A literature review of parameter tuning

within the context of big data processing frameworks

by

Mayank Tiwary

B. Engineering, 2016

AN ESSAY SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2023

Abstract

This essay explores the challenges and solutions in automatic machine learning (ML)-based configuration tuning of Big Data processing frameworks. The use of Big Data processing systems has increased due to the availability of high performance computing resources and the realization of the value of data in making business decisions. However, configuring the parameters of these systems is difficult due to the high diversity of workloads, rapid growth of data, changing software and hardware infrastructures, and inter-dependency between the configurations. The literature has extensively explored ML-based tuners to solve the auto-tuning problem. In this essay we focus on answering four survey questions related to model generalization, search-space pruning, workload characterization, and tuning benchmarks. We follow a Systematic Literature Review (SLR) plan to answer the survey questions. As part of our SLR plan, we finalized 47 research papers and articles (published between 2012 and 2022) from different database sources. Using these papers, we perform an extensive literature review and summarize our findings.

Lay Summary

Given the numerous configuration parameters significantly impacting performance, manually tuning them can be challenging and error-prone. Therefore, machine learning-based systems have become indispensable in automating this process, constantly adapting to changing workload patterns and identifying the best configurations. Moreover, the tuning systems analyze workload execution-related observational data generated by Big Data frameworks, enabling them to optimize the performance. This, in turn, leads to improved performance, allowing organizations to process and analyze massive amounts of data more efficiently, resulting in better decision-making and insights. However, there are several challenges while using Machine Learning algorithms to tune the configurations of Big Data processing frameworks.

In this essay, we define four survey questions (as part of a systematic literature review plan) based on the challenges of Big Data processing frameworks. Further, we do an extensive literature review to answer the survey questions and summarize our findings.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank Ivan Beschastnikh and Thomas Pasquier for their invaluable support and feedback on this essay.

Dedication

To my parents.

Chapter 1

Introduction

The first Big Data processing framework, Apache Hadoop, was released in 2011 by the Apache Software Foundations [6]. Other Big Data processing frameworks like Apache Spark [7], Storm [9], and Flink [4] were released soon after. Since that time, many organizations have adopted Big Data processing systems. Different sectors started this adoption because high-performance computing resources, such as graphics processing units (GPUs) and cloud computing, made it easier to process large datasets and train complex machine learning models. This has made it possible to use more advanced algorithms and models that were previously infeasible. Additionally, organizations started realizing the value of data, which can help inform various business decisions. Early data processing challenges were due to high volume, velocity, variety, variability, and veracity. Big data processing systems, or frameworks, were designed to manage these 5 "Vs." (Volume, Velocity, Variety, Veracity, and Value) [29].

However, Big data processing frameworks expose hundreds of tunable parameters that must be optimized [27]. These configuration parameters control execution behavior. For example, in Apache Spark, *spark.executor.memory* controls the amount of memory available for the underlying container that will process the data, *spark.serializer* controls the type of Serializer to use (Java default or Kryo Serializer), etc. *Configuration* tuning of a Big Data framework refers to optimizing the configuration parameters for the framework to achieve optimal performance and resource utilization across a cluster of machines. Configuring the parameters of

these frameworks has continued to bedevil developers and administrators.

Configuration tuning of these systems is difficult for the following reasons:

1. The high diversity of the workloads [34, 55]: It is difficult for a human to find the optimal configuration values for hundreds of configurations.

2. Rapid growth of data: As data size increases, these systems need to be reconfigured. The optimal configuration values for executing a workload that processes a relatively smaller volume of data might not work for the same workload when the data is larger [35].

3. Changing Software and Hardware infrastructures: When the underlying software or hardware stack changes, the previously found optimal configuration values might not work [43].

4. Inter-dependency between the configurations [24]: There are often dependencies between configuration parameters. This dependency can be within or across the software components. And this is often not clearly documented.

When these frameworks are misconfigured, the systems usually continue executing correctly but with degraded performance. Under-provisioned cluster setups have shown degraded execution times of 12x [15], and misconfigurations on Apache Spark can lead to a slowdown of 89x [73]. Hadoop was found to be slower than parallel database systems for multiple data-intensive analytical workloads [59]. However, Hadoop can perform well when its configuration is tuned. Auto-tuning of systems improves the efficiency of these systems and can tune the configuration per workload without much human intervention. Out of the different types of tuners, Machine Learning (ML)-based tuners have been explored extensively in the literature to solve the auto-tuning problem.

In this MSc essay, we perform a literature survey of ML-based tuners for Big Data frameworks. There is a class of ML-based tuners known for their ability to reuse information (knowledge transfer) gained from tuning a system that executes a specific workload to tune the same system that executes a similar workload [34, 68]. Cloud-based service providers, which provision thousands of instances of Big Data processing framework, can easily use the knowledge transfer capability [34,

75] of ML-based tuners as they can generate observational data (to train the ML-based tuners) from different tuner deployments (a tuner deployment is responsible for tuning one or more Big Data processing frameworks).

Other tuners [43], like cost-modeling- or simulation-based tuners, need a profound understanding of the frameworks to tune. Rule-based tuners use rules and cannot guarantee an optimal solution [43]. Experiment-based tuners [**?**] repeatedly run an application or a workload on a cluster using various configurations until it reaches a suitable set of settings. This class of tuners cannot perform knowledge transfer. By contrast, ML-based tuners do not need a deep understanding of the system and can find the optimal configurations in the least number [35, 50] of experiments or tuning iterations.

Another motivation for us to survey ML-based tuners is the extensive list of recent publications focusing on using ML for tuning configurations. Hence, our survey focuses on studying the automatic ML-based tuners that treat the underlying system as a black box with little understanding of its internals.

Chapter 2

Systematic Literature Review Plan

With the proliferation of Big Data applications in recent years, the need for efficient and effective management of large-scale data processing has become increasingly important. One of the key challenges in this area is tuning the configuration parameters of Big Data frameworks to achieve optimal performance. Machine learning-based tuning techniques have emerged as a promising approach to automate the tuning process and improve the overall efficiency of Big Data frameworks. This chapter defines the specific research questions related to ML-based tuning frameworks we want to explore as part of our survey.

2.1 Introduction

Any ML-based tuner shares certain characteristics, irrespective of the underlying framework it aims to tune. Such a tuner typically has a core tuning engine with a search space explorer and a performance prediction model (also shown in Fig. 3.1). Other optional components, such as search-space pruning [45], trim down the non-influential search space; and a workload classifier, classifies a given workload to a similar workload seen in the past. We summarize the main components of ML-based tuners as follows:

- **Core Tuning Engine**: This component is responsible for sampling optimal configurations from the search space using a trained performance prediction model. After every observation, the tuner re-trains the performance prediction models as the tuner starts its exploration.
- **Search Space Pruning**: This component aims to reduce the total search space for effective training of the tuning engine. This saves a lot of time the tuner would have spent exploring the non-influential regions.
- **Workload Classifier**: The workload classifier classifies a given execution profile (or workload execution data) to a class of execution profiles that it has already seen in the past. Using the classification result, the tuner will reuse the previously trained model or train a new one using the past workload's observational data. This helps in generalizing the tuner to reuse tunings it has previously learned.

Based on the above components of the tuning framework, we describe the challenges, and based on these challenges, we define our survey/research questions.

A tuning framework is always expected to **generalize** [77] because it should be able to tune various workloads within a small number of iterations. Typically, tuning frameworks use knowledge transfer to leverage past tuning experiences to achieve better tuning. Each type of tuner performs knowledge transfer differently. For example, the Bayesian Optimization (BO) style tuners can easily adapt to a similar pattern as described in RGPE [36, 77] (ranking-weighted Gaussian process ensemble). RGPE combines Gaussian process models learned from past tasks with different weights computed through relative ranking loss, allowing for generalization across diverse workloads and hardware configurations. Leveraging past tuning experiences is important for any tuning framework as it would help to amortize the tuning costs. Previous surveys [27, 43, 45] do not specifically talk about the challenges and effectiveness of transfer learning. **Hence, in this survey, we summarize the tuning frameworks based on the performance prediction model and study the challenges related to the knowledge transfer of different tuning frameworks (R1).** However, the discussion on the challenges of knowledge transfer is motivated by a similar discussion in the survey on database tuners conducted by Zhang et al. [77].

In the literature, various techniques have been proposed to find the most optimal set of parameters for search-space pruning that can have a significant impact on the execution time of a job. Pruning the non-influential parameters is important because, for most workloads, only a handful of parameters impact the execution time [34, 43]. The pruning techniques mainly include - manual approach [63], search-space division based on parameter independence [57], incremental (stepwise) pruning [51], sensitivity analysis [46] and recursive feature elimination with approximation-based approaches [35]. Prior surveys [27, 43, 45] do not discuss the trade-offs (like the cost of finding the significant configuration parameters vs. tuning cost amortization). **Hence, in this survey, we categorize the different pruning techniques based on the trade-offs and the features used for pruning (R2).** The discussion on the challenges of pruning techniques is motivated by a similar discussion in the survey on database tuners conducted by Zhang et al. [77].

Multiple definitions exist of a big data workload [34, 37, 47, 72]. And every definition relies on the granularity of metrics/features being used, hardware and software stacks used to find the statistical differences/variations. Also, with every new definition of a Big Data workload, a new workload characterization or classification technique is required [34, 37, 47, 72]. The surveys done in the past [27, 43, 45] on performance optimization of big data frameworks using parameter tuning do not systematically categorize the big data workload classification techniques, the level of granularity, and features used for classification. In the previous surveys [43], [27], the works have implicitly just mentioned the name of the techniques used to characterize the workloads. **Hence, in this work, we systematically categorize the workload classification techniques used, the set of features used, the definitions & granularity at which these techniques have been proposed (R3).** The discussion on the workload classification techniques is motivated by a similar discussion in the survey on database tuners conducted by Zhang et al. [77].

The surveys done in the past on parameter tuning [27, 43, 45] on performance optimization of big data processing frameworks have not discussed the evaluation and benchmarking of the tuning frameworks. The evaluation metrics used for evaluating the tuners help explain a tuner's goal. Further, the benchmarks used help us understand the kind of scenarios that the tuner is designed to manage the best.

For example, the tuner evaluation benchmarks might not execute a production-like workload. This will help us understand the effective scope of the tuner in tuning the big data processing frameworks. **Hence, in this work, we survey the metrics chosen for evaluating the tuning frameworks and the different benchmarks they use (R4).** Costa et al. [27] discuss the benchmarking of the Big Data frameworks instead of benchmarking the tuning frameworks.

We summarize the research questions from above in the next section.

2.2 Survey questions

Based on the above discussion and prior surveys done in the area of using ML for parameter tuning of the big data frameworks, we defined the following four survey questions for our study:

- **(R1)** What are the different performance prediction models based on the tuning frameworks, and the challenges related to the generalization of different tuning frameworks?
- **(R2)** What are the different search-space pruning techniques, their pros & cons, and what kind of trade-offs do these techniques expect the users to take care of?
- **(R3)** What are the techniques and features used to classify/characterize the workloads?
- **(R4)** Which benchmarks and metrics have been used in the literature to evaluate tuner performance?

To study the above components in depth, we aim to do a taxonomy-based survey covering meaningful categories and sub-categories for each sub-component. This study will use the similar Systematic Literature Review (SLR) approach [76]. In the first phase, we decide our scope of review, survey questions, list of conferences and journals (data sources), inclusion, and exclusion criteria. Based on the defined scope, we do the literature review in the second phase to answer the above survey questions.

In the next section, we define the criteria based on which we select or reject the research papers or articles we would use for the literature review.

2.3 Inclusion and exclusion parameters

We decided to filter papers based on the publication year as the first criterion. We pick those research papers which have a publication year later than 2011. The rationale for choosing the parameter tuning papers after 2011 is that the big data processing frameworks we surveyed were released after 2011. Apache Software Foundation released the first version of Apache Hadoop in December 2011.

The second criterion to filter the paper would be based on the paper's aim and objective. We only select the articles aiming to tune the parameters of big data processing frameworks. There are several papers/works which aim to improve the performance of big data processing frameworks by optimizing the query plan, partition size, selective caching, etc. However, as per the scope of this study, we only consider the papers that try to tune the parameters of big data processing frameworks in an automated fashion using machine learning.

The big data processing frameworks we consider in this survey are the same as those considered in previous surveys [27, 43]. These include Apache Spark, Apache Hadoop, Apache Yarn, Apache Flink, Apache Storm, and Heron. We are focusing on these specific frameworks because of their success in the industry, where these frameworks are ranked in the top 10 based on their usage and adoption [11].

In the next section, we define the database sources from which we would search the research papers or articles and then filter them based on the set of defined criteria.

