Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines

Bonaventura Del Monte[1,2]   Steffen Zeuch[1,2]   Tilmann Rabl[3]   Volker Markl[1,2]

[1]Technische Universität Berlin   [2]DFKI GmbH   [3]HPI, University of Potsdam
bdelmonte@tu-berlin.de   steffen.zeuch@dfki.de   tilmann.rabl@hpi.de   volker.markl@tu-berlin.de

ABSTRACT

Scale-out stream processing engines (SPEs) are powering large big data applications on high velocity data streams. Industrial setups require SPEs to sustain outages, varying data rates, and low-latency processing. SPEs need to transparently reconfigure stateful queries during runtime. However, state-of-the-art SPEs are not ready yet to handle on-the-fly reconfigurations of queries with terabytes of state due to three problems. These are network overhead for state migration, consistency, and overhead on data processing.

In this paper, we propose *Rhino*, a library for efficient reconfigurations of running queries in the presence of very large distributed state. Rhino provides a handover protocol and a state migration protocol to consistently and efficiently migrate stream processing among servers. Overall, our evaluation shows that Rhino scales with state sizes of up to TBs, reconfigures a running query 15 times faster than the state-of-the-art, and reduces latency by three orders of magnitude upon a reconfiguration.

1 INTRODUCTION

Over the past years, increasingly complex analytical queries on real-time data have made Stream Processing Engines (SPEs) an important component in the big data toolchain. SPEs power stateful analytics behind popular multimedia services, online marketplaces, cloud providers, and mobile games. These services deploy SPEs to implement a wide range of use-cases, e.g., fraud detection, content recommendation, and user profiling [5, 23, 27, 34, 35, 38, 41]. Furthermore, cloud providers offer fully-managed SPEs to customers, which hide operational details [1, 2]. State in these applications scales with the number of users, events, and queries and can reach terabyte sizes [27]. These state sizes originate from intermediate results of large temporal aggregations or joins. We consider state as very large when it exceeds the aggregated main-memory available to the SPE.

To run applications, SPEs have to support continuous stateful stream processing under diverse conditions, e.g., fluctuating data rates and low latency. To handle varying data rates and volumes, modern SPEs scale out stateful query processing [39]. Furthermore, SPEs have to transparently handle faults and adjust their processing capabilities, regardless of failures or data rates fluctuatations. To this end, they offer runtime optimizations for running query execution plans (QEPs), resource elasticity, and fault tolerance through QEP reconfigurations [5, 6, 19, 30, 35, 39, 45, 47, 49]. State management is necessary to enable fault-tolerance, operator rescaling, and query re-optimization, e.g., load balancing. Therefore, scale-out SPEs require efficient state management and on-the-fly reconfiguration of running queries to quickly react to spikes in the data rate or failures.

The reconfiguration of running stateful queries in the presence of very large operator state brings a multifaceted challenge. First, *network overhead*: a reconfiguration involves state migration between workers over a network, which results in more resource utilization and latency proportional to state size. As a result, this migration overhead increases the end-to-end latency of query processing. Second, *consistency*: a reconfiguration has to guarantee exactly-once processing semantics through consistent state management and record routing. A reconfiguration must thus alter a running QEP without affecting result correctness. Third, *processing overhead*: a reconfiguration must have minimal impact on performance of query processing. An SPE must continuously

and robustly process stream records as if no reconfiguration ever occurred.

Today, several industrial and research solutions provide state migration. However, these solutions restrict their scope to small state sizes or offer limited on-the-fly reconfigurations. Apache Flink [3, 5], Apache Spark [46, 47], and Apache Samza [35], enable consistency but at the expense of performance and network throughput. They support large, consistent operator state but they restart a running query for reconfiguration [5, 35, 47]. Research prototypes, e.g., Chi [30], ChronoStream [45], FUGU [19], Megaphone [21], SDG [15], and SEEP [6] address consistency and performance but not network overhead. They enable fine-grained reconfiguration but support smaller state sizes (i.e., tens of gigabytes).

In this paper, we show that representatives of the above systems, i.e., Flink and Megaphone, do not cope well with large state sizes and QEP reconfigurations. Although the above systems support stateful processing, they fall short in providing efficient large state migration to enable on-the-fly QEP reconfigurations.

To bridge the gap between stateful stream processing and operational efficiency via on-the-fly QEP reconfigurations and state migration, we propose *Rhino*. Rhino is a library for efficient management of very large distributed state compatible with SPEs based on the streaming dataflow paradigm [2].

Rhino enables on-the-fly reconfiguration of a running query to provide resource elasticity, fault tolerance, and runtime query optimizations (e.g., load balancing) in the presence of very large distributed state. To the best of our knowledge, Rhino is the first system to specifically address migration of large state. Although state-of-the-art systems provide fine-grained state migration, Rhino is optimized for reconfiguration of running queries that maintain large operator state. In particular, Rhino proactively migrates state so that a potential reconfiguration requires minimal state transfer. Rhino applies a state-centric, proactive replication protocol to asynchronously replicate the state of a running operator on a set of SPE workers through incremental checkpoints. Furthermore, Rhino applies a handover protocol that smoothly migrates processing and state of a running operator among workers. This does not halt query execution and guarantees exactly-once processing. In contrast to state-of-the-art SPEs, our protocols are tailored for resource elasticity, fault tolerance, and runtime query optimizations.

In our evaluation, Rhino reduces reconfiguration time due to state migration by 50x compared to Flink and 15x compared to Megaphone, as shown in Figure 1. Furthermore, Rhino shows a reduction in processing latency by three orders of magnitude for a reconfiguration with large state migration. We show that Rhino does not introduce overhead on query processing, even when state is small. Finally, we show that Rhino can handle state migration in a multi-query

scenario with a reduction in reconfiguration time by one order of magnitude. Overall, Rhino solves the multifaceted challenge of on-the-fly reconfiguration involving large (and small) stateful operators.

In this paper, we make the following contributions:

- We introduce the full system design of Rhino and the rationale behind our architectural choices.
- We devise two protocols for proactive large state migration and on-the-fly reconfiguration of running queries. We design these protocols to specifically handle migrations of large operator state. To this end, Rhino proactively migrates state to reduce the amount of state to move upon a reconfiguration.
- With Rhino, we enable resource elasticity, fault tolerance, and runtime re-optimizations in the presence of very large distributed state. Rhino's state migration is tailored to support these three features.
- Using the NEXMark benchmark suite [40] as representative workload, we validate our system design at terabyte scale against state-of-the-art SPEs.

We structure the remainder of this paper as follows. In § 2, we provide background knowledge. We describe the system design of Rhino in § 3 and its replication and handover protocols in § 4. We evaluate Rhino in § 5 and provide an overview of related work in § 6. Finally, we summarize our work in § 7.

2 BACKGROUND

In this section, we introduce the concepts of *stateful stream processing* (see § 2.1) and *state management* (see § 2.2) as the underlying foundation of our work.

2.1 Stateful Stream Processing

Modern scale-out SPEs leverage the dataflow execution model [8] to run continuous queries. This system design enables the execution of queries on a cluster of servers. A query consists of stateful and stateless operators [1, 2]. A QEP is represented as a weakly-connected graph with special vertices called source and sink operators. Sources allow for stream data ingestion, whereas sinks output results to external systems. To process data, users define transformations through higher-order functions and first-order functions (UDFs). Operators and UDFs support internal, mutable state (e.g., windows, counters, and ML models).

In the rest of this paper, we follow the definitions introduced by Fernandez et al. [6] to model streaming processing. In this model, a query is a set of logical operators. Each logical operator o consists of p_o physical instances $o_1, \ldots, o_i, \ldots, o_{p_o}$. A stream represents an infinite set of records $r = (k, t, a)$, where $k \in K$ is a partitioning key, K is the key space, t is a

strictly monotonically increasing timestamp, and a is a set of attributes. An operator produces and consumes n and m streams, respectively.

To execute a query, an SPE maps the parallel instances of operators to a set of worker servers. Each operator has its producer operators (*upstream*) and its consumer operators (*downstream*). A parallel instance receives records from upstream operators through channels according to an exchange pattern, e.g., hash-partitioning. An inter-operator channel is durable, bounded, and guarantees FIFO delivery. A channel contains fixed-sized buffers, which store records of variable size and control events, e.g., watermarks [2]. A channel imposes a total order on its records and control events based on their timestamp, however, instances with multiple channels are non-deterministic. As a result, there exists a partial order among records consumed by an instance.

A stateful operator o holds its state S_o, which is a mutable data set of key-value pairs (k, v). To scale out, an SPE divides S_o in disjoint partitions and assigns a partition $S_{o_i}^t$ to an instance o_i using k as partitioning key. With t, we denote the timestamp of the last update of a state partition. A value v of the state is an arbitrary data type. A parallel instance o_i processes every record in a buffer, reacts to control events, emits records, and reads or updates its state $S_{o_i}^t$.

