Cluster Computing
https://doi.org/10.1007/s10586-022-03758-1

zStream: towards a low latency micro-batch streaming system

Suyeon Lee[1] · Yeonwoo Jeong[1] · Kyuli Park[1] · Gyeonghwan Jung[1] · Sungyong Park[1]

Received: 19 April 2022 / Revised: 29 July 2022 / Accepted: 18 September 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Conventional micro-batch streaming systems introduce high tail latency for the following two reasons. First, they trigger data processing on a fixed time interval, without taking into account the data ingestion rate and data analysis processing rate. Second, in a multi-query environment, their scheduling methods do not consider the real-time characteristics of the application and each query. This paper presents zStream, a low latency micro-batch streaming system implemented on Apache Spark. The main idea of zStream's design is to use a reference value called *deadline* for each query to constrain latency around a specific time. Considering the deadline information of each query, zStream controls the admission of every micro-batch dynamically and schedules them by their priorities, significantly reducing the average and tail latency while maintaining throughput. Experimental results show that the tail latency of zStream is reduced by a maximum of 34% compared to the original Spark. Also, zStream reduces the average latency per query by up to 48% while stably maintaining throughput compared to the original Spark.

Keywords Micro-batch streaming system · Admission control · Scheduling · Spark

1 Introduction

As we enter the big data era, not only is the amount of new data generated increasing, but its cycle is also getting shorter. Large-scale industries such as smart factories, embedded sensor technologies, and web service providers generate vast amounts of data in everyday life, unconstrained by time and space. These applications emphasize the need for streaming systems to process these explosively generated data in real-time. The fundamental goal of stream processing is to deliver valuable knowledge to users in the shortest possible time [2–4]. In addition, applications such as periodic Internet-of-Things (IoT) analysis even have specific latency requirements (e.g., deadline) for practical purposes [5]. Therefore, streaming systems need to provide low latency services by minimizing both average and tail latency of each dataset[1].

Among various streaming systems, micro-batch model-based frameworks [6–10] are gaining popularity due to their fine performance, allowing high throughput while providing considerably acceptable latency. The micro-batch streaming system first buffers one or more datasets for a certain period to form a single micro-batch (i.e., a minimal processing unit) per query. Then, it uses basic scheduling algorithms (e.g., FIFO/FAIR) to execute each micro-batch of multiple queries. To decide a specific period interval, the system expects users to set a fixed time value called *trigger* to buffer datasets. This conventional

Suyeon Lee and Yeonwoo Jeong have contributed equally to this work.

A preliminary version of this article [1] was presented at the 2021 IEEE 1st International Workshops on Autonomic Computing and Self-organizing Systems (ACSOS), Washington DC, USA, September, 2021.

✉ Sungyong Park
parksy@sogang.ac.kr

Suyeon Lee
leesy0506@sogang.ac.kr

Yeonwoo Jeong
akssus12@sogang.ac.kr

Kyuli Park
kyuli0909@sogang.ac.kr

Gyeonghwan Jung
siblue202@sogang.ac.kr

[1] Department of Computer Science and Engineering, Sogang University, 35, Baekbeom-ro, Mapo-gu, Seoul, Republic of Korea

[1] A *dataset* refers to a minimum unit of data, which is a collection of raw records (e.g., single file).

mechanism allows improved throughput at the minimal cost of latency. However, it has two fundamental problems that result in high tail latency of each dataset. This high tail latency degrades the overall system's performance more and more as time goes by, and eventually prevents the system from functioning as a desirable streaming system.

The first problem is that conventional systems use static buffering time, which means the trigger time interval is immutable during runtime. It becomes a problem when the query processing rate becomes lower than the input data arrival rate. Processing delays lead to increased buffering times, which again leads to increased processing times. These endless cycles increase the latency of individual datasets, which is the sum of buffering time and processing time. The second problem is that existing systems only use basic scheduling algorithms that do not consider application characteristics. For example, the FIFO scheduler cannot cope with specific types of delaying queries, and this causes considerable latency. The FAIR scheduler does not have a policy for adaptive resource allocation among queries, which is crucial for preventing the latency of overall queries. Both schedulers are unaware of the user's latency requirements and are not responsive to input data fluctuations.

To solve the tail latency problem mentioned above, this paper proposes zStream, a deadline-aware micro-batch streaming system that significantly reduces tail latency by minimizing the latency of each dataset. Our main idea is to use the *deadline*[2] concept as a reference value for constraining the system to maintain each dataset's latency around a specific time value. Instead of using a static buffering time option (i.e., trigger), zStream uses a deadline time option, which is the same as the trigger time value per query, and applies it to every dataset of various queries. For example, if the deadline time is 5 s for query Q_1, it means the system needs to process all datasets executing Q_1 in around 5 s. It not only helps the system to maintain latency near the specific value, but it is also much more intuitive for users to state their latency requirements.

zStream uses two main modules to reduce tail latency: (i) *admission controller* and (ii) *priority-based scheduler*.

Admission controller module Instead of the existing static buffering method, this module uses a dynamic buffering method that controls the trigger time interval in real-time. While monitoring the input data arrival rate and query processing rate, if an individual dataset's latency is within the allowable range, the module increases the trigger time to allow more datasets to be buffered, thereby improving throughput. Otherwise, it reduces the trigger time so that the latency of individual dataset can be maintained within a specific range, leading to the prevention of high tail latency.

Priority-based scheduler module This module schedules all datasets considering application characteristics so that the deadline specified by the user for each query can be satisfied. This method effectively prevents high tail latency from occurring because it allows latency to be maintained at a certain level. This module includes two scheduling policies such as *Earliest-Deadline-First* (*EDF*) policy and *Resource-Deadline-Aware-Scheduling* (*RDAS*) policy. In both policies, the module uses a priority in the order of the shortest remaining time until the deadline. The EDF policy, which improves the FIFO policy, rearranges the execution order in real-time so that a micro-batch with high priority can be executed first. The RDAS policy, which improves the FAIR policy, redistributes resources in real-time so that more resources can be allocated to a micro-batch with high priority.

We implemented zStream on top of Apache Spark Structured Streaming [7], which is a representative framework supporting a micro-batch streaming system. Experimental results show that zStream reduces the tail latency per query up to a maximum of 34% compared to the original Spark. Furthermore, zStream reduces the average latency per query by up to 48% while showing slight increased throughput compared to the original Spark.

To summarize, this paper makes the following specific contributions.

- zStream proposed a deadline-aware dynamic buffering method in the micro-batch streaming system, which effectively reduces the latency of individual dataset.
- zStream introduced two deadline-aware scheduling policies in the micro-batch streaming system, which minimizes the latency of each dataset per query in a multi-query environment.
- We implemented a working system on Apache Spark and demonstrated its effectiveness with real-world stream processing benchmarks.

The rest of this paper is organized as follows. Section 2 briefly explains the background of the micro-batch stream model in Apache Spark. Section 3 discusses our motivation which reveals the fundamental problem of the micro-batch stream model. Section 4 presents the overview of zStream, followed by Sect. 5, which explains its design and implementation details. Section 6 describes the experimental setup and presents the results of the evaluations. Section 7 provides an overview of the related studies. Section 8 concludes this paper.

[2] A *deadline* indicates a reference value to sustain latency of dataset.

2 Background

In this section, we explain the background of Apache Spark [11] and its micro-batch stream model. The background is limited to the Spark framework's context, but the details are also applicable to other micro-batch streaming systems.

