StreamSwitch: Fulfilling Latency Service-Layer Agreement for Stateful Streaming

ABSTRACT

Distributed stream systems provide low latency by processing data as it arrives. However, as dynamic stream workload exhibits temporal bursts and spacial skewness in data, existing systems do not provide latency guarantee, a critical requirement of real-time analytics, especially for stateful operators. We present StreamSwitch, a control plane for stream systems to bound operator latency while optimizing resource usage. Based on a novel *stream switch* abstraction that unifies dynamic scaling and load balancing into a holistic control framework, our design incorporates reactive and predictive metrics to deduce the healthiness of executors and prescribes practically optimal scaling and load balancing decisions in time. We implement a prototype of StreamSwitch and integrate it with Apache Flink and Samza. Experimental evaluations on real-world applications and benchmarks show that StreamSwitch provides cost-effective solutions for bounding latency and outperforms the state-of-the-art alternative solutions.

1 INTRODUCTION

With the ubiquitous sensors and mobile devices continuously generating data, analytics over unbounded data streams has become very common. Because a large part of this big data is most valuable if it can be analyzed quickly when generated, distributed stream systems [1, 2, 7, 27, 35, 41, 50] are often preferred by time-critical analytics over batch-based systems. Although analyzing data directly as it is produced or received, existing stream systems do not provide any form of latency guarantee, a crucial requirement for real-time analytics. For example, algorithmic trading strategies need to be generated within seconds of receiving real-time market data [38]; otherwise, a profitable trade could easily turn to be a losing one.

Stream analytics often consists of a pipeline of fine-grained *stream operators*, i.e., the basic logical units that perform computation on tuples from input streams to generate output streams. To summarize the data seen so far, some operators also need to maintain *states* to encapsulate the full status of computation, such as the counter operator in the word-count application. To enable parallel processing, data stream is often defined over a *key space*, where the state and computation for each *substream*, i.e., the subset of data associated with each key, can be maintained and executed independently. Correspondingly, a stream operator can be instantiated by multiple physical *executors*, e.g., virtual machines in a distributed cluster, to process the substreams. However, as in-order execution is required on a per-key basis for maintaining the consistency of keyed-state, any substream cannot be processed by more than one executor at any time. Because per-tuple latency guarantee is extremely difficult and expensive to achieve if not impossible, while average latency is not well defined for unbounded data, we consider a generalized average tuple latency defined over a sliding time-window, where the window length provides the flexibility for users to specify various requirement stringencies. Since bounding latency is not without costs, another main concern of users, our solution also takes the resource consumption into consideration while fulfilling our primary goal for latency guarantee. *In summary, our goal is to bound the average latency of data tuples over a sliding time-window for general stream operators, including the ones that take multiple input streams such as Join, while optimizing the computation resources used.*

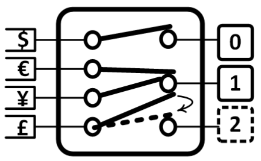

Figure 1: The stream switch abstraction.

The challenge of performance guarantee originates from the uncontrollable and unpredictable stream workload. From a temporal perspective, an operator's workload may surge significantly in seconds, requiring operators to scale out fast. For example, the number of requests made to an online ticket seller may suddenly increase when a popular game or show becomes available. Nevertheless, the scaling features of existing systems[7, 41] are very primitive if any, and cannot support runtime per-operator scaling well. From a spatial perspective, the dynamic workload of substreams over the executors can be highly skewed, resulting in overload and excessive latency in some executors. For example, in a stock market, compare to an average stock, a popular stock may have hundreds times more trading every second in a period of time. Under such scenarios, substreams need to change their designated executors, involving non-trivial synchronization and state migration. To the best of our knowledge, no system solution has provided latency SLA for stateful operators even under heterogeneous resources.

To define the solution space, we propose the *stream switch* abstraction. Figure 1 shows that each substream, e.g., input orders denominated in certain currency, connects to an executor, and a switching solution defines the *switching decision*, i.e., the substream-to-executor mapping for all substreams, at all times. Since a solution can switch the binding of a substream possibly to a non-existing executor, e.g., for the substream with key *£* in the example, the stream switch abstraction includes both scaling and load balancing decisions that need to be performed to handle dynamic workload. Furthermore, the solution space defined by this abstraction is general, because resolving any performance issue boils down to two related static decisions: 1) how many executors to use and 2) how to assign the substreams to the executors; and a switching solution specifies both decisions at all times. Therefore, to guarantee a latency SLA, StreamSwitch continuously monitors the operator's latency performance and makes appropriate *switching decision* if necessary.

Inspired by clinical procedures, the design of StreamSwitch follows an *Examine-Diagnose-Treat* framework shown in Figure 2 under which *switching decisions* are derived and deployed. The

Examine module collects substream metrics to estimate the average latencies of executors. Even in a heterogeneous resources environment, it can accurately estimate the latencies. Based on the metrics and estimated latencies, the Diagnose module prescribes cost-effective switching decisions, while considering the state migration cost and the heterogeneous resources environment. The decisions are further executed by the Treat module via operator-level scaling and load balancing in the underlying stream systems.

We implemented StreamSwitch as an engine-agnostic control plane for the systems that follow the dataflow [2] model, and demonstrate its generality by integrating it with two representative stream engines: Flink[7] and Samza [35]. In particular, native system mechanisms of Flink and Samza are leveraged to realize operator-level scaling, load balancing and fault tolerance. StreamSwitch provides latency guarantee for stateful stream operators, where a service-level agreement (SLA) is specified by a latency bound over a sliding window and the service is delivered in a serverless paradigm [8], i.e., the scaling, load balancing, and maintenance operations are hidden from the application developers. Due to the unpredictability of workload, we aim for a cost-effective solution that achieves close to 100% SLA rather than perfect guarantee. Notice that the Function as a Service offerings of serverless computing are incapable of guaranteeing the performance for stream jobs, because they are only suitable for short-run and stateless applications with simple workload of independent tasks [19], while stream jobs are long-run and often consisting of complex inter-dependent and stateful tasks.

We evaluate StreamSwitch via extensive experiments using a real financial application and multiple stream benchmarks. We show that 1) StreamSwitch is able to trade off resource frugality against success rate via a tunable safety margin ϵ and achieve close to 100% SLA for bursty workload in the order of seconds and the granularity of substreams; 2) by responding to workload dynamics with holistic decisions of switching, i.e., scaling or load balancing, it outperforms state-of-the-art alternatives and the spectrum of static solutions; 3) it can be applied to multi-operator scenarios for bounding the end-to-end latency of a critical path of the pipeline; and 4) its performance is fundamentally limited by the *switching overhead*, measured by the percentage of executor runtime spent on re-configuring new switching decisions.

StreamSwitch is the first system solution that provides latency SLA for stateful operators even under heterogeneous resources. Its unique and principled design is derived from a novel stream switch abstraction that unifies scaling and load balancing into a holistic control framework. Its generality is shown by integrations with Samza and Flink and evaluations on various applications and benchmarks. We plan to make StreamSwitch open source, and it is available at https://github.com/anonymousupload/StreamSwitch.

2 MOTIVATION AND OVERVIEW

In a typical stream engine, an application pipeline is expressed as a directed graph, where each vertex represents an operator associated with user-defined computation logic and each edge represents the data stream that specifies the input-output relationship between the operators. With respect to a logical operator, an input data tuple's latency is defined as the time between the tuple's arrival

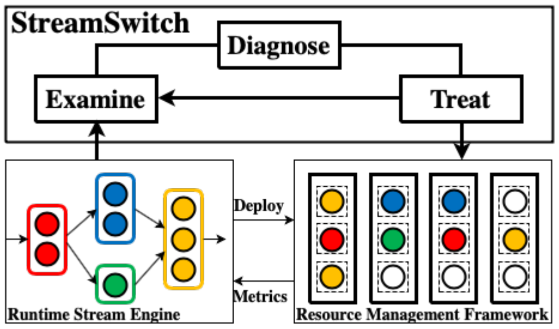

Figure 2: The Examine-Diagnose-Treat (EDT) framework.

to and its completion by the executor that performs the computation. Typically, each operator can be instantiated by multiple executors deployed as VMs or containers in a machine cluster. Instead of managing VMs or containers directly , many stream engines [7, 27, 35, 41] support deployments over resource management frameworks such as YARN [44] and Kubernetes [5]. The design of StreamSwitch is based on the premise that a stream engine leverages such a framework, through which executors can be added/removed in a runtime.

2.1 Motivation and Objective

To optimize the performance of stream systems, prior work primarily targeted two performance metrics: throughput [13, 15, 17, 24, 48] and latency [11, 14, 21, 29]. When a stream application's latency is bounded, its throughput will reach the maximum value, which is equal to the input rate. Therefore, the goal of latency is more desirable than that of throughput. Although plenty of latency-oriented work tried to reduce latency via efficient state migration [21, 31] and load balancing [11, 46], we found that only two works [14, 29] specifically targeted latency guarantee, a matter of life and death for applications such as self-driving and healthcare [45].

Unlike both works which aimed at end-to-end latency, we focus on per-operator latency for two reasons. First, end-to-end latency is not well-defined for application pipelines that consist of partially overlapping paths towards multiple sinks. For a path with well-defined end-to-end latency, we will show that our fine-grained per-operator mechanisms could be used to bound its end-to-end latency. Second, end-to-end latency has components of network latency induced by data transmission between distributed executors. Although bandwidth allocation [4] can be used to control network latency, any solution that relies purely on scaling and scheduling on computation resources, including both [14, 29] and ours, is unable to guarantee network latency. Therefore, we consider the network latency out of the scope of this paper and do not focus on end-to-end latency. Although we focus on per-operator latency, we have also demonstrated that our design can be extremely effective in guaranteeing end-to-end latency if network latency is not the bottleneck in Section 6 .

