To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics

Ayat Fekry
Lucian Carata
Computer Laboratory
University of Cambridge, UK

Thomas Pasquier
Department of Computer Science,
University of Bristol, UK

Andrew Rice
Andy Hopper
Computer Laboratory
University of Cambridge, UK

ABSTRACT

This experimental study presents a number of issues that pose a challenge for practical configuration tuning and its deployment in data analytics frameworks. These issues include: 1) the assumption of a static workload or environment, ignoring the *dynamic* characteristics of the analytics environment (e.g., increase in input data size, changes in allocation of resources). 2) the amortization of tuning costs and how this influences what workloads can be tuned in practice in a cost-effective manner. 3) the need for a comprehensive incremental tuning solution for a *diverse* set of workloads. We adapt different ML techniques in order to obtain efficient incremental tuning in our problem domain, and propose Tuneful, a configuration tuning framework. We show how it is designed to overcome the above issues and illustrate its applicability by running a wide array of experiments in cloud environments provided by two different service providers.

1 INTRODUCTION

Optimizing the runtime configuration when executing data processing tasks in the cloud is crucial for meeting desired requirements (timeliness of results, accuracy, energy efficiency) at a minimal cost. Sometimes, this can be achieved by setting the execution time as the sole optimization goal. At other times, multiple objectives need to be taken into account simultaneously and trade-offs considered. For example, how much are you willing to spend for getting a result 10% faster? Can you sacrifice accuracy and get probabilistic answers faster than exact ones, while incurring similar cost?

While the position above is a usual way of framing optimization problems, we have found that the *amortization* of the optimization costs through the resulting savings is often overlooked. This is because tuning is usually performed using ML performance/-cost models [17, 25, 27]. Once trained, it is assumed the model can produce predictions about what configurations yield close-to-optimum executions for a given workload for a long period of time. This makes the implicit assumption that the workloads and their runtime environment during prediction never drift too far from conditions during training. However, a significant number of real-world situations do not fit this assumption. There are workloads for which the distribution of input data and its size change over time; workloads which might be executed for a limited period of time or with reduced frequency, in cluster configurations that may change dynamically. Little data has been publicly shared about those workload characteristics, so it is of interest to build intuition on how they may affect the decision to optimize configurations.

To better understand this, we perform experiments in two cloud environments, Amazon Web Services (AWS) [3] and Google Cloud (GCP) [16] – 7429 hours of runtime in total, and use the lessons learned to define a practical automated optimization framework.

We also explore how to identify scenarios where: 1) the cost of incremental optimization and execution when compared to executing the workload with a fixed configuration cannot be amortised; and 2) optimization is only practical when performed by the cloud provider based on historical data collected across tenants.

While all methods discussed are not limited to a particular system or cost function, we have targeted Spark as the data processing framework to configure, because it is both popular and poses significant challenges for state-of-art configuration tuners (huge configuration search space, a variety of ways to process data) [27].

The paper makes 3 fundamental contributions to the discussion around configuration tuning:

- Bringing into focus the trade-offs that need to be considered when performing tuning *without* assuming a static environment or workload, and proposing "rules of thumb" for evaluating the opportunity to optimize.
- Proposing a comprehensive framework for incremental tuning of configurations that reaches results comparable to existing state-of-art tuners but using significantly fewer executions. The source code of our implementation for Spark [2] is released under an Apache License v2.0 at [6] and can be transparently integrated into existing processing workflows.
- Showing how data-efficient machine learning techniques can be leveraged to widen the types of workloads that can be tuned in a cost-effective manner.

All collected data is published as a reference dataset usable for significant configuration parameter identification, efficient tuning and workload similarity analysis. The data contains five workload types, each executed with multiple data input sizes and hundreds of configurations across 2 different types of clusters.

1.1 Tuning Cost Amortization

Is workload-specific tuning necessary? It is reasonable to ask whether tuning can not be efficiently done using a single cost model to predict the best configurations for a wide range of data processing workloads. If that were the case, amortization could be done over the lifetime of a cluster rather than for individual workloads. The main difficulties posed for training such a model are: 1) hundreds of executions are needed to build it [27]; 2) difficulties in adapting to dynamic resource allocation in the cluster; 3) the high diversity of the workloads makes it harder to build a single cost model of a good accuracy [8]; 4) the high dimensionality of the search space: one dimension per configuration parameter. Complex data processing frameworks such as Spark commonly have 20 - 60 parameters

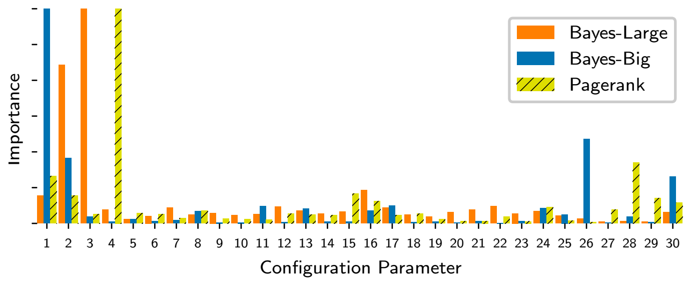

Figure 1: Relative configuration parameter importance for 3 Hibench workloads, showing that the parameters (Table 3) which affect runtime the most are workload-specific.

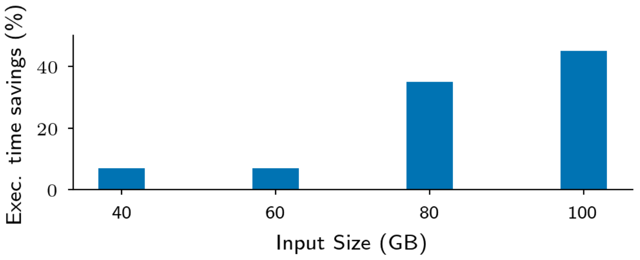

Figure 2: Execution time savings when retuning the configuration for each input size, compared against reusing the best configuration found for input size 20GB (TPCH benchmark)

that are relevant for tuning, and our experimental evaluation with system-wide models yields results around 40% worse than optimal.

While the first 3 issues can be directly solved by moving to workload-specific tuning, the typical way to solve the dimensionality problem is to employ dimensionality-reduction methods such as factor analysis, PCA, autoencoders, etc. The assumption is that almost all information can be retained in a lower-dimensionality space. This is certainly true for configurations when looking at individual workloads (Fig. 1). However, the problem is that different workloads are sensitive to different sets of parameters and respond to each in different ways, increasing the minimal number of dimensions that need to be considered. This makes it difficult to obtain models that generalize well, even after hundreds of training examples (workload executions).

We see a similar issue with generically-tuned configurations suggested by PaaS Spark services from Amazon and Google, which are tuned only taking into account available cluster resources. In the typical case, those yield execution times 43 - 68% longer than the optimum for a given workload. Where available, those configurations are a significantly better starting point than the default Spark configuration. However, it is difficult to bring them closer to the optimum while staying workload-agnostic. There is no configuration that is close-to-optimum for numerous workload types. Lacking workload-similarity measures, running a workload with the optimum configuration of another leads to unpredictable results. As an extreme example, the best configuration for Pagerank run on 10 million pages yields the worst runtime for input size 15 million.
Dynamic configuration tuning: The idea of workload and data-aware tuning is further strengthened if we consider a dynamic context, where both the input data and the cluster resources may change. This means that the conditions under which we ask for predictions drift away from the conditions during training, unless we spend resources for updating the model.

