Adaptive Stream Processing using Dynamic Batch Sizing

Tathagata Das
University of California Berkeley

Yuan Zhong
Columbia University

Ion Stoica Scott Shenker
University of California Berkeley

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 effects of the batch size 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 the batch size as the situation necessitates. We show through extensive experiments that it can ensure system stability and low latency for a wide range 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 content distribution network may wish to monitor its distribution system in real-time for anomalies. A social network may wish to quickly find trending conversation topics. The processing demands of such workloads has led to the development of distributed stream processing frameworks [5, 16–18, 25], that are designed to scale to large clusters, and tolerate system failures.

Recently proposed frameworks have chosen to treat stream processing as a continuous series of MapReduce-style batch processing jobs on batches of received data [14, 25]. This model leverages the fault-tolerance properties of the MapReduce [13] processing model to allow faster fault-recovery (parallel recovery in [25]) and straggler mitigation (speculative execution in [13]). This enables efficient and fault-tolerant stream processing at scale.

A practical requirement of such systems is robustness against variations in streaming workloads. For example, a social network wishing to find trending conversation topics 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.

Traditionally, stream processing systems have managed such scenarios either by elastically increasing available resources [8, 19, 24], or by discarding part of the data streams (i.e., load shedding) [9, 23]. However, losing data is often not an option, and apriori provisioning of resources for handling unpredicatably high loads can be expensive. Hence, its important to explore other approaches that allow the processing system to adapt to changes in operating conditions. In this work, we focus on the batch size of batched stream processing systems as it can significantly affect the performance of the system even with the same amount of available resources.

Depending on the workload, larger batches of data may allow the system to process data at higher rates. However, larger batch size also increases the end-to-end latency between receiving a data record and getting the corresponding results. Ideally, the system should 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 dynamically varying 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 the 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 latencies. Furthermore, the algorithm must be robust under noisy operating conditions that arise from continuous variations in the data rates, available resources, etc.

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2670995

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 latencies that are 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 behaviors 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 the related work and Section 8 concludes by summarizing our contributions.

2. The Case for Dynamic Batch Sizing

There are many factors that affect the performance of a stream processing system - cluster size, parallelism of operators, batch sizes, etc. Previous literature have studied various techniques to adapt to changes in operating conditions, either by elastically scaling the workload [8, 19, 24] or by discarding data to shed load [9, 23]. However, in many practical use cases (stock ticks, bank transactions, etc.), data loss is unacceptable. And dynamic scaling may require apriori provisioning of resources (especially in non-cloud environments) which can be expensive if peak surges need to be accounted for. This motivated us to explore adaptive methods that neither require extra resources, nor require data loss.

The batch size of a batched streaming systems significantly affects the performance. Hence, dynamically adapting the batch size may allow the system to adapt in our desired manner. This is what we argue for in this section. First, we describe our simple system model of batched stream processing systems. Then we discuss the impact of batch size on the performance, highlighting the drawbacks of statically set batch sizes. Finally, we present our goal of dynamically adapting the batch size and the challenges associated with it.

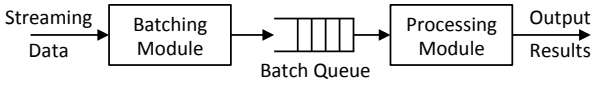


Figure 1: Model of a batched stream processing system

2.1 System Model

Figure 1 illustrates a simple model that we use to analyze 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 batches of 1-second, then the data received in each 1-second interval will form new batches which are put into the *batch queue*. The *processing module* dequeues and processes them one by one.

Note that we chose to define the 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. This is hard to model as data streams with different rates would generate batches at different intervals. This is incompatible with the model of batched streaming systems. Also, as discussed next, important metrics like latency can be naturally characterized in terms of time intervals. We shall henceforth use the term *batch interval* instead of batch size to avoid confusion.

Also note that we do not model the capacity (i.e., available resources) of the processing module. It is considered as a black box and any change to it is treated no differently from any other changes in the conditions (e.g., data rates).

