Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing

Li Wang$^+$, Tom Z. J. Fu†, Richard T. B. Ma*, Marianne Winslett‡, Zhenjie Zhang$^+$
$^+$Yitu Technology, †Advanced Digital Sciences Center, *National University of Singapore, ‡UIUC

ABSTRACT

Elasticity is highly desirable for stream systems to guarantee low latency against workload dynamics, such as surges in arrival rate and fluctuations in data distribution. Existing systems achieve elasticity using a *resource-centric* approach that repartitions keys across the parallel instances, i.e. *executors*, to balance the workload and scale operators. However, such operator-level repartitioning requires global synchronization and prohibits rapid elasticity. We propose an *executor-centric* approach that avoids operator-level key repartitioning and implements executors as the building blocks of elasticity. By this new approach, we design the Elasticutor framework with two level of optimizations: i) a novel implementation of executors, i.e., *elastic executors*, that perform elastic multi-core execution via efficient intra-executor load balancing and executor scaling and ii) a global model-based scheduler that dynamically allocates CPU cores to executors based on the instantaneous workloads. We implemented a prototype of Elasticutor and conducted extensive experiments. We show that Elasticutor doubles the throughput and achieves up to two orders of magnitude lower latency than previous methods for dynamic workloads of real-world applications.

ACM Reference Format: Li Wang, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, Zhenjie Zhang. 2019. Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing. In 2019 Int'l Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3319868

1 INTRODUCTION

Distributed stream systems [8, 12, 40, 43, 45, 50, 51] enable real-time data processing over continuous streams, and have been widely used in applications including fraud detection,

SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06...$15.00
https://doi.org/10.1145/3299869.3319868

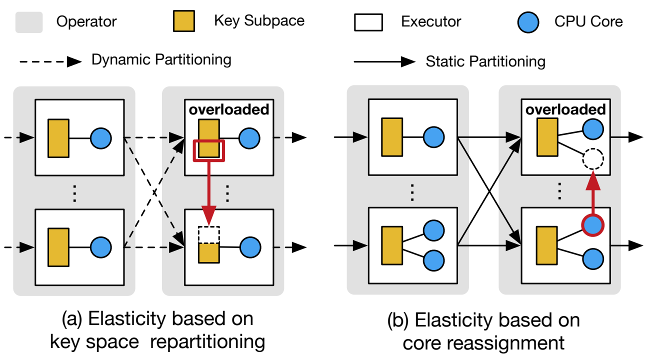


Figure 1: Comparison of elasticity mechanisms: resource-centric (left) vs. executor-centric (right).

surveillance analytics and quantitative finance. In such systems, the application logic is modeled as a graph of computation, where each vertex represents an *operator* associated with user-defined processing logic and each edge specifies the input-output relationship of data streams between the operators. To enable large-scale data processing, the input stream to an operator is often defined under a *key space* that can be partitioned into subspaces. Parallel execution instances, i.e. *executors*, are created to statically bind each key subspace to an amount of computational resource, typically a CPU core. As a result, each executor can conduct computation associated with its key subspace independently.

However, in real applications such as stock trading and video analytics, the workload fluctuates greatly with time, leading to severe performance degradation [15, 39]. From a temporal perspective, the aggregate workload fed to an operator might surge significantly in a short period of time, e.g, 10 seconds, making the operator a bottleneck for the entire processing pipeline. From a spatial perspective, the workload distribution over key space might be unstable, resulting in a skewed workload across the executors with low CPU utilization in some and overload in the others. To adapt to workload fluctuation, prior work [14, 15, 39, 41] proposed solutions to enable *elasticity*, i.e., operator scaling and load balancing. All these solutions are *resource-centric*, in that executors are bound to particular resources and elasticity is achieved by dynamically repartitioning keys across executors.

Figure 1(a) illustrates a scenario where an executor is overloaded due to imbalance in workload distribution. To relieve the performance bottleneck, the key space is repartitioned such that a certain amount of workload along with the corresponding keys in the overloaded executor is migrated to a lighter-loaded executor. However, this process requires a time-consuming protocol [15, 39] to maintain state consistency. In particular, the system needs to perform the following operations: (a) stop upstream executors from sending tuples downstream; (b) wait for all in-flight tuples to be processed; (c) migrate the state among executors according to the new key space partitioning; (d) update the routing tables of the upstream executors; and finally (e) resume upstream executors sending tuples downstream. Because both inter-operator routing update and inter-executor state migration require expensive global synchronization, key space repartitioning may last over seconds, during which new incoming tuples cannot be processed and significant delays incur.

To achieve rapid elasticity, we propose an *executor-centric* paradigm. The core idea is to statically partition the key space of an operator among executors but dynamically assign CPU cores to each executor based on its instantaneous workload. Figure 1(b) illustrates that instead of repartitioning key space, the new approach balances workload by reassigning CPU cores from a lighter-loaded executor to the overloaded executor. As each executor possesses a fixed key subspace, the new approach achieves *inter-operator independence*, i.e., upstream operators do not need to synchronize with downstream ones, and *inter-executor independence*, i.e., states associated with key subspaces do not need to be migrated across executors. In other words, this new approach gracefully decouples the binding between operator-level key space repartitioning and dynamic provisioning of computational resources.

Based on the executor-centric approach, we design the Elasticutor framework with two levels of optimization. At the executor level, implemented as a lightweight distributed subsystem, each *elastic executor* evenly distributes its workload over its assigned CPU cores and scales rapidly when the scheduler allocates/deallocates CPU cores to or from it. At the global level, a model-based dynamic scheduler is designed to optimize the core-to-executor assignment based on the measured performance metrics in order to accommodate the workload dynamics with minimum state migration overhead and maximum locality of computation. We implement a prototype of Elasticutor and conducted extensive experiments using both synthetic and real datasets. The results show that Elasticutor doubles the throughput and achieves orders of magnitude lower latency than existing methods.

The rest of this paper is organized as follows. Section 2 introduces the executor-centric paradigm and gives an overview of the Elasticutor framework. Sections 3 and 4 present the designs of elastic executors and the dynamic scheduler, respectively. Section 5 discusses experimental results. Section 6 review the related work. Section 7 concludes the paper.

2 PARADIGM AND FRAMEWORK

2.1 Basic Concepts

We consider a real-time stateful stream processing system on a cluster of machines, called *nodes*, connected by fast network devices. A *stream* is an unbounded sequence of tuples. Tuples from the input stream(s) continuously arrive at the system and are immediately processed. A user application is modeled as a directed graph of computation, called a topology, where the vertices are the operators with user-defined processing logic and the edges represent the sequence of processing among the operators. For each pair of adjacent operators, tuples of a stream are generated by the upstream operator and consumed by the downstream operator. In stateful computation, an operator maintains an internal *state*, which is used for computation and will be updated during the processing of input tuples. To distribute and parallelize the computation, the state of an operator is implemented as a divisible data structure defined on a key space. The system partitions the key space into *subspaces* and creates a parallel instance, called an executor, with identical data processing logic for each of them. To guarantee the consistency of states maintained on such a distributed system, tuples need to be correctly routed to downstream executors. Because processing the same sequence of input tuples in different orders may result in different output tuples and states, another basic requirement in stateful computation is to process the tuples of the same key in the order of arrival.

Stream processing workloads are often dynamic in that the input rate to an operator and the key distribution of tuples fluctuate over time. To guarantee the performance under a dynamic workload, computational resources, i.e., CPU cores, should be appropriately provisioned to the operators so as to ensure 1) *operator scaling*, i.e., CPU cores are dynamically allocated to operators according to their workloads; and 2) *load balancing*, i.e., the workload of each operator is evenly distributed across the allocated CPU cores. Without achieving the former, some operators may be overloaded or over-provisioned, becoming a performance bottleneck or wasting computational resources, respectively. Without achieving the latter, some CPU cores will be overloaded while others will be underutilized, resulting in poor performance. We refer to the mechanism of operator scaling and load balancing as *elasticity*. To retain high performance under dynamic workloads, rapid elasticity is a crucial requirement.

2.2 The Executor-Centric Paradigm

Table 1 summarizes the main features of the two existing paradigms of elasticity: the static and the resource-centric approach. The static approach implements each operator with a fixed number of executors and uses static operator-level key partitioning to distribute the workload among the executors.

Table 1: Comparison of three execution paradigms.

paradigms	operator-level key partitioning	CPU-to-executor assignment	elasticity
static	static	one-to-one	N/A
resource-centric	dynamic	one-to-one	slow
executor-centric	static	many-to-one	rapid

Each executor consists of a single data processing thread bound to an assigned CPU core. Due to the static key partitioning and one-to-one binding of CPU cores to executors, the static approach simplifies system implementation and is adopted in most state-of-the-art systems [30, 43]. However, since it can neither balance the workload across the allocated CPU cores nor adjust the number of CPU cores assigned to a particular operator, this approach is very sensitive to the partitioning schema and works inefficiently under a dynamic workload due to the lack of elasticity.

The resource-centric approach resolves the limitation of the static approach by supporting dynamic operator-level key partitioning, while following the same implementation of the executors as in the static approach. With the capability of operator-level key repartitioning, the resource-centric approach achieves elasticity, as it can migrate some keys with their corresponding workload from overloaded executors to the lighter-loaded executors to balance the workload, or from existing executors to a newly created executor to scale out an operator. However, as discussed in the introduction section, this operator-level key repartitioning is a time-consuming procedure, during which expensive global synchronization is required to migrate the state and to update the routing tables of all the upstream executors. Therefore, the resource-centric approach does not achieve rapid elasticity and can only tackle a very limited degree of workload dynamics.

