Adaptive Stream Processing using Dynamic Batch Sizing

Tathagata Das
Yuan Zhong
Ion Stoica
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-133
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-133.html

June 3, 2014

Acknowledgement

Many thanks to Yuan Zhong, Ion Stoica and Scott Shenker for making this thesis possible. Also thanks to David Zats, Shivaram Venkataraman, and Neeraja Yadwadkar for providing feedback on earlier versions of the text.

Finally, a very special thanks to both my advisers, Scott Shenker and Ion Stoica, for guiding me through my ups and downs in life and putting up with my idiosyncrasies.

Adaptive Stream Processing using Dynamic Batch Sizing

by
Tathagata Das
tdas@eecs.berkeley.edu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Scott Shenker
Research Advisor

(Date)

* * * * * * *

Professor Ion Stoica
Second Reader

(Date)

Acknowledgements

Many thanks to Yuan Zhong, Ion Stoica and Scott Shenker for making this thesis possible. Also thanks to David Zats, Shivaram Venkataraman, and Neeraja Yadwadkar for providing feedback on earlier versions of the text.

Finally, a very special thanks to both my advisers, Scott Shenker and Ion Stoica, for guiding me through my ups and downs in life and putting up with my idiosyncrasies.

Abstract

The need for real-time processing of "big data" has led to the development of frameworks for distributed stream processing in clusters. It is important for such frameworks to be robust against variable operating conditions such as server failures, changes in data ingestion rates, and workload characteristics. To provide fault tolerance and efficient stream processing at scale, recent stream processing frameworks have proposed to treat streaming workloads as a series of batch jobs on small batches of streaming data. However, the robustness of such frameworks against variable operating conditions has not been explored.

In this paper, we explore the effect of the size of batches on the performance of streaming workloads. The throughput and end-to-end latency of the system can have complicated relationships with batch sizes, data ingestion rates, variations in available resources, workload characteristics, etc. We propose a simple yet robust control algorithm that automatically adapts batch sizes as the situation necessitates. We show through extensive experiments that this algorithm is powerful enough to ensure system stability and low end-to-end latency for a wide class of workloads, despite large variations in data rates and operating conditions.

1 Introduction

Complex real-time processing of "big data" has become increasingly important. Many systems need to process large volumes of live data and take actions based on the results as soon as possible. For example, a social network may wish to quickly find trending conversation topics, and a content distribution network may wish to monitor its distribution system in real time. Such workloads require large clusters to process the data as soon as it is received. This has led to the development of many distributed stream processing frameworks [4, 15, 16, 17, 24].

Besides scalability, fault-tolerance and low latency, another important requirement in distributed stream processing systems is robustness against variations in streaming workloads. For example, a social network wishing to find trending conversation topics using a stream processing system would like the system to be robust to surges in social activity. Similarly, a content distribution network would like its distribution system to adapt quickly to sudden spikes in the content demands. Furthermore, server faults may suddenly reduce the available processing resources and a stream processing system should be able to adapt automatically.

Every stream processing system makes architectural choices based on its desired performance, fault-resilience and consistency properties. Recently proposed frameworks treat streaming processing as a continuous series of MapReduce-style batch processing jobs on batches of received data [13, 24]. This model leverages the fault-tolerance properties of the MapReduce [12] processing model to allow faster fault-recovery (parallel recovery in [24]) and straggler mitigation (speculative execution [12]). This enables efficient stream processing at scale. However, the robustness of this processing model against changes in data rates and operating conditions has not be well explored. In this work, we analyze the effect of batch size on processing rates and the ability of a system to automatically adapt the batch size to such changes.

The size of the batches can have a significant effect on the throughput and the end-to-end latency of the system. Depending on the nature of the workload, larger batches of data may allow the system to process data at higher rates. However, larger batch sizes also increase the end-to-end latency between receiving a data record and getting the results generated from it. Hence, it is necessary for the system to operate at a batch size that minimizes latency while ensuring that the data is processed as fast as it is received. Furthermore, this desired batch size varies with data rates and other operating conditions. Therefore, a statically set batch size may either incur unnecessarily high latency under low load, or may not be enough to handle surges in data rates, causing the system to destabilize.

To address these issues, we propose an online adaptive algorithm that allows the system to automatically adapt the batch size as operating conditions change. Developing such an algorithm is challenging. The throughput of a streaming workload can behave non-linearly with respect to the batch size. Despite this, given any workload, the algorithm must be able to quickly adapt to changes and provide low latency. Furthermore, the algorithm must be robust with respect to noisy operating conditions that arise from continuous variations in the data rates, available resources, etc.

To develop this algorithm, we made an intuitive observation that applies to a wide range of

workloads – the processing time of a batch increases smoothly and monotonically with the batch size. This allowed us to design our online adaptive algorithm based on Fixed-Point Iteration [2], a well-known numerical optimization technique. By using job statistics of prior batches, our algorithm continuously learns and adapts in order to provide low latency while maintaining system stability.

We demonstrate our algorithm's efficacy for a wide range of workloads. Our contributions are as follows.

- With absolutely no prior knowledge of a workload's characteristics, our algorithm is able to achieve latency that is comparable to the minimum latency achievable by any statically configured batch size.
- It is able to quickly adapt the batch size under changes in data rates, workload behavior and available resources.
- It is simple and requires no workload-specific tuning.

In the rest of the paper, Section 2 discusses the relationship between the batch size and the performance of a streaming workload, and argues for the necessity of dynamic sizing of batches. Section 3 first presents some of our initial unsuccessful approaches and then explains in detail our algorithm based on the fixed-point iteration technique. Section 4 details our implementation of the algorithm. We evaluate our algorithm in Section 5 and discuss some of the finer details in Section 6. Section 7 discusses related work and Section 8 concludes by summarizing our contributions.

2 The Case for Dynamic Batch Sizing

In this section, we argue for the need to dynamically adapt the batch size of a streaming computation as operating conditions vary. To this end, we first give a brief description of the simple system model that we use to characterize batched stream processing systems. Then we discuss the effects of batch size on the performance of streaming workloads and discuss the drawbacks of a statically defined batch sizes. Finally, we highlight our goals and the challenges involved in dynamically adapting the batch size that works across a wide range of workloads.

2.1 System Model

Figure 1 illustrates a simple system model that we use to understand the behavior of a batched stream processing system. The batching module is responsible for dividing the data streams into batches containing data received in discrete time intervals. The length of this interval is the *batch size*. For example, if the system is configured to process data in 1-second batches, then all the data received in a 1-second interval will form a new batch of data to be processed. Batches of data from the batching module is pushed to a batch queue. The processing module dequeues batches from the queue and processes them one by one.

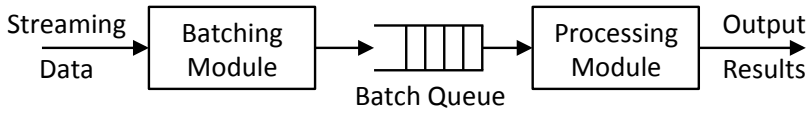


Figure 1: Model of a batched stream processing system

It is important to note that we choose to define size of batches in terms of time intervals. Batch size may also be defined in terms of the data size – for example, every 10 MB of received data becomes a batch. However, this significantly complicates the system model. First, if a streaming workload has multiple data streams with different data rates, then synchronizing the batch boundaries becomes more complicated as each stream would generate batches at different intervals. Second, we shall see shortly that important performance metrics such as end-to-end latency and system throughput are naturally characterized in terms of time intervals. We shall henceforth use the term *batch interval* instead of batch size to avoid confusion.

2.2 Effect of Batch Interval on Performance

Here we discuss the effect of batch interval on the latency and the throughput of a streaming workload.

2.2.1 Effect on Latency

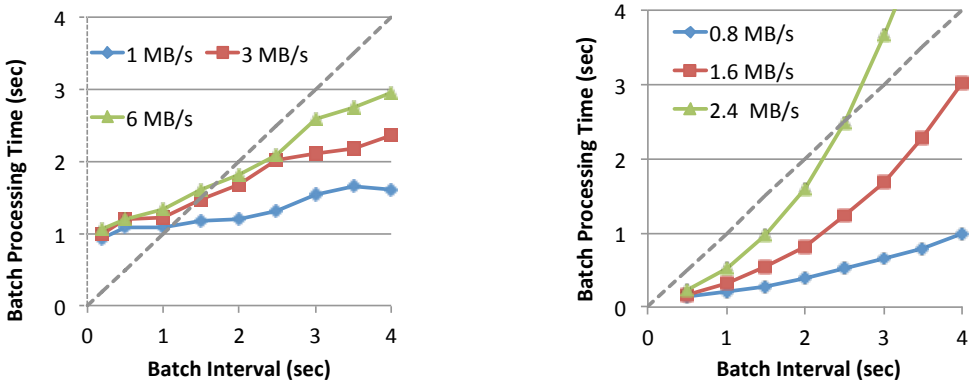
We define the end-to-end latency of a streaming workload as the duration between the time when a data record enters the system and the time when the results corresponding to that record is generated. Based on the aforementioned system model, this latency can be calculated as the sum of the following three terms.

