Adaptive Block and Batch Sizing for Batched Stream Processing System

Quan Zhang
Wayne State University
Detroit, MI, USA
quan.zhang@wayne.edu

Yang Song Ramani R. Routray
IBM Almaden Research
San Jose, CA, USA
{*yangsong, routrayr*}*@us.ibm.com*

Weisong Shi
Wayne State University
Detroit, MI, USA
weisong@wayne.edu

***Abstract*—The need for real-time and large-scale data processing has led to the development of frameworks for distributed stream processing in the cloud. To provide fast, scalable, and fault tolerant stream processing, recent Distributed Stream Processing Systems (DSPS) treat streaming workloads as a series of batch jobs, instead of a series of records. Batch-based stream processing systems could process data at high rate but lead to large end-to-end latency. In this paper we concentrate on minimizing the end-to-end latency of batched streaming system by leveraging adaptive batch sizing and execution parallelism tuning. We propose, DyBBS, a heuristic algorithm integrated with isotonic regression to automatically learn and adjust batch size and execution parallelism according to workloads and operating conditions. Our approach does not require workload specific knowledge. The experimental results show that our algorithm significantly reduces end-to-end latency compared to state-of-the-art : i) for *Reduce* workload, the latency can be reduced by 34.97% and 48.02% for sinusoidal and Markov chain data input rates, respectively; and ii) for *Join* workload, the latency reductions are 63.28% and 67.51% for sinusoidal and Markov chain data input rates, respectively.**

I. Introduction

The volume and speed of data being sent to data centers has exploded due to increasing number of intelligent devices that gather and generate data continuously. The ability of analyzing data as it arrives leads to the need for stream processing. Stream processing systems are critical to supporting application that include faster and better business decisions, content filtering for social networks, and intrusion detection for data centers. In particular, the ability to provide low latency analytics on streaming data stimulates the development of distributed stream processing systems (DSPS), that are designed to provide fast, scalable and fault tolerant capabilities for stream processing. Continuous operator model, that processes incoming data as records, is widely used in most DSPS systems [1]–[7], while recently proposed frameworks [8]–[12] adopt batch operator model that leverage Mapreduce [13] programming model and treat received data as continuous series of batch processing jobs.

It has become essential for organizations to be able to stream and analyze data in real time. Many streaming applications, such as monitoring metrics, campaigns, and customer behavior on Twitter or Facebook, require robustness and flexibility against fluctuating streaming workloads. Traditionally, stream processing systems have managed such scenarios by i) dynamic resource management [14], [15], or ii) elastic operator fission (i.e., parallelism scaling in directed acyclic graph (DAG)) [3], [16], [17], or iii) selectively dropping part of the input data (i.e., load shedding) [18]–[20]. Especially for batch based streaming systems, dynamic batch sizing adapts the batch size according to operating conditions [21]. However, dynamic resource allocation and elastic fission require expensive resource provisioning to handle burst load, and discarding any data may not be acceptable for exactly-once aggregation applications. The dynamic batch sizing also suffers long delay and overestimation for batch size prediction. In this paper, we focus on a batch-based stream processing system, Spark Streaming [12], that is one of the most popular batched stream processing systems, and minimize the end-to-end latency by tuning framework specified parameters in Spark Streaming.

Ideally, a batch size in Spark Streaming should guarantee that a batch could be processed before a new batch arrives, and this expected batch size varies with time-varying data rates and operating conditions. Moreover, depending on the workload and operating condition, a larger batch size leads to higher processing rate, but also increases the end-to-end latency. On the contrary, a smaller batch size may decrease the end-to-end latency while destabilize the system due to accumulated batch jobs, which means the data cannot be processed as fast as it is received. With the exception of batch size, the processing time of a batch job also significantly affects the end-to-end latency. With the same amount of available resources, less data processing parallelism (i.e., the number of blocks in a batch) may incur less execution overhead of task creation and communication but lower resource utilization, while massive parallel may dramatically increase the overheads even the resources may be fully utilized. By default, Spark Streaming adopts static batch size and execution parallelism (i.e., $\frac{batch\ size}{block\ size}$), which makes it possible that the system may involve any aforementioned issue. In this work, we focus on both *batch interval* and *block interval* as they are the most important factors affecting the performance of Spark Streaming.

To address these issues in Spark Streaming, we propose an online heuristic algorithm called DyBBS, which dynamically adjusts batch size and execution parallelism as the

workload changes. This algorithm is developed based on two significant observations: i) the processing time of a batch is a monotonically increasing function of batch size, and ii) there is an optimal execution parallelism for each particular batch size to achieve the minimal latency. These two observations (addressed in Section II) inspired us to develop our heuristic algorithm based on *Isotonic Regression* [22] to dynamically learn and adapt the batch size and execution parallelism in order to achieve low end-to-end latency while keeping system stability.

We compared our algorithm with three other solutions available on Spark Streaming for two representative workloads. The contributions of the paper are following:

- Without any assumption of workload characteristics, our algorithm is able to achieve low latency, that is i) significantly lower than the state-of-the-art solution [21] (implemented on Spark Streaming [12]), and ii) comparable to the optimal case.
- To the best of our knowledge, we are the first effort to introduce the ability of simultaneously managing batch size and execution parallelism in Spark Streaming, which adapts to time-varying workload and operating conditions.
- DyBBS, which is implemented in Spark 1.4.0, requires no workload specific tuning or user program modification, which makes the optimization transparent to end users and easy to deploy.

The remainder of this paper is organized as follows. In Section II, we analyze the relationship between batch size and execution parallelism and the performance of Spark Streaming. Section III presents design of our algorithm based on isotonic regression. The implementation details are addressed in Section IV, and we evaluate our algorithm in Section V. Section VI discusses the limitations of our work. Finally, we present the related work in Section VII and concludes in Section VIII.

