified between two state replications, the incremental parts still occupy large proportions of the global state.

Eventually Rhino degrades to the Partial-Pause approach with latency spikes. In contrast, Meces uses a fetch-on-demand state accessing model for efficient state sharing among operator instances and less processing latency.

In addition, we compare Rhino and Meces in terms of the disruption to non-rescaling stream processing, by measuring the extra network bandwidth introduced by both systems compared to Native Flink. In Rhino, when replication interval decreases from 10 minutes to 1 minute, the extra network bandwidth ratio grows from 35% to 56% (Figure 19). This is consistent with the conclusion reported in the Rhino paper [43], where Rhino uses 30% network bandwidth during a replication. As Rhino periodically replicates operator states among all workers, it helps fault-tolerance in the face of very large states, but incurs extra communication across the network even if the system performs no state migration at all. In contrast, Meces incurs no network overhead to non-rescaling stream processing, because it performs no additional operations during non-rescaling periods.

5.6.2 Comparison with Megaphone

Megaphone [26] is a state migration approach that splits the state load into batches and embeds the migration flows into data flows for lower latency. It relies on two specific SPE features [26]: (1) state extraction from upstream operators (2) dataflow frontiers. However, both are still not natively supported in many widely-used SPEs, including Flink [16], Heron [33], Spark Streaming [53], Samza [50], etc. Therefore, to run Megaphone in a widely-used SPE, we meet the above requirements of Megaphone in Flink with naive implementations for state extraction and splitting the data stream



(a) Megaphone on Flink (b) Meces

Figure 20: Latency comparison with Megaphone

into micro batches underhook. Based on that, we implement Megaphone's state migration mechanism on Flink.

We then run the key-count job in both Meces and Megaphone with 10^8 unique keys. Megaphone is configured to use 2^{14} bins, according to the original suggestions [26]. The latency comparison is demonstrated in Figure 20.

When not rescaling, Megaphone on Flink shows an order of magnitude higher latency than Meces. There are two reasons for this phenomenon: (1) Megaphone's strong pre-requisite of SPE features calls for extra synchronization and communication techniques to fulfill the requirements, thus bringing dramatic overhead to the system performance [43]. (2) To prepare for state migration, Megaphone splits the original operators into two operators before the streaming job is submitted. However, this incurs extra partition overhead between the two new operators, resulting in increased processing latency for the regular execution of the job. As for Meces, it can work without expensive system requirements or logic modification to a non-rescaling SPE, and thus brings no overhead when the system is not rescaling.

During rescaling, both systems show a limited amount of latency increase. However, the latency of Megaphone stays at the level of around 8000 ms, while Meces never reaches the bar of 1000 ms. This verifies that compared with Megaphone, Meces's prioritized state migration can efficiently reduce the processing latency during rescaling with low overhead.

6 Related Work

Stateful stream processing has been an active research field in the past years, both in single-machine [1, 9, 44] and distributed settings [3, 8, 16, 50, 53, 62]. To meet diversified requirements of real-time computing in different scenarios, research works focus on various aspects of stream processing, including performance [32, 36, 37, 58, 59], reliability [45, 64], scalability [14, 15, 56], flexibility [23], programmability [5, 40], etc. In addition, some researchers propose SPEs with enriched semantics [38, 42, 57, 65, 66] to support more sophisticated analysis of streaming data.

Elasticity in batch processing systems has also been studied [34, 35], but they mainly deal with scenarios with rather higher processing latency. As for the field of elastic stream processing, the past decade also witnessed many advances [14–16, 25, 26, 33, 40, 43, 47, 52, 53, 61].

The most recent works related to ours are Megaphone [26] and Rhino [43]. Megaphone [26] provides efficient on-the-fly state migration for SPEs. This is achieved by transferring the operator states in a small granularity upon dataflow reconfiguration. It also supports trading off low latency against high throughput of state migration. Rhino [43] periodically replicates operator states among all worker machines. It can speed up the process of state migration by asking the operator instances to read/write from an incremental checkpoint instead of a global state snapshot. They both reduce processing latency during state migration at the expense of SPE performance during non-rescaling periods, as neither of them considers the migration order of states. In contrast, Meces uses a fetch-on-demand state accessing mechanism to enable prioritized state migration during rescaling, without extra resource usage in non-rescaling periods.

Another critical issue about the elasticity of stream processing is to decide when and how to rescale. Many SPE controllers [4, 19, 20, 28, 31, 39, 41, 54] have been proposed for adaptive rescaling to meet the QoS targets in various scenarios. These works are orthogonal to ours and can be combined with Meces's on-the-fly rescaling mechanism to provide self-regulating streaming systems.