- *Batching delay*: The duration between the time a data record is received by the system and the time that it is sent to the batch queue – this quantity is upper bounded by the corresponding batch's interval;
- *Queuing delay*: The amount of time that the batch spends waiting in the batch queue;
- *Processing time*: The processing time of the batch. Note that this is also dependent on the corresponding batch's interval.

It is clear that the latency directly depends on the batch interval – lower batch interval leads to lower latency. Hence, it is desirable to keep the batch interval as low as possible. However, it is also necessary to ensure that the stability of the streaming workload. This is discussed next.

2.2.2 Effect on Throughput

Intuitively, a streaming workload can be stable only if the system can process data as fast as the data is being receiving. In case of batched stream processing systems, the sufficient condition for



(a) Reduce **(b)** Join

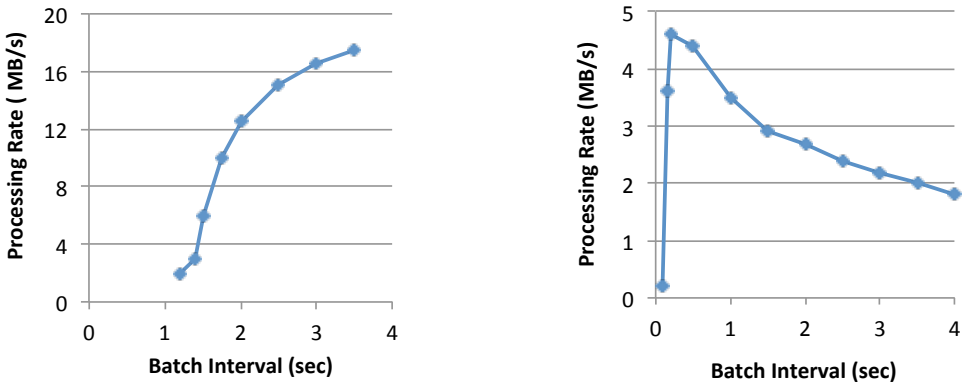
Figure 2: Behavior of the processing time of two streaming workloads with respect to batch interval and data rate.

stability is that the processing time of batches must not exceed the batch interval, so that each batch is completely processed by the time next batch arrives. Failure to do so leads to building up of the batch queue.

Typically, the processing time monotonically increases with the batch interval – larger interval implies more data to process, which leads to a higher processing time. However the exact relationship between the batch interval and processing time can be non-linear. In fact, this relationship heavily depends on (i) the characteristics of the processing system, (ii) the available resources for processing, (iii) the nature of the workload, and (iv) the data ingestion rates. Any variations in these can change the behavior of processing time with respect to batch interval.

This complex behavior is illustrated in Figure 2. It shows the processing time of two different streaming workloads against various batch intervals for different data rates. First one, named *Reduce*, is based on a streaming aggregation. The second one, named *Join*, joins two batches of data received from two separate data streams. The details of these two workloads will be explained in Section 5.1.

Note the stability condition line (i.e., batch processing time = batch interval) that identifies the stable operating zone. Any batch interval whose processing time is below this line will be stable and vice versa. *Reduce* has a roughly linear behavior with respect to batch interval and higher intervals leads to more stable operation (i.e., below the stability line). *Join*, however, has a distinctly superlinear behavior. This is because joining two datasets can potentially generate $O(M \times N)$ records (where N and M are number of records in each dataset), and as the batch interval of both data streams are varied simultaneously, the resultant number of records and processing time can increase superlinearly. Therefore, batch interval needs to be kept low to achieve stability. This leads to a completely non-monotonic behavior between the batch interval and the maximum processing rate that can be sustained, as shown in Figure 3.



(a) Reduce **(b)** Join

Figure 3: Behavior of the maximum processing rate of two streaming workloads with respect to batch interval.

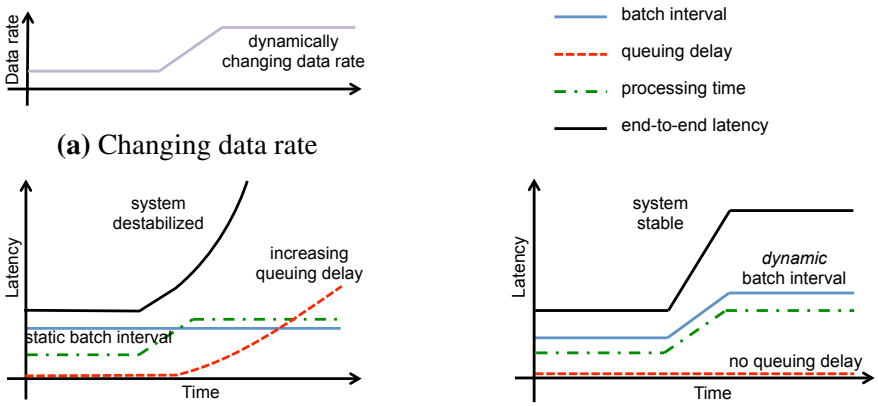
2.3 The Cost of a Static Batch Interval

Having understood the effect of the batch interval on performance, it is clear that the batch interval needs to be carefully chosen to achieve both goals – stability and low latency. A plausible strategy to determine such a batch interval is to apply offline learning. Given a streaming workload and its processing resources, a characteristic profile of the workload may be developed that predicts the batch processing times and end-to-end latencies with respect to different batch intervals and data ingestion rates. Accordingly, an interval can be chosen that best balances the end-to-end latency requirements and the maximum ingestion rate that is expected.

However, this method has significant limitations.

- Any profile developed offline would be very specific to the cluster resources (i.e., memory, CPU, network, etc.) as well as workload characteristics that were used to generate the profile. Any changes in the cluster resources (e.g., failed nodes, stragglers, new resources, etc.) or workload characteristics (e.g., changes in the number of aggregation keys, etc.) would change the actual behavior of the workload thus rendering the profile useless.
- Often it is hard to account for unpredictable changes in data ingestion rates. Even large services like Twitter experience outages due to sudden surges in tweet rates during various global events [5]. If such peak loads are accounted in the static batch interval, then this interval may be very large. Correspondingly, the latency will be needlessly high for normal conditions defeating the original goal of achieving low latency.

Figure 4 schematically illustrates a single scenario of how the system destabilizes with a statically set batch interval when the operating conditions change. As the data rate increases (Figure 4a), the processing time increases as well (dash-dot line in Figure 4b). When it exceeds the static batch interval, the batches start getting queued, causing the queuing delay (dashed line) and the end-to-end latency (solid black line) to build up indefinitely. As a result, the system destabi-



(a) Changing data rate

(b) Instability with static batch interval

(c) Stability with dynamic batch sizing

Figure 4: Instability of statically defined batch interval vs stability of dynamic batch sizing when data ingestion rate changes.

lizes.

In summary, a statically defined batch interval may neither minimize the latency in the current operating conditions, nor ensure system stability when operating conditions change. Thus, it is important to dynamically adapt the batch interval as the operating conditions demand.

2.4 The Goal of Dynamic Batch Sizing

Figure 4c illustrates our goal. As the data ingestion rate increase, we want to detect the change and accordingly adjust (in this case, increase) the batch interval such that the stability condition is maintained. Note that there will be no buildup of queuing delay and the system can remain stable at the higher data rate, although with a higher latency.

In order to make this dynamic sizing possible, we propose to introduce a control loop between the processing module and the batching module. The control module, as shown in Figure 5, is going to receive statistics (e.g., processing time, queuing delay, etc.) of completed jobs from the processing module, run a control algorithm based on the statistical data to decide the desired batch interval. After every batch interval is over, the batching module queries the control module for the next batch interval and creates batches accordingly. The goal of this paper is to design the control algorithm that allows this feedback loop to appropriately adapt the batch interval. Next we describe the desirable properties of the algorithm and discuss why it is hard to achieve them.

2.4.1 Desirable Properties of the Control Algorithm

In order to dynamically adapt the batch size of a streaming workload, the control module of the processing system has to continuously estimate the operating condition and accordingly increase

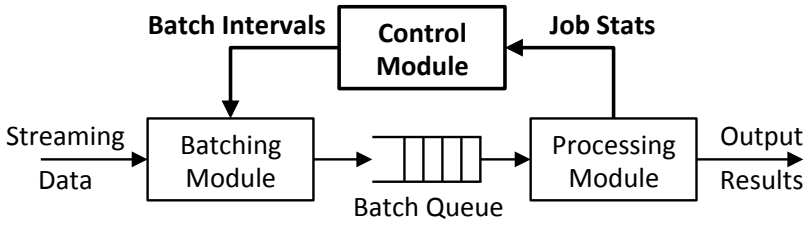


Figure 5: Model of our proposed system with a control loop that dynamically adapts the batch interval based on statistics of completed jobs.

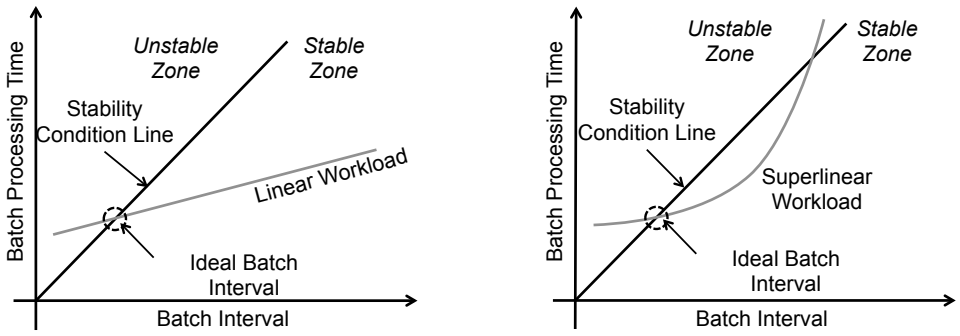
or decrease the batch size. This calls for an online control algorithm. Such an algorithm should have the following desirable properties.

- *Low Latency*: The control algorithm should be able to achieve a low batch interval and hence low end-to-end latency, while ensuring the stability of the system.
- *Agility*: The control algorithm should be able to quickly detect any changes in operating conditions, and quickly converge to the desired batch interval.
- *Generality*: The algorithm should be applicable to any kind of batched stream processing system, and be able to deal with any workload having any kind of processing time versus batch interval relationship. This is important because we do not want the developers implementing their workloads to be concerned about their workloads' behavior and the applicability of the control algorithm on their workloads.
- *Easy to configure*: A control algorithm may have parameters that may need tuning depending on intended operating conditions (e.g., target cluster utilization, etc.). The number of parameters should be small and should be easy for a system administrator to configure.

2.4.2 Why is this Hard?

Achieving the properties outlined above is challenging due to the following reasons.

- *Nonlinear Behavior of Workloads*: As discussed earlier, the relationship between the processing rate and the batch interval can be complex and potentially hard to model by a control algorithm. For example, one can attempt to apply a simple approach that increases the batch interval as the system starts to fall behind and reduces the batch interval when the more resources are available. While this will work for a linear workload like *Reduce*, it will fail to converge for superlinear workloads such as *Join* since increasing the batch interval can actually destabilize the system. This is explained in more detail in Section 3.2.



(a) Linear workloads **(b)** Superlinear workloads

Figure 6: Stability condition and ideal batch duration in two type of workloads.

- *Noise*: A practical processing system may have noisy behavior due to continuous variations in data ingestion rates and cluster conditions. This makes it challenging for a control algorithm to converge to a stable batch interval.
- *Trade-off between Accuracy and Agility*: We are attempting to simultaneously learn the workload's behavior and adapt the batch interval based on it. In such online adaptive algorithms, there is a fundamental trade-off between the rate of learning and the accuracy of the model learned – if it uses more time and data to learn a more accurate system model, it may provide better results but it may be slower to adapt to changes in the model.

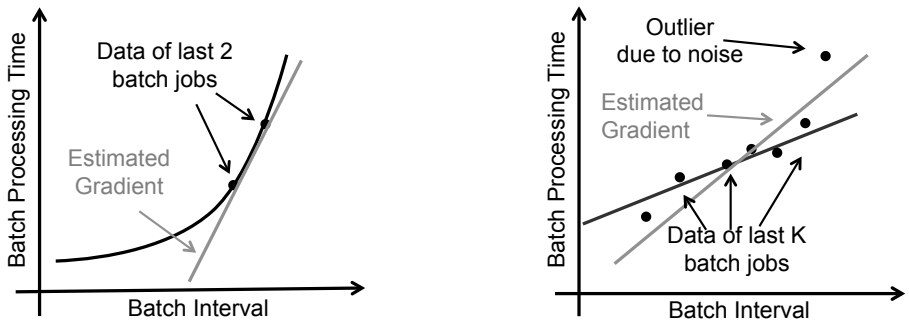
3 Dynamic Batch Sizing

In this section, we first describe in detail our problem formulation. Then we discuss why some intuitive solutions fail to achieve the desired properties. Finally we present our algorithm based on the fixed-point iteration technique.

3.1 Problem Formulation

Our goal is to achieve the minimum batch interval that ensures system stability at all times. This is illustrated in more detail in Figure 6 for two possible workloads [1]. Given the current operating conditions, let us assume that we know the relationship between the processing time and the batch interval. For the linear workload (Figure 6a), it is evident that the desired minimum batch interval that ensures stability is at the intersection (marked with a dashed circle). The superlinear workload (Figure 6b) has two intersections, and the smaller intersection (marked with a dashed circle) is the desired batch interval. We wish our algorithm to quickly locate the desired batch interval without prior knowledge of the workload characteristics. Thus, the algorithm needs to gather information

[1] We restrict ourselves to these two workloads to keep the problem tractable. More details in Section 6.



(a) Gradient estimation using two points

(b) Inaccurate gradient estimation due to noisy statistics

Figure 7: Gradient estimation and its sensitivity to noise.

from past job statistics to learn the characteristics of the workload, and to adapt batch intervals based on whatever it has learned. Furthermore, the characteristics are continuously changing as the data ingestion rates and other operating conditions vary over time. The desired algorithm will continuously adapt to these changes and converge to the right batch interval.

3.2 Why have Some Initial Solutions Failed

Here we discuss some initial approaches toward devising a robust and stable control algorithm with good performance and why they failed to achieved our desired goals. Note that we do not claim any impossibility result – it may be possible to get good results using these initial approaches. We found them to be hard to configure.

3.2.1 Controls Based on Binary Signals

Control algorithms based on binary signals have been used extensively in many different contexts. The general idea is that the controller uses a one-bit information about the current system state to update a parameter that controls the system's performance. The performance must have a monotonic relationship with the parameter. For example, in the context of congestion control in the Internet [11], the additive increase / multiplicative decrease (AIMD) controller uses a binary signal of congestion level (such as packets drops) to control the size of the congestion window. Increasing the window size increases the data throughput of the system (assuming bandwidth is available) and vice versa.

Our goal is very similar – we wish to adapt the batch interval based on stability of the streaming workload. Hence, at a first glance, one can devise a simple control algorithm that increases the batch interval if the operating point is in the unstable zone and vice versa. Even though this approach works for linear workloads, it fails for superlinear workloads, since increasing the batch interval in superlinear workloads can increase processing times such that we enter the unstable

zone and therefore reduce sustainable processing rate (as shown in Figure 3b for the *Join* workload). This non-monotonic behavior in performance makes control systems based on binary signal unsuitable, as without any more information, it is impossible to know whether to increase or decrease the batch interval to achieve the necessary processing rate.

3.2.2 Controls based on Gradient Information

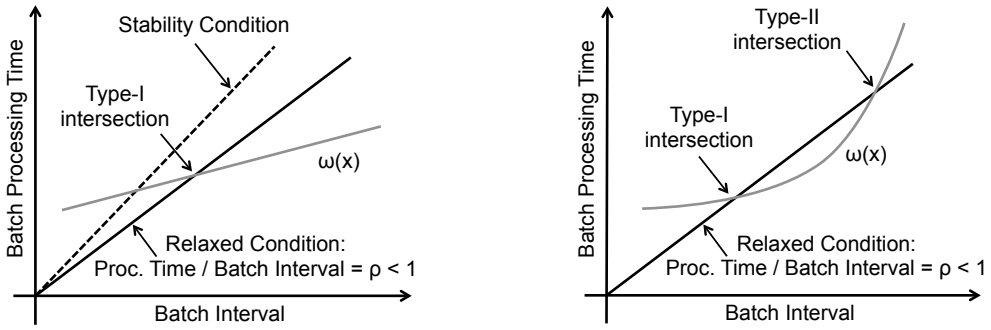
Consider the ideal case in which we know the workload profile completely, and consider the gradient of the processing time with respect to batch interval. If the gradient is high, we are operating near the Type-II intersection. Thus, a plausible approach to building a dynamic control algorithm is to use the statistics of complete jobs to estimate the gradient, and increase/decrease the batch interval accordingly. For example, we can do the following.

- Estimate gradient based on prior batch processing times as shown in Figure 7a.
- If the gradient is large (i.e., very steep), it is likely to be a superlinear workload and near the higher intersection. So, reduce the batch interval by a configured step size.
- Otherwise, if the gradient is small, increase or decrease the batch interval by the configured step size, depending on whether the current operating point is in the stable or unstable zone (similar to the binary control techniques).

Note that this algorithm attempts to identify when the system is operating close to the higher intersection (Figure 6b) based on the gradient and reduces the batch interval.

The simplest way to estimate the gradient is to calculate the slope based on the batch intervals and the processing times of the last two completed jobs (as shown in Figure 7a). In a static environment and for a smooth workload profile, this could be a sufficiently good estimate. However, in practice, in the presence of varying data rates and noise in processing times, the error in the estimate can be large which can make the system apply incorrect changes to the batch interval, sometimes leading to complete destabilization. The estimates can be improved by applying linear regression on more than two data points. However, in our extensive experimentation, the system is still very vulnerable to noise. Figure 7b is an illustration of this vulnerability - one outlier can drastically change the result of the linear regression.

Another practical challenge was to tune the step size of this control algorithms. Too large step sizes It is often not clear what step sizes should be used in these algorithms, and good choices of step sizes depend very much on the specific problem context. If the step sizes are too small, the system takes a long time to converge. In our case, the system may be unable to keep up with changes in data ingestion rates. If the step sizes are too large, while it may converge faster, it may not converge to the desired batch sizes. This issue makes it harder for application developers to tune the control algorithms for their own applications.



(a) Linear workloads **(b)** Superlinear workloads

Figure 8: Relaxed stability condition and 2 types of intersections.

3.3 Our Solution - Fixed-point Iteration

Here we present the details of our control algorithm based on Fixed Point iteration technique that can achieve the desired properties.

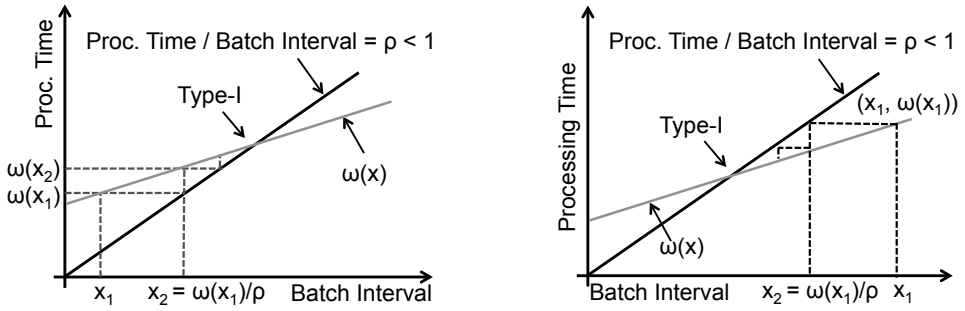
3.3.1 Relaxing the Requirements to cope with Noise

From our initial attempts, we learned that it is very hard to achieve the optimal batch interval and keep the system stable in the presence of continuously changing workload function, noise and other unpredictable variations in operating conditions. At the optimal batch interval (i.e., batch processing time = batch interval), even with slight increases in the processing times due to any change in the operating conditions, the queuing delay will start building up immediately before the controller can adapt, potentially resulting in instability. To hedge against this, we choose our desired batch interval based on the intersection between the workload function $\omega(x)$ and the condition line $batch\ processing\ time\ =\ \rho \times batch\ size$, where $\rho < 1$ is a pre-configured parameter. As illustrated in Figure 8, this allows for a certain slack in the system – increases in processing times will not immediately put the system in the unstable region, thus giving the control algorithm time to adapt the batch interval. Note that this relaxation increases the batch interval that the controller will target. We argue that this is an acceptable trade-off for ensuring stability of the system.

For clarity, we henceforth refer to our desired intersection with the condition line (i.e., our desired batch interval) as *Type-I* intersection and the second intersection in the case of superlinear workloads as *Type-II* intersection. This is also shown in Figure 8. The goal of our algorithm is converge to Type-I intersection in both cases.

3.3.2 Fixed-point Iterations for Type-I intersections

The Fixed-point Iteration method [2] is an iterative algorithm to find the solution x^* of a fixed-point equation $f(x) = x$, for a function f. The algorithm is extremely simple and operates as follows.



(a) Convergence from below **(b)** Convergence from above

Figure 9: Fixed-point iterations for finding Type-I intersection.

1. Start with an initial guess x_1;
2. iterate using $x_{n+1} := f(x_n)$ for $n = 1, 2, ...$

This method of finding fixed points is applicable to a class of functions that satisfy conditions imposed by the Contraction Mapping Theorem [1].

In our system, we wish to find the point x^* so that $\omega(x^*) = \rho x^*$ (x represents the batch interval). For the purpose of this explanation, let us assume we *know* the function $\omega(x)$. Finding the Type-I intersection is essentially solving the fixed point of the function $f(x) = \omega(x)/\rho$, so that $f(x^*) = \omega(x^*)/\rho = x^*$.

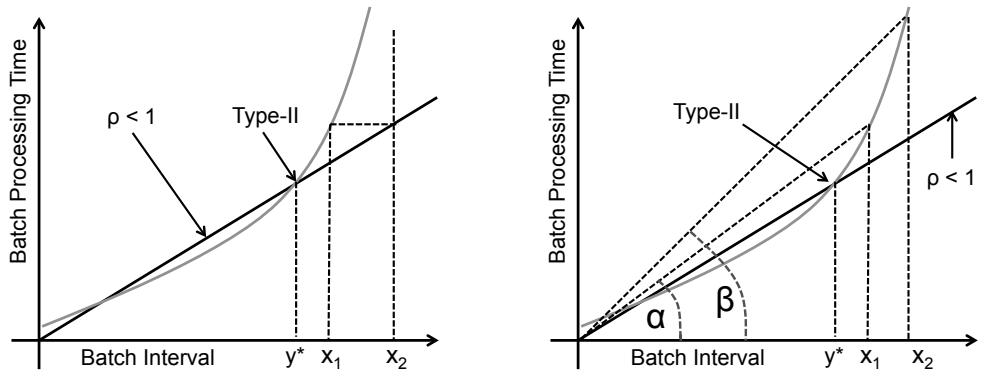
We can then iterate as follows:

1. Start with an initial guess x_1;
2. iterate using $x_{n+1} := f(x_n) = \omega(x^*)/\rho$ for $n = 1, 2, ...$

The iterations are illustrated in Figures 9a and 9b. Pictorially, they work as follows. At any batch interval x_n, the next batch interval will be determined by the intersection between the horizontal line with y-intercept at $\omega(x_n)$, and the threshold line $batch\ processing\ time = \rho \times batch\ interval$. As we can see from the figures, the fixed-point iterations converge quite fast to the desired Type-I intersection.

From the figures, it is also intuitively clear that if we start at a batch interval close to the Type-I intersection, then the system should converge to this intersection. This observation is in fact very general, and can be justified as follows. As it is mentioned above, the slope of ω at a Type-I intersection is less than ρ. This implies that at this intersection, the derivative (i.e., slope) of the function f, defined by $f(x) = \omega(x)/\rho$, is less than ρ. We can then invoke the Contraction Mapping Theorem to prove convergence.

Finally, we point out a very attractive property of the fixed-point iterations – no step size configuration is necessary. The algorithm automatically adjusts the step size based on how near or far it is away from the intersection, resulting in a fast convergence without any tuning. This makes the algorithm very robust with respect to changing operating conditions (i.e., changing $\omega(x)$) and is the primary advantage of this solution over our earlier attempts.



(a) Divergence above the Type-II intersection

(b) Reducing batch intervals near Type-II intersections

Figure 10: Handling Type-II intersections.

3.3.3 Handling Type-II Intersections

For linear workloads, Type-II intersections do not exist, hence the fixed-point iteration explained above will converge to the desired Type-I intersection. However, for a superlinear workload, we need a mechanism to detect whether the system is near the Type-II intersection. In this region, the Contraction Mapping Theorem breaks down and the fixed-point iterations will render the system unstable (see Figure 10a, where the batch intervals diverge to infinity).

Let y^* be the batch interval corresponding to a Type-II intersection. Notice that at a Type-II intersection, if our last two batch intervals are x_1 and x_2 with $y < x_1 < x_2$, then since the workload grows in a superlinear manner, we must have a) $\frac{\omega(x_1)}{x_1} < \frac{\omega(x_2)}{x_2}$, and b) $\omega(x_1) > \rho x_1$, and $\omega(x_2) > \rho x_2$. Condition a) is pictorially illustrated in Figure 10b, where for the two angles α and β, we have $\tan\alpha = \omega(x_1)/x_1$ and $\tan\beta = \omega(x_2)/x_2$. Since $\alpha < \beta$, $\tan\alpha < \tan\beta$ and $\frac{\omega(x_1)}{x_1} < \frac{\omega(x_2)}{x_2}$.

In this case, we want to decrease the batch interval, to move it closer to the Type-I intersection. More specifically, for a pre-configured parameter $r < 1$, we set the next batch interval $x_{n+1} := (1 - r) \times min(x_1, x_2)$.

3.3.4 Putting it All Together

In both cases above, we had made the assumption that we know $\omega(x)$. However, we actually do not. We only have data points based on the batch intervals and processing times of completed jobs. We use this data to approximate the underlying function $\omega(x)$. Specifically, we use the processing times of last two jobs to calculate the batch interval of the next batch to be received.