II. Spark Streaming Insights

For a batched stream processing system, there are several key factors affecting the performance, which include cluster size, execution parallelism, batch size, etc. In this paper, we explore adaptive methods that minimize the end-to-end latency while maintaining the stability of system. In this section, we first describe the system model of Spark Streaming, which is our target platform. Then we discuss the basic requirements to minimize the latency. Finally, we show the impact of batch sizing and execution parallelism tuning on the performance, and how these insights inspire our algorithm design.

A. System Model

Spark Streaming is a batched stream processing framework, which is a library extension on top of the large-scale data processing engine Apache Spark [23]. Figure 1 demonstrates a detail overview of Spark Streaming. In general, Spark Streaming divides the continuously input data stream into batches in discrete time intervals. To form a batch, two important parameters, *block interval* and *batch interval* are used to scatter the received data. First, the received data is split with a relatively small block interval to generate a *block*. Then after a batch interval, all the blocks in the block queue are wrapped into a *batch*. Finally, batches are put into a batch queue, and the spark engine processes them one by one. Basically, the batch interval determines the data size (i.e., number of data records), and the execution parallelism of a batch is decided by $\frac{batch\ interval}{block\ interval}$, which is exactly the number of blocks in that batch. The end-to-end latency of a data record consists of three parts:

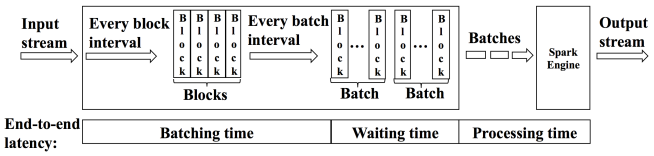


Figure 1. Stream processing model of Spark Streaming.

- *Batching time*: The duration between the time a data record is received and the time that record is sent to batch queue;
- *Waiting time*: The time a data record waits in the batch queue, which depends on the relationship between batch interval and processing time;
- *Processing time*: The processing time of a batch, which depends on the batch interval and execution parallelism.

Note that the batching time is upper bounded by the batch interval. The waiting time could be infinitely large, and eventually the accumulated batches exhaust available resources (e.g. OutOfMemory), which causes the system destabilized. Moreover, the processing time depends on not only the batch interval and execution parallelism but also the available resources of the process engine.

B. Requirements for Minimizing End-to-End Latency

To make sure that the batched stream processing system handles all workloads under any operating situations (e.g., different cluster, unpredictable data surge), the desired interval needs to be chosen either by offline profiling, or by sufficient resource provisioning. However, offline profiling is vulnerable to any change of available resources and workloads. It is also hard to provision enough resources to handle unpredictable data surges, which is also not necessary for normal conditions that may dominate most of the running time. Based on the definition of end-to-end latency, it is clear that the batch interval and execution parallelism have a significant impact on the latency. To minimize the latency, a small batch interval is preferred since

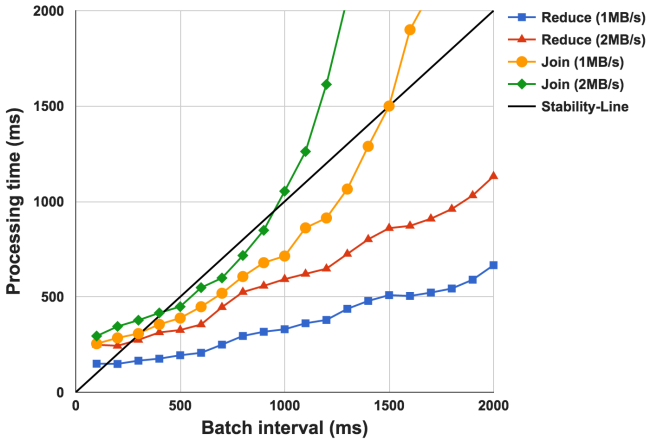


Figure 2. The relationships between processing time and batch interval of two streaming workloads with respect to batch intervals and data rates. The block interval is fixed to 100 ms.

it means less data to process in the processing stage, which may also lead to less processing time. To avoid indefinitely increasing waiting time, the desired batch interval is also required to be larger than the processing time. Therefore, the first requirement for minimizing latency is keeping the stability of the system, which essentially means on average the processing time of a batch should be less than the batch interval. With a fixed batch interval, larger execution parallelism may incur less processing time and consequently reduce the total latency, but it is needless to be true since larger parallelism also accompanies other overhead for task creation and communication. Thus, to minimize the latency, the second requirement is to identify how batch interval and execution parallelism affect the performance.

Static batch interval and execution parallelism cannot maintain the stability and minimize the latency. In this work, we propose to dynamically adjust the batch interval and block interval such that the system is stable as well as the latency is minimized. To find a batch interval that satisfies the stability condition and the optimal execution parallelism that minimizing the latency, we need to identify the relationship between batch interval and processing time as well as the relationship between block interval and processing time. Thus, in the following two sections, we will discuss how the batch interval and block interval affect the latency in the following section.

