Q-Spark: QoS Aware Micro-batch Stream Processing System Using Spark

Suyeon Lee[†], Yeonwoo Jeong[‡], Minwoo Kim[‡] and Sungyong Park[*]
Department of Computer Science and Engineering, Sogang University
Seoul, South Korea
Email: [†]leesy0506@sogang.ac.kr, [‡]akssus12@sogang.ac.kr, [‡]joey@sogang.ac.kr, [*]parksy@sogang.ac.kr

***Abstract*—Unlike the event-driven stream processing systems, the micro-batch stream processing systems collect input data for a certain period of time before processing. This is because they focus on improving the throughput of the entire system rather than reducing the latency of each data. However, ingesting a continuous stream of data and its real-time analysis is also necessary in micro-batch stream processing systems where reducing the latency is more important than improving the throughput. This paper presents Q-Spark, a QoS (Quality of Service) aware micro-batch stream processing system that is implemented on Apache Spark. The main idea of Q-Spark design is to set a deadline time for each query and dynamically adjust the batch size so as not to exceed it. Since Q-Spark executes a micro-batch by buffering as much as possible until the deadline set for each query is exceeded, it guarantees the QoS requirement of each query while maintaining the throughput as much as the original Spark batching mechanism. Experimental results show that the tail latency of Q-Spark is always bound to the deadline compared to the original Spark where data is buffered using triggers for a certain period. As a result, Q-Spark reduces the tail latency per query by up to 75%, while maintaining the throughput stably compared to the original Spark without the concept of a deadline.**

***Keywords*—Micro-batch Stream Processing, Spark, QoS, Admission Control**

I. Introduction

[1] As we enter the big-data era, not only is the amount of new data generated increasing, but its cycle is also getting shorter. Advances in IoT (Internet of Things) and embedded sensor technologies have enabled the collection of vast amounts of data generated in everyday life, unconstrained by time and space, emphasizing the need to process these data in real-time. As a result, a stream processing system has recently been attracting attention.

Stream processing systems can be largely categorized into two mechanisms: *event-driven* and *micro-batch* [1]. The event-driven mechanism processes each data immediately whenever it occurs. Pipelining allows the system to process different data across multiple operations immediately, which optimizes the latency of each data. The micro-batch mechanism buffers real-time data for a certain period and processes it in a small batch unit, which improves throughput at the cost of latency. Recently, micro-batch stream processing systems such as Apache Spark [2] [3] have been widely used to optimize the overall throughput. However, in micro-batch stream processing systems, there are several cases where data continuously comes in at short intervals, and applications require real-time analysis results for the data. Accordingly, it is not always appropriate to lower the priority of latency and focus only on throughput in stream processing systems. Spark uses the *trigger* concept to ensure that processing is performed periodically in a certain interval of time. Since the trigger value is a static value set by the user, it often cannot become the boundary value of latency in fluid traffic. For example, if the processing time becomes longer than the trigger time due to bursty input data, the amount of data waiting increases, forming a faulty cycle. From this point on, the latency of each data gradually increases without an upper limit.

To solve this problem, this paper proposes Q-Spark, a QoS (Quality of Service) aware micro-batch stream processing system that processes the incoming stream without violating the QoS for each query. The QoS in Q-Spark is a deadline time that is set by the user for each query. Deadline values are not applied to a single execution unit (i.e., micro-batch), but to the individual data that constitutes the unit. In the event-driven systems, data immediately enters the processing phase as soon as its generation. Thus, the deadline concept can be easily applied to each data. On the other hand, in the micro-batch systems, the data first goes through a buffering phase for a constant period of time to form a single execution unit, and then the data is sent to the processing phase. In this case, latency cannot be defined solely by the processing time of individual data in micro-batch stream systems. In other words, within a single micro-batch, the latency depends on the event time of each data. Q-Spark is designed to ensure that the system can handle data with tail latency within the deadline time.

Q-Spark is implemented on top of Apache Spark, which supports micro-batch multi-stream processing. Q-Spark has a micro-batch constructor module that replaces the trigger concept used in the original Spark. This module constructs a temporary micro-batch as soon as new data to be processed for each query is generated. Q-Spark also has an admission controller module that decides whether the micro-batch generated per query can be processed or not. Considering the tail latency of the data in a micro-batch, the admission controller module permits processing if the micro-batch has an imminent deadline. Otherwise, it aborts processing and signals the micro-batch constructor. In this case, the micro-batch constructor buffers more data and creates a new micro-batch. That is, Q-Spark is a multi-stream processing system

[1]This work was supported by IITP grant funded by MSIT (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System). (Corresponding author: Sungyong Park)

that collects and processes as much data as possible within the range where all individual data does not violate the deadline.