2.1 Apache spark

Apache Spark is a general-purpose distributed in-memory data processing platform. It provides an in-memory data structure (i.e., *RDD* [12]), an immutable distributed collection of objects partitioned in distributed memory across cluster nodes. Spark processes datasets in parallel by partitioning and distributing them through query planning and task scheduling. For parallel processing of queries, a user first writes a Spark program by calling query operations as a series of RDD functions shown in Code 1.

```
var spark =
    SparkSession.builder...getOrCreate()

/* make RDD reading files */
var fileInputRDD = spark.readStream
                .schema()
                .csv("file_path")

// RDD transformations
var Query1 = fileInputRDD
       .flatMap(_.split("."))
      .filter{_ => _.startsWith("s")}
      .reduceByKey(_ + _)
      .count()
```

Code 1: Spark program with RDD transformations.

The user submits the program to *SparkDriver* to execute queries. SparkDriver analyzes each query logic and builds a logical operation flow, represented as a Directed Acyclic Graph (DAG). Then, it submits the DAG to *DAGScheduler* to determine optimal query operation orders and *DAGScheduler* in turn divides the whole DAG into multiple stages according to the data dependency. Each stage is a set of parallel tasks to be scheduled. Finally, DAGScheduler submits a set of tasks to *TaskScheduler* to schedule them with the FIFO or FAIR scheduling policy.

The FIFO scheduling method sequentially prioritizes queries in the order of arrival in a single queue. The query with the highest priority occupies all available computing resources in the cluster. On the other hand, the FAIR scheduling method allows queries to be executed with fair shares of available computing resources. In addition, it supports the pool concept, which is a queue that groups one or more queries. It is up to users to define pools and designate several options per pool, such as the number of minimum resources and the second-step scheduling method. For example, suppose that there is a specific pool with two queries given a minimum of 6 CPU cores, and the second-step scheduling method is FAIR. In this case, each query in the pool will run concurrently with at least 3 CPU cores, while other queries will run with a different number of cores depending on their pool options.

2.2 Micro-batch stream model

Apache Spark supports two micro-batch streaming systems such as Spark Streaming [6] and Structured Streaming [7]. These systems buffer real-time data for a certain period and process them in small batch units (i.e., micro-batch), which improves throughput at the cost of latency. Before query execution starts, the system goes through buffering and scheduling phases. During the buffering phase, it collects data for a specific time interval that users set in their applications with the *trigger* option. If the trigger value is 0, the system creates a new micro-batch as soon as it completes the previous micro-batch. In this case, the buffering phase of the current micro-batch will be the same as the previous query execution time. Once a micro-batch of a query is formed, the system enters the scheduling phase to determine the query execution order of partitions of the micro-batch.

Figure 1 describes the detailed steps of buffering and scheduling phases in Spark. At the front end of the streaming systems, message brokers (e.g., Apache Kafka [13]) periodically aggregate real-time raw data produced from IoT/edge devices into a dataset. In the buffering phase, new datasets are buffered into the streaming system until they are loaded for query execution in every trigger cycle. To retrieve data from the message broker without data loss, the micro-batch system uses an offset information completed up to the last micro-batch execution. When the trigger is initiated (①), SparkDriver calculates the start offset using the information committed to the checkpoint. Then, it creates a micro-batch (②) by fetching all datasets buffered after the start offset and commits the updated offset information in the checkpoint. Finally, SparkDriver transfers the created micro-batch and query information to the scheduling phase.

In the scheduling phase, DAGScheduler analyzes the query operation orders and data dependencies to make a DAG (③) and divides it into several stages. Each stage is a set of parallel tasks, the smallest execution units in Spark. Finally, TaskScheduler maps each task to the CPU core and determines the execution order according to scheduling policy such as FIFO or FAIR (④). The buffering phase for subsequent micro-batches occurs concurrently with the scheduling phase and final query execution. It continues until the next trigger occurs regardless of the completion time of the previous micro-batch.

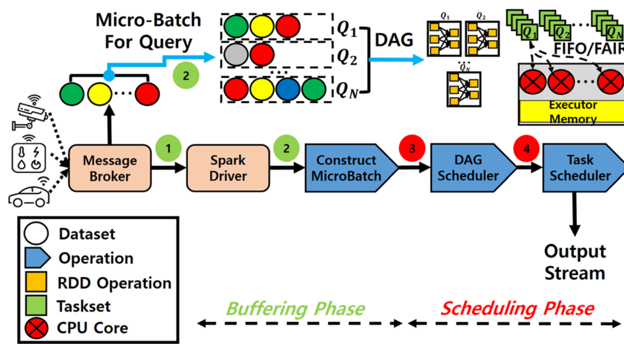


Fig. 1 Buffering and scheduling phases before query executions in Spark

3 Motivation

This section presents a fundamental tail latency problem in the micro-batch streaming system and reveals the primary causes. For this, we ran three queries of the Linear Road streaming benchmark [14] (we call them LR1, LR2, LR3) on the Apache Spark cluster. We have described experimental settings and query details in Sect. 6.1. The trigger time interval is set to 5 s.

We obtained the cumulative distribution function of maximum dataset latency per micro-batch, and the correlation between latency and micro-batch data size for each scheduling policy (i.e., FIFO and FAIR policies). Figure 2 shows the results for LR1 in the case of using FIFO scheduling, and Fig. 3 shows the results for the same query in the case of using FAIR scheduling. We only present the results for LR1 because the trend of experimental results for other queries is similar. Our results clearly show that high tail latency occurs regardless of scheduling policy. In the left-side graph of Fig. 2, the 95th percentile latency is around 27.4 s, which is about 3.1$\times$ of 50th percentile latency (i.e., around 8.7 s). In the left-side graph of Fig. 3, the 95th percentile latency is around 22.2 s, which is about 2.8$\times$ of 50th percentile latency (i.e., around 7.8 s).

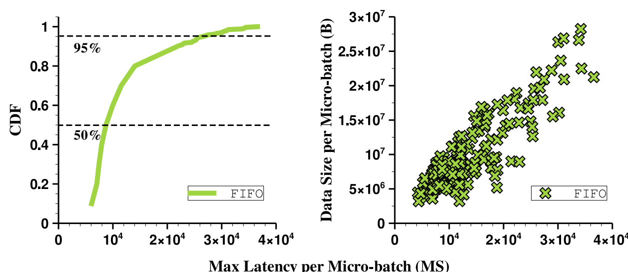


Fig. 2 The cumulative distribution function of the maximum dataset latency per micro-batch (left), and the correlation between latency and data size (right) in the case of using FIFO scheduling

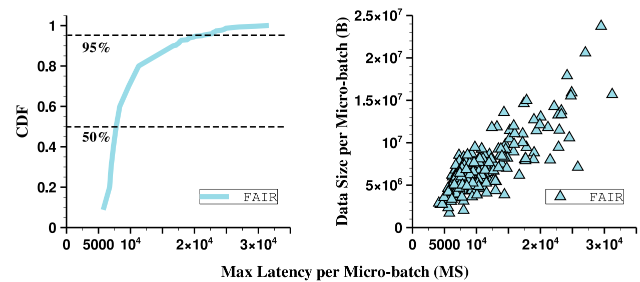


Fig. 3 The cumulative distribution function of the maximum dataset latency per micro-batch (left), and the correlation between latency and data size (right) in the case of using FAIR scheduling

As for the cause of such high tail latency, we point out two main problems of the conventional micro-batch streaming systems below.