We would like to emphasize that for guaranteeing the per-operator component, both [14, 29] assumed stateless operators and homogeneous resources for executors. In [29], the overall latency is minimized by always routing an input tuple to the least loaded executor when the tuple comes. However, this method is not accessible for stateful operators. For example, in a word-counter operator, same words cannot be simultaneously processed by the different executor; otherwise, the counting result will be incorrect. In a heterogeneous environment, executors may have different processing capacities, and the capacities may change sometimes. Although [14] claims that it supports heterogeneous executors, it only assumes homogeneous executors and cannot capture the correct capacities of heterogeneous executors. Compare to them , StreamSwitch works for stateful operators with both homogeneous and heterogeneous resources.

Because per-tuple latency guarantee may not be required by applications and can be expensive to achieve if not impossible, we allow users to specify their requirements in terms of an average latency of tuples. From a temporal perspective, as stream jobs are often long-run with unbounded inputs, we need to define a sensible average over a finite time horizon. We aim to bound the average latency of tuples completed in a sliding time window $(t - \mathrm{T}, t]$ of length T at any time t. Not only does the length T generalize the per-tuple requirement, as per-tuple guarantee is specified as $\mathrm{T} \to 0$, but it also provides the flexibility to differentiate the stringencies of requirements. From a spatial perspective, the average latency can be scoped on a per-operator, per-executor or per-substream basis. Despite being more stringent, the fine-grained substream scope could be desirable for applications that need to guarantee latency with respect to any substream, we aim to bound the average latency of tuples on a per-substream basis. From a user's point of view, a latency SLA can be easily specified as a pair of (L, T), where L specifies the maximum acceptable latency and T provides an upper-bound for the "downtime" when latency exceeds L, because tuple latencies cannot all exceed L if their average within T is bounded by L. In our implementation, latency SLAs can be specified in the configuration file of stream application and become known to StreamSwitch when the configurations are submitted to the stream engines; however, this does not prevent StreamSwitch from adapting to possibly new latency SLAs on the fly.

To execute *switching decisions*, scaling and load-balancing functionalities in the distributed stream systems are required. However, these functionalities are either not or poorly supported by existing stream systems [7, 27, 35, 41, 50]. Therefore, one needs to either develop these functionalities based on existing functionalities or build their own engines from nothing. After evaluating different systems, we choose to implemented our functionalities Flink [7] and Samza [35], since they are representative and provide required mechanisms such as metrics and failure-recovery mechanisms.

To summarize, our objective is to bound the average latency of data tuples over a sliding time-window for stateful stream operators, while optimizing the computation resources used.

2.2 Overview of StreamSwitch

In this subsection, we provide a high-level overview of the main components in StreamSwitch as shown in Figure 2. To bound tuples' latencies while optimizing resource consumption, the key is to constantly generate and realize *switching decisions* that specify necessary changes in 1) the number of executors to use, and/or 2) the substreams assigned to them. To identify and execute such switching decisions, StreamSwitch periodically goes through an **Examine-Diagnose-Treat** (**EDT**) procedure.

Table 1: Notations

$\mathcal{K}, \mathcal{K}_i$	Input substreams set of operator and executor i.
$\mathcal{N}$	Executors set of the operator.
s, d	Source and destination executors of a strategy.
K	The set of substreams moved by a strategy.
x_{NO}	Passive strategy that does nothing.
$x^*_{SI}, x^*_{LB}, x^*_{SO}$	Optimal strategies of type SI, LB and SO.
l	Alert threshold of instantaneous latency l_i.
l_i	Instantaneous latency of executor i.
$L_i(x)$	Projected latency of executor i under strategy x.
$\lambda_i(x)$	Arrival rate of executor i under strategy x.
λ^k	Arrival rate of substream k.
μ_i	Processing rate of executor i.

The main goal of the **Examine** module is to estimate the per-executor latency. To determine the number of executors needed, one needs to know whether the existing executors are about to violate the latency SLA; however, due to unpredictable workload, any predictive model can be unreliable. Therefore, the **Examine** module estimates the short-term reactive measure on the average latency over the current sliding window. To further determine the assignment of substreams to executors, one needs to predict whether an executor will fulfill the latency SLA under potential substream-to-executor mappings. Based on a queueing model, the **Examine** module projects the long-term proactive measure on the steady-state latency under any substream-to-executor mapping. To calculate the latency estimates, the **Examine** module interfaces with stream engines to sample substream metrics.

The **Diagnose** module is responsible for generating new switching decisions. Based on the latency SLA (L, T) and the estimated latency provided by the Examine module, the **Diagnose** module decides the *healthiness* of executors. An executor's healthiness is defined to be good (or severe) if the executor is (or is not) able to fulfill the latency SLA. If a severe executor exists, the **Diagnose** module will search for a cost-effective load balancing or scale-out to cure it; if all the running executors are in good health, the **Diagnose** module will search for the best possible scale-in for cost saving.

The **Treat** module finally executes the switching decisions made by the **Diagnose** module in the underlying stream engines.It interfaces with the resource management framework to acquire and release executor instances, and dynamically remaps the substreams to executors. To maintain the consistency of substream states, the **Treat** module synchronizes the executors and migrates substreams' states when substreams need to be migrated between executors.

Table 1 summarizes frequently used notations throughout paper.

3 THE DIAGNOSE MODULE

As the core of StreamSwitch, the Diagnose module makes switching decisions for an operator. We denote the set of input substreams and executors of the operator by $\mathcal{K}$ and $\mathcal{N}$. A *switching decision* is defined as a substream-to-executor mapping $\mathcal{F} : \mathcal{K} \to \mathcal{N}$, under which $\mathcal{K}_i = \{k \in \mathcal{K} : \mathcal{F}(k) = i\}$ defines the assigned set of substreams to an executor $i \in \mathcal{N}$. At any decision epoch, the Diagnose module can move a set K of substreams to form a new *switching decision* $\mathcal{F}'$. So we can equivalently define any new decision $\mathcal{F}'$ via a *strategy*, which specifies the set K of substream with their destination executors.

3.1 Strategy Space and Treatment Types

In our design, at each decision epoch, the Diagnose module only writes a *unit strategy* that moves a set of substreams K from a single source executor s to a single destination executor d for several reasons. First, a strategy involving more executors induces higher execution overhead and takes the system a longer time to execute it, since moving substreams involves expensive operations of synchronization and state migration that degrade the performance of the source and destination executors, especially for stateful operators. For example, in our stock operator, synchronization and state migration usually take hundreds of milliseconds. In this case, more involved executors means more executors are stopped from processing tuples during this period of time. As workload may change within seconds, a complex strategy that involves many executors could become unsuitable before it is fully executed. Second, as any strategy can be expressed as a composite of multiple unit ones, without loss of generality, any switching decision can be achieved by executing a series of unit strategies. With its lower complexity of generation and lower cost of execution, the use of unit strategy supports a higher decision frequency which makes StreamSwitch more adaptive to dynamic stream workloads. A unit strategy can be denoted as a triple $x = (K, s, d)$.

The possible unit strategies can be classifed into 3 types of treatments: 1) Scale In (SI) treatment if $K = \mathcal{K}_s$; 2) Scale Out (SO) treatment if $\mathcal{K}_d = \emptyset$; 3) Load Balancing (LB) treatment if otherwise. All of the three types of treatments make trade-offs between performance and costs. The SI treatments are resource-saving strategies at a cost of latency, while the SO and LB treatments are latency-improving strategies at a cost of computation resource for the former and system overhead for the latter. As the capacity of a newly acquired executor is unknown and may not be higher than that of an existing one, the strategy $(K, \mathcal{K}_d) = (\mathcal{K}_s, \emptyset)$ that only replaces an executor is excluded from our strategy space by design. Finally, we denote the passive strategy that keeps the existing switching decision by x_{NO} when $K = \emptyset$.

3.2 Two-Level Diagnostic Logic

The Diagnose module uses a two-level logic to generate strategies of different treatments, sorted from the most resource-saving to the most latency-improving ones from left to right at the bottom of Figure 3. The top-level logic marked by $H(x_{NO})$ differentiates three scenarios where 1) a resource-saving strategy might be used to reduce resource consumption, 2) the passive No Operation (NO) strategy x_{NO} is used to keep the switching decision, or 3) a latency-improving strategy must be used to fulfill the latency SLA. Accordingly, a *healthiness* metric $H(x)$ is used to classify the operator's performance under any strategy x into three levels: *good* (G), *moderate* (M) and *severe* (S). In particular, the top-level logic uses the healthiness $H(x_{NO})$ of the operator projected under the current switching decision.

The bottom-level logic further nails down the type of treatment T and an optimal unit strategy x^*_T of that type. When a latency-improving strategy is needed, it will prescribe an optimal load balancing strategy x^*_{LB} if the healthiness $H(x^*_{LB})$ projected under x^*_{LB} can relieve the latency stress; otherwise, an optimal scale-out strategy x^*_{SO} will be prescribed. For example, when an operator

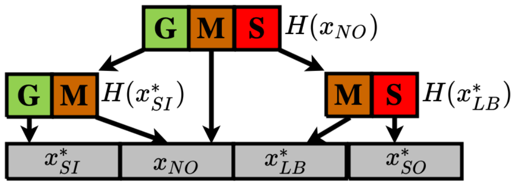

Figure 3: Two-level diagnostic logic.

almost or already violates the latency SLA, its healthiness metrics $H(x_{NO})$ is severe, so the Diagnose module follows the arrow on the right in Figure 3. Then it tries to find an optimal LB strategy x^*_{LB} that can reduce the operator's latency most. If the healthiness metric $H(x^*_{LB})$ under the strategy x^*_{LB} is improved from severe to moderate, it means load-balancing is enough to avoid the operator from violating the latency SLA, so the Diagnose module generates x^*_{LB} to the Treat module. Otherwise, the Diagnose module will find and generate an optimal SO strategy. When resource-saving is possible, it will prescribe an optimal scale-in strategy x^*_{SI} if the healthiness $H(x^*_{SI})$ projected under x^*_{SI} remains *good*; otherwise, the passive strategy x_{NO} will be prescribed. In both cases, we prefer a resource-efficient strategy if it does not sacrifice the healthiness with respect to latency.

