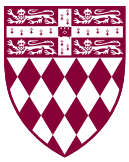
UNIVERSITY OF
CAMBRIDGE

Optimizing Data-Intensive Computing with Efficient Configuration Tuning

Ayat Fekry



Fitzwilliam College

This dissertation is submitted on July, 2020 for the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Ayat Fekry
July, 2020

Abstract

Optimizing Data-Intensive Computing with Efficient Configuration Tuning

Ayat Fekry

As the complexity of distributed analytics systems evolves over time, more configuration parameters get exposed for tuning. While these numerous parameters allow users more control over how their workloads are executed, this flexibility comes at a cost, since finding the right configurations for such systems in a cost-effective way becomes challenging.

In practice, several factors contribute to the complexity of tuning the configuration of those systems: the large configuration space, the diversity of the served workloads (each workload possibly requiring a different resource allocation strategy to run optimally), and the dynamic characteristics of these systems' environment (e.g., increase in input data size, changes in the allocation of resources). Paradoxically, existing solutions for workload tuning either assume static tuning environment or workloads that are inexpensive to run (i.e. requiring hundreds of execution samples).

Recently, Bayesian Optimisation (BO) strategies have been applied as a solution to enable efficient autotuning. They build a probabilistic model incrementally to predict the impact of the parameters on performance using a small number of execution samples. The incrementally constructed BO model is used to guide the tuning process and accelerate convergence to a near-optimal configuration. Unfortunately, for distributed analytics systems, the configuration space is too large to construct a good model using traditional BO, which fails to provide quick convergence in high dimensional configuration space.

I argue that cost-effective tuning strategies can only be developed when taking into account: the frequent changes that can happen in the analytics workload/environment, the *amortization* of tuning costs and how this influences tuning profitability, the high dimensionality of configuration space and the need to cater for *diverse* workloads.

To tackle these challenges, I propose Tuneful, an efficient configuration tuning framework for such expensive to tune systems. It works efficiently both initially (when little data is available) as well as later (as more tuning knowledge is acquired). It starts with learning

workload-specific influential parameters incrementally and tunes those only, then when more tuning knowledge becomes available, it detects similarity across workloads and utilizes multi-task BO to share the tuning knowledge across similar workloads. I show how augmenting the BO approach with parameters' significance and workload similarity characteristics enables an efficient configuration tuning in high dimensional configuration space. Over diverse analytics workloads, this significantly accelerates both configuration tuning and cost amortization, saving search time by 2.7-3.7X at median compared to the-state-of-the-art approaches.

Acknowledgements

"And my success is not but through Allah. Upon him I have relied, and to Him I return."
—Quran, surah Hud, 88

I'd like to sincerely thank Andy Hopper. I feel deeply indebted for the great support he gave me. During the course of securing funds for my PhD tuition fees, the first year of my PhD and even after he moved to the royal society: he has been knowledgeable, supportive and understanding in every possible way. I am impressed by how he fully understood what it means to be a female student dragging your family to pursue a PhD in computer science—let alone moving to a new country with a completely different culture. I really consider myself lucky to be one of his former students.

A special thanks to my supervisor Andy Rice for his invaluable guidance, useful feedback and always support. I am also extremely grateful to Lucian Carata for his devoted guidance and tireless support during this PhD journey.

I had long discussions with Clayton Rabideau about parallelising his work which certainly helped me to shape the dissertation's main contribution.

I am also indebted to Prof.Reem Bahgat, my dear professor back home in Cairo university and the chancellor of the Egyptian cultural bureau, for her support and encouragement.

My PhD was funded by Cambridge Trust, the Egyptian cultural bureau and the Cambridge computer laboratory (through the support of Andy Hopper), to whom I am really grateful. I am also thankful to Google and Amazon for generously supporting me with cloud research credit to run my experiments.

On the personal front, I'd like to express my gratitude to my beloved husband Mohamed for being supportive. Surprisingly, he quit his competitive job back home and decided to take a leap in the dark, joining me in achieving my long held dream of doing my PhD at Cambridge—unusual sacrifice taken by an understanding husband.

Special thanks to my adorable son Hamza for being really understanding. I was impressed by how he kept pushing me to finalize my PhD work, just to free me up before his little sister arrives!

Last but not least, I am grateful to my mum, brothers and sister (Mohamed, Mahmoud, Abdelrahman and Rofida) for their infinite support.

Contents

List of Figures

List of Tables

Publications and awards

Peer-reviewed publications

- Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. "Towards Seamless Configuration Tuning of Big Data Analytics." In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1912-1919. IEEE, 2019. (Parts of Chapter 7)

- Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics. In Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'20) - Applied Data Science track. (Chapter 4 and parts of Chapter 5 and Chapter 7)

- Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. Accelerating the Configuration Tuning of Big Data Analytics with Similarity-aware Multitask Bayesian Optimization. IEEE BigData'20. (Chapter 6 and Chapter 7)

Preprints

- Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. "Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics." arXiv preprint arXiv:2001.08002 (2020). (Chapter 5)

Funding Awards

- Google Cloud Platform research credit award ($12,000), 2018.
- Amazon Web Services research credit award ($37,000), 2019.
- Google Cloud Platform research credit award ($14,000), 2019.

List of Abbreviations

DISC Data Intensive Scalable Computing

BO Bayesian Optimization

GP Gaussian Process

SVM Support Vector Machine

ML Machine Learning

RFR Random Forest Regression

AE AutoEncoder

PCA principal Component Analysis

PaaS Platform as a service

CC Cloud Computing

DBMS Database Management System

HPO Hyperparameter Optimization

SA Sensitivity Analysis

MTBO Multitask Bayesian optimization

MTGP Multitask Gaussian Process

PR Pagerank

TS Terasort

AWS Amazon Web Services

GCP Google Compute Platform

VM Virtual Machine

RDD Resilient Distributed Dataset

DAG Directed Acyclic Graph

PI Probability of Improvement

EI Expected Improvement

SQL Structured Query Language

RL Reinforcement Learning

JCT Job Completion Time

JVM Java Virtual Machine

GC Garbage Collection

HPC High Performance Computing

MCMC Markov Chain Monte Carlo

RFE Recursive Feature Elimination

HDFS Hadoop File System

SVR Support Vector Regression

LR Linear Regression

STGP Single Tasked Gaussian Process

GP-UCB Gaussian Process-Upper Confidence Bound

ARD Automatic Relevance Determination

Chapter 1

Introduction

Recent years have witnessed a rapid growth in data volume in both research and industry [1]. Processing this ever growing data is crucial to extract valuable information that can be used to support decision making in businesses and strengthen their competitive advantage, or uncover insights and hidden patterns in research areas.

DISC (Data Intensive Scalable Computing) systems such as Hadoop [2], Spark [5] and Flink [3] have become a common choice when processing this "big data". These systems are typically characterized by the ease of writing analytics workloads and scaling them to accommodate the ever growing volume of data. They enable the manipulation and analysis of data by distributing work over a cluster of machines, making it easier to scale horizontally. Importantly, their abstracted programming model enables the programmer to simply write few lines of code to define a parallelizable function—the programmer is not exposed to any under the hood issues such as load balancing and fault tolerance. These characteristics have led to the wide adoption of DISC systems in businesses (53% of companies have embraced big data analytics in 2017 [48]), as well as in research areas such as biology [54], physics [102] and astronomy [132].

While DISC systems provide a convenient abstraction for a programmer to express a parallelizable workload, how to execute the workload is left to the system itself. Therefore, to allow the programmer to *govern* workload execution, DISC systems inevitably expose a set of configuration parameters. These include factors ranging from number of cores, settings of data transfer across cluster nodes (e.g., buffer size, compression technique).

1.1 Problem Statement

A poor choice of configuration parameters can have a dramatic impact on execution time. Misconfiguration can lead to either resource contention/exhaustion or under-utilization, with the former potentially triggering errors hours after the start of execution (e.g., the optimum configuration for a workload can lead to a late failure after hours of computation when used for another workload). Finding the right configuration for these systems is daunting, with each workload potentially requiring a different resource allocation strategy to run optimally, as will be shown

in the experiments of Chapter 5 and Chapter 7. Even worse, changes in workload or environment characteristics (input size, data, Virtual Machine (VM) migration) means that re-tuning might be needed. As an example on how running a workload with the optimum configuration of another leads to unpredictable results, when the optimal configuration of a Pagerank workload receiving 10 million pages is used for evolving input size of 15 million pages, it yields 85% longer execution time compared to the optimal configuration that can be found for the 15 million pages workload (according to the experiments illustrated in Chapter 7). Consequently, developers spend significant time and resources identifying the appropriate configuration for their workload.

Some work has been proposed towards the automation of configuration tuning that ranges from generic autotuners [135, 21] to system-specific ones [83, 129, 56], requiring hundreds of execution trails and incurring thousands of hours of computational power and associated monetary cost. Those high tuning costs are not practical for workloads that either take a long time to execute or change over time and require frequent re-tuning due to running in a *rapidly changing environment*, where workloads are subject to changes in their characteristics as a result of growth in the volume of data received or a change in the underlying physical infrastructure (e.g., VM migration, adding/shrinking cluster nodes). This is because retuning the configuration based on the existing work requires restarting the whole process from scratch, yielding an investment in time and resources comparable to the initial tuning costs.

To alleviate these high tuning costs, Bayesian Optimization (BO) strategies have been applied recently as a solution to enable efficient autotuning. They build a probabilistic model incrementally to predict the impact of the parameters values on performance using a small number of execution samples. The incrementally constructed BO model is used to guide the tuning process and accelerate the convergence to a near-optimal configuration. Unfortunately, for DISC systems, the configuration space is too large to construct a good model using traditional BO, which fails to provide quick convergence in high dimensional configuration spaces (more than 10 parameters [103]).

1.2 Proposed solution

This dissertation tackles the configuration tuning of systems that are complex and expensive to tune such as DISC systems. I propose an efficient incremental tuning framework that automates the configuration tuning. It is incremental since tuning is not done as an offline stage but incrementally using real workload executions, aiming to provide a better configuration each time. It can also consider similarities between workloads to reduce exploration costs and provide faster amortization. It is designed to work efficiently both initially (when little data is available) as well as later (as more tuning knowledge is acquired).

To achieve this, the framework augments BO with incrementally acquired knowledge. It starts with incremental learning of workload-specific influential parameters and tunes those

only, then when more tuning knowledge becomes available (through tuning more workloads), it detects similarity across workloads and utilizes Multitask Bayesian optimization (MTBO) to share the tuning knowledge across these similar workloads.

I show how augmenting the BO approach with incrementally acquired knowledge (i.e., parameters' significance and workload similarity characteristics) enables efficient configuration tuning in high dimensional configuration space. Over a diverse range of analytics workloads, this significantly accelerates both the configuration tuning and cost amortization. I argue the following thesis:

> *Augmenting BO with incrementally acquired knowledge can enable cost-effective tuning of diverse analytics workloads in a rapidly changing environment.*

1.3 Contributions

This dissertation makes three particular contributions, each of which is a fundamental aspect to enable efficient configuration tuning of data intensive computing:

1. My first contribution is to demonstrate, with a series of experiments, a number of overlooked issues that pose a challenge for practical configuration tuning in data analytics frameworks. I demonstrate that: 1) ignoring the dynamic characteristics (e.g., increase in input data size, changes in allocation of resources) of the analytics environment has a significant impact on performance; 2) cost-effective tuning requires that one amortizes tuning costs over saved execution time: many tuning solutions fail to provide a benefit in this respect; 3) optimal tuning varies between workloads and over the dynamic evolution of a workload.

2. My second contribution is to show that augmenting BO with incrementally acquired knowledge of parameters significance can efficiently tune high-dimensional configurations of diverse analytics workloads, outperforming the-state-of-the-art tuning approaches and saving search time by 2.7X at the median in comparison. This is the first work to propose a *data-efficient* [40] tuning of high-dimensionality configurations. Earlier solutions either require expensive offline phases or address low-dimensionality configurations. To achieve this, I develop an **Online Significance-aware Configuration Tuner** that leverages incremental Sensitivity Analysis (SA) to identify workload-specific influential configuration parameters, which represent the ones that influence performance most and are detected out of a high-dimensional space. It then tunes those detected parameters only using BO.

3. My third contribution is to provide evidence that extending BO with workload similarity characteristics can significantly accelerate the amortization of tuning costs and enable effective tuning in a rapidly changing environment—even when tuning configurations of high-dimensionality, which was previously thought to be impractical with BO. This extension is developed as a **Similarity-aware Configuration Tuner** that leverages neural encoding of workload execution metrics to detect similarity across workloads. It then shares the tuning

knowledge between similar workloads using MTBO. Over diverse analytics workloads, this significantly accelerates both configuration tuning and cost amortization (saving 56% of the search time when compared to an independent single tasked BO based tuning).

1.4 Dissertation Outline

The rest of the dissertation is structured as follows: I start with describing the needed background information and survey the existing related work in the first two chapters. I then present the thesis contributions in the following chapters.

Chapter 2 presents a background on big data, analytics workloads, Spark—which is the DISC system used to run the analytics workloads—and finally describes the machine learning algorithm that the proposed tuning approaches are based on. In particular, Gaussian Process (GP) and how it is used in the context of the BO framework.

Chapter 3 provides a survey of the existing approaches for tuning the configuration parameters of DISC systems. It also covers the tuning approaches used in generic tuning systems and other domains such as Cloud Computing (CC), Hyperparameter Optimization (HPO) and Database Management System (DBMS).

Chapter 4 describes the first contribution. It studies the challenges of configuration tuning and demonstrates, using a series of experiments, a set of overlooked issues in existing solutions. Lastly, it presents Tuneful, a comprehensive incremental tuning framework to overcome those issues and how it works efficiently both initially (when little data is available, I refer to this as zero tuning knowledge) as well as later (as more tuning knowledge is acquired). In a top-down manner, it briefly describes the fundamental components of this framework and how they work incrementally to enable efficient configuration tuning in both modes (zero and extended tuning knowledge). A detailed description of each mode is covered separately in the following chapters.

Chapter 5 presents the second contribution, an online *significance-aware* tuning approach and illustrates how Tuneful works efficiently when *zero tuning knowledge* is available initially. It describes Tuneful's incremental SA algorithm to identify the influential configuration parameters in a high-dimensional space and how it efficiently tune those parameters using BO. It then evaluates the proposed approach using four applications from two well known big data benchmarks (Hibench and TPC-H), and compares against the state-the-art tuning approaches.

Chapter 6 presents the third contribution, a *similarity-aware* multitasked tuning approach and illustrates how Tuneful works efficiently later when *more tuning knowledge* is acquired. Firstly, it describes Tuneful's learning of workload representation through encoding workload execution metrics into a lower dimensional space. Secondly, it shows how Tuneful detects workload similarities using this learnt representation. Lastly, it illustrates Tuneful sharing of the tuning knowledge gained previously across the similar workloads using MTBO.

Chapter 7 evaluates the *similarity-aware* multitasked tuning approach (Chapter 6) against: 1) Independent single tasked BO tuning as described in Chapter 5, 2) Direct transfer of the configuration across the similar workloads, 3) Transfer learning based tuning with single tasked BO and 4) Random search.

Chapter 8 concludes this dissertation and highlights avenues for future work.

Chapter 2

Background

The fundamental contribution of this dissertation is to enable cost-effective configuration tuning for DISC systems. This is done through augmenting BO with incrementally acquired knowledge (i.e., parameters' significance and workload similarity characteristics). In order to facilitate presenting how this contribution is realized, I start with providing the needed background information in this chapter. It gives an overview of big data, analytics workloads, and the DISC systems in particular Apache Spark [5], which is the representative DISC system chosen to run the analytics workloads for evaluating the algorithms developed in this dissertation. Then, I describe BO and GP and how they can optimize the exploration of the configuration space. Lastly, I demonstrate BO and GP limitations which make directly employing them to enable cost-effective tuning is not practical.

2.1 Big data and analytics workloads

In spite of being the "big data" era, there is no unified definition for big data. One of the most common definitions is: the data that can not be captured, stored, managed and processed using traditional computing tools in acceptable time [33]. Big data is characterized by 3V's: Volume refers to the amount of data; Velocity is the speed of capturing, preprocessing and analyzing data; and Variety refers to the heterogeneous types of data [136]. Oracle added a fourth 'V' to represent the value that could be obtained by analyzing data. Recently, veracity was added as a fifth 'V', it is the need to deal with uncertainty due to the inherited bias and inaccuracy in data [87]. Gartner defines big data as "*high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making*" [52].

The successful use of big data solutions mainly depends on the prompt extraction of valuable insights from data. This will continue to become more challenging due to the growth in data volume without a corresponding increase in the velocity to process this data. It is crucial to accelerate processing done by analytics pipelines to catch up with the ever growing data volume. For example, IDC predicts the global datasphere, which represents the amount of data created,

captured and replicated per year across the world, to reach 175 zettabytes (10^{21}) by 2025 [100], shaping 5X growth compared to 33 zettabytes in 2018.

The value of big data is revealed only when harnessed in decisions of big impact [51]. Timely analytics helps organizations make better and faster decisions that strengthen the competitive advantage of their businesses. With data volume growing exponentially in a way that poses the prompt distillation of insights as a fundamental challenge, the ability to process and analyze this exponentially growing amount of data will be critical to the success of every industry. The straightforward solution to process this huge data volume is to add more nodes and scale horizontally. However, this solution incurs high deployment, operational and maintenance costs—with data centres across the world already using around 3% of the world's electricity [9].

To accelerate the processing of this data, it is important to leverage the readily available optimization opportunities (scale up), thus saving extra scaling out costs.

2.1.1 Analytics workloads

Unlike previously, where businesses used basic analytics through spreadsheets or traditional relational database systems to uncover insights and trends in data [7]. Now, as the amount of data continued to grow, those systems failed to cope with this ever growing data volume, therefore distributed analytics systems have been developed to cope with bigger amounts of data. These distributed analytics systems enabled advanced analytics of high diversity (ranging from text analysis, machine learning, Structured Query Language (SQL) to graph analysis) to take place on large datasets, automating the discovery of hidden patterns and insights in data. A recent study shows that 53% of companies have adopted big data analytics in 2017 [48].

A workload in general is defined as all inputs received by a technological infrastructure [30]. In our context, an analytics workload is an application/service deployed on the cloud and consuming resources such as CPU, network, I/O. Workloads can be classified based on their resource requirements. A workload can be CPU-intensive requiring high computational power such as Wordcount, memory-intensive such as Pagerank, I/O intensive requiring frequent access to the data and causing significant data shuffle such as Terasort (also known as shuffle-intensive workload), or a hybrid workload that intensively requires multiple resources (e.g., Pagerank is both memory and shuffle intensive) [80]. It is worth noting that the classification of these workloads is subject to change depending on cluster setup, configuration parameters and input content and size. However, the classification is provided here to reflect the possible diversity in resource usage across workloads.

Another way to classify workloads is based on their usage patterns: while some workloads are static receiving fixed amount of data and have a predictable resource requirements, other workloads can be dynamic receiving evolving inputs and their resource requirements are susceptible to change. Unfortunately, there's very limited data published about workloads execution patterns in data centres [125, 8]. Google open-sourced 29 days logs of 12,500-machine cluster that does not tell much about workload execution patterns. Alibaba open-sourced two

datasets (one is the logs of 12 hours cluster usage and the other is 8 days usage). Those datasets are not sufficient to accurately describe the usage patterns of a cluster's workload.

Recently, a study reported that recurring jobs, where similar/static workloads are executed repeatedly (e.g., daily log analysis) shapes up to 40% of analytics jobs [16, 45].

Analytics benchmarks: In order to characterize the performance and trade-offs made by various data processing systems in a consistent way. It was essential to come up with representative benchmarks for the analytics workloads. Two of the common comprehensive benchmarks for analytics workloads are Hibench [62] and TPCH [6]. Hibench consists of real-life analytics workloads such as Pagerank, Bayesian classifier, wordcount, terasort and KMeans clustering. TPCH is a benchmark for SQL-based workloads. It consists of 22 decision support SQL queries. These benchmarks are used throughout the dissertation since they represent numerous real-life analytics workloads (e.g., while Pagerank was initially developed by Google to rank the pages of the search result, it can also be used in product recommendation, friend suggestion in social network, or ranking news feeds).

Diversity-representative workloads: To cover diverse workloads throughout the dissertation, I selected a set of applications commonly used in the top three reported application domains. According to a recent study [121], the top three application domains are search engines, social networks, and e-commerce, contributing with 80% page views of all the internet services in total. The applications selected to build a diversity-representative set of analytics workloads are:

1. *Bayes* is an application that builds a Bayesian classification model. This kind of application is used in domains such as social network (e.g., Facebook status sentiment analysis [113], spam detection in social network [117]) and e-commerce (e.g., product review classification [118], product recommensation [109]).
2. *Pagerank* is a link analysis application used broadly in web search engines to calculate the ranks of web pages based on the number of reference links. It is also used as a graph analytics application that ranks the influence of graph vertices in social network domain.
3. *Wordcount* is a textual analysis application that counts word occurrences. It is commonly used in sentiment analysis applications [36] in social network domain.
4. *TPC-H* is a benchmark for big data systems that runs a suite of business oriented SQL queries that support decision making. The queries and the data populating the database is designed to have industry-wide relevance. SparkSQL is used to run these queries. This is a representative application in the e-commerce domain.
5. *KMeans* is an application that performs unsupervised machine learning to cluster data. It has many applications in social network (e.g., community detection [96]) and e-commerce (e.g., clustering customer trust [99]).
6. *Terasort* is a numeric data sorter that can be used in various applications in those top three domains.

2.2 DISC systems

DISC Systems such as Hadoop [2], Spark [5], Storm [4] and Flink [3] have been well received as platforms for processing huge data volumes. These DISC systems enable the manipulation and analysis of data by distributing the data over a cluster of machines. Each machine performs the same operation but on a different subset of data, hence the systems are sometimes referred to as "data-parallel" processing systems. Unlike compute-intensive systems where most of the execution time spent to meet computational requirements, DISC systems receive large volumes of data and spend most of the execution time in I/O and manipulation of data. There are a set of characteristics that distinguish DISC systems form other systems: 1) Scalability: they are designed to seamlessly scale horizontally by adding more nodes to facilitate accommodating the growing data volume. 2) Abstracted programming model: which enables the programmer to simply write few lines of code to define a parallelizable function—the programmer is not exposed to any under the hood issues such as load balancing and fault tolerance. 3) Reliability and availability: they are designed to be resilient to faults such as node failures or communication errors. Such faults are tolerated through failure detection, selective re-computation of missing results instead of re-executing the whole job.

DISC systems can be categorized into batch, stream or hybrid systems [32]. The batch processing systems (e.g., Hadoop MapReduce) receive a huge volume of data collected over a period of time (e.g., daily logs) then process them. In contrast, stream processing systems (e.g., Storm) receive continual input which should be processed in a small time period (i.e., near real time). An example of workloads running using stream processing systems is real-time sensor data analysis. On the other hand, the hybrid systems (e.g., Spark, Flink) combines both the capabilities of batch and stream processing systems. This is usually done through micro-batching, wherein streams of data is considered as a series of very small batches that can be processed using the batch engine.

In this section, a system of each category is described—with Spark described in more detail since it is the representative DISC system chosen for evaluating the algorithms developed in this dissertation. Spark is chosen not only due to its wide popularity [48], but also as it poses significant challenges for state-of-the-art configuration tuners (huge configuration search space, a variety of ways to process data) [129].

Storm is a stream processing system characterized by its scalability, fault tolerance and low latency. Storm processing model is based on defining togologies, which describe the transformations applied on each piece of data received by the system. The topology consists of: 1) streams of data continuously received by the system. 2) Spouts which represent sources of data stream such as queues producing data to be processed. 3) Bolts represent a processing step that takes one or more input stream(s) as an input (from a Spout), applies certain transformation on it and possibly outputs new stream.

Storm has a number of configuration parameters that govern its execution behaviour. Re-

#	Parameter	Default
	Memory Tuning	
1	mapreduce.map.memory.mb	1GB
2	mapreduce.reduce.memory.mb	1GB
3	mapreduce.task.io.sort.mb	100
46	mapreduce.map.sort.spill.percent	0.8
5	mapreduce.reduce.shuffle.input.buffer.percent	0.7
6	mapreduce.reduce.shuffle.merge.percent	0.66
7	mapreduce.reduce.shuffle.memory.limit.percent	0.25
8	mapreduce.reduce.merge.inmem.threshold	1000
9	mapreduce.reduce.input.buffer.percent	0
	CPU Tuning	
10	mapreduce.map.cpu.vcores	1
11	mapreduce.reduce.cpu.vcores	1
12	mapreduce.task.io.sort.factor	10
13	mapreduce.reduce.shuffle.parallelcopies)	5

Table 2.1: Example of performance related Hadoop configuration parameters

cently, some work has been proposed to tune a set of parameters that impact performance most [27, 65]. This includes the number of worker Java VMs that Storm generates for a topology, number of Spout executors (threads generating tuples for the topology), number of Bolt executors and the max Spout pending which determines the maximum number of tuples (per spout) that can be at once in the topology.

MapReduce is a programming model proposed by Google to process large volumes of data in parallel in a cluster of nodes. It used to be one of the most adopted models in DISC systems. The MapReduce model offers developers to write their workloads using two functions: map and reduce. The Map function takes a set of inputs in the form of <key,value>and transforms them to another set of <key,value>. The output of the Map is fed as an input to the Reduce function to combine the data that has the same key into a smaller set of outputs. Apache Hadoop [2] embraces the MapReduce programming model and has been broadly used as big data processing system. More recently, Apache Spark [5] has gained a wide adoption, it is a successor of Hadoop MapReduce outperforming its performance by 10X in iterative applications [131]. Hadoop has a number of configuration parameters that control its execution behaviour in terms of CPU and memory. Table 2.1 lists examples of those performance related configuration parameters tuned in [81].

2.2.1 Spark

Spark has been widely adopted for in-memory big data analytics—with 30% of businesses considering Spark framework critical to their big data analytics strategies and 73% considering

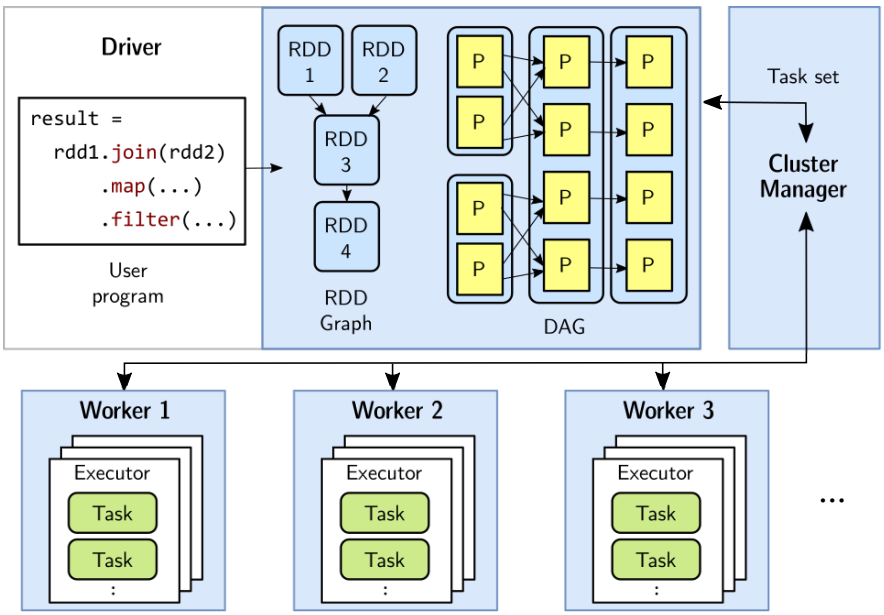


Figure 2.1: Spark internal architecture, redrawn with further edits from [72]

SparkSQL critical to their big data access strategies [48].

While Spark is designed with batch-oriented workloads in mind, it can support stream processing capabilities. To achieve that, it employs micro-batching, considering a streams of data as a series of very small batches that can be processed like a normal batch workload. It buffers the stream in sub-second increments, with this buffered data received as small input datasets for batch processing.

On the batch processing side, Spark has been developed to overcome the limitations of the MapReduce [39] paradigm in handling iterative workloads. MapReduce forces mappers to write data to disk for reducers to read, which consumes significant I/O resources for iterative applications. Spark keeps the data in memory as Resilient Distributed Dataset (RDD)s [130], a choice that significantly reduces I/O costs and speeds up iterative job execution time by up to 10X compared to Hadoop [131]. RDDs are immutable collections distributed over a cluster of machines to form a restricted shared memory, with each RDD consisting of a set of partitions. Figure 2.1 shows how Spark works internally. Users write a program and submit it to the Spark Driver, a separate process that executes user applications and schedules them into executable jobs. The Spark programming model is based on two types of function: transformations and actions. Transformations represent lazy computations on the RDD that create a new RDDs (e.g., map, filter). Actions trigger computation on an RDD and produce an output (e.g., count, collect). When an application invokes an action on an RDD, it triggers a Spark job. Each job has an RDD dependency graph containing all the ancestor RDDs, representing a logical execution plan for the set of transformations. The RDD graph is mapped into a Directed Acyclic Graph (DAG) defining the physical execution plan: a split of the job into stages, the dependencies between stages and the partitions processed in each stage. The driver uses this DAG to define the set of tasks to execute at each stage. Typically, an RDD partition is given as input to a stage

#	Parameter	Default	Description
	Execution Behavior		
1	spark.executor.cores	1	The number of cores per each Spark executor
2	spark.executor.instances	2	The number of Spark executor instances
3	spark.default.parallelism	-	The number of partitions in a returned RDD by the distributed shuffle operation, its default value varies depending on the distributed shuffle operation
46	spark.broadcast.blockSize(MB)	4	The size of the broadcatsed blocks in Spark, larger value would decrease broadcast parallelism
5	spark.storage.memoryMapThreshold	2	A block size beyond which Spark performs memory mapping of the disk read blocks, memory mapping very small blocks will incur higher overheads
	Memory Management		
6	spark.executor.memory(GB)	1	The memory size of each Spark executor
7	spark.memory.offHeap.enabled	false	If set to true, Spark will try to use the off-heap space for cetrain operations
8	spark.memory.offHeap.size(MB)	0	The size of memory that can be used for off-heap allocation
9	spark.memory.fraction	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.
10	spark.memory.storageFraction	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
	Shuffle Behavior		
11	spark.shuffle.file.buffer(KB)	32	The size of each shuffle buffer output stream in-memory
12	spark.reducer.maxSizeInFlight(MB)	48	The maximum allowed size to fetch from a map output of each reduce task
13	spark.shuffle.sort.bypassMergeThreshold	200	How often Spark avoids merge-sorting data in the sort-based shuffle manager
14	spark.shuffle.compress	true	Specifies if Spark compresses map output file, compression happens using spark.io.compression.codec
15	spark.shuffle.spill.compress	true	Determines if Spark compresses data spilled during shuffles
	Scheduling		
16	spark.speculation	false	If set to true, Spark will check if one task or more are running slowly in a stage, it will re-launch them
17	spark.speculation.interval(MS)	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
18	spark.speculation.multiplier	1.5	Defines how to consider a task for speculation w.r.t the successful tasks median exection time
19	spark.speculation.quantile	0.75	The fraction of tasks that should be finished before starting speculation on a stage
20	spark.locality.wai	3	The amount of time Spark waits to launch a data-local task before moving the task to a less-local node
	Compression and Serialization		
21	spark.io.compression.codec	lz4	The compression technique spark uses for its internal data such as RDD
22	spark.io.compression.lz4.blockSize(MB)	32	The block size used by lz4 compression
23	spark.io.compression.snappy.blockSize(MB)	32	The block size used by snappy compression
24	spark.kryo.referenceTracking	true	Determines if Spark will track references to the same object when using kryo serializer
25	spark.kryoserializer.buffer.max(MB)	64	The maximum size of the buffer used by Kryo serializer
26	spark.kryoserializer.buffer	64	The size of kryo serialization buffer initially
27	spark.broadcast.compress	true	Decides if Spark compresses broadcast variables before sending them
28	spark.rdd.compress	false	Specifies if Spark compresses serialized RDD partitions
29	spark.serializer	JavaSerializer	Sets the serialization strategy in Spark
	Networking		
30	spark.network.timeout	120	The timeout of all network interactions in Spark

Table 2.2: Example of Spark configuration parameter with categories

and is processed by a Spark task. Finally, the driver sends tasks to the cluster manager, which assigns them to worker nodes. A worker node can have multiple executors, with each of them being a process executing an assigned task and sending the result back to the driver.

For each workload, the user needs to find the best choice for configuration parameters covering different execution aspects such as processing, memory, networking and data shuffling. Example to these choices are: how many executor instances? what is the size of memory per executor? what is the number of cores per executor? how many partitions within RDD? what is the size of shuffled data buffer? should the shuffled data be compressed?

Spark has around 200 configuration parameters with many of them do not influence Spark performance (e.g., Spark UI parameters). However, there are various parameters that control shuffle behavior, compression and serialization, memory management, execution behavior, networking and scheduling and can significantly impact Spark performance. Table 2.2 categorizes and describes example parameters that govern the behaviour of each these aspects. Exposing all of those knobs makes the system flexible. It also makes it difficult to run efficiently without significant expertise and measurement. Even so, human expertise is of little help in dynamic tuning.

While all the configuration tuning methods proposed throughout the dissertation are not limited to a particular system or cost function, I have targeted Spark as the data processing framework to configure, because it is both popular and poses significant challenges for state-of-the-art configuration tuners (huge configuration search space, a variety of ways to process data) [129].

2.3 Bayesian Optimization

While several machine learning algorithms can be employed to tune the configuration parameters, this dissertation leverages BO algorithms due to their data-efficient learning, making them widely applicable for modeling expensive cost functions (e.g., big data workloads). Other employed machine learning algorithms (e.g., hierarchical models [129], neural network [88], support vector regression [78]) incur high tuning costs, requiring hundreds of workload executions to build a tuning model. In a rapidly changing environment such as big data systems, these tuning costs are hardly amortized within workload's lifetime (as will be shown in Chapter 4).

This section describes the machine learning algorithm, which the proposed tuning approaches throughout the dissertation are based on. In particular, GP and how it is used in the context of the BO framework. It demonstrates how BO can optimize the exploration of the configuration space. Lastly, it illustrates BO limitations which make directly employing them to enable cost-effective tuning is not practical.

BO [89] is a method for minimizing blackbox functions f iteratively, using a limited number of samples. This is useful when it is expensive to evaluate f at a given point (such as running a big-data workload with a given configuration). BO is characterized by its *prior model* and

acquisition function: the prior model represents a space of possible target functions f, and the acquisition function guides the selection of the next evaluation point based on the prior modelled knowledge.

One of the widely accepted prior models for BO is GP. It represents a distribution over functions (a sample drawn from this process is a function) with given mean and covariance. Here, the mean function describes expected values at each point and the covariance function (also known as kernel) defines the smoothness of the functions which can be drawn as samples, encoding prior assumptions about the data that we want to model [97].

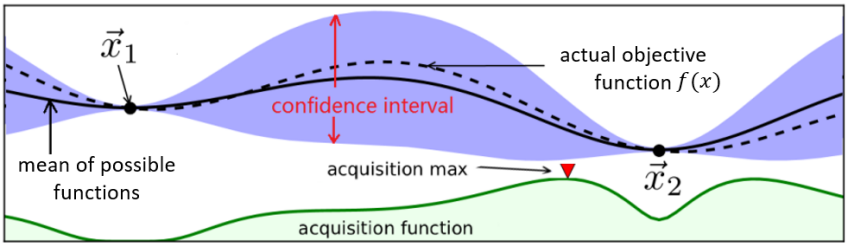
On of the broadly adopted covariance functions is Automatic Relevance Determination (ARD), which determines the relevance of each input dimension to the modelled function. This relevance is represented using the length scale, indicating how far a movement (along a particular dimension in the input space) is needed so that the function values become uncorrelated. The inverse of the length scale indicates the relevance. If the length-scale is high, then covariance is almost independent of that input dimension. This can be leveraged to get rid of exploring irrelevant input points. One of the widely used ARD kernel implementations is Matern 5/2 [98] which proved to be effective for modelling practical functions [105].

The GP maintains a probabilistic belief about what functions f are possible, given known characteristics and already seen data. This belief is updated by using an acquisition function, which determines the best point of f to sample next. After sampling, the prior belief about possible functions f is updated and a new sampling decision can be made, iteratively. At each step, the posterior distribution has filtered-out functions that are not consistent with the sampled data and will ideally have a narrower candidate function space.

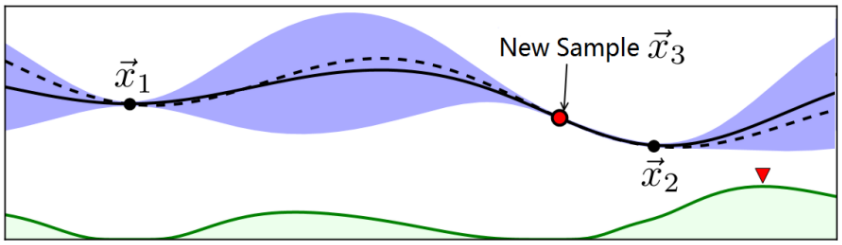
The BO acquisition function represents the metric by which the next input point to sample is picked so that it improves the probabilistic model function. It is typically a function that is cheap to evaluate at a given point x and its value is proportional to how useful evaluating $f(x)$ would be for the optimisation problem. Various acquisition functions have been proposed to define the way the BO samples the input space, e.g., random, sequential, Probability of Improvement (PI) or Expected Improvement (EI) [116, 107, 57].

The existing acuistion functions are either parametric (requiring setting its own hyperparameter(s) such as Gaussian Process-Upper Confidence Bound (GP-UCB)) or nonparameteric which optimizes its own hyperparameters without a need to set them priori.

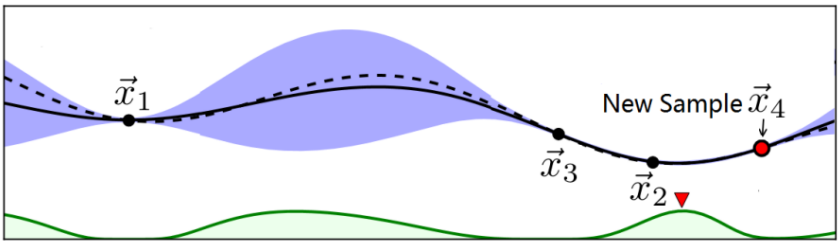
GP-UCB [97] has a hyperparameter that balances exploration and exploitation. As shown in Equation 2.1, the function finds the next point to sample x_{t+1} that maximizes a weighted sum of both mean μ_t (represents the exploitation) and standard deviation σ_t (represents the uncertainty) across all the function's input space. The function is fine adjusted by k to address the exploitation/exploration trade-off of the BO algorithm. While a small k value yields choosing input points that are expected to be high-performing, a high k focuses on exploring new areas of the search space that potentially have high-performing solutions. It is important to pick the right



(a) $t = 2$



(b) $t = 3$



(c) $t = 4$

Figure 2.2: An example of how BO works incrementally(figure source is [20]).

value for k in order to accurately model the function while bounding the number of sampled input data.

$$x_{t+1} = \arg\max_{x}(\mu_t(x) + k\sigma_t(x)) \tag{2.1}$$

On the other hand, EI with Markov Chain Monte Carlo (MCMC) [105] is an example of nonparametric acquisition function, wherein the grid is densely sampled and its hyperparameters are fine-tuned to optimize the EI.

I discuss the BO as it applies to DISC system's configuration tuning in Chapter 5 and Chapter 7.

2.3.1 How it works?

Figure 2.2 shows an example of how BO works incrementally on modelling an objective function and finding its minimum, the dashed line represents the actual objective function that BO tries to model, the black line represents the mean function of the possible functions. The purple area indicates the confidence interval, the range that the actual function should fall in with a high probability. The green curve at the bottom represents the acquisition function. The acquisition is high when the GP predicts high exploitation (high objective) and/or high explo-

ration (wider confidence interval/high uncertainty). BO samples the areas with both exploration and exploitation are high first then starts to compromise exploration for exploitation when a model of acceptable certainty is reached. At $t = 2$, BO has witnessed two observations (samples) that maps input into output, it filters out any functions that are not consistent with those samples, update the mean function and finds the next point to sample with the max acquisition. At $t = 3$, the point with the max acquisition is sampled leading to a narrower confidence interval (lower uncertainty), then at $t = 4$, the suggested next new sample is the one that maximizes the exploitation.

It is important to note that the BO did not select any points from the left side of the figure, since those points do not have the potential to maximize the acquisition. Leaving them unexplored saves modelling costs and makes BO a common choice for modelling expensive cost functions.

2.3.2 Advantages

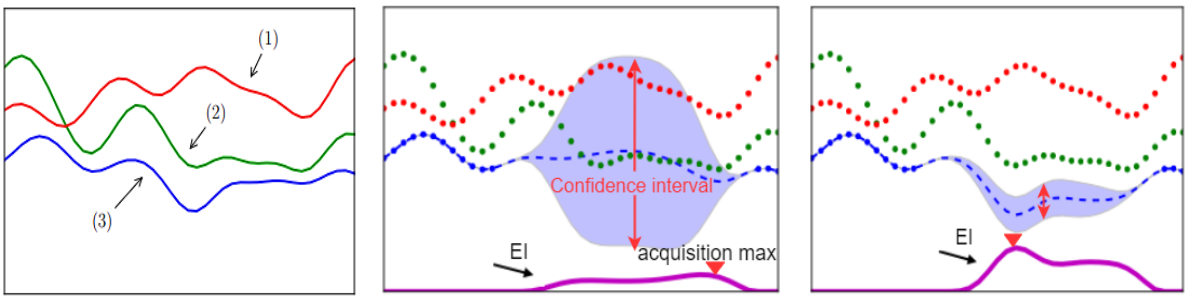
BO has three fundamental advantages that makes it appealing for modelling expensive blackbox functions: Firstly, BO models objective function using a small number of observations/samples since it focuses on sampling the points that has the highest potential gain. It is not only efficient in terms of the number of needed samples, but also the cost of the picked samples. This means that it can model blackbox function with minimal exploration cost. Secondly, BO does not commit the modelled function to a predefined shape, because it is nonparametric. This flexibility allows the GP to model diverse functions. Lastly, BO can model the uncertainty region of the modeled function, this uncertainty can happen in several contexts such as modelling the performance of cloud-hosted analytics. The BO's computed confidence interval can be used as a representation of this uncertainty.

2.3.3 Limitations

Despite the numerous advantages of BO, it converges slowly when instructed to model high dimensional space (more than 10 variables) [103]. Another limitation is that its computation complexity grows significantly with the number of observations (i.e. its complexity is $O(N^4)$, where N is the number of observations/execution samples). This computational overhead does not shape an issue in the context of this dissertation since I target using the BO to model the performance of analytics workload using a small number of observations (i.e., building a performance model from a maximum of 15 executions for any given workload).

2.3.4 Multitask Bayesian Optimization

MTBO is an extension to the standard BO that enables modelling and minimizing different tasks. This modelling principally relies on defining a covariance function between input-task pairs. While the covariance function of the standard GP defines the relationship between inputs, the covariance function of Multitask Gaussian Process (MTGP) defines the relationship between inputs and tasks. It learns the degree of correlation between tasks and utilizes this information



(a) Multitask GP sample functions (b) Independent GP Modelling (c) Multitask GP Modelling

Figure 2.3: (a) an example sample function from a Multitask GP with three tasks. Task 2 and 3 are strongly correlated, 1 and 3 are anti-correlated, and 1 and 2 are not correlated. (b) Independent single tasked GP modelling for Task 3. (c) Multitask GP modelling for Task 3, utilizing the other tasks (figure source is [108] with some edits applied for more clarity).

to guide the search for the best point to sample. Figure 2.3 shows an example MTGP with three tasks. In 2.3(a) the actual functions of the three tasks are depicted. Task 2 and 3 are strongly correlated, 1 and 3 are anti-correlated, and 1 and 2 are not correlated. The goal is to model task 3, the dots represent observations and the dashed lines represent the predictive mean. The curve at the bottom represents the EI for each input location on Task 3. The confidence interval indicates the range that the actual function should fall in with a high probability, the narrower the better. Whereas the independent GP in 2.3(b) has a wide confidence interval and optimizing EI leads to a false minimum as the next point to sample, the MTGP in 2.3(c) leverages the correlation across the tasks, thus the confidence interval is improved (narrowed) and the maximal EI point corresponds to the true minimum.

2.4 Summary

This chapter provided the needed background information in order to facilitate presenting the dissertation's key contributions. It gave an overview of big data, analytics workloads, the DISC systems and the different aspects of BO and GP in terms of how they work, limitations and their advantage of optimizing the exploration of the configuration space.

With this background in mind and before proceeding to the contributions, it is essential to demonstrate the context of these contributions through describing the related work. This is to show where the proposed contributions fit. The next chapter studies the existing configuration tuning approaches in various domains, spots the existing gap and illustrates what distinguishes our work to fill this gap.

Chapter 3

Related Work

In order to distinguish the proposed work in this dissertation from others and show how it fits in, this chapter studies existing approaches for tuning the configuration parameters of DISC systems. It also covers tuning approaches used in generic tuning systems and other domains such as CC, HPO and DBMS.

Most frequently, end-users tune system configuration manually based on expertise and/or trial-and-error. However, this manual tuning approach can lead to missing optimization opportunities. Further, it is extremely challenging to apply in a high-dimensional space. Similarly, a number of general methods (USE, TSA)[1] have been suggested as recipes for figuring out the location of bottlenecks and explaining the time spent by applications. Those are ways of guiding tuning or debugging performance issues rather than automating the tuning process for dynamic workloads.

Therefore, the problem of configuration autotuning gained traction from developers, researchers and end-users, with dedicated efforts put to propose strategies that address configuration autotuning in different domains. Generally, the tuning approaches either follow a search-based or model-based strategy:

Model-based configuration Tuning where a model is built to predict the execution time given a set of workload characteristics (resource consumption metrics) and a particular configuration [56, 78]. The model is pre-trained on numerous executions of different workload types and configurations, after which it can cheaply estimate run times for a new (workload, configuration) pair. However, tuning results are highly dependent on the model accuracy and on the similarity between workloads seen during training and actual workloads.

Search-based configuration Tuning where a random or incremental search of the configuration space is done. The latter feeds back information about the workload actual execution cost under a given configuration to select a better configuration for the next run [83]. Current approaches require a long search phase (around 500 executions) [135]. The practicality of such solutions hinges on the challenge of amortizing the cost of the optimisation, a function of both

[1] `http://www.brendangregg.com/methodology.html`

how fast the algorithm converges to very good configurations and how many bad (slow) configurations are explored in order to get there.

Out of the several domains in which autotuning approaches have been proposed, this chapter covers the recent work done in five domains of relevance and growing popularity, illustrating a set of representative autotuning approaches from each domain. I start with studying the configuration tuning in DISC systems, then cloud computing, machine learning, and lastly database management systems and generic tuning systems. Additionally, I give an overview on existing work that leverages workload similarity.

I characterize the existing work based on a set of features: 1) The domain which the work targets. 2) The tuning approach, wherein a search-based or model-based approach is employed. 3) Configuration dimensionality, which represents whether the work supports high dimensional configuration tuning or not. 4) The need for upfront profiled dataset, indicates if the work requires an expensive offline phase to build a dataset upfront, which is used to enable the tuning approach (e.g., build a performance prediction model, detect system-wide significant parameters). 5) Similarity-awareness, indicating whether the work can detect similarity across workloads and leverage this similarity to accelerate configuration tuning. 6) Dynamic tuning, this represents whether the work is designed to to detect the need for configuration retuning and performs retuning efficiently in response. This is crucial to accommodate workload/environment changes to avoid workload performance degradation. Table 3.1 shows a summary of the studied work based on those features, with each of this work described in detail in the next sections.

Work	Domain	Approach	upfront profiled dataset?	high dimensional conf. space?	similarity-aware?	dynamic-tuning?
Cherrypick [20]	Cloud	Model-based	×	×	×	✓
Scout [59]	Cloud	Model-based	×	×	×	×
Arrow [58]	Cloud	Model-based	×	×	✓	×
PARIS [127]	Cloud	Model-based	✓	×	×	×
Autotune [74], Optuna [19]	HPO	Search-based	×	×	×	×
Flash [133]	HPO	Model-based	✓	×	×	×
[34], [101]	HPO	Search-based	×	×	×	×
Ottertune [114]	DBMS	Model-based	✓	✓	✓	×
Rafiki [88], ituned [42]	DBMS	Model-based	✓	✓	×	×
AROMA [78]	Hadoop	Model-based	✓	×	✓	×
Starfish [56], RFHOC [24], H-Tune [61]	Hadoop	Model-based	✓	×	×	×
Gunther [83], JellyFish [41], MROnline [81]	Hadoop	Search-based	×	×	×	×
DAC [129]	Spark	Model-based	✓	✓	×	×
ATCS [82]	Spark	Model-based	✓	✓	×	×
BestConfig [135]	Generic	Search-based	×	✓	×	×
Spearmint [105]	Generic	Model-based	×	×	×	×
OpenTuner [21]	Generic	Search-based	×	✓	×	×
Tuneful (this work)	Spark	Model-based	×	✓	✓	✓

Table 3.1: The characteristics of the work addressing configuration tuning in different domains.

3.1 Configuration Tuning for DISC systems

Several solutions have been proposed for tuning the configurations of Hadoop/MapReduce workloads and more recently for Spark.

3.1.1 Hadoop Configuration Tuning

Some early work [50, 104] targeted the analysis of MapReduce performance without tackling configuration autotuning. Ganapathi et al. [50] leverage Hadoop logs for a period of 6 months to build a model that predicts jobs' performance under different execution features (e.g., inputs bytes, number of map and reduce stages, query operator). Shi et al. [104] utilize logs of Yahoo's M45 supercomputing cluster to analyse the performance of MapReduce jobs, leading to the characterization of job patterns, completion times, job failures and resource utilization. This characterization has suggested improvements in Hadoop MapReduce to mitigate performance bottlenecks and limit job failures.

Nearly a decade ago, the work on Hadoop configuration tuning has started with proposing *whitebox* model-based tuning approaches such as StarFish [56] and MRTuner [104]. These whitebox tuning approaches adopt the idea of transparent modelling, which requires a deep understanding of the system's internals to estimate the cost of executing a job under certain configuration. This is done through profiling numerous jobs under various configurations, then analyzing how they perform by breaking down their execution cost (e.g., execution time) into a set of basic operations attributing to the total cost (e.g., CPU utilization, memory utilization, size of shuffled data, configuration parameters), this breakdown is used to derive formulas for execution cost estimation (some work refers to these formulas as a cost model).

StarFish is a Hadoop tuning system inspired by the self-tuning database systems. It uses a configuration tuning approach in which cost estimation is derived by a What-If engine, predicting the cost of different configurations given some profiled data. For example, the engine can answer a query like "Given the profile of a job A, input data x, cluster resources c1, what will the performance of job A be with input data y and cluster resources c2". Here, finding good configurations hinges on the accuracy of the what-if engine itself. Similarly, MRTuner embraces a whitebox model-based tuning approach in which an anlytical model is built to estimate the tradeoffs in MapReduce execution plans and the execution cost accordingly. It also detects the relationship among parameters to develop a fast search algorithm for good configurations.

However, this whitebox model-based tuning approach has some limitations: It is very complex to capture such system's internal knowledge to build an accurate cost model. Moreover, the built cost model is tightly coupled to the system's internals and a change in the system would trigger a change in the model. Overall, this approach is hard to adopt to build a cost model with good accuracy for complex system such as Hadoop while keeping flexible and robust [128, 83].

These limitations have motivated the wide adoption of Machine Learning (ML) model-based tuning approaches, since they do not require extensive knowledge of the system's internals and

work in a black-box fashion. The work on ML model-based configuration tuning has started with AROMA [78], which is a system for Hadoop resource provisioning and configuration tuning. It uses the k-medoids algorithm to cluster the executed jobs and leverages Support Vector Machine (SVM) to build a ML model that predicts job's performance under different provisioned resources and configurations. The built model is used to automate resource provisioning and the tuning of 10 Hadoop configuration parameters.

Lee et al. [79] have proposed a fuzzy prediction controller to optimize the number of concurrent MapReduce jobs. The fuzzy prediction controller constructs rules that can predict resource usage and the number of concurrent tasks. Those predicted values are used to dynamically tune a single Yarn (Hadoop job scheduler) parameter (i.e., number of jobs in running state).

Liu et al. [86] have proposed a Reinforcement Learning (RL) based approach for managing resources in Hadoop. The proposed approach proved to outperform rule-based approaches for resource management in terms of reducing the chances of missing deadline for time-critical applications and lowering average job delay for all jobs.

Some work has been proposed leveraging ensemble learning [24, 61] where multiple models are combined to build a performance prediction model of better accuracy, the model is then used to search for good configuration thereafter. In H-Tune [61] a two-level fusion model is built to predict job performance considering the configurations and input data size. The constructed model is leveraged to search for the optimal configuration using a modified genetic algorithm.

Similarly, RFHOC [24] constructs an ensemble prediction model using random forest then employs a genetic algorithm to search the configuration space.

On the search-based tuning front, Gunther [83] has been proposed leveraging genetic algorithms to tune the Hadoop configuration parameters. Then, MROnline [81] proposed a modified Hill climbing technique to find good configurations; it limits the search space using predefined tuning rules. Lastly, JellyFish [41] tunes the configuration parameters using a divide-and-conquer approach (dividing the parameters into map-related and reduce related ones) then searches for good configurations using hill climbing algorithm. It also re-schedules resources in nodes to maximize the utilization of the cluster's resources.

Those MapReduce tuning systems are optimized for a low number of parameters, which fits Hadoop well but not other systems in general (for example, Spark). To demonstrate that, the proposed work in this dissertation will be comparatively evaluated against one of the MapReduce configuration tuners (Gunther [83] in particular due to the sufficient details presented in the paper to implement it and compare against, as will be discussed in Chapter 5).

3.1.2 Spark Configuration Tuning

More recently, a few systems have been developed for tuning Spark configurations—addressing a larger number of tunable parameters compared to MapReduce configuration tuners. Yu et al. have proposed DAC [129], a data-size aware Spark configuration tuning system, which uses a hierarchical modelling approach to approximate workload execution time as a function of input

data-size and configuration. It then leverages Genetic algorithms to search for good configurations based on the execution time estimated by the model. The work in this dissertation aims to avoid incurring the high data collection costs of DAC when modelling execution time as a function of a given configuration. Those high costs are hard to amortize before re-tuning is needed (e.g. because of environment change). DAC improves performance by 30-80X with respect to the default configuration and tunes 41 configuration parameters in Spark 1.6. Throughout the dissertation, I consider all parameters (30) of a more recent Spark release (some of the parameters have been deprecated). Wang et al. [119] leverage regression trees to tune Spark configurations; they tune 16 configuration parameters and improve performance by 36% with respect to the default configuration. This approach also needs a significant number of execution samples to build an accurate regression tree model. BestConfig [135] is a general-purpose tuning system that uses a divide-and-diverge sampling method and a recursive bound-and-search algorithm to tune configurations. BestConfig was used to tune 30 Spark configuration parameters using 500 execution samples and achieved 80% runtime performance improvement with respect to the default configuration. Zhao et al. [134] have proposed an Adaptive Serialization Strategy to improve Spark performance. It tunes Spark's serialization strategy dynamically based on runtime statistics. The work in this dissertation addresses tuning a large set of Spark configuration parameters, not just the serialization strategy.

MEMTUNE [126] is a memory manager for in-memory data analytics. It dynamically tunes the memory allocated for computing and caching based on workload memory demand. Likewise, ATuMm [66] enables dynamic memory allocation for Spark while taking into consideration both memory demands and the latency caused by garbage collection.

PETS [93] uses a resource bottleneck-aware approach to iteratively tune parameter ensembles. Gu et al. [53] proposed a neural network based tuning approach, wherein a neural network model is built to predict the next step to take during the configuration search (i.e. each configuration parameter increase/decrease). It also utilizes a random forest prediction model to predict the performance under the suggested configuration. It tunes 15 parameters and outperforms the default configuration by 42.8%. ATCS [82] have proposed a Generative Adversarial Nets (GAN)-based tuning approach, which builds a performance prediction model using less data without degrading the accuracy using GANs. The constructed model is then leveraged to search for good configuration using an optimized genetic algorithm. A+Tuning [120] proposed to tune the architecture and application parameters to optimize energy consumption. It starts with profiling the applications and clustering them based on their resource usage characteristics (e.g., compute-bound, memory-bound), then co-locate the applications and leverage a machine learning model to tune the architecture and applications parameters of the co-located applications. ReB [60] proposed balancing the resource allocation of Spark jobs based on predicting Job Completion Time (JCT). It employs a prediction model to estimate the JCT then leverages this model to minimize the JCT of multiple Spark jobs. This results in an average JCT decrease

by 10%-30% compared to existing solutions. Ousterhout et al. [92] highlight the effects of resource bottlenecks in Spark, but do not address the autotuning problem.

Some work has been proposed to study Spark performance on HPC systems [111, 122, 123], tuning a few selected Spark parameters. There are also some developer guides [5, 71] that document an explanation of each Spark parameter and guide end-users to the impact of tuning them. Nevertheless, those guides do not address the automated tuning of Spark parameters and the challenge of manually tuning them remains a burden on the end-user.

In a relevant work, Bilal et al. [27] have proposed a framework to tune the configuration of stream processing systems such as Storm. The tuning takes place using a modified hill-climbing search with heuristic sampling inspired by Latin Hypercube.

Unlike this earlier work, the proposed work in this dissertation aims to maintain a significantly lower tuning cost, making dynamic tuning in a rapidly changing environment possible.

3.2 Configuration Tuning for Cloud Computing

Some other systems have looked at tuning cloud configurations (selecting cloud instances, the number of instances, number of cores). Cherrypick [20] finds near-optimal cloud configurations iteratively. It leverages GP to build a performance model that allows picking good configurations using a small number of samples. This work suits well the low dimensional space of cloud configurations, but is harder to apply directly to the higher-dimensionality search space such as Spark. Similarly, Arrow [58] and Scout [59] employed GP to find near-optimal configurations while augmenting the GP with performance metrics [58] to improve the accuracy of the GP constructed performance model, or historical information to explore the search space more effectively [59]. Ernest [115] efficiently builds a performance model by utilizing the internal structure of the machine learning jobs. The model predicts workload performance under different VM configurations (e.g., number of cores, memory size, number of VM instances). However, the constructed model is tied to a specific VM type. PARIS [127] is a system for selecting the best VM for certain workloads based on user defined metrics. It uses offline profiling for benchmarking various VM types, then combines this with an online fingerprint of each workload. The combined data is used to build a decision tree and a random forest-based model that predicts performance and cost of workloads on different VM types.

Selecta [73] finds near-optimal configurations for cloud compute and storage of the analytics workloads. It employs latent factor collaborative filtering to predict the application's performance under different configurations. Lynceus [31] tunes both cloud and application (e.g., hyperparameters of machine learning algorithms) related configuration parameters. It reduces the tuning cost by employing a time-out approach that aborts the less promising configurations. Besides, it employs a budget-aware tuning technique that decides which configuration to try next. The decision is taken based on predicting the impact of exploring a candidate configuration.

Those systems are designed to address the low dimensionality of the cloud configuration parameters and hard to adopt for the high dimensionality of Spark. To demonstrate that, I compare Cherrypick for tuning Spark configurations against Tuneful (Chapter 5). Tuneful can work together with those CC tuning systems to enable optimization at the DISC system level after finding the best cloud configuration.

3.3 Hyperparameter Optimization (HPO)

HPO is the process of finding the right hyperparameters for a learning algorithm to achieve a certain objective (e.g., better model accuracy and/or learning speed). A hyperparameter is a parameter used to control the learning process during the training (e.g., learning rate in gradient descent or batch size in neural networks).

One of the widely used approaches for hyperparameters optimization is the manual tuning based on the user's experience. However, this approach is tedious to apply given the growing complexity of machine learning models. Further, it yields a workable solution but does not guarantee optimal tuning. Therefore, automating the optimization of machine learning models through hyperparameter tuning gained wide traction over the last two decades.

Historically, the work on hyperparameter tuning started with employing grid search in 1994 [63], which specifies a grid of each combination of algorithm parameters. However, the growth in the data (i.e the training data) and parameters dimensionality made exhaustive grid search not a viable solution for HPO. Therefore, random search and adaptive optimization methods such as pattern search [90] have been employed, leading to faster tuning compared to grid search.

In 2004, evolutionary algorithms such as genetic algorithms have been employed in the hyperparameter optimization of SVM [34] and neural networks [101]. As the size of training data continued to grow exponentially, the application of data-efficient hyperparameter optimization approaches has become urging. Consequently, HPO using BO gained a wide adoption. It has been employed to tune two hyperparameters of SVM, outperforming grid search by a factor of 10 in terms of the tuning speed [49]. Then BO has been utilized to tune the hyperparameters of deep neural networks [75, 105]. Similar in spirit, recent work has been proposed not only to tune the hyperparameters of the deep neural network but also to search for the best network architecture (i.e number of layers, neurons per layer) [23, 85, 94]. The proposed work revolves around either predicting the performance of a given architecture [23], searching for the best architecture by growing the architecture's complexity iteratively [85], or sharing weights between architectures [94].

In 2009, the problem of HPO has been reshaped to include both algorithm selection and HPO [43], which addresses all the decisions across a machine learning pipeline, starting from selecting a preprocessing algorithm, feature selection algorithm, classifier/regressor to tuning their associated hyperparameters. For example, Flash [133] proposes a two-layer BO frame-

work for algorithm selection and hyperparameter optimization. The first layer employs a linear model to quickly explore the learning algorithms space, then the second layer applies nonparametric tuning of the hyperparameters.

Starting from 2013, existing machine learning frameworks have been extended to facilitate picking the right algorithm and hyperparameters for end-users [110, 47, 70]. More recently, some work has been proposed to speed up HPO through distributed tuning [74, 19], allowing to try multiple candidate configurations simultaneously.

Overall, some of these techniques are common across other configuration tuning systems in different domains (e.g., BO and genetic algorithm based tuners). The cost-effective tuning approaches proposed in this dissertation would be of great value if successfully applied in HPO. This is because of the rapidly growing number of configuration parameters, demanding cost of each workload execution sample and the dynamic nature of the running workloads. Experimenting with the tuning strategies proposed in this dissertation for HPO is an interesting future avenue.

3.4 Configuration Tuning for Database Management Systems (DBMS)

On the DBMS front, iTuned [42] has been proposed to tune SQL database systems. It employs GP to build a performance prediction model that enables automatic configuration tuning. Similarly, OtterTune [114] is a recent work that leverages GP. It also performs a feature selection step in which it reduces the number of parameters. This step finds the most influential configuration parameters (system-wide) to mitigate the complexity of the configuration tuning problem. However, it requires hundreds of trials in an offline phase to define a system-wide significant parameters for the DBMS workloads. Likewise, Rafiki [88] performs a similar step to define system-wide significant parameters. Then builds a neural network performance model using those parameters and tune them using genetic algorithms.

More recently, SageDB [76] has been proposed as a new vision for data processing systems, which departs from processing the data in a general purpose approach to a specialized one that takes advantage of the characteristics of a particular database, query workload and execution environment. This is done through modeling data distribution, workload and hardware characteristics. Then learning the best ways to structure the data, access data efficiently and plan query execution.

Generally, those proposed tuning solutions proved to be effective in the DBMS domain. However, applying such techniques directly in DISC systems is not only expensive, requiring hundreds of executions to define a system-wide set of significant parameters, but also impractical given the higher diversity of analytics workloads (e.g., graph analytics, machine learning, text analysis and SQL based analytics)—with the significant parameters possibly varying from workload to another.

3.5 Generic configuration Tuners

Generic autotuners are general-purpose tuning systems in which the end-user defines the configuration parameters, their types and range of values. The end-user also defines an objective function that the tuning systems aim to minimize/maximize (e.g., execution time, energy consumption).

OpenTuner [21] is a general tuning system that uses ensembles of search techniques such as hill climbing, differential evolution, particle swarm optimization and pattern search. OpenTuner evaluates which techniques perform well over a window of time and picks them more frequently than the ones that have poor performance (those can even get disabled).

Spearmint [105] is a Python BO library that employs GP prior model to tune a user-defined configuration parameters with respect to a user-defined objective function. Similarly, SMAC [64] is a BO configuration tuning system that employs a random forest prior model to tune user-defined configuration parameters.

More recently, BestConfig [135] has been proposed as a generic autotuning system that leverages a divide-and-diverge sampling method and a recursive bound-and-search algorithm to tune configurations. BOAT [38] a BO based autotuner that enables developers to provide contextual information, in the form of a bespoke probabilistic model of the workload's behaviour, to accelerate tuning convergence. Since it is tedious to define those probabilistic models for non expert users, the work in this dissertation aims at minimizing user's intervention. FLASH [91] is a generic autotuning system that starts with 30-50 execution samples to build a decision tree model then incrementally execute more samples till it converges.

Overall, those systems are either tedious to specialize to enable faster convergence to near-optimal configuration [38], or hard to converge to near-optimal configuration quickly while keeping generic [21, 135]. The proposed tuning approach in the dissertation will be comparatively evaluated against one of those generic autotuners (Opentuner in particular since it covers a wide array of search algorithms).

3.6 Similarity-aware configuration Tuning

In practice, detecting similarity across workloads has a significant potential for accelerating the configuration tuning. However, as a prerequisite step, it is necessary to characterize the different workloads, wherein the main components distinguishing each workload are identified. The aim of workload characterization is either detecting similarity across workloads [18], coming up with representative benchmarks of workloads [68], or revealing existing performance issues [35].

Historically, research into workload characterization started four decades ago [46] to help selecting representative workloads during computer performance measurement, then kept evolving to cope with advances in various domains that ranges from web workloads, social network workloads, mobile services, and cloud services [30]. Generally speaking, workload charac-

terization usually involves two fundamental steps: first, collecting measurements of various execution features describing how a workload is executed (e.g., CPU utilization, memory footprint). Then measurement analysis which aims at summarizing the patterns of how various workloads are executed (e.g., I/O intensive, CPU intensive). This is usually done through applying statistical approaches to represent the properties of each collected feature (e.g., mean, percentile, variance), then employing multivariate analysis approaches to analyze each feature in a multidimensional space of features (e.g., principal component analysis, factor analysis, clustering). This analysis enables summarizing the overall properties of a workload. Lastly, visualization approaches can be employed to uncover insights across workloads (e.g., scatter plots, dendrograms) [30].

Recently, a relatively few work has been proposed to characterize and cluster Hadoop and Spark analytics workloads, with the majority of work employing principal Component Analysis (PCA) to identify the most important metrics to lower the dimensionality of the captured execution metrics [68]. Besides finding the important execution metrics, some work has been proposed to cluster the workloads using KMeans, grouping the various workloads based on their execution patterns [68, 18].

Jia et al. [67] characterize data analytics workloads (most of them are Hadoop based workloads) in data centers based on their microarchitectural characteristics (e.g., L1 instruction cache misses per thousand instructions). They concluded that data analytics workloads have inherent different characteristics from traditional workloads (e.g., High Performance Computing (HPC) workloads). Similarly, some work has been proposed characterizing in-memory analytics workloads [69, 22]. Jiang et al. [69] characterize Spark workloads and compare them against Hadoop and traditional HPC workloads. Their work shed light on the significant differences in Spark in terms of memory utilization, memory access and disk I/O frequency. Awan et al. [22] highlight thread scalability issues in Spark (i.e., Spark does not scale linearly when more than twelve threads are running). Chiba et al. [35] characterize the memory, network, Java Virtual Machine (JVM), and Garbage Collection (GC) usage to tune the performance TPC-H workloads on Spark.

On top of workload characterization, some configuration tuning systems have leveraged similarity to accelerate the tuning process: AROMA [78] clusters the Hadoop workloads then builds a performance model that guides the tuning of each workload cluster. Scout [59] explores the search space more effectively for cloud configuration tuning. Lastly, OtterTune [114] guides the tuning of similar workloads in DBMS.

3.7 Summary

This chapter studied the work addressing configuration tuning in different domains that range from DISC systems, cloud computing, machine learning systems, database management systems to generic tuning systems.

As Table 3.1 showed, none of the existing work addresses dynamic high-dimensional configuration tuning—keeping in mind the dynamic environment of the analytics workloads and the need for frequent retuning. This dissertation tackles filling this gap to enable efficient tuning of the analytics workloads in a rapidly changing environment.

Chapter 4

Incremental Configuration Tuning Framework

Automatic configuration tuning has been based on developing strategies for the exploration of the configuration space, building on search-based techniques such as hill climbing [81] and genetic algorithms [83], or model-based techniques that explore hundreds of configurations to build a prediction model of workload performance under a given configuration.

Existing solutions incur expensive tuning cost since they require numerous execution samples. This is impractical for workloads that take long time to execute (e.g., big data analytics workload) or change over time (e.g., due to change in the environment or growth in the volume of data the workload receives) and might require re-tuning. This is because getting optimal configurations through re-tuning requires an investment in time and resources comparable to the initial tuning costs.

This chapter brings into focus these trade-offs and how they should be considered when performing tuning without assuming a static environment or workload. It further sheds light on a set of overlooked issues in the existing solutions that pose a challenge for data analytics configuration tuning in a dynamic environment. Ultimately, I propose Tuneful, a framework that makes tuning in those scenarios possible. Tuneful does not require an *expensive* offline phase to build performance prediction model and reaches close-to-optimal configurations significantly faster than existing incremental search strategies. It does this by running the actual workloads—instead of relying on predicted execution cost using offline model—making better decisions about which configuration to explore next. This is done online, in the context of workloads that are run periodically. Each execution is used to: First, pick a configuration that helps exploring workload-specific influential parameters. Then, once the influential parameters are known, select a configuration that has the maximum probability to minimize a cost function (e.g. runtime or actual costs). It is designed from the start to perform *incremental* optimizations, enabling a cost-effective re-tuning across a diverse set of workloads.
The key contributions of this chapter are:

- Show that configuration tuning of data analytics need to be addressed differently to enable cost-effective tuning. This is done through a series of experiments that: 1) Bring into focus the trade-offs that need to be considered when performing tuning without assuming a static environment or workload. 2) Shed light on a set of overlooked issues that pose a challenge for cost-effective tuning of data analytics workloads.
- Propose the first work to address cost-effective tuning of high dimensional configurations in a dynamic environment, Tuneful, a comprehensive incremental configuration tuning framework that overcomes those overlooked issues and enables cost-effective configuration tuning.

I demonstrate the applicability of the proposed framework through its implementation as an open-source extension to the Spark framework, directly usable within existing PaaS deployments. Tuneful optimizes configuration parameters without requiring any user input or changes to Spark. The source code is available online under the Apache 2.0 open-source license at [15]. This work has been published in the applied data science track of ACM knowledge discovery and data mining conference (KDD'20) [44].

4.1 Amortization of tuning cost

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 (§ 4.2.3 shows a case study demonstrating that). Here, the window to get benefits from optimization depends on how quickly the configuration tuning takes place, the workload execution frequency, the data growth rate and how quickly an existing performance model becomes obsolete.

Ass an illustrative toy example of how tuning cost amortization works, Figure 4.1 shows the amortization of tuning costs over the number of workload executions. This compares the

[1]This figure is a result of collaboration with Lucian Carata

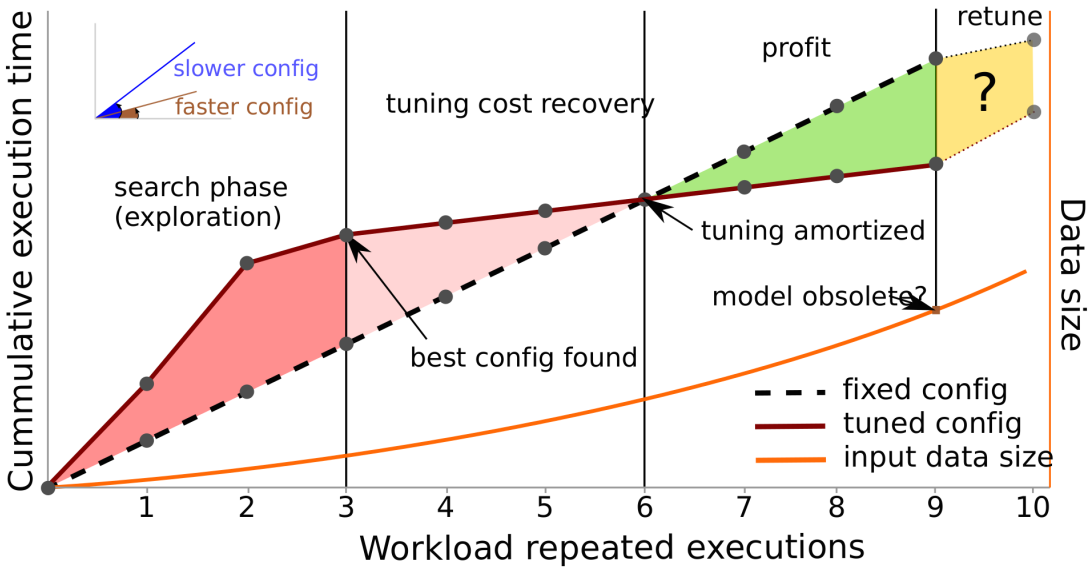


Figure 4.1: The amortization of tuning cost over workload executions [1]

cumulative execution time of running with a fixed configuration to the one obtained through incremental tuning. Normally, 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 point at which this change happens represent a break-even deadline (longest possible time for the tuning cost recovery phase while not losing money), after d executions. In Figure 4.1, assuming $d = 9$. It is crucial for the tuning costs to be recovered before execution number 9. The faster the tuning costs are recovered, the higher the tuning profit.

4.2 Configuration Tuning Challenges

This section demonstrates a set of issues that shape a challenge for cost-effective tuning of data analytics configuration. Those challenges are not solved yet by existing solutions, firmly showing that configuration tuning of data analytics needs to be addressed differently.

4.2.1 Finding best estimated "ground truth" for the optimum configuration:

Given the high dimensionality of the configuration parameters and the demanding cost to run a workload execution sample, it is impossible to exhaustively search for the optimum configuration. For example, considering tuning the configuration parameters listed in Table 4.3, there will be approximately $2 \cdot 10^{41}$ configurations possible in total (this represents the size of the search space). Nevertheless, it is essential to have a good estimate of the optimum con-

#	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)	[3,43]	1	The memory size of each Spark executor
3	spark.executor.instances	[5,80]	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 cetrain 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

Table 4.1: System-wide set of configuration parameters

figuration, since it works as a reference the one uses to compare against. To achieve this, I employ an extensive exploratory approach using Random search with the configurations being generated using low-discrepancy sequences [106], since they cover the search space quicker and more evenly than the standard random numbers[2]. This enables exploring the search space with a smaller number of execution samples. A budget of 100 execution samples is used to limit the cost of the experiments. In our experiment, some workloads take more than 2 hours to execute once, which means more than 8 days of processing to find an estimate of the ground truth for a single workload, let alone the need to experiment with workloads of evolving input sizes running across different clusters. I have also experimented with allocating larger search space exploration budgets (more execution samples) to other methods (e.g., OpenTuner [21] and Gunther [83]) but the exploratory approach with low-discrepancy sequences guaranteed finding comparable configurations using lower number of executions. § 5.3.3 will also show how when the same budget (execution samples) is allocated to those methods, sampling using low-discrepancy sequences guarantees finding the best estimated optimal.

[2]The Random search approach used throughout the dissertation always generate the configuration using low-discrepancy sequences, a decision taken to make sure the search space is widely covered.

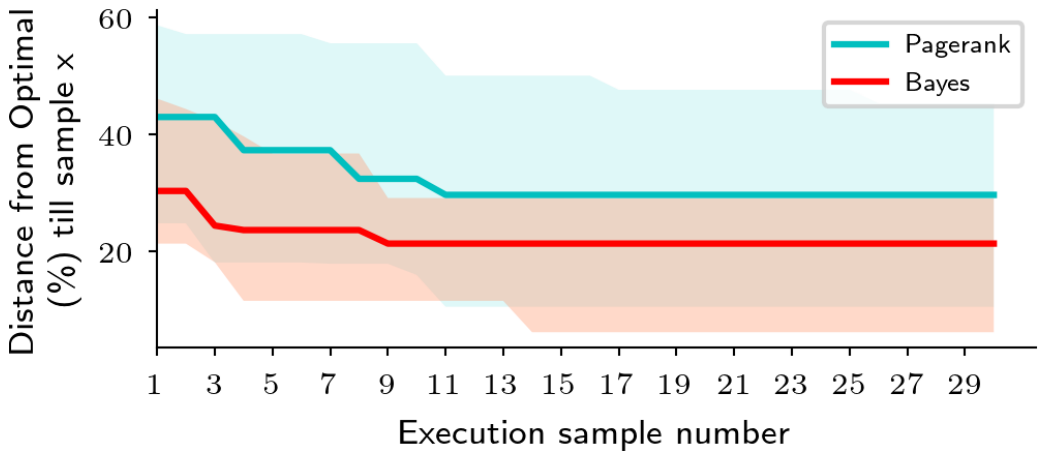


Figure 4.2: The percentile-based interval of each execution sample using Bayesian Optimization with system-wide set of configuration parameters.

4.2.2 High dimensionality and significant number of execution samples

The search space for good configurations grows exponentially with the number of configuration parameters. Spark [5] has over 180 configuration parameters. Not all of them impact performance, but the relevant ones are still of high dimensionality and with large intervals of possible values. Complex data processing frameworks such as Spark commonly have 20 - 60 parameters that are relevant for tuning. In practice, tuning the configuration using evolutionary or hill climbing approaches [83, 21] requires hundreds of execution samples, representing thousands of hours of computational power and associated monetary cost. It is therefore crucial to minimize the number of configurations evaluated to reach an optimal configuration. BO strategies have been applied recently as a solution to this problem. However, BO does not provide quick convergence in high dimensional configuration spaces (more than 10 parameters) [103].

Figure 4.2 demonstrates this slow convergence when tuning a set of 10 performance-related configuration parameters (listed in Table 4.1) using BO, the lines show the median and the shaded area represents the space between the 10th and 90th percentile of 10 experiments. At the median, the BO finds a configuration within 30% and 20% of the estimated optimal configuration for Pagerank and Bayesian workloads respectively, while finding a configuration of up 45% and 30% distance from the estimated optimal at the 90th percentile. The estimated optimal configuration is found as illustrated in § 4.2.1. A Google Compute Platform (GCP) cluster of 6 *n1-standard-16* nodes is used to run this experiment.

This experiment supports existing reports in the literature showing BO converges relatively slowly when tasked with approximating functions in a high dimensional space [112]. This is because the BO relies on the Euclidean distance to define input space correlations. Euclidean distance becomes less informative as the input space dimensionality increases [26] and the number of samples required to learn the model grows exponentially—a phenomena coined as the *curse of dimensionality* [25]. Withe Complex data processing frameworks such as Spark commonly have 20 - 60 parameters that are relevant for tuning, this means that employing the

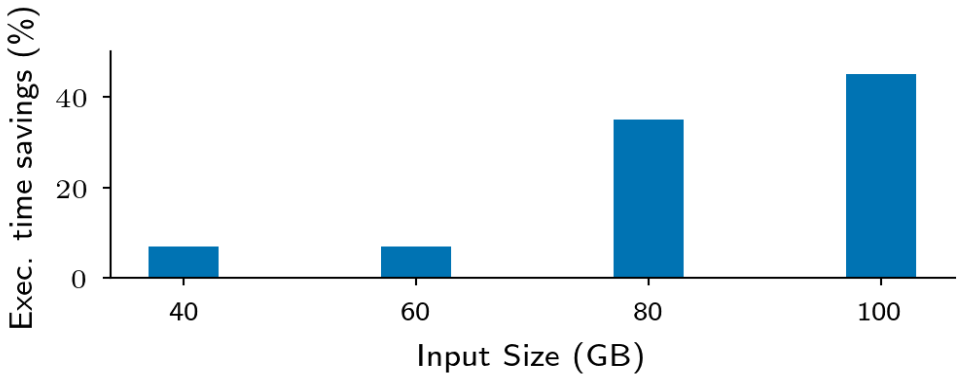


Figure 4.3: Execution time savings when retuning the configuration for each input size, compared against reusing the best configuration found for input size 20GB (TPCH benchmark))

standard BO in isolation is not enough, the percentile-based interval of each execution sample in Figure 4.2 is too wide to be acceptable in practice. This will ultimately yield the exploration cost incurred by the execution samples to outweigh the gains arising from the tuning.

4.2.3 Frequent workload changes

Most of the existing configuration tuning work assumes either static or inexpensive workloads running in a fixed environment. Paradoxically, the analytics workloads upon which they are evaluated have different characteristics: they are expensive to run, can receive evolving inputs, run in a dynamic environment and are susceptible to frequent changes. A change in a workload takes place either through modifying the execution logic or through a significant change in the processed data size (e.g., new records need analysis) or the physical underlying environment. While the former is rare in a deployed system, the later is common in the big data world. For example, let's consider a TPCH workload, initially processing a dataset of 20GB of data. Running once daily and receiving roughly 1GB of additional data at each run, the dataset grows to 80GB in 2 months (60 executions of the workload). Such changes can lead to significant changes in the workload's behaviour under particular configurations.

Figure 4.3 shows that the potential savings from retuning can be significant, evaluated as shown in Equation 4.1, where $runtime_{reusedConfig}$ represents the runtime of the workload with the reused configuration of the initial input size (20GB), $runtime_{retuned}$ represents the runtime of the workload when the configuration is retuned for the evolving input size.

$$savings = ((runtime_{reusedConfig} - runtime_{retuned})/runtime_{reusedConfig}) * 100 \quad (4.1)$$

It demonstrates tuning TPCH workloads of evolving input sizes as illustrated in § 4.2.1. The configuration parameters under tuning are listed in Table 4.3 (expanding the previous list of considered parameters in Table 4.1 towards a more comprehensive list, the configurations' value range also may vary since a different cluster is used to run this experiment—an Amazon Web Services (AWS) cluster of 4 *h1.4xlarge* instances). Those parameters are selected as they rep-

resent a superset of the ones used in related work [135, 129]). Initially, a TPCH workload of scale factor 20 (20GB of compressed input) is tuned. Then the scale factor is incremented to represent an extra data received for processing and configurations are tuned again for each input size in Figure 4.3. The aim of this experiment is to compare the amount of potential savings of retuning the configuration for the increased input sizes against reusing the original configuration of the 20 scale factor workload. As shown in the figure, reusing old configuration does not guarantee near optimal tuning and the potential savings of retuning the configuration can be significant, reaching up to 45%.

The configuration should therefore be re-tuned accordingly. While the decision about *when* to re-tune is independent from the tuning strategy, its *feasibility* relies on how expensive is to re-run tuning in the setting of gradually changing workloads. Existing approaches require a significant number of executions for finding close-to-optimal solutions. In particular, the number of executions required generally exceeds the number of times the workload might run before re-tuning is necessary, making such solutions impractical. For example, the `BestConfig` [135] system requires 100-500 samples to identify a good configuration, exceeding the 60 "normal" runs of our TPCH example workload over the 2 months period. This dissertation focuses on developing strategies that are able to identify good configurations using a significantly smaller number of samples.

4.2.4 One model does not fit all

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. With this "*one model fits all*", amortization of tuning costs 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 [129]; 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 [20]; 4) the high dimensionality of the search space: one dimension per configuration parameter.

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. 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).

I demonstrate this with three workloads from Hibench [62]: Pagerank and two Bayesian classifiers with different input sizes, each workload is executed 100 times using random configurations for 30 parameters. Table 4.3 describes each configuration parameter and its range.

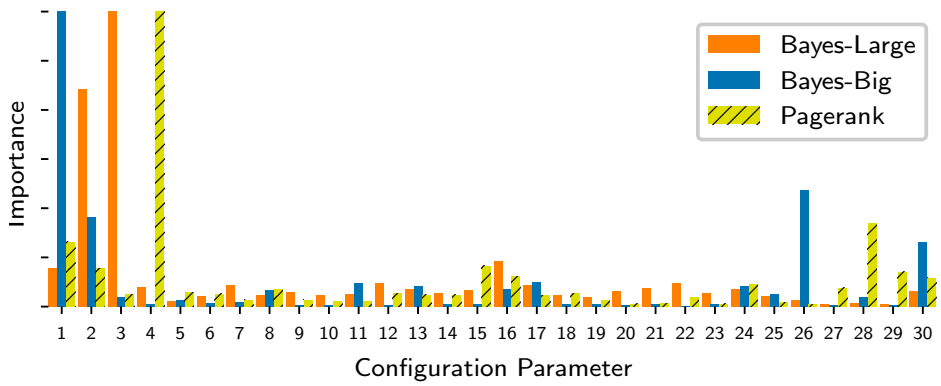


Figure 4.4: Relative configuration parameter importance for 3 Hibench workloads, showing that the parameters which affect runtime the most are workload-specific.

Application	Input Size	Dataproc exec. time (m)	Best exec. time (m)	Potential savings
Pagerank	10m pages	22	10	54%
TPC-H	40GB	28	10	63%
Terasort	40GB	14	10	29%
Bayes Classifier	20m pages	41	23	43%

Table 4.2: The execution time of the configuration suggested by Dataproc against the best configuration found by random search.

Figure 4.4 depicts the significant configuration parameters for the three workloads. The calculation of the significance is based on the contribution of each configuration parameter in predicting the execution time, as determined from the 100 executions of each workload. More details on the importance calculation of each parameter is given in § 5.1.1. As expected, for each of these workloads, the set of significant configuration parameters is different. Even for the same application, different input data sizes (Bayes-large, Bayes-Big) have different important parameters. This suggests that even parameters' significance need to take palace in a workload-specific approach rather than learning system-wide significant parameters.

4.2.5 Cloud provider's workload-agnostic tuning

Cloud providers such as Amazon and Google offer Platform as a service (PaaS) Spark services [11, 12], the configurations in such services are generically-tuned taking only into account available cluster resources. Table 4.2 shows the difference in execution time between the configuration suggested by Dataproc [12][3] and the best configuration that can be found as described in § 4.2.1. A GCP cluster of 4 *n1-standard-16* nodes is used to run this experiment. As shown in the table, in the typical case, the suggested configurations by cloud provider yield execution times 29–63% longer than the optimum for a given workload. The configurations provided 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. Obviously there will be workloads with similar characteristics. To exploit this one needs a concept of workload

[3]Spark as a service provided by GCP [77]

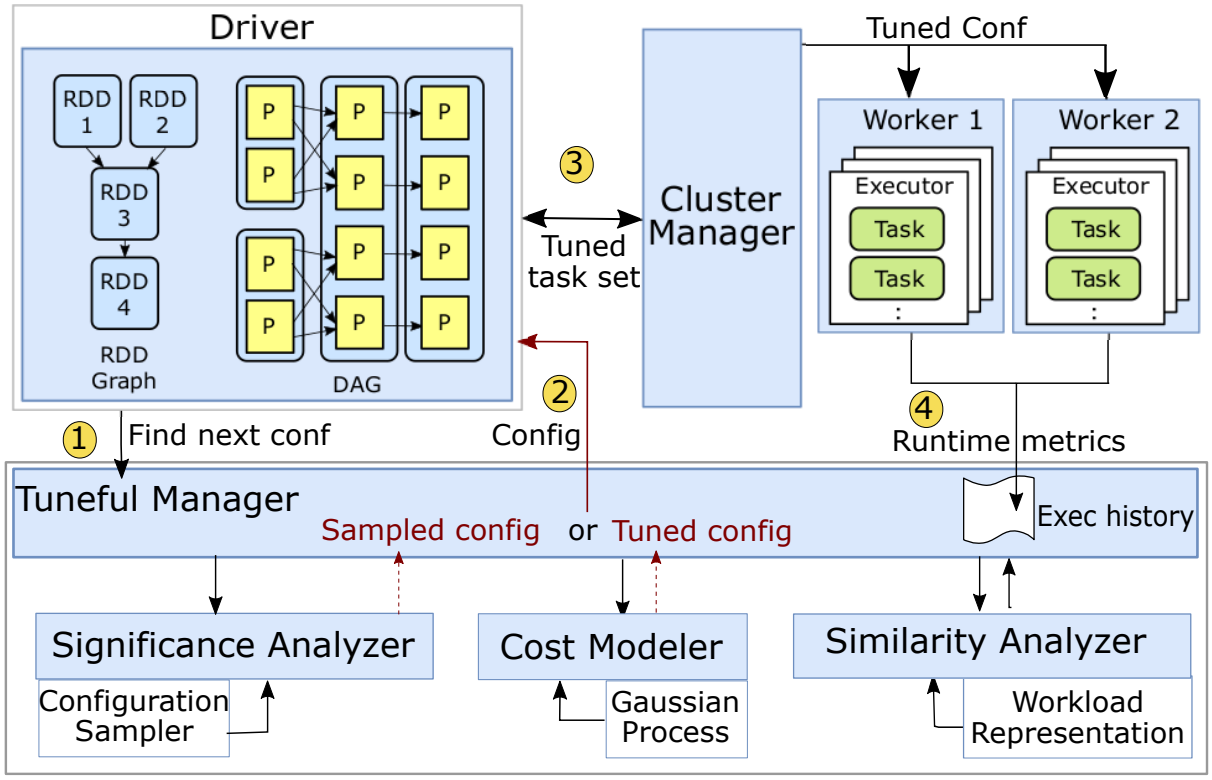


Figure 4.5: Tuneful–Spark integration: (1) On workload submission, the Driver requests the next configuration from Tuneful. (4) After workload execution, metrics are collected and used to update existing optimization models.

similarity, which is developed in Chapter 6.

4.2.6 The need for incremental tuning

The idea of workload and data-aware tuning is further strengthened if a dynamic context is considered, 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 resources are spent for updating the model. For example, DAC [129] takes 92 hours to collect data for KMeans clustering workloads of evolving input sizes. This data is used to build an offline data-size aware prediction model that works for a *given* cluster. This model is then used to perform configuration tuning for KMeans workloads with various input sizes. However, a change in the cluster resources will degrade this model accuracy and imply recollecting more data to maintain the model's accuracy, incurring high tuning costs in a rapidly changing environment.

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.

4.3 Incremental Configuration Tuning Framework

The challenges demonstrated above firmly show that configuration tuning of data analytics should be addressed differently. Taking into consideration the *diverse* workloads, high dimensionality of configuration parameters, dynamic workload environment and importantly the amortization of tuning cost and how this should be accelerated. I advocate that tuning should follow an incremental workload-specific approach to address these issues. Tuneful is a tuning framework which makes incremental tuning a viable proposition. It aims to use information from each execution sample efficiently, in order to minimize the time spent in significant parameters exploration and tuning, even when considering evolving workloads.

4.3.1 Overview

The Tuneful framework is designed to work efficiently both initially (when zero/ little data is available) as well as later (as more tuning knowledge is acquired).

It 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 Figure 4.5. It consists of three main components controlled by the Tuneful Manager: Significance Analyzer, Cost Modeler and Similarity Analyzer. 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, the 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 Figure 4.5, 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 demonstrated in Chapter 7.

I illustrate the zero-knowledge tuning approach of the framework, namely, significant parameters exploration and tuning in Chapter 5—as a use case of a new cluster receiving workloads without having any prior tuning knowledge. Then Chapter 6 covers how to characterize workload similarity and perform similarity-aware tuning (Tuneful's extended knowledge tuning mode).

4.4 Summary

The chapter argued that configuration tuning should be addressed differently. This is done through bringing into focus the trade-offs that need to be considered when performing tuning without assuming a static environment or workload, demonstrating a set of overlooked issues that pose a challenge for practical configuration tuning for data analytics: the high dimensionality of configuration parameters and the large search space, the dynamic environment of the analytics workloads, and the diversity of the analytics workloads and lastly advocating that configuration tuning should take place in an incremental workload-specific approach, proposing Tuneful, a comprehensive framework for efficient high-dimensional configuration tuning. In the next chapters, I present Tuneful's high-dimensional significance-aware tuning approach (Chapter 5). Then, explore how to leverage the knowledge acquired during tuning a workload to accelerate another similar workload. To answer this question, as a first step, I start with defining similarity in this context through: studying how to represent a workload based on its execution metrics and how to leverage the tuning knowledge across similar workloads(Chapter 6). Lastly, Chapter 7 evaluates the benefits of similarity-aware tuning.

#	Configuration name	Range	Default	Description
1	spark.executor.cores	[3,16]	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	[19,304]	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 cetrain 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 exection 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 broadcatsed 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 4.3: Configuration parameters under tuning

Chapter 5

High Dimensional Significance-aware configuration Tuning

This chapter illustrates how Tuneful employs a significance-aware configuration tuning approach in high dimensional configuration space. It is the first work to propose a *data-efficient* [40] tuning of high-dimensionality configurations. It is designed to perform *incremental* optimizations and make retuning cost-effective, requiring 62% less search time at the median and 97% less in the best-case, while finding configurations with similar runtimes when compared to current state-of-the-art approaches. Tuneful achieves this by leveraging incremental SA to identify the influential configuration parameters in a high-dimensional space and BO using GP for efficient tuning of those parameters.

The key contributions of this chapter are:

- Show how data-efficient machine learning techniques can be leveraged to find *workload-specific* influential parameters incrementally using a small number of workload executions, within a high dimensional configuration space.
- Demonstrate how applying BO to tune the influential configuration parameters incrementally widen the types of workloads that can be tuned in a cost-effective manner.
- Show that successfully applying data-efficient learning enables obtaining comparable or better configurations than prior work but converging to those significantly faster.

5.1 Approach

5.1.1 Identifying Significant Parameters

Virtually all complex systems have numerous configuration parameters that can be set to user-given values. However, only a small subset of those parameters has a significant impact on workload overall performance. Figure 4.4 demonstrates an example of this for a set of workloads, showing that out of the selected 30 Spark configuration parameters only a small fraction influence the execution time significantly. Tuning the others (non-influential parameters) simply wastes resources and does not bring the algorithm closer to the optimal solution.

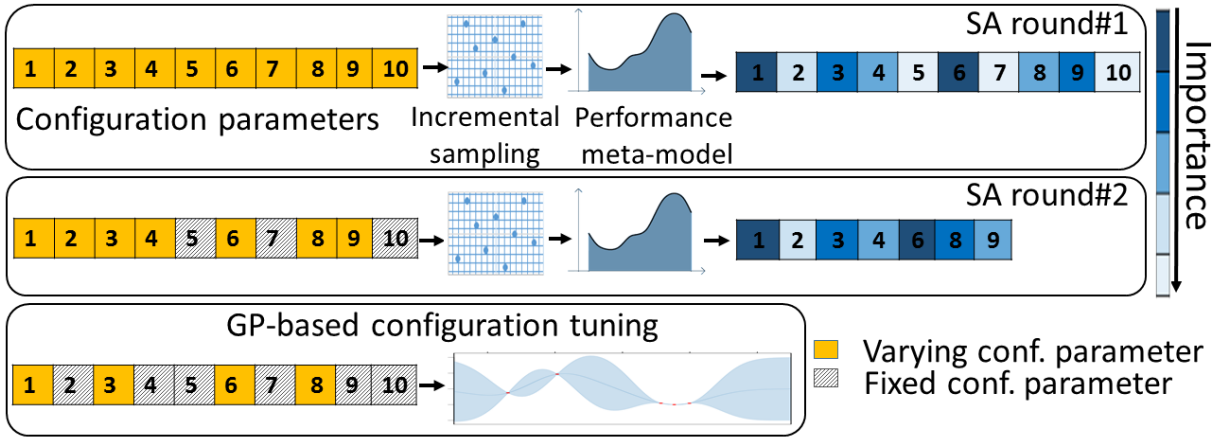


Figure 5.1: Tuneful's incremental algorithm for detecting the significant parameters and tuning them.

A common approach for identifying parameter significance is to perform SA [28]. In our context, SA studies how the variation of the cost function (e.g., execution time) can be attributed to the different configuration parameters.

The SA 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 [88, 114]. However, applying such techniques directly to a DISC system is not only expensive, requiring hundreds of executions, but also impractical given the diversity of workloads.

Tuneful does not seek to identify system-wide influential parameters. Instead, it incrementally identifies *workload-specific* influential parameters using a multi-round SA method that is efficient in our problem domain.

For each workload, Tuneful *incrementally* defines the set of parameters that impact its performance the most. Figure 5.1 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, Tuneful first builds a metamodel for predicting the execution cost of a given configuration. The implementation uses Random Forest Regression (RFR), as it improves the prediction accuracy compared to single learning models [84]. Because it is empirically observed that most parameters will have little effect and only few parameters dominate workload performance, this metamodel does not need to be extremely accurate; even a rough approximation will enable correct detection of important parameters, since they highly dominate the performance and are amenable for detection with an approximate metamodel. However, the metamodel needs to provide estimates over a wide area of the configuration search space. This is why Tuneful will first suggest configurations sampled incrementally using low-discrepancy sequences [106]. Experimentally, I have determined that

the lowest number of executions that provides acceptable accuracy of the RFR model (less than 40% error) across diverse workloads is 10. More details about this design choice can be found in § 5.1.1.1.

Tuneful then calculates the importance of each input configuration parameter in the metamodel based on Gini importance [124]. 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. Across all the forest trees, important features are used more frequently in decision tree splits, so they have higher Gini importance.

This allows selecting the most influential parameters and considers the remaining ones as non-influential. Non-influential parameters are fixed for the remaining SA rounds to the mean of their value range (integer and float parameters) or their default values (boolean and enum parameters); a decision made since the default values are excluded from the values range of some integer/float parameters due to their poor performance (e.g., the default Spark.executor.memory is 1GB). Fixing those non-influential parameters allows a faster detection of the influential ones. Then the next SA round is started to determine the most influential parameters among the ones that can still vary. Tuneful can additionally detect dependencies between parameters by including polynomial features, similar to [114], 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, a feature that represents the product of these parameters' values can be included. If this feature has high importance, then those two dependent parameters will be detected.

The algorithm is described in the rest of this subsection. It is repeatedly called during the SA phase for each execution of a workload. Its 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 (configurable and § 5.1.1.1 provides more details on how it is determined);
- n the number of samples required per SA round;

Conceptually, Tuneful considers three global variables: n_SA_rounds, the number of SA rounds remaining; $n\_executions$, initialized to 0; and P_{fixed}, represents the set of fixed parameters and is 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 Tuneful's SA algorithm, $C(X_i)$ its execution cost, and $M(\mathbf{X}, \mathbf{C})$ the constructed metamodel 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. Alg. 1 shows the steps of the SA algorithm:

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 $n\_executions \leftarrow 0$;

The algorithm is run for each workload execution, until the influential parameters are determined. An incremental sampling is performed at line 1 using low-discrepancy sequences [106], which represent numbers that are evenly distributed in a given space to provide a *quicker* coverage. This quick coverage is crucial in the high-dimensional space as exploring the whole space is not a viable option. Once Tuneful has n samples and their associated $C(X_i)$, a RFR model M is built, approximating the true dependence between C and X (line 5). At line 6, the importance of each input configuration parameter in M is calculated based on Gini importance [124].

Then the highest influence parameters are selected (line 7), a fraction α of the total d parameters, and the remaining ones are considered as non-influential (line 8). Those are fixed for the remaining rounds to the mean of their value range and the next iteration is started to determine the most influential parameters among the ones that can still vary. The algorithm stops after the pre-defined number of SA rounds n_SA_rounds.

5.1.1.1 Design choices:

Tuneful has three internal variables for which I empirically set their values:

- **α, the fraction of configuration parameters retained in each SA round:** I observed the impact of different values of α from 0.1 to 0.9 across different workloads. Table 5.1 illustrates this impact. It shows the S_{error} for the various values of α. S_{error} represents the proportion of the best estimated significant parameters (estimated ground truth of parameter importance foreach workload, found through running known SA algorithms with a large number of sample executions) missing in the output of Alg. 1 (the top 6 important parameters in P_s). More details on S_{error} evaluation is in § 5.3.2. While a very small value for α leads to pruning influential parameters (higher S_{error}), 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. As the table shows, setting α to 0.6 in the SA stage represents a good compromise between the accuracy of detecting the influential parameters and bounding

the number of SA runs.

- n**, the number of samples required per SA round:** I experimented with different values for n. While larger values for n generates a metamodel with lower learning error, setting n to 10 execution samples 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 while limiting the number of execution samples.
- $n\_SA\_\mathbf{rounds}$**, the number of SA rounds:** needs to be selected to provide good guarantees, while minimizing cost. I discuss how it is selected in practice for Tuneful in section § 5.3.2.

With the values of those Tuneful's variables set as specified above, Tuneful is experimented with various workloads running on different clusters (§ 5.3).

Workload\Alpha	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
Bayes	0.92	0.16	0.12	0.12	0.08	**0.04**	0.04	0.04	0.04
Pagerank	0.92	0.92	0.92	0.88	0.88	**0.08**	0.08	0.08	0.04
TPCH	1.	1.	0.2	0.16	0.12	**0.12**	0.12	0.12	0.08
Wordcount	0.2	0.18	0.16	0.16	0.16	**0.16**	0.16	0.12	0.04

Table 5.1: The sensitivity error (S_{error}) over different alpha values.

5.1.2 Configuration Tuning

To perform data-efficient tuning, the aim is to get as close to the optimal configuration as possible, using the minimum number of executions. Tuneful employs BO using GP due to its data-efficient learning performance, which makes it an effective approach for modelling expensive functions [40, 20]. Furthermore, GP is nonparametric, 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.

Problem formulation: The objective function that the BO tries to minimize is defined as the execution time of the workload. Tuneful's proposed approach is extensible and can handle any other objectives such as minimizing energy consumption, cluster utilization or a weighted sum of objectives.

Acquisition function: Tuneful uses the GP EI with a MCMC hyperparameter marginalization algorithm [105] as an acquisition technique. This is an EI based [29] function, which iteratively selects the next configuration sample as one that has the highest potential to minimize the objective function. It has the distinct advantage of not requiring any external tuning of GP hyperparameters unlike other acquisition functions such as GP-UCB [97]. Other acquisition functions are possible, such as random, sequential and PI. EI MCMC is used as it has shown better performance compared to other acquisition functions across a wide array of applications and test cases [105].

Prior and covariance functions: I assume that the multi-dimensional function that maps a set

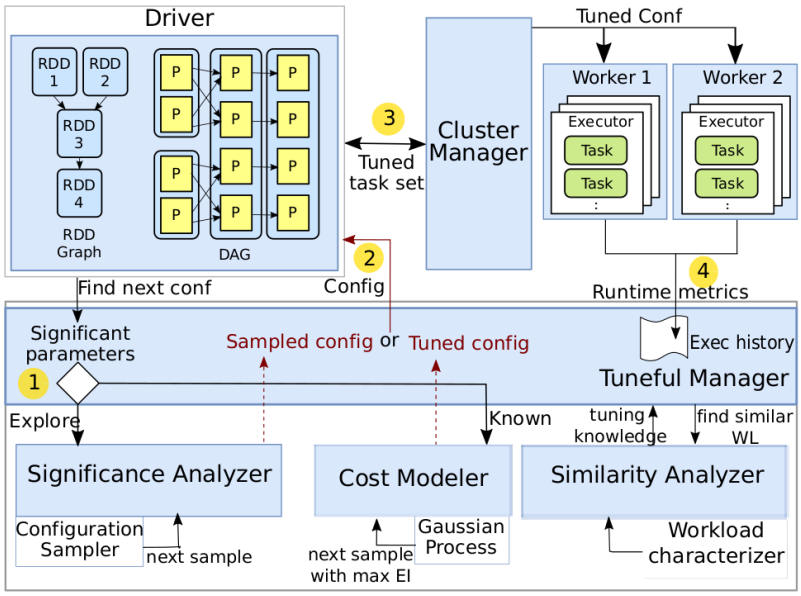


Figure 5.2: Details on how Tuneful performs an incremental significant parameter detection and tuning

of configuration parameter values to the runtime cost can be modelled by a GP (this is a GP prior model). This assumption is made as GP can provide a wide range of flexible nonparametric statistical models over the function space. ARD Matern 5/2 kernel [97] is chosen as the GP covariance function (kernel), because it is able to control how smooth the estimated function is along each of its dimensions independently. This allows Tuneful to learn the objective function quickly and just sample the data that most likely has the minimum objective function, while leaving the other less useful points unexplored. This kernel has been successfully adopted for modelling practical functions [105].

Starting points: The GP cost model is built incrementally, starting with just three samples generated using low-discrepancy sequences [106]. Then the model is improved after each further execution, picking the next candidate configuration that is estimated to reduce the execution time.

Stopping Criteria: The GP modeling stops after suggesting a minimum of n samples (e.g 10 samples) and then after the EI drops below 10%. This decision is made to make sure that the exploration of the tuning space and exploitation of the best configuration found is balanced.

5.2 Implementation

This section shows how Tuneful can be seamlessly integrated into Spark in order to automatically tune configurations. This integration has been tested both in AWS and Google Cloud services, without requiring changes to the Spark binary that is used. Tuneful consists of four components: the Configuration Sampler, the Significance Analyzer, the Cost Modeller and the Tuneful Manager. In the rest of this section, Tuneful's components and their interactions with Spark are described (Figure 5.2).

When a workload execution starts, Spark's driver calls into Tuneful, registered as an exten-

```
$ ./bin/spark-submit
  --jars tuneful-with-dependencies.jar
  --conf spark.extraListeners=TunefulManager
  --class BayesClassifier.java
   bayes-workload.jar
```

Listing 5.1: Tuneful example usage

sion. In response, it returns the configuration that should be used next.

Tuneful Manager: This component first identifies the workload being submitted and retrieves its profile if it exists. During a first phase, configurations are obtained through the Significance Analyzer aiming to identify the significant parameters (as described in § 5.1.1). In a second phase, once those parameters have been identified, the Cost Modeller takes over to minimise the objective function (as described in § 5.1.2). Once a workload execution is finished, relevant performance metrics are provided to Tuneful and used either by the Significance Analyzer or the Cost Modeller. This process is performed automatically every time a workload is scheduled for execution, identifying the significant parameters incrementally in the first phase then optimising the configuration by reducing cost of the objective function iteratively. The practical effectiveness of the approach is demonstrated in § 5.3.

The Tuneful Manager monitors the performance of the workload over time to determine if reconfiguration is necessary. This can be due to a number of factors, such as changes to the size of the data to be analysed, or to the workload logic or changes in the underlying physical architecture of the cloud platform hosting the workload. Once performance degradation is detected, the Tuneful process restarts (i.e. Tuneful suggests a configuration that helps with identifying new significant parameters then tuning them, following Alg. 1 steps again). Currently, this can be detected through monitoring the gap between the predicted execution time by the GP model and the actual execution time. If continuous degradation takes place, then retuning can be triggered.

Significance Analyzer: The Significance Analyzer finds the influential configuration parameters using Alg. 1. It uses the Configuration Sampler to sample values for these potential influential parameters and fixes the non-influential ones. The implementation uses Python3 and the Scikit library [37].

Configuration Sampler: During the phase that explores parameter significance, the Configuration Sampler is used by the Significance Analyzer to sample the configuration values in a manner that accelerates the coverage of the exploration space. It uses low-discrepancy indices, which provide a good coverage of the sampling space [106].

Cost Modeler: The Cost Modeler uses GP optimization to build the configuration cost model of a workload. The model is built incrementally and the GP suggests the next configuration that has high potential to minimize the objective function. To implement this module, I used Spearmint [105], which is a Python BO library.

Example usage: Listing A.1 shows an example of using Tuneful when tuning the Bayes workload. In order to use Tuneful, a Spark user simply adds Tuneful library as a dependency and TunefulManager as an extra Spark listener while submitting his workload to Spark. In other words, Tuneful can run seamlessly on an unmodified Spark infrastructure without complex user intervention.

5.3 Evaluation

Tuneful is evaluated in two stages: first, examining the properties of the algorithm proposed for picking significant configuration parameters; then, examining Tuneful as a whole performing online tuning of typical cloud-computing workloads. For the latter I consider the savings in execution time obtained through tuned configurations and the search time required to get close to the optimal when compared to three state-of-the-art approaches (Opentuner, Gunther and Random search).

5.3.1 Experimental setup

Cluster and configuration specification: A cluster of 20 GCP [77] instances (1 driver + 19 workers) is used, with the driver being an *n1-highmem-8* instance with 8 vCPUs, 52 GB memory and 300GB SSD storage and the 19 workers being *n1-standard-16* instances with 16 vCPUs, 60GB memory and 500GB storage each. The total cluster memory and storage size is 1.2 TB and 5.48 TB respectively. The cluster is over-provisioned so that resources can be restricted through configuration rather than due to physical limitations. A smaller cluster of 4 AWS *h1.4xlarge* instances is also used to validate the robustness of Tuneful in a more resource-constrained environment, but experiments are run on the 20 nodes cluster unless otherwise mentioned. HDFS [13] version 2.7 is used for accessing the shared data and Spark version 2.2.1 as the system under tuning. I tune 30 configuration parameters that cover Spark memory, processing, shuffle and network aspects, with approximately $2 \cdot 10^{41}$ configurations possible in total (this represents the size of the search space). A list of the configuration parameters and their ranges are in Table 4.3[1]. The same ranges are used when evaluating the other tuning approaches.

Applications: 4 diversity-representative workloads are chosen to experiment with the effectiveness of Tuneful in searching for close-to-optimal configurations. The workloads are chosen from the well known big data benchmarks (Hibench [62] and TPC-H [6]) due to their broad real-life applications as described in § 2.1.1:

1. *Bayes* workload with a total Spark executors input of 350GB.
2. *Pagerank* workload with the Hibench-defined *huge* data size is used.
3. *Wordcount* workload that counts word occurrences in 320GB of input.

[1]For the experiments executed on the 20 instances cluster, the range of spark.executor.cores configuration parameter starting from 1 instead of 3, since the cluster is well provisioned already in terms of the number of nodes.

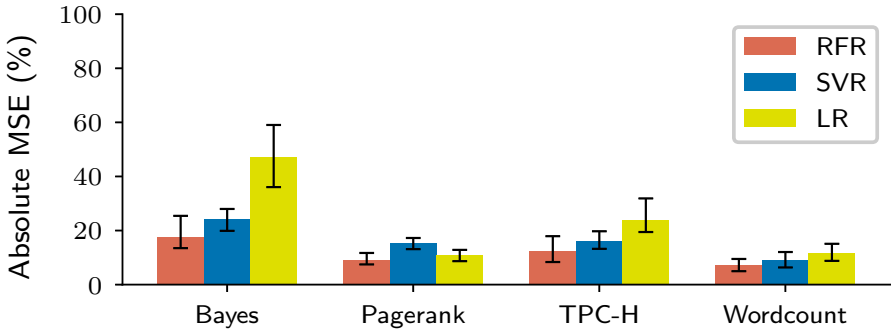


Figure 5.3: Error across the different models when computing influential configuration parameters using a large number of samples (lower is better).

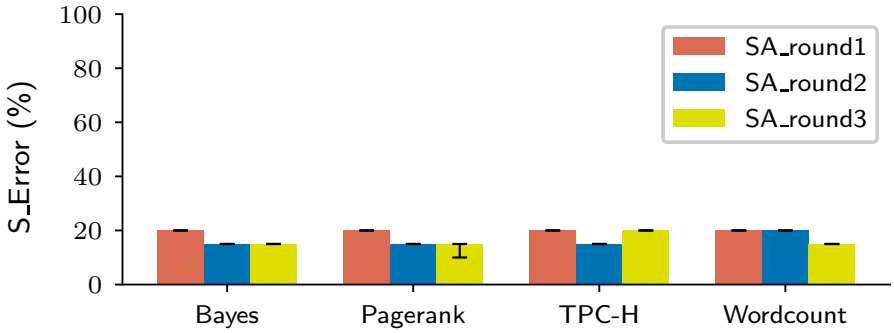


Figure 5.4: Weighted sensitivity error over each of three SA rounds. The error remains low across all workloads (lower is better).

4. *TPC-H* workload that runs 22 decision support SQL queries on data of scale factor 20. This scale is chosen to limit the expenses of our experiments; however, I made sure that this scale is representative enough in terms of input size, with 300GB of total input to Spark executors.

5.3.2 Significant Parameters Exploration

This section evaluates the accuracy with which Tuneful's algorithm detects significant configuration parameters in each SA round. I start by estimating a ground truth parameter importance for each workload, running known SA algorithms (requiring a large number of sample executions). This is then compared with the output of our algorithm (identification from a small number of executions).

5.3.2.1 Estimated significant configuration parameters

Each workload is run 100 times with different configurations sampled using low-discrepancy indices, then a RFR execution time prediction model is built. From this, the most significant configuration parameters are selected using Recursive Feature Elimination (RFE) [55]. To make sure that the built model has an acceptable prediction error, 20% of the samples are used as the test dataset and excluded from the training data when building the model. Figure 5.3 demonstrates the accuracy of the RFR model compared to other other strategies, namely, Support Vector Regression (SVR) and Linear Regression (LR). The y axis shows the absolute mean

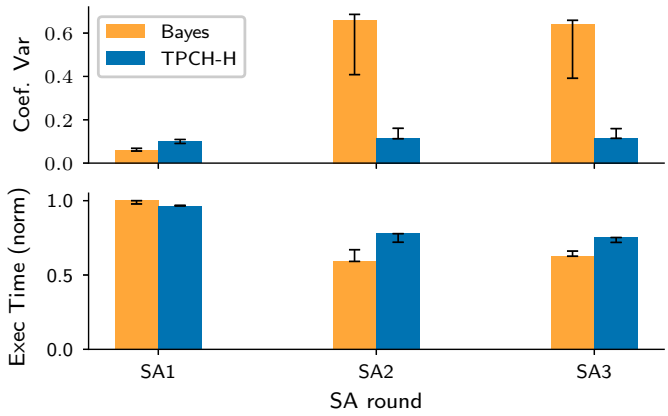


Figure 5.5: Coefficient of Variation (CV, $c_v = \frac{\sigma}{\mu}$, top) and normalized workload execution time (bottom) when using the parameters picked after each SA round.

square error, and the plot displays median values, with bars for the 10th and 90th percentile.

The RFR model has a median error less than 20% across all workloads, and also maintains the lowest error for all the workloads at the 90th percentile compared to SVR and LR. This concludes that it represents the best estimate of the true significant parameters and select the top 6 of those (out of 30) for comparing with the SA stage of Tuneful (that also selects the top 6). This selection is made following the Pareto principle (80/20 rule).

Tuneful detects the significant configuration parameters within 20 samples: To validate our algorithm, I measure the classification error S_{error}, representing the proportion of the best estimated significant parameters missing in the output of Alg. 1 (the top 6 important parameters in P_s after each round). S_{error} is weighted based on importance, with each configuration parameter classified either as high-influence or low-influence according to its normalised importance (as determined by RFR). The error weight of misclassifying a configuration parameter varies depending on its class, as follows: the weight of misclassifying all the high-influence parameters is 0.8 of the total error, while the weight of misclassifying all the low-influence ones is 0.2 of the total error. Figure 5.4 shows the S_{error} associated with each SA round, I ran the entire experiment 10 times to plot the median, 10th percentile and 90th percentile.

Across all workloads, Tuneful reaches a 90th percentile error of 20% or less using only 2 SA rounds, detecting all the highly-influential parameters. For TPC-H, the error slightly increases when running 3 SA rounds. This can happen with all workloads having very few influential parameters, and it means that in top 6 some non-influential parameters have been selected; those are unstable across the SA rounds and differ from the estimated ground truth. Tuneful relies on the GP phase later to detect those non-influential parameters and focus the tuning towards the high-influence ones. I experiment with tuning the configuration of TPC-H workload using the parameters detected by the second and third SA rounds and found similar results, since the high-influence parameters were already detected by the second SA round. Taking this into account, only two SA rounds are used, picking the parameters selected by the second, as a compromise

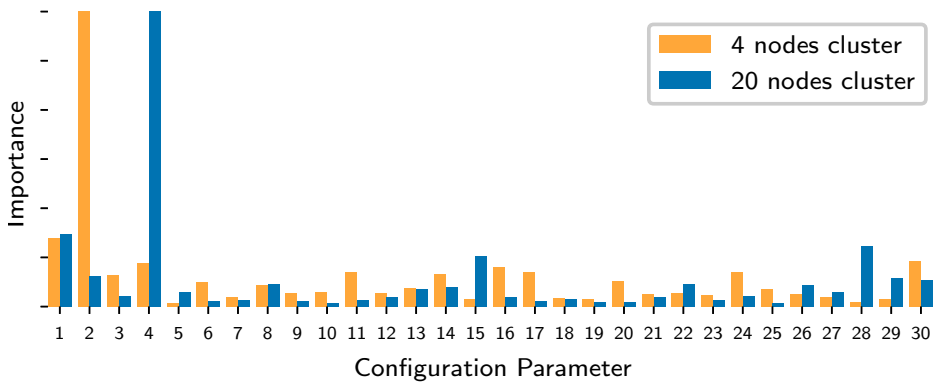


Figure 5.6: Relative configuration parameter importance for the Pagerank workload executed in 2 different clusters. Parameters with the largest impact on runtime can be cluster-specific.

between exploration (number of samples) and exploitation (information extracted from each round).

I also experiment with a smaller cluster of 4 AWS `h1.4xlarge` instances to validate that two SA rounds are enough over clusters of different resources. Figure 5.5 (top) shows the coefficient of variation (CV, defined as the ratio between the standard deviation and the mean) in execution time for the Bayes and TPCH-H workloads if I picked parameters selected by different SA rounds for tuning. At the 90th percentile, the CV at the first SA round is small as the picked parameters for tuning do not include all the highly-influential ones. The parameters detected as important by the second SA round have higher influence, leading to higher variation (the changes made by the tuning stage to the values of parameters selected as important lead to wider variations in execution time). The third SA round has only a marginal change over the previous round. Figure 5.5 (bottom) shows the execution time when using tuned parameters picked by each SA round. Similarly, at the 90th percentile, the parameters picked by the second SA round achieve a better execution time compared to the first SA round, with only marginal improvements made by a third SA round.

Significance-awareness in different environments: Figure 5.6 demonstrates how the significant parameters of the *same* workload vary in different clusters. It shows the estimated significant parameters along with their importance of the Pagerank workload across two different environments: a AWS cluster of 4 nodes and a Google compute engine cluster of 20 nodes. When the Pagerank workload was deployed in the two clusters, Tuneful detected some entirely expected differences, e.g. CPU and the parallelism level (param. 4) being the most important in the large, over-provisioned cluster and memory (param. 2) the most influential in the small cluster. However, there are also less obvious differences in parameter importance: speculative execution of tasks (param 15) and whether to compress variable broadcasts (param 28) being more important on the large cluster, and the choice of the serializer (param 30) being more important in the small cluster. Overall, the algorithm proposed is able to tune accordingly for changes in the environment that reflect indirectly on how configuration values map to workload runtime.

5.3.3 Tuning Effectiveness and Efficiency

I evaluate Tuneful in the zero-knowledge tuning mode, the evaluation of tuning a new workload in a new cluster, assuming 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 and *zero knowledge* mode, I only look at workloads that do not evolve. This is for the scope of this chapter, evolving workloads are experimented in the *extended knowledge* mode of Tuneful in Chapter 7; a decision made to compare the behaviour of Tuneful's strategy against existing state-of-art tuners and establish a performance baseline. Each state-of-the-art system that I compare against was allowed a maximum budget of 100 executions (representing a good balance between space exploration and experiment cost) for reaching a stable tuned configuration. Then I compare the configurations picked by Tuneful with the configuration tuned by:

1. *OpenTuner* [21], a general tuning system that uses ensembles of search techniques such as hill climbing, differential evolution, particle swarm optimization and pattern search. OpenTuner evaluates which techniques perform well over a window of time and picks them more frequently than the ones that have poor performance (those can even get disabled). OpenTuner is selected as it covers a wide range of search algorithms.
2. *Gunther* [83], is a Hadoop configuration tuning system that leverages genetic algorithms to search for good configurations. To compare Gunther with Tuneful, I implemented it for Spark based on the details given in the paper [83], choosing a population size of 60 and evolving for 20 generations (as reasonable limits for the number of workload executions required).
3. A configuration picked through *Random search*.
4. Cherrypick [20], is a GP based configuration tuner for the cloud configuration (low dimensional number of parameters). I compare against it to demonstrate that directly applying the standard GP to tune the high dimensionality of Spark parameters is not practical. Other published work either uses search techniques already covered by OpenTuner, or the implementation details were too sparse to reproduce the approach.

Metrics: Three metrics are used to evaluate Tuneful:

1. *Execution time saving*, the amount of execution time saved by Tuneful and competing approaches with respect to the default configuration. 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 while repeatedly running workloads in a cloud environment. The target is to get close-to-optimal configurations (within 5% 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 of workload execu-

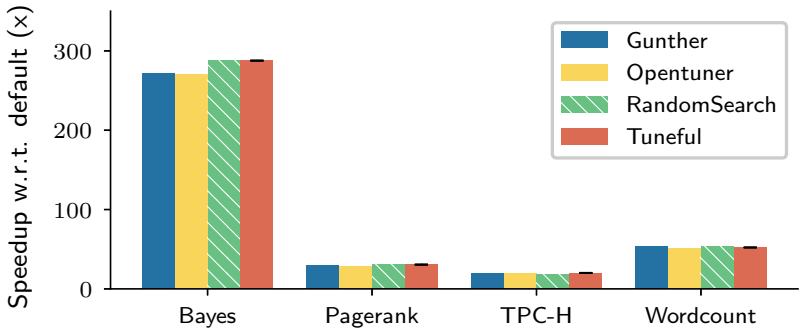


Figure 5.7: Execution time acceleration (X times) w.r.t Spark default configuration (higher is better).

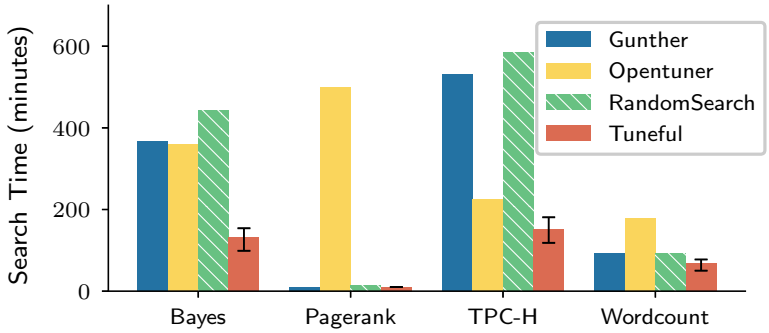


Figure 5.8: Search time of the different tuning algorithms to find 5% of the optimal configuration (the lower the better).

tions.

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

Tuneful finds configurations comparable to the state of the art tuning systems: Figure 5.7 shows that at the median, Tuneful is able to obtain effective configurations for all workloads, with a very small inter-percentile (10-90) range. Tuneful maintains a comparable performance to the other non-incremental tuning systems. I am comparing against the default configuration as a baseline usable across workloads. The relative differences between algorithms are relevant rather than the precise acceleration figures (in a realistic setting those would be lower when starting from reasonably hand-tuned configurations). In evaluating the execution time acceleration with the best configuration found by each tuner per workload, the other algorithms are not penalized if they are slow in finding good configurations. Therefore each of them is allowed to use 100 execution samples per workload (our fixed maximum budget). In comparison, the results of Tuneful is presented using just 35 execution samples per workload (as it would normally be used).

For evaluating differences in time taken to *find* configurations close-to-optimal, each algorithm is allowed to execute workloads until it finds the first configuration resulting in a runtime

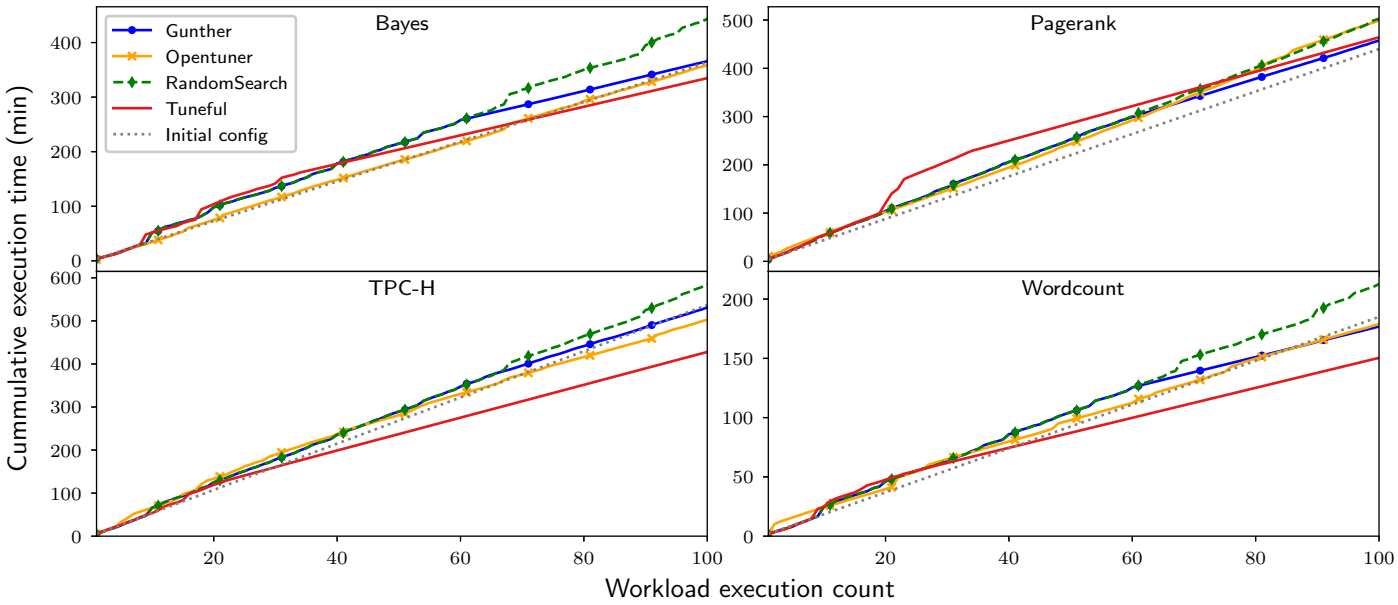


Figure 5.9: Cumulative execution time over 100 workload executions, under iterative tuning. Shallow slopes represent better configurations (smaller time increment for executing the workload once). Steeper slopes represent worse configurations.

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: Figure 5.8 shows that for the Bayes workload, OpenTuner, Random Search and Gunther need 2.7X, 3.3X and 2.7X more search time compared to the median Tuneful search time. For Pagerank, Random Search takes 1.4X the time while Gunther has comparable performance. However, Opentuner never finds a configuration with a runtime within 5% of the optimum within the allocated 100 executions. That represents a 49X increase in search time. For TPC-H, Opentuner, Random Search and Gunther take 2.8X, 3.6X and 3.2X more time than Tuneful, respectively. For Wordcount, Opentuner takes 2.6X the time while Random Search and Gunther 1.3X. 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, (ii) tune those to their optimal values. For each execution, the time spent in the tuning algorithm for actions such as sampling, running Bayesian optimisation on GP, etc is included. Tuneful runs 2 SA rounds, each using 10 execution samples, followed by tuning (15 execution samples at maximum). 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 slow execution of the workload. 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. Finally, the algorithm overhead is negligible: a few seconds to run and pick the next

configuration for exploration/tuning. This is due to the small number of samples used during the SA rounds and optimization, in addition to restricting the tuning to the influential parameters. The search time for the other three systems is significantly higher than Tuneful, mostly due to them not being data-efficient in finding close-to-optimal configurations.

The search cost is estimated based on GCP's [77] per-second pricing, which charges the user based on the number of seconds a machine instance is used. The search cost for tuning a workload is calculated through: summing up the number of seconds spent to execute the workload during the tuning phase, then multiplying the total number of seconds by the per-second usage price and the number of instances. 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 (incurred during the normal workload executions not in a separate offline phase like the other approaches).

Tuneful accelerates 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. Figure 5.9 shows the cumulative execution time of running each workload over multiple configurations, as determined iteratively by the tuning algorithms considered. Here, I start from a plausible developer-guided configuration to reduce exploration costs across the large search space. The dotted line shows cumulative execution time for this configuration *without any tuning*.

The tuning "pays off" only after the lines intersect the dotted line (which represents the plausible initial configuration), even if good configurations were found much earlier (finding them required exploring some configurations worse than the initial). Tuneful explores the search space for 35 executions (20 during SA and 15 for tuning), then it picks the best configuration it found and continue only using that. I let other tuning algorithms run longer (100 executions) to see if they find configurations that are better or equivalent to Tuneful's. 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 (shallower slope), the exploration cost for certain configurations is high (the hump after execution 20) and can not be amortized in 100 executions.

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 the workload's lifetime. Here, the benefits of the tuning hinge on how quickly the tuning approach converges to a good configura-

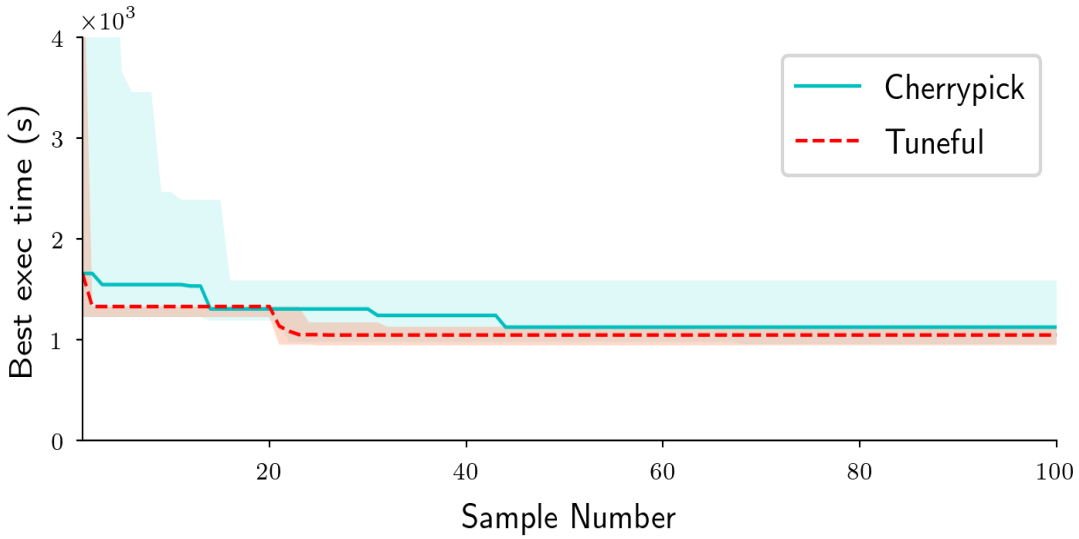


Figure 5.10: Convergence speed of Cherrypick versus Tuneful (Pagerank workload). The shaded space represents the area between the 90th and 10th percentile of execution time.

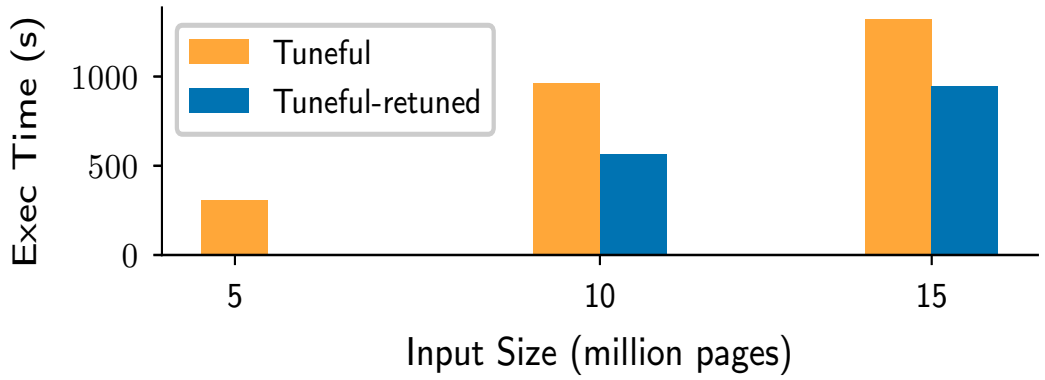


Figure 5.11: Execution time of Tuneful picked configuration over the evolving input sizes.

tion and how long the workload remains "alive" (with the same characteristics) after amortizing the tuning cost.

Tuneful compared to other GP approaches: By leveraging significance-aware GP optimization, Tuneful is able to find good configurations significantly faster than the other approaches. In comparison, it is known that just using GP directly for the high dimensional configuration space will converge slowly or get stuck in local minimums [112].

I experiment with tuning the Pagerank workload using Cherrypick [20], with the same 30 configuration parameters are used, and compare against Tuneful. The experiment is performed 10 times, each starting with a different initial random configuration, given to both systems (they start exploring the space from the same point). In Figure 5.10 the area between the 90th and 10th percentile of execution time is shaded, and the line shows the median of 10 experiments. I plot the minimum execution time found by the tuning algorithm until each sample on the x axis (akin to stopping the tuning at that point and running with the best configuration so far).

At the 90th percentile, Cherrypick tries configurations that are more costly (more variation, Cherrypick's median at execution 100 is above the 90th percentile of Tuneful) and takes a significantly longer time to find configurations close to the ones found by Tuneful: Tuneful's median best execution time still 7% faster than Cherrypick's median at sample 100; Tuneful's 90th percentile is 29% faster than Cherrypick's.

On the other hand, strategies employing SA on its own, without data-efficient tuning approaches such as GP fail as well: For example, even after reducing the search space, Gunther tunes the 6 significant configuration parameters for the Bayes workload using 2X the search cost needed by Tuneful.

Tuneful accommodates the need for configuration retuning over evolving input sizes through its data-efficient tuning: Figure 5.11 shows the execution time of Tuneful-picked configurations when the input size evolves. As shown in the figure, reusing the same configuration over evolving input sizes misses further optimization opportunities and retuning the configuration enables additional optimizations, with execution time savings of up to 40%.

Retuning the configuration by restarting Tuneful is still efficient and happens incrementally compared to the other state-of-the-art approaches, which need hundreds of execution samples. Nevertheless, understanding the reasons behind the need for retuning can help making retuning more efficient. The need for retuning usually happens due to a change in workload characteristics. This change causes a previously used configuration to become obsolete and degrade the performance. This happens due to: 1) a need to adjust the values of the significant parameters to adapt to the workload change (e.g., increase the shuffle buffer size or the executor allocated memory or cores), or 2) a need to consider other parameters as influential for this workload due to change in workload/environment (e.g., the workload becomes not only CPU intensive but also shuffle intensive, with a need to consider shuffle related configuration parameters for tuning).

In the latter scenario, retuning by restarting Tuneful from scratch is needed to detect the new significant parameters and tune them accordingly. On the other hand, the first scenario means the influential parameters are still similar over the evolving input sizes, yet their values need to be adjusted. In this case, leveraging the knowledge of these influential parameters is crucial to saving the cost of re-exploring them, and utilizing the previously tried executions during tuning them has the potential to guide the retuning process. These potential benefits trigger finding an answer to a couple of prerequisite intriguing questions: 1) how to define if two workloads are similar and share the same influential parameters? 2) how to share the tuning experience (the tried executions during tuning) between similar workloads? Sharing such knowledge not only has the potential to accelerate the retuning of a specific seen workload, but also the tuning of a similar unseen workload. Chapter 6 tackles those two questions.

5.4 Summary

This chapter showed how data-efficient machine learning techniques can be leveraged to accelerate the tuning of DISC system configurations, presenting the first work to enable data-efficient tuning of high dimensional configuration parameters. I argued that configuration tuning should take place in an incremental workload-specific approach, proposing Tuneful's data-efficient tuning approach, which leverages incremental SA and BO.

Further, this chapter showed that Tuneful significantly reduces the exploration costs and accelerates the amortization of tuning costs, while online finding configurations that are comparable to the ones from state-of-the-art tuning algorithms. It also illustrated how Tuneful is designed to be integrated into Spark to provide an efficient configuration tuning with negligible overhead.

In the next chapter, I explore how to leverage the knowledge acquired during tuning a workload to accelerate another similar workload. To answer this question, as a first step, I start with defining similarity in this context. This is done through studying how to represent a workload based on its execution metrics to detect similar workloads and share the tuning knowledge across them (Chapter 6).

Chapter 6

Similarity-aware Configuration Tuning

Virtually all of the existing work for DISC systems configuration tuning assume a static environment, such as a stable workload running in a fixed underlying cluster. Therefore, it is assumed that changes in the workload/environment are infrequent and re-tuning from scratch is cost-effective.

On the contrary, in a rapidly changing environment such as big data systems where the input sizes keep evolving and the underlying cluster dynamically changes (as demonstrated in Figure 5.11), it is crucial to perform tuning/retuning efficiently using a small number of executions to accomodate these frequent environment changes, which eventually will accelerate the amortization of tuning costs.

The existing work retunes the configuration either by re-learning the models or restarting the search algorithm, which is slow and prohibitively expensive. Ignoring any prior knowledge acquired during workload tuning poses poor adaptivity to such changes and slows down the tuning process, incurring high re-tuning costs. In this chapter, I propose solving this problem using a similarity-aware multitasked tuning approach (Similarity-aware Tuneful). To achieve this, the similarity-aware Tuneful leverages neural encoding of workload execution metrics to detect workload similarities, then shares the tuning knowledge gained previously across the similar workloads using Multitask Bayesian Optimization (MTBO). The main focus of this chapter is on *how* to define similarity across workloads and leverage to accelerate the configuration tuning. I consider workloads already tuned using Tuneful's significance-aware tuning algorithm (Chapter 5) and address accelerating the tuning of similar/evolving workloads using the proposed similarity-aware tuning approach.

The key contributions of this chapter are:

- Show how to learn a representation of a workload through encoding its execution metrics in a lower dimensional space, while capturing nonlinear dependencies across these metrics—a prerequisite step for similarity identification across workloads.
- Propose a novel similarity-aware tuning approach that detects similarity across workloads and shares the tuning knowledge between them using MTBO to accelerate workload configuration

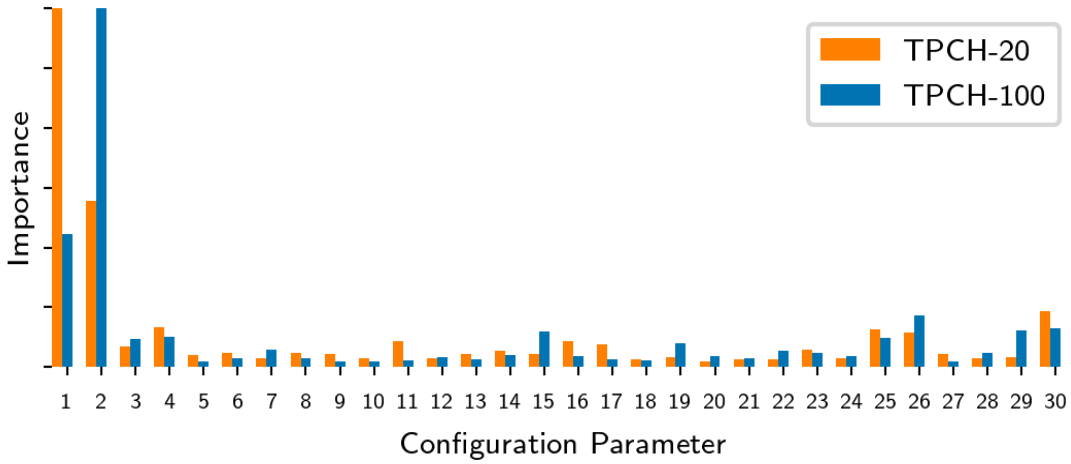


Figure 6.1: Configuration parameter importance for TPCH-20 and TPCH-100 workloads, showing that the parameters which affect runtime the most are similar yet vary in their impact.

tuning.

6.1 Motivation

As Figure 4.3 showed that the best configuration of TPCH-20 becomes obsolete for the evolving input sizes such as TPCH-100. Despite this, both workloads have similarities in terms of the configuration parameters that influence performance most. Still, there is a crucial need to adjust these parameters to accommodate evolving input sizes. Figure 6.1 shows the importance of the configuration parameters of TPCH-20 and TPCH-100 workloads. The calculation of the significance is based on the contribution of each configuration parameter in predicting the execution time, as determined from the 100 executions of each workload (as described earlier in § 5.3.2.1). For interest, *spark.executor.cores*, *spark.executor.memory*, *spark.shuffle.compress*, and *spark.serializer* (parameter 1, 2, 26 and 30) have the highest influence on performance. While the influence of each parameter varies from workload to another depending on its resources' needs, they still share the same significant parameters. For example, *spark.executor.memory* and *spark.shuffle.compress* are of significance for both workloads but have more importance for TPCH-100 than TPCH-20 due to the growth in the amount of processed and shuffled data.

This motivates sharing the knowledge of these influential parameters and the experience of tuning them across these similar workloads. This shared knowledge can save the cost of significant parameters exploration (an expensive process) and the previous experience of tuning them has the potential to guide the tuning process, ultimately accelerating the convergence to near-optimal configuration. However, the success of exploiting the similarity hinges on the accuracy of detecting similar workloads in the first place.

6.2 Approach

To accommodate the need for cost-effective configuration tuning, I propose a similarity-aware tuning approach that leverages the tuning knowledge between the similar workloads and effectively accelerates the tuning process.

Assumptions: I assume the existence of readily available workloads tuned using a *base* tuner. I build on top of Tuneful (Chapter 5) and use it as a *base* tuner, since it finds a configuration comparable to the-state-of-the-art significantly faster and exposes workload-specific influential parameters. Moreover, it tunes the configurations using Single Tasked Gaussian Process (STGP), which has been widely adopted in several configuration tuning systems ([42, 20]).

Leveraging the proposed approaches in this dissertation to automate the extraction and sharing of tuning knowledge in any GP-based tuner is an interesting future avenue.

This work addresses *how* to define similarity across workloads and *efficiently* leverage this similarity in twofold: 1) tune a new similar workload, 2) retune already seen workload to accommodate workload changes. In this context, similar workloads are workloads that have similar resource usage characteristics and thus have similar influential parameters. For example, they might have a similar amount of data shuffled with respect to the whole input, or spend a similar amount of time doing GC with respect to the whole CPU time.

To define similarity across workloads, I start with identifying the main components that distinguish each workload. In order to achieve this, the execution metrics of a diverse set of workloads are monitored to learn the main aspects that represent each workload. I then detect workload similarity using the learnt workload representation, and lastly employ MTBO to accelerate the tuning of these similar workloads.

6.2.1 Workload Monitoring

For each workload, the monitored metrics involve CPU time, number of tasks per stage, number of stages per job, input and output size, GC time, execution time, data serialization/deserialization time, the size of shuffled data, memory spilled data and disk spilled data. All the numeric metrics that relate to those workload execution features are collected, Table 6.1 lists the captured metrics and their description. The overhead of collecting these metrics is minimal as they are already present in Spark logs.

To build representative statistics of each metric, for each stage in the Spark job, the 95% quantile of the metric across the tasks of the stage is calculated. Then, the captured metric is averaged across all the stages and represented in terms of ratios comparable across workloads. For example, I consider the amount of GC time relative to the total CPU time and the amount of shuffled data with respect to the total input data, rather than the quantitative values of GC, CPU time, shuffled and input data size. This choice is made since this characterization is ultimately used to detect similarity between workloads, in this context two workloads are similar if they inherent similarity in terms of their relative resource usage- not the quantitative resource

Metric	Description
Executor runtime	Average executor runtime across the job's stages
Executor CPU time	Average executor CPU time across the job's stages
Input size	Average input size across the job's stages
Shuffle read bytes	Average number of bytes read during data shuffle across the job's stages
Shuffle written bytes	Average number of bytes written during data shuffle across the job's stages
Memory spilled bytes	Average number of bytes spilled into memory across the job's stages
Disk spilled bytes	Average number of bytes spilled into disk across the job's stages
Executor deserialization time	Average time spent to deserialize data across the stages
Executor deserialization CPU time	Average CPU time spent to deserialize data across the stages
Result size	average result size across the stages
Number of stages	number of stages of the job
GC time	Average time spent performing GC across the job's stages
Stage output size	Average output size of tasks within a stage
Result serialization time	Average time spent to serialize data across the stages
Input records read	Average number of records read across the job's stages
Output records written	Average number of records written across the job's stages
Shuffle remote blocks fetched	Average number of remote blocks read during data shuffle across the job's stages
Shuffle remote bytes read	Average number of remote bytes read during data shuffle across the job's stages
Shuffle local blocks fetched	Average number of local blocks read during data shuffle across the job's stages
Shuffle fetch waiting time	Average waiting time during data shuffle across the job's stages
Total shuffle blocks fetched	Average number of blocks read during data shuffle across the job's stages
Shuffle write time	Average time spent writing bytes during data shuffle across the job's stages
Shuffle written records	Average number of records written during data shuffle across the job's stages
Shuffle read records	Average number of records written during data shuffle across the job's stages

Table 6.1: The set of execution metrics monitored to learn workload representation.

usage. Intuitively, the workloads that are similar in their relative resource usage are expected to have similar influential parameters (as shown in Figure 6.1, two TPCH workloads have similar influential parameters. § 6.2.3 will show how they are also highly similar in terms of resource usage with a very small distance as depicted in Figure 6.3).

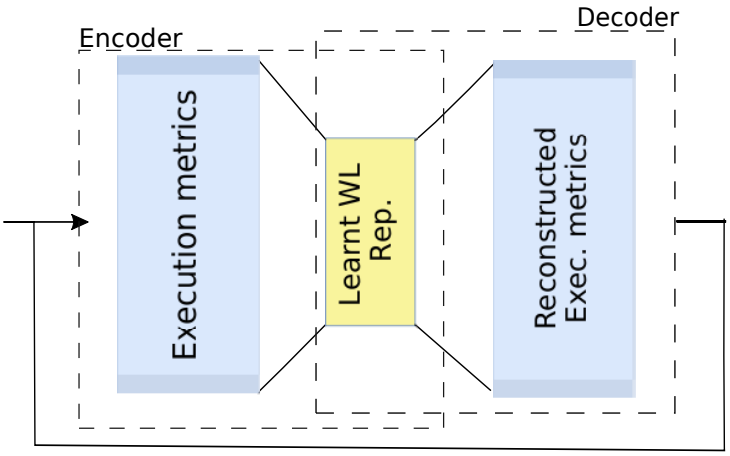


Figure 6.2: AE learning how to represent workload execution metrics.

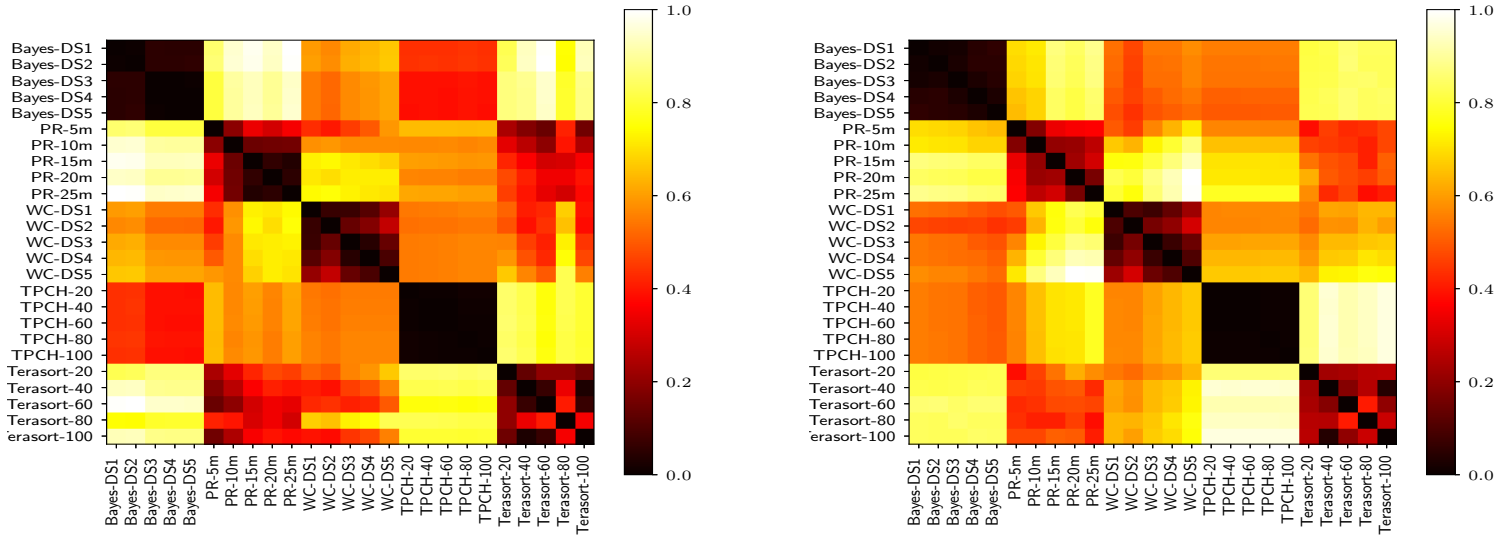
6.2.2 Workload Representation Learning

One approach for detecting similarities relies on first projecting the workload execution metrics in a low dimensional space then calculating the distance within this space, this is due to the fact that distance metrics such as L_p norm become more informative as you move to a lower-dimensional space [17].

I embrace the same approach but instead use neural encoding to embed workload execution metrics in a low-dimensional space. This is in contrast to previous work which uses PCA for this purpose. The benefit of using a neural encoding is that it can capture nonlinear dependencies in the latent space as opposed to PCA which projects into linear space, leading to a higher data reconstruction loss. Therefore, a nonlinear neural autoencoder is employed to lower this loss. The nonlinear autoencoder has been successfully applied to encode data in different domains such as image processing, natural language processing and signal compression. This is the first work to encode workload execution metrics using nonlinear autoenecoder and validate it against PCA (as will be shown in § 7.1.3).

An AutoEncoder (AE) is a neural network that learns how to represent data in a low dimensional space. It consists of two components: encoder and decoder. The encoder has one or more neural layers that learns $f(x)$, which transforms input $x \in \mathbb{R}^d$ into $h \in \mathbb{R}^n$ latent space, such that $n < d$. The decoder learns $g(h)$, which reconstructs the latent low dimensional h back to $\hat{x}$ of the same dimensionality as the original input space. The AE compares the decoder's output (reconstructed input) $\hat{x}$ to the encoder's original input x and update the neurons' weights accordingly to minimize the loss of the re-constructed input L as shown in Equation 6.1, where L is a function that penalizes $g(f(x))$ for not being similar to x (e.g., the mean square error).

$$L(x, \hat{x}) = ||x - g(f(x))||^2 \tag{6.1}$$



(a) 1 instance

(b) 5 instances

(c) 10 instances

(d) 15 instances

Figure 6.3: Distance across workloads using different number of fingerprint instances.

Figure 6.2 illustrates how the AE learns workload representation, starting from the high dimensional execution metrics as encoder's input to learn the low dimensional representation, then feeding back the reconstructed execution metrics using the decoder to the encoder. The reconstructed metrics are compared against the original execution metrics and the weights of the encoder's neurons are updated accordingly, minimizing the information loss in the learnt workload representation.

A one layer nonlinear AE is used to encode the execution metrics into a low dimensional latent space. This decision was made as it minimizes the reconstruction loss compared to linear AE and PCA, moreover it has a smaller number of model parameters compared to multilayer AE.

6.2.3 Similarity Analysis

The aim of the similarity Analysis is to find a *source*—already tuned—workload and share its influential parameters with the *target* workload (the one under tuning). The source workload is the most similar one based on the Manhattan distance between the encoded execution metrics h. Manhattan distance is used as the distance metric since it provides more contrast compared to the Euclidean distance and other L_p norm metrics [17].

The distance is calculated between workloads running under the *same* configuration, using Equation 6.2. A *fixed* representative configuration is used to run the workload once and extract workload execution metrics using the learnt workload representation (this works as a workload fingerprint). The workload's fingerprint is then matched to a fingerprint of the seen workloads. I experimented with varying the number of workload executions used for workload fingerprinting and how this influences the accuracy of workload matching. As shown in Figure 6.3, using a single representative configuration was sufficient to provide the same matching as multiple executions. Thus a single execution instance is used to limit the cost of workload fingerprinting. The benefit from running multiple executions might exist but the returns diminish quite quickly. The 1 instance case still captures the essential similarity patterns (e.g., the high similarity across TPCH workloads, the distances between Terasort-80 and other workloads). Two workloads are considered similar if they have the smallest distance in terms of their relative resource usage, since this implies that they have similar influential parameters. Figure 6.2 shows how the similarity analysis takes place online, upon receiving a new *target* workload, the Tuning Manager, as a first step, checks if this workload has a similarly matched workload or not. If not, it suggests the fingerprinting configuration to the Spark driver, which then forwards to the Spark workers and execute the workload. After workload execution, the Tuning Manager provides the workload's execution metrics to the Similarity Analyzer, which encodes the execution metrics of this workload—leveraging the learnt workload representation from the offline phase. Then the workload with the smallest distance is selected as the source workload and shares its

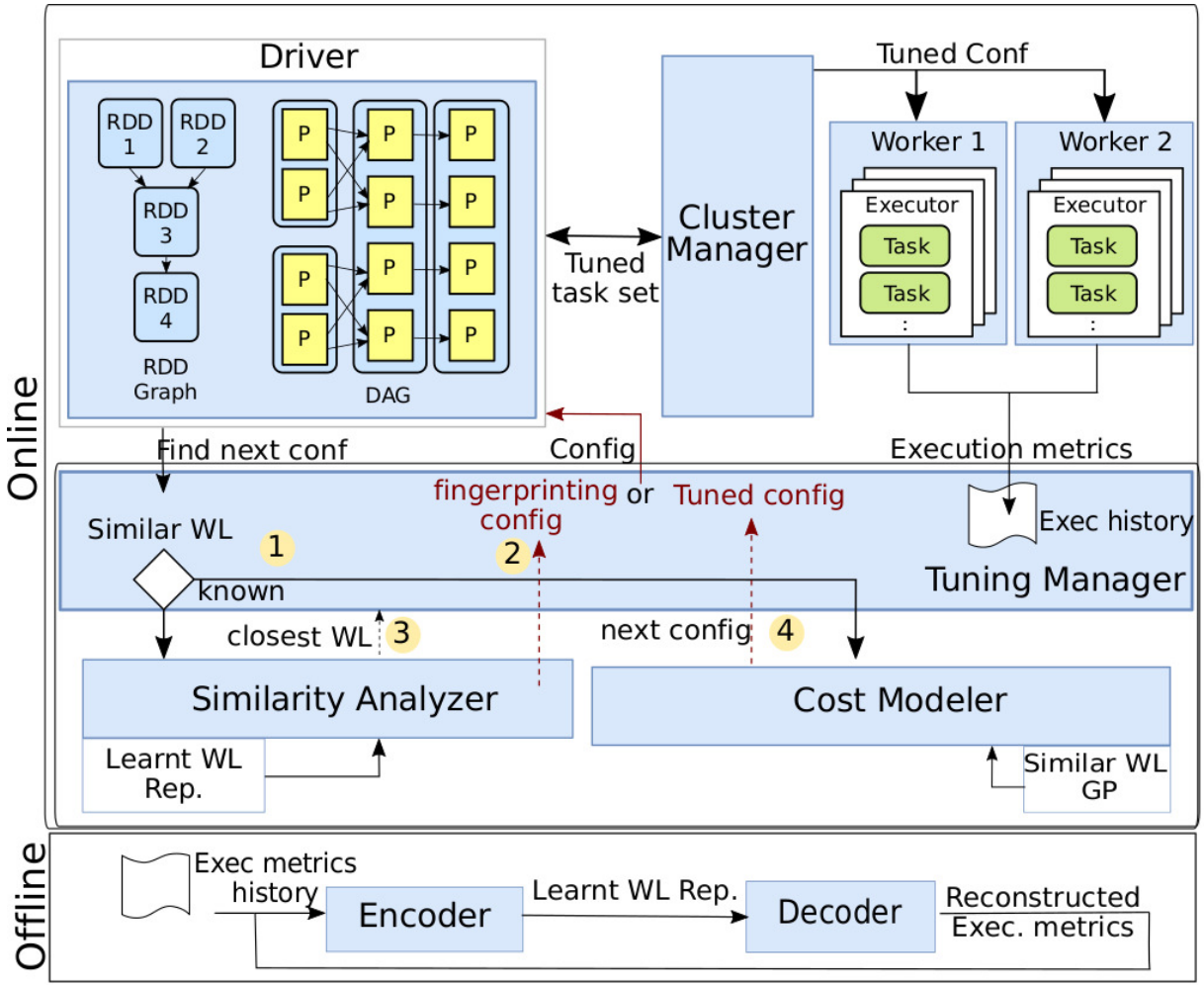


Figure 6.4: Similarity-aware Tuneful in the offline and online phases to tune Spark

influential parameters with the target workload.

$$d = \sum_{i=1}^{n} |h_{1i} - h_{2i}| \tag{6.2}$$

6.2.4 Multitask Configuration Tuning

After finding a similar *source* workload, already tuned using the *base* tuner, the knowledge gained during tuning the *source* workload is transferred to the *target* workload. This knowledge includes: 1) the significant configuration parameters. As illustrated in Chapter 5, in DISC systems, not all the workloads share the same influential parameters, this is due to the high diversity of the workloads served (e.g., graph analytics, machine learning, SQL, text analysis). 2) the execution samples evaluated during tuning this source workload.

The Tuning Manager passes the tuning knowledge of the source workload to the Cost Modeler to reuse the significant configuration parameters and leverages the previously seen execution samples. This knowledge guides the Cost Modeler to suggest the next configuration

Algorithm 2: Similarity-aware Multitask configuration tuning

Input : α, 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: $GP_s \leftarrow GP_s|(t_j, x_i, y_i)$

sample, with the highest potential to maximize the EI. In the beginning, it suggests the configuration samples that help to learn the covariance across the different workloads (modelled as tasks). Then it decides the next sample to try based on this covariance (i.e. if the covariance is strong, it picks the sample that maximizes the EI of the source task, otherwise it picks the sample that maximizes the EI of the target task). More details on how MTGP works can be found in [108] and § 2.3.4.

In Alg. 2, the steps of the multitask configuration tuning algorithm is shown, its input arguments are as follows:

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

After fingerprinting the new *target* workload w (as described in § 6.2.3) through finding its execution metrics $w_{metrics}$, the algorithm performs the followings steps:

- The algorithm starts at line 2 with calculating the distance between the execution metrics of the target workload $w_{metrics}$ and the seen workloads W_{seen}, the distance is calculated based on the learnt low dimensional encoding of workload execution metrics.
- Then workload s with the smallest distance is selected to use as the *source* task for the multi-task tuning at line 3.
- The significant parameters of s is reused to tune w and a new task t_j is added to the GP of s for w (line 5).
- At line 6, I leverage the GP of s to select the next configuration to execute w, the next configuration is selected so that it maximizes α acquisition function over task t_j.
- Lastly, workload w is executed using the selected configuration (line 7) and the execution cost C is used to update the GP model of task t_j(line 8).

Figure 6.4 shows how the tuning takes place online during the normal workload executions. The Cost Modeler suggests the next configuration to explore, then the Tuning Manager feeds the execution cost (execution time) back to the Cost Modeler to update the MTGP model. The

Cost Modeler detects if it is time to stop the optimization. It stops modelling a workload after suggesting a minimum of n samples (e.g 10 samples) and then after the EI drops below 10%—a decision made to ensure balancing between the exploration of the tuning space and exploitation of the best configuration found.

Resilience to workload mismatching: The inaccuracies in workload matching can be detected through monitoring the gap between the predicted execution time by the MTGP model and the actual execution time. If a continuous degradation takes place, then workload matching is triggered again (given that sufficient workloads have been seen, otherwise a standalone tuning is performed using Tuneful's Significance-aware tuning, as described in Chapter 5). Even if a workload mismatching happens due to the limited number of seen workloads, Similarity-aware Tuneful guarantees that the found configuration is not worse than the reused state-of-the-art's configuration, which implies reusing the configuration suggested by the state-of-the-art for a similar workload (I refer to this as Direct Transfer and illustrate that more in § 7.1.1).

Similarity-aware Tuneful is implemented in Python on top of Spearmint [105], a Python BO framework that implements MTBO based on [108].

6.3 Summary

The chapter argued that extending BO with workload similarity characteristics has the potential to significantly accelerate the amortization of tuning costs and enable effective tuning in a rapidly changing environment—even when tuning configurations of high-dimesnionality, which was previously thought to be impractical with BO. It proposed a similarity-aware tuning approach that detects similarity across workloads leveraging a neural encoding of workload execution metrics, then shares the tuning knowledge across similar workloads using MTBO.

The next chapter evaluates how effective is the proposed similarity-aware tuning approach in accelerating the configuration tuning and cost amortization.

Chapter 7

Evaluation of the Similarity-aware Configuration Tuning

In the previous chapters, I started with addressing the configuration tuning in the case of zero knowledge, proposing Tuneful's significance-aware tuning approach (Chapter 5), which proved to be cost-effective for recurring workloads (i.e., repeatable stable workloads, representing 40% of the analytics workloads [16, 45]).

To have a wider application of the cost-effective tuning to other types of workloads (e.g., evolving dynamic workloads), Chapter 6 argues that the tuning knowledge acquired over time (using Tuneful significance-aware tuning) can be shared to make the tuning efficient further for similar workloads, with the success of exploiting that being subject to the accuracy of detecting similar workloads in the first place. This chapter puts things together and evaluates how effective can the tuning be for evolving workloads when more tuning knowledge becomes available and shared across similar workloads. This is done through assuming the existence of readily available workloads tuned using the Tuneful's significance-aware approach (Chapter 5), then employing similarity matching for a newly received workload against the seen workloads (§ 6.2.3) and lastly, sharing the pre-acquired tuning knowledge with this new workload.

This chapter validates the contributions of Chapter 6:

- It shows, using a series of experiments, how exploiting a nonlinear neural encoder can effectively learn the nonlinear dependencies between workload execution metrics, leading to a representation learning accuracy that surpasses other earlier adopted techniques such as PCA. This learnt representation is used to detect similarity between workloads.
- It demonstrates that extending BO with workload similarity characteristics can significantly accelerate the configuration tuning—even when tuning configurations of high-dimensionality, which were previously thought to be impractical with the traditional BO.
- It shows how the proposed similarity-aware tuning approach obtains configurations comparable to prior work but converging to those significantly faster, enabling a quicker amortization of the tuning costs in a rapidly changing environment.

Application	Abbr.	Input data sizes (DS)
Pagerank	PR	5, 10, 15, 20, 25 (million pages)
TPC-H benchmark	TPCH	20, 40, 60, 80, 100 (compressed GB)
Terasort	TR	20, 40, 60, 80, 100 (GB)
Bayes Classifier	Bayes	5, 10, 30, 40, 50 (million pages)
Wordcount	WC	32, 50, 80, 100, 160 (GB)

Table 7.1: The set of applications and input sizes used to learn workload representation.

7.1 Evaluation

7.1.1 Methodology

I start with evaluating the accuracy of learning how to represent a workload execution metrics in a low dimensional space (§ 6.2.2) using nonlinear AE, linear AE and PCA (a prerequisite step for similarity detection across workloads). I then evaluate how effective is the similarity-aware Tuneful in two scenarios: 1) 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. 2) as more tuning knowledge is acquired. This is to assess the impact of having extended tuning knowledge on the 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 (e.g., Tuned-Configuration-As-A-Service).

7.1.2 Experimental setup

Cluster and configuration specification: I use a cluster of 4 AWS *h1.4xlarge* instances with 16 vCPUs, 64 GB memory, and 2TB storage each.

HDFS [13] version 2.7 is used for accessing the shared data and Spark version 2.2.1 as the system under tuning. A list describing each configuration parameter and its range can be found in Table 4.3. I use the same ranges when evaluating the other tuning approaches.

7.1.3 Workload Representation Learning

Environment setup: Keras v2.3.0 is used to build the AE running on AWS `c5.large` instance, with epoch (the number of complete passes over the dataset to update the internal model parameters) of 300 and a sigmoid activation function. Multiple other activation functions were tried and this one was selected since it yields the minimal information loss. For setting the epoch, as shown in Figure 7.1 different values until 1000 were tried and the information loss converged at 300. The same number of epoch is used for the linear AE.

Dataset: A dataset of execution metrics using two well known big data benchmarks (Hibench [62] and TPC-H [6]) was built. I selected five different applications from those benchmarks due to their wide adoption in real-life applications (§ 2.1.1). For each application, I

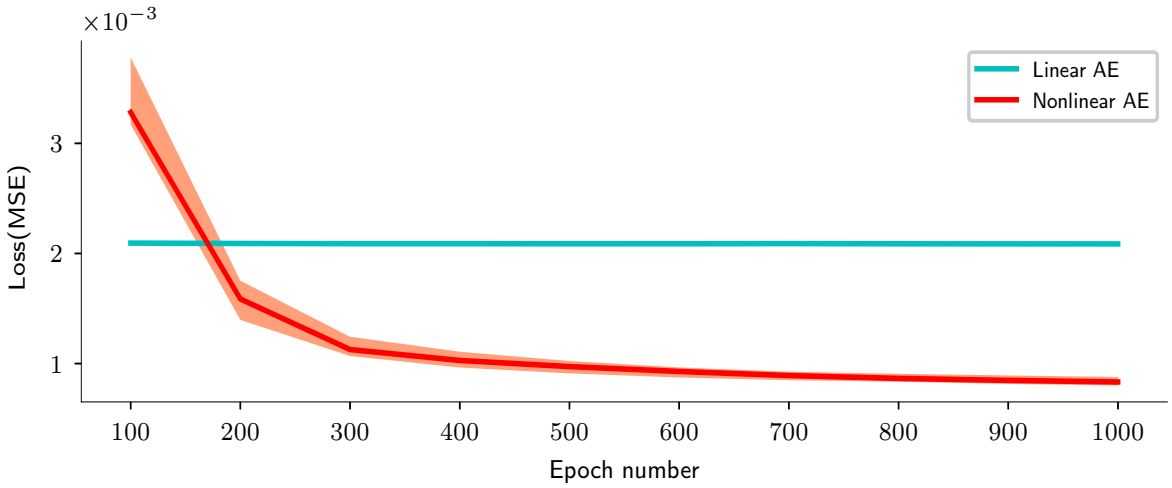


Figure 7.1: Reconstruction loss over the number of epochs at median, the shaded area represents that space between the 90th and 10th percentile of 10 experiments.

experimented with five different input sizes under different configurations for the 30 parameters in Table 4.3. with a total of 2200 application executions that took 2188 compute hours to execute. This time is not considered as an overhead of the proposed tuning approach as this data will be readily available over time in a deployed system. Table 7.1 lists the applications and input sizes, the dataset is publicly available on the project's data repository [14].

To evaluate the accuracy of learning how to represent workload's execution features in a low dimensional space, I compare the loss of our representation using nonlinear AE against PCA and linear AE. Figure 7.2 shows the reconstruction loss over the number of components/encoded dimensions during learning the representation using PCA, linear AE and nonlinear AE at the median, 90th and 10th percentile. The nonlinear encoder can represent the execution metrics using a smaller number of components/dimensions while maintaining less reconstruction loss compared to PCA and the linear AE. For example, at the 90th percentile the nonlinear AE can encode the execution metrics using 5 dimensions with a loss lower than PCA and linear AE using 6 dimensions, since they restrict the encoding of data into a linear space that does not capture the nonlinear relationships across the execution metrics. The close performance of PCA and the linear AE is due to the fact that a single layer linear AE learns a representation that spans the same subspace as PCA—the learnt representations are not identical though [95].

The workload execution metrics are encoded using 5 dimensions, representing a good compromise between the reconstruction loss and the number of components for all the three approaches (e.g., the PCA represents 92% of the data variance using 5 components).

20% of the dataset was excluded as a hold out validation dataset to evaluate the learnt representation. Further, to evaluate the generalization of the learnt representation in a different environment, a test dataset was built including the execution metrics of 5 applications running on a cluster of different characteristics: a 20 GCP instances (1 driver + 19 workers), with the

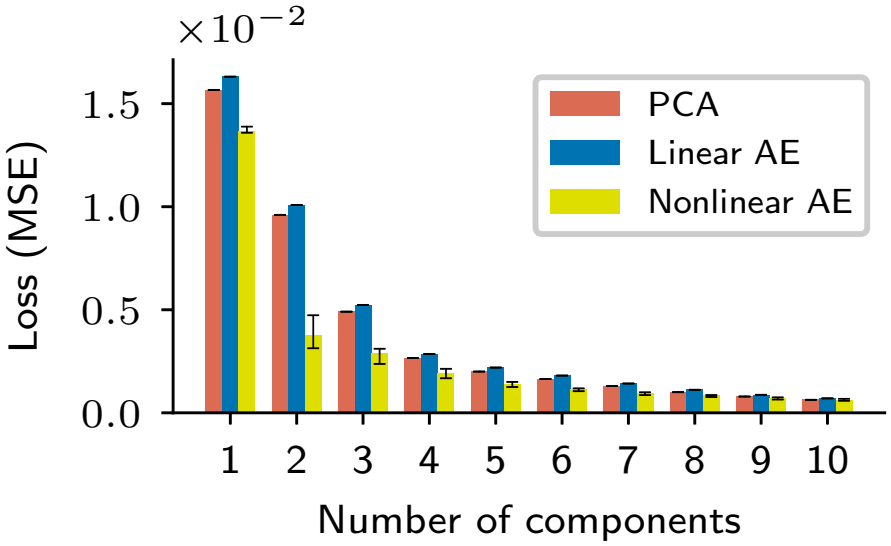


Figure 7.2: Reconstruction loss over the number of encoded low dimensional components, showing median, 90th and 10th percentile of 10 experiments.

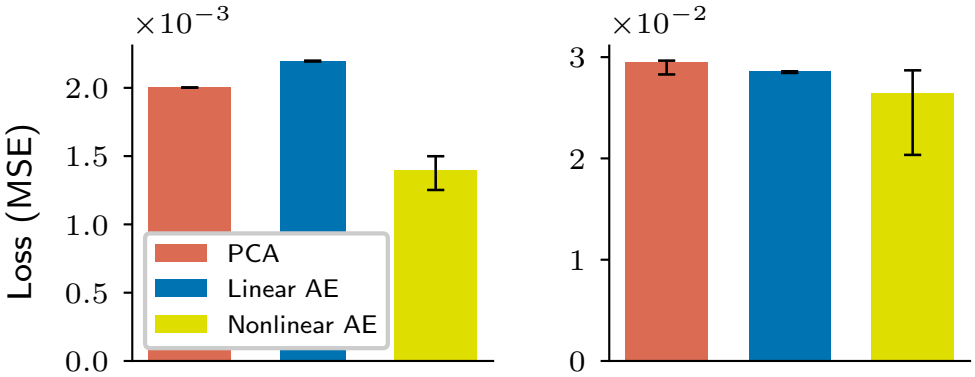


Figure 7.3: Reconstruction loss for 20% holdout validation dataset (left) and test dataset of 500 workloads execution metrics running on a different cluster (right).

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 dataset contains the execution metrics of 500 executions of TPC-H, PR, bayes, WC, TR and KMeans workloads. Figure 7.3 shows the reconstruction loss for the validation and test dataset. As shown in the figure the nonlinear AE maintains the lowest loss for the validation dataset at the 90th percentile and generalizes better than the other approaches for the test dataset with a smaller reconstruction loss at the median. Unlike the linear methods, the nonlinear AE has numerous ways of non-linearly representing the execution metrics in a lower dimensionality space (varies depending on the used random seed). Therefore, its percentile bar is larger than the other linear methods.

7.1.4 Tuning Effectiveness and Efficiency

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), I evaluate the effectiveness of tuning a set of workloads with growing input data and evaluate how this influences the speed of tuning cost amortization. I consider a limited set of "auxiliary workloads" that represents the workloads already seen and tuned using standalone Tuneful (Chapter 5), then evaluate tuning a set of similar workloads with evolving input sizes using Tuneful's similarity-aware tuning algorithm (Alg. 2), comparing against:

1. *Direct transfer*, which implies transferring the tuned configuration from the source workload directly to the target workload—comparing against this to show the importance of retuning the configuration.
2. Independent workload tuning using Tuneful's significance aware algorithm (as described in Chapter 5), comparing 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. I refer to this as *Standalone Tuneful* throughout this chapter.
3. *Transfer learning and single tasked GP tuning* (TL+STGP), which implies transferring the significant parameters from a similar workload then performing configuration tuning using STGP, comparing against this approach to evaluate the benefits of using MTGP against STGP.
4. For the sake of completeness, I also compare the tuned configuration against Random search, with a budget of 100 executions generated using low-discrepancy sequences [106]. I compare against this approach to assess how far the configurations provided by the Similarity-Aware Tuneful is from the best estimated configuration using this intensive exploratory approach.

Metrics: I use the same metrics as discussed in the evaluation of Standalone Tuneful (Chapter 5):

1. The execution time of the tuned configuration, akin to the running cost of the tuned configurations. The target here is to obtain tuned configurations close to what the-state-of-

the-art achieves.

2. Search Cost, the amount of time and actual cost (in $) required by each approach to find good configurations while repeatedly running workloads in a cloud environment. The target is to get close-to-optimal configurations (within 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 cost. The target is to amortize the tuning cost after a small number of workload executions.

The results are always presented as the median of 10 experiments, bars represent the 10th and 90th percentile.

Auxiliary workload set: consists of three workloads from three applications (Bayes-DS1, PR-DS1 and WC-DS1), each tuned using Tuneful as described in Chapter 5, performing an independent significant parameter detection and tuning for each workload. Table 7.2 shows the auxiliary workloads colored in green. I exclude two applications (TPC-H and TS) from the set of the auxiliary workload to evaluate the effectiveness of tuning unseen applications.

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

Workloads under tuning: To evaluate the effectiveness of Similarity-aware Tuneful in finding good configurations, for each application in the auxiliary workload set, I tune three workloads of evolving input sizes using Similarity-aware Tuneful and compare against direct transfer, standalone Tuneful, and Random search. Table 7.2 shows each workload's input size. I also consider tuning two workloads of two applications that were not included in the auxiliary workload set (TPC-H-DS1 and TS-DS1), evaluating the benefit of the proposed approach in tuning unseen applications.

Similarity-aware Tuneful finds configurations comparable to the state-of-the-art outperforming the direct transfer by 32% at median and 79% at the 90th percentile: Figure 7.4 shows the execution time of the configurations found by the different tuning approaches, the direct transfer of the configurations does not guarantee near-optimal performance, with a potential performance degradation of up to 85% compared to retuning the configuration. Across all workloads, similarity-aware Tuneful finds on average a configuration with an execution time within 8% of Tuneful's and 12% of Random search's. This average is computed over the dif-

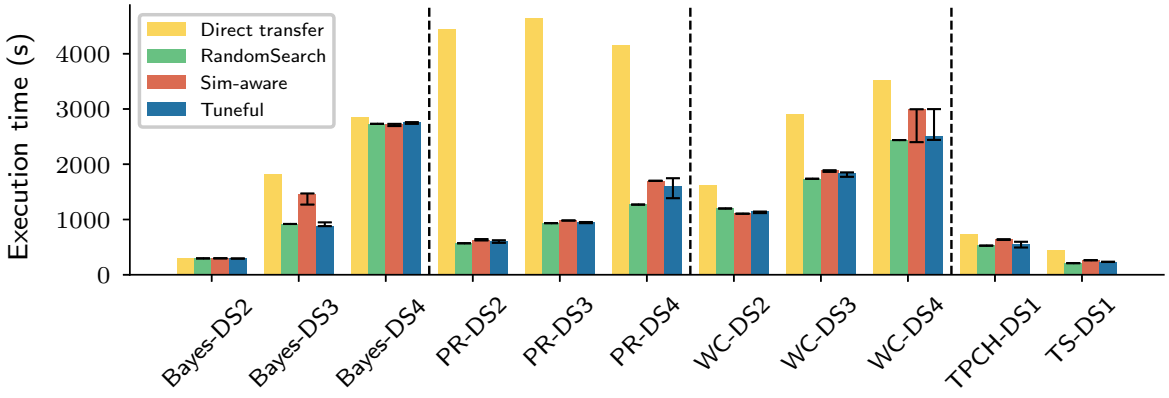


Figure 7.4: Execution time of the direct transferred configuration, the tuned configurations using Tuneful, Random search and the proposed Similarity-aware Tuneful approach.

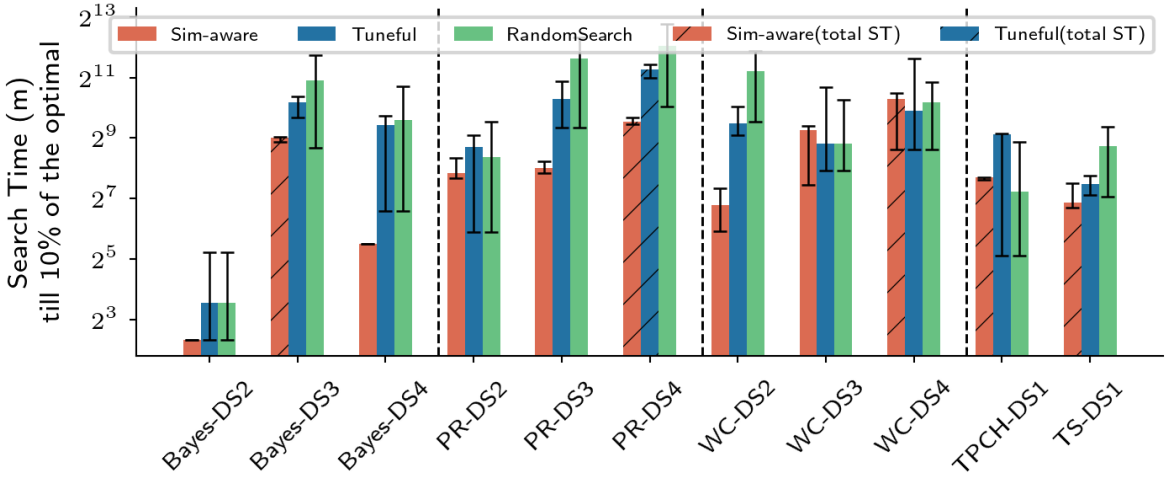


Figure 7.5: Search time till finding 10% of the optimal configuration using Tuneful, Random search and our proposed Similarity-aware Tuneful approach (note the logarithmic y axis). The dashed bars represent the total Search Time (ST) when a configuration within 10% is not found.

ference in median execution times for each workload that is shown in Figure 7.4. For some workloads, namely Bayes-DS3 and WC-DS4, standalone Tuneful and Random search outperform the similarity-aware Tuneful. This is due to the limited set of source workloads. The next section shows how this can be mitigated through extending the tuning knowledge (having a larger set of source workloads).

To evaluate differences in time taken to find configurations close-to-optimal, each approach is allowed to execute workloads until it finds the first configuration resulting in a runtime within 10% 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 approach or experiment.

The median search time for the-state-of-the-art is 2.3-3.7X longer when compared to Similarity-aware Tuneful: Figure 7.5 shows the search time of each approach until find-

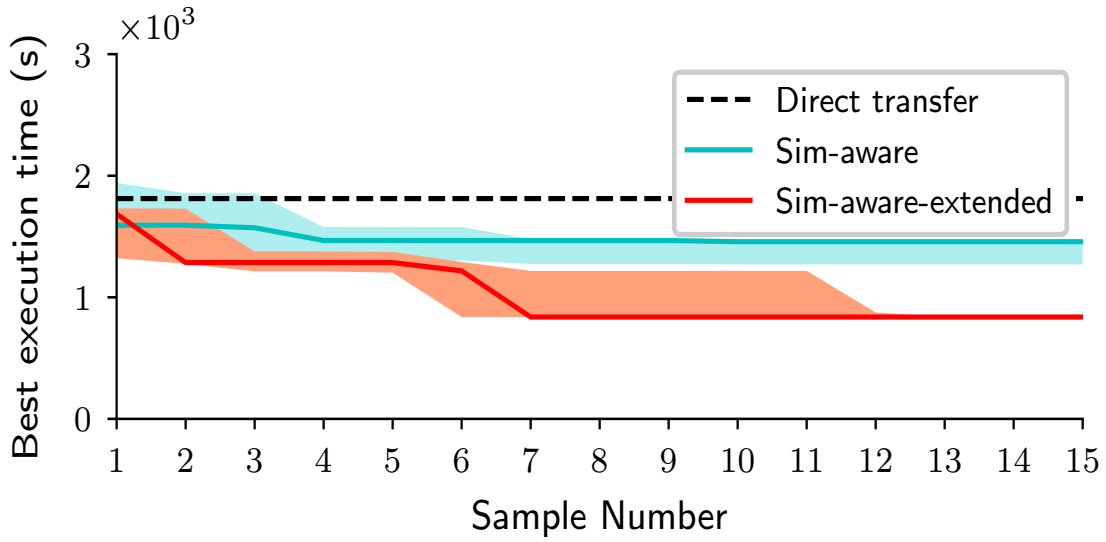


Figure 7.6: Convergence speed of the Bayes-DS3 workload when the significant parameters are transferred from Bayes-DS2.

ing a configuration within 10% of the estimated optimum. For some workloads, a configuration within 10% of the estimated optimal configuration is never found (due to the limited number of seen workloads), and I 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 Random search is 2.3 - 3.7X longer when compared to the similarity-aware Tuneful. Out of the 11 workloads considered and given a 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 Random search.

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 suggests mitigating that through using a larger set of auxiliary workloads.

Tuning with extended knowledge: I 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. Although Figure 6.3 shows that the Bayes workloads are close in distance (have similar resource usage and influential parameters), the covariance of the workloads (closeness in behaviour and how they respond to the different configuration values) still might vary. This is why Bayes-DS2 has a closer fingerprint than Bayes-DS1. Those work-

Approach/application	Bayes	PR	WC	TPC-H	TS
Random Search	$168	$483	$247	$9	$26
Tuneful	$116	$257	$132	$34	$9
Similarity-aware Tuneful	$35	$77	$123	$13	$7

Table 7.3: The search cost based on AWS [10] per-second pricing of the different tuning approaches.

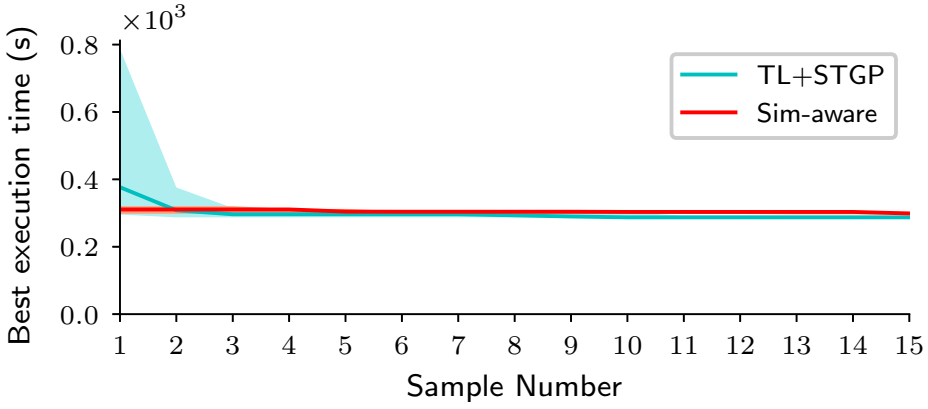
loads transition between bottlenecks as processed data size increases. Hence it's expected to see better behaviour correlation when using a workload with a fingerprint closer to the target workload.

As shown in Figure 7.6, extending the auxiliary workload set enables finding a configuration comparable to Tuneful and Random search reported values in Figure 7.4 (outperforming direct transfer by 54%), while the similarity-aware tuning happens in 38% less time compared to results in the previous section. This completely eliminates the trade-off between search time and optimality of the configuration that was present in the limited auxiliary workload set case.

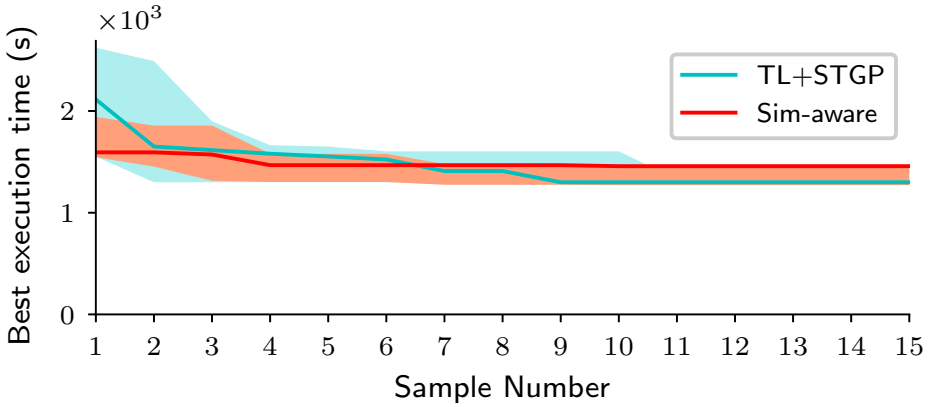
It also suggests that cloud providers, if able to observe executions across clients, would be in the ideal position to offer tuning-as-a-service in a way that minimizes costs, even for workloads that are repeated just a couple of times. Individual customers doing their own tuning will have a more complex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

The search time in Similarity-aware Tuneful is the sum of the *workload execution times* needed by the tuning algorithm to run the fingerprinting instance and tune the configuration parameters using MTGP. The MTGP needs 15 execution samples at maximum to tune the configurations. 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 slow execution of the workload. Leveraging similarity suggests samples that most likely have the minimum execution time, minimizing the chance of exploring other costly configurations. Finally, the algorithm overhead is negligible: a few seconds to select the most similar workload, transfer the significant parameters and pick the next configuration during tuning. This is due to the small number of samples that I use during the MTGP optimization. The search time for the standalone Tuneful is significantly higher than Similarity-aware Tuneful, since it is the sum of the workload execution times needed by the tuning algorithm to: (i) explore for significant configuration parameters, (ii) tune those to their optimal values.

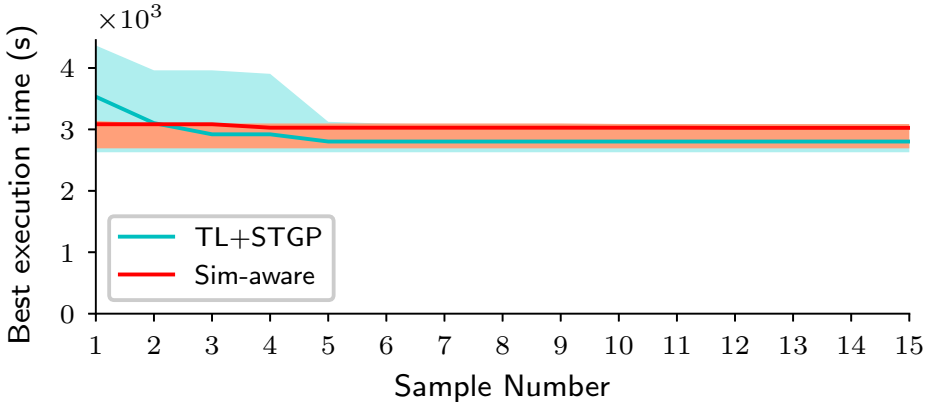
To evaluate the amount of cost saving, I estimate the search cost based on AWS [10] per-second pricing (the cloud infrastructure used to run the experiments). Table 7.3 shows the total cost for tuning the workloads of Bayes, PR, WC, TPC-H, and TS at the median. Overall, the similarity-aware Tuneful achieves a median cost saving of 3.3X and 4.7X compared to Tuneful and Random search.



(a) Bayes-DS2



(b) Bayes-DS3



(c) Bayes-DS4

Figure 7.7: Convergence speed of transfer learning and single tasked GP against the Similarity-aware Tuneful.

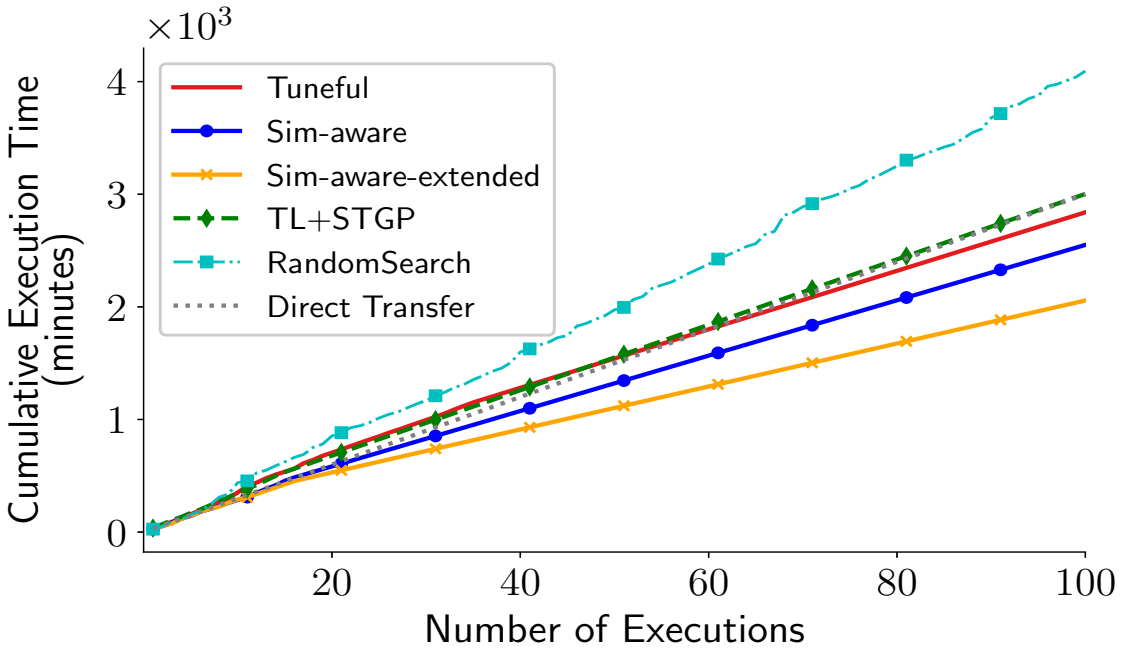


Figure 7.8: Amortization speed of using the different tuning approaches (Bayes-DS3 workload).

Single tasked versus multitasked tuning: In order to evaluate the benefits of MTGP over STGP, I experiment with TL+STGP for three evolving input sizes of the Bayes workloads. This is to show the benefits of sharing the tuning experience (execution samples tried during tuning) of a similar workload in limiting the exploration time. By leveraging similarity-aware MTGP optimization, as shown in Figure 7.7, at the 90th percentile TL+STGP tries configurations that are more costly with a wider inter-percentile range. Similarity-aware Tuneful finds configurations close to the ones found by TL+STGP (within 10%) while not only avoiding the cold start problem of TL+STGP but also bounding the inter-percentile range overall. It is worth noting that the closer the source workload is to the target workload (e.g., Bayes-DS2), the smaller the inter-percentile range of similarity-aware Tuneful and the faster the convergence to near-optimal configuration. This suggests that having more seen workloads in the auxiliary set has the potential to save more search time.

The similarity-aware tuning accelerates the amortization of tuning costs: the previous experiments do 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. Figure 7.8 shows the cumulative execution time of running Bayes-DS3 workload over multiple configurations, as determined iteratively by the tuning algorithms considered. It represents the amortization of the tuning cost under different tuning scenarios. I 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 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 Figure 7.8, 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.

7.2 Summary

This chapter presented an evaluation of a similarity-aware tuning approach that puts together the benefits of the zero knowledge tuning (Chapter 5) with similarity detection and utilization across workloads (Chapter 6).

This evaluation proved that extending BO with workload similarity characteristics can significantly accelerate the amortization of tuning costs and enable effective tuning in a rapidly changing environment. The chapter studied the different ways for learning how to represent workload execution metrics in low dimensional space—a prerequisite step for similarity identification across workloads. I experimented with a diverse set of workloads running on two different clusters. These experiments showed that representing the execution metrics using nonlinear AE outperforms PCA and the linear AE, minimizing information loss. The chapter then evaluated how effective is the proposed similarity-aware tuning approach in accelerating the configuration tuning and cost amortization. This showed that the proposed approach significantly reduced the exploration cost and accelerated the amortization of tuning costs, while finding configurations that are comparable to the ones from the-state-of-the-art (when provided with sufficient auxiliary workloads).

This ultimately enabled a cost-effective tuning of the configuration parameters in a dynamic and rapidly changing environment.

Chapter 8

Conclusion and Future work

The dissertation demonstrated that augmenting BO with incrementally acquired knowledge can enable cost-effective tuning of diverse analytics workloads in a rapidly changing environment. The effectiveness of the proposed approaches firmly shows the need to address the configuration tuning of data analytics differently. The dissertation presented three particular contributions, each of which is a fundamental aspect to enable cost-effective configuration tuning of data intensive computing:

1. It brought into focus the trade-offs that need to be considered when performing tuning without assuming a static environment or workload. Further, I demonstrated, with a series of experiments, a number of overlooked issues that pose a challenge for practical configuration tuning in data analytics frameworks. These issues include: 1) 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. (Chapter 4)

2. It showed that augmenting BO with incrementally acquired knowledge of parameters significance can efficiently tune high-dimensional configurations of diverse analytics workloads, outperforming the-state-of-the-art tuning approaches and saving search time by 2.7X at the median in comparison. This was the first work to propose a *data-efficient* [40] tuning of high-dimensionality configurations. Earlier solutions either require expensive offline phases or address low-dimensionality configurations. To achieve this, I developed an Online Significance-aware Configuration Tuner that leverages incremental SA to identify workload-specific influential configuration parameters in a high-dimensional space, then tunes those parameters using BO. (Chapter 5)

3. It showed that extending BO with workload similarity characteristics can significantly accelerate the amortization of tuning costs and enable effective tuning in a rapidly changing environment—even when tuning configurations of high-dimensionality, which were previ-

ously considered to be impractical with BO. This extension was developed as a Similarity-aware Configuration Tuner that leverages neural encoding of workload execution metrics to detect similarity across workloads. It then shares the tuning knowledge between similar workloads using MTBO. Over diverse analytics workloads, this significantly accelerated both configuration tuning and cost amortization (saving 56% of the search time when compared to an independent single tasked BO based tuning). (Chapter 6 , Chapter 7)

8.1 Limitations

The proposed data-efficient tuning approaches 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 these approaches (e.g., workloads that execute once or few number of times, workloads that execute in a cluster with very limited types of workloads and lacks sufficient similarity knowledge). However, the workloads that are recurrent and subject to variations in input data size or in the resources allocated for execution fit the proposed tuning cost amortization model (§ 4.1). Those will be tuned in a cost-effective manner using the approaches proposed here.

8.2 Future work

As a future work, I will investigate a set of interesting aspects:

- **Extended-knowledge Tuning:** I experimented the impact of tuning the configuration with an extended tuning knowledge (§ 7.1.4). As expected, extending the tuning knowledge showed better performance and enabled finding better configuration with less search time. However, there is a need to gauge when exploiting the readily acquired knowledge in similarity-aware tuning is a better strategy than acquiring more tuning knowledge (i.e., by tuning more workloads using standalone Tuneful as described in Chapter 5). I plan to formulate that as an exploration-exploitation problem, where I find an answer to when should I exploit the existing knowledge rather than waiting to explore more workloads.

- **Experiment with different systems and domains:** The tuning approaches adopted by Tuneful have application beyond Spark. I selected 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). Experimenting Tuneful with other DISC systems such as Storm is an interesting future work. Further, experimenting with systems from other domains such as HPO is an appealing future avenue.

- **Port Similarity-aware Tuning to any GP-based Tuner:** I plan to leverage the proposed approaches in this dissertation to automate the extraction and sharing of tuning knowledge for any GP-based tuner, converting any GP-based configuration tuner in any domain into a

significance and similarity-aware tuner to enable cost-effective tuning.

- **Dynamic identification of configuration search space:** Throughout the dissertation, I considered developer-guided configuration ranges (Table 4.3), which I slightly tweaked across the different cluster (i.e., depending on the cluster characteristics) to enable a faster exploration of the search space. For example, starting the range of spark.executor.cores configuration parameter from 1 core in the AWS 4 nodes cluster would slow down the exploration and it is sensible to start the search space from a larger number of cores — which might not be the case in a larger cluster such as the GCP 20 nodes cluster and the configuration range should be adjusted accordingly. I plan to explore how to automate the exploration of the parameter search space dynamically across different clusters.

- **Dataset Maintenance:** The dataset of workload representation learning (Chapter 6) along with the experiment data (Chapter 5, Chapter 7) are open-sourced and can be found in [14]. This data can be used as a starting point for bootstrapping a database of tuning models and fingerprints against which similarity-aware tuning is possible. I plan to maintain this data 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.

Bibliography

[1] Growth forecast for the worldwide big data and business analytics market through 2020. https://www.idc.com/getdoc.jsp?containerId=US46760920.

[2] Apache Hadoop, 2006. http://hadoop.apache.org/.

[3] Apache Flink, 2011. http://flink.apache.org/.

[4] Apache Storm, 2011. http://storm.apache.org/.

[5] Apache Spark: fast and general engine for large-scale data processing, 2014. https://spark.apache.org/.

[6] TPC-H SQL benchmark, 2014. http://www.tpc.org/tpch/.

[7] Spreadsheet in big data analytics, 2016. https://www.forbes.com/sites/bernardmarr/2016/06/16/spreadsheet-reporting-5-reasons-why-it-is-bad-for-business/#39c9b2ee65e3.

[8] Alibaba cluster data. 2017. https://github.com/alibaba/clusterdata.

[9] Data centers electricity usage, 2017. https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/#5e81d9e55a30.

[10] Amazon EC2 instance Pricing, 2018. https://aws.amazon.com/ec2/pricing/on-demand/.

[11] Amazon EMR, 2018. https://aws.amazon.com/emr/.

[12] Google Dataproc, 2018. https://cloud.google.com/dataproc/.

[13] Hadoop distributed file system, 2018. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[14] Tuneful: Experiment data repository, 2020. https://github.com/ayat-khairy/tuneful-data.git.

[15] Tuneful: project repository, 2020. https://github.com/ayat-khairy/tuneful-code.git.

[16] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 281–294, 2012.

[17] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.

[18] Sonali Aggarwal, Shashank Phadke, and Milind Bhandarkar. Characterization of hadoop jobs using unsupervised learning. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 748–753. IEEE, 2010.

[19] 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.

[20] 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.

[21] 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.

[22] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Performance characterization of in-memory data analytics on a modern cloud server. In *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*, pages 1–8. IEEE, 2015.

[23] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

[24] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. Rfhoc: a random-forest approach to auto-tuning hadoop's

configuration. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1470–1483, 2015.

[25] Richard Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

[26] Yoshua Bengio, Olivier Delalleau, and Nicolas L Roux. The curse of highly variable functions for local kernel machines. In *Advances in neural information processing systems*, pages 107–114, 2006.

[27] Muhammad Bilal and Marco Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 189–200, 2017.

[28] Emanuele Borgonovo and Elmar Plischke. Sensitivity analysis: a review of recent advances. *European Journal of Operational Research*, 248(3):869–887, 2016.

[29] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[30] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization: A survey revisited. *ACM Computing Surveys (CSUR)*, 48(3):1–43, 2016.

[31] Maria Casimiro, Diego Didona, Paolo Romano, Luis Rodrigues, Willy Zwaenepoel, and David Garlan. Lynceus: Cost-efficient tuning and provisioning of data analytic jobs. In *The 40th International Conference on Distributed Computing Systems*.

[32] CL Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347, 2014.

[33] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[34] Peng-Wei Chen, Jung-Ying Wang, and Hahn-Ming Lee. Model selection of svms using ga approach. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, volume 3, pages 2035–2040. IEEE, 2004.

[35] Tatsuhiro Chiba and Tamiya Onodera. Workload characterization and optimization of TPC-H queries on Apache Spark. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 112–121. IEEE, 2016.

[36] Dongjin Choi and Pankoo Kim. Sentiment analysis for tracking breaking events: a case study on twitter. In *Asian conference on intelligent information and database systems*, pages 285–294. Springer, 2013.

[37] D Cournapeau. Sci-kit learn. *Machine Learning in Python. online,[cit. 8.5. 2017]. URL http://scikit-learn. org*, 2015.

[38] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, pages 479–488, 2017.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[40] 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.

[41] Xiaoan Ding, Yi Liu, and Depei Qian. Jellyfish: Online performance tuning with adaptive configuration and elastic container in hadoop yarn. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 831–836. IEEE, 2015.

[42] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[43] Hugo Jair Escalante, Manuel Montes, and Luis Enrique Sucar. Particle swarm model selection. *Journal of Machine Learning Research*, 10(Feb):405–440, 2009.

[44] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andrew Hopper. To tune or not to tune? in search of optimal configurations for data analytics. In *26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20)*, 2020.

[45] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112, 2012.

[46] Domenico Ferrari. Workload charaterization and selection in computer performance measurement. *Computer*, 5(4):18–24, 1972.

[47] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[48] Forbes. Companies adoption of big data analytics, 2017. https://www.forbes.com/sites/louiscolumbus/2017/12/24/53-of-companies-are-adopting-big-data-analytics/?sh=22e0910c39a1.

[49] Holger Frohlich and Andreas Zell. Efficient parameter selection for support vector machines in classification and regression via model-based global optimization. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 3, pages 1431–1436. IEEE, 2005.

[50] Archana Ganapathi. *Predicting and optimizing system utilization and performance via statistical machine learning*. PhD thesis, UC Berkeley, 2009.

[51] Amir Gandomi and Murtaza Haider. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, 35(2):137–144, 2015.

[52] Gartner. What is big data, 2016. https://www.gartner.com/en/information-technology/glossary/big-data.

[53] Jing Gu, Ying Li, Hongyan Tang, and Zhonghai Wu. Auto-tuning spark configurations based on neural network. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[54] Runxin Guo, Yi Zhao, Quan Zou, Xiaodong Fang, and Shaoliang Peng. Bioinformatics applications on Apache Spark. *GigaScience*, 7(8):giy098, 2018.

[55] 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.

[56] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[57] Matthew Hoffman, Bobak Shahriari, and Nando Freitas. On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning. In *Artificial Intelligence and Statistics*, pages 365–374, 2014.

[58] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 660–670. IEEE, 2018.

[59] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. Scout: An experienced guide to find the best cloud configuration. *arXiv preprint arXiv:1803.01296*, 2018.

[60] Zhiyao Hu, Dongsheng Li, and Deke Guo. Balance resource allocation for spark jobs based on prediction of the optimal resource. *Tsinghua Science and Technology*, 25(4):487–497, 2020.

[61] Xingcheng Hua, Michael C Huang, and Peng Liu. Hadoop configuration tuning with ensemble modeling and metaheuristic optimization. *IEEE Access*, 6:44161–44174, 2018.

[62] 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.

[63] F Hutter, L Kotthoff, and J Vanschoren. Automl: methods, systems, challenges (2018). *Book in preparation. Current draft at https://www. automl. org/book/. Accessed July*, 2019.

[64] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

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

[66] Danlin Jia, Janki Bhimani, Son Nam Nguyen, Bo Sheng, and Ningfang Mi. Atumm: Auto-tuning memory manager in apache spark. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2019.

[67] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 66–76. IEEE, 2013.

[68] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–201. IEEE, 2014.

[69] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *2014 IEEE*

International Symposium on Workload Characterization (IISWC), pages 22–30. IEEE, 2014.

[70] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-Keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956. ACM, 2019.

[71] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[72] Anton Kirillov. Spark Internal Architecture, 2016. `http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/`.

[73] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 $USENIX$ Annual Technical Conference ($USENIX$ ATC 18)*, pages 759–773, 2018.

[74] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune: A derivative-free optimization framework for hyperparameter tuning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 443–452, 2018.

[75] Wolfgang Konen, Patrick Koch, Oliver Flasch, Thomas Bartz-Beielstein, Martina Friese, and Boris Naujoks. Tuned data mining: a benchmark study on different tuners. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1995–2002, 2011.

[76] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. 2019.

[77] 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.

[78] 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.

[79] Gil Jae Lee and José AB Fortes. Hadoop performance self-tuning using a fuzzy-prediction approach. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 55–64. IEEE, 2016.

[80] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[81] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 165–176. ACM, 2014.

[82] Mingyu Li, Zhiqiang Liu, Xuanhua Shi, and Hai Jin. Atcs: Auto-tuning configurations of big data frameworks based on generative adversarial nets. *IEEE Access*, 8:50485–50496, 2020.

[83] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: Search-based autotuning of MapReduce. In *European Conference on Parallel Processing*, pages 406–419. Springer, 2013.

[84] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[85] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[86] Zixia Liu, Hong Zhang, Bingbing Rao, and Liqiang Wang. A reinforcement learning based resource management approach for time-critical workloads in distributed computing environment. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 252–261. IEEE, 2018.

[87] Tatiana Lukoianova and Victoria L Rubin. Veracity roadmap: Is big data objective, truthful and credible? 2014.

[88] 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.

[89] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*, volume 37. Springer Science & Business Media, 2012.

[90] Michinari Momma and Kristin P Bennett. A pattern search method for model selection of support vector regression. In *Proceedings of the 2002 SIAM International Conference on Data Mining*, pages 261–274. SIAM, 2002.

[91] Vivek Nair, Zhe Yu, and Tim Menzies. Flash: A faster optimizer for sbse tasks. *arXiv preprint arXiv:1705.05018*, 2017.

[92] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th $USENIX$ Symposium on Networked Systems Design and Implementation ($NSDI$ 15)*, pages 293–307, 2015.

[93] Tiago BG Perez, Wei Chen, Raymond Ji, Liu Liu, and Xiaobo Zhou. Pets: Bottleneck-aware spark tuning with parameter ensembles. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2018.

[94] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[95] Elad Plaut. From principal subspaces to principal components with linear autoencoders. *arXiv preprint arXiv:1804.10253*, 2018.

[96] E Raju, MA Hameed, and K Sravanthi. Detecting communities in social networks using unnormalized spectral clustering incorporated with bisecting k-means. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–5. IEEE, 2015.

[97] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

[98] Carl Edward Rasmussen and Christopher KI Williams. Covariance functions. 2005.

[99] Iran Roudsar and Johor Malaysia. Application of ahp and k-means clustering for ranking and classifying customer trust in m-commerce. *Australian Journal of Basic & Applied Sciences*, 5(12):1441–1457, 2011.

[100] David Reinsel-John Gantz-John Rydning. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 2018.

[101] B Samanta. Gear fault detection using artificial neural networks and support vector machines with genetic algorithms. *Mechanical systems and signal processing*, 18(3):625–644, 2004.

[102] Saba Sehrish, Jim Kowalkowski, and Marc Paterno. Spark and hpc for high energy physics data analyses. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1048–1057. IEEE, 2017.

[103] 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.

[104] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *Proceedings of the VLDB Endowment*, 7(13):1319–1330, 2014.

[105] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[106] Ilya M Sobol. On quasi-monte carlo integrations. *Mathematics and computers in simulation*, 47(2-5):103–112, 1998.

[107] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[108] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012, 2013.

[109] Anand Shanker Tewari, Tasif Sultan Ansari, and Asim Gopal Barman. Opinion based book recommendation using naive bayes classifier. In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, pages 139–144. IEEE, 2014.

[110] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[111] Ruben Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguadé, Jesús Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 299–306. IEEE, 2015.

[112] Rohit Tripathy, Ilias Bilionis, and Marcial Gonzalez. Gaussian processes with built-in dimensionality reduction: Applications to high-dimensional uncertainty propagation. *Journal of Computational Physics*, 321:191–223, 2016.

[113] Christos Troussas, Maria Virvou, Kurt Junshean Espinosa, Kevin Llaguno, and Jaime Caro. Sentiment analysis of facebook statuses using naive bayes classifier for language learning. In *IISA 2013*, pages 1–6. IEEE, 2013.

[114] 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.

[115] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion 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, 2016.

[116] Julien Villemonteix, Emmanuel Vazquez, and Eric Walter. An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*, 44(4):509, 2009.

[117] Alex Hai Wang. Detecting spam bots in online social networking sites: a machine learning approach. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 335–342. Springer, 2010.

[118] Chao Wang, Jie Lu, and Guangquan Zhang. A semantic classification approach for online product reviews. In *The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05)*, pages 276–279. IEEE, 2005.

[119] 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.

[120] Han Wang, Setareh Rafatirad, and Houman Homayoun. A+ tuning: Architecture+ application auto-tuning for in-memory data-processing frameworks. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 163–166. IEEE, 2019.

[121] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*, pages 488–499. IEEE, 2014.

[122] Yandong Wang, Robin Goldstone, Weikuan Yu, and Teng Wang. Characterization and optimization of memory-resident mapreduce on hpc systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 799–808. IEEE, 2014.

[123] Md Wasi-ur Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K DK Panda. Mr-advisor: A comprehensive tuning, profiling, and prediction tool for mapreduce execution frameworks on hpc clusters. *Journal of Parallel and Distributed Computing*, 120:237–250, 2018.

[124] John A Weymark. Generalized Gini inequality indices. *Mathematical Social Sciences*, 1(4):409–430, 1981.

[125] John Wilkes and Charles Reiss. Details of the clusterdata-2011-1 trace. 2011. `https://github.com/google/cluster-data`.

[126] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 383–392. IEEE, 2016.

[127] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465. ACM, 2017.

[128] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. Towards machine learning-based auto-tuning of mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 11–20. IEEE, 2013.

[129] 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.

[130] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[131] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[132] Petar Zečević, Colin T Slater, Mario Jurić, Andrew J Connolly, Sven Lončarić, Eric C Bellm, V Zach Golkhou, and Krzysztof Suberlack. Axs: A framework for fast astronomical data processing based on apache spark. *arXiv preprint arXiv:1905.09034*, 2019.

[133] Yuyu Zhang, Mohammad Taha Bahadori, Hang Su, and Jimeng Sun. Flash: fast bayesian optimization for data analytic pipelines. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2065–2074, 2016.

[134] Yao Zhao, Fei Hu, and Haopeng Chen. An adaptive tuning strategy on Spark based on in-memory computation characteristics. In *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, pages 484–488. IEEE, 2016.

[135] 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.

[136] Paul Zikopoulos and Chris Eaton. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

Appendix A

Experiment Reproducibility

A.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 A.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 A.1: Tuneful example usage

A.2 Experiment data and workload representation learning dataset

The experiment data of Tuneful and the-state-of-the-art approaches are publicly available on [14] 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.

The built dataset of execution metrics—used for workload representation learning—is publicly available on the project data repository [14] under dataset folder.