Amoeba: Aligning Stream Processing Operators with Externally-Managed State

Antonis Papaioannou and Kostas Magoutis

{papaioan,magoutis}@ics.forth.gr

Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH)

Computer Science Department, University of Crete

Heraklion, Greece

ABSTRACT

Scalable stream processing systems (SPS) often require external storage systems for long-term storage of non-ephemeral state. Such state cannot be accommodated in the internal stores of SPSes that are mainly geared for fault tolerance of streaming jobs, lack externally visible APIs, and their state is disposed of at the end of such jobs. Recent research have pointed to scalable in-memory key-value stores (KVS) as an efficient solution to manage external state. While such data stores have been interconnected with scalable streaming systems, they are currently managed independently, missing opportunities for optimizations, such as exploiting locality between stream partitions and table shards, as well as coordinating elasticity actions. Both processing and data management systems are typically designed for scalability, however coordination between them poses a significant challenge. In this work we describe Amoeba, a system that dynamically adapts data-partitioning schemes and/or task or data placement across systems to eliminate unnecessary network communication across nodes. Our evaluation using state-of-the-art systems, such as the Flink SPS and Redis KVS, demonstrated 2.6x performance improvement when aligning SPS tasks with KVS shards in AWS deployments of up to 64 nodes.

CCS CONCEPTS

• **Information systems** → **Stream management**; **Key-value stores**.

KEYWORDS

data stream processing, scalable cloud services, data locality

1 INTRODUCTION

Scalable stream-processing systems (SPSs) and analytics are key to a number of high data-volume Internet services [3, 6, 10, 14, 23, 35, 39]. As real-time data analytics provide valuable insights, there are cases where stream-processing platforms process trillions of events per day [6]. A stream application can be modeled as a directed dataflow graph comprising processing operators [9]. Each operator

transforms data flowing from its inputs to its outputs. Stateful operators preserve a sequence of recent data as ephemeral internal state and apply a transformation on a subset of its incoming data (e.g. aggregating values over a time window). Storing intermediate (ephemeral) state of streaming applications in a reliable, recoverable manner is important for the fault-tolerance of long-running jobs.

Early SPSs stored ephemeral operator state in memory or on an external data store (e.g., an SQL server), whereas modern systems increasingly use embedded local key-value stores (KVS). Other systems externalize data management to distributed storage systems for out-of-the-box fault-tolerance, recovery, and elasticity [8, 19].

Persistence of *external state* of streaming jobs is a separate but equally important concern. External state refers to data produced by streaming jobs or by other applications/systems [10, 14, 23, 39], or ground-truth table data that can be accessed during the execution of streaming applications. External state has typically a lifecycle that is disconnected from that of streaming applications and is often stored on scalable KVSs [16, 23, 39]. Current state-of-the-art solutions, such as IBM InfoSphere Streams, provide support for general-purpose access to external state on scalable KVSs [35, 37]. However, lack of coordination in the placement/partitioning policies of SPSs and external-state KVSs, results to unnecessary network cross-talk (often on the critical path of a streaming application), and mis-alignment of adaptation policies such as elasticity.

In this work, we build upon the idea that the performance of accessing external state can be improved via co-location of internal and externally-managed state partitions. This is possible either via appropriate (informed) initial placement of such partitions and alignment of SPS and KVS key partitioning schemes or via migration of tasks (along with their internal state) or externally-managed state partitions, with the ensuing data movement where appropriate. Early experimental results (poster [32]) report the benefits of aligning data and task placement/partition policies in scalable SPSs with externally-managed state KVSs. That study used a simple application derived from the Yahoo streaming benchmark [16] deployed on the Flink [13] SPS using Redis as an external-state KVS. The results provided early evidence of significant performance benefits on manually created configurations and small-scale setups.

In this work we extend those early results in a number of directions. We describe the design and implementation of an actual system that we call Amoeba¹ that implements the alignment between SPS task partitions and externally-managed KVS-based state. We experimentally evaluate Amoeba on an advanced, complex

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).
UCC'21, December 6–9, 2021, Leicester, United Kingdom

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8564-0/21/12...\$15.00
<https://doi.org/10.1145/3468737.3494096>

¹An amoeba is a type of cell or unicellular organism that has the ability to alter its shape, primarily by extending and retracting pseudopods [1]. Its adaptivity as a defining characteristic inspired us to name the system presented in this paper after it.

streaming benchmark application, Linear Road [12], on large-scale deployments of up to 64 AWS nodes. Our work in this paper makes the following research contributions:

- (1) A broad investigation of the benefits of automatically aligning SPS tasks with externally-managed state in large-scale production-quality workloads.
- (2) Determination of the benefits possible with shared access on externally-managed tables, and how these benefits change with increasing system scale.
- (3) Investigation of combined actions (shipping data to tasks or vice versa, adaptation of partitioning schemes) to align externally-managed state partitions with SPS task partitions to exploit locality.

The research work reported in this paper provides a novel solution to the research problem of automatically streamlining and integrating state management capabilities across stream and storage technologies, a complex undertaking that today requires deep human expertise, making it a hard and failure-prone undertaking.

The rest of the paper is structured as follows. In Section 2 we position Amoeba in relation to previous research. In Section 3 we detail our design and prototype implementation of Amoeba. In Section 4 we describe our implementation of Linear Road for the Flink SPS using Redis for storing externally-managed state [31]. In Section 5 we discuss our extensive evaluation of Amoeba on AWS deployments of up to 64 nodes. Finally, in Section 6 we conclude.

2 RELATED WORK

Stream processing applications can be expressed as a collection of processing tasks (operators) connected in a dataflow graph [9, 40], with each operator transforming data flowing from its inputs to its outputs. Operators can be *stateless* or *stateful*. The output of stateless operators (such as filters) is based on the current inputs only and does not take into account state from previous computations. Stateful operators (such as aggregations) build state from past inputs and use it to influence the processing of future input.