2.4 Academic database sources for the survey

Based on the big data frameworks that we consider to survey in this work, we use the following keywords:

1. Performance optimization
2. Big data tuning

3. Big data parameter tuning
4. Spark parameter tuning
5. Hadoop parameter tuning
6. Yarn parameter tuning
7. Flink parameter tuning
8. Storm parameter tuning

Using the keyword above, we searched the following digital libraries to gather the set of relevant papers:

1. IEEE Xplore
2. Science Direct (Elsevier)
3. Springer
4. ACM Digital Library

After searching the digital libraries and reading the survey papers, we compiled the final list of papers. For articles published in conferences, we picked the conferences with a minimum of B ranking per www.conferenceranks.com. As per the search criteria we defined, we found 47 relevant papers (summarized in Table 2.1). The rest of our survey will use these papers.

We defined the survey questions, inclusion and exclusion parameters, database sources, and final research papers and articles in the previous sections. In the next section, we conclude the chapter.

2.5 Conclusion

In this chapter, we have discussed the main components of ML-based tuners, which include the core tuning engine, search space pruning, and workload classifier. These components are essential in enabling the tuner to explore the search space and learn optimal configurations efficiently. There are several challenges associated with each of these components, such as the high dimensionality of the search

Table 2.1: Summarizing the selected research papers and venues

Publication Venue	Type	Papers	Publication Year
SigMod	Conference	[59]	2009
ACM CGO	Conference	[57]	2012
ICCCN	Conference	[49, 60]	2012, 2018
ICAC	Conference	[51, 66]	2012, 2019
EDBT	Conference	[32]	2013
Euro-Par	Conference	[56]	2013
IEEE IISWC	Conference	[47]	2014
VLDB	Conference	[55, 63]	2014, 2020
Int. Congress on Big Data	Conference	[23]	2015
IEEE TPDS	Journal	[18, 42, 72]	2015, 2017, 2021
Int. Conference on (Big Data)	Conference	[16, 34, 37, 67, 74]	2015, 2018, 2019, 2020
HPCC	Conference	[38, 70]	2016
IEEE PACT	Conference	[48]	2016
IEEE TBD	Journal	[25, 53]	2016, 2019
IEEE TKDE	Journal	[69]	2017
SoCC	Conference	[20]	2017
IEEE FASS	Workshop	[65]	2017
ASPLOS	Conference	[73]	2018
IEEE ICC	Conference	[40]	2018
IEEE ICDE	Conference	[64, 75]	2018, 2021
Elsevier Big Data Research	Journal	[27, 39, 41]	2018, 2021, 2022
ICDCS	Conference	[33]	2019
Elsevier FGCS	Journal	[21]	2019
ACM SIGKDD	Conference	[35]	2020
IEEE TSE	Journal	[50]	2020
IEEE Cloud	Conference	[71]	2020
IEEE TSC	Journal	[79]	2021
IEEE ISCC	Conference	[62]	2021
Elsevier JSS	Journal	[26]	2021
Springer Cluster Computing	Journal	[31]	2022
Springer Applied Intelligence	Journal	[19]	2022
Elsevier JPDC	Journal	[22]	2022
Elsevier KBS	Journal	[28]	2022

space, the difficulty of accurately modeling performance, and the need for effective workload classification. These challenges have motivated several research efforts in the area of ML-based tuning. We have identified our survey questions that remain to be addressed. These questions represent essential research directions for advancing the state-of-the-art in ML-based tuning and improving the performance of complex systems. The next set of chapters will answer our survey questions. The next chapter discusses and answers the first research question (**R1**).

Chapter 3

ML-based Tuning Frameworks and Challenges

Big Data frameworks are developed for managing massive and intricate datasets by distributing the workload across multiple nodes. To ensure peak performance, it is essential to tune the configuration settings of these frameworks for the execution of a variety of workloads. However, this can be an arduous task that requires specialized knowledge of the framework and a comprehensive understanding of its configuration options. Furthermore, different Big Data framework configurations can be tuned by tuning frameworks, adding to the complexity of transferring knowledge across platforms. **Consequently, this chapter explores the challenges of transferring tuning knowledge in Machine Learning (ML) based tuners.** Specifically, this chapter aims to answer the first research question (**R1**) raised in Chapter 2.

3.1 Introduction

In literature, there are various tuning framework types [43] that aim to tune the configurations of Big Data frameworks. Compared to other tuning frameworks like cost-model-based tuners and experiment-based tuners [43], ML-based tuners have three clear advantages:

1. They treat the underlying system to tune as a black box. ML-based tuners

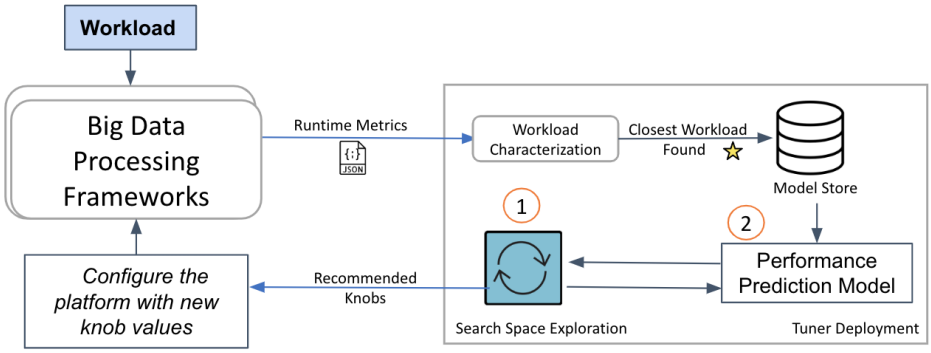

Figure 3.1: A generic architecture for ML-based tuner. This architecture is motivated by the architecture used for database tuners [77]

can tune the configurations of the big data framework with a minimum understanding of the internals.

2. The tuners can leverage past tuning experiences. They have the capability to reuse the trained performance prediction models from the past to tune the system for a new or unknown workload.

3. The tuners use performance prediction models to explore the search space. This helps in saving time as the performance prediction model can predict the outcome without executing a candidate configuration, unlike the experiment-based tuner, which has to execute every configuration candidate to get the optimal solution.

A generic ML-based tuner (Fig. 3.1) mainly has two core components [54], [17]: (1.) search space exploration algorithm, which explores the search space (i.e., generates candidate configurations and returns the best among them), and (2.) performance prediction model, which can predict the metric which the tuner wants to optimize.

As shown in Fig. 3.1, the search-space explorer generates configurations to explore. These configurations are used to execute the workload on the Big Data framework. The tuning framework aims to generate the best values for configurations that can lead to optimal performance for workload execution. The workloads

are external to the tuning framework and can also change. Once any workload executes, it generates runtime metrics or observational data. Using the generated observational data, the tuners can optionally classify the workload execution data to the closest one seen in the past. Based on the closest source found, the tuner picks up a relevant performance prediction model from the model store. The search space explorer further performs the exploration using the model and generates new configurations to explore. Here workload characterization is an optional step, and tuning frameworks can opt not to use it, which would result in training performance models from scratch instead of reusing models trained in the past. Tuners can optionally rank or prune the configuration parameters with the least impact on the outcome. This process is similar to the feature pruning in any classical ML task.

The deployment of ML algorithms in a tuning framework can optimize Big Data framework deployments by predicting performance based on previous observational data. Still, challenges arise when attempting to leverage past tuning experiences due to the architecture and framework of the performance prediction model. A typical tuning framework deployment runs a set of ML algorithms that can be used to tune the configurations for one or many Big Data framework deployments. The performance prediction models in the tuning framework learn the system behavior of executing different workloads in the ML-based tuning framework using the generated observational data. Workload classification and characterization algorithms in the tuning framework are responsible for detecting new unknown workloads. This means the tuner would reuse a previous performance prediction model if the new workload is similar to the workload with which the performance prediction model was trained earlier. However, the core challenges of leveraging past tuning experiences depend on the architecture and framework of the performance prediction model. Hence, in the next section, we summarize the performance models based on the tuning frameworks used in the literature and then discuss the knowledge transfer challenges in Section 3.3.

3.2 Summarizing the performance prediction models

We found that tuning frameworks use different performance prediction models in the literature. In this section, we discuss each type of performance prediction

model used along with bit details of the respective tuning framework. Then, based on the identified performance prediction model types, we discuss the knowledge transfer challenges in the next section. This chapter sets the context for the discussion of challenges related to the first research question in chapter 2 (**R1**). We categorized the performance prediction models into six following groups:

1. Gaussian Process Regression-based performance prediction models
2. Regression-based performance prediction models
3. Tree or Ensemble-based performance prediction models
4. Classifier-based performance prediction models
5. Performance predictions in Reinforcement style tuning frameworks

We have summarized the performance prediction models in Table 3.1. Further, we discuss the performance models and start with the Gaussian Process Regression-based performance prediction model in the next section.

Gaussian process regression-based performance prediction models

Gaussian Process Regressors (GPRs) have been widely used to model the performance of a black box system mainly because they measure uncertainty over the prediction. This helps the search space explorer to model the unknown and uncertain regions. GPRs typically assume that the data distribution is Gaussian.

In Tuneful [35], the authors use GPR to model the performance of Spark. Here the authors model Spark's performance using configuration parameters as the features and execution time as the outcome. Further, in Tuneful, the authors use the Expected Improvement-based MCMC (EI-MCMC) algorithm to explore the uncertainty over the GPR.

In [31], authors proposed a tuning framework that tunes not only Spark's configuration parameters but also the kernel's and JVM's configuration parameters. It uses an importance score to rank the parameters and a dropout Bayesian Optimization algorithm (which internally uses a GP regression) to address the challenge of high dimensionality (or huge search space).

Table 3.1: Summarizing the ML-based tuners based on performance prediction models and search space exploration algorithm

Overall Approach	Prediction Model Type	Search Space Exploration
GPR Based	Single GP [35]	Expected Improvement MCMC
	Single GP [33]	Expected Improvement
	Single GP [31]	Expected Improvement
	Single GP [49]	No Searching Needed
	Two GPs [74]	State-Transition based
	Single GP [65]	Genetic Algorithm
	Multiple GPs [64]	Multi-Objective Gradient Descent
	Multi-Task GPs [34]	Expected Improvement
Regression-based	Support Vector Regressor [51]	Pattern Search Algorithm
	NN based regression [22]	Global & Local Search
	Support Vector Regressor [53]	Heuristic Based Search
Tree or Ensemble-based	Random Forest [16]	Multi-Bound Search Algorithm
	SGB Trees [42]	Genetic Algorithm
	Hierarchical Modelling [73]	Genetic Algorithm
	AdaBoost Algorithm [26]	Genetic Algorithm
	Random Forests [18] [71]	Genetic Algorithm
	Stacked Neural Network [79]	NSGA-II
	PeiceWise Regression [25]	Heuristic Based Search
	EDK Regression [69]	No Searching Needed
	Random Forests [23]	Random Sampling + Hill Climbing
Classifier-based	XGBoost Classifier [28]	No Searching Needed
	K-Nearest Neighbour [48]	No Searching Needed
	SVM [66]	Genetic Algorithm
	Decision Tree Classifier [70]	Recursive Random Search
RL-style performance prediction models	Q-Learning	Reinforcement Learning [19]
	Similarity Matrix + SVD [21]	Two Phase Random Search
	Population-based search	Evolutionary MCMC [50]

Kadirvel and Fortes [49] after multiple evaluations on different workloads, they selected the five most important features "Map Input Size, Number of Reduces, I/O Sort Record Percent, Map-Reduce Parallel Copies" [49]. Using these features, the authors found that Regression by Multilayer Perceptron, Discretization, Gaussian Process Regression, and M5 Pruned Model Tree were the models which showed encouraging results in predicting the performance of map-reduce jobs.

To tune Storm configurations, Zacheilas et al. [74] proposed a tuning framework that tries to find the optimal Esper engines to use for balancing the costs (cost of missing tuples, monetary cost of the computing resources, and the cost of rebalancing the topology). To tune this parameter, it uses two GPRs. Zacheilas et al. state that "The first GP model will have as target variable y the Latency values and as features will use a multidimensional vector that contains the timestamp of the measurement, the time of day, the day of week and the number of engines." [74]. The second GPR predicts the number of input tuples (future load). Further, they model a state transition model where transiting from one state to another quantifies the impact of the transition on performance.

The tuning framework proposed by Trotter et al. [65] tunes the total number of executors and worker processes. It reads metrics from JMX - CPU load, memory usage, and Java class related profiling data. And from Storm, it reads each executor's processing throughput within a Storm topology. Further, using this collected performance data, the tuner uses GPR for performance prediction and Genetic Algorithm (GA) to explore the search space and find the optimal configurations. Also, they bootstrap the tuner's exploration with constraints (added to the optimization problem) from the standard rules of thumb.