2.2 State Management in SPEs

Today's SPEs provide state management through two techniques: *state checkpointing* and *state migration*.

2.2.1 State Checkpointing.
State checkpointing enables an SPE to consistently persist the state of each operator, recover from failures, and reconfigure a running query.

Fernandez et al. propose a checkpointing approach [6, 7] that stores operator state and in-flight records. This produces local checkpoints that are later used to reconfigure or recover operators. In contrast, Carbone et al. [5] propose a distributed protocol for global, epoch-consistent checkpoints. It captures only operator state (by skipping in-flight records) and does not halt query processing. This protocol divides the execution of a stateful query into *epochs*. An epoch is a set of consecutive records that the SPE processes in a time frame. If query execution fails before completing the $i + 1$-th epoch, the SPE rollbacks to the state of the i-th epoch.

The distributed protocol of Carbone et al. [5] assumes reliable data channels and FIFO delivery semantics. In addition, it assumes upstream backups of ingested streams, i.e., sources can consume records again from an external system. Finally, it assumes checkpoints to be persisted on a distributed file system for availability. The drawback of this protocol is the full restart of a query upon a reconfiguration.
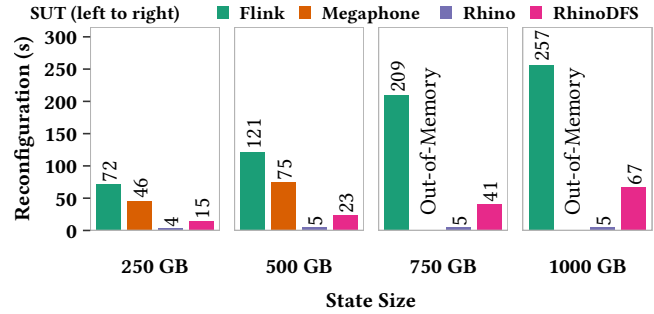

Figure 1: Time spent to reconfigure the execution of NBQ8.

2.2.2 State Migration.
State migration enables the handover of the processing and the state of a key partition of an operator among workers [21]. This enables on-the-fly query reconfiguration to support fault-tolerance, resource elasticity, and runtime optimizations.

Megaphone is the latest approach to state migration [21]. Megaphone is a state migration technique built on Timely Dataflow [31] that enables consistent reconfiguration of a running query through fine-grained, fluid state migration. Megaphone introduces two migrator operators for each migratable operator in a QEP to route records and state according to a planned migration. It requires from an SPE: 1) out-of-band progress tracking, i.e., operators need to observe each other's progress and 2) upstream operators need to access downstream state. Its authors state that Megaphone can be compatible with other SPEs but with some overhead.

3 SYSTEM DESIGN

In this section, we present the design of *Rhino*, a library that integrates stateful stream processing with on-the-fly query reconfiguration. First, we present a benchmark that shows the shortcomings of current state management techniques for migration of large operator states (see § 3.1). Second, driven by our findings, we propose an architectural revision for scale-out SPEs to handle large state in § 3.2. Finally, we provide an overview of the protocols and components that we use to implement this architectural change and build Rhino (see § 3.3 to § 3.5).

3.1 Benchmark state migration techniques

With large state sizes in mind, we run a benchmark to assess the stateful capabilities of modern scale-out SPEs. We select Flink as a representative, industrial-grade SPE due to its wide adoption and its built-in support for state. We assume a distributed file system (DFS) [18, 37, 43] to be commonly deployed with scale-out SPEs for checkpoint storage [5, 35]. Furthermore, we choose Megaphone [21] as it is the most recently proposed scale-out state migration technique. Chi [30]

is another SPE that provides state migration, yet it is not available for public access at the time of writing.

In Figure 1, we report the impact on processing latency during a reconfiguration with varying state size (i.e., ranging from 250 GB to 1 TB) and show results in Figure 1. We observe that Flink requires a full query restart, which results in a significant latency spike that hinders processing performance. Our time breakdown, which we fully describe in § 5.2.1, indicates that Flink spends the majority of the time in state materialization from a previous checkpoint prior to resuming query processing. Upon a reconfiguration, a parallel instance of a stateful operator retrieves in bulk its new state from DFS. Every instance quickly retrieves its local blocks (if any), yet the materialization of remote blocks entails network transfer. As a result, state retrieval introduces additional latency because every instance pulls its state from multiple workers.

Although Megaphone provides fine-grained state migration, we find that it does not handle large state size. All its executions in our benchmark with more than 500 GB of state terminated with an out-of-memory error. By inspecting its source code [20], we consider the lack of memory management to support state migration as the main responsible for this error. However, we observe that Megaphone can successfully complete fine-grained state migrations, if state fits main memory. Note that the introduction of a data structure that supports out-of-core storage, e.g., a key value store (KVS), would enable Megaphone to store larger-than-memory state. However, this does not solve the problem of large state migration as Megaphone migrates key-value pairs in one batch.

3.2 Rhino

In the previous section, we show that current SPEs do not properly support large operator state when reconfiguring a running query. To fill this gap, we build Rhino to support large state and enable on-the-fly reconfiguration of running queries by design. Rhino is thus tailored to provide fine-grained fault-tolerance, resource elasticity, and runtime optimizations, e.g., load balancing.

The design of Rhino focuses on efficient reconfigurations and migration of large state. To this end, Rhino introduces three key techniques. First, Rhino proactively migrates state such that a reconfiguration requires minimal state transfer. In contrast to the state-of-the-art, Rhino proactively and periodically persists the state of an operator instance on exactly r workers of the SPE. Rhino thus executes fast reconfiguration (handover) from an origin instance to a target instance that runs on one of the r workers. Second, to control the state size to be migrated, Rhino asynchronously replicates incremental checkpoints of the state. As a result, Rhino migrates only

the last incremental checkpoint of the state of an operator upon a handover. In contrast, other systems transfer state in bulk [5, 21]. Third, Rhino introduces consistent hashing with virtual nodes to further partition the key partition of an operator instance. In Rhino, virtual nodes are the finest granularity of a reconfiguration.

3.3 Components Overview

Rhino introduces four components in an SPE, i.e., a *Replication Manager (RM)*, a *Handover Manager (HM)*, a distributed runtime, and modifications to existing components.

Replication Manager. The replication manager runs on the coordinator of the SPE and builds the replica groups of each instance based on the bin-packing algorithm. After that, the RM assigns replicas to workers. We implement this component with less that 1K lines of code (loc).

Handover Manager. The handover manager serves as the coordinator for in-flight reconfiguration with state transfer. Based on a human or automatic decision-maker (e.g., Dhalion [17], DS2 [24]), our HM starts a reconfiguration. After that, it monitors Rhino's distributed runtime (see below) for a successful and timely completion of the triggered handover. We implement this component in about 1K loc.

Distributed Runtime. The distributed runtime runs on the workers of the SPE. This runtime uses our two application-level protocols for handover and state-centric replication. We implement this runtime with less than 5K loc.

Modifications. Besides introducing new components, Rhino also extends stateful operators to 1) receive and broadcast control events on their data channel, 2) buffer in-flight record of specific data channels, 3) reconfigure inbound and outbound data channels, and 4) ingest checkpointed state for one or more virtual nodes of an instance. In total, these modifications require about 2K loc in Flink.

3.4 Host System Requirements

We design Rhino as a library that can be deployed on top of a scale-out SPE. However, the target SPE has to fulfill the following requirements.

R1: Streaming dataflow paradigm. Our handover protocol requires the SPE to follow the streaming dataflow paradigm because Rhino relies on markers that flow along with records from source operators. Common SPEs feature this paradigm with two processing models: the record-at-a-time model, adopted by SPEs such as Apache Flink and Apache Samza [5, 35], and the bulk synchronous processing model, adopted by SPEs such as Apache Spark Streaming [47]. With a record-at-a-time model, Rhino can inject a marker in the record flow at any point in time. In contrast, the bulk synchronous processing model introduces coarse-grained synchronization at the end of every micro-batch. Rhino could

piggyback a handover on this synchronization barrier to enable reconfigurations at the expense of increased latency.

R2: Consistent hashing with virtual nodes. Our handover protocol requires techniques for fine-grained reconfiguration. A fine-grained reconfiguration requires fine-grained state migration and record rerouting. To this end, Rhino uses consistent hashing with virtual nodes [29] instead of traditional consistent hashing [12, 25]. This enables Rhino to reassign state to instances and route records at a finer granularity, upon a reconfiguration. Thus, Rhino further partitions streams and state into a fixed number of virtual nodes, which are the finest granularity of a reconfiguration.

R3: Mutable State. It is necessary for Rhino that each operator supports local, mutable state along with distributed checkpointing [5, 10]. In addition, Rhino requires read and write access to the internal state of a parallel instance to update it upon a handover. Embedded KVS, e.g., RocksDB [13] and FASTER [9], provide mutable state compatible with Rhino. We do not consider external KVS, e.g., Cassandra [29] and Anna [44], as they would introduce extra synchronization in a distributed environment.

