NoStop: A Novel Configuration Optimization Scheme for Spark Streaming

Qianwen Ye, Wuji Liu, and Chase Wu
New Jersey Institute of Technology
Newark, New Jersey, USA
{qy57,wl87,chase.wu}@njit.edu

ABSTRACT

An increasing number of big data applications in various domains generate datasets continuously, which must be processed for various purposes in a timely manner. As one of the most popular streaming data processing systems, Spark Streaming applies a batch-based mechanism, which receives real-time input data streams and divides the data into multiple batches before passing them to Spark processing engine. As such, inappropriate system configurations including batch interval and executor count may lead to unstable states, hence undermining the capability and efficiency of real-time computing. Hence, determining suitable configurations is crucial to the performance of such systems. Many machine learning- and search-based algorithms have been proposed to provide configuration recommendations for streaming applications where input data streams are fed at a constant speed, which, however, is extremely rare in practice. Most real-life streaming applications process data streams arriving at a time-varying rate and hence require real-time system monitoring and continuous configuration adjustment, which still remains largely unexplored. We propose a novel streaming optimization scheme based on Simultaneous Perturbation Stochastic Approximation (SPSA), referred to as NoStop, which dynamically tunes system configurations to optimize real-time system performance with negligible overhead and proved convergence. The performance superiority of NoStop is illustrated by real-life experiments in comparison with Bayesian Optimization and Spark Back Pressure solutions. Extensive experimental results show that NoStop is able to keep track of the changing pattern of input data in real time and provide optimal configuration settings to achieve the best system performance. This optimization scheme could also be applied to other streaming data processing engines with tunable parameters.

KEYWORDS

Spark Streaming; Performance Optimization; Stochastic Approximation; Big Data Systems.

ACM Reference Format:
Qianwen Ye, Wuji Liu, and Chase Wu. 2021. NoStop: A Novel Configuration Optimization Scheme for Spark Streaming. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472515

ICPP'21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472515

1 INTRODUCTION

In the past decade, with the widespread deployment of pervasive services in e-commerce, e-finance, social networking, and many other fields, there is a rapidly increasing need to process streaming data in real time for various purposes such as security event detection, dynamic content delivery, and user profile analysis. To meet such demands, a plethora of computing engines for streaming data processing have been developed and used, including Apache Spark (Spark Streaming) [5], Apache Flink [3], Apache Storm [6], Apache Samza [9], Apache Apex [8], and Google Cloud Dataflow [15]. In fact, real-time streaming data processing engines have now become an essential building block in big data ecosystems, especially in business domains. Taking China as one example, the e-commerce transaction service of Alibaba, the most popular search engine from Baidu, and the social network platform of Tencent, all depend on Spark-based computing solutions at scale. Particularly, Tencent has eight hundred million active users generating over 1 PB of data per day, which is being processed on a cluster consisting of more than 8,000 computing nodes [19].

System stability and processing speed are among the utmost important performance metrics for streaming data processing. System stability is generally considered as the bottom line for streaming data processing. On the other hand, for a long-running streaming application, if the system fails to process data items as fast as they arrive, the data may accumulate at the input queue, hence leading to possible data loss or system failure eventually. In general, processing speed is determined by end-to-end latency that denotes the duration from the time when the system receives a data entry to the time when a corresponding output is produced. Obviously, such latency reflects the system's capability of providing real-time results for fast user response.

As one of the most popular systems for streaming data processing, Spark Streaming applies a batch-based mechanism, which receives real-time input data streams and splits the data into multiple batches, each of which is processed by Spark engine as a regular Spark job. As shown in Fig. 1, input data are divided into batches according to the setting of a batch interval, which defines the size of the batch in seconds. Typically, a longer batch interval results in a larger batch size, and hence incurs a longer batch processing time. Considering the processing dynamics, the stability of the system is largely affected by batch interval and batch processing time. If batch processing time is longer than batch interval, the batch queue may be overwhelmed by unprocessed batches, resulting in an unstable state. Hence, a larger batch interval is conducive to

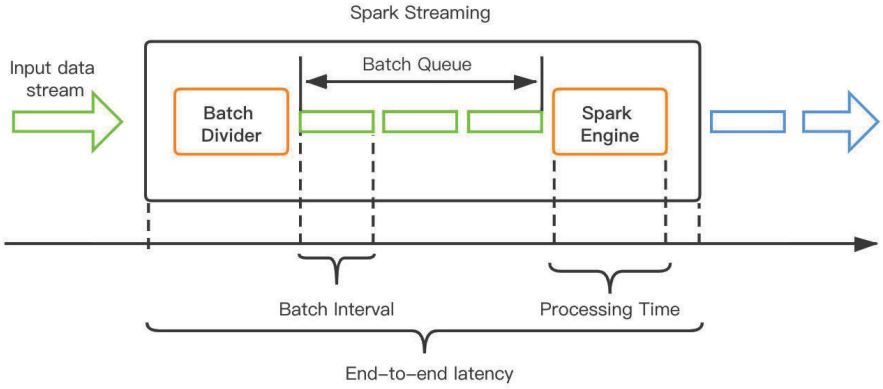


Figure 1: Spark Streaming model.

the stability of the system, but meanwhile may lead to a longer end-to-end delay. In addition to batch interval, the configuration of Spark engine, which mainly determines resource allocation, also has a critical impact on batch processing time. Therefore, it is of paramount importance to decide an appropriate batch interval and configure suitable Spark system parameters, especially in practical applications where data items arrive at a time-varying speed.

Optimizing the performance of streaming data processing is challenging because the compound effects of the aforementioned factors are complex and opaque to application users. In general, finding a proper system configuration requires significant domain knowledge and an in-depth understanding of the streaming data processing model adopted in big data systems. Due to the stochastic dynamics of streaming data processing in distributed environments and the lack of accurate performance models, it is extremely difficult, if not impossible, to analytically derive the best configuration setting for a given application. Especially, if the input data rate fluctuates without a predictable pattern, performance optimization becomes a non-deterministic, intractable problem. Due to the nature of the problem, misconfigured settings are not uncommon even in production systems and oftentimes lead to poor performance or system failure in extreme situations.