2.2 Effect of Batch Interval on Performance

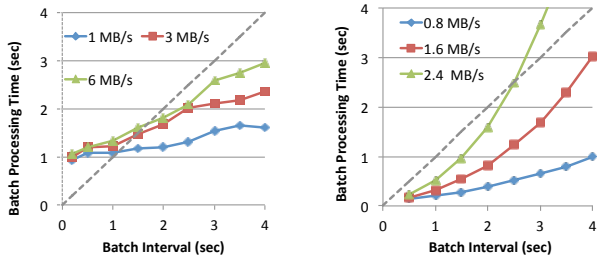
Here we discuss the effect of the batch interval on the latency and the throughput of a streaming workload.

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.

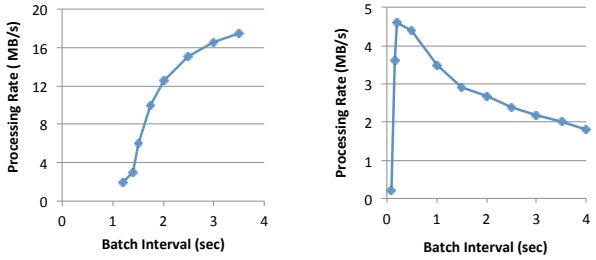
- *Batching delay*: The duration between the time a data record is received by the system and the time that record is sent to the batch queue – this quantity is upper bounded by the corresponding batch's interval;
- *Queuing delay*: The time that a batch spends waiting in the batch queue;
- *Processing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

It is clear that the latency directly depends on a batch interval – lower batch interval leads to lower latency. Hence, it is desirable to keep the batch interval as low as possible. How-



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

Figure 2: Behavior of the processing times of two streaming workloads with respect to batch intervals and data rates.



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

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

ever, 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 received. In case of batched stream processing systems, the sufficient condition for stability is that the batch processing time 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 the build up of the batch queue.

The processing time monotonically increases with the batch interval – longer intervals imply more data to process, hence higher processing times. However, the exact relationship between the intervals and the processing times can be non-linear, as it depends on the characteristics of the processing system, the available resources, the nature of the workload, and the data ingestion rates. Variations in these can change the behavior of the processing times with respect to the intervals.

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 M and N are the numbers of records in the two datasets. As the batch interval of both data streams are varied simultaneously, the resultant number of records and processing time can increase superlinearly[1]. 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.

2.3 The Cost of a Static Batch Interval

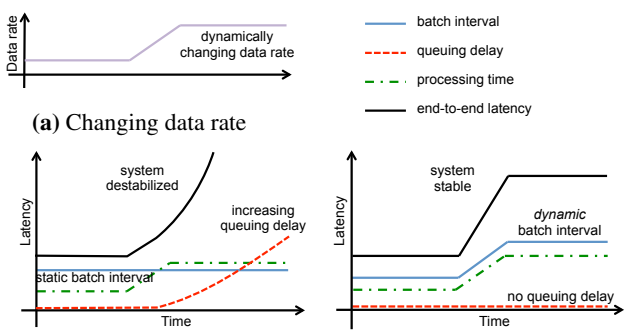
Having understood the effects of the batch intervals on the performance, it is clear that batch intervals need to be carefully chosen to achieve both goals – stability and low latency. A plausible strategy to determine such a batch interval is to apply offline learning. Given a streaming workload, a 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 required to be processed.

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 well-equipped services like Twitter experience outages due to surges in data rates during various global events [6]. If set statically, large batch intervals may be needed to account for such surges. Correspondingly, the latency will be needlessly high for normal conditions.

Figure 4 schematically illustrates a single scenario of how the system with a statically set batch interval destabilizes 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

[1] The implementation details of Join operation and the operating conditions (selectivity, etc.) are less important. As long as there exists a workload that exhibits superlinear behavior on a streaming system, our cause is justified.



(a) Changing data rate

(b) Instability with a static batch interval

(c) Stability with dynamic batch sizing

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

black line) to build up indefinitely. As a result, the system destabilizes.

In summary, a statically defined batch interval may neither minimize the latency in the current operating conditions, nor ensure system stability when the 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. We want to detect changes in the operating conditions and accordingly increase the batch interval such that the stability condition is maintained. Queuing delay will stay low, and the system will 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, as shown in Figure 5. The *control module* receives statistics (e.g., processing time, queuing delay, etc.) of completed jobs from the processing module and runs an online control algorithm to continuously learn the system behavior and 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 this control algorithm. Next, we discuss the desirable properties of the algorithm and discuss why it is hard to achieve them.

