Towards Automatic Parameter Tuning of Stream Processing Systems

Muhammad Bilal
Université catholique de Louvain

Marco Canini
KAUST

ABSTRACT

Optimizing the performance of big-data streaming applications has become a daunting and time-consuming task: parameters may be tuned from a space of hundreds or even thousands of possible configurations. In this paper, we present a framework for automating parameter tuning for stream-processing systems. Our framework supports standard black-box optimization algorithms as well as a novel gray-box optimization algorithm. We demonstrate the multiple benefits of automated parameter tuning in optimizing three benchmark applications in Apache Storm. Our results show that a hill-climbing algorithm that uses a new heuristic sampling approach based on Latin Hypercube provides the best results. Our gray-box algorithm provides comparable results while being two to five times faster.

CCS CONCEPTS

• **Social and professional topics → Management of computing and information systems**;

ACM Reference Format:
Muhammad Bilal and Marco Canini. 2017. Towards Automatic Parameter Tuning of Stream Processing Systems. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 12 pages.
https://doi.org/10.1145/3127479.3127492

1 INTRODUCTION

The use of stream processing is rapidly increasing in industry because real-time streaming analytics allows companies to react to changes in data quickly [8]. For example, an e-commerce company can use streaming analytics to create targeted ads and real-time promotional offers for its customers. A study showed that more than 90% of surveyed organizations plan to increase investment in stream processing [8]. Dozens of companies are already using Apache Storm [5] and several others are likely using other stream-processing frameworks.

Popular stream-processing systems such as Apache Storm [3], Heron [32], Apache Flink [1] and Spark Streaming [2] have dozens of available configuration parameters. The performance of big-data applications that utilize these frameworks for real-time processing crucially depends on parameter tuning. For example, Apache Storm has about 201 parameters that can be set by the user. Among these, there are 49 parameters that can be controlled on a per-application basis. Not all configurable parameters affect performance; however, several of these parameters should be tuned to achieve performance goals, such as meeting Service Level Objectives (SLOs) for (tail) latency and throughput in production deployments. Moreover, suitable parameter configurations vary depending upon the available resources, workloads and applications.

Currently, these parameters are manually tuned, possibly requiring several hours of investigation and testing by performance engineers [4, 6, 10, 12, 18]. In addition, performance engineers may require detailed knowledge of the stream-processing system itself to find a configuration that meets the performance requirements of an application because a best-practices approach is not always sufficient—a case that we make later in our evaluation. This makes configuration optimization a daunting and time-consuming task; we believe that automating configuration optimization is a viable alternative to manual tuning.

The configuration parameters in big-data frameworks can be broadly classified into two categories: (*i*) resource allocation parameters and (*ii*) application- or system-specific parameters. Resource allocation has been the subject of prior work, especially in conjunction with MapReduce jobs [29, 42]. On the other hand, there have been fewer studies on determining the optimal application-specific parameters for big-data frameworks. Most existing works present solutions specific to MapReduce jobs [20, 28, 31]. Only a few recent studies have considered performance optimization for stream-processing systems like Storm [30, 44]. Although BO4CO [30] deals directly with parameter tuning, the focus in this work is on latency optimization and not on throughput. To the best of our knowledge, a broader consideration of complete performance optimization in stream-processing systems has yet not been attempted.

Stream-processing systems are unique and present different challenges than those related to tuning databases and batch-processing systems. First, stream-processing applications are long-running (potentially infinite) programs. Tuning such applications requires determining an experiment's duration such that it is long enough to provide accurate performance measurements of a configuration yet short enough to quickly converge. Second, there are generally two metrics of interest that are jointly optimized: throughput and latency (typically at a high latency distribution percentile). Tuning is thus a case of multi-objective optimization. Most previous works considered single metrics [20, 24, 28, 30, 31] rather than multiple metrics. Third, measuring latency with no overhead is challenging when throughput is high, such as at hundreds of thousands of processed data items per second. Without sacrificing accuracy to measure latency, we turn to a probabilistic data structure called T-Digest (§5), which is able to handle millions of latency numbers

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127492

per second. Fourth, unlike MapReduce-based frameworks, stream-processing applications are generally multi-stage data processing pipelines. Thus, we need to configure the parameters and the level of parallelism of each stage, yet each stage is also dependent on its interconnected stages and only a suitable balance in the levels of parallelism leads to the best achievable performance.

In this paper, we introduce a framework for automated off-line parameter tuning of stream-processing systems. We use iterative optimization algorithms on measurements from actual runs of streaming applications, from which the framework collects feedback. We focus on Apache Storm as a stream-processing system for our experiments. However, several of the concepts can be directly translated to other stream-processing systems because component-level parallelism is common across all stream-processing systems and the black-box approaches presented here are not bound to specific parameters or systems.

In summary, we make the following main contributions:

- A framework for automated parameter tuning of stream-processing systems along with an implementation of five optimization algorithms. Our framework is released as open-source code [11].
- A novel gray-box optimization algorithm based on a rule-based heuristic approach.
- A new sampling method for the hill-climbing algorithm that accounts for desirable initial configurations of stream-processing applications.
- An evaluation of the performance of different algorithms using three benchmark applications for Apache Storm.

2 PRELIMINARIES

2.1 Stream-Processing Systems

Stream-processing systems are software systems that are designed to analyze and act on (potentially unbounded) streaming data, generally in real time or in near real time (considered as having latencies in the range of sub-second up to a few minutes). Stream-processing systems have the ability to perform streaming analytics, i.e., to continuously calculate mathematical or statistical analytics on the data stream. Modern stream-processing systems are constructed on distributed, scalable, and fault-tolerant architectures that handle high volumes of data in real time.

There are several open-source stream-processing systems currently available and the number of systems is growing steadily. Apache Storm [3], Heron [32], Apache Flink [1] and Spark Streaming [2] are a few examples of production-grade stream-processing systems. In this paper, we use Apache Storm as a case study; however, our concepts and approach are not specific to Storm and can be generalized to other systems.

Apache Storm is a distributed, real-time stream-processing system written in Java. An application in Storm is called a topology. A Storm topology is a Directed Acyclic Graph (DAG) composed of nodes that perform computation and edges that represent the flow of data being passed between nodes. The nodes send data between one another in the form of tuples. A tuple is an ordered list of values. The data stream originates at the source(s) node of the DAG, which is called a "spout". A spout generates a stream of tuples that are consumed and processed by "bolt" components according to a user-defined logic that defines the processing performed on the tuples at the node.

Apache Storm provides an application programming interface (API) to access metrics on the cluster, topology and component levels. The metrics that are provided include: average latency, number of tuples processed and per-component capacity. Capacity provided by Storm's metrics API is the utilization of each component/executor. We use the number of tuples processed by the topology as well as the per-component capacity for our framework.