One naive approach to optimizing system configuration for streaming data processing would be to exhaust all possible settings and choose the "best" configuration. However, such exhaustive search or brute force approaches are prohibitively time-consuming when there is a large value range for the control parameters in the configuration space. This is actually almost always the case in real life, and thus is impractical especially for applications with variable input data rates.

We propose a NOvel STreaming OPtimization scheme, referred to as NoStop, based on Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm to dynamically and adaptively determine the most suitable system configuration. The proposed NoStop scheme is implemented and tested in real-life computing environments, and extensive experimental results show that it is able to keep track of the changing pattern of input data and provide the best configuration in real time to maintain system stability and minimize end-to-end delay. This optimization scheme could be used to automatically configure and execute streaming data applications in complex big data systems.

In sum, the work in this paper makes the following contributions to the area of big data processing:

- We construct the models for Spark Streaming and investigate the impact of batch interval and batch processing time on system stability and end-to-end delay. Based on such models and analyses, we formulate an optimization problem to minimize end-to-end delay while maintaining system stability in streaming data processing with variable input data rates.
- We propose NoStop, a streaming data processing performance optimization scheme based on SPSA algorithm, to quickly determine the most appropriate configuration setting in real time in response to time-varying system status with negligible overhead.
- NoStop significantly improves Spark Streaming performance for different application types in comparison with existing approaches.
- The proposed SPSA-based performance optimization framework is generic and hence is applicable to other big data computing systems to systematically and adaptively determine the most suitable system configuration.
- NoStop tackles hardware heterogeneity in a transparent manner and conducts real-time analytics for big data system performance optimization.

The rest of the paper is organized as follows. In Section 2, we conduct a survey of related work. In Section 3, we discuss the relationship between batch interval and batch processing time, analyze two performance-related factors, and formulate a performance optimization problem for streaming data processing. In Section 4, we detail the design of NoStop. We conduct real-life experiments for performance evaluation in Section 6. Section 7 concludes our work and sketches a plan for future research.

2 RELATED WORK

As driven by many emerging business and scientific applications, performance optimization for streaming data processing has been extensively studied in the literature and continues to be the focus of research in the field of big data processing. We conduct a brief survey of existing work in related areas.

Spark configuration for performance improvement has been investigated in various contexts. In [11], the effects of parameters are investigated and a systematic method for parameter tuning is proposed by Gounaris *et al.* They created a set of candidate configurations off-line through benchmark Spark jobs, and tested these configurations for Spark jobs to be executed. Petridis *et al.* identified the impact of the most important tunable parameters and proposed a trial-and-error methodology for tuning in an arbitrary application based on a very small number of experimental runs [21]. In [16], a simulated annealing algorithm is used to dynamically adjust parameters to obtain optimal Spark configuration. Also a least squares method is integrated to improve convergence speed. However, the spark applications are categorized based on job types, which are too general.

Many machine learning-based approaches have been proposed to tune Spark parameters. In [14], a binary classification and multi-classification machine learning model is proposed to accelerate the tuning of Spark parameters. In [20], Nguyen *et al.* constructed application-specific performance influence models based on three different machine learning models, namely, Artificial Neural Network, Support Vector Regression, and Decision Trees, and employed a recursive random search algorithm to tune configuration settings by leveraging the aforementioned models. Since they mainly focused on configuration recommendation for Spark instead of Spark Streaming, their suggested configuration may not be adaptive to changing input data even with the same fed-in data rate. In [13], Oliverira *et al.* focused on spark parameter tuning for scientific workflow. Instead of considering spark parameters solely, they also took domain-specific parameters into account, including the size of the DNA sequence. Decision trees were used to extract interpretable rules for parameter tuning guidance. However, it requires extensive experiments to train decision trees for different scientific workflows.

There also exist a number of efforts on Spark Streaming performance optimization. In [12], Cheng *el al.* proposed an adaptive scheduler that can automatically adjust scheduling parameters to improve performance and resource efficiency. Their work mainly targets applications where input data are fed at a constant speed, which is extremely rare in real life, and such systems are vulnerable to spikes of input data. So far, limited research efforts have been made to address varying data rates. In [10], Cheng *et al.* integrated an expert fuzzy control mechanism to dynamically change batch size in Spark Streaming to account for input fluctuations, which, however, requires significant historical data for training. In [18], Petrov *et al.* designed an adaptive performance model to identify the optimal amount of resources for a given Spark Streaming job. However, their work requires an ability to scale up cluster resources on demand, which is almost infeasible for non-cloud users in practice.

Different from the aforementioned work, we propose a stochastic approximation-based streaming optimization scheme to minimize end-to-end delay while keeping the system robust to varying input data rates by dynamically allocating computing resources and adjusting batch intervals. Compared with machine learning-based approaches, this scheme requires no historical data and incurs negligible computing overhead.

3 PROBLEM STATEMENT

3.1 Optimization Objectives

Maintaining system stability and minimizing end-to-end delay are closely related and are reflective of the relationship between batch interval and batch processing time. We characterize this relationship in the following three scenarios and discuss their respective effects on system performance:

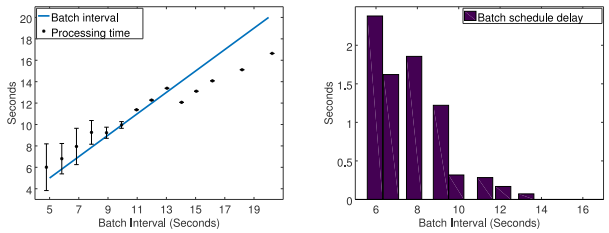
- When *Batch Processing Time* > *Batch Interval*: The unprocessed batches would pile up in the batch queue as the system fails to process data items as fast as they arrive. The system is therefore unstable and would even crash with long-running applications.
- When *Batch Interval* > *Batch Processing Time*: the system is stable because batches are retrieved from the batch queue and are processed immediately. However, computing resources are underutilized and Spark engine would sit idle waiting for batches to arrive, hence resulting in an unnecessarily long end-to-end delay.
- When *Batch Interval* = *Batch Processing Time*: This is the ideal situation, where there is no data accumulation in the input queue (hence the system is stable), and there is no idle time for the use of computing resources. However, this situation rarely happens because of complex system dynamics, data rate variation, and resource sharing complexity.