To achieve rapid elasticity, we propose a new execution paradigm: *the executor-centric* approach. Our idea comes from the observation that the operator-level key repartitioning is too expensive to achieve rapid elasticity. Consequently, the executor-centric approach uses *static* operator-level key partitioning but implements each executor as the building block of elasticity to handle workload fluctuation. In particular, each executor is designed to utilize various amount of computation resources by creating or removing data processing threads on the fly. Therefore, to achieve load balancing and operator scaling, the system can dynamically assign an appropriate number of CPU cores to each elastic executor rather than performing the expensive operator-level key repartitioning. Compared with operator-level key repartitioning, reassignment of CPU cores and intra-executor load balancing can be achieved efficiently, since they do not need any inter-operator or inter-executor synchronization. Fundamentally, our new approach achieves rapid elasticity by avoiding global synchronization.

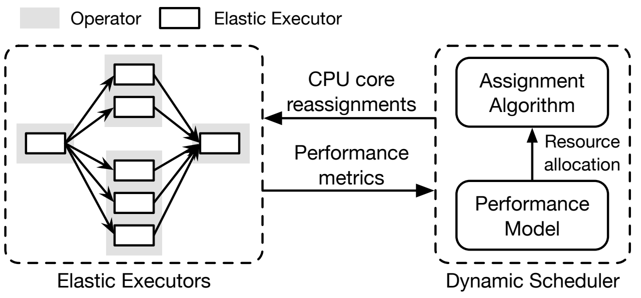


Figure 2: Overview of the Elasticutor framework.

2.3 Overview of Elasticutor Framework

Following the executor-centric approach, we design Elasticutor that focuses on supporting stateful stream processing. To process large-scale data on stream systems, we assume that data and states are defined under a key space, based on which partitioned data streams and states can be processed and maintained in parallel by distributed computational units. We assume that the key space is fine-grained enough so that even skewed workload could potentially be distributed and balanced on the increasing amount of computation resource, i.e., CPU cores. For other state-of-the-art streaming systems like Heron, Flink and Samza, this assumption is also needed to enable highly parallelized stateful stream processing.

Our design goal is to enable real-time responses, which boils down to guaranteeing low latency. However, excessive delay could be due to insufficient system resources caused by higher arrival rates of data, or inefficient resource allocation and scheduling that induce workload imbalances. The former requires resource scaling, while the latter does not. Based on the principle of separation of concerns, we design Elasticutor as a two-level architecture, as illustrated in Figure 2.

The high-level scheduler (described in Section 4) handles dynamic workload that may surge in periods of time, during which the existing system capacity is insufficient and needs to scale. Overprovisioning is not needed, but we assume that resources can be acquired on-demand from cloud-based platforms. We assume that this overall surge in workload doesn't happen too frequently, e.g., at a time-scale of minutes to hours. The dynamic scheduler determines the desirable number of CPU cores each elastic executor should be provisioned under the instantaneous workload. It employs a performance model based on queuing networks and uses collected performance metrics of the elastic executors as inputs to generate resource allocation decisions. Based on the existing core-to-executor assignment and the availability of CPU cores in the cluster, the scheduler refines the assignment to accommodate the new resource allocation plan, while taking both the CPU reassignment overhead and the locality of computational resources into consideration.

Each low-level executor (described in Section 3) is designed as a lightweight, self-contained, distributed subsystem, called an *elastic executor*, responsible for processing inputs under a fixed key-subspace. To adapt to the workload fluctuations, an elastic executor can utilize a dynamic number of CPU cores, possibly from multiple nodes, as determined by the dynamic scheduler. To fully utilize its allocated CPU cores in presence of workload fluctuation, an elastic executor has an efficient internal load balancing mechanism that evenly distributes the computation of its input stream across the allocated CPU cores in much shorter time-scales.

The design space of streaming systems that target stateful processing also include dimensions such as the state size and the characteristics of data streams, i.e., per-tuple computation and size, and the skewness and dynamicity of data streams under the key space. We will discuss the trade-offs Elasticutor makes compared to alterantive methods in Section 5.

3 ELASTIC EXECUTOR

To efficiently utilize CPU resources, an elastic executor is designed to adapt to two dynamics: 1) *changes in key distribution* and 2) *CPU core reassignments*, as illustrated in Figure 3. The former results from fluctuations in the input stream, while the latter is determined by the scheduler for global optimization. To distribute the workload over its computational resources, an elastic executor creates a *task* for each assigned CPU core and distributes input data tuples over them. Upon a CPU reassignment, a new task will be created or an existing task will be deleted. Both dynamics introduce unbalanced workload among the tasks, leading to resource underutilization or performance degradation. Therefore, a central design question is *how to keep balanced workload distribution among tasks in presence of such dynamics.*

3.1 Components and Working Mechanism

As illustrated in Figure 4, an elastic executor is implemented as a lightweight, self-contained distributed subsystem that can utilize computational resources on multiple physical nodes. Each elastic executor primarily resides in one physical node, called its *local node*, where it runs a local *main process* to receive input tuples and send output tuples. For each allocated CPU core, a task, implemented as a data processing thread, is created in the process. To utilize CPU cores on a remote node, a remote process can be created to host remote tasks for remote data processing.

Intra-Executor Routing: We employ a two-tier design, implemented in the routing table shown in the central rectangle in Figure 4, to dynamically map input tuples to the tasks based on the instantaneous workload distribution. The first tier statically partitions the key subspace into *shards* using a static hashing function; the second tier explicitly

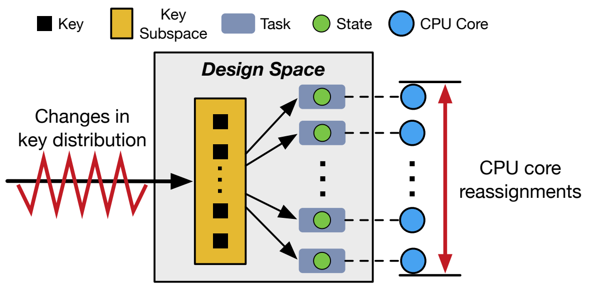


Figure 3: The design space of elastic executor against changes in key distribution and core reassignment.

maintains a dynamic shard-to-task mapping, which gets updated upon shard reassignments. We balance the workload on a coarser-grained rather than a per-key basis, mainly because a fine-grained method needs to maintain the workload for every single key and thus suffers from high memory consumption. The choice of the number of shards provides trade-offs between the quality of load balancing and maintenance overhead. However, in practice, a reasonable number of shards, e.g., 4 or 8 times to the number of tasks, achieves good balancing quality while keeping low maintenance overhead. We will discuss how shard number affects system performance in some extreme settings in Section 5.3.

To guarantee state consistency, states have to be migrated along with their shards among the tasks, leading to migration overhead and delay. Consequently, for rapid load balancing, the number of shard reassignments should be minimized. This optimization problem can be interpreted as a NP-hard multi-way partitioning problem [29]. We use a simple heuristic algorithm similar to the First-Fit-Decreasing algorithm [19] to solve it. Our intra-executor load balancing algorithm refines the shard-to-task assignment in rounds until the workload imbalance factor δ, defined as the ratio of the maximum task workload to the average workload of all tasks, is below a predefined threshold θ. In each round, among all the possible reassignments that reassign a shard from the most overloaded task to the least loaded task, the algorithm picks the shard reassignment that reduces δ the most. In our implementation, the workload of a particular shard is measured as the aggregated computation cost of the input data tuples processed within one-second sliding windows. We choose $\theta = 1.2$, allowing a maximum imbalance of 20% deviation from the average workload of the tasks. The reassignment is triggered frequently, e.g., every 0.5 second, to guarantee responsiveness to workload changes.

Intra-Executor State Management: To minimize the state migration overhead and enable efficient state access simultaneously, we employ an *intra-process state sharing* mechanism in the elastic executors. In particular, each process of an elastic executor maintains the states of its tasks in a lightweight

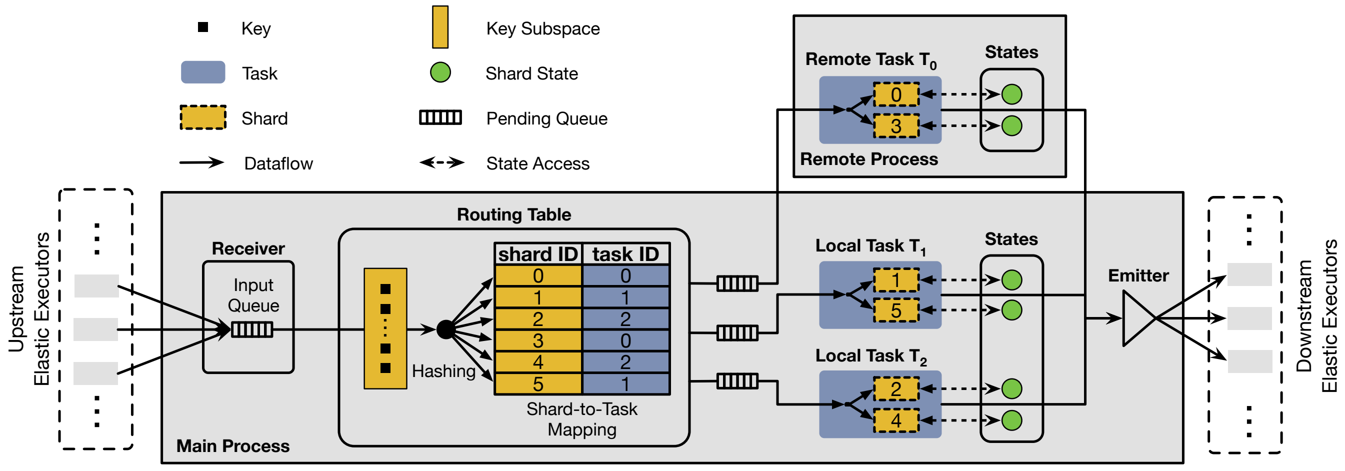


Figure 4: The internal structures and working mechanisms of an elastic executor.

in-memory key-value store and provides a *state access interface* to its tasks for state reads and updates on a per-key basis. When shards are reassigned across nodes, up-to-date states are always migrated among them to guarantee the state consistency, which will be discussed in Section 3.2. While retaining efficient state access performance, this design avoids state migration when shards are reassigned between tasks on the same node, because the newly assigned task can always access the shard's state via the interface without state migration. Given the increasing number of CPU cores on modern processors, many tasks can be created on a single node. Consequently, intra-process state sharing can significantly reduce shard reassignment overhead. Furthermore, our dynamic scheduler also optimizes the locality of CPU resources for the elastic executors, providing the executors more opportunities to benefit from state sharing.

