

CHALMERS



Predictive Resource Tuning of Apache Spark Jobs in Kubernetes

A Machine Learning driven approach, with uncertainty aware predictions

Master's thesis in Computer science and engineering

ALEXANDER PETERSSON

LEAH WANJA NDIRANGU

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

MASTER'S THESIS 2026

Predictive Resource Tuning of Apache Spark Jobs in Kubernetes

A Machine Learning driven approach, with uncertainty aware
predictions

ALEXANDER PETERSSON
LEAH WANJA NDIRANGU



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Predictive Resource Tuning of
Apache Spark Jobs in Kubernetes

A Machine Learning driven approach, with uncertainty aware predictions
ALEXANDER PETERSSON
LEAH WANJA NDIRANGU

© ALEXANDER PETERSSON, LEAH WANJA NDIRANGU, 2026.

Supervisor and Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2026

Predictive Resource Tuning of Apache Spark Jobs in Kubernetes

A Machine Learning driven approach, with uncertainty aware predictions

ALEXANDER PETERSSON, LEAH WANJA NDIRANGU

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Apache Spark is a widely used distributed data processing engine for large-scale analytics and machine learning workloads, typically executed in multi-node cluster environments. Kubernetes is increasingly used to deploy and manage these workloads in container orchestration, resource scheduling, and scalable cluster management. Despite this infrastructure, Spark workloads require users to specify CPU and memory allocations in advance. In practice, these resource requests are often overestimated to prevent job failures under uncertain workload requirements. This leads to inefficient resource utilization in shared clusters, reduced overall throughput, and increased operational cost in multi-tenant environments. This thesis addresses the problem of static and overly conservative resource provisioning for Spark jobs running on Kubernetes.

This thesis proposes a machine learning-based resource tuning system that learns from historical execution behavior to predict more accurate resource allocations for defined workload types. In addition, the system incorporates uncertainty awareness to ensure that resource adjustments are applied more conservatively when prediction confidence is low. The proposed approach aims to reduce resource overprovisioning while maintaining execution stability, thereby improving cluster efficiency in distributed Spark environments.

The results demonstrate that the tuning approach reduced memory from 64 GB to 8 GB and cores from 4 to 1 per executor for a designed test job. The total savings was 616 GB of memory and 30 cores per such job after tuning for 40 iterations. This increased resource utilization, although execution time increased by approximately 55%. Uncertainty-aware Wilson intervals were successfully incorporated into memory tuning but not core tuning due to its continuous cost-based optimization objective.

Keywords: Computer science, Masters Thesis, ML, Wilson Interval, Apache Spark, K8s

Acknowledgements

We would like to pass our gratitude to our supervisor and examiner Ahmed Ali-Eldin Hassan for his support and for the mentorship from the planning phase to the completion of this thesis. We also extend our appreciation to our workplace supervisors Devendra Dodiya and Binay Mishra for their guidance, advice, expertise and wisdom which were instrumental in developing our skills and helping us overcome challenges throughout the project. To our manager Pierre Johansson, thank you for providing moral support and encouragement. Your guidance has created a safe environment to express our thoughts, ideas and concerns. Finally, we would like to thank our families for their motivation, patience and support through the thesis work.

Alexander Petersson, Leah Wanja Ndirangu, 2026-10-13

Contents

List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Problem Formulation	1
1.2 Aim	2
1.3 Research Questions	2
1.4 Delimitations	3
1.5 Use of Generative AI	3
1.6 Thesis Outline	4
2 Background	5
2.1 Kubernetes	5
2.2 Apache Spark	6
2.2.1 Spark Architecture	6
2.2.2 Spark Lazy Evaluation	8
2.2.3 Catalyst Optimizer	9
2.2.4 Spark Query Plans	9
2.2.5 Spark Event Logs	10
2.2.6 Spark Dynamic Resource Allocation	10
2.3 Spark on Kubernetes	11
2.4 Baseline: Fixed Class Resource Allocation	12
2.4.1 Baseline System Overview	12
2.4.2 Resource Configuration Model	12
2.4.3 Limitations of the Baseline Approach	13
2.5 Relevant Machine Learning Concepts	14
2.5.1 Tree-based Machine Learning Models	14
2.5.2 Gradient Boosting Classifier	14
2.5.3 Clustering Algorithms	15
2.5.4 DBSCAN Clustering Algorithm	16
2.5.5 Dimensionality Reduction Techniques	16
2.6 Uncertainty and Wilson Confidence Intervals	17
3 Related Work	19
3.1 Predictive Resource Allocation	19

3.2	Memory Optimization	20
3.2.1	Machine Learning Methods	20
3.2.2	Tuning Apache Spark Jobs	20
3.2.3	Improving Execution in Kubernetes	20
3.2.4	Queries as ML Features in Apache Spark.	21
3.3	Gaps in Related Work	21
3.3.1	Predictive Resource Tuning for Spark on K8s	21
3.3.2	Uncertainty-Aware Resource Correction Policies	21
4	Methods	23
4.1	System Overview	23
4.2	Offline Footprint Model Construction	24
4.2.1	Training Data Collection	24
4.2.2	Feature Engineering	25
4.2.3	Feature Selection	25
4.2.4	ML Model Selection	27
4.2.5	Footprint Class Generation using Clustering	28
4.3	Online Profiling and Footprint Classification	28
4.4	Uncertainty-Aware Resource Tuning	29
4.4.1	Memory tuning	29
4.4.2	Cores tuning	31
4.5	Execution Outcome Modeling	32
5	Evaluation	33
5.1	Clustering Analysis	34
5.2	Footprint Model performance	35
5.3	Main Experiments	37
5.3.1	Experiment 1: Workload classification	37
5.3.2	Experiment 2: Iterative Resource Tuning	38
5.3.3	Experiment 3: Simulated resource tuning of jobs of different footprints	40
5.3.4	Experiment 4: Tuning up due to OOM failures	46
5.3.5	Experiment 5: Uncertainty aware tuning	47
6	Discussion	49
6.1	Answering the Research Questions	49
6.2	Method Limitations	50
6.2.1	Query Plan Feature Extraction Limitations	50
6.2.2	Performance Limitations of Interactive Spark Sessions	50
6.3	Design Trade-offs	51
6.3.1	Feature Trade-offs	51
6.3.2	Tuning Algorithm Considerations	51
6.3.3	Model Drift	52
6.3.4	Failure Handling	52
7	Conclusion	53
7.1	Conclusion	53

7.2	Future Work	53
	Bibliography	55
A	Test job code listings	I
A.1	test_job_heavy.py	I
A.2	test_job_oom.py	II

List of Figures

2.1	Image showing the relationship between a cluster manager (e.g. K8s), driver and executors. Taken from Apache Spark documentation [13]. .	7
2.2	Difference in execution between narrow and wide transformations. Image taken from [18]	8
2.3	Spark on Kubernetes architecture. Image taken from [26].	11
2.4	Code listing showing the current configurations users can run Spark with.	13
2.5	Illustration of the sequential training process in gradient boosting. Image taken from [29].	15
4.1	Overview of the proposed resource tuner architecture.	23
4.2	Instrumentation of Spark jobs for feature extraction.	29
5.1	Cluster Distribution K=17	34
5.2	Graph of tuned memory and cores of <code>test_job_heavy.py</code>	39
5.3	Graph of simulated memory, footprint 0, 50 iterations	41
5.4	Graph of simulated memory, footprint 1, 50 iterations	41
5.5	Graph of simulated memory, footprint 2, 50 iterations	42
5.6	Excerpt from <code>state/fp_7.json</code> showing the accumulated statistics for the 8 GB and 16 GB memory levels.	46
5.7	Last three entries from <code>history/fp_7.json</code> , showing two OOM failures at 8 GB followed by an execution at 16 GB.	47
5.8	State of footprint 7 after discarding 8 GB.	48
A.1	Test Job 1 Query plan	IV
A.2	Test Job 2 Query plan	IV
A.3	Test Job 3 Query Plan	V
A.4	Test Job 4 Query plan	V

List of Tables

1.1	Thesis outline and chapter contents.	4
5.1	Top 5 Feature Loadings for the First Three Principal Components . .	35
5.2	10-Fold ($k = 17$ clusters, $\epsilon = 0.8$, $\text{min_samples} = 10$)	35
5.3	Cross-validation mean metrics	36
5.4	Out-of-fold classification report. Class 16 was noted to have had only one instance.	36
5.5	Mapping of jobs to their footprint IDs	38
5.6	Average execution times for the memory+cores tunings experiment 2 observed.	39
5.7	Execution times for footprint 0 at different core counts and memory configurations.	40
5.8	Execution times for footprint 1 at different core counts and memory configurations.	40
5.9	Execution times for footprint 2 at different core counts and memory configurations.	40
5.10	Core tuner decisions and cumulative average cores-cost, footprint 0 .	43
5.11	Core tuner decisions and cumulative average cores-cost, footprint 1 .	44
5.12	Core tuner decisions and cumulative average cores-cost, footprint 2 .	45

1

Introduction

The exponential growth of data usage has led to an increase in the demand of scalable and high-performance data processing frameworks across many industries. As organizations increasingly adopt data-driven approaches into their operations, the need for efficient processing of large-scale datasets has become more prevalent [1]. Apache Spark has emerged as a leading data processing engine due to its in-memory computation model, rich ecosystem of libraries, and integration with major cloud providers. Spark is commonly deployed as a shared, multi-tenant system when using resource managers such as Kubernetes or YARN, where many users submit jobs concurrently and compete for limited cluster resources [2]. This creates challenges for efficient resource allocation, as workloads compete for limited cluster capacity in shared environments.

1.1 Problem Formulation

Users often lack precise knowledge of the CPU and memory resources their applications require [3]. Systems such as Kubernetes, Apache Spark, and SLURM require users to specify resource requests or job configurations prior to execution, even though the true resource requirements of the workload are typically unknown beforehand. This uncertainty frequently leads to human mistakes like conservative over-estimation, and sometimes but less frequently naive under-estimation of resources. Over-estimation is often caused by trying to avoid performance degradation and job failures, while under-estimations often are a result of attempting to save cluster resources and meet Service Level Agreements (SLA) with clients. Incorrect resource provisioning can lead to either over-provisioning, leaving parts of the cluster underutilized, or under-provisioning, resulting in application failures and costly task restarts. Both outcomes reduce overall scheduling efficiency. Applications may be forced to wait in queues because the requested resources appear unavailable, even though they are not fully utilized in practice [4] [5].

This has led to the introduction of predictive resource allocation approaches. However, many of these approaches, such as Paragon [6], Quasar [7], and HPA+ [8], suffer from a key limitation, which is: allocation decisions are commonly based on point estimates and do not explicitly quantify predictive uncertainty through methods such as confidence intervals or probabilistic distributions. To compensate for this uncertainty, systems and users often rely on static safety margins or manually

padded resource requests that are applied independent of the workload characteristics of the specific application. Although these conservative safety buffers may reduce the likelihood of resource contention and job failures in some cases, they can also lead to unnecessary over-provisioning, inefficient cluster utilization, or insufficient resource allocation when the workload behavior deviates from what was expected.

A third problem addressed in this thesis, specifically observed in the studied environment but potentially more general, is that users are currently required to specify resource requests using a predefined set of job or Spark session size classes, similar to those reported in large-scale production environments such as ByteDance [3]. These classes are defined independently of the actual resource requirements of individual applications, and there is no mechanism for validating whether the selected class appropriately reflects the workloads runtime behavior (see Background section 2.4 for class definitions). This can lead to inefficient resource utilization, as the memory and CPU requirements of different workloads may vary by orders of magnitude, while the available resource classes remain coarse-grained and fixed.

1.2 Aim

The aim of this thesis is to design and implement a machine learning-driven resource tuner that corrects resource requests for Apache Spark jobs running on multi-node Kubernetes clusters. The proposed tuner includes an uncertainty-aware component that estimates confidence bounds for expected resource usage, with the goal of improving robustness in resource allocation under workload variability.

1.3 Research Questions

To formulate the contributions of this thesis into a single problem statement, this masters thesis investigates how predictive, uncertainty-aware scheduling can safely improve resource allocation for multi-node Spark jobs on Kubernetes. Specifically, it aims to answer the following two research questions:

"How effective is a machine learning-based resource tuner in safely identifying and correcting resource requests for incorrectly provisioned Apache Spark jobs in in multi-node Kubernetes clusters?"

And:

"How can uncertainty awareness be incorporated into resource tuning in this context?"

1.4 Delimitations

This thesis focuses on the design and evaluation of a machine learning-based resource prediction and correction system for Apache Spark workloads running on a Kubernetes cluster specifically. The implementation is developed as a prototype and is not intended to meet production-grade requirements in terms of reliability, security, or large-scale operational robustness.

The scope is limited to Spark workloads. Extending the approach to a universal model applicable to all types of Kubernetes workloads is outside the scope of this work, as different application types exhibit fundamentally different resource usage patterns and would require substantially more heterogeneous training data and feature extraction work.

Furthermore, the thesis does not modify or redesign Spark's internal scheduling or dynamic resource allocation mechanisms. Existing Spark scheduling behavior is treated as a fixed component of the system, and any modifications to its core scheduling logic are left for future work.

Finally, advanced cluster-level scheduling concerns such as load balancing across nodes, and global fairness optimization are not explicitly addressed in this work, as the focus is on improving resource request estimation rather than redesigning cluster scheduling policies. While execution traces may contain signals reflecting such behavior, these are only used indirectly as descriptive features of historical workload behavior.

1.5 Use of Generative AI

With the recent genesis of generative Artificial Intelligence, it has been deemed necessary to disclose how it has been used in this project, and what policies have been followed to ensure ethical use of AI tools.

It has been widely used in the project to generate ideas, speed up programming and writing, and to summarize and extract key information from sources when necessary. ChatGPT has been used as an all-purpose idea generation, content parsing, and report writing tool. Github Copilot was primarily used to speed up code implementation and debugging incorporating different agents such as Claude (Haiku 4.5, Sonnet 4.5, Opus 4.5 & 4.6), GPT (5, 5-mini, 5.2, 5.2 & 5.3-Codex, 5.4), and Consensus AI was used to find real sources to support any scientific claims made in the thesis.

With that being said, we, the authors of this paper, take full responsibility for the validity and correctness of every claim and idea in this paper. We also take full responsibility to make sure our working methods have been strictly ethical in the context of sensitive information handling, plagiarism, and any other applicable area.

1.6 Thesis Outline

This section describes the outline of this thesis.

Chapter	Content
Chapter 1: Introduction	Presents the scope, motivation, and objectives of the thesis.
Chapter 2: Background	Provides the necessary theoretical background and foundational knowledge required to understand the methodology and findings of the thesis.
Chapter 3: Related Work	Present any relevant work done before on the subject to anchor the thesis context
Chapter 4: Method	Describes the system design of the proposed resource tuner, the implementation considerations, and the theoretical reasoning behind the design choices.
Chapter 5: Results	Presents the effects of the proposed resource tuner in a cluster environment, with a focus on how CPU and memory utilization are affected by the system.
Chapter 6: Discussion	Discusses and evaluates the results in relation to the research questions, presents limitations.
Chapter 7: Conclusion	Thesis summary and ideas on future work

Table 1.1: Thesis outline and chapter contents.

2

Background

2.1 Kubernetes

Kubernetes is a container orchestration platform used for deploying, managing, and scaling applications in distributed computing environments such as data clusters. At a high level, Kubernetes manages applications through a cluster-based architecture consisting of a control plane and a set of worker nodes. Worker nodes host application workloads in the form of containers grouped into pods, which represent the smallest deployable unit in Kubernetes. The control plane is responsible for making global scheduling decisions, maintaining cluster state, and ensuring that the desired number of application replicas are running at any given time [9].

Applications deployed on a Kubernetes cluster specify their computational resource requirements through CPU and memory requests and limits. The Kubernetes scheduler assigns pods to nodes based on these resource requirements, node availability, placement constraints, and scheduling policies. This process ensures that workloads are placed on suitable nodes while respecting the declared resource constraints of each application [10]. Resource requests represent the amount of CPU and memory that Kubernetes reserves for scheduling decisions, while limits define the maximum resources a container is allowed to consume [11].

Kubernetes also provides runtime scaling mechanisms, such as Horizontal Pod Autoscaling (HPA) and Vertical Pod Autoscaling (VPA), which can modify resource allocation based on observed workload behavior. HPA adjusts the number of pod replicas based on metrics such as CPU or memory utilization, while VPA adjusts pod resource requests and limits. However, these mechanisms operate after workloads have been deployed and are not designed to replace accurate initial resource configuration. In particular, resource changes that require pod replacement can be disruptive for long-running or stateful workloads, where restarting containers may result in loss of temporary application state [12]. Therefore, initial resource requests remain an important factor in Kubernetes scheduling and cluster efficiency.

In summary, Kubernetes provides a flexible and scalable foundation for containerized workload execution, but its reliance on static resource specifications introduces challenges in achieving optimal resource utilization, particularly in environments with heterogeneous and unpredictable workloads.

