Resource Efficient Stream Processing Platform with Latency-Aware Scheduling Algorithms

Yuta Morisawa, Masaki Suzuki, Takeshi Kitahara
KDDI Research, Inc.

Abstract

We presented a novel platform dedicated to stream processing that improved resource efficiency by sharing resources among applications. The platform utilized latency-aware schedulers to handle stream applications with heterogeneous SLAs and workloads. We implemented the prototype in Spark Structured Streaming and evaluated the platform with pseudo IoT services. The result showed that our platform outperformed default Spark Structured Streaming while reducing the necessary CPU cores by 36%. We further compared the adaptability of the schedulers and found that one of the schedulers reduced the SLA violations by 90% compared to the default FAIR when the platform was overloaded.

1 Introduction

Stream processing has become a major region in big data analytics, such as the monitoring of IoT services, web-site anomaly detection, and dynamic pricing in e-commerce. These services require low-latency and high-throughput processing and may have a stringent latency service level agreement (SLA) as a deadline for execution. Moreover, current business situations demand systems that cover a variety of aspects: scalability, durability of 24x7 processing, adaptability to varying workload pattern, and mitigation of the operation burden.

Several stream processing platforms have been published to meet the demand. Apache Storm [6] and Apache Flink [2] are major stream processing platforms that adopted the continuous processing model for lower latency processing. Spark Structured Streaming [11], which is an extension of Apache Spark, inherited the micro-batch feature from Spark Streaming [30], which divides infinite data into a group of records.

Today, it is common to run hundreds of stream applications in one organization because of business divergence and complexity. In such an environment, improvement of resource efficiency has become a critical challenge because low resource utilization leads directly to an increase in platform cost. However, resource optimization is quite challenging since each stream application has different latency requirements and different workloads. Traditional resource schedulers do not consider the applications' latency to observe the SLAs since these schedulers are designed for batch computing [4, 7, 23]. As the input data rate often varies unexpectedly, the resource scheduler must detect resource shortages to reallocate resources dynamically in real time. The reallocation should be completed on the order of milliseconds since stream applications sometimes have sub-second latency SLAs. An existing coarse-grained resource sharing mechanism, where the cluster manager removes JVMs from an application and adds to another application, is too time consuming to achieve the real-time reallocation [5]. The scheduler must also determine the priority that applications use resources if the platform is overloaded. This is a significantly different perspective from that of traditional schedulers.

Many studies have been made on inner platform schedulers, for example, mitigating data skew within workers [20, 29], reusing the results of prior jobs [13, 18, 22], adapting the scheduler for a heterogeneous environment [21, 27], modeling the platform behavior to mitigate the performance bottle necks [16, 17], and adaptively rebuilding job graphs based on the workload [28]. Xu et al. proposed a task level scheduler that assigns tasks based on the input data rate [26]. Cheng et al. optimized job parallelism in Spark Streaming [15]. These studies have proved that pre-embedded schedulers in major platforms [2, 6, 30] might not work sufficiently since stream processing has completely different requirements from traditional batch computing. However, these studies assumed that a cluster accommodated a single stream application, and effectiveness was limited in an environment where multiple applications with different latency requirements exist.

To minimize unused resources, a straightforward approach is to share resources with other applications in the same cluster. Dynamic resource allocation (DRA) [5] and Mesos fine-grained scheduler [4] provide mechanisms to adjust resources in Apache Spark. These mechanisms allow applications to release resources back to the cluster if they no longer use them and request them when the applications have urgent tasks. Spark Job Server [10] is a library for managing Spark jobs, which allows adding or removing jar files, data, and contexts via REST APIs. This library also supports running multiple jobs inside a single context. Yuzhao et al. proposed a method to interconnect schedulers among different layers [25]. Guolu et al. evaluated a latency-aware scheduler with multiple applications environment [24]. These studies provided the frameworks to improve resource efficiency but had no scheduling algorithms that could optimize resource allocation in real time while considering the applications' SLAs.