Algorithm 1 presents the function that calculates the next batch interval. This function is called after every batch has been received and new batch is about to start. Let batch intervals of the last two completed jobs be b_{last} and $b_{2nd-last}$. Also, let $x_{small} = \min\{x_{last}, x_{2nd-last}\}$, and $x_{large} = \max\{x_{last}, x_{2nd-last}\}$. Let the corresponding processing times of each batch interval be defined accordingly, that is, p_{last} for batch x_{last}, p_{small} for batch x_{small}, etc. The `CalculateNextBatchInterval`

Algorithm 1 Dynamic Batch Sizing Algorithm (Simplified)

Require: $x_{last},\ x_{2nd-last}$: *batch intervals of last 2 batches*
Require: $p_{last},\ p_{2nd-last}$: *proc. times of last 2 batches*
function CALCULATE NEXT BATCH INTERVAL
$x_{small} \leftarrow \min(x_{last},\ x_{2nd-last})$
$x_{large} \leftarrow \max(x_{last},\ x_{2nd-last})$
$p_{small} \leftarrow processing\ time\ of\ batch\ x_{small}$
$p_{large} \leftarrow processing\ time\ of\ batch\ x_{large}$
if $\frac{p_{large}}{x_{large}} > \frac{p_{small}}{x_{small}}$ **and** $p_{last} > \rho x_{last}$ **then**
$x_{next} \leftarrow (1 - r)x_{small}$
else
$x_{next} \leftarrow p_{last}/\rho$
end if
return x_{next}
end function

function essentially tests the two conditions for Type-II intersection, and accordingly reduces the batch interval by a factor of r or applies the fixed-point iteration.

We omitted two finer details from the above pseudo-code for brevity. They are as follows.

- In practice, a corner case often arises where the batch intervals of the last two completed jobs were exactly the same [2]. Checking for Type-II intersection is inconclusive in this case (the definition of p_{small} and p_{large} is ambiguous) and we simply choose to apply fixed-point iteration.

- At the start of the streaming computation, until the first job has completed, the algorithm has absolutely no knowledge of what the ideal batch interval would be. Manual configuration of the first batch interval is prone to errors – a large gap between this and the ideal batch interval can significantly delay convergence. We chose to start from a very small batch interval and exponentially increase it in every subsequent batch until the first job has completed. This is similar in principle to Slow Start in TCP [21] and ensures fast convergence.

3.4 Parameter Considerations

An advantage of the current control algorithm is that there are only two parameters to set - the slack factor ρ and the superlinear reduction factor r. The rationale for setting these parameters are as follows. A smaller value of ρ ensures greater robustness to fluctuating operating conditions at the cost of higher batch interval and lower throughput of the system (see Section 3.3.1). A larger r may bring the system closer to Type-I intersection more quickly, but it can also make the system

[2]This arises frequently in our implementation as all batch intervals are rounded to multiples of 100 ms.

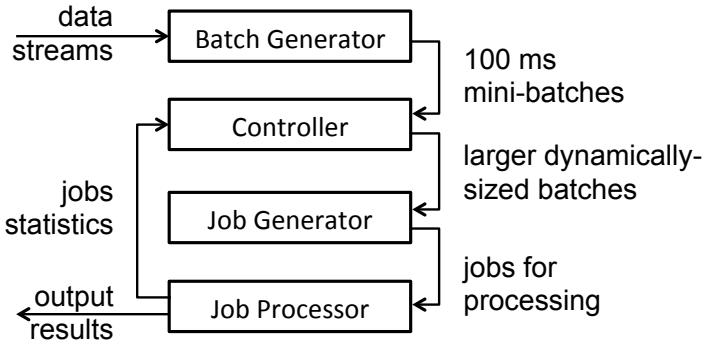


Figure 11: High-level architecture of our system.

vulnerable to the noise in the system (noise can make the algorithm identify the type of intersection incorrectly.). Through simulation and real workloads we found the values of $\rho = 0.7 - 0.8$ and $r = 0.25$ to provide good performance across many workloads. This is further discussed in Section 5.6.

3.5 Limitations

Till now, we have assumed the presence of the Type-I intersection which ensures existence of a batch interval at which the stability and low latency can be achieved. However, if for a particular set of operating conditions, the Type-I intersection does not exist (i.e., the processing rate cannot match the data rate under any batch interval), then the algorithm will not converge. It will continue to increase the batch interval in the hope of finding the Type-I intersection. Since we are operating under zero prior knowledge, identifying such a scenario is non-trivial. For the purpose of this work, we assume the presence of rate limiters or load shedder [22] that prevent the system from being overloaded.

4 Implementation

We choose to implement our solution in an existing open-source cluster computing framework called Spark [3]. While originally a batch processing framework, it also has extensions for batched stream processing. In this section, we describe the details of the implementation. Figure 11 presents the high-level architecture of the modified Spark system. The main modules of our system are as follows.

- *Batch Generator*: This module generates the batches based on a pre-configured batch interval. We kept this unmodified and configured it to generate batches of 100 ms. These *mini-batches* are handed off to the *Controller*.

- *Controller*: This is the new module we introduced that runs our control algorithm. It uses job statistics from the *Job Processor* to calculate the batch intervals in multiples of 100 ms

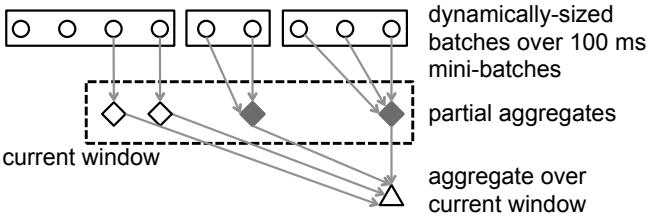


Figure 12: Adaptive window-based aggregation – Mini-batches (circles) are coalesced into dynamically-sized batches (rectangles). Partial aggregates from the dynamically-sized batches (filled diamonds) and the mini-batches (empty diamonds) are used to generate the final aggregate (triangle) over a window (dotted rectangle).

(i.e., the size of mini-batches). Based on that interval, the mini-batches are coalesced into larger batches and handed off to the *Job Generator*.

- *Job Generator*: This module generates the batch jobs on the batches of data it receives. Further details about the modifications to this module to support window-based operations are discussed later in this section.

- *Job Processor*: This unmodified module maintains a queue of jobs and runs them on a cluster one at a time. The generated job statistics are sent to the *Controller*.

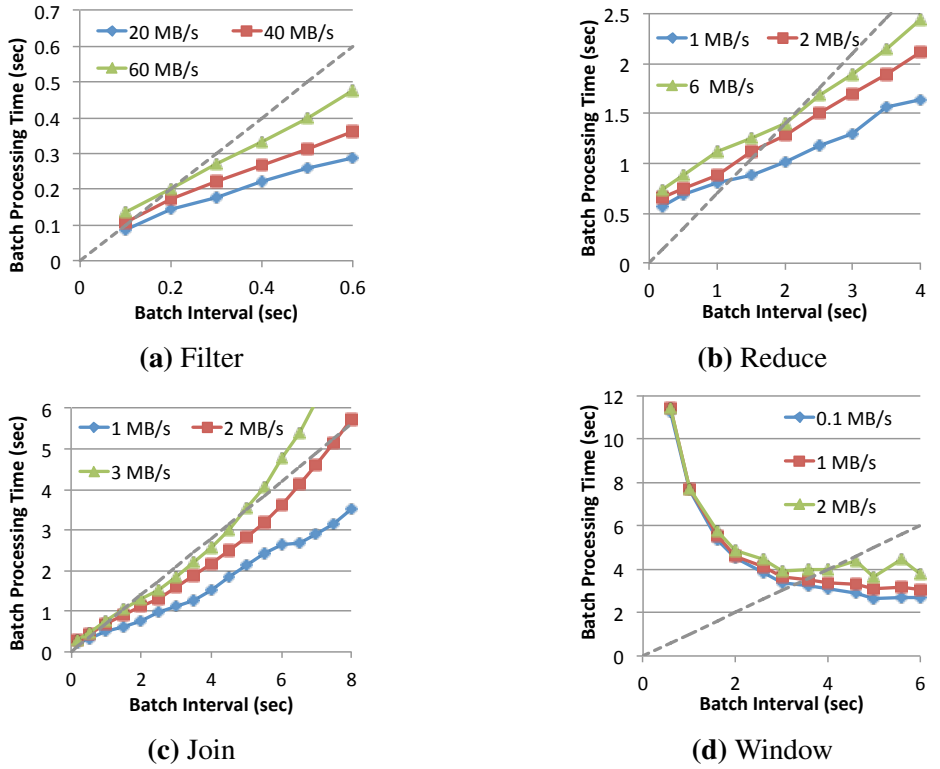
We chose 100 ms as the mini-batch size as anything lower significantly increased overheads in the system. Note that we slightly modified the control algorithm to round batch intervals to multiples of 100 ms.

A particularly challenging aspect of the implementation was supporting Spark Streaming's sliding window-based operations such as "continuously generate the count of records from the last 30 seconds". In Spark Streaming, the window operation has to be defined by a window length and a sliding interval, both being multiples of the constant-batch interval. After every sliding interval, the job for with window operation would be executed. We extend the model by allowing the system to automatically adapt how frequently window operation will be executed. In fact, for window-based aggregations, we allow partial aggregations over variable-sized batches to be reused for the window-based aggregations for greater performance. This is illustrated in Figure 12.

To summarize, our modification have been fairly simple and did not require any changes to programming interface. Hence, we believe that our dynamic batch sizing technique can be implemented on any batched stream processing system.

5 Evaluation

In this section, we present the results of our evaluation of our algorithm by subjecting it to various streaming workloads under different combinations of data rates and operating conditions.



(a) Filter **(b)** Reduce

(c) Join **(d)** Window

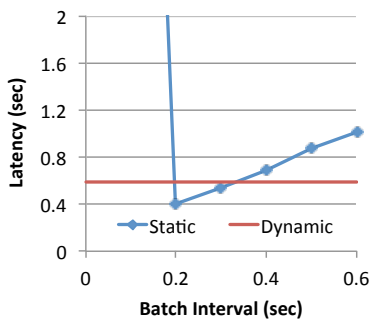
Figure 13: Relationship of the processing time of the workloads with batch interval. The dotted line is the stability line.

5.1 Setup and Workloads

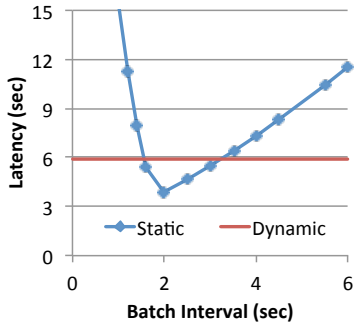
For evaluation, we used a cluster of 20 m1.xlarge EC2 instances. We ran four different workloads, each having unique workload characteristics as shown in Figure 13. Note that they have been illustrated in the context of the stability threshold (i.e., batch processing time = batch interval). The detailed descriptions of the workloads are as follows.

- **Filter**: This is a simple data filtering workload in which streaming text data is split into words, matched against a list of filters and counted. Compared to other workloads, the desired batch interval of this workload is small as shown in Figure 13a. This workload evaluates our algorithm's ability to converge to the small batch intervals.

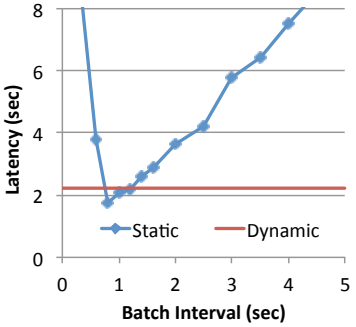
- **Reduce**: This is an aggregation workload in which data is reduced based on a key and the reduced data is committed to a database. Committing the data for each key takes about 1 ms. This sets the lower bound on the batch processing time of about about 500 ms, as shown in Figure 13b. Note that this minimum processing time is a function of the number of unique keys – higher number of unique keys would require higher batch processing time. Figure 13b shows this relationship for 20000 unique keys. Furthermore, as the data rate changes, the desired batch interval changes significantly. So this workload evaluates the



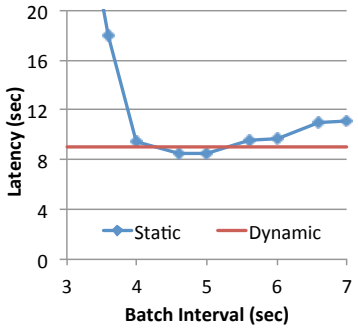
(a) Filter with 40 MB/s of data



(b) Reduce with 5 MB/s of data



(c) Join with 4 MB/s of data



(d) Window with 1 MB/s of data

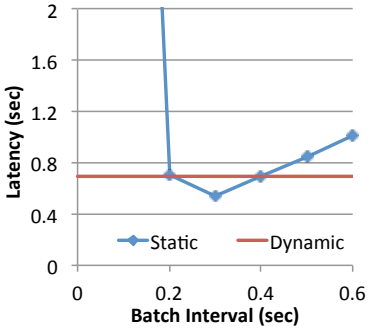
Figure 14: Comparison of the end-to-end latency observed with constant data rates and static / dynamic batch intervals. Our algorithm is able to ensure latency that is comparable to the minimum latency achieved with any statically defined batch interval.

control algorithm's ability to adapt to large variations in the desired batch interval as the data input rate changes.

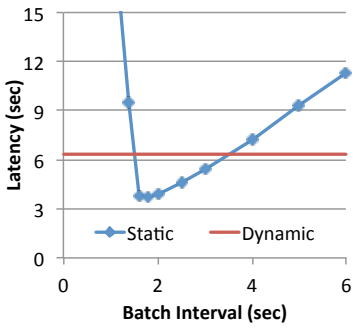
- **Join**: This workload joins batches of data from two different data streams. As it has been discussed earlier, this workload has superlinear characteristics. Note that as the data rate increases, the range of batch intervals in the stable zone reduces. Needless to say, this workload tests our algorithm's ability to adapt to superlinear workloads.

- **Window**: This extends the *Reduce* workload by applying the reduction over a moving 20 second window. The movement of the window is decided adaptively by our algorithm. As discussed in Section 4, this is a multi-stage workload where partial reductions over the batch intervals are re-used to compute reductions over the window. It has a significantly different behavior from other workloads as shown in Figure 13d – the processing time increases at low batch intervals. This is because partial reductions over smaller intervals increases the computational window-based reduce compared to larger intervals [3]. This workload tests the

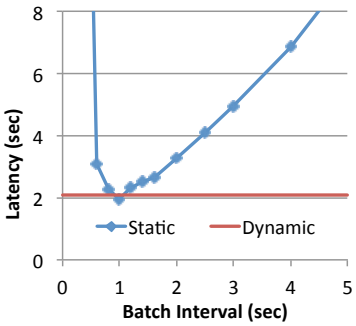
[3]This is an artifact of underlying system where smaller intervals lead to larger number of tasks needed to compute the window-based reduce, thus increasing the system overheads.



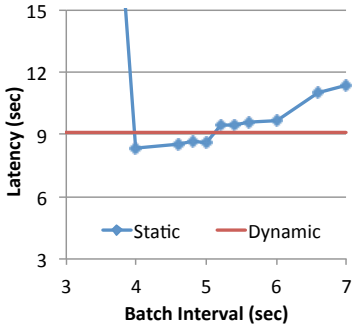
(a) Filter with sinusoidal rates, cycling every minute between 20 and 60 MB/s



(b) Reduce with sinusoidal rates, cycling every minute between 1 & 6 MB/s



(c) Join with sinusoidal rates, cycling every minute between 1 and 4 MB/s



(d) Window with sinusoidal rates, cycling every minute between 0.1 and 0.4 MB/s

Figure 15: Comparison of the average end-to-end latency observed with time-varying data rates and static / dynamic batch intervals. Our algorithm is able to ensure latency that is comparable to the minimum latency achieved with any statically defined batch interval.

robustness of our algorithm against more complex workload characteristics that the algorithm was not explicitly designed for.

These characteristics may be specific to the cluster and processing framework used to run these workloads. It is important to note that we are not evaluating the performance of these workloads under the framework. Our goal is to evaluate our algorithm's ability to optimize and adapt under a wide range of workload characteristics. We simply assume out-of-the-box performance characteristics of these workloads and try to adapt based on them. Hence, absolute performance numbers are best ignored.

We configured our algorithm to use $\rho = 0.7$ and $r = 0.25$ in all cases. Our goal is to minimize end-to-end latency which is defined as the sum of batch interval, queuing delay and processing time for each batch.

5.2 Comparison with Static Batch Intervals

Recall that a desirable property of the control algorithm is that it should be converge to a low batch interval that can sustain the data rate as well as provide a low end-to-end latency. To evaluate this, for each workload, we first measured the minimum latency that can be achieved with statically defined batch intervals. Then, under the same operating conditions, we measured the latency achieved by applying our control algorithm and compared it to the minimum. We did this for constant data rate as well as time-varying data rates. The results are presented next.

5.2.1 Constant Input Data Rate

First, we compare the performance of our algorithm under constant data rate. Figure 14 presents the average end-to-end latencies observed for each workload with statically defined batch intervals as well as with dynamically adapted batch intervals. In all cases, the average latency achieved with our algorithm is comparable to the minimum latency achieved with statically defined batch intervals. Specifically, it was able to achieve a low batch interval for the *Filter* workload and it correctly identified the higher desired batch interval for the *Window* workload. In case of the *Reduce* workload, the difference between the minimum latency and the one achieve by our algorithm is relatively larger than the other workloads. This is largely due to the approximation introduced by the slack parameter ρ, as discussed in Section 3.3.1. This approximation was larger in case of the *Reduce* workload than the others.

Note that this was achieved without any workload-specific configuration of the control algorithm. This highlights the power of the algorithm – as long as the data rate is sustainable by the cluster resources, the algorithm can achieve low latency without any prior knowledge about the workload characteristics.

5.2.2 Time-varying Input Data Rate

We subjected each workload to input data whose rate varies in a sinusoidal fashion with period of 1 minute. As shown in Figure 15, even with time-varying input data rates, our control algorithm is able to achieve latency comparable to the minimum that can be achieved with any fixed batch interval. We explored further and analyzed the detailed behavior of processing times and queuing delays that a workload experiences under static batch intervals. This is illustrated in Figure 16. This is the time line observed with the *Reduce* workload under sinusoidal data rates and a static batch size of 1.3 seconds (chosen to specifically to highlight the phenomenon sketched out in Figure 4). While the data rate cycles between 1 and 6.4 MB/s, the processing times repeatedly becomes more than the static batch interval of 1.3 seconds. The queuing delay starts building up (around 40 seconds mark), thus increasing the end-to-end latency. In contrast, our algorithm reacts to the increasing processing times and increases the batch interval to keep up, as illustrated in Figure 17. This lowered the average end-to-end latency in this case from 4.7 seconds (static batch interval of 1.3 seconds) to 3.9 seconds.

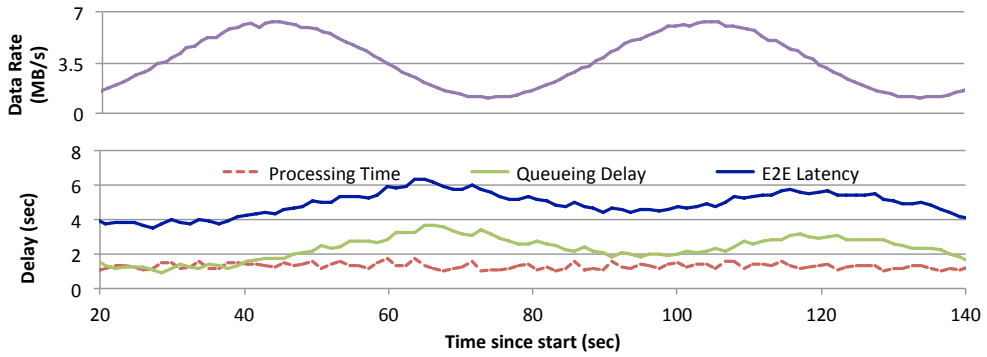


Figure 16: Timeline of the batch interval and other times for the *Reduce* workload with sinusoidal input data rate and static batch interval. The queuing delay builds up when the processing time exceeds above the batch interval, thus increasing the latency.

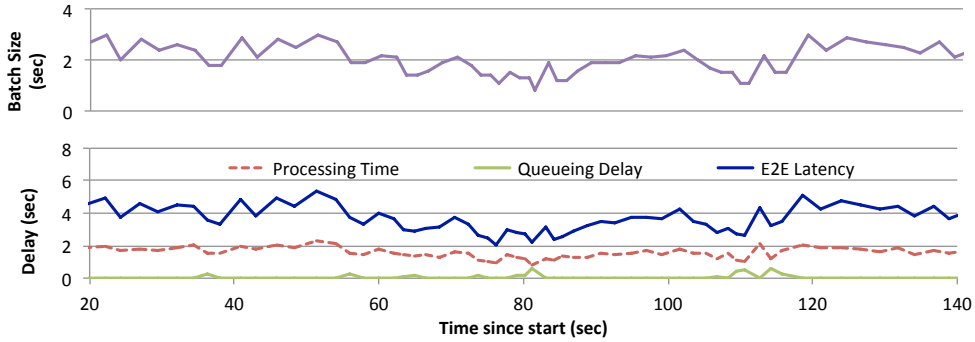


Figure 17: Timeline of the batch interval and other times for the *Reduce* workload with sinusoidal data rate as shown in Figure 16. Unlike static batch intervals, our algorithm automatically adapts to increasing processing times, thus preventing any queue buildup and ensuring a lower latency.

5.3 Comparison with Other Techniques

In Section 5.2.2, we had argued that our initial approaches based on gradient information were insufficient due to errors in the estimation of the gradient. To understand this, we also ran the algorithm discussed in Section 3.2.2 with the *Reduce* workload. Figure 18 shows that under a sinusoidal data rate the system destabilizes very soon. The sudden spike at 45 second mark caused a large error in the estimated gradient forcing the system to reduce the batch interval to almost zero. The resulting buildup of queuing delay completely destabilized the system.

5.4 Robustness under Workload Variations

The characteristics of a workload may change even if the data rate stays the same. For example, in *Reduce*, the number of keys determine the number of database commits that one has to make while

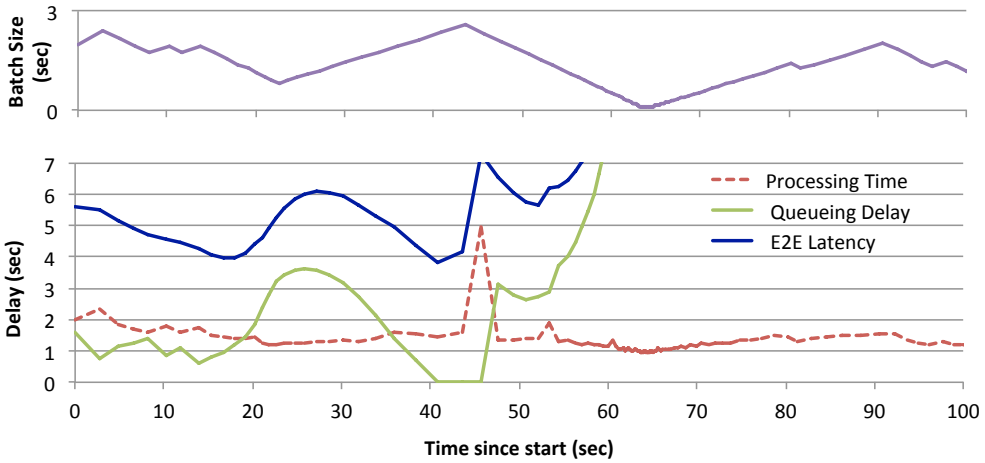


Figure 18: Timeline of batch interval and other times for the *Reduce* workload with a gradient based control algorithm. This highlights the vulnerability of this algorithm to noise.

processing each batch changes. Therefore, any changes in the number of keys significantly affects the workload characteristics and a control algorithm should be able to adapt to such variations.

To test this, we ran *Reduce* with constant data rate of 6 MB/s, but the number of keys was varied between 500 and 40000. Figure 19 illustrates that our algorithm is quickly able to adapt to such changes. It can provide latencies as low as 600 ms in the presence of 500 keys as well as adapt to latencies as high as 4.3 seconds when required. Also note that it adapts quite swiftly preventing the system from significant queue buildup. This further illustrates the power of our algorithm to adapt to arbitrary changes in workload characteristics (assuming that it is possible for the system to sustain the load at some batch interval).

5.5 Robustness under Resource Variations

In Section 2.3, we had argued that setting a static batch interval based on offline learning of a workload's characteristics is vulnerable to cluster resource changes. To further emphasize this argument, we illustrate the our algorithm's ability to adapt to changes in cluster resources[4]. We consider a common scenario where multiple processing frameworks are sharing the same cluster resources. It is possible that other batch jobs submitted to a shared cluster reduce the amount of resources available for the streaming workload running on the same cluster.

We emulate this scenario by running other Spark batch jobs that consume 25% of the resource of the same cluster that is running the *Reduce* workload. Figure 20 illustrates that our control algorithm is able to automatically adapt the batch interval when the jobs are running, increasing the latency from 4.1 seconds to 5.1 seconds.

[4]We assume that enough resources are available for the streaming workload to continue running.

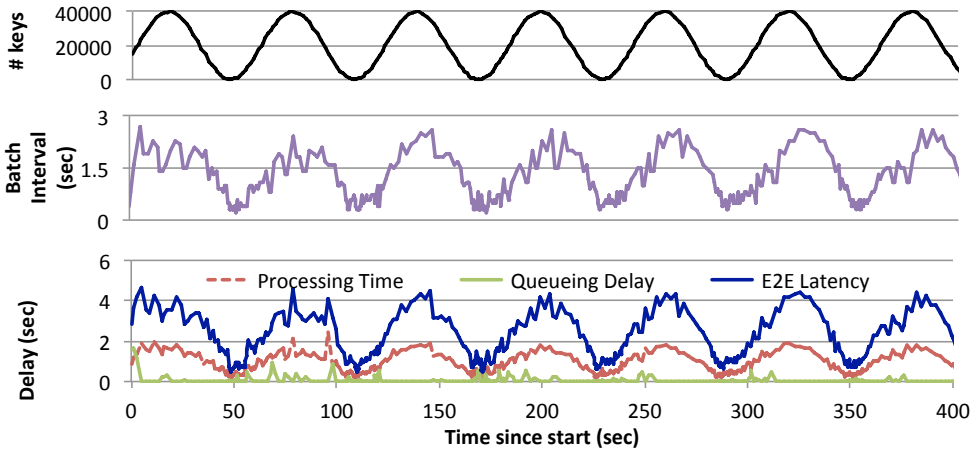


Figure 19: Timeline of number of keys, batch interval and other times for the *Reduce* workload under variations in the number of keys (data rate is constant at 6 MB/s).

5.6 Parameter Study

Finally, we present justification for our choice of values for ρ and r. We ran our algorithm on *Reduce* and *Join* workloads for various combinations of values of the two parameters. Data rate was constant. Figure 21 illustrates the average latency achieved in each case. $\rho > 0.8$ tends to destabilize the system as the slack becomes too low to accommodate the noisy processing times and $\rho < 0.7$ tends to increase the approximate in the batch duration (see section Section 3.3.1). On the other hand, the reduce in batch interval in the superlinear case (Figure 21b) is either too aggressive or not aggressive enough with $r = 0.4$ and $r = 0.1$, respectively. Hence, we chose $\rho = 0.7$ and $r = 0.25$. This seems to work well across the workloads obviating the need for tuning.

6 Discussion

Workload Semantics: For map-only streaming workloads like *Filter*, the impact of variable batch interval on the quality of the result is expected to be minimal. However, for aggregation-based workloads like *Reduce* and *Join*, the quality of the result may depend on the batch interval. For example, the count of unique keys in 100 ms batches may be different from that in 10 second batches on the same data stream. Developers must be aware of this aspect. However, it is important to note that this change in quality is different from lossy techniques like load shedding [8, 22] where data is dropped to effectively reduce the data rate.

Control Loop Delay: This is the delay between the time when a control decision is taken (e.g., batch interval is increased) and the time when the effect of that decision is observed by controller (the job processing the corresponding larger batch completes). Like many other control systems [20], our algorithm is vulnerable to large control loop delays. Sudden large changes in the

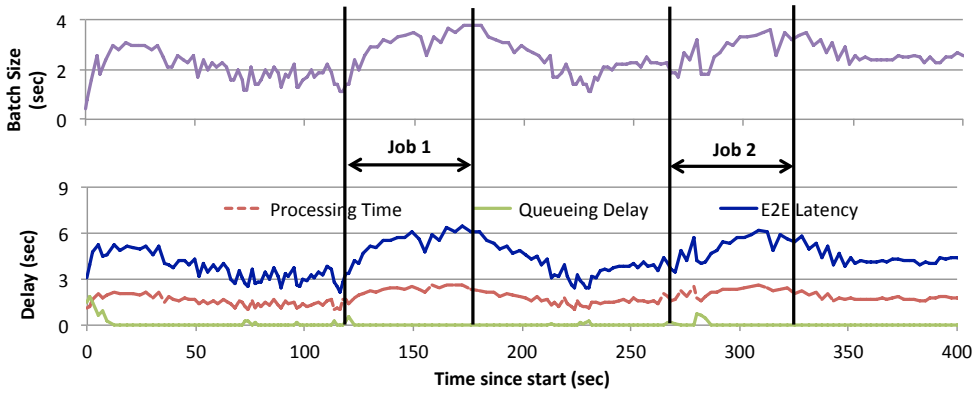
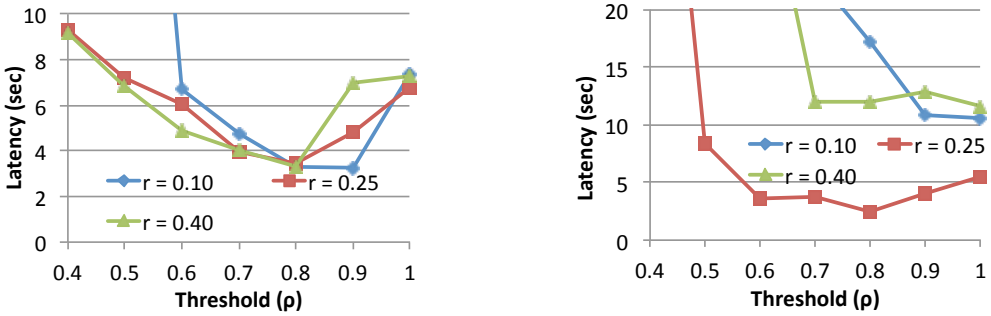


Figure 20: Timeline of batch interval and other times for the *Reduce* workload under variations in the available resources. Our algorithm is able to adaptively increase the batch interval when external batch jobs 1 and 2 take away 25% of the cluster resource.



(a) Reduce **(b)** Join

Figure 21: Average latency under various combinations of values of ρ & r. $\rho =$ 0.7 - 0.8 and $r = 0.25$ works well for both workloads.

workload characteristics (e.g. 10x increase in processing time between consecutive batches) can introduce large delays. Dealing with them is non-trivial and is left for future work.

More Complex Workloads: As evidenced by the *Window* workload (see Section 5.1), there may be workloads with more complex relationships than the two (i.e., linear and superlinear) we assumed in this work. Things become particularly complex if there are more than two intersection points. Such cases are non-trivial to solve and is left for future work.

7 Related Work

Stream Processing and Incremental Data Processing Systems: Our proposed algorithm is primarily focused on batched stream processing systems such as Comet [13] and Spark Streaming [24]. They collect received streaming data into batches and periodically process them using MapReduce-style batch computations. In both systems, the periodicity of the batch computation is

left to the developer to figure out, which is non-trivial as discussed in Section 2.3. Our technique to dynamically adapt the size based on system progress alleviates this issue.

Other stream processing systems such as Borealis [6], TeleGraphCQ [10], TimeStream [17], Naiad [15], and Storm [4] are based on the *continuous operator model*. In this model, the streaming computation is expressed as a graph of long-lived operators that exchange messages with each other to process the streaming data. For efficiency, many of these systems employ batching of messages between pairs of operators. While such systems are not the direct focus of this paper, our adaptation technique may be applicable to message streams within each pair of operators. Incremental bulk data processing systems such as CBP [14], Percolator [16] and Incoop [9] allows updated view of processed data set to be maintained by incrementally and efficiently recomputing the updates to the input data. In such systems, the recomputation is triggered whenever an update to the input dataset is detected. Percolator briefly discusses the adjustment of batch size of data updates based on server load. In general, this has not been well-explored in such systems, and our algorithm may be prove useful in them.

Adaptive Stream Processing Systems: There has been much work in area of load-based adaptation in stream processing systems based on the continuous operator model. Load shedding techniques adaptively discards a fraction of the received data when the processing load exceeds the capacity of the system [8, 22]. This is of course a lossy technique, which introduces errors in the result. By comparison, our dynamic batch sizing is a lossless technique because as long as there is a batch interval at which the system can be stable, no load will be shed. However, we do assume the presence of load shedder when the data rate too high for the system to handle at any batch interval. Some have proposed elastic adjustment of available resources based on the progress of operators [7, 18, 23]. Others have proposed dynamic reconfiguration of the operator graph [17, 19]. Since these techniques have the designed with continuous operator model, it remains to be seen whether they are directly applicable to the batched streaming model. However, they are orthogonal to our technique, and can be used in conjunction.

Control and Queueing Theory The dynamic batch sizing problem falls under the broad category of feedback control [20], since our batching decisions are only based on the feedback and statistics that we receive from earlier batching decisions. We can also view the problem as a queueing theory problem, where customer inter-arrival times correspond to batch intervals, and customer service requirements correspond to processing times of the batches. However, our problem is distinct from traditional queuing problems in that the batch intervals (or inter-arrival times) are control decisions, and processing times depend on batch intervals in a complicated manner. To the best of our knowledge, our problem formulation is new, hence it is difficult to compare the performance of our algorithm with those in other contexts.

8 Conclusion

In this paper, we have present a novel control algorithm for dynamically adapting the batch size in batched stream processing systems. We have shown that, without any workload-specific configuration, it is able to adapt to changing data rates and cluster resources and also provide a low

end-to-end latency.

References

[1] Banach fixed-point theorem. http://en.wikipedia.org/wiki/Banach_fixed_point_theorem.

[2] Fixed-point iteration. http://en.wikipedia.org/wiki/Fixed-point_iteration.

[3] Spark. http://spark-project.org.

[4] Storm. https://github.com/nathanmarz/storm/wiki.

[5] Spotty twitter service. http://goo.gl/XUBqd9.

[6] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[7] 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*, pages 71–71. IEEE, 2006.

[8] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[9] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 7. ACM, 2011.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[11] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] 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*, pages 63–74. ACM, 2010.

[14] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. SoCC, 2010. ISBN 978-1-4503-0036-0. doi: http://doi.acm.org/10.1145/1807128.1807138. URL `http://doi.acm.org/10.1145/1807128.1807138`.

[15] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP '13*, 2013.

[16] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010*.

[17] 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 *EuroSys '13*, 2013.

[18] 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*, pages 1–12. IEEE, 2009.

[19] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. *SIGMOD*, 2004.

[20] E. D. Sontag. *Mathematical control theory: deterministic finite dimensional systems*, volume 6. Springer, 1998.

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

[22] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320. VLDB Endowment, 2003.

[23] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 791–802. IEEE, 2005.

[24] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013.