In addition to the relationship between batch interval and batch processing time, their settings also affect end-to-end delay. Minimizing end-to-end delay while maintaining system stability is equivalent to minimizing batch interval under a constraint that batch interval is no less than batch processing time.

3.2 Parameter Effects

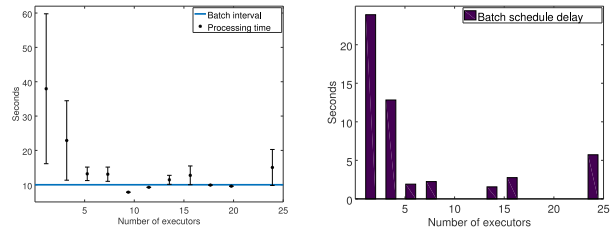
Streaming data processing in distributed environments is a complex process that involves the batch divider as well as Spark computing engine. Although Spark Streaming provides over 150 configurable parameters, not all of them play an equally important role in system performance, and some of them can only be configured at the beginning of Spark launching and remain unchanged during job execution. For example, the specification of executors, memory size, and number of CPU cores cannot be adjusted dynamically in response to varying input data rates. In this paper, we focus on two most influential parameters, namely, number of executors and batch interval, the latter of which is made tunable at runtime through system modification.

Batch interval decides the number of records in each batch, and it directly affects system stability and end-to-end delay. To instantiate our analysis, we collect and present the experimental results of Streaming Logistic Regression using Spark deployed on a local



(a) Batch interval vs. batch processing time (b) Batch interval vs. batch schedule delay

Figure 2: Illustration of effects of batch interval on the performance of streaming logistic regression using Spark Streaming.



(a) Number of executors vs. batch processing time (b) Number of executors vs. batch schedule delay

Figure 3: Illustration of effects of executor counts on the performance of streaming logistic regression using Spark Streaming.

testbed consisting of ten nodes to illustrate the effect of batch interval. We plot in Fig. 2a the comparison of batch processing time in response to different batch intervals. From these performance measurements, we observe that the batch processing time increases slowly as the batch interval grows. A small batch interval results in a limited number of records in each batch, and hence the overhead of initializing batch processing would be non-negligible. On the other hand, with a large batch interval, the time for actual data processing dominates the batch processing time. Particularly, when the batch interval is smaller than 10 seconds, the batch processing time is greater than the batch interval and hence the system is unstable and the processing of the next arriving batch is delayed, as shown in Fig. 2b. The batch schedule delay is defined as the time duration a batch must wait before it starts to be processed. For a batch whose batch processing time is smaller than its batch interval, it is processed immediately once ready, hence resulting in no batch schedule delay. In this experiment, when the batch interval is set to be around 10 seconds, the system yields the minimum end-to-end delay.

Executors are processes running on worker nodes and are responsible for executing individual tasks in Spark applications. They are launched with specific memory size and number of CPU cores at the beginning of a Spark application and run through its whole lifetime. To illustrate how the number of executors affects system performance, we compare the batch processing time in response to different numbers of executors with fixed batch intervals, as shown in Fig. 3a. These measurements show the same relationship as our theoretical analysis that a smaller number of executors lead to a longer processing time due to limited computing power. However, with an excessively large number of executors, the overhead of managing all executors and task execution would negatively affect the batch processing time. Also, launching more executors consumes more computing resources. We observe in Fig. 3a that when the number of executors are 10, 12, 18, and 20, the batch processing time is smaller than the batch interval, and therefore the system is stable. Particularly, when the number of executors is around 20, the batch processing time is the closest to the batch interval while the system still remains stable with the smallest end-to-end delay.

3.3 Problem Formulation

Based on the above analysis of optimization objectives and parameter effects, we define a generic Spark Streaming performance optimization problem, referred to as SSPO, to minimize the end-to-end delay of a Spark application while maintaining system stability:

Definition 3.1. Spark Streaming performance optimization problem (SSPO): Given a Spark Streaming application executed in a distributed computing environment and an input data stream arriving at a varying speed, we wish to find a proper setting for batch interval and number of executors in real time to achieve minimum end-to-end delay:

$$\underset{\textit{Batch Interval, Number of Executors}}{\text{argmin}} \quad \textit{End-to-end Delay}, \tag{1}$$

subject to the following constraint:

$$\textit{Batch Interval} \geq \textit{Batch Processing Time}, \tag{2}$$

where the constraint guarantees system stability.

4 DESIGN OF NOSTOP FOR STREAMING OPTIMIZATION

4.1 The Goals of NoStop Design

The main goal of NoStop is to achieve low end-to-end latency and maintain system stability. Meanwhile, we also consider the following properties of NoStop:

- Noise Tolerance: As shown in Fig. 3a and Fig. 2a, it is well recognized that randomness exists in the dynamics of streaming data processing in distributed environments, including network jitters, resource contentions, etc. One salient feature of NoStop is to explicitly account for such randomness.
- Generality: It is applicable to different types of Spark Streaming applications executed in different computing environments, e.g., heterogeneous/homogeneous cluster, cloud-based cluster, etc.
- Efficiency: It converges to minimum end-to-end delay promptly for fluctuating input data with negligible overhead.
- Performance Guarantee: It provides a theoretically proved performance bound.

4.2 SPSA-Based Optimization Algorithm

To meet the aforementioned goals, we employ Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm [22] to determine the optimal parameter setting at runtime.