2.2 Problem Formulation

Consider a system for which we want to perform parameter tuning for a set of N parameters. The goal of the configuration optimization process is to find the configuration, X, i.e., the vector of assigned parameter values, $X = x_1, \ldots, x_N$, that provides the desired performance according to a specific metric of interest (e.g., throughput or latency). The possible values of these parameters come from range R_{x_i} $\forall\ x \in X$. $DOM = \prod_{i=1}^{N} R_{x_i}$ represents the space of all possible configurations that the system can have. More concretely, let y denote the performance metric of interest as a function of the configuration, i.e., $y = G(X)$. Then, the configuration optimization problem can be defined as finding the parameter values, X^*, such that:

$$X^* = \underset{X \in DOM}{\arg\min}\ G(X) \tag{1}$$

In our case, since we consider throughput in addition to the minimization of latency, we define $G(X)$ as a discontinuous objective function consisting of two underlying black-box functions, $T(X)$ and $L(X)$, representing throughput and latency, respectively, such that:

$$G(X) = \begin{cases} L(X) & T(X) \geq t \\ \infty & T(X) < t \end{cases} \tag{2}$$

where t is the throughput objective. Thus, a configuration needs to attain a certain minimum throughput objective to be considered a candidate solution. In practice, $T(X)$ and $L(X)$ are often unknown or do not have a closed form.

A possible approach to solving this optimization problem would be to resort to analytical modeling to predict the throughput and latency of a given stream-processing application. Although performance models are useful because they provide insight into the system's behavior, can be solved quickly, and allow for large parameter space explorations, such models are unfortunately time- and labor-intensive to obtain, typically rely on simplifying assumptions that hinder accuracy, and need to be recalibrated and revalidated as the conditions change.

Instead, we seek to automate parameter tuning to achieve specific performance goals with minimal human effort. Thus, we run the actual system in a simpler and more accurate way to determine its performance.

2.3 Our Approach

We use actual experiments that run the real system in a profiling environment using a small number of machines to evaluate the throughput and latency functions and taking measurements of

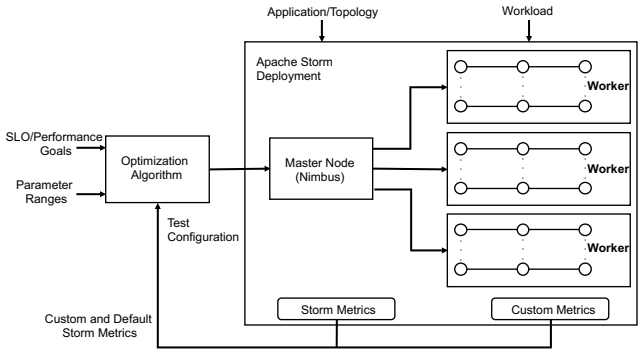


Figure 1: Configuration optimization framework.

these metrics. Such an experimental approach has been shown to produce system configurations that are as good as those resulting from idealized, perfectly accurate models [24, 52, 55].

Figure 1 presents an overview of our optimization framework. The user provides the application, a sample workload, SLO/performance goals and the possible ranges of parameters. The initial configuration is generated by the optimization algorithm and is used to submit the application/topology to the Storm cluster. After a specific duration, the application is terminated and the metrics exposed via Storm API (i.e., throughput, utilization, processing latency) as well as custom metrics (tail latency in our case) are collected to evaluate the current configuration. Subsequently, a new configuration is generated according to the optimization algorithm and the same application is submitted with this new configuration. This process continues until the optimization algorithm terminates the search based on some allocated resource budget (e.g., the amount of time we allow for configuration optimization) and provides the user with the best configuration according to the performance goals.

Our framework supports multiple optimization algorithms (§4). To demonstrate this generality, we later implement and evaluate several algorithms.

2.4 Background on Latin Hypercube Sampling

Exploring the complete space of all configurations is intractable when the number of parameters is large. We rely on an experimental design approach to obtain an initial sample from the configuration space using the Latin Hypercube Sampling (LHS) method. Here, we provide a short description of this method and refer the reader to McKay et al. [38] for a detailed introduction to LHS.

LHS is a stratified sampling formulation that is used to generate a near-random sample of parameter values from a multidimensional distribution. Since it is a stratified sampling approach, LHS can provide better coverage of the sample space than random sampling can. The LHS algorithm for generating K random configurations of dimension N (the number of parameters) can be briefly described as follows:

(1) Generate N random permutations, $P^1, \ldots, P^N$, of set $S = 1, \ldots, K$, where $P^i = (P^i_1, \ldots, P^i_K)$.
(2) For the i-th dimension with $i = 1, \ldots, N$, divide the parameter range, R_i, into K non-overlapping intervals of equal probabilities.
(3) The k-th sampled point is an N-dimensional vector, with the value for dimension i uniformly drawn from the P^i_k-th interval of R_i

Unfortunately, LHS by itself does not rule out bad spreads. For example, all samples can be spread along the diagonal. This problem can be addressed by generating multiple sets of LHS samples and choosing the one that maximizes the minimum distance between any pair of samples. That is, suppose l different sets of LHS samples, $L1, \ldots, Ll$, were generated. Then, the approach selects the set L^* such that:

$$L^* = \arg\max_{1 \le i \le l} \min_{X^{(j)}, X^{(k)} \in L_i, j \neq k} \text{dist}(X^{(j)}, X^{(k)}), \quad (3)$$

where dist is a distance metric like Euclidean distance. This method is called Maximin Latin Hypercube Sampling [39].

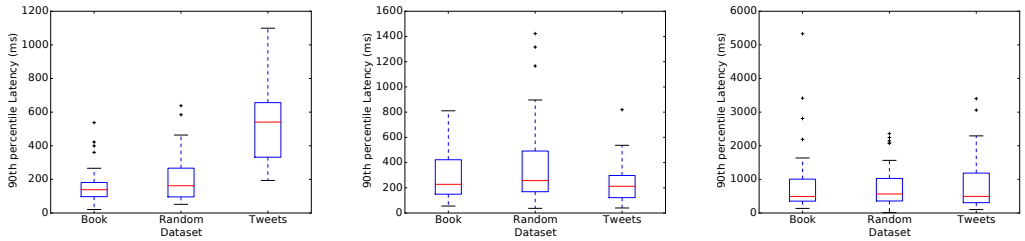
However, we cannot directly use LHS for our case because different parameters have different numbers of values. For example, the number of workers might only four discrete values while the number of bolt executors might have 15 different values. If we were to use LHS, then we would not generate more than four samples because a specific value of a parameter cannot appear more than once in an LHS design. Thus, we use a modified version of LHS that we call mLHS. mLHS generates samples using LHS in a [0-1] N-dimensional design space. These samples are then mapped to the discrete bounded ranges in the N-dimensional space, which means that the end design is not a Latin Hypercube since an input parameter can have the same discrete value in two or more configurations, depending on the number of samples generated.

3 IMPACT OF PARAMETER TUNING

Actual applications of stream processing on the industrial scale typically adhere to strict SLOs . The most common performance metrics used to specify SLOs for stream processing consist of (i) latency: how much time the system should take to process each input so that outputs reflect the latest inputs, and (ii) throughput: how much data the system should process within a time interval. An SLO specifies a desired level for throughput and/or latency that the stream-processing system must attain so that data will not be queued and eventually dropped or so that data will not cause cascading errors.

Performance tuning is therefore a crucial task that occurs when an application is deployed or when the application or workload changes. Changing the configuration of the parameters is one of the main ways to address performance tuning and to ensure that an SLO is met. Although improper tuning of parameters can have dire consequences on system performance, making sure that an application is running as efficiently as possible is a daunting task given the very large space of possible configurations. To understand the variation in performance achievable through parameter tuning, we conducted experiments on a multi-node Apache Storm cluster.

