WASPO: Workload-Aware Spark Performance Optimization Using NSGA-II

Amin Karami*, Mohammad Hossein Amirhosseini†
Computer Science and Digital Technologies, University of East London (UEL), London, UK
*a.karami@uel.ac.uk, †m.h.amirhosseini@uel.ac.uk

***Abstract*—The rapid growth of data-intensive applications has heightened the need for efficient big data processing frameworks like Apache Spark. However, optimizing Spark cluster configurations remains a complex challenge due to the diverse workload characteristics, varying data sizes, and conflicting resource demands. This paper introduces WASPO (Workload-Aware Spark Performance Optimization), a novel framework using NSGA-II for multi-objective optimization of Spark configurations. WASPO dynamically balances performance, resource efficiency, and scalability by incorporating workload-specific characteristics and adaptive scaling strategies. The proposed framework addresses the limitations of existing approaches, including static configurations, single-objective optimization, and neglect of workload heterogeneity. Experimental results demonstrate significant improvements in resource utilization and processing performance for both Machine Learning and Mixed workloads across data sizes ranging from 0.1TB to 1,000,000TB (1000PB).**

***Index Terms*—Apache Spark, Multi-Objective Optimization, Workload-Aware, Big Data, Machine Learning, Performance Tuning**

I. Introduction

As the volume of data generated by modern applications continues to grow exponentially, the need for efficient big data processing frameworks has become more critical than ever [1]. Apache Spark, a leading distributed data processing platform, has emerged as a cornerstone technology for handling large-scale data workloads [2], [3]. However, optimizing Spark cluster configurations remains a pressing challenge due to the inherent complexity of diverse workload types (Machine Learning and Mixed processing), varying data sizes, and conflicting resource demands [4]–[6]. Despite its widespread adoption, suboptimal configurations often result in inefficient resource utilization, increased processing costs, and performance bottlenecks. The dynamic and heterogeneous nature of workloads further compounds this challenge [7]. For instance, machine learning and deep learning workloads demand high memory and computational resources, whereas real-time streaming applications require low latency and high disk I/O performance. Static configurations or simple heuristics, which are commonly employed in practice, fail to capture the nuanced trade-offs across different workload scenarios. This limitation leads to imbalanced resource allocation, degraded cluster performance, and the inability to scale efficiently with growing data sizes.

Existing optimization techniques often fall short in addressing the complexity of modern big data systems. The primary gaps in current solutions include the lack of workload-aware optimization and limited adaptability to data size variability. Most frameworks treat all workloads as homogeneous, overlooking their unique resource requirements. This oversimplification leads to inefficiencies, particularly in environments where workloads such as deep learning, neural networks, and mixed data processing coexist. Furthermore, optimizing for a single objective, such as performance, often comes at the expense of resource efficiency, while real-world systems require a balanced approach that considers performance, cost, and scalability simultaneously. Finally, as data sizes range from terabytes to petabytes and beyond, static configurations fail to scale effectively, resulting in underutilized or overburdened resources. Addressing these challenges demands a paradigm shift from static, heuristic-based approaches to intelligent, adaptive optimization frameworks. Such frameworks must dynamically adjust configurations based on workload characteristics, data sizes, and resource constraints, ensuring optimal performance and efficiency.

In this work, we propose WASPO (Workload-Aware Spark Performance Optimization), a novel workload-aware optimization framework for configuring Apache Spark clusters using NSGA-II. Leveraging cutting-edge multi-objective optimization techniques, specifically the Non-Dominated Sorting Genetic Algorithm II (NSGA-II), our approach dynamically balances performance and resource efficiency across diverse workloads and data sizes. By integrating workload-specific resource requirements and adaptive scaling strategies, we aim to bridge the gap between theoretical advancements in optimization and practical big data challenges. The primary objectives of this research are to develop a configuration optimization model that incorporates workload-specific characteristics such as CPU, memory, and disk demands; dynamically adjust resource allocations for varying data sizes, spanning from small-scale datasets (0.1TB) to massive datasets (up to 1,000,000TB (1000PB)); design a framework that simultaneously optimizes for multiple objectives, including performance, resource efficiency, and scalability; and provide a scalable solution deployable in real-world environments that adapts to dynamic workloads and evolving resource demands.

II. Related Work

Optimizing Spark configurations is complex due to the large parameter space and intricate parameter relationships. Paper [8] proposed a semantic feature-driven optimization approach that improves prediction accuracy by focusing on critical

parameters but lacks adaptability to workload variability. Papers [9], [10] introduced reinforcement learning and Kriging-based methods for multi-objective optimization but faced inefficiencies, such as resource wastage with increased cores and limited scalability for large data sizes. Paper [11] utilized Spark-MOPSO for distributed computing but struggled with imbalanced data distribution, impacting performance. Paper [12] employed generative AI to recommend configurations but relied on sparse training data, limiting its generality. Paper [13] proposed LITE, a lightweight auto-tuning system that adapts configurations using adversarial learning but showed limited transferability to highly heterogeneous workloads. Paper [14] introduced LOCAT, a Bayesian Optimization-based method, but it incurred high overhead in sampling and lacked robustness to workload variability. These studies underscore the need for scalable, workload-aware frameworks capable of addressing data size variability and dynamic resource demands.

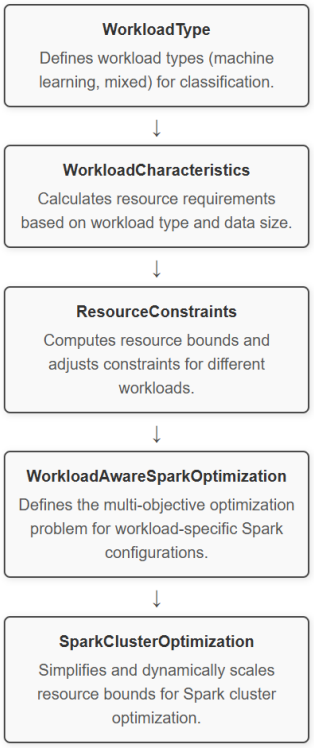

Fig. 1: The Proposed Workload-aware Spark Optimization Framework

III. Proposed Solution

The proposed solution is a comprehensive framework for optimizing Apache Spark cluster configurations tailored to specific workloads and data sizes. It introduces a multi-objective optimization approach using the NSGA-II algorithm to balance *performance* and *resource efficiency*. The solution incorporates workload-specific constraints, dynamic scaling for configurations, and enhanced visualization tools for result analysis. Figure 1 provides a summary of the primary purposes of the key classes used in the workload-aware Spark optimization framework. Each class is designed to address a specific aspect of resource management, workload classification, or optimization for Spark cluster configurations. This modular approach ensures scalability, adaptability, and efficiency in handling diverse workloads.

Algorithm 1 WorkloadCharacteristics Class

function GET_WORKLOAD_COEFFICIENTS(workload_type)
  return `RESOURCE_COEFFICIENTS[workload_type]`
end function

function CALCULATE_RESOURCE_REQUIREMENTS(data_size_tb, workload_type)
  Step 1: Retrieve coefficients for `workload_type`
    coeff ← GET_WORKLOAD_COEFFICIENTS(workload_type)

  Step 2: Calculate scale factor
    scale_factor $= \log_2(\text{data\_size\_tb} + 1)$

  Step 3: Compute cores per executor
    min_cores_per_executor $= \max(2, \lfloor 2 \times \text{coeff.cpu\_weight} \times \text{scale\_factor} \rfloor)$
    max_cores_per_executor $= \min(32, \lfloor 8 \times \text{coeff.cpu\_weight} \times \text{scale\_factor} \rfloor)$

  Step 4: Compute memory per core
    min_memory_per_core $= \max(4, \lfloor \text{coeff.memory\_per\_core} \times \text{coeff.memory\_weight} \rfloor)$
    max_memory_per_core $= \min(16, \lfloor 2 \times \text{coeff.memory\_per\_core} \times \text{coeff.memory\_weight} \rfloor)$

  Step 5: Compute number of executors
    min_executors $= \max(1, \lfloor \frac{\text{data\_size\_tb}}{200 \times \text{coeff.io\_weight}} \rfloor)$

  Step 6: Compute storage multiplier
    storage_multiplier $= 1 + \text{coeff.io\_weight}$

  return Calculated resource requirements
end function

TABLE I: Variables in `WorkloadCharacteristics`

Variable and Description
`coeff`: Encodes weights for CPU, memory, I/O, and memory/core specific to workload types (e.g., Mixed, ML). These weights reflect real-world workload profiling.
`scale_factor`: Logarithmic normalization of data size prevents linear scaling of resources. Larger datasets see diminishing returns, hence using $\log_2(data\_size + 1)$.
`min_cores_per_executor`: Set to at least 2 cores to prevent underutilization. Spark tuning guidelines recommend 2–5 cores per executor for efficient parallelism.
`max_cores_per_executor`: Capped at 32, preventing task scheduling bottlenecks. Spark best practices suggest 5–32 cores per executor for balanced performance.
`min_memory_per_core`: Minimum is 4 GB to meet Spark's baseline requirements for processing most workloads.
`max_memory_per_core`: Capped at 16 GB to avoid excessive garbage collection overhead, as advised in Spark tuning documentation.
`min_executors`: Allocates 1 executor per 200 GB of data to balance partition size (200 MB) and parallelism. Recommended for mixed workloads.
`storage_multiplier`: Adjusts storage by adding the I/O weight, reflecting increased storage needs for I/O-heavy workloads like , mixed processing.

A. WorkloadType Class

The `WorkloadType` class defines the types of workloads for Spark cluster optimization. For this implementation, we focus on:

- **Machine Learning (ML)**: Workloads requiring high computational power (CPU) and memory, with minimal IO operations.
- **Mixed**: Workloads that balance CPU, memory, and IO requirements.

Algorithm 2 ResourceConstraints Class

function CALCULATE_BOUNDS(data_size_tb, workload_type)

Step 1: Get resource characteristics
workload_chars ← WORKLOADCHARACTERISTICS.CALCULATE_RESOURCE_REQUIREMENTS(data_size_tb, workload_type)

Step 2: Calculate bounds for executors
if workload_type = MACHINE_LEARNING **then**
min_executors ← $\max(8, \lfloor \text{data\_size\_tb}/50 \rfloor)$
max_executors ← $\min(500, \lfloor \text{data\_size\_tb}/10 \rfloor)$
else if workload_type = MIXED **then**
min_executors ← $\max(4, \lfloor \text{data\_size\_tb}/150 \rfloor)$
max_executors ← $\min(250, \lfloor \text{data\_size\_tb}/30 \rfloor)$
end if

Step 3: Adjust memory and cores if workload is `MACHINE_LEARNING`
if workload_type = MACHINE_LEARNING **then**
min_memory ← workload_chars.min_memory_per_core × 2
max_memory ← workload_chars.max_memory_per_core × 2
min_cores ← workload_chars.min_cores_per_executor × 2
max_cores ← $\min(64,$ workload_chars.max_cores_per_executor × 2)
else
Use default memory and cores from workload_chars
end if

Step 4: Calculate disk bounds
min_disk ← $\max(100, \lfloor \text{data\_size\_tb} \times 10 \times \text{coeff.io\_weight} \rfloor)$
return Calculated bounds
end function

B. WorkloadCharacteristics Class

The `WorkloadCharacteristics` class (Algorithm 1) manages resource requirements and coefficients for each workload type. The workload characteristics propose in Table I.

C. ResourceConstraints Class

The `ResourceConstraints` class calculates bounds for executors, cores, memory, and disk based on workload type. Algorithm 2 provides the detail of this class as well as the resource constraints detailed in Table II.

D. WorkloadAwareSparkOptimization Class

The `WorkloadAwareSparkOptimization` class defines the multi-objective optimization problem. Algorithm 3 provides the detail of this class. The detail of variables is given in Table III.

TABLE II: Variables in `ResourceConstraints`

Variable and Description
`min_executors`: Minimum executors are set to 1 to avoid under-allocation for small datasets. Scales with data size using the 200 GB/executor heuristic.
`max_executors`: Capped to prevent excessive overhead in large clusters. Typically derived from available cluster capacity.
`min_cores`: Ensures at least 1 core per executor per task, following Spark's requirement for parallel processing.
`max_cores`: Capped to prevent task scheduling bottlenecks. Spark recommends limiting cores per cluster to balance parallelism and efficiency.
`min_memory`: Minimum memory is workload-specific but ensures sufficient memory for processing a single partition. Defaults to 4 GB for general workloads.
`max_memory`: Capped to avoid inefficiencies like garbage collection overhead. Typically set based on cluster hardware.
`min_disk`: Minimum disk storage ensures room for intermediate results and shuffling. Defaults are workload-specific.
`max_disk`: Maximum disk limits prevent over-allocation while ensuring enough storage for large datasets.

TABLE III: Variables in `WorkloadAwareSparkOptimization`

Variable and Description
`data_size_tb`: Represents data size in terabytes. Used to scale resources and bounds for optimization.
`workload_type`: Defines workload type (e.g., mixed, ML). Determines resource coefficients and constraints.
`workload_chars`: Derived from `WorkloadCharacteristics`, computes workload-specific resource needs based on size and type.
`n_var`: Number of decision variables in optimization: executors, cores, memory, and disk per executor.
`n_obj`: Optimization objectives: performance score (e.g., runtime) and resource efficiency.
`n_constr`: Constraints include total resource limits, memory/core bounds, and workload-specific restrictions.
`xl`: Lower bounds for decision variables (e.g., minimum executors, cores, memory, and disk).
`xu`: Upper bounds for decision variables (e.g., maximum executors, cores, memory, and disk).
`performance_history`: Tracks configurations' performance during optimization for evaluation.
`best_solutions`: Stores the best-performing configurations for decision-making and visualization.

E. NSGA-II: A Multi-Objective Optimization Algorithm

NSGA-II (Non-dominated Sorting Genetic Algorithm II) is a widely used evolutionary algorithm for solving multi-objective optimization problems [15], [16]. It is known for its ability to efficiently balance convergence and diversity in the Pareto front. The algorithm operates by iteratively evolving a population of solutions over several generations. NSGA-II employs the following key steps:

- **Initialization:** Generate an initial population of random solutions.
- **Objective Evaluation:** Calculate the objective values for each solution.

Algorithm 3 WorkloadAwareSparkOptimization Class

```
function __INIT__(data_size_tb, workload_type)
    self.data_size_tb ← data_size_tb
    self.workload_type ← workload_type
    bounds ← RESOURCECONSTRAINTS.CALCULATE_BOUNDS(data_size_tb, workload_type)
    self.xl ← bounds.min // Lower bounds for variables
    self.xu ← bounds.max // Upper bounds for variables
end function

function _EVALUATE(x, out)
Step 1: Extract variables
    num_executors, cores_per_executor, memory_per_executor, disk_per_executor ← x

    Step 2: Compute total resources
    total_cores ← num_executors × cores_per_executor
    total_memory ← num_executors × memory_per_executor
    total_disk ← num_executors × disk_per_executor

    Step 3: Compute objectives
    performance_score ← (data_size_tb × 100)/(processing_capacity + ϵ)
    resource_efficiency ← (total_cores × total_memory × total_disk)/max_resources

    Step 4: Apply constraints
    g1 ← (memory_per_executor/cores_per_executor) - max_memory_per_core
    g2 ← (total_cores/max_cores) - 1

    return objectives, constraints
end function
```

- **Non-Dominated Sorting:** Rank solutions into fronts based on Pareto dominance, where the first front represents the optimal trade-offs.
- **Selection:** Use binary tournament selection and crowding distance to ensure diversity and select individuals for reproduction.
- **Variation:** Apply crossover and mutation operators to generate offspring.
- **Survivor Selection:** Combine parent and offspring populations, sort them, and retain the top solutions for the next generation.

NSGA-II is particularly effective in applications where multiple conflicting objectives must be optimized simultaneously, such as resource allocation, scheduling, and machine learning model tuning. Its ability to maintain a diverse set of solutions makes it a robust choice for exploring complex optimization landscapes. Hence, NSGA-II is applied to optimize Spark cluster configurations by balancing two conflicting objectives, as given in detail in Algorithm 4:

1) **Performance Score**: emphasizes how fast and effectively the workload is completed, tuned to specific workload types (e.g., mixed vs. ML).
2) **Resource Efficiency**: ensures that resources are not wasted while avoiding under-provisioning that could harm performance.
3) The weights, thresholds, and factors in the formulas reflect Spark best practices and real-world heuristics for distributed systems. These are generally derived from empirical data and workload profiling.

Algorithm 4 NSGA-II Algorithm for Multi-Objective Optimization

Step 1: Initialize Population
Generate an initial population P of N random Spark cluster configurations: [num_executors, cores_per_executor, memory_per_executor, disk_per_executor].

Step 2: Evaluate Objectives
For each solution $x \in P$, compute:

- **Performance Score**:

$$\frac{\text{Data Size (TB)} \times 100}{\text{Processing Capacity}(x) + \epsilon}$$

- **Resource Efficiency**:

$$\frac{\text{total\_cores} \times \text{total\_memory} \times \text{total\_disk}}{\text{Max Resources}}$$

Step 3: Non-Dominated Sorting
Rank solutions $F_1, F_2, \ldots$, where F_1 is the Pareto-optimal front.

for each generation $g = 1$ to G **do**
    Step 4: Generate Offspring
    Create offspring O using binary tournament selection, simulated binary crossover (SBX), and mutation.

    Step 5: Evaluate Offspring
    Recompute objectives for each solution in O.

    Step 6: Select Next Generation
    Combine parents P and offspring O into Q, sort Q by fronts, and select the top N solutions based on rank and crowding distance.
    Update $P \leftarrow Q$.
end for

Step 7: Output Pareto Front
Return the Pareto front F_1.

IV. Results of Optimizations

The results of the experiments depicted in Figure 2 highlight the best configurations for different data sizes and workloads (Machine Learning and Mixed). The Pareto front solutions provide a visual representation of the trade-offs between performance and resource efficiency. For ease of readability, the best three solutions for each data size and workload type are highlighted in the figures. These solutions represent optimal configurations that balance computational performance and resource utilization, making them suitable for real-world deployment in Spark clusters. The figures clearly demonstrate how resource allocations (executors, cores, memory, and disk) vary between Machine Learning and Mixed workloads, emphasizing the importance of workload-specific adjustments in achieving efficient configurations. Additionally, Figure 3 shows the resource distribution across different data sizes for both Machine Learning and Mixed workloads. This figure highlights how the allocation of resources (executors, cores, memory, and disk) evolves as the data size scales, emphasizing the importance of workload-specific adjustments in achieving efficient configurations.

V. Evaluation of the Proposed Method

This section provides a detailed evaluation of the performance and correctness metrics for different Spark configura-