4.2.1 Rationale of Using SPSA. In Spark Streaming, batch interval is in unit of milliseconds. Therefore, there are at least thousands of combinations of batch interval and number of executors, which makes it practically infeasible to conduct exhaustive search. In distributed computing environments, randomness is caused by network fluctuations, system dynamics, input data rate variations, etc. Since our work is focused on minimizing end-to-end delay, the Spark execution workflow could be treated as a "black box", where the input is the set of control parameters θ and the output is the objective $G(\theta)$. Based on this model, SPSA is suitable for finding the optimal configuration quickly as it has the following desirable features:

- It requires only two measurements of the objective function, regardless the dimension of θ in the optimization problem, which means that we only need to change the configuration twice in each iteration, hence greatly reducing the overhead for real-time configuration adjustment.
- It does not require an explicit formula of $G(\theta)$, which is essentially unknown, but only its "noise corrupted" measurements $y = G(\theta) + \xi$. Note that the results of $y(\theta, \xi)$ can be measured directly through Spark Streaming API [5].
- It incurs a low time complexity to determine the next optimization step.
- It has a proven convergence property (4.2.4), ensuring that each optimization step is effective.

4.2.2 Objective Function. Since batch interval is set by NoStop, to apply SPSA algorithm, we convert the optimization problem in Eq. 1 to the following:

$$\underset{\theta}{\operatorname{argmin}} \left(\textit{Batch Interval} + \rho \cdot \max\left(p, 0\right)\right), \tag{3}$$

where $\theta = \{\textit{Batch Interval}, \textit{Number of Executors}\}$, p denotes the difference between batch processing time and batch interval, and ρ is a penalty coefficient, increasing at a certain interval for each round of optimization up to a preset limit. This is because in the beginning of the SPSA optimization process, the gain sequence is large, and a large coefficient ρ may produce a large gradient, making the step size too large to approach the optimal point. As the SPSA optimization progresses with the increase of iteration k, the gain sequence becomes smaller, and we can increase the coefficient ρ to avoid unstable optimization results. However, an excessively large coefficient ρ would dilute the minimization goal of batch interval, and should be upper limited according to our empirical study.

4.2.3 SPSA Method. Stochastic Approximation (SA) is a family of iterative methods widely used for optimization problems. They can approximate extreme values of functions that cannot be computed directly, but only estimated via noisy observations [2].

In our problem, the goal is to find the optimal control parameters θ^* that minimize $G(\theta)$ within feasible space $\Theta, \theta \in \Theta$. Following the standard Kiefer-Wolfowitz Stochastic Algorithm (KWSA) [17], the standard stochastic approximation form is

$$\hat{\theta}_{k+1} = \hat{\theta}_k + a_k \cdot \hat{g}_k\left(\hat{\theta}_k\right),$$

where $\hat{\theta}_k$ denotes the control parameters θ at the k-th iteration, Gain Sequence a_k is a non-negative scalar, $g(\theta) = \frac{\partial G(\theta)}{\partial \theta}$ is the gradient of $G(\theta)$, and $\hat{g}_k\left(\hat{\theta}_k\right)$ is an approximation of $g(\theta_k)$ with simultaneous perturbation.

We define a "noise corrupted" measurement of the objective function as $y(\theta), \theta \in \Theta$, and $y(\theta) = G(\theta) + \xi$, where ξ denotes the random impact of the system at runtime. In general, $y(\theta)$ is the objective value under control parameters θ during a specific time duration.

The gradient of $G(\theta)$ can be approximated by

$$\hat{g}_k\left(\hat{\theta}_k\right) = \frac{y\left(\hat{\theta}_k + c_k\right) - y\left(\hat{\theta}_k - c_k\right)}{2c_k},$$

where $c_k > 0$. Note that the coefficients a_k and c_k in the above equations should satisfy the following conditions to guarantee the convergence [17]:

$$\lim_{k\to\infty} a_k = 0, \lim_{k\to\infty} c_k = 0, \sum_{k=1}^{\infty} a_k = \infty, \sum_{k=1}^{\infty} \left(\frac{a_k}{c_k}\right)^2 < \infty.$$

Furthermore, to accelerate configuration optimization, we employ the Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm [22], which generates the simultaneous perturbation approximation to the unknown gradient $g(\hat{\theta}_k)$:

$$\hat{g}_k\left(\hat{\theta}_k\right) = \frac{y\left(\hat{\theta}_k + c_k\Delta_k\right) - y\left(\hat{\theta}_k - c_k\Delta_k\right)}{2c_k} \begin{bmatrix} \Delta_{k1}^{-1} \\ \Delta_{k2}^{-1} \\ \vdots \\ \Delta_{kp}^{-1} \end{bmatrix},$$

where p is the dimension of θ, and Δ_{ki} ($i = 1, 2, ..., p$ for p-dimensional control parameters) is independent and symmetrically distributed around 0 with finite inverse $E\left|\Delta_{ki}^{-1}\right|, (i = 1, 2, \cdots, p)$ in the k-th iteration [22].

In SPSA, the gain sequence is

$$a_k = \frac{a}{(A + k + 1)^{\alpha}}, c_k = \frac{c}{(k + 1)^{\gamma}},$$

where α and γ are oftentimes set to be 0.602 and 0.101, respectively (practically effective) [22], and k is the number of iterations. We will discuss the choices of a, A and c in Section 5.6.

4.2.4 Convergence of SPSA. The convergence of the proposed SPSA-based NoStop scheme is important as it affects both the quality and the efficiency of configuration optimization. To explore the applicability of SPSA and investigate its convergence property, we need to validate the conditions that ensure the convergence of θ^* in the context of SSPO.

As pointed out by Spall in [23] (pp. 161), the conditions for convergence can hardly be all verified in real-life problems since $G(\theta)$ is practically unknown. We provide the following arguments to justify the use of SPSA in our problem.

According to Theorem 7.1 in [23] (pp. 186), if Conditions $B.1'' - B.6''$ hold and θ^* is a unique minimum of $G(\theta)$, then for SPSA, θ_k almost surely converges to θ^* as $k \to \infty$.

Conditions $B.1''$, $B.4''$, and $B.6''$ are the most relevant since we govern the gains sequences a_k and c_k and the random perturbation Δ_k. $\sum_{k=0}^{\infty} \frac{a_k^2}{c_k^2} < \infty$ in Condition $B.1''$ balances the decay of a_k against the decay of c_k. Specifically, it prevents c_k from going to zero too quickly, thereby preventing the gradient estimate from becoming too wild and overpowering the decay associated with a_k. Conditions $B.1''$ can be easily validated by the coefficient sequences a_k and c_k we choose and the symmetric Bernoulli ± 1 distribution we follow to generate the simultaneous perturbations $\{\Delta_{ki}\}$.