2.2 Apache Spark

Apache Spark, commonly known as Spark, is a data processing engine that supports multiple programming languages for executing data engineering, data science and machine learning applications on single-node machines or data clusters [13]. Spark performs in-memory computations to achieve high-speed data processing on large-scale data. This leads to a demand in expensive hardware, scalability limits, and challenges for data intensive applications. A Spark data cluster is deployed as a multi-tenant system, where users or data pipelines submit jobs or run interactive sessions concurrently [13]. They rely on cluster resource scheduling mechanisms to manage computing resources and deliver acceptable response times to meet Service Level Agreements. The fair scheduler is implemented through cluster managers such as K8s, YARN or Apache Mesos to control the job flow and resource allocation before reaching Spark’s internal scheduler [14].

2.2.1 Spark Architecture

Apache Spark employs a master-worker architecture composed of several core components and abstractions that enable high-performance distributed data processing, fault tolerance, and scalable execution.

A user submits a job to Spark, after which the master node initializes a driver process, while worker nodes launch multiple executors to execute tasks. The application interacts with the driver through a Spark session. The driver is responsible for planning and coordinating execution by transforming the user program into a Directed Acyclic Graph (DAG) of stages and tasks, which are then scheduled across the cluster. Data is partitioned and distributed across executors, where each partition is processed independently. Executors execute tasks in parallel using available CPU cores (slots), enabling concurrent task execution. Failed tasks are rescheduled by the driver to available execution slots. The driver also requests resources with the cluster manager and continuously monitors task execution. Executors are responsible for loading partitioned data, performing computations, and persisting intermediate or final results either in memory or on disk. They also send execution status and results back to the driver. Both the driver and executors run as processes on a Java Virtual Machine (JVM) [13], [15]. Figure 2.1 below illustrates this description.

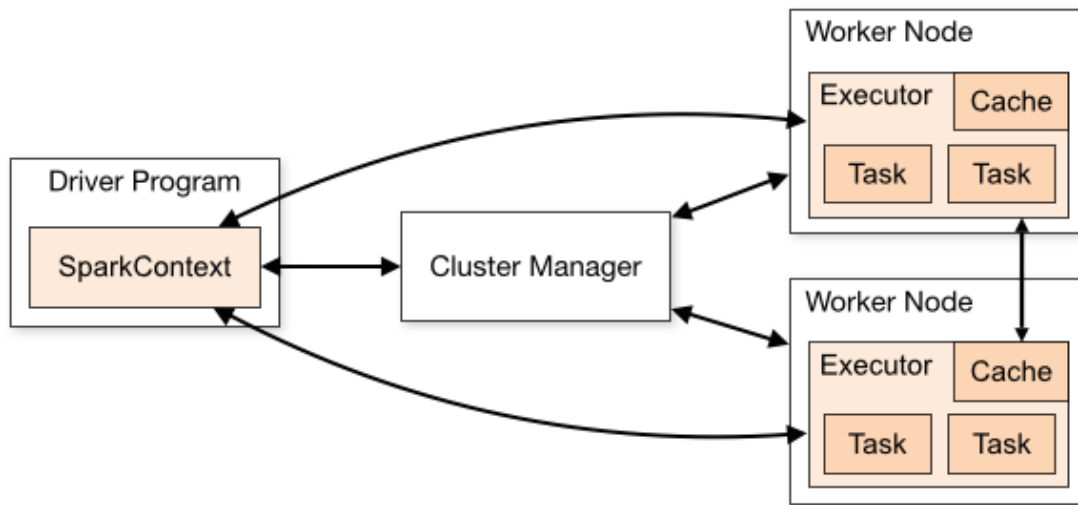


Figure 2.1: Image showing the relationship between a cluster manager (e.g. K8s), driver and executors. Taken from Apache Spark documentation [13].

Apache Spark employs three primary abstractions for handling data at scale: Resilient Distributed Datasets (RDDs), Dataframes and Datasets.

- **Resilient Distributed Datasets (RDDs):** RDDs are the core distributed data structure in Apache Spark. They are immutable and fault-tolerant, and are designed to support distributed in-memory computation across clusters. RDDs were introduced to address limitations in existing distributed computing frameworks when handling iterative algorithms and interactive data processing workloads. They provide a programming interface based on coarse-grained transformations, making them suitable for parallel processing where operations are applied across entire datasets rather than individual records [16].
- **DataFrames:** A DataFrame is an immutable distributed collection of data organized into named columns, similar to relational database tables or Pandas DataFrames. DataFrames are built on top of Spark's execution engine and enable query optimizations through the Catalyst optimizer and the Tungsten execution engine. Compared to RDDs, DataFrames provide a higher-level abstraction that enables more expressive APIs for structured and semi-structured data processing. This abstraction allows the system to automatically optimise query execution and improve performance for complex workloads [15]. The studied environment uses DataFrames for SQL-like queries over structured and semi-structured data.
- **Datasets:** Datasets combine the advantages of RDDs and DataFrames by supporting functional transformations together with query optimizations. They provide type safety in Scala and Java, making them suitable for complex data processing workflows where compile-time safety is beneficial. Like DataFrames, Datasets are optimised by the Catalyst optimizer. RDDs provide greater control over low-level execution but are less expressive and less optimised compared to DataFrames and Datasets [15], [17].

2.2.2 Spark Lazy Evaluation

A Spark application consists of several jobs. Within a job, there are two primary types of operations: transformations and actions [15].

Transformations are operations on DataFrames that return other DataFrames, as illustrated in Figure 2.2. They can be classified into two categories:

- **Narrow Transformations:** Each partition of a DataFrame is used by at most one partition of the child DataFrame. Examples include operations such as `filter()` and `map()`.
- **Wide Transformations:** These require data to be reshuffled across the cluster, as a single output partition depends on multiple input partitions. An example is `groupBy()`.

Wide transformations introduce shuffle operations, which are computationally expensive due to network communication and disk I/O, and therefore negatively impact performance [15], [18].

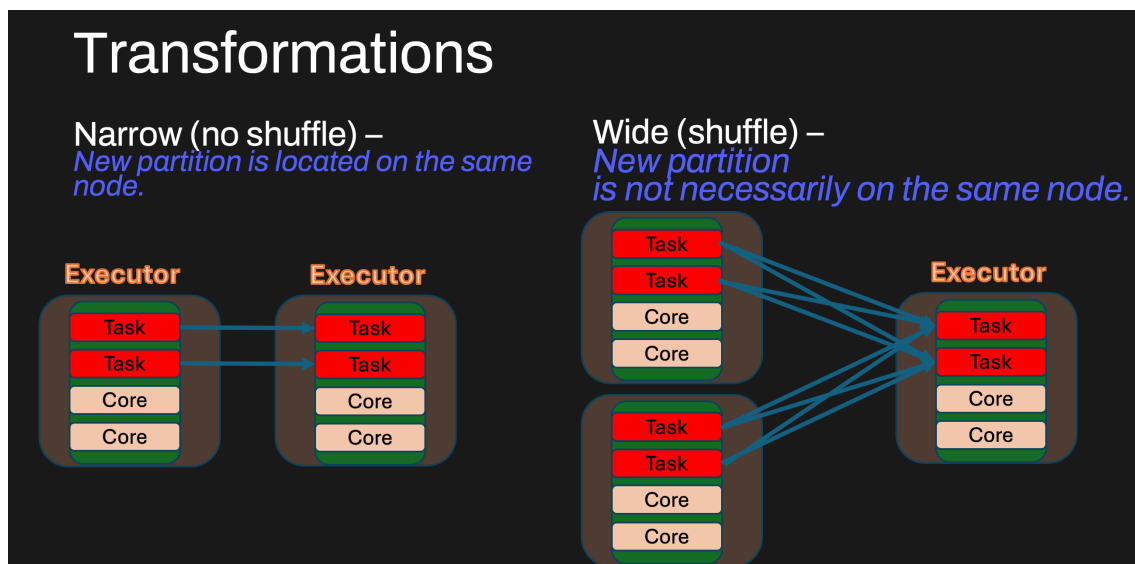


Figure 2.2: Difference in execution between narrow and wide transformations. Image taken from [18]

Actions are operations on DataFrames that trigger computation and return results, such as `collect`, `count`, and `save`. Transformations are not executed immediately; instead, Spark uses lazy evaluation, where execution is deferred until an action is invoked.

This approach improves resource efficiency by avoiding unnecessary materialisation of intermediate results. Instead of executing transformations immediately, Spark constructs a logical execution plan and optimises it before execution. This optimization is performed by the Catalyst optimizer, which determines an efficient execution strategy for the query plan [15], [17], [19].

2.2.3 Catalyst Optimizer

The Catalyst optimizer is a rule-based query optimization framework used in Spark SQL [20]. It represents queries as tree structures, where each node corresponds to logical or physical operations in the execution plan. Transformations are applied to these trees using sets of rules, which are executed iteratively in batches until a fixed point is reached, meaning no further optimizations can be applied [21].

Catalyst operates on logical and physical query plans rather than execution DAGs. The DAG-based execution model is part of Sparks execution engine, which is constructed after query optimization. Optimizations are performed before execution, when the full logical query plan is available [20], [22].

Catalyst query optimization can be described in four main phases: Analysis, Logical Optimization, Physical Planning, and Code Generation [20]:

- **Analysis:** A query begins as an unresolved logical plan generated either from a SQL parser or the DataFrame API. During analysis, Spark resolves references using the Catalog, mapping attributes to tables and columns, assigning unique identifiers, and performing type checking and propagation through expressions [20].
- **Logical Optimization:** This phase applies rule-based optimizations to the logical plan. Common optimizations include constant folding, predicate push-down, projection pruning, and Boolean expression simplification [20].
- **Physical Planning:** In this phase, Spark SQL converts the optimized logical plan into one or more physical plans composed of executable operators. A cost-based model may be used to select the most efficient plan [22]. Physical optimization can include operator fusion and reordering of operations where beneficial.
- **Code Generation:** Spark uses runtime code generation to improve execution efficiency, particularly for CPU-intensive workloads. Instead of interpreting expression trees row by row, Spark compiles parts of the execution plan into Java bytecode at runtime using whole-stage code generation, reducing interpretation overhead [22].

Catalyst may generate multiple candidate physical plans and selects the most efficient one based on cost estimation. It provides a library of logical and physical operators, as well as expression representations used throughout the optimization process [17].

2.2.4 Spark Query Plans

A Spark query plan is structured as a tree, where the depth corresponds to the longest chain of dependent operations from input to output. Logical plan depth reflects the complexity of the logical relational expressions, while physical plan depth reflects the execution operators such as aggregations and joins. Shuffle operations introduce stage boundaries in the execution plan [20], [22].

Deeper query plans typically indicate more complex sequences of dependent transformations. This can increase execution overhead due to additional intermediate operations and stage boundaries. However, performance impact depends on the effectiveness of optimizations applied during planning, and deeper plans are not inherently inefficient [20].

A Spark job is divided into stages based on shuffle boundaries. Each stage consists of a set of tasks, which represent the smallest unit of execution. Tasks within a stage are executed in parallel where possible. Transformations that do not require shuffling are typically pipelined within the same stage to reduce execution overhead [22].

2.2.5 Spark Event Logs

A Spark event log is a record of Spark applications, capturing detailed runtime information about execution. Each log consists of a stream of JSON events that record metrics about jobs, stages, and tasks, enabling post-execution analysis and reconstruction of the Spark UI. These logs are primarily used for debugging and performance tuning. Event logs are stored in persistent storage and can be accessed after application completion [23]. They contain user-specified resource configurations such as driver and executor memory, number of executor instances, CPU cores per executor, number of executors, and shuffle partition settings, among others.

From these logs, it is possible to reconstruct job and stage lifecycles, as well as extract detailed task-level metrics. These include task duration, waiting time, completion time, shuffle read and write volumes, and execution timing. The logs also capture dynamic cluster behaviour such as scaling events, where computing resources are added or removed during execution, as well as the full Spark configuration, system properties, and JAR dependencies used for the run [13], [23].

2.2.6 Spark Dynamic Resource Allocation

Spark offers Dynamic Resource Allocation (DRA), which adjusts the number of executors assigned to an application based on workload demand and available cluster resources. It enables executors to be added when there are pending tasks and removed when they remain idle for a configurable period [22].

Conceptually, DRA allows Spark applications to scale resources dynamically instead of requiring a fixed executor allocation at submission time. This behaviour is similar to Kubernetes Horizontal Pod Autoscaling (HPA), as both mechanisms adapt resource allocation based on workload characteristics. However, Spark DRA is primarily designed for resource utilization efficiency within the Spark execution model, rather than enforcing strict fairness policies or directly optimising execution time under congestion [24], [25].

2.3 Spark on Kubernetes

Apache Spark can be deployed on Kubernetes in cluster mode, where both the Spark driver and executor processes run as K8s pods, as illustrated in Figure 2.3. In this deployment model, K8s acts as the cluster manager and is responsible for scheduling pods onto worker nodes, while Spark manages application execution, task scheduling, and fault recovery [26].

When a Spark application is submitted, K8s creates a driver pod that initializes the application. The driver communicates with the K8s API server to request executor pods, which are scheduled onto available worker nodes according to the requested CPU and memory resources. Once started, executor pods process the tasks assigned by the driver and return execution results throughout the lifetime of the application [26]. Resource requests for the driver and executor pods are specified using Spark configuration parameters before execution begins. K8s uses these requests when scheduling pods and reserves the corresponding resources for each pod during its lifetime.

Spark also supports Dynamic Resource Allocation (DRA), which allows the driver to request additional executor pods when increased parallelism is required and remove idle executors when demand decreases. The number of executors is adjusted within user-defined minimum and maximum limits, while the CPU and memory assigned to each executor pod remain fixed after the pod has been created [26], [27].

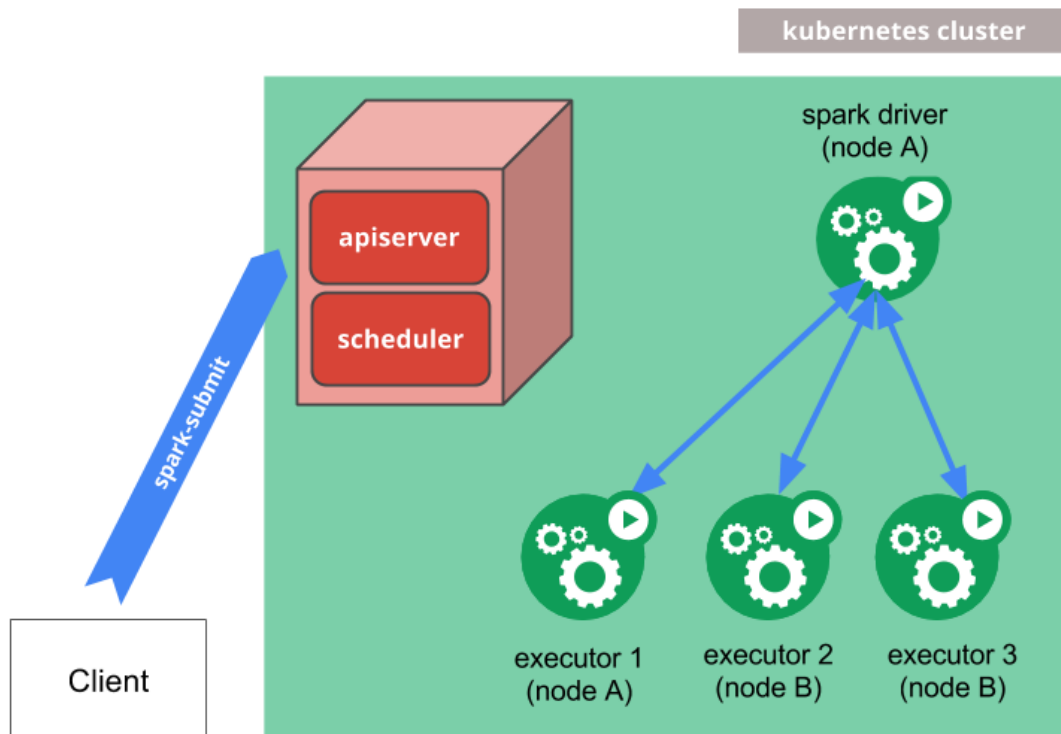


Figure 2.3: Spark on Kubernetes architecture. Image taken from [26].

2.4 Baseline: Fixed Class Resource Allocation

This section describes the current resource tuning system baseline, which will serve as the benchmark results of this thesis are compared against. The baseline reflects how Spark jobs are currently configured in practice, relying on predefined heuristics and coarse-grained job classes. Understanding its structure and limitations provides context for the design choices made in the proposed system.

2.4.1 Baseline System Overview

The baseline resource allocation strategy is based on a predefined set of job size classes, where each Spark application is assigned a fixed configuration prior to execution. These classes, defined as *small*, *medium*, *large*, and *dynamic*, are intended to approximate different workload sizes at a coarse level. In the studied environment, this class-based scheme is used as the standard mechanism for resource allocation. Decisions are made without considering workload-specific characteristics, such as query structure or input data properties. Instead, configurations are selected from a small set of predefined options designed to ensure that jobs complete successfully under typical operating conditions. While simple to use, this strategy provides only a coarse approximation of workload requirements, as it does not account for fine-grained differences between individual Spark applications.