Experimental results show that `Q-Spark` improves the tail latency per query up to 75%, while maintaining the throughput stably compared to the original Spark without a separate QoS concept. Also, `Q-Spark` always bound the tail latency to the user-specified deadline, which solves the critical problem of increasing latency without an upper limit.

To summarize, this paper makes the following specific contributions.

- `Q-Spark` is the first attempt to introduce a deadline concept to ensure QoS of each query in micro-batch streaming processing system.
- `Q-Spark` proposes a simplified dynamic batching mechanism to reduce QoS violation per streaming query. This is a general-purpose mechanism that can be applied to any framework that uses the micro-batch model, since the scheduling modules are not modified.
- `Q-Spark` solves the problem that the processing time of multi-queries in a single streaming application increases non-linearly on the original Spark.

II. BACKGROUND

A. Micro-batch Stream Processing

Stream processing can be performed in two models: event-driven model and micro-batch model. Event-driven model prepares data by creating a topology or DAG (Directed Acyclic Graph) in the form of a pipeline before data ingestion. When data is generated from the input source, it is processed immediately without buffering or local storage, ensuring low latency. Since the latency in the event-driven model includes only processing time, a deadline can be easily applied.

As the need for analyzing data generated over a certain period of time increases, a micro-batch model that collects and processes a certain amount of streaming data at once has emerged to improve throughput. The processing step of micro-batch streaming processing system is divided into buffering phase and processing phase. In the buffering phase, data ingested in real-time is aggregated for a certain period to construct a single execution unit called micro-batch. The micro-batch is then fed to the processing phase for further processing.

Apache Spark, a representative framework that supports the micro-batch model, uses *trigger* [3] to allow users to determine the execution cycle by setting an appropriate value. After the system starts processing a specific micro-batch, it begins to process subsequent micro-batch only after the trigger time. If micro-batch execution is completed before the trigger time, the system performs buffering for the remaining time. The trigger value cannot be changed at runtime. If the trigger value is 0, a new micro-batch is created as soon as the previous micro-batch is processed.

B. Related Works

Event-driven QoS-aware stream processing Hoseiny et al. [4] proposes an event-driven adaptive feedback controller system to consider QoS for latency-sensitive streaming applications. It satisfies the QoS requirements of each query by predicting the load intensity of each workload in the sampling phase and responding quickly to the workload fluctuations. Wang et al. [5] suggests a priority-based resource allocation strategy to ensure QoS requirements of real-time streaming applications. It solves a multi-objective optimization problem to minimize the sum of QoS violations across all applications and maximize the CPU utilization of the node on which the application is deployed. Hoseiny et al. [6] proposes a dynamic resource control mechanism based on the consideration of traffic fluctuations that satisfies the QoS required for each query, reduces query response time, and maximizes resource utilization.

Micro-batch QoS-aware stream processing Das et al. [7] proposes an adaptive online-based dynamic batching algorithm that considers workload patterns and data ingestion rates in order to solve QoS violations of queries processing streaming data. It dynamically adjusts the micro-batch size so that the system performance is stable by continuously monitoring the input rate and the throughput of the streaming processing engine. Although this dynamically adjusts the batch size to satisfy the QoS for each query, it does not consider the environment where multiple queries are performed in a single application and the deadline for each query. Cheng et al. [8] designs an adaptive query scheduling mechanism to maximize resource efficiency and system performance in an environment where multiple queries are executed in parallel in a single streaming application. In addition, this study devises a dynamic micro-batch interval control algorithm considering the workload fluctuation and input speed. Most studies considering QoS in the micro-batch model focus on scheduling queries to match the processing capacity of the processing engine for each framework. On the other hand, our proposed approach sets a deadline for each query in the buffering phase and fetches data up to the guaranteed line. To the best of our knowledge, this is the first micro-batch stream processing system applying the deadline concept among multi-queries in a single application.

III. MOTIVATION

Streaming applications based on the micro-batch model fetch data from the input source by calling the trigger at regular intervals. Considering that the trigger value is statically determined by the user and does not change during entire executions, this method raises two main issues.