The setup included three worker nodes and a master node, each with a dual CPU with eight cores with enabled hyper-threading.



(a) Rolling Count Topology (b) Rolling Top Words Topology (c) Sentiment Topology

Figure 2: Variation of the 90^{th}-ile latency for different configurations and workloads in three benchmark topologies.

Parameter	Range	Step Size
Number of workers	3–12	3
Number of Acker executors	3–12	3
Number of Spout executors	3–12	3
Number of Bolt executors	3–45	3
Max Spout pending	2000–10000	2000

Table 1: Selected parameters and their value ranges.

This meant that a total of 32 threads could simultaneously run on each node. We configured Storm to use the Netty transport layer, as it achieves better performance than the default transport layer, ZeroMQ [9]. We left Acking, which is the process by which bolts inform Storm when they have finished processing an input tuple, enabled to access performance metrics from Storm UI for our measurements. We used three topologies as benchmarks [7, 13, 14]: (*i*) rolling word count (RC), (*ii*) rolling top words (RTW) and (*iii*) sentiment analysis (SA). Details about these topologies are presented in Sec. 5.1 below.

Table 1 shows the parameters that we selected to tune and the ranges and increment step sizes considered in our experiments, unless otherwise noted. We empirically chose this subset of parameters from all possible parameters of Apache Storm based on their importance according to documented parameter-tuning best practices [4, 6, 10, 12, 18]. The number of workers indicates the number of worker Java VMs that Storm generates for the topology. "Number of Spout executors" and "Acker executors" defines the number of threads that generates tuples for the topology to process and the number of threads responsible for handling tuple acknowledgments, respectively. Similarly, users can also specify the "Number of Bolt executors" for each bolt in the topology. Lastly, "Max Spout pending" specifies the maximum number of tuples (per spout) that can be simultaneously on-the-fly in the topology, i.e., the max number of tuples per spout that have not been acknowledged yet.

The step size used for all parameters except "Max Spout pending" is a direct result of the number of worker nodes that we use. We selected the step size to be equal to the number of nodes to avoid any kind of imbalance in the DAG placement, which is chosen by the default Storm scheduler. This strategy might be unnecessary if a different scheduling algorithm is used.

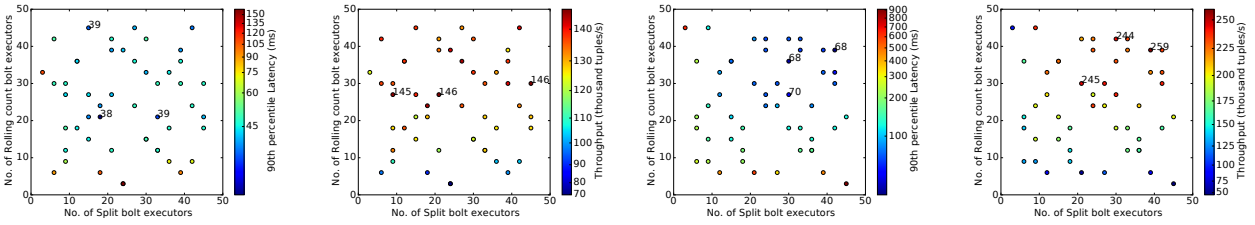
Note that the "Number of Bolt executors" parameter is a per-bolt parameter. Thus, the number of parameters to be tuned depends on the topology size, which is between 2 to 5 bolts in our topologies. The total number of parameters that we use for tuning purposes therefore ranges from 6 to 9 parameters.

Parameters significantly affect latency. We used mLHS to generate 30 different configurations for each topology. The purpose of this sample set was to illustrate the variation in performance metrics that can occur due to different configurations. For each configuration, the topologies were executed for 200 s while samples from the 90^{th} percentile of latencies and throughput were recorded for the last 100 s of the run to exclude noisy measurements, which are typical in the initial phase during execution. We execute each configuration with three different data streams based on a text corpus from a book, a random process and a dataset of real Twitter tweets.

Figure 2 shows a boxplot of the variation in the 90^{th} percentile of latency for the 30 configurations tested for different workloads. We observe that latency depends on all three factors, configuration, workload and application. Importantly, the configuration of parameters has a significant impact on latency, influencing the measured 90^{th}percentile of latency by even two orders of magnitude in some cases (e.g., the sentiment topology with the book dataset).

Parameters have complex second-order (or higher) effects. The relationship between different configuration parameters can be complex and can include second-order or higher effects. Thus, simply increasing the value of one parameter does not correspond to a monotonic increase or decrease in the performance metric. To illustrate the second-order effects, we generate 50 configurations of the number of executors using mLHS.

The results for the RC topology are presented in Figure 3. There are two separate experiments shown there. Each experiment varies the number of executors for split bolt (x-axis) and rolling count bolt (y-axis) while keeping the remaining parameters fixed. Figure 3a and 3b show the different values of latency and throughput obtained when varying the number of executors while keeping the number of spouts, workers and ackers set at 12 (Setting 1). Figure 3c and 3d show similar results with the fixed parameters set at 6 instead (Setting 2). Each figure also highlights the top three candidate configurations according to its respective metric. While using more executors for the bolts can be a good strategy when the throughput objective is high, it does not necessarily yield the best configuration. These figures show that often comparable performance is achievable by multiple different configurations in the configuration space. For example, in Figure 3a, the best configurations in terms of the 90^{th} percentile of latency are with $x = 42$ and $y = 39$ as well as



(a) Latency with Setting 1 (b) Throughput with Setting 1 (c) Latency with Setting 2 (d) Throughput with Setting 2

Figure 3: Latency and throughput for different values of executors for Rolling Count and Split bolt for the RC topology.

$x = 33$ and $y = 36$. In Figure 3c, the best 90^{th}-ile latency achieved is when $x = 18$, $y = 21$; or $x = 15$, $y = 45$; or $x = 33$, $y = 21$. In the case of Setting 2, since the throughput is limited by the number of spout executors and the number of workers, increasing the number of executors to either bolt beyond 18 does not significantly impact the 90^{th} percentile in latency. Depending on the throughput objective, the suitable configuration for minimizing latency varies. The value selected for one parameter dictates the value that needs to be selected for another parameter to achieve the desired performance.

Summary. Based on a sample of 30 configurations, we observed 8 to 25 times variation in 90th percentile of latency (excluding the outliers). The same configurations provide different performance metrics for different workloads. In addition, we found that there is an inter-dependence between parameters. The desired throughput objectives and values set for other parameters dictates the best achievable latency and the values that need to be set in terms of the number of executors to achieve that latency.

4 OPTIMIZATION ALGORITHMS

In this section, we describe the algorithms that we have used to automatically tune the parameters for Apache Storm. These algorithms constitute the "Optimization Algorithm" block of our framework shown in Figure 1. We implemented the optimization algorithms listed in Table 2. Two of these five algorithms are introduced in this work. The remaining algorithms were previously documented in the literature to achieve good results, though they were not previously considered for parameter tuning in the stream-processing context.

Algorithm	Abbr.	Introduced in	Target setting
Hill-climbing algorithm	HC	[52]	Application server
Modified hill-climbing algorithm	mHC	This work	Stream processing
Rule-based approach	RB	This work	Stream processing
Genetic algorithm	GA	[35]	Batch processing
Random recursive search algorithm	RRS	[54]	Network protocols

Table 2: Algorithms used in our framework for parameter tuning.