Desirable Properties of the Control Algorithm

- *Low Latency*: The control algorithm should be able to achieve small batch intervals and hence low end-to-end latencies, while ensuring the stability of the system.
- *Agility*: The control algorithm should be able to quickly detect any changes in the operating conditions, and quickly converge to the desired batch interval.
- *Generality*: The algorithm should ideally be able to deal with any workload having any kind of relationship between processing times and batch intervals. We do not want the developers to worry about the applicability of the algorithm on their workloads.
- *Easy to configure*: A control algorithm can 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.

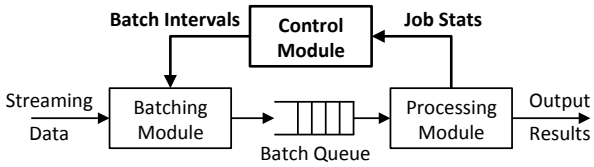


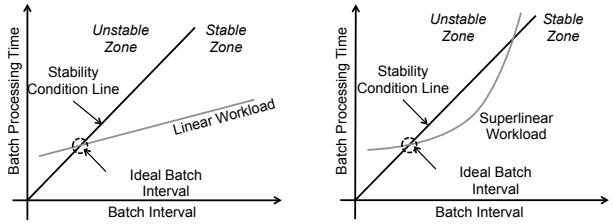
Figure 5: Model of our proposed system with a control loop that dynamically adapts the batch interval based on job statistics.

Why is it Hard to Achieve these Properties?

- *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 vice versa. 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.
- *Noise*: A practical processing system may have noisy behavior due to continuous variations in the data ingestion rates and the cluster conditions. Converging to a stable batch interval under such conditions is challenging for a control algorithm.
- *Trade-off between Accuracy and Agility*: Online control algorithms that simultaneously model a system and change it suffer from fundamental trade-off between the rate of learning and the accuracy of the model learned – more time and information helps learning more accurate but stale models, making the system slow to adapt.

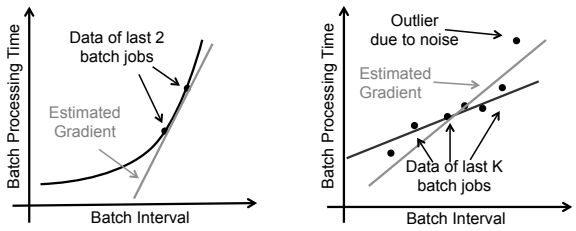
2.5 Assumptions on Workload Semantics

For simple workloads that does not involve aggregation (e.g., filtering events), the size of the batch does not make a difference to the results. However, for aggregation-based workloads, changing the interval may make a difference to the final results (e.g., counting unique records with 1-second batches will be different from that with batches of 10 seconds). However, for many workloads (e.g., maintaining an approximate count of active users in a social network), this variation in the semantics is an acceptable tradeoff for achieving greater stability without load shedding. Detailed analysis of the implications of this is beyond the scope of this paper and is left as future work.



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

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



(a) Gradient estimation using two points **(b)** Inaccurate gradient estimation due to noisy statistics

Figure 7: Gradient estimation and its sensitivity to noise.

3. Dynamic Batch Sizing

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

3.1 Problem Formulation

The goal of our control algorithm 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[2]. 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 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 properties. 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 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 [12], 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.

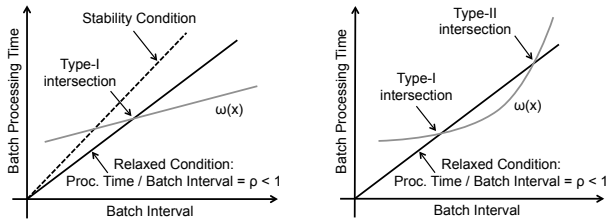
Our goal is very similar – we wish to adapt the batch interval based on the stability of the streaming workload. Hence, at the first glance, one can devise a simple control algorithm that increases the batch interval if the operating point is in the unstable zone and vice versa. Even though this approach may work for linear workloads, it fails for superlinear workloads. 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 signals unsuitable for our purpose – without any more information, it is impossible to know whether to increase or decrease the batch interval for increasing the 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 most likely operating near the higher intersection of the superlinear workload. Thus, a plausible approach to building a dynamic control algorithm is to use the statistics of complete jobs to estimate the gradient, and change the batch interval accordingly. For example, we can do the following.