Amoeba addresses the alignment between tasks managing stream-processing state and partitions of externally-stored and managed state. We review related research in three areas: SPS access to externally-stored and managed state (§2.1); the management of SPS internal state (ephemeral operators state) (§2.2); and, task placement/alignment to improve performance in different contexts (§2.3).

2.1 Storing and accessing external data

Streaming applications often require access to *external data* on the common path, typically stored on external systems whose lifecycle is decoupled from that of the stream-processing application. Access to external data may be needed in the context (and for the purposes) of either stateful or stateless operators. Management of external state is orthogonal to the management performed by SPS of internal state (ephemeral operator state); the latter is the focus of Section 2.2.

Several production use-cases highlight the need to use external storage systems to share state across different processing applications and management systems [10, 14, 37]. Existing scalable SPSs provide ways to interact with scalable external stores. One way is through source/sink operators channeling streams to scalable key-value stores such as Redis [13, 29] and through user-defined

operators with custom code accessing external state via publicly-available KVS client drivers and APIs. Another way is to extend streaming frameworks with general-purpose put/get access to remote stores [37] as supported by IBM InfoSphere Streams *distributed process store*, and demonstrated in commercial use cases [35]. This model also allows combining "data-at-rest" (batch processing) and "data-in-motion" (stream processing) analysis.

Accessing external state *in the common path of stream processing* is known to be a cause of overhead (higher latency, as well as protocol overhead). With out-of-core datasets stored as external state (e.g., user profiles in large Internet services [14, 29, 39] or user activity [10]), external state is a performance pain point that is not well addressed so far. Amoeba addresses this challenge through the alignment between tasks managing stream-processing state and partitions of externally-stored and managed state.

2.2 Storage solutions for internal operator state

Stateful operators build state from past inputs. This internal state is ephemeral and is tightly coupled with the lifecycle of the operator. Modern SPSs follow two directions to manage the operators internal state: using an embedded data management system such as RocksDB [13, 29], or storing it on external data stores [8, 25, 26]. An embedded store results in efficient local data access, while an external data store has the additional benefit of independent failure modes, although at a cost of higher access latency.

A challenge with internally-managed state is the need to reorganize operator state via network transfer across nodes during reconfigurations of the SPS, which has led to recent research in reducing the impact of such reconfigurations [17, 22, 43]. Using a distributed storage system to persist internal operator state has the benefit of allowing systems to quickly recover from failures and to efficiently reallocate stateful operators across several newly partitioned operators to provide scale out. With certain SPS using external storage systems for reliability and reconfiguration (elasticity), each processing operator creates periodic checkpoints that are mirrored to an external distributed storage system (e.g. HDFS). In case of failure or reconfiguration, the stream processing system can recover its state from the external store [13, 17, 29, 43]. The external store is *not involved in the common path of stream processing* and is only used during the checkpointing or recovery process.

Exposing internal operator state and allowing users to query the state of the streaming job from the outside has recently been introduced in the Flink SPS as *querable state* [5]. This approach applies to internal job state only and does not extend to external data whose lifecycle is disconnected from that of a streaming application.

Combining distributed state management solutions to support both external state and internal operator state via a single platform in modern stream processing frameworks could lead to efficiency and versatility benefits. However, determining how to effectively combine the two types of systems remains a challenging research problem, as has also been pointed elsewhere [41].

2.3 Placement/alignment of SPS tasks

Previous work studied improvements in the placement of processing tasks to reduce the communication overheads between operators, migration of operators and replication [18, 30, 34] or study the

Figure 1: Amoeba system design

checkpoint placement policies [17, 36]. Controlling the placement of externally-managed state to align with streaming tasks for lower latency, a focus of this work, has not been made so far. Aligning data and processing partitions on a common topology to improve data locality is a challenging task. Work on modeling load balancing, operator instance co-locations and horizontal scaling as one integrated optimization problem that can be solved using mixed-integer linear programming (MILP) [27] are related to Amoeba in that it pursues opportunities for co-location (in this case, operator instances) to improve system performance. Previous research looked into task assignment for state sharing [42] and on shared store [26] to support this. Previous research on improving SPS load balancing [20, 24, 28, 38] relates to Amoeba in the use the partitioning scheme to improve SPS performance. Amoeba is the first system to achieve data locality across two systems, an SPS and a KVS, by adapting the SPS and/or KVS partitioning schemes.

3 DESIGN AND IMPLEMENTATION

In this section we describe the design and implementation of Amoeba, a system for adaptive and dynamic alignment of partitioning schemes and assignment of task and data partitions to nodes to improve system performance by minimizing data access over the network.

Figure 1 shows the Amoeba system design. It functions as a typical autonomic monitor-analyze-plan-execute (MAPE) loop [15] whose main purpose is to monitor and coordinate SPS-KVS systems, aiming to achieve efficient SPS access to external state. Amoeba follows a modular design approach using SPS and data store specific drivers as plugins to communicate with the underlying systems. Drivers are responsible to discover metadata and to realize (effect) the alignment plan on the underlying systems.

At the core is the Amoeba **Coordinator**, whose role is to monitor and control inter-dependent SPS and KVS platforms and their relationships, through the driver modules that encapsulate domain-specific knowledge. The Coordinator incorporates a metadata repository responsible for storing information, such as application descriptions, deployment topologies of both systems. It periodically contacts SPS and KVS to discover topology and partition related information (e.g. partition assignment across instances). The coordinator correlates the metadata of both systems to create alignment plans (§3.2). The design of the Coordinator is agnostic to the underlying SPS and KVS.

An **SPS Driver** is a module responsible for interaction with an SPS, incorporating domain-specific knowledge about discovering information and controlling each SPS via external management APIs. An SPS Driver discovers streaming application descriptions, physical deployment topologies (assignment of operator partitions to SPS worker tasks), and how streaming operators are routing their inputs into partitions of downstream operators. Operators partition their outputs when their downstream operators have multiple instances to each instance of each downstream operator. The partitioning scheme is usually based on hashing, although it can also be based on simpler round-robin key distribution approaches or more complex load balancing methods. Following an alignment plan decided by the coordinator, an SPS may adapt its partitioning scheme and assign partitions across processing tasks and/or reconfigure to execute task-migration actions.