Executor-Level Fault Tolerance: Fault tolerance [8, 11, 14, 35, 49] has been extensively studied in stream processing systems and thus is neither the focus nor a contribution of this paper. Here, we only discuss how to recover the remote tasks of each elastic executor from failures so that Elasticutor can utilize state-of-the-art state checkpoint techniques, such as Pipelined Snapshotting Protocol [11], for fault tolerance.

The main process of an elastic executor logically maintains a primary copy of the state of its tasks. By default, the state is maintained in memory in the main process for efficient access, but can also be stored on an external storage when the state is too large to fit in memory. The main process also marks each input data tuple with an increasing tuple ID. For each remote task T, the elastic executor maintains a pending tuple queue to backup the data tuples sent to T and a value t_s indicating that all the data tuples with IDs smaller than t_s have been processed and the state updates resulted from processing those tuples have been flushed back to the primary copy. Each remote task periodically, e.g., every 10 seconds, sends the updates of its local state together with t_{max}, i.e., the largest ID of the tuples it has been processed, to the main process of the elastic executor. Upon receiving the state updates from a remote task T, the elastic executor updates the main copy of the state, removes the tuples with ID no large than t_{max} in the pending queue of T, and updates $t_s = t_{max}$. When a remote task T fails, the elastic executor creates a new tasks with the main copy of the state and starts the execution of the new task by replaying tuples with IDs larger than t_s in the pending queue of T.

3.2 Consistent Workload Redistribution

Although state sharing improves the efficiency of shard reassignment, attention needs be paid to guarantee consistency. Generally speaking, despite using the similar procedure as in key repartitioning of the resource-centric approach, we achieve efficient shard reassignment with state consistency by taking advantage of the inter-operator and inter-executor independence enabled by the executor-centric approach.

Consider the case in Figure 4, where a tuple t_1 is in the pending queue of task T_2, a tuple t_2 just arrived at the executor's main process, and a tuple t_3 is to be emitted by an upstream executor. Suppose all three tuples belong to shard r_4. If shard r_4 is reassigned from the *source task* T_2 to a new destination task before t_1 is processed or before the routing of t_2 and t_3 are updated, the state will become inconsistent. In particular, if the destination task is local, e.g., T_1, then t_2 might be processed before t_1, violating the order preserving requirement. If the destination task is remote, e.g., T_0, the modifications to states made by t_1 will be lost.

Inter-Operator Consistent Routing: To guarantee consistent routing, e.g., t_3, from upstream operators to the correct processes where the assigned tasks reside, an elastic executor implements a *receiver* daemon in its local main process as the single entrance for all tuples coming from upstream operators. The receiver routes tuples to the appropriate tasks, local or remote, based on the internal routing table. Similarly, an *emitter* daemon is implemented in the main process as the single exit of the executor to forward output tuples generated by the tasks to downstream operators. Remote processes only communicate with the receiver and the emitter on the main process of the elastic executor. Therefore, regardless of how shards are dynamically reassigned among the tasks within an elastic executor, upstream and downstream operators always send tuples to or receive tuples from the executor via its receiver and emitter, avoiding any inter-operator synchronization caused by shard reassignments. In contrast, the resource-centric approach redistributes workload by operator-level key space repartitioning, leading to synchronization with all the upstream executors.

Note that compared with the resource-centric approach where tuples from upstream executors are directly routed to the downstream operator, Elasticutor may involve additional remote data transfer between the receiver/emitter and the remote tasks. This is the trade-off we make to achieve rapid elasticity. In typical workloads, the remote data transfer is not the performance bottleneck, as shown in Figure 13. In Section 5.3, we discuss how to avoid/reduce remote data transfer in some extreme workloads by properly configuring the number of executors of an operator.

Intra-Executor State Consistency: To guarantee state consistency during the reassignment of a shard, the elastic executor employs a key repartitioning procedure similar to the operator-level repartitioning used in the resource-centric approach, but does not involve any global synchronization. The key is to ensure that a) the pending tuples, i.e., the unprocessed tuples of the shard queued in the source task, must be processed before the shard state is migrated to the destination task; and b) tuples with the same keys will not be processed by any two tasks concurrently. During the reassignment of shard r_4 in Figure 4, the routing for tuples of r_4 is paused and a labeling tuple is sent to its source task T_2. Since tasks process their input tuples on a first-come-first-served basis, any pending tuple already sent to T_2 is guaranteed to be processed when T_2 pulls the labeling tuple from its pending queue. After that, the state of r_4 is migrated to the destination task. State migration is omitted if the shard is reassigned to a task local to its source task. After the state migration, the shard-to-task mapping is updated in the routing table before the routing for tuples of r_4 is resumed.

Discussions: It is worth noting that our proposed executor-centric paradigm is applicable to other existing distributed streaming systems, e.g., Apache Flink, Apache Heron, and Apache Samza, where stateful processing can be parallelized by partitioning states and data under a key space. For stateless applications, our approach can still be applied but may not necessarily be the best choice, since load balancing can be easily achieved by simply sending tuples in round robin or to the least-loaded executors.

Although our approach does not apply to batch-based systems, our two-tier load balancing design has some similarities with the approaches taken by the mini-batch oriented Spark Streaming [50]. Two major differences are: 1) our design of an extra intermediate layer of shards provides trade-offs between maintenance costs and balanced load, and 2) our design for measuring and balancing are more natural to streaming system where operations process input tuples upon arrival rather than based on mini-batches.

4 DYNAMIC SCHEDULER

The objective of the dynamic scheduler is to satisfy user-defined latency requirements by adaptively allocating CPU cores to the elastic executors under a changing workload. By using instantaneous performance metrics measured by the system, the scheduler first estimates the number of cores needed for each executor based on a queueing network model, and further (re)assigns the physical cores to the executors so as to minimize the reallocation overhead and maximize the locality of computation within the executors.

4.1 Model-Based Resource Allocation

We model a topology $\mathcal{E} = \{1, \cdots, m\}$ of m elastic executors as a Jackson network, in which each executor $j \in \mathcal{E}$ is regarded as an M/M/k_j system [42], where k_j denotes the number of allocated CPU cores to j. The average processing latency of an input stream, denoted as $\mathbb{E}[T]$, can be calculated as a function of the resource allocation decision $\mathbf{k}$ as

$$\mathbb{E}[T](\mathbf{k}) = \frac{1}{\lambda_0} \sum_{j=1}^{m} \lambda_j \mathbb{E}[T_j](k_j), \tag{1}$$

where λ_0 denotes the arrival rate of the input stream, T_j and λ_j denote the average processing time and the arrival rate of executor j, respectively. Each $\mathbb{E}[T_j](k_j)$ is bounded when $k_j > \lambda_j/\mu_j$, where μ_j denotes the processing rate of elastic executor j and can be calculated as a function of the parameters λ_0, $\{\lambda_j\}$ and $\{\mu_j\}$ measured by the system. Based on Equation (1), the scheduler attempts to find an allocation $\mathbf{k}$ to ensure that $\mathbb{E}[T]$ is no larger than the user-specified latency target T_{max}, while minimizing the total number of CPU cores, i.e., $\sum k_j$. In particular, each k_j is initialized to be $\lfloor \lambda_j/\mu_j \rfloor + 1$, which is the minimal requirement to make the system stable. We repeatedly add 1 to the value in the vector $\mathbf{k}$ that leads to the most significant decrease in $\mathbb{E}[T]$, until $\mathbb{E}[T] \le T_{max}$ or $\sum k_j$ exceeds the number of available CPU resources. This greedy algorithm has shown to be optimal [22] in finding the solution $\mathbf{k}$.

4.2 CPU-to-Executor Assignment

The performance model only suggest a new allocation, i.e., the number of CPU cores each executor needs, resulted from the workload fluctuation; the scheduler still needs to accommodate the new allocation plan by updating the existing core-to-executor assignment. The CPU reassignment for a new allocation plan is key to the system performance, since it may introduce 1) the state migration costs during the transition, and 2) the remote data transfer costs afterwards. For instance, upon the reassignment of a CPU core, an elastic executor creates a new task, which involves in state migration and future remote data transfer if the CPU core is remote to the elastic executor. To optimize execution efficiency, we search for CPU-to-executor assignments that minimize migration costs, while constraining the computation locality to limit future remote data transfer costs.