As an operator is instantiated by executors, $H(x)$ is naturally a function of the healthiness $H_i(x)$ of each executor i. We define $H(x)$ to be 1) severe if there exists *any* executor i whose healthiness $H_i(x)$ is severe, and therefore, a latency-improving strategy is needed to alleviate the executor, 2) good if $H_i(x)$ is good for *all* executors, under which we might consider a resource-saving strategy, and 3) moderate if otherwise.

3.3 Healthiness Metrics & Optimal Strategies

To complete the discussions of the Diagnose module, two remaining questions are: 1) how to construct the healthiness metric $H_i(x)$ of executor i under a strategy x, and 2) how to define and find an optimal strategy x^*_T for a treatment type T?

To aim at latency guarantee, we construct $H_i(x)$ based on how likely it is for an executor to violate the latency constraint. At any decision epoch t, a direct metric is the average latency l_i of the tuples processed by executor i over the current sliding window $(t - \mathrm{T}, t]$. Although this reactive metric l_i can be used to diagnose how critical the current situation is for the executor, it does not yet capture the potential impact of a strategy x and is insufficient for predicting whether executor i will likely violate the latency constraint in the near future. Thus, we utilize a predictive metric $L_i(x)$ that projects the steady-state latency of executor i under any potential strategy x based on the current workload trends of the substreams.

Based on the metrics l_i and $L_i(x)$, provided by the Examine module and to be discussed in the next section, we define the healthiness $H_i(x)$ to be 1) severe if $(l_i, L_i(x)) > (l, \mathrm{L})$, 2) good if $(l_i, L_i(x)) \leq (l, \mathrm{L})$, and 3) moderate if otherwise, where l is a configurable parameter that defines an alert threshold. l usually needs to be smaller than L; otherwise, the healthiness $H_i(x)$ will only be severe when $l_i > l \geq \mathrm{L}$, which means the tuples' latencies in i already exceeded L, and the SLA was already violated. For L =1s, we will show that l works well in a wide range from 20ms to 500ms and is robust to the performance in Section 6.

The above definition relies on a reactive condition $l_i > l$, i.e., the achieved average latency is over an alert threshold, and a proactive condition $L_i(x) > \mathrm{L}$, i.e, the projected latency will be larger than the latency bound, as indications of potential constraint violation. It predicts that an executor is higly likely to violate the latency contraint, i.e., *severe*, if both conditions are satisfied, while very unlikely to violate the contraint, i.e., *good*, if neither condition is met.

If an executor is diagnosed with *severe* under the existing switching decision, i.e., x_{NO}, a latency-improving strategy will be triggered since the constraint is expected to be violated without making any changes; otherwise, such a strategy will not be triggered, e.g., by a *moderate* executor, because either the workload trends might alleviate the instantaneous latency l_i or the laterncy buffer might absorb the surge of workload before it goes down. However, the existence of a *moderate* executor will prevent the system from using any resource-saving strategy, because such an executor does have some pressure on latency and might need more resources in the near future.

Because the average latencies of the substreams associated with the same executor are close, as executors do not differentiate substreams, our primary objective of bounding per-substream latency boils down to bounding the worst latency on a per-executor basis. Thus, we define an optimal strategy x_T^* to be any unit strategy of treatment type T under which the maximum of the projected executor latency $L_i(x)$ is minimized. In the next subsection, the algorithms of finding optimal strategies are described.

3.4 Algorithms for Optimal Strategies

Algorithm 1: Find the optimal SI strategy x_{SI}^*

```
function FindOptimalSIStrategy($x_{NO}$)
    $L_{OPT} \leftarrow L_{NO}, K_{OPT} \leftarrow \emptyset, s_{OPT} = d_{OPT} =$ NULL ;
    for $s$ in $\mathcal{N}$ do
        for $d$ in $(\mathcal{N} - \{s\})$ do
            $L_{sd} \leftarrow \{L_i(x_{NO}) : i \in \mathcal{N} - \{s, d\}\} \cup \{CalcL(\lambda_s + \lambda_d, \mu_d)\}$ ;
            if $L_{sd} < L_{OPT}$ then
                $L_{OPT} \leftarrow L_{sd}$;
                $K_{OPT} \leftarrow \mathcal{K}_s$ ;
                $s_{OPT} = s$ ;
                $d_{OPT} = d$ ;
    return $K_{OPT}, s_{OPT}, d_{OPT}$;
```

Algorithm 1 shows the algorithm to find the optimal scale-in strategy x_{SI}^*. As the passive strategy x_{NO} does not change the switching decision, the vector $\boldsymbol{L}_{NO} = \{L_i(x_{NO}) : i \in \mathcal{N}\}$ projects the latency of executors under the current setting and serves as a baseline for finding an optimal strategy x_T^* that minimizes the worst projected latency $L_i(x)$ for each treatment type T. In algorithm 1, we iterate over all pairs of executors to be the source s and destination d at a complexity of $O(N^2)$, where N is the number of executors. As will be discussed in Section 4, the projected latency L_i of executor i is a function $L_i = (\lambda_i, \mu_i)$ of i's arrival rate λ_i and service rate μ_i. Since all substreams of s are moved to d, we project the latency of the destination by $L_d = L(\lambda_d(x_{NO}) + \lambda_s(x_{NO}), \mu_d)$ to form a latency vector of $n - 1$ executors with its components sorted in a descending order. An optimal strategy x_{SI}^* induces the sorted vector $\boldsymbol{L}_{OPT}$ that has the smallest lexicographic order.

Algorithm 2: Find the optimal LB strategy x_{LB}^*

```
function FindOptimalSubstreams(s, d)
    $\lambda'_s = \lambda_s, \lambda'_d = \lambda_d$ ;
    $L_{sd} = \max(CalcL(\lambda_s, \mu_s), CalcL(\lambda_d, \mu_d)), K \leftarrow \emptyset$ ;
    sort $\mathcal{K}_s$ by $l^k$ in ascending order ;
    for $k$ in sorted($\mathcal{K}_s$) do
        $\lambda'_s$ -= $\lambda^k$ ;
        $\lambda'_d$ += $\lambda^k$ ;
        $K \leftarrow K \cup k$ ;
        if $\max(CalcL(\lambda'_s, \mu_s), CalcL(\lambda'_d, \mu_d)) < L_{sd}$ then
            $L_{sd} = \max(CalcL(\lambda'_s, \mu_s), CalcL(\lambda'_d, \mu_d))$
        else
            break ;
    return $L_{sd}, K$ ;
function FindOptimalLBStrategy($x_{NO}$)
    $s_{OPT} = arg\,max_{i:l_i>l}(CalcL(\lambda_i, \mu_i))$ ;
    $L_{OPT} = \max_i(CalcL(\lambda_i, \mu_i))$ ;
    $K_{OPT} \leftarrow \emptyset$ , $d_{OPT}$ = NULL ;
    for $d$ in $\mathcal{N}$ do
        $L_{sd}, K \leftarrow$ FindOptimalSubstreams($s_{OPT}, d$) ;
        if $\max(\max_{i \neq s_{OPT}, d}(CalcL(\lambda_i, \mu_i)), L_{sd}) < L_{OPT}$ then
            $L_{OPT} = \max(\max_{i \neq s_{OPT}, d}(CalcL(\lambda_i, \mu_i)), L_{sd})$ ;
            $K_{OPT} \leftarrow K$ ;
            $d_{OPT} = d$ ;
    return $K_{OPT}, s_{OPT}, d_{OPT}$;
```

The function *FindOptimalLBStrategy* in Algorithm 2 show our algorithm to find the optimal LB strategy . When a latency-improving treatment, i.e., LB or SO, is triggered, we will take the *severe* executor with the maximum $L_i(x_{NO})$ as the source s from which some substreams will be migrated to reduce latency. For load balancing, we iterative through all other executors as the destination d with measured service rate μ_d. For scaling out, the only difference between it and the load balancing algorithm is that the newly acquired executor is chosen as the destination d. Since StreamSwitch does not know the service rate of the newly acquired executor d in a heterogeneous environment, an estimation of $\mu_d = \mu_s$ is used. Although the estimated service rate may have some error in a heterogeneous environment at the beginning, it will soon be corrected by the Examine module when the executor d starts.

In principle, it is desirable to move a subset of substreams $K \subseteq \mathcal{K}_s$ from the source such that the maximum of the source's projected latency $L_s = L(\lambda_s(x_{NO}) - \lambda_K, \mu_s)$ and the destination's projected latency$L_d = L(\lambda_d(x_{NO}) + \lambda_K, \mu_d)$ is minimized, a combinatorial problem with high complexity. The function *FindOptimalSubstreams* in Algorithm 2 illustrates how we find the moving substreams K for finding optimal LB and SO strategy. Although relocating substreams mitigates latency pressure, it exacerbates the latency of relocated substreams since processing will temporarily be paused for them during migration. Therefore, it is desirable to migrate the substreams that are experiencing lower latency in practice. To balance between theoretical optimality and practical impact, we 1) sort the set $\mathcal{K}_s$ of substreams by the average latency l^k, calculated by the per-time slot metric $l^k[m]$ over the sliding window, in an ascending order, and 2) move them one by one into the subset K until the difference $|L_s - L_d|$ is minimized. In total, it takes $O(N|\mathcal{K}_s|)$ and $O(|\mathcal{K}_s|)$ complexity to search for the optimal strategies x_{LB}^* and x_{SO}^*, respectively. Because $L_s - L_d$ is monotonic in K, a binary search can be used and the complexity can be optimized to $O(\log|\mathcal{K}_s|)$ and $O(N \log|\mathcal{K}_s|)$ respectively.