A **Store Driver** is a module responsible for the interactions with a *data store* (in principle any scalable storage system, although scalable KVS are the prevailing technology today), focusing on data partitioning, the assignment of one or more partitions across data-store instances, and partition migrations. Amoeba plans its actions upon the understanding that data stores may adapt similarly to SPS tasks, i.e. by adapting their partitioning scheme, re-assign partitions across its instances, as well as via data migration actions.

Naturally, some actions have a higher impact than others, and there may also be constraints on what actions are feasible given the configuration of resources on the field. Amoeba takes these factors into account in proposing actions. Next we focus on details of Amoeba alignment plans for improving the efficiency of SPS access to external state.

3.1 Amoeba inputs and metadata discovery

Amoeba requires users to provide handles and credentials (access rights) to management components and APIs of the underlying systems (SPS and KVS) as well as monitoring endpoints. Amoeba discovers (rather than require a human expert to provide) the deployment topology of the SPS and KVS and the execution graph of streaming applications. Through static analysis of the application's code or deployment package, Amoeba may pinpoint the operators accessing external state. This information is augmented and cross-checked by dynamic monitoring of data-access requests. For completeness, Amoeba allows application owners to augment this information by identifying operators (names) that access external data along the associated external data tables. The Amoeba coordinator periodically communicates with the SPS and data store(s) to draw the above metadata and to discover opportunities for alignment to improve data locality.

3.2 Planning alignment actions

An alignment plan is a collection of alignment actions that aim to improve locality of one or more operators. Amoeba generates alignment plans in three ways, exemplified in Figure 2, where a processing operator with 3 parallel instances accesses an external data table split into 2 partitions (shards) (Figure 2(a)). Amoeba may adjust the parallelism of either the SPS or the KVS via elasticity actions to improve the potential for alignment between the two

Figure 2: Example: (a) Initial deployment, (b) creation of new shard, (c) task migration and partition-scheme change

systems; in Fig. 2(b) it creates an additional KVS server on Node 3. Amoeba may next migrate processing tasks or data partitions, leading to a deployment that increases co-location between the operators and the corresponding data tables; in Fig. 2(c) it migrates a task from Node 4 to Node 1. Finally, Amoeba improves data locality by aligning the partitioning schemes of the two systems to exploit the co-location between processing operators and data tables on Nodes 1-3. Amoeba considers known global constraints (e.g. data tables not allowed to move to or out of specific nodes) and resource limitations (e.g. overloaded nodes that cannot host more processing operators or data shards) in its decision-making process.

Next we describe the process of alignment plan generation and the decisions involved in more detail (Figure 3).

Adjust the parallelism of SPS or KVS: A difference in the degree of parallelism between the SPS and KVS deployments (as in Figure 2(a)) will prohibit full co-location between the two systems, even after migration and partitioning-scheme changes (described next). Where possible, Amoeba considers adjusting the parallelism (via elasticity actions) on either the SPS or KVS as a way to achieve optimal alignment of the systems. Even when SPS and KVS may have had the same degree of parallelism at some point in time, this may be disrupted by an externally triggered elasticity action on one of the two systems (e.g. the SPS, due to a load surge), without a corresponding action by the other. Amoeba can detect this and trigger a corresponding elasticity action (when feasible) to re-align both systems. Amoeba prefers scaling up the system with the lower parallelism, as scaling down may negatively impact overall performance.

In Fig. 2(b), Amoeba adds a new KVS instance on Node 3 so that processing tasks and data partitions have the same parallelism, and re-balances the KVS. Then, it explores potential for more co-location through data or task migrations:

Migrate tasks or data to increase co-location: Where processing operators and data tables are hosted on separate nodes (as in Nodes 1 and 4 in Fig. 2(b)), Amoeba will consider migration actions to increase the co-location. Available options of migrating SPS partitions or KVS data partitions (shards) are taken into account in deciding the appropriate adaptation action. Amoeba selects either SPS or KVS migration depending on the availability of online, low-impact migration mechanisms [17, 22, 33, 43] in either system. Amoeba assumes apriori knowledge of whether such mechanisms are supported in a SPS or KVS in its metadata catalog. If both support such mechanisms, it defaults to the SPS. Sometimes there are constraints that do not permit such actions (e.g. no access rights or

Figure 3: Amoeba decision-making process

limited processing capacity on certain nodes). Amoeba takes into account constraints, rules, and policies in creating migration plans. If migrating a KVS entirely off of its original deployment is not an option, creating new data replicas on the SPS nodes (rather than effecting a full migration there) may be the best possible middle ground.

In Fig. 2(b), Amoeba moves the processing task initially placed on Node 4 to Node 1. Next, it aligns the partitioning scheme of the systems to fully exploit data locality:

Align SPS/KVS partitioning schemes: In any deployment topology, a key Amoeba step is to align the partitioning scheme of the systems to improve data locality. Where it finds that data partitions and processing tasks are already co-located on certain nodes, Amoeba can trigger changes to the partitioning scheme to improve alignment, i.e. the same stream and data keys be routed to the same node. This alignment can be realized on the partitioning scheme of either the SPS or the KVS. Although it is feasible to dynamically repartition a scalable KVS [21], it is usually considered a heavy-weight task. In addition changes on the partitioning scheme of the external database may also impact other systems or applications that access the external data. Thus partition-scheme change on the SPS is typically preferable. In Fig. 2(c), Amoeba detects co-locations on Nodes 1-3 and changes the SPS partitioning scheme to assign keys in each KVS shard to the SPS task hosted in the same node.

Through continuous monitoring, Amoeba analyzes the performance impact of its alignment plans and may re-evaluate if needed

(e.g. in case there are still remote data accesses), similar to previous approaches in online optimization of allocation decisions [11].