7 Conclusion

This paper presents Meces, a latency-efficient on-the-fly rescaling mechanism using prioritized state migration for stateful distributed SPEs. Meces uses a fetch-on-demand model with hierarchical state data structure and gradual strategy, to achieve prioritized state migration with global consistency and high efficiency. This design puts all the operations in rescaling periods and requires no work during non-rescaling periods. We implement Meces in Apache Flink and evaluate our design on diversified workloads. The experimental results show that compared to state-of-the-art approaches, Meces improves the latency and throughput performance during rescaling by orders of magnitude without disrupting non-rescaling periods or using huge amounts of resources.

In the future, we plan to integrate Meces with stream performance monitoring tools and further study more adaptive rescaling policies for diversified scenarios on Meces.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We sincerely thank our shepherd for her/his guidance and time. This work is funded in part by the National Natural Science Foundation of China (No. 62072230), Alibaba Group through Alibaba Innovative Research Program, the National Natural Science Foundation of China (No. U1811461), Jiangsu Province Science and Technology Key Program (No. BE2021729), the Collaborative Innovation Center of Novel Software Technology and Industrialization. Rong Gu is the corresponding author of this paper.

References

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal '03*, 12(2):120–139, November 2003.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 739–753, November 2016.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *Proceedings of the VLDB Endowment (PVLDB '15)*, volume 8, pages 1792–1803, April 2015.

[4] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, pages 71–71, July 2006.

[5] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the 37th ACM International Conference on Management of Data (SIGMOD '18)*, pages 601–613, June 2018.

[6] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 39th ACM International Conference on Management of Data (SIGMOD '20)*, pages 2651–2658, June 2020.

[7] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[8] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM Conference on Programming Language Design and Implementation (PLDI '10)*, pages 363–375, June 2010.

[9] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. In *Proceedings of the VLDB Endowment (PVLDB '14)*, volume 8, pages 401–412, April 2014.

[10] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS '85)*, 3(1):63–75, March 1985.

[11] Debezium. Building audit logs with change data capture and stream processing. https://debezium.io/blog/2019/10/01/audit-logs-with-change-data-capture-and-stream-processing/, 2019.

[12] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations. In *Proceedings of the 11st ACM International Conference on Distributed and Event-based Systems (DEBS '17)*, pages 227–238. ACM, June 2017.

[13] Exastax. Real-time stream processing for internet of things. https://medium.com/@exastax/real-time-stream-processing-for-internet-of-things-24ac529f75a3/, 2017.

[14] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 32nd ACM International Conference on Management of Data (SIGMOD '13)*, pages 725–736, June 2013.

[15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. Making state explicit for imperative big data processing. In *Proceedings of 2014 USENIX Annual Technical Conference (ATC '14)*, pages 49–60, June 2014.

[16] Apache Flink. https://flink.apache.org/, 2015.

[17] A deep dive into rescalable state in Apache Flink. https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html/.

[18] flink-rescalable-state. https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html/, 2021.

[19] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. In *Proceedings of the VLDB*

Endowment (PVLDB '17), volume 10, pages 1825–1836, April 2017.

[20] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Auto-Scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking (TON '17)*, 25(6):3338–3352, June 2017.

[21] Yupeng Fu and Chinmay Soman. Real-time data infrastructure at uber. In *Proceedings of the 40th ACM International Conference on Management of Data (SIGMOD '21)*, pages 2503–2516, June 2021.

[22] Can Gencer, Marko Topolnik, Viliam Durina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, József Bartók, Grzegorz Gierlach, Frantisek Hartman, Ufuk Yilmaz, Ondrej Lukás, Mehmet Dogan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. In *Proceedings of the VLDB Endowment (PVLDB '21)*, volume 14, pages 3110–3121, 2021.

[23] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 213–231, October 2018.

[24] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems (TPDS '12)*, 23(12):2351–2365, 2012.

[25] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*, pages 13–22, May 2014.

[26] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. In *Proceedings of the VLDB Endowment (PVLDB '19)*, volume 12, pages 1002–1015, April 2019.

[27] Apache Kafka. `http://kafka.apache.org/`, 2011.

[28] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 783–798, October 2018.

[29] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th ACM Symposium on the Theory of Computing (STOC '97)*, pages 654–663, May 1997.

[30] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE '18)*, pages 1507–1518, August 2018.

[31] Alireza Khoshkbarforoushha, Alireza Khosravian, and Rajiv Ranjan. Elasticity management of streaming data analytics flows on clouds. *Journal of Computer and System Sciences (JCSS '17)*, 89:24–40, October 2017.