Conditions $B.4''$ to $B.6''$ on the perturbation distribution and smoothness of L guarantee that the gradient estimate $\hat{g_k}(\hat{\theta}_k)$ is an unbiased estimate of $\hat{g}(\hat{\theta}_k)$ within error $O(c_k^2)$.

The simultaneous perturbations $\{\Delta_{ki}\}$ we generate ensure that Δ_{ki} is a mutually independent sequence, symmetrically distributed around zero and uniformly bounded in magnitude for all k, i, and is independent of $\hat{\theta}_1, \hat{\theta}_2, \cdots, \hat{\theta}_k$. Hence, Condition $B.6''$ holds.

The noise in Spark streaming applications is mainly caused by the dynamics and randomness of distributed computing systems and network devices, and is essentially unpredictable. Since the loss measurements $y(\cdot)$ are corrupted by both positive noise and negative noise, the long-term conditional expectation of the observed noise can be considered as zero, i.e., for all k, $E[\varepsilon_k^+ - \varepsilon_k^- | \{\hat{\theta}_1, \hat{\theta}_2, \cdots, \hat{\theta}_k\}, \Delta_k] = 0$. As $\{\Delta_{ki}\}$ is generated following the symmetric Bernoulli ± 1 distribution with a probability of 0.5 for each outcome of either $+1$ or -1, $E[\Delta_{ki}^{-1}]$ is uniformly bounded. In addition, the loss measurements $y(\hat{\theta}_k \pm c_k \Delta_k)$ are bounded by the feasible regions of controlling parameters, so the ratio of measurement to perturbation $E[y(\hat{\theta}_k \pm c_k \Delta_k)\Delta_{ki}]$ is uniformly bounded over i and k. Therefore, Condition $B.4''$ holds.

$B.5''$ asks $G(\theta)$ to be three-times continuously differentiable and bounded by R^p. As the feasible regions of the control parameters are finite and the computing resources are limited, the end-to-end delay is obviously bounded. However, the smoothness and differentiability of $G(\theta)$ is very difficult to verify due to the lack of knowledge on $G(\theta)$. For this optimization problem, we assume that $G(\theta)$ meets this condition.

Conditions $B.2''$ and $B.3''$ impose the requirement that $\hat{\theta}_k$ (including the initial condition) is close enough to θ^* so that there is a natural tendency for an analogous deterministic algorithm to converge to θ^*. These two conditions are intuitively satisfied in our optimization scenario because: i) $sup_{k \geq 0} \left\| \hat{\theta}_k \right\| < \infty$ can be satisfied since the control parameter values (batch interval and number of executors) are both finite positive integer numbers; ii) since the feasible regions of the control parameters in Spark application execution are finite and mapped to a limited range of iterative variables, $\hat{\theta}_k$ (including the starting point) is sufficiently close to θ^*; iii) in Spark streaming processing environments, due to the system dynamics and randomness, different runs with identical parameter settings may yield different end-to-end delay, which makes θ^* be not a single point but an "acceptable area" in order to tackle this optimization problem.

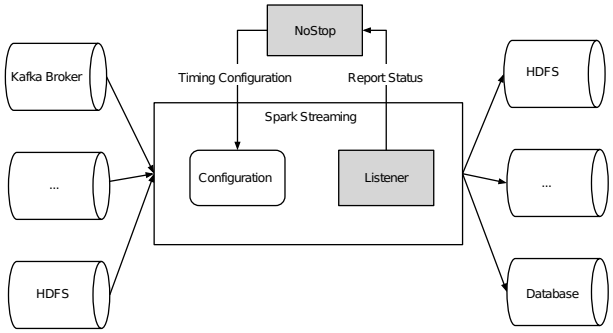


Figure 4: NoStop architecture.

With all conditions $B.1'' - B.6''$ satisfied in SSPO, the SPSA-based NoStop scheme is able to converge to the minimum end-to-end delay.

4.3 System Architecture

The NoStop architecture is shown in Fig. 4, where Spark Streaming receives input data streams from multiple channels for processing, and output data to Hadoop Distributed File System (HDFS) for storage or other Databases for management. We design Spark Streaming Listener to report real-time system status to NoStop in JSON format. Based on each newly updated performance vector, NoStop computes the next-step configuration parameters and makes a real-time adjustment to the system configuration. NoStop is capable of optimizing system configurations online without rebooting the entire cluster, hence greatly reducing operational cost and computing overhead.

Note that in the current NoStop scheme, we focus on two tunable system parameters, i.e., the number of executors and batch interval, as explained in Section 3.2.

5 IMPLEMENTATION OF NOSTOP

NoStop is implemented in Java and integrated into Spark engine. We present the implementation details of NoStop in this section.

5.1 Determine Configuration Range

According to the cluster capacity and the resources allocated to each executor, we determine the range for the number of executors:

$$Min_{Executors} <= \theta_{Executors} <= Max_{Executors}.$$

Similarly, we determine the range for batch interval according to the application requirement:

$$Min_{Batch\ Interval} <= \theta_{Batch\ Interval} <= Max_{Batch\ Interval}.$$

For convenience, we use $\theta = \{\theta_{Executors}, \theta_{Batch\ Interval}\}$ to denote the configuration vector. We apply a scale function (e.g., min-max normalization) to normalize parameters into the same range.

5.2 Set SPSA Parameters

We choose an either fixed or random starting point $\theta_{initial}$ within the configuration space, and set nonnegative coefficients a, c, A, α,

and γ in the SPSA gain sequence $a_k = \frac{a}{(k+1+A)^\alpha}$ and $c_k = \frac{c}{(k+1)^\gamma}$. The choices of A, a and c are discussed in Section 5.6.

5.3 Optimize Configuration

5.3.1 Generate Random Perturbation Vector. We generate a two-dimensional random perturbation vector Δ_k, where each component of Δ_k is independently generated from a zero-mean symmetric Bernoulli ±1 distribution with probability of $\frac{1}{2}$ for each ±1 outcome.