First, the user does not know the processing capacity of the entire system precisely. This makes it difficult for the user to set an optimal trigger value considering both input rate and processing rate at the same time. Second, in streaming workloads, the input rate changes in real-time [4]. As a consequence, the fixed polling interval is likely to create a micro-batch that puts a load on the system. If the size of a single micro-batch becomes too large, the time required for processing phase increases. In the meantime, since real-time data continues to flow in, the size of the subsequent

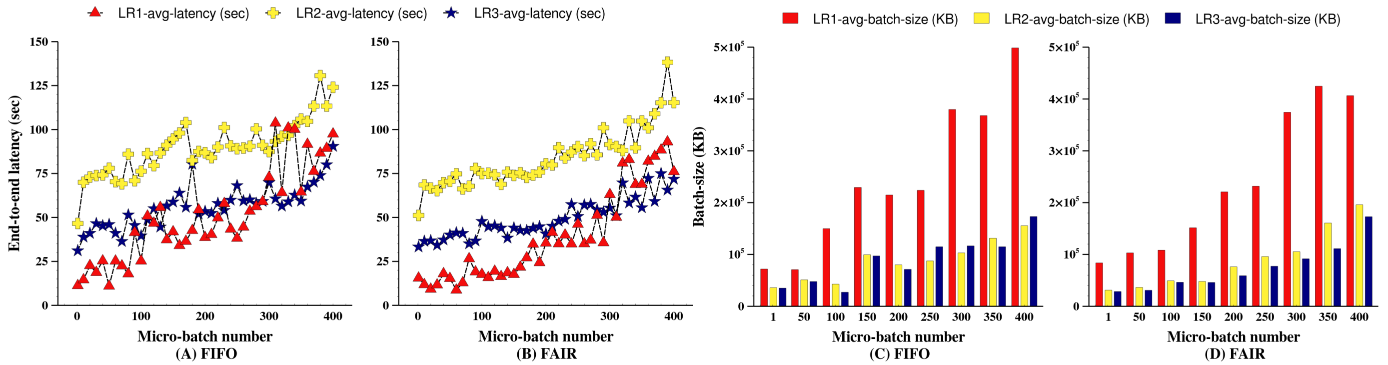

Figure 1: (A) and (B) are end-to-end latency graphs for each micro-batch for each scheduling algorithm. (C) and (D) are batch size graphs for each micro-batch for each scheduling algorithm.

micro-batch further increases. This faulty cycle eventually degrades the performance of the system and makes the stream processing unstable.

To confirm this, we have measured the end-to-end latency and the size of processed data for each micro-batch in the original Spark where we ran three most widely used queries [9] simultaneously, noted as LR1, LR2, and LR3. We used the same experimental setup and workloads explained in Chapter V. In this paper, the end-to-end latency is the sum of the times required both at buffering phase and processing phase for individual data.

Figure 1 shows the performance results over the original Spark configured with two main scheduling algorithms: FIFO and FAIR. We can observe in Figure 1 that the end-to-end latency of three queries increases almost linearly as we increase the batch size, regardless of the scheduling algorithms used. This indicates that if the stream processing engine overlooks the micro-batch size, the performance of the stream processing engine continues to deteriorate. The main reason for this is that the original Spark aggregates all newly created data during buffering phase as a single micro-batch. Therefore, there is a high possibility that the original Spark ingests an immense amount of data by bursty traffic, which causes the streaming engine to spend more time on processing than the previous micro-batch. Since traffic continues to occur while processing for a more extended period, subsequent micro-batch must process more data, and the end-to-end latency continues to increase.

To cope with this problem, this paper introduces the concept of a deadline. Since the trigger value in the original Spark determines the execution cycle, it becomes a lower bound value in terms of latency. However, what is needed to guarantee QoS in real-time systems is to secure the upper boundary of latency. Therefore, designing a streaming system that receives the user-specified deadline value and operates not to violate it is essential.

IV. Design

A. Definition of QoS in Micro-batch Processing

In an event-driven model, the execution unit is individual data. As soon as data is generated, the stream processing engine proceeds processing through multiple pipelines. Because the latency of each data in the event-driven model is equal to the processing time, it is easy to apply a deadline to each data to ensure QoS. In contrast, the execution unit in a micro-batch model is created by buffering incoming data from the input source in real-time. Buffered data is passed to the processing phase simultaneously, distributed to each executor, and processed in parallel. Since the processing time per each micro-batch and the latency of individual data are different, applying deadline is not easy. Also, if the deadline is applied based on the average latency of data included in single micro-batch, there may be data with tail latency that exceeds the query deadline. In this case, it is not appropriate to state that the system meets the user-specified deadline.