Thus, a changing environment suggests tuning that is not done as an offline stage but incrementally using real workload executions, aiming to provide a better configuration each time. This can also consider similarities between workloads to reduce exploration costs and provide faster amortization.
When does dynamic tuning pay off? An entire class of workloads can be optimized *without* considering the cost of tuning: those are workloads that will be recurring indefinitely (e.g. every day, week), without significant input size changes (e.g. stable size of data since last execution). It is also likely that those workloads will not require dynamic resource allocation, having stable computational, I/O, and network requirements (e.g. delta log processing). This is intuitive: a static, predictable workload can be tuned once and executed optimally thereafter. However, any departure from those characteristics implies a finite window of time to amortize optimization costs:

If the workload will only be executed x times, then the cost of exploring the search space for a good configuration needs to amortize well within those x runs; In the extreme case of a single execution, tuning can only be cost effective if "guessing" a good configuration by static analysis or matching to similar workloads.

If the input data size grows over time, previous optimal configurations lose their efficiency, as data gets re-partitioned, communication costs grow (i.e. increased data shuffling) or some processing gets serialized. At some point, more resources (cores, disks, VM instances) need to be allocated at the cluster level. Taken together, those will change the performance characteristics and invalidate previously learned models, requiring re-tuning (shown in Fig. 2). Here, the window to get benefits from optimization depends both on the workload execution frequency, the data growth rate and how quickly an existing performance model becomes obsolete.

1.2 A Simple Amortization Model

Building on those observations, we introduce a simple model for tuning cost amortization (Fig. 3). This compares the cumulative execution time of running with a fixed configuration to the one obtained through incremental tuning. In the search phase, the tuning algorithm will incur higher costs as it tries to find good configurations. Once a stopping criteria is met, the best found configuration is used for the following executions. The sufficient condition for tuning profitability is for the expenses to be recovered before the cost model built through tuning becomes obsolete. This might happen due to changes in input data size or in the environment where the workload executes. Once that happens, it is impossible to predict how either the original configuration or the tuned one will behave, and re-tuning needs to be triggered.

The amortization model we just described can be used to obtain a rough measure on whether breaking even is likely or possible at all. First, one must estimate after how many workload executions a tuned configuration will no longer provide the optimal execution.

This estimate is our a break-even deadline (longest possible time for the tuning cost recovery phase while not losing money), after d

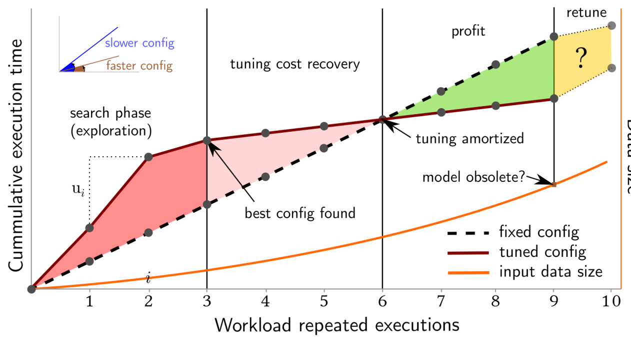

Figure 3: An idealized model for cost amortization

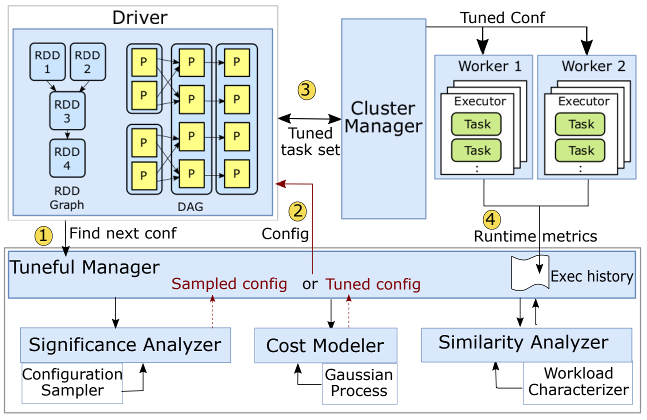

Figure 4: Tuneful–Spark integration: (1) On workload submission, the Driver requests the next configuration from Tuneful. (4) After workload execution, metrics are collected and used to update existing optimization models.

executions. In Fig. 3, $d = 9$. In practice, one sets d as a rough estimate for when workload performance characteristics will change (due to increase in input data size, cluster resource allocation, etc). Based on this, for execution i taking time u_i we can compute how much faster the next configuration needs to be such that all costs are recovered exactly at execution d. We determine this in (1) by constraining the next point, $(i+1, \sum_{j=1}^{i+1} u_j)$ to be on the line connecting the current tuning cumulative time and the time of running with the initial configuration d times (t_d).

$$u_{i+1} = \frac{t_d - \sum_{j=1}^{i} u_j}{d - i} \quad (1) \qquad \Delta_{\%} = \frac{u_i - u_{i+1}}{u_i} \cdot 100 \quad (2)$$

We can consider stopping the tuning if the best configuration found so far, $\min_{1 \le j \le i} u_j$ would amortize the cost before d.

When comparing two subsequent executions during tuning in our experiments, the second execution is faster 54% of the time, and on average reduces the execution cost by 23% (4% at the 25th percentile, 13% at median and 34% at the 75th percentile). Those can be considered as guidelines to see if the cost decrease required to break even, $\Delta_{\%}$ as computed in (2) after plugging in u_i and u_{i+1} can be achieved.

2 AN INCREMENTAL TUNING FRAMEWORK

The presented amortization model can of course be made more realistic, but has all the required components for a more nuanced discussion around tuning. Based on its characteristics, we explore ML techniques that are suitable for tuning configurations in practice. The resulting framework, Tuneful, aims to use information from each execution sample efficiently, in order to minimize the time spent in the search phase and allow the best chances for cost recovery and profit, even when considering evolving workloads.

2.1 System Overview

Tuneful is designed to efficiently tune the high-dimensional configurations of various data analytics workloads (e.g., graph analytics, machine learning, SQL, text analysis).

Unlike usual configuration tuning approaches, Tuneful is designed to avoid expensive offline phases for significant parameters identification or tuning, computing everything as part of incremental optimization. An illustration of the framework is shown in Fig. 4. It consists of three main components controlled by the Tuneful Manager: Significance Analyzer, Cost Modeler and Similarity Analyzer.

We have explored multiple solutions for each component in order to obtain the fastest amortization of costs, but every component is pluggable in the sense of allowing new techniques to be tested while keeping every other component fixed. At the beginning of the search phase, the Significance Analyzer uses executions to pick configurations that enable a quick exploration of *workload-specific* influential parameters. Once those are determined, the Cost Modeler can take over and build a lower-dimensionality model. However, our experiments show that those two steps still require around 35 workload executions to obtain good results. Once sufficient workload/environment changes and types of workloads have been seen, the Similarity Analyzer reduces the exploration costs further by reusing existing information when tuning new workloads.