To model the migration costs, we consider a cluster of n nodes where each node i has c_i CPU cores. For any executor $j \in \mathcal{E}$, we denote the node where its main process resides by $I(j)$ and the number of cores assigned to it on all nodes by a column vector $\mathbf{x_j} = (x_{1j}, \cdots, x_{nj})^T$. We define $X_j = \sum_{i=1}^{n} x_{ij}$ as the total number of assigned cores for j and denote a CPU-to-executor assignment by a matrix $X = (\mathbf{x_1}, \cdots, \mathbf{x_m})$. Given any new allocation $\mathbf{k}$, a transition from an existing assignment $\tilde{X}$ to a new assignment X needs to perform a set of CPU allocations/deallocations. The overhead of core reassignment is dominated by the state migration cost, which is proportional to the size of state moved across the network. We denote the aggregate state size of any executor j by s_j. For simplicity, we assume the shards of an elastic executor are evenly distributed across the allocated CPU cores; and therefore, the amount of state data associated with each CPU core is approximately s_j/X_j. Given any allocation $\mathbf{k}$, available cores $\mathbf{c}$ and an existing assignment $\tilde{X}$, we formulate the CPU assignment problem as follows.

$$\underset{X}{\text{minimize}} \quad C(X|\tilde{X}) = \sum_{j=1}^{m}\sum_{i=1}^{n} \max\left(0, \frac{s_j\tilde{x}_{ij}}{\tilde{X}_j} - \frac{s_j x_{ij}}{X_j}\right)$$

$$\text{s.t.} \quad \text{(a)} \sum_{j=1}^{m} x_{ij} \le c_i, \quad \forall i \le n;$$

$$\text{(b)}\ X_j \ge k_j, \quad \forall j \in \mathcal{E};$$

$$\text{(c)}\ x_{I(j)j} = X_j, \quad \forall j \in \mathcal{E}(\varphi).$$

The above optimization problem minimizes the migration costs $C(X|\tilde{X})$ of transition from an existing assignment $\tilde{X}$ to a new assignment X, where each term in the summation measures the cost for executor j to migrate its state out of node i. The constraints include (a) the number of CPU cores, (b) the allocation requirement and (c) the *computation locality*, i.e., requiring all cores assigned to the set $\mathcal{E}(\varphi)$ of executors to be on their local nodes. The system measures the instantaneous per-core data-intensity of any executor j by its total input and output data rates divided by the number of cores k_j, and $\mathcal{E}(\varphi)$ denotes the set of executors whose data-intensity is above a threshold φ. Because data-intensive executors will incur higher network costs if their assigned cores are remote, we enforce the computation locality by avoiding assigning remote cores to members of $\mathcal{E}(\varphi)$. This integer programming problem can be reduced to the NP-hard multiprocessor scheduling problem [23]. Thus, we design an efficient greedy Algorithm 1 to find an approximate solution. For any assignment X, we define $\mathcal{E}^+ = \{j \in \mathcal{E}|X_j < k_j\}$ to be the set of under-provisioned executors, $\mathcal{E}^+_\Delta = \{j \in \mathcal{E}^+ \cap \mathcal{E}(\varphi)\}$ to be the subset of data-intensive executors, and $\mathcal{E}^- = \{j \in \mathcal{E}|X_j > k_j\}$ to be the set of over-provisioned executors. We use $C^+_{ij}(X)$ and $C^-_{ij}(X)$ to denote the overhead of allocating/deallocating a CPU core on node i to/from executor j, respectively, which can be derived as $C^+_{ij}(X) = s_j(X_j - x_{ij})/(X_j(X_j+1))$ and $C^-_{ij}(X) = s_j(X_j - x_{ij})/(X_j(X_j-1))$.

Algorithm 1: Dynamic Allocation Algorithm

```
Input: allocation $\mathbf{k}$, assignment $\tilde{X}$, CPU cores $\mathbf{c}$, threshold $\varphi$
Output: new assignment $X$
1  Initialize the new partitioning as $X = \tilde{X}$;
2  Find the under- and over-provisioned executors $\mathcal{E}^+$ and $\mathcal{E}^-$,
     and the data-intensive executors $\mathcal{E}^+_\Delta$;
3  Sort $\mathcal{E}^+$ based on the data-intensity of the executors;
4  for each $j \in \mathcal{E}^+$ in non-descending order do
5      while CPU cores are insufficient, i.e., $X_j < k_j$ do
6          if $j$ is data-intensive, i.e., $j \in \mathcal{E}(\varphi)$ then
7              $i = I(j)$;   $j' = \arg\min_{\hat{j} \in \mathcal{E} \setminus \mathcal{E}^+_\Delta} C^-_{i\hat{j}}(X)$
8          else
9              $(i, j') = \arg\min_{\hat{j} \in \mathcal{E}^-, 1 \le \hat{i} \le n} C^-_{\hat{i}\hat{j}}(X) + C^+_{\hat{i}\hat{j}}(X)$
10         if $(i, j')$ is found then
11             $x_{ij'} = x_{ij'} - 1$;   $x_{ij} = x_{ij} + 1$
12         else
13             return FAIL;
14     return $X$
```

Algorithm 1 sorts the executors in $\mathcal{E}^+$ by data-intensity in descending order and tries to assign the target number of CPU cores to each executor j one by one by deallocating cores from other executors. Specifically, if elastic executor j is data-intensive, i.e, $j \in \mathcal{E}(\varphi)$, it only accepts CPU cores on node $i = I(j)$, to avoid creating remote tasks. Consequently, among all the non-data-intensive executors, the algorithm finds a CPU core on node $I(j)$ that can be reassigned to j with minimal deallocation overhead (Line 7). In contrast, if j

is not data-intensive, it accepts CPU cores on any node. The algorithm searches all the executors in $\mathcal{E}^-$ for an executor with a CPU core that can be reassigned to j with the minimal deallocation and allocation overhead (Line 9). In either case, if such a valid core reassignment is found, the algorithm added it to the new assignment X; otherwise, it returns `FAIL`, which indicates that no feasible solution can be found and implies that a higher data-insensitivity threshold φ is required to obtain a feasible solution.

The choices of φ provide trade-offs between the feasibility of Equation 4.2 and the computation locality of the elastic executors. Since the dynamic assignment algorithm is very efficient, we run the algorithm using a low default value $\varphi = \tilde{\varphi}$. If no feasible solution is found, we double φ and rerun the algorithm until we find one. In our experiments, we set $\tilde{\varphi}$ to be 512 KB/s, below which the benefit of computation locality is negligible.

Discussions: Our design of the dynamic scheduler applies to stream processing that uses continuous operators and follows the dataflow model [5]. The scheduler determines the resources needed for each executor to fulfill latency requirements, and calculates the resource assignment for minimizing state migration costs. Other schedulers work at this level include DS2 [28] for Flink, Dhalion [21] for Heron, RAS [36] for Storm, and so on. In contrast, cloud-based resource management systems such as YARN [44] and Mesos [25] are more cluster-centric [26, 27], i.e., they primarily aim to manage cluster resources among different applications. They usually receive resource demands from the application manager and make decisions on how to provision resources based on criteria like efficiency and fairness. A negotiator/coordinator module is commonly developed for assisting interactions between schedulers at different levels. Typical examples include Storm-on-Yarn [3] and Flink-on-Yarn [2].

5 PERFORMANCE EVALUATION

We implemented a prototype of Elasticutor in about 10,000 lines of Java on Apache Storm [43]. The source codes of Elasticutor are available at [4]. Storm is a popular distributed stream processing system which exposes low level APIs, such as the Bolt API. This is relatively easier for prototyping research ideas. Storm follows the static approach and its operators are implemented by users via an abstract class, *Bolt*. We added a new abstract class, *ElasticBolt*, which provides the same programming interface as Bolt but exposes a new state access interface to the user space. For any operator defined as an ElasticBolt, Elasticutor creates a number of elastic executors with built-in state management, metrics measurement and elasticity functionalities. The dynamic scheduler is implemented as a daemon process running on Storm's master node (nimbus).

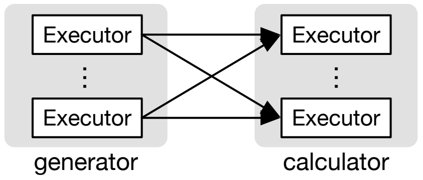
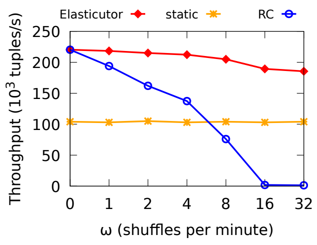


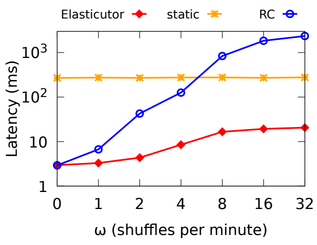
Figure 5: Micro-benchmarking simulation topology.

Our experiments are conducted on Amazon EC2 with 32 t2.2xlarge instances (nodes), each with 8 CPU cores and 32GB RAM running Ubuntu 16.04. The network is 1Gbps Ethernet. Executors are assigned to the nodes in a round-robin manner under all approaches. Unless otherwise stated, Elasticutor uses 32 elastic executors per operator and 256 shards per executor (8192 shards per operator). For fair comparison, we create enough executors for the operators in the static approach to fully utilize all CPU cores in the cluster; and set the granularity of key space partitioning in the RC approach to be 8192 shards per operator, the same as in Elasticutor. To ensure system stability, existing stream systems such as Storm, Heron and Flink implement back-pressure mechanisms to control the input rates to the operators. To focus on system performance, we evaluate pressured scenarios where sufficiently high arrival rates keep the input queues non-empty and possibly trigger Storm's back-pressure mechanism.