C. Effect of Batch and Block Interval Sizing on Latency

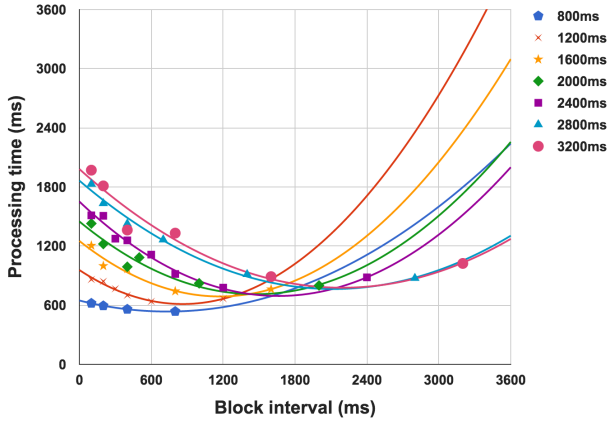
In this section we first show the relationships between block interval and processing time for two representative streaming workloads. Then we further explore how block interval affects the processing time for the same workloads.

The Case for Batch Interval Sizing: Intuitively, the processing time should be a monotonically increasing function of batch interval. Moreover, the exact relationship could be any monotonically functions (e.g., linear, super-liner, exponential), that depends on the characteristics of workload, input data rate, and the processing engine. Thus, we choose two representative streaming workloads in the real world to explore the relationship between batch interval and processing time. The first workload is *Reduce*, which aggregates the received data based on keys. In the following sections of this paper, we used networked word count as an example of *Reduce* workload. The second one is *Join*, which joins two different data streams.

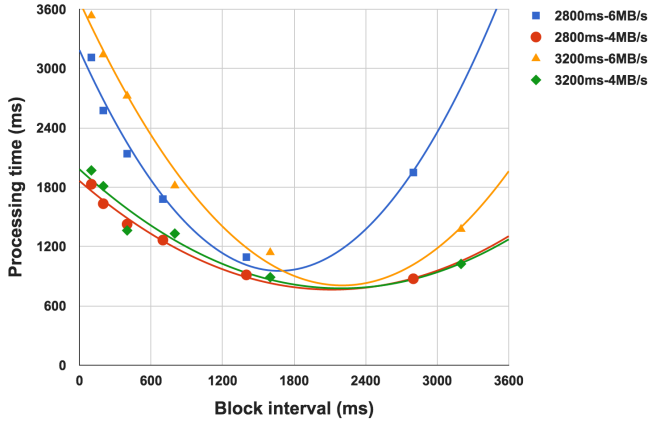
Figure 2 illustrates the relationship between batch interval and processing time for *Reduce* and *Join* workloads with different data ingestion rates, in which case the block interval is set to 100 ms. Basically, it is the linear relationship between batch interval and processing time for *Reduce* workload and superlinear for *Join* workload. The area below the *stability-line* is the stable zone where the batch interval is larger than processing time. Note that with a longer batch interval, *Reduce* has more stable operating status (i.e., batch interval is much larger than processing time), while the stable operating condition for *Join* is limited as the larger batch interval leads to unstable operating zone. For both workloads, the ideal batch interval is the smallest batch interval that meets the stability conditions. For linear workload (i.e., *Reduce*), there will be only one point of intersection, while for superlinear workload (i.e., *Join*), multiple intersection points may exist. For superlinear case, we expect our adaptive algorithm is able to find the lowest intersection point.

The Case for Block Interval Sizing: As we discussed, the batch interval and block interval determine the execution parallelism, which significantly affects the performance on processing time. Figure 3 illustrates the effect of block sizing on processing time for *Reduce* workload. Figure 3(a) shows the relationship between block interval and processing time for different batch intervals. We applied quadratic regression, and the optimal block interval is achieved around the extreme point of that parabola. For the same batch interval, the optimal block interval with different data ingestion rate may be the same (i.e., 3200ms-6MB/s and 3200ms-4MB/s) or different (2800ms-6MB/s and 2800ms-4MB/s) as shown in Figure 3(b). Note that we do not claim that the relationship between block interval and processing time is a quadratic function. Undoubtedly, the optimal block interval varies along with batch interval and data rate. For *Join* workload, it has similar relationships as *Reduce* workload. Being aware of above observations, for a given batch interval that meets the stability condition, we can further reduce the end-to-end latency by using a block interval that minimize the processing time.

To this point, we have explored how batch and block sizing affect the processing and consequently the end-to-end latency. In the next section, we introduce our online adaptive batch- and block-sizing algorithm that is designed according to these insights.



(a) The relationship between block interval and processing time with their corresponding trend lines (Quadratic) for *Reduce* workload with respect to batch intervals. The data rate is 4MB/s.



(b) The optimal block interval varies along with batch interval and data rate. The number pair in the legend represents batch interval and data ingestion rate, respectively.

Figure 3. The effect of block sizing on processing time for *Reduce* workload.

III. DyBBS: Dynamic Block and Batch Sizing

In this section, we first address the problem statement that describes the goal of our control algorithm and remaining issues in existing solutions. Then we introduce how we achieve the goal and address the issues with isotonic regression and heuristic approach. Finally, we present the overall algorithm.

A. Problem Statement

The goal of our control algorithm is to minimize the end-to-end latency by batch and block sizing while ensuring the system stability. The algorithm should be able to quickly converge to the desired batch and block interval and continuously adapt batch and block intervals based on time-varying data rates and other operating conditions. We also assume the algorithm has no prior knowledge of workload characteristics. Compared to the works in literature, we adapt both batch interval and block interval simultaneously, which is the most challenging part in the algorithm design. In addition to this major goal, we also wish to solve several issues in current solutions:

- *Overestimation*: Generally, a configurable ratio (e.g. $\rho < 1$) is used to relax the convergence requirements to cope with noise. In this case, a control algorithm converges when the processing time is $\rho \times batch\_interval$, and the gap between batch interval and processing time (i.e., $(1-\rho) \times batch\_interval$ increases linearly along with the batch interval. This overestimation induces non-negligible increment on latency;
- *Delay response*: In ideal case, the waiting time is small and the algorithm adapts the workload variations in near real-time manner. However, when the waiting time is large (e.g., few times of the batch interval), the statistics of the latest completed batch used to predict new batch interval is already out of date, which usually cannot reflect the immediate work load conditions. This long loop delay may temporarily enlarge the waiting time and latency;

To achieve the major goal and address the issues, we introduce batch sizing using isotonic regression and block sizing with heuristic approach, which are explained in detail in following two sections, respectively. Moreover, we concentrate the algorithm design for *Reduce* and *Join* workloads. The case for other workloads is discussed in Section VI.