In this paper, we select Apache Flink as host system to deploy Rhino as it meets R1 and R3 (via RocksDB) by design and requires minimal changes to meet R2. However, Rhino is compatible with any SPE that fulfills the above requirements, e.g., Apache Spark, Apache Storm, and Google Dataflow.

3.5 Benefits of Rhino

In this section, we show how current SPEs could benefit from Rhino and its ability to reconfigure running queries at a fine granularity and in different scenarios. In particular, Rhino enables higher operational efficiency through fine-grained load-balancing (see § 3.5.1), resource elasticity (see § 3.5.2), and fault-tolerance (see § 3.5.3). Note that the handover protocol ensures that query execution preserves exactly-once processing semantics during all operations.

3.5.1 Load balancing. Rhino enables fine-grained load balancing, which is missing in current SPEs [21, 33]. Load balancing is beneficial when the SPE detects that a physical instance O is overwhelmed (e.g., due to data skewness) but instance T is underloaded. In this case, the SPE requests Rhino's HM to migrate the processing and state of some virtual nodes of O to T, which runs on a worker that has a copy of the state of O. After that, the HM triggers a handover that involves these virtual nodes. When the handover completes, T takes over the processing of migrated virtual nodes of O.

3.5.2 Resource Elasticity. Rhino enables resource elasticity for current SPEs and handles rescaling as a special case of load balancing. To this end, Rhino moves some virtual nodes

of a running instance to a newly spawned instance. In particular, Rhino enables adding operator instances on an in-use worker (i.e., vertical scaling) and deploys more instances on newly provisioned workers (i.e., horizontal scaling).

Overall, Rhino follows a similar course of action for resource elasticity as for load balancing. Rhino defines O and T as the origin and target instances, respectively. In the case of vertical scaling, Rhino assumes that the SPE deploys T on an in-use worker, which has a copy of the state of O. In the case of horizontal scaling, Rhino provisions a new worker, which requires a bulk copy of the state. The cost of a bulk transfer is mitigated by early provision of workers and parallel copies.

3.5.3 Fault Tolerance. Rhino enables fine-grained fault tolerance that results in faster recovery time (see Figure 1). Upon the failure of O, the SPE requests the HM to migrate O on a the workers that stores a copy of its state and can run a new instance T. Thus, Rhino triggers a handover that instructs T to start processing using the last checkpointed state of O. In addition, the handover instructs upstream and downstream operators to rewire their channels to connect to T.

4 THE PROTOCOLS

In this section, we describe Rhino's protocols in detail. We first introduce the handover protocol in § 4.1 and then the replication protocol in § 4.2.

4.1 Handover Protocol

In this section, we define our handover protocol (see § 4.1.1), its steps (see § 4.1.2), and correctness (see § 4.1.3).

4.1.1 Protocol Description. To enable fine-grained reconfiguration of a running query, our handover protocol defines three aspects. First, it defines the concept of configuration epochs for a running stateful query. Second, it defines how to transition from one configuration epoch to the next. Third, it defines what properties hold during a handover.

Configuration epoch. Our protocol discretizes the execution of a query into configuration epochs of variable length. A configuration epoch $E_{a,b}$ is the non-overlapping time interval between two consecutive reconfigurations that occur at time a and b, respectively. An epoch is independent from any windowing semantics or checkpoint epochs. The first epoch begins with the deployment of a query. During an epoch, each instance processes records using fixed parameters, i.e., assigned worker, input and output channels, state partition, and assigned virtual nodes. The task of the handover is to consistently reconfigure the parameters of a running instance. This involves state migration as well as channel rewiring. Furthermore, it triggers the transition to the next epoch.
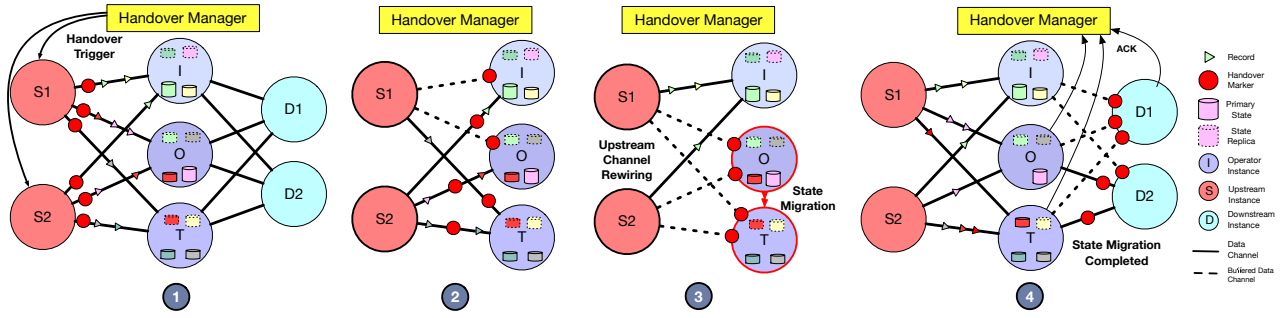
Figure 2: Steps (left to right) of the Handover Protocol of Rhino.

Handover. A handover reconfigures a non-source operator of a running query at time t. This ends the epoch $E_{a,t}$ and starts $E_{t,b}$. As a result, records with timestamp in interval $[a, t]$ are processed with a configuration and records with timestamp in $[t, b]$ are processed with a new configuration.

A handover involves 1) the propagation of handover markers, 2) state migration, and 3) channel rewiring. A handover marker (1) is a control event that flows from source operators to the instances to reconfigure through all dataflow channels. A handover marker carries the configuration update that involves the origin and target instances and their upstream and downstream instances. This is inspired by Chi's idea of embedding control messages in the dataflow to reconfigure queries [30]. As soon as the origin instance receives a marker on all its inbound channels, it migrates state (2) to the target instance. When the target instance receives its markers and the state from the origin instance, it takes over processing. Channel rewiring (3) ensures that records are consistently routed among upstream, downstream, origin, and target instances.

Epoch alignment. To perform a consistent reconfiguration at time t, we use handover markers to induce an instance-local synchronization point. To this end, we use an epoch alignment algorithm, similarly to the checkpointing approach of Carbone et al. [5]. Recall that an instance nondetermistically polls records and control events from its channels. Therefore, it is necessary to ensure a total order among otherwise partially ordered records. When a marker with timestamp t arrives at an instance from a channel, it signals that no record older than t is to be expected from that channel. Since newer records have to be processed with the new configuration, the instance has to buffer upcoming records on that channel to process them later. When an instance receives all markers, no record older than t is in-flight. As a result, the instance reaches its synchronization point and the reconfiguration takes place. Target instances, however, must wait for state migration to complete.

4.1.2 Protocol Steps. In this section, we define the steps of our handover protocol. To do so, we first provide an example and then a formal description of each step.

Example. The SPE requests Rhino to migrate the processing of red records from origin instance O to target instance T (see Figure 2). Instances O and T run on distinct workers. To fulfill the SPE's request, the HM triggers a handover and source instances S1 and S2 inject a handover marker into the dataflow ①. When receiving a marker on a channel, instances O and T buffer records further arriving on that channel ②. Upon receiving all markers, O migrates the red state ③. At the completion of the state transfer, O and T acknowledge the HM ④. Next, upstream instances S1 and S2 send red records to T instead of O. Finally, instance O, T, D1, and D2 acknowledge the HM.

Step 1. Assume that running instances completed a checkpoint at time t'. Based on a policy, an SPE triggers a handover at time t between the origin instance O and the target instance T for a virtual node (k_l, k_q). The handover assumes previously checkpointed state of O to be replicated on the worker running T and that no checkpoint is in-flight.

Step 2. The protocol defines the following actions for an operator instance, which we summarize in Figure 2.

① Source operators injects a handover marker h_t in their outbound channels.

② When an instance I receives h_t on one of its inbound channel, it buffers incoming records on that channel.

③ Upon receiving h_t on all its inbound channel, I broadcasts a handover marker on its outbound channels and performs Step 3.

④ When T receives the checkpointed state for (k_l, k_q), it processes buffered records that arrived after m_t.

Step 3. According to its position (upstream or downstream) with O and T, I reacts using one of the following routines. First, if I is an *upstream* instance, it rewires the output channels for (k_l, k_q) to send records to T. Second, if I is a *downstream* instance and T is a new instance, it rewires its inbound channels to process records from T. Third, if I is the origin instance, it triggers a local checkpoint t, which Rhino transfers to T, and releases unneeded resources. Fourth, if I is the target instance, it loads checkpointed state of O and then consumes buffered records. If origin instance has failed, Rhino does not migrate state and relies on upstream backup

to replay records, if necessary. Note that operators are aware of an in-flight handover and ignore seen records based on their timestamps. As a result, O processes records of $(k_l, k_q]$ with a timestamp smaller than t, whereas T processes records with a timestamp greater than t.