[32] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. SABER: Window-Based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 35th ACM International Conference on Management of Data (SIGMOD '16)*, pages 555–569, July 2016.

[33] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 34th ACM International Conference on Management of Data (SIGMOD '15)*, pages 239–250, May 2015.

[34] Alok Kumbhare, Marc Frincu, Yogesh Simmhan, and Viktor K. Prasanna. Fault-tolerant and elastic streaming mapreduce with decentralized coordination. In *Proceedings of the 35th International Conference on Distributed Computing Systems*, pages 328–338, June 2015.

[35] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. In *Proceedings the 5th Open Cirrus Summit*, pages 1–5, June 2011.

[36] Gyewon Lee, Jeongyoon Eo, Jangho Seo, Taegeon Um, and Byung-Gon Chun. High-performance stateful stream processing on solid-state drives. In *Proceedings of the 9th ACM Asia-Pacific Workshop on Systems (APSys '18)*, pages 9:1–9:7, August 2018.

[37] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated Key-Value storage management for balanced I/O performance. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC '21)*, pages 673–687, July 2021.

[38] Thomas Lindemann, Jonas Kauke, and Jens Teubner. Efficient stream processing of scientific data. In *Proceedings of the 34th IEEE International Conference on Data Engineering Workshops (ICDEW '18)*, pages 140–145, April 2018.

[39] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS '15)*, pages 399–410, June 2015.

[40] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. In *Proceedings of the VLDB Endowment (PVLDB '18)*, volume 11, pages 1303–1316, April 2018.

[41] Tiziano De Matteis and Gabriele Mencagli. Elastic scaling for distributed latency-sensitive data stream operators. In *Proceedings of the 25th IEEE Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP '17)*, pages 61–68, March 2017.

[42] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. In *Proceedings of the VLDB Endowment (PVLDB '20)*, volume 13, pages 1793–1806, March 2020.

[43] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 39th ACM International Conference on Management of Data (SIGMOD '20)*, pages 2471–2486, June 2020.

[44] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 439–455, November 2013.

[45] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pages 1–14, April 2013.

[46] Rabbitmq. https://www.rabbitmq.com/, 2007.

[47] Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman P. Amarasinghe. Gloss: Seamless live reconfiguration and reoptimization of stream programs. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, pages 98–112, March 2018.

[48] Redis Sentinel. https://redis.io/topics/sentinel/, 2021.

[49] Apache RocketMQ. http://rocketmq.apache.org/, 2012.

[50] Apache samza. http://samza.apache.org/, 2013.

[51] Twitter sentiment analysis: A tale of stream processing. https://towardsdatascience.com/twitter-sentiment-analysis-a-tale-of-stream-processing-8fd92e19a6e6/, 2020.

[52] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE '03)*, pages 25–36, March 2003.

[53] Apache Spark Streaming. https://spark.apache.org/streaming/, 2013.

[54] Rafael Tolosana-Calasanz, Javier Diaz Montes, Omer F. Rana, and Manish Parashar. Feedback-control & queueing theory-based resource management for streaming applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS '17)*, 28(4):1061–1075, October 2017.

[55] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark - a benchmark for queries over data streams draft. Technical report, School of Science and Engineering at OHSU, March 2008.

[56] Taegeon Um, Gyewon Lee, Sanha Lee, Kyungtae Kim, and Byung-Gon Chun. Scaling up IoT stream processing. In *Proceedings of the 8th ACM Asia-Pacific Workshop on Systems (APSys '17)*, pages 25:1–25:7, September 2017.

[57] Pourya Vaziri and Keval Vora. Controlling memory footprint of stateful streaming graph processing. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC'21)*, pages 269–283, July 2021.

[58] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS '11)*, pages 120–129, May 2011.

[59] Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *Proceedings of the 5th ACM International Systems and Storage Conference (SYSTOR '12)*, pages 1–12, June 2012.

[60] Virtuslab. Preventing fraud and fighting account takeovers with kafka streams. https://www.confluent.io/blog/fraud-prevention-and-threat-detection-with-kafka-streams/, 2020.

[61] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic stateful stream computation in the cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE '15)*, pages 723–734, April 2015.

[62] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 423–438, November 2013.

[63] The Z Garbage Collector. https://wiki.openjdk.java.net/display/zgc/Main/, 2021.

[64] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 1187–1204, November 2020.

[65] Yunhao Zhang, Rong Chen, and Haibo Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, pages 614–630, November 2017.

[66] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 37th ACM International Conference on Management of Data (SIGMOD '18)*, pages 741–756, June 2018.