Earliest deadline first (EDF) is a scheduler in the context of real-time systems [12]. EDF considers the deadline of each application and schedules the application with the closed deadline first. However, the algorithm does not work correctly in an environment where applications have significantly different SLAs. For instance, there are two tasks with 6 seconds and 100 seconds in the SLAs. Assuming that when both have the same deadline and the remaining time is 5 seconds, the former task can afford to wait for resource assignment since the remaining time is as much as 83% of the SLA until the deadline, but the latter task appears to face the deadline.

To the best of our knowledge, no prior work had been published that could improve resource efficiency in an environment where multiple stream applications existed while keeping the applications' SLAs. The challenge was to schedule heterogeneous applications with different latency requirements and input data rates. Motivated by our observations, we developed a novel stream processing platform that provided latency-aware resource sharing mechanisms. We summarize our contributions as follows:

- We propose a platform that achieves better resource efficiency by accommodating multiple applications in one cluster. The platform can also handle heterogeneous SLAs with latency-aware schedulers. We implement the prototype by extending Structured Streaming.

- We implement three latency-aware task schedulers motivated by the scheduler of real-time systems [12]. The schedulers determine the priority of each application based on the current load, deadline, and the applications' SLAs. We evaluate these schedulers with pseudo IoT services.

- The experimental result showed that one of the implemented schedulers outperformed the default Structured Streaming while reducing the necessary CPU cores by 36%. We compared the latency-aware schedulers with default scheduling algorithms and found that the scheduler reduced the average latency at most by 61% compared to FIFO. Another experiment showed that one of the schedulers reduced the SLA violation by 90% compared to FAIR when the platform was overloaded.

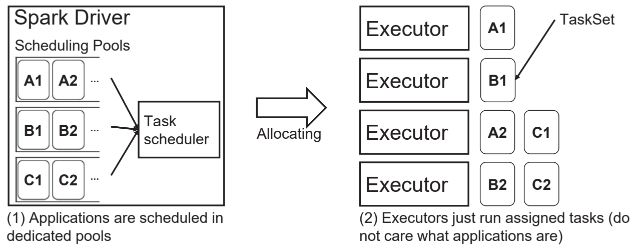
Spark Driver
Scheduling Pools
A1
A2
B1
B2
C1
C2
Task scheduler
Allocating
Executor
Executor
Executor
Executor
A1
B1
A2
C1
B2
C2
TaskSet
(1) Applications are scheduled in dedicated pools
(2) Executors just run assigned tasks (do not care what applications are)

Figure 1: Architecture Overview

2 Platform Design

2.1 Architecture

Our platform is designed for an environment where multiple stream applications coexist. The key features are as follows: 1) accommodate multiple applications in one Spark application, 2) schedule at task level granularity with latency-aware algorithms. We chose the design that accommodating multiple applications in one cluster since it achieved the finest granularity of resource scheduling. Moreover, reallocating processes can be completed with lower latency since the master process can monitor every application and manipulate its scheduling order. This feature is desirable for the situation that latency-aware applications increase. On the other hand, since the design does not provide resource isolation, it may occur interference between applications. The detailed discussion is conducted in section 5.

Figure 1 shows the architecture overview. Our platform basically runs as a Spark cluster composed of the Spark Driver and Executors. Structured Streaming applications in our platform behave as if they run on the default Spark cluster since we did not make any change on the Structured Streaming APIs. We assume that every application has its own data source and sink outside the platform, which means we do not allow applications to read other applications' data even though this is theoretically possible. Otherwise, the scheduler must consider the dependency between applications, which makes the scheduling algorithms complex. Moreover, the recovery from failures also becomes slow since the platform needs to trace the data-lineage in order to re-compute dependent data.

The jobs generated by each application are buffered in the dedicated scheduling pool. Our latency-aware scheduler allocates resources to the pools according to the priorities. The priorities are determined by the latency-aware algorithms described in the next subsection.

2.2 Latency-aware task scheduler

Background Spark is a distributed computing platform for general purpose processing. Structured Streaming is an extension of Spark, which is a high-level API for stream processing.