Step 4. As soon as an instance completes its steps, it acknowledges the HM. When all instances successfully send an acknowledgment, the HM marks the handover as completed.

Our handover protocol needs upstream backup when a failure occurs. The upstream backup can be a source operator or windowed operator that keeps in-flight windows. They can replay records from an external system or the window state. In the latter case, downstream operators must acknowledge when it is safe to delete the content of a window.

Note that the handover protocol is not fault-tolerant. A failure that occurs during a handover may restart the protocol. We plan to make our protocol fault-tolerant as future work, e.g., via fine-grained, upstream buffering of records.

4.1.3 Correctness guarantees. A handover between two instances has the following correctness guarantees.

THEOREM 1. Consider a handover that migrates the state S^{t-1} of the virtual node $(k_l, k_q]$ from O to T at timestamp t. The protocol guarantees that: 1) T receives S^{t-1} at t and then processes records with keys in $(k_l, k_q]$ and timestamps greater than t and 2) the handover completes in finite time.

Proof. Let $t - 1$ be the timestamp of the last processed record and S^{t-1} the state of O and assume that T has received all previous incremental checkpoints from O. Recall that an instance is aligned at time t if it has processed all records bearing timestamps smaller than t. In our setting, the alignment at time t occurs when an instance receives the handover marker h_t on its channels and has no in-flight checkpoint. The alignment at an upstream instance results in the routing to T of records with keys in $(k_l, k_q]$ and timestamp greater than t. Note that these records are not processed until the handover on T occurs. The alignment at t on O results in a incremental checkpoint of the state S^{t-1}, which is migrated to T. The alignment of T occurs upon receiving markers on its channels and the incremental checkpoint of S^{t-1} from O. Thus, T receives S^{t-1} at time t and only then processes records with keys in $(k_l, k_q]$ and timestamps greater than t (1). State transfer, channel rewiring, and acknowledgment to coordinator are deterministic operations. In addition, channels are FIFO and durable, thus, they eventually deliver handover markers to instances. As a result, a handover completes in finite time (2).

4.2 Replication Protocol

In this section, we describe our state replication protocol and how we use it to guarantee timely handovers between two stateful instances.
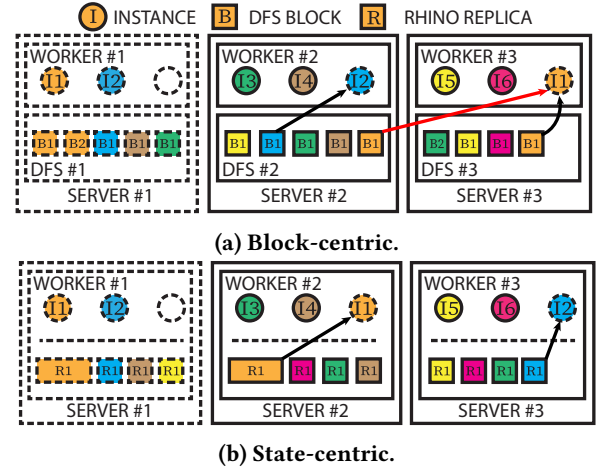

(a) Block-centric.

(b) State-centric.

Figure 3: Block-centric vs. state-centric replication. Black and red arrows indicate local and remote fetching, respectively.

4.2.1 Protocol description. Our state-centric replication protocol enables us to asynchronously replicate the local checkpointed state (primary copy) of a parallel instance of an operator on r workers of the SPE (secondary copies). It assumes that the SPE periodically triggers global checkpoints. Rhino replaces the *block-centric* replication of traditional DFS with a *state-centric* protocol. The intuition behind our protocol is that Rhino proactively migrates state of each instance through incremental checkpoints to ensure that a target instance during a handover has the latest copy of its state. The protocol assumes each worker to have dedicated storage units to locally fetch checkpointed state. This allows the SPE upon a reconfiguration to spawn or reuse an instance on a worker that already stores the state for this instance. Traditional DFSs and disaggregated storage do not guarantee this property as block placement is transparent to client systems. As a result, the SPE must query the DFS to retrieve necessary blocks. The retrieval is either local or remote depending on the placement of each block.

In Figure 3, we show an example of configuration of block-centric and state-centric replication. With block-centric replication, the state of each operator instance is split into multiple blocks, which are replicated on 3 servers. With state-centric replication, the state of each operator instance is entirely replicated on 2 servers. This improves replica fetching upon a recovery. When server 1 fails, its stateful instances are to resume on servers 2 and 3. With block-centric replication, server 3 fetches B1 and B2 from server 2. With state-centric replication, fetching is local.

4.2.2 Protocol phases. Our state-centric replication protocol consists of two phases. The first phase of the protocol instructs how to build *replica groups* for each stateful instance. A *replica group* is a chain of r workers, which owns

the secondary copy of the state of a parallel instance, i.e., the state of all virtual nodes. The second phase defines how to perform asynchronous replication of a primary copy to its *replica group*.

Phase 1: Protocol Setup. The RM configures the protocol at deployment-time of a query or upon any change in a running QEP. Through a placement algorithm, e.g., bin packing, the RM creates a replica group of r distinct workers for each stateful instance. In this paper, we assume that workers have equal capacities and use all available workers for the replication of the global state. However, we leave the investigation of other placement algorithms and heuristics as future work. Overall, the RM assigns to every worker a set of secondary copies to maintain.

Phase 2: Distributed Replica Transfer. Our distributed replication runtime implements a state-centric replication protocol that follows the primary/secondary replication pattern [4, 36] with chain replication [42]. We choose chain replication as it guarantees parallel replication with high network throughput. In addition, we use credit-based flow control [28] for application-level congestion control.

Each member of the chain asynchronously stores the received data to disk and forwards the data to its successor in the chain. As soon as the last worker in a chain stores the copy of the checkpointed state to disk, it acknowledges its predecessor in the chain, which does the same. When the owner of the primary copy receives the acknowledgment, it marks the checkpoint as completed and releases unnecessary resources. The replication of a checkpoint completes successfully only if the runtime successfully acknowledges all copies within a timeout. When the replication process is completed, Rhino marks the checkpoint as completed.

4.2.3 Correctness guarantees. Our replication protocol strictly follows the fault-model behind chain replication [42]. We assume *fail-stop* workers that halt upon a failure, which they report to replication manager (RM). Our protocol distinguishes two failures: 1) failure on a running operator and 2) failure on the replica group. If the primary owner fails, the RM halts the replication for the failed operator and only when it resolves the failure, the replication resumes. If a worker in a replica group fails, the RM removes the failed worker and replaces it with another worker.

5 EVALUATION

In this section, we experimentally validate the system design of Rhino and compare it against three systems under test (SUTs [32]): Apache Flink, Megaphone, and RhinoDFS (a variant that uses HDFS for state migration). First, we describe the setup of our evaluation in § 5.1. After that, we evaluate state migration in the case of a virtual machine (VM) failure in § 5.2. In § 5.3, we show the overhead that Rhino introduces

during query processing. Next, we evaluate vertical scaling (§ 5.4.1) and load balancing (§ 5.4.2). Finally, we examine our SUTs under varying data rates (§ 5.5) and sum up the evaluation (§ 5.6).

5.1 Experiment Setup

In this section, we introduce our experimental setup (§ 5.1.1), our workloads (§ 5.1.2), SUT configuration (§ 5.1.3), stream generator (§ 5.1.4), and SUT interaction (§ 5.1.5).

5.1.1 Hardware and Software. In our experiments, we deploy our SUTs on a mid-sized cluster that we rent on Google Cloud Platform. The cluster consists of 16 `n1-standard-16` virtual machines. Each VM is equipped with 16 2 Ghz Intel Xeon (Skylake family) vcores, 64 GB of main memory, and two local NVMe SSDs. These VMs run Debian 9 Cloud, use the OpenJDK JVM (version 1.8), and are connected via a 2 Gbps/vcore virtual network switch.

In our evaluation, we use Apache Flink 1.6 as baseline and Apache Kafka 0.11 as a reliable message broker that provides upstream backup [22]. Apache Flink ships with RocksDB [13] as out-of-core storage layer for state. Each stateful operator instance in Apache Flink has its own instance of RocksDB. We pair Flink with the Apache Hadoop Distributed File System (HDFS) version 2.7 as persistent checkpoint storage. Finally, we use the Megaphone implementation (rev: `abadf42`) available online [20].

5.1.2 Workloads. We select the NEXMark benchmark suite to experimentally validate our system design [40]. This suite simulates a real-time auction platform with auction, bids, and new user events. The NEXMark benchmark consists of three logical streams, i.e., an auction stream, a bid stream, and a new user event stream. Records are 206 (new user), 269 (auction), and 32 (bid) bytes large. Every record stores an 8-bytes primary key and an 8-bytes creation timestamp. The suite features stateless (e.g., projection and filtering) and stateful (e.g., stream joins) queries.