Figure 3 depicts the Amoeba decision-making process incorporating its three key mechanisms along with decision points that check on the impact of actions or constraints. As an alignment of SPS-KVS partitioning schemes is needed after either a migration or an elasticity action, the decision-making process starts by considering elasticity first, followed by migration, and ends with partition-scheme alignment.

In our evaluation we demonstrate the decision-making process of all three key alignment mechanisms, namely elasticity actions to adjust parallelism (§5.5), data migrations (§5.4) and partitioning scheme alignment (§5.2 - §5.3)

3.3 Prototype implementation specifics

Our prototype implementation of Amoeba includes a Coordinator and drivers for the Flink SPS and Redis Cluster KVS. The Coordinator incorporates a scheduler that periodically invokes both drivers to discover changes in the deployment topologies. The Flink SPS Driver contacts the Flink Job Manager and extracts the necessary metadata for the running jobs (streaming applications), including details on operators and their accessed state, deployment parallelism, partition placement, etc. The Redis Store Driver is based on the *lettuce* Redis client library². This Redis driver contacts any Redis nodes to get metadata information regarding partition assignment across Redis instances. Both Flink and Redis drivers require no modifications to the underlying platforms.

The Flink and Redis drivers are also used to collect monitoring data exposed by the underlying platforms. In Flink we use the REST API to access the monitoring data exposed by the JobManager. We also enriched the Lettuce Redis client library to expose the Redis requests distribution across nodes to the JobManager metrics. Monitoring is performed via periodic sampling and thus has low overhead. Amoeba can access the monitoring data through its Flink driver. An alternative would be to get the access pattern of Redis requests from the Redis servers directly using the *monitor* command³.

Elasticity and migration actions. Amoeba discovers nodes joining or leaving the cluster on the SPS and can trigger a corresponding KVS action, a Redis *cluster rebalance*. Redis splits data horizontally to 16384 partitions, also called *shards* or *slots*, mapped to Redis node instances. Our implementation supports Redis cluster rebalance to a different set of instances by re-assigning its 16384 partition slots across the available Redis instances. Amoeba effects elasticity on Flink via its standard mechanism based on savepoints [2].

When KVS nodes do not have processing capacity (or are otherwise constrained) to host SPS processing tasks, Amoeba migrates KVS data to SPS nodes by creating new replicas within each Redis replica group. Redis supports the leader-follower (master-slave) replication scheme. Every partition has a master replica that serves read and write requests and a configurable number of follower replicas that asynchronously get updated by the leader replica. When migration of a data partition is needed, the Amoeba Redis driver creates a new follower replica on an appropriate SPS node. When

the new replica catches up with the primary, the driver can re-configure the group, promoting that replica to primary, allowing for local reads and writes between co-located SPS tasks and KVS partitions. Amoeba may also maintain the master in its original node (providing SPS tasks with read-only access to the new local replica), so that other applications that share access to Redis may still access their data on the original replicas.

Partition-scheme alignment. An important mechanism exercised by Amoeba is changing the partitioning scheme in either the SPS or the KVS platform (or both) so as to achieve alignment, namely that external state and operator key routing (and internal state, in case of stateful operators) are partitioned the same way. Flink uses the *KeyBy* transformation, which logically partitions a stream into disjoint partitions based on some field(s) of the tuples (called partition key) [4]. All records with the same key are assigned to the same partition. Internally, Flink KeyBy is based on the MurmurHash algorithm⁴. Since KeyBy does not allow customization, in our prototype we use the Flink *Partitioner* API interface⁵ that allows users to implement their own partitioning method. We have implemented a remote communication API to the Flink Partitioner API through which the Coordinator dynamically adapts the partitioning scheme of certain operators according to its alignment plan. The communication between the Coordinator and Flink partitioner is carried over a sockets-based protocol performing data exchange in a simple JSON format. Redis uses a hash-based sharding scheme based on CRC-32 to distribute data across multiple instances across nodes [7]. The Redis partitioning method differs from the native Flink partitioning scheme. Although Amoeba could make use of database repartitioning as another adaptation mechanism, we have not implemented it in this context. However, ways to dynamically repartition a scalable KVS have been studied in the past [21].

Our prototype offering the above described functionality can support a wide range of experimental scenarios (§5).

4 LINEAR ROAD

To support our evaluation of Amoeba under a complex stream-processing application, we selected Linear Road [12], an extensively used performance evaluation benchmark. Linear Road simulates a tolling system on expressways of a metropolitan area based on variable pricing, i.e., tolls calculated based on dynamic factors such as traffic congestion and accident proximity. Linear Road processes position reports emitted periodically by every vehicle containing its position and speed on an expressway. Besides continuously-evaluated toll pricing and accident detection, the benchmark is also designed to answer historical queries (e.g., account balance and travel time estimates) that are issued less often.

The implementation of Linear Road used in this work [31] fully leverages stream-processing capabilities of the underlying platform (in this case, Flink), e.g., using sliding window operators where appropriate rather than ad-hoc management of internal state, is able to store and manage shared tables in an external key-value store, and carefully follows the original benchmark specification [12]. In

²<https://lettuce.io>

³<https://redis.io/commands/MONITOR>

⁴<https://en.wikipedia.org/wiki/MurmurHash>

⁵<https://ci.apache.org/projects/flink/flink-docs-master/api/java/org/apache/flink/api/common/functions/Partitioner.html>

Figure 4: Linear Road dataflow graph

Section 4.2 we describe Linear Road’s use of internal and externally-managed state, namely ephemeral operators state (e.g. window buffers) that are used to produce the application output vs. state of broader interest (such as historical table data) even beyond the streaming job that should be accessible outside of the SPS.