It optimizes user programs and generates micro batch jobs that are transferred to the Spark core engine as traditional batch jobs. Spark has two scheduling layers, i.e., job level and task level. The job is a unit of execution composed of tasks, and the task is an atomic unit of execution. The former determines the scheduling policy for execution and schedules jobs in the Spark cluster. The latter then assigns tasks individually to the appropriate executor according to the priority.

The task scheduling mechanism is similar to Mesos fine-grained resource allocation [4]. Executors transmit `resourceOffer` to the master when they have available resources. The master then assigns `TaskSet`, which is a set of tasks that can be scheduled concurrently, to the executor transmitted `resourceOffer`. There are two scheduling levels for resource allocation, i.e., inter-pool and intra-pool. The inter-pool level scheduling prioritizes the scheduling pools, which behave as queues for `TaskSet`. The intra-pool level scheduling determines the priority of the `TaskSet` inside the pool.

Overview: The goal of the scheduling is to make the latency of every application less than each SLA. In this paper, we treated the job duration as the latency. Ideally, we must consider the queuing delay occurred in the storage layer (e.g., Kafka, HDFS, and DB). The reason we excluded the queuing delay from the priority calculation is that to obtain the precise queuing delay makes the system too complex. The detail is discussed in section 5. We note that we evaluated the platform and the scheduler by measuring end-to-end latency including the queuing delay.

To satisfy the SLAs of applications, we had two design choices: estimating the necessary resources to achieve the SLAs for each application and scheduling them according to the calculated plans, or treating applications as black boxes and scheduling them according to the urgency. The former is suited for batch computing because the scheduler can know the input data size in advance. However, as for stream processing where the data rate varies in time, this kind of estimation is difficult to apply, especially for our platform that accommodates multiple applications. Inaccurate estimation can lead to inappropriate resource allocation and result in SLA violations. We chose the latter design, which leverages the existing Spark scheduler features. We utilized the Spark task scheduler for allocating resources to each application. As we described in section 2.1, we assumed that the applications belonged to the scheduling pools separately. Unlike the default schedulers that statically determine the priority of the scheduling pools, our scheduler calculates priority for each `resourceOffer` that comes in, and then allocates resources dynamically based on the priority. We limited the intra-pool scheduling method to FIFO since applications may have inter-dependent jobs. We designed three scheduling algorithms to calculate the priority based on the SLAs.

Earliest Deadline First (EDF): The EDF is a traditional scheduling algorithm that schedules the task first that has the closest deadline [12]. When we denote the active job of application s as Job_s, then the submission time of Job_s as T_s and the current time as T_c, $T_c - T_s$ defines the spent time of Job_s. In this paper, we assumed that every job that belonged to the same application had the same latency SLA, so we defined the SLA of application s as L_s (equal to the maximum process time allowed for each job). Under this condition, when we denote the priority of Job_s as P_s, P_s is defined as $-1 * (L_s - (T_c - T_s)) = T_c - T_s - L_s$. The scheduler calculates the priority for each group of tasks and finds the biggest value in each pool as the representative value. Resources are allocated in order of the pool with the largest representative value.

Priority based EDF: Because the EDF simply considers the deadline, it does not reflect the potential priority of each application. For example, when there are two applications, one is anomaly detection and another is aggregation for visualization, thus the former should have higher priority than the latter. Reflecting the potential priority, we modified the original EDF formula as follows: $P_s = \frac{T_c - T_s}{L_s} * p_s$ (Let p_s be the potential priority of the application s). Different from Guolu's work [24], we moved the L_s to the denominator to align the weight of the remaining time among the applications. Stream applications sometimes have more than ten times the SLAs that differ due to the variety of the roles (e.g., aggregation is prone to have a longer SLA than anomaly detection and the machine learning inference). To treat every application in the proper manner and priority, one common criterion is necessary.