2.4.2 Resource Configuration Model

Each job class is associated with a fixed set of Spark configuration parameters. In the baseline setup, all classes use the same executor size, with `spark.executor.memory` set to 64 GB and `spark.executor.cores` set to 4. The primary distinction between classes lies in the number of executor instances: 2, 5, and 10 for the small, medium, and large classes, respectively. The dynamic class uses Spark's dynamic resource allocation mechanism, where the number of executors can vary within a predefined range. In this case, the initial number of executors is set to 4, with a minimum of 0 and a maximum of 10. In addition, the number of shuffle partitions is derived from available parallelism and is defined as a fixed multiple of compute capacity:

$$shuffle\ partitions = 2 \times N_{executors} \times N_{cores}$$

where $N_{executors}$ corresponds to the configured executor count for each class, or the maximum number of executors in the case of dynamic allocation. As a result, scaling between classes is achieved by increasing parallelism, while keeping executor size fixed. The code listing in figure 2.4 illustrates the aforementioned resource allocation heuristics.

```

if size not in ["large", "medium", "small", "dynamic"]:
    logger.warning(
        "Allowed size is 'dynamic', 'large', 'medium' "
        "or 'small'. Size 'test' is no longer valid."
        " Setting size to dynamic (default).")
    )
    size = "dynamic"

if size == "large":
    instances = "10"
    memory = "64g"
    cores = "4"

if size == "medium":
    instances = "5"
    memory = "64g"
    cores = "4"

if size == "small":
    instances = "2"
    memory = "64g"
    cores = "4"

if size == "dynamic":
    memory = "64g"
    cores = "4"
    dynamic = True
    min_executors = "0"
    max_executors = "10"
    initial_executors = "4"
    shuffle_partitions = f"{int(max_executors) * int(cores) * 2}"
else:
    shuffle_partitions = f"{int(instances) * int(cores) * 2}"

```

Figure 2.4: Code listing showing the current configurations users can run Spark with.

2.4.3 Limitations of the Baseline Approach

While simple to use, the baseline strategy provides only a coarse approximation of workload requirements, as it does not account for differences in workload characteristics between individual Spark applications. Applications may vary in their resource demands depending on factors such as data volume, query structure, execution patterns, and whether they are primarily CPU-bound, memory-bound.

A first limitation is the rigidity of the configuration model. All job classes use a fixed executor size, with 64 GB of memory and 4 cores per executor. As a result, scaling

between classes is primarily achieved by varying the number of executors, which significantly restricts the configuration space and limits adaptation to workload-specific resource needs.

Furthermore, the coarse-grained class definitions fail to capture the diversity of real-world Spark applications. Jobs with similar input sizes may exhibit different execution behavior depending on query structure, while identical query structures may still require different resources when applied to different data volumes. This mismatch makes it difficult for the baseline system to reliably align resource allocation with actual workload characteristics.

The limitations of fixed resource allocation motivate approaches that can learn workload characteristics from historical execution data and make more workload-aware resource allocation decisions without relying solely on coarse predefined classes. Machine learning provides methods for identifying patterns in complex datasets and can therefore be applied to tasks such as workload characterization and resource prediction. Historical execution data also needs mechanisms to handle inconsistent or sparsely available data, which uncertainty-awareness is a possible solution for.

2.5 Relevant Machine Learning Concepts

This section presents the machine learning concepts required to understand the methods used in this thesis, including tree-based models, gradient boosting, clustering algorithms, and dimensionality reduction techniques.

2.5.1 Tree-based Machine Learning Models

Tree-based models are widely used within the machine learning domain because they can model relationships between input variables and target values using a hierarchical tree structure. Each node in the tree represents a decision based on one or more input variables, which are used to produce a prediction for the target value. Decision trees employ a divide-and-conquer strategy by conducting a greedy search to identify suitable split points within the tree. Machine learning uses tree-based models for both classification and regression tasks. Classification predicts discrete, categorical labels, while regression predicts continuous numerical values. The input variables are repeatedly segmented into subsets to build decision trees, where each split is selected based on a defined optimization criterion to improve prediction performance [28].

2.5.2 Gradient Boosting Classifier

Gradient Boosting Classifier is a supervised machine learning algorithm that applies the gradient boosting framework to classification problems. It combines multiple weak prediction models, typically decision trees, into a stronger predictive model. Instead of training each tree independently, gradient boosting builds the ensemble sequentially, where each new tree is trained to reduce the errors made by the current model.

The training process consists of repeatedly fitting new trees to the errors, or residuals, produced by the current ensemble. As illustrated in Figure 2.5, each additional tree contributes to correcting previous predictions and improving the overall model. The contribution of each individual tree is controlled by a learning rate parameter η , which reduces the impact of each update and helps control overfitting.

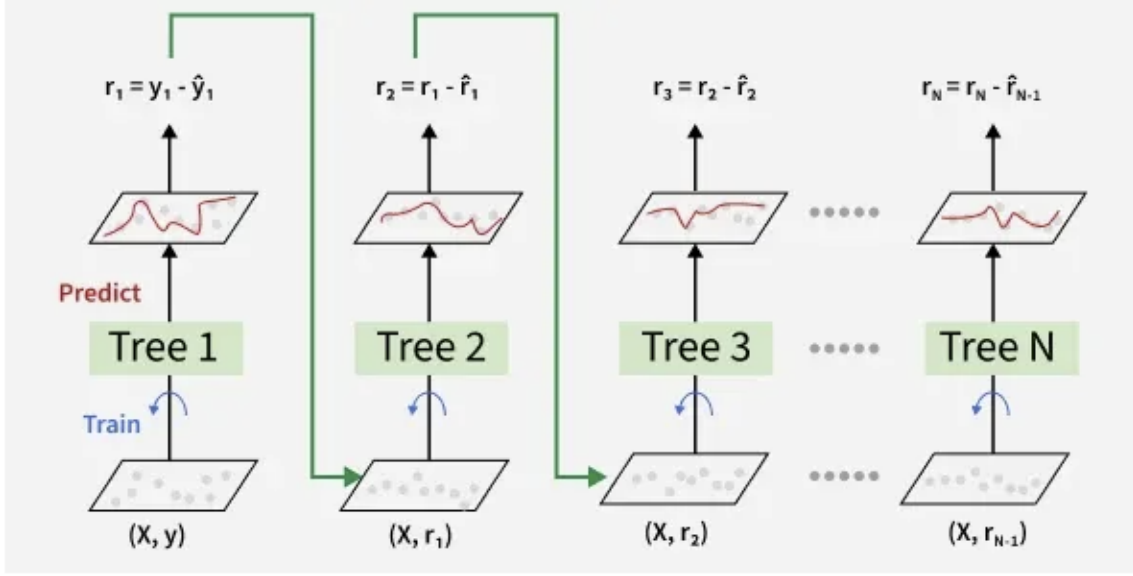


Figure 2.5: Illustration of the sequential training process in gradient boosting. Image taken from [29].

The final prediction is obtained by combining the contributions from all individual trees in the ensemble. A general formulation of the model after M boosting iterations is:

$$F_M(x) = F_0(x) + \eta \sum_{m=1}^M h_m(x)$$

where $F_0(x)$ represents the initial prediction, $h_m(x)$ represents the prediction from the m -th weak learner, and η is the learning rate controlling the contribution of each tree.

In this thesis, gradient boosting is used as a classification model to predict workload-related categories from Spark execution characteristics. The specific model configuration, feature selection, and evaluation process are described in the methodology chapter.

2.5.3 Clustering Algorithms

Clustering is an unsupervised machine learning technique designed to group unlabeled data based on similarity [30]. It helps identify patterns, summarize information, and support decision-making processes. Manually inspecting large datasets can

be a slow and tedious process. By condensing complex datasets into manageable groups, clustering allows data instances with similar characteristics to be grouped together. Clustering algorithms can be used to generate structured data for further analysis, primarily through feature engineering, dimensionality reduction, and data labeling. Clustering was used in this thesis to turn a regression problem into a classification problem, which will later be motivated further in the Methods section.

2.5.4 DBSCAN Clustering Algorithm

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm introduced by Ester et al. [31]. It identifies dense regions separated by areas of lower density and groups together points that are closely located while marking points in low-density regions as noise. DBSCAN is particularly useful when clusters have arbitrary shapes, the number of clusters is unknown, and outliers or noise are important factors to consider [31].

Clusters are formed based on ε (epsilon), which defines the radius of a neighborhood, and the minimum number of points required to form a dense region. Each point in the dataset can be classified as either a core point, a border point, or a noise point [31]. A core point is a point that has at least the minimum number of points within its ε -neighborhood. A border point has fewer than the minimum required number of neighbors but lies within the neighborhood of a core point. A noise point is neither a core point nor a border point.

DBSCAN begins by selecting an unvisited point and identifying its ε -neighborhood. If the point satisfies the density requirement, a new cluster is created and neighboring points are recursively added. This process continues until no additional points can be assigned to the cluster [31]. DBSCAN is capable of identifying clusters of arbitrary shapes, detecting outliers, and separating noise from meaningful data.

2.5.5 Dimensionality Reduction Techniques

Data analysis is required to extract meaningful information, support decision-making, and draw conclusions from datasets. Different types of analysis exist, including descriptive, diagnostic, exploratory, and inferential analysis. Descriptive analysis summarizes and describes datasets using statistical measures such as central tendency, variability, frequency, and position. To interpret high-dimensional data more effectively, dimensionality reduction techniques such as PCA and t-SNE can be applied. These techniques were used in this thesis for feature extraction and visualization, which will later be explained further in the Methods section.

Principal Component Analysis (PCA) is a dimensionality reduction technique used for feature extraction, noise reduction, and visualization. PCA transforms correlated variables into a smaller set of orthogonal components while preserving the maximum possible variance in the data [30], [32]. It is useful for reducing multicollinearity because the generated principal components are uncorrelated. The contribution of the original features can be analyzed by examining their weights within each principal component.

PCA transforms the coordinate system of the data so that the directions containing the greatest amount of variance are identified first. The resulting orthogonal axes, known as principal components, capture decreasing amounts of variance, with the first principal component capturing the maximum possible variance [32].

t-distributed Stochastic Neighbour Embedding (t-SNE) is an unsupervised, non-linear dimensionality reduction technique primarily used for data exploration and visualization. Unlike PCA, which preserves global variance, t-SNE focuses on preserving local similarities between data points, making it suitable for identifying clusters in high-dimensional datasets [33].

2.6 Uncertainty and Wilson Confidence Intervals

Finally, it is also relevant to describe the relevant uncertainty-aware statistics this thesis uses. When making decisions based on observed data, there is always some degree of uncertainty. An observed result is based on a finite number of observations and may therefore not perfectly represent the underlying behaviour. In general, the fewer observations available, the greater the uncertainty in the estimated result. As more observations are collected, the estimate generally becomes more reliable and the uncertainty decreases.

A Wilson confidence interval is a method for representing this uncertainty when estimating a proportion. For example, if 8 out of 10 attempts of an experiment are successful, the observed success rate is 80%. However, this does not necessarily mean that the true success rate is exactly 80%. A Wilson confidence interval instead provides a range of plausible values for the true success rate. The interval is generally wider when only a small number of observations are available and becomes narrower as more observations are collected.

$$\hat{p}_{\pm} = \frac{\hat{p} + \frac{z^2}{2n} \pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}$$

$\hat{p}$: Sample success proportion.

n : Sample size.

z : Critical value from standard normal distribution

3

Related Work

This section presents previous research related to improving resource utilization in Apache Spark using both machine learning-based and traditional approaches. It also covers related work on resource allocation optimization in Kubernetes environments. Across the reviewed literature, memory allocation has been the primary area of focus.

3.1 Predictive Resource Allocation

Machine learning has become an increasingly important approach for improving resource management in distributed systems by learning patterns from historical execution data and predicting future resource requirements. In Apache Spark environments, jobs often exhibit diverse and workload-dependent resource demands, making static resource allocation inefficient. Consequently, machine learning techniques have been explored to predict resource utilization and support more adaptive resource allocation decisions.

Predictive resource allocation approaches learn resource usage patterns from historical job executions using metrics such as historical resource consumption, application-specific characteristics, and execution behaviour. These learned patterns are then used to estimate future resource requirements and guide resource allocation decisions that improve resource utilization while maintaining application performance.

One example is the HPA+ scheduler proposed by Toka et al. [8], which explicitly balances the likelihood of SLA violations against overall system utilization. Higher utilization increases the risk of SLA violations due to reduced spare capacity, whereas lower utilization leaves more resources idle but reduces the likelihood of violations. Other approaches include Paragon [6] and Quasar [7], which profile applications and use classification models trained on historical performance data to predict how applications will behave under different resource configurations. These predictions are then used to select resource allocations that satisfy performance objectives while improving overall cluster utilization.

3.2 Memory Optimization

3.2.1 Machine Learning Methods

Machine learning methods have also been explored to optimize Apache Spark task configurations and improve memory allocation efficiency. One study, [34], investigates the use of machine learning models, including Linear Regression, Decision Trees, Support Vector Machines (SVM), and Artificial Neural Networks (ANN), for optimizing memory allocation in cloud-based Spark environments.

The experiments evaluate different workload types, including memory-intensive, iterative, CPU-intensive, and I/O-intensive workloads, by varying `spark.executor.cores` and `spark.executor.memory`. Performance is evaluated using benchmark workloads such as Word Count, PageRank, K-Means, and Terasort. The authors report that ANN achieves the highest prediction accuracy of approximately 97%. However, the study does not disclose the features used by the models, making it difficult to evaluate the generalizability of the approach. Furthermore, if sparse features were used with a small dataset, there is a potential risk of model overfitting.

3.2.2 Tuning Apache Spark Jobs

Apache Spark has tunable parameters such as memory allocation, number of executors/cores and shuffle settings. However, it can be challenging to tune these manually because in most cases, they depend on workloads; different jobs behave differently and while cluster conditions change, static tuning might not be optimal. The paper “Towards General and Efficient Online Tuning for Spark” enables meta-learning to improve early stage tuning. Whenever a periodic Spark job is about to run, the system will suggest the configuration, observe results after execution, and uses the feedback to improve the next suggestion [35].

3.2.3 Improving Execution in Kubernetes

Job Scheduling Mechanism (JSM) an infrastructure designed to lower costs and improve resource management for big data applications running on Apache Spark cluster managed by Kubernetes [36]. JSM overcomes the challenges of First In First Out Scheduler which does not balance resources allocation across varying big data applications. JSM monitors cluster load where it calculates dynamic resource coefficients and weights for both CPU and memory and predicts upcoming workloads and ranks the active pods from over-loaded to low-loaded. By taking advantage of the Best-Fit algorithm, JSM determines the optimal pod allocation for a new job by consolidating active jobs into underutilized pods and migrates tasks from overloaded clusters. When an under-loaded pod is cleared of tasks, the system terminates it to prevent idle resource waste.

3.2.4 Queries as ML Features in Apache Spark.

Cloud analytics systems often adjust resources only after a query has already started running, which can be inefficient because the query may have already passed the point where extra resources help most. The paper *Predictive Price-Performance Optimization for Serverless Query Processing* presents AutoExecutor, a Spark SQL system in Azure Synapse that tries to make resource adjustment decision earlier by estimating how a query will perform under different resource levels before execution begins [37]. Instead of relying on runtime statistics, which are available after the query finishes, it uses information available from the query plan and input data. The authors tested two ways of representing the runtime to resources relationship, a power-law with saturation model and an Amdahls Law model. They first trained a machine learning model to predict the parameters of these curves from query features, then used the predicted curve to pick a resource setting that balanced performance and cost. The main takeaway was that query characteristics can be used to forecast a runtime curve and choose a near-optimal Spark configuration ahead of time.

3.3 Gaps in Related Work

The literature review identified two under-explored areas of research: predictive resource management for Spark on Kubernetes, and uncertainty-aware resource correction policies. The following sections summarize the limitations of existing approaches that motivate these research directions.

3.3.1 Predictive Resource Tuning for Spark on K8s

Resource prediction and allocation have been studied for Apache Spark, including approaches that use workload characteristics and historical execution data to estimate resource requirements. However, existing approaches do not address resource tuning for Spark workloads running on Kubernetes. For example, Dong et al. focus on performance- and energy-aware resource allocation in Kubernetes but assume that submitted resource requests are reasonable and do not address the correction of inaccurate requests [38].

This leaves a more specific gap in predictive resource tuning for Apache Spark workloads running on Kubernetes. In particular, there is limited work on using pre-execution Spark workload characteristics to identify similar workloads and using their previous execution outcomes to correct resource requests for future executions.