- *Static buffering time set by users* The existing micro-batch streaming system queues a newly arriving dataset in a buffer. Afterward, when the trigger time interval (T_{intv}) set by the user has passed, all buffered datasets are configured as one micro-batch, and then the system performs query execution on it. In this case, the value of T_{intv} does not change during runtime. This static buffering method causes latency problems when the query processing rate becomes lower than the input data arrival rate. If the time it takes to execute the query on the ith micro-batch is more prolonged than T_{intv}, more datasets remain buffered for the increased time. The $i+1$th micro-batch contains many more datasets than before, and the time it takes to perform a query increases further while more datasets are buffered again. This vicious cycle increases the latency of each dataset, causing high tail latency. We demonstrated this chain action by showing a correlation between the latency and data size of each micro-batch in the right-side graphs of Figs. 2 and 3. It shows a positive correlation between latency and data size, meaning that high latency occurred as the data size per micro-batch gradually increased due to the static buffering method.
- *Basic scheduling algorithms that ignore application characteristics* The current micro-batch streaming system supports only the most basic scheduling methods such as FIFO and FAIR. This method does not consider application characteristics such as users' latency requirements specified for each query, types of operators used in each query, and input data traffic. Therefore, it is difficult for the system performance to reach the user's expectations, and it is impossible to optimize the performance of all queries evenly in a multi-query environment. In the case of FIFO scheduler, since it only processes in the order of query execution requests without considering the latency

requirement per query, it cannot respond even if a specific query is delayed and causes a latency problem. In the case of FAIR scheduler, since all queries use resources equally, it causes latency in queries with much computation depending on the type of operation used. In particular, while the amount of query calculation changes in real-time due to the fluctuating input data traffic, each existing scheduler cannot react adaptively to it. As a result, high tail latency appears in all cases regardless of the existing scheduling policies, as shown in Figs. 2 and 3. Therefore, no scheduler prevents tail latency from occurring considering application characteristics.

Our results strongly indicate the need for a mechanism and algorithm that resolves each problem, which would effectively reduce tail latency. Our main idea is to use the *deadline* concept as a tool to constrain the system to keep each dataset's latency around a specific time value. We propose a deadline-aware dynamic buffering method and deadline-aware scheduling policies to solve the above problems.

4 Overview of zStream

Existing systems unconditionally create micro-batches at fixed time intervals according to a trigger set by the user (Sect. 2.2 ③). Then it executes a query for the micro-batch according to a scheduling algorithm (e.g., FIFO or FAIR) (Sect. 2.2 ④). zStream adds two modules to the original Spark: (i) *admission controller* and (ii) *priority-based scheduler*. These modules allow the system to prevent the tail latency of every query by using each deadline information. Figure 4 shows the overall system structure of zStream. We highlight the two newly designed modules. To explain the role of each module in more detail, we present the micro-batch execution cycle of *buffering phase*, *scheduling phase*, and *analysis phase*.

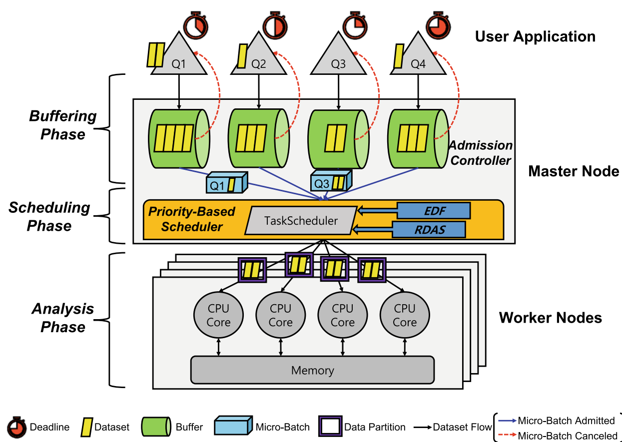


Fig. 4 Overview of zStream architecture

- *Buffering phase* zStream adaptively adjusts the trigger interval time to maintain the latency of datasets around the deadline time with an *admission controller module*. The arriving dataset accumulates in the admission controller's query-specific buffer until a coordinated trigger occurs. The admission controller calculates the following three times at 10 ms intervals per query to find the right moment for the trigger to occur. Let $DS_{earliest}$ be the dataset waiting the longest in a specific query Q_n and MB_i be the micro-batch containing $DS_{earliest}$. ① First, the admission controller calculates the difference between the arrival time of $DS_{earliest}$ and the current time to get the buffering time of $DS_{earliest}$ (T_{buff}). ② Next, it predicts the time for $DS_{earliest}$ to be scheduled and to wait for its execution (T_{sched}). For example, if Q_1 and Q_2 are already waiting for their execution in a simple FIFO scheduler queue, Q_n must wait until the completion of both of them. Therefore, it is necessary to predict the amount of time in advance. ③ Finally, the scheduler estimates the time the system performs query executions for the MB_i to get the final analysis result (T_{anal}). The admission controller adds all of T_{buff}, T_{sched}, and T_{anal}. If the sum is greater than or equal to the deadline, the system activates the query trigger immediately to authorize the construction of the micro-batch. If not, the system cancels the construction of the micro-batch and repeats the same process to adjust the buffering phase, while considering continuous data ingestion. More detailed algorithms and prediction methods of T_{sched}/T_{anal} are described in Sect. 5.1.
- *Scheduling phase* Spark's query planner creates a task, a scheduling unit, combining the query operation set and the micro-batch dataset. zStream schedules tasks with a *priority-based scheduler module* to sustain the latency of datasets around the deadline time. There are two scheduling policies, and in all approaches, the priority is determined based on the time remaining until the deadline. The first is *Earliest-Deadline-First* (*EDF*). This policy sorts task execution orders by assigning priority based on specific factors that reduce tail latency. The primary factor is the time remaining until the deadline, and the shorter the time, the higher the priority. EDF performs sorting whenever a new task arrives. The second is *Resource-Deadline-Aware-Scheduling* (*RDAS*). This policy also gives priority in the order of the shortest time remaining until the deadline. However, the allocation of resources (CPU cores) differs according to the priority, and tasks with a high priority can be processed quickly using many

resources. The detailed algorithm for each policy is described in Sect. 5.2.

- *Analysis phase* After zStream finally determines a specific task to execute, the system divides the datasets in the task equally and forms units called data partitions. Each data partition is mapped 1:1 to the core, executes the specified query in-memory method, and delivers meaningful analysis results to the user.

5 Design and implementation

In this section, we describe the details of our design and implementation for the two core modules of zStream. To explain our own mechanisms, we use several notations which are summarized in Table 1.

5.1 Admission controller

This section describes the detailed mechanism of the admission control module. The execution unit in a micro-batch model is created by buffering incoming data from the input source in real-time. Buffered datasets are passed to the processing phase simultaneously, distributed to each executor, and processed in parallel. The buffering time is different per dataset because the arrival time of the individual dataset is different. The processing time is the same for all datasets within a single micro-batch. Therefore, we define max latency per micro-batch as the sum of the buffering time of the earliest-arrived dataset and the processing time of the micro-batch.

To prevent the tail latency of queries, admission controller estimates the max latency of the future micro-batch, so that it can sustain every max latency per micro-batch around the deadline time. It sums up three types of time to get the max latency of prospective micro-batch as shown in Eq. 1. Each type of time components is defined in Eqs. 2, 3, and 4.

$$MaxLat_{(q,i)} = T^{Buff}_{(q,i)} + T^{Sched}_{(q,i)} + T^{Anal}_{(q,i)} \tag{1}$$

$$T^{Buff}_{(q,i)} = currentTime - arrivalTime(EearlyDS_{(q,i)}) \tag{2}$$

$$\forall(x,y) \in (queuedQuery, queuedMicroBatch), \quad T^{Sched}_{(q,i)} = \sum T^{Anal}_{(x,y)} \tag{3}$$

$$T^{Anal}_{(q,i)} = \begin{cases} \dfrac{BatchSize_{(q,i)}}{AvgThPut_q}, & \text{if FIFO/EDF} \\ \dfrac{BatchSize_{(q,i)}}{AvgThPut_q \times \dfrac{\#allocCore}{\#totalCore}}, & \text{if FAIR/RDAS} \end{cases} \tag{4}$$