Process time estimation: The former two algorithms do not consider how busy the application is. The data rate of the stream applications naturally varies, and some trends can be predicted (daily trend) but other peaks are not (sudden increase of events). To adapt such a trend, we define the estimated job execution time as $F(I(s, T_s))$ where $I(s, T_s)$ denotes the amount of input data the job should process. In the end, we define a new priority as: $P_s = \frac{T_c - T_s + F(I(s, T_s))}{L_s}$.

2.3 Limitations

Our platform is strongly motivated the observation that stream applications have the variety of the latency requirements and traffic patterns. When one application is at peak time, other applications are not, so resources can be accommodated. If every application refers the same source, the performance of our platform and schedulers is limited, but we believe it still improves resource efficiency.

The latency-aware task scheduler just determines which tasks use executors' cores so it cannot take other resources (e.g., memory, network, and storage) into consideration. We understand the necessity of the more general resource scheduler for treating the variety demands of applications and plan to investigate the benefit in the future. We note that today's version of our platform is still useful since stream applications are naturally CPU-bound [31].

3 Evaluation

Testbed Environment: We deployed our platform on a Hadoop cluster consisting of five bare metal nodes. Each node had 16 CPU cores, 128 GB of memory, 128 GB of SSD storage, and 15 TB of HDD storage. We used Hadoop 2.9.2 for the shared storage and Kafka 2.2.0 as the data source and sink. We also used YARN to adjust the resources to compare our platform with a default Spark cluster.

To evaluate the performance of schedulers, we referred to the white paper published by the 5G automotive association (5GAA) [1] and developed pseudo-connected car applications. The applications were written in the Structured Streaming API, and the details were as follows: **Parsing** it parsed the binary input data for conversion into the `DataFrame` format. The result was used by the latter two applications. **Searching** it searched targets to report road warnings by calculating the distance between a specific landmark and each car. **Windowing** it counted the number of cars in the target areas with the sliding window for statistics, visualization, and traffic congestion detection. We called these three applications the application set.

Implementation: We implemented schedulers in Spark 2.4.5 by modifying the scheduling and metrics classes. The priority calculation was implemented mainly in the `SchedulingAlgorithm` class that determined the priorities of the `Pool` and the `TaskSet`. We modified `DAGScheduler` and `TaskScheduler` to acquire input data history. We could not obtain real time metrics from the data source since Spark executed the tasks in a lazy manner. Instead of the live data, we then utilized the statistics of the data rate for $I(s,T_s)$. In this paper, we measured the actual latency of our connected car applications to determine the estimation function $F(I(s,T_s))$ and finally defined the following: $F(I(s,T_s)) = C_1 * I(s,T_s) + C_2$, where ($C_{1,Parsing}$, $C_{2,Parsing}$) = (1.48, 271), ($C_{1,Searching}$, $C_{2,Searching}$) = (0.0008, 509) and ($C_{1,Windowing}$, $C_{2,Windowing}$) = (0.009, 6570). $F(I(s,T_s))$ and $I(s,T_s)$ are in milliseconds and records, respectively.

Our platform also supported a fault tolerance mechanism using Spark Checkpoint. Checkpoint is a function to periodically save progress information in order to resume the processes if the platform is in failure status. Our platform allowed each application to use the Checkpoint mechanism separately and we confirmed that all applications could be resumed through a simple experiment.

Workload: We created test data using the traffic simulator Vissim [8]. Vissim simulates how cars and humans behave on a virtual map, and we customized it to simulate connected cars (e.g., cars share their location, velocity, and health data). In the experiments, we ran three copies of the application set (i.e., three copies of Parsing, Searching, and Windowing) and replayed the traffic with different scenarios for each application set.

Latency: We measured the end-to-end latency in Parsing and Searching by the timestamp appended by Kafka brokers [3] and employed the batch duration as the Windowing latency. The reason we used two measurement methods was that the aggregation functions must wait for the watermark in order to trigger the execution [11]. In other words, if the data stamped with a sufficiently new timestamp did not come in, Windowing would never start processing records. Including this queuing time in the latency was clearly incorrect. Thus, we used the batch duration as the latency criterion that purely measured processing time.