We use query 5 (NBQ5) and query 8 (NBQ8) and we define a new query (NBQX) that consists of five sub-queries. Based on the these queries, we define three workloads to assess our SUTs. First, NBQ5 contains a window aggregation with a window of 60 secs and 10 secs slide on the bid stream. We select NBQ5 as it features small state sizes and a read-modify-write state update pattern. Second, NBQ8 consists of a tumbling window join on 12 hours in event time on the auction and new user streams. We choose NBQ8 since it reaches large state sizes due to its append state update pattern. Third, NBQX contains multiple four session window joins with 30, 60, 90, 120 minutes gap and a tumbling window join of four hours on the auction and bid stream. We select NBQX as it contains multiple queries that alone have

mid-sized state but result in large state size by running concurrently. Furthermore, NBQX features append and deletion update patterns.

5.1.3 SUT Configuration. We deploy Flink, Megaphone, RhinoDFS, and Rhino on eight VMs, Kafka on four VMs, and our data generator on four VMs. We use this configuration to ensure that Kafka and our data generator have no throughput limitations. We spawn HDFS instances on the same VMs where the SUTs run. In addition, we reserve one VM to run the coordinators of Flink and Rhino.

Flink and Rhino. To configure Flink (and Rhino), we follow its configuration guidelines [16]. On each VM, we allocate half cores for processing and the other half for network and disk I/O. We allocate 55% of the OS memory to Flink and Rhino, which they divide equally in on-heap and off-heap memory. Flink and Rhino use the off-heap memory for buffer management and on-heap memory for data processing. We assign the remaining memory to RocksDB and the OS page cache. In addition, we assign one dedicated SSD to Rhino's replication runtime. Finally, we use 2^{15} key groups for consistent hashing and 4 virtual nodes for Rhino as these values lead to best performance for our workloads.

Megaphone. We configure Megaphone to use half of the cores for stream processing and the other half for network transfer, following TimelyDataflow's guidelines [20]. In addition, we configure Megaphone to use 2^{15} bins to be compliant to Rhino's setup.

RocksDB. We configure RocksDB for SSD storage, following best practices [14]. To this end, we use fixed-sized memtables (64 MB), bloom filters (for point lookup), and 64 MB as block size for *string sorted tables*.

HDFS. We configure HDFS for Flink and RhinoDFS with a replication factor of two, i.e., each block is replicated to two VMs. As a result, HDFS replicates the first copy of a state block locally and the second block on a different VM. To simulate this replication factor, we configure Rhino to store a local copy (primary copy) and a remote copy (secondary copy) of the state.

Kafka. We let Kafka use all SSDs, all cores, and 55 GB of page cache on each VM, following best practices [11]. We configure Kafka to batch records in buffers of 32 KB with a maximum waiting time of 100 ms. We use three Kafka topics with 32 partitions each to represent 32 new user, 32 bid, and 32 auction streams.

5.1.4 Stream Generator. We implement a stream generator to produce three logical streams of new user, bid, and auction events. Our generator concurrently creates 32 physical streams for each logical stream. To minimize JVM overhead, we implement our generator by omitting managed object allocation and garbage collection.

To assess the SUTs, we let our generator produce records at the maximum sustainable throughput of all SUTs. The maximum sustainable throughput is the maximum rate at which an SPE can ingest a stream and output results at constant, target latency [26]. To this end, we experimentally find the sustainable throughput for each query. Thus, we configure the generator to generate each stream at 128 MB/s for NBQ5 (target latency 500 ms) and 8 MB/s for NBQ8 (target latency 500 ms) and NBQX (target latency 5 s). Overall, we deploy four generators with eight running threads each for a total throughput of 4 GB/s (~135 M records/s) for NBQ5, 256 MB/s for (~500 K records/s) for NBQ8, and 256 MB/s (~3.45 M records/s) for NBQX. Primary keys (auction id, person id) are generated randomly following uniform distribution.

Note that Megaphone does not support real-time data ingestion via external systems as this results in increasing latency. Therefore, Megaphone generates records on-demand within its source operators.

5.1.5 SUT Interaction. Unless stated otherwise, we run our SUTs in conjunction with Kafka and our own custom generator. We follow the methodology of Karimov et al. [26] for end-to-end evaluation of SPEs. To this end, we decouple the data generator, Kafka, and SPE. We denote the end-to-end processing latency as the interval between the arrival timestamp at the last operator in a pipeline and the creation timestamp of the record. Our generator creates timestamped records in event time and sends them to Kafka, which acts as reliable message broker and replays streams when necessary.

We implement our queries in Flink and Rhino using built-in operators. Each source runs with a degree of parallelism (DOP) of 32 to fully use Kafka parallelism, i.e., one thread per partition. We run the join and aggregation operators with a DOP of 64 to fully use SUT parallelism. We instrument the join and aggregation operators to measure processing latency. We repeat each experiment multiple times and report mean, minimum, and 99-th percentile measurements.

5.2 Rhino for Fault Tolerance

In this set of experiments, we examine two important aspects of an SPE. First, we evaluate the performance that a modern SPE exhibits in restoring a query with large operator state (see § 5.2.1). Second, we measure the impact of recovering from a failure on processing latency (see § 5.2.2).

5.2.1 Recovery Time. In this experiment, we run a benchmark using NBQ8 to measure the time required for our SUTs to recover from a failure with varying state sizes. Then, we perform a time breakdown of the time spent in the recovery process. Finally, we compare our results, which we summarize in Table 1.

State Size	SUT	Scheduling	State Fetching	State Loading
250 GB	Flink	2.2 ± 0.1	68.2 ± 5.7	1.3 ± 0.2
	Rhino	2.8 ± 0.2	0.2 ± 0.1	1.3 ± 0.3
	RhinoDFS	2.9 ± 0.2	10.7 ± 3.1	1.3 ± 0.1
	Megaphone	46.3 ± 2.8		
500 GB	Flink	2.5 ± 0.2	116.6 ± 4.9	1.8 ± 0.3
	Rhino	3.1 ± 0.3	0.2 ± 0.1	1.3 ± 0.3
	RhinoDFS	3.0 ± 0.3	18.9 ± 3.7	1.3 ± 0.5
	Megaphone	74.8 ± 3.0		
750 GB	Flink	2.6 ± 0.3	205.3 ± 5.2	1.3 ± 0.1
	Rhino	3.0 ± 0.2	0.2 ± 0.1	1.5 ± 0.1
	RhinoDFS	2.6 ± 0.1	36.1 ± 2.3	1.5 ± 0.2
	Megaphone	Out-of-Memory		
1000 GB	Flink	2.4 ± 0.3	252.9 ± 5.9	1.5 ± 0.2
	Rhino	3.0 ± 0.2	0.2 ± 0.1	1.5 ± 0.2
	RhinoDFS	2.9 ± 0.3	62.7 ± 0.9	1.5 ± 0.1
	Megaphone	Out-of-Memory		

Table 1: Time breakdown in seconds for state migration during a recovery.

Workload. For this experiment, NBQ8 runs until it reaches the desired state size of 250 GB, 500 GB, 750 GB, and 1 TB. Next, we terminate one VM and measure the time spent by our SUTs to reconfigure the running query. Furthermore, we configure the interval of incremental checkpointing to three minutes for Flink and Rhino variants.

We distinguish three operations that comprise a recovery, i.e., scheduling, state fetching, and state loading. Scheduling is the time spent in triggering a reconfiguration . State fetching is the time spent in retrieving state from the location of the previous checkpoint. State loading is the time spent in loading checkpointed state into the state backend.

Result. The time-breakdown in Table 1 shows three aspects. First, we observe that the most expensive operation for block-centric replication is state fetching for RhinoDFS and Flink. This dominates the overall recovery process and depends on the state size. In contrast, Rhino's state-centric replication reduces the state fetching overhead significantly. Performance increases due to local state fetching, which involves hard-linking instead of network transfer.

Second, scheduling and state loading have negligible impact on the recovery duration. In particular, scheduling is about 25% faster in Flink because Flink triggers a recovery immediately. In contrast, Rhino starts a reconfiguration through handovers that reach target instances based on their processing speed. For Rhino and Flink, state loading in RocksDB only needs to open the data files and process metadata files. After the load, the state resides in the last level of the LSM-Tree and is ready to be queried by the SPE.

Third, we observe that Megaphone does not support workloads with more than 500 GB of state as it runs out of memory.

For state smaller than 500 GB, Megaphone spends the majority of time to schedule migrations, serialize state into buffers, write buffers on the network, deserialize buffers, and restore state. Note that Megaphone overlaps the above operations on a key basis, e.g., it schedules and migrates state of different keys at the same time. Therefore, it was unfeasible for us to break down its migration times.