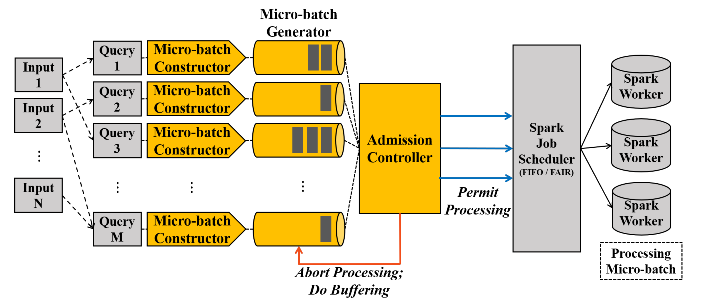

Figure 2: Yellow figures indicate system modules newly implemented in `Q-Spark`.

In this paper, we define QoS as the deadline time applied to individual data processed in the micro-batch stream processing system. Before starting a streaming application, users can set a deadline time for processing individual data in each query. For example, if you set a deadline of 5 seconds as an option for stream query A, the latency of individual data for query A cannot exceed 5 seconds, regardless of which micro-batch the query belongs to. In this case, the latency of each data is the sum of the buffering time waiting for micro-batch creation and the processing time it takes for the micro-batch to be processed.

B. Overview of `Q-Spark`

Figure 2 shows the overall architecture of `Q-Spark` and yellow boxes indicate new modules implemented over original

Table I: Parameters of cost models and rules. All parameters are visible through entire systems.

Type	Notation	Description
Specified by User	$Deadline_q$	Required deadline time specified per query q in user application.
Used in System	$TailLatency_{(q,i)}$	Tail latency of dataset within micro-batch i of query q.
	$EstimatedRunTime_{(q,i)}$	Estimated runtime of micro-batch i of query q.
	$BatchSize_{(q,i)}$	Size of micro-batch i of query q.
	$MaxThrouhgput_q$	Maximum throughput of query q.
	$L_{throughput}$	Loss of throughput depend on scheduling mode.
	$L_{latency}$	Loss of latency depend on scheduling mode.
	$NumActive$	Number of active queries (jobs) in a Spark scheduler. (When scheduling mode is FAIR)
	$NumWaiting$	Number of waiting queries (jobs) in a Spark scheduler. (When scheduling mode is FIFO)

Spark.

As shown in Figure 2, multiple stream queries can be executed simultaneously in a streaming application, and each query can receive data from different input sources. `Q-Spark` dynamically adjusts the batch size by applying a QoS deadline to the micro-batch to be executed per query instead of the existing trigger function. First, the *micro-batch constructor module* that operates for each query periodically polls the input source and creates a temporary micro-batch as soon as new data is created. After that, the *admission controller* receives the micro-batch and the deadline of corresponding query, and decides whether it can process the micro-batch based on the cost model. If the admission controller permits processing, it confirms the micro-batch and passes the query job to the Spark scheduler for execution. If the admission controller aborts processing, it cancels the micro-batch and proceeds the admission review of another query job. Cancelled micro-batch data is queued to the *micro-batch generator module* for buffering. During this process, the latency of the previously queued data continues to increase. Therefore, even if new data from the previous one does not arrive during the buffering phase, it re-requests processing from the admission controller at regular intervals to satisfy the deadline.

C. Admission Control Mechanism

Algorithm 1: Admission Control Mechanism

```
1  Def AdmissionControl(microBatch, D):
2      BatchSize_(q,i) = sizeof(microBatch)
3      Get TailLatency_(q,i) // Eq (1)
4      Compute EstimatedRunTime_(q,i) // Eq (2)
5      if TailLantecy_(q,i) +
         EstimatedRunTime_(q,i) ≥ Deadline_q then
           // Admit processing
6          numBuffered = 0
7          return (True, microBatch)
       // Abort Processing; Do buffering
8      Set #files in microBatch as numBuffered
9      BatchSize_(q,i) = 0
10     return (False, ∅)
```

This section describes the admission control mechanism and the cost models used throughout this paper. The detailed specifications of system parameters are summarized in Table I.