Table 1: Minimum number of CPU cores required to keep the SLAs

Platform	Application Type	Total Cores
Separating	Parsing	24
	Searching	12
	Windowing	3
Proposal	Parsing, Searching, Windowing	25

3.1 Resource Efficiency

We first evaluated the benefits of co-locating applications in one cluster. For the experiment, we adjusted the amount of resources and investigated the minimum resources needed to maintain the latency under the SLAs. We allocated sufficient memory statically and only changed the number of CPU cores in the experiment. We compared the case that running applications in different Spark clusters separately (Separating) and the case that co-locating applications in our platform (Proposal). In the experiment, we used the Priority-based EDF algorithm for the scheduler in our platform. Table 1 shows that each application type (i.e., Parsing, Searching, and Windowing) has heterogeneous resource demands, and when running applications separately, a total of 39 cores is needed. Compared to that, our proposed platform consumed only 25 cores to maintain the same performance for all applications. This was because our platform allowed applications to share resources dynamically and fewer CPU cores were in the idle state. This fact was also evident in CPU utilization.

Figure 2 shows the total CPU time per core during the workload performed by Separating, Earliest Deadline First (EDF), Priority-based EDF (PT), task time estimation (EST), FIFO, and FAIR. Except for Separating, the platform co-located applications in one cluster and used a corresponding scheduling algorithm. The graph shows that by co-locating applications, the platform achieved approximately twice as much CPU utilization than Separating. That means, in Separating, CPU spent more underutilized time due to input data rate variations.