Discussion. This experiment outlines that Rhino efficiently recovers a query from a VM failure in a few seconds, even in the presence of large operator state. This improvement is due to local state fetching, which is only marginally impacted by state size. In contrast, state fetching in Flink and RhinoDFS entails network transfer and introduces latency proportional to the state size. In particular, RhinoDFS is faster than Flink because of fine-grained operator restart but does not achieve low-latency. Instead, Flink fully restarts NBQ8, which results in a 50x higher latency. In our setup, Megaphone does not reach all desired state sizes due to lack of memory management for state-related operations. This prevents Megaphone from supporting large state workloads. In the cases where Megaphone can sustain a workload, its state migration is up to 1.5x faster than Flink's one. Overall, Rhino improves recovery time by 50x compared to Flink, 15x compared to Megaphone, and 11x compared to RhinoDFS.

Note that we run the same benchmark on NBQ5 and all SUTs had comparable recovery time. This suggests that Rhino's design is beneficial if queries require large state.

5.2.2 Impact on Latency. In this experiment, we examine the impact of state migration on the end-to-end processing latency in Flink, Rhino, and RhinoDFS. We exclude Megaphone in the following experiments as it provides state migration but does not have mechanisms for fault-tolerance and resource elasticity. Therefore, the optimizations provided by Megaphone are orthogonal to our work.

Workloads. In this experiment, we run NBQ8, NBQ5, and NBQX on eight VMs. We terminate one VM after three checkpoints and let each system recover from the failure. Upon the failure, the overall state size of the last checkpoint before the failure is approximately 190 GB (NBQ8), 26 MB (NBQ5), and 180 GB (NBQX). After the failure, we let the SUTs run for other three checkpoints and then we terminate the execution. In Figure 4a, 4b, and 4c, we report our latency measurements, which we sample every 200 K records.

Result. Figures 4a, 4b, and 4c shows that Rhino and Flink can process stream events with low overhead at steady state. For NBQ8 and NBQ5, the average latency for both systems is stable around 100 ms (min: 2 ms, p99: 6.8 s). For NBQX, the average latency is 5 s (min: 1 s, p99: 12 s). Note that the higher max latencies are due to the synchronous phase of a checkpoint, which pauses query processing.

(a) Fault Tolerance NBQ8.

(b) Fault Tolerance NBQ5.

(c) Fault Tolerance NBQX.

(d) Vertical Scaling NBQ8.

(e) Vertical Scaling NBQ5.

(f) Vertical Scaling NBQX.

(g) Load Balancing NBQ8.

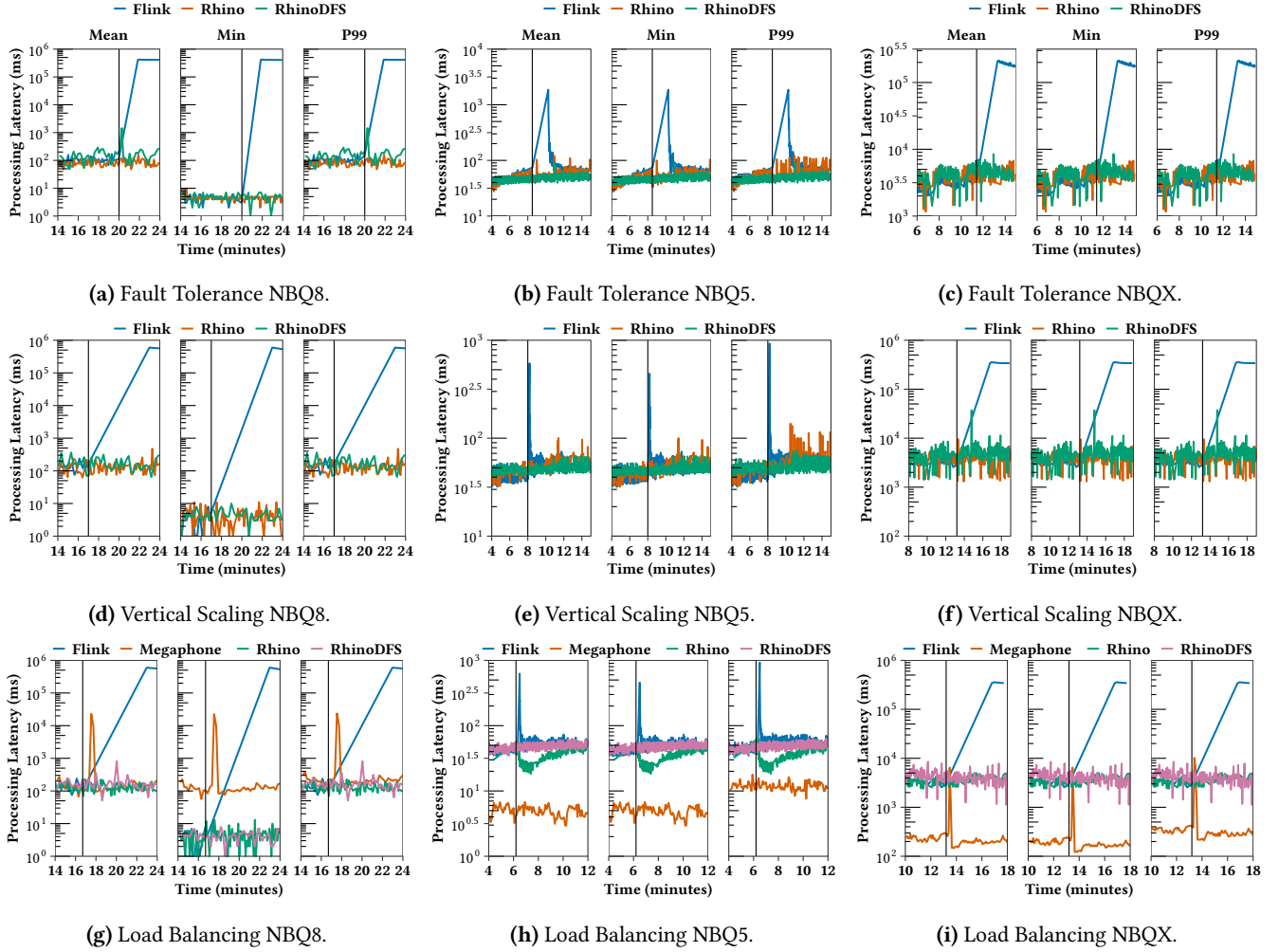(h) Load Balancing NBQ5.

(i) Load Balancing NBQX.

Figure 4: End-to-end Processing Latency in ms (using log-scale) for our fault tolerance, vertical scaling, and load balancing experiments. The vertical black bar represents the moment we trigger a reconfiguration.

Upon a VM failure, we observe that latency in Rhino is not affected, whereas the latency of Flink increases up to 300 s. After recovery, Rhino's variants do not exhibit any impact on latency. In contrast, Flink accumulates up to 300 s of latency lag from the data generator and cannot resume processing with sub-second latency after an outage.

Discussion. Overall, our evaluation confirms that Rhino's design choices result in a robust operational behavior. Rhino handles a VM failure without increasing latency, whereas Flink increases latency by three orders of magnitude.

The main reasons for Flink's decreased performance is three-fold. First, Flink needs to stop and restart the running query, which takes a few seconds. Second, deployed operator instances need to fetch state through HDFS prior to resuming processing. Third, Flink needs to replay records from the upstream backup, which negatively affects latency, whereas Rhino buffers records internally.

Although latency in Rhino is stable after a handover, the SPE is expected to provision a new worker to replace the failed VM. Afterwards, Rhino can migrate again the operators to rebalance the cluster.

5.3 Overhead of Rhino

In this experiment, we evaluate the impact of state-centric replication on query processing. To this end, we measure latency and OS resource utilization with the SUTs running below their saturation point, i.e., the generator rate is set to maximum sustainable throughout. In Figure 4, we show the processing latency of Rhino and Flink before and after a handover takes place (i.e., indicated by the vertical black line). As shown, the latency of query processing (on the left of the black line) is negligibly affected by Rhino's approach in comparison to block replication of Flink and RhinoDFS.

As a result, Rhino does not increase processing latency of a query when there is no in-flight reconfiguration.

In Figure 5, we report aggregated CPU, memory, network, and disk utilization of our cluster. Rhino and Flink use network to read from Kafka and to perform data exchange and state migration among VMs. They use disk to update state stored in RocksDB and to migrate state among VMs. In contrast, Megaphone requires network for processing and migrations but no disk. Before the reconfiguration, we observe that Rhino and Flink have similar resource utilization during query processing as they execute the same routines. We also notice periodical peaks in all charts, which occur at every checkpoint/replication. In contrast, CPUs usage in Megaphone is constant because of a fiber-based scheduler. Megaphone's memory consumption increases over time since it allocates memory on-demand from the OS.