5.3.2 Collect Two Performance Observations. We run two perturbed executions and collect two corresponding performance observations:

$$y(\theta_k^+) = y(\hat{\theta}_k + c_k\Delta_k),$$
$$y(\theta_k^-) = y(\hat{\theta}_k - c_k\Delta_k).$$

5.3.3 Approximate Gradient. We generate the simultaneous perturbation approximation to the unknown gradient $\hat{g}_k\left(\hat{\theta}_k\right)$:

$$\hat{g}_k\left(\hat{\theta}_k\right) = \frac{y(\theta_k^+) - y(\theta_k^-)}{2c_k\Delta_k}.$$

5.3.4 Update θ Estimate. We use the standard SA form $\hat{\theta}_{k+1} = \hat{\theta}_k - a_k\hat{g}_k\left(\hat{\theta}_k\right)$ to update $\hat{\theta}_k$ to a new value $\hat{\theta}_{k+1}$.

5.3.5 Continue or Pause Optimization. The above process is repeated with the increasing iteration number k until a pause condition is met. Once NoStop reaches the optimal configuration, it halts the optimization process until the system becomes unstable (e.g., there is an abrupt change in the input data rate). We employ impeded progress rules to guarantee optimization halt with satisfactory performance: if the standard deviation of the end-to-end delay resulted from N best configurations is smaller than a threshold S, we pause the optimization process.

5.4 Collect System Metrics

According to the features of Spark Streaming, we consider the following rules for system metrics collection:

- The first processed batch after changing configurations is not considered, because after each configuration adjustment, Spark Streaming performs a series of initialization tasks including sending application jar to the newly added executors, which lead to a longer processing time.
- System metrics are collected for a certain number of batches, and the average processing time is calculated and recorded. To enhance the robustness of NoStop with varying input data rates, we follow an additive-increase rule to gradually relax on the number of batches for performance calculation: if the system is in the optimal state based on the current measurements, for each newly completed batch, we increase the number of collected batches by one. Also, we prevent insensitivity to system status change by setting a maximum number of batches allowed to be collected. This method can prevent the system from starting the next round of optimization due to the system's temporary unstable state.

5.5 Handle Different Input Data Rates

In real-life streaming applications, the arriving speed of input data always varies over time. If the fluctuation range of the input data rate is relatively small, SPSA treats this fluctuation as noise, which is explicitly accounted for. However, in some scenarios, there may be surges in traffic (e.g., E-commerce promotion, spike activities, etc.). After going through a large number of iterations, SPSA may generate a small step size, resulting in a tardy process of configuration optimization. To prevent this, we set a threshold for input data speed variation $threshold_{speed}$. If the standard deviation of the recent input data speed is greater than this threshold, it triggers NoStop to reset the coefficients and restart the optimization process.

5.6 Choose Gain Sequences A, a_k, and c_k

The choices of the gain sequences A, a_k, and c_k are critical to the performance of SPSA (as in all stochastic optimization algorithms) [22]. In our implementation, we set A, a, and c before running NoStop, as guided by the following rules based on the suggestions in [22].

- A is much less than (usually 10% or less of) the maximum number of iterations expected. Our empirical study recommends setting $A = 1$.
- a, which determines the step size generated after each iteration, is recommended to be set as half of the configuration range.
- c is set to be approximately the standard deviation of measurement $y(\theta)$.

The pseudocode of NoStop is provided in Algorithms 1 and 2, and the functions are described in Table 1.

Algorithm 1: NoStop

Input: $A, a, c, \theta_{initial}$
Initialize $\alpha = 0.602, \gamma = 0.101, k = 0, \rho = 1$;
$N = getDimension(\theta_{initial})$;
$x = \theta_{initial}$;
while *True* **do**
  if *needResetCoefficient()* **then**
    resetCoefficient();
  $k++$;
  $a_k = a \div (k + 1.0 + A)^\alpha$;
  $c_k = c \div (k + 1.0)^\gamma$;
  $\Delta = getDelta(N)$;
  $\theta^+ = checkBound(x + c_k \times \Delta)$;
  $\theta^- = checkBound(x - c_k \times \Delta)$;
  $grad = \frac{adjust(\theta^+,\rho) - adjust(\theta^-,\rho)}{2\Delta c_k}$;
  $\rho = \rho + 0.1$;
  $\rho = \min(\rho, 2)$;
  $x = checkBound(x - a_k \times grad)$;
end

Table 1: Descriptions of functions used in NoStop.

Function	Description	Return
getDimension(θ)	Return the dimension of vector θ	Integer
needResetCoefficient()	If the input speed changes significantly, reset the coefficient	Boolean
resetCoefficient()	Reset $k = 0, x = \theta_{initial}$	Void
getDelta(n)	Return n-dimensional random perturbation	Vector
checkBound(θ)	Return θ within the configuration range	Vector
changeConfigurations(θ)	Dynamically adjust the configuration of Spark Streaming	Void
getSystemStatus	Return the real-time system status	Object
satisfyPauseCondition(status)	Check whether the current state satisfies the pause condition	Boolean

Algorithm 2: Adjust Function

Input: θ, ρ
Output: Objective function value
$changeConfigurations(\theta)$;
$batchInterval = \theta_{BatchInterval}$;
while *True* **do**
  $(batchProcessingTime, status) = getSystemStatus()$;
  if *satisfyPauseCondition(status)* **then**
    /* Prevent fetch data frequently. */
    sleep(10 seconds);
  else
    break;
  end
end
/* Objective function. */
$G = batchInterval + \rho \cdot max(0, batchProcessingTime - batchInterval)$;
return G;

6 PERFORMANCE EVALUATION

In this section, we conduct a set of Spark Streaming data processing experiments on different types of workloads to examine the optimization evolution process and evaluate the performance of NoStop.