The algorithms of finding optimal strategies make no assumptions on homogeneous resources. Combining with the projected latency $L_i(x)$ and other metrics provided by the Examine module, the algorithm can function normally in a heterogeneous environment. More detail will be discussed in Section 4.

4 THE EXAMINE MODULE

The previous section describes how a strategy is formed at a decision epoch. Although the stream switch abstraction does not limit the frequency at which such decisions are made, StreamSwitch is designed to trigger the EDT framework at regular intervals with a configurable length δ. If it starts at a logical time 0, the Examine module will be triggered at the end of each time interval $((m-1)\delta, m\delta]$, referred to as time slot m, to retrieve system metrics and build states, based on which the latency metrics l_i and $L_i(x)$ are constructed.

4.1 Instantaneous Average Latency l_i

The instantaneous average latency l_i is the average latency of the tuples that are recently processed in executor i. However, a direct calculation of l_i based on counting individual tuple latencies is practically expensive, since it requires each executor to record the times of arrival and completion for all tuples it processes. The Examine module utilizes much sparser information at the per-substream granularity: the accumulated numbers of arrived and completed tuples sampled at each time epoch m, denoted by $a^k[m]$ and $c^k[m]$ for any substream $k \in \mathcal{K}$.

Because the per-executor metric l_i is defined over a sliding window, within which the substream-to-executor mapping $\mathcal{F}$ may change, we construct l_i using the average tuple latency within each time slot m, denoted by $l_i[m]$, during which the set $\mathcal{K}_i[m]$ of substreams assigned to executor i remains the same. At any epoch n, l_i is the weighted average of $l_i[m]$ for the time epochs m that fall within the sliding window, i.e., $m\delta \in (n\delta - \mathrm{T}, n\delta]$, weighted by the number of completed tuples $C_i[m]$ in each time slot m, calculated by $C_i[m] = \sum_{k\in\mathcal{K}_i[m]} c^k[m] - c^k[m-1]$.

For each time slot m, the average tuple latency $l_i[m]$ of executor i can be calculated as the average of per-substream average latency $l^k[m]$ over $k \in \mathcal{K}_i[m]$, weighted by the number of completed tuples $c^k[m] - c^k[m-1]$, because the index of completed tuples in time slot m ranges from $c^k[m-1]+1$ to $c^k[m]$.

Because in-order processing is required by applications, FIFO scheduling is used by stream systems on a per-stream basis; thus, the time slot m' at which a tuple arrived can be traced: for the rth tuple, m' satisfies $r \in (a^k[m'-1], a^k[m']]$. Accordingly, we approximate each tuple's latency by $(m-m')\delta$, i.e., the difference between its completion and arrival times in the granularity of time slots. Consequently, the approximation error of the estimated average latency l_i is no more than the length δ of a time slot, since any tuple's true latency is lower and upper bounded by $[(m-1)-m']\delta$ and $[m-(m'-1)]\delta$.

Property: *StreamSwitch will not trigger unnecessary latency-improving strategies, i.e., SO or LB.* This is because l_i accurately measures the incurred average latency and $l_i < l$ is a necessary condition for an executor i to be considered severe.

4.2 Projected Steady-State Latency $L_i(x)$

In principle, the average latency of an executor i depends on two fundamental quantities: *arrival rate* λ_i, i.e., the rate at which tuples arrive at the executor, and *service rate* μ_i, i.e., the maximum rate at which the executor can complete tuples. To a first approximation, we project the future average latency based on a generalized M/M/1 sojourn time formula $L_i = L(\lambda_i(x), \mu_i) = [(1-\epsilon)\mu_i - \lambda_i(x)]^{-1}$, where $\lambda_i(x)$ is the projected arrival rate under the strategy x and $\epsilon \in [0,1)$ is a tunable safety margin. We specify ϵ as a percentage used to control the upper-limit of arrival rate, because the latency cannot be bounded as the arrival rate approaches the service rate. Therefore, the system may trigger a latency-improving strategy when the arrival rate approaches or exceeds $(1-\epsilon)$ times the service rate. For example, when an executor's service rate and arrival rate are 2000 and 1799 tuples-per-second, and ϵ is 0.1, the projected latency of the executor is $\frac{1}{(1-0.1)\times 2000-1799} = 1$ seconds.

Although fine-grained metrics such as higher-order statistics of the inter-arrival and service times can be incorporated into a general G/G/1 queueing model to project latency, we expect that the characteristics of workload may change, and higher-order statistics are too sensitive to such changes. Without making predictions on how workloads will change, we make use of the M/M/1 steady-state metric as a neutral estimate on the near-future based on the instantaneous trends. To apply the M/M/1 formula, the projected arrival rate at any decision epoch n is calculated by $\lambda_i(x) = \sum_{k\in\mathcal{K}_i(x)} \left(a^k[n] - a^k[n-\mathrm{T}/\delta]\right)/\mathrm{T}$, which counts the number of arrived tuples within the recent sliding window $(n\delta-\mathrm{T}, n\delta]$ divided by the duration T.

Unlike the arrival rate $\lambda_i(x)$, the service rate μ_i purely depends on the computation capacity of executor i. Because executors may experience idle times during which no pending tuples need to be processed, the number of processed tuples $C_i[m]$ generally underestimates the service rate. Furthermore, normalizing $C_i[m]$ with the CPU utilization of the executor may still underestimate μ_i, since CPU utilization may include I/O blocking time that does not contribute to data processing. Following prior work [24], we estimate the instantaneous service rate of executor i at each time slot m by $C_i[m]/\rho_i[m]$, where $\rho_i[m]$ denotes the per-executor *useful time*[24], i.e., the time spent in deserialzation, processing and serialization activities. In a heterogeneous environment, this estimation can also distinguish different types of executors and capture their capacities dynamically.

Since the instantaneous service and arrival rates might fluctuate, we maintain an exponential weighted moving average (EWMA), for example, the average service rate is calculated as $\mu_i[m] = (1-\eta)\mu_i[m-1] + \eta(C_i[m]/\rho_i[m])$, where the decay factor η is set to be $1/8$, following how the TCP protocol smoothens its measurements of round trip time.

Property: *StreamSwitch will trigger a scale-in strategy, only if all the projected arrival rates λ_i of the executors are below $(1-\epsilon)$ times their service rates μ_i.* This prevents StreamSwitch from unnecessary scale-in operations that may lead to instability and oscillation in switching decisions.

4.3 Metrics Retrieval, Validity & Calibration

The Examine module maintains a data structure called `State` to store the metrics $a^k[m]$, $c^k[m]$ and $\rho_i[m]$, retrieved from the runtime system. We define a common interface `StreamSwitchMetricsRetriever` for stream systems to provide such metrics via their metrics library or customized method, and implement two retriever instances. For Samza, as Java Management Extensions (JMX) is enabled by default in Samza containers [39], the metric retriever runs as a JMX client and invokes remote procedure calls to retrieve metrics from the executor containers. In Flink, because JMX-based metrics are provided at the `TaskManager` that includes excessive details, to reduce overhead, we implement a customized retrieval process `MetricManager` that exports the metrics from individual executors, i.e., a stream task in Flink, into a Kafka stream for the metric retriever to consume. In both cases, the *useful time* $\rho_i[m]$ is obtained by directly measuring the serialization, deserialization and processing time.

Although the Examine module is triggered every δ amount of time to update `State`, as the metric retriever may fail to retrieve some metrics in a time slot, which makes `State` *invalid* or incomplete for calculating the latency metrics l_i and $L_i(x)$, the Diagnose module will not be triggered if `State` is invalid. To enable a valid `State` as much as possible, whenever a valid value of $a^k[m]$ or $c^k[m]$ is obtained in a time slot, we calibrate any previously missing entries using linearly interpolated values. Besides a valid `State`, the Diagnose module will be triggered only if no active treatment is on-going, because substreams are changing hands between executors during treatment, which changes the substream-to-executor mapping $\mathcal{F}$ and may make the calculation of l_i inconsistent. We will discuss how stream systems inform StreamSwitch about the completion of a treatment in the next section.

5 THE TREAT MODULE

After the Diagnose module generates a strategy, the Treat module realizes the corresponding treatment in the native stream systems. To interface any generic runtime system with any control plane, we defined a pair of Java interfaces `OperatorController` and `OperatorControllerListener`. The former API specifies the callback methods for stream engines to notify system events and is implemented by the control plane; the latter API defines the system operations exposed and invoked by the control plane and is implemented by the stream engines. In particular, `OperatorControllerListener` consists of methods `scale` and `remap` for performing scaling and load balancing operations, respectively, whose system-specific implementations for Samza and Flink will be discussed. We built an abstract class `StreamSwitch` to maintain the states of the stream switch abstraction and the life cycle of StreamSwitch, and implemented the specific EDT framework by a concrete subclass `LatencyGuarantor` that inherits `StreamSwitch`. To activate the control plane in a plug-and-play manner, native stream systems can instantiate StreamSwitch via supplying it with the methods `scale` and `remap` encapsulated in the `OperatorControllerListener` interface.

Although most stream systems including Samza and Flink implement fault tolerance mechanisms, e.g., checkpointing and stream barrier, to recover executors from failures, information of the executors such as their mapped substreams could be lost during failures, causing the breakdown of StreamSwitch. To address this problem, we implemented a callback method `onExecutorFailed` in the `OperatorController` interface to respond to the failure of executors. When a failure happens, besides recovering the failed executors, stream systems will notify StreamSwitch by invoking the callback method, and StreamSwitch will then publish the newest substream-to-executor mapping to the recovered executors. This ensures the correctness of state migration and the exactly-one semantics of processing, even if failures happen during a treatment.