B. Batch Sizing with Isotonic Regression

First we look at online control algorithms that model and modify a system and at the same time suffer the trade-off between learning speed and control accuracy. The more information the algorithm learns, the higher accuracy it achieves but also requires longer convergence time. We chose an algorithm with slow convergence time in exchange for high accuracy. The reason here is that compared to record-based stream processing system, batched stream system has relatively loose constraints in terms of convergence speed since the time duration between two consecutive batches is at least the minimal batch interval that is available in that system. The control algorithm can learn and modify the system during that short duration.

Given that a time-consuming regression based algorithm can be used, an intuitive way to model the system is to directly use linear or superlinear function to fit a curve with statistics of completed batches. However, it requires prior knowledge of workload which violates our intention. As shown in Section II, the processing time is a monotonic increasing function of batch interval for both *Reduce* and *Join*. We chose a well-known regression technique, Isotonic Regression [22], which is designed to fit a curve where the direction of the trend is strictly increasing. A benefit of isotonic regression is that it does not assume any form for the target function, such as linearity assumed by lin-

ear regression. This is also what we expect that using a single regression model to handle both *Reduce* and *Join* workloads. Another benefit of using regression model is that the overestimation can be eliminated since we can find the exact lowest intersection point as long as the fitting curve is accurate enough. However, in reality noisy statistics may affect the accuracy of the fitting curve, and thus we still need to cope with noisy behavior. Compared to ratio based (e.g., $processing\_time < \rho \times batch\_interval$) constraint relaxing, we use a constant (e.g., c) to relax the constraint, that is $processing\_time + c < batch\_interval$. With a constant difference between processing time and batch interval, the algorithm can converge as close as possible to the optimal point even with a large batch interval, in which case the ratio based method introduces a large overestimation.

The optimal point is achieved at the lowest point of intersection between stability-line and workload specified line. We first fit a curve with isotonic regression using the gathered statistics (i.e., a batch's interval and its corresponding processing time) of completed batches that have the same block interval. Suppose the regression curve is $IsoR(x)$, where x is the batch interval, and $IsoR(x)$ is the processing time estimated with the regression model. Then we identify the lowest point of intersection by finding the smallest x such that $IsoR(x) + c < x$. Note that we do not need to examine all data ingestion rate to find the intersection point. We only fit the curve for the immediate data ingestion rate that is the same rate in the latest batch statistics as we assume the data ingestion rate keeps the same in the near future (within the next batch interval).

C. Block Sizing with Heuristic Approach

The block interval affects the end-to-end latency via its impact on execution parallelism. As shown in Section II, with a fix block interval, the best case a control algorithm can reach is actually a local optimal case in the entire solution space, which is far from the global optimal case that can significantly further reduce the end-to-end latency. Therefore, we wish our algorithm were able to explore all solution space through block sizing. As aforementioned, the relationship between block interval and processing time is not quadratic although we use it as the trend line in above results. Even though this relationship is quadratic, our control algorithm still needs to enumerate all possible block intervals to find the global optimal case, which is the same as brute-force solution. Therefore, we propose a heuristic approach performing the block sizing to avoid enumeration and frequent regression computation.

Similar to the batch sizing, with different data ingestion rates, the curves of processing time and block interval are also identical. Thus, we use the same strategy used in batch sizing, which is that for each specific data rate we use the same heuristic approach to find the global optimal point. Basically, this heuristic approach starts with the minimal block interval and gradually increase the block interval size until we cannot benefit from larger block interval. The detailed heuristic approach is described as following: i) Starting with the minimal batch interval, we use isotonic regression to find the local optimal point within that block interval (i.e., only applying batch sizing with a fix block interval); ii) Then increase the block interval by a configurable step size and apply the batch sizing method until it converges; and iii) If the end-to-end latency can be reduced with new block interval, then repeat step ii; otherwise the algorithm reaches the global optimal point. Besides that, we also track all local optimal points for all block intervals that have been tried. Note that with the same data ingestion rate and operating condition, the global optimal point is stable. When the operation conditions change, the global optimal point may shift, in which case the algorithm needs to adapt to the new condition by repeating the above three steps. However, restarting from scratch slows down the convergence time, thus we choose to restart the above heuristic approach from the best suboptimal solution among all local optimal points to handle operating condition changes.

Up to this point, we have addressed batch sizing that leverages isotonic regression and block sizing with a heuristic approach. The next step is to combine these two parts to form a practical control algorithm, which is discussed in next section.

D. Our Solution - DyBBS

Here we introduce our control algorithm - *Dynamic Block and Batch Sizing* (DyBBS) that integrates the discussed isotonic regression based batch sizing and heuristic approach for block sizing. DyBBS uses statistics of completed batches to predict the block interval and batch interval of the next batch to be received. Before we explain the algorithm, we first introduce two important data structures that are used to track the statistics and current status of the system. First, we use a historical table (denoted as $Stats$ in the rest of this paper) with entries in terms of a quadruple of $(data\_rate, block\_intvl, batch\_intvl, proc\_time)$ to record the statistics, where the $data\_rate$ is the data ingestion rate of a batch, $block\_intvl$ is the block interval of a batch, $batch\_intvl$ is the batch interval of a batch, and $proc\_time$ is the processing time of a batch. The $proc\_time$ is updated using a weighted sliding average on all processing times of batches that have the same $data\_rate$, $block\_intvl$, and $batch\_intvl$. Second, we track the current block interval (denoted as $curr\_block\_intvl$ in the remaining paper) for a specific data ingestion rate by mapping $data\_rate$ to its corresponding $curr_block\_intvl$, which indicates the latest status of block sizing for that specific data rate, and this mapping function is denoted as DR_to_BL.