The random recursive search and genetic algorithms did not achieve as good results as the other three approaches in most of our experiments that adhered to our optimization duration budget (which we set at 50 experiments or equivalently 250 minutes). We therefore do not discuss them in this paper due to space limitations.

4.1 Hill-Climbing Algorithm (HC)

Our implementation is based on a hill-climbing algorithm previously introduced in the literature [52]. However, unlike the previous approach, our version of the algorithm does not use weighted LHS since this assumes that a priori information is available regarding the correlation between parameters, whereas this might not be the case for us in general. The rest of the algorithm is implemented as it is described by Xi et al. [52].

The hill-climbing algorithm used in this work consists of the following phases:

(1) Initial Sampling: In this phase, n initial samples are generated using mLHS from the design space and the application is executed using these configurations.
(2) Local Search: In this phase, the algorithm uses mLHS sampling to generate m samples from the neighborhood of the best configuration found in the Initial sampling phase. It also checks whether a configuration in these m samples is better than the best configuration previously found.
(3) Potential Best Configurations: After the local search, the algorithm uses a polynomial fitting on the parameter values to create a configuration that might provide better performance. If a better configuration is found, it goes back to the local search phase. If not, it proceeds with including this new result and performing polynomial fitting again. If a better configuration is still not found, it proceeds to the shrink phase; otherwise, it goes back to local search.
(4) Shrink: The shrink phase decreases the size of the neighborhood and restarts the local search phase. If the neighborhood size is below a threshold, then the algorithm enters a restart phase.
(5) Restart: The restart phase generates l samples from the design space using mLHS and checks if the best configuration found in these l samples is better than the k-th percent of the configurations tested until now; if so, then it enters the local search phase again. Otherwise, it initiates the restart phase again.

The hill-climbing algorithm terminates after a fixed optimization budget is reached. Note that we avoid repeating experiments of configurations already explored during the local search phase. Thus, unique samples from the neighborhood are tested every time. If no new samples are generated by mLHS, then we enter the restart phase.

In all our tests, we set $n = 12$, $m = 5$ and $l = 5$ to achieve a balance between local search and global search. The neighborhood threshold value, t, is set to 0.4 times the size of the design space and k is set to 80%. The values for these settings are chosen according to the recommendations provided in [54] and through empirical testing to obtain the best performance in our settings; however, we did not perform an exhaustive analysis and these settings might need to be reconfigured according to need.

4.2 Modified Hill-Climbing Algorithm (mHC)

The difference between the hill-climbing algorithm (HC) and the modified hill-climbing algorithm (mHC) is the way in which the initial sample set is generated. In HC, we use mLHS to generate the initial sample set. In the case of mHC, we resort to a custom heuristic. In particular, the values of the number of workers and "Max Spout pending" are generated through mLHS, while the numbers of spout and acker executors are set equal to the number of workers. The number of executors for different bolts are generated randomly using a Dirichlet distribution to distribute the remaining number of cores among different bolts, according to Equation 4. The Dirichlet distribution is a family of continuous multivariate probability distributions parameterized by a vector, α, of positive real numbers. It can be used to generate random numbers such that their sum is equal to 1.

$$t_{bolts} = t_{cores} - t_{spout} - t_{ackers} \tag{4}$$

where t_{cores} is the total number of cores in the cluster, t_{spout} and t_{ackers} is the number of spout and acker executors, respectively, and t_{bolts} is the number of executors that will be divided equally among the bolts in a topology.

4.3 The Rule-Based Approach (RB)

The rule-based approach (RB) is a gray-box heuristic approach. It takes hints from the user who provides a ranking of parameters according to a priority level and specifies for each parameter whether an increase in parameter value has an overall positive or negative impact on latency and throughput. This is a greedy approach. It favors finding a suitable configuration quickly at the expense of optimality. Also, it attempts to minimize the number of resources (executors) that are required to achieve the required throughput objective.

Algorithm 1 shows the rule-based approach. The algorithm has three distinct phases: (*i*) throughput objective satisfaction phase, (*ii*) latency minimization phase, and (*iii*) executor adjustment phase.

(*i*) The throughput objective satisfaction phase changes the value of the parameters to achieve the throughput objective specified by the user. To make this phase aggressive, we update all the parameters at the same time according to their expected effect on the throughput (i.e., if increasing the parameter values increases the throughput, then the parameter values are increased or vice versa).

(*ii*) Once the throughput objective is satisfied, RB enters the latency minimization phase during which it does one-at-a-time tuning of parameters to minimize the latency. This phase iteratively goes through each parameter according to its user-provided ranking and tunes the value of the parameter to achieve a better performance than previous performances. If change in a parameter's value does not improve performance, the change is reverted and the algorithm moves to the next parameter. Once this iterative phase has taken all the parameters into account, it selects the best configuration it has found and enters the executor adjustment phase.

(*iii*) The executor adjustment phase involves two sub-phases. The first sub-phase reduces the number of executors assigned to each of the bolts as long as their capacity reported by Storm metrics is less than 80%. The second sub-phase reassigns the executors from other bolts to the bolt that is running at the highest capacity. Both of the phases are executed repeatedly for as long as the throughput objective is met. This last phase might also lead us to a configuration that can satisfy the throughput objective but with higher latency, while using a smaller number of executors.

The basic parameter ranking that we use for all topologies ranks bolt executors at the top (based on their utilization; a higher utilization gets higher priority) followed by executors for spouts, ackers, max spout pending and number of workers.

Discussion. We note that all algorithms described in this section are not specific to Apache Storm and could be used with other stream-processing systems. For instance, both mHC and RB algorithms mainly provide heuristics for selecting the parallelism levels for the number of executors for bolt stages. These heuristics are directly applicable to systems like Apache Flink and Twitter Heron, which also have parallelism-level configuration parameters.

```
Data: Ranked parameters, Parameter ranges, application and tputObjective
Result: Best configuration
Divide executors equally among all bolts;
Set values for all other configurations to their minimum;
while throughput objective unmet do
    change values of all parameters simultaneously to increase throughput;
    if throughput objective met then
        bestConfig = currentConfig;
        bestLatency = currentLatency;
        break;
    else
        continue;
    end
end
foreach parameter by ranked priority do
    while true do
        change value of the parameter by one step;
        get throughput and latency ;
        if throughput ≥ tputObjective and latency <bestLatency then
            bestConfig = currentConfig;
            bestLatency = latency;
        else if throughput < tputObjective or latency ≥ bestLatency then
            break ;
    end
end
Decrease the number of executors per bolt depending on the capacity;
if latency ≤ BestLatency then
    bestLatency = latency;
    bestConfig = currentConfig;
while throughput ≥ tputObjective do
    Reallocate executors from bolts with low capacity to bolt with max
    capacity;
    if latency ≤ BestLatency then
        bestLatency = latency;
        bestConfig = currentConfig;
end
```

Algorithm 1: Rule-based approach.

5 EVALUATION

We now demonstrate that automated parameter tuning is both practical and beneficial for stream-processing systems by conducting an evaluation of the algorithms on a range of topologies and workloads to answer the following questions.