$T^{Buff}_{(q,i)}$ is the buffering time measured based on the current time. Admission controller calculates the value by subtracting the arrival time of the earliest-arrived dataset from the current time as shown in Eq. 2. $T^{Sched}_{(q,i)}$ is the scheduling delay time. Admission controller gets the value by adding all estimated analysis time of active queries in the scheduling queues as shown in Eq. 3. $T^{Anal}_{(q,i)}$ is the analysis time, which is the time for query execution. Admission controller estimates the value by dividing the data size of micro-batch with average throughput of the query as shown in Eq. 4. To maintain average throughput information per query, it calculates and stores the throughput of every micro-batch after its completion. If the scheduling policy is FAIR or RDAS (the new policy of zStream which

Table 1 Notations for describing the detailed mechanism of each module in zStream

Notation	Description
$Buff_{(q,i)}$	Buffered data for ith micro-batch of query q
$MB_{(q,i)}$	ith constructed micro-batch of query q
$Deadline_q$	Deadline time specified per query q
$EearlyDS_{(q,i)}$	Earliest-arrived dataset of micro-batch i of query q
$BatchSize_{(q,i)}$	Total size of all dataset of micro-batch i of query q
$AvgThPut_q$	Average throughput of query q
$MaxLat_{(q,i)}$	Max latency per micro-batch i of query q
$T^{Buff}_{(q,i)}$	Buffering time of micro-batch i of query q
$T^{Sched}_{(q,i)}$	Scheduling delay time of micro-batch i of query q
$T^{Anal}_{(q,i)}$	Query execution time of micro-batch i of query q
$T^{Remain}_{(q,i)}$	Time remaining until deadline for micro-batch i of query q
$Priority_{(q,i)}$	Scheduling priority of micro-batch i of query q

improves the FAIR policy), concurrent queries share the computing resources (CPU cores). In these cases, admission controller penalizes average throughput by multiplying the ratio of the number of allocated CPU cores to the total number of CPU cores.

Algorithm 1: Admission Controller

```
1  Def AdmissionControll(Buff_(q,i), Deadline_q):
2      BatchSize_(q,i) = sizeof(Buff_(q,i))
3      Get T^Buff_(q,i) // Eq 2
4      Estimate T^Sched_(q,i) // Eq 3
5      Estimate T^Anal_(q,i) // Eq 4
6      MaxLat_(q,i) = T^Buff_(q,i) + T^Sched_(q,i) + T^Anal_(q,i)
7      if MaxLat_(q,i) ≥ Deadline_q then
           // Micro-Batch Admitted
8          Buff_(q,i) = ∅
9          return (True, MB_(q,i))
       // Micro-Batch Cancelled
10     Set all datasets as Buff_(q,i)
11     BatchSize_(q,i) = 0
12     return (False, ∅)
```

Algorithm 1 shows the dynamic buffering method in the admission controller module. After admission controller gets the max latency of prospective micro-batch ($MaxLat_{(q,i)}$), it compares the value with query-specific deadline ($Deadline_q$). If the max latency value is bigger or the same, it admits the micro-batch so that it could be scheduled and perform query executions. In the max latency value is smaller, it cancels the micro-batch and continues its buffering. In zStream, the admission controller module is called every 10 ms per query to judge whether to admit or cancel each micro-batch.

Scenario of admission controller An example scenario of dynamic buffering in admission controller is shown in Fig. 5. We assume that datasets with timestamp of multiple IoT devices are stored in real-time in the message broker. We also assume that the user sets the deadline as T_6 for query Q_k. To create a micro-batch for Q_k, the admission controller determines how many datasets to include. First, it creates a temporary buffer for micro-batch creation and

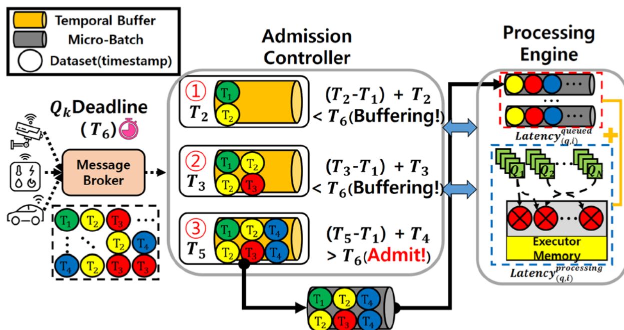


Fig. 5 Example scenario of admission controller mechanism

stores some datasets fetched from the message broker. In the first step of the scenario, admission controller finds the earliest dataset from the buffer and subtracts it from the current time T_2. It is then compared to the deadline T_6 in addition to the estimated time T_2 to be processed when the temporary batch is handed over to the processing engine. Since the result of calculation T_3 does not exceed the deadline, admission controller buffers more datasets from the message broker. This process continues until the deadline is exceeded. The last step in the scenario shows the process of creating a micro-batch. The sum of the dataset's longest waiting time T_4 in the temporary buffer and the estimated time T_4 in the processing engine exceeds the deadline T_6. At this time, the admission controller stops buffering and creates a micro-batch of the dataset collected so far and delivers it to the processing engine.

5.2 Priority-based scheduler

This section explains two new scheduling policies of the priority-based scheduler module. Although admission controller dynamically buffers data to sustain the latency, it is not enough to minimize the tail latency without further schedules. For example, there might be a micro-batch whose max latency is already behind its target deadline when it arrives at the scheduling phase. In this case, rather than using a conventional FIFO policy, it needs another approach to increase the execution order priority to minimize the increase in latency of the query. Priority-based scheduler module proposes deadline-aware task scheduling policies, which effectively sustain the latency of individual dataset of every queries.

Algorithm 2: Priority-based Scheduler

```
1  procedure Scheduler(schedulingQueue):
2      Get T^Remain_(q,i)
3      Compute Priority_(q,i)
4      if EDF then
5          Sort schedulingQueue by Priority_(q,i)
6          Run in FIFO fashion
7      else if RDAS then
8          Do min-max normalization with all Priority_(q,i)
9          Update weight of each pools in schedulingQueue
10         Run in FAIR fashion
```

Algorithm 2 shows how each policy works. To determine the deadline-aware priority, it is necessary to get the remaining time until the deadline ($T^{Remain}_{(q,i)}$) as shown in Eq. 5.

$$T^{Remain}_{(q,i)} = Deadline_q - arrivalTime(EearlyDS_{(q,i)}) \quad (5)$$

If the scheduling policy is set as EDF, priority-based scheduler sorts the scheduling queue by deadline-aware priority to rearrange the execution order. It performs

sorting whenever a new micro-batch arrives. Right after sorting, it follows the FIFO fashion and conducts tasks according to the priority. If the scheduling policy is RDAS, priority-based scheduler redistributes resources according to deadline-aware priority. Then, it transforms priority into normalized weight value to make a criterion for allocating resources to each query. For the normalization, we use min-max normalizing method. With given weights, priority-based scheduler allocates more CPU cores to higher-weighted tasks.

Scenario of priority-based scheduler An example scenario of priority-based scheduler considering a deadline is shown in Fig. 6. We assume that a whole taskset for each query has a timestamp and the deadline. Moreover, there are additional delays, such as queuing delay and task execution time according to scheduling modes. We also assume that t_1 is the additional delay that occurs in this scenario. In the current Spark scheduler, queries are executed in timestamp order, and while one query is being executed, other queries are waiting. In this scenario, the timestamps of Q_1, Q_2, and Q_3 are T_1, T_2, and T_3, respectively, so Q_1, Q_2, and Q_3 are executed sequentially. Assuming that the processing time of task for each query is T_1, in the case of Q_1, its latency would be around the target deadline because it is processed immediately without queuing delay. However, in the case of Q_3, a queuing delay occurs while Q_1 and Q_2 are executed, therefore Q_3 would have a latency far behind the target deadline T_3. To solve this problem, each query is sorted in the order of the shortest remaining time, and as a result, the latency of Q_3 can be reduced.