We briefly discussed Elasticutor in the design space of stateful streaming processing in Section 2.3. In this section, we compare the performance of Elasticutor with that of the static approach (the default Storm) and resource-centric (RC) approaches. The key differences in the three approaches are summarized in Section 2.2. We implemented RC based on Storm by enabling creation/deletion of executors and operator-level key repartitioning. For fair comparison, RC uses the same performance model, load balancing algorithm and intra-process state sharing mechanism as Elasticutor. We will evaluate the peformance and trade-offs that Elasticutor makes along the different dimensions in the design space, including the state size, per-tuple computation and size, and the skewness and dynamicity of data streams. In general, the design of Elasticutor amis to accommodate computation-intensive workloads as long as sufficient computational resources are available for scaling in the system. However, because the introduction of remote tasks might incur data transmission and state migration overhead and delays, our design assumes that the workload will not be too data-intensive with respect to tuple size and state size, and the network bandwidth capacity does not become the bottleneck. We assume that the skewness exhibited in the data distribution on the key space is a norm and we focus on more challenging scenarios where the skewness also changes abruptly, which can be shown in the equity trading data set and evaluation.



(a) Throughput



(b) Processing latency

Figure 6: Comparison with varying ω.

5.1 Micro-Benchmarking

In this subsection, we use a simple yet representative topology, shown in Figure 5, which allows easy control over the workload characteristics, such as input rates, computation cost and data distribution. The topology consists of a generator and a calculator, and input data stream are fed by the generator to the calculator for processing. We ensure that the data generation rate saturates the input queues of the calculator. The processing time for each tuple in the calculator operator follows a normal distribution $\mathcal{N}(\mu, \delta^2 = 0.5\mu)$. The computation is implemented by running data encryption in loop within the execution time to exhaust the CPU cycles and simulate the computation-intensive workload. Unless otherwise stated, each tuple consists of an integer key and a 128-byte payload, and takes an average CPU cost of 1 ms for processing. The key space contains 10K distinct values, whose frequencies follow a zipf distribution [37] with a skew factor of 0.5. The default state size is 256MB, with 32KB for each shard. To emulate workload dynamics, we shuffle the frequencies of tuple keys by applying a random permutation ω times per minute.

Robustness to workload dynamics: Figure 6 plots the throughput and average processing latency under the three approaches as ω varies along the x-axis. We observe that Elasticutor consistently outperforms the others in terms of both metrics when the workload is dynamic, i.e., $\omega > 0$. Specifically, the performance of the static approach is poor due to workload imbalance caused by skewed key distribution, but is relatively stable across all scenarios as no elasticity operations are performed. Since both RC and Elasticutor are able to adapt to skewed key distribution, they outperform the static considerably when ω is small. However, as ω increases, although the performance of both RC and Elasticutor decreases due to higher operational costs for elasticity, the performance degradation of Elasticutor is marginal, while that of RC becomes 2-3 orders of magnitude larger, making RC useless as ω reaches 16.

To better explain the performance of the three approaches as ω varies, we focus on the scenario of $\omega = 2$, i.e., shuffle every 30 seconds, and plot the instantaneous throughput measured in a sliding time window of 1 second in Figure 7.

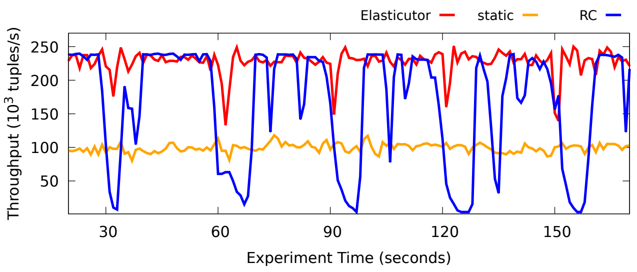
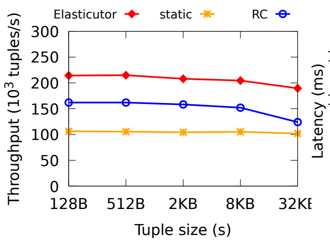


Figure 7: Instantaneous throughput with $\omega = 2$.

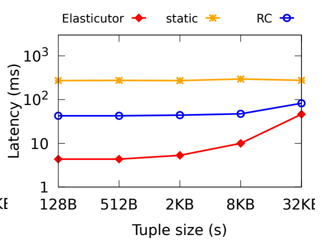
We observe that the throughput of the static approach is consistently much lower than that of RC and Elasticutor, although it does not vary much. Both RC and Elasticutor exhibit a transient throughput degradation every 30 seconds, due to the executions of elasticity operations triggered by key shuffles. However, the degradation in RC is much worse and its transient period lasts 10 to 20 seconds, while that of Elasticutor only lasts 1 to 3 seconds. This explains the reason behind the widening performance gap in the two approaches as the workload becomes more dynamic.

Performance under varying data intensity: To evaluate how data intensity of the workload affects the performance of the three approaches, we vary the tuple size, denoted as s, and the computation cost per tuple, denoted as c, and compare the performance of the three approaches in Figure 8. The results show that with higher data intensity, e.g., with large tuple size or smaller computation cost per tuple, throughput drops for the three approaches due to the higher data transfer overhead. For instance, when $c = 0.01$ms and $s = 2$KB, the data transfer requirement for full-speed tuple processing on one CPU core is 2Gbps, which exceeds the network bandwidth, i.e., 1Gbps, and thus leads to significant performance degradation for all the approaches. However, Elasticutor is generally more sensitive to tuple size than its competitors especially when the computation cost is extremely low, e.g., $c = 0.01$ms per tuple, because of its unique two-level tuple routing mechanism, which introduces higher data transfer overhead with higher data intensity.

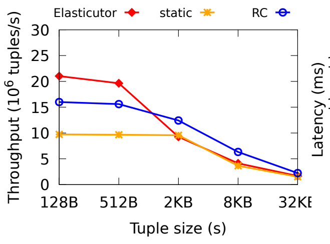
Performance under varying state size: Figure 9 compares the performance of three approaches in terms of both throughput and latency as the state size varies along the x-axis. Note that as there are 8192 shards per operator, the state size of an operator will be 256GB when the state size per shard is 32MB, which is considerably large. The result shows that as the state size increases, the performance of both the RC and Elasticutor drops, due to the increased state migration overhead associated with larger state sizes. When the state size approaches 32MB, as an extreme case, both Elasticutor and the RC perform worse than the static approach because of the huge operational costs of performing elasticity. We also observe that with the same state size, Elasticutor performs better than the RC approach. This indicates



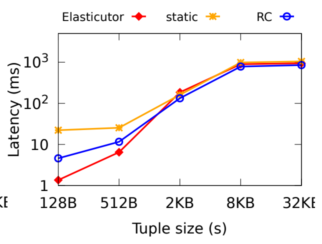
(a) Throughput with c=1ms



(b) Latency with c=1ms



(c) Throughput with c=0.01ms

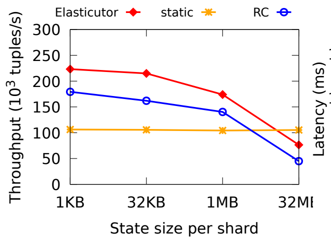


(d) Latency with c=0.01ms

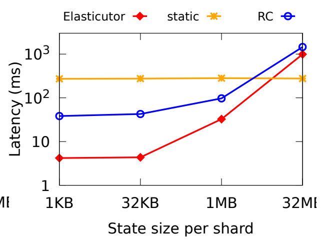
Figure 8: Varying computation costs and tuple sizes.

that the techniques used in Elasticutor, such as state sharing mechanism and the dynamic schedule, can effectively reduce the state migration overhead in the elasticity operations.

Robustness to skewed executor workload distribution: In practice, the workload may not be evenly distributed among the executors, either due to the skewed key distribution or due to the improper configuration of the operator-level key partitioning function. To evaluate the robustness of the three approaches to the skewed executor workload distribution, we evaluate their performance with varying key distribution skewness controlled by the skewness factor α in Figure 10. Note that the larger α is, the larger skewness the key distribution has. For instance, the key follows a uniform distribution when $\alpha = 0$, while most workload falls into a few keys when $\alpha \geq 0.8$. The results show that the static approach suffers greatly from load imbalance with no surprise, while Elasticutor and the RC are much more resistant to the executor load imbalance when $\alpha < 0.8$. The major observation is that Elasticutor consistently outperforms the RC when $\alpha \leq 0.6$, but its performance drops sharply and is worse than the RC under extremely skewed workload distribution, e.g., $\alpha \geq 0.7$. This indicates that despite relying on creating more remote tasks to handle skewed executor workload distribution, the executors in Elasticutor are able to handle workload imbalance up to $\alpha = 0.5$, without introducing noticeable latency increase and throughput degradation in running remote tasks. However, when $0.6 \leq \alpha \leq 0.8$, the most overloaded executors cannot further offload its workload by effectively utilizing more remote tasks, mainly due to the congested network bandwidth, consequently becoming the performance bottleneck and resulting in poor system throughput and latency.