- How well do the considered optimization algorithms perform?
- What impact does the ranking of parameters have on the performance of the rule-based approach?
- Is it possible to scale via extrapolation a small cluster's best configuration to a larger cluster?
- Given a resource budget, are the benefits of using automated parameter tuning greater than employing such resources to scale the cluster?

Results Highlights. Based on our evaluation described below, we observe that:

• Our results demonstrate the feasibility and benefits of automated parameter tuning across multiple benchmark Storm applications and workloads, suggesting that our framework is a viable solution to the problem.

• Overall, mHC and RB perform comparably while outperforming HC in most cases. However, HC performs better when we are dealing with low throughput objective cases because of the uniformly sampled design space.

• RB is two to five times faster than the hill-climbing algorithms in terms of convergence time, thanks to the initial sample and throughput objective satisfaction phases.

5.1 Applications

We selected three topologies from publicly available Storm benchmarks [7, 13, 14]. The topologies are: Rolling count (RC), Rolling top words (RTW) and Sentiment analysis (SA). These topologies consist of three, five and six stages, respectively. These topologies can be found at [11] In our experiments, we use HDFS as the source of input data to the topologies. The data is streamed from HDFS to bolts using Storm spouts. To provide the impression of infinite data stream, since the data in HDFS is limited, the spouts simply restart from the beginning of the data file once they reach the end of the file.

5.2 Experimental Setup

Our main setup consists of a four-node testbed, with each node having 16 physical cores and 32 virtual cores. Three of the nodes are used as workers for Apache Storm and one node is used as the master.

Latency measurements are done on a per-tuple basis. Latency is measured based on the time it takes for the tuple to progress from a spout to the last bolt. Spouts tag each tuple with a timestamp and the total latency is calculated by subtracting the time at the last bolt from the timestamp tagged by the spout. To make sure that the timestamps are consistent across multiple nodes, we use Network Time Protocol (NTP) to synchronize the time.

The Storm Metrics API provides only average latency numbers, which are not sufficient for our case. Measuring per-tuple latencies is essential to measure tail latencies. Since throughput levels can reach hundreds of thousands of tuples per second, we cannot use a conventional method to measure tail latencies based on per-tuple latencies. To handle this challenge, we use a probabilistic data structure called T-Digest [15]. We have implemented a client-server application based on T-Digest that is able to handle several millions of tuple latencies/s [16].

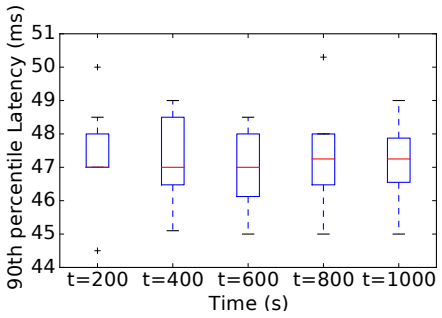


Figure 4: Latency results for different duration of experiments

Since we have a stream processing system with long-running applications, we needed to limit the duration of a single experiment. We tested different time durations for our experiments and selected 200 s as the duration of a single run. This time limit provides a good trade-off between the stability of the results and the time it takes for a single experiment to run. The results are stable because the tail latency does not change significantly if the experiments are executed for longer durations, as shown in Figure 4.

The latency measurements were collected for the specified time interval, in each experiment, for a total of 10 runs. We observed that the median 90^{th} and 99^{th} percentiles of latency across all runs differed by less than 2%. Therefore, we executed each experiment for 200 s. To allow for the setup phase to finish, we measured the latency only after the first 100 s. All subsequent latency readings were of the last 100 s of each experiment.

5.3 Performance of Parameter Tuning

We evaluated each optimization algorithm based on two criteria: (*i*) *Convergence*: how quickly the algorithm is able to search through the configuration space and find the best configuration, and (*ii*) *Best latency achieved*: what performance is attained by the best configuration that the algorithm finds within a certain amount of time.

In our evaluation, we focused on the minimization of latency (particularly at its tail) while sustaining a certain average throughput objective as the primary goal of the parameter tuning. This is a typical scenario envisioned for parameter tuning.

We present results obtained after five repetitions of each algorithm for each scenario. Due to space limitations, we present only a representative subset of the results.

The convergence graphs (Figures 5a, 5c, 6a, 7a) show the progress of the algorithms in terms of the best latency found at each point during execution of the experiments. The y-axis reports the median best latency found at each point across five repetitions of the algorithm. Note that the convergence graph for RB shows the runtime with the longest execution.

The best latency graphs (Figures 5b, 5d, 6b, 7b) show a boxplot of the best latencies found by each algorithm across multiple repetitions.

Overall, mHC performed the best in our experiments although RB outperformed it in some cases.

Figure 5b shows the best 90^{th} percentile of latency achieved by the three algorithms on the RC topology with a throughput objective of 150k tuples/s. We observe that RB has the least variance across multiple runs but the best configuration it finds can have higher latency as compared to the other two algorithms. This is because at a relatively low throughput objective, at 150k tuples/s, the configurations that provide the best latency are the ones with a low number of executors. Since RB starts from a configuration that utilizes all resources to begin with, it never arrives at these configurations. The mHC algorithm can suffer from a similar issue. While RB is able to find a good configuration within 10 iterations, it might miss some of the configurations that could improve the performance even further. On the other hand, HC generates a more uniformly distributed initial sample set, which leads it to configurations that provide on average 30% and 20% better latency as compared to RB and mHC, respectively.

Figure 5d illustrates the best 90^{th} percentile of latency when we increase the throughput objective to 300k tuples/s. The importance of the initial sampling technique for hill-climbing algorithms becomes clear at this throughput objective. HC uses LHS to sample the configuration space in a stratified manner, which means that it includes plenty of samples from the configuration space that are not suitable in high-throughput objective scenarios. On the other hand, mHC uses a sampling strategy that includes only configurations that have the total number of executors equal to the total number of cores, which enables it to find configurations that are easily able to achieve the throughput objective. In this case, on average only 2 out of 12 initial samples for HC were able to achieve 300k tuples/s throughput as compared to 9 out 12 initial samples for mHC. In terms of best latency achieved, RB and mHC perform comparably while HC provides 40% higher latency, on average.

The results for the RTW topology with the 150k tuples/s throughput objective are shown in Figure 6. HC and mHC perform comparably on average, while RB outperforms both. This is the case primarily because the initial configuration used by RB is very close to the configuration that provides the best latency. Thus, after a few iterations, RB can reach a configuration with the 90^{th} percentile of latency that is 25% lower compared with results from both HC and mHC.

For the sentiment analysis topology, several bolts have similar per-tuple execution latencies, which means that having a configuration where the number of executors is equally divided among multiple bolts is actually one of the best configurations. The Logging bolt is the bottleneck bolt in this case; however, within our cluster setup, we could not increase the parallelism level to that bolt enough to improve the performance further. The latency provided by mHC, on average, is only 5% and 10% lower than that provided by RB and HC, respectively.