With the above process, RDAS can also schedule by the assigned priority of task for each query. Although RDAS follows the FAIR policy, RDAS changes the weight of the task for each query with the calculated priorities and assigns different number of cores accordingly, whereas original FAIR policy shares cores evenly. In the scenario, original FAIR policy would distribute the 6 available CPU cores evenly, assigning Q_1, Q_2, and Q_3 two cores each.

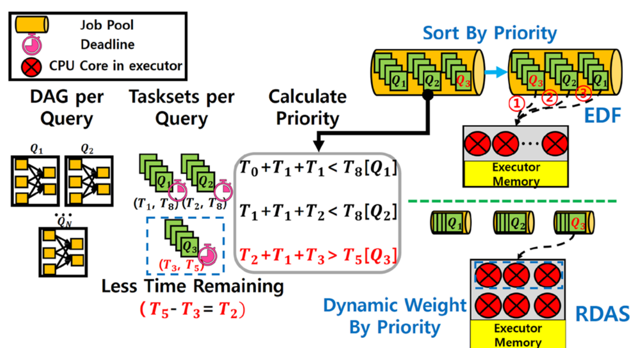


Fig. 6 Example scenario of priority-based scheduler mechanism

Suppose that Q_3, which has the shortest remaining time, will take 2 s to execute with two cores. In this case, since an additional core would be assigned to Q_3 with RDAS, execution time can be reduced to 1.5 s, which will eventually reduce the latency.

5.3 Implementation

We implemented zStream on Apache Spark Structured Streaming [7] (version 3.0.1) that supports micro-batch streaming system with SQL-based optimized query processing. To implement zStream, we disabled the trigger option and set the deadline time per query in the application. Then, we added our new two modules on top of the Structured Streaming. We built the admission controller module before the step of constructing a micro-batch per query (Sect. 2.2 ②) and the priority-based scheduler module as a part of existing TaskScheduler operations (Sect. 2.2 ④). We used Scala language to implement overall system components of zStream. The source codes are available at https://github.com/sylee0506/zStream.

6 Evaluation

This section compares the performance of structured streaming, which supports the streaming API in Spark, and the proposed mechanism to show the possible trade-offs.

6.1 Experimental setup

Experimental environments For the experiment, we configured Spark cluster consisting of one master node and two worker nodes. All experiments are performed on a server with AMD Ryzen 9 3900X 12-core 3.80 GHz supporting hyper-threading and 64 GB of memory. All nodes are interconnected by 10 Gbps Ethernet. We set 1 executor per worker node with 16 CPU cores and 24 GB of memory. Both the proposed system and the comparison targets are run on the Oracle JVM 8 with G1 garbage collector. The rest of the framework settings followed the default (Table 2).

Workloads and stream traffic types The workload used in the experiment is a real-world streaming workload, Linear Road benchmark [14] that includes various query operations, such as filter, project, aggregate, and join. We used query 2, 3, 4 [15], which are frequently used in previous works. Each query notation is LR1, LR2, LR3. To represent a whole streaming system, we built Apache Kafka [13] as message broker and Spark as stream processing engine. Kafka producer sends a dataset to message queues in random rates. Then, Kafka consumer stores a dataset into the local file system of master node. According

Table 2 Query details of real-world streaming workloads used in our experiments

Notation	Linear road benchmark [14] streaming query
*LR*1	SELECT L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment FROM SegSpeedStr (range 30 slide 1) as A, SegSpeedStr as L WHERE (A.vehicle == L.vehicle)
*LR*2	SELECT timestamp, highway, direction, segment, AVG(speed) as avgSpeed FROM SegSpeedStr (range 30 slide 1) GROUPBY (highway, direction, segment) HAVING (avgSpeed < 40.0)
*LR*3	SELECT timestamp, highway, direction, segment, COUNT(vehicle) as numVehicle FROM SegSpeedStr (range 30 slide 1) GROUPBY (highway, direction, segment)

to the trigger value, the processing engine fetches datasets and starts to make micro-batches. To observe performance according to traffic distribution, we defined low and high traffic. The detailed description of traffic rate is as follows.

- Traffic (low): a random record is generated every second so that its average converges to 1000 records per second.
- Traffic (high): a random record is generated every second so that its average converges to 10,000 records per second.

Comparison targets In order to show the effectiveness of our approach, an overall system performance was compared with the original Spark [7].

Spark processes queries by fetching data for each trigger value set by the user. As the processing engine gradually slows down, even if numerous datasets are accumulated in the meantime, a fixed trigger fetches all. Since the dataset waits for a long time in the buffering phase, there is a problem that the tail latency of the query increases. In contrast, zStream leverages query-specific triggers to adjust the micro-batch size enough to handle the current processing capacity. It also minimizes the tail latency by prioritizing queries that are imminent to the deadline during the scheduling phase. In this way, the performance effect depends on the trigger value set by the user.

For the performance comparison between original Spark and zStream, we used three trigger intervals such as 5, 8, and 10 s. While the original Spark used the static trigger interval and default scheduling policies (i.e., FIFO and FAIR), zStream dynamically adjusts the trigger interval and schedules tasks by considering the trigger interval for each query. The comparison targets are summarized below.

* *SB-FIFO* Original Spark (static buffering) with FIFO scheduling policy.
* *SB-FAIR* Original Spark (static buffering) with FAIR scheduling policy.
* *DB-EDF* zStream (dynamic buffering) with EDF scheduling policy.
* *DB-RDAS* zStream (dynamic buffering) with RDAS scheduling policy.

We compare the performance of the above mechanisms in terms of tail latency, average latency, and average throughput. Finally, we analyze the effectiveness of each module, such as the scheduling policy and dynamic batching proposed in this paper.

6.2 Tail latency

This section compares the tail latency of original Spark and zStream by varying scheduling policy and trigger value according to traffic distributions. From Figs. 7, 8, 9, 10, 11, and 12, it shows a cumulative distribution function of the tail latency of both mechanism per query with increasing trigger interval. We calculated the average value of the tail latency by performing the experiment five times for 30 min for each case.

Overall, zStream shows a performance improvements in median and 95th/99th percentile latency regardless of traffic distributions compared to original Spark. Interestingly, there are different tail latency patterns when traffic is varied.

When traffic is low, zStream significantly reduces the average and tail latency in all trigger cases. Figure 7 shows the tail latency per query when trigger interval is set to 5 s in low traffic. Fixed trigger-based stream processing can increase tail latency regardless of query characteristics. In the case of median latency, DB-EDF and DB-RDAS remain close to the trigger interval, but SB-FIFO and SB-FAIR show higher tail latency than the trigger interval. This trend is more evident in LR1. The median latency of DB-EDF is 5.5 s, while the median latency of SB-FIFO is 7.3 s, resulting in a maximum difference of 32%. This is because dynamic admission control adjusts micro-batch size so that each dataset waiting in the data source can be processed as much as possible within the user-defined trigger interval. In contrast, the original Spark fetches all datasets without considering the processing capacity, resulting in a continuous increase in processing latency than the trigger interval. As a result, the tail latency increases indefinitely because the buffering phase is longer than the time to process the micro-batch.

As the trigger interval increases, the tail latency pattern of zStream and the original Spark also varies. For all queries in Fig. 8, we can observe a constant difference between the median and 95th/99th percentile latency between the two mechanisms. When more datasets accumulated in a micro-batch due to a longer trigger interval, zStream significantly reduces the tail latencies.

Although the query processing is slowed by the data traffic ingestion rate exceeding the processing capacity, we can also observe that zStream maintains tail latency similar to the trigger interval until 95th percentile from the median. The reason why tail latency is maintained with trigger interval is that zStream has a point to determine when to micro-batch a data set waiting in the buffering phase. If there is no point in deciding how much buffering to be done, the slower the processing capacity, the longer the buffering is, and the tail latency increases without bound.