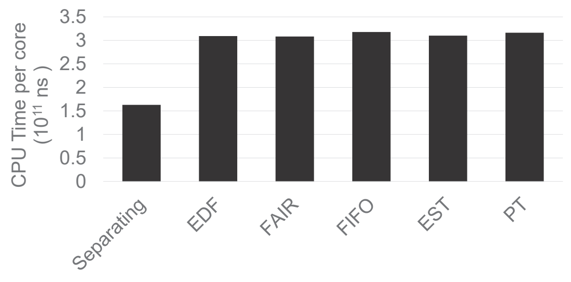

Figure 2: CPU time per core

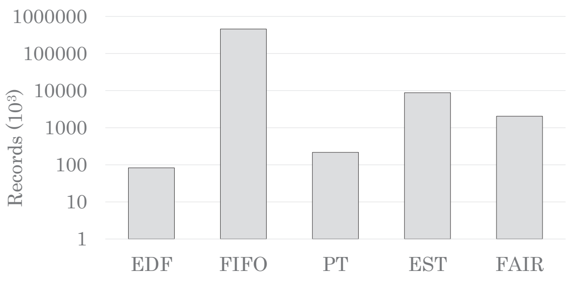

Figure 4: The number of records violated SLA

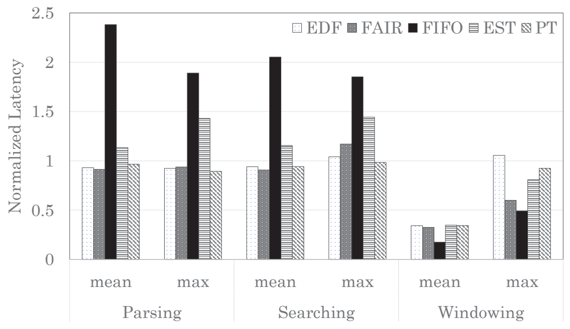

Figure 3: Normalized latency comparison

3.2 Scheduler Performance

Figure 3 shows the latency performed by EDF, PT, EST, FIFO, and FAIR. The latency is averaged in the same application type and normalized by the Separating's latency. Allocated cores (including Separating) are as shown in table 1. The figure shows that EDF, PT, and EST reduced the average latency of Parsing by approximately 61%, 59%, and 52% compared to FIFO. These scheduling algorithms also reduced the average latency of Searching by approximately 54%, 54%, and 44% compared to FIFO. By contrast, FIFO achieved the smallest latency for both mean and max in Windowing. This was because FIFO did not consider any other parameters except the job order and allocated surplus resources to Windowing even though the latency of Windowing fully satisfied the SLA. Compared to FAIR, EDF and PT reduced maximum latency in Searching by 11% and 16%; however, the latency of Windowing increased. The reason was that EDF and PT preceded urgent tasks to avoid SLA violations even if that resulted in interrupting other running jobs. The policy eventually obtained the feature that reduced maximum latency by slightly sacrificing average latency. EST was inferior to FAIR in the maximum latency of Parsing and Searching. This was a surprising result because the intuition was that the most sophisticated algorithm should achieve better performance. We found that this result was caused by the failure of the task time estimation due to the unexpected traffic patterns. We assumed that cars were generated randomly in the simulated area so that the data rate was too elusive to estimate with statistics. This indicated that EST had a natural limitation on the applicable services. Overall, PT achieved smaller latency for every application type compared to Separating, which meant our platform and the scheduler could handle the same workload with only 25/39 = 64% of the resources of the default platform.

We overloaded the cluster and evaluated the elasticity of schedulers. Figure 4 illustrates the number of records violating the SLA. The figure shows EDF and PT reduced the SLA violated records by more than one thousandth compared to FIFO. Even though FAIR achieved almost the same latency as PT and EDF, the number of SLA violated records was ten times greater. Compared to the default FIFO and FAIR, PT and EDF outperformed for elasticity of the workload.

4 Conclusion

We proposed a novel stream processing platform that improved resource efficiency by co-locating applications in one cluster. The main contributions of this work were designing an architecture that co-located multiple applications and evaluating the scheduling algorithms that handled the applications with heterogeneous SLAs and workloads. We implemented a platform prototype with three latency-aware schedulers and evaluated them with a realistic workload. The result shows that our platform reduced the necessary resources by 36%. The latency-aware schedulers outperformed the default schedulers FIFO and FAIR. We plan to develop a more sophisticated scheduling algorithm that fully treats the heterogeneous requirements of applications (e.g., task localization, heterogeneous data source, and memory allocation) in the future. We also plan to investigate the architecture in terms of security, fault tolerance, and operation.

5 Discussion

SLA Measurement: As we mentioned in section 2.2, we treated the task duration as the deadline criterion for each application. However, task duration does not reflect the actual latency imposed on records. In the real world, there are various elements that increase end-to-end latency, for example, waiting fetch in a messaging system and loading time of storage. Actually, in our environment, about 40% of the end-to-end latency was the queuing delay that occurred in the Kafka topic. The fact indicated that the measurement of task duration was not enough to determine the true deadline. One of the concrete solutions was to receive latency information from external systems. However, it was not realistic since the combinations of data source and sink were numerous, and the interface of the system was too complex. Although some prior studies have been proposed for estimating inside latency [14, 19], it is still challenging to estimate the real deadline with limited information.

Isolation: Isolation is a major tradeoff with resource efficiency. Our platform did not provide an isolated environment for each application; therefore, if an application fell into infinite loops, it caused non-negligible effects on the performance of other applications. This situation could be avoided by restricting the maximum amount of resources per application and the maximum duration of occupying resources. However, we need to carefully consider elaboration of the scheduler because a complicated algorithm causes an additional operational burden. An operational burden is one of the reasons that Mesos fine-grained mode is removed from Apache Spark [9]. (The main reason is that a fine-grained scheduler is not necessary for batch computing.) We plan to investigate the tradeoff of resource efficiency in the future.

Another problem caused by the absence of isolation is security risk. Because applications are executed on the same process, one application can refer other applications' data. That may cause security concerns, so we recommend to limit the applications run in the platform that belongs to the same team or department.