While not shown in the figures, best practices suggested in various blog posts [4, 6, 10, 12, 18] and one-shot configurations were not able to satisfy the throughput objectives that we used in our tests. Their results are therefore not comparable. The best-practice approach that we adopted suggested that there should be one worker per node and the number of executors should be equal to the number of cores. We set the "Max Spout pending" configuration to 2000, with one spout and acker thread per worker. These approaches were able to achieve 70k tuples/s throughput for the sentiment analysis topology and 100k tuples/s for the RC and RTW topologies. Following the best practices did not therefore provide us with even the throughput objective that we wished to satisfy . The main reason for this behavior is the fact that our setup required more than one worker per node to achieve higher throughput.

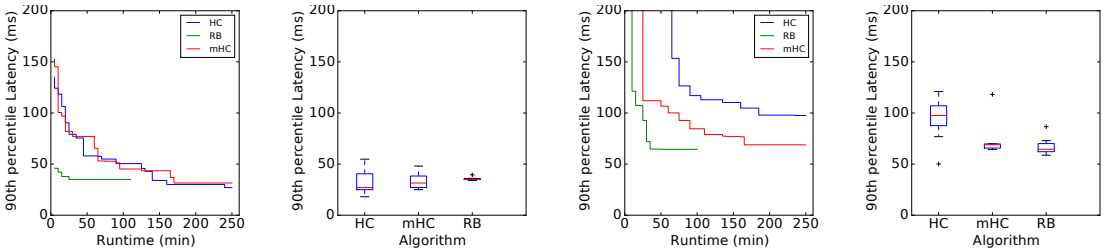
5.4 Impact of Parameter Ranking on RB

We now turn to an evaluation of the impact of parameter ranking on the performance of RB. Since RB does not backtrack on the configurations in most of its steps, the role of parameter ranking might be important to the achievable performance. To evaluate this aspect, we took four random parameter rankings for three test cases along with the ranking that we used for our previous experiments. Note that our previous ranking was based on our intuition and it might not always be the best ranking. Thus, we compare five parameter rankings to observe their influence on the best achievable performance. We execute RB three times using each ranking.

The results are shown in Figure 8. Surprisingly, we observe that parameter ranking does not significantly impact the best latency achieved. Even with different rankings, RB is able to achieve comparable performance. This observation can be explained based on the fact that since there are multiple configurations that are able to perform comparably, RB is able to arrive at one of those configurations from different starting points. More importantly, we observe that the throughput objective satisfaction phase in our test cases gets us very close to one of the best configurations in terms of latency. Because the changes in parameter ranking only affect the latency minimization phase, the affect of parameter ranking on best latency achieved is limited. While this may be the case for our tests, we believe that there will be other scenarios in which parameter ranking might be important and an approach to determine the ranking might be necessary, especially since RB does not backtrack to previous parameters.

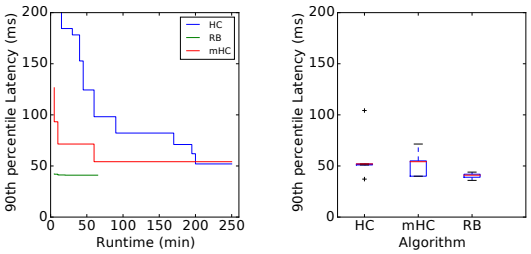
5.5 Comparison with Other Approaches

To the best of our knowledge, the state of the art in parameter optimization for stream-processing systems is the work by Jamshidi et al. [30]. Their work follows a similar approach to iTuned [24] and uses Gaussian processes to do Bayesian optimization. The methodology and techniques presented in their paper are *suitable when latency is the only metric under consideration.* We argue that only optimizing for latency can limit the applicability of such a technique because often a throughput requirement is present as well. We ran their system on our RC topology and observed that because the system only optimizes for latency, it misses configurations for which latency values with up to 30% higher throughput were achievable. In addition, such a method does not allow for any trade-off between throughput or latency. For example, there were cases in which up to 45% better throughput was achievable while sacrificing 1 ms



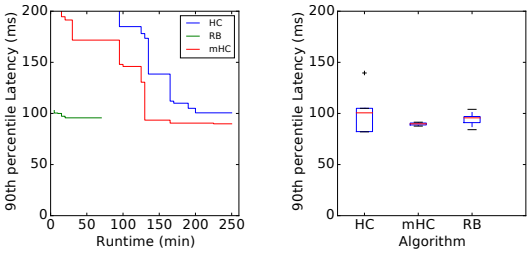
(a) Convergence (b) Best latency achieved (c) Convergence (d) Best latency achieved

Figure 5: Performance of the three algorithms on the RC topology with book workload and throughput objective of 150k (a,b) and 300k (c,d) tuple/s.



(a) Convergence (b) Best latency achieved

Figure 6: Performance of the three algorithms on RTW topology with book workload and 150k throughput/s objective.



(a) Convergence (b) Best latency achieved

Figure 7: Performance of the three algorithms on SA topology with book workload and 100k throughput/s objective.

of latency (increasing the 90th percentile of latency from 5 ms to 6 ms). Their techniques are clearly unable to navigate the trade-off between latency and throughput.

Bayesian optimization, as it has been discussed in [24, 30], is a single objective optimization technique. We have transformed our multi-objective optimization problem into a single objective one, using Equation 2. The function produced as a result has discontinuities, violating the smoothness assumptions of the generic covariance functions used in Gaussian processes [21]. Hence, Bayesian optimization with Gaussian processes used in these prior works is not applicable here. We leave as future work a study of multi-objective Bayesian optimization or Bayesian optimization with inequality constraints.

5.6 Extrapolation to Large Clusters

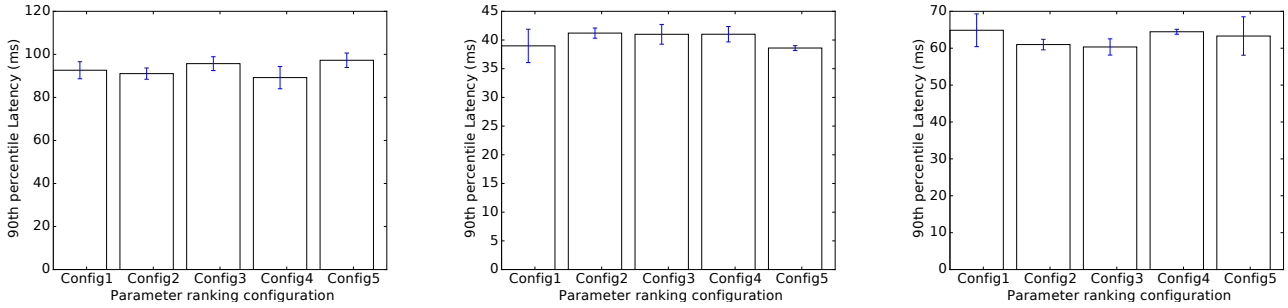
We now explore the following question: is it possible to use the results obtained on a small cluster of machines to tune the parameters of larger production clusters? This would be useful in several scenarios where it would be impractical or too costly to run the algorithms on the production cluster; however, parameter tuning over a few machines is possible if the best configuration found can inform the best configuration on the larger cluster. To evaluate the applicability of configurations found on smaller test clusters to larger production clusters, we use a simple extrapolation process.

We find that a combined throughput and latency objective is difficult to predict because the throughput does not scale linearly. However, we observe that in terms of best latency, the configurations found using a small cluster can be scaled to larger clusters while preserving the small cluster's latency optimization characteristic.