A user-defined deadline $Deadline_q$ applies to individual data within a micro-batch of a query q. Buffering is performed as much as possible in the range where the sum of latency $TailLatency_{(q,i)}$ and estimated runtime $EstimatedRunTime_{(q,i)}$ of the oldest data in a specific micro-batch i of the query does not violate the deadline.

Algorithm 1 shows the QoS-aware dynamic admission control mechanism in detail. In Algorithm 1, the tail latency of the data in the i-th micro-batch in the query q can be defined as Equation 1. Moreover, the estimated runtime of the micro-batch is defined as Equation 2.

$$TailLatency_{(q,i)} = currentTime - oldestDataCreationTime \quad (1)$$

$$EstimatedRunTime_{(q,i)} = \frac{BatchSize_{(q,i)}}{(MaxThroughput_q * L_{throughput})} + L_{latency} \quad (2)$$

In Equation 2, $MaxThroughput_q$ is defined as the maximum throughput of the query and is the value when the query is executed alone by occupying all resources. As a result, the execution time can be estimated by dividing $BatchSize_{(q,i)}$(i.e., the size of the data to be processed) by $MaxThroughput_q$. However, additional overhead can be occurred depending on the query scheduling algorithm in a multi-query environment. If the scheduling algorithm is FAIR, the query can be executed immediately, but resources must be shared equally with other running queries. Therefore, the throughput degrades more than $MaxThroughput_q$. On the other hand, if the scheduling algorithm is FIFO, the query q is executed by occupying all resources after other queries are terminated. Since this causes a scheduling delay that waits until the query is executed in the processing phase, $EstimatedRunTime_{(q,i)}$ increases accordingly. According to the scheduling algorithm s, the loss in throughput and latency can be determined as Equation 3 and Equation 4.

$$L_{throughput} = \begin{cases} 1, & \text{if } s \text{ is FIFO} \\ \frac{1}{NumActive}, & \text{if } s \text{ is FAIR} \end{cases} \quad (3)$$

$$L_{latency} = \begin{cases} \sum_{i=0}^{NumWaiting} \frac{BatchSize_{(q,i)}}{MaxThroughput_q}, & \text{if } s \text{ is FIFO} \\ 0, & \text{if } s \text{ is FAIR} \end{cases} \quad (4)$$

Based on Algorithm 1, data buffering is performed until the sum of the values of $TailLatency_{(q,i)}$ and $EstimatedRunTime_{(q,i)}$ does not exceed $Deadline_q$.

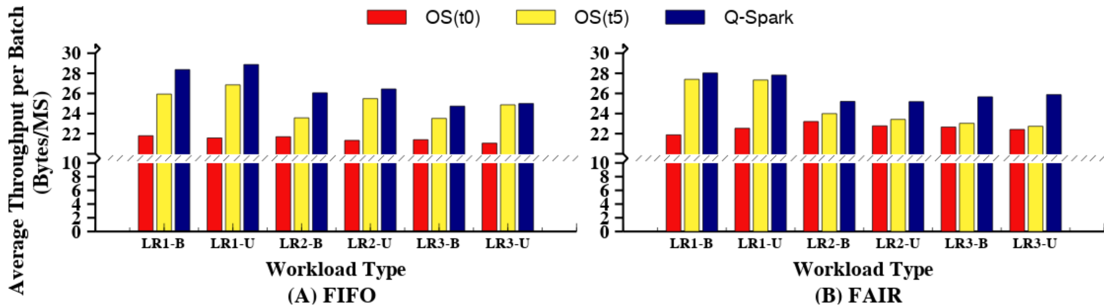

Figure 3: Average throughput per batch by each workload during continuous execution.

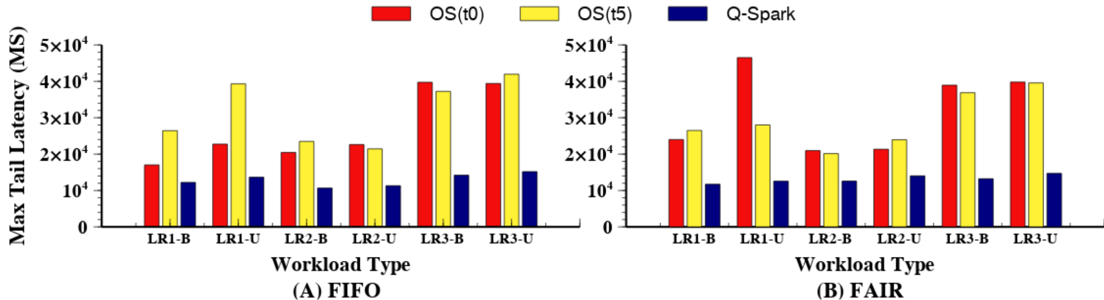

Figure 4: Max tail latency by each workload during continuous execution. The deadline is set to 10 seconds.