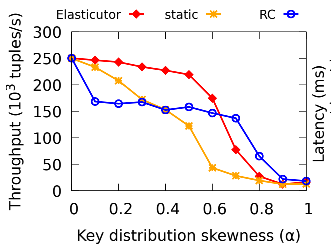


(a) Throughput

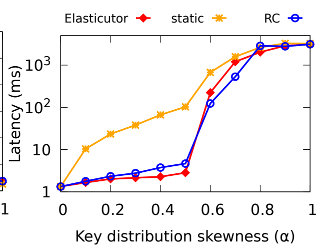


(b) Processing latency

Figure 9: Comparison with varying state size.



(a) Throughput



(b) Processing latency

Figure 10: Comparison with different key skewness.

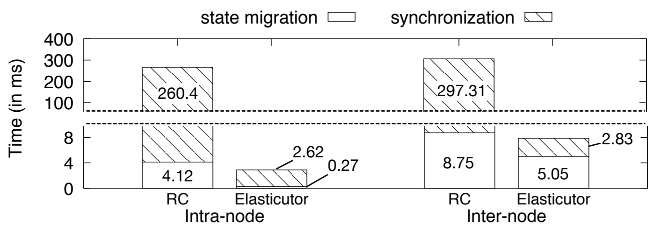


Figure 11: Breakdown of shard reassignment time.

Shard reassignment cost: Because both the RC approach and Elasticutor use shard reassignment to balance the workload, we compare their costs to better understand the different delays incurred. Figure 11 shows the average intra- and inter-node reassignment time per shard, broken down into synchronization time and state migration time. We observe that the shard reassignment time is much longer in RC than in Elasticutor, mainly due to the extremely long synchronization time in the RC approach. We can also see that Elasticutor takes shorter time in state migration than RC, but the difference between the two approaches in state migration is minor compared to that in synchronization time.

To gain insights into the synchronization time differences between the two approaches, we vary the number of upstream executors and find that RC takes 2-3 orders of magnitude longer time to synchronize than Elasticutor and their difference widens with more upstream executors, as shown in Figure 12(a). Elasticutor follows the executor-centric paradigm and thus avoids synchronization with upstream executors during the shard reassignments. As a result, its synchronization time is around 2 ms regardless of the number of upstream executors. In contrast, in the RC approach, the

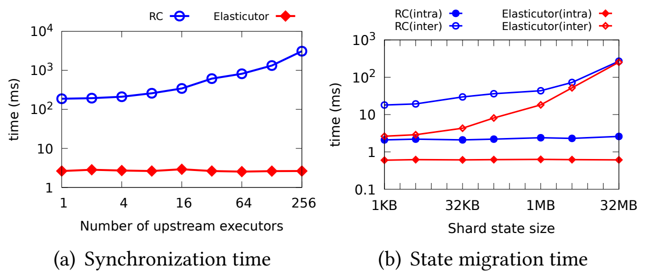


Figure 12: Effect of the number of upstream executors and the state size.

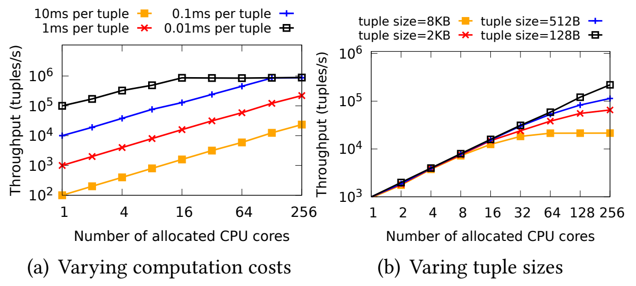


Figure 13: The scalability of a single elastic executor.

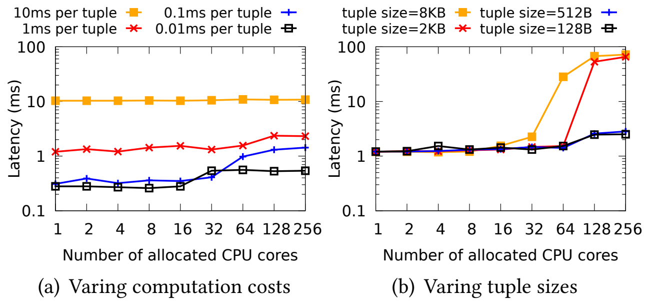


Figure 14: The 99th percentile latency as scaling out CPU cores.

routing tables of upstream executors need to be updated and global synchronization is required to clean the in-flight tuples between the executor and the upstream executors. Consequently, the synchronization time in RC is much higher and grows greatly with the number of upstream executors.

Figure 12(b) plots the state migration time as the state size varies. We observe that the latency of intra-node state migration is negligible in both approaches, because of the intra-process state sharing mechanism. The time of inter-node state migration increases significantly as the state size reaches 32 MB, where network data transfer of the state is the dominant overhead in the state migration process. The figure also shows that given the same state size, the Elasticutor takes slightly shorter time to migrate the state than RC, due to inter-executor independence enabled by the executor-centric paradigm.

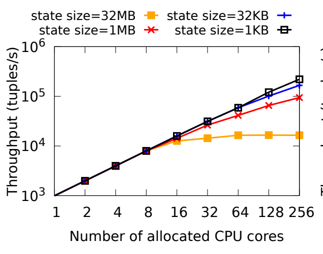
5.2 Scalability of a Single Elastic Executor

The major advantage of Elasticutor is that it handles workload dynamics by allocating more CPU cores rather than operator-level key space repartitioning. Although in a reasonable setting an operator typically has enough executors to amortize the workload on a single executor, it is still possible that an executor may be so heavily loaded that many remote tasks are needed, due to skewed key distribution, improper operator-level partitioning or unnecessarily few executors. Consequently, for robustness of Elasticutor, it is crucial that an elastic executor has good *scalability*, i.e., being able to efficiently scale out to many CPU cores, and not introducing noticeable latency when running remote tasks.

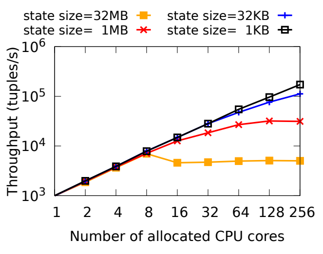
To evaluate to what extend the elastic executor can efficiently scale out, we set up only ONE elastic executor for the calculator operator, but gradually allocate more CPU cores and measure its throughput and processing latency. As each node has 8 CPU cores, the first 8 cores allocated are local, with the subsequent ones being remote. In our evaluation, we vary data intensity and operational cost of elasticity, which are the major factors affecting the scalability. The former decides the long-term cost of remote data transfer in running a remote task, and is proportional to tuple size and reversely proportional to the computational cost per tuple. The latter affects the short-term transit overhead in performing elasticity operations, and has positive correlation with the state size and workload dynamics (ω).

Figure 13 plots the scalability of an executor under different computational costs (left) and tuple sizes (right). We observe that the single elastic executor generally can efficiently scale out to the whole cluster (256 CPU cores), indicating that cost of remote data transfer is negligible. We also observe that the elastic executor cannot efficiently utilize more than 16 CPU cores with a very large tuple size, e.g., 8KB, or very low computation cost, e.g., 0.01ms per tuple, indicating that the huge remote data transfer linked to the high data intensity prevents the executor from scaling. Figure 14 shows the 99% latency as an elastic executor scales out. We can see that in most settings, processing latency does not increase noticeably as the elastic executor scales out, due to the efficient network data transfer enabled by Netty [1]. However, in the data-intensive workload, e.g., computational cost $\leq$ 0.1ms or tuple size $\geq$ 2KB, the latency increases greatly as the number of allocated CPU cores exceeds the points where remote data transfer becomes the performance bottleneck. Note that the latency does not grow infinitely, due to the back-pressure mechanism we implemented between any pair of input-output executors.

Figure 15 shows the scalability of an elastic executor under various shard state sizes with $\omega = 2$ (left) and 16 (right). The results show that the elastic executor scales efficiently



(a) $\omega = 2$



(b) $\omega = 16$

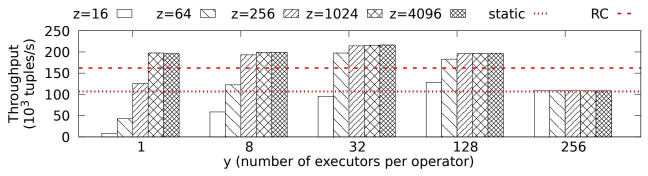
Figure 15: Varying the operational cost of elasticity.

under all state sizes but 32MB. With a large state size, the state migration becomes a performance bottleneck, which prevents the executor from efficiently using remote CPU cores. By comparing both sub-figures, we observe that as ω increases to 16, the scalability under the large state size decreases considerably, due to the increased requirement of state migration linked to higher workload dynamics.

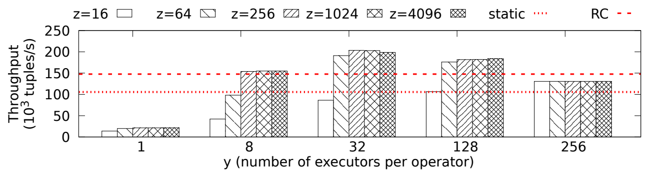
5.3 Choosing Appropriate Parameters

There are two important parameters in Elasticutor: the number of shard per executor, denoted as z, and the number of executors per operator, denoted as y. As a rule of thumb, setting z to be between 256 and 1024 can achieve good intra-executor load balance, and setting y to be the number of nodes for computation-intensive operators can provide those operators with sufficient potential in scaling out upon workload bursts. However, in what follows, we evaluate the system performance with a large range of (y, z) under various workload, so as to understand why and how the two parameters impact system performance and how to choose proper parameters under extreme workloads. To make comprehensive observation, we use three representative workloads, namely the *default* workload, *data-intensive* workload and *highly dynamic* workload. Let s and ω denote the tuple size in bytes and key shuffles per minute, respectively. In the default workload, $(s, \omega) = (128B, 2)$. We get data-intensive workload and highly dynamic workload by increasing s to $8K$ and ω to 16, respectively. Thus, $(s, \omega) = (8K, 2)$ for data-intensive workload and $(s, \omega) = (128B, 16)$ for highly dynamic workload. Figure 16 shows the system throughput with various y and z under the three workloads. For comparison, we also show the throughput of the static and RC approaches in the figures.

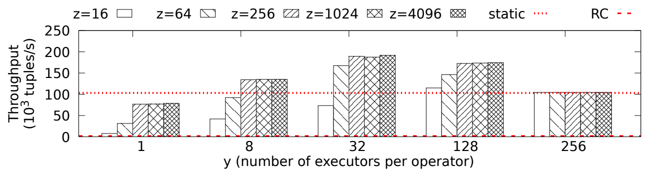
Number of shards: From Figure 16, we observer that as z increases, the throughput generally increases though the marginal increase is diminishing. This shows when using too few shards, e.g., $z \leq 64$, poor quality of intra-executor load balancing prevents elastic executors from efficiently utilizing multiple cores; however, too fine-grained sharding, e.g., $z \geq 1024$, does not further improve throughput as intra-executor load balancing is already effective. Base on those



(a) default workload (s = 128 B, ω = 2)



(b) data-intensive workload ($\underline{s = 8\text{ KB}}$, ω = 2)



(c) highly dynamic workload (s = 128 B, $\underline{\omega = 16}$)

Figure 16: The impact of number of executors (y) and number of shards (z) on the throughput of Elasticutor.

observations, we verify that 256 to 1024 shards per executor achieves good performance.

Number of executors: As shown in Figure 16(a), for a sufficiently large z, Elasticutor achieves promising performance except for y = 256. When y = 256, i.e., the number of CPU cores in the cluster, each elastic executor can only be allocated one CPU core. As such, executors lose elasticity and Elasticutor is downgraded to the static approach. By comparing Figure 16(a) with Figure 16(b), we can see that as tuple size increases to $8K$, the performance of the static and the RC does not change much, while that of Elasticutor under y = 1 drops severely. Compared with the default workload, the cost of remote data transfer in running a remote task in the data-intensity workload is 64 times higher. This limits the scalability of a single executor and thus results in poor performance for small y where a single executor needs to scale to many remote CPU cores. By comparing Figure 16(a) to Figure 16(c), we observe that as the shuffle frequency increases from 2 to 16, although the throughput decreases in general, the reduction is much greater when y is small, i.e., 1 or 8. Under a dynamic workload with frequent shuffles, e.g., ω = 16, more shards need to be reassigned for load balancing, incurring high migration cost. In contrast, when y is sufficiently large, most executors can scale using local CPU cores and thus avoid state migration due to intra-processing state sharing mechanism; and therefore, the throughput does not decrease much. In conclusion, setting one or two executors per node is robust to various workloads.

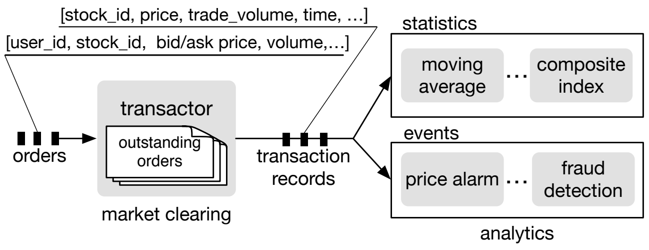


Figure 17: The topology of the SSE application.

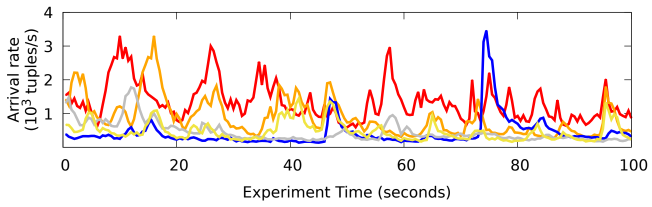
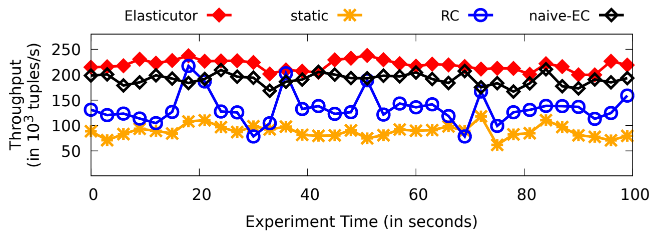


Figure 18: Arrival rates of 5 most popular stocks.

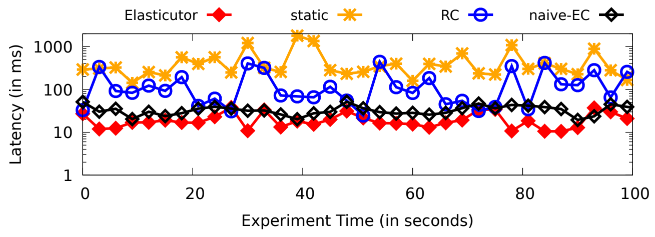
5.4 Evaluation of Realtime Application

To evaluate the performance of Elasticutor for practical applications, we use a dataset of anonymized orders for stocks traded in the Shanghai Stock Exchange (SSE), collected over three months with around 8 million records per trading hour. The application performs the market clearing mechanism of the stock exchange and provides real-time analytics. The topology of the application is shown in Figure 17. The input stream consists of limit orders from buyers and sellers, who specify their bid and ask prices for a particular volume of a particular stock. An order tuple is 96 bytes in size. Upon the arrival of a new order, a *transactor* operator executes it against the outstanding orders and determines the quantities traded and the cash transfers made. Once such a transaction is made, a 160-byte transaction record, including the time, number of shares and price of the transaction and IDs of the seller, buyer and stock, is sent to the downstream operators, including 6 operators for statistics and 5 operators for event processing. The analytics operators generate statistics, such as the moving averages and the composite index, and trigger user-defined events, such as alarms when the transaction price of a particular stock exceeds a predefined threshold. The state for each statistics operator is around 200MB to 400 MB in size, while the event processing operators have relatively small state below 10MB. As transactions and analytics concern individual stocks, we partition the space of stock IDs for parallel processing. Due to the unpredictable nature of stock trading, both the arrival rates and distribution of the orders of stocks fluctuate greatly over time, resulting in a highly dynamic workload. To illustrate the workload dynamics, Figure 18 shows the instantaneous arrival rate of 5 most popular stocks.

Besides the static, RC and Elasticutor, we test a naive executor-centric (naive-EC) implementation, which is the same as Elasticutor except that optimizations for migration cost and computation locality are disabled in the scheduler. Figure 19 plots the instantaneous throughput and the 99th percentile processing latency under the four approaches running on 32 nodes. We observe that both naive-EC and Elasticutor outperform the static and RC approaches, approximately doubling the throughput and reducing the latency by 1-2 orders of magnitude. Although the performance gaps between the naive-EC and Elasticutor are recognizable, they are small compared to those between the executor-centric approaches and the other two approaches. This observation indicates that despite the considerable performance improvement enabled by the optimizations in the dynamic scheduler, the better performance of Elasticutor is mainly due to the advantageous executor-centric paradigm employed.



(a) Throughput



(b) The 99th percentile processing latency

Figure 19: Performance comparison on real workload.

To further show the reason behind the performance gap between the naive-EC and Elasticutor, we show their state migration rate and the remote data transfer rate in Table 2. The former rate is the aggregated size of state the whole system migrates across network in a unit of time. The latter rate is the aggregated amount of data transfered in a unit of time between all the elastic executors and their remote tasks. We observe that the rates of state migration and remote data transfer under naive-EC are 5x and 10x higher than those under Elasticutor, respectively. With less state migration, it will be more efficient for the elastic executors to transition to a new resource allocation plan, thus achieving higher performance. Similarly, with less remote data transfer, more network bandwidth can be used by inter-operator data transfer, further improving the performance.

Table 2: Comparing naive-EC with Elasticutor.

Metrics	naive-EC	Elasticutor
State migration rate (MB/s)	13.9	2.4
Remote data transfer rate (MB/s)	235.3	21.6

Table 3: Scalability of Elasticutor.

number of nodes in the cluster	8	16	32
throughput (10^3 tuples/s)	66.6	121.3	218.6
scheduling time (ms)	4.1	5.2	5.7

Finally, we evaluate the scalability of Elasticutor under the SSE workload. We vary the size of the computing cluster, i.e., the number of nodes, and measure Elasticutor's throughput and scheduling cost, i.e., the average time needed for the dynamic scheduler to calculate a new CPU-to-executor assignment. Keeping a low scheduling cost is important for the system to be adaptive to a dynamic workload. Table 3 shows the throughput and scheduling cost as the scale increases. We observe that the throughput grows nearly linearly as the cluster grows; and the scheduling cost is around several milliseconds and grows slightly with the number of nodes.