Besides the acquisition and release of resources, conceptually, a treatment involves three stages for the source and destination: 1) stop processing and stop pulling tuples from upstream, 2) migrate states from source to destination and update the routing information, and 3) resume operations. To fulfill the exactly-once semantics of processing, the challenge of the first stage is to synchronize the executors so as to maintain the state consistency during and after migration. In the rest of this section, we will discuss our design and implementation of the operator-level scaling and load balancing mechanisms in Samza and Flink, i.e., the system-specific methods `scale` and `remap`, mainly focusing on how we leverage native system mechanisms to achieve executor synchronization.

5.1 Integration with Apache Samza

In Samza, operators can be deployed as single-stage *jobs* via two options. The *standalone* mode abstracts each executor as a `StreamProcessor` with a ZooKeeper-based `JobCoordinator` to coordinate the distributed entities among themselves. The YARN mode manages and initializes the distributed executors centrally, but it does not actively manage the executors at runtime.

To take the advantages of both options, we unified the two approaches by incorporating the `StreamProcessor` abstraction into the YARN mode. Our `YarnApplicationMaster` not only initializes the cluster-based deployment, but also instantiates StreamSwitch, invokes YARN APIs to acquire and release container resources during scaling events, and invokes the `onExecutorFailed` method to notify StreamSwitch about failures of executors. Furthermore, it deploys executors as instances of `StreamProcessor` and leverages *job coordinators* to synchronize them. In particular, we designed a leader-follower paradigm of the coordinators, where a `LeaderJobCoordinator` is run by the control plane and a `FollowerJobCoordinator` is associated with each executor. The `remap` method is implemented by the `LeaderJobCoordinator` publishing new substream-to-executor mapping, while each `FollowerJobCoordinator` judging whether a new set of substreams is mapped to its executor. Consequently, during treatments only the source and destination executors will be involved to update their states while other executors will keep running. Because Samza utilizes Kafka[26] as its inter-operator connector, the upstream and downstream operators only need to produce and consume Kafka streams, respectively, without managing the dynamic routing information.

5.2 Integration with Apache Flink

In Flink, we built a `SwitchCoordinator` to perform treatments within the `JobManager`. When `SwitchCoordinator` receives treatment strategies from StreamSwitch, it updates the *execution plan* of the Flink job and the routing information for the involved executors. `SwitchCoordinator` is also responsible for notifying StreamSwitch the failure of executors via invoking the callback method `onExecutorFailed`. Unlike Samza, Flink's executors, i.e., instances of `StreamTask`, maintain message queues to communicate with other executors of upstream and downstream operators. To maintain consistent routing between executors during treatments, we also built a `TaskConnectionManager` to update routing information in each executor. Although Flink supported runtime reconfiguration, it required a full restart of the pipeline and induced significant latency; and thus, was retracted from v1.9 onwards.

By injecting *stream barriers* into data streams, Flink generates consistent checkpoints or snapshots of the distributed system for failure recovery. We leveraged this fault tolerance mechanism [6] to synchronize executors using a new type of checkpoint `Switchpoint` and migrate substreams through a four-stage process. Notice that when performing the scaling and load balancing operations, similar to that in Samza, our implementation of the `scale` and `remap` methods only involves the source and destination executors for synchronization and state migration, while other executors keep processing data tuples as usual.

6 EVALUATION

We implemented StreamSwitch on Samza and Flink, and conducted experiments to answer the following questions.

(1) What is the overall performance of StreamSwitch?
(2) How does it handle dynamic workload with bursts?
(3) What are the impacts of its system parameters?
(4) Does it accommodate heterogeneous resources well?
(5) What are the limitations of StreamSwitch?
(6) How about end-to-end latency for multiple operators?

Metrics We measure the benefit of a solution by the average *success rate* of substreams, defined as the percentage of time windows under which latency constraint is fulfilled. We record the ground truth latency of data tuples to calculate the success rates in all our experiments. We measure the cost of a solution by the average number of executors used, because we primarily run executors over *homogeneous* CPU resources. For self-adapting solutions that leverage resource elasticity, we compare StreamSwitch with DRS [14] that specifically targets cost minimization with latency constraints. Since neither load balancing nor substream-to-executor assignment is specified by DRS (as it focused on stateless operators), we implemented 1) DRS with the optimal substream migration of StreamSwitch to realize the scaling decisions made by DRS, and 2) DRS-LB that further incorporates load balancing enhancement of StreamSwitch if DRS does not trigger any scaling decision. We also compare StreamSwitch with the spectrum of static solutions that use a fixed number of executors without incurring *switching overheads* such as scaling and state migration. However, as static solutions are not self-adapting, it is impractical to derive the optimal number of executors for unpredictable workload.

Application Workload We evaluated StreamSwitch using three different application and benchmark workloads.

1. Financial Application We use a trace of anonymized orders traded in the Shanghai Stock Exchange (SSE). It consists of 14,854,215 purchasing/selling orders of 1,156 stocks traded in the morning (09:30-11:30) and afternoon (13:00-15:00) sessions of a typical trading day. The application runs a continuous auction and maintains the states of pending buy/sell orders, each includes around a dozen fields including quote id, stock id, quote price, purchase direction and volume. When a new order arrives, it checks account balances and tries to match pending orders to generate trading transactions.

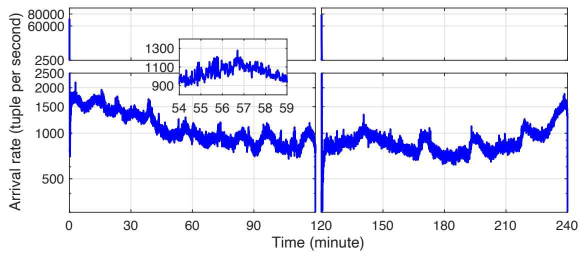

Figure 4: Workload of the limit orders in the 4 trading hours.

As shown in Figure 4, the workload varies frequently from hundreds to thousands of tuples per second, and has spikes of 70K-80K/s at the opening of the trading sessions due to the nature of equity trading. Furthermore, this financial data workload is also highly skewed and exhibits a power-law type of distribution, where the top 1% (11 stocks) and the heaviest stock contribute more than 10% and 2% of the total orders.

2. NEXMark [42] This benchmark suite models an online auction system where a stream of users, auctions and bids arrive. We choose representative queries[1] Q5 and Q8 to demonstrate stateful operators: sliding window and tumbling window join.

3. Word Count This is a benchmark used by Dhalion[13] and DS2 [24] and has a two-operator Split-Count topology.

Experimental Setup We run experiments on a cluster of 4 machines, each has a Intel Xeon Silver 4216 @2.10GHz processor with 16 cores and 64GB of RAM, running Ubuntu 18.04.4 LTS. All the machines are in located in the same server rack, connected by a high-speed switch with 1 Gbps ports. Samza 1.0.0 and Flink 1.8.1 are used as stream engines and Kafka 0.10.1.1, YARN 2.6.1 and ZooKeeper 3.4.3 are used for metrics delivery, resource management and coordination, respectively. We use a default value of 100ms for the latency threshold l and examine interval δ of StreamSwitch.

6.1 Overall Performance

To stress-test StreamSwitch under long-run bursty workload, we concatenate 2 trading sessions into a 4-hour input stream, partitioned into 64 substreams by equity IDs. We built an application whose operator runs a continuous auction to check the validity of orders and generate trading transactions. The operator needs to maintain a state of pending and partially fulfilled orders, whose quantities starts around 370K entries at the opening call auction and reaches over 2M at the end.

Figure 5 compares the performance of StreamSwitch with other solutions under $(\mathrm{L}, \mathrm{T}) = (1s, 1s)$ on Samza, showing the success rates along the y-axis. To make fair comparisons with DRS and DRS-LB, we generalize DRS's M/M/k model by introducing the same safety margin ϵ in the service rate. The left subfigure shows when ϵ increases, the self-adapting solutions trade off resource frugality against success rate. This illustrates that users could increase or decrease the safety margin ϵ to achieve higher success rate or lower resource usage. In particular, StreamSwitch achieves much higher success rates using comparable amounts of resources, while DRS and DRS-LB perform worse than the chosen static solutions. The mid subfigure boxplots the 64 per-substream success rates for the

[1]We tested StreamSwitch for all the queries of the NEXMark benchmark suite. Due to space limit, we show the adaptivity of StreamSwitch on heterogeneous resource using query Q2.

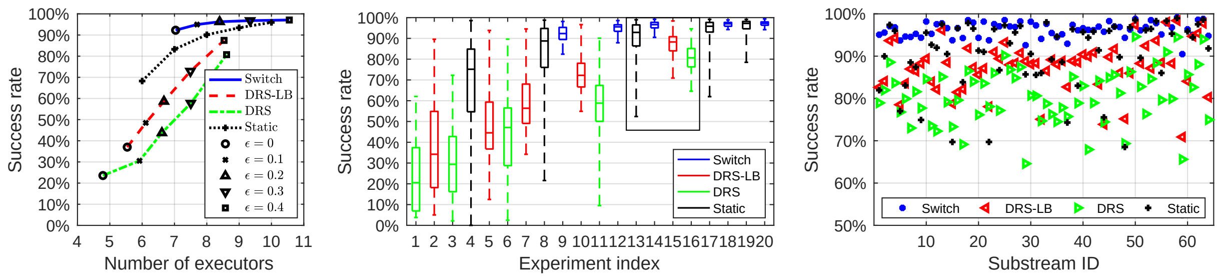

Figure 5: Performance comparison under $(\mathrm{L}, \mathrm{T}) = (1s, 1s)$. Left: success rates vs. numbers of executors; Middle: boxplots sorted by amount of used resources; Right: the success rates of 64 substreams under Switch(0.2), Static(8), DRS-LB(0.4), and DRS(0.4).