V. Evaluation

A. Experimental Setup

For the experiment, we configured Spark cluster consisting of one master node and two worker nodes. All experiments are performed on a server with Intel Xeon Silver 4210 2.20GHz CPUs with 10 physical CPU cores and 128 GB of memory. All nodes are interconnected by 10 Gbps Ethernet. We set 2 executors per node with 12 CPU cores and 48GB of memory.

B. Workloads and Comparison Target

The workload used in the experiment is a real-world streaming workload, Linear Road Benchmark [10] that includes various query operations, such as filter, project, aggregate, and join. We used query 2,3,4 [9], which are frequently used in previous works. Each query notation is LR1, LR2, LR3. In addition, we used two types of workload traffic, and the detailed description is as follows.

- B(300): A constant 300 records are generated every second.
- U(300): A random record is generated every second so that its average converges to 300 records per second.

In order to show the effectiveness of our approach, an overall system performance was compared with the original Apache Spark [3]. In the original Spark, the trigger value is set to 0 (OS(t0)) or 5 (OS(t5)) seconds. In contrast, we set the deadline to be 10 seconds in `Q-Spark`. The deadline value of 10 seconds is assumed to be sufficient for LR1 and tight for LR2 and LR3.

C. Overall Performance

In this section, we compared the overall performance during the entire execution of applications. We ran each workload for 30 minutes and LR1, LR2, and LR3 were performed simultaneously in the same traffic for the multi-query environment.

Figure 3 shows the average throughput obtained immediately after completing micro-batch in each platform. In this paper, we defined the throughput, which is calculated by the size of data processed in each micro-batch, as the processing time of the micro-batch. Looking at the results, the average throughput of OS(t0), which does not proceed with buffering at all, is the lowest, followed by the average throughput of OS(t5) and `Q-Spark` in that order. In the case of `Q-Spark`, the deadline is 10 seconds, but it does not mean it buffers for 10 seconds. Considering the situation where multiple other queries are running or waiting for scheduling, the admission controller adjusts each query to meet the deadline. Therefore, the buffering time is different from each query and micro-batch set, depending on the system situation. In addition, the deadline time contains the time taken during processing phase, which means `Q-Spark` sometimes buffers less period than OS(t5). Therefore, there may not be a big difference in average throughput with the OS(t5), which performs regular buffering for 5 seconds each time.

Figure 4 shows the maximum tail latency values obtained from each micro-batch. Considering the maximum tail latency of each system together, we can see that the overall performance of `Q-Spark` outperforms other approaches. Since the latency of individual data constantly increases in the original Spark, the maximum value appears relatively high. However, in `Q-Spark`, the average latency of each data converges to the deadline. As a result, the tail latency is reduced by up to 65% when the scheduling algorithm is FIFO and up to 73% when the scheduling algorithm is FAIR. In particular, in the original Spark, the overhead imposed by the scheduling algorithm is directly related to data latency. For example, if the scheduling algorithm is FIFO, the size of a single micro-batch to be processed by each query gradually increases. As the processing time of a specific query increases, the waiting

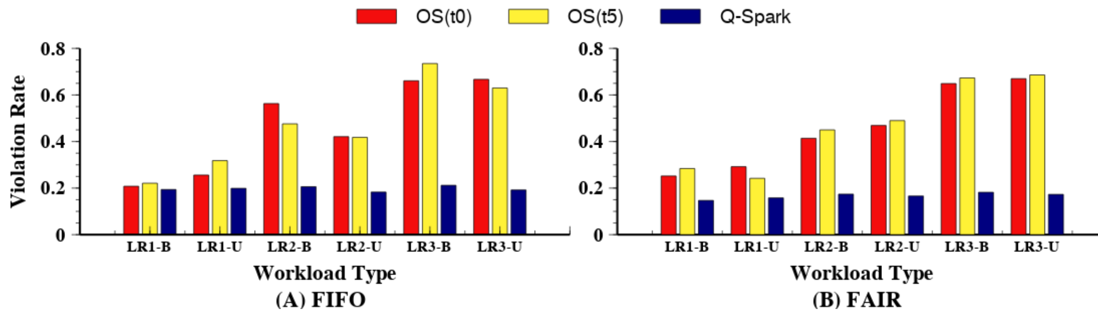

Figure 5: Violation rate for each workload during continuous execution. The deadline is set to 10 seconds.