Algorithm 1 presents the core function that calculates the next block and batch interval. The input of DyBBS is the $data\_rate$ that is the data ingestion rate observed in

Algorithm 1 ***DyBBS***: Dynamic Block and Batch Sizing Algorithm

Require: $data\_rate$: data ingestion rate of last batch
1: **Function** *DyBBS*($data\_rate$)
2: curr_block_intvl = DR_to_BL($data\_rate$)
3: (block_intvl, proc_time)[] = Stats.getAllEntries($data\_rate$, $curr\_block\_intvl$)
4: new_batch_intvl = IsoR(($block\_intvl, proc\_time$)[])
5: proc_time_of_new_batch_intvl = Stats.get($data\_rate$, $curr\_block\_intvl$, new_batch_intvl)
6: **if** proc_time_of_new_batch_intvl + $c <$ new_batch_intvl **then**
7: **if** curr_block_intvl is the same of block interval of current global optimal point **then**
8: new_block_intvl = curr_block_intvl + block_step_size
9: **else**
10: new_block_intvl is set to the block interval of the best suboptimal point
11: **endif**
12: **else**
13: new_block_intvl = curr_block_intvl
14: **endif**
15: Update(new_block_intvl, new_batch_intvl)
16: **EndFunction**

the last batch. First, the algorithm gets the block interval for the $data\_rate$. Secondly, all the entries with the same $data\_rate$ and $curr\_block\_intvl$ are extracted, which are used for function $IsoR$ to calculate the optimal batch interval, as shown on Line 3 and 4. Thirdly, the algorithm compares the estimated new_batch_intvl and the sliding averaged $proc\_time$. We treat the isotonic regression result converges as if the condition on Line 6 is true and then the algorithm will try different block interval based on operating condition. If the $curr\_block\_intvl$ is the same as that of the global optimal point stated on Line 7, then increase the block interval by certain step size. Otherwise, we treat it as if the operating condition changing and choose the block interval of the suboptimal point. If the condition on Line 6 is not satisfied (Line 12), it means the $IsoR$ for $curr\_block\_intvl$ has not converged and keep the block interval unchanged. Finally, DyBBS updates the new block interval and batch interval. Note that the new_batch_intvl is always rounded to a multiplier of the new_block_intvl, which is omitted in the above pseudo code. In addition, when the $IsoR$ is called, if there is less than five samples points (i.e., the number of unique batch intervals is less than five), it will return a new batch interval using slow-start algorithm [24].

In this section, we have introduced our adaptive block and batch sizing control algorithm and showed that it can handle the overestimation and long delay issues. We will discuss the implementation of DyBBS on Spark Streaming.

IV. Implementation

We implement our dynamic block- and batch-sizing algorithm in Spark Streaming (version 1.4.0). Figure 4 illustrates the high-level architecture of the customized Spark Streaming system. We modified the *Block Generator* and *Batch Generator* modules and introduced the new *DyBBS* module such that Spark Streaming can generate different size blocks and batches based on the control results. First, the *Block Generator* generates blocks with block interval, and the blocks are handed off to the *Batch Generator*. Then,

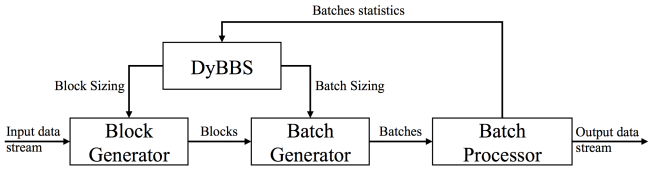


Figure 4. High-level overview of our system that employs DyBBS.

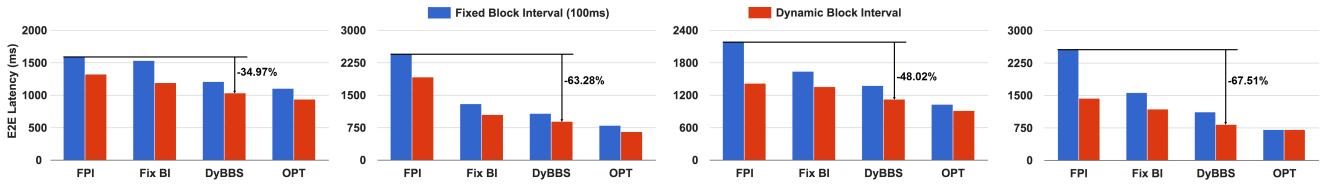
blocks are wrapped into batches, and the batches are put in the batch queue. Finally, the batches are processed in the *Batch Processor*. The *DyBBS* module leverages the statistics to predict the block interval and batch interval for next batch.

There are three configurable parameters used in our algorithm. The first one is constant c used to relax the stability condition. A larger c keeps the system more stable and more robust to noisy, but it also leads to a larger latency. The second parameter is a block interval incremental step size. A larger step size incurs fast convergence speed on block sizing procedure. However, it may also never converge to the global optimal case due to too large of a step size. The last parameter is used for data ingestion rate discretization. In DyBBS, the isotonic regression is conducted on batch interval and processing time for a specific data rate. In our implementation, we choose 50ms as the constant value c, 100ms for the block step size, and 0.2MB/s for the data rate discretization.

In summary, our modifications do not require any changes to the user's program or programming interfaces. Although our implementation is Spark Streaming specified, it is easy to implement our control algorithm on other batched stream processing systems that have similar design principle of block and batch.

V. Evaluation

We evaluate the DyBBS with *Reduce* and *Join* streaming workloads under various combinations of data rates and operating conditions. Our results show that DyBBS is able



(a) Comparison for *Reduce* workload with sinusoidal input.
(b) Comparison for *Join* workload with sinusoidal input.
(c) Comparison for *Reduce* workload with Markov chain input.
(d) Comparison for *Join* workload with Markov chain Input.

Figure 5. The performance comparison of average end-to-end latency over 10 minutes for *Reduce* and *Join* workloads with sinusoidal and Markov chain input rates. Four different approaches are compared, including FPI, FixBI, DyBBS, and OPT.

to achieve latencies comparable to the optimal case. The comparison with the other two practical approaches illustrates that our algorithm achieves the lowest latency. We demonstrate that the convergence speed our algorithm is slower than the state-of-art only for the first time a data rate appears, and for the second and following appearance of the same data rate our algorithm is much faster. Finally, we also show our algorithm can adapt to resource variation.

A. Experiment Setup

In the evaluation, we used a small cluster with four physical nodes. Each node can hold four Spark executor instances (1 core with 2 GB memory) at most. We ran two workloads as mentioned-*Reduce* and *Join*, with two different time-vary data rate patterns. The first data rate pattern is sinusoidal fashion of which the data rate varies between 1.5 MB/s to 6.5 MB/s for *Reduce*, and 1 MB/s to 4 MB/s for *Join*, with periodicity of 60 seconds. The other one is Markov chain [25] fashion of which the data rate changes every 15 seconds within the same rate duration as sinusoidal fashion. For the input data, we generated eight bytes long of random strings as the key for reducing and join operations, and the number of unique key is around $32,000$. The output is written out to a HDFS file [26].

As comparison, we implemented three other solutions that are as follow:

- *FPI* [21]: the state-of-the-art solution, that adopts fix-point iteration (FPI) to dynamically adapt batch interval and is the only practical one in Spark Streaming.
- *FixBI*: We compare our algorithm with this unpractical hand-tuning method. It is the best case (i.e., with the lowest average latency over entire running duration) by enumerating all possible pair of block and batch intervals. However, it still uses static setting over the running time and cannot change either block or batch interval at runtime.
- *OPT*: This is the oracle case, which we dynamically change the block and batch intervals at the runtime based on prior knowledge.

For the performance experiments, we implemented two different versions for all methods: i) *Fixed Block Interval* that changes the batch interval while it keeps the block interval fixed (i.e., 100ms); and ii) *Dynamic Block Interval* that adapts both block and batch intervals. For the FPI that is not designed for block sizing, we set a block interval based on prior knowledge such that it achieves local optimal with the batch interval FPI estimates. To implement the *Fixed Block Interval* algorithm in DyBBS, we disable the block sizing and set block interval to 100ms. The end-to-end latency of a batch is defined as the sum of $batch\ interval$, $waiting\ time$, and $processing\ time$, and we used the average end-to-end latencies over entire experiment duration as the criteria for performance comparison.

B. Performance on Minimizing Average End-to-End Latency

We compared the average end-to-end latency over 10 minutes for *Reduce* and *Join* workloads with sinusoidal and Markov chain input rates. Figure 5 shows that the DyBBS achieves the lowest latencies that are comparable with the oracle case (OPT). In general, by introducing block sizing, the latency is significantly reduced compared to that only applies batch sizing. As Figure 5 showed, our DyBBS outperforms the FPI and FixBI in all cases. Specifically, compared with FPI with fixed block interval, DyBBS with dynamic block interval reduced the latencies by 34.97% and 63.28% for *Reduce* and *Join* workloads with sinusoidal input rate (the cases in Figure 5(a) and 5(b)), respectively. For Markov chain input rates, DyBBS with dynamic block interval reduced the latencies by 48.02% and 67.51% for *Reduce* and *Join* workloads (the cases in Figure 5(c) and 5(d)) respectively, compared to FPI with fixed block interval.

In terms of real-time performance, Figure 6 shows the dynamic behaviors including block interval, batch interval, waiting time, processing time, and end-to-end latency for *Reduce* workload with sinusoidal input, for DyBBS. During the first three minutes (the first three sine waves), our algorithm was learning the workload characteristics, and hence the latency is relatively high. After our algorithm converged (from the fourth minute), the latency is relatively low and maintained the same for the rest of time. In Figure 7, we compared the batch interval and end-to-end latency of FPI and DyBBS. During the first 60 seconds, the end-to-end latency of DyBBS is higher than FPI due to imprecise isotonic regression model. In the next two cycles, the end-to-end latency of DyBBS reduces since the isotonic regression

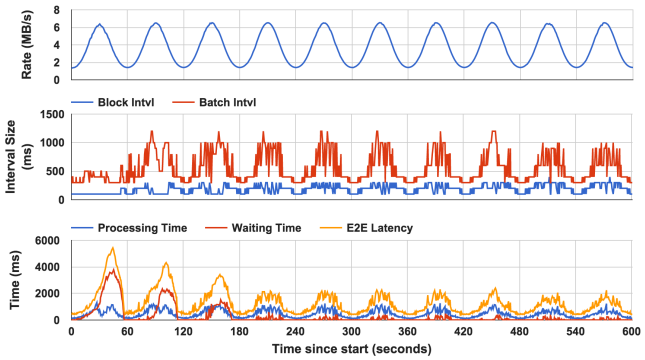


Figure 6. The DyBBS's real-time behavior of block interval, batch interval, waiting time, processing time, and end-to-end latency for *Reduce* workload with sinusoidal input data rate.

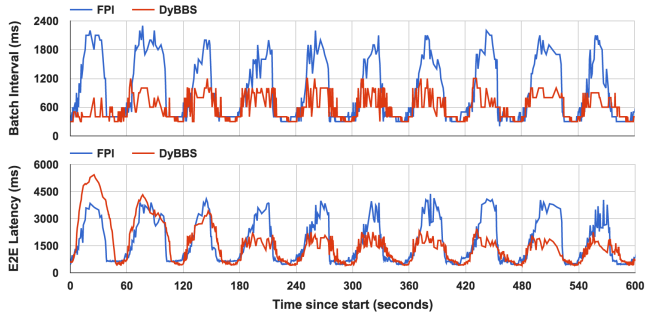


Figure 7. Comparison of batch interval and end-to-end latency of FPI and DyBBS for *Reduce* workload with sinusoidal input data rate.

model is becoming more and more accurate.

C. Convergence Speed

We first compared the convergence speed of *Reduce* workload for FPI and DyBBS with a step changing data rate. Since FPI does not support block sizing, we only present the behavior of batch interval along with the data rate in Figure 8. Before the star time, we run FPI and DyBBS long enough such that both of them have converged with the 1MB/s data rate. At the 5th second, the data rate changes to 3MB/s and that is the first time the rate appears during entire experiment. The FPI first converges roughly after the 11th second (the red vertical line on the left), while DyBBS converges until the 18th seconds (the blue vertical line on the left). This is caused by the longer learning time of isotonic regression used in DyBBS. When the second time that the data rate changes to 3MB/s, in contrary to the first time, DyBBS converges at the 47th second (the blue vertical line on the right), while FPI converges seven seconds later (the red vertical line on the right). Since FPI only uses the statistics of last two batches that means it cannot leverage historical statistics, hence FPI has to re-converge every time the data rate changes. Thus, for a specific data rate, DyBBS has long convergence time for the first time that rate appears

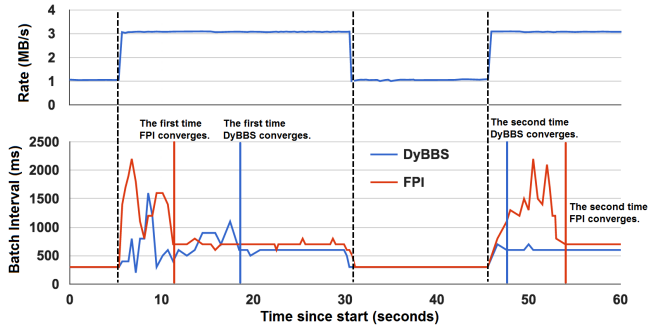
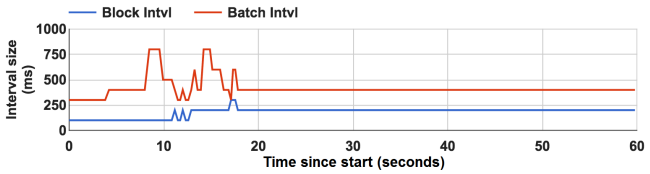
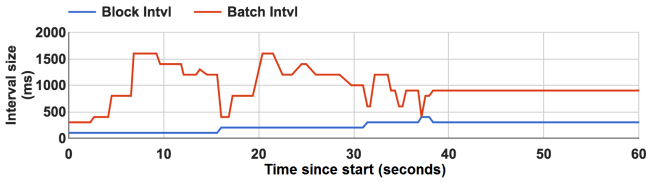


Figure 8. Timeline of data ingestion rate and batch interval for *Reduce* workload using FPI and DyBBS with fixed block interval. Our algorithm spent longer time to converge for the first time when a specific rate occurs, but less time for the second and later times that rate appears.



(a) *Reduce* workload with 3MB/s constant data ingestion rate.



(b) *Reduce* workload with 6MB/s constant data ingestion rate.

Figure 9. Timeline of block interval, batch interval for *Reduce* workload with different data ingestion rates using DyBBS with dynamic block intervals. Larger data ingestion rate leads to longer convergence time.

and relatively small convergence time for the second and rest times the same data rate appears.

With block sizing enabled in DyBBS, the convergence time is longer than that when block sizing is disabled. Figure 9 shows the convergence time for *Reduce* workload with two different constant data rates. With larger data ingestion rate, our algorithm spent more time to search for the optimal point since there are more potential candidates. Although our algorithm spends tens of seconds to converge for large data ingestion rate, this convergence time is relatively small compared to the execution time (hours, days, even months) of long run streaming application. Therefore, we believe that our algorithm is able to handle large data ingestion rate for long run applications.

D. Adaptation on Resource Variation

In above experiments, we have showed that our algorithm is able to adapt to various workload variation. When the operating condition changes, our algorithm should detect the resource variation and adjust the block and batch interval

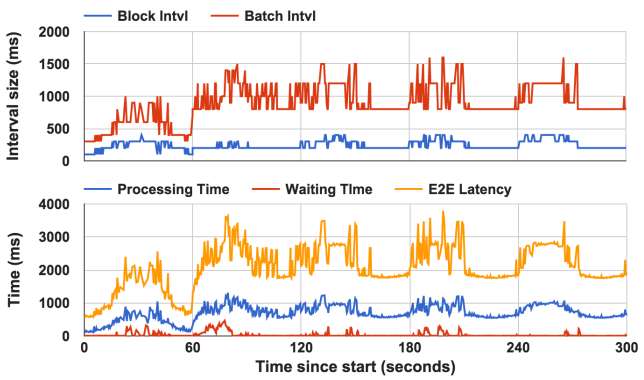


Figure 10. Timeline of block interval, batch interval, and other times for *Reduce* workload with sinusoidal input rate under resource variation. At the 60th second, external job takes away 25% resources of the cluster.

consequently. To emulate the resource reduction, we run a background job on each node such that one core on each node is fully occupied by the background job. Figure 10 illustrates that our algorithm can adapt the resource reduction (at 60th second) by increasing the batch interval and end-to-end latency.

VI. Discussion

In this paper, we narrowed down the scope of our algorithm to two workloads with simple relationships (i.e., linear for *Reduce* workload and superlinear for *Join* workload). There may be a workload with more complex characteristics than *Reduce* and *Join* workloads. An example of such workload is *Window* operation, which aggregates the partial results of all batches within a sliding window. This issue is also addressed in [21], and the *Window* operation is handled by using small mini-batch with 100ms interval. In our approach, there are two issues for handling *Window* workload. One is that it is possible that we cannot find a serial of consecutive batches such that the total time interval exactly equals the window duration. If we always choose the next batch interval such that the previous condition is satisfied, then we lose the opportunity to optimize that batch, as well as all the batches in the future. Another problem is that our approach dynamically changes the block interval, which means our approach cannot directly employ the method used in [21]. Therefore, we leave the work to support *Window* workload for the future. Another limitation is that the scale of our experiment test bed is relatively small. For large scale environment, the solution space is much larger, which takes longer time to converge. In addition, large scale cluster can handle more workload (e.g., larger data ingestion rate), which needs to adjust the granularity of data rate discretization. Next step we plan to explore these features of DyBBS algorithm in large cluster environment.

VII. Related Work

There has been much work on workload-aware adaption in stream processing systems. The work in [21] is the closest one to our work. In [21], the authors proposed adaptive batch sizing for batched stream processing system based on fix point iteration, which targets to minimize the end-to-end latency while keeping the system stable. It changes the batch interval based on the statistics of last two completed batches. This work is also designed for handling *Reduce*- and *Join*-like workloads, but it also handles *Window* workload. Compared to this work, our approach is different from it in two folds: i) Our approach dynamically adapts not only batch interval but also the block interval that is not considered in their work, and our block sizing method shows that it can further reduce the end-to-end latency compared than batch sizing only approach; and ii) For the batch interval prediction, our work utilizes all historical statistics so that the batch interval prediction is much accurate. Furthermore, our algorithm also eliminates the overestimation and control loop delay.

For continuous operator based stream processing system, DAG based task scheduling is widely used in systems including Dryad [16], TimeStream [3], MillWheel [10], Storm [2], and Heron [6]. One approach used to achieve elasticity in DAG is graph substitution. In DAG (or a sub graph of DAG), graph substitution requires that the new graph used to replace is equivalent to the one being replaced, and two graphs are equivalent only if they compute the same function. This approach is used in Dryad, TimeStream and Apache Tez [17]. A similar elastic resource adjustment is based on the operators' process [27]–[29]. All of those methods identify the bottleneck of a system based on resource utilization. After that, the system increases the parallelism of the bottleneck nodes via either splitting (e.g. increase map operators) or repartition (e.g. increase shuffle partitions). In our work, we dynamically change the execution parallelism by block interval sizing, which requires no resource provisioning and partition functions. Load shedding discards part of received data when the system is overloaded [18]–[20], which is a lossy technique. Our approach does not lose any data as long as the system can handle the workload with certain batch interval while keeping the system stable. However, the presence of the load shedding technique is necessary when the processing load exceeds the capacity of system.

VIII. Conclusion

In this paper, we have illustrated an adaptive control algorithm for batched processing system by leveraging dynamic block and batch interval sizing. Our algorithm is able to achieve low latency without any workload specific prior knowledge, which is comparable to the oracle case due to our accurate batch interval estimation and novel execution parallelism tuning. We have shown that compared with the state-of-the-art solution, our algorithm can reduce the

latency by at least 34.97% and 63.28% for *Reduce* and *Join* workloads, respectively. We have also presented the abilities of DyBBS to adapt to various workloads and operating conditions.

REFERENCES

[1] Apache, "Apache storm," https://storm.apache.org/, [Online; accessed June 12, 2015].

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[3] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 1–14.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[5] Apache, "Apache samza," http://samza.apache.org/, [Online; accessed June 12, 2015].

[6] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[7] IBM, "Ibm infosphere streams," www.ibm.com/software/products/en/infosphere-streams.

[8] Apache, "Apache storm trident," https://storm.apache.org/documentation/Trident-tutorial.html, [Online; accessed June 12, 2015].

[9] L. Hu, K. Schwan, H. Amur, and X. Chen, "Elf: efficient lightweight fast stream processing at scale," in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 25–36.

[10] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[11] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 63–74.

[12] 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*. ACM, 2013, pp. 423–438.

[13] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[17] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1357–1369.

[18] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 37–48.

[19] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Y. He, and S. Elnikety, "Measuring and managing answer quality for online data-intensive services," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 167–176.

[20] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 159–170.

[21] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[22] H. Brunk, R. Barlow, D. Bartholomew, and J. Bremner, "Statistical inference under order restrictions.(the theory and application of isotonic regression)," DTIC Document, Tech. Rep., 1972.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, p. 10.

[24] W. R. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," 1997.

[25] "Markov chain," https://en.wikipedia.org/wiki/Markov_chain.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[27] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1447–1463, 2014.

[28] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[29] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*. IEEE, 2006, pp. 71–71.