20 experiments, sorted by the number of used executors in an ascending order, showing that StreamSwitch also achieves much lower variances. We take $\epsilon = 0.2$ as our StreamSwitch baseline, referred to as Switch(0.2), and choose the solutions that use the closest numbers of executors shown in an inner box and plot their per-substream success rates in the right subfigure. We find that Switch(0.2) achieves high success rates across all 64 substreams: only 6 and 1 of them are slightly lower than those under Static and DRS-LB, respectively.

Overall, vanilla DRS does not perform well and DRS-LB is sub-optimal, implying that it is necessary to integrate scaling and load balancing in a unified framework like *stream switch*. We show that StreamSwitch is able to trade off resource frugality against success rate via the safety margin ϵ and outperforms alternative solutions by achieving uniformly high success rates for all substreams using comparable resources. Because success rates increase naturally as the latency SLAs relax, our evaluations under $(\mathrm{L}, \mathrm{T}) = (1s, 1s)$ show that StreamSwitch is able to bound latency in the order of seconds. Unless specified particularly, we will use $(\mathrm{L}, \mathrm{T}) = (1s, 1s)$ as the target latency SLA for all the remaining experiments. [2]

6.2 Detailed Behavior of StreamSwitch

We illustrate the control decisions and the resulting latencies within the 4-hour runtime under the baseline Switch(0.2) in Figure 6. The changes in the number of executors correspond to the scaling decisions and the load balancing decisions are marked by circles in the upper subfigure. We find that StreamSwitch strives to balance load most of time, while adapts to the workload by scaling the operator accordingly. For instance, it scales out at the opening of trading sessions and towards the day-end closing when the aggregate workload surges. The lower subfigure shows that the mean and P99 latencies are mostly in the range of tens to hundreds of milliseconds, although the P99 latency overshots the 1s constraint some times, especially at the opening times when the instantaneous arrival rates are 70-80 times higher than their average rate.

Figures 7-9 illustrate how StreamSwitch responds to a data burst in a 5-minute period (shown by the embedded graphs in Figure 6) using scaling and load balancing. Figure 7 shows when executor E18's arrival rate approaches its service rate at 56.80min, because its latency also exceeds the threshold l, a scale-out is triggered to move approximately half workload, i.e., 6 substreams, to a new

[2] We also conducted evaluations for various SLAs (L, T) and tested StreamSwitch on Flink, from which similar observations can be made. Due to space limit, we cannot include them in this paper.

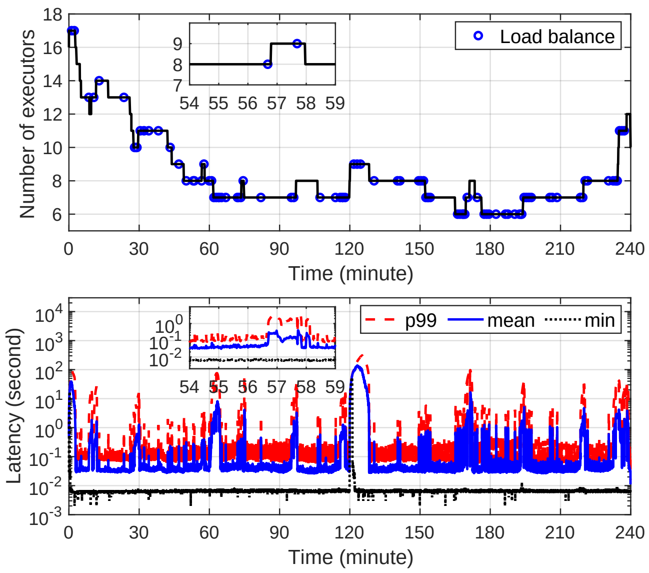

Figure 6: Strategies and the latency of tuples.

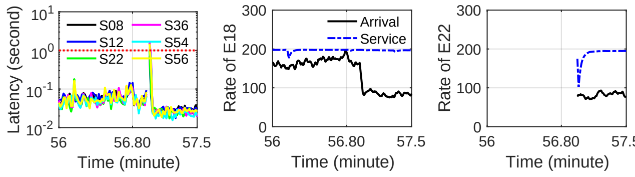

Figure 7: Scale out (6 substreams) at minute 56.80.

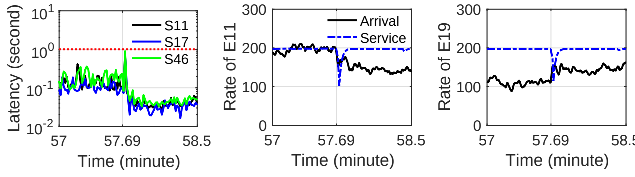

Figure 8: Balance load (3 substreams) at minute 57.69.

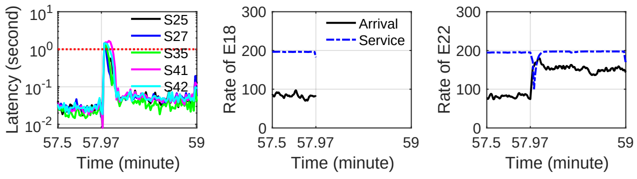

Figure 9: Scale in (5 substreams) at minute 57.97.

executor E22. Figure 8 shows when E11 becomes severe at 57.69min, around a quarter of its workload is migrated to E19 to balance

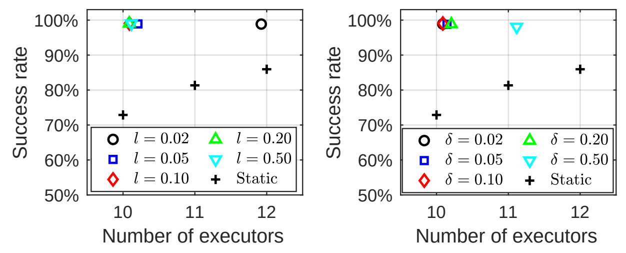

Figure 10: Impact of the parameters l and δ on performance.

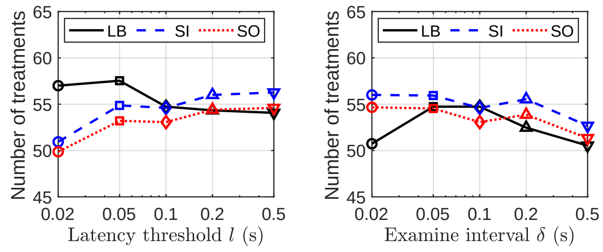

Figure 11: The number of treatments performed.

load. Figure 9 shows when the data rate mitigates at 59.97min, E18 becomes underloaded and migrates all workload to E22 with a scale-in. In all three scenarios, we find that although the latency of the migrated substreams slightly overshots 1s due to the overheads of synchronization and state migration, the latency returns back to a normal region quickly afterwards.

In summary, we show that StreamSwitch adapts to changes in the aggregate workload via scaling, while responds to load skewness via load balancing. A source of latency constraint violation may come from the overheads of switching decisions, a fundamental limitation to be explored a later subsection.

6.3 Impact of System Parameters

Besides the safety margin ϵ, we evaluate how the other two parameters l and δ impact performance. We run query Q8 of window join from the NEXMark suite [42] over Flink with Switch(0.2). We use dynamic workloads of input streams that follow sinusoidal patterns with a base rate of 30K per second, parameterized by an amplitude of 30K and a period of 10 minutes. We set the parameters l and δ to be 0.1s by default and vary them from 0.02s to 0.5s in Figure 10. For each setting, we run fifteen 30-minute runs and take an average. We observe that the success rates do not get affected much as the parameters change. Although the numbers of consumed executors are slightly larger when $l = 0.02$s and $\delta = 0.5$s, StreamSwitch is robust in performance and still outperforms other solutions under all scenarios as before.

Figure 11 plots the numbers of scaling and load balancing events triggered, as the parameters increase along the x-axis. In the left subfigure, we observe when l is within 100ms, due to the low alert threshold, the system is sensitive to load and does not scale-in much. Under these scenarios, most severe executors can be cured via load balancing prescribed by the system. In contrast, when l increases beyond 100ms, instead of load balancing, more scaling-out treatments need to be carried out to cure severe executors. Nevertheless, because typical latency ranges from tens to hundreds of milliseconds, values of l within that range all provide reasonable thresholds for achieving high success rates. In the both subfigures, we also observe that the numbers of scale-in and scale-out are very similar and have the same trend with varying parameters. The reason is that the rising and falling of workload, which trigger scale-out and scale-in respectively, are symmetrical and periodical under sinusoidal patterns. This demonstrates that StreamSwitch adapts to dynamic workloads very well.

6.4 Heterogeneous Resources

In this subsection, we evaluate whether StreamSwitch accommodates platforms that consist of heterogeneous resources. Since our experimental cluster consists of homogeneous CPU resources, we evaluate the performance of StreamSwitch over a heterogenous resource environment by emulating the different service rates achieved by different types of resources. In particular, we use a baseline service rate of container and determine an allocated container to be of a different type with the service rate α times of that of the baseline with probability β, where $\alpha < 1$ or $\alpha > 1$ emulates resources that have inferior or superior processing power, and β models a hybrid platform with a percentage β of mixed resource.