3.3.2 Uncertainty-Aware Resource Correction Policies

Many existing predictive resource management approaches provide point estimates of resource usage. For example, approaches such as Toka et al. and Qiu et al. predict expected resource consumption but do not incorporate uncertainty estimates, such as confidence intervals or probabilistic predictions, into tuning decisions [8], [39].

In dynamic environments, resource requirements may vary significantly between workloads and during execution. Scheduling policies that account for uncertainty could improve reliability by considering the likelihood of resource deviations rather than relying on a single predicted value. This is particularly relevant in shared clusters, where inaccurate predictions can contribute to inefficient resource utilization, increased queueing times, and failed or delayed executions.

Together, these gaps motivate further investigation into predictive resource tuning for multi-node Spark workloads on shared Kubernetes clusters. Specifically, exploring predictive resource tuning for Apache Spark on Kubernetes, and incorporating uncertainty awareness into resource tuning, represent opportunities to develop more adaptive and robust scheduling approaches for data-intensive applications.

4

Methods

This chapter first presents the overall system architecture and the interaction between workload characterization, resource configuration selection, and execution feedback. It then describes the construction of the workload "footprint" model, followed by the online classification and feedback-based tuning process.

4.1 System Overview

Spark workloads can exhibit highly variable execution behavior and small differences in workload characteristics can result in different resource requirements. Instead of attempting to directly predict an optimal configuration, the proposed approach first identifies similar workload patterns and then uses historical execution outcomes to refine configuration decisions over time. This decision is motivated in sections 4.2.4 and 4.2.5.

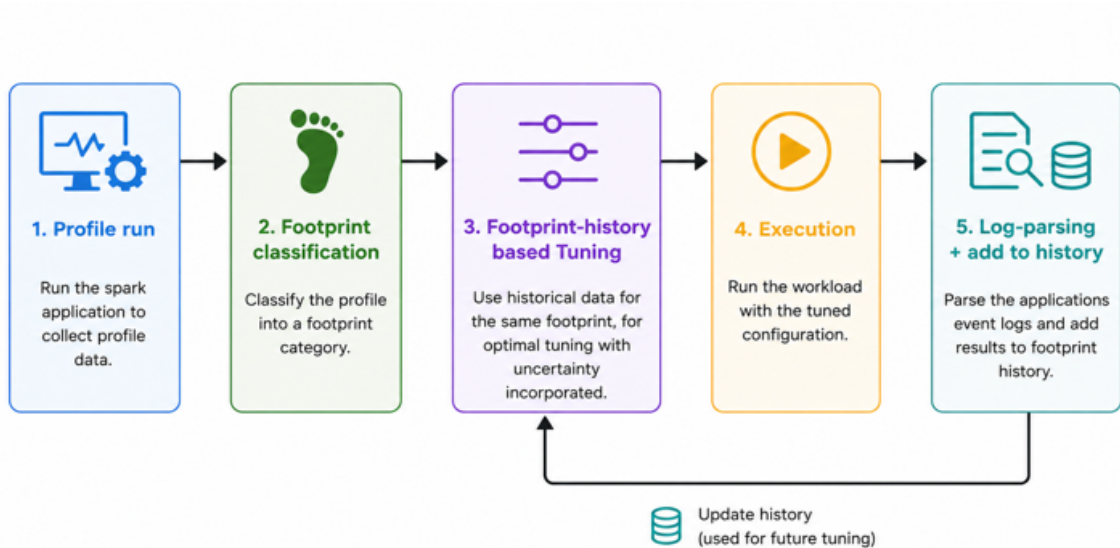


Figure 4.1: Overview of the proposed resource tuner architecture.

As shown above in figure 4.1, the system is structured as a modular pipeline that maps Spark job submissions to resource allocation decisions through a sequence

of feature extraction, workload classification, and policy selection steps. Upon job submission, pre-execution features are extracted from the Spark physical execution plan and input file metadata. These features form the input to a footprint classification ML model trained on past job executions, which maps each job to a discrete workload category representing its execution behavior. Based on the predicted footprint, resource tuning based on that footprint’s historical data is applied prior to execution, and the job is then executed on the Spark cluster. After execution, result indicators are collected and stored, along with the predicted footprint and selected configuration.

The system operates as a feedback-enabled decision pipeline over time. The footprint classifier is trained offline using historical event-log data and remains fixed during online operation. During execution, the predicted footprint is combined with historical observations of similar footprints to guide resource allocation decisions. These historical observations are maintained as an expanding set of footprint, configuration, outcome tuples, which are used to refine future decisions under the same footprint class.

4.2 Offline Footprint Model Construction

This section describes how the machine learning model capable of classifying Spark jobs into the aforementioned footprint categories was built. The process consisted of data cleaning, extraction of structural and metadata-based signals.

4.2.1 Training Data Collection

The footprint model was trained using Spark Event Logs from ETL (Extract, Transform, Load) workloads collected in the studied environment. The environment-specific nature of the data was considered a feature of the system, because the proposed system targets similar or repeated workloads. Each event log was treated as a single workload, although an event log technically corresponds to a Spark Session. This was considered defensible because most logs originated from individual Apache Airflow jobs, where each job initialized a Spark Session, executed the workload, and terminated the session.

Prior to feature extraction, non-informative executions were removed. This included no-op applications that initialized a Spark Session without executing any jobs, stages, or tasks; lightweight metadata-probe applications performing trivial operations such as `.head()` or `.first()`; and applications without meaningful input data or execution structure, where runtime primarily reflected JVM startup and cluster allocation overhead. These executions were excluded because they provided little information about the relationship between workload characteristics and resource requirements.

Lightweight but valid workloads, such as simple file or DataFrame reads, were retained. Although these workloads contained limited query plan structure, their input metadata still provided useful information about workload scale. Retaining them also ensured that the training data covered simple but realistic Spark workloads.

4.2.2 Feature Engineering

The feature representation was constructed from Spark Event Logs of historical ETL workloads. The system targets Python-based Spark applications using the DataFrame and SQL APIs, which are commonly used for ETL processing [40]. Each workload was represented using features extracted from its physical query plan and selected input metadata.

Structural features were extracted from the physical query plan using text parsing and regular-expression matching. These features captured the presence and frequency of operators such as scans, joins, aggregations, sorts, exchanges, and window functions. The number of input files was also included as a coarse measure of input scale, as this was the only information consistently available both from historical event logs and during pre-execution profiling. The resulting features were aggregated into a fixed-length numerical vector and used as training data to the footprint model. This representation therefore captured structural characteristics of the workload and an available indicator of its input scale.

4.2.3 Feature Selection

The initial feature set consisted of query plan features and job-level metadata described in the previous section. PCA and t-SNE were applied to the full feature set to examine how the workloads were distributed in the feature space and whether some features provided similar information.

Feature importance was also evaluated using a Gradient Boosting model trained for footprint classification. Features with consistently low importance were considered for removal. The effect of removing features was then evaluated using cross-validation on multiple data splits to determine whether the reduced feature set maintained classification performance.

Based on these analyses, the 44 candidate features were reduced to a final set of 14 features. The final features were selected from the original feature set rather than being principal components. The set of features for the footprint clustering and classifier algorithms was found empirically, by starting with a wide initial range of possible metrics from Spark query plans and memory, and editing the feature set based on the feature selection strategies formulated. The following list of 14 features was found to produce the best clustering. Sources supporting the plausibility of each feature as a representation of workload characteristics are also provided.

input_num_files

The number of input files read by the Spark application, extracted from the query plan’s scan nodes. A high file count typically indicates a wide, partitioned read pattern and tends to increase scheduling and partition-management overhead, making it a useful indicator of I/O-bound workload scale [41], [42].

plan_max_depth

The maximum depth of the physical query plan tree. Deeper plans can be treated as a proxy for more complex dependency structure, since Spark executes jobs through

DAG-based physical execution plans and workload-aware optimization methods explicitly use DAG structure to choose execution settings [42], [43].

max_scan_columns

The largest number of columns read in any single scan node across the plan. Wider scans plausibly increase per-task memory footprint and serialization cost, because Spark memory behavior is sensitive to in-memory representation and serialized data layout, although the supplied sources do not test scan width directly [44], [45].

num_scans

The total count of scan nodes in the query plan. Multiple scans indicate that the workload reads from several distinct data sources or re-reads the same source, which tends to increase I/O cost and can amplify executor memory demand when scans are broad [45], [46].

num_sort_nodes

The number of sort operators in the plan. Sorts are memory-intensive operations that can spill to disk when executor memory is insufficient, so a high sort node count signals elevated memory pressure and sensitivity to execution-memory sizing [44], [47].

num_window_nodes

The count of window function operators. Window functions require ordered, partitioned data and can maintain substantial in-memory state per partition, making them a strong signal for elevated memory demand and spill risk, particularly on large datasets [48].

num_joins

The total number of join operators across the plan. Joins, especially sort-merge joins, are strong drivers of shuffle volume and peak memory usage, so a higher join count increases the likelihood of memory pressure and longer stage runtimes [44], [49].

num_exchanges

The number of exchange (shuffle) operators in the plan. Each exchange materializes intermediate data across the network and often to disk, making this feature a direct proxy for shuffle intensity and a strong indicator of memory and CPU resource demand [46], [47].

has_subquery

A binary indicator for the presence of correlated or uncorrelated subqueries. In this feature set, subquery presence is best treated as a possible complexity marker, since the supplied sources support the importance of stateful operators and optimizer rewrites more directly than repeated subquery evaluation itself [47].

has_window_function

A binary flag indicating whether the workload uses any window functions. As a complement to **num_window_nodes**, this feature captures the qualitative presence of windowed computation as a distinct workload type, even when only a single window node exists [48].

has_sort

A binary indicator for the presence of any explicit sort operations. Even a single sort can be a useful qualitative signal because sort uses execution memory and can become spill-sensitive under constrained executor memory [44].

has_repartition

Indicates whether the plan contains an explicit repartition operation. Repartitioning forces data redistribution and directly influences network traffic and executor memory requirements, regardless of the number of downstream operations [46].

has_distinct

A binary flag for the presence of distinct operations in the plan. Distinct computations require deduplication across partitions, typically via aggregation or sort, which increases shuffle-space demand and spill risk in a manner similar to other stateful operators [47], [50].

has_cache

Indicates whether the application explicitly caches a `DataFrame` or `RDD`. Caching retains materialized data in executor memory across actions, increasing steady-state memory consumption while reducing recomputation and disk I/O cost [51], [52].

4.2.4 ML Model Selection

The available data was represented in tabular form, where each workload was described by a set of numerical and categorical features. This representation motivated the use of tree-based learning methods, which are well-suited for handling heterogeneous feature types, non-linear relationships, and interactions between variables without requiring extensive feature scaling or transformation.

Gradient Boosting Trees were selected as the model family for the footprint classification task. Gradient boosting combines multiple decision trees sequentially and can model non-linear relationships and interactions between features. These properties made it a suitable candidate for the structured, tabular feature representation, produced by the feature extraction explained in section 4.3.

A key design decision was the choice between regression and classification. An initial approach based on direct regression was considered, where the model would predict Spark resource configuration parameters such as executor memory and cores. However, this formulation proved unstable in practice, with unreliable predictions and high variance in model outputs. This suggested that the mapping from pre-execution features to exact resource values was not feasible to learn directly.

To address this limitation, the learning problem was formulated as a supervised task operating on workload representations. Each workload was associated with a target label derived from its observed execution behavior, and the objective was to learn a mapping from pre-execution features to this target representation. This will be presented in the next section.

4.2.5 Footprint Class Generation using Clustering

To avoid learning a direct mapping from high-dimensional workload features to recommended resources, the system instead introduced an intermediate representation of workloads. Each Spark job was associated with a discrete footprint class that captured similarities in execution behavior.

A footprint was defined as a cluster of Spark workloads that exhibited similar structural and performance-related characteristics. These characteristics were derived from the aforementioned query plan structure and input metadata, available from Spark event logs. The goal of this representation was to group workloads that were expected to have similar resource requirements under comparable execution conditions.

Footprint classes were constructed using an unsupervised clustering approach, specifically DBSCAN. The clustering process operated on the combined feature representation introduced earlier. In this feature space, workloads with similar computational patterns were grouped together, while variations in input scale contributed to cluster separation where relevant.

The resulting clusters defined the set of footprint classes used in the system. This abstraction reduced variability in the learning problem by replacing individual job-level prediction with classification over a finite set of workload types. It provided a stable intermediate representation that supported more robust downstream resource allocation decisions.

4.3 Online Profiling and Footprint Classification

To enable feature extraction from Spark workloads prior to execution, instrumentation code had to be written in the job source code. Each Spark job was extended with a profiling mode that exposed the final DataFrame lineage without triggering job execution. This is illustrated in the pseudocode in Figure 4.2 below.

```

# example_spark_job.py
import sys
from app_extract import create_feature_row

...

last_df = # last transformation defined in the Spark application

mode = sys.argv[1]
if mode == "profiling":

    # Extract features from the logical query plan
    create_feature_row(last_df)

elif mode == "execute":
    # Execute the full Spark job

```

Figure 4.2: Instrumentation of Spark jobs for feature extraction.

In profiling mode, the Spark application constructed the transformation graph without triggering execution using lazy evaluation. The resulting physical query plan was extracted and passed to the feature extraction function, which converted the relevant properties into the feature vector described in Section 4.2.3.

Categorical and binary features were one-hot encoded where required. The resulting feature vector was passed to the trained footprint classifier, which assigned the workload to a footprint class. This class was then used by the resource tuning stage to select the resource configuration for the workload.

4.4 Uncertainty-Aware Resource Tuning

The limitations of the baseline system described in Section 2.4.3 motivated a workload-aware approach to resource allocation. The proposed tuner used observations from previous executions of the same footprint to determine the resource configuration for the next execution. The tuner adjusted two Spark parameters: `spark.executor.cores` and `spark.executor.memory`. The number of executor instances was not tuned in this thesis and was fixed at 10, to make it comparable to the baseline "large" configuration.

4.4.1 Memory tuning

A concise way to describe the algorithm would be through pseudocode. This section will present the algorithm like that in Algorithm 1 below, and then explain the steps in more detail. The pseudocode for this algorithm looks like the following:

Algorithm 1 Uncertainty-Aware Memory Tuning Algorithm

Require: Initial memory allocation $M \leftarrow 64$ GB; minimum allocation $M_{\min} \leftarrow 8$ GB; step size $\Delta M \leftarrow 8$ GB

Require: Confidence level $\gamma \leftarrow 0.80$; lower threshold $\tau_L \leftarrow 0.75$; upper threshold $\tau_U \leftarrow 0.95$

Ensure: Selected memory allocation M

```

1: Initialize observation statistics for each memory level
2: Initialize set of discarded memory levels  $\mathcal{D} \leftarrow \emptyset$ 
3: while tuning continues do
4:   Execute Spark job using memory allocation  $M$ 
5:   Record execution outcome and update statistics for  $M$ 
6:    $n \leftarrow$  number of observations at level  $M$ 
7:   if  $n < 2$  then
8:     Retain  $M$  and continue
9:   end if
10:  Calculate the  $\gamma$ -level Wilson interval  $[L, U]$ 
11:  using the cumulative success rate at level  $M$ 
12:  if  $L \geq \tau_L$  then
13:    if  $M - \Delta M \geq M_{\min}$  then
14:       $M_{\text{next}} \leftarrow M - \Delta M$ 
15:      if  $M_{\text{next}} \notin \mathcal{D}$  then
16:         $M \leftarrow M_{\text{next}}$ 
17:      end if
18:    end if
19:  else if  $U < \tau_U$  then
20:    Add  $M$  to  $\mathcal{D}$ 
21:     $M \leftarrow M + \Delta M$ 
22:    while  $M \in \mathcal{D}$  do
23:       $M \leftarrow M + \Delta M$ 
24:    end while
25:  else
26:    Retain the current memory allocation  $M$ 
27:  end if
28: end while
29: return  $M$ 

```

The memory tuner evaluated Spark executor memory configurations using historical execution outcomes associated with each footprint (first requirement). A memory level represented an executor memory allocation, with levels separated by 8 GB (e.g., 64 GB, 56 GB, 48 GB). Starting from the baseline system’s initial allocation of 64 GB, the tuner maintained cumulative statistics for each memory level, recording the number of observations, successful executions, and failed executions (lines 1-2).

A memory level was evaluated once at least two observations had been collected (lines 7-9). The tuner then calculated an 80% two-sided Wilson confidence interval for the cumulative success rate of the current memory level (line 10). The Wilson

interval was used instead of relying directly on the observed success rate, thereby accounting for the uncertainty associated with a limited number of observations. An 80% confidence level was selected to balance uncertainty and responsiveness. Higher confidence levels would have required stronger evidence before changing configurations, slowing down the tuning process. A lower confidence level would do the opposite, tune faster, but fail more often.