During a replication, Rhino uses up to 30% network bandwidth and 5% disk write bandwidth more than Flink. However, the higher utilization of network bandwidth results in an up to 3.5x faster state transfer than Flink. Furthermore, Rhino requires 25% less CPU in comparison to Flink, whereas memory consumption is the same. We do not observe spikes in network utilization for Megaphone as it multiplexes state migration with data exchange. After a reconfiguration, Rhino and Flink exhibit similar resource utilization. However, Rhino resumes proactive migration after the reconfiguration (minute 23), while Flink is still resuming query processing. To sum up, the experiment shows that proactive state migration of Rhino does not negatively impact processing latency.

5.4 Rhino for Resource Efficiency

In this section, we assess the performance of Rhino, RhinoDFS, and Flink when they perform vertical scaling (see § 5.4.1) and load balancing (see § 5.4.2). We find that horizontal scaling performs similarly to vertical scaling with a slowdown proportional to the size of the state scheduled for migration. Therefore, we omit its evaluation in this section.

5.4.1 Vertical Scaling. In this section, we evaluate the vertical rescaling of NBQ8, NBQ5, and NBQX by adding extra instances on running workers.

Workload. In this experiment, we run NBQ8, NBQ5, and NBQX on eight VMs (max DOP is 64). We configure each worker to not use full parallelism, i.e., the DOP of the stateful join and aggregation operators is 56 (seven instances per VM). We set the checkpoint interval to two minutes to avoid overhead due to continuous checkpointing. After three checkpoints, we trigger a rescaling operation and switch to full parallelism (64 instances, eight per VM). The overall state size of the last checkpoint before rescaling is approximately 220 GB (NBQ8), 26 MB (NBQ5), and 170 GB (NBQX). After

rescaling, we let the SUTs run for three checkpoints and then stop the execution. We collect latency measurements and report them in Figure 4d, 4e, and 4f.

Result. Figures 4d, 4e, and 4f show that the average latency for Rhino's variants is stable. The SUTs can keep average latency of NBQ8 around 130 ms (min: 2 ms, p99: 11 s) for Rhino's variant and 129 ms (min: 2 ms, p99: 9 s) for Flink. In NBQ5, average latency is about 75 ms (min: 2 ms, p99: 119 ms) for all SUTs. For NBQX, average is about 5 s (min: 1 s, p99: 10 s) for all SUTs. As in previous experiment, the high maximum latency is due to the synchronous phase of a checkpoint and higher complexity of NBQX.

Upon rescaling in NBQ8, we observe that latency of Flink increases up to 570 s, whereas RhinoDFS has a sudden spike to 30 s. After scaling, processing latency for Rhino increases to 146 ms in NBQ8. We observe that latency drops to 118 ms after 120 s. In contrast, Flink accumulates ~10 m of latency lag from the data generator in NBQ8. NBQX has similar behavior due to large state sizes. Finally, the execution of NBQ5 is stable in all SUTs before and after the reconfiguration. Upon rescaling, Flink exhibits a spike in latency of 1 s.

Discussion. Overall, this experiment confirms that Rhino supports vertical rescaling without introducing excessive latency on query processing among all queries. In contrast, Flink induces an increased latency of up to three orders of magnitude in NBQ8. This latency spike is significant in NBQ8 as Flink needs to restart a query and reshuffle state among workers. In contrast, Rhino need to checkpoint and migrate ~32 GB of state among workers and have similar performance in NBQ8 and NBQX. When state is small (NBQ5), Flink and Rhino have similar performance because state migration is not a bottleneck.

5.4.2 Load balancing. In this section, we examine how fast Rhino and Megaphone can reconfigure NBQ8, NBQ5, and NBQX to balance the load among stateful join instances. As there is no implementation of load balancing in Flink, we compare load balancing against vertical scaling.

Workload. In this experiment, we deploy NBQ8, NBQ5, and NBQX on eight VMs. We follow the same methodology of the experiment in § 5.4.1. After three checkpoints, we trigger a load balancing operation that moves half virtual nodes from eight instances (one per VM) to other eight instances. We do the same for Megaphone but do not trigger checkpoints.

Result. Our latency measurements in Figures 4g, 4h, and 4i show that Rhino sustains large state stream processing with an average latency of 110 ms in NBQ8. During a load balancing operation, we observe a latency increase of ~60 ms, which Rhino mitigates in one minute. Afterwards, latency is constant with minor fluctuations during checkpointing and proactive state migration. In NBQ5, we observe that Rhino's rebalancing is highly effective such that latency decreases

from 60 ms to 20 ms for some minutes. In contrast, the latency of Flink reaches 500 ms for a few seconds after the reconfiguration. In NBQX, we observe that Rhino does not introduce latency after a rebalancing, which stays constant at around 4 sec. Instead, Flink's latency reaches 3.5 min after the reconfiguration. During the reconfiguration of NBQ8 and NBQX, the latency of Megaphone reaches 23.6 s and 10.2 s for ~90 s, respectively.

Discussion. This experiment shows that Rhino supports load balancing via migration of virtual nodes with minimal impact on latency in NBQ8, NBQ5, NBQX. This experiments shows that Megaphone's migration affects latency if migrated state is large, i.e., 27 GB. Overall, compared to vertical scaling, our load balancing technique keeps latency constant, whereas Flink shows an increment in latency by three orders of magnitude.

5.5 Migration under varying data rates

In the following, we show how Rhino supports query processing and state migration under varying data rates. We select NBQ8 for this experiment as it results in larger state.

Workload. We use the same setup as in § 5.2.2 with the following changes. We configure our data generator to produce records at varying speed. Each producer thread initially generates data at 1 MB/s. Every 10 s, data rate increases by 0.5 MB/s until it reaches 8 MB/s (the max sustainable rate for NBQ8). When this happens, data rate decreases by 0.5 MB/s every 10 s, until it reaches 1 MB/s. This repeats throughout the whole experiment. We let Rhino, RhinoDFS, and Flink run until they reach approx. 150 GB of state. Then, we trigger a reconfiguration to migrate 8 operators from one server to the remaining 7 servers.

Result. Our latency measurements in Figure 6 show that all SUTs can sustain stream processing under varying data rates. Average latency for all SUTs is 205 ms (min: 9 ms, p99: 826 ms). Upon the reconfiguration, the latency of Rhino and RhinoDFS remains constant, whereas the latency of Flink reaches 225 s. After 2 minutes from the reconfiguration, minimum latency of Flink goes down to 20 ms.

Discussion. Overall, this experiment confirms that Rhino also supports reconfiguration in the presence of fluctuating data rates. We also show that Rhino and Flink are resistant to varying data rates during query processing. However, latency in Rhino is not affected during a reconfiguration, whereas Flink accumulates latency up to 225 s.

5.6 Discussion

The key insights of our evaluation are four-fold. First, we show that Rhino and RhinoDFS can perform a reconfiguration to provide fault-tolerance, vertical scaling, and load balancing with minimal impact on processing latency. When

state is large, Rhino achieves up to three orders of magnitude lower latency compared to Flink and two compared to Megaphone. In contrast, if the state size is small, Rhino performs similarly to baseline. Second, we show that Rhino's state migration protocol is beneficial when operator state reaches TB sizes. In particular, Rhino reconfigures a query after a failure 50x faster than Flink, 15x faster than Megaphone, and 11x faster than RhinoDFS. Third, we show that Rhino has minor overhead on OS resources and stream processing. In particular, Rhino uses 30% more network bandwidth than baseline but achieves up to 3.5x faster state transfer. However, we expect our replication runtime to become a bottleneck if an incremental checkpoint to migrate is large, e.g, above 50 GB per instance. We leave investigating this issue as future work, e.g., using adaptive checkpoint scheduling. Fourth, we show that Rhino migrates state and reconfigures a query under fluctuating data rates with no overhead. Overall, our evaluation shows that Rhino enables on-the-fly reconfigurations of running queries in the presence of large operator state on common workloads.

6 RELATED WORK

In the following, we group and describe related work, which we did not fully cover in previous sections.

Fernandez et al. propose SEEP [6] and SDG [7] to address the problem of scaling up and recovering stateful operators in cloud environments. Their experiments target operators with state size up to 200 GB and confirm that larger state leads to higher recovery time. Both approaches are based on migration of checkpointed state, which captures also in-flight records. SEEP stores checkpoints at upstream operators, which may become overwhelmed. SDG improves on SEEP by using incremental, per-operator checkpoints that are persisted to m workers. However, SDG does not control where an operator is resumed, which leads to state transfer and network overhead. In this paper, we explicitly address these shortcomings to provide low reconfiguration time for queries with very large operator state. To this end, we target large state migration (up to 1 TB) by proactively migrating state to workers where a reconfiguration will take place.