To confirm these observations, we ran the RB algorithm to find the configuration that provides the lowest 90^{th} percentile of latency in a four-node cluster (1 master and 3 worker nodes) and, afterwards, we ran the configuration with executors/number of workers scaled by five on a 16-node cluster (1 master and 15 workers). We compared the results of this scaled configuration with the best configuration provided by a run of all three algorithms on the 16-node cluster. We found that the scaled configuration provides 22 ms latency at the 90^{th} percentile while the best configurations provided by the runs of mHC, HC and RB algorithm were 27 ms, 33 ms and 21 ms, respectively. Hence, the scaled configuration is able to perform comparably to the best configurations found by running the optimization algorithm with the optimization budget of 50 runs. This intuitively makes sense because all the configurations except for "Max Spout pending" are scaled by five times. Thus, with five times greater load and five times more executors to handle that load, the configuration should perform just as well. The major condition here is that other resources, such as the network, do not become the bottleneck.

5.7 Cost-Benefit Analysis

To understand whether automated parameter tuning is viable in practice, we performed a cost-benefit analysis. We considered a cloud environment where machines could be used for configuration optimization or could simply be added to the cluster to scale out the stream-processing systems configured according to documented



(a) SA topology (100k tuple/s) (b) RTW topology (150k tuple/s) (c) RC topology (300k tuple/s)

Figure 8: Performance variation due to different parameter rankings for mentioned throughput limit.

Throughput (k tuples/s)	W/o opt.		With opt.		Break-even point
	Nodes	Cost(in $)	Nodes	Cost(in $)	
200	4	0.75/h	3	0.6 + 0.6/h	6h
250	5	0.90/h	3	0.6 + 0.6/h	3h
300	6	1.05/h	3	0.6 + 0.6/h	2h

Table 3: Cost-benefit analysis contrasting the cost of running parameter tuning versus the cost of scaling out the stream processing cluster. The benefits of automated parameter tuning always occur within a few hours.

best practices. We calculate the break-even point where the cost of running the cluster with configuration optimization is better than simply putting in more resources and using the best practices. For this experiment, we used three-worker nodes in AWS EC2 and a throughput objective of 200k, 250k and 300k tuples/s.

The total cost is the sum of the costs for the configuration optimization (fixed cost) and per hour based on the number of instances and their types. We used m4.4xlarge instances at a price of $0.15 per instance per hour. In our experiments, we observed that RB takes an hour to find a configuration that satisfies each of the throughput objectives. When using the best-practices approaches, we had to add more worker nodes to be able to achieve the desired throughput objective. Thus, the additional cost of parameter tuning is amortized over time.

Table 3 shows the break-even point, i.e., the duration of the application run after which the cost of using the automatic configuration is amortized. In the worst case, the break-even point is after six hours. If the application is supposed to be executed for periods longer than this point, then using automated parameter tuning provides cost savings.

6 RELATED WORK

Parameter Tuning. There exists a large body of work on automatic parameter tuning, primarily in the fields of application servers, batch-processing systems and databases. We briefly discuss works closely related to ours.

iTuned [24] is a system for online parameter tuning of databases. It uses Bayesian optimization with Gaussian processes to find the best values for a set of parameters. One limitation with Gaussian processes is that they require a large amount of training data to be able to accurately model a complex multi-dimensional black-box function. iTuned's goal is to optimize for completion time, which makes it a single objective Bayesian optimization problem. On the other hand, our goals are to improve latency while achieving a certain average throughput objective, making our problem a multi-objective Bayesian optimization. Moreover, as the workload changes, the model will need to be trained again. OtterTune [49] also uses Gaussian processes with workload characterization to tune DBMS configurations. Similarly, BO4CO [30] uses Gaussian processes for parameter tuning but in the context of stream-processing systems. As detailed in 5.5, their work focuses on a single performance metric and the techniques presented are not directly applicable to multi-objective optimization. Dhalion [25] proposes a policy-based self-regulation system for Twitter Heron. Users can define policies comprising a set of symptom, diagnosis and resolution actions. Symptoms provide information about observed anomalies in the behavior of the stream-processing system, which leads to the generation of a single diagnosis. As a result, a corresponding resolution action is carried out by the framework to bring the stream-processing system back to the normal state. Thus, policies are essentially a flexible form of rule-based automation. Their work is not limited to parameter tuning and can be used for parameter tuning. When the correct policies are written for each parameter and performance metric under consideration, their work can be used as an online tuning framework complementing the initial parameters selected using offline tuning by our approach.

For optimizing batch processing jobs, MROnline [34] uses a hill-climbing algorithm along with custom rules to tune certain MapReduce parameters to optimize the average execution time of MapReduce jobs. Xi et al. [52] also used a hill-climbing algorithm to find appropriate configurations for application servers. However, as we show in our evaluation, LHS is not necessarily the best sampling strategy for finding the best configuration in stream-processing systems. Tao et al. [54] and Heinze et al. [27] used recursive random search algorithm for parameter tuning in network protocols and an elastic scaling algorithm, respectively. However, in our experience, recursive random search is out-performed by the hill-climbing algorithm in the context of stream-processing systems.

Gunther [35] is a search method based on genetic algorithms to optimize parameters for MapReduce. However, their primary

goal is to decrease the execution time of the job. They focus on optimizing only for one metric.

Venkataraman et al. [51] train an analytical model for Apache Spark using experimental design techniques to create an efficient performance predictor to select the number of instances and their types for running jobs in Amazon EC2. CherryPick [17], on the other hand, uses Bayesian optimization with Gaussian priors to find best VM types and the number of VMs to minimze the deployment cost given a maximum tolerable time. Our approach instead considers the tuning of parameters assuming that the deployment scale and resources are known. Lin et el. [36] determined the degree of parallelism for each stage of the processing DAG based on input data and the computational cost. Thus, they were able to adjust the parallelism level to accommodate required throughput but they did not consider latency requirements. Zhu et al. [57] proposed online tuning of parameters for perception applications by learning a cost-based model using a parameter dependency graph. Cost-based models have also been used in parameter tuning for database systems as in [33]. However, cost-based models can have difficulty in accurately predicting performance across different types of hardware and application versions. For example, changes to the architecture of a database might render its cost-based model inapplicable.

Several works have proposed sophisticated approaches for designing experiments to run when benchmarking servers [46] or optimizing their configuration parameters [48, 56], though they do not directly apply to the stream-processing context. In addition to search-based algorithms, there have been several efforts to use machine learning for the purpose of automatic parameter tuning [26, 47]. However, approaches using machine learning generally require a large amount of training data to be able to build a classifier with good accuracy because they can only learn about scenarios and configurations that have been seen in the past.

Online Adaptation in Stream Processing. Several works proposed adaptive scaling and load balancing of stream-processing systems [22, 40, 41, 45, 53]. Lohrmann et al. [37] adaptively adjusted the buffer sizes and performs task chaining according to the QoS constraints. Das et al. [23] used dynamic batch sizing for stream processing in Apache Spark to avoid queuing delays as the input data rate changes. Venkataraman et al. [50] dynamically adjusted the number of batches that were grouped together for scheduling. We view these works as complementary to ours because we first need to find an efficient configuration to have an efficient scaling strategy.