A Artifact Appendix

Abstract

Meces is implemented on Apache Flink. It also relies on Kafka and Redis to function properly. We prepare the programs, assemble a workflow of Meces and package the artifact into a Docker image.

Scope

The artifact rescales a key-count job with different state migration mechanisms. It verifies the basic functions of Meces and validates the performance improvement brought by the prioritized state migration strategy proposed in this paper.

Contents

The artifact includes a compiled version of Meces, along with dependencies such as Kafka, Redis, Java and Python, etc. A "README.md" file can be also found in the image. It contains detailed description of the artifact and a step-by-step instruction for the rescaling workflow.

Hosting

The artifact is hosted on Docker Hub. It can be installed by downloading the pre-built Docker image from the public dockerhub repository and initiating a container from it:

- docker pull njupasalab/meces:latest
- docker run -it njupasalab/meces:latest

Artifact Check-list:

- Run-time Environment: Linux OS with Docker installed.
- Experiments: Workflow of rescaling a key-count job.
- Expected Experiment Running Time: About half an hour.
- Output: Data plots of latency comparison.

Expected Running Result

There are scripts that help pre-check the testing environment:

- source scripts/install.sh
- scripts/start_background_environment.sh
- scripts/check_environment.sh

Then, an experiment script evaluates the rescaling performance of Meces using the key-count job on your environment:

- scripts/rescale_exp.sh

It generally goes through three stages in series, namely *Meces*, *Order-Unaware*, *Native-Flink*. In each stage, the system submits the key-count job, runs for a while and then rescales the operator with its corresponding mechanism. For further details, please refer to "README.md" in the artifact.

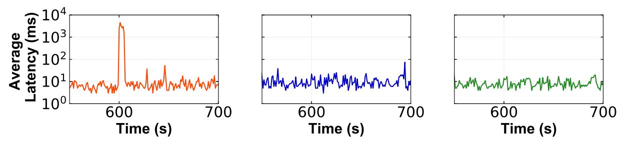
After the process finishes, the experimental data is collected in the *data* folder. Each experiment should generate a plot of the latency curve, similar to what is reported in Section 5.2.

B Appendix: Latency Performance Evaluation on full NEXMark suite during Rescaling

We evaluate the latency performance of the SPEs during rescaling. The SPEs are initially configured with global operator parallelism of 25. Then, each job runs at a steady input rate of 800K records/s for 600 seconds. After that a rescaling operation is triggered. It increases the parallelism of critical operators (e.g., join or window operators in NEXMark queries, counting operator in the key-count job) to 30. This causes 115 out of 128 key-groups to be relocated during the scale-out.

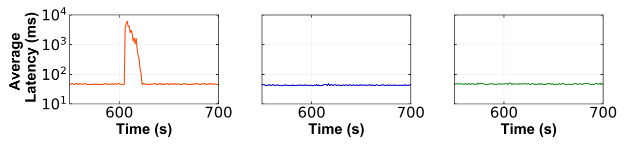
Figure 21~28 illustrate the end-to-end latency change of each query in this process. We compare Meces with Native Flink (stopping the whole job when rescaling) and Order-Unaware (online block-based state migration without order prioritization).

NEXMark Q1 and Q2 do currency conversion or filtering operations on a bid stream. These simple transformation tasks do not maintain states and the behavior is demonstrated as a basic case for our evaluation. As shown in Figure 21 and 22, both Meces and Order-Unaware reveal no latency peak but only system noise, because they incur no state migration cost during rescaling and operations can be done asynchronously. In contrast, Native Flink still needs to stop and restart the job even if there is no state to migrate.



(a) Native Flink (b) Order-Unaware (c) Meces

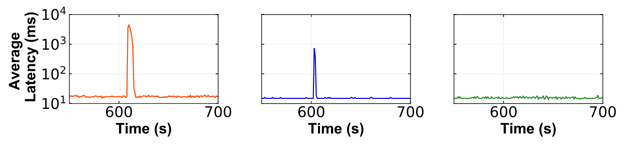
Figure 21: End-to-end latency of NEXMark Q1



(a) Native Flink (b) Order-Unaware (c) Meces

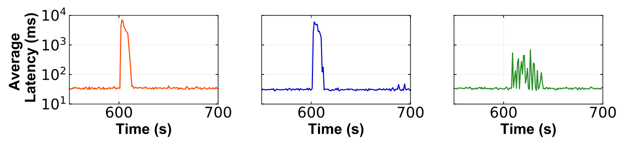
Figure 22: End-to-end latency of NEXMark Q2