Mai et al. [30] propose Chi to enable user-defined reconfiguration of running queries. Their key idea is the use of control events in the dataflow to trigger a reconfiguration, which reduces latency by 60% on queries with small state. Rhino uses a similar approach, i.e., control events, but it differs from Chi in the following aspects. First, user-defined reconfigurations are orthogonal to Rhino as it supports reconfiguration to transparently provide fault-tolerance, resource elasticity, and load balancing. Second, Chi reactively migrates state in bulk upon a user-defined reconfiguration, which leads to migration time proportional to state size. In contrast,
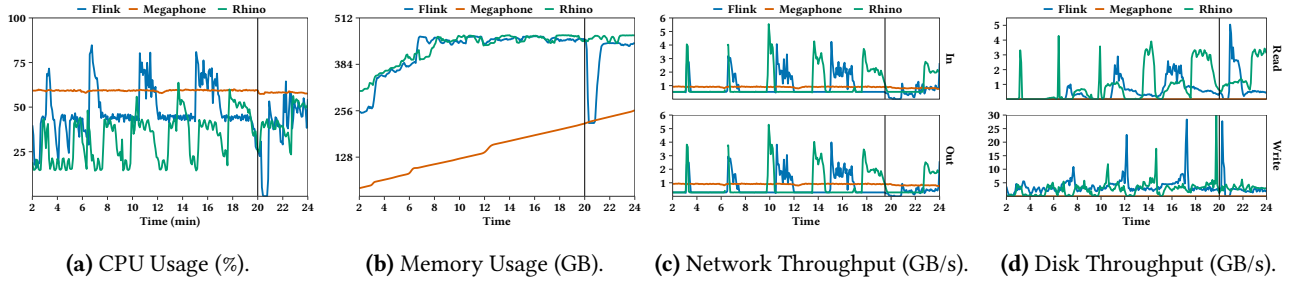
(a) CPU Usage (%). **(b)** Memory Usage (GB). **(c)** Network Throughput (GB/s). **(d)** Disk Throughput (GB/s).

Figure 5: Comparison of resource utilization of Apache Flink and Rhino for NBQ8. The black line indicated when we perform a reconfiguration.
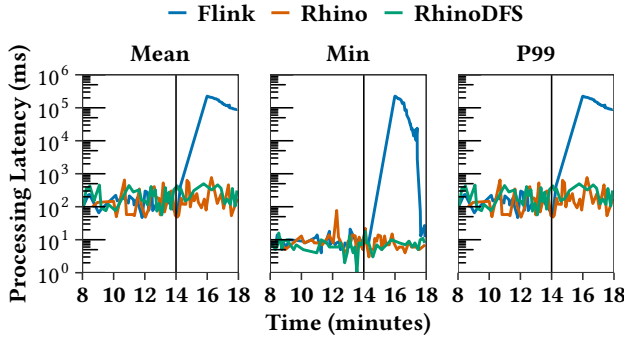

Figure 6: End-to-end latency of NBQ8 under varying data rate.

Rhino proactively and incrementally migrates state to provide timely reconfigurations. Finally, a state migration for a logical operator in Chi affects all its physical instances. As a result, all running instances send state to a newly spawned instance. Instead, Rhino migrates state through virtual nodes at a finer granularity to involve the least number of instances.

Megaphone [21] is a system that provides online, fine-grained reconfigurations for SPEs with out-of-band tracking of progress, e.g., Timely Dataflow [31] and MillWheel [1]. There are four key differences that make our contribution different than Megaphone. First, Megaphone and Rhino have different scope. Megaphone enables programmable state migrations in the query DSL, whereas Rhino provides transparent fault-tolerance, resource elasticity, and load balancing via state migration. Second, Megaphone migrates state in bulk and in reaction to user request, whereas Rhino proactively migrates state to reduce reconfiguration time. In particular, Rhino explicitly targets large state migration, while Megaphone handles migration as long as state fits in memory. Third, Megaphone requires out-of-band tracking of progress, i.e., operators must observe each other's progress. Moreover, downstream operators must expose their state to their upstream to perform state migration. While Timely Dataflow provides these features out-of-the-box, other SPEs need costly synchronization and communication techniques to fullfil

Megaphone's requirements. In contrast, Rhino has an internal progress tracking mechanism based on control events that only requires an SPE to follow the streaming dataflow paradigm. Furthermore, Rhino has its own migration protocol, which does not require operators to expose state. As a result, Rhino's protocols could run on Timely Dataflow. Finally, Megaphone and Rhino use different migration protocols. Megaphone uses two migrator operators for every migratable operator. This could lead to scalability issues on larger queries. Instead, Rhino multiplexes state migration of every operator through its distributed runtime.

Dhalion [17] and DS2 [24] are monitoring tools that trigger scaling decisions. Their contributions are orthogonal to ours as Rhino provides a mechanism to reconfigure a running SPE. We envision an SPE, a monitoring tool, and Rhino that cooperatively handle anomalous operational events to ensure robust stream processing.

7 CONCLUSION

In this paper, we present Rhino, a system library that enables fine-grained fault-tolerance, resource elasticity, and runtime optimizations in the presence of large distributed state. Through proactive state migration, Rhino removes the bottleneck induced by state transfer upon a reconfiguration. We plan to incorporate Rhino in our new data processing platform NebulaStream [48].

We evaluate our design choices on a common benchmark suite and compare two variants of Rhino against Flink and Megaphone. Rhino is 50x faster than Flink, 15x faster than Megaphone, and 11x faster than RhinoDFS in reconfiguring a query. Moreover, Rhino shows a reduction in processing latency by up to three orders of magnitude after a reconfiguration. With Rhino, we enable robust stream processing for queries with very large distributed operator state, regardless of failures or fluctuations in the data rate.

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *PVLDB* (2013).

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB* (2015).

[3] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, J. Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* (2014).

[4] Peter A. Alsberg and John D. Day. 1976. A Principle for Resilient Sharing of Distributed Resources. In *ICSE*.

[5] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. *PVLDB Endow.* (2017).

[6] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *ACM SIGMOD*.

[7] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*.

[8] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN*.

[9] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. *SIGMOD*.

[10] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM TOCS* (1985).

[11] Confluent. 2017. Running Kafka in Production. https://docs.confluent.io/current/kafka/deployment.html

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS* (2007).

[13] Facebook. 2012. RocksDB.org. Facebook Open Source. https://rocksdb.org/

[14] Facebook. 2017. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making state explicit for imperative big data processing. In *2014 USENIX ATC)*.

[16] Apache Flink. 2015. Apache Flink Configuration. https://ci.apache.org/projects/flink/flink-docs-master/

[17] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *PVLDB* (2017).

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *ACM SOSP*.

[19] Thomas Heinze, Yuanzhen Ji, Lars Roediger, Valerio Pappalardo, Andreas Meister, Zbigniew Jerzak, and Christof Fetzer. 2015. FUGU: Elastic Data Stream Processing with Latency Constraints. *IEEE Data Eng. Bull.* (2015).

[20] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration. https://github.com/strymon-system/megaphone

[21] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, and Timothy Roscoe. 2019. Megaphone: Latency-conscious State Migration for Distributed Streaming Dataflows. *VLDB* (2019).

[22] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. 2005. High-Availability Algorithms for Distributed Stream Processing. In *IEEE ICDE*.

[23] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, Anirban Banerjee, Benjamin Heintz, Shridar Iyer, and Anshul Jaiswal. 2018. Providing Streaming Joins As a Service at Facebook. *PVLDB* (2018).

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *USENIX OSDI*.

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees.. In *ACM STOC*.

[26] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *IEEE ICDE 2018*.

[27] Klaviyo. 2019. Apache Flink Performance Optimization. https://klaviyo.tech/flinkperf-c7bd28acc67

[28] H. T. Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *ACM SIGCOMM*.

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS* (2010).

[30] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *VLDB* (2018).

[31] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *ACM SOSP*.

[32] Raghunath Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Andrew Bond, Forrest Carman, and Michael Majdalany. 2013. TPC State of the Council 2013. In *5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*.

[33] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *IEEE ICDE*.

[34] Netflix. 2017. Keystone Real-time Stream Processing Platform. https://medium.com/netflix-techblog/keystone-real-time-stream-\processing-platform-a3ee651812a

[35] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *PVLDB* (2017).

[36] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM PODC*.

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *IEEE MSST*.

[38] Tencent. 2018. Oceanus: A one-stop platform for real time stream processing. https://www.ververica.com/blog/oceanus-platform-powered-by-apache-flink

[39] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A Survey of State Management in Big Data Processing Systems. *The VLDB Journal* (2018).

[40] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2004. NEXMark - A Benchmark for Queries over Data Streams DRAFT. (2004).

[41] Uber. 2018. Introducing AthenaX, Uber Engineering's Open Source Streaming Analytics Platform. https://eng.uber.com/athenax/

[42] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *USENIX OSDI*.

[43] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *USENIX OSDI*.

[44] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2019. Anna: A kvs for any scale. *IEEE TKDE* (2019).

[45] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *IEEE ICDE*.

[46] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *ACM SOSP*.

[47] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache Spark: A unified engine for big data processing. *CACM* (2016).

[48] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.

[49] Yali Zhu, Elke Rundensteiner, and George Heineman. 2004. Dynamic Plan Migration for Continuous Queries over Data Streams. In *ACM SIGMOD*.