When the trigger interval is set to 10 s, zStream still shows shorter tail latency. In case of LR1, LR2 shown in Fig. 9, we can observe that DB-EDF reduces 95th percentile/99th percentile tail latency by up to 26% and 27%,

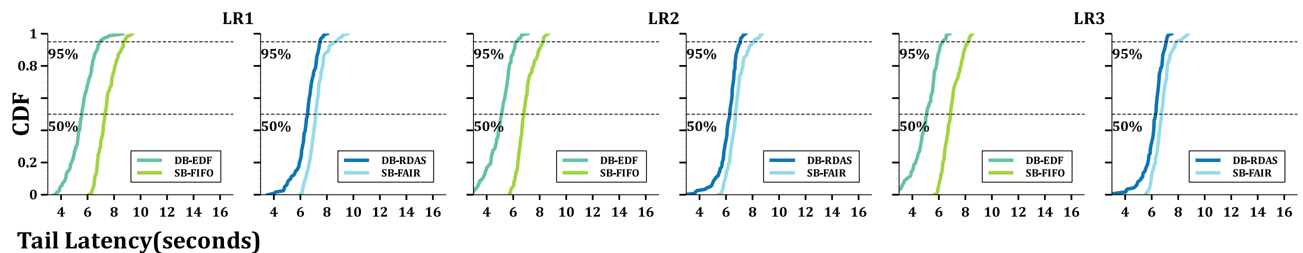


Fig. 7 CDF of the maximum dataset latency per scheduling policy (Trigger = 5, Traffic = Low)

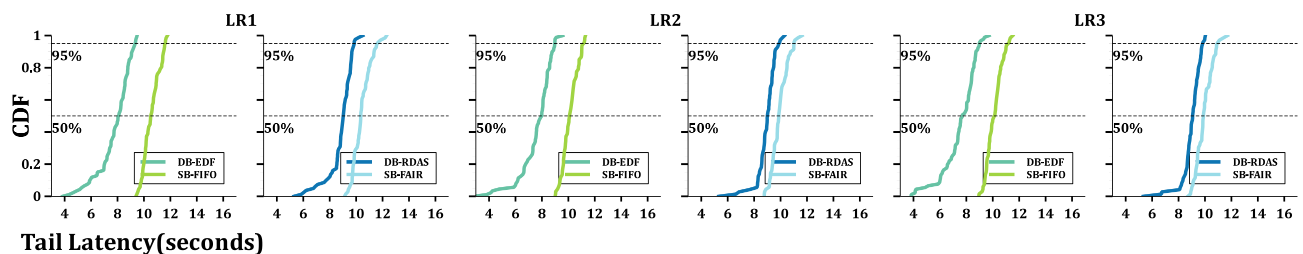


Fig. 8 CDF of the maximum dataset latency per scheduling policy (Trigger = 8, Traffic = Low)

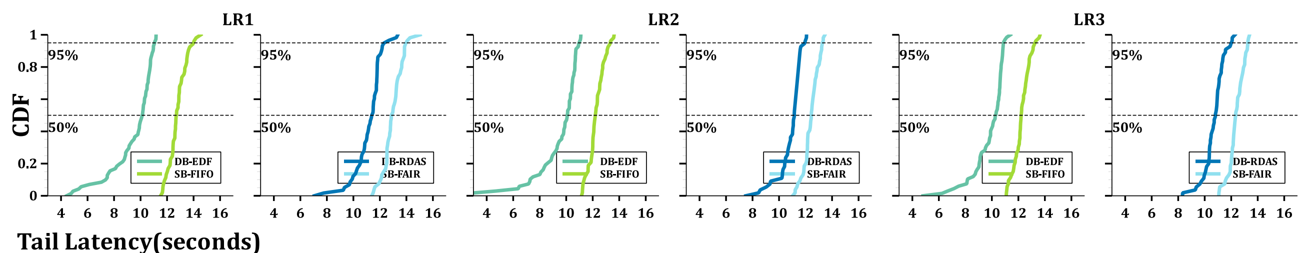


Fig. 9 CDF of the maximum dataset latency per scheduling policy (Trigger = 10, Traffic = Low)

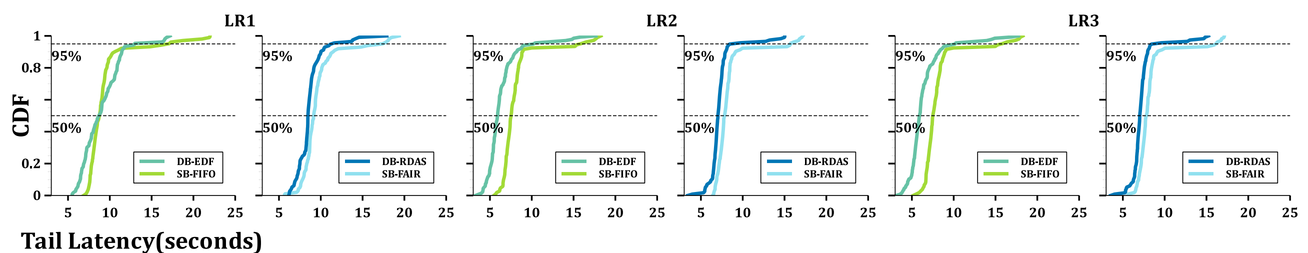


Fig. 10 CDF of the maximum dataset latency per scheduling policy (Trigger = 5, Traffic = High)

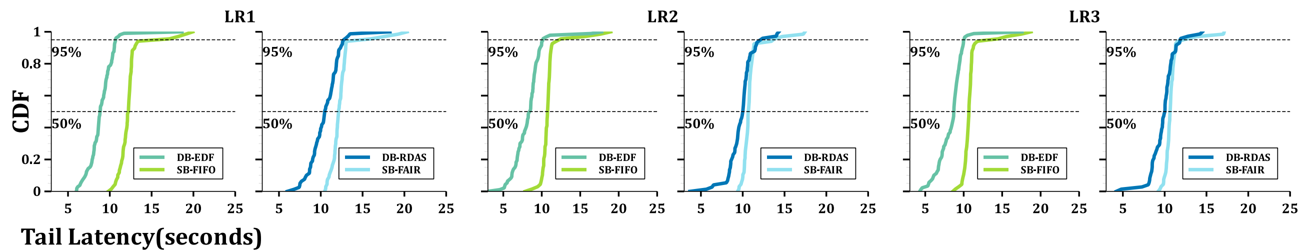


Fig. 11 CDF of the maximum dataset latency per scheduling policy (Trigger = 8, Traffic = High)

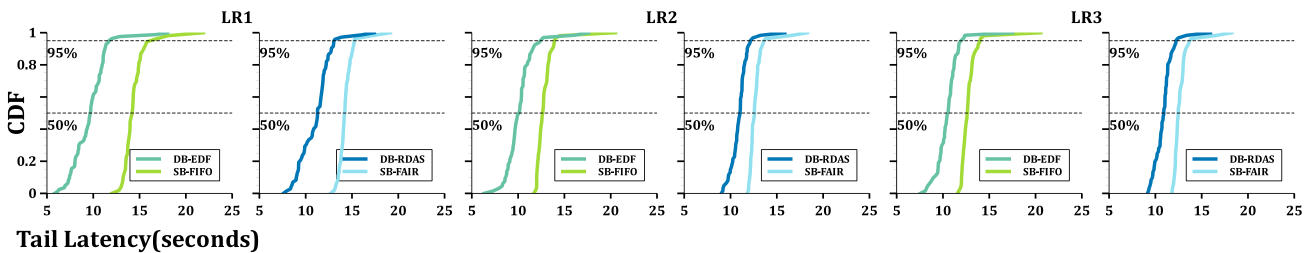


Fig. 12 CDF of the maximum dataset latency per scheduling policy (Trigger = 10, Traffic = High)

respectively. Interestingly, the original Spark shows tail latency above the average of 20% of the trigger interval from the initial micro-batch step. This is a case where too many datasets are fetched from the data source to process with the available processing capacity. Also, default task scheduling that executes queries based on the arrival time cannot reduce tail latency in this case. In contrast, zStream leverages the trigger interval to adjust the micro-batch size and accelerate query processing by prioritizing the dataset with the least time remaining until trigger interval processing. For this reason, regardless of trigger interval, zStream processes queries near the trigger interval from the initial batch stage and shows a tail latency with less even 95th and 99th percentile.