7 CONCLUSION

We have presented a framework to automate parameter tuning in stream-processing frameworks and we have applied it to Apache Storm as a case in point. The concepts and approaches presented in the paper are not specific to Storm and could be extended to other stream-processing systems. We have shown that our modified hill-climbing algorithm and rule-based approach outperform the basic hill-climbing algorithm in three of the four test scenarios. The rule-based approach might provide up to 40% higher latency as compared to hill-climbing algorithms, in the worst case. However, RB can converge two to five times faster to that point as compared with the other two approaches and thus might be suitable for online parameter tuning.

Discussion A limitation of the rule-based approach is that it moves in a unidirectional fashion to change the parameters and improve performance. It does not backtrack to change the value of a previous parameter once a change leads to desired performance improvement. This means that there can be configurations that might lead to improved performance but that are never explored by the rule-based approach. This is a trade-off between the speed of the optimization algorithm and its effectiveness.

According to our initial experiments, the throughput of our Storm topologies does not scale linearly with the cluster size. Thus, we are unable to predict the throughput of the topology as the cluster is scaled out. In addition, the optimization algorithms need to be run again in case of a significant workload change as well as a change in the cluster hardware.

The framework introduced in this paper is for offline parameter tuning. Online parameter tuning requires the stream-processing system to be dynamically configurable without incurring long reconfiguration times. An online parameter-tuning system is also limited by the constraint that it should not use configurations that can possibly degrade the performance of the production system. Developing online parameter tuning to work alongside the offline framework is part of our future work.

Our framework assumes that the scheduling is static. Dynamic scheduling can change placements of different executors of Storm (e.g., [19, 43]), which can impact performance and might make the same configuration behave differently from one run to another.

Lastly, interesting challenges arise in a multi-tenant Storm cluster. In a shared environment, the parameters of a topology can be tuned automatically using the approaches discussed in this paper as long as enough resources are available to avoid overloading and other topologies running with a fixed configuration. On the other hand, if multiple topologies need to be tuned simultaneously, we see two possible avenues, which we leave for future work. One possible approach would be to optimize the topologies separately, while measuring and taking into account the interference from other topologies. A second approach would be to optimize the parameters of multiple topologies jointly so that each achieves its respective performance goal, in essence achieving Pareto optimality where each topology has its performance constraints met.

Acknowledgements. We thank the anonymous reviewers and our shepherd, Avrilia Floratou, for their useful feedback. We would like to thank Ivan Beschastnikh, Nabila Bounceur, Marco Chiesa, Noel Cody, Paolo Costa, Bobby Evans and Eno Thereska for helpful comments and discussions. Muhammad Bilal was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (EACEA) (FPA 2012-0030).

REFERENCES

[1] Apache Flink. http://flink.apache.org/.
[2] Apache Spark. http://spark.apache.org/.
[3] Apache Storm. http://storm.apache.org/.
[4] Apache storm performance tuners. https://www.ericsson.com/research-blog/data-knowledge/apache-storm-performance-tuners.
[5] Companies using Apache Storm. http://storm.apache.org/Powered-By.html.

[6] How spotify scales apache storm. https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm.

[7] Intel storm benchmark. https://github.com/intel-hadoop/storm-benchmark.

[8] Investment in Fast Data Processing Growing: Survey. http://www.enterpriseappstoday.com/data-management/investment-in-fast-data-processing-growing-survey.html.

[9] Making storm fly with netty. https://yahooeng.tumblr.com/post/64758709722/making-storm-fly-with-netty.

[10] Notes on tuning storm+trident. https://gist.github.com/mrflip/5958028.

[11] Parameter tuning framework repository. https://github.com/MBtech/stormbenchmark/.

[12] Performance tuning for hdinsight storm and microsoft azure eventhubs. https://blogs.msdn.microsoft.com/shanyu/2015/05/14/performance-tuning-for-hdinsight-storm-and-microsoft-azure-eventhubs.

[13] Rolling top words topology from bigdatabench. http://prof.ict.ac.cn/bdb_uploads/bdb_streaming/SocialNetwork-JStorm.tar.gz.

[14] Sentiment analysis storm. https://github.com/zdata-inc/StormSampleProject.

[15] T-digest. https://github.com/tdunning/t-digest.

[16] T-digest service. https://github.com/MBtech/TDigestService.

[17] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[18] S. T. Allen, M. Jankowski, and P. Pathirana. *Storm Applied: Strategies for Real-time Event Processing*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[19] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive Online Scheduling in Storm. In *DEBS*, 2013.

[20] S. Babu. Towards Automatic Optimization of MapReduce Programs. In *SoCC*, 2010.

[21] R. Calandra, J. Peters, C. E. Rasmussen, and M. P. Deisenroth. Manifold gaussian processes for regression. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 3338–3345. IEEE, 2016.

[22] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *SIGMOD*, 2003.

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

[24] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. *VLDB Endowment*, 2(1), 2009.

[25] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. In *VLDB*, 2017.

[26] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *HotPar*, 2009.

[27] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 276–287. ACM, 2015.

[28] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *VLDB Endowment*, 4(11), 2011.

[29] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.

[30] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 39–48. IEEE, 2016.

[31] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *HotCloud*, 2009.

[32] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, 2015.

[33] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for IBM DB2 universal database. *IBM Perf. Technical Report*, 2002.

[34] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. MRONLINE: MapReduce Online Performance Tuning. In *HPDC*, 2014.

[35] G. Liao, K. Datta, and T. L. Willke. Gunther: Search-Based Auto-Tuning of MapReduce. In *European Conference on Parallel Processing*, 2013.

[36] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. of NSDI*, pages 439–454, 2016.

[37] B. Lohrmann, D. Warneke, and O. Kao. Massively-parallel stream processing under qos constraints with nephele. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 271–282. ACM, 2012.

[38] M. McKay, R. Beckman, and W. Conover. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21(2), 1979.

[39] M. D. Morris and T. J. Mitchell. Exploratory designs for computational experiments. *Journal of statistical planning and inference*, 43(3):381–402, 1995.

[40] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 137–148. IEEE, 2015.

[41] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 589–600. IEEE, 2016.

[42] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *SC*, 2011.

[43] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[44] M. J. Sax, M. Castellanos, Q. Chen, and M. Hsu. Performance optimization for distributed intra-node-parallel streaming systems. In *ICDEW*, 2013.

[45] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[46] P. Shivam, V. Marupadi, J. Chase, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *USENIX ATC*, 2008.

[47] J. Singer, G. Kovoor, G. Brown, and M. Luján. Garbage collection auto-tuning for Java MapReduce on multi-cores. *ACM SIGPLAN Notices*, 46(11):109–118, 2011.

[48] R. Thonangi, V. Thummala, and S. Babu. Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2008.

[49] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[50] S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale, 2016. http://shivaram.org/drafts/drizzle.pdf.

[51] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378. USENIX Association, 2016.

[52] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, 2004.

[53] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *21st International Conference on Data Engineering (ICDE'05)*, pages 791–802. IEEE, 2005.

[54] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS Performance Evaluation Review*, 31(1), 2003.

[55] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *USENIX ATC*, 2009.

[56] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic Configuration of Internet Services. In *EuroSys*, 2007.

[57] Q. Zhu, B. Kveton, L. Mummert, and P. Pillai. Automatic tuning of interactive perception applications. *arXiv preprint arXiv:1203.3537*, 2012.