- Estimate the 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.

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



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

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

- 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.

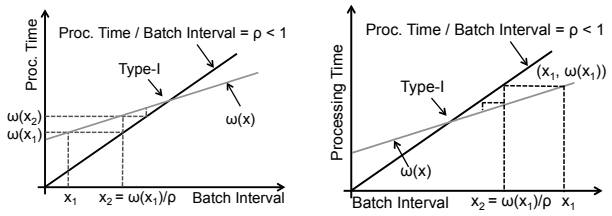
The simplest way to estimate the gradient is to calculate the slope based on the batch intervals and the processing times of the last two completed jobs (Figure 7a). This could be a sufficiently good estimate for a smooth workload orofile in a static environment. However, in practice, varying data rates and noise in processing times can increase the estimation error. This can make the system adapt the batch intervals incorrectly, 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 - outliers can drastically change the results of the linear regression.

3.2.3 General Control Algorithms

We also attempted to apply general control theory principles to model this problem to apply known control theory techniques. However, we found it difficult to build an intuitive model of our system where the processing times and the resultant queuing delays depend on the batch intervals in a complicated manner. Furthermore, another practical challenge was to tune the step size of such control algorithms. It is often not clear what step sizes should be used in these algorithms. Too small steps sizes can increase convergence times making the system unable to keep up with changes in conditions. Too large step sizes can make the system vulnerable to noise preventing it from finding the desired batch interval. This issue makes it harder for the application developers to tune the control algorithms for their own applications. Instead we found our chosen approach more stable and intuitive.

3.3 Our Solution - Fixed-point Iteration

We chose our control algorithm to be based on a well-known optimization technique, Fixed-point Iteration [2], which is designed to find the fixed point of a static, well-defined



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

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

function. In this section, we present our novel approach of applying this technique to estimate the dynamically-varying function of a workload.

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 the Type-I intersection in both cases.

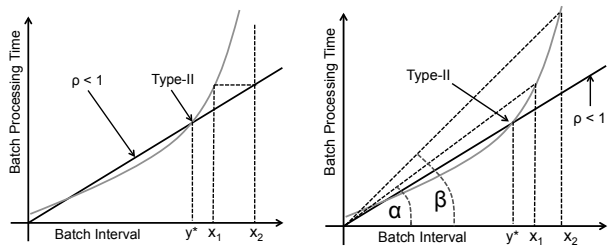
3.3.2 Fixed-point Iterations for Type-I intersections

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.

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



(a) Divergence above the Type-II intersection

(b) Reducing batch intervals near Type-II intersections

Figure 10: Handling Type-II intersections.

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.

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).

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

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 the two cases discussed, we assumed that the $\omega(x)$ is known. However, it reality, it is an unknown function. Consequently, the data points based on the past job statistics are used 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 every time a batch has been received and the interval of the next batch is to be decided. Let batch intervals of the last two completed jobs be x_{last} and $x_{2nd-last}$. 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` 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

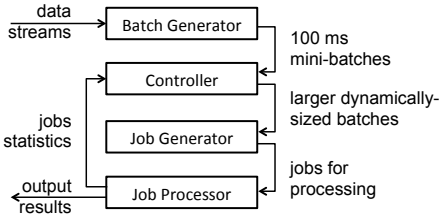


Figure 11: High-level architecture of our system.

same [3]. 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 [22] 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 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 [23] that prevent the system from being overloaded.