4.1 Linear Road dataflow graph

Figure 4 depicts the application as a dataflow graph. Operators are annotated with their type (filter, map, window, etc.) and a name that describes their functionality. Several operators access external state on an externally-managed store (KVS) as shown on the right of Figure 4 (in Section 4.2 we describe the state management needs of the application). The application ingests events from a Kafka topic and deserializes the raw data into structured event objects through a FlatMap operator (named *deserialize*). Next the stream splits into two substreams based on the event type, position report (event Type 0) or historical query (event Type 1). In what follows we describe Linear Road through the workings of its key operators:

VehicleStatus operator: The position report events sub-stream (upper sub-graph of Figure 4) accounts for 99% of all generated events. The stream is partitioned based on the vehicle ID (vid) that emitted the position report, and following that, the *VehicleStatus* operator identifies the status of the vehicle: moving, stopped, or crossing segment. A vehicle is considered as stopped if four consecutive position reports indicate exactly the same position identified by the triplet (xway, segment, position within the segment). To identify the status of a vehicle, the *VehicleStatus* operator maintains in its internal state the previous position of the vehicle. The internal state of the operator is maintained using Flink’s embedded state management system as the lifecycle of the state is coupled with the operator’s lifecycle (ephemeral state) and there is no need for sharing that state with other operators (more details on state management in §4.2). The operator marks the corresponding fields in the tuple accordingly and forwards it to its downstream operators. The output stream of the *VehicleStatus* operator is partitioned based on the segment ID (segID). *VehicleStatus* has three downstream operators consuming its output.

AccidentManager operator: This operator discovers accidents that occurred or cleared in the segment. An accident occurs when two stopped vehicles emit reports at the same position at the same time. For every stopped vehicle, the operator accesses the external database to check whether there is already another stopped vehicle

in the segment and reports a new accident. If this is the first stopped vehicle of the segment, the operator updates the database accordingly. For every vehicle marked as “started moving”, i.e. previously marked as stopped, now reporting a new position indicating it has started moving, *AccidentManager* updates the external database in order to clear the accident involving the vehicle.

SegStats operator: The application updates the statistics of each segment of the metropolitan area. The statistics include the number of unique vehicles and the average speed of the moving vehicles in the segment in the last 5 minutes. The statistics are updated on a per-minute basis and are used to determine toll charges. We use a 5-minutes sliding window that slides every 60 seconds emitting its output to the *SegStats* operator, which updates the external KVS.

Toll operator: For position reports marked as *crossing*, i.e. the position report indicates that the vehicle enters a new segment, the *Toll* operator dynamically calculates toll charges based on segment statistics and proximate accidents according to the variable tolling system [12]. The operator gets the information by accessing the external database. Finally, it notifies vehicles for the charges and updates the account balance of the vehicle in the database.

AccountBalance operator: Type 1 queries are events indicating that a vehicle requests its account balance. The corresponding account balance operator accesses the database and notifies the vehicle accordingly (bottom of Figure 4). Similar to the Aurora implementation of Linear Road [12], we did not implement travel-time estimation queries and input for this type of query is ignored.

4.2 Linear Road state

According to the Linear Road specification [12], historic data are usually loaded offline into a storage facility of choice. In a historical query, the system reports an account balance, a total of all assessed tolls on a given expressway on a given day, or an estimated travel time and cost for a journey on an expressway. Usually such queries provide useful insights about the metropolitan area and are often used by multiple systems and applications (e.g. reporting and visualization tools). Historic data should naturally be stored in an external store, accessible and able to be updated by the SPS. Such an external data store allows historical data to outlive the stream-processing job and be available after the end of it. In addition, the external store allows sharing of information across multiple applications/systems. This is a common practice in production environments where multiple systems involved in real-time data processing including SPSs

Redis table	Updated by	Read by
Segment stats	AccidentManager, SegStats	Toll
Account balance	Toll	AccountBalance

Table 1: Linear Road external state and its consumers

and data stores (§2). For state that is not shared across operators or between processing jobs and whose lifecycle is coupled to the lifecycle of an operator, there is no need to use an external store (e.g., state of *VehicleStatus*). Similar design principles were applied by Jain *et al.* [23] and others [35].

In our implementation of Linear Road we use Redis as an external KVS. After analyzing the semantics of Linear Road state we decided to use two separate Redis tables (summarized in Table 1). The first table stores *segment statistics*, such as number of vehicles, average velocity, and the occurrence of accidents. This table is updated periodically by two different operators, *Accident Manager* and *SegStats* and is read by the *Toll* operator, which calculates the dynamic toll charges. Operator instances processing the same substream partitions should be placed on the same node to maximize the benefit of alignment. Flink does this by default in our deployments.

The second Redis table stores the *account balance of each vehicle*. This structure is updated by the *Toll* operator and used by the *Account Balance* operator to answer the corresponding query. These operators consume different substreams, partitioned by different keys. The corresponding operator instances may be placed on different nodes by the SPS. This is the case where multiple operators that access the same data partition are placed on different physical hosts. In this case, Amoeba opts to co-locate the table with the operator that performs the most frequent access to it, determined based on monitoring data (in this case, the *Toll* operator).

5 EVALUATION

Our evaluation was carried out on the Amazon EC2 cloud. The resources used are summarized in Table 2, with all VMs allocated in AWS region us-east-1. Our Linear Road application ingests data from a 3-node distributed Kafka service. These nodes also host our data generator. The Kafka topic which Linear Road subscribes to is configured with 64 partitions. The Flink cluster deployment consists of a Flink Job Manager and up to 64 nodes hosting Flink task managers. Each node used for the task managers also hosts a Redis instance, part of the Redis cluster.

In this section we evaluate the performance improvements in various Linear Road deployments and scenarios achieved by Amoeba adaptation actions. In particular, in Section 5.1 we focus on the impact of unaligned SPS-KVS platforms and demonstrate that the throughput *per node* gets worse (if not aligned) with scale from 1 to 64 nodes. In Section 5.2 we evaluate the impact of alignment (vs. random placement) in deployments of the SPS and KVS platforms ranging from 1 to 64 AWS nodes. In Section 5.3 we focus on the dynamic alignment between SPS and KVS deployed on the same nodes and demonstrate the speed at which performance benefits are realized over time. In Section 5.4 we demonstrate an alignment plan that includes data migration as well as partition alignment across systems. Finally, in Section 5.5 we evaluate Amoeba’s support for coordinating elasticity actions: From an originally co-located and aligned setup, the SPS triggers an elasticity action. To restore

	#VMs	VM type	cores	DRAM	SSD
Kafka cluster	3	r5ad.large	2	16GB	75GB
Flink Job Mgr	1	c5d.large	2	4GB	50GB
Flink Task Mgr	1-64	c5d.xlarge	4	8GB	100GB

Table 2: AWS VM types used in our evaluation **Figure 5: Clusters with more nodes result to lower local-hit rate (table in figure) affecting the throughput per node**

an optimized configuration, Amoeba will trigger a matching KVS elasticity action, followed by partition alignment.