6 RELATED WORK

Stream Processing Systems. Early stream processing systems, such as Aurora [7], Borealis [6], TelegraphCQ [17] and STREAM [10] were designed to process massive data updates by exploiting distributed but static computational resources. With cloud computing technologies, a new generation of stream systems emerged, with emphasis on parallel data processing, availability and fault tolerance, to fully exploit flexible resource management schemes on cloud-based platforms. Spark Streaming [50], Storm [43], Samza [35], Heron [31], Flink [12] and Waterwheel [46] are the most popular open-source systems providing distributed stream processing and analytics. Big industrial players are also developing in-house distributed stream systems such as Muppet [32], MillWheel [8], Trill [16], Dataflow [9] and StreamScope [33].

Elasticity. A large body of work explores the possibility of achieving elasticity. Castro et al. [15] combine the resource re-scaling operation with fault tolerance functionality in distributed stream systems, such that the intermediate states bound with the processing logic are written to persistent storage before migrating to new computation nodes. Wang et al. [47] propose elastic pipelining to enable dynamic, workload-aware run-time reconfiguration for distributed SQL query. An adaptive partitioning operator is proposed in Flux [39] to enable partition movement among nodes for load balance. However, as their workload migration is on a per-partition basis, this method faces difficulties when a single partition exceeds the processing capability of any node in the cluster. ChronoStream [48] partitions computation states into a collection of fine-grained slice units and dynamically distributes them across nodes to support elasticity. Gedik et al. [24] propose mechanism to scale stateful operators without violating state consistency. Chi [34] is a control panel with capability of monitoring and dynamic re-configuration. However, those methods achieve elasticity following the resource-centric paradigm which incurs expensive synchronization and prevents rapid elasticity. They are applicable in the scenarios where elasticity functionalities are used at a coarse time granularity, i.e., every 5 minutes; the achieved elasticity is too slow to be applicable in the application with highly dynamic workloads. Elasticutor avoids the problem by employing a new executor-centric approach. This approach greatly reduces the synchronization overhead in performing workload rebalance and therefore enables workload redistribution within several milliseconds.

Workload Distribution. Generic workload distribution for distributed stream systems is a challenging problem, due to the high skewness and huge variance in the incoming data stream over time. Shah [39] et al. designed dynamic workload redistribution mechanisms for individual operations in a traditional stream processing framework, e.g., Borealis [6]. [24] and [20] studied the mixed routing strategies to group the workload by its keys to dynamically balance the load in terms of CPU, memory and bandwidth resource. TimeStream [38] adopts a graph restructuring strategy, by directly replacing the original processing topology with a completely new one. It is, however, challenging for the system to monitor and optimize the topology in a huge search space of all applicable graphical structures. Cardellini et al. [13] studied stateful task migration on top of Storm. Ding et al. [18] discussed the long-term optimization on making task migration plans based on Markov Decision Processes (MDPs) to improve the resource utilization of distributed stream engines. However, Elasticutor not only achieves load balancing in workload distribution, but also takes migration cost minimization and computation locality into consideration.

7 CONCLUSION

We have designed and implemented Elasticutor, which enables rapid elasticity for stream processing systems. Elasticutor follows a new executor-centric approach that statically binds executors to operators, but allows executors to scale independently. This approach decouples the scaling of operators from the global synchronization needed for stateful processing. The Elasticutor framework has two building blocks: elastic executors, which perform dynamic load balancing, and a scheduler that optimizes the use of computational resources. Experiments show that compared with a traditional resource-centric approach to providing elasticity, Elasticutor doubles the throughput and achieves an average latency orders of magnitude lower.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their constructive and helpful comments. This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme. Richard Ma is supported by the Ministry of Education of Singapore AcRF under Grant R-252-000-A67-114. Li Wang and Tom Fu are partially supported by Natural Science Foundation of China (61702109, 61702113), China Postdoctoral Science Foundation (2017M612615, 2017M612613).

REFERENCES

[1] 2008. http://netty.io.
[2] 2016. https://github.com/apache/flink/tree/master/flink-yarn.
[3] 2016. https://github.com/yahoo/storm-yarn.
[4] 2018. https://github.com/ADSC-Cloud/Elasticutor.
[5] 2019. https://en.wikipedia.org/wiki/Dataflow programming.
[6] Daniel J Abadi et al. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*. 277–289.
[7] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 120–139.
[8] 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. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.
[9] 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. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.
[10] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. 2003. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.* 26, 1 (2003), 19–26.
[11] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.
[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[13] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. 2016. Elastic stateful stream processing in storm. In *Proceedings of International Conference on High Performance Computing & Simulation (HPCS)*. 583–590.
[14] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator scheduling in a data stream manager. In *VLDB*. 838–849.
[15] 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 *SIGMOD*. 725–736.
[16] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.
[17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. 2003. TelegraphCQ: continuous dataflow processing. In *SIGMOD*. 668–668.
[18] Jianbing Ding, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, Yin Yang, Zhenjie Zhang, and Hongyang Chao. 2015. Optimal Operator State Migration for Elastic Data Stream Processing. *CoRR* abs/1501.03619 (2015).
[19] Gyorgy Dosa. 2007. The tight bound of first fit decreasing bin-packing algorithm is FFD (I)? 11/9OPT (I)+ 6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 1–11.
[20] Junhua Fang, Rong Zhang, Tom ZJ Fu, Zhenjie Zhang, Aoying Zhou, and Xiaofang Zhou. 2018. Distributed Stream Rebalance for Stateful Operator Under Workload Variance. *IEEE Transactions on Parallel and Distributed Systems* 29, 10 (2018), 2223–2240.
[21] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.
[22] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3338–3352.
[23] Michael R Gary and David S Johnson. 1979. Computers and Intractability: A Guide to the Theory of NP-completeness.
[24] Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. *VLDBJ* 23, 4 (2014), 517–539.
[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.
[26] Navendu Jain, Ishai Menache, Joseph Seffi Naor, and Jonathan Yaniv. 2015. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. *ACM Transactions on Parallel Computing* 2, 1 (2015), 3.
[27] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*. 117–134.
[28] 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 *OSDI*. 783–798.
[29] Richard Earl Korf. 2009. Multi-way number partitioning. In *Twenty-First International Joint Conference on Artificial Intelligence*.
[30] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *SIGMOD*. ACM, 239–250.
[31] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *SIGMOD*. 239–250.
[32] Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. 2012. Muppet: MapReduce-style processing of fast data. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1814–1825.
[33] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed

Processing of Big Data Streams. In *NSDI*. 439–453.

[34] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.

[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. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[36] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th annual middleware conference*. ACM, 149–161.

[37] David MW Powers. 1998. Applications and explanations of Zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 151–160.

[38] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: reliable stream computation in the cloud. In *EuroSys*. 1–14.

[39] Mehul A Shah, Joseph M Hellerstein, Sirish Chandrasekaran, and Michael J Franklin. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*. 25–36.

[40] Julian James Stephen, Savvas Savvides, Vinaitheerthan Sundaram, Masoud Saeida Ardekani, and Patrick Eugster. 2016. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *SoCC*. ACM, 348–360.

[41] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[42] Henk C Tijms. 1986. *Stochastic modelling and analysis: a computational approach*. John Wiley & Sons, Inc.

[43] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *SIGMOD*. ACM, 147–156.

[44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 5.

[45] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*. ACM, 374–389.

[46] Li Wang, Ruichu Cai, Tom ZJ Fu, Jiong He, Zijie Lu, Marianne Winslett, and Zhenjie Zhang. 2018. Waterwheel: Realtime Indexing and Temporal Range Query Processing over Massive Data Streams. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 269–280.

[47] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. 2016. Elastic pipelining in an in-memory database cluster. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1279–1294.

[48] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *ICDE*. 723–734.

[49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2–2.

[50] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. 423–438.

[51] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *SOSP*. ACM, 614–630.

A EXTENSIVE EXPERIMENTAL RESULTS

In Section 5.4, we have conducted experiments with the realtime application, e.g., the Shanghai Stock Exchanged (SSE application), to compare the throughput and 99 percentile processing latency under the four approaches running on 32 nodes, as plotted in Figure 19. In Figure 20, we plot the average processing latency of the four approaches running on 32 nodes. We observe that the Elasticutor outperforms the other three approaches in terms of the average processing latency, which further verifies the advantageous of the executor-centric paradigm we proposed and employed.

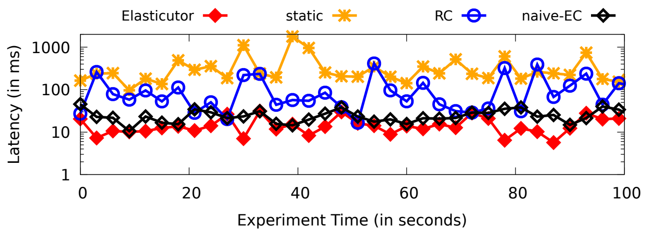


Figure 20: Performance comparison on real workload.

B DISCUSSION ON THE FUTURE WORK

Although our scheduling algorithm improves computation locality effectively and prefers to allocate tasks local to the main process of the executor, it is possible that in some extreme workloads, e.g., highly skewed key distribution, some executors may run excessive tasks with most tasks being remote, thus introducing extensive remote data transfer. To tackle this problem, we can detect the overloaded executors and split them into more executors with operator-level key repartitioning at a coarse time granularity, e.g., every 10 minutes. This is also useful when the workloads has increased significantly and the system needs to gracefully scale out to many new nodes, e.g., from initial 10 nodes to 100 nodes. Similarly, when the total workload decreases substantially, it is desirable to merge some idle executors so that some nodes can be freed up. In the future, we plan to develop a hybrid framework that uses elastic executors for rapid elasticity and infrequent operator-level key space repartitioning for long-term optimizations, such as offloading congested executors or scaling out/in the entire system.