The tuner proposed by Song et al. [64] optimizes k different objectives. The users use the framework to specify the objectives, such as latency, throughput, CPU utilization, memory utilization, etc. And then, the framework forms a multi-objective problem and solves it. It proposes a new Progressive Frontier approach to compute a decent Pareto frontier in practical time. It transforms the multi-objective functions into single objectives and solves them individually. With the decoupling, it can train multiple performance models (GPs and DNNs) offline, which would eventually get consumed by the optimization pipeline.

Fekry et al. [34] propose a tuning framework that uses MultiTask Gaussian Pro-

cess (MTGP) to model individual workloads as tasks. MTGP learns the correlation between the tasks and inputs. In this work, authors specifically use MTGP with Bayesian Optimization to utilize the knowledge transfer of significant parameters from past similar workloads.

Next, we discuss the regression-based performance prediction models (apart from Gaussian Regression).

Regression-based performance prediction models

Unlike a Gaussian Process Regressor, which assumes that the data distribution is a Gaussian Distribution, other regression algorithms may or may not assume any distribution to be specific. In literature, various other regression methods have also modeled the performance of Big data frameworks.

The tuning framework proposed by Lama and Zhou [51] uses an SVM-based regressor to predict a job's relative performance for jobs grouped by k-medoid. The framework trains a performance model for every cluster formed using k-medoid and keeps track of the significant configuration parameters detected by the step-wise regression approach. It uses a pattern search algorithm to find the optimal configuration values.

The tuning framework proposed by Chen et al. [22] tunes the Hadoop level configuration parameters and the resource level configuration parameters. The core of the tuner consists of a Neural Network based prediction model and a proposed global and local search-space exploration strategy. The search-space exploration algorithm first generates (samples uniformly) random configuration values and performs a global search over them using the performance prediction model. It performs a local search on the prediction model based on the selected global best.

The stream processing system's performance is evaluated in terms of the latency of tuple processing. Modeling the execution latency of a tuple is challenging as it involves execution at multiple Processing Units PUs. The tuning framework proposed by Li et al. [53], Li et al. uses the topology of execution DAG to model the tuple processing latency of a thread (task) and tuple transfer latency using Support Vector Regression (SVR). This scheduling algorithm uses the prediction, and a heuristic search to assign threads, total cores, and memory to machines (schedul-

ing).

Next, we discuss the tree-based performance prediction models.

Tree or Ensemble-based performance prediction models

Many different tuning frameworks have used decision tree-based regressors to model the performance of black-box systems. The trees-based regressors have been widely used for performance predictions when the features (configuration parameters) are categorical or discrete in nature [77]. Compared to any traditional database system, the execution flow in the big data processing framework is divided into stages. Executing the stages in serial or parallel depends on the workflow scheduler. Some tuning frameworks model individual stage execution and then aggregate the total execution time from individual stage-level predictions. This benefits in terms of higher prediction accuracy, as modeling the individual stages is easier than modeling the end-to-end execution time. Some tuning frameworks also use an ensemble of regressors or classifiers for modeling the performance, where the final prediction is decided based on the weights or scores of individual models in the ensemble.

To generate sufficient samples for the predictive model of the tuning engine, work by Bao et al. [16] utilizes the computation properties of the workload and the internal communication patterns between different machines in the cluster. Internally, the framework trains a Random Forest Regressor as the performance prediction model and Latin Hypercube Sampling to explore the search space.

Guo et al. [42] propose a tuning framework that consists of two main components, i.e., synthetic training data generator and Genetic Algorithm based tuner, which uses a performance model to explore the search space effectively. It trains a Generative Adversarial Network (GAN), which generates training data for the performance model using these sampled data points. In this framework, they use Stochastic Gradient Boosted Regression Trees as the performance model and use a Genetic Algorithm to explore the search space.

Yu et al. [73] use a tuning framework where the search-space exploration is done using GA. The performance model is a Hierarchical Model (HM) of regression trees. The submodels are trained with more optimized hyperparameters (num-

ber of trees in the random forest, learning rate, tree complexity). Here the submodels are arranged in hierarchy-based first, second, and higher-order levels (at each level, the model's variance reduces). HM model picks the configurations randomly at any level, and the whole process continues till a target accuracy from the overall HM is achieved.

The tuning framework by Cheng et al. [26] proposes a decomposition-based multi-objective optimization (MOEA/D). To optimize resource usage cost, and execution time, MOEA/D also uses a performance prediction model to solve the multi-objective problem. Overall, the MOEA/D is responsible for exploring the configuration space, and for every exploration, the performance prediction model (AdaBoost model) evaluates the fitness of the given candidate.

The tuning framework proposed by Bei et al. [18] utilizes an ensemble of Random Forest regressors to model the total execution time of the map-reduce jobs. To model the total job execution time, the tuning framework models the map phase separately with four regressors to model the read phase, map phase, collect phase, and merge & spill phase, respectively. Similarly, for the reduce phase, they model the shuffle & sort phase, merge phase, reduce phase, and write phase, respectively. With this ensemble of regressors, where the features are the configurations and the outcome is a performance metric, the exploration is performed using the GA approach.

The tuning framework proposed by Chen et al. [23] uses Random Forest Regressor to predict total job execution time. Here they use two predictors: one for predicting the total execution time of the map stage where the input feature is only the configuration parameters that impact the map phase, and the second predictor is used to predict the execution time of the reduce phase, where the input feature is the configuration parameters that impact the reduce phase. After predicting the execution time of the individual phases, they aggregate the individual times into the total execution time. They use Random Sampling with a Hill Climbing algorithm to explore the search space.

The tuning framework proposed by Zong et al. [79] targets optimizing the configurations when multiple workloads execute concurrently on the same cluster. It proposes a multiobjective optimization framework that optimizes execution time and resource usage across the cluster. They propose a stacked neural network ap-

proach to model the performance. Further, they use NSGA-II to solve the multi-objective optimization problem.

To tune the configuration parameters of Hadoop, a tuning framework proposed by Wang et al. [71] builds Random Forest-based performance models which can separately predict the execution time for map tasks, reduce tasks and shuffle timings. Using an ensemble of regressors and a packing algorithm, it can predict the total job execution time. Further, it uses a Genetic Algorithm approach to explore the search space.

In the framework proposed by Chen et al. [25], the authors suggest breaking down the total search space into a d-dimensional mesh using the Delaunay Triangulation method. Then, for each d-dimensional mesh, they build a piece-wise linear regression model. To get the best samples for training the performance model, Chen et al. propose an adaptive sampling method where they state that the proposed method "heuristically searches the area with significant runtime changes and more unknown configurations".

The tuning framework by Wang et al. [69] aims to optimize both latency and resource usage. It uses an ensemble of ML algorithms - Naive Bayes, HoeffdingTree, Online bagging, and Nearest neighbors. The framework models resource usage with data-, plan-, operator-, and cluster-level features. They also propose an outlier removal algorithm, increasing the training data quality. Using the ensemble of regressors, they predict compute CPU & Memory usage, latency, and throughput from a given data sample.

Next, we discuss the classification-based performance prediction models.

Classifier-based performance prediction models

Search space exploration algorithms evaluate a candidate configuration using performance prediction models. In literature, some tuners have used classification-based performance models which can predict if a given configuration can lead to performance improvement or not. Some tuners in the literature also use classifiers to predict the optimal configuration values.

The tuning framework proposed by Daud et al. [28] does not need re-training performance models when the workload changes. This tuner is overengineered for

Twitter link prediction workloads (Daud et al. specifically state "Graph Clustering (GC), Overlapping Community Detection (OCD), and Redundant Graph Detection (RGD)" algorithms). The tuner's core has an XGBoost classifier, which can classify workload and resource consumption (of master and worker nodes) related features to executors needed per node value. From the prediction (which is executors needed per node value), the framework use rules which can set the values of other configuration parameters. No search-space exploration is performed here as the classifier can predict the optimal configuration directly.

Jia et al. [48] propose a tuning framework that tunes the processor's SMT configuration for different Spark Workloads. This operates at the state level, where for any given stage in a big data processing framework, it aims to predict the optimal value of the SMT configuration. The operations of the tuning framework are split into two phases. The first phase (offline) trains the prediction model where the features are handpicked set of processor's hardware counters, and the label is the SMT configuration. Once it has the offline trained model, it integrates this model with Spark's internal hooks. These hooks get triggered once a stage starts execution. It collects the processor's metrics after collecting initial execution details from stage execution and uses the prediction model to get an optimal value of the SMT.

The tuner proposed by Wang et al. [70] uses a Recursive Random Search to explore the configurations and decision tree-based classifier as the performance prediction model. The tuner uses two classifiers which helps it to perform search-space exploration. The first is a binary classifier which predicts whether a configuration value will improve performance. And the second is a binary classifier which predicts how much improvement the given configuration value will yield. For the second classifier, the labels are relative performance improvements (for example, 5% improvement, 10% improvement, 20% improvement, etc.).

The tuner proposed by Trotter et al. [66], uses Genetic Algorithm to explore the search space. Further, instead of using a regression-based performance model where regression-based models usually overfit (as per the workload), they train a classifier that can predict whether a given set of configurations will lead to performance improvement (binary classification). To generate the training dataset, they use three workloads, SOL, WordCount, and RollingCount, where the configurations are sampled randomly. They label a configuration good if the measured

throughput exceeds the default throughput by a certain margin.

Next, we discuss the performance predictions in Reinforcement style tuning frameworks.

Performance predictions in Reinforcement style tuning frameworks

Ben Slimane et al. [19], propose a tuning framework that uses Reinforcement Learning (RL) based Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm. Here they use TD3 algorithm for tuning the tasks instead of DDPG. Further, they propose to use three reward functions i.e.

1. The reward is decided based on the last performance of the agent.
2. The reward is based on the agent's last and initial performance.
3. Where the reward is decided based on past performance, best performance, and the initial performance of the agent

To find the optimal configurations for populating the configuration repository by executing the workloads offline, the tuning framework proposed by Cai et al. [21] uses a two-phase random search algorithm, a variant of a hill-climbing algorithm for the global search. The global search uses Latin hypercube sampling to generate the configurations. Followed by this, it uses a simulated-annealing algorithm for local search. Once they generate performance data for all the selected workloads, they build a similarity matrix (using the collaborative filtering approach) with performance data (execution time) for all the configurations and all the workloads. They use stochastic gradient descent (SGD) with singular value decomposition (SVD) to fill in the missing values. Now they have the similarity matrix. Now for the new job, they map it to the closest previous workloads by measuring the relative standard deviation on the execution time.

Tuning framework by Genkin et al. [38] aims to assign resources to the YARN containers optimally. It measures the response time of the containers by their total execution duration. It receives the resource allocation requests for YARN containers and modifies them. With the modified resource requests, it creates containers and executes the tasks. This mode (the Analyze mode) only observes and watches for any significant performance variation in the container's response time. If it finds

one, it systematically explores the resource allocation parameters until it finds the best setting.

Tuning framework by Perez et al. [60] uses a tuner that identifies and tunes the resource bottlenecks of the systems. Specifically, they monitor and quantify CPU, Memory, Disk, and Network resource bottlenecks and use this information with a fuzzy controller to tune the values of the associated configurations. Further, they use rule-based search-space exploration.

The tuner proposed by Krishna et al. [50] takes advantage of the fact that configurations that lead to better performance are often nearby in the search space. Hence, to use this fact, they propose to use the Evolutionary MCMC algorithm to explore the search space accompanied by metropolis hasting. Here metropolis hasting helps the explorer to make random jumps so that it does not always get stuck in local minima or maxima.

This tuning framework proposed by Guo et al. [41] optimizes the overall platform execution stack in three phases. In the first two phases, it optimizes system-level and application-level configuration parameters; it uses a rule-based system (a heuristic approach). After setting these values and experimenting with the underlying system with these values, they further optimize the system (for both system-level and application-level configuration parameters) using a searching-based algorithm. In this phase, they use a hill-climbing algorithm to optimize the configuration parameters further. After configuring the system from the first two phases, they get a base system performance and a default performance value. The hill climbing algorithm uses the base system performance numbers to explore the search space.

In this section, we categorized and discussed the performance prediction models used by different tuning frameworks in the literature. This sets a context for **R1**. Now based on the performance prediction types discussed in this section, we further discuss the knowledge transfer challenges in the next section.

3.3 Knowledge transfer challenges

This section discusses the knowledge transfer challenges based on the previous section's identified set of performance prediction model types. To start with, we first discuss the core aspects of knowledge transfer and then move on to discuss the

challenges.

The knowledge transfer process involves two main steps:

1. Comparing the execution of the current workload with all workloads executed in the past. Based on the comparison, the framework has to figure out the closest workload in the past.

2. Reusing past tuning experiences. This can be done by either reusing the already trained model or reusing the observational data (generated while tuning a similar workload) and training a new model.

Comparing workloads or workload classification/characterization has been attempted by different tuning frameworks in the past, and we also investigate this in the next chapter 5.

Reusing past tuning experiences is not as straightforward as it looks [34]. Replacing the performance prediction model with the past model (trained while tuning a similar workload) does not guarantee that the replaced model will lead the tuner to generate optimal configurations. The primary reason is that the current workload to tune might be similar to the past workload (with which the past model was trained) but not identical. The second reason is that the models that were trained in the past used influential configurations as features. This means the influential configuration parameters for a previous workload may differ for a new workload.

The tuning frameworks in the literature that use GPR-based, Regression-Based, and Decision Tree based performance models (as shown in Table 3.1) can easily reuse the trained models from the past after figuring out similar workloads. However, in the worst case, the tuning framework might take more than double the iterations and explorations for the replaced model to learn the new workload compared to training a new model from scratch [77]. This problem arises when the tuner picks a wrong model from the past and uses it to tune a new unknown workload. Though this is a complex problem to solve, some tuning frameworks have tried minimizing the impact of the problem in different ways.

To avoid the above-mentioned problem, the tuning framework presented by Fekry et al. [34] uses a multi-task Gaussian Process where each workload is modeled as a separate task. For a new unknown workload, the framework models it as

a new task. Once the framework identifies a similar workload (source task) seen in the past for the target workload (target task), it starts exploring the source task. It then trains the target task (defined over the significant parameters of the found source task) using the new observations. The proposed acquisition function in the tuning framework further maximizes the Expected Improvement of the source or target task.

The same problem has also been addressed by the framework proposed by Feurer et al. [36], where the tuning framework uses an ensemble of GP models, which includes a source GP model (the model trained using observational data of a similar workload) and a target GP model (the new model created for the target workload). In this ensemble, the weights are assigned based on relative ranking loss. This generalizes the ensemble across the models trained in the past and models trained when the workloads are executed on different infrastructures.

The tuning frameworks which use ensemble-based performance prediction models (as shown in Table 3.1) do not model end-to-end workload execution but instead model individual stage-level execution. Any new target workload might or might now have the same stage-level distribution. Hence, in the case of ensemble-based performance prediction models, the tuning frameworks need to map individual execution phases (or stages) to the stages seen in the past. Once the mapping is complete, the tuning framework reuses the trained models from the past for every stage and could form a new ensemble. This process becomes more challenging in the case of Big Data Frameworks like Spark or Storm, where the workload execution is divided into hundreds of stages. This is typically seen in ML-based workloads in HiBench. With this method, the tuner's insufficient experience (when the tuner cannot find a similar execution profile from the past) will impact the convergence and exploration process of new ensemble-based models.

Any RL-based [19] or MCMC-based tuner [50] could easily train the algorithm by replaying the workload offline and then further reusing when a similar workload has to be tuned. Further, once the offline trained model is used for tuning actual workloads, the tuning framework can continuously update the model based on the live observations by executing the workload [77]. The same approach could be used for any search-based tuning framework which does not use a performance prediction model.

The classifier-based performance prediction models are divided into two further sub-groups.

1. Using Classifiers to predict the optimal configuration - Tuning Frameworks [28], [48] use a classification-based approach where the labels are configuration parameter values.

2. Using Classifiers to predict the performance - Tuning Frameworks [70], [66] use a classification-based approach to predict if a given candidate configuration would lead to better performance compared to a base performance.

The classifiers that directly predict the optimal configuration value are trained to learn the system's behavior. This means they can predict the optimal configuration for any given workload. Hence, there is no need for knowledge transfer required. The main problem with this approach is that it can predict the value for only one configuration parameter. Also, it is expected that the configuration parameter that has to be tuned should be discreet or categorical. Predicting the optimal configuration value when the configuration range is continuous would be challenging. Another challenge for this tuning framework is that the selection of features (application-level metrics) needs to be done by an expert. The chosen features should sufficiently quantify the system behavior for any workload.

The classifiers used to predict the performance can be replaced with classifiers trained in the past using a similar approach described above. The challenges would also remain the same.

3.4 Conclusion

ML-based tuning frameworks offer several advantages, such as treating the system as a black box, leveraging past tuning experiences, and using performance prediction models to explore the search space. The challenges arise when attempting to leverage past tuning experiences due to the architecture and framework of the performance prediction model. In this chapter, we summarized the performance models used in the literature and presented the knowledge transfer challenges. This chapter answers the research question (**R1**) presented in Chapter 2.

Some tuning frameworks do not rely on knowledge transfer and use different mechanisms to amortize the tuning costs. The following section discusses configuration parameter pruning (one such mechanism used to amortize the tuning cost), its pros & cons pruning, and the trade-offs. The next chapter aims to answer the second research question (**R2**) presented in Chapter 2.

Chapter 4

Configuration Parameter Pruning and Challenges

With the increasing popularity of Big Data platforms, such as Apache Spark, managing many configurations for optimal performance can pose a significant challenge. To address this issue, configuration pruning, also known as dimensionality reduction, is used to eliminate configuration options that do not significantly impact performance. This approach simplifies the system's tuning and enhances its overall efficiency.

However, it is essential to consider the different search-space techniques and their respective trade-offs when implementing configuration pruning. Multiple pruning techniques are available, each with its own advantages and disadvantages. To make informed decisions, users must understand these trade-offs.

This Chapter examines the various search-space pruning techniques, their benefits and drawbacks, and the trade-offs that users must consider when implementing them in Big Data platforms, such as Spark. By providing insights into the strengths and limitations of each technique, our study can aid users in selecting the most suitable approach for their specific use case. This Chapter will answer the second research question **(R2)**.

4.1 Introduction

Out of all the configurations of a Big Data processing framework, only a handful significantly impact the execution [35]. Hence, in the literature, multiple tuning frameworks prune the non-influential configurations to reduce the search space [34, 35, 42, 51, 57, 60, 63, 79]. Different tuning frameworks use various pruning (dimensionality reduction) techniques.

Configuration pruning techniques require observational data which can increase the cost of tuning, but once an influential set of configurations is found, it can reduce the search space and training time, creating a trade-off between tuning efficiency and cost amortization. The configuration pruning techniques are statistical and need some amount of observational data. This brings a trade-off between tuning efficiency and tuning cost amortization. Tuning efficiency reduces if the configuration pruning techniques need more observational data (as the tuning framework has to run multiple initial experiments to generate the observational data) to find influential configuration parameters. At the same time, once the tuning framework finds out the influential set of configurations, it can reduce the search space for exploration, reducing the training time. However, if the pruning techniques need a lot of observational data, amortizing the initial cost of generating the observational data will become problematic. Many tuning frameworks opt for manually selecting a bunch of configurations for tuning.

Our study finds that many tuning frameworks opt for manually handpicking a set of configurations. Manually selecting configurations always has a risk of missing influential configurations. However, by manually picking the configurations, the tuning framework gains an advantage in search space as its size is relatively smaller than the search space with all the configurations. Further, we summarize and discuss the different pruning techniques used in the literature.

In the next Section 4.2, we summarize and discuss the pruning techniques used by different tuning frameworks.

4.2 Summarizing the pruning techniques

We summarize all the pruning techniques with the respective challenges in Table 4.1.

Table 4.1: Summarizing the configuration pruning techniques

Approach	Pruning Techniques	Challenges
[42]	Genetic Algorithm Mutation using Gini's Importance score	Needs sufficient size of observational data
[79]	Statistical Significance test using Gini's Importance score with p-value	Needs sufficient size of observational data
[34, 35]	Approximating Gini's Importance score using Recursive Feature Elimination	Needs sufficient size of observational data
[60]	Prunes the non-influential configurations using Resource Bottleneck scores	Needs solid understanding of configurations per resource bottlenecks
[51]	Significance analysis using t-values and p-values	Needs sufficient size of observational data
[63]	Finds out configuration dependencies using execution cost models	Needs solid understanding of systems internals
[57]	Finds out dependencies manually for pruning non-influential configs	Needs solid understanding of systems internals
[56]	Manually prunes the configuration based on workload classification	Needs solid understanding of systems internals
[30]	Manually prunes the configurations as per the stage type	Needs solid understanding of systems internals
[39]	Manually explores the entire search space to prune configs	Hard to scaled when configs are in hundreds

Different works in the literature use Gini's importance score to rank the configurations and then prune the non-influential ones. The traction toward Gini's importance score was mainly because it can rank the configurations as per their importance using a minimal size of observational data. Gini's score also works well because the majority of configurations of Big Data frameworks are categorical. This fact suits the Decision Tree based regressors the most [77] and is used for computing the importance scores. This configuration ranking method must also trade off between tuning efficiency and tuning cost amortization. This is because

computing the Gini's score involves training a decision tree-based meta-model, which requires observational data. Ginis score can be used to rank the configurations. Different tuning frameworks have used it differently.

Guo et al. [42] use Genetic Algorithm (GA) to explore the search space. The tuning framework does not explicitly prune out the non-influential configurations in this work. Instead, the GA implicitly uses the Gini's importance scores. While selecting the next set of candidate configurations to explore, the GA uses the importance score in the mutation phase. This also speeds up the convergence. Using both p-value and Gini's importance score-based measures in the tuning framework [79] can help identify and prune non-influential configurations, improving the efficiency of the tuning process.

Zong et al. [79] use both p-value and Gini's importance score-based measures to find the influential configurations. To detect the configurations that do not impact the execution, they do null-hypothesis testing by finding out the p-values of the configurations. Further, the configuration whose p-values are less than a specific threshold is pruned in this phase. Further, the framework finds the Gini's importance score among the selected configurations and prunes the configurations with a score less than a defined threshold value. Linear Regressors are tools that help in computing the p-value. To identify the most effective features for predicting job performance, Lama and Zhou [51] employ a stepwise linear regression technique. It uses a systematic approach to prune the configurations from the performance prediction model based on the statistical significance. This approach recursively computes the parameter's importance scores using the p-value of a t-statistic. In this approach, they define certain thresholds to reject the null hypothesis (i.e., configurations with p-value less than a threshold are pruned). The proposed statistical approach also needs observational data and has to trade off between tuning efficiency and cost amortization. Gini's importance score is computed using Decision Tree based regressors. Tuneful's [35] approach of approximating the Gini's importance score with Recursive Feature Elimination (RFE) provides an efficient method to manage the trade-off between tuning efficiency and tuning cost amortization, allowing for a reduction in training time and computational resources.

To manage the trade-off between tuning efficiency and tuning cost amortization, Tuneful [34, 35] proposed a method to approximate the Gini's importance

score or feature importance with Recursive Feature Elimination (RFE). Tuneful's approach takes advantage of the fact that non-influential knobs will have little effect on the outcome, for which even a rough approximation of the meta-model's prediction would be enough. These techniques compute the feature importance at a global level (i.e., for all the data points from the training data and not individually for every data point or observation) based on the decrease in meta-models performance (like - a decrease in RMSE, model impurity, etc.). Search space can be explored in an easy way when similar configurations are grouped. Gini's importance scores do not give any information about the dependencies of configurations, or configurations that represent a specific subsystem of the Big Data framework.

The PETS [60] groups organize 18 parameters into different ensembles where each ensemble affects the workload execution in a specific way. This allows multiple parameters to be adjusted in certain directions. The framework forms eight groups or ensembles, each indicating a possible direction for adjustment. Perez et al. explain the adjustment process as - "As such we elaborate on a total of eight groups/ensembles, indicating two possible directions (increasing or decreasing) for four different resources (CPU, memory, network and disk)". Further, they use a fuzzy controller and resource bottleneck scores to narrow the search-space exploration to specific parameters per the identified bottleneck. They compute the resource bottleneck scores by averaging the scores across the nodes in a cluster based on the fraction of total tasks executed on the specific node. The problem with this approach is that it needs an excellent understanding of the configurations per the resource bottlenecks. Hence, this approach cannot be entirely categorized as a block-box approach. The statistics gained from the execution model of a Big Data framework like Hadoop can also be used to infer optimal values of specific configurations.

In MRTuner [63], authors propose to use the execution model to find the optimal values for four key parameters. Using this, they further derive the dependencies between the configurations. With the dependencies, the framework reduces the search space. This approach can be used specifically for frameworks like Hadoop, but extending it to other frameworks like Spark or Flink isn't easy. This is because their proposed execution model is built based on the Map and Reduce execution stages. At the same time, other frameworks like Spark can perform workload

execution using any number of stages and not just map and reduce. One of the problems with this pruning technique is that it needs a solid understanding of the system internals and is more inclined towards a gray-box approach. Some works in the literature showcase the benefits that a tuning framework can achieve if it knows the dependencies between the configurations.

Liu et al. [57] propose a pruning strategy that first finds out the set of parameters which can be independently searched and further divides the exploration process accordingly. And then, the framework further recursively performs a similar search (in parallel) across the newly created sub-search spaces to find the influential configurations and prunes the non-influential ones. This framework also needs a good understanding of the system internals to determine the dependencies. Apart from pruning strategies, workload characterization can also help in reducing dimensionality.

The tuning framework proposed in Gunther [56] classifies any given application into four classes. For classifying applications, authors selected specific metrics that can help quantify the resource bottlenecks in Hadoop. Further, they use threshold values for the metrics to group the applications. The authors manually prune specific configuration parameters based on the classified group to which the application belongs. For example, if the application is classified into the group which signifies a low shuffle intensity, then most of the parameters that impact the Shuffling behavior can be pruned out. The pruning of non-influential configurations is a manual process based on understanding and human experiences. As Big Data frameworks execute the workloads in stages, it is observed in [30] that configurations that impact specific stages can be grouped.

Jellyfish [30] reduces the search space by grouping the configuration parameters into two groups, i.e., map-phase relevant configurations and reduce-phase relevant configurations. They group the configurations based on the understanding of the documentation. This framework also follows a manual pruning process based on understanding and human experiences. With a limited set of configurations, Gounaris and Torres [39] filters out the non-influential configurations by manually exploring the configurations.

The tuning framework proposed by [39] [39] gains a significant advantage from its proposed knob pruning method. Based on its experience, it initially picks up

Spark's top 15 configuration parameters. Then it manually finds the highly impacting parameters by executing all possible configuration pairs. They select the top nine configuration parameters using a systematic approach and the generated observational dataset (from the experiments). They find the best configurations from three workloads and use the best set for new workloads. This framework assumes that dependencies (among the configurations) only exist in pairs. However, static analysis tools like [24] and documentation mention that dependencies can exist across various configurations, not just pairs.

In the next section, we conclude this Chapter.

4.3 Conclusion

While configuration pruning techniques can effectively reduce the search space and training time of Big Data processing framework tuning, they require observational data and bring a trade-off between tuning efficiency and cost amortization. Many tuning frameworks opt for manually selecting a set of configurations, which has a risk of missing influential configurations but gains an advantage in search space. The literature presents various pruning techniques, which we summarize and discuss in this study.

The set of influential configurations can be the same for similar workloads. However, figuring out similar workloads can be challenging. Hence, in the next Chapter, we explore the workload characterization techniques, which specifically aim to answer the third research question **(R3)** (from Chapter 2).

Chapter 5

Workload Characterization Methods

With unprecedented data volume growth, big data workloads are becoming increasingly diverse and complex [77]. Understanding and characterizing these workloads are crucial for efficient resource allocation, capacity planning, and performance optimization [43]. Workload classification categorizes similar workloads based on their characteristics, which helps identify suitable previously learned ML models. To classify big data workloads, researchers have proposed various techniques and features. This Chapter investigates the present state-of-the-art workload classification methods, focusing on the techniques and features used to characterize and classify big data workloads. In particular, it aims to answer the third research question (**R3**) from Chapter 2.

5.1 Introduction

Big Data frameworks execute applications where different metrics are available at different layers of the execution stack that show variations with statistical significance. These variations can be used to define similar workloads. Big Data frameworks execute applications that process massive volumes of data. The execution occurs in stages where the stage executions are part of the physical execution plan. Metrics available at different layers (i.e., Kernel, JVM, Application, etc.) show

variations when the applications execute. These variations are statistically significant when different applications execute and when individual stages of different applications execute. In literature, workloads are defined in terms of the statistical significance observed across different layers of metrics. Two different jobs are considered similar workloads if the observed metrics variances are statistically the same [37]. Selection of features (that define a Big Data workload) should be performed in such a way that the variations in the features should be as minimum as possible when the same instance of workload is executed multiple times and should be maximum when a different workload gets executed. [32].

The selection of physical and logical features in defining a Big Data workload defines its scope. Two workloads that look similar when using one set of metrics might look different when using another. Multiple works in the literature use physical features like metrics variations to define a workload. However, some works also use logical features like execution plans, features extracted from the execution profile, etc., to define a workload [75]. And some works use logical and physical features to define workloads [32]. In any definition of a Big Data workload, the selected features define its scope. For example, a workload defined over the metrics exposed by the Big Data processing framework has a lesser scope than a workload defined over the metrics exposed by both the OS kernel layer and Big Data processing framework. This also means two workloads that look similar when using the Big Data framework metrics might look very different when using the kernel metrics [72]. ML-based tuners are better than others as they can save time and costs by transferring knowledge from past workloads. The process of characterizing workloads is crucial for optimal knowledge transfer.

The workload characterization process is crucial for ML-based tuners to optimally map the current workload to a similar workload in the past to transfer knowledge and save time and tuning costs. ML-based tuners are better than other tuners because they can transfer knowledge and save a lot of time and tuning costs. We mentioned the challenges of knowledge transfer in Chapter 3.3. A tuning framework has to find a workload from the past that very closely resembles the characteristics of the current executing workload. This mapping of the current workload to a workload from the past has to be done optimally so that the tuning framework can use knowledge or learning from the past. In literature, mapping the current

workload to a similar workload in the past is referred to as the workload characterization/classification process [34, 68]. The workload characterization process groups the workloads that exhibit similar variation patterns in the features (logical or physical). The workload characterization process is critical for an ML-based tuning framework to accurately detect a workload change that may arise due to a change in data size, executing application, application code, logic, or the underlying software or hardware stack.

For an ML-based tuning framework, the workload characterization process is expected to detect a workload change and map a current workload to the optimal one. To detect workload change, tuning frameworks often compare the features obtained from every single execution of the workload. The core challenge for the tuning framework is accurately detecting a workload change. A workload change can happen for the following reasons [37]:

1. When the data size that the workload is executing significantly changes.
2. When the executing application changes (for example, an OLAP workload changes to an OLTP workload).
3. When the application code or logic changes. This might cause a change in the execution plan.
4. When the underlying software or hardware stack changes.

We summarize the different approaches and features used in Table 5.1.

Many works in the literature have preferred physical features over logical ones. This is because it is easy to detect workload change with physical features compared to the logical features [37, 72]. Hence, we first describe the different features that the tuning frameworks have used in the literature (Section 5.2) to define workload and then discuss the workload characterization techniques (Section 5.3).

5.2 Features used for defining workloads

This section presents the different features used in the literature to quantify the workload execution behavior on Big Data frameworks.

Table 5.1: Summarizing the workload characterization features and techniques

Approach	Input Metrics	Characterization Techniques
[37]	yarn-Container level	Sliding Window
[72]	job and micro-architectural level	Benchmark Similarity Matrix
[48]	micro-architectural level	XGBoost Classifier
[22]	resource utilization & Hadoop counters	MLP Classifier
[56]	Hadoop counters	Thresholding based
[51]	resource utilization	LCS & K-Medoid
[50]	System Calls	Jaccard Index
[34]	Spark stage level	AE & Euclidean distance
[28]	workload properties & cluster level	XGBoost Classifier
[75]	Spark stage DAG	Graph Edit Distance

Genkin and Dehne [37] measures the system performance metrics and Yarn container metrics. They measure the system performance metrics using *nmon* and the Yarn container metrics using KERMIT logs [38]. Some Yarn container metrics include the number of containers created, average active memory, average container response time, etc. They convert the observed metrics to a time-series dataset and characterize the workload. Genkin and Dehne define workload as if it can be "used to represent any continuous sequence of observation windows with feature vectors that do not show any statistically meaningful differences." [37]. Using job-level and micro-architectural metrics allows for a more granular understanding of system behavior, which is essential for accurately characterizing workloads and improving tuning performance.

Yu et al. [72] use a set of job-level and micro-architectural-level metrics. These metrics quantify the system behavior at the granularity of the application level. The job-level metrics are chosen to quantify the amount of input and output data, communications at the job level (shuffles), and processing & execution. Similarly, the micro-architectural-level metrics are collected to quantify the performance at the processor level for every worker node in the cluster. These metrics are collected at every node and then aggregated globally. Similarly, Jia et al. [48] only measure the simultaneous multi-threading (SMT) features (micro-architectural-level) to define

a workload. They precisely measure 11 performance events using the *Perf* tool to observe the SMT behavior accurately. The system metrics and job counters are collected from different computing resources to quantify the workload execution behavior.

Chen et al. [22], the tuning framework collects system metrics and Hadoop counters for workload characterizations. The system metrics are collected from computing resources like CPU, Memory, Disk, and Network. Many tuning frameworks assume that the best set of metrics that quantify the workload execution comes from the application layer (the Big Data processing framework layer).

The Hadoop counters measure the job execution details (precisely the runtime behavior) at the Yarn layer. Similarly, Liao et al. [56] use five specific job counters to measure the shuffle intensity, and the amount of data read and written to HDFS. The tuning framework uses these five metrics to quantify a workload in this work. Some of the tuning frameworks in the literature rely on the assumption that resource utilization patterns serve as a good way of quantifying workload execution on Big Data frameworks.

Lama and Zhou [51] use metrics to measure the resource utilization patterns of the workload execution. They assume that the applications that exhibit similar resource utilization patterns can be tuned using the same performance prediction model. They precisely measure the resource utilization (i.e., CPU, disk, and network usage on all the worker machines using *dstat*). Based on the same assumption, Krishna et al. [50] profile the runtime behavior using system call traces. Further, they extract features from the system call traces measured across all worker nodes. The actual execution of workload in Big Data frameworks follows a stage execution model where the workload is executed in stages. Some of the tuning frameworks in the literature use stage-level metrics to quantify workload execution.

Unlike Hadoop's MapReduce, Spark's execution can be divided into several hundred stages. Fekry et al. [34] measure the applications execution behavior by aggregating the stage-level metrics. These metrics are exposed by Spark and are available for every stage. The stage-level metrics are intended to precisely capture the shuffle overhead, CPU overhead, serialization overhead, memory overhead, and garbage collection overhead. Big Data frameworks execute the workloads using

clusters of physical machines. The cluster-specific metrics also help in quantifying the workload execution.

Daud et al. measure the applications properties and cluster metrics using "*masterMemory*, *masterCore*, *workerNode*, *workerMemoryNode*, *workerCoreNode*, *dataSize*, *applicationComplexity* and *memoryCapacity*" [28]. However, these metrics are used only to study the behavior of executing Twitter's link prediction workloads on Spark. Stage DAGs (execution plans) are easy to model the Workloads. Using the stage DAGs, the workload comparison also becomes easier.

Zacheilas et al. [75] exploit that two workloads must be similar if their respective stage graphs are similar. Hence, they directly compare the stage graphs generated by the DAG scheduler in Spark to compare the two workloads.

Next (Section 5.3), we summarize and discuss the workload characterization techniques used in the literature by different Big Data tuning frameworks.

5.3 Techniques used for workload characterization

This section presents the different workload classification techniques used in the literature to categorize Big Data workloads.

In [37], authors define workload change in terms of:

1. Workload cycles: shifts caused by a change in usage pattern (i.e., read heavy to write heavy).

2. Workload drifts: shifts characterized by long-term change. Changes in user volume can cause changes in data volume or changes in underlying software or hardware infrastructure.

3. Workload anomaly: shifts characterized by sudden, abrupt, short term and random variation in usage pattern.

4. Workload transition: deeper in granularity, at the level of stages in the big data framework.

The architecture proposed by Genkin and Dehne [37] introduces a multivariate observation window that moves over time series data. The observation window is

combined with analytical sliding windows with feature vectors as mean and standard deviations of individual metrics. They statistically characterize a change in the observational workload pattern based on the changes observed in the feature vectors of the analytical sliding window. The overall workload of a big data framework is statistically defined as an observation window with a steady-state period (represented by an analytical sliding window) connected to other non-steady-state periods. Here non-steady state periods are the same as the workload transition defined above. For example, in a map-reduce workload execution, the steady state periods would be the sliding window capturing the execution of the map and reduce phase. In contrast, the transition phase from the map would characterize the non-steady state period to reduce. On a high level, they define a workload change detector which classifies analytical sliding windows into a steady state or a non steady state. This component to mark any significant difference between the given sliding windows performs a set of statistical significance tests. They use a random forest ensemble as a workload classifier to classify steady-state observational windows to workload sub-types. Further, using a transition classifier with the same random forest ensemble method to figure out long-term or short-term workload transitions from non-steady state analytical windows by computing the rate of change of feature vectors. Finally, they use a workload predictor using the LSTM approach to find the temporal dependency in the time-series sliding window data. These techniques are specifically designed for system performance metrics and Yarn container metrics and might not work with job-level and micro-architectural-level metrics.

For the job-level and micro-architectural-level metrics [72] authors, try to quantify the system behavior at the granularity of the application level. They introduce two new metrics for comparing the benchmarks or workloads: Metrics Importance Analysis based on Benchmark Similarity Matrix (BSM) and Kiviat plots. For both metrics, the first step is to compute the importance of metrics. For this, the authors use Gini's based importance score. Then they compute the important metrics for all workloads and rank them. These important metrics are then used to plot the Kiviat plot using which the authors compare the workloads. Also, they use the same important metrics and compute the Manhattan distance to find the BSM. Similarly, Jia et al. [48] use micro-architectural-level metrics to define workload. And further, they train an XGBoost-based classifier to directly predict the SMT

configurations' optimal value. They train the classifier with training data generated from the execution of many different workloads. When a new/unknown workload starts execution, the trained classifier implicitly maps the new metrics data to the closest workload and predicts the optimal configuration. A similar approach has been seen in [28] where the tuning framework uses a classification model (trained using many different workload execution data) to implicitly map the new metrics to the closest workload. Some works in the literature have used a simple classifier to classify the workload with labels.

Chen et al. [22] train a Multi-Layer Perceptron (MLP) based Neural Network classifier offline using the workload execution data. They generate the training data created for every workload by labeling each data point with its respective workloadId. For generating the training data, they execute the workload on a specific testbed setup using a minimal dataset. Some works [56] in the literature create a set of rules to classify/characterize workloads. Instead of a classification engine, tuning frameworks have used a manual set of rules [56]. Liao et al. [56] classify the workloads using manually created rules. Here the rules are the thresholds defined for the specific metrics. The classification frameworks do not specifically consider the noise generated by resource contention (in the observational data) while workload execution.

To quantify the resource usage at the job level [51], authors aggregate the time-series data by taking the average. Further, they use the Longest Common Subsequence (LCS) based distance metric and group similar jobs together using the k-medoid clustering approach. Another motivation to use the LCS distance metric is that it works better even with noise created by resource contention (when multiple jobs execute). Besides using the application and kernel-level metrics to measure resource utilization, system call traces can quantify resource usage patterns.

To assess how similar resource usage patterns are, Krishna et al. [50] employed a method that involves converting system call trace records to a four-tuple dataset. The first tuple in the dataset represents the system call sequence, while the second tuple counts the categorical arguments across all system calls. The third tuple quantifies the term frequency and counts the categorical arguments for every system call, and the fourth tuple computes the numerical arguments mean in the system calls. To measure the similarity between two workloads, the Jaccard Index

is used as a metric, which has been previously utilized for comparing sets. The high dimensionality of the metrics presents challenges for workload classification or characterization.

To model the workload execution data, authors in Fekry et al. [34] convert the aggregated high dimensional stage-level execution data to lower dimensions using Auto Encoders. Further, to quantify the similarity between any two workloads, Fekry et al. [34] measure the Manhattan distance between the lower dimensional vectors. The above-mentioned techniques cannot find workload similarity when modeled using the graph data structure.

To find similar workloads for a given new workload, Zacheilas et al. [75] measure the dissimilarity score using the Graph Edit Distance (GED) on the physical execution plan (stage graphs). GED has been used in literature to find the similarity between two graphs. Authors initially assume the two graphs are the same and then find the dissimilarity by measuring the cost to delete a stage from the stage graph modeled as Delete Stage Cost (DSC).

5.4 Conclusion

The effectiveness of configuration tuning of Big Data frameworks relies on the precise characterization and classification of workloads, achieved through analyzing various features across different layers of the execution stack. Workloads can be defined based on physical features, such as metric variations, logical features, like execution plans and profiles, or a combination of both. The choice of features needs to be made cautiously, considering the workload's scope and variances among multiple executions of the same or different workloads. ML-based tuning frameworks are preferable because they transfer knowledge, minimize time, and save costs. Nevertheless, accurately detecting a workload change is challenging for these frameworks, with physical features being preferred over logical ones. This Chapter presents an overview of the features used to define workloads and the workload characterization techniques employed in the existing literature.

Based on the diversity of workloads and different tuning frameworks in literature, designing a tuner benchmarking or evaluation strategy is challenging. Hence, in the next Chapter 6, we survey the benchmarks and metrics used in the literature

to evaluate the tuner's performance.

Chapter 6

Benchmarking Tuning Frameworks

Tuning Big Data processing frameworks is a challenging task that has led to the developing of various automated tuning frameworks in the literature. Effective evaluation of these frameworks' performance necessitates identifying and using suitable benchmarks and metrics. This Chapter explores the literature and highlights the different benchmarks and metrics used to assess the tuning of Big Data processing frameworks. Specifically, this Chapter answers the fourth research question (**R4** from Chapter 2).

6.1 Introduction

As discussed in Chapter 3, the main components of ML-based tuners are a search-space explorer engine, performance prediction model, workload mapping engine, and configuration pruning engine. This section discusses how different approaches in the literature have evaluated proposed tuning frameworks. Specifically, in this Chapter, we survey the metrics and benchmarks used to evaluate the tuning frameworks and some of their core components.

Next, we present how the performance prediction models are evaluated in Section 6.2, the configuration pruning in Section 6.3, the search space exploration in Section 6.4 and the end-to-end tuning in Section 6.5.

6.2 Evaluating the performance prediction models

Performance Prediction Models are ML models trained to learn the impact of a configuration change on the system. Here the features are the configurations or specific system metrics and workload characteristics, and the outcome is a performance metric. The performance metric varies based on the underlying Big Data Framework to be tuned. For example, for batch processing systems, the outcome is mostly end-to-end execution time or system resource utilization, whereas for streaming systems, the outcome could be throughput or latency. This section focuses on the metrics used to evaluate the performance prediction models.

Root Mean Square Error (RMSE) has been used by Zacheilas et al. [75] and Lama and Zhou [51] to quantify the prediction error. The frameworks use standard cross-validation mechanisms to evaluate the RMSE of the regression model. Similarly, Mean Squared Error (MSE) has been used by Gu et al. [40], and Bei et al. [18] to evaluate the regression models. In the case of RMSE, it is measured using the same unit as the outcome variable. However, MSE is used to penalize the more significant errors more severely. These metrics measure the actual error instead of using a percentage scale.

Wang et al. [69] use Relative Absolute Error (RAE) to evaluate the regression model. The RAE compares the mean error with the actual error. Mean Absolute Percentage Error (MAPE) has also been used to evaluate the regression models [23, 25, 69, 71]. This metric is more often used as the scale used is a percentage. Like MAPE, some works [18, 22, 26, 42, 73, 79] use error percentages to evaluate the regression models. These metrics can evaluate regression-based performance prediction models and will not work when the performance prediction model is a classification model.

We also observed that some of the tuning frameworks in the literature use classification models as performance prediction models (Table 3.1). Instead of predicting the actual performance numbers, the classification models can predict whether the given features (configurations) would lead to performance improvement. This could be a binary or multi-class classification where the class labels are expected to performance improvement (in percentage scale). Also, some tuning frameworks use classifiers to directly predict the optimal values of specific configuration param-

eters. Accuracy, precision, F1-score and Recall has been used [22, 28, 48, 66, 70] for evaluating classification based models. Generating observational data for training performance prediction models is costly as the tuning frameworks need to perform experiments on the target system. Hence, Guo et al. [42] propose to create and evaluate synthetic observational data.

Guo et al. [42] use Generative Adversarial Networks (GANs) to generate synthetic data that the performance prediction models can consume. To evaluate the effectiveness of the generated synthetic data, authors visualize and compare the distribution of generated synthetic data with ground truth data. For this, authors use 3d-Kiviat plots.

The next section presents the metrics used to evaluate configuration pruning (dimensionality reduction).

6.3 Evaluating configuration pruning engines

Tuning Frameworks sometimes use configuration pruning engines to prune the non-influential configuration parameters [35] to reduce the search space. Once the search space is trimmed, it is easier for the configuration explorer to search for the optimal configurations. Evaluating the configuration pruning is difficult for unknown (production) workloads because of the absence of the ground truth about the actual influential configuration parameters for the unknown workload. Evaluating configuration pruning is easier if the tuning framework knows the ground truth.

Fekry et al. [35] first use a huge set of observational data (generated offline by randomly setting the configurations) to find the ranking of the configurations and term it as ground truth. Using the ground truth, they evaluate the performance of the tuning engine when it is used with minimal observational data. They compute the accuracy to evaluate the pruning engine. The accuracy is computed using the ranking produced by the pruning engine. Using the ranking, the tuner finds the significant configuration parameters and non-significant parameters. The authors compute the accuracy based on the computed significant and non-significant parameters and the ground truth data. It is difficult to evaluate the ranking of the influential configuration parameters using accuracy.

Bao et al. [16] define Discounted Cumulative Gain (DCG) to evaluate the quality of ranking for the configuration pruning engine. Given a set of significant configurations (n) and their respective ranking (r), ranked in the order of the importance score (calculated using Gini's importance), the DCG is described as follows:

$$DCG = \sum_{i=1}^{n} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{6.1}$$

Here rel_i is the graded relevance of the configuration i. Authors in [16] used this metric to compare the DCG of the ranked configurations generated by the tuner with the ground truth.

Search Space pruning inherently helps search space exploration by filtering out non-effective regions. Evaluating the search space pruning is different from assessing search space exploration. In the next section, we present the metrics used to evaluate the search space exploration process.

6.4 Evaluating search space exploration

The search space exploration algorithm is responsible for exploring the configurations to find the optimal one within the fewest possible iterations. The search space exploration algorithms use performance prediction models to evaluate candidate configurations. In this section, we present the different metrics used in the literature to assess the search space exploration process.

Fekry et al. [35] use the searching time as the metric to evaluate the search space exploration. Using this metric, the authors measure the total search time until the explorer finds the best configuration. In the literature, convergence has been widely used to evaluate the effectiveness of search space exploration. Previous works [34, 35, 42, 79] use convergence (best execution time vs. total iterations) of the tuner to evaluate the search space explorer. Similarly, to measure the convergence, Perez et al. [60] normalizes the iterations based on the iterations it takes for a baseline tuner to converge. Convergence evaluates the search space exploration and the performance prediction model. This is because finding the best execution time with the least iterations is a function of both search space exploration and learning of the performance prediction model.

Apart from evaluating different sub-components of the tuning frameworks, it is also essential to assess the end-to-end tuning. Hence, in the next section, we survey the overall tuning efficiency (or end-to-end tuning efficiency).

6.5 Evaluating the end-to-end tuning performance

Any tuning framework aims to find the optimal configuration in the fewest iterations. This results in better scores on the outcome metrics. However, the tuner performs many experiments (by setting the configurations) to learn the workload execution behavior of the underlying system. This is the tuning cost [35] which the tuner incurs in the initial runs and is expected to amortize once the learning is complete. Some of the tuners which tune the infrastructure configurations, like the number of VMs, the size of VMs, etc., can easily suggest higher configurations where the workload execution becomes faster, but the infrastructure cost increases. Due to this, various metrics have been discussed in the literature, which we summarize in this section. We also summarize the benchmarks and workloads these tuners have used to understand a tuner's applicability and scope.

The end-to-end performance metrics vary for different Big Data frameworks. For example, stream processing frameworks' outcome metrics are often throughput or latency, whereas batch processing frameworks' outcome metrics are total execution time. Due to the difference in outcome metrics, benchmarks vary for different frameworks. In the next section, we present the metrics used by tuning frameworks that aim to tune the Big Data streaming frameworks.

Metrics used for evaluating the streaming frameworks

Streaming frameworks like Apache Storm, Heron, and Flink follow a per-record execution model [77] and care about higher throughput and lower latency. There are also streaming systems like Spark streaming, which process the workload in micro batches (or batched streaming model). Typically the throughput is higher in the batched streaming model than in per-record processing frameworks.

Previous works [42, 69, 74] have used 99^{th} percentile latency to measure the system performance. In a stream processing system, when the 99^{th} percentile latency is recorded as x seconds, it means that 99% of the records or micro-batches

Table 6.1: Summarizing the streaming workloads and benchmarks

Workload	Benchmark	Used in
Bus Traces	collection of traces	[74]
ClickStream	extended [52]	[64]
FixWindow	HiBench [5]	[42]
Streaming WordCount	HiBench [5]	[42, 53, 65, 66, 69]
Repartition	HiBench [5]	[42]
Identity	HiBench [5]	[42]
Traffic Monitoring	Manually Created	[69]
Yahoo Ads	Streaming Benchmark [12]	[42]
Log Stream	Storm Benchmark [10]	[53]
SQL Queries	Storm Benchmark [10]	[53, 65, 66]
Rolling Count	Storm Benchmark [10]	[65, 66]
TPC-H	Spark Cyclone [3]	[69]

were processed in under x seconds. Similarly, performance gains are measured using throughput. Some of the previous works [42, 53, 65, 66, 69] use throughput for measuring the performance gain. Guo et al. [42], propose a framework that measures the throughput in events (graph operations) per second. Similarly, some works use [65, 66, 69] tuples per second to measure the throughput. Likewise, Trotter et al. [66] also measures the normalized (compared to the throughput obtained by the baseline tuner) throughput to understand the system's performance. Tuner efficiency can also be measured by the total search time for optimal configurations. Trotter et al. also measures the total runtime for the tuner to find the optimal configurations. The total time also inherently signifies the tuner's convergence (take taken for the tuner to converge). Apart from searching time, the optimal configurations obtained from a tuning framework can also lead to side effects, such as an increase in error rate, etc.

Stream processing systems might miss records or tuples when the incoming rate peaks. Hence, Zacheilas et al. [74], measure the tuples missed over the time tuner finds the best configurations. The tuning framework proposed by Zacheilas et al. [74], adjusts the total number of engines which process the records. Hence it is essential to measure the total cost and resource utilization [69].

We summarize the workloads and benchmarks used for evaluating tuning frame-

works that tune the stream processing systems in Table 6.1. As per the findings, we observed that HiBench had been used by most of the tuning frameworks, followed by Storm Benchmark. Regarding workload, it is clear that Streaming Wordcount is the highest used workload from HiBench by different tuning frameworks. And followed by Streaming Wordcount, other highly preferred workloads are SQL Queries and Rolling Count.

In the next section, we present the different metrics used in the literature for evaluating tuning frameworks that tune batch-processing Big Data frameworks.

Metrics used for evaluating the batch processing frameworks

The batch processing systems like Apache Spark and Hadoop process the data and execute the workloads in stages. Hadoop executes the data in a map and reduce stages, whereas the Spark processing framework executes the workload in multiple stages. If there is no dependency between the stages, Spark executes them in parallel or serially. Typically in batch processing systems, workload execution time may vary from minutes to several hours [29]. Hence, the most important end-to-end metric is total execution time. However, SQL workloads that execute queries can also be interested in observing the system's overall throughput.

Cumulative execution time [21, 34, 35, 50] helps visualize tuning cost amortization. Any ML-based tuner initially incurs the cost of training performance prediction models, which must be amortized after training. Cumulative execution time helps in the visualization of the amortization process. However, visualizing the overall gains achieved using different configurations is challenging to visualize using the cumulative execution time metric.

Some of the tuning frameworks [16, 38, 50] use execution time improvement percentages to measure the end-to-end performance gain. They compute the improvements (gain in execution time) caused by the tuner's best-recommended configuration and convert it to a percentage by comparing it with the execution time achieved with a baseline tuner. To evaluate the best-recommended configuration (after the tuner converges) from the tuner, many works in the literature have measured the execution time. Raw execution time achieved from a tuner's best configuration is also used to compare the efficiency of different tuners [18, 19, 21, 22, 25,

26, 31, 34, 38, 40, 41, 70, 71, 73, 79]. Some tuning frameworks [22, 38, 48, 60] measure the execution time by normalizing it using the base execution time (base execution time is achieved using a base tuner or default configuration). Similarly, some of the works [23, 32, 39] measure the speedup compared to the execution time achieved using the default configurations to evaluate the best configurations found by different tuners. Apart from the end-to-end execution time, some works in the literature have used stage-specific distributions to visualize and evaluate the impact of an obtained configuration on different stage execution times rather than the end-to-end execution time.

Bei et al. [18] and Yu et al. [73] use stage-specific distribution to measure the impact of good configurations at the stage level. When Big Data batch processing frameworks are used to execute SQL queries, it becomes challenging to measure the overall performance of a workload (where a workload consists of a set of SQL queries) as some queries will execute in parallel. There is a high possibility that parallel executing queries will not complete simultaneously. Hence, the end-to-end execution time will not be a good indicator for evaluating the performance of executing a SQL workload.

For SQL workloads, where the batch processing systems execute transactions (in the form of queries), different queries will have different execution times. Hence, in such cases, it is often feasible to measure the system's performance using throughput, where the throughput is measured in queries executed in unit time [38] and [41].

Measuring the performance improvement rate also helps evaluate the configuration quality [19, 31, 64]. The improvement rate signifies the gain in execution time (compared with a baseline) over the total tuning iterations. Dou et al. [31] measure the execution time distribution obtained using the best configuration, comparing the distributions' mean, median, and standard deviations. Similarly, Chen et al. [23] plot the Cumulative Distribution Function to compare systems' behavior of executing workload with different configurations. Some tuning frameworks aim to solve a multi-objective problem. Hence, the above-discussed metrics might not be a good choice to evaluate the tuning of such tuners.

Tuners that solve multi-objective optimization problems are often interested in measuring systems resource utilization [28, 41, 64, 79] as the configuration can greatly impact resource utilization. To measure the quality of configurations

(where the tuner is supposed to minimize the infrastructure cost), Cheng et al. [26] measure the total cost of individual executions. In multi-objective problems where the result is a Pareto optimal set. [64] [64] use uncertain space to find the most optimal candidates among the Pareto optimal candidates. Daud et al. [28] measure the resource utilization rate, where the utilization rate is represented as the sum of CPU & memory utilization and packet delivery ratio.

We summarize the workloads and benchmarks used for evaluating tuning frameworks that tune the batch processing systems in Table 6.2. As per our findings, it is clear that HiBench is the most popular benchmark used for batch-processing frameworks. Some highly used workloads in Hibench are PageRank, Naive Bayes, WordCount, KMeans, TeraSort, Sort, etc. The tuning frameworks have also preferred HiBench's SQL queries and SQL Benchmark [1] for SQL workloads.

6.6 Conclusion

This Chapter has presented a survey of the benchmarks and metrics used to evaluate the performance of tuning frameworks for Big Data processing frameworks. Our survey has explored the metrics used to assess the various components of the tuning frameworks, including the search-space explorer engine, performance prediction model, configuration pruning engine, and end-to-end tuning. The study has revealed the strengths and weaknesses of existing benchmarks and metrics by analyzing the current evaluation techniques. Ultimately, this chapter is a valuable resource for researchers and practitioners who aim to evaluate and compare the performance of tuning frameworks for Big Data processing frameworks.

In the next Chapter, we discuss related work, such as previous surveys in the configuration tuning of Big Data frameworks.

Table 6.2: Summarizing the batch workloads and benchmarks

Workload	Benchmark	Used in
Grep	BigDataBench [2]	[51, 70, 75]
PageRank	HiBench [5]	[16, 34, 35, 40, 60] [19, 25, 31, 48, 79]
Naive Bayes	HiBench [5]	[16, 34, 35, 60, 79] [25, 26, 70]
WordCount	HiBench [5]	[16, 34, 35, 40, 60] [19, 23, 26, 71, 79] [18, 23, 25, 32, 51, 70]
KMeans	HiBench [5]	[16, 19, 40, 60, 79] [25, 26]
Linear Regression	HiBench [5]	[26]
Support Vector Machine	HiBench [5]	[16, 31]
Gradient Boosting Trees	HiBench [5]	[16]
TeraSort	HiBench [5]	[34, 38, 60, 71, 79] [18, 26, 31, 41]
Sort	HiBench [5]	[19, 32, 48, 60, 70] [18, 23, 51]
NWeight	HiBench [5]	[26, 79]
PCA	HiBench [5]	[19, 48]
DFSio	HiBench [5]	[41]
SQL Queries	HiBench [5]	[25, 32, 41]
Alternating Least Squares	Manually Created	[75]
SGD Regression	Manually Created	[75]
Lasso's Regression	Manually Created	[75]
Twitter Link Prediction	Manually Created	[28]
Collaborative Filtering	Manually Created	[32]
Frequent Itemset Mining	Manually Created	[32]
Inverted Index	PUMA Benchmark [14]	[18, 32, 71]
LogQuery	Spark Default	[40]
TPC-H	SQL Benchmark [1]	[34, 35, 38]
TPCx-BB	SQL Benchmark [1]	[64]
Word2Vec	Spark-Perf [8]	[48]
PyPearson	Spark-Perf [8]	[48]
YCSB	YCSB Bench [13]	[41]

Chapter 7

Related Work

In this Section, we survey and closely review the previous surveys conducted to address the challenges of configuration tuning for Big Data frameworks.

Multiple works can be found in the literature that aims to improve the performance of Big Data Frameworks. Ousterhout et al. [58] showcase how networks can become bottlenecks while executing workloads on massive data sets. They further showcase how much performance gain the system would achieve if these bottlenecks were cleared using configuration parameter tuning. For stream processing systems, Hirzel et al. [44] surveys several optimization techniques that aim to optimize the performance of the stream processing systems. However, just a part of this work discusses tuning the batch-size-related configuration parameters. Similarly, it is observed that in [61], only the improvement of performance of stream-processing systems through parallelization and elastic methods is discussed, while the parameter tuning aspects are not addressed. And a lot of work in the literature aims to improve the execution plan for Big Data frameworks. We keep these works out of scope and only consider the works that aim to tune the configuration parameters of Big Data frameworks automatically.

There are already surveys in the literature investigating the different tuning approaches used for parameter tuning [27, 43, 45, 77, 78]. However, some surveys focused on tuning database system parameters using Machine Learning algorithms [77, 78] and not big data frameworks. Traditional SQL or NoSQL databases were not designed to handle the 5“Vs” (Volume, Velocity, Variety, Veracity, and

Value). Our scope of this survey is to focus only on big data processing frameworks or tools that can cater to the needs of 5 "Vs.". Hence, in this work, we limit the scope of our work to studying tuning systems for Big Data processing frameworks.

Other work [27, 43, 45] focuses on surveying the different categories of parameter tuning of big data processing frameworks (including Machine Learning based approaches). The survey in [43] studies six different types of tuners: rule-based, cost-modeling-based, simulation-based, experiment-based, ML-based, adaptive-learning-based tuners. A similar survey can be seen in [27] where authors summarize the ML-based tuners without explicitly discussing the challenges of different components. Autonomic computing principles (specifically related to control communities) are discussed in [45] that aim to reduce/minimize human involvement in finding optimal configurations. However, the techniques and methods discussed in [45] are quite different than ML-based techniques, so we keep them out of scope for this study.

On the other hand, tuners that use Machine-Learning algorithms face challenges like knowledge transfer, huge search space, workload classification, etc. To the best of our knowledge, the survey conducted by Zhang et al. [77] is the closest survey in terms of the survey questions they follow. However, their survey considers the database, not Big Data frameworks. The second closest survey to the current one is conducted by Herodotou et al. [43]. This survey studies the different types of tuning frameworks for Big Data frameworks but not discusses the specific challenges faced by the different components of ML-based tuning systems.

In this chapter, we covered various works in the literature that aim to improve the performance of Big Data frameworks, focusing on tuning configuration parameters. The previous surveys explore and discuss the different tuning approaches, including machine learning-based approaches. This Chapter also highlights the challenges machine learning-based tuners face, such as knowledge transfer, huge search space (high dimensionality), workload classification, etc.

Chapter 8

Conclusion

Configuration tuning in big data processing frameworks is necessary to optimize performance and resource utilization across the cluster of machines. This task is challenging due to the high diversity of workloads, rapid growth of data, changing software and hardware infrastructures, and inter-dependency between configurations.

The literature has extensively explored ML-based tuners to solve the auto-tuning problem. This essay studies the challenges related to the automatic ML-based tuners that treat the underlying Big Data processing frameworks system as a black box. More specifically, we consider the challenges faced by different components of ML-based tuning frameworks and define four survey questions based on the challenges. Further, following a systematic literature review plan, we discussed a comprehensive overview of the state-of-the-art techniques and summarized our findings for every survey question.

We investigated four research questions: (1) the different performance prediction models based on tuning frameworks and challenges related to knowledge transfer, (2) search-space pruning techniques and their trade-offs, (3) workload classification techniques and features, and (4) benchmarks and metrics used to evaluate tuner performance. The insights gained from these questions will be valuable for researchers and practitioners working in the field of performance tuning of big data frameworks.

Bibliography

[1] Tpc-h sql benchmark. http://www.tpc.org/tpch/. Accessed: 2023-01-06. → pages 54, 55

[2] Ibm bigdatabench. https://www.benchcouncil.org/BigDataBench/. Accessed: 2023-01-06. → page 55

[3] Spark cyclone benchmark. https://github.com/XpressAI/SparkCyclone. Accessed: 2023-01-06. → page 51

[4] Apache flink. https://flink.apache.org/. Accessed: 2023-01-06. → page 1

[5] Intel hibench. https://github.com/Intel-bigdata/HiBench. Accessed: 2023-01-06. → pages 51, 55

[6] Hadoop release history. https://www.geeksforgeeks.org/hadoop-history-or-evolution/. Accessed: 2023-01-06. → page 1

[7] Apache spark. https://spark.apache.org/, . Accessed: 2023-01-06. → page 1

[8] Databricks sparkperf. https://github.com/databricks/spark-sql-perf, . Accessed: 2023-01-06. → page 55

[9] Apache storm. https://storm.apache.org/, . Accessed: 2023-01-06. → page 1

[10] Inel storm benchmark. https://github.com/yahoo/streaming-benchmarks, . Accessed: 2016. → page 51

[11] Top 10 big data frameworks in 2022. https://www.eztek.vn/top-10-big-data-frameworks-in-2022/. Accessed: 2023-01-06. → page 8

[12] Yahoo streaming benchmark. https://github.com/yahoo/streaming-benchmarks. Accessed: 2023-01-06. → page 51

[13] Ycsb benchmark suite. https://github.com/brianfrankcooper/YCSB/. Accessed: 2023-01-06. → page 55

[14] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012. → page 55

[15] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017. → page 2

[16] L. Bao, X. Liu, and W. Chen. Learning-based automatic parameter tuning for big data analytics frameworks. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 181–190. IEEE, 2018. → pages 10, 16, 19, 48, 49, 52, 55

[17] J. C. Barsce, J. A. Palombarini, and E. C. Martínez. Towards autonomous reinforcement learning: Automatic setting of hyper-parameters using bayesian optimization. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–9. IEEE, 2017. → page 13

[18] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng. Rfhoc: A random-forest approach to auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1470–1483, 2015. → pages 10, 16, 20, 47, 52, 53, 55

[19] N. Ben Slimane, H. Sagaama, M. Marwani, and S. Skhiri. Mjolnir: A framework agnostic auto-tuning system with deep reinforcement learning. *Applied Intelligence*, pages 1–15, 2022. → pages 10, 16, 23, 26, 52, 53, 55

[20] M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 189–200, 2017. → page 10

[21] L. Cai, Y. Qi, W. Wei, J. Wu, and J. Li. mrmoulder: a recommendation-based adaptive parameter tuning approach for big data processing platform. *Future Generation Computer Systems*, 93:570–582, 2019. → pages 10, 16, 23, 52

[22] C.-C. Chen, K.-S. Wang, Y.-T. Hsiao, and J. Chou. Albert: An automatic learning based execution and resource management system for optimizing hadoop workload in clouds. *Journal of Parallel and Distributed Computing*, 2022. → pages 10, 16, 18, 39, 40, 43, 47, 48, 52, 53

[23] C.-O. Chen, Y.-Q. Zhuo, C.-C. Yeh, C.-M. Lin, and S.-W. Liao. Machine learning-based configuration parameter tuning on hadoop system. In *2015 IEEE International Congress on Big Data*, pages 386–392. IEEE, 2015. → pages 10, 16, 20, 47, 53, 55

[24] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 362–374, 2020. → pages 2, 35

[25] Y. Chen, P. Goetsch, M. A. Hoque, J. Lu, and S. Tarkoma. d-simplexed: Adaptive delaunay triangulation for performance modeling and prediction on big data analytics. *IEEE Transactions on Big Data*, 2019. → pages 10, 16, 21, 47, 52, 55

[26] G. Cheng, S. Ying, and B. Wang. Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model. *Journal of Systems and Software*, 180:111028, 2021. → pages 10, 16, 20, 47, 53, 54, 55

[27] R. L. d. C. Costa, J. Moreira, P. Pintor, V. dos Santos, and S. Lifschitz. A survey on data-driven performance tuning for big data analytics platforms. *Big Data Research*, 25:100206, 2021. → pages 1, 5, 6, 7, 8, 10, 56, 57

[28] N. N. Daud, S. H. Ab Hamid, M. Saadoon, C. Seri, Z. H. A. Hasan, and N. B. Anuar. Self-configured framework for scalable link prediction in twitter: Towards autonomous spark framework. *Knowledge-Based Systems*, 255:109713, 2022. → pages 10, 16, 21, 22, 27, 39, 41, 43, 48, 53, 54, 55

[29] A. Davoudian and M. Liu. Big data systems: A software engineering perspective. *ACM Computing Surveys (CSUR)*, 53(5):1–39, 2020. → pages 1, 52

[30] X. Ding, Y. Liu, and D. 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. → pages 31, 34

[31] H. Dou, K. Wang, Y. Zhang, and P. Chen. Atconf: auto-tuning high dimensional configuration parameters for big data processing frameworks. *Cluster Computing*, pages 1–19, 2022. → pages 10, 15, 16, 53, 55

[32] M. Ead. Pstorm: Profile storage and matching for feedback-based tuning of mapreduce jobs. Master's thesis, University of Waterloo, 2013. → pages 10, 37, 53, 55

[33] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper. Towards seamless configuration tuning of big data analytics. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1912–1919. IEEE, 2019. → pages 10, 16

[34] A. Fekry, L. Carata, T. Pasquier, and A. Rice. Accelerating the configuration tuning of big data analytics with similarity-aware multitask bayesian optimization. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 266–275. IEEE, 2020. → pages 2, 6, 10, 16, 17, 25, 30, 31, 32, 38, 39, 40, 44, 49, 52, 53, 55

[35] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper. To tune or not to tune? in search of optimal configurations for data analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2494–2504, 2020. → pages 2, 3, 6, 10, 15, 16, 30, 31, 32, 48, 49, 50, 52, 55

[36] M. Feurer, B. Letham, and E. Bakshy. Scalable meta-learning for bayesian optimization using ranking-weighted gaussian process ensembles. In *AutoML Workshop at ICML*, volume 7, 2018. → pages 5, 26

[37] M. Genkin and F. Dehne. Autonomic workload change classification and prediction for big data workloads. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 2835–2844. IEEE, 2019. → pages 6, 10, 37, 38, 39, 41

[38] M. Genkin, F. Dehne, M. Pospelova, Y. Chen, and P. Navarro. Automatic, on-line tuning of yarn container memory and cpu parameters. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 317–324. IEEE, 2016. → pages 10, 23, 39, 52, 53, 55

[39] A. Gounaris and J. Torres. A methodology for spark parameter tuning. *Big data research*, 11:22–32, 2018. → pages 10, 31, 34, 53

[40] J. Gu, Y. Li, H. Tang, and Z. Wu. Auto-tuning spark configurations based on neural network. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018. → pages 10, 47, 53, 55

[41] Q. Guo, Y. Xie, Q. Li, and Y. Zhu. Xdataexplorer: a three-stage comprehensive self-tuning tool for big data platforms. *Big Data Research*, 29:100329, 2022. → pages 10, 24, 53, 55

[42] Y. Guo, H. Shan, S. Huang, K. Hwang, J. Fan, and Z. Yu. Gml: Efficiently auto-tuning flink's configurations via guided machine learning. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):2921–2935, 2021. → pages 10, 16, 19, 30, 31, 32, 47, 48, 49, 50, 51

[43] H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)*, 53(2): 1–37, 2020. → pages 2, 3, 5, 6, 8, 12, 36, 56, 57

[44] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4): 1–34, 2014. → page 56

[45] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):1–28, 2008. → pages 4, 5, 6, 56, 57

[46] P. Jamshidi and G. 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. → page 6

[47] Z. Jia, J. Zhan, L. Wang, R. Han, S. A. McKee, Q. Yang, C. Luo, and J. Li. Characterizing and subsetting big data workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–201. IEEE, 2014. → pages 6, 10

[48] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee. Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading. In *2016 international conference on parallel architecture and compilation techniques (pact)*, pages 387–400. IEEE, 2016. → pages 10, 16, 22, 27, 39, 42, 48, 53, 55

[49] S. Kadirvel and J. A. Fortes. Grey-box approach for performance prediction in map-reduce based platforms. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2012. → pages 10, 15, 16, 17

[50] R. Krishna, C. Tang, K. Sullivan, and B. Ray. Conex: Efficient exploration of big-data system configurations for better performance. *IEEE Transactions on Software Engineering*, 2020. → pages 3, 10, 16, 24, 26, 39, 40, 43, 52

[51] P. Lama and X. 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, 2012. → pages 6, 10, 16, 18, 30, 31, 32, 39, 40, 43, 47, 55

[52] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015. → page 51

[53] T. Li, J. Tang, and J. Xu. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, 2(4): 353–364, 2016. → pages 10, 16, 18, 51

[54] Y. Li, Y. Shen, J. Jiang, J. Gao, C. Zhang, and B. Cui. Mfes-hb: Efficient hyperband with multi-fidelity quality measurements. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 8491–8500, 2021. → page 13

[55] Y. Li, Y. Shen, H. Jiang, W. Zhang, J. Li, J. Liu, C. Zhang, and B. Cui. Hyper-tune: Towards efficient hyper-parameter tuning at scale. *arXiv preprint arXiv:2201.06834*, 2022. → pages 2, 10

[56] G. Liao, K. Datta, and T. L. Willke. Gunther: Search-based auto-tuning of mapreduce. In *European Conference on Parallel Processing*, pages 406–419. Springer, 2013. → pages 10, 31, 34, 39, 40, 43

[57] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir. Panacea: Towards holistic optimization of mapreduce applications. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 33–43, 2012. → pages 6, 10, 30, 31, 34

[58] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. 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. → page 56

[59] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178, 2009. → pages 2, 10

[60] T. B. Perez, W. Chen, R. Ji, L. Liu, and X. 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. → pages 10, 24, 30, 31, 33, 49, 53, 55

[61] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52(2): 1–37, 2019. → page 56

[62] H. Sagaama, N. B. Slimane, M. Marwani, and S. Skhiri. Automatic parameter tuning for big data pipelines with deep reinforcement learning. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. IEEE, 2021. → page 10

[63] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *Proceedings of the VLDB Endowment*, 7(13):1319–1330, 2014. → pages 6, 10, 30, 31, 33

[64] F. Song, K. Zaouk, C. Lyu, A. Sinha, Q. Fan, Y. Diao, and P. Shenoy. Spark-based cloud data analytics using multi-objective optimization. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 396–407. IEEE, 2021. → pages 10, 16, 17, 51, 53, 54, 55

[65] M. Trotter, G. Liu, and T. Wood. Into the storm: Descrying optimal configurations using genetic algorithms and bayesian optimization. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 175–180. IEEE, 2017. → pages 10, 16, 17, 51

[66] M. Trotter, T. Wood, and J. Hwang. Forecasting a storm: Divining optimal configurations using genetic algorithms and supervised learning. In *2019 IEEE international conference on autonomic computing (ICAC)*, pages 136–146. IEEE, 2019. → pages 10, 16, 22, 27, 48, 51

[67] M. Twaty, A. Ghrab, and S. Skhiri. Graphopt: a framework for automatic parameters tuning of graph processing frameworks. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3744–3753. IEEE, 2019. → page 10

[68] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017. → pages 2, 38

[69] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang. Automating characterization deployment in distributed data stream management systems. *IEEE Transactions on Knowledge and Data Engineering*, 29(12): 2669–2681, 2017. → pages 10, 16, 21, 47, 50, 51

[70] G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters of spark based on machine learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 586–593. IEEE, 2016. → pages 10, 16, 22, 27, 48, 53, 55

[71] X. Wang, J. Zhang, and Y. Shi. Resource and job execution context-aware hadoop configuration tuning. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 116–123. IEEE, 2020. → pages 10, 16, 21, 47, 53, 55

[72] Z. Yu, W. Xiong, L. Eeckhout, Z. Bei, A. Mendelson, and C. Xu. Mia: Metric importance analysis for big data workload characterization. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1371–1384, 2017. → pages 6, 10, 37, 38, 39, 42

[73] Z. Yu, Z. Bei, and X. 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, 2018. → pages 2, 10, 16, 19, 47, 53

[74] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos. Elastic complex event processing exploiting prediction. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 213–222. IEEE, 2015. → pages 10, 16, 17, 50, 51

[75] N. Zacheilas, S. Maroulis, and V. Kalogeraki. Dione: Profiling spark applications exploiting graph similarity. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 389–394. IEEE, 2017. → pages 3, 10, 37, 39, 41, 44, 47, 55

[76] X. Zeng, S. Garg, M. Barika, A. Y. Zomaya, L. Wang, M. Villari, D. Chen, and R. Ranjan. Sla management for big data analytical applications in clouds: A taxonomy study. *ACM Computing Surveys (CSUR)*, 53(3):1–40, 2020. → page 7

[77] X. Zhang, Z. Chang, Y. Li, H. Wu, J. Tan, F. Li, and B. Cui. Facilitating database tuning with hyper-parameter optimization: A comprehensive experimental evaluation. 2021. → pages vii, 5, 6, 13, 19, 25, 26, 31, 36, 50, 56, 57

[78] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020. → page 56

[79] Z. Zong, L. Wen, X. Hu, R. Han, C. Qian, and L. Lin. Mespaconfig: Memory-sparing configuration auto-tuning for co-located in-memory cluster computing jobs. *IEEE Transactions on Services Computing*, 2021. → pages 10, 16, 20, 30, 31, 32, 47, 49, 53, 55