When a workload is submitted for execution, as shown in Fig. 4, Tuneful suggests either an exploratory configuration (generated by the Significance Analyzer), a tuned one (generated by the Cost Modeler), or starts tuning from an existing model based on workload similarities (determined by the Similarity Analyzer). After execution, performance and cost metrics are fed back into Tuneful where they are used to update choices for the next configuration. Over time, the Similarity Analyzer is able to match more and more workloads, which significantly improves the tuning speed as we demonstrate in § 3.

2.2 Exploring Significant Parameters

All complex systems have numerous configuration parameters that can be set to user-given values. However, we have observed that for each workload only a small subset of those parameters has a significant impact on overall performance. Attempting to tune the non-influential parameters simply wastes resources and slows down the search for a near-optimal solution. This is especially true for Bayesian optimization (BO) strategies. In spite of their wide adoption as an efficient state-of-art solution for configuration tuning in various domains [8, 12], the problem of providing quick convergence in high dimensional configuration spaces (more than 10 parameters) remains [21]. A common approach for identifying parameter significance is to perform sensitivity analysis (SA) [10]. In our context, SA studies how the variation of the cost function (e.g., execution time) can be attributed to the different configuration parameters. This is usually done by running intensive offline benchmarks under numerous configuration values, analyzing the variance in performance with respect to each configuration parameter and ultimately building a system-wide set of influential parameters. This approach has been applied to tune systems such as DBMSs [20, 24].

Applying such techniques directly is not only expensive, requiring hundreds of executions, but also impractical given the diversity of Spark workloads. Instead, Tuneful *incrementally* identifies workload-specific influential parameters using a multi-round SA method that is efficient in our problem domain.

Fig. 5 shows the general strategy used, with each workload execution being used to advance through the steps of the algorithm. In each SA round, the aim is to prune some low-influence parameters in order to get better information about the highly influential ones in the next round. To achieve that efficiently, we first build a metamodel for predicting the execution cost of a given configuration. Our implementation uses Random Forest Regression (RFR), as it improves the prediction accuracy compared to single learning models [19]. Because we know most parameters will have little effect, this metamodel doesn't need to be extremely accurate; even a rough approximation will enable correct detection of important parameters. However, the metamodel needs to provide estimates in a wide area of the configuration search space. This is why Tuneful will first suggest configurations sampled using low-discrepancy sequences [22]. Experimentally, we have determined that the lowest number of executions that provides acceptable accuracy of the RFR model (less than 40%) is 10 (§ A.2).

We then calculate the importance of each input configuration parameter in the metamodel based on Gini importance [26]. This is a measure of each feature's contribution to the prediction of the execution time, considering the number of times a given feature is used in a tree split.

This allows us to select the most influential parameters and consider the remaining ones as non-influential. Those are fixed for the remaining SA rounds to the mean of their value range (int and float parameters) or their default values (boolean and enum parameters). Then the next SA round is started to determine the most influential parameters among the ones that can still vary. With the significant parameter detection approach described, we can still detect dependencies between parameters by including polynomial features, similar to [24], i.e. the method does not change, the metamodel just works on polynomial representation of the parameters- instead of individual parameters. For example, to detect if two parameters depend on each other, we can include a feature that represents the product of these parameters' values. If this feature has high importance, then those two dependent parameters will be detected.

2.3 Tuning of Significant Parameters

Unlike the SA rounds where a perfectly accurate model was not essential, in this stage we need more fidelity to ensure we are able to get as close to the optimal configuration as possible, using the minimum number of executions. Tuneful leverages Gaussian Processes (GP) due to their data efficient learning performance, which makes them ideal for modelling expensive functions such as the execution cost of the analytics workloads [8, 11]. GP enables Tuneful to model the execution cost under different configurations quickly, suggesting the samples that most likely contain the minimum point, and leaving the other costly samples unexplored.

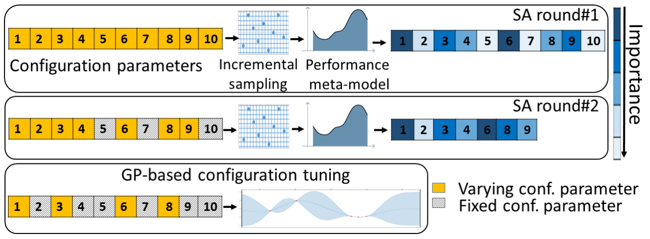

Figure 5: Tuneful's incremental algorithm for detecting the significant parameters and tuning them.

Furthermore, GP is non-parametric, which means that it does not need users to pre-commit to the shape of the function that models the cost. This flexibility allows GP to model the runtime of heterogeneous workloads and the influence of various configuration parameters on them. In practice, directly applying GP to the high-dimensional configuration parameter space of Spark is insufficient for obtaining satisfactory results, and therefore this stage is only called after the workload-specific significant parameters have been detected, as shown in Fig. 5. The GP incrementally builds the cost model considering only the significant parameters. After execution i, the model guides the choice for the point in the configuration space that has the best chances of minimizing the objective function at execution $i + 1$.

The GP modeling stops after suggesting a minimum of n samples (e.g., 10 samples) and then after the expected improvement (EI) drops below 10%. We made this decision to make sure that we balance between the exploration of the tuning space and exploitation of the best configuration found. Our empirical evaluation of this stopping criteria shows that it guarantees near-optimal tuning within 15 executions at maximum.

Tuneful Manager monitors the difference between the GP predicted execution cost and the actual execution cost of the workload over time. If a continuous degradation is captured, it is a sign that the current model has become obsolete and re-tuning is necessary. Once a performance degradation is detected, the Tuneful process restarts (leveraging previously gained knowledge to speed up optimization, as we explain in § 2.4).

2.4 Similarity Analysis

Tuneful adopts a similarity-aware tuning approach to further accommodate the need for efficient configuration tuning. The aim of the Similarity Analysis is to find a source – already tuned – workload and use knowledge acquired during its tuning for the current workload (the target). This knowledge is leveraged to effectively accelerate the tuning process for: 1) tuning a new similar workload, 2) retuning an already seen workload to accommodate workload changes. Characterizing workloads for similarity matching is an offline step that can be performed once a sufficient number of workloads have been executed by Tuneful. This characterization involves capturing numerous runtime metrics and expressing them as ratios comparable across workloads (e.g., GC time relative to total CPU time, shuffled data relative to input size etc.). Based on those offline executions, a nonlinear autoencoder is trained to predict a low-dimensionality fingerprint for each given workload. Data enabling this type of training is published in our dataset (§ A.1.3).

At runtime, we find the workload most similar to the current one based on the Manhattan distance between their fingerprints. The knowledge gained while tuning the source is then transferred to the target. This includes: 1) significant configuration parameters; 2) the source execution samples and their runtime costs.

Sharing the knowledge of influential parameters saves the cost of running SA rounds, and the previous experience in tuning them helps to guide the tuning process. This ultimately accelerates the convergence to a near-optimal configuration and enables a quicker amortization of the tuning costs.

We employ multitask GP (MTGP) [23] to share the tuning samples across similar workloads (with each workload modelled as a task in the GP). The full similarity-aware tuning algorithm can be found in Alg. 2. We can detect inaccuracies in workload matching through monitoring the gap between the predicted execution time by the MTGP model and the actual execution time. If a continuous degradation takes place, we re-trigger workload matching (if we haven't yet seen sufficient workloads, we perform standalone tuning as described in § 2.2 and § 2.3 instead).