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

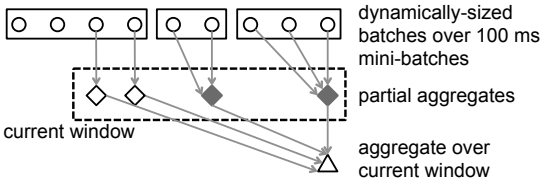


Figure 12: Adaptive window-based aggregation – Mini-batches (circles) are coalesced into dynamically-sized batches (rectangles). Partial aggregates from the dynamically-sized batches (filled diamonds) and the mini-batches (empty diamonds) are used to generate the final aggregate (triangle) over a window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

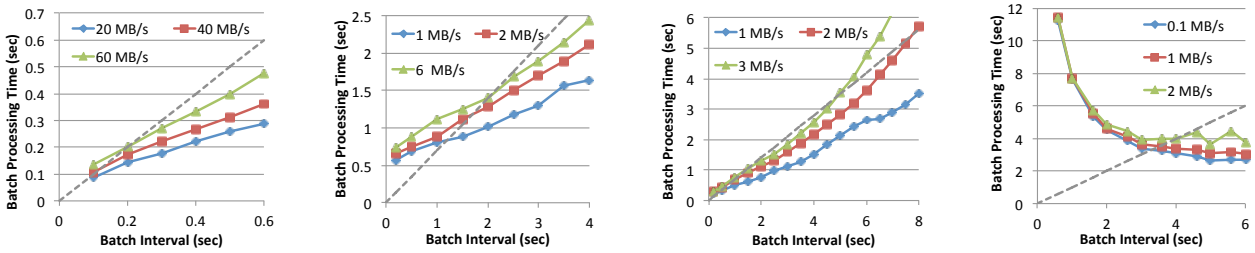
4. Implementation

We choose to implement our solution in an existing open-source distributed, batched stream processing framework called Spark Streaming [4], which is an extension of the Apache Spark framework [3]. 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 batches based on a pre-configured 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 (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 interval 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 of the window operation would be executed. We extend the model by allowing the



(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.

system to automatically adapt how frequently the 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 the programming interface. Hence, we believe that our dynamic batch sizing technique can be implemented on any batched stream processing system.

5. Evaluation

We evaluated our algorithm by subjecting it to various streaming workloads under different combinations of data rates and operating conditions. The key takeaways are:

- For a diverse set of workloads (single-stage as well as complex multi-stage) with very different behaviors, both at constant and at time-varying input data rates, our algorithm was able to achieve latencies comparable to the minimum latencies achievable by hand-tuning. This demonstrates the *generality* of our algorithm, and the ability to achieve *low latency* (note the desirable properties in Section 2.4).
- Our algorithm achieved these results without any prior knowledge of the workload characteristics and without workload-specific tuning (*easy to configure* property).
- Even with variations in the workloads' characteristics, and variations in the available cluster resources, our algorithm was able to adapt, thus demonstrating its *agility*.

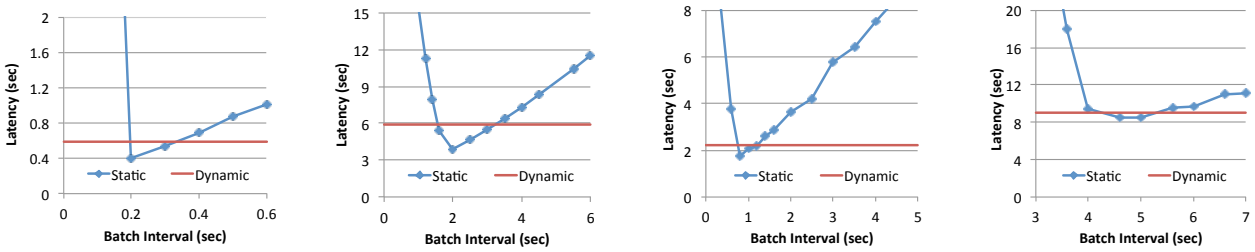
5.1 Setup and Workloads

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**: A simple data filtering workload in which streaming text is split into words, matched against a list of filters and counted. The desired batch interval of this workload is small as shown in Figure 13a. This will evaluate our algorithm's ability to converge to the small batch intervals.
- **Reduce**: An aggregation workload in which data is reduced based on a key and the results are stored into a database. Committing the record corresponding to each key takes 1 ms. This sets the lower bound on the batch processing time of 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 longer batch processing time. Figure 13b shows this relationship for 20000 unique keys. Note that the desired batch interval changes significantly with data rate. So this will evaluate our algorithm's ability to adapt to large variations in the desired batch interval.
- **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. As discussed in Section 4, this is a complex multi-stage workload where partial aggregations over the batch intervals are re-used to compute aggregations 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 as partial aggregations over smaller intervals increases the computation of compared to larger intervals[4]. This workload tests the robustness of our algorithm against more complex workload characteristics that the algorithm was not explicitly designed for.