When traffic is higher, the tail latency gap between zStream and original Spark is narrowed, but it still performs well. As with traffic at low, the latency gap between zStream and original Spark gets wider in median and 95th percentile as the trigger increases. In case of LR2, LR3 in Fig. 10, it shows a slight latency difference in median and 95th percentile. The reason is that the expected time for processing the temporary micro-batch often exceeds the relatively short target latency as the traffic ingestion rate increases. It indicates that a small trigger value might limit the effectiveness of dynamic buffering and priority scheduling. However, when the trigger interval increases, zStream reduces the median and 95th/99th percentile latency. Compared to LR1 in Fig. 12, it is shown that DB-EDF and DB-RDAS reduces 95th percentile latency by up to 34% and 16%, respectively. This is because it not only aggressively leverages the point that creates the optimal micro-batch to process, but also dynamically schedules tasks to give priority in the order of imminent time remaining until the target latency. In contrast, original Spark collects the datasets until a fixed trigger interval is reached and creates the micro-batch. As a result, a numerical dataset due to high traffic is fetched at once, resulting in long tail latency because the dataset waits for a long time in the buffering phase.

6.3 Average latency and throughput

This section analyzes the average latency and throughput of original Spark and zStream to verify the performance improvement of zStream. Figures 13, 14 shows an average latency and average throughput when the trigger interval for each traffic distribution varies from 5 to 8 s. Since the overall performance of each query showed a similar pattern, we used LR1, which showed the most distinct pattern among them. The average latency is the sum of the processing time and waiting time for each dataset included in the micro-batch divided by the total number of batches. The larger the trigger interval, the smaller the average latency for zStream compared to the original Spark in two traffic distributions. In Fig. 14, when the trigger interval is 10 s, DB-EDF and DB-RDAS show up to 48% and 32% reduction in average latency, respectively. The reason is that zStream adjusts the micro-batch appropriate to the current processing capacity. Moreover, it also tries to minimize the tail latency of each dataset belonging to the micro-batch as much as possible in the scheduling phase.

Interestingly, we can observe that the average throughput also increased slightly. In general, the average throughput is the total batch size processed up to the entire

micro-batch divided by the total time. It is expected that the average throughput decreases compared to the original Spark, as zStream is partially fetched from datasets loaded in the data source during the previous micro-batch step. However, the average throughput is slightly increased regardless of the trigger interval. The average throughput increases even though the micro-batch size is small because the throughput capacity is well maintained in zStream compared to the original Spark. This indicates that dynamic admission control and priority-based task scheduling policies proposed by zStream affect overall performance improvement.

6.4 Scalability

We primarily configured a our cluster with one master node and two worker nodes. To evaluate the scalability of the zStream, we also conducted experiments upon the cluster with four and six workers. Figure 15 describes the performance scalability comparison between orginal Spark and zStream. It indicates that even if the number of workers increases, original Spark still incurs high tail latency without performance improvement. In particular, using the FAIR policy in original Spark even shows increasing tail latency as the number of workers scales. On the other hand, zStream shows degrading tail latency as the number of workers increases, regardless of its scheduling policies. The results show that zStream has better scalability than original Spark.

6.5 Module effectiveness

In order to figure out the performance effectiveness of zStream, we evaluate the performance per algorithm according to the module options. For evaluation, we apply the dynamic admission control and the priority-based scheduling algorithm for each scheduling policy. We also normalized the average latency of LR1 query with static trigger interval and FIFO scheduling policy. Figure 16 shows the performance improvement by zStream module

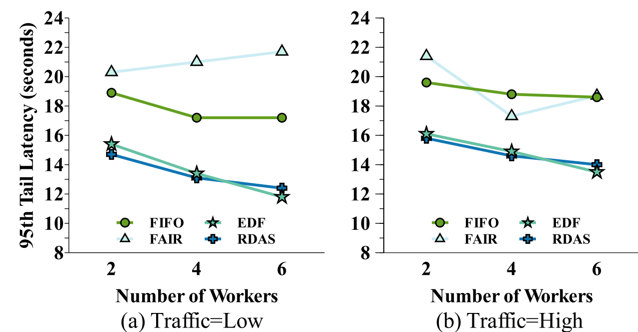
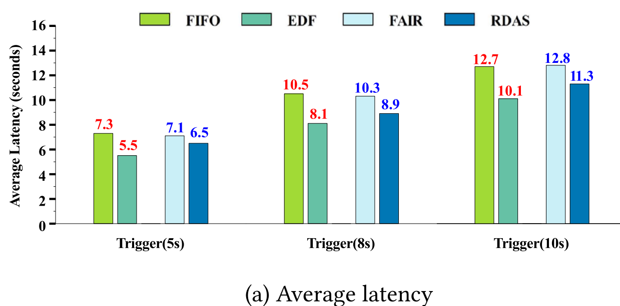


Fig. 15 Scalability of 95th percentile tail latency in LR1 (Trigger = 10)



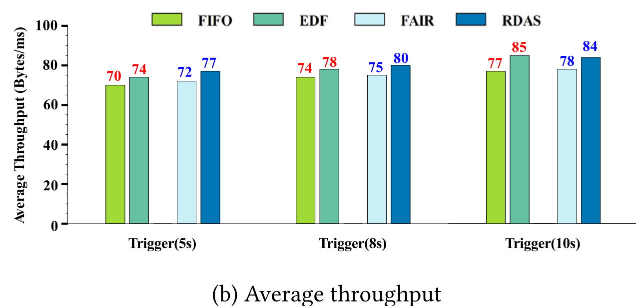
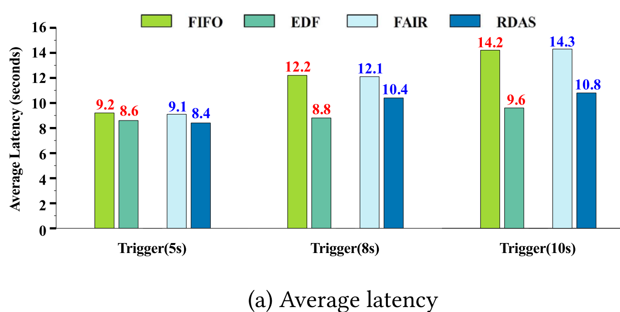


Fig. 13 Evaluation of the average latency and throughput in LR1 (Traffic = Low)



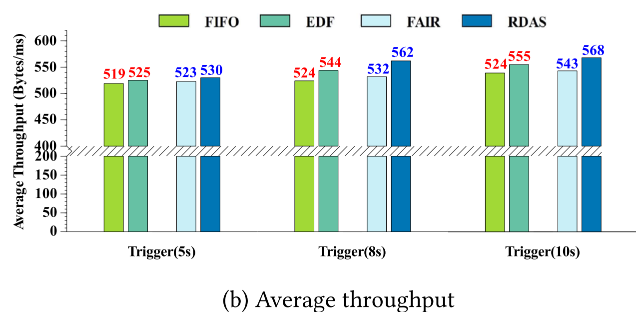
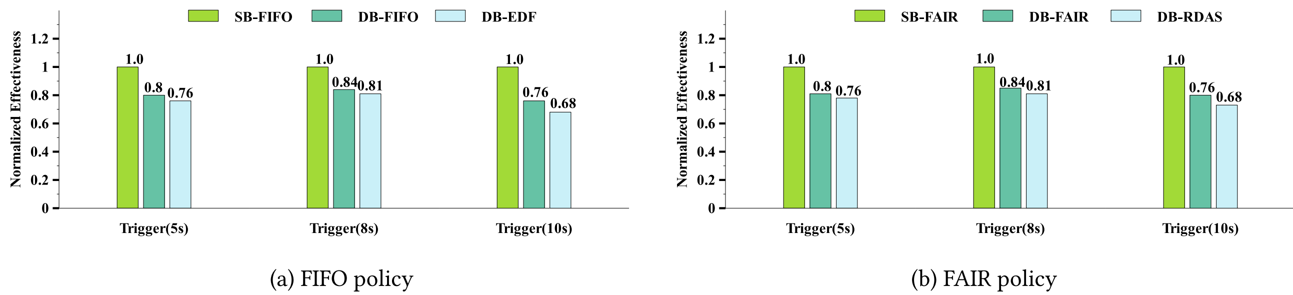


Fig. 14 Evaluation of the average latency and throughput in LR1 (Traffic = High)



(a) FIFO policy

(b) FAIR policy

Fig. 16 Module effectiveness per scheduling policy (query = LR1)

according to the trigger interval. The results show that the performance improvement of the dynamic admission control of zStream is the most dominant factor. In particular, the performance effectiveness of dynamic admission control increases as the trigger interval increases. Meanwhile, priority-based scheduling algorithm shows a slight performance improvement. When the trigger interval is 10 s, priority-based scheduling algorithm shows a performance improvement of up to 8% compared to when only dynamic admission control is applied. This shows that not only adjusting the micro-batch, but also prioritizing queries with imminent deadlines effectively reduces tail latency.

7 Related works

Das et al. [16] 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.

In addition to dynamically adjusting the batch interval according to processing capacity to accelerate query execution, there are also studies that optimize spark task scheduling. Cheng et al. [17] 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. Garefalakis et al. [18] proposes Neptune, which suspends tasks in batch jobs to prioritize latency-sensitive tasks. It is a proactive scheduler that suspends low-priority batch jobs when there are streaming jobs. Most studies in the micro-batch model focus on scheduling the queries to match the processing capacity of the processing engine not considering user's latency requirements.

On the other hand, zStream presents a deadline-aware micro-batch streaming system that controls the micro-batch size dynamically and schedules multi queries according to priority considering user's latency requirements. 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.

8 Conclusion

In this paper, we have proposed a low latency micro-batch streaming system called zStream that supports a deadline-aware dynamic buffering mechanism and priority-based scheduling. zStream effectively reduces the latency for the individual dataset by dynamically adjusting the buffering time. In addition, zStream uses deadline-aware priority-based scheduling policies to prevent tail latency of every query. In particular, zStream 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. Experiments show that the proposed mechanisms effectively reduce tail latency per streaming queries while maintaining system throughput.

In future work, we aim to design a system that automatically finds the optimal deadline reference value within zStream, rather than delegating it to users.

Acknowledgements This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (Grant No. NRF-2021R1A2C2014386). This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (Grant No. IITP-2021-2017-0-01628) supervised by the IITP (Institute for Information & communications Technology Promotion).

Author contributions All authors contributed to the study conception and design. Design and analysis were performed by SL, YJ, and SP. SL, YJ, KP, GJ, and SP contributed in writing and proofreading of the manuscript.

Funding This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (Grant No. NRF-2021R1A2C2014386). This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (Grant No. IITP-2021-2017-0-01628) supervised by the IITP (Institute for Information & communications Technology Promotion).

Data availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests The authors have no relevant financial or non-financial interests to disclose.

References

1. Lee, S., Jeong, Y., Kim, M., Park, S.: Q-spark: Qos aware micro-batch stream processing system using spark. In: 2021 IEEE International Conference on Autonomic Computing and Self-organizing Systems Companion (ACSOS-C), pp. 38–43 (2021)
2. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. SIGMOD Rec. **34**, 42–47 (2005)
3. Akidau, T., Chernyak, S., Lax, R.: Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. O'Reilly Media Inc., Sebastopol (2018)
4. Sun, D., Zhang, G., Zheng, W., Li, K.: Key technologies for big data stream computing. In: Big Data Algorithms, Analytics, and Applications, pp. 1–22. Chapman and Hall/CRC (2004)
5. Xu, L., Venkataraman, S., Gupta, I., Mai, L., Potharaju, R.: Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pp. 389–405. USENIX Association (2021)
6. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 423–438 (2013)
7. Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., Zaharia, M.: Structured streaming: a declarative API for real-time applications in Apache Spark. In: Proceedings of the 2018 International Conference on Management of Data, pp. 601–613 (2018)
8. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In: Proceedings of the VLDB Endowment, vol. 8, pp. 1792–1803 (2015)
9. Chen, F., Wu, S., Jin, H., Yao, Y., Liu, Z., Gu, L., Zhou, Y.: Lever: towards low-latency batched stream processing by pre-scheduling. In: Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, (New York, NY, USA), p. 643. Association for Computing Machinery (2017)
10. He, B., Yang, M., Guo, Z., Chen, R., Su, B., Lin, W., Zhou, L.: Comet: batched stream processing for data intensive distributed computing. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, (New York, NY, USA), pp. 63–74. Association for Computing Machinery (2010)
11. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache Spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
12. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), (San Jose, CA), pp. 15–28. USENIX Association (2012)
13. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. Proc. NetDB **11**, 1–7 (2011)
14. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30, pp. 480–491 (2004)
15. Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A. L., Costa, P., Pietzuch, P.: Saber: window-based hybrid stream processing for heterogeneous architectures. In: Proceedings of the 2016 International Conference on Management of Data, pp. 555–569 (2016)
16. Das, T., Zhong, Y., Stoica, I., Shenker, S.: Adaptive stream processing using dynamic batch sizing. In: Proceedings of the ACM Symposium on Cloud Computing. SOCC '14, Seattle, WA, pp. 1–13. Association for Computing Machinery, New York, NY (2014)
17. Cheng, D., Zhou, X., Wang, Y., Jiang, C.: Adaptive scheduling parallel jobs with dynamic batching in spark streaming. IEEE Trans. Parallel Distrib. Syst. **29**(12), 2672–2685 (2018)
18. Garefalakis, P., Karanasos, K., Pietzuch, P.: Neptune: scheduling suspendable tasks for unified stream/batch applications. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC), pp. 233–245 (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Suyeon Lee is currently working as a researcher in distributed computing and operating systems laboratory at Sogang University, Seoul, Republic of Korea. She received her B.S./M.S. degree in computer science and engineering from Sogang University. Her research interests include high-performance systems for data-intensive environments.

Yeonwoo Jeong is currently pursuing Ph.D. degrees in computer science and engineering from Sogang University, Seoul, Republic of Korea. He received his B.S. degree in computer software from Kwangwoon University and M.S. degree in computer science and engineering in Sogang University. His research interests include cloud computing and resource management.

Kyuli Park is currently pursuing Master degrees in computer science and engineering from Sogang University, Seoul, Republic of Korea. She received her B.S. degree in computer science and engineering from Sogang University. Her research interests include cloud computing, streaming system and resource management.

Gyeonghwan Jung is currently pursuing Master degree in computer science and engineering from Sogang University, Seoul, Republic of Korea. He received B.S. degree in computer science from Sangmyung University. His research interests include cloud computing and resource management.

Sungyong Park is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.