NEXMark Q3 and Q4 test join functionality. Q3 joins the stream of open auctions and the stream of people to local item suggestions for users. Q4 joins the stream of closed auctions and the stream of items to output the average deal price of items for a category. Both queries have to store information of records in the data stream as states of join operators. The per-record latency is demonstrated in Figure 23 and Figure 24. In our settings, Q3 maintains a rather small size of states. Meanwhile, operator states in Q4 grow large when millions of items have been sold. As a result, for Native Flink and Order-Unaware, a sharp and short-lasting rise of latency can be seen in Figure 23, because the task executions are globally or partially blocked until the state migration is completely done. As a comparison, there is no obvious latency change in Meces as it reduces the disturbance caused by state migration to a lower granularity. A similar conclusion can also be drawn from the results in Figure 24 for Q4. The degradation of real-time performance becomes more significant for Native Flink and Order-Unaware with a larger size of states, while Meces only incurs a small range of latency fluctuations. The latency peak of Meces is an order of magnitude smaller than the other two mechanisms, due to its fetch-on-demand accessing and gradual migration strategy.



(a) Native Flink (b) Order-Unaware (c) Meces

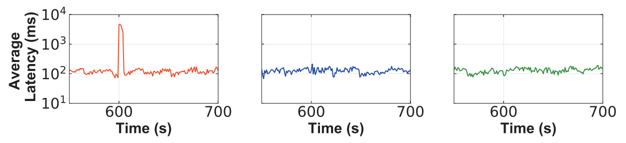
Figure 23: End-to-end latency of NEXMark Q3



(a) Native Flink (b) Order-Unaware (c) Meces

Figure 24: End-to-end latency of NEXMark Q4

NEXMark Q5 tests window operator with small states. It repeatedly selects the hottest item in the past period of time, which is the item with the most number of bids. The stateful operator maintains item counts in each time window. In our experiments, the job reports every second the hottest item over the last 60 seconds. As this query does not accumulate large states, it exposes similar behavior with Q1 and Q2, as shown in Figure 25. While the latency of Native Flink increases significantly due to a full restart, the record processing of Meces and Order-Unaware is hardly affected because the state migration cost is minor.



(a) Native Flink (b) Order-Unaware (c) Meces

Figure 25: End-to-end latency of NEXMark Q5

NEXMark Q6, Q7 and Q8 test window operator with bigger states. Q6 includes a sliding window over a single stream to calculate the average of items recently sold by a seller. Q7 and Q8 perform tumbling window join of two streams, to find out the items with the highest price and the new users who just registered in the last period of time. These queries maintain large states when the window size grows. We set the window sizes to be 10 seconds, 10 seconds, and 100 seconds for the three queries respectively. The performance comparison is illustrated in Figure 26~28, where the latency peak of Meces during rescaling is an order of magnitude smaller than the others. When the rescaling begins, Order-Unaware needs to block the currently processed records while migrating a considerable amount of states. As a result, Order-Unaware goes through a performance degradation near to Native Flink (Full-Restart), and reaches a latency peak of dozens of seconds. For comparison, Meces can give priority to the hot keys being processed and smoothen the latency peaks.

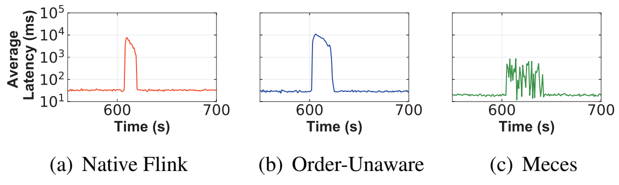


Figure 26: End-to-end latency of NEXMark Q6

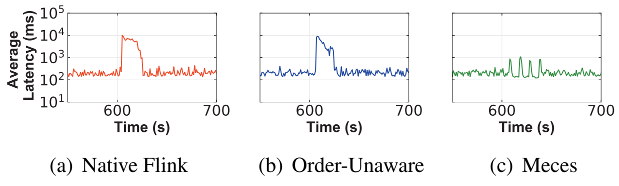


Figure 27: End-to-end latency of NEXMark Q7

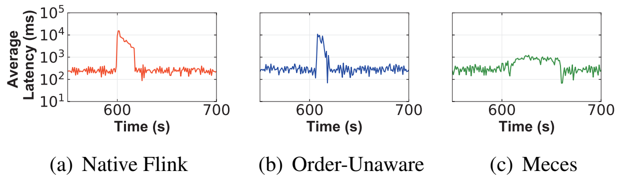


Figure 28: End-to-end latency of NEXMark Q8

In conclusion, Meces achieves better performance than existing approaches during rescaling. Native Flink and SPEs with Order-Unaware can suddenly become out-of-service when migrating states, while Meces relieves the impact of rescaling to a lower granularity and reduces the maximum latency.