If the lower bound of the Wilson interval was at least 0.75, the tuner considered the current memory level sufficiently reliable to attempt a reduction (lines 12-18, 25-27). It therefore decreased the allocation by 8 GB, provided that the resulting level was at least 8 GB and had not previously been discarded. Conversely, if the upper bound of the Wilson interval was below 0.95, the tuner considered the current memory level insufficiently reliable. The level was then discarded and the memory allocation was increased by 8 GB (lines 19-25). Discarded levels were not evaluated again, and the tuner skipped over any subsequently encountered discarded levels when increasing the allocation. If neither condition was satisfied, the tuner retained the current memory allocation. Thus, the tuner did not increase memory only because the evidence was inconclusive, but rather only when the current level’s Wilson upper bound fell below the required reliability threshold.

The 75% lower bound was selected to not tune down too fast on little evidence. The 95% upper bound in order to not tune up too fast on small signs of failure, while still keeping a high standard for the long-term success rate. Overall, the tuner used the confidence interval to distinguish between configurations for which the available execution history provided sufficient evidence for a change and those for which the evidence was inconclusive. In the latter case, the current allocation was retained rather than changed based on limited observations.

4.4.2 Cores tuning

The cores tuner evaluated Spark executor core configurations using the execution time observed for each configuration within a footprint. The allowed configurations were 1, 2, 3, and 4 executor cores, with 4 cores corresponding to the baseline system configuration. The tuner collected five observations for each core configuration. For each execution, the observed cores-cost was calculated as the product of the number of assigned executor cores and the execution time of the job. The five observations collected for each configuration were then averaged to obtain an estimate of their mean cores-costs. Once all four configurations had been evaluated, the tuner selected the configuration with the lowest average cores-cost.

Following this initial comparison, every subsequent execution contributed an additional observation to the statistics of the selected configuration. The corresponding average cores-cost was therefore updated as new evidence was added. This made the cores tuner robust in the sense that configuration decisions were based on empirical evidence rather than on a single execution, but not truly uncertainty-aware. As the number of observations increased, the estimated cores-cost became less sensitive to individual executions, providing increasing confidence in the efficiency of the selected configuration.

4.5 Execution Outcome Modeling

In order for the resource tuning algorithms defined in the previous section to get the data they needed, the Spark event log produced by each run was used to derive four metrics: assigned executor cores, assigned executor memory, job execution time, and execution outcome. The first three metrics were trivial to obtain from the event logs. But the execution outcomes were more nuanced, which is why the following categorization of job outcome was defined. Execution outcomes were classified into three categories:

- **Success:** The job completed successfully.
- **OOM failure:** The job failed or was sub-optimally executed due to an observed out-of-memory condition.
- **Other failure:** The job failed for another reason, e.g crashed or cancelled etc.

The execution outcome was classified based on the failure evidence observed in the Spark event log. An execution was classified as OOM if at least one executor was removed with exit code 52 or 137, or if at least one failed job (jobs in the event log correspond to actions) contained the corresponding OOM indication. If no OOM was detected but at least one executor was removed with a non-zero exit code, or at least one job failed for another reason, the execution was classified as *other*, and not counted in order to not affect the historical data. Otherwise, the execution was classified as *success*. Normal executor removals with exit code 0, such as those caused by Dynamic Resource Allocation, were not considered failures. When both failure types were observed, OOM was assigned.

5

Evaluation

This section describes the experimental methodology used to evaluate the proposed resource configuration tuning system. The experiments are designed to assess three main capabilities of the system: (1) whether Spark workloads can be meaningfully grouped into workload footprints based on pre-execution characteristics, (2) whether the system can use these workload footprints to select more appropriate resource configurations, and (3) whether the feedback-based tuning mechanism can gradually refine configurations based on observed execution outcomes.

The experiments took place in a RKE2 Rancher Kubernetes Cluster. The project was allocated a separate namespace that had resource quotas of 48 CPU cores and 832 GB of RAM. This is the list of versions of the software used in this project:

- Kubectl Client: v1.33.0
- RKE2 Server: v1.33.8+rke2r1
- Python: 3.11
- Apache Spark: 4.0.2
- Docker: 29.1.3

This is some training dataset feature metadata:

- Number of Spark jobs (rows) used (raw): 14038
- Number of Spark jobs (rows) used (clean): 3674
- Footprint (class) count: 17
- Feature count (before PCA): 44
- Feature count (after PCA): 14
- DBSCAN eps value: 0.8
- Minimum samples to form core point: 10

The GB-classifier was implemented using the `xgboost` library. Initial development and experimentation was conducted using a containerized Spark environment deployed on Kubernetes Minikube, a virtual Kubernetes cluster that is deployed on your local machine. When prototyping was finished, the implementation was moved to the cluster environment.

In order to test the implementation, some experiments were designed. Each experiment aimed to demonstrate and verify how a specific part of the system worked. The parts that were tested were:

1. Ability to classify workloads into footprints based on execution nature (RQ1).
2. Ability to gradually tune down a given test job over 40 iterations (RQ2).
3. Ability to tune workloads of different resource profile differently (RQ1, RQ2).
4. Ability to tune up when encountering OOM fails (RQ2).
5. Ability to allow some failure in the case that previous history strongly backs a configuration (RQ2).

5.1 Clustering Analysis

The data varied across several dimensions, though it could not be classified as random noise 5.1. There were dominant plan complexities and meaningful independent variations in window, input, scan and sort vs repartition behavior. The data had a variance of about 64%, for the selected features as seen in table 5.1.

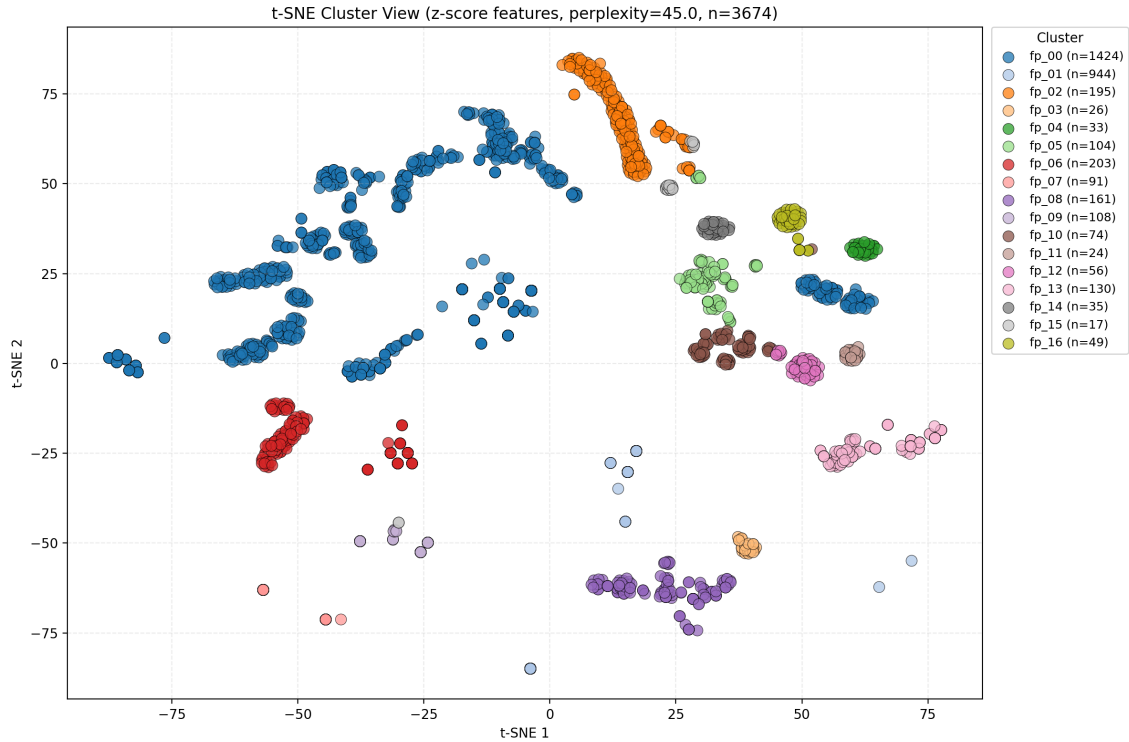


Figure 5.1: Cluster Distribution K=17

PC1 (36.12% Var.)		PC2 (14.75% Var.)		PC3 (13.32% Var.)	
Feature	Loading	Feature	Loading	Feature	Loading
num_sort_nodes	0.3989	window_function	0.4521	has_sort	0.6002
window_nodes	0.3815	num_scans	-0.4029	has_repartition	-0.4724
plan_max_depth	0.3571	num_exchanges	-0.3955	max_scan_columns	0.3619
num_exchanges	0.3313	input_num_files	0.3915	has_subquery	0.3244
num_joins	0.2739	has_subquery	-0.3501	has_distinct	-0.2464

Table 5.1: Top 5 Feature Loadings for the First Three Principal Components

5.2 Footprint Model performance

To verify how the model performed from a set 3674 entries of training data, cross fold validation($k=10$) was performed 5.2. The clustering was strong but uneven. There was a difference between balanced accuracy and accuracy for the fold but the score shows that the model performed well on the majority of the clusters. It was observed that one of the footprints obtained most of the data, causing the disparity between the accuracy and balanced accuracy as well as the Macro F1 from weighted F1. Clusters like 1,3,4,6 were predicted well, while clusters 11,12,13 and 15 performed poorly based on the recall value from the table 5.4.

Fold	Accuracy	Balanced Accuracy	Macro F1	Weighted F1	Top-3 Accuracy
1	0.9565	0.8477	0.8346	0.9673	0.9620
2	0.9484	0.8660	0.8518	0.9640	0.9592
3	0.9538	0.8251	0.8370	0.9720	0.9728
4	0.9429	0.8173	0.8199	0.9631	0.9674
5	0.9455	0.8316	0.8100	0.9583	0.9564
6	0.9401	0.8320	0.8343	0.9606	0.9591
7	0.9510	0.8822	0.8685	0.9668	0.9619
8	0.9428	0.8261	0.8056	0.9610	0.9591
9	0.9537	0.8625	0.8562	0.9699	0.9591
10	0.9237	0.8206	0.8214	0.9481	0.9482

Table 5.2: 10-Fold ($k = 17$ clusters, $\epsilon = 0.8$, min_samples = 10)

The CV average metrics summarizes the performance across the 10 folds. The standard deviation showed that the system was consistent across a majority of the clusters, but the small clusters were inconsistent. The data in table 5.3. The accuracy, weighted F1 and top 3 accuracy shows low variability across folds. The balanced accuracy and macro F1 varied more, hence the performance in the class was less uniform, and the minority clusters were challenging to separate reliably.

Metric	Mean	$\pm$ Std. Dev.
Accuracy	0.9458	0.0090
Balanced Accuracy	0.8411	0.0211
Macro F1	0.8339	0.0194
Weighted F1	0.9631	0.0065
Top-3 Accuracy	0.9605	0.0061

Table 5.3: Cross-validation mean metrics

Class	Precision	Recall	F1-score	Support
0	1.00	0.91	0.95	235
1	1.00	0.99	0.99	1287
2	1.00	0.82	0.90	65
3	1.00	0.99	0.99	293
4	1.00	0.99	1.00	702
5	1.00	0.95	0.97	129
6	1.00	0.96	0.98	275
7	1.00	0.83	0.91	187
8	0.99	0.95	0.97	148
9	0.99	0.96	0.97	99
10	0.96	0.89	0.93	28
11	1.00	0.70	0.82	33
12	1.00	0.79	0.88	63
13	0.96	0.81	0.88	27
14	1.00	0.94	0.97	34
15	1.00	0.26	0.42	68
16	0.00	0.00	0.00	1
Accuracy			0.95	3674
Macro avg	0.88	0.76	0.81	3674
Weighted avg	1.00	0.95	0.97	3674

Table 5.4: Out-of-fold classification report. Class 16 was noted to have had only one instance.

Classes 15 and 16 showed substantially lower performance than the other classes. Class 16 contained only one instance, making its scores unreliable, while class 15 had 68 instances but still showed notably lower recall and F1-score. These results indicate that the classifier had greater difficulty distinguishing these two classes.

5.3 Main Experiments

This section describes the methodology and results of each of the five experiments.

5.3.1 Experiment 1: Workload classification

A set of Spark workloads with different execution characteristics was used to evaluate whether the footprint classifier produced meaningful and consistent workload categories. The workloads were selected to represent different combinations of computational patterns, such as varying levels of CPU, memory, and shuffle activity. Each workload was executed through the pre-execution profiling process and assigned a footprint by the trained classifier.

The resulting footprint assignments were compared with the characteristics of the workloads. The experiment evaluates whether workloads with substantially different characteristics are assigned to different footprints, while repeated or sufficiently similar workloads are consistently assigned to the same footprint. This provides evidence that the footprint representation captures meaningful distinctions between workload types and can be used to associate incoming workloads with relevant historical execution data.

The jobs were classified based on SQL operations complexity. Window functions and multi-stage joins maximize memory usage per executor with limited parallelization, broadcast joins minimize memory but require small tables, `DISTINCT` operations trigger global shuffles that distribute load across cores at the cost of network I/O, and UDFs severely limit core utilization despite high memory overhead.

Test Case 1:

This job uses window functions, `AVG OVER` with `PARTITION BY` and row-based frames, combined with `INNER JOINS` and multi-stage `GROUP BY`. Each window function requires a partition of memory for computation, while `JOINS` build hash tables that consume memory proportional to the smaller join input.

Test Case 2:

This job has window functions with large frame sizes (100+ row `PRECEDING` windows) dramatically amplify memory pressure as each partition's data must remain in executor memory during window computation. In addition, multiple chained window functions force sequential, memory-intensive evaluations.

Test Case 3:

This job uses a small broadcast join, where the table fits in memory and is replicated to every executor and a single-stage `GROUP BY`, minimizing shuffle operations and allowing efficient parallel aggregation across cores with minimal memory overhead.

Test Case 4:

This job has `UNION DISTINCT` with filter pushdown. The `DISTINCT` operation forces a global `HashAggregate` deduplication across all executors, triggering a shuffle that redistributes data to remove duplicates, consuming memory for the hash table but benefiting from parallelization.

Experiment 1 Results:

Table 5.5 shows that jobs with different functionalities fell into the same or different class depending on their features. The feature vectors for these jobs can be found in appendix A (Figures A.1, A.2, A.3, and A.4).

Job ID	Footprint ID
1	13
2	8
3	9
4	8

Table 5.5: Mapping of jobs to their footprint IDs

Table 5.5 confirms that different jobs with different feature vectors get classified accordingly, either in to different footprints when they differ, or in the same footprint if they are similar enough (test cases 2 and 4 here).

5.3.2 Experiment 2: Iterative Resource Tuning

A Spark workload was created to evaluate how the resource tuner adapts resource requests over repeated executions. The workload was designed to create CPU and memory pressure by generating a configurable number of records with fixed-size binary payloads, persisting the resulting DataFrame in memory, and performing a Python UDF, aggregation, and ordering operation. This workload was called `test_job_heavy.py`, and can be found in appendix A.1.

The workload and input data were kept constant across all iterations to isolate the effect of resource tuning. First, the workload was executed without the tuner using the "large" configuration described in Section 2.4.2, which served as the baseline. The "large" baseline was picked because it was the mostly common selection in the studied environment, and was therefore deemed the most appropriate config to compare against. The allocated cores and memory history was extracted from their event logs and saved in a persistent volume on the cluster. Peak CPU and memory consumption were recorded using the cluster's Prometheus and Grafana monitoring stack.

The workload was then executed with the resource tuner enabled for 40 iterations. The same metrics were recorded for each execution to measure how the requested resource configuration changed as additional execution history became available. 40 iterations were selected to provide sufficient execution data for multiple tuning updates, showing the system capability when getting past the "cold start" phase, and to observe whether the configuration stabilized. The resulting resource allocations were compared with the baseline to quantify the reduction in resource requests over time.

Experiment 2 results:

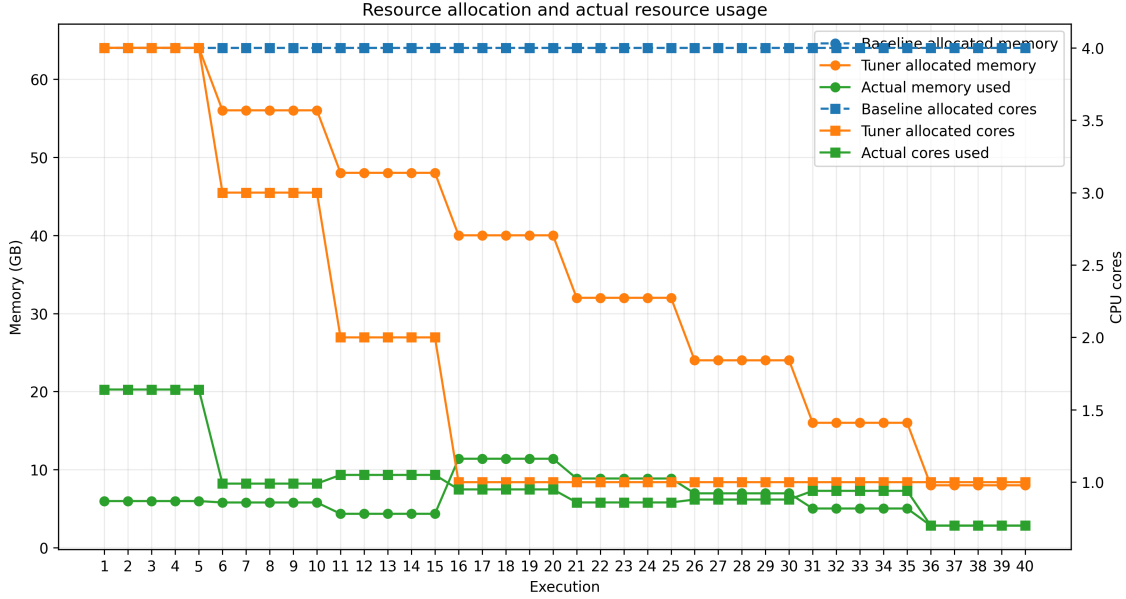


Figure 5.2: Graph of tuned memory and cores of `test_job_heavy.py`.

The graph 5.2 shows the baseline and tuned configurations (blue and orange respectively) after 40 runs, and actual resource usage observed in Grafana (green). True resource usage varied differently when both cores and memory got tuned at the same time independently. All statistics are per executor. This experiment demonstrated the ability of the tuner to approach the observed resource requirements of real applications of a given footprint over the course of several runs. The divergence of the orange (allocated by tuner) graph from the blue (allocated by baseline) line quantifies the potential for resource savings, and the convergence of the orange line towards the green line (observed need) shows the adaptive ability of the algorithm based on its history of data. The observed correlation between allocated and observed resources for both resource types is further discussed in the Discussion chapter.

Config (GB, Cores)	64, 4	56, 3	48, 2	40, 1	32, 1	24, 1	16, 1	8, 1
Avg exec time (s)	40.21	43.87	57.67	77.98	74.23	70,71	66.53	62.54

Table 5.6: Average execution times for the memory+cores tunings experiment 2 observed.

Table 5.6 shows that the average execution time did not linearly scale with allocated resources, and that the trend was that less resource sometimes reduced execution time. These observations are further discussed in the Discussion chapter.

5.3.3 Experiment 3: Simulated resource tuning of jobs of different footprints

A simulation environment was developed to replicate the gradual tuning process using the same memory tuning algorithm as the production system. Three different footprints were simulated, each consisting of three possible workload configurations. Random noise was added to the resource usage and execution time to approximate the variability of real workloads.

For each execution, one of the three configurations associated with the footprint was randomly selected as the ground truth. An execution was classified as successful if the memory allocated by the tuner was sufficient for the workload; otherwise, it resulted in an out-of-memory (OOM) failure.

Cores	Exec Time 4 GB (s)	Exec Time 8 GB (s)	Exec Time 16 GB (s)
1	52	68	95
2	39	48	62
3	36	42	49
4	34	39	44

Table 5.7: Execution times for footprint 0 at different core counts and memory configurations.

Cores	Exec Time 32 GB (s)	Exec Time 40 GB (s)	Exec Time 48 GB (s)
1	125	145	180
2	82	91	105
3	68	70	76
4	61	60	62

Table 5.8: Execution times for footprint 1 at different core counts and memory configurations.

Cores	Exec Time 64 GB (s)	Exec Time 80 GB (s)	Exec Time 108 GB (s)
1	220	250	300
2	145	160	185
3	115	120	130
4	100	102	105

Table 5.9: Execution times for footprint 2 at different core counts and memory configurations.

Each footprint was simulated for 50 runs. Every run started with the baseline configuration of 64 GB memory and 4 CPU cores, after which the tuner adjusted the memory allocation based on the observed outcomes.

Experiment 3 Results:

The following three figures 5.3, 5.4, and 5.5 will show plots from a simulation of memory being tuned:

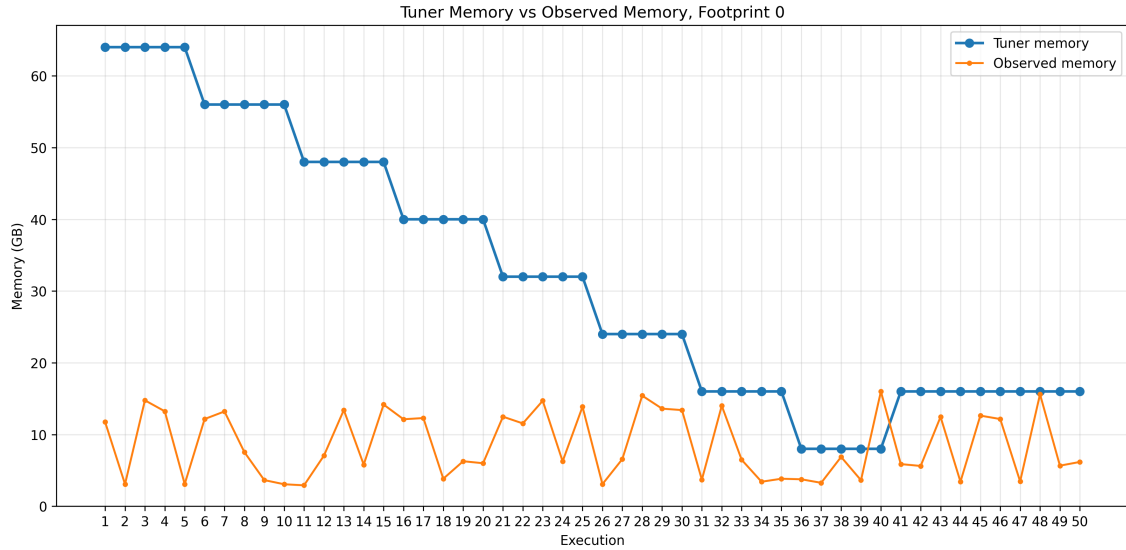


Figure 5.3: Graph of simulated memory, footprint 0, 50 iterations

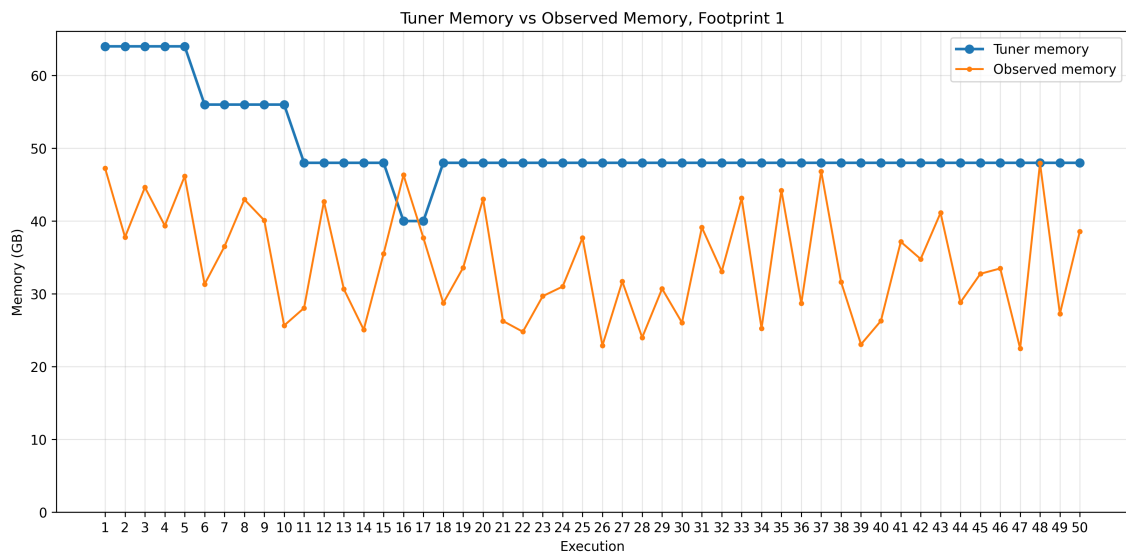


Figure 5.4: Graph of simulated memory, footprint 1, 50 iterations

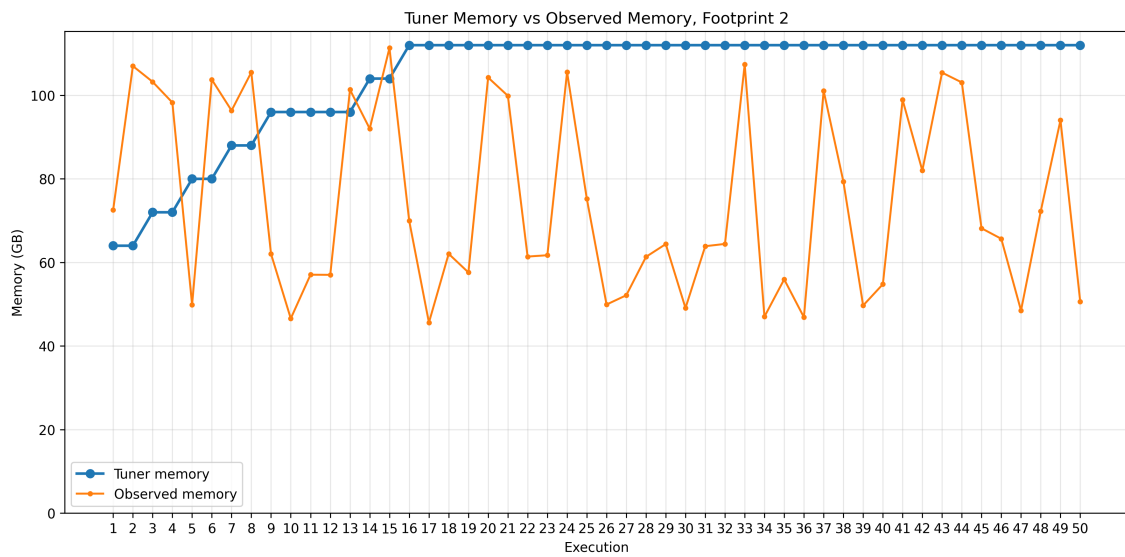


Figure 5.5: Graph of simulated memory, footprint 2, 50 iterations

Figures 5.3, 5.4, and 5.5 show the simulated memory level approaching the lowest level that consistently made all variable workloads within the same footprint class get sufficient memory, scaling both up or down depending on data history. This verifies that the tuner handles different footprints and resource profiles differently, and also that it works when there is variation in workloads demands within the same footprint.

The three tables show a simulation of tuned cores. The reason it wasn't plotted in a graph is because the core tuning criterion worked fundamentally different, with the cores-cost being the goal.

Run	Cores	Exec. time (s)	Avg. cost (4/3/2/1 Cores) (core-seconds)	Next Cores
1	4	46.13	184.52 / - / - / -	4
2	4	35.61	163.48 / - / - / -	4
3	4	44.40	168.19 / - / - / -	4
4	4	39.86	166.00 / - / - / -	4
5	4	34.69	160.55 / - / - / -	3
6	3	50.47	160.55 / 151.41 / - / -	3
7	3	48.50	160.55 / 148.45 / - / -	3
8	3	37.85	160.55 / 136.82 / - / -	3
9	3	34.85	160.55 / 128.75 / - / -	3
10	3	37.90	160.55 / 125.74 / - / -	2
11	2	35.85	160.55 / 125.74 / 71.70 / -	2
12	2	50.95	160.55 / 125.74 / 86.80 / -	2
13	2	57.35	160.55 / 125.74 / 96.10 / -	2
14	2	46.01	160.55 / 125.74 / 95.08 / -	2
15	2	66.49	160.55 / 125.74 / 102.66 / -	1
16	1	86.82	160.55 / 125.74 / 102.66 / 86.82	1
17	1	91.00	160.55 / 125.74 / 102.66 / 88.91	1
18	1	55.81	160.55 / 125.74 / 102.66 / 77.88	1
19	1	69.85	160.55 / 125.74 / 102.66 / 75.87	1
20	1	66.03	160.55 / 125.74 / 102.66 / 73.90	1
21	1	103.30	160.55 / 125.74 / 102.66 / 78.80	1
22	1	97.56	160.55 / 125.74 / 102.66 / 81.14	1
23	1	88.60	160.55 / 125.74 / 102.66 / 82.19	1
24	1	73.79	160.55 / 125.74 / 102.66 / 81.14	1
25	1	98.51	160.55 / 125.74 / 102.66 / 83.07	1
26	1	47.13	160.55 / 125.74 / 102.66 / 80.09	1
27	1	62.10	160.55 / 125.74 / 102.66 / 78.70	1
28	1	91.48	160.55 / 125.74 / 102.66 / 79.38	1
29	1	102.31	160.55 / 125.74 / 102.66 / 80.91	1
30	1	90.53	160.55 / 125.74 / 102.66 / 81.55	1
31	1	52.41	160.55 / 125.74 / 102.66 / 79.61	1
32	1	102.56	160.55 / 125.74 / 102.66 / 80.33	1
33	1	74.76	160.55 / 125.74 / 102.66 / 79.83	1
34	1	47.75	160.55 / 125.74 / 102.66 / 78.23	1
35	1	48.39	160.55 / 125.74 / 102.66 / 77.12	1
36	1	51.19	160.55 / 125.74 / 102.66 / 76.38	1
37	1	53.00	160.55 / 125.74 / 102.66 / 75.73	1
38	1	74.41	160.55 / 125.74 / 102.66 / 75.70	1
39	1	47.99	160.55 / 125.74 / 102.66 / 74.73	1
40	1	81.10	160.55 / 125.74 / 102.66 / 74.90	1
41	1	67.11	160.55 / 125.74 / 102.66 / 74.68	1
42	1	71.02	160.55 / 125.74 / 102.66 / 74.59	1
43	1	95.01	160.55 / 125.74 / 102.66 / 75.06	1
44	1	47.91	160.55 / 125.74 / 102.66 / 74.42	1
45	1	97.64	160.55 / 125.74 / 102.66 / 74.93	1
46	1	92.60	160.55 / 125.74 / 102.66 / 75.30	1
47	1	54.90	160.55 / 125.74 / 102.66 / 74.85	1
48	1	96.88	160.55 / 125.74 / 102.66 / 75.31	1
49	1	73.84	160.55 / 125.74 / 102.66 / 75.27	1
50	1	64.48	160.55 / 125.74 / 102.66 / 74.85	1

Table 5.10: Core tuner decisions and cumulative average cores-cost, footprint 0

Run	Cores	Exec. time (s)	Avg. cost (4/3/2/1 Cores) (core-seconds)	Next Cores
1	4	56.86	227.44 / - / - / -	4
2	4	65.74	245.20 / - / - / -	4
3	4	57.39	239.99 / - / - / -	4
4	4	55.98	235.97 / - / - / -	4
5	4	61.05	237.62 / - / - / -	3
6	3	71.47	237.62 / 214.41 / - / -	3
7	3	73.14	237.62 / 216.92 / - / -	3
8	3	75.06	237.62 / 219.67 / - / -	3
9	3	72.17	237.62 / 218.88 / - / -	3
10	3	69.20	237.62 / 216.62 / - / -	2
11	2	73.92	237.62 / 216.62 / 147.84 / -	2
12	2	99.31	237.62 / 216.62 / 173.23 / -	2
13	2	74.96	237.62 / 216.62 / 165.46 / -	2
14	2	81.76	237.62 / 216.62 / 164.98 / -	2
15	2	114.15	237.62 / 216.62 / 177.64 / -	1
16	1	134.75	237.62 / 216.62 / 177.64 / 134.75	1
17	1	136.02	237.62 / 216.62 / 177.64 / 135.38	1
18	1	121.36	237.62 / 216.62 / 177.64 / 130.71	1
19	1	151.64	237.62 / 216.62 / 177.64 / 135.94	1
20	1	185.26	237.62 / 216.62 / 177.64 / 145.81	1
21	1	120.98	237.62 / 216.62 / 177.64 / 141.67	1
22	1	117.26	237.62 / 216.62 / 177.64 / 138.18	1
23	1	135.82	237.62 / 216.62 / 177.64 / 137.89	1
24	1	157.27	237.62 / 216.62 / 177.64 / 140.04	1
25	1	155.32	237.62 / 216.62 / 177.64 / 141.57	1
26	1	137.48	237.62 / 216.62 / 177.64 / 141.20	1
27	1	135.66	237.62 / 216.62 / 177.64 / 140.73	1
28	1	124.64	237.62 / 216.62 / 177.64 / 139.50	1
29	1	135.06	237.62 / 216.62 / 177.64 / 139.18	1
30	1	137.13	237.62 / 216.62 / 177.64 / 139.04	1
31	1	153.27	237.62 / 216.62 / 177.64 / 139.93	1
32	1	158.26	237.62 / 216.62 / 177.64 / 141.01	1
33	1	179.52	237.62 / 216.62 / 177.64 / 143.15	1
34	1	136.72	237.62 / 216.62 / 177.64 / 142.81	1
35	1	164.19	237.62 / 216.62 / 177.64 / 143.88	1
36	1	147.44	237.62 / 216.62 / 177.64 / 144.05	1
37	1	181.12	237.62 / 216.62 / 177.64 / 145.74	1
38	1	114.50	237.62 / 216.62 / 177.64 / 144.38	1
39	1	114.20	237.62 / 216.62 / 177.64 / 143.12	1
40	1	136.04	237.62 / 216.62 / 177.64 / 142.84	1
41	1	183.40	237.62 / 216.62 / 177.64 / 144.40	1
42	1	185.67	237.62 / 216.62 / 177.64 / 145.93	1
43	1	195.65	237.62 / 216.62 / 177.64 / 147.70	1
44	1	120.35	237.62 / 216.62 / 177.64 / 146.76	1
45	1	149.98	237.62 / 216.62 / 177.64 / 146.87	1
46	1	157.44	237.62 / 216.62 / 177.64 / 147.21	1
47	1	128.03	237.62 / 216.62 / 177.64 / 146.61	1
48	1	164.64	237.62 / 216.62 / 177.64 / 147.15	1
49	1	115.81	237.62 / 216.62 / 177.64 / 146.23	1
50	1	156.00	237.62 / 216.62 / 177.64 / 146.51	1

Table 5.11: Core tuner decisions and cumulative average cores-cost, footprint 1

Run	Cores	Exec. time (s)	Avg. cost (4/3/2/1 Cores) (core-seconds)	Next Cores
1	4	80.34	321.36 / - / - / -	4
2	4	86.35	333.38 / - / - / -	4
3	4	87.53	338.96 / - / - / -	4
4	4	87.43	341.65 / - / - / -	4
5	4	107.79	359.55 / - / - / -	3
6	3	98.08	359.55 / 294.24 / - / -	3
7	3	99.91	359.55 / 296.99 / - / -	3
8	3	97.86	359.55 / 295.85 / - / -	3
9	3	129.80	359.55 / 319.24 / - / -	3
10	3	113.24	359.55 / 323.33 / - / -	2
11	2	154.67	359.55 / 323.33 / 309.34 / -	2
12	2	138.10	359.55 / 323.33 / 292.77 / -	2
13	2	155.61	359.55 / 323.33 / 298.92 / -	2
14	2	132.28	359.55 / 323.33 / 290.33 / -	2
15	2	149.69	359.55 / 323.33 / 292.14 / -	1
16	1	232.40	359.55 / 323.33 / 292.14 / 232.40	1
17	1	214.04	359.55 / 323.33 / 292.14 / 223.22	1
18	1	207.24	359.55 / 323.33 / 292.14 / 217.89	1
19	1	213.56	359.55 / 323.33 / 292.14 / 216.81	1
20	1	294.38	359.55 / 323.33 / 292.14 / 232.32	1
21	1	325.55	359.55 / 323.33 / 292.14 / 247.86	1
22	1	240.95	359.55 / 323.33 / 292.14 / 246.87	1
23	1	199.09	359.55 / 323.33 / 292.14 / 240.90	1
24	1	316.94	359.55 / 323.33 / 292.14 / 249.35	1
25	1	268.20	359.55 / 323.33 / 292.14 / 251.23	1
26	1	232.64	359.55 / 323.33 / 292.14 / 249.54	1
27	1	199.70	359.55 / 323.33 / 292.14 / 245.39	1
28	1	265.83	359.55 / 323.33 / 292.14 / 246.96	1
29	1	266.03	359.55 / 323.33 / 292.14 / 248.32	1
30	1	227.04	359.55 / 323.33 / 292.14 / 246.91	1
31	1	268.22	359.55 / 323.33 / 292.14 / 248.24	1
32	1	250.47	359.55 / 323.33 / 292.14 / 248.37	1
33	1	302.17	359.55 / 323.33 / 292.14 / 251.36	1
34	1	240.70	359.55 / 323.33 / 292.14 / 250.80	1
35	1	241.28	359.55 / 323.33 / 292.14 / 250.32	1
36	1	217.12	359.55 / 323.33 / 292.14 / 248.74	1
37	1	296.18	359.55 / 323.33 / 292.14 / 250.90	1
38	1	323.97	359.55 / 323.33 / 292.14 / 254.07	1
39	1	229.19	359.55 / 323.33 / 292.14 / 253.04	1
40	1	233.49	359.55 / 323.33 / 292.14 / 252.26	1
41	1	314.52	359.55 / 323.33 / 292.14 / 254.65	1
42	1	295.88	359.55 / 323.33 / 292.14 / 256.18	1
43	1	307.39	359.55 / 323.33 / 292.14 / 258.01	1
44	1	313.19	359.55 / 323.33 / 292.14 / 259.91	1
45	1	258.34	359.55 / 323.33 / 292.14 / 259.86	1
46	1	239.78	359.55 / 323.33 / 292.14 / 259.21	1
47	1	227.26	359.55 / 323.33 / 292.14 / 258.21	1
48	1	270.14	359.55 / 323.33 / 292.14 / 258.57	1
49	1	294.36	359.55 / 323.33 / 292.14 / 259.62	1
50	1	207.25	359.55 / 323.33 / 292.14 / 258.13	1

Table 5.12: Core tuner decisions and cumulative average cores-cost, footprint 2

Each of the simulated runs show the executor cores value being tuned based on what the average lowest cores-cost values is, as seen in the "average cores-cost per core level" column in tables 5.10, 5.11, and 5.12. This verifies that the core tuning algorithm works as intended.

5.3.4 Experiment 4: Tuning up due to OOM failures

A workload with a known memory requirement below the initial allocation was used to evaluate whether the tuner could identify and reduce an over-provisioned memory allocation. The workload was executed repeatedly at the initial memory level until the minimum number of observations required by the tuner had been collected. The Wilson confidence interval was then evaluated to determine whether its lower bound reached the reduction threshold. If the lower bound was at least 0.75, the tuner was expected to reduce the allocation by 8 GB. The subsequent execution was used to verify that the reduced allocation was selected by the tuner.

Experiment 4 Results:

```
"memory\_levels": {  
  "discarded": ["8"], ...  
  ...  
  "16": {  
    "observations": 5,  
    "successes": 5,  
    "failures": 0  
  },  
  "8": {  
    "observations": 8,  
    "successes": 6,  
    "failures": 2  
  }  
}
```

Figure 5.6: Excerpt from `state/fp_7.json` showing the accumulated statistics for the 8 GB and 16 GB memory levels.

```

{
  "end_reason": "oom",
  "assigned": {"memory": "8g", "cores": "1"},
  "execution_time": 50.61
},
{
  "end_reason": "oom",
  "assigned": {"memory": "8g", "cores": "1"},
  "execution_time": 50.71
},
{
  "end_reason": "success",
  "assigned": {"memory": "16g", "cores": "1"},
  "execution_time": 66.14
}

```

Figure 5.7: Last three entries from `history/fp_7.json`, showing two OOM failures at 8 GB followed by an execution at 16 GB.

The two listings together demonstrate the tuner’s response to OOM failures. The history in Figure 5.7 shows two OOM failures at the 8 GB level, while the accumulated state in Figure 5.6 recorded these as two failures from eight observations. The resulting Wilson interval had an upper bound below the 95% threshold, causing the 8 GB level to be discarded. The tuner consequently increased the allocation to 16 GB for the subsequent execution.

5.3.5 Experiment 5: Uncertainty aware tuning

A memory level with a high observed success rate was used to evaluate the tuner’s behaviour under statistical uncertainty. The workload was executed repeatedly to establish a reliable history at the current memory level. The number of observations was then varied to examine how the Wilson confidence interval changed as additional evidence was collected. The experiment evaluated whether the tuner remained at the current allocation while the confidence interval was inconclusive and reduced the allocation only once the lower Wilson bound reached the 0.75 threshold. This tests whether the tuner accounts for uncertainty rather than making decisions based solely on the observed success rate.

Experiment 5 Results:

```
"memory_levels": {  
  "discarded": [  
    "8"  
  ], ...  
  
  ...  
  "16": {  
    "observations": 11,  
    "successes": 10,  
    "failures": 1  
  },  
  "8": {  
    "observations": 7,  
    "successes": 5,  
    "failures": 2  
  }  
}
```

Figure 5.8: State of footprint 7 after discarding 8 GB.

Ten successful executions of the test job were performed, all of which succeeded, then the 11th one was the oom inducing job. The "discarded" field did not show 16 GB, indicating that the system did not react to the failure since the upper confidence bound did not drop below 95%. This experiment showed runs that don't decrease the upper bound of the confidence interval enough (see Algorithm 1 in section 4.4) allow the tuner to keep the configuration, and not change the level too aggressively on too few failure occurrences.

6

Discussion

6.1 Answering the Research Questions

RQ1: The results from figure 5.2 showed allocated memory dropping from 64 GB to 8 GB over the course of a 40 run cold start of `test_job_heavy`, thereby saving 56 GB per executor after the tuning stabilized. The memory tuner also managed to find the lowest consistently safe memory level for the simulated runs as seen in figures 5.3, 5.4, and 5.5, even if that was above the baseline. The cores tuner always stayed at 1 core after exploring regardless of workload, even for the CPU-heavier workloads that were simulated (see tables 5.10, 5.11, 5.12). Even more CPU demanding jobs that would trigger a move to 2 cores would validate the cores tuner further. All results were per executor, making the total savings for this job after the cold start compared to the baseline 560 GB and 30 cores since there were 10 instances (executors) per job. Adding in the 10 % memory overhead which pods use on top of Spark executors, the total memory savings increase to 616 GB of RAM per Spark job.

Memory utilization increased from around 8.5% ($5.98/(64 \times (1+0.1))$) in the baseline configuration, to 32.3% ($2.84/(8 \times (1+0.1))$) in the 8 GB configuration arrived at in figure 5.2. Cores utilization increased from around 41% ($1.64/4$) to stabilizing around 70-90% ($0.7-0.9/1$) in the 1 core configurations, depending on how much memory was allocated. The results show a sharp decline back to around 40% utilization in the 8 GB case, which is worth investigating further.

Looking at table 5.6, execution times started at around 40 seconds and ended at about 62 seconds as the configurations changed, meaning that these improvements came at a cost in execution time of around 55% ($40.21/62.54$). As configs went from 40 GB 1 core down to 8 GB 1 core, execution time sped up, which seemed unintuitive. However, other research corroborates these findings. Some plausible explanations are that there are less resources to manage in a cluster which can drastically improve data locality and resource management overheads potentially in Apache Spark.

RQ2: Uncertainty aware Wilson intervals were successfully implemented in the memory tuning algorithm, but uncertainty awareness is not used to tune cores. This is because the objective of cores tuning for this thesis isn't based on levels like the memory tuning, but rather continuous core-cost levels.

6.2 Method Limitations

The proposed system was subject to several limitations imposed by Apache Spark, the execution model adopted in this thesis, and the available mechanisms for pre-execution feature extraction.

6.2.1 Query Plan Feature Extraction Limitations

The proposed approach relied on extracting workload features from the Spark query plan prior to execution. This required the complete computation graph to be fully defined before any action was triggered, allowing Spark to generate the complete logical and physical execution plans. Consequently, the approach was limited to batch-style workloads where the entire computation could be represented before execution.

Another limitation of the chosen feature extraction strategy concerned Apache Spark’s Adaptive Query Execution (AQE). While pre-execution profiling extracted features from the initial physical query plan, Spark could dynamically modify this plan during execution through runtime optimizations such as join strategy selection and shuffle partition coalescing. This meant that the execution characteristics captured in event logs did not necessarily correspond to the query plan used to generate training data. This introduced an additional source of inconsistency between training data and deployment, potentially reducing the quality of the learned workload representations.

6.2.2 Performance Limitations of Interactive Spark Sessions

Interactive Spark sessions and batch-oriented job submissions differ fundamentally in their execution characteristics. Batch jobs, defined as immutable workloads, have fixed execution plans, enabling predictable job execution. In contrast, interactive Spark sessions are more exploratory in nature, where queries and transformations can change within the session. Due to this variability, interactive workloads were inherently more difficult to optimize using pre-execution techniques. Unpredictable execution behaviour, and therefore query plan, could change significantly within a single session. As a result, the method of this thesis only suited pre-defined batch workloads, and not interactive execution.

Furthermore, Spark sessions persisted across multiple tasks within a single application to reduce session creation overhead. While isolated Spark sessions would have enabled more accurate workload-specific tuning, session reuse was generally recommended to reduce initialization costs. Consequently, the proposed approach was expected to provide the greatest benefit for longer-running workloads, where the additional profiling and session initialization overhead could be amortized over the execution time.

6.3 Design Trade-offs

Several implementation decisions were made to balance predictive capability, implementation complexity, and applicability in a production environment. While these choices enabled the development of a functioning prototype, they also introduced a number of trade-offs that influenced the system’s capabilities.

6.3.1 Feature Trade-offs

The machine learning model was intentionally restricted to features that could be extracted entirely before execution from the Spark query plan and consistently available workload metadata. This enabled resource predictions to be produced before scheduling, but excluded potentially informative runtime metrics that would only become available during execution. Furthermore, the feature extraction pipeline is only compatible with workloads implemented using the Spark DataFrame and SQL APIs, since these expose structured query plans that can be inspected prior to execution. Applications implemented directly using lower-level RDD operations were therefore not fully supported. Although DataFrames and SQL based workloads represent the recommended and most common programming model in modern Spark applications, this nevertheless reduced the generality of the proposed system.

6.3.2 Tuning Algorithm Considerations

As shown in the results of experiment 2, figure 5.2, used memory and cores varied along with the given configuration. That means that in practice, there was no defined, hard memory or cores need that had to be respected for a given job. This observation is particularly interesting, because it implies that tuning a given job is more complex than finding a specific value for cores and memory that satisfies static resource requirements. It suggests that the tuning problem should be treated as more of a 2-dimensional search space of memory and cores pairs rather than independent tuning of memory and cores. Independent tuning did however produce improved results compared to the stated baseline, which makes it defensible as a starting point.

Another aspect to consider is about which historical observations should be considered, and the effects that would have on the exploration of tuning configurations. The memory tuning algorithm of this thesis permanently discarded unreliable memory configurations according to the confidence intervals. This meant that the tuner permanently stopped exploring memory below that level. Even if, for example, changing the number of cores would make a memory level reliable, that would not get the chance to be explored, which lead to the type of problem discussed in the last paragraph.

This mechanism could be affected by introducing a sliding window mechanism that "forgets" old history, and therefore makes permanently discarded configs available again. This could also improve cores config exploration, since a footprint settled on 1 core might start considering 2,3, and 4 cores again if that data gets forgotten,

possibly at a stage where memory has been tuned to other levels than when those core levels were explored last, allowing for wider exploration of configs. Another idea is to remove the discarding of memory levels completely, and allow the confidence intervals total control of the outcome. The price for this exploration would be paid in fails when the tuner would repeatedly try the same lower configuration level, but this would have the advantage of reacting to changing frequencies of workloads within a footprint.

6.3.3 Model Drift

Although the online tuning mechanism continuously updated resource recommendations, the underlying workload clustering model remained fixed after training. As workload types and therefore characteristics change over time, the learned cluster boundaries may gradually become less representative of the current workload distribution, reducing prediction quality.

To maintain effectiveness over longer deployment periods, the workload footprint model would therefore require periodic retraining and re-clustering using newly collected execution data. Determining suitable retraining intervals and detecting when model drift has become significant remain open design considerations.

6.3.4 Failure Handling

The implemented system did not perform automatic re-execution following failed workload executions. If a job failed because of incorrect resource allocation, the failure was added to the historical data, but no immediate corrective re-submission was attempted. Consequently, the system operated as an open-loop learning framework, where improvements were realised across successive workload executions rather than within a single execution cycle.

Incorporating automatic failure detection, resource adjustment, and job resubmission would allow failed jobs to be retried automatically within the same execution cycle. This would constitute an important step towards production readiness, but it fell outside the scope of the present work due to time constraints.

7

Conclusion

7.1 Conclusion

This thesis presented a predictive resource tuner for Apache Spark workloads running on a multi-node Kubernetes cluster. The objective has been to align allocated resources with the runtime requirements of applications through pre-execution prediction of resource needs, with the aim of improving cluster utilization, reducing job queue times, and improving performance stability.

The results demonstrate that the general concept of predicting resource requirements for batch-oriented workloads is promising, and yields real results in terms of resource savings. The current implementation is limited by feature consistency and the difficulty of accurately capturing runtime behaviour from pre-execution signals. In particular, mismatches between event-log-derived metrics and pre-execution features reduce the robustness of the learned models.

Despite these limitations, the approach shows potential in environments where workload types are well-defined and training data accurately reflects production usage patterns. With improved feature engineering and more controlled data collection strategies, as outlined in the Future Work section below, the proposed approach could form the basis for effective pre-execution resource allocation for Spark jobs in K8s data clusters.