We carefully instrument each operator of the application graph (Figure 4) to measure the throughput (processed tuples per second) and operators processing latency. The latter metric refers to the duration from the time a tuple enters the operators’ processing task until the corresponding output is produced. For operators accessing external state, processing latency includes the cost of (local or network) data access to the KVS. Our evaluation focuses on the Toll operator, as it is the most heavily-loaded task accessing the external database, and on the critical path of the application. We thus consider it representative of overall application performance.

All reported results are the average of at least 5 runs of each experiment with a standard deviation of <1%.

5.1 Unaligned access gets worse with scale

In this section we study data-access locality between Flink tasks and Redis data partitions, in a cluster that scales from 4 to 64 AWS nodes. Each node in the cluster hosts a Flink task manager and a Redis instance. Each Toll operator instance issues data requests to Redis, served either by the local (co-located with the Flink task) Redis instance or by a remote instance hosted on another node in the cluster. We define the *local-hit rate* metric as the percentage of the total data-requests issued by an operator instance towards Redis that are served by the local Redis instance. Without proper partition alignment through Amoeba, partitions are randomly placed across cluster nodes. While each system may individually achieve good load balance, lack of coordination leads to high remote access penalizing overall performance. In this section we study the local-hit rate of data requests from a single node across all Redis instances and its performance impact on the throughput of that node.

Figure 5 depicts the drop in per node throughput for different cluster sizes, as the *per node* local-hit rate decreases (table in figure).

Figure 6: Single-partition deployment. Alignment yields 2.1x in throughput and 56% lower latency on Toll operator

We observe that the local-hit rate of each processing task decreases inversely proportional to the cluster size, penalizing per-node performance. With more Redis instances, the probability of hitting the local instance decreases even more. Scaling beyond 16 nodes does not seem to reduce per node throughput further as the local-hit rate is already close to the worst-case scenario where nearly all requests are served by remote Redis instances.

As we will see next, aligning systems through Amoeba results in 100% local hit rate, leading to high performance of each node (§5.2).

5.2 Benefits of alignment

To evaluate the performance overhead of remote data access we first deploy the application with parallelism one, where each operator is deployed as a single instance (single partition). Redis is also deployed in standalone mode (single node), either on the same node as Flink (*local*) or on a different node (*remote*). Figure 6 depicts the throughput of the Toll operator in the local vs. remote cases. Co-location leads to a 2.1x speedup in throughput. Figure 6 shows also the latency (processing time) of the Toll operator. Co-location leads to 56% lower latency compared to remote data access, avoiding the cost of a network round trip. The base network latency is measured at $11.25\mu s$ using `qperf`⁶.

Next we examine the impact of local vs. remote access to Redis tables in scaled deployments of 4 to 64 AWS nodes. Each processing operator is partitioned to multiple instances and each instance is assigned a non-overlapping key range. Each node hosts a Flink task manager and a Redis instance. By default each system follows its own partitioning scheme and partition assignment method across nodes in the cluster. This leads to cross-talk communication over the network as some of the data requests are served by the local (co-located) Redis instance while others are served by remote Redis instances (§5.1). Figure 7 depicts the aggregated throughput of all Toll operator instances as we scale our deployment from 4 to 64 nodes. The figure compares the unaligned case where each system uses its own partitioning scheme and partition placement across nodes, to the system resulting from Amoeba partition-alignment, re-assigning Flink processing partitions to operator instances across nodes. We observe that Amoeba coordination more than doubles

Figure 7: Throughput of Toll. Amoeba improves overall throughput consistently for cluster sizes 4-64 AWS nodes

Figure 8: Dynamic alignment of partition schemes

the achieved throughput in all cases, reaching up to 2.6x improvement. This experiment provides evidence that partition-alignment benefits persist with increasing cluster sizes. The speedup levels off at 2.4-2.6x beyond 16 nodes following the same trend on per-node throughput as the *local-hit rate* drops under 6% (Fig. 5, §5.1).

5.3 Dynamic alignment of partition schemes

In this section we demonstrate the ability of Amoeba to dynamically adapt SPS-KVS key routing based on the partitioning schemes it dynamically receives by the Coordinator. The Coordinator periodically probes the Flink Job Manager and Redis instances in the cluster to discover the partitions of each system and their distribution within the cluster. Amoeba applies its aligning actions whenever it discovers opportunities to improve data locality. For the sake of this demonstration we configure our adaptive partitioner to contact the Coordinator 60 seconds into a run and use a Flink-Redis deployment over 8 AWS nodes. Figure 8 presents the performance improvement achieved as the adaptive partitioner effects the optimal alignment between systems at run time. Sixty seconds into the run we see a 2.4x improvement on the aggregated throughput as well as on the throughput per node in the 8-node cluster. Note that adaptation is seamless, exhibiting no downtime.

⁶<https://github.com/linux-rdma/qperf>

Figure 9: An alignment plan that includes data migration

5.4 Combining alignment with data migration

The Amoeba coordinator takes into account topology (placement) metadata of both systems and possible constraints and generates an alignment plan to improve overall performance (§3). Here we demonstrate a scenario where the SPS operators and the KVS partitions are deployed on a disjoint set of nodes, and the nodes hosting the data (KVS) are dedicated storage nodes, thus not a target for placing or migrating processing tasks. In this case the only option to achieve alignment (locality) is to move KVS data partitions onto the nodes hosting stream-processing tasks. This experiment focuses on the 4 nodes that are initially hosting only SPS partitions.

Our Redis driver allows data migration to the SPS nodes without any noticeable service disruption by leveraging Redis data-replication mechanisms (available in many other KVSs), as described in §3.3. Having both processing tasks and data shards co-located on the same set of nodes, the next step of the adaptation plan is to align the SPS partitioning scheme to achieve locality.

Figure 9 depicts the dynamic adaptation of the system in this scenario. To clearly demonstrate the impact of each alignment step (data migration, then partition-scheme alignment) we configured the Amoeba coordinator to start the alignment process 60 seconds into the run and space alignment actions 30-seconds apart. Starting the experiment using a different set of nodes for processing operators and KVS partitions (interval 0-60 sec), requests are served by remote Redis instances incurring a performance cost for data accesses to external state over the network. At 60 seconds, the Amoeba coordinator creates the alignment plan based on the topology metadata and instructs the Redis driver to create follower replicas on the SPS nodes according to the plan.

In the interval 60-90 seconds, follower nodes catch up with their primaries, and are then promoted to primaries and start serving requests. There is no service disruption during this period, in fact we observe a 23% higher throughput as some of the data requests are served by a co-located Redis instance (in Section 5.1 we saw that local-hit rate is non-zero even without partition alignment in the systems). However the performance can be further improved: The final step of the plan is realized 90 seconds into the run, when our Flink partitioner adapts its partitioning scheme according to the plan, leading to 100% local-hit rate improving overall performance by 2x compared to the co-located but unaligned Redis instances

Figure 10: Amoeba matches elasticity actions of the SPS with corresponding actions in the KVS to maintain alignment

(consistent with the results reported in Figure 7) or 2.38x compared to the case of remote Redis nodes.

5.5 Coordinating elasticity actions

Amoeba continuously strives to achieve alignment of elasticity actions between SPS and KVS to preserve locality achieved in previous configurations. Here we demonstrate a scenario where Amoeba notices a change in a fully aligned system that is stirred by an elasticity action decided and effected by the SPS. Amoeba will then respond by triggering a corresponding action for the KVS and then re-adjust the SPS partitioning scheme to restore data locality.

The initial configuration includes 3 nodes, each hosting a Flink task manager and a Redis instance. For the sake of demonstration we intentionally space adaptation actions 60 seconds apart. Figure 10 depicts the different stages of a run and the impact of alignment steps triggered by Amoeba. The experiment starts with 3 nodes (0-60 sec), with no coordination between systems, achieving a throughput of about 17,000 tuples/sec. At 60 seconds the Amoeba coordinator aligns the key-partitioning scheme of the SPS to that of the KVS, improving overall performance in the 3 node-cluster to 27,000 tuples/sec (60-120 sec).

Next at about 130 seconds, an elasticity controller⁷ of the SPS (external to Amoeba) triggers a reconfiguration action, scaling the SPS to 4 cluster nodes. This elasticity action is effected on the Flink streaming job by externally triggering a *savepoint* (a consistent image of the execution state of a streaming job [2]), then stopping and restarting the job from that savepoint with a new parallelism property. During its elasticity action (130-137 seconds in Figure 10), the SPS stops processing tuples. Right after the job resumes processing, the system achieves about 21,000 tuples/sec. Amoeba automatically discovers that the new Flink configuration disrupts the locality achieved by previous actions. To re-align the SPS and KVS it first triggers a reconfiguration action on the KVS, creating a new Redis instance on the new node, and then triggers a background rebalancing action on the Redis cluster. The rebalancing action does not result in measurable downtime as Redis can serve requests while there are active data migrations. When the rebalancing action is over, the system reaches a stable state where both systems have

⁷The nature of such a controller is beyond the scope of our work

a processing operator and a KVS instance on each node. In the last step of the alignment plan, Amoeba instructs the SPS adaptive partitioner to apply the new partitioning scheme improving the overall system performance (at around 190 seconds), reaching about 35,000 tuples/sec. Note that had Amoeba actions not been spaced for demonstration, the throughput increase would have happened much more rapidly.

Figure 10 depicts that without coordination, the *per node* throughput is affected by the cluster size, i.e. the *per node* throughput for the 4-node cluster (135–190 seconds) is lower than the 3-node cluster (0–60 seconds) (as analyzed in §5.1). Amoeba discovers changes in the topology and produces alignment plans to preserve locality, resulting in high throughput and stable per-node performance.

As Amoeba's actions overlap in time with actions of external management systems, such as the elasticity controller in this experiment, a controller should be aware (via a distributed policy coordination solution [15]) that Amoeba adaptation actions are in progress and factor-in their delay when probing the effect of its actions in its control loop.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented a broad investigation of the benefits of Amoeba, a system we designed and implemented for exploiting data locality between processing tasks (SPS) and external state (KVS) via dynamic adaptivity actions (appropriate data or task placement, data or task migrations, alignment of partitioning schemes, and coordination of elasticity actions). Our experimental evaluation of Amoeba coordinating various deployments of Flink and Redis executing the Linear Road application, demonstrate significant (up to 260%) performance improvements in large-scale setups of up to 64 AWS nodes by reducing cross-talk between processing tasks and the corresponding external state over the network. To the best of our knowledge, dynamic adaptation to achieve co-location in this context has not been explored in previous work. Control over data placement may at times call for adaptation actions requiring state redistribution, as in elasticity actions involving stateful operators. In our ongoing work we are pursuing the integration and coordination of efficient elasticity mechanisms for KVSs and SPSs [17, 22, 33, 43] in SPS-KVS systems managed by Amoeba.

ACKNOWLEDGMENTS

The authors thankfully acknowledge funding by the Hellenic Foundation for Research and Innovation through the STREAMSTORE faculty grant (Grant ID HFRI-FM17-1998) and by the EVOLVE H2020 (GA no. 825061) project. We want to thank Chrysostomos Zeginis for discussions on the research space relating to Amoeba.

REFERENCES

- [1] [n.d.]. <https://en.wikipedia.org/wiki/Amoeba> (Accessed Aug 2021).
- [2] [n.d.]. <https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/savepoints.html> (Accessed Aug 2021).
- [3] [n.d.]. Blink: How Alibaba Uses Apache Flink. <https://www.ververica.com/blog/blink-flink-alibaba-search>. (Accessed Aug 2021).
- [4] [n.d.]. Flink Operators. <https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators> (Accessed Aug 2021).
- [5] [n.d.]. Flink Queryable State (Beta). https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/queryable_state.html. (Accessed Aug 2021).
- [6] [n.d.]. Keystone Real-time Stream Processing Platform at Netflix. <https://netflixtechblog.com/keystone-real-time-stream-processing-platform-a3ee651812a>. (Accessed Aug 2021).
- [7] [n.d.]. Redis Cluster Specification. <https://redis.io/topics/cluster-spec> (Accessed Aug 2021).
- [8] Tyler Akidau et al. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 734–746.
- [9] Tyler Akidau et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8 (2015), 1792–1803.
- [10] R. Ananthanarayanan et al. 2013. Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams. In *Proc. of SIGMOD'13*.
- [11] Eric Anderson et al. 2002. Hippodrome: Running Circles Around Storage Administration. In *Proc. of FAST '02* (Monterey, CA, USA). USA.
- [12] Arvind Arasu et al. 2004. Linear Road: A Stream Data Management Benchmark. In *Proc. of VLDB'04* (Toronto, Canada).
- [13] P. Carbone et al. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729.
- [14] Guoqiang Jerry Chen et al. 2016. Realtime Data Processing at Facebook. In *Proc. of the SIGMOD '16* (San Francisco, California, USA).
- [15] D. M. Chess and J. O. Kephart. 2003. The Vision of Autonomic Computing. *Computer* 36, 01 (jan 2003), 41–50.
- [16] S. Chintapalli et al. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE IPDPS Workshops*.
- [17] B. Del Monte et al. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proc. of SIGMOD '20*.
- [18] Y. Feng et al. 2011. Efficient and Adaptive Stateful Replication for Stream Processing Engines in High-Availability Cluster. *IEEE TPDS* 22, 11 (2011), 1788–1796.
- [19] M. Fragkoulis et al. 2020. A Survey on the Evolution of Stream Processing Systems. arXiv:CoRR abs/2008.00842
- [20] Buğra Gedik. 2014. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *The VLDB Journal* 23, 4 (Aug. 2014), 517–539.
- [21] M. Ghosh et al. 2017. Morpheus: Supporting Online Reconfigurations in Sharded NoSQL Systems. *IEEE Trans. on Emerg. Topics in Computing* 5, 4 (2017), 466–479.
- [22] Moritz Hoffmann et al. 2019. Megaphone: Latency-Conscious State Migration for Distributed Streaming Dataflows. *Proc. VLDB Endow.* 12, 9 (May 2019), 1002–1015.
- [23] N. Jain et al. 2006. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD 2006* (Chicago, IL, USA).
- [24] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthos. 2017. A Holistic View of Stream Partitioning Costs. *Proc. VLDB Endow.* 10, 11 (Aug. 2017).
- [25] Y. Kwon et al. 2008. Fault-Tolerant Stream Processing Using a Distributed, Replicated File System. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 574–585.
- [26] G. Losa et al. 2012. CAPSULE: Language and System Support for Efficient State Sharing in Distributed Stream Processing Systems. In *Proc. of DEBS'12*.
- [27] K. G. S. Madsen et al. 2017. Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine. In *Proc. of 33rd ICDE'17*.
- [28] M. A. U. Nasir et al. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *Proc. of 31st ICDE'15*.
- [29] Shadi A. Noghbi et al. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (Aug. 2017).
- [30] B. Ottenwälder et al. 2013. MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing. In *Proc. of DEBS'13* (Arlington, TX, USA).
- [31] Antonis Papaioannou. 2021. Linear Road implementation. <https://github.com/antonis-papaioannou/linearRoad>
- [32] A. Papaioannou et al. 2020. The Case for Better Integrating Scalable Data Stores and Stream-Processing Systems. In *Proc. IEEE CLUSTER'20*.
- [33] A. Papaioannou and K. Magoutis. 2017. Incremental Elasticity for NoSQL Data Stores. In *Proc. 36th IEEE Symposium on Reliable Distributed Systems (SRDS)*.
- [34] P. Pietzuch et al. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *Proc. 22nd Int. Conf. on Data Engineering (ICDE'06)*.
- [35] R. Rea. 2016. Walmart & IBM Revisit the Linear Road Benchmark. <https://www.slideshare.net/RedisLabs/walmart-ibm-revisit-the-linear-road-benchmark>
- [36] Y. Robert et al. 2012. On the complexity of scheduling checkpoints for computational workflows. In *Proc. IEEE/IFIP DSN Workshops (DSN 2012)*.
- [37] N. Senthil and G. Bugra. 2013. Using InfoSphere Streams with memcached and Redis. In *IBM developerWorks*. <https://www.ibm.com/developerworks/library/bd-streamsmemcached/>
- [38] M. A. Shah et al. 2003. Flux: an adaptive partitioning operator for continuous query systems. In *Proc. 19th Int. Conf. on Data Engineering (ICDE'03)*.
- [39] Roshan Sumbaly et al. 2013. The Big Data Ecosystem at LinkedIn. In *Proc. of the 2013 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'13)*.
- [40] The Borealis Team. [n.d.]. Borealis Application Programmer's Guide [White Paper]. http://cs.brown.edu/research/borealis/public/publications/borealis_application_guide.pdf. (Accessed Aug 2021).
- [41] QC. To, J. Soto, and V. Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* 27 (2018), 847–872.
- [42] S. Wu et al. 2012. Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should You and How Much?. In *DEBS'12* (Berlin, Germany).
- [43] Y. Wu and K. Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *Proc. of 31st IEEE Int. Conf. on Data Engineering (ICDE'15)*.