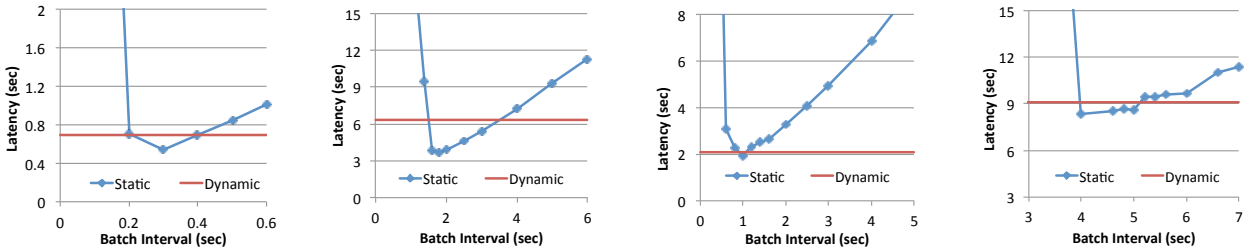
These characteristics may be specific to the cluster and processing framework used to run these workloads. It is important to note that we are not evaluating the performance of these workloads under the framework. Our goal is to evaluate our algorithm's ability to optimize and adapt under a

[4] 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 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.



(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.

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, highlighting the power of the algorithm – as long as the input data rate is sustainable by the cluster resources, the algorithm can achieve low latency without apriori knowledge about the workload.

5.2.2 Time-varying Input Data Rate

We varied the input data rate in each workload in a sinusoidal fashion with periodicity 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

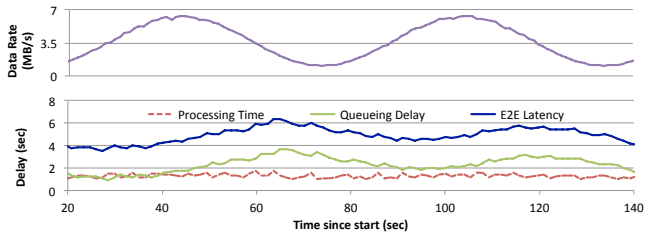


Figure 16: Timeline of the batch interval and other times for the *Reduce* workload with sinusoidal input data rate and static batch interval. The queuing delay builds up when the processing time exceeds above the batch interval, thus increasing the latency.

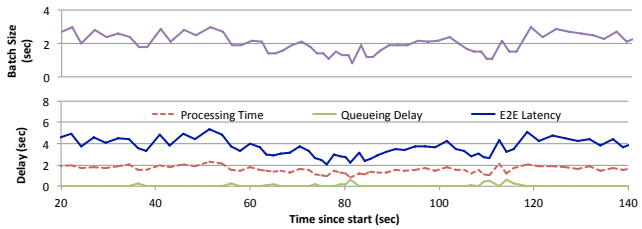


Figure 17: Timeline of the batch interval and other times for the *Reduce* workload with sinusoidal data rates as shown in Figure 16. Unlike static batch intervals, our algorithm adapts to increased load, thus preventing queue buildup and ensuring low latency.

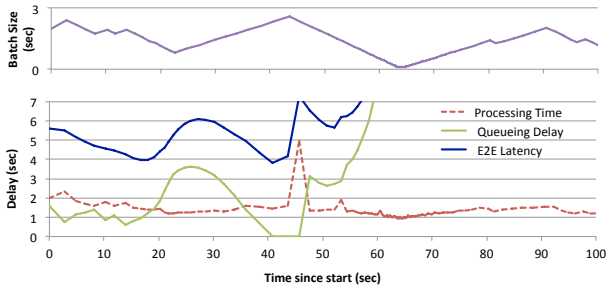


Figure 18: Timeline of batch interval and other times for the *Reduce* workload with a gradient based control algorithm. This highlights the vulnerability of this algorithm to noise.

processing 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.