7.2 Future Work

The findings and experience gained throughout this thesis suggest several areas where the proposed system could be improved or extended. The first idea is the design of a unified data collection pipeline. Instead of deriving training data from Spark event logs while extracting inference-time features through pre-execution query plan profiling, both could be generated using the same instrumentation and feature extraction mechanism. In that case, every feature row produced by the profiling could be added to a data store containing training data and used when retraining the model. This would eliminate any inconsistencies in features between training and inference data since they would both be produced in the same way, namely through the profiling. This would also allow all research effort to go toward obtaining the best candidate features possible from the profiling, which provides a much richer

selection than what was available in this thesis.

Another possible improvement is to extend the system architecture so that it could support automatic handling of failed executions through closed-loop feedback. This would allow unsuccessful jobs to be resubmitted immediately with adjusted resource configurations. Likewise, mechanisms for detecting the ML model drift described in Section 6.3.3 and periodically retraining the workload model from newly collected job feature data could be integrated to maintain prediction quality as workload distributions evolve over time.

While extending the system to support interactive Spark sessions may be desirable, the highly iterative and exploratory nature of notebook-based development presents challenges for pre-execution resource prediction. Since both the query plan and resource requirements may change between successive executions, interactive workloads are less suited to the current approach. Future development could therefore investigate broader workload support while retaining the pre-execution prediction approach.

The proposed scheduler could also be integrated more closely with existing Kubernetes resource management mechanisms. Whereas Apache Spark already supports horizontal scaling through Dynamic Resource Allocation, this work primarily addressed pre-execution vertical resource allocation. Future iterations could therefore investigate how workload-aware resource prediction may complement Kubernetes Vertical Pod Autoscaling, combining proactive resource estimation with adaptive runtime scaling to achieve more efficient cluster utilization.

Finally, other tree-based machine learning models could be evaluated as alternatives to Gradient Boosting Trees. Comparing different tree-based approaches could determine whether another model provides improved workload classification or resource prediction. Additional query plan and runtime features could also be evaluated to provide a richer representation of workload characteristics.

Bibliography

- [1] A. Badshah, A. Daud, R. Alharbey, A. Banjar, A. Bukhari, and B. Alshe-maimri, “Big data applications: Overview, challenges and future,” *Artificial Intelligence Review*, vol. 57, 2024. DOI: 10.1007/s10462-024-10938-5.
- [2] S. Tang, B. He, C. Yu, Y. Li, and K. Li, “A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, pp. 71–91, 2018. DOI: 10.1109/tkde.2020.2975652.
- [3] Y. Wu et al., “Towards resource efficiency: Practical insights into large-scale spark workloads at bytedance,” *Proc. VLDB Endow.*, vol. 17, pp. 3759–3771, 2024. DOI: 10.14778/3685800.3685804.
- [4] R. Kessler, S. Volpert, and S. Wesner, “Towards improving resource allocation for multi-tenant hpc systems: An exploratory hpc cluster utilization case study,” in *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2024, pp. 66–75. DOI: 10.1109/clusterworkshops61563.2024.00019.
- [5] S. Malik, M. Tahir, M. Sardaraz, and A. Alourani, “A resource utilization prediction model for cloud data centers using evolutionary algorithms and machine learning techniques,” *Applied Sciences*, 2022. DOI: 10.3390/app12042160.
- [6] C. Delimitrou and C. Kozyrakis, “Paragon: Qos-aware scheduling for heterogeneous datacenters,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: ACM, 2013, pp. 77–88. DOI: 10.1145/2451116.2451125.
- [7] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-efficient and Qos-aware cluster management,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: ACM, 2014, pp. 127–144. DOI: 10.1145/2541940.2541941.
- [8] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, “Adaptive AI-based autoscaling for Kubernetes,” *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 599–608, 2020.
- [9] Kubernetes Authors, *Kubernetes components and architecture*, <https://kubernetes.io/docs/concepts/architecture/>, Accessed: 2026-07-02, 2026.
- [10] Kubernetes Authors, *Kubernetes scheduler*, <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>, Accessed: 2026-07-02, 2026.

- [11] Kubernetes Authors, *Resource management for pods and containers*, <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>, Accessed: 2026-07-02, 2026.
- [12] Kubernetes Authors, *Kubernetes overview*, <https://kubernetes.io/docs/concepts/overview/>, Accessed: 2026-07-02, 2026.
- [13] Apache Software Foundation, *Spark cluster mode overview*, <https://spark.apache.org/docs/latest/cluster-overview.html>, Accessed: 2026-07-02, 2026.
- [14] Apache Software Foundation, *Spark job scheduling*, <https://spark.apache.org/docs/latest/job-scheduling.html>, Accessed: 2026-07-02, 2026.
- [15] Apache Software Foundation, *Spark rdd programming guide*, <https://spark.apache.org/docs/latest/rdd-programming-guide.html>, Accessed: 2026-07-03, 2026.
- [16] M. Zaharia et al., “Resilient distributed datasets: A fault-tolerant abstraction for In-Memory Cluster Computing,” (*UCB/EECS-2011-82*) by the EECS Department, University of California, Berkeley, 2011.
- [17] M. Armbrust, Y. Huai, C. Liang, R. Xin, and M. Zaharia. “Deep dive into spark sql’s catalyst optimizer.” [Online]. Available: <https://www.databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>.
- [18] D. Aronovich. “Understanding wide vs. narrow transformations in apache spark: Why it matters for performance.” [Online]. Available: <https://bigdataperformance.substack.com/p/transformations-vs-actions-in-apache>.
- [19] J. Tringham. “Spark concepts simplified: Lazy evaluation.” [Online]. Available: https://medium.com/@john_tringham/spark-concepts-simplified-lazy-evaluation-d398891e0568.
- [20] Apache Software Foundation, *Spark sql, dataframes and datasets guide*, <https://spark.apache.org/docs/latest/sql-programming-guide.html>, Accessed: 2026-07-06, 2026.
- [21] Apache Spark Project, *Spark sql internals*, <https://spark.apache.org/docs/latest/sql-internals.html>, Accessed: 2026-07-06, 2026.
- [22] Apache Spark Project, *Tuning spark*, <https://spark.apache.org/docs/latest/tuning.html>, Accessed: 2026-07-06, 2026.
- [23] Apache Spark Project, *Spark monitoring and instrumentation*, <https://spark.apache.org/docs/latest/monitoring.html>, Accessed: 2026-07-06, 2026.
- [24] A. Satheesh. “Spark’s dynamic resource allocation, does it really help?” Medium article, accessed 2026-05-26. [Online]. Available: <https://medium.com/@anands282/sparks-dynamic-resource-allocation-does-it-really-help-1a81fb085bc9>.
- [25] Kubernetes Authors, *Horizontal pod autoscaling*, <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>, Accessed: 2026-07-06, 2026.
- [26] Apache Spark Project, *Running spark on kubernetes*, <https://spark.apache.org/docs/latest/running-on-kubernetes.html>, Accessed: 2026-07-06, 2026.

- [27] D. Kaemaks, L. Versluis, B. K. Ozkan, and J. Decouchant, “Balancing Fairness and Performance in Multi-User Spark Workloads with Dynamic Scheduling,” *Proceedings of the 2025 ACM Symposium on Cloud Computing*, pp. 867–880, 2025.
- [28] c3 ai. “Tree-based models. ”[Online]. Available: <https://c3.ai/glossary/data-science/tree-based-models/>.
- [29] GeeksforGeeks. “Gradient boosting in machine learning.” Accessed: 2026-07-03, Accessed: Jul. 3, 2026. [Online]. Available: <https://www.geeksforgeeks.org/machine-learning/ml-gradient-boosting/>.
- [30] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
- [31] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.
- [32] K. Pearson, “On lines and planes of closest fit to systems of points in space,” *Philosophical Magazine*, vol. 2, no. 11, pp. 559–572, 1901.
- [33] L. van der Maaten and G. Hinton, “Visualizing data using t-sne,” *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.
- [34] A. Zainab, A. Ghayeb, H. Abu-Rub, S. S. Refaat, and O. Bouhali, “Distributed tree-based machine learning for short-term load forecasting with Apache Spark,” *IEEE Access*, vol. 9, Apr. 2021.
- [35] Y. Li et al., “Towards general and efficient online tuning for spark,” *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3570–3583, 2023. DOI: 10.14778/3611540.3611548.
- [36] J. M. K. R. M. Rao2, “Scheduling of jobs allocation for apache spark using kubernetes for efficient execution of big data application,” *Intelligent Engineering & Systems*, Apr. 2023.
- [37] R. Sen, A. Roy, and A. Jindal, *Predictive price-performance optimization for serverless query processing*, Dec. 2021. DOI: 10.48550/arXiv.2112.08572. arXiv: 2112.08572 [cs.DB]. [Online]. Available: <https://arxiv.org/abs/2112.08572>.
- [38] H. Dong et al., “Towards Performance and Energy Aware Kubernetes Scheduler,” *ACM SIGENERGY Energy Informatics Review*, vol. 5, no. 2, pp. 69–75, 2025.
- [39] Qiu, H. and Mao, W. and Wang, C. and Franke, H. and Youssef, A. and Kalbarczyk, Z. T. and Baar, T. and Iyer, R. K., “AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems,” *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 387–402, 2023.
- [40] Apache Software Foundation, *Spark sql, dataframes and datasets guide*, <https://spark.apache.org/docs/latest/sql-programming-guide.html>, Accessed: 2026-06-17, 2026.
- [41] N. Ahmed, A. Barczak, T. Sunjak, and M. A. Rashid, “A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using hibenach,” *Journal of Big Data*, Aug. 2020.

- [42] A. Gounaris, G. Kougka, R. Tous, C. Montes, and J. Torres, “Dynamic configuration of partitioning in spark applications,” *IEEE Transactions on Parallel and Distributed Systems*, Jul. 2017.
- [43] J. Bae et al., “Jointly optimizing task granularity and concurrency for in-memory mapreduce frameworks,” in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017.
- [44] Q.-l. He, F. Zhang, G. Bian, W. Zhang, and Z. Li, “Research on performance optimization of spark distributed computing platform,” *Computers, Materials & Continua*, 2024.
- [45] A. Tzenetopoulos, M. Gazzetti, D. Masouros, C. Pinto, S. Xydis, and D. Soudris, “Disaggregated rdds: Extending and analyzing apache spark for memory disaggregated infrastructures,” in *2024 IEEE International Conference on Cloud Engineering (IC2E)*, Sep. 2024.
- [46] S. Raj, D. Ramesh, and K. K. Sethi, “A spark-based apriori algorithm with reduced shuffle overhead,” *The Journal of Supercomputing*, Mar. 2020.
- [47] A. Modi et al., “New query optimization techniques in the spark engine of azure synapse,” *Proceedings of the VLDB Endowment*, Dec. 2021.
- [48] S. Teryoshkin, D. Nasonov, and N. Butakov, “Parwop: Optimization for window functions partitioning,” in *2025 IEEE International Conference on Data Mining Workshops (ICDMW)*, Nov. 2025.
- [49] K. Chen, L. Zhao, Z. Ke, D. Xianjun, and C. Xudong, “A spark join algorithm based on memory monitoring and batch processing,” in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, Nov. 2018.
- [50] J. Choi, J. Lee, J.-S. Kim, and J. Lee, “Optimization techniques for a distributed in-memory computing platform by leveraging ssd,” *Applied Sciences*, Sep. 2021.
- [51] L. Lu et al., “Lifetime-based memory management for distributed data processing systems,” *Proceedings of the VLDB Endowment*, Feb. 2016.
- [52] K. Jiang, S. Du, F. Zhao, Y. Huang, C. Li, and Y. Luo, “Effective data management strategy and rdd weight cache replacement strategy in spark,” *Computer Communications*, Jul. 2022.

A

Test job code listings

A.1 test_job_heavy.py

```
import sys
import time
from pyspark.sql import SparkSession, functions as F
from pyspark.sql.types import StructType, StructField, IntegerType,
    BinaryType
from pyspark.storagelevel import StorageLevel
from app_extract import add_result_to_json

# Hardcoded tuning knobs (adjust here)
ROWS = 200000
PARTITIONS = 200
PAYLOAD_KB = 64
NUM_KEYS = 1000
SLEEP_AFTER_PERSIST = 15

def make_payload(i):
    # deterministic key distribution
    key = int(i % NUM_KEYS)
    payload = bytearray(b'x') * (PAYLOAD_KB * 1024)
    return (key, bytes(payload))

# --- TRANSFORMS ---
spark = SparkSession.builder.appName('test_job_heavy').getOrCreate()
sc = spark.sparkContext

schema = StructType([
    StructField('key', IntegerType(), False),
    StructField('payload', BinaryType(), True),
])

# Create RDD and DataFrame (transforms only)
rdd = sc.parallelize(range(ROWS), PARTITIONS).map(make_payload)
df = spark.createDataFrame(rdd, schema=schema)

# Define derived aggregation (length per key) using a Python UDF
```

```
payload_len_udf = F.udf(lambda b: len(b) if b is not None else 0,
    ↪ IntegerType())
lengths_df = (
    df.withColumn(
        'payload_len',
        payload_len_udf(F.col('payload'))
    )
    .groupBy('key')
    .agg(
        F.sum('payload_len').alias('total_bytes')
    )
)

# Also provide a small top-N ordering transform (not executed yet)
top10_df = lengths_df.orderBy(F.desc('total_bytes')).limit(10)

if len(sys.argv) > 1 and sys.argv[1] == "profile":
    # No actions here.
    # Only inspect the final transformation's physical plan.
    add_result_to_json(top10_df, "/work")

# --- ACTIONS ---
elif len(sys.argv) > 1 and sys.argv[1] == "execute":
    # materialize and show results
    df.persist(StorageLevel.MEMORY_ONLY)
    df.count()
    time.sleep(SLEEP_AFTER_PERSIST)
    # show summary and top keys
    print("\n-- Aggregated lengths (per key) --")
    lengths_df.show(truncate=False)
    print("\n-- Top 10 keys by aggregated bytes --")
    top10_df.show(truncate=False)
    df.unpersist()

else:
    print("\nError: specify 'profile' or 'execute'\n")

spark.stop()
```

A.2 test_job_oom.py

```
import sys
import time
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType
from app_extract import add_result_to_json

spark = SparkSession.builder.appName("test_job").getOrCreate()
```

```
# OOM test workload
ALLOCATE_GB = 40
CHUNK_MB = 256

def oom_partition(_):
    chunks = []
    for _ in range(ALLOCATE_GB * 1024 // CHUNK_MB):
        chunks.append(
            bytearray(CHUNK_MB * 1024 * 1024)
        )
    # Keep the allocations alive.
    time.sleep(60)
    return iter([1])

# Transforms
rdd = (
    spark.sparkContext
    .parallelize([0], 1)
    .mapPartitions(oom_partition)
)

schema = StructType([
    StructField("value", IntegerType(), False)
])

df = spark.createDataFrame(
    rdd.map(lambda x: (x,)),
    schema=schema
)

# Profile / Execute
if len(sys.argv) > 1 and sys.argv[1] == "profile":
    add_result_to_json(df, "/work")

elif len(sys.argv) > 1 and sys.argv[1] == "execute":
    # Actions
    df.count()

else:
    print("\nError: specify 'profile' or 'execute'\n")

spark.stop()
```

```
{
  "input_num_files": 1,
  "plan_max_depth": 15,
  "max_scan_columns": 0,
  "num_scans": 0,
  "num_sort_nodes": 3,
  "num_window_nodes": 0,
  "num_joins": 1,
  "num_exchanges": 4,
  "has_subquery": 0,
  "has_window_function": 0,
  "has_sort": 1,
  "has_repartition": 0,
  "has_distinct": 0,
  "has_cache": 0
}
```

Figure A.1: Test Job 1 Query plan

```
{
  "input_num_files": 1,
  "plan_max_depth": 4,
  "max_scan_columns": 0,
  "num_scans": 0,
  "num_sort_nodes": 0,
  "num_window_nodes": 0,
  "num_joins": 0,
  "num_exchanges": 0,
  "has_subquery": 0,
  "has_window_function": 0,
  "has_sort": 0,
  "has_repartition": 0,
  "has_distinct": 0,
  "has_cache": 0
}
```

Figure A.2: Test Job 2 Query plan

```
{
  "input_num_files": 1,
  "plan_max_depth": 14,
  "max_scan_columns": 0,
  "num_scans": 0,
  "num_sort_nodes": 3,
  "num_window_nodes": 0,
  "num_joins": 1,
  "num_exchanges": 4,
  "has_subquery": 0,
  "has_window_function": 0,
  "has_sort": 1,
  "has_repartition": 0,
  "has_distinct": 1,
  "has_cache": 0
}
```

Figure A.3: Test Job 3 Query Plan

```
{
  "input_num_files": 1,
  "plan_max_depth": 10,
  "max_scan_columns": 0,
  "num_scans": 0,
  "num_sort_nodes": 0,
  "num_window_nodes": 0,
  "num_joins": 0,
  "num_exchanges": 2,
  "has_subquery": 0,
  "has_window_function": 0,
  "has_sort": 0,
  "has_repartition": 0,
  "has_distinct": 0,
  "has_cache": 0
}
```

Figure A.4: Test Job 4 Query plan