6.1 Cluster Setup and Streaming Workloads

In our experiments, we consider a heterogeneous cluster of five nodes, one master node and four worker nodes, as detailed in Table 2. All of these nodes have CentOS8 installed as well as Apache Hadoop 3.2.1 [7], Apache Spark 3.0.0 [5] compiled with NoStop, Apache Kafka 2.5.0 [4] and OpenJDK 1.8. We deploy Spark Streaming in the standalone mode and deploy Kafka Broker on each node. To prevent performance bottlenecks in Kafka, we set the number of Kafka partitions to be larger than the number of cores owned by the entire cluster.

Furthermore, we deploy a streaming data generator outside the cluster, which sends data to Kafka Brokers at varying data rates. The data are sent to each Kafka Broker uniformly to avoid data skew. We set up and execute four computing workloads: Logistic Regression, Linear Regression, WordCount, and Page Analyze. Logistic Regression and Linear Regression belong to machine learning algorithms, which require multiple iterations of processing. WordCount is a CPU-intensive application. Log Analyze simulates the common scenarios in industry, receiving Nginx [1] log from Kafka, washing and analyzing data, and writing results back into HDFS.

Table 2: List of cluster nodes.

Node ID	CPU	Disk	Type
1	I5-9400 2.9GHz	SSD	Master
2	I5-9400 2.9GHz	SSD	Worker
3	Xeon Bronze 3204 1.9GHz	HHD	Worker
4	I5-10400 2.9GHz	HHD	Worker
5	I5-10400 2.9GHz	HHD	Worker

6.2 Experimental Settings

6.2.1 Parameter Settings. In our experiments, we allocate one CPU core and 1GB of memory to each executor. According to our cluster capacity, we set the parameter range of NoStop as follows:

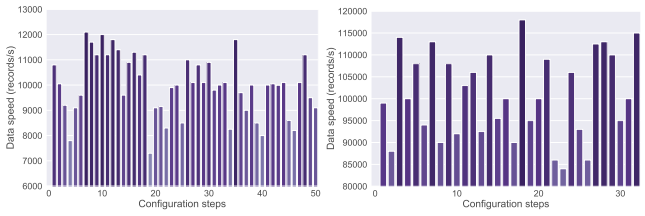
$$1 <= Num_{Executor} <= 20,$$
$$1 <= Batch\ Interval <= 40.$$

Then, we scale two parameters into the same range ($[1, 20]$) to produce better results. Following the guidelines in Section 5.6, we set coefficients $A = 1$, $a = 10$, $c = 2$, and the initial point in the middle of the parameter range, $\theta_{initial} = \{10, 10\}$. For the pause condition, we set the number of consecutive optimization rounds $N = 10$, and the standard deviation $S = 1$.

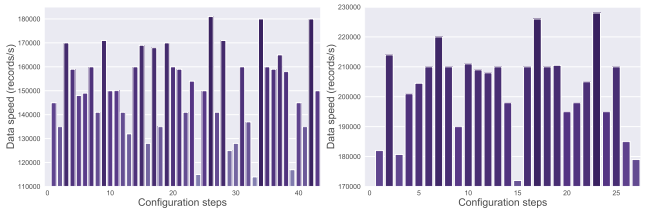
6.2.2 Variation in Input Data Rates. The data generation process in real-life applications is affected by different factors, such as the nature of the data source, the tool used for data collection, etc., hence oftentimes resulting in varying data rates. To evaluate the robustness of NoStop in handling such situations, we trigger the data generator to send data items at a random rate within a certain range:

$$MinRate <= Rate <= MaxRate.$$

This range is typically determined based on the application scenario being considered. In practice, the input data rate could also be restricted in the streaming data processing system to avoid instantaneous surge rates (e.g., by controlling the Kafka producing rate).

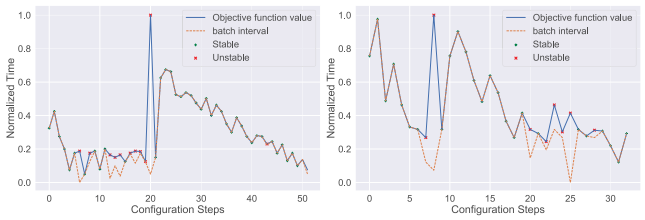


(a) Logistic Regression (b) Linear Regression

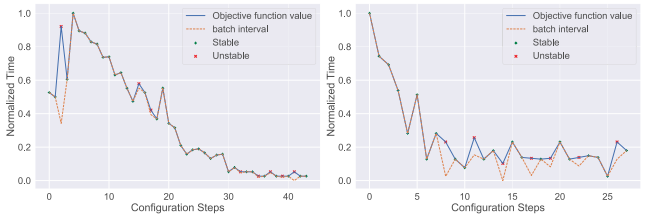


(c) WordCount (d) Page Analyze

Figure 5: Performance measurements in response to varying data arriving speeds with four different types of workloads.



(a) Logistic Regression (b) Linear Regression



(c) WordCount (d) Page Analyze

Figure 6: Optimization evolution process with four different types of computing workloads.

As shown in Fig. 5, in our experiments, we vary the arriving speed of data items within the range of [7000, 13000] for Logistic Regression, [80000, 120000] for Linear Regression, [110000, 190000] for WordCount, and [170000, 230000] for Page Analyze.

6.3 Improvement Over Default Configuration

We collect and plot in Fig. 6 the performance measurements and batch interval as the optimization process proceeds. With the machine learning workloads, the optimization process appears more dynamic than the others. It shows that even the data input speed

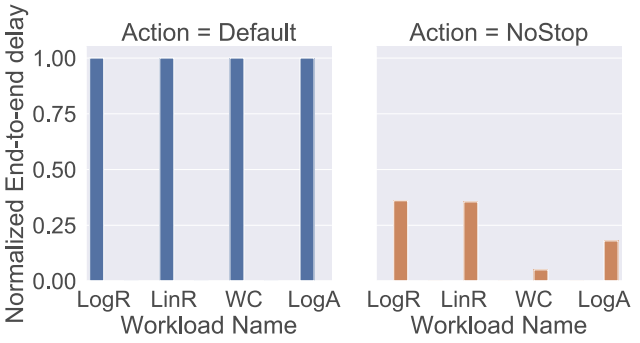
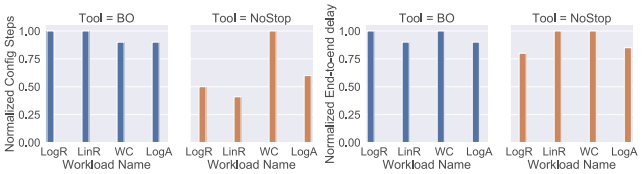


Figure 7: Performance improvement over initial configurations set by default.



