T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 127

On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines

Adrian Bartnik[1] Bonaventura Del Monte[2] Tilmann Rabl[3] Volker Markl[4]

Abstract: Stream Processing Engines (SPEs) must tolerate the dynamic nature of unbounded data streams and provide means to quickly adapt to fluctuations in the data rate. Many major SPEs however provide very little functionality to adjust the execution of a potentially infinite streaming query at runtime. Each modification requires a complete query restart, which involves an expensive redistribution of the state of a query and may require external systems in order to guarantee correct processing semantics. This results in significant downtime, which increase the operational cost of those SPEs. We present a modification protocol that enables modifying specific operators as well as the data flow of a running query while ensuring exactly-once processing semantics. We provide an implementation for Apache Flink, which enables stateful operator migration across machines, the introduction of new operators into a running query, and changes to a specific operator based on external triggers. Our results on two benchmarks show that migrating operators for queries with small state is as fast as using the savepoint mechanism of Flink. Migrating operators in the presence of large state even outperforms the savepoint mechanism by a factor of more than 2.3. Introducing and replacing operators at runtime is performed in less than 10 s. Our modification protocol demonstrates the general feasibility of runtime modifications and opens the door for many other modification use cases, such as online algorithm tweaking and up- or downscaling operator instances.

Keywords: Data Stream Processing, Resource Elasticity, Query Plan Maintenance, Fault Tolerance

1 Introduction

Stream processing engines (SPEs) have become an essential component of many business use cases and need to reliably ensure correct processing of the incoming, high-speed workload. However, the characteristics of a streaming workload may vary over time, since it includes foreseeable and predictable changes within a time frame (e.g., the day and night usage patterns for social media posts) but also bursty spikes in the case of irregular events such as sport or weather events. The ability of an SPE to adapt to workload changes at runtime is one aspect of *elasticity*. Currently, many SPEs (e.g., Apache Flink [Ca17], Apache Storm [To14], and Apache Spark [Za13]) provide limited functionality to support changes in their running configuration, if any at all. These systems allow modifications of a running streaming query only by restarting its execution with a new configuration. However,

[1] Technische Universität Berlin, bartnik@campus.tu-berlin.de
[2] DFKI GmbH, bonaventura.delmonte@dfki.de
[3] Technische Universität Berlin - DFKI GmbH, rabl@tu-berlin.de, tilmann.rabl@dfki.de
[4] Technische Universität Berlin - DFKI GmbH, volker.markl@tu-berlin.de, volker.markl@dfki.de

doi:10.18420/btw2019-09

this may violate SLAs when other, critical systems rely on the output of the SPEs and cannot tolerate any downtime. As a result, end-users of those SPEs face potentially ever-running streaming queries that they cannot alter at runtime. Motivated by this technical challenge, our goal is to enable modifications of a running query at runtime, without stopping the SPE. We analyze diverse modifications and we derive a set of generic protocols that alter the physical plan of a running query. Our protocols enable introducing, tuning, and removing operators in a running query. This opens the door for a new class of online optimizations. Through our protocols, the SPE can tune the behavior of an operator, e.g., increasing its number of network buffers to sustain higher incoming load. Our protocols also enables the SPEs to migrate operators among different nodes, e.g., when a node is scheduled for maintenance. This is particularly challenging for jobs with large state, which have long recovery times. Finally, our protocols potentially allow in- or decreasing the degree of parallelism of an operator, which is beneficial for the system to adapt to changes of the incoming workload. Our contribution are as follows:

- Migration of stateless and stateful stream operators at runtime
- Introduction of new operators into a running streaming query
- Changing user defined functions of operators in running query
- Implementation of our protocols in Apache Flink
- Evaluation of our solution on Nexmark and a custom benchmark

This paper is structured as follows: we first provide background concepts in Section 2 and then we describe the design of our protocols in Section 3. After that, we present the architecture of an SPE integrating our protocols in Section 4 and its new features in Section 5. In Section 6, we show the experimental evaluation of our system. In Section 7, we conclude by summarizing our contributions and providing insight about future works that our current work enables.

2 Background

In the following, we present the background concepts that lay the foundation to our work. In particular, we provide an overview of data stream processing and a description of Apache Flink with its checkpoint mechanism, which we use as building block for our techniques.

2.1 Data Stream Processing

Data stream processing enables continuous analysis of real-time, unbounded data. Although SPEs are part of the data processing stack for more than a decade [ScZ05], only recently the need for real-time big data processing has fostered the development of a new generation of SPEs. Those new SPEs target massively parallel cloud infrastructures by extending the batch-oriented MapReduce paradigm [DG04]. The new engines improve on MapReduce as they consider low latency an important constraint. New SPEs adopt one of two processing models: micro-batching and tuple-at-a-time processing.
The micro-batching model discretizes the processing of an unbounded stream in a series of

finite chunks of data. While this approach ensures higher throughput, the size of each chunk drastically impacts the latency of running queries [Za13, Ku15]. The tuple-at-a-time model allows for a more fine-grained processing of incoming records, thus achieving lower latency. This model still uses a batching mechanism at the physical level to ensure high throughput, i.e., operators pack tuples into buffers [Ca17].
Regardless of the processing model, a natural way of modeling data flows in SPEs is by means of a *Directed Acyclic Graph* (DAG), which contains source, processing, and sink nodes. *Source nodes* continuously emit a stream of data elements, which are also commonly referred to as *tuples* or *records*. A stream is a potentially unbounded sequence of tuples generated continuously over time. *Processing nodes* consume, process, and emit new data elements. A *Sink* consumes but does not forward any new elements, e.g., it writes its output to disk. Source, processing, and sink nodes are connected through edges, which represent the communication channels for the data exchange among operators.
Processing nodes are either *stateful* or *state-less*. For state-less nodes, the output only depends on each incoming data element. In contrast, the output for stateful nodes depends on the incoming data elements as well as on some internally managed state.

2.2 Apache Flink

Apache Flink is an open-source dataflow processing framework for batch and stream data [Al14]. Flink uses the parallelization contract (PACT) programming model, a generalization of the MapReduce programming model, and second order functions to perform concurrent computations on distributed collections in parallel [Hü15]. Flink compiles submitted queries into DAGs that it optimizes and executes on a cluster of nodes. Flink relies on pipelining and buffering to avoid the materialization of intermediate results. Stream operators in Flink exchange intermediate results via buffers, i.e., an operator sends its buffer downstream when it is full or after a timeout. This enables processing data with high throughput and low latency. Furthermore, Flink provides operator fusion [Hi14]. This ensures that fused operators exchange tuples in a push-based fashion, whereas not-fused operators exchange buffers in a pull-based fashion. Back-pressure occurs in Flink when an operator receives more data than it can actually handle. Back-pressure is usually due to a temporary spike in the incoming workload, garbage collection stalls, or network latency.

2.3 Fault Tolerance and Checkpointing in Apache Flink

Flink's *Checkpointing Mechanism* consistently stores and recovers the state of a streaming query [Ca17]. The mechanism ensures fault tolerance in the presence of failures, meaning upon recovery, the program's state eventually reflects the stream state from before the failure. The checkpointing settings enable specifying message delivery guarantees, which ensure that every record from the data stream is processed exactly-once, at-least-once, or at-most-once. This mechanism lays the foundation for the *Savepoint Mechanism*, which enables deliberately stopping and resuming a running query. When restarting from a savepoint, Flink allows for updating aspects of the query itself, e.g., adding or removing

operators as well as adjusting their degree of parallelism. The checkpointing mechanism periodically triggers *checkpoints* of the streaming query, which act as independent snapshots to which the system can fall back in case of a failure. For queries with small states, the checkpoints have only negligible impact on the overall system performance. For queries with large state, checkpointing may drastically affect the query performance, mainly because of the time needed to copy the state on a persistent storage. In case of a program failure, Flink stops and restarts the streaming query by resetting the operator state to the one captured in the last successful checkpoint. The checkpointing mechanism requires a persistent stream source (e.g., Apache Kafka) that allows for rewinding the stream to a specific point in time and resend all subsequent messages for strict processing semantics. Non-replayable stream sources instead loose intermediate records upon query restart.

3 Protocol Description

This section describes the main features of the migration protocol that enables runtime modifications on a running streaming query. These are migrating stateful operator instances across nodes, introducing new operators into a running query, and replacing user-defined functions (UDFs) of operators.

3.1 System Model

Before we discuss the design of the migration protocols, we provide a description of our system model. We assume that our SPE ingests an infinite set of tuples $r_1, r_2, \cdots$. A tuple $r_j = (A, t)$ consists of a set of attributes A and a timestamp t. Each operator in our system processes a tuple at a time through a UDF $f(r_j, S)$, where r_j is the input tuple and S is the current state of the operator. According to the semantic of its UDF, the operator emits zero or more output tuples upon processing an input tuple. Each operator runs with its own degree of parallelism. A logical query plan consists of a set of source, sink, and processing operators, which the SPE models as nodes of a DAG. A physical query plan contains all the parallel instances of all the operator. The SPE also represents this as a DAG. We refer to a job as a running physical plan in the SPE. The edges of a DAG act as communication channels between those machines and represent the data exchange among operators, which follows three patterns. A parallel instance of an operator can 1. forward a record to a single parallel instance of a downstream operator, 2. broadcast a record to all parallel instances of a downstream operator, and 3. send a record downstream based on some partitioning function. To support checkpointing, the SPE injects special markers m_i through the sources into the DAG, periodically or upon user request. Marker m_i triggers the i-th checkpoint for a job. Each parallel instance of an operator receives a marker on all its inbound communication channels and reacts by snapshotting its state and forwarding the marker downstream. As soon as all parallel instances successfully complete their snapshot, they asynchronously send the checkpointed state to persistent storage (e.g., a distributed file system). As soon as those asynchronous copies finish, the SPE marks the checkpoint as completed. Checkpoint markers logically divide the processing of an input stream in finite sets of tuples. The

SPEs guarantees exactly-once processing of each tuple in every set. The SPEs consists of a coordinator process and several workers processes, which run on a cluster of nodes. Each worker process has a set of execution slots that determines the number of parallel instances of an operator it can run. In the rest of this section, we provide a thorough description of our proposed protocols based on the system model presented above.

3.2 Migration Protocol Description

This first protocol enables the migration of operator instances across machines, e.g., when detecting an upcoming hardware failure on a node in the cluster. Upon a migration request, the SPE retrieves all operator instances on the faulty node and allocates resources for these instances on others nodes. The SPE starts a migration by creating a special *modification marker* that contains all necessary information to perform the migration and ingests it into the DAG at the source operators (following the ideas of Del Monte [DM17]). Each operator instance eventually receives these markers and decides whether it needs to react on that migration marker. The key concept here is that a migration does not only affect the actual migrating instances but also the up- and down-stream instances. The upstream instances temporarily buffer their records during the migration duration as well as guarantee processing semantics and maintain FIFO order. The migrating instances store their state in persistent storage in order to consistently resume processing tuples once restarted on a different worker. The downstream instances rewire their inbound communication channels to correctly consume records from the migrated instances.

3.3 Modification Protocols Description

In addition to migrating operator instances, our protocol also enables the modifications of the data flow, e.g., the introduction of new operator instances or replacing the operator function in a running job. Both operations require the distribution of the UDFs in the cluster at runtime. The SPE starts each modification similarly to a migration by ingesting the modification marker at the source operators.

3.3.1 Introduction of New Operators

Upon receiving the modification marker, each operator checks how it needs to react. The upstream operators have to start buffering their outgoing records. Then, they broadcast the upcoming location of the newly introduced operator to all downstream operators. As soon as all upstream operators have successfully acknowledged buffering, all new operator instances will be started. Before the actual modification starts, the SPE distributes the UDFs compiled code to the target nodes such that the operators can immediately start instantiating those UDFs and processing records. The downstream operators instead attempt to connect to the machine of the newly instantiated operator instances or fail otherwise.

3.3.2 Changing the Operator Function

The last modification operation enables replacing the UDF of an operator at runtime. Similar to introducing an operator, the user first needs to provide the SPEs with the new UDFs compiled code for the operator to update The SPEs distributes the compiled code to all nodes on which the target operator runs. The SPE then introduces a special modification marker at the DAG sources that sets up all operator instances for the modification. Upon receiving the checkpoint marker for that modification, each instance of the target operator instantiates the new UDF and continues processing records.

4 System Architecture

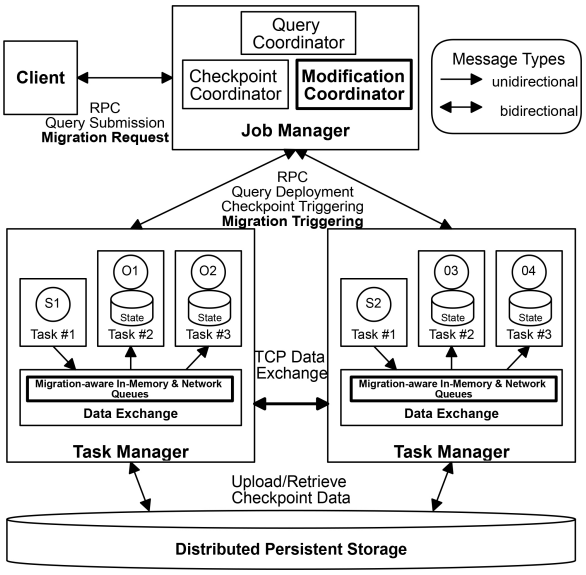


Fig. 1: System Architecture. Our contributions are marked in bold.

This section deals with our system architecture that support the migration and modification protocol from the previous section. We explain the components that drive and supervise the modifications as well as the integration with the checkpointing mechanism of Apache Flink (Version 1.3.2). We first provide an overview of the existing components and then we illustrate our changes on the coordinator and worker sides.

4.1 Vanilla Components Overview

The client sends modification control messages to the coordinator (i.e., the Job Manager), which interacts with the Flink cluster. It delegates and distributes the actions between all workers (i.e., the Task Manager), that run the actual operator instances. These control messages are not part of the actual data flow and are realized as Remote Procedure Call (RPC) via the actor model [HBS73]. The actor model represents a system, in which all entities, namely actors, run concurrently and solely communicate via message passing. The client, the Job and Task Managers use the actor system to concurrently communicate via asynchronous messages. Apache Flink handles the data exchange among parallel instances of two operators in a produce-consumer relation through an internal protocol built on top of TCP. All operator instances on a Task Manager share some of their resources, such as TCP connections via multiplexing as well as common data sets and data structures. Each instance is responsible for establishing a connection to its upstream instances in order to retrieve the input tuples. Flink leverages either the Job Manager or a third-party storage system to persistently store the checkpoint data. In Figure 1, we present a conceptual view of the resulting system architecture.

4.2 Our Changes on the Coordinator Side

Besides controlling the health of every Task Manager and monitoring queries execution, the Job Manager is responsible for the checkpoint mechanism. The existing *Checkpoint Coordinator* handles all functionality related to the checkpointing mechanism, i.e., it triggers checkpoints, supervises the lifecycle of a checkpoint, and may restart jobs based on past savepoints. Similarly, our newly introduced *Modification Coordinator* handles processing and validating modification commands as well as triggering and supervising the execution of these modifications. The validation includes whether the requested modification is applicable to the current, running job and some modification-specific checks. If successful, the Modification Coordinator prepares a modification-specific trigger messages, which it introduces in the data flow at source vertices of the target job. The coordinator keeps track of all vertices contained in the DAG and their modification life-cycle. Should an error occur during a modification, the coordinator aborts the current modification and notifies involved operators about its cancellation. Each operator may react to a certain trigger message or choose to ignore it, yet in any case it will acknowledge the reception to the Modification Coordinator. Depending on whether all vertices successfully acknowledge the trigger message as well as all subsequent, modification-specific state changes, the coordinator eventually marks a modification as *completed* or as *failed*. Normally, a task starts in the *Created*-state, gets *Scheduled* and *Deployed* by the Job Manager, processes all its input in the *Running* and eventually finishes by entering the *Finished* state. A task may fail for various reasons and subsequently enters the *Failed* state. Finally, if the SPE cancels a task due to an external reason, it enters the *Canceling* and *Canceled* state consecutively. We introduce two additional states, i.e., *Pausing* and *Paused*, which the task enters when the SPEs triggers a migration. In case of stateful operators, the tasks checkpoint and submit their state to a distributed persistent storage system.

4.3 Our Changes on the Worker Side

Each Task Manger executes *tasks*, which provide the environment for running the concrete operator instances. A task is mainly responsible for establishing the connection between Task Managers according to the up- and downstream operators, deserializing incoming records, processing them, serializing output records, and placing them in outgoing queues. The Job Manager determines the location of the up- and downstream operators in the job's initialization phase. The type of connections between operator instances depends on their relative location. In the case both producing and consuming instances are located on the same Task Manager, two instances transfer records through an in-memory queue. In the case they run on different Task Managers, the producer sends its output records via the network. When performing an operator migration, it is necessary to buffer intermediate records as long as the new consuming operator is not yet ready to receive records. Therefore, we implement a dedicated queue that is able to retain buffers while not blocking the upstream producers. It holds the buffers as long as possible in memory but it can also spill these buffers to disk. The spilling phase is asynchronous, i.e., writing to disk will not block. As

soon as a new consuming operator starts, it first reads all spilled buffers and only afterwards fetches newly-arrived buffers from memory to preserve FIFO semantic.

4.4 Query Plan Modifications

The protocol must at all times guarantee data integrity by not loosing any records or events and maintain the processing semantics. For each single operator instance to modify, we need to also consider its up- and downstream operators, since they should continue to process records. Therefore, instead of handling each operator individually, the modification message contains complete instructions for all operators. The logic for preparing this modification message is located in the Modification Coordinator, each operator simply follows the instructions contained in the modification message. We trigger the modification through RPCs to the source vertices, whereas we rely on the dataflow channels to propagate the modification message to remaining vertices with the same speed of records. Additionally, by relying on Flink data flow mechanism, all messages maintain FIFO semantics and are processed exactly once.

4.4.1 Upstream Operators

The checkpointing mechanism guarantees fault tolerance by aligning the checkpoint markers for each incoming data stream in each operator. Each operator instance receives records and events from the upstream operator. It also buffers pending records during the alignment phase of a checkpoint. This phase occurs when an instance has not received all the markers for its upstream instances and thus needs to buffer the incoming records for the blocked channels. Therefore, in case an operator wants to migrate to a different Task Manager, the SPE needs to migrate those buffers as well, increasing the state size. Because these buffers are not part of the internally-managed state, we need to handle them separately. To this end, the modification mechanism ensures that all upstream operators perform a custom alignment procedure on the sending side. Modification messages contain an upcoming checkpoint id. When the checkpoint with that id is triggered, the upstream operators broadcast checkpoint barriers along with a modification acknowledge message to the migrating downstream operators. This event signals to consumer operators that the upstream operator is spilling all further records and events to disk. Hence, it is safe to expect no new buffers from this operator instance. Synchronizing on a checkpoint on the sending side guarantees that no data are currently in-flight between the operators and the target operator has no buffered data. Therefore, it is safe to take further actions, such as pausing the operator for a migration. However, this also means that the modification has to wait until that specific checkpoint is acknowledged at the producing operators. The checkpointing interval determines how often a checkpoint is triggered by the Checkpoint Coordinator. The duration of taking a specific checkpoint depends on many factors, but most importantly on the number of operators in the job and the actual state size. Therefore, for jobs with small state and a low checkpointing interval, the SPE completes a modification trigger message in a matter of seconds. For jobs with large state and thus long checkpointing interval, the waiting time for the checkpoint to finish may be significantly longer than the duration of the actual modification itself.

4.4.2 Target and Downstream Operators

If an operator reacts to a trigger message, it will transition into a new state according to the specific modification and wait for further actions or events. Instances of downstream operators generally only wait for the target operators to react. In case the location of the target operator changes, such as during a migration, downstream instances receive the new location in the last record and then update the connection to that new upstream instance accordingly.

5 Protocol Implementation

The following section describes each modification operation in detail. These are migrating operator instances across Task Managers, introducing new operators into a running job and replacing user-defined functions of operators at runtime.

5.1 Operator Migration

In Figure 2, we provide an overview of the four steps involved in a migration. Upon a migration request, the Modification Coordinator retrieves all operators on that specific Task Manager and allocates new execution slots. It checks if enough resources are available, but also assigns these new slots to each migrating operator. This step is essential as each slot determines the location on which the operator instances will restart after state submission. The Modification Coordinator creates the trigger message that contains instructions for the involved operators and ingests it into the data flow through the source operators. This message determines the operators to migrate as well as the upstream operators that react by spilling their records to disk. Involved operators wait for the upcoming checkpoint marker to trigger the actual migration. When the Checkpoint Coordinator introduces that marker in Step 2, all upstream operators stop sending records to the migrating instances. Every operator starts the migration sequence by collecting and transmitting its current state to the Modification Coordinator as well as transitioning into the *Pausing* state. Additionally, it sends its upcoming new location to all downstream operators, as shown in Step 3. Finally, the task initiates the release of all allocated resources and enter the *Paused* state. As soon as the Modification Coordinator receives the confirmation of a successful transition to the *Paused* state, it restarts the operator. In Step 4, the Modification Coordinator attaches the state location and starts the execution in the previously assigned task slot. In addition, it will compute the new in- and outputs of the instance of the migrating operator based on the updated location of all other migrating operators instances. We rely on the checkpoint mechanism of Flink to collect and reassign the state of every operator.

5.2 Introduction of new Operators

The modification protocol also enables the modification of the data flow itself, in particular, the introduction of new operators into a running query. The requirement for this modification

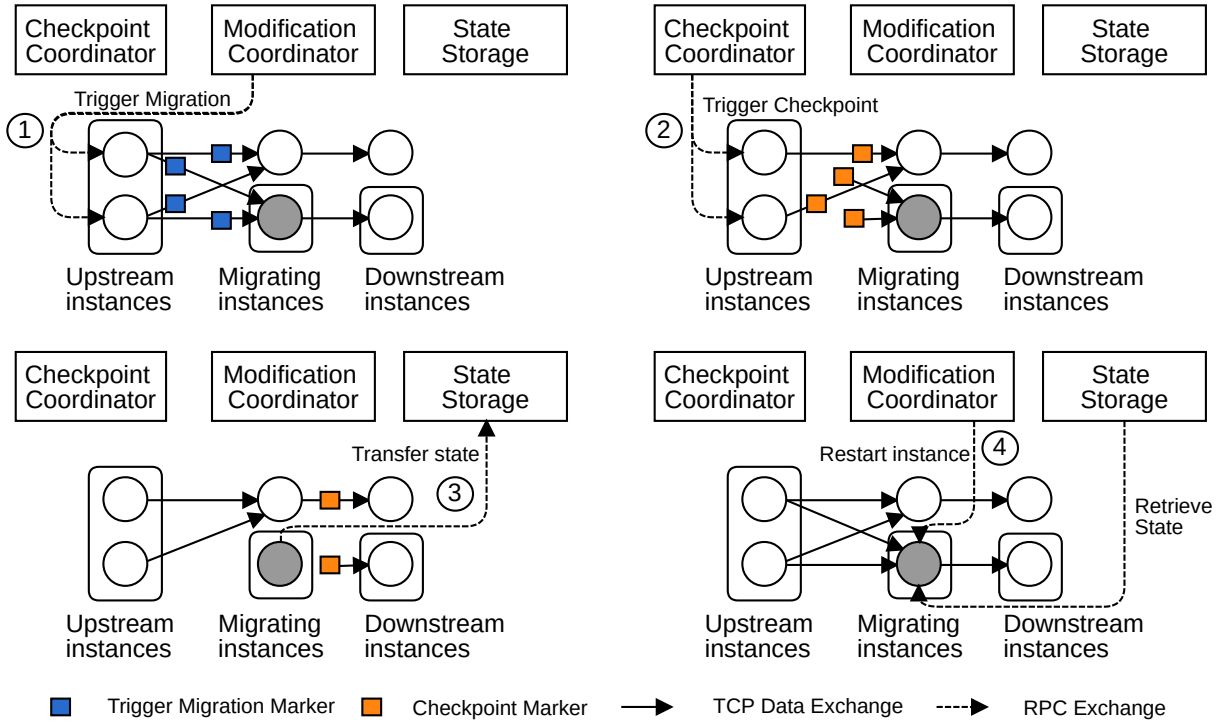


Fig. 2: Overview of modification protocol procedure

is to provide the SPE with compiled code for the new operators. Apart from that, introducing operators works similarly to migrating operator instances. The set of all upstream operators is formed by all operator instances directly before the new operator. Similarly, the set of downstream operators consists of all operators after the operator that should be inserted. As soon as all upstream operators successfully acknowledge the spilling phase, the Modification Coordinator deploys the new operator instances. The deployment payload also contains the compiled code for the new operators. Prior to a modification, our protocol enforces that new operators do not violate any type constraints of the up- and downstream operators.

5.3 Changing the Operator Function

This modification operation enables replacing the UDF of an operator at runtime. Similar to introducing an operator at runtime, the user needs to provide the new code for the target UDFs. The Modification Coordinator then introduces the modification message in the dataflow containing the checkpoint id and the reference for the new UDF. Through this reference, each Task Manager involved in the modification asynchronously fetches the actual new code once. Afterwards, each operator instantiates the new UDF and registers a callback. When the SPE completes the checkpoint with the ID specified in the modification message, every Task Manager triggers the callback locally. The purpose of the callback is to replace the operator function with the new one only when the SPE completes a global checkpoint. The rationale behind this choice deals with guaranteeing exactly-once processing semantic. We need to ensure that we consistently process each record with the new UDFs only after a specific point in time.

6 Evaluation

In the following, we present the empirical evaluation of our proposed modification protocols. We use the Nexmark and a custom benchmark to evaluate our protocols. We present the performance metrics for the operator migration, introduction a new operator; as well as the replacement of UDFs. Finally, we discuss the results and findings of our work.

6.1 Data Generator

In order to minimize the dependencies and not benchmark external systems, we implement a custom data generation and a message queuing system based on the work of Karimov et al. [Ka18]. We prefer an external generator over consuming from an internal Flink source because this may lead to unrealistic throughput and latency metrics as the source's record generation affects the overall system performance. In case of back-pressure, for instance, Flink is not able to produce new data because of the unavailability of network buffers. We use our data generator in all the following benchmarks to create a constant, non-fluctuating workload.

6.2 Cluster Environment and Benchmark Setup

We run all the experiments on an 8-nodes cluster, each with 48 cores and 48 GB of memory. Out of the 8 machines, one is dedicated to the Flink Job Manager, five run the Task Managers and the remaining two machines host three data generator instances each. This ensures that the generator instances do not interfere with neither the Job Manager nor the Task Managers. Our benchmarks have a warm-up phase of 10 minutes, after which each modification is triggered. For all benchmarks without windowing, we calculate the latency by subtracting the record's creation timestamp from its arrival time at the sink. If not mentioned otherwise, the filesystem state backend is used for the checkpointing mechanism.

6.3 Workloads

This section introduces the workloads with which we have evaluated the migration protocol.

6.3.1 Stateful Map Query

The first benchmark is a stateful map query (SMQ) with a source, a stateful map, and a sink operator. The sources read monotonically-increasing numbers along with a creation timestamp. Each map operator counts the number of elements this particular instance has received so far and appends that number to the output tuple. Finally, the sink computes the overall latency for each element by subtracting the event's timestamp from the current timestamp and writes everything to disk. This benchmark is used to demonstrate the correctness of the migration mechanism with small state size. In addition, it demonstrates the mechanism with many operators of varying degree of parallelism. The source operator, the map operator, and the sink operator have 60, 80, and 70 parallel instances, respectively.

6.3.2 Nexmark Benchmark

The modification protocol is suited for streaming queries with large state, which need longer to write and recover their state in case of failures. For this purposed, we select Query 8 (NBQ8) of the Nexmark Benchmark Suite [Tu18]. This benchmark represents a three-entity online auction system and is designed to measure the performance of various aspects of a streaming system. The three entities are the stream sources *person*s, *auction*s, and *bid*s. The *person* source emits a registration event, every time a new user registers in the auction system. The *auction* source emits events for each newly created auction by a specific person. Finally, persons may bid on auctions, which creates a *bid* event emitted by the respective source. Query 8 finds those persons who created a new auction within a certain time frame after signing up at the auction service. We implement this query through a windowed join using a tumbling window of 20 minutes in event time. Since this benchmark generates a substantially larger state than the previous benchmarks, it is not feasible to write the state to disk every single time. For job with large state, writing gigabytes-sized state to disk simply takes too much time. Therefore, we leverage RocksDB[3], i.e., the Flink-embedded key-value store, which offers *incremental checkpointing*. We set the checkpointing interval to 30 s.

6.4 Migration Protocol Benchmark

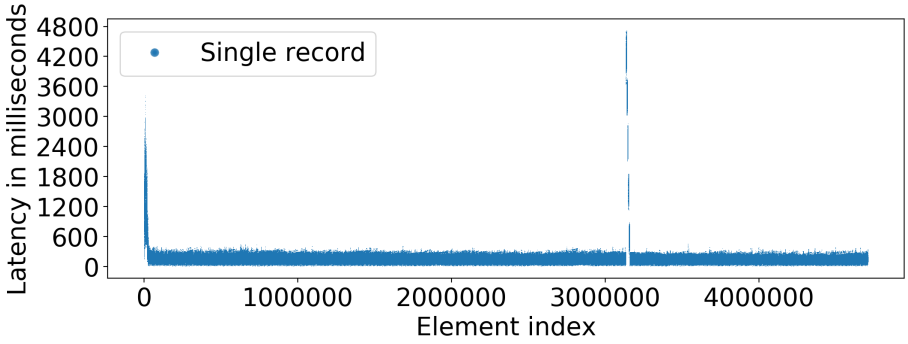
This section demonstrates the migration operation on the SMQ and NBQ8. We compared our migration technique against the Flink mechanism of canceling and restoring a job with a savepoint. As Flink has no special operation for simultaneously taking the savepoint and canceling the job, those operations can only occur subsequently. Therefore, Flink waits until all operators have successfully acknowledged the savepoint and then cancels the query. While taking a savepoint, the job continues to consume records, which means that all in-flight records are not actually part of the savepoint and are therefore lost upon restoring the job.

6.4.1 Stateful Map Job Performance Drill Down

Savepoint. Figure 3a shows the duration for each event, when canceling and restoring the job from a savepoint. Initially, most of the time is spent on establishing the connection to submit the cancellation command to the job, whereas actually canceling takes only about 300 ms. The system stores savepoint (100 KB) on a shared file system. Restoring and scheduling the job takes roughly 3 s. This adds up to a total duration of around 7.5 s. The latency spike in Figure 3b represents the time in which the streaming job restarts and corresponds to the actual restarting duration of about 4.5 s.
Migration. Figure 4a shows the duration for each step when migrating all operator instances from one specific Task Manager. The initialization for sending the `TriggerMigration` messages takes a bit less than 3 s. The next 2 s are spent waiting for the upcoming checkpoint barriers to arrive at the job sources, upon which they will spill to disk and broadcast their

[3] `http://rocksdb.org/`

(a) Event timeline for canceling and restoring the stateful map job via a savepoint



(b) Individual records latency

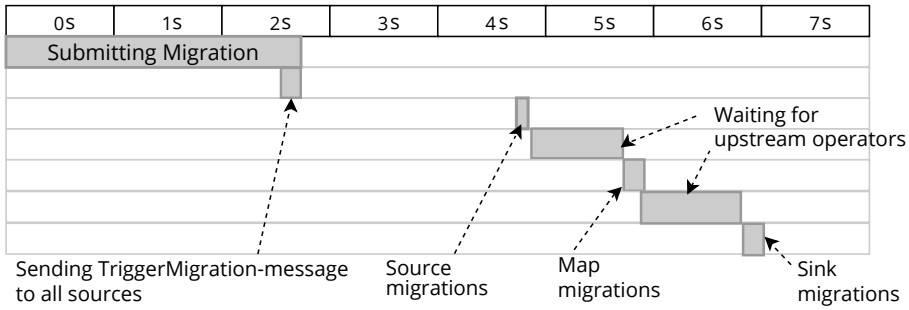
Fig. 3: Benchmark results for canceling and restoring the Stateful Map Job via a savepoint

new location. It takes almost another 1 s until the map operators are able to enter their migration. This happens once they receive either the spilling to disk events or the new location messages from all upstream sources. The sink migrations are fast and take less than 0.3 s. The long duration of almost 1.5 s might be explained by communication overhead introduced by the high degree of parallelism. In total, our mechanism migrates 34 operator instances, while 112 operator instances spill to disk during the migration. The migrating instances are 10 source, 12 map and 12 sink operators with a total migrated state size of 17 kB. We see the latency spike at the time of the migration of about 3500 ms, as shown in Figure 4b. This is because the actual duration of the migration without waiting for the checkpoint barriers takes about 3 s plus some additional delay for reconnecting to the operators. The migrated operators continue to process records after the migration, whereas the old operator instances stop their execution.

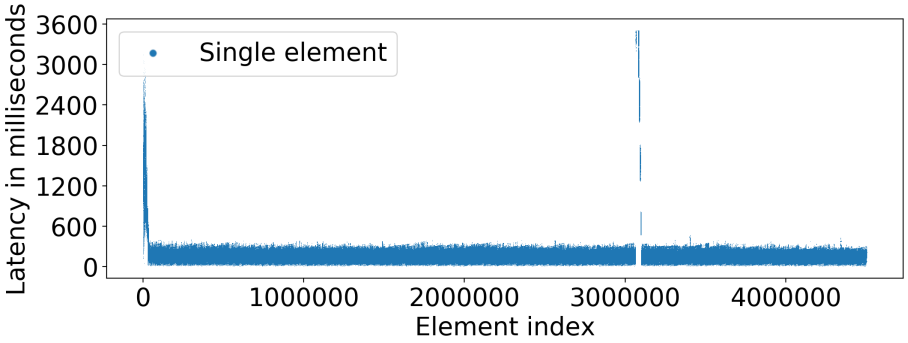
6.4.2 Nexmark Benchmark Performance Drill Down

Savepoint. Figure 5a shows the complete duration for canceling and restoring the job via a savepoint. The system needs about 161 s from submitting the savepoint command until the state has actually been stored in the state backend. We show this in Figure 5b as there is no decrease in sent records due to the fact that the job sources still consume data. The state size after running the Benchmark for about 12 min is about 13.4 GB. The complex setup is the cause of the long cancellation duration.

Migration. Figure 6a shows the complete duration for migrating all operators from one specific Task Manager. The system spends almost 1 min waiting for the upcoming checkpoint barrier to arrive. This is due to the 30 s checkpointing interval of which at least one full cycles needs to pass. Since two times the checkpointing interval is also the maximum time

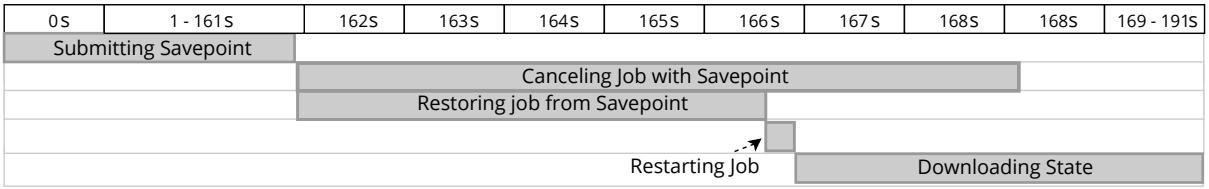


(a) Event timeline for migrating operators of the stateful map job

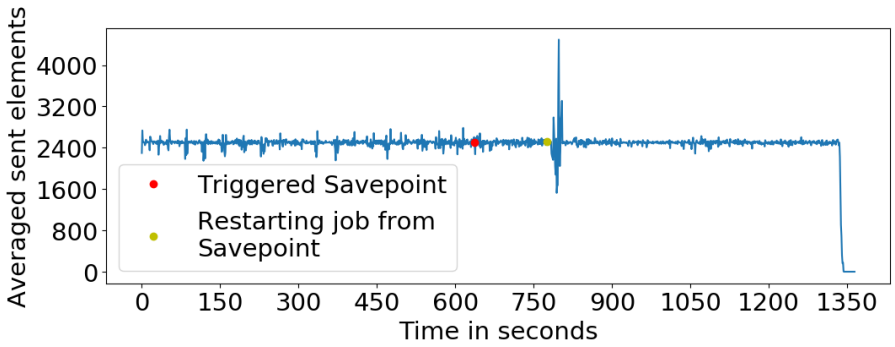


(b) **Individual records latency**

Fig. 4: Benchmark results for performing a migration on the Stateful Map Job



(a) Event timeline for canceling and restoring the Nexmark Benchmark via a savepoint

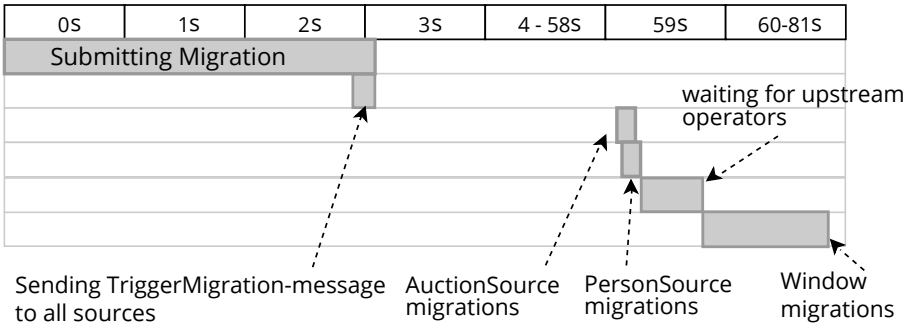


(b) Average throughput at the 80 generators

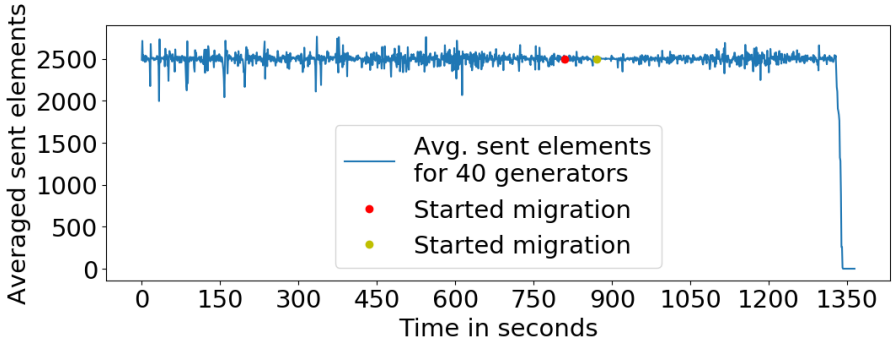
Fig. 5: Benchmark results for canceling and restoring NBQ8 via a savepoint

it takes until the modification mechanism triggers, this represents almost the worst case

waiting duration. The migration moves 24 operators in total across Task Managers, 8 person and 8 auction sources, as well as 8 window operators. The remaining 64 job sources are instead spilling to disk. The total state size for the job at the point of the migration is about 13.5 GB. However, only about 2.7 GB are actually stored and retrieved via the state backend for the window operators in about 21 s. Figure 6b shows that the throughput at the generators is not affected by the migration, since pausing and restarting the sources on different Task Manager takes less than 250 ms.



(a) Event timeline for migrating operators of NBQ8



(b) Average throughput at the 80 generators

Fig. 6: Benchmark results for performing a migration on NBQ8

6.5 Introducing new operators at runtime

To demonstrate the introduction of an operator at runtime, we use the SMQ (see Section 6.4.1). Its function filters the input stream by only letting every 5th record pass. We set the checkpointing interval to 1 s with concurrent checkpoints enabled. Figure 7 gives an overview of the modification duration. Most of the time for uploading the custom jar and submitting the command to Flink is spent establishing the connection. The actual preparation and sending of the trigger message takes less than 300 ms. It takes around 2 s for the corresponding checkpoint barrier to arrive, upon which all sources broadcast the new filter operator location. The source operators acknowledge the spilling to disk, while the map operators simultaneously update their inbound connections. As soon as all acknowledgments have been received, the new filter operator starts, which takes about 300 ms. In total, it takes only about 550 ms from the first spilling source to start and connect all 60 filter operators. The average throughput of the generators remains unaffected.

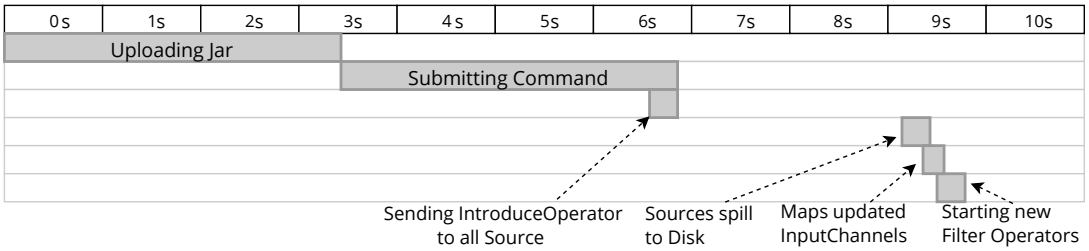


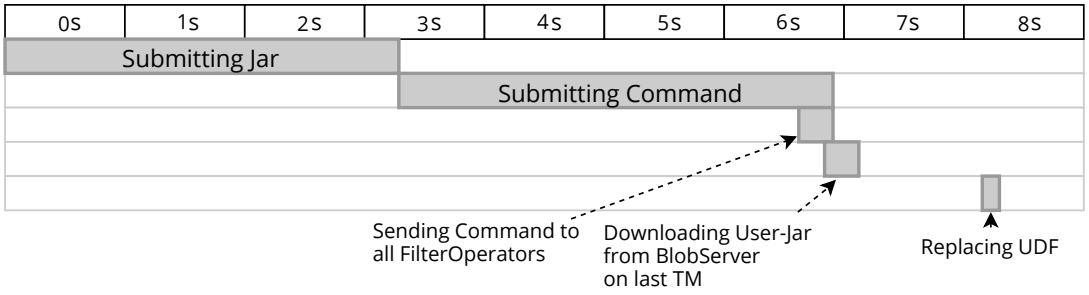
Fig. 7: Event timeline for introducing the filter function

6.6 Replacing the operator function at runtime

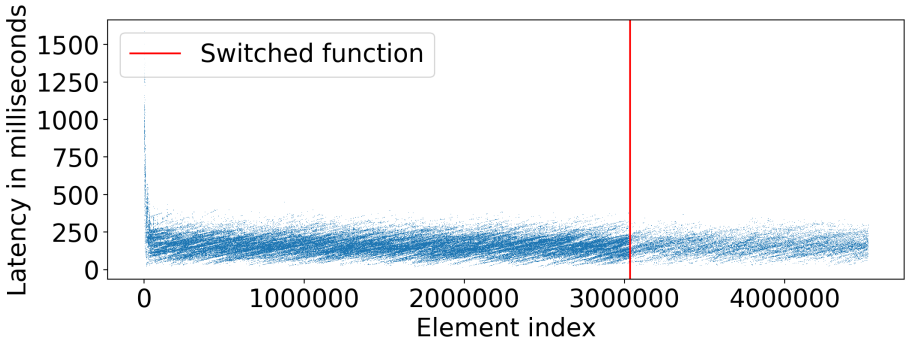
The last benchmark evaluates the replacement of a UDF of an operator at runtime. We reuse the SMQ with the filter operator already running. The source, filter, and map operators have 60 parallel instances, whereas the sink has 40. We set the checkpointing interval to 1 second with concurrent checkpoints enabled. Figure 8a shows the timeline of the introduction mechanism. Most of the time for uploading the new compiled code and submitting the command to Flink is spent on instantiating the client and establishing the connection. The actual preparation and sending of the trigger messages takes around 300 ms. Downloading the code on each Task Manager takes around 350 ms. Waiting for the upcoming checkpoint barrier, upon which all filter operators will replace their UDF with the one from the jar-file, takes around 1 s. In total, it takes a bit more than 8 s to replace the operator function. The synchronization on the checkpoint barriers may be omitted depending on the use case. Synchronizing only ensures that switching the function is aligned with the checkpoint mechanism, hence, it simplifies the recovery procedure. However, the synchronization is not necessary, e.g., all operators could immediately replace their function when receiving the trigger message. Replacing the operator function has nevertheless no impact on the throughput. Figure 8b shows the record latency for every 200th element in order to visualize the drop in incoming records after the function switch.

6.7 Discussion

The checkpointing settings of a given query have a big influence on the overall modification duration because of the eventual synchronization on every checkpoint barriers. For jobs with small state, fast checkpointing intervals are feasible, thus enabling the mechanism to react quickly on events, such as user input. For jobs with large state, the synchronization drastically limits the reaction time, however, the overall mechanism still performs better than the Flink baseline with savepoints. We are able to trigger all modification in less than 6 s for jobs with small state. For jobs with larger state, this duration increases up to about 60 s. This includes the heavy initialization overhead for the initial submission of the trigger as well as further messages. The experimental results show that it is feasible to migrate stateful operator instances at runtime with a negligible impact on the query performance. The operator state job demonstrated the migration of multiple, subsequent operators in 7.1 s. The modification duration grows with the number of subsequent operator instances, however, the impact is still low compared to the initial submission and the waiting duration. For the NBQ8, the migration mechanism showed big performance improvements, outperforming



(a) Event timeline for switching the filter operator



(b) Individual records latency

Fig. 8: Benchmark results for switching the filter function

the savepoint mechanism by a factor of 2.3, even including the long waiting and state transferring time. This is due to the long shutdown sequence of the savepoint mechanism, which transfers each operator state to the state backend. Table 1 summarizes the results of the migration operation. Additionally, Flink savepoint mechanism has the disadvantage of loosing in-flight records that flow from the sources during the savepoint procedure. When consuming from a persistent data source, such as Apache Kafka, we may still achieve exactly-once processing guarantees by replaying the missed records. However, the system looses records in the presence of non-persistent data source. Our migration mechanism prevents this from happening because stream sources never consume records that they cannot process. Furthermore, our mechanism does not introduce tangible throughput spikes at the generator because sources generally have very small state, which allows for quick migration.

Benchmark	Migrated State Size	Migrated Operators	Overall Duration	Waiting Time	Migration Duration
SMQ	17 kB	34	7.1 s	2 s	2.4 s
NBQ8	2.7 GB	24	81.8 s	54.9 s	23.7 s

Tab. 1: Migration overview for SMQ and NBQ8

Although introducing an operator into a running job suffers a few limitations, as mentioned in Section 5.2, it still enables many complex modification scenarios. Introducing operators enables altering the physical data flow of a DAG in under 10 s, depending on the checkpointing settings. With some further work, it would enable more sophisticated modification scenarios,

such as operator reordering. Replacing the operator function is a modification operation that enables fast tweaking of the streaming job in under 9 s with no performance impact. In the current version, users can leverage it to avoid restarting a running job, e.g., by adding additional logic to a UDF. It also enables more complex modification scenarios, such as tweaking the functionality of operator function based on internal or external triggers, such as runtime metrics.

7 Related work

Schneider et at. present an auto-parallelization mechanism for general-purpose, distributed data stream processing systems that deals with workload and resource aware scaling of operator instances as well as runtime state migration [Sc12]. We differentiate from their work as our migration mechanism is specific for queries that need exactly-once processing semantic as well large state. Wu et al. present ChronoStream, a distributed stream processing system specifically designed for elastic stateful stream computation [WT15]. ChronoStream achieves horizontal and vertical scalability in order to cope with workload fluctuation by treating internal state as a first-class citizen. The delta-compressed state of a task is separated into computational slices, which are duplicated and managed by a locality-aware placement mechanism across all machines. Through both mechanisms, they drastically lower the overhead caused by network I/O in case of migrations. Although their evaluation demonstrates linearly elasticity without sacrificing system performance or affecting collocated tenants, Ding et al. argue that their transactional migration protocol may cause incorrect results due to synchronization issues [Di15]. In contrast to our solution, they do not consider changes to the data flow and take placement decisions only based on the migration protocol, but not overall system performance. Heinze et al. [He14] deal with finding the right point in time to take scaling decisions through an online learning algorithm. They empirically compare their solution [He15] against local and global threshold-based mechanisms presented by Lorido et al. [LML14]. Although their results indicate that their auto-scaling technique performs better than the previous solutions, their solution only applies a simplistic migration protocol. In contrast, we design our protocols to cope with more demanding requirements, e.g., large state, exactly-once semantic. Nasir et al. [Na15b] propose the concept of *partial key grouping* in response to load imbalance caused by skewness in the key distribution of the input. Their approach monitors the number of tuples sent to two downstream instances. Each operator instance sends a tuple to the one with lower load estimation. As a result, tuples with the same key are routed to different parallel instances of the same operator. The authors extend their work to also include a mechanism for the "hottest" keys in the stream and assign more operator instances to those keys [Na15a]. In contrast to our approach, their solution does not alter the running query (i.e., no state migration) but only uses a streaming algorithm to determine the number of workers for "heavy hitter" keys, which is minimal yet sufficient for load balancing. Mai et al. introduce Chi [Ma18], a system that allows for dynamic reconfiguration of a running query. They use a migration mechanism similar to ours, however, they do not consider queries that result in large state and they do not further investigate such a specific scenario. Castro Fernandez