3 EVALUATION

We'll now examine how the set of techniques described above work together to ensure tuning in a variety of scenarios typical for data analytics workloads. We consider the execution time of the tuned configurations, the search time required to get close to the optimum, and importantly the way in which tuning cost are amortized. To achieve this, we evaluate Tuneful in three stages: 1) zero-knowledge tuning, the evaluation of tuning a new workload in a new cluster. We assume that there is no previous knowledge about tuning this or other workloads, which would be a typical scenario if you have just adopted a tuning strategy. In this bootstrapping context, we only look at workloads that do not evolve; Therefore, it is possible to compare the behaviour of our strategies against existing state-of-art tuners and establish a performance baseline. 2) we then evaluate Tuneful's effectiveness as workloads evolve, but assuming the need of running a limited number of types of workloads – this would be the case for targeted use of cloud data processing. 3) finally, we asses the impact of having more extended tuning knowledge on Tuneful's performance and the speed of tuning cost amortization. The context might be one where the user executes a wide array of types of workloads, or that of a cloud provider offering a PaaS solution to its customers.

3.1 Experimental setup

Cluster and configuration specification: We use two different clusters: a GCP cluster of 20 nodes (over-provisioned for most of our workloads) and an AWS cluster of 4 nodes (used to test tuning in a more resource-constrained environment). The detailed specification of each cluster and the configuration parameters considered for tuning can be found in § A.1.2.

Applications: We chose 5 applications of different characteristics to evaluate the effectiveness of Tuneful. The applications are chosen from the well known big data benchmarks (Hibench [15] and TPC-H [1]): 1) Bayes: an application that builds a Bayesian classification model. 2) Pagerank (PR): a graph analytics workload that ranks the influence of graph vertices. 3) Wordcount (WC): a text analysis application that counts word occurrences. 4) TPC-H: a benchmark that runs 22 decision support SQL queries. We use SparkSQL to run these queries. 5) Terasort (TS): a numeric data sorter.

3.2 Significant Parameters Exploration

We evaluate the accuracy with which Tuneful's algorithm detects the significant configuration parameters in each SA round. The ground truth parameter importance for each workload is estimated running Recursive Feature Elimination (RFE) [14] on top of a performance model built with a large number of sample executions (100 per workload). We compare this with the output of our algorithm (identification from small number of executions). Across all workloads and the different clusters, Tuneful can correctly detect the significant configuration parameters within 20 executions (using 2 SA rounds with 10 executions each).

Overall, the proposed algorithm generalizes well across the two different clusters, adapting to differences in parameter importance caused by resource availability (cores, memory, I/O). We report the representative example of Pagerank executed on a 5 million pages input and deployed in both clusters. Tuneful does not only detect some entirely expected differences (i.e., CPU and the parallelism level being the most important parameters in the large, over-provisioned cluster; and memory the most influential parameter in the small cluster), but also less obvious differences (i.e., speculative execution and variable compression during broadcast on the large cluster due to the large number of workers and the resulting data shuffle; comparatively, on the small cluster the choice of serializer is more important). The detailed results of Tuneful's significant parameters detection can be found in the published data as well as in [13]. While this proves that dimensionality reduction can be achieved relatively quickly in our domain space, 20 executions with unknown cost at the beginning of the search phase already means that some workloads will only amortize their tuning cost after 60 repetitions (Fig. 8). This is why beyond tuning in zero-knowledge scenarios or explicit retuning from scratch, we will prefer to reuse the significant parameters already identified for similar workloads.

3.3 Tuning Effectiveness and Efficiency

We use three metrics to evaluate Tuneful: 1) Execution time: the execution time of the tuned configuration by Tuneful and the competing approaches. The target here is to obtain tuned configurations similar to what state-of-the-art tuners achieve. 2) Search Cost: the amount of time and actual cost (in $) required by each system to find good configurations. The target is to get close-to-optimal configurations (within 5-10% of the estimated best configuration) significantly faster than the state-of-the-art. 3) Amortization speed: the number of needed workload executions to amortize the tuning costs. The target is to amortize the tuning cost after a small number

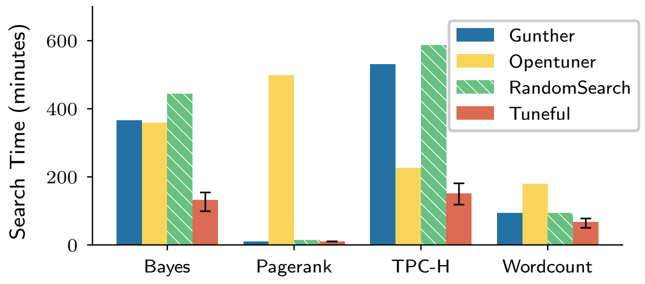

Figure 6: Search time of the different tuning algorithms to find configurations within 5% of the optimal (the lower the better)

of workload executions. The results are always presented as the median of 10 runs, with bars for the 10th and the 90th percentile.

3.3.1 Zero-knowledge Tuning: We first evaluate Tuneful without prior tuning knowledge (i.e., without leveraging the workload Similarity Analyzer) against 3 tuning approaches. The aim of this experiment is to assess the effectiveness of tuning without any earlier gained knowledge. This ultimately suggests when it is worthwhile to tune the configuration under different scenarios.

Workloads: 1) Bayes with a total executors input of 350GB. 2) PR with the Hibench-defined *huge* data size (5 million pages). 3) WC with a total executors input of 320GB. 4) TPC-H with a scale factor of 20. We chose this scale to limit the expenses of our experiments; however, we make sure that this scale is representative enough, with 300GB of total input to executors.

Baselines: we use two state-of-the-art techniques and random search as baselines in this experiment: 1) OpenTuner [9], a general tuning system that uses an ensembles of search techniques such as hill climbing, differential evolution, and pattern search. OpenTuner evaluates which techniques perform well over a window of time and picks them more frequently than the ones that have a poor performance (those can even get disabled). We selected OpenTuner as it covers a wide range of search algorithms. 2) Gunther [18], is a Hadoop configuration tuning system that leverages genetic algorithms to search for good configurations. To compare Gunther with Tuneful, we ported it to Spark following the details given in the paper. We set the population size to 60 and the number of generations to 20. We have made this port available online for future baseline evaluation [6]. 3) A configuration picked through *Random Search* (RS) using low-discrepancy sequences [22], since they cover the search space quicker and more evenly than the standard random numbers. Other surveyed work either uses search techniques already covered by OpenTuner, or the implementation details were too sparse to reproduce the approach.

We allow each state-of-the-art system a maximum budget of 100 executions for reaching a stable tuned configuration. Then we compare the configurations picked by Tuneful with the configuration tuned by the-state-of-the-art; we used the GCP 20 nodes cluster to run this experiment.

Tuneful finds configurations comparable to the state of-the-art tuning systems: Table 1 shows that at median, Tuneful is able to obtain effective configurations for all workloads. Tuneful maintains a comparable performance to the other non-incremental tuning systems. We are comparing against the default configuration as a baseline usable across workloads, so the actual values are not relevant in practice but the differences between them are (in a realistic setting the accelerations would be significantly lower when starting from reasonably hand-tuned configurations). In evaluating the execution time acceleration with the best configuration found by each tuner per workload, we don't penalize other algorithms if they are slow in finding good configurations. We therefore allow each of them to use 100 execution samples per workload (our fixed maximum budget). In comparison, we present the results of Tuneful using just 35 execution samples per workload (as it would normally be used, performing 2 SA rounds with 10 execution samples each, followed by 15 execution samples for tuning at maximum).

Application	Gunther	Opentuner	RandomSearch	Tuneful
Bayes	271.6X	270.5X	287.5X	287.7X
Pagerank	29.4X	28X	30.6X	30.6X
TPC-H	19X	19.7X	18.5X	19.9X
Wordcount	54X	50.7X	53.6X	52X

Table 1: Execution time acceleration (X times) w.r.t Spark default configuration (higher is better).

For evaluating differences in time taken to *find* configurations close-to-optimal, we allow each algorithm to execute workloads until it finds the first configuration resulting in a runtime within 5% of the one produced by the *estimated optimal configuration*. This is defined as the best configuration ever found across all our tests for each workload, irrespective of the tuning algorithm or experiment.

The median search time for other algorithms is 2.7X longer when compared to Tuneful: The search time in Tuneful is the sum of the *workload execution times* needed by the tuning algorithm to: (i) explore for significant configuration parameters (§ 2.2), (ii) tune those to their optimal values (§ 2.3). It is important to note that the reported search time does not only depend on the number of samples but also on the actual samples that are picked, as exploring a bad configuration leads to a slow execution of the workload (what we care about in our amortization model). The GP model suggests samples that most likely have the minimum execution time, leaving others unexplored. However, it still needs to explore the configuration space in order to build an accurate cost model (§ 3.3.2 will show how this exploration cost is minimized when more tuning knowledge is gained and the similarity-aware tuning comes into action).

We estimate the search cost based on GCP's [16] per-second pricing. The total cost for tuning the four workloads is $379, $354, $288 using Opentuner, Gunther and Random Search, respectively. In comparison, tuning the four workloads with Tuneful costs $94.

3.3.2 Tuning With Limited Knowledge: In order to mimic a big data analytics environment executing dynamic workloads (e.g. growing input sizes or receiving workloads similar to already seen ones), we assess the effectiveness of tuning a set of workloads with growing input data and evaluate how this influences the speed of tuning cost amortization. We consider a limited set of "auxiliary workloads" that represents the workloads already seen and tuned in a zero-knowledge context, then evaluate tuning a set of similar workloads with evolving input sizes using Tuneful's similarity-aware

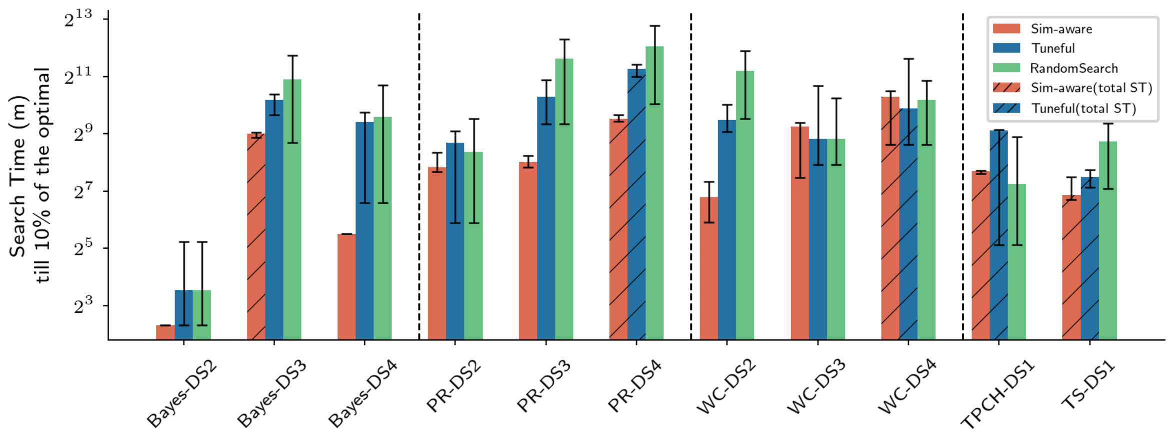

Figure 7: Search time until finding 10% of the optimal configuration using Random search, similarity-aware Tuneful (Sim-aware) and Standalone Tuneful (in log_2). The dashed bars represent the total Search Time (ST) when a configuration within 10% of the optimal is not found.

tuning algorithm (Alg. 2), comparing against: 1) Direct transfer, which implies transferring the tuned configuration from the source workload directly to the target workload (the source workload is found as described in § 2.4). This comparison will show the importance of retuning the configuration. 2) Independent workload tuning using Tuneful's significance aware algorithm (as described in § 2.2 and § 2.3). We compare against it to assess the accuracy of reusing the significant parameters of a similar workload against performing an independent detection of the significant parameters and tuning. We refer to this as *Standalone Tuneful.* 3) For the sake of completeness, we also compare the tuned configuration against Random Search, with a budget of 100 executions generated using low-discrepancy sequences [22]. We compare against this approach to asses how far the configurations provided by a Similarity-Aware Tuneful is from the best estimated configuration using this intensive exploratory approach. We used the AWS 4 nodes cluster to run this experiment.

Application	Input data sizes (DS{1,2,3,4})
PR	5, 10, 15, 20 (million pages)
Bayes	5, 10, 30, 40 (million pages)
WC	32, 50, 80, 100 (GB)
TPC-H	20 (compressed GB)
TS	200 million rows

Table 2: The set of applications and input sizes used to evaluate the dynamic configuration tuning.

Auxiliary workload set: consists of three workloads from three applications (Bayes-DS1, PR-DS1 and WC-DS1). Table 2 shows the auxiliary workloads colored in green. We excluded two applications (TPC-H and Terasort) from the set of the auxiliary workloads to evaluate the effectiveness of tuning unseen applications.

Workloads: for each application in the auxiliary workload set, we tune three workloads of evolving input sizes using Tuneful's similarity-aware tuning algorithm and compare against the other 3 approaches. Table 2 shows each workload's input size.

Our experiment with tuning those workloads shows that the direct transfer of the configurations does not guarantee near-optimal performance. The similarity-aware Tuneful finds configurations comparable to standalone Tuneful and RandomSearch, outperforming the direct transfer by 32% at median and 79% at the 90th percentile. Across all the workloads in Fig. 7, the similarity-aware Tuneful finds on average a configuration with an execution time within 8% of the Standalone Tuneful and 12% of RandomSearch.

Similarity-aware Tuneful search times: Fig. 7 shows the search time of each approach until finding a configuration within 10% of the estimated optimum. For some workloads, a configuration within 10% of the estimated optimal configuration is never found, and we present those cases as dashed bars in the figure, with their height representing the total search time.

Overall, the median search time for the standalone Tuneful and RandomSearch is 2.3 - 3.7X longer when compared to the similarity-aware Tuneful. Out of the 11 experimented workloads and given the limited set of auxiliary workload, the similarity-aware Tuneful significantly accelerates the search time of 6 workloads while finding a configuration within 10% of the estimated optimal. For the remaining workloads, the total search time of the similarity-aware Tuneful is still considerably smaller, with a configuration that is not only notably outperforming direct transfer, but also comparable to the standalone Tuneful for most of the workloads and within 17%-37% of RandomSearch.

This shows that a significantly shorter search phase is possible even with a limited number of source workloads to transfer tuning data from. However, in around 45% of the cases this also means that the reached configuration is slightly further away from the optimum. The next experiment will suggest that this can be mitigated by starting the use of similarity-aware tuning once a larger set of "auxiliary workloads" exists.

Tuning with extended knowledge: We give an example of better matching to related workloads by extending the auxiliary workload set to include the Bayes-DS2 workload. This allows the similarity-aware algorithm to select it as the source workload for Bayes-DS3, since it has a closer fingerprint than Bayes-DS1. Extending the auxiliary workload set enables the similarity-aware tuning to happen in 38% less time compared to results in the previous section, while

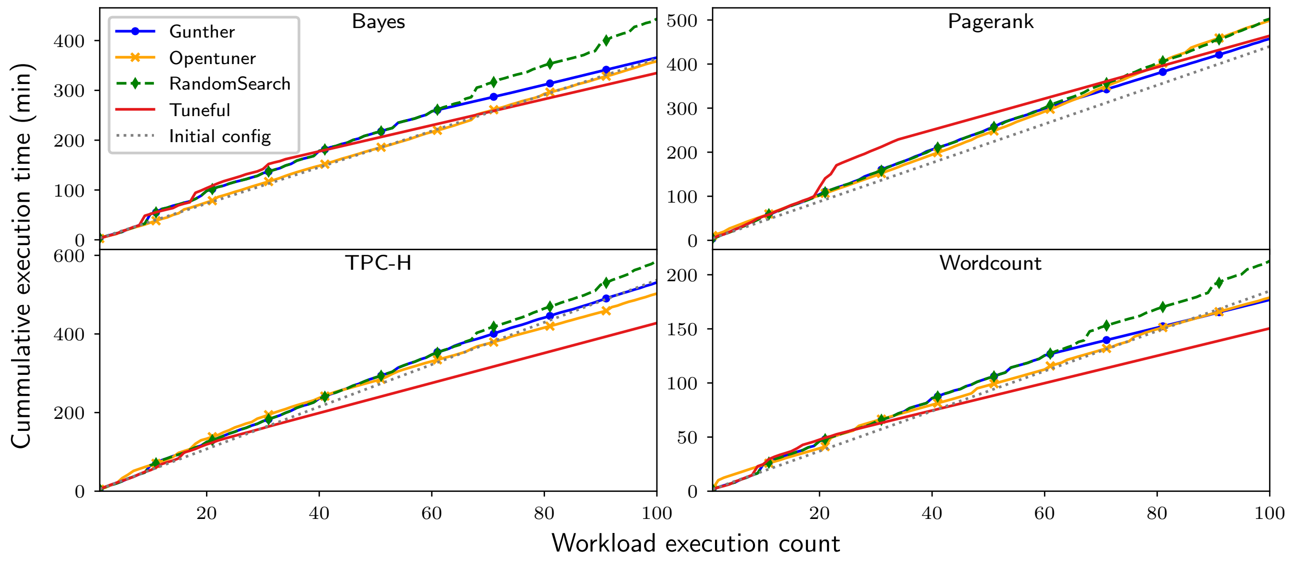

Figure 8: Cumulative execution time over 100 workload executions, under iterative tuning. Shallow slopes represent better configurations (smaller time increment for executing the workload once). Steeper slopes represent worse configurations.

finding a configuration comparable to Tuneful and RandomSearch (outperforming direct transfer by 54%). This completely eliminates the trade-off between search time and optimality of the configuration that was present in the limited knowledge case.

It also suggests that cloud providers, if able to observe executions across clients, would be in the ideal position to offer tuning-as-a-service in a way that minimizes costs, even for workloads that are repeated just a couple of times. Individual customers doing their own tuning will have a more complex decision to make depending on their workload and estimates provided by an amortization model.

Single tasked versus multitasked tuning: It is important to note that by leveraging MTGP optimization to perform Tuneful's similarity-aware tuning (Alg. 2), we are able to find good configurations faster than other approaches while bounding the search cost. When we experimented with TL+STGP, which transfers the significant parameters from the source workload then applies the standard single tasked GP (STGP) optimization. It incurred a higher exploration cost to find good configurations. In comparison, our similarity-aware tuning finds configurations comparable in execution time to TL+STGP, while significantly bounding the exploration cost by leveraging the data from the tuning of the source workload. This minimizes the chance of trying costly configurations, eventually enabling a quicker amortization of the tuning costs. The next section discusses this in more detail.

3.3.3 The amortization of tuning costs: The previous experiment does not show the full story on how the different approaches compare in behaviour as they perform incremental tuning from one execution to the next. For that, it is useful to have a timeline view.

The amortization of tuning costs with zero knowledge: Based on the experiment in § 3.3.1, Fig. 8 shows the cumulative execution time of running each workload over multiple configurations, as determined iteratively by the tuning algorithms considered. The graphs are the real-data versions of Fig. 3 and can be interpreted in the same way. Here, the fixed configuration and the starting point for tuning is a plausible developer-guided configuration that reduces exploration costs across the large search space. The dotted line shows cumulative execution time for this configuration *without any tuning*.

Tuneful explores the search space for 35 executions (20 during SA and 15 for tuning), then picks the best configuration it found and continues only using that. We let other tuning algorithms run longer (100 executions) to see if they find configurations that are better or equivalent to Tuneful's. However, most of the time they become stuck in local minima of the cost function. Better configurations are shown as lines with shallower slopes (e.g. when Wordcount is tuned by Gunther, after execution 60), while equivalent configurations appear as lines parallel to Tuneful's (e.g. Gunther for the Bayes workload). For Pagerank, the initial configuration proved to be a very good one and hard to beat through tuning. While both Tuneful and Gunther find better configurations than it, the exploration cost is not amortized in 100 executions. For Tuneful, the difficulty of cost amortization comes from a couple of very slow configurations sampled at the beginning of tuning (after execution 20).

This result suggests that if a workload is executed a small number of times, then tuning does not pay off during the workload's lifetime and it is more practical to use a plausible configuration (developer-guided or default cloud provider configuration), even if it is suboptimal. On the other hand, if a workload is to be executed more frequently, then it is worthwhile to tune the configuration as long as the tuning cost is amortized during workload's lifetime. Here, the benefits of the tuning hinge on how long the workload remains "alive" (with the same characteristics) after amortizing the tuning cost.