(a) Comparison of steps (b) Comparison of end-to-end delay

Figure 8: Comparison between SPSA and Bayesian optimization.

changes overtime, and the batch interval can keep decreasing while maintaining the stability of the system. This is because in machine learning algorithms, the processing time of each batch is not identical, and the batch processing time of an unfitted model usually takes longer than that of a fitted model. Different random data may have different number of iterations. WordCount is a simple workload as it only requires two mapping/reducing operations and has a fixed processing flow. Hence, its processing time is the most stable than the other workloads. In Log Analyze, we use several Transformations and Actions to perform various analysis. Although the operations are complex, the processing time for each batch remains nearly the same, and hence it carries out a relatively stable optimization process.

Considering the randomness in the streaming optimization process, we repeat NoStop optimization experiments five times for each workload and plot the average performance measurement with the standard deviation for each workload in Fig. 7. These results illustrate that NoStop significantly reduces end-to-end delay in comparison with the system's default configurations.

6.4 Comparison with Random Search

Similar to Stochastic Approximation, Random Search does not require an explicit analytical form for optimization. Bayesian Optimization is among the most commonly used algorithms in Random Search, and has been widely employed for hyper-parameter tuning in machine learning.

Again, we repeat each experiment five times and compute the average measurement and standard deviation. As shown in Fig. 8,

the final optimization results are comparable, but the search time and configure steps of SPSA are less than that of Bayesian Optimization, which clearly illustrates the run-time efficiency of NoStop.

7 CONCLUSION AND FUTURE WORK

As data volumes continue to grow in many domains, the importance of streaming data processing has been increasingly recognized in both system research and application communities. We proposed an SPSA-based NoStop scheme to dynamically tune Spark Streaming configurations to optimize the performance of Spark-based data processing. Our analysis shows that SPSA is a good choice for parameter optimization and extensive experiments illustrate the efficacy of the proposed scheme in processing different types of computing workloads.

We will improve the performance and applicability of our streaming optimization scheme in multiple directions. For example, the SPSA algorithm is able to optimize multiple parameters simultaneously without additional overhead. In our current design of NoStop, we consider only two parameters for tuning. Also, it is still a challenging task for end users, who are primarily domain experts, to choose appropriate gain sequences. We plan to incorporate more system parameters in the optimization framework to reap the full benefits of SPSA. It is also of our future interest to design intelligent approaches to determine gain sequences systematically based on some user-level knowledge such as cluster capacity and throughput estimate.

ACKNOWLEDGMENT

This research is sponsored by U.S. National Science Foundation under Grant No. CNS-1828123 with New Jersey Institute of Technology.

REFERENCES

[1] [n.d.]. *Nginx*. http://nginx.org/
[2] [n.d.]. *Wikipedia stochastic approximation*. https://en.wikipedia.org/wiki/Stochastic_approximation
[3] Apache. 2011. Flink. https://flink.apache.org/.
[4] Apache. 2011. Flink. http://kafka.apache.org/.
[5] Apache. 2016. Spark. http://spark.apache.org.
[6] Apache. 2016. Storm. http://storm.apache.org.
[7] Apache. 2016. Hadoop. http://hadoop.apache.org.
[8] Apache. 2018. Apex. http://apex.apache.org/.
[9] Apache. 2019. Samza. http://samza.apache.org/.
[10] Dazhao Cheng, Xiaobo Zhou, Yu Wang, and Changjun Jiang. 2018. Adaptive scheduling parallel jobs with dynamic batching in spark streaming. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2672–2685.
[11] A. Counaris and J. Torres. 2018. A methodology for spark parameter tuning. *Big data research* 11 (2018), 22–32.
[12] Y. Chen D. Cheng and etc. 2017. Adaptive scheduling of parallel jobs in spark streaming. In *IEEE Conference on Computer Communications (INFOCOM 2017)*. Atlanta, GA, USA.
[13] C. Boeres D. Oliveira, F. Porto and etc. 2020. Towards optimizing the execution of spark scientific workflows using machine learningâĂŘbased parameter tuning. *Concurrency and Computation Practice and Experience* 33, 5 (2020).
[14] B. He etc. G. Wang, J. Xu. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In *2016 IEEE 18th Intl. Conf. on HPC and Communications*. Sydney, NSW, Australia.
[15] Google. 2015. Google Cloud Dataflow. https://cloud.google.com/dataflow.
[16] W. Chen H. Du, P. Han and etc. 2018. Otterman: A Novel Approach of Spark Auto-tuning by a Hybrid Strategy. In *2018 5th International Conference on Systems and Informatics (ICSAI)*. Nanjing, China, 831–836.
[17] J. Wolfowitz J. Kiefer. 1952. Stochastic Estimation of the Maximum of a Regression Function. *Ann. Math. Statist* 23, 3 (1952), 462–466.
[18] D. Nasonov M. Petrov, N. Butakov and etc. 2018. Adaptive performance model for dynamic scaling Apache Spark Streaming. *Procedia Computer Science* 136 (01 2018), 109–117.
[19] P. Wendell M. Zaharia, R.S. Xin and etc. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
[20] M. Khan N. Nguyen and K. Wang. 2018. Towards Automatic Tuning of Apache Spark Configuration. In *2018 IEEE 11th Intl. Conf. on Cloud Computing (CLOUD)*. San Francisco, CA, USA.
[21] P. Petridis1 and A. Gounaris1. 2016. Spark Parameter Tuning via Trial-and-Error. *Advances in Intelligent Systems and Computing* 529 (2016), 226–237.
[22] James C. Spall. 1998. Implementation of the simultaneous perturbation algorithm for stochastic optimization. *IEEE Trans. Aerospace Electron. Systems* 34 (July 1998), 817 – 823.
[23] James C. Spall. 2005. *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons.