Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

A Lightweight Performance Estimation Approach for Spark

Author: Yancheng Zhuang (2688067)

1st supervisor:	Dr. Adam S. Z. Belloum	University of Amsterdam
daily supervisor:	Kristy James	Elsevier B.V.
2nd reader:	Dr. Zoltán Ádám Mann	University of Amsterdam

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

December 9, 2022

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Apache Spark has been broadly adopted as the standard data processing framework to derive values from big data. However, this brings new challenges for the cluster administrators, especially in terms of selecting the right cloud configuration, since there are many available instance types with different sizes. A bad selection leads to unnecessary excessive costs. The situation will get more challenging if the administrators have no knowledge of the Spark application which will be executed. There exist automatic solutions to identify the best configuration for a broad range of options, with high search costs. However, there are many large-scale non-recurring data analytic applications, which have practical cost constraints. The goal of this paper is to develop a performance model for Spark for simulation, which can estimate the optimal or near-optimal number of parallelisms for the application by running the application only once. The system also leverages sampled data to further lower the cost. Our experiments show that the system has a high chance of finding the optimal number of parallelisms, without the high overhead of developing the model and collecting the data.

Keywords: *Performance Model, Performance Estimation, Configuration Selection, Apache Spark, Big Data, Data Sampling*

Acknowledgements

This project was finished during my internship in the ICSR Lab at Elsevier. The internship offered me an invaluable chance to not only hone my technical skills but also conduct in-depth academic research. I feel very grateful to be able to join and be an intern at Elsevier. I would like to thank all my colleagues, especially my daily supervisor, Kristy James. I have learned so much while working alongside her, and I cannot thank her enough for the impact she has had on my professional growth. I am also grateful to my academic supervisor, Dr. Adam Belloum for all the academic guidance on this project, and to Dr. Zoltán for his valuable feedback on this thesis. Last but not least, a big thanks to all the support from my partner, my family, and my friends.

Contents

List of Figures

LIST OF FIGURES

List of Tables

Glossary

DAG **D**irected **A**cyclic **G**raph; a directed graph with no directed cycles

EBS **E**lastic **B**lock **S**tore, a cloud-based block storage system that is best used for storing persistent data, the volumes of which can be mounted as devices on the instances

EC2 **E**lastic **C**ompute **C**loud; secure, resizable compute in the cloud offered by Amazon Web Services

HDFS **H**adoop **D**istributed **F**ile **S**ystem; a distributed file system designed to run on commodity hardware.

Hive a data warehouse software project built on top of Apache Hadoop for providing SQL-like data query and analysis

JVM **J**ava **V**irtual **M**achine; a virtual machine that enables a computer to run Java programs

NVMe **N**on-**V**olatile **M**emory **e**xpress, a storage access and transport protocol for SSDs that delivers high throughput and fast response times

pandas **Pan**el **Da**ta; a fast, powerful, flexible and easy to use open source data analysis and manipulation tool

Parquet Apache Parquet, a popular columnar storage format in the Hadoop ecosystem

RDD **R**esilient **D**istributed **D**ataset; a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel

S3 **S**imple **S**torage **S**ervice; an object storage service offering industry-leading scalability, data availability, security, and performance offered by Amazon Web Services

SQL **S**tructured **Q**uery **L**anguage; a standardized programming language that is used to manage relational databases and perform various operations on the data in them

Standalone a simple cluster manager incorporated with Spark

YARN **Y**et **A**nother **R**esource **N**egotiator; a cluster resource management platform for distributed computing paradigms

1

Introduction

In the past decade, we all have witnessed a huge adoption of large-scale big data technologies across all kinds of sectors like banking(2), retail(3), healthcare(4) and IoT(5). Organizations are all starting to make use of the massive amount of data generated and collected to solve complex real-world data science questions, in order to further optimize their decisions and services. The ICSR Lab[1] is a platform that enables bibliometricians, scientometricians, informetricians and other researchers to run bibliometric data analyses using Elsevier metadata such as Scopus(6) and PlumX(7). Through free access to the Lab for non-commercial research studies, we want to help researchers to test their innovative ideas and methods, as well as contribute to the community in terms of collaboration and reproducibility.

In order to derive value from big data efficiently, numerous big data frameworks have been introduced to handle 5 V's of big data (velocity, volume, value, variety and veracity (8)). Among all these frameworks, Apache Spark(9) has become the de-facto standard for big data processing for its ease of use and high performance. At the time of being awarded the 2022 SIGMOD Systems Award, it has been downloaded 45 million times in PyPI and Maven Central alone and has been used in at least 204 countries and regions[2]. The ICSR Lab, which runs on the Databricks platform, runs on Spark.

As public/private cloud adoption grows in the enterprise sector, more and more users realize the benefits of deploying Spark on the cloud and start to run Spark on cloud infrastructure. In a cloud environment like AWS EC2, administrators have a wide range of choices in terms of the instance types and number of instances. Different instance types bring about different computing and I/O capability, while the number of cores set the

[1] https://www.elsevier.com/icsr/icsrlab

[2] https://databricks.com/blog/2022/06/15/apache-spark-and-photon-receive-sigmod-awards.html

maximum parallelism of a Spark cluster. For the time being, AWS currently offers nearly 400 different instances with choices across storage options, networking, and operating systems[1]. Given so many options to choose from, it is often quite hard for administrators to answer questions like *What is the cheapest instance configuration to finish this job given this configuration* or *Bigger clusters with fewer nodes or smaller clusters with more nodes*, especially without the a priori knowledge of the Spark application which is going to be executed on the cluster. Administrators need to read the source code of the application to get a sense of the scale of the workload, which is time-consuming and may not be practical because of other considerations like privacy. Administrators can consult the developers of the application, but the native support of SQL in Spark has attracted users like data analysts and data scientists, who typically don't have a deep knowledge of the underlying big data system. A more practical way to get around this problem is to run the application several times using different cluster configurations to observe the performance differences. It is, however, costly to develop such an empirical performance model, especially for applications that don't need to be run repeatedly and frequently. Also, it is often the case that data analysts and data scientists develop their applications on the sampled data since the ad-hoc queries on smaller data are more efficient, which helps their proof of concept and debugging of the application. The dry runs on the sampled data should provide great insights to the administrators on how to allocate the resources. Therefore, a lightweight approach to estimating the performance of the Spark applications and selecting the optimal instance using sampled data should be of great value to the cluster administrators.

This thesis aims at developing an approach to estimate the performance of nonrecurring Spark applications under different available resources. The approach includes a performance model, which should both have good accuracy in the estimated running time, and be lightweight, which means it won't incur excessive costs in building the model. How to use the model on sampled data for selecting the optimal instances is also investigated in this thesis, which covers a domain-driven stratified data sampling method to produce the sampled data properly. The rest of the thesis is organized as follows. In chapter 2, the background information about Spark and related works regarding the performance modeling on Spark are described, followed by a detailed problem statement with research questions. In chapter 3, the design of the solution to solve the research problems is described. In chapter 4, the implementation of the purposed solutions is introduced in detail. Experiments, comparisons, and results are presented in chapter 5. Finally, the discussion and future work of this project are presented in chapter 6.

[1]https://aws.amazon.com/ec2/instance-types/

2

Background

In this chapter, the background of this thesis work is introduced. Firstly, some technical background information on Spark is introduced. Then, current related works of performance modeling of Spark are described. Based on the actual requirements in the ICSR Lab and the related works in this field, problem statements and research questions are proposed.

2.1 Spark

2.1.1 Architecture

Spark has a well-defined layered architecture where all the Spark components and layers are loosely coupled. The architecture can be separated into user layer, data processing layer, resource management and task scheduling layer and physical execution layer. In the user layer, users need to prepare data, develop the application and configure parameters. The data processing layer converts the application code into a logical and physical plan according to the code and configuration. The resource management and task scheduling layer allocate resources needed and schedule the tasks to the suitable container. Finally, the physical layer executes the processing task.

From the system perspective, Spark adopts Master-Worker architecture. Take the standalone version as an example: if a Spark cluster consists of three nodes, the deployed architecture is shown in Figure 2.1. The master node is responsible for managing applications and tasks. Specifically, the *Master* daemon on the master node manages all worker nodes, i.e. allocating tasks to Worker nodes, collecting task run-time information on the worker nodes, detecting the heartbeats of the worker nodes, etc. The *Worker* daemons need to communicate with the master node and be responsible for the actual task execution

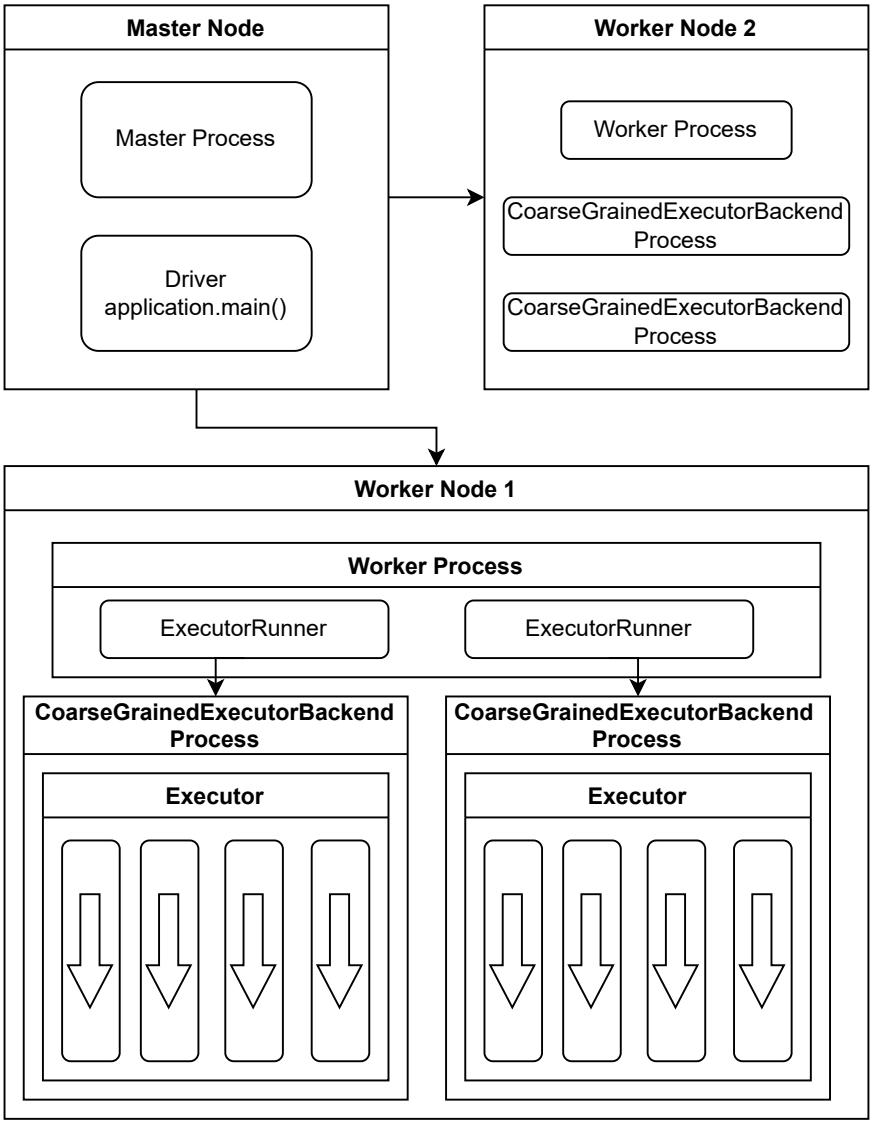

Figure 2.1: Spark system architecture

like starting up the executors for specific Spark tasks, monitoring the states of the tasks, etc. Some core concepts are described below:

- **Application**. One runnable Spark application, like *WordCount.scala*, which includes main() function. The typical process is reading the data from the data source, processing the data, and finally outputting the results. The application includes some configurations, like the number of CPUs needed, the size of the executor memory, etc. In order to implement the application, users can use either the data operations provided by Spark or some other frameworks like Spark SQL (which can translate

the SQL languages into Spark applications).

- **Driver**. The process runs the main() function of the application and creates the SparkContext. In Figure 2.1, the Spark application process (which is usually created by the SparkSubmit script) running on the Master node is the Spark Driver, which is independent of the Master process. SparkContext includes DAG scheduler, task scheduler, and SparkEnv which is a set of Spark environment managers.

- **Executor**. One unit of Spark computing resources. Spark uses cluster resources in the unit of executor and then allocates the detailed computing tasks to the executors. Physically, the executor in fact is a JVM process (called CoarseGrainedExecutorBackend in Standalone deployment mode), upon which can be run multiple threads (computing tasks). SparkEnv is also included in the JVM process on each executor.

- **Task**. When the driver is running the main() function in the Spark application, it divides the application into multiple computing tasks and allocate them to the executors. The task is the smallest computing unit in Spark and cannot be divided further. Tasks are run as threads in the process of the executor and execute computing tasks like the map operation, reduce operation, etc. Because the executor can be configured with multiple CPU cores, and usually one task only uses one CPU, multiple tasks can run in parallel in the executor. For example, in Figure 2.1, worker node 1 has 8 CPU cores. 2 executors are launched on this worker node, so 4 tasks can run in parallel in each executor. Notice that in the Databricks architecture, only 1 executor is launched on the worker node [1].

2.1.2 Logical Plan

There are four main sections to the logical plan:

- **Data source**. Spark needs to read the data from a certain source. The data can be placed on distributed file systems like HDFS, object stores like Amazon S3, and distributed key-value databases like HBase, etc.

- **Data model**. Hadoop MapReduce abstracts input, intermediate and output data as <Key, Value> record. The fine-granularity of the data representation means that the only way to work with data is using the map or reduce function, which lacks flexibility. Spark abstracts all the data on a higher level as RDD. Different from elementary

[1] https://docs.databricks.com/clusters/cluster-config-best-practices.html

data structures like *ArrayList*, RDD is only a logical concept, which means it will not be allocated storage space in memory (unless being cached explicitly). Three characteristics are associated with an RDD: dependencies, partitions and compute function. Firstly, dependencies can provide Spark with the ability to construct the RDD with its required input. Secondly, different partitions enable RDD to be processed by different tasks in parallel on different nodes. Finally, compute function produces the iterator for the data in the RDD.

- **Data operation**. Spark divides the operations into two kinds: transformation and action. The main difference is that action operations generally post-process data to generate the results, and will trigger Spark to submit an actual job. Transformation actions, on the other hand, are used to generate new RDDs.

- **Results processing**. There are two ways for Spark to generate results. One way is to aggregate the data to the driver side for the final computation, like counting the number of elements. The other way is to save the computation results directly to the distributed file system.

To generate the logical plan, Spark firstly generates the RDDs according to the transformation operations in the application code. The number of RDDs generated by Spark is more than the number of transformation operations because some complex transformation operations like *join* need to transform the source RDD multiple times. Then, Spark will develop data dependency between the partitions of the RDDs. Spark splits the data dependency into two groups: narrow dependency and shuffle dependency. Narrow dependency is the base class for dependencies where each partition of the child RDD depends on a small number of partitions of the parent RDD. More specifically, this kind of dependency can be divided further into 4 kinds:

- **One-to-one dependency**. The number of partitions of the child RDD and the parent RDD is equal, and the partitions in two RDDS have a one-to-one mapping relationship.

- **Range dependency**. A one-to-one dependency between ranges of partitions in the parent and child RDDs.

- **Many-to-one dependency**. One partition in the child RDD relies on multiple partitions in the parent RDDs.

- **Many-to-many dependency**. One partition in the child RDD relies on multiple partitions in the parent RDD, while one partition in the parent RDD relies upon multiple partitions in the child RDD.

Narrow dependencies facilitate data pipelining, which means transformations can be done in memory in a pipelined way to achieve better performance. In contrast, shuffle dependency means that one partition in the child RDD relied on multiple parts of the partitions in the parent RDDs, which require data from all parent partitions to be available and to be shuffled across the nodes. Executors need to fetch the data required for further transformations, either locally or remotely.

2.1.3 Physical Plan

When there are a large number of tasks, they put pressure on scheduling and data storage for intermediate data. In order to overcome this problem without sacrificing parallelism, Spark divides the jobs, stages and tasks in an appropriate way. There are four main procedures for Spark to generate the physical plan:

- **Divide the application into jobs according to the action operation**. The job corresponds to the whole process starting from the initial data input to the final action operation. If there are multiple action operations in the application, Spark will generate jobs in turn.
- **Divide the job into stages according to the shuffle dependency**. For each job, Spark backtraces the whole logical processing process from the last RDD. If it is a narrow dependency, then the parent RDD is brought into the stage and the backtracing continues. If it is a shuffle dependency, Spark stops back-tracing and make a new stage including all the stages that have been brought into.
- **Divide stage into tasks according to partitions**. Since the computation logic on each partition is the same and independent, Spark will decide the number of tasks according to the number of partitions in the last RDD in each stage.

The dependency between stages is shuffle dependency, which means each task in the child RDD needs to obtain part of the data from every partition in the parent RDD. The parent stage needs to partition the output data in advance, the number of which equals the number of tasks in the child stage. This process is referred to as Shuffle Write. Then, the tasks in the child stage read the data for its own partition through the network and then

aggregate the data from different partitions in the parent stage together. This process is referred to as Shuffle Read.

2.1.4 Memory Management

As shown in Figure 2.1, tasks are threads in the executor, which means multiple tasks share the same memory space of the executor. So memory management in Spark needs to balance the memory consumption from different sources and solve the memory sharing and competition between the tasks. In version 1.6, Spark implemented a unified model called UnifiedMemoryManager. The memory space is divided into 4 parts: reserved memory, storage memory, execution memory and user memory. Reserved memory is for storing internal objects in Spark. Storage memory is used for storing RDD caching data, broadcast data, part of the computation results from tasks, etc. Execution memory is for storing intermediate data during the shuffle phase. User memory is for storing intermediate computation results from user code and user-defined data structures in map functions, etc. Instead of using a static ratio to divide the memory space, the memory model adopts a dynamic approach.

The size of reserved memory is fixed at 300MB. Storage memory and execution memory are together called framework memory, of which the size is about 60% of the memory space ($spark.memory.fraction \times (heap - reserved\_memory)$, $spark.memory.fraction$ is 0.6 by default), and an initial ratio is given. The ratio can be adjusted dynamically, for example, if there is not enough space for shuffling then part of the storage memory can be borrowed by the execution memory. But the size of the storage memory should at least be around 50% of the memory ($spark.memory.storageFraction \times (spark.memory.fraction \times (heap - reserved\_memory))$, $spark.memory.storageFraction$ is 0.5 by default).

2.1.5 Spark SQL

Spark SQL(10) is a package built upon Spark that allows developers to issue SQL queries. The underlying engine can translate the queries automatically to RDD transformations and actions. At the core of the Spark SQL engine are the Catalyst optimizer and Project Tungsten. Together, these support the high-level DataFrame and Dataset APIs and SQL queries. Spark SQL is now de facto the primary and feature-rich interface to Spark's underlying in-memory distributed platform.

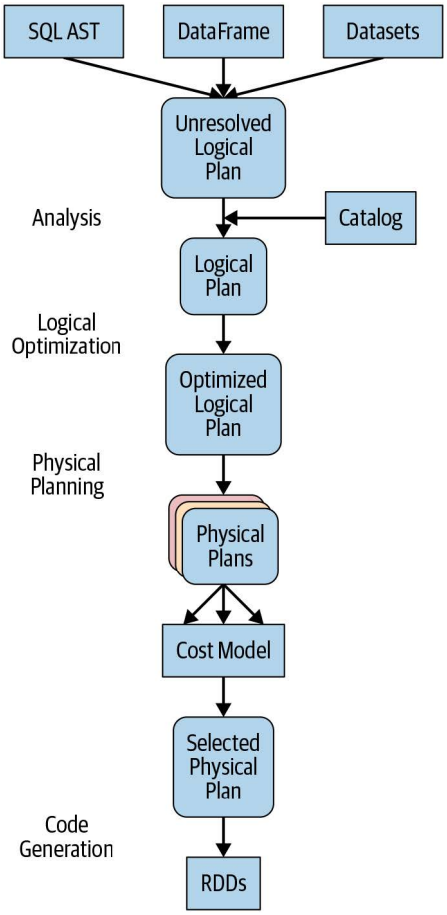

Figure 2.2: Major steps in Spark SQL (1)

2.1.6 Spark Listener

Spark provides several useful internal listeners that track metrics about tasks and jobs. During the development cycle, for instance, these metrics can help users to understand when and why a task takes a long time to finish. Examples of metrics are the number of active tasks, jobs/stages completed and failed, executor CPU used, executor run time, garbage collection time, shuffle metrics, I/O metrics, metrics with memory usage details, etc. Developers can attach to Spark monitoring data using the developer API, which is called a Spark Listener. So developers can write a custom class, extend the Spark Listener, write methods that react to events and collect data and process data. The executor metrics instrumentation can measure data of memory usage per memory component, and also provide values of the peak measurements.

2.2 Performance Modeling of Spark

Several works exist that look into various aspects of Spark performance modelling using different approaches. These approaches can be roughly divided into three categories: black-box, white-box and grey-box modelling. White-box models need to symbolize the interactions between various system-internal components which needs a strong understanding of the underlying system, while the black-box model or machine learning model needs historical performance data of the workloads to establish a model which automatically learns the relationship of performance with the interactions of the system components. A grey-box model is a combination of the white-box model and black-box model and is intended for taking the best of both approaches.

2.2.1 Black-Box Modelling

The black-box modelling approach is the most popular approach to address this problem due to its simplicity and high accuracy. Black-box models can be based on regression models (11, 12, 13, 14, 15, 16, 17), classification models (18) and parameter optimization (19, 20).

Regression is the most suitable model for performance modelling since the most expected modelling performance metric is the running time, which is a continuous value. *Ernest* (11) is one of the first black-box models to predict the execution time based on a specified instance configuration, given the job and input data size. To be more specific, they summarised high-level computation and communication patterns in Spark and tried to predict the overall execution time using a linear regression model. Only input data size and number of nodes are used as features in the model. For computation patterns, time is positively correlated to input and negatively correlated to a number of nodes. For communication patterns, tree DAG communication has a logarithmic relation with time, all-to-one communication has a linear relationship with time, and one-to-one communication is a constant factor in time. Instead of collecting the training of all the combinations of input data size and number of machines, they made use of *optimal experiment design* from Statistics which is supposed to minimize the trace of the matrix that represents all the options, while the matrix is subjected to a bounded total budget. We also use *Ernest* as the the baseline to compare with our work, since it is the widely used and open-sourced.

Based on *Ernest*, other works try to make improvements in two aspects. Firstly, add more features to a more sophisticated model including LASSO(12), neural network(12, 13, 17), decision tree(12, 13), support vector machine(12), ensemble learning models(12, 14, 17).

Secondly, control the cost of training by using techniques like Latin hypercube sampling (13), projective sampling (14), and simulated Bayesian optimization (16).

Wang et al. (18) also proposed a classification-based black-box model to predict the execution time with Spark configurations and the size of input data. Random search is performed over the parameters for collecting training data, of which values are selected uniformly at random. A binary classification model is built to predict whether the execution time of a job based on a set of configuration parameter values is improved. The ones classified as improved will be further used in a finer-grained multi-classification model based on the improvement.

Cherrypick is a pure parameter optimization-based model proposed by Alipourfard et al. (19), which generates the optimal or near-optimal instance configuration that minimizes cloud usage cost, guarantees application performance and limits the search overhead. To be more specific, the model is based on *Bayesian optimization*, which estimates a confidence interval of the cost and running time of each candidate cloud configuration. It has two functions. A prior function is used for black box modelling, while an acquisition function is used for choosing the next configuration based on the calculated expected improvement in comparison to the current best configuration. When the expected improvement is less than a threshold and at least N cloud configurations have been observed, the model stops the search.

Instead of predicting the execution time, Marco et al. (15) proposed a black-box model predicting the memory footprint. For an incoming application, the framework first extracts the features of the program using system-wide profiling tools including *vmstat*, *perf* and *PAPI*. Based on the feature values, it predicts which of the off-line learned memory functions best describes the memory behaviour of the application based on *K-nearest neighbours*. It then instantiates the function parameters by profiling the application on some small sets of input data items.

Black-box-based models typically produce good results, however, it incurs a substantial cost for building the model, because black-box models are empowered by enough training data. While several alternative solutions have been proposed to control the cost, black-box-based models are still best to be used for recurring jobs.

2.2.2 White-Box Modelling

Fewer works build the performance model based on the white-box modelling approach. Researchers have tried different methods, including simulation (21), queuing network (22, 23), Petri net (24) and computational geometry (25).

Wang and Khan (21) proposed a fine-grained white-box model to predict execution time and I/O cost, under the same instance configuration. The model built is of a hierarchical top-down fashion, firstly considering job time, secondly stage time and lastly task time. In order to predict the performance, various performance metrics like run time, I/O cost and memory cost need to be collected. The number of tasks executed in the actual job is the main factor for predicting the performance of the actual job.

Ardagna et al. (22, 23) evaluated two different white-box parallel computation performance modelling approaches to predict the execution time of the Spark application. The first model is based on a simple upper bound on the average execution time for Fork-Join queuing networks(26) and is referred as Fork-Join. In this model, tasks are forked into identical subtasks which are joined once they are completed by corresponding servers. It depends only on the number of parallel tasks and the average execution time of a single task which can be estimated based on historical data. The key factor that affects the estimation is the harmonic number. The second model modifies the Mean Value Analysis technique for queuing networks to account for delays caused by synchronization and resource constraints in DAG(27) and is referred to as the Task Precedence model. The model uses DAG and the average execution time of each individual stage as input. The model estimates the overlap probability between each pair of tasks based on DAG which is used as an inflation factor.

Karimian-Aliabadi et al. (24) proposed an analytical white-box model to predict the execution time of Spark applications using the YARN scheduler. The YARN scheduler is widely adopted for enabling Spark to run alongside other Hadoop workloads. The model is based on Stochastic Activity Networks, which is a probabilistic generalization of Activity Networks from Petri Nets. A monolithic model was firstly proposed for the simple double-queues scenario, which is proved not scalable because of the proportional growth of the state space size with the multiplicity.

Chen st al. (25) developed a white-box model to predict the execution time of Spark applications. By taking inspiration from the field of Computational Geometry, they constructed a d-dimensional mesh using Delaunay Triangulation over a selected set of features. Delaunay Triangulation partitions the d feature space into a set of interconnected d-simplexes, which helps to avoid overfitting. The prediction of the execution time is done by calculating a hyperplane in d+1 dimensional space by bringing in the runtime dimension. Also, adaptive sampling was integrated to minimize the samples needed.

While the white-box model doesn't need lots of training data to make the model effective, typically the performance is worse than black-box models, since white-box models need to

make some assumptions when building the model, which is sometimes at the cost of the prediction accuracy. Also, some models (24) are complex to solve, which might also incur extra costs for running the model.

2.3 Problem Statements

Performance modelling on Spark has been under active research recently. Several works have achieved great accuracy regarding the prediction of the running time of Spark jobs using different black-box modelling techniques. It can be seen that these works majorly aim at recurring jobs, which means that the Spark application is probably executed frequently, e.g. daily or hourly. It is reasonable to develop a sophisticated performance model for this type of job, which potentially trims an enormous amount of costs that cover the costs to develop the model. However, it is not practical for non-recurring jobs. To illustrate this case, we give a simple example - it takes $200 to run a Spark application on the worst cluster configuration A, while $50 on the optimal cluster configuration B. For a black-box model like Cherrypick (19), it typically needs 10 iterations on the sampled configuration to converge. Suppose the application is executed 5 times, and since Cherrypick continues to find a better configuration, we suppose the average cost for the system to run the application is $125. So, the total cost will be $1500, which is enough for the application to be executed on the worst cluster configuration 7 times.

Based on these practical problems that were encountered with cluster administration, the most important problems are summarised in the following points:

- Cluster administrators need an approach to choose a suitable cluster configuration given an application. Ideally, without performance modelling, optimal cluster configuration can be chosen based on the knowledge of both the application and the system. However, it is usually not possible for administrators due to some objective and practical reasons. Allocating the cluster in an arbitrary way will simply lead to unnecessary costs or even the failure of the execution.

- Proposed black-box-based solutions require training data or guided optimization-based searching on the cluster configuration candidates, which incurs the excessive cost of building the model. It is unreasonable to introduce extra costs for nonrecurring jobs.

- Developers typically develop their applications on the sampled data, which should provide insights into the cluster configuration. Running the application on the sampled data will also finish in a shorter time, which leads to lower costs. However, only running the application on the representative sampled data can contribute to the selection of the optimal instance for the application to run on the full data.

2.4 Research Questions

According to the problems encountered with performance modelling on Spark, the research questions defined are described as follows:

- RQ1: How to estimate the performance of the Spark applications so that we can choose the optimal number of instances while minimizing the cost?
  - I wish to estimate the performance with inconsiderable cost, while still achieving decent accuracy. As discussed above, most of the current works focus on improving the accuracy for recurring Spark applications, which are not suitable for the nonrecurring workloads like the ones in the ICSR Lab.
- RQ2: How to make use of sampled data to estimate the optimal parallelism given a Spark application?
  - I wish to further reduce the cost of the estimation of optimal parallelism for running the application on the sampled data, with acceptable accuracy loss. Several works mention using sampled data for the model, however the sampling methods are not described or just done randomly.

3

Design

This chapter demonstrates the performance model we propose. Firstly, the components that need to be incorporated into the performance model of Spark are described. Then, the proposed analytical hierarchical performance model is introduced in details.

3.1 Performance Model of Spark

As described previously, Spark generates a physical plan by dividing the application in a layered fashion, which includes jobs, stages, and tasks. Intuitively, a hierarchical model can be built to predict the execution time. However, there are some intricacies to the plan generated by the modern Spark.

Firstly, there are multiple levels of parallelism in Spark, which is demonstrated in Figure 3.1. As described previously, the tasks are the smallest unit in the execution in the Spark applications, which are executed in parallel on the executors. However, it doesn't necessarily means that the tasks that can be executed in parallel all come from one single stage. In fact, Spark introduces stage-level parallelism and job-level parallelism, which means the tasks can come from different stages, even different jobs. Stage-level parallelism often occurs when two stages have no direct dependencies on each other, so they can be scheduled to run safely in parallel. In Figure 3.1, in each job, stage 1 and stage 2 don't have serial dependencies, so they can be grouped together so that they can run safely in parallel. In terms of job-level parallelism, firstly, within each Spark application, multiple jobs can be running concurrently if they were submitted by different threads. Spark's scheduler is fully thread-safe and supports this use case to enable applications that serve multiple requests from multiple users. Secondly, in Spark SQL, an execution id is defined for every generated Spark job, so that the ones can be stitched together and can be seen a

App	Spark application
Job_d	job sets executed solely on the driver
Job_e	job sets executed on the executors
$JobG$	groups of jobs
$JobSG$	subgroups of jobs which can be executed in parallel
$StageG$	groups of stages whichcan be executed in parallel
$Stage$	stages
$Task$	tasks

Table 3.1: Notations in the model

single structured query execution sharing the same ID. However, these stitched jobs don't necessarily mean that they can run in safely parallel. According to our observations, jobs in these groups can also have some sort of dependencies.

Secondly, the compute time on the driver should also be included in the model. First of all, given a Spark SQL query, Spark generates an abstract syntax tree, applies local optimizations using pre-defined rules and a cost-based optimizer, generates efficient Java bytecode, and finally constructs the DAG graph between stages. All these steps will be performed on the driver side. If the query is extremely complex, then it takes Spark a significant amount of time for these steps before executing the application. Moreover, it is often the case the driver is running some serial tasks while the executors are idling. For example, the collect operation is often used by users to collect some intermediate results from the executors and do some sequential computation for the following jobs in the application. Similarly, data scientists and analysts tend to convert the Spark DataFrames to pandas DataFrames for further analysis and visualizations, whereas internally Spark needs to collect the RDDs to the driver. I/O operations also contribute to the running time of the driver. Take Hive tables as an example: Spark writes the Hive table in a temporary location. Once the computation is over, Spark copies the table to the final location. It is a minor issue if the underlying file system is HDFS since it is just a constant-time virtual rename operation. But if it is object storage like S3, files are copied to the new location physically and then files under the temporary location are deleted, which significantly increases the time on the driver side.

With these two points in mind, the hierarchical analytical model is developed as follows. We use the following notations to represent a Spark application:

$$App = \{Job_d, Job_e\} \tag{3.1a}$$
$$Job_e = \{Job_eG_i \mid 1 \le i \le \#Job_eG\} \tag{3.1b}$$
$$Job_eG_i = \{Job_eGS_{i,j} \mid 1 \le j \le \#Job_eGS_i\} \tag{3.1c}$$
$$Job_eGS_{i,j} = \{StageG_{i,j,k} \mid 1 \le k \le \#StageG_{i,j}\} \tag{3.1d}$$
$$StageG_{i,j,k} = \{Stage_{i,j,k,m} \mid 1 \le m \le \#Stage_{i,j,m}\} \tag{3.1e}$$
$$Stage_{i,j,k,m} = \{Task_{i,j,k,m,n} \mid 1 \le n \le \#Task_{i,j,k,m,n}\} \tag{3.1f}$$

And it should meet the following condition:

$$\forall i \in [1, \#Job_eG], j \in [1, \#Job_eGS_i], k \in [1, \#StageG_{i,j}],$$
$$\sum_{m=1}^{m=\#Stage_{i,j,k,m}} \sum_{n=1}^{n=\#Task_{i,j,k,m,n}} Task_{i,j,k,m,n} \le \sum_{e=1}^{e=\#executors} \#cores_e \tag{3.2}$$

The meanings of the notations in the equations are listed in Table 3.1. Eq. 3.2 means that at any time, the total number of tasks running in parallel on the cluster should be less or equal to the maximum parallelism, which is the total number of cores in the system. Then, the execution time of a spark application can be represented as follows:

$$T(App) = T(Job_d) + \sum_{i=1}^{\#Job_eG} T(Job_eG_i) \tag{3.3a}$$
$$T(Job_eG_i) = \sum_{j=1}^{\#Job_eSG_i} T(Job_eSG_{i,j}) \tag{3.3b}$$
$$T(Job_eSG_{i,j}) = \sum_{k=1}^{\#StageG_{i,j}} T(StageG_{i,j,k}) \tag{3.3c}$$
$$T(StageG_{i,j,k}) = \max_{m=1}^{\#Stage_{i,j,k}} T(Stage_{i,j,k,m}) \tag{3.3d}$$
$$T(Stage_{i,j,k,m}) = \max_{n=1}^{\#Task_{i,j,k,m}} T(Task_{i,j,k,m,n}) \tag{3.3e}$$
$$T(Task_{i,j,k,m,n}) = T(SF\_RD) + T(Ser) + T(Run) + T(DSer) + T(SF\_WT) \tag{3.3f}$$

Here $T(Job_d)$ is the running time for the sequential executions on the driver which precedes the following dependent executor jobs. $T(SF\_RD)$ is the time for the executor to read remote shuffle blocks. T(Ser) is the result serialization time. T(Run) is the actual running time for the transformations. T(DSer) is the time spent to deserialize the task.

T(SF_WT) is the time spent blocking on writes to disk or buffer cache. So, the overall model (with the constraint condition Eq. 3.2) can be developed as follows:

$$T(App) = T(Job_d) + \sum_{i=1}^{\#Job_eG} (\sum_{j=1}^{\#Job_eSG_i} (\sum_{k=1}^{\#StageG_{i,j}} (\max_{m=1}^{\#Stage_{i,j,k}} (\max_{n=1}^{\#Task_{i,j,k,m}} (T(SF\_RD) + T(Ser) + T(Run) + T(DSer) + T(SF\_WT)))))) \tag{3.4}$$

Based on the above equations, the running time of the Spark application can be modelled. The developed application needs to be executed on the cluster once in order to collect the application context information, which includes the metrics of the tasks and the dependencies between stages. However, in order to estimate the performance under the different numbers of available parallelisms, a simulation method based on this model is needed, which is described in the following chapter.

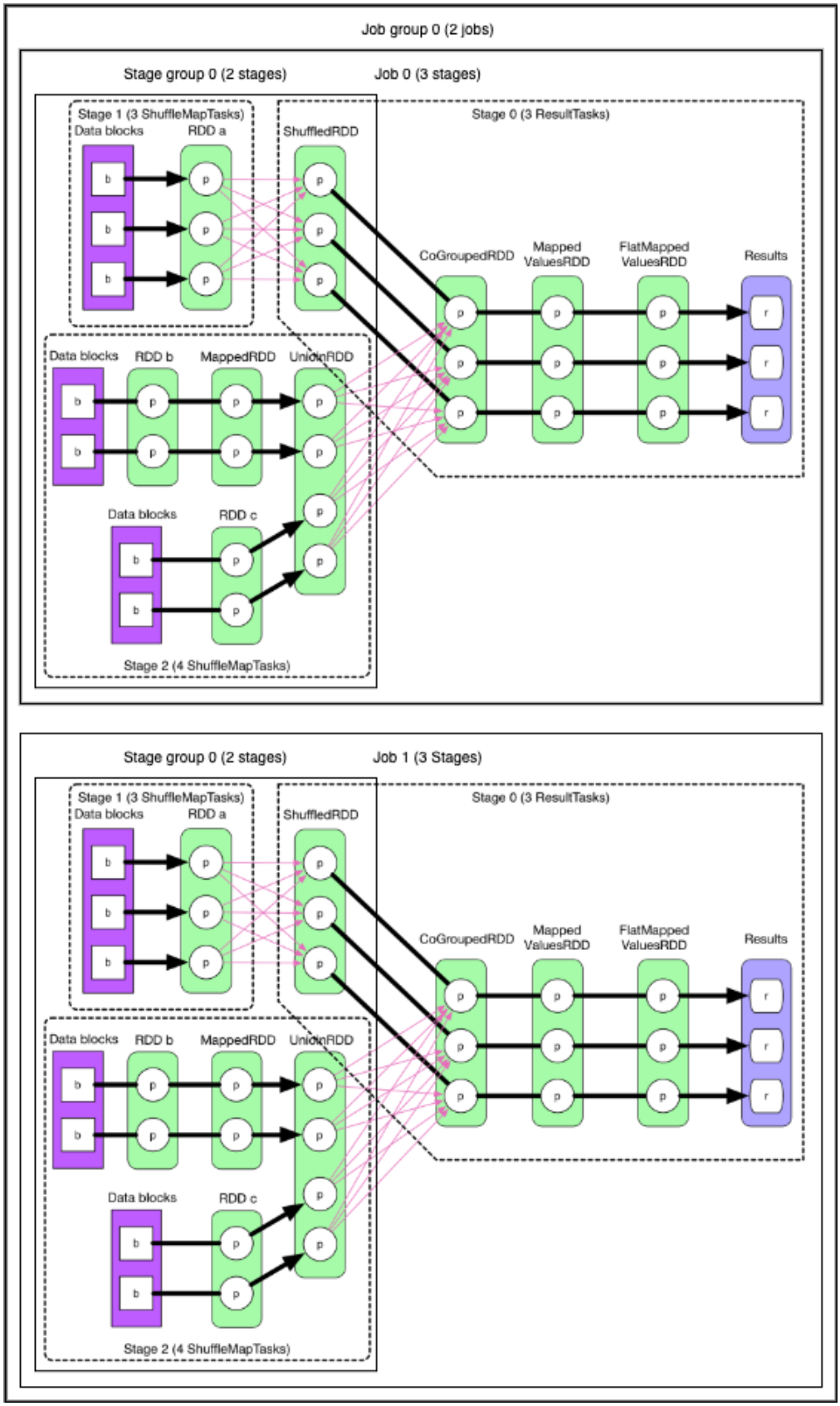

Figure 3.1: Multiple levels of parallelism in Spark

4

Implementation

In this chapter, we illustrate how to use the performance model built in Chapter 3 to estimate the performance of the Spark application under different available parallelisms. We first describe how to collect application context information in Spark. Then, the simulation method is introduced. Lastly, how to make use of the proposed method to select the best cloud configurations using limited amounts of sampled data is introduced with the corresponding data sampling approach.

4.1 Application Context Collection

In the dry run of the application, the application context information is collected by a set of customized job listeners extended from SparkListener. A corresponding listener is triggered once a certain event is sent from the DAG scheduler to collect the data. For example, when a task is completed, the DAG scheduler posts a SparkListenerTaskEnd event. Then, the internal SparkListenerBus relays the event to its listeners. Among all the listeners, the onTaskEnd listener is matched to the event which leads to the triggering of the customized data collection logic implemented in the listener. The overall structure of the Spark listeners is shown in Figure 4.1.

Listeners collect the metrics needed as shown in Equation 3.4.

4.2 Simulation Method

Based on the collected application context, we make some transformations so that the the collected information can be applied to the analytical model. Since the time spent on the driver is not captured by the metrics and events, it needs to be extracted based on the

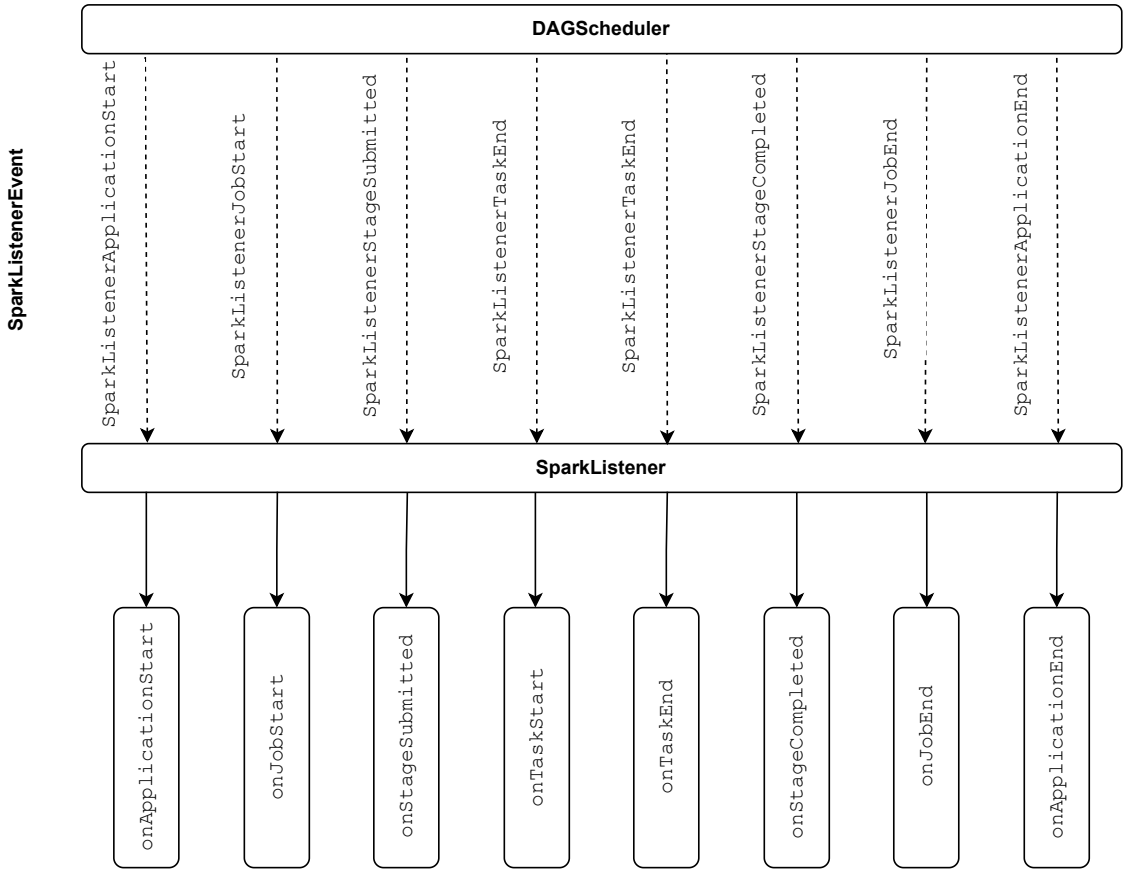

Figure 4.1: The overall structure of the Spark listeners

measured running time on the executors. Firstly, the job groups Job_eG are constructed by being grouped by the *spark.sql.execution.id*. If some jobs don't have this property, they themselves form individual job groups containing themselves. Thus, in each job group, all the jobs have the same *spark.sql.execution.id*. However, in some cases, there still exists some dependencies inside such formed job groups, which are not captured by any events in Spark. So, the start time and end time information of a job is utilized to recognize such dependencies. It is assumed that if the start time of group A is bigger than the end time of group B, then group A is dependent on group B. Once such dependencies are recognized, the groups are divided into subgroups such that the final job groups are composed of a series of subgroups which are executed in parallel.

After finishing constructing the job groups Job_eG, based on the assumption that the driver time remains a constant number regardless of the available parallelism of the cluster, the driver time can be computed as:

$$T(Job_d) = T^*(App) - \sum_{i=1}^{\#Job_eG} T^*(Job_eG_i) \tag{4.1}$$

where T^* stands for the measured time in the first dry run. The construction of stage groups and stages is similar to the job group and jobs, which is based on the time span.

Then, the simulator takes the constructed stages and their dependencies between each other as inputs. There are four states for the stages in the same job group: waiting, runnable, running, and completed. Each stage can only be associated with one of the four states. Firstly, the simulator initializes the states of the stages and finds the list of runnable stages by traversing the dependency graph for each job in the job groups, which is described in Alg. 1. The traversal starts from the largest stage ID inside each job group, and during the traversal, the stages are categorized and apprehended to one of the ordered state sets.

Algorithm 1 Stage States Initialization in the Job Group

```
1: waitingStages ← ∅
2: runnableStages ← ∅
3: runningStages ← ∅
4: function scheduleStage(stage)
5:     if stage has no parents then
6:         runnableStages ← {stageID} ∪ runnableStages
7:     else
8:         if stage has no parents in stageGroup then
9:             runnableStages ← {stageID} ∪ runnableStages
10:        else
11:            waitingStages ← {stageID} ∪ waitingStages
12:            scheduleStage(parentStage)
13:        end if
14:    end if
15: end function
```

After initializing the correct states for the stages, the actual simulation begins. The simulator consists of stage simulator and task simulator. The stage simulator is mainly responsible for the state transition of the stages. It loads stages from the runnable stages, performs the state transitions and tries to push the stages to the task scheduler. The task scheduler performs the simulation based on the priority queue, using the collected metrics

in the dry run and the given number of available parallelism that can be utilized. If there are no more runnable stages for the stage simulator to process, the task scheduler will be called to dequeue all the tasks in the queue which leads to the completion of the stage. Once one stage is completed, runnable stages are updated and then the stage simulator moves to the next iteration. Alg. 2 shows the detailed procedures.

After all the simulations of the job groups finish, the final estimated time of the Spark application can be computed by adding the driver time and the sum of all the simulated running times of the job groups.

Figure 4.2 shows the major steps. The proposed solution is implemented as a plugin, which can be attached to Spark without modifying the source code of Spark.

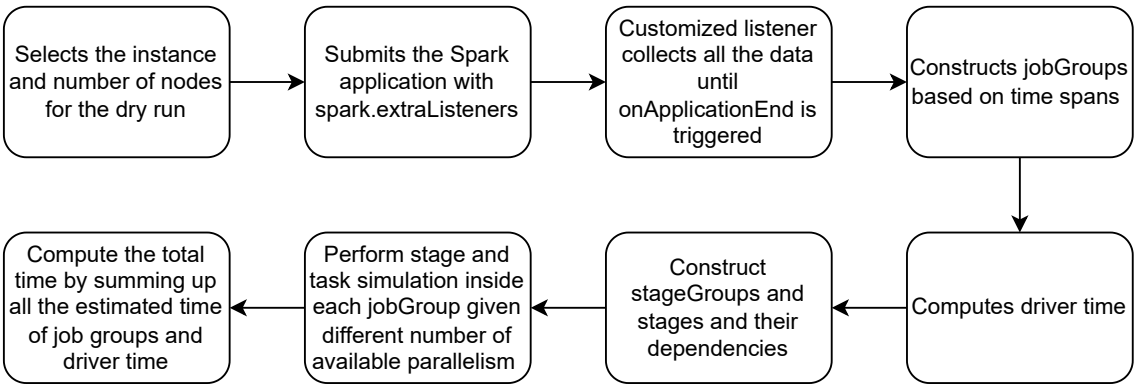

Figure 4.2: Major steps of the performance estimation

4.3 Sampling Method

The previous section describes how to estimate the performance of the Spark application under different parallelisms. However, we would like to further estimate the performance using only a subset of data. Sample data indicates smaller data, which leads to a shorter running time for the simulation which incurs less cost. It also makes the model more useful since according to the use case, the Spark application will only be executed a couple of times, even once or twice, which is the most frequent case encountered in the ICSR Lab. Under such cases, our current two-pass approach can hardly suffice the cost-saving goal, except that the instance is chosen really badly, for example, to run a tiny-scale application across tens of nodes. However, it is not feasible to predict the performance of the application on the full data by running on the sample data a limited number of times. Take the

Algorithm 2 Stage and Task Simulation in the Job Group

```
1: function DequeOneTask(stage)
2:     finishedTask ← dequeue(taskQueue)
3:     wallClock ← finishingTime(finishedTask)
4:     #pendingTasks(currentStageID) ← #pendingTasks(currentStageID) − 1
5:     if pendingNumTasksInStage = 0 then
6:         runningStages ← runnableStages \ {currentStageID}
7:         nowRunnableStages ← ∅
8:         for all stage ∈ waitingStages do
9:             if parents(stage) are completed then
10:                nowRunnableStages ← nowRunnableStages ∪ {stage}
11:            end if
12:        end for
13:        waitingStages ← waitingStages \ nowRunnableStages
14:        runnableStages ← runnableStages ∪ nowRunnableStages
15:    end if
16: end function
17: while #completedStage ≠ #stages do
18:    if runnableStages ≠ ∅ then
19:        eligibleStages ← runnableStages \ runningStages
20:        if eligibleStages ≠ ∅ then
21:            currentStageID ← head(eligibleStages)
22:            runningStages ← runningStages ∪ {currentStageID}
23:            runnableStages ← runnableStages \ {currentStageID}
24:            tasks ← {task | getStageID(task) = currentStageID}
25:            for all task ∈ tasks do
26:                if |taskQueue| = numOfCores then
27:                    DequeOneTask()
28:                end if
29:                Enqueue(taskQueue, wallClock + taskTime, stageID)
30:            end for
31:        end if
32:    else
33:        while |taskQueue| ≠ 0 do
34:            DequeOneTask()
35:        end while
36:    end if
37: end while
```

sorting process in the shuffle read in Spark as an example (we assumed that there is a need for aggregation and sorting), the records in the buffer will be aggregated using ExternalAppendOnlyMap, which is a HashMap-like customized data structure. Instead of the native implementation using Array and LinkedList in Java HashMap, ExternalAppendOnlyMap in Spark only uses Array for storing the elements and the hash value of the element is used for locating the position. Spark uses Timsort algorithm to sort such data structure (28), of which the time complexity is $\mathcal{O}(n \log n)$ in the worst case (29), but it is not possible to estimate the time since the constants and low-order terms are unknown. The parameters can be estimated using curve fitting techniques or machine learning. However, these solutions are inappropriate in our case since they need to collect a lot of data generated from the actual application executions for modeling the relationship.

In order to still achieve the goal of estimating the performance, we make use of the established simulation method to select the optimal instance directly instead of firstly estimating the running time. This is done by choosing sampled data that is representative of the full data so that the performance characteristics can be captured. The generated running time estimation curve should look alike the curve on the full data.

As discussed above, a good data sampling approach that results in representative sample data is selecting the optimal instance. By the word representative, we mean the sample should keep the probability distribution of the population of the full data under a reasonable significance level. If we take a look at the histogram of the sample data, ideally it should be the same as the histogram of the population. This also indicates the sample can reflect the data skew or biased information in the full data. However, it is challenging to get representative sample data. In most cases, a dataset is made of several variables, and it is very difficult to perform the sampling on a multivariate distribution, especially if the dataset is mixed with both numerical and categorical variables. Also, the more variables in the data, the less chance the sample data follows the distribution of the full data.

In order to address these challenges, a domain knowledge-driven stratified sampling-based approach is proposed. To simplify the problem, each variable is assumed to be independent of the others. If each one of the single, uni-variate histograms of the sample columns is comparable with the correspondent histogram of the population columns, we can assume that the sample is not biased. The general process has the following steps:

- Identify N variables which need stratification. The subset of the columns selected should be based on the domain knowledge of how end users use the data. For example, the columns frequently used as keys in the Spark transformation operations should

be selected since the data skew problem caused by imbalanced data distribution has a strong negative effect on the performance of Spark. In contrast, columns that contain text or binary data should not be included.

- Perform equal-frequency bucketing on the selected N variables into M buckets. For the columns with numeric variables, the data bucketing technique is used so that they are grouped into a smaller number of buckets. In order to output the columns with binned categorical data, the quantiles of the columns are needed. Greenwald-Khanna algorithm (30) is utilized for the computation. The number of buckets should be chosen appropriately since a smaller number leads to worse sampling performance while a bigger number leads to small groups (especially if there are many columns that need to be sampled). For the columns with categorical data, if the number of groups is large, similar groups should be merged.

- Use a hash list to generate the final column for the sampling. All the selected groups need to be hashed in order to unify the length and be concatenated and hashed together. The reason behind using a hash list is that plain concatenation of the input values without hashing them can be unsafe, e.g. "12" || "3" = "1" || "23".

- Perform stratified sampling is performed based on the final generated column. The sampling ratio R is a key factor that decides the similarity and distance between the cost curve using the full data and the cost curve using the sampled data.

Since the new variable is categorical, Pearson's chi-square test (31) can be used to compare the sample with the full data to see if they come from the same distribution. If the test is not significant, either perform another stratified sampling or adjust the parameters N and M. These two parameters affect the data sampling process in different ways. N enables us to choose the number of variables to merge. A large N leads to an excessive number of buckets in the new variable, thus negatively impacting the sampling result. This number is derived based on the domain knowledge of the usage pattern of the end-users, so the identified variable contains the most used variables. M helps to fix the resulting number of buckets for the numerical variables. A large M retains more information, but it will also results more buckets in the new variable, so that there are more distinct values in the final generated column which affects the result of the sampling negatively.

To store data in the file system, the sampled data will be partitioned into P partitions, where P is the number of partitions the full data has. The same partition function is used on the sample data as the one used on the full data. These two steps ensure the physical

similarity of the data layouts between these two data and leads to the same number of tasks launched in every single stage.

5

Evaluation

In this chapter, we describe and discuss the results of the experiments performed. Various experiments have been performed in order to evaluate how well the proposed solution works. Relative error as defined in Eq. 5.1 and running cost are the main metrics. The simulations are performed using the different numbers of cores to evaluate their effect on the accuracy. We also compare the system with Ernest (11) to show how well the system performs.

$$RelErr = \frac{|measuredTime - realTime|}{realTime} \tag{5.1}$$

5.1 Experiment setup

5.1.1 Applications

Two different big data analytics benchmark applications are chosen for evaluation:

- Standardized citation metrics computation (32) is chosen as the main application for evaluation since it is one of the most representative Spark applications in the ICSR Lab. The application uses the data from the large-scale Scopus database which provides full publication metadata, as well as involves complicated query logics with a vast range of built-in or user-defined functions. All the data are stored as Parquet files instead of plain vanilla text files in order to utilize partitioning used in data sampling.
- TPC-H, which is a SQL benchmark that contains a suite of business-oriented ad-hoc queries (33). We run TPC-H with a scale factor of 10. All 22 queries are combined as a single Spark application for evaluation.

5.1.2 Cloud configurations

Three families in Amazon EC2 are chosen: m5a (general purpose) [1], r5a (memory optimized) [2] and i3en (compute optimized) [3]. For m5a and r5a family, *large* instance size are chosen. For the i3en family, *large*, *xlarge* and *2xlarge* instance sizes are chosen. We limit the maximum number of cores available in the cluster to 24. Table 5.1 shows the number of cores and executors for the chosen instances. There are more available instances for evaluation like compute-optimized instances or larger ones, but they are not chosen due to practical constraints on time and budget. The i3en family is used quite frequently since it offers a low price per GB of SSD instance storage on EC2. The m5a and r5a family are in accord with our cost-saving motivation since they offer *large* instance sizes[4].

Instance	**#executors**											
i3en.large (2 CPU, 16 GB)	1	2	3	4	5	6	7	8	9	10	11	12
i3en.xlarge (4 CPU, 32 GB)		1		2		3		4		5		6
i3en.2xlarge (8 CPU, 64 GB)				1				2				3
r5a.large (2 CPU, 16 GB)	1	2	3	4	5	6	7	8	9	10	11	12
m5a.large (2 CPU, 8 GB)	1	2	3	4	5	6	7	8	9	10	11	12
#cores	2	4	6	8	10	12	14	16	18	20	22	24

Table 5.1: Instance configurations

5.1.3 Runtime

Databricks Runtime 7.3 LTS is used for experiments, which includes Apache Spark 3.0.1, Ubuntu 18.04.5 LTS, Java Zulu 8.48.0.53-CA-linux64 (build 1.8.0_265-b11), Scala 2.12.10 and Python 3.7.5.

5.2 Relative Error

Firstly, the relative errors using different types of instances on the full data are measured. In the experiment, we configure the dry run of the simulator with different numbers of cores, collect the performance metrics in the system, and then predict the performance of cases where the application runs on other numbers of cores as shown in Table 5.1. All

[1] https://aws.amazon.com/ec2/instance-types/m5/
[2] https://aws.amazon.com/ec2/instance-types/r5/
[3] https://aws.amazon.com/ec2/instance-types/i3en/
[4] https://aws.amazon.com/ec2/pricing/on-demand/

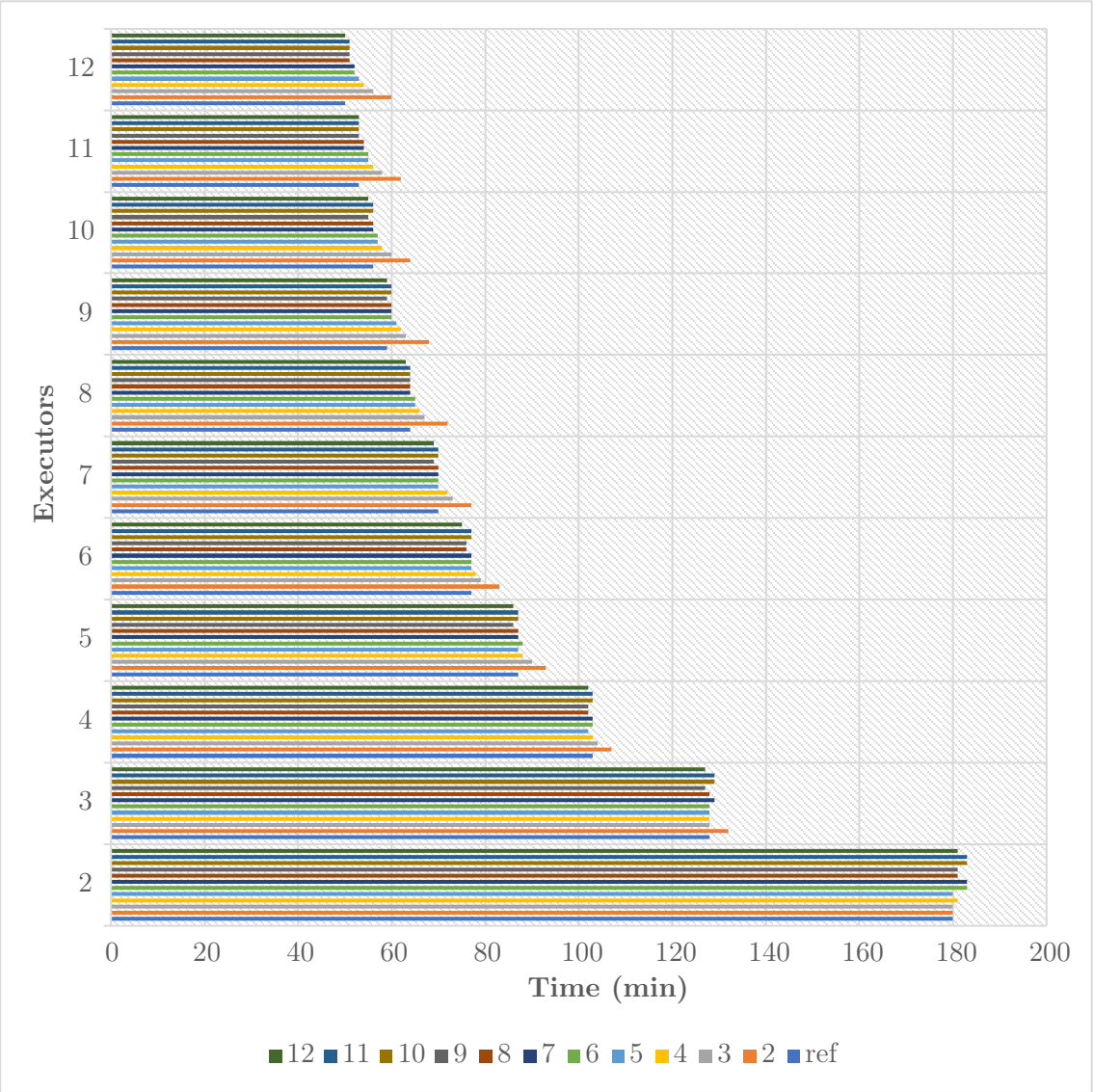

Figure 5.1: Simulation results using different numbers of cores on i3en.large

the simulation results are presented in Figure 5.1, 5.2, 5.3, and Figure 7.1, 7.2 and 7.4 in the Appendix. Simulations are launched using different numbers of instances which are indicated by different colors, where blue lines (which are the bottom ones in each group with a different number of executors in the figures) represent the actual running time of the execution. The system produces similar results on the two different applications. For standardized citation metrics computation, the average relative error of simulation running using i3en.large, i3en.xlarge, i3en.2xlarge, r5a.large and m5a.large is 2.3%, 2.7%,

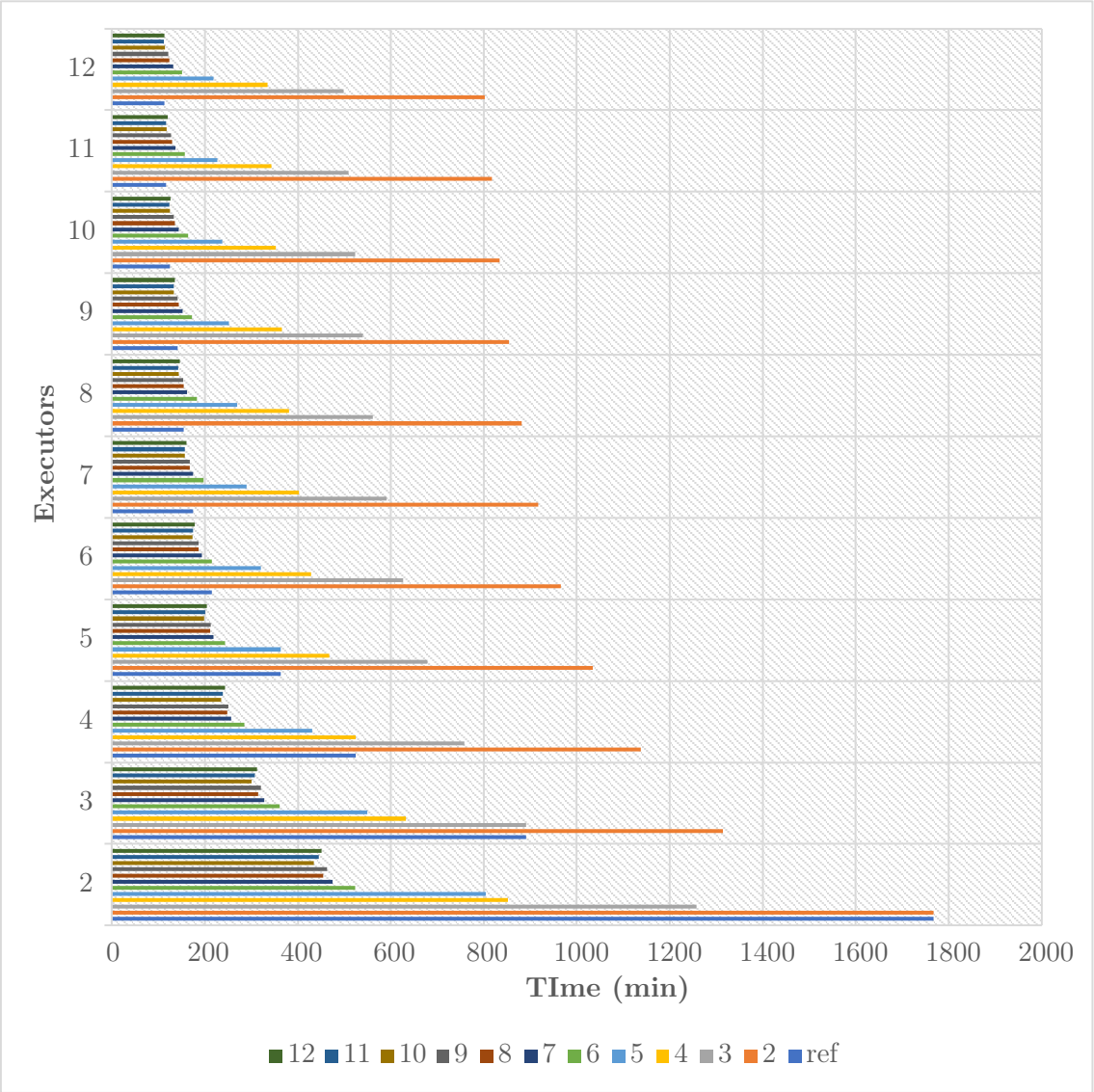

Figure 5.2: Simulation results using different numbers of cores on m5a.large

5.3%, 51.6% and 82.1%, as shown in Table 5.2. In comparison, we run Ernest under the same settings with 10 iterations, which results in average relative errors of 16.2%, 14.2%, 17.2%, 62.6%, and 87.1% on each instance. Predictions on i3en families showed much better performance, which has two possible reasons - memory size and EBS. m5a.large only has 8 GB memory on each node, which leads to frequent disk spilling of Spark. However, the effect of disk spilling during Spark transformations is not captured in the model. If the simulation is only launched using two executors, the metrics captured in

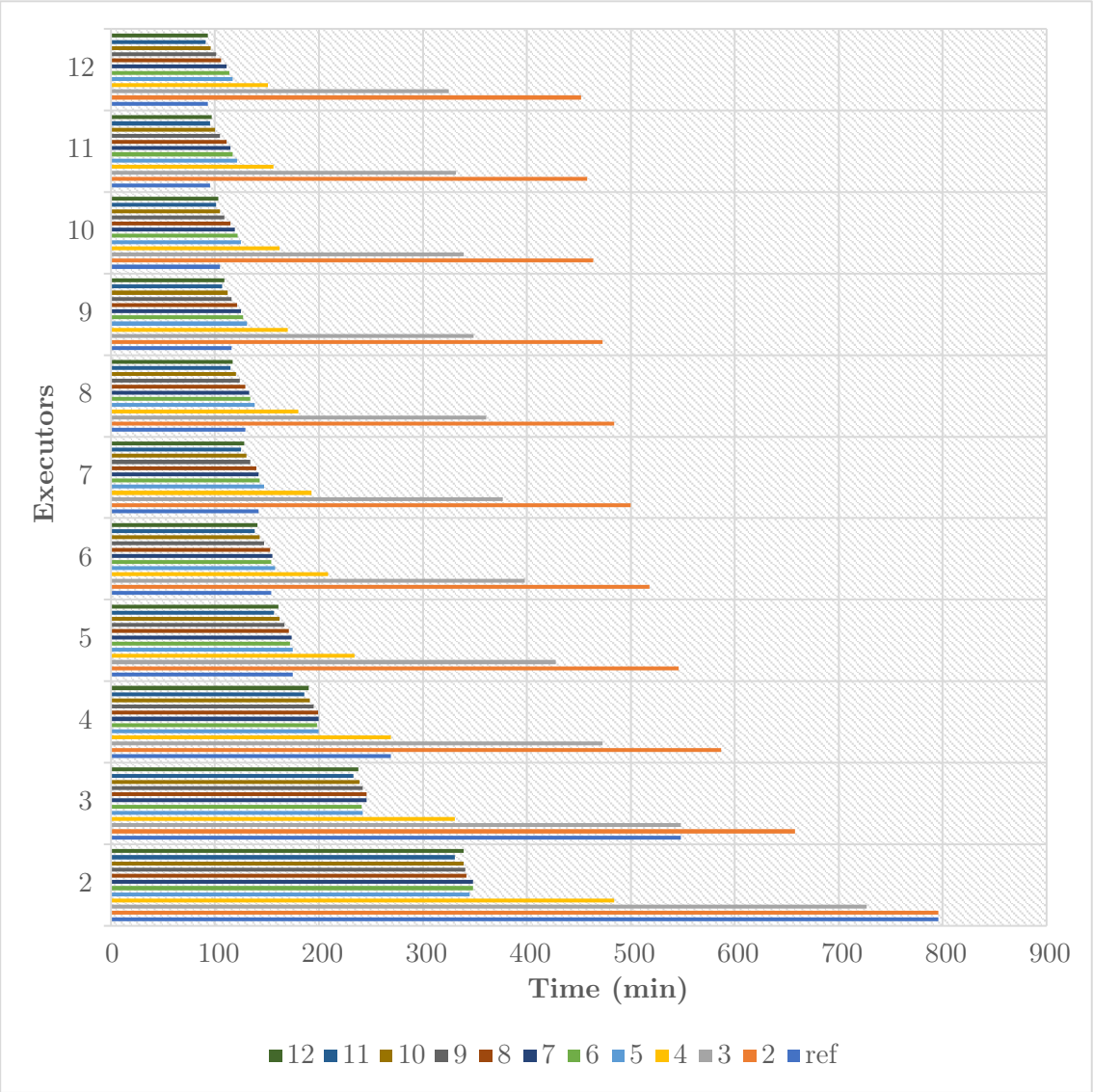

Figure 5.3: Simulation results using standardized citation metrics computation on r5a.large

the model are affected by disk spilling, which leads to the magnified estimated running time on more executors. Moreover, both r5a.large and m5a.large are EBS-only, which is network-attached storage and the slightly worse performance further deteriorates the estimation accuracies of the simulation. In contrast, the chosen i3en families have big NVMe SSD instance storage, which means the low-latency disks are physically attached to the servers so there is no network latency as well. However, we can see that the prediction errors of simulation on r5a.large and m5.large become smaller and stay consistent with an

#executors	i3en.large	i3en.xlarge	i3en.2xlarge	r5a.large	m5a.large
1	/	9.72%	6.95%	/	/
2	10.07%	1.26%	5.54%	230.16%	352.39%
3	4.67%	0.45%	3.31%	150.50%	193.73%
4	2.85%	0.90%	/	41.01%	110.59%
5	1.59%	2.30%	/	21.49%	59.64%
6	1.42%	1.44%	/	19.67%	32.80%
7	0.91%	/	/	18.44%	27.17%
8	0.76%	/	/	17.02%	25.65%
9	0.91%	/	/	16.55%	24.71%
10	0.56%	/	/	16.79%	25.97%
11	0.56%	/	/	18.54%	25.71%
12	0.98%	/	/	17.18%	25.03%
Average	2.30%	2.68%	5.27%	51.58%	82.13%

Table 5.2: Relative errors of simulation (results of simulations using only one executor on i3en.large, r5a.large and m5a.large are missing since they timed out)

acceptable error rate as the number of executors used for simulation becomes bigger. We can also find that for the i3en families, the simulation can achieve a satisfactory result using either a small number or a large number of executors.

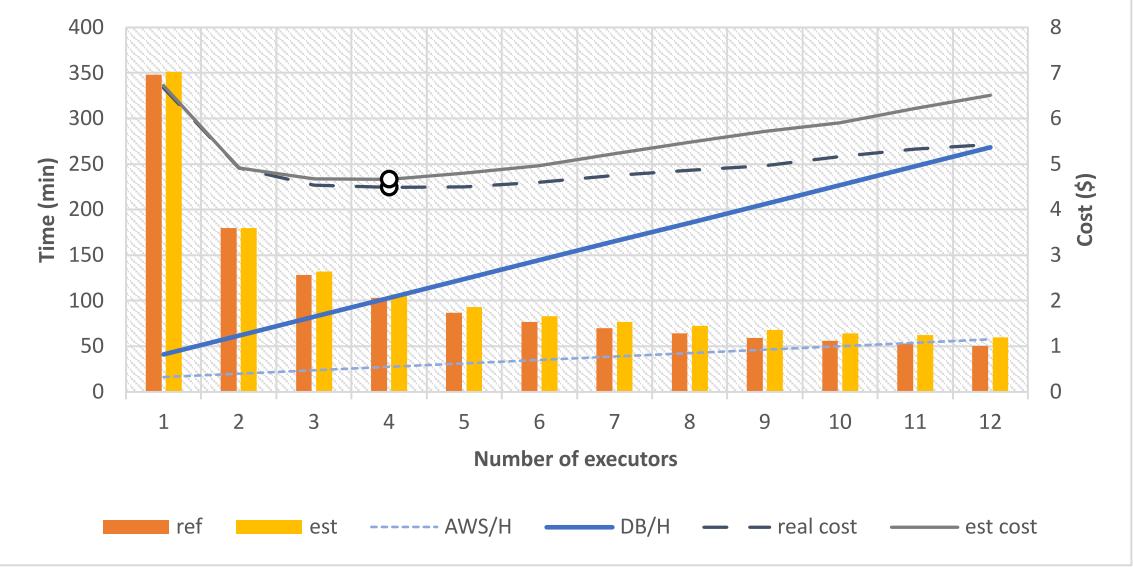

Figure 5.4: Estimated cost curve line on i3en.large using standardized citation metrics computation application

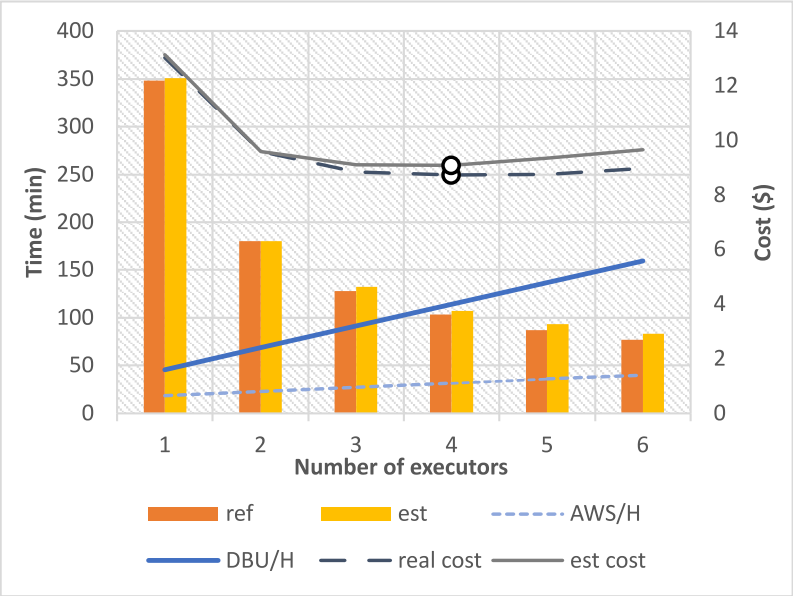

Figure 5.5: Estimated cost curve line on i3en.xlarge using standardized citation metrics computation application

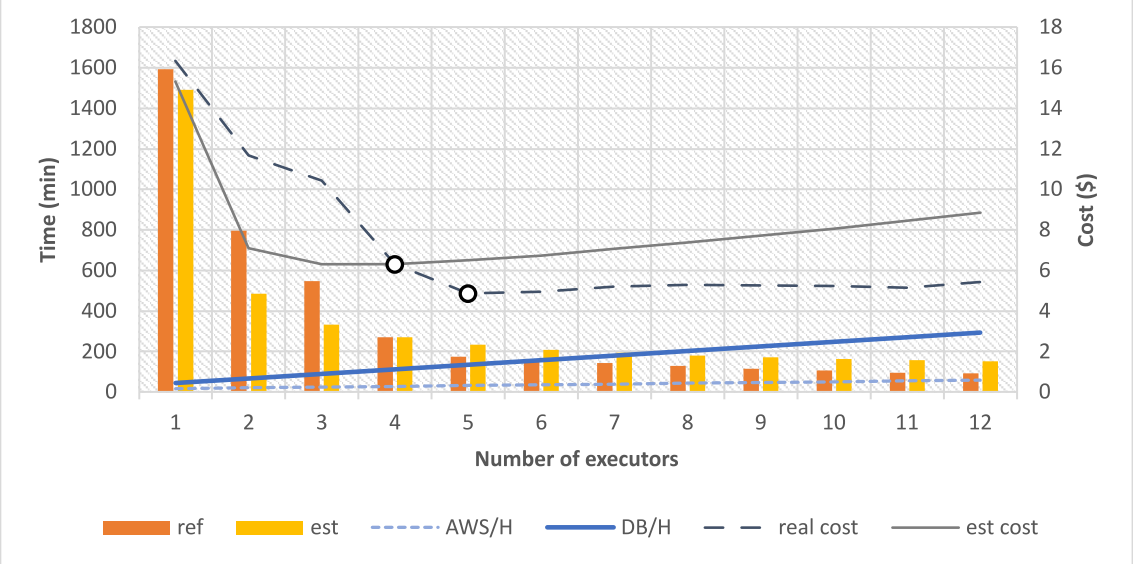

Figure 5.6: Estimated curve line on r5a.large using standardized citation metrics computation application

5.3 Choose Optimal Number of Instances

We try to identify the optimal number of instances using the estimated time. For the i3en family, the results of the simulation with 4 available cores are shown here as the

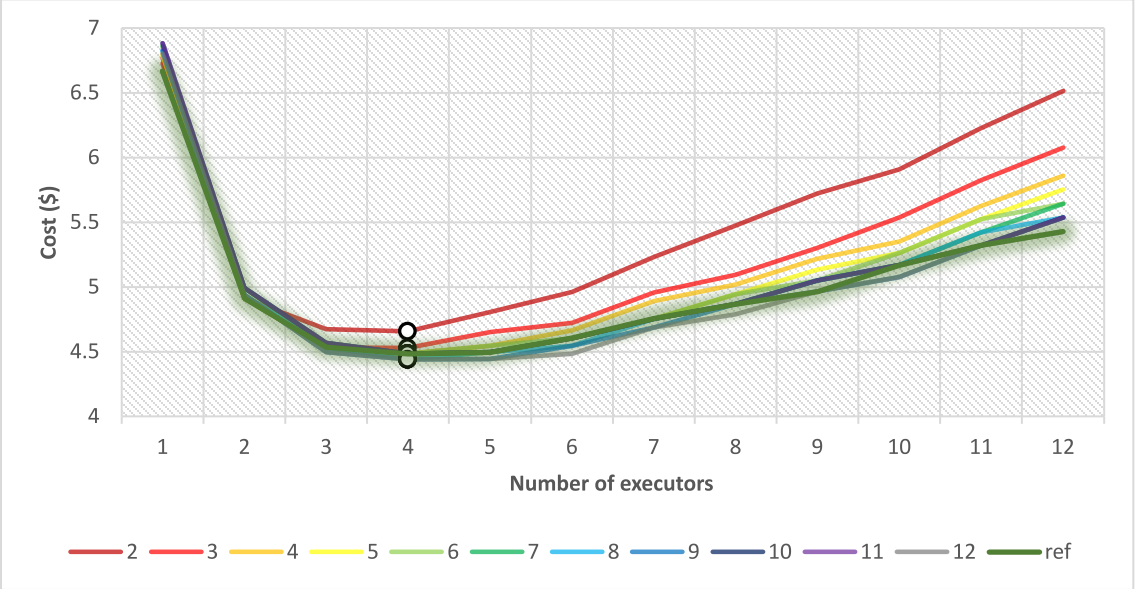

Figure 5.7: Estimated cost curves on i3en.large using standardized citation metrics computation application on full data

representative one in Figure 5.4. For r5a, the results of the simulation with 8 available cores is used, which is shown in Figure 5.6. Figure 5.7 shows the full result. As shown in the figures, the simulation produces an accurate estimation of running time, which generates a similar cost curve line. The optimal number of instances can be chosen by finding the minimum of the curve. Due to the distortion of the estimated running time curve on r5a.large and m5a.large, the simulation generates cost curves with a different shape and minimum point, but the suggested numbers of executors are still close to the real optimal number.

We also run the system on the sampled data to determine the optimal number of instances. Different numbers of cores are used for the simulation to observe the estimation differences. As shown in Figure 5.8, most of the recommended parallelisms for the application (which are shown as the white points in the plot) are around 4 to 5, which are very close to the ideal one (which is 4 as shown in Figure 5.4 and Figure 5.7). We also compared the results with one using random sampling, as shown in Figure 5.9. The simulations are performed 10 times on different randomly sampled data with different random number seeds. As can be seen in the figure, the recommended parallelisms are more scattered and fail to produce robust results that are close to the optimal number.

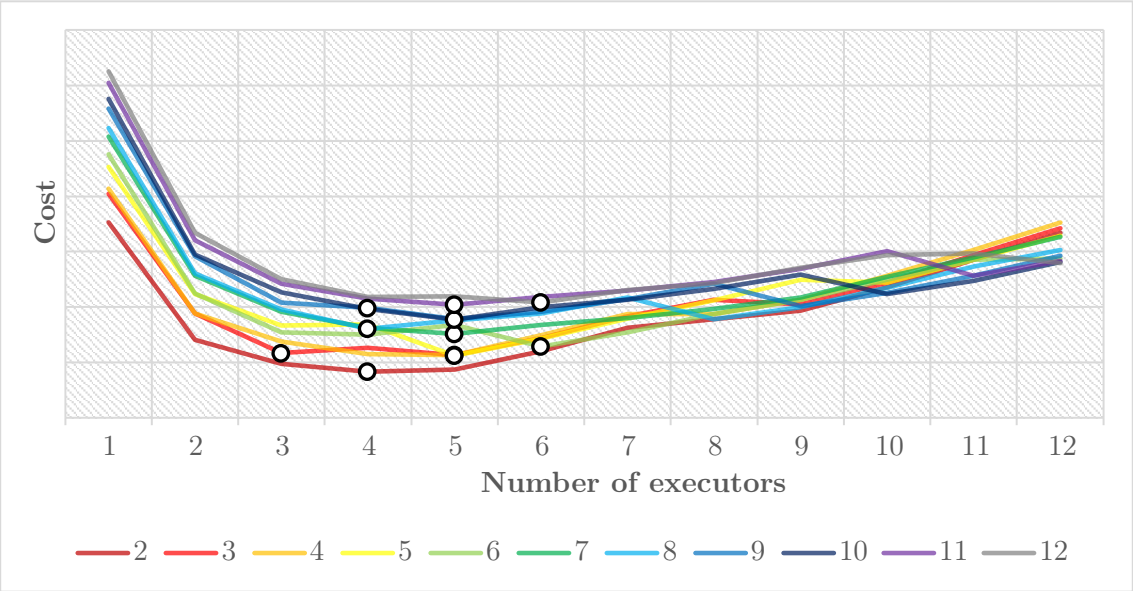

Figure 5.8: Estimated cost curve on i3en.large using standardized citation metrics computation application on stratified sampled data

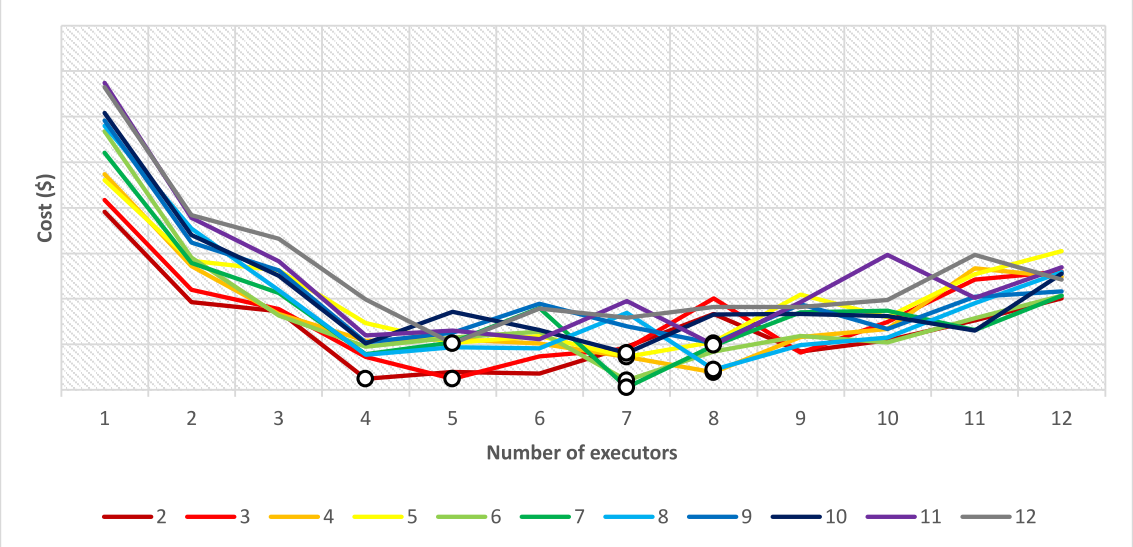

Figure 5.9: Estimated cost curve on i3en.large using standardized citation metrics computation application on randomly sampled data

5.4 Cost

The cost of the system consists of two types of the cost: the cost of running the application for obtaining the performance metrics and the cost of running simulations. The latter cost of running simulations is almost negligible. Although some CPUs have less powerful computing capabilities than others, generally the simulation can be finished in less than 5 minutes using any type of cluster chosen in the experiment. The former type of cost of running the application for obtaining the performance metrics depends on the application itself and the chosen instance for running the system. The actual cost can be found in Figure 5.4, 5.5, 5.6 and 5.7 with different cost curves. If the system runs on the sampled data, the running time of the application can be decreased 5-10x less than running on the full data, which further decreases the cost. In comparison, Ernest typically needs to run the application at least 10 times on different-sized sampled data generated by the optimal experiment designs.

6

Discussion

In this chapter, we discuss the benefits and limitations of the proposed solution of choosing instances.

6.1 Benefits

According to the experiment results in Chapter 5, we can see in the case of non-recurring lightweight Spark applications, the proposed Spark performance model can have a good and robust estimation of the running time of the application given a different number of resources, and thus accurate recommendations for the optimal number of instances. If the end goal is only about choosing the optimal number of instances, then the user can only run the system on the sampled data to further reduce the cost. Because of the low cost of the system, users can launch the application with the system with different types of instances and find out the most suitable cluster for the application. Although the resulting estimated time and cost cannot be used to estimate the final cost of the application on the full data, it can be used to compare across different types of instances. The proposed way of stratified data sampling also helps to produce robust results which are close to the optimal number, in comparison the random sampling results in unstable curve lines, and the estimated optimal numbers of executors are scattered around.

6.2 Limitations and Future Work

The system is designed for estimating the running time for a smaller scale workload, which does not capture the negative effect brought by disk spilling of Spark. If the application needs to process a huge amount of data with frequent disk spilling, the system overestimates

the running time of the application given more resources, which is also reflected in the results of the experiments performed on r5a.large and m5a.large instances. In the future, we plan to have further research on how to incorporate disk spilling behaviour into the model.

In the experiments, we only selected five different instance types for the evaluation. However, there are more different EC2 instances with various different hardware configurations. Also, there are bigger instances in the r5a family and m5a family. To further verify the negative impact on disk spilling, we can limit the available cores that can be used for Spark so that Spark can run using the same available resources with *large* instances except for bigger memory to prevent disk spilling.

More and more cloud service providers started to provide the auto-scaling/managed-scaling feature for Spark. Auto-scaling is typically achieved by periodically reporting various statistics on idle executors and the location of intermediate files within the cluster. It would be interesting to evaluate the effect of the integration of the system into the auto-scaling-enabled clusters.

Due to the practical constraints on time and resources, experiments are not performed on larger workloads with more powerful instances. The effect of the new factorized query engine Photon (34) on the system is also not evaluated, mainly because the current engine does not support user-defined functions which are frequently used in the typical workloads in the ICSR Lab. The engine exploits data-level and instruction-level parallelism in CPUs which is not yet captured in the proposed model, and also introduces lots of different Photon specific actions. It would be insightful to see how adaptive the model is when performing on the new engine.

7

Conclusion

To conclude, in this thesis, we purpose a lightweight approach to estimate the optimal parallelism given an Apache Spark application and evaluate its accuracy on two different applications and types of instances. The experiment results demonstrate the high accuracy of the system with low cost compared to other solutions. The answers to our proposed research questions are summarized in the following paragraphs.

Regarding how to estimate the performance of the Spark applications while minimizing the cost in the first research question, we proposed a hierarchical analytical performance model of Spark and implemented a simulator based on the model, instead of black-box-based models which need a lot of training data in order to develop a performance model with decent accuracy. The system, which uses the performance metrics collected in the listeners during the execution of the application, is implemented as a plugin so that users just need to attach it to the Spark application without the need to modify the source code. We performed the experiments on our proposed solution using different applications and types of instances and compared it with Ernest.

Regarding how to make use of sampled data to estimate the optimal parallelism given a Spark application in the second research question, we proposed a domain knowledge-driven stratified sampling-based approach, which can result in representative sampled data for Spark so that the system can still generate performance curves with similar shapes based on the performance metrics collected. The experiments show that our data sampling approach can successfully find out the optimal or sub-optimal parallelism, and can achieve better results compared to the random data sampling approach. It indicates further reduced cost which leave room for users to run the system across several instances in order to pick the most suitable one for the application.

References

[1] Jules Damji, Brooke Wenig, Tathagata Das, and Denny Lee. *Learning spark: lightning-fast big data analysis.* " O'Reilly Media, Inc.", 2020. v, 9

[2] Hossein Hassani, Xu Huang, and Emmanuel Silva. **Digitalisation and Big Data Mining in Banking**. *Big Data and Cognitive Computing*, **2**(3), 2018. 1

[3] John A. Aloysius, Hartmut Hoehle, Soheil Goodarzi, and V. Venkatesh. **Big data initiatives in retail environments: Linking service process perceptions to shopping outcomes**. *Annals of Operations Research*, **270**:25–51, 2018. 1

[4] Sabyasachi Dash, Sushil Kumar Shakyawar, Mohit Sharma, and Sandeep Kaushik. **Big data in healthcare: management, analysis and future prospects**. *Journal of Big Data*, **6**(1):1–25, 2019. 1

[5] Abhishek Singh, Ashish Payal, and Sourabh Bharti. **A walkthrough of the emerging IoT paradigm: Visualizing inside functionalities, key features, and open issues**. *Journal of Network and Computer Applications*, **143**:111–151, 2019. 1

[6] Judy F Burnham. **Scopus database: a review**. *Biomedical digital libraries*, **3**(1):1–8, 2006. 1

[7] Robin Champieux. **PlumX**. *Journal of the Medical Library Association: JMLA*, **103**(1):63, 2015. 1

[8] Abiodun Oguntimilehin and Emmanuel-Ojo Ademola. **A review of big data management, benefits and challenges**. *A Review of Big Data Management, Benefits and Challenges*, **5**(6):1–7, 2014. 1

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing**. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association. 1

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. **Spark SQL: Relational Data Processing in Spark**. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. 8

[11] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. **Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics**. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 363–378, USA, 2016. USENIX Association. 10, 29

[12] Alexandre Maros, Fabricio Murai, Ana Paula Couto da Silva, Jussara M. Almeida, Marco Lattuada, Eugenio Gianniti, Marjan Hosseini, and Danilo Ardagna. **Machine Learning for Performance Prediction of Spark Cloud Applications**. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 99–106, 2019. 10

[13] Nhan Nguyen, Mohammad Maifi Hasan Khan, and Kewen Wang. **Towards Automatic Tuning of Apache Spark Configuration**. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 417–425, 2018. 10, 11

[14] Guoli Cheng, Shi Ying, Bingming Wang, and Yuhang Li. **Efficient Performance Prediction for Apache Spark**. *Journal of Parallel and Distributed Computing*, **149**:40–51, 2021. 10, 11

[15] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. **Improving Spark Application Throughput via Memory Aware Task Co-Location: A Mixture of Experts Approach**. In *Proceedings of the 18th ACM/I-*

FIP/USENIX Middleware Conference, Middleware '17, page 95–108, New York, NY, USA, 2017. Association for Computing Machinery. 10, 11

[16] David Buchaca Prats, Felipe Albuquerque Portella, Carlos H. A. Costa, and Josep Lluis Berral. **You Only Run Once: Spark Auto-Tuning From a Single Run**. *IEEE Transactions on Network and Service Management*, **17**(4):2039–2051, 2020. 10, 11

[17] Zemin Chao, Shengfei Shi, Hong Gao, Jizhou Luo, and Hongzhi Wang. **A gray-box performance model for Apache Spark**. *Future Generation Computer Systems*, **89**:58–67, 2018. 10

[18] Guolu Wang, Jungang Xu, and Ben 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, 2016. 10, 11

[19] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. **CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics**. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association. 10, 11, 13

[20] Jinhan Xin, Kai Hwang, and Zhibino Yu. **LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications**. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*, SIGMOD '22, page 457–468, New York, NY, USA, 2022. Association for Computing Machinery. 10

[21] Kewen Wang and Mohammad Maifi Hasan Khan. **Performance Prediction for Apache Spark Platform**. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 166–173, 2015. 11, 12

[22] Danilo Ardagna, Enrico Barbierato, Athanasia Evangelinou, Eugenio Gianniti, Marco Gribaudo, Túlio B. M. Pinto, Anna Guimarães, Ana Paula Couto da Silva, and Jussara M. Almeida. **Performance Prediction of Cloud-Based Big Data Applications**. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 192–199, New York, NY, USA, 2018. Association for Computing Machinery. 11, 12

[23] Danilo Ardagna, Enrico Barbierato, Eugenio Gianniti, Marco Gribaudo, Túlio B. M. Pinto, A. P. C. da Silva, and Jussara M. Almeida. **Predicting the performance of big data applications on the cloud**. *The Journal of Supercomputing*, pages 1–33, 2020. 11, 12

[24] Soroush Karimian-Aliabadi, Mohammad-Mohsen Aseman-Manzar, Reza Entezari-Maleki, Danilo Ardagna, Bernhard Egger, and Ali Movaghar. **Fixed-point Iteration Approach to Spark Scalable Performance Modeling and Evaluation**. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021. 11, 12, 13

[25] Yuxing Chen, Peter Goetsch, Mohammad A. Hoque, Jiaheng Lu, and Sasu Tarkoma. ***d*-Simplexed: Adaptive Delaunay Triangulation for Performance Modeling and Prediction on Big Data Analytics**. *IEEE Transactions on Big Data*, **8**(2):458–469, 2022. 11, 12

[26] R. Nelson and A.N. Tantawi. **Approximate analysis of fork/join synchronization in parallel queues**. *IEEE Transactions on Computers*, **37**(6):739–743, 1988. 12

[27] A. R. Lebeck, M. Thottethodi, and S. S. Mukherjee. **Exploiting Global Knowledge to Achieve Self-Tuned Congestion Control for k-Ary n-Cube Networks**. *IEEE Transactions on Parallel Distributed Systems*, **1**(03):257–272, mar 2004. 12

[28] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. **Graysort on apache spark by databricks**. *GraySort Competition*, page 65, 2014. 26

[29] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. **On the worst-case complexity of TimSort**. *arXiv preprint arXiv:1805.08612*, 2018. 26

[30] Michael Greenwald and Sanjeev Khanna. **Space-Efficient Online Computation of Quantile Summaries**. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, page 58–66, New York, NY, USA, 2001. Association for Computing Machinery. 27

[31] Karl Pearson. **X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling**. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, **50**(302):157–175, 1900. 27

[32] John PA Ioannidis, Jeroen Baas, Richard Klavans, and Kevin W Boyack. **A standardized citation metrics author database annotated for scientific field**. *PLoS biology*, **17**(8):e3000384, 2019. 29

[33] Meikel Pöss and Chris Floyd. **New TPC Benchmarks for Decision Support and Web Commerce**. *SIGMOD Rec.*, **29**(4):64–71, 2000. 29

[34] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. **Photon: A fast query engine for lakehouse systems**. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2326–2339, 2022. 40

Appendix

7.1 Example Policy Definition and Inital Script for Extra Listeners on Databricks Platform

```
{
  "spark_version": {
    "type": "fixed",
    "value": "7.3.x-scala2.12",
    "hidden": true
  },
  "spark_conf.spark.extraListeners": {
    "type": "fixed",
    "value": "com.elsevier.magpie.NotebookListener",
    "hidden": true
  },
  "aws_attributes.instance_profile_arn": {
    "type": "fixed",
    "value": "arn:aws:iam::123:xxx",
    "hidden": true
  },
  "enable_elastic_disk": {
    "type": "fixed",
    "value": true,
    "hidden": true
  },
  "init_scripts.0.dbfs.destination": {
    "type": "fixed",
    "value": "dbfs:/databricks/scripts/magpie-install.sh",
    "hidden": true
  },
  "cluster_type": {
    "type": "fixed",
    "value": "job",
    "hidden": true
  },
```

```
  "node_type_id": {
    "type": "regex",
    "pattern": "[rmci][3-5][rnade]*.[0-2]{0,1}[x]{0,1}large",
    "defaultValue": "i3en.large"
  }
}
```

Listing 7.1: Example policy file

```
#!/bin/bash
cp $pathToJar /databricks/jars
```

Listing 7.2: Example init script

7.2 More Evaluation Results

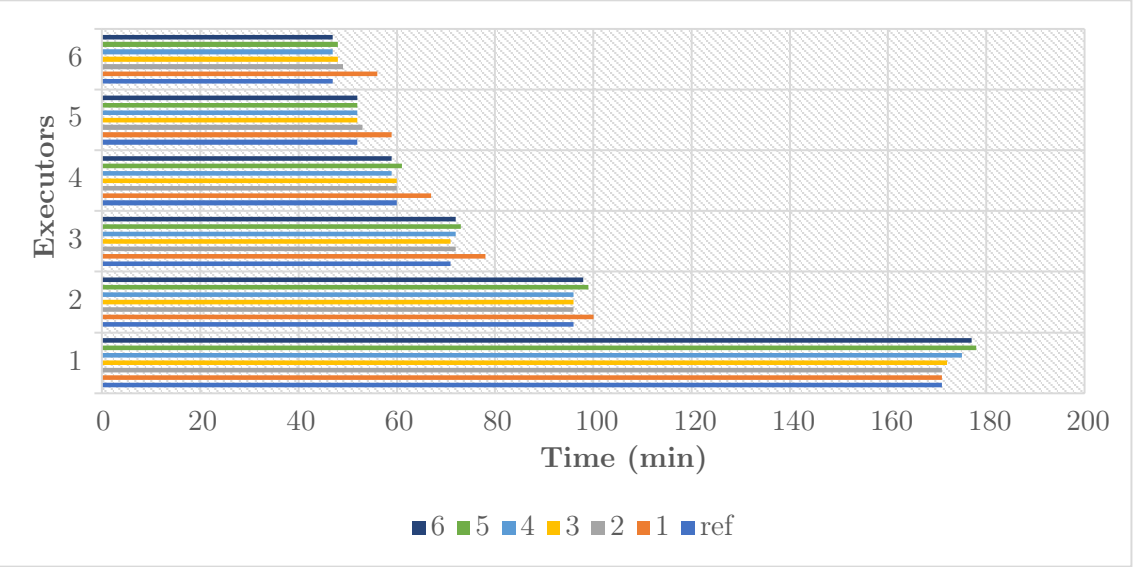

Figure 7.1: Simulation results using standardized citation metrics computation on i3en.xlarge

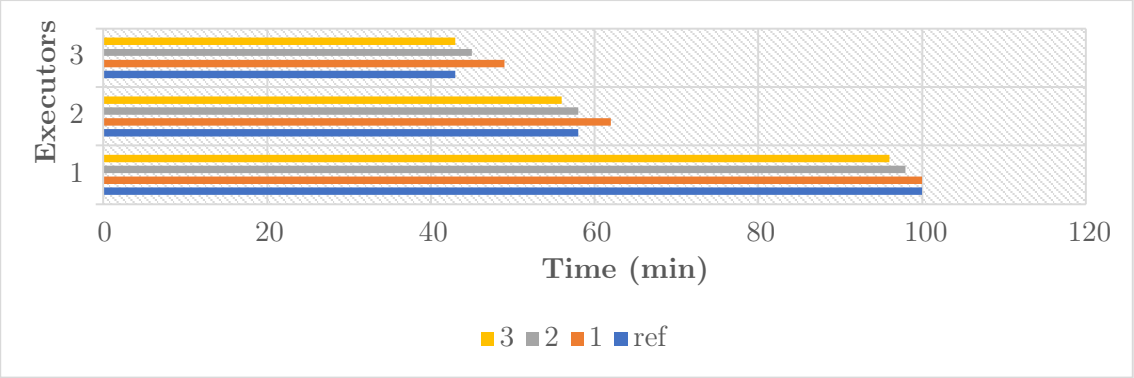

Figure 7.2: Simulation results using standardized citation metrics computation on i3en.2xlarge

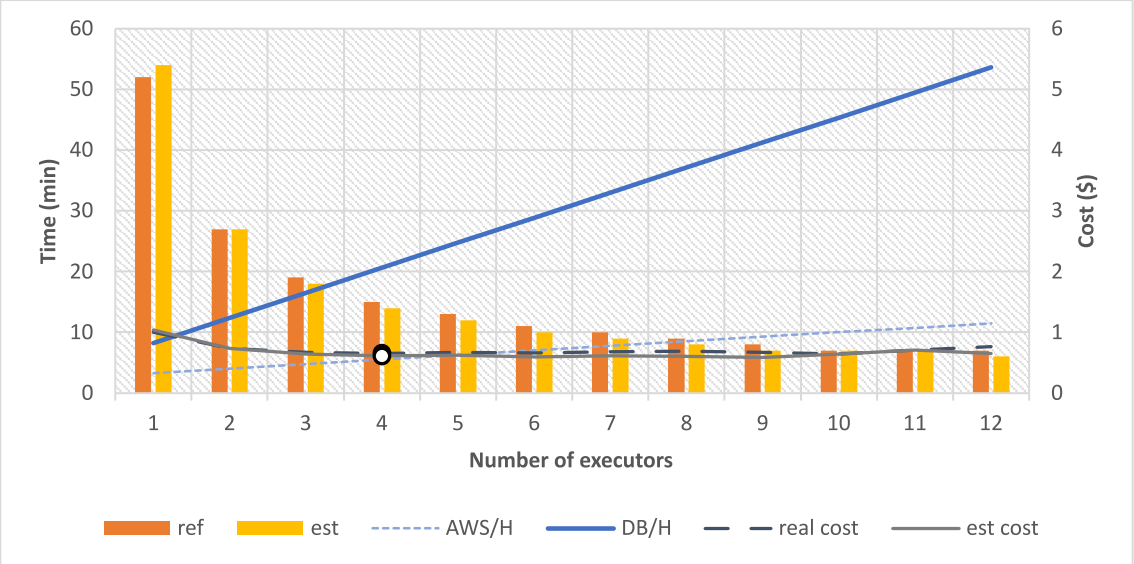

Figure 7.3: Estimated cost curve using TPC-H application on i3en.large

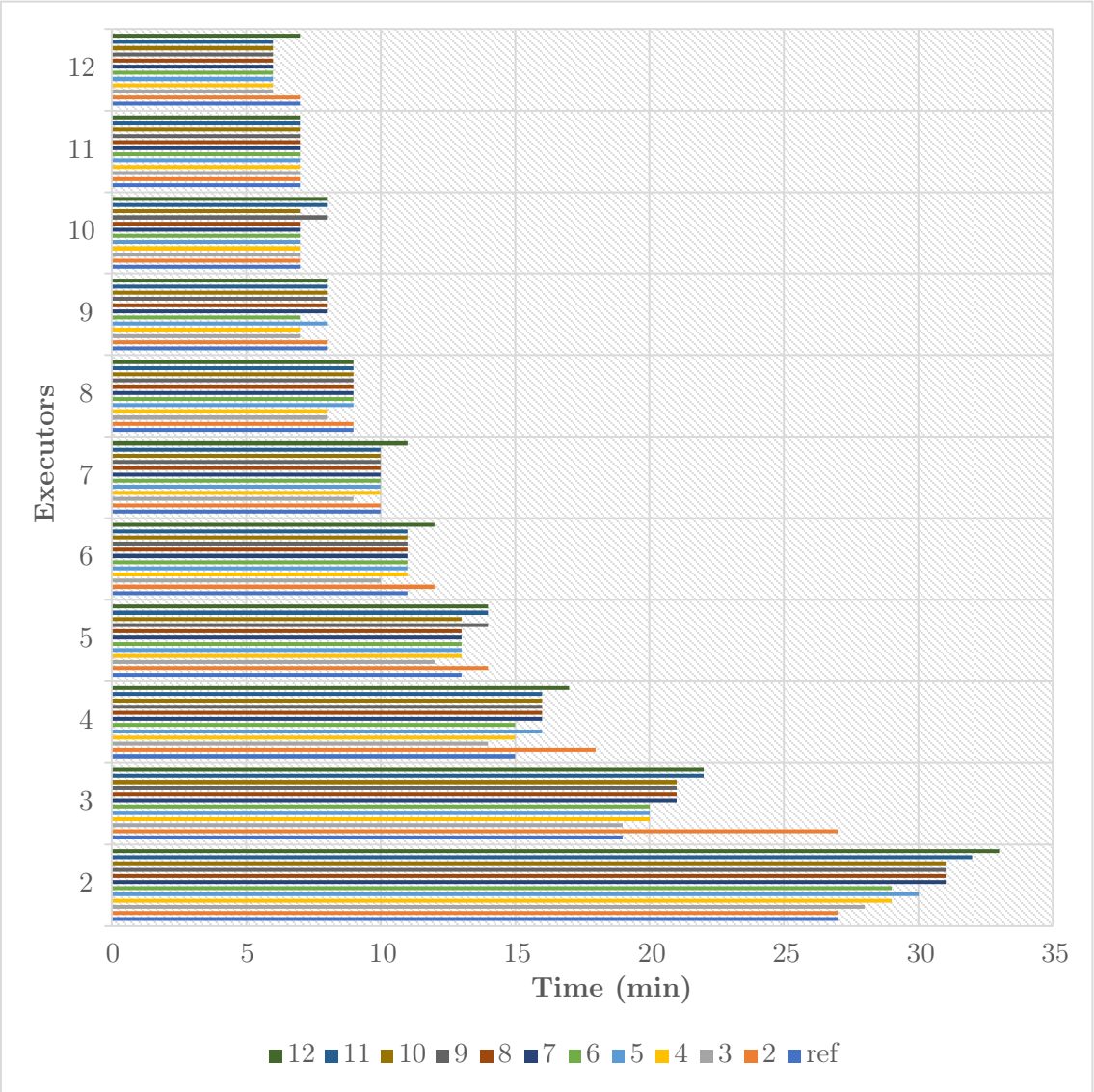

Figure 7.4: Simulation results using TPC-H application on i3en.large