The amortization of tuning costs happens faster with more tuning knowledge: Based on the experiment in § 3.3.2, Fig. 9 shows the amortization of the tuning cost for Bayes-DS3 under different tuning scenarios. We compare against the directly transferred configuration from a source workload. This represents the *static* tuning approach followed by most of the existing tuners, in which the workload is tuned once and the tuned configuration is reused – ignoring the need for dynamic workload retuning. The dotted line shows cumulative execution time for this configuration without any tuning. The *dynamic* tuning "pays off" only after the

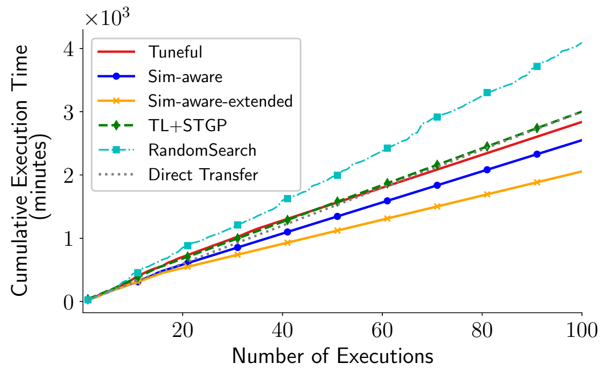

Figure 9: Cumulative execution time over 100 workload executions, representing the amortization speed of the different tuning approaches (Bayes-DS3 workload).

lines intersect the dotted line. The similarity-aware Tuneful runs a fingerprinting execution sample once and tunes the configurations incrementally within 15 executions, then picks the best configuration it found and continues only using that until retuning is needed. As shown in Fig. 9, it takes less than 5 executions to amortize the tuning cost using the similarity-aware Tuneful, with a better configuration (shallower slope) found when the tuning knowledge is extended (Sim-aware extended). This enables a quicker adoption for workload retuning in a rapidly changing environment. On the other hand, standalone Tuneful needs more executions to amortize this cost (e.g. 50 executions), but remains necessary for building an initial tuning database against which similarity analysis can be done.

4 CONCLUSIONS

The effectiveness of Tuneful firmly shows the need to address the configuration tuning of data analytics differently. The proposed data-efficient tuning methods can significantly reduce the exploration costs and accelerate their amortization, while finding configurations that are comparable to the ones of existing state-of-art tuning algorithms.

Not all workloads will benefit from this approach. However, the ones that are recurrent and subject to variations in input data size or in the resources allocated for execution fit our proposed tuning cost amortization model. They will likely be able to be tuned in a cost-effective manner using the methods proposed here. Our experiments suggest that incremental configuration tuning is the right approach when data analytics workloads are executed in a dynamic environment.

The dataset we release together with this paper can be used as a starting point for bootstrapping a database of tuning models and fingerprints against which similarity-aware tuning is possible. If maintained in the open-source space, with external contributions, this could make fast-amortizing tuning a reality even if tuning-as-a-service does not materialize on the cloud provider side.

REFERENCES

[1] TPC-H SQL benchmark, 2014. http://www.tpc.org/tpch/.
[2] Apache Spark: fast and general engine for large-scale data processing, 2015. https://spark.apache.org/.
[3] Amazon EC2 instance Pricing, 2018. https://aws.amazon.com/ec2/pricing/on-demand/.
[4] Hadoop distributed file system, 2018. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[5] Tuneful: Experiment data repository, 2020. https://github.com/ayat-khairy/tuneful-data.git.
[6] Tuneful: project repository, 2020. https://github.com/ayat-khairy/tuneful-code.git.
[7] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.
[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, volume 2, pages 4–2, 2017.
[9] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315. IEEE, 2014.
[10] Emanuele Borgonovo and Elmar Plischke. Sensitivity analysis: a review of recent advances. *European Journal of Operational Research*, 248(3):869–887, 2016.
[11] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423, 2015.
[12] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.
[13] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. Tuneful: An online significance-aware configuration tuner for big data analytics, 2020. https://arxiv.org/pdf/2001.08002.pdf.
[14] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.
[15] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.
[16] SPT Krishnan and Jose L Ugia Gonzalez. Google compute engine. In *Building Your Next Big Thing with Google Cloud Platform*, pages 53–81. Springer, 2015.
[17] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of MapReduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, pages 63–72. ACM, 2012.
[18] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: Search-based auto-tuning of MapReduce. In *European Conference on Parallel Processing*, pages 406–419. Springer, 2013.
[19] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.
[20] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A middleware for parameter tuning of NoSQL datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 28–40. ACM, 2017.
[21] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.
[22] Ilya M Sobol. On quasi-monte carlo integrations. *Mathematics and computers in simulation*, 47(2-5):103–112, 1998.
[23] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012, 2013.
[24] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan 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.
[25] Guolu Wang, Jungang Xu, and Ben He. A novel method for tuning configuration parameters of Spark based on machine learning. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 586–593. IEEE, 2016.
[26] John A Weymark. Generalized Gini inequality indices. *Mathematical Social Sciences*, 1(4):409–430, 1981.
[27] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 564–577. ACM, 2018.
[28] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350. ACM, 2017.

A APPENDIX

A.1 Experiment Reproducibility

A.1.1 Tuneful Usage: In order to use Tuneful, a Spark user simply adds Tuneful library as a dependency and Tuneful as an extra Spark listener while submitting his workload to Spark. In other words, Tuneful can run on an unmodified Spark infrastructure. Listing 1 shows how to clone Tuneful code, build and use to tune one of Spark's example workloads (SparkPi).

```
$ git clone https://github.com/ayat-khairy/tuneful-
     code.git
$ cd tuneful-code
$ mvn clean package
$ /usr/lib/spark/bin/spark-submit
  --jars target/tuneful-0.0.1-SNAPSHOT-jar-with-
      dependencies.jar
  --conf spark.extraListeners=TunefulListener
  --class org.apache.spark.examples.SparkPi
   /path/to/examples.jar 100
```

Listing 1: Tuneful example usage

A.1.2 Cluster and configuration specification: We use a cluster of 20 Google Compute Engine [16] instances (1 driver + 19 workers), with the driver being an *n1-highmem-8* instance with 8 vCPUs, 52 GB memory and 300GB storage and the 19 workers being *n1-standard-16* instances with 16 vCPUs, 60 GB memory and 500GB storage each. The total cluster memory and storage size is 1.2 TB and 5.48 TB respectively. We also use a smaller to validate the robustness of Tuneful, a cluster of 4 AWS *h1.4xlarge* instances. We use HDFS [4] version 2.7 for accessing the shared data and Spark version 2.2.1.

The default set of parameters that Tuneful tunes is in Table 3. We selected those parameters as they cover a wide range of Spark's internal aspects (memory, processing, shuffle and network aspects) and represent a superset of the ones used in the related work [27, 28], with approximatively $2 \cdot 10^{40}$ configurations possible in total (this represents the size of the search space).

A.1.3 Workload characterization Dataset: We built a dataset of execution metrics using two well known big data benchmarks (Hibench [15] and TPC-H [1]). We selected 5 applications of heterogeneous characteristics from these benchmarks. For each application we experimented with five different input sizes under different configurations for the 30 parameters in Table 3, with a total of 2200 application executions that took 2188 compute hours to execute (on a cluster of 4 AWS *h1.4xlarge* instances). The dataset is publicly available on the project data repository [5] under dataset folder.

A.1.4 Experiment data: The experiment data of Tuneful and the-state-of-the-art approaches are publicly available on [5] under experiment folder. This data can be leveraged for significant configuration parameter analysis for various workloads over different clusters. This data also shows the benefits of Tuneful's similarity-aware tuning in accelerating the tuning of a similar workload or retuning a seen workloads to accommodate the growing input sizes.