et al. also present a mechanism to scale up and down streaming topologies and to deal with operator failures [Ca13]. Our solution is different because of the explicit handling of large state and extended range of modifications, e.g., altering an execution plan at runtime. Systems based on micro-batching such as Apache Spark allow for reconfiguration at the end of each micro-batch, however, this results in higher latency and lower throughput because of synchronization. Finally, we point out that our approach is orthogonal to public and private cloud [Am] and resource managers [Hi11, Va13] because we tackle fault-tolerance and resource elasticity at application-level, whereas they isolate running applications through virtual machines or containers.

8 Conclusion

This paper examines the feasibility of modifying the execution of a running streaming job in Apache Flink. Our major contribution is the implementation of a modification protocol that enables three types of modifications, namely migrating operator instances, introducing new operator instances, as well as changing an operator's UDF. We evaluate the protocol using one custom benchmark and the Nexmark Benchmark. The results show that migrating operators for jobs with small state is as fast as using Flink's savepoint mechanism. Migrating operators of a job with 15 GB of state even outperforms the savepoint mechanism by a factor of 2.3. Furthermore, our migration mechanism solves the problem of data loss during job restart that arises when not consuming records from a persistent data source. Changing the data flow itself by either introducing new or replacing existing operators can be performed in less than 10 s and 8 s, respectively. The modification protocol opens the door for a variety of further enhancements to a running streaming job. It allows for online optimizations of the running job that were not possible before. An example is up- and down-scaling operator instances, which enables the SPE to dynamically adapt to changes in the incoming workload without halting the whole job. Lastly, removing the synchronization on the checkpoint barriers can significantly reduce the overall modification duration and increase the responsiveness of our mechanism.

Acknowledgment. This work was funded by the European Union through PROTEUS (ref. 687691) and STREAMLINE (ref. 688191).

References

[Al14] Alexandrov, A.; Bergmann, R.; Ewen, S.; Freytag, .; Hueske, F.; Heise, A.; Kao, O.; Leich, M.; Leser, U.; Markl, V. et al.: The Stratosphere Platform for Big Data Analytics. The VLDB Journal, 2014.

[Am] Amazon EC2. https://aws.amazon.com/ec2/.

[Ca13] Castro Fernandez, R.; Migliavacca, M.; Kalyvianaki, E.; Pietzuch, P.: Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. ACM SIGMOD, 2013.

[Ca17] Carbone, P.; Ewen, S.; Fóra, G.; Haridi, S.; Richter, S.; Tzoumas, K.: State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. VLDB, 2017.

[DG04] Dean, J.; Ghemawat, S.: MapReduce: simplified data processing on large clusters. USENIX OSDI, 2004.

[Di15] Ding, J.; Fu, T.; Ma, R.; Winslett, Ma.; Yang, Y.; Zhang, Z.; Chao, H.: Optimal Operator State Migration for Elastic Data Stream Processing. CoRR, abs/1501.03619, 2015.

[DM17] Del Monte, B.: Efficient Migration of Very Large Distributed State for Scalable Stream Processing. VLDB PhD Workshop, 2017.

[HBS73] Hewitt, C.; Bishop, P.; Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI, 1973.

[He14] Heinze, T.; Pappalardo, V.; Jerzak, Z.; Fetzer, C.: Auto-scaling techniques for elastic data stream processing. In: IEEE ICDE Workshops. 2014.

[He15] Heinze, T.; Ji, Y.; Roediger, L.; Pappalardo, V.; Meister, A.; Jerzak, Z.; Fetzer, C.: FUGU: Elastic Data Stream Processing with Latency Constraints. IEEE Data Eng. Bull., 2015.

[Hi11] Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.; Katz, R.; Shenker, S.; Stoica, I.: Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In: USENIX NSDI. 2011.

[Hi14] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R.: A Catalog of Stream Processing Optimizations. ACM CSUR, 2014.

[Hü15] Hüske, F.: Specification and Optimization of Analytical Data Flows. PhD thesis, TU Berlin, 2015.

[Ka18] Karimov, J.; Rabl, T.; Katsifodimos, A.; Samarev, R.; Heiskanen, H.; Markl, V.: Stream Processing Performance in Online Game Scenarios. IEEE ICDE, 2018.

[Ku15] Kulkarni, S.; Bhagat, N.; Fu, M.; Kedigehalli, V.; Kellogg, C. et al.: Twitter Heron: Stream Processing at Scale. ACM SIGMOD, 2015.

[LML14] Lorido, T.; Miguel, J.; Lozano, J.: A review of auto-scaling techniques for elastic applications in cloud environments. Journal of grid computing, 2014.

[Ma18] Mai, L.; Zeng, K.; Potharaju, R.; Xu, L.; Venkataraman, S.; Costa, P.; Kim, T.; Muthukrishnan, S.; Kuppa, V.; Dhulipalla, S.; Rao, S.: Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. VLDB, 2018.

[Na15a] Nasir, M.; Morales, G.; Kourtellis, N.; Serafini, M.: When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing. CoRR, abs/1510.05714, 2015.

[Na15b] Nasir, M; Morales, G.; Soriano, D.; Kourtellis, N.; Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. IEEE ICDE, 2015.

[Sc12] Schneider, S.; Hirzel, M.; Gedik, B.; Wu, K.: Auto-parallelizing stateful distributed streaming applications. ACM PACT, 2012.

[ScZ05] Stonebraker, M.; Çetintemel, U.; Zdonik, S.: The 8 Requirements of Real-time Stream Processing. ACM SIGMOD, 2005.

[To14] Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J.; Kulkarni, S.; Jackson, J.; Gade, K.; Fu, M.; Donham, J. et al.: Storm@ twitter. In: ACM SIGMOD. 2014.

[Tu18] Tucker, P.; Tufte, K.; Papadimos, V.; Maier, D.: NEXMark - A Benchmark for Queries over Data Streams. 2018.

[Va13] Vavilapalli, V.; Murthy, A.; Douglas, C.; Agarwal, S.; Konar, M. et al.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: ACM SOCC. 2013.

[WT15] Wu, Y.; Tan, K. L.: ChronoStream: Elastic stateful stream computation in the cloud. IEEE ICDE, 2015.

[Za13] Zaharia, M.; Das, T.; Li, H.; Hunter, T.; Shenker, S.; Stoica, I.: Discretized Streams: Fault-tolerant Streaming Computation at Scale. ACM SOSP, 2013.