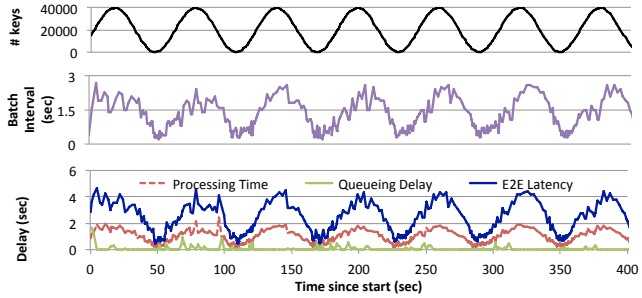


Figure 19: Timeline of number of keys, batch interval and other times for the *Reduce* workload with varying number of keys (data rate is 6 MB/s).

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 batch processing time depends on the number of keys in each batch. 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 able to quickly 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. Note that its agility prevents any significant queue buildup, which further illustrates its power 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 test out algorithm against variations in cluster resources. 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

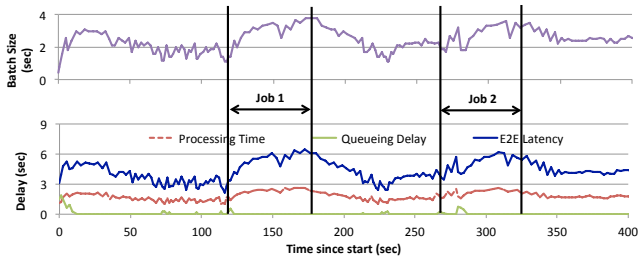
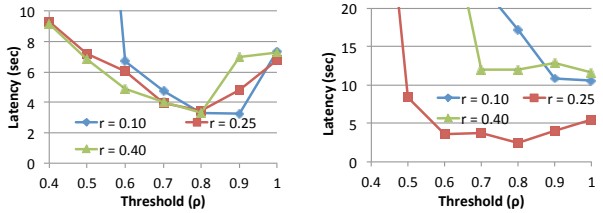


Figure 20: Timeline of batch interval and other times for the *Reduce* workload under variations in the available resources. Our algorithm is able to adaptively increase the batch interval when external batch jobs 1 and 2 take away 25% of the cluster resource.



(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.

shared cluster reduce the amount of resources available for the streaming workload running on the same cluster[5].

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

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

Control Loop Delay: This is the delay between the control decision (e.g., batch interval is increased) and its observed effect (i.e., the corresponding larger batch is processed). Like other control systems [21], our algorithm is vulnerable to large loop delays. Sudden large changes in the workload (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.

Other Adaptive Techniques: Other adaptive techniques like load shedding and elastic resources can be applied in conjunction with batch resizing. Interactions of these techniques with each other can be very complex, and therefore hard to analyze. In this paper, we narrowed the scope to batch resizing alone to keep the problem tractable, and have left the more general problem of the interaction of all these techniques for future.

7. Related Work

Stream Processing and Incremental Data Processing Systems: Our proposed algorithm is primarily focused on batched stream processing systems such as Comet [14] and Spark Streaming [25]. They collect received 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 [7], TeleGraphCQ [11], TimeStream [18], Naiad [16], and Storm [5] 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 processing systems such as CBP [15], Percolator [17] and Incoop [10] 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 load. Generally, this has not been explored.

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 ca-

[5] Assumption: reduced resources are still sufficient for streaming workload.

pacity of the system [9, 23]. 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 [8, 19, 24]. Others have proposed dynamic reconfiguration of the operator graph [18, 20]. 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.

8. Conclusion

In this paper, we have presented 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 adapt to variations in operating conditions for a wide range of workloads.

Acknowledgments

We thank the SOCC reviewers and our shepherd for their detailed feedback. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

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.apache.org`, .

[4] Spark streaming. `http://spark.apache.org/streaming/`, .

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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. . URL `http://doi.acm.org/10.1145/1807128.1807138`.

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

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

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.