A.2 Significance Parameter Detection Algorithm

The input arguments are as follows:

- d the number of configuration parameters considered;
- $P = \{p_1, ..., p_d\}$ the configuration parameters;
- $R = \{r_1, ..., r_d\}$ the range of values of each p_i configuration parameter;
- α the fraction of configuration parameters retained in each SA round;
- n the number of samples required per SA round;

Conceptually, we consider three global variables: *n_SA_rounds*, the number of SA rounds, *n_executions*, initialized to 0, and P_{fixed} initialized to the empty set. Variable states are maintained between calls, so that P_{fixed} grows between SA rounds.

Let $X_i = \{x_{i1}, x_{i2}, ...x_{id}\}$ be a particular configuration chosen by our algorithm, $C(X_i)$ its execution cost, and $M(\mathbf{X}, \mathbf{C})$ the constructed meta-model that maps a given configuration X to its execution cost C, used to distinguish the influential parameters. The goal is to identify $P_s = \{p_1, p_2, ..p_s\}$ where $|P_s| < |P|$ such that P_s contains the *selected* top s influential parameters.

Algorithm 1: Significant Parameter exploration

```
Input  : $d, \alpha, n, n\_SA\_rounds, P, R$
Output : $P_s = P_{\alpha*d}$
1  $X_i = sample(P, R, P_{\text{fixed}})$
2  run workload using $X_i$ and get $C_i(X_i)$
3  $n\_executions \leftarrow n\_executions + 1$
4  if $n\_executions > n$ and $n\_SA\_rounds > 0$ then
5      build $M(\mathbf{X}, \mathbf{C})$
6      find_the_importance $imp\{p_1, ..., p_d\}$ using $M$
7      find $P_{\alpha*d} \subset P$ with the highest importance
8      $P_{\text{fixed}} \leftarrow P - P_{\alpha*d}$
9      $d \leftarrow \alpha * d$
10     $n\_SA\_rounds \leftarrow n\_SA\_rounds - 1$
11     $n\_executions \leftarrow 0$;
```

Design Choices: We empirically observed the impact of different values of α from 0.1 till 0.9. While a very small value for α leads to pruning influential parameters, high values for α will eventually lead to a wider search space that includes many uninfluential parameters and slows the identification of the highly-influential ones. We set $\alpha = 0.6$ in the SA stage as it represents a good compromise between the accuracy of detecting the influential parameters and bounding the number of SA runs. We set n to 10 execution samples, as it guarantees generating an RFR model of an acceptable accuracy (less than 40% error), enabling a good *approximation* of the true dependence between the configuration parameters and execution time. *n_SA_rounds* needs to be selected to provide good guarantees, while minimizing cost. We experimented with different number of *n_SA_rounds* for Tuneful in [13].

A.3 Similarity-aware tuning Algorithm

The input arguments are as follows:

- w target workload;
- $W_{seen} = \{w_1, w_2, ..\}$ the matrix of the execution metrics for the seen workloads;
- α the acquisition function;
- $GP = \{GP_1, GP_2, ..\}$ the Gaussian process models of the seen workloads;

#	Configuration name	Range	Default	Description
1	spark.executor.cores	[1,16]	1	The number of cores per each Spark executor
2	spark.executor.memory(GB)	[5,43]	1	The memory size of each Spark executor
3	spark.executor.instances	[8,48]	2	The number of Spark executor instances
4	spark.default.parallelism	[8,50]	-	The number of partitions in a returned RDD by the distributed shuffle operation, its default value varies depending on the distributed shuffle operation
5	spark.memory.offHeap.enabled	[true,false]	false	If set to true, Spark will try to use the off-heap space for certain operations
6	spark.memory.offHeap.size(MB)	[10,100]	0	The size of memory that can be used for off-heap allocation
7	spark.memory.fraction	[0.5,1]	0.6	The fraction of heap space used for execution and storage, if set to a low value spills and cached data eviction takes place more often.
8	spark.memory.storageFraction	[0.5,1]	0.5	The fraction of spark.memory.fraction that is not evicted by Spark, if set to high value, less memory will be available to execution and disk spill will occur more frequently
9	spark.shuffle.file.buffer(KB)	[2,128]	32	The size of each shuffle buffer output stream in-memory
10	spark.speculation	[true,false]	false	If set to true, Spark will check if one task or more are running slowly in a stage, it will re-launch them
11	spark.reducer.maxSizeInFlight(MB)	[2,128]	48	The maximum allowed size to fetch from a map output of each reduce task
12	spark.shuffle.sort.bypassMergeThreshold	[100,1000]	200	How often Spark avoids merge-sorting data in the sort-based shuffle manager
13	spark.speculation.interval(MS)	[10,100]	100	How frequently Spark will check for tasks to speculate, Spark speculation is a procedure that detects the tasks running slower than the median of all the successful tasks, Spark then restart these tasks
14	spark.speculation.multiplier	[1,5]	1.5	Defines how to consider a task for speculation w.r.t the successful tasks median execution time
15	spark.speculation.quantile	[0,1]	0.75	The fraction of tasks that should be finished before starting speculation on a stage
16	spark.broadcast.blockSize(MB)	[2,128]	4	The size of the broadcasted blocks in Spark, larger value would decrease broadcast parallelism
17	spark.io.compression.codec	[snappy,lzf,lz4]	lz4	The compression technique spark uses for its internal data such as RDD
18	spark.io.compression.lz4.blockSize(MB)	[2,128]	32	The block size used by lz4 compression
19	spark.io.compression.snappy.blockSize(MB)	[2,128]	32	The block size used by snappy compression
20	spark.kryo.referenceTracking	[true,false]	true	Determines if Spark will track references to the same object when using kryo serializer
21	spark.kryoserializer.buffer.max(MB)	[8,128]	64	The maximum size of the buffer used by Kryo serializer
22	spark.kryoserializer.buffer	[2,128]	64	The size of kryo serialization buffer initially
23	spark.storage.memoryMapThreshold	[50,500]	2	A block size beyond which Spark performs memory mapping of the disk read blocks, memory mapping very small blocks will incur higher overheads
24	spark.network.timeout	[20,500]	120	The timeout of all network interactions in Spark
25	spark.locality.wait	[1,10]	3	The amount of time Spark waits to launch a data-local task before moving the task to a less-local node
26	spark.shuffle.compress	[true,false]	true	Specifies if Spark compresses map output file, compression happens using spark.io.compression.codec
27	spark.shuffle.spill.compress	[true,false	true	Determines if Spark compresses data spilled during shuffles
28	spark.broadcast.compress	[true,false]	true	Decides if Spark compresses broadcast variables before sending them
29	spark.rdd.compress	[true,false]	false	Specifies if Spark compresses serialized RDD partitions
30	spark.serializer	[JavaSerializer, KryoSerializer]	JavaSerializer	Sets the serialization strategy in Spark

Table 3: Configuration parameters description and values range.

Algorithm 2: Similarity-aware configuration tuning

```
  Input  : $\alpha, w, W_{seen}, GP$
1 forall $x_i \in W_{seen}$ do
2     calculate distance $d$ between $w_{metrics}$ and $x_i$;
3 Find workload $s$ with the smallest $d$
4 Transfer $s$ significant parameters to $w$
5 Add new task $t_j$ for workload $w$ to $GP_s$
6 Find config: $x_i \leftarrow \arg\max_x \alpha(GP_s(x, t_j))$
7 Evaluate config: $y_i \leftarrow C_w(x_i)$
8 Update: $GP_s \leftarrow GP_s|(t_j, x_i, y_i)$
```