We run query Q2 from the NEXMark suite over Flink with Switch(0.2). We use dynamic workload that follows sinusoidal patterns with a base rate of 30K per second, parameterized by an amplitude of 10K and a period of 10 minutes. The baseline resource is rate-limited to have a service rate around 4K per second. We choose α to be 0.5, 0.75, 1.25 or 1.5 to emulate resources whose processing rates are ±25% and ±50% from the baseline. We plot the average success rate (left) and number of used executors (right) in Figure 12 where β varies from 0.2 to 0.8 along the x-axis. To eliminate transient behaviors, for each setting, we show the average of 10 experiments, each runs 20 minutes for warm-up and an hour for the performance statistics. We find that close to 100% success rates are achieved under all scenarios, while the number of executors increases when the mixed resource becomes more inferior under any fixed percentage β. Furthermore, the number of executors increases with β if the mixed resource is inferior to the baseline, i.e., $\alpha < 1$, and vice versa, because more executors are needed when each becomes less powerful. Figure 13 plots the corresponding percentage of mixed resources used (left) and the total number of treatments performed (right). Compared to the supplied percentage β, we find that a higher proportion of superior resource is used in every scenario, i.e., a lower percentage of mixed resource is used when $\alpha < 1$, and vice versa. This is a consequence of the scale-in mechanism of StreamSwitch, which always tries to minimize the worst latency of the executors and tends to abandon inferior resources. We also observe that more treatments get triggered with worse resource, since more executors need to be used to accommodate workload. Consequently, the success rates drop below 99% when many treatments are triggered under $\alpha = 0.5$. This is related to the fundamental limitation of StreamSwitch that we explore next.

6.5 Limitations of StreamSwitch

In this subsection, we reveal the limitations of StreamSwitch, and understand under what circumstances it is not able to guarantee latency regardless of resource consumption. Under our stream switch abstraction, since any switching decision involves scaling or load

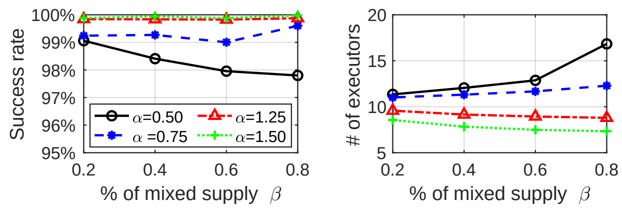

Figure 12: Performance under heterogeneous resources.

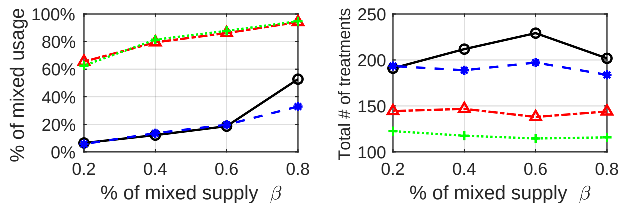

Figure 13: Resource distribution and number of treatments.

balancing, during which substreams are migrated from a source to a destination executor, switching decisions introduce overheads in the involved executors, e.g., executors cannot process data tuples at their normal service rates during synchronization or state migration. To ease our discussion, we call any treatment a *switching event* and define the amount of time spent on switching-related operations as *switching time*. Although influenced by various factors, the achievable success rate is fundamentally limited by the percentage of time executors spend on switching.

We run NEXMark query Q5 over Flink with Switch(0.2) and use sinusoidal workloads with a baseline rate of 30K per second. Each executor is rate-limited to a service rate of 10K per second to emulate bottleneck resources. We run five workloads with amplitudes and periods varying along the x-axis and boxplot the performance of 10 runs in Figure 14. Each experiment runs for 10 minutes with an additional 1-minute warm-up. We observe when the amplitude increases from 10K to 30K and the period decreases from 90s to 70s, although the average number of executors used by StreamSwitch increases, the achieved success rate keeps decreasing and drops below 80%. The increase in amplitude makes the workload more bursty and the decrease of its period makes the workload change more frequently. To cope with such dynamics, we observe that 1) as the amplitude increases, the number of switching events increases from 60 to more than 130, while the average switching time increases from around 0.5s to a bit longer than 1s; 2) as the period further decreases, although the average number of switching events stays around 130, its variance and the average per-event switching time increase prominently. As a result, a clear negative correlation can be observed between the success rate and the percentage of time spent in the switching operations shown in the mid-subfigure, which is contributed by both the number of switching events and the per-event switching time.

Besides the dynamics of workload, there exists other factors that influence the per-event switching time, for example, the synchronization protocols implemented by the stream systems may induce various amounts of latency and larger sizes of states may induce longer migration time. To understand the impacts of such factors, we use 10K-amplitude and 90s-period as a baseline and introduce an additional latency to each switching event, varying along the x-axis in Figure 15. We observe that the additional latency amplifies the per-event switching times,which reach 6s and 13s after adding delays of 1s and 2s, respectively, because additional delays cause large amount of backlogs to be accumulated at executors. Although the average number of executors used does not increase much and the number of switching events even decreases (as the system is unable to adapt when the per-event switching time is large), we still observe that the percentage of time executors spent on switching operations increases monotonically, exhibiting a clear negative correlation with the success rates.

6.6 Multi-Operator Scenarios

Although aiming at per-operator SLAs, StreamSwitch can also be used to bound end-to-end latency in multi-operator scenarios. Following prior works [13, 24], we run the word count benchmark on Samza with a base rate of 15K sentences per second and rate limit the operators such that 10 Split and 20 Count operators are needed to accommodate the base rate. We vary the workload using a sinusoidal pattern with ±25% amplitude and 5-minute period, and consider an end-to-end latency SLA of L=2s over a T=1s time window.

We vary the latency bounds (L_S, L_C) of the Split and Count operators to be $(0.5s, 1.5s)$, $(0.75s, 1.25s)$, $(1s, 1s)$, $(1.25s, 0.75s)$, and $(1.5s, 0.5s)$ along the x-axis in Figure 16. For each setting, we run twenty 10-minute runs, each with 5-minute warm-up, and boxplot the success rates of the two operators and the end-to-end path. In the upper subfigure, we observe that the success rate of an operator, i.e., Split or Count, shows an increasing trend when its latency bound, i.e., L_S or L_C, increases. Intuitively, the latency SLA is more likely to be fulfilled as the bound becomes more relaxed. We also observe that when L_S or L_C decreases below 0.5s, the success rate of Split or Count sometimes drops below 90%, showing that StreamSwitch does not bound sub-second latency well.

In the lower subfigure, we observe that the end-to-end success rates, however, are mostly between 97% to 99%, showing the flexibility of StreamSwitch for achieving end-to-end latency SLAs, even when some operator's latency is not well bounded. This is because most of time the per-operator latency does not exceed 0.5s too much and therefore, the aggregate latency does not exceed the 2s end-to-end bound. Furthermore, we observe that the highest success rate for the end-to-end latency is achieved at $(L_S, L_C) = (1.25s, 0.75s)$, under which the worse success rate of the two operators Split and Count is maximized among the five configurations. The intuition is that such a solution prevents either operator from exceeding its latency bound L_S or L_C too much, resulting in a higher chance of fulfilling the end-to-end latency SLA.

From the above observations and reasoning, a strategy to maximize the success rate of an end-to-end latency SLA is to distribute the end-to-end bound L to the individual operators such that the resulting vector of the success rates of the operators maximizes its worst success rate components, i.e., the max-min solution in terms of the per-operator success rates. As the success rate of an operator increases with its latency bound, given a fixed end-to-end latency SLA, the max-min solution is achieved when all the success rates are equalized. In practice, one could measure the success rates of operators by their instantaneous latency metrics l_i at intervals and

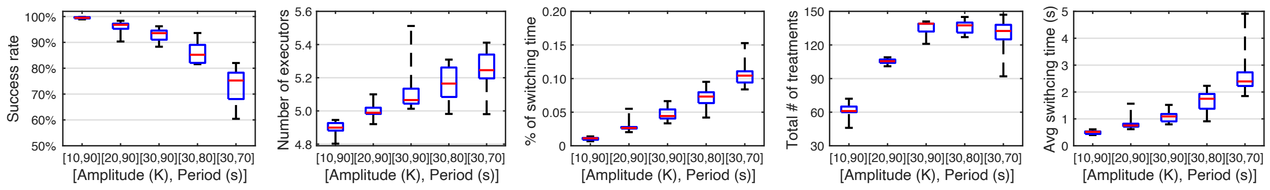

Figure 14: Performance degradation as workload becomes more bursty and dynamic.

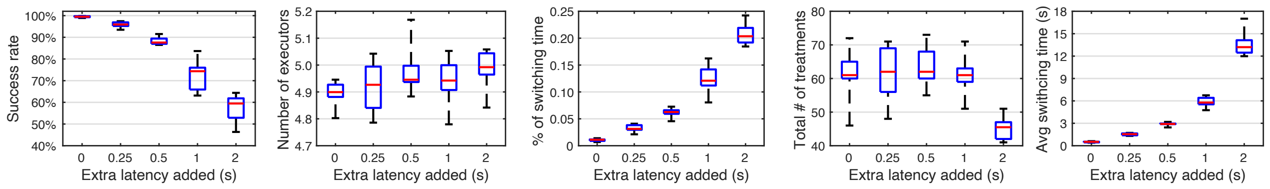

Figure 15: Performance degradation as the per-treatment switching time increases.

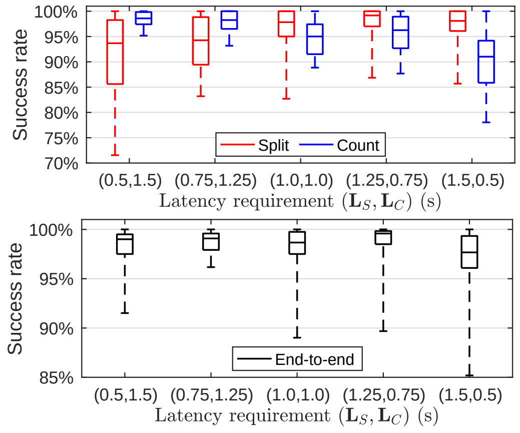

Figure 16: Performance under different configurations.

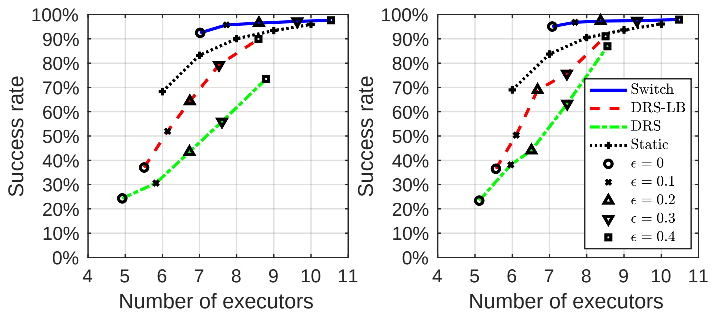

Figure 17: Performance comparison under $(L, T) = (1s, 5s)$ in the left subfigure and $(L, T) = (5s, 1s)$ in the right subfigure.

adjust the allocated latency bounds L_i to equalize the success rates.

6.7 More Evaluations for Financial Application

We present additional evaluations for the financial application presented in Section 6. In the following, we first show the performance comparison between StreamSwitch and alternative solutions under different latency SLAs (L, T) in Figure 17, similar to that of Figure 5. We then show the performance of our baseline Switch(0.2) under Flink in Figure 18, similar to that of Figure 6. Through these additional evaluations, we can make similar observations, confirming the conclusions shown previously.

Figure 17 compares the performance of StreamSwitch with DRS and DRS-LB under different values of the margin ϵ and the static solutions, where the latency SLA (L, T) is set to be $(1s, 5s)$ and $(5s, 1s)$ in the left and right subfigures, respectively. Similar performance trends can be observed with success rates higher than that of $(L, T) = (1s, 1s)$ under these more lenient constraints. In particular, StreamSwitch 1) still outperforms alternative solutions similarly as previously shown, 2) improves the success rate as T or L increases under any fixed margin ϵ, and 3) increases the success rates by 0.5% and 1% on average when $T = 5s$ and $L = 5s$, respectively.

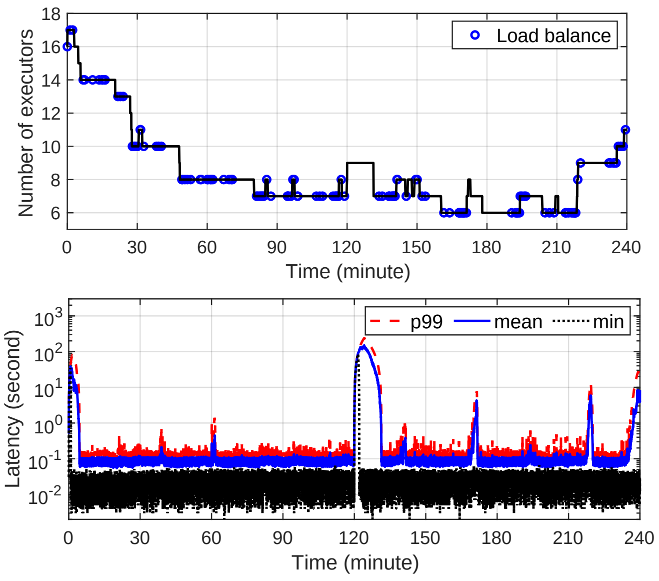

Figure 18: Strategies and the latency of tuples on Flink.

We also evaluated StreamSwitch on Flink and show the control decisions and the resulting tuple latencies within the 4-hour runtime of the financial application under the baseline Switch(0.2) in Figure 18. Compared to the 96.28% success rate and 8.40 executors on average under Samza, StreamSwitch achieves slightly lower success rate at 95.24% with a comparable average of 8.39 executors under Flink. It is interesting to notice that although Flink achieves faster state migration during treatments, under which the latency SLA rarely gets violated; however, due to the use of stream barrier for synchronization, *switchpoint* gets delayed by the surge of the data peak at 120min, which takes a longer time for Flink to recover from the latency spike and contributes to the lower success rate. Furthermore, Flink also exhibits larger variances on the minimum latencies compared to Samza.

7 RELATED WORK

Dataflow engine With the rise of cloud computing, plenty of distributed data processing frameworks have emerged from industry [1, 2, 23, 28, 32, 36] and academia [3, 16, 33, 49, 51]. some of which have evolved into open-source engines [7, 27, 35, 41, 50]. Many of them [1, 28, 36, 49] were centered around fault tolerance. Besides reliability, [2, 16, 32, 33, 51] were proposed to support various requirements. CIEL[33] and Naiad[32] used data-dependent control-flow to handle dataflows with cycles. Noria [16] and Wukong+S [51] provided low-latency query with a relational schema and persistent store. Dataflow[2] model provided semantics for event-time ordering. Unlike the *bulk synchronous parallel* (BSP) model [43] that is adopted by Dryad [23], CIEL[33] and Spark [50] a continuous operator model is adopted by pure streaming engines, for which StreamSwitch is designed as a control plane.

Control plane There have been a few recent developments of control planes [13, 14, 20, 24, 29, 30] for stream processing. Chi [30] supports continuous monitoring and dynamic re-configuration by inserting control messages into data streams. Snailtrail [20] applies critical path analysis for BSP-based systems andmakes reconfigurations between micro batches. StreamSwitch applies a similar idea by embedding control messages as stream barriers in Flink to achieve treatments. Both Dhalion[13] and DS2 [24] make auto-scaling decisions to maximize system throughput. StreamSwitch however targets latency objectives under which throughput is also maximized. In particular, the EDT framework is similar to the `Health Manager` in Dhalion and the calculation of the steady-state latency L_i relies on metrics like *useful time* suggested by DS2. DRS [14] and Nephele [29] also target latency guarantee for stream processing; however, both rely on the assumptions of homogeneous resources and stateless operators, under which the decisions of which substreams to migrate are not specified, nor are the state migration mechanisms implemented.

Data plane To achieve high-availability [22, 40] and fault tolerance [1, 10, 28, 36, 49] for stateful stream operators, state management [6, 9, 12, 21, 31] has become a recent focus. SEEP [9] and SDG [12] can integrate state recovery and scale out asynchronously by exposing internal operator state. Flink [6] achieves distributed checkpoints via stream barriers. Megaphone [21] proposes a fluid migration mechanism for state migration. Recently, Rhino [31] has been proposed to handle states with TB sizes. Although state management and fault tolerance are orthogonal to the focus of StreamSwitch, the efficiency of the underlying state management will limit the effectiveness of StreamSwitch, since the treatments induce unavoidable latency overhead. Besides state management, plenty of work explored the load balancing [25, 34, 37] and elasticity [11, 15, 17, 18, 46–48] aspects for performance optimization. StreamSwitch unifies both under the *stream switch* abstraction and makes principled control decisions within a holistic decision space.

8 CONCLUSIONS

We presented StreamSwitch, a system solution to guarantee latency for stream analytics. Under fairly weak assumptions of resource elasticity and dataflow-based paradigm, it makes holistic control decisions of scaling and load balancing in the solution space defined by a novel stream switch abstraction. To demonstrate its effectiveness, we built StreamSwitch as control planes for Samza and Flink. Although switching overhead limits StreamSwitch's capability of bounding sub-second latency, this barrier can be reduced by more efficient system mechanisms in the data plane, for example, fast state migration mechanisms such as Megaphone [21] and Rhino [31].

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at Internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[3] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (2014), 939–964.

[4] Walid A. Y. Aljoby, Xin Wang, Tom Z. J. Fu, and Richard T. B. Ma. 2019. On SDN-Enabled Online and Dynamic Bandwidth Allocation for Stream Analytics. *IEEE Journal on Selected Areas in Communications* 37, 8 (2019), 1688–1702.

[5] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[6] 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.

[7] 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).

[8] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (2019), 44–54.

[9] 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 *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 725–736.

[10] Badrish Chandramouli and Jonathan Goldstein. 2017. Shrink: prescribing resiliency solutions for streaming. *Proceedings of the VLDB Endowment* 10, 5 (2017), 505–516.

[11] Tiziano De Matteis and Gabriele Mencagli. 2017. Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators. In *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 61–68. https://doi.org/10.1109/PDP.2017.31

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

[13] 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.

[14] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 411–420.

[15] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1447–1463.

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

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

[18] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 276–287.

[19] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[20] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. 2018. SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 95–110.

[21] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. 2019. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.

[22] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. 2005. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 779–790.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*. ACM, 59–72.

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 783–798.

[25] Nikos R Katsipoulakis, Alexandros Labrinidis, and Panos K Chrysanthis. 2017. A holistic view of stream partitioning costs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1286–1297.

[26] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*. 1–7.

[27] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.

[28] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. Streamscope: continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 439–453.

[29] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[30] 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.

[31] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM.

[32] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[33] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[34] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. 2016. When two choices are not enough: Balancing at scale in distributed stream processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 589–600.

[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] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 1–14.

[37] Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 80–91.

[38] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. 2010. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1525–1528.

[39] Apache Samza. [n.d.]. *JMX-based Metrics*. https://samza.apache.org/learn/documentation/0.7.0/container/jmx.html.

[40] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. 2004. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 827–838.

[41] 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 *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[42] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2004. NEXMark – A benchark for queries over data streams. In *Technical Report, OGI School of Science and Engineering at OHSU*.

[43] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing (SoCC)*.

[45] Di Wang, Elke A Rundensteiner, Han Wang, and Richard T Ellison III. 2010. Active complex event processing: applications in real-time health care. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1545–1548.

[46] Li Wang, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, and Zhenjie Zhang. 2019. Elasticutor: rapid elasticity for realtime stateful stream processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 573–588.

[47] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 723–734.

[48] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *Proceedings of the International Conference on Cloud Engineering (IC2E*. IEEE, 22–31.

[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 *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. 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 *Proceedings of the 24th ACM symposium on operating systems principles*. 423–438.

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