time for other queries increases as well. In the meantime, the latency of each data in the micro-batch of the query continues to increase accordingly. When the scheduling algorithm is FAIR, all queries share the system resources. Therefore, the processing time becomes longer and generates high latency to process the data. In contrast, Q-Spark shows constant tail latency regardless of the scheduling algorithm.

D. Deadline Violation Analysis

In this section, we compare the deadline violation rate of Q-Spark with the original Spark. In the case of the original Spark, we treat that the deadline violation happens if the latency exceeds 10 seconds (i.e., deadline in Q-Spark). Figure 5 shows the ratio of the number of datasets with the latency of 10 seconds or more among all datasets. Regardless of the scheduling algorithm, the violation rate of Q-Spark is the lowest among all systems in common. In particular, in OS(t0) and OS(t5), LR2 and LR3, which have longer unit processing times than LR1, have much higher violation rates. As mentioned in Section V-C, a query with a long processing time affects the latency of other queries according to the scheduling algorithm. For example, if the scheduling mode is FIFO, the data latency in other waiting queries keeps increasing. However, Q-Spark shows a violation rate below a certain value regardless of the workloads.

In the case of LR1-B in Figure 5-(A), the difference between the violation rate measured in Q-Spark and other systems is not significant. The rate calculated by simple counts may be similar, but there is a big difference in the degree and pattern of violation. Also, as mentioned in Section III, the size of the processed data and tail latency is continuously increasing in the case of the original Spark. Therefore, over time, the degree of violation will increase significantly, and this trend continues. On the other hand, in Q-Spark, the phenomenon does not persist. The degree of violation is much smaller than the original Spark, and the tail latency value quickly recovers below the deadline value. The results show that Q-Spark effectively manages the latency of streaming data by introducing the concept of deadline compared to the original Spark.

VI. Conclusion

In this paper, we have proposed a QoS-aware micro-batch stream processing system called Q-Spark that supports a dynamic batching mechanism for the streaming applications. Q-Spark introduces a new QoS concept in micro-batch multi-stream processing systems. Q-Spark meets the user-specified deadline for individual data by dynamically adjusting the batch size. In addition, Q-Spark aggregates incoming data as much as possible within the range that does not violate the deadline, which allows us to maximize throughput. In particular, Q-Spark improves the micro-batch decision logic without modifying the core system so that it can be generally applied to various micro-batch model frameworks other than Spark. Through experiments, we have shown that the proposed mechanism is effective for minimizing QoS violation per streaming queries while maintaining system throughput.

References

[1] G. Van Dongen and D. Van den Poel, “Evaluation of stream processing frameworks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020.

[2] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

[3] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, “Structured streaming: A declarative api for real-time applications in apache spark,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 601–613.

[4] M. R. Hoseiny Farahabady, A. Jannesari, J. Taheri, W. Bao, A. Y. Zomaya, and Z. Tari, “Q-flink: A qos-aware controller for apache flink,” in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 629–638.

[5] Y. Wang, Z. Tari, M. R. HoseinyFarahabady, and A. Y. Zomaya, “Qos-aware resource allocation for stream processing engines using priority channels,” in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, 2017, pp. 1–9.

[6] M. R. Hoseiny Farahabady, H. R. Dehghani Samani, Y. Wang, A. Y. Zomaya, and Z. Tari, “A qos-aware controller for apache storm,” in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, 2016, pp. 334–342.

[7] T. Das, Y. Zhong, I. Stoica, and S. Shenker, “Adaptive stream processing using dynamic batch sizing,” 2014, p. 1–13.

[8] 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*, vol. 29, no. 12, pp. 2672–2685, 2018.

[9] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, “Saber: Window-based hybrid stream processing for heterogeneous architectures,” in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 555–569.

[10] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, “Linear road: a stream data management benchmark,” in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 480–491.