In spite of the disadvantage, we believe that the design still useful for the situation the same team will operate dozens of applications for one service and will want to save the platform cost. One example is the IoT service, it consists of many functions and the data sources, for example, parsing the multiple sensor data, then aggregating them, and finally detecting anomaly events. The platform must handle the bunch of sensor data, on the other hand, the requirement for the system durability is often not as severe as the other services (e.g., monitoring the security events of web servers requires severer durability than monitoring animals in a forest). In such a condition, to improve resource efficiency with sacrificing isolation is reasonable.

References

[1] 5gaa releases white paper on c-v2x use cases: Methodology, examples and service level requirements. https://5gaa.org/news/5gaa-releases-white-paper-on-c-v2x-use-cases-methodology-examples-and-service-level-requirements.

[2] Apache Flink. https://flink.apache.org/.

[3] Apache Kafka. https://kafka.apache.org/.

[4] Apache Mesos. http://mesos.apache.org/.

[5] Apache Spark: Job Scheduling. https://spark.apache.org/docs/latest/job-scheduling.html.

[6] Apache Storm. https://storm.apache.org/.

[7] Kubernetes. https://kubernetes.io/.

[8] PTV Vissim. https://www.ptvgroup.com/en/solutions/products/ptv-vissim.

[9] Remove mesos fine-grained mode subject to discussions. https://issues.apache.org/jira/browse/SPARK-11857.

[10] Spark Job Server. https://github.com/spark-jobserver/spark-jobserver.

[11] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[12] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer US, 2011.

[13] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM '16*, 2016.

[14] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *2011 IEEE 27th International Conference on Data Engineering*, pages 255–266, April 2011.

[15] D. Cheng, X. Zhou, Y. Wang, and C. Jiang. Adaptive scheduling parallel jobs with dynamic batching in spark streaming. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2672–2685, Dec 2018.

[16] Nihal Dindar, Nesime Tatbul, Renée Miller, Laura Haas, and Irina Botan. Modeling the execution semantics of stream processing engines with secret. *The VLDB Journal*, 22, 08 2013.

[17] G. Jacques-Silva, Z. Kalbarczyk, B. Gedik, H. Andrade, K. Wu, and R. K. Iyer. Modeling stream processing applications for dependability evaluation. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 430–441, June 2011.

[18] T. Jianchao, Y. Shuqiang, H. Chaoqiang, and Y. Zhou. Design and implementation of scheduling pool scheduling algorithm based on reuse of jobs in spark. In *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, pages 290–295, June 2016.

[19] T. Li, J. Tang, and J. Xu. A predictive scheduling framework for fast and distributed stream data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 333–338, Oct 2015.

[20] Anis Nasir, Gianmarco Morales, Nicolas Kourtellis, and Marco Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. pages 589–600, 05 2016.

[21] F. Pan, J. Xiong, Y. Shen, T. Wang, and D. Jiang. H-scheduler: Storage-aware task scheduling for heterogeneous-storage spark clusters. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–9, Dec 2018.

[22] J. Tang, M. Xu, S. Fu, and K. Huang. A scheduling optimization technique based on reuse in spark to defend against apt attack. *Tsinghua Science and Technology*, 23(5):550–560, Oct 2018.

[23] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Guolu Wang, Jungang Xu, Renfeng Liu, and Shanshan Huang. A hard real-time scheduler for spark on yarn. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 645–652. IEEE Press, 2018.

[25] Yuzhao Wang, Lele Li, You Wu, Junqing Yu, Zhibin Yu, and Xuehai Qian. Tpshare: A time-space sharing scheduling abstraction for shared cloud via vertical labels. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 499–512, New York, NY, USA, 2019. Association for Computing Machinery.

[26] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544, June 2014.

[27] L. Xu, A. R. Butt, S. Lim, and R. Kannan. A heterogeneity-aware task scheduler for spark. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 245–256, Sep. 2018.

[28] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496, New York, NY, USA, 2016. Association for Computing Machinery.

[29] J. Yu, H. Chen, and F. Hu. Sasm: Improving spark performance with adaptive skew mitigation. In *2015 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 102–107, Dec 2015.

[30] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[31] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 12(5):516–530, January 2019.