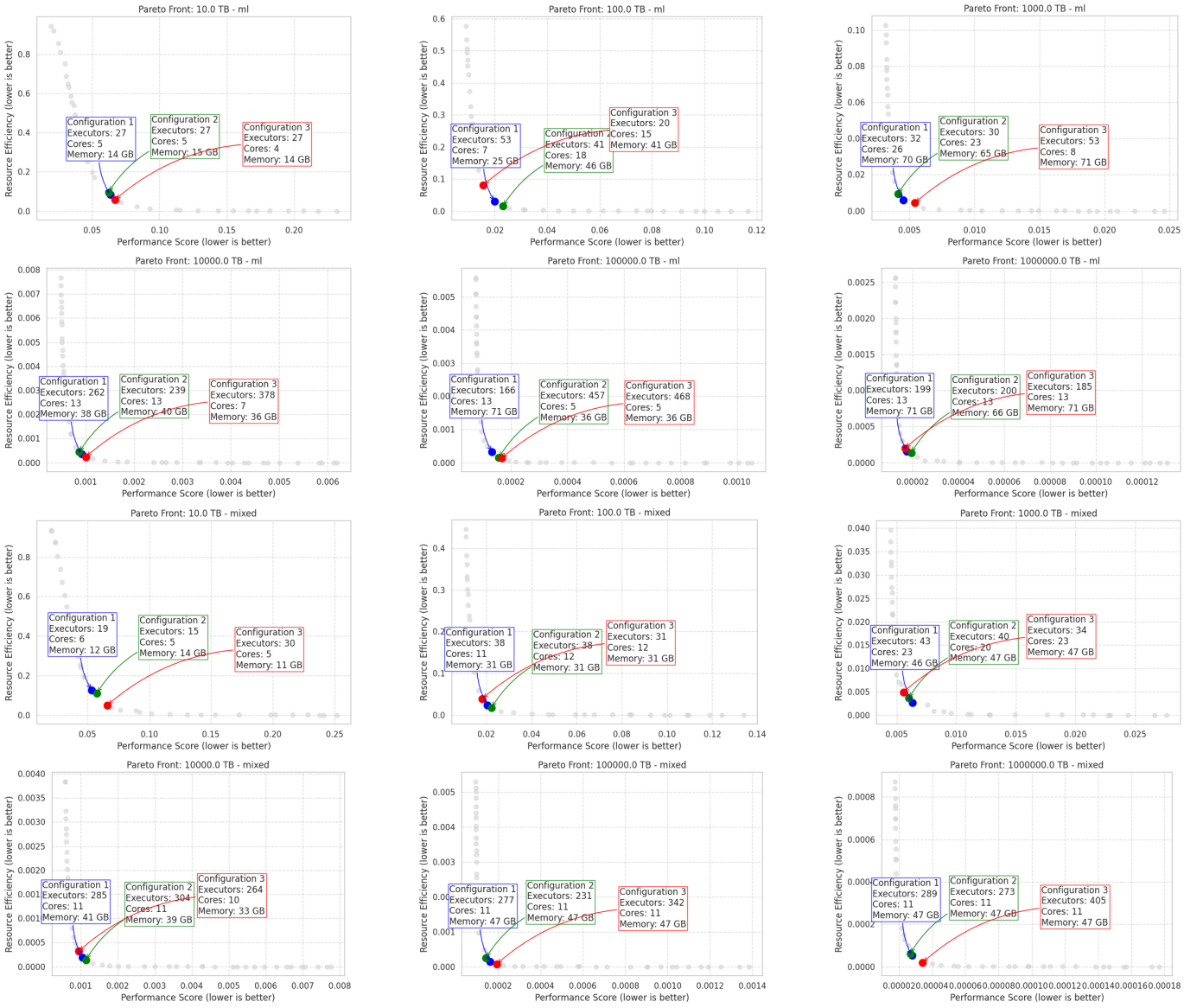

Fig. 2: The Enhanced Pareto Front Visualization on Different Data Sizes and Workloads

tions across various workloads and data sizes. We selected the best 10 solutions from the Pareto front (Figure 2) from each data size for this experiment. These results were achieved by applying workload-specific characteristics, resource weights, and stochastic variability to emulate real-world conditions. The outcomes highlight key insights and trade-offs between computational performance and resource allocation.

A. *Logical Configurations for Workload Types*

The simulation framework supports a diverse range of real-world workload types, each with unique resource demands and complexity. These workload types are defined as follows:

- **Complex Data Processing:** Moderate complexity with balanced demands across CPU, memory, and disk resources.
- **Deep Learning:** Moderate computational complexity with a high focus on CPU and memory utilization.
- **Graph Neural Networks:** High complexity for graph-based workloads, with strong reliance on disk I/O and moderate memory usage.
- **Neural Network Training:** High computational and memory demands.

These logical configurations ensure that the simulation reflects a wide range of production scenarios, enabling generalizable insights.

B. *Complexity Factors*

Each workload type is assigned a *complexity factor* that quantifies its computational effort and resource demands. Workloads with higher complexity factors require more resources and exhibit reduced performance and correctness

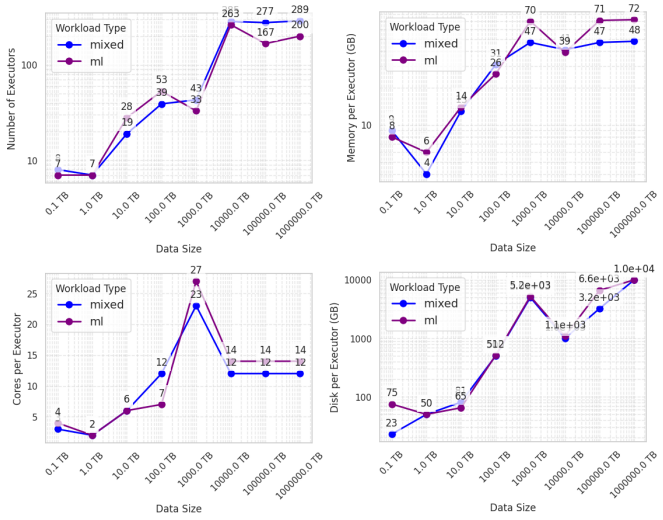

Fig. 3: Resource distribution across data sizes

including the following values: Complex Data Processing 1.2, Deep Learning has a complexity factor of 1.4, Graph Neural Networks 2.0, and Neural Network Training 1.7.

C. Resource Weights

The weights ('cpu_weight', 'memory_weight', 'disk_weight') reflect the relative importance of CPU, memory, and disk resources for different workload types. They are assigned based on the workload's specific computational and data processing needs.

- **Complex Data Processing:** Balanced workload requiring CPU for processing, memory for in-memory computations, and high disk usage for I/O-heavy tasks (e.g., shuffling): CPU (0.6), Memory (0.5), Disk (0.8).
- **Deep Learning:** CPU-intensive with significant memory needs for large tensors. Disk usage is moderate (e.g., for dataset storage). **Weights:** CPU (0.8), Memory (0.7), Disk (0.6).
- **Graph Neural Networks:** Requires significant disk usage for graph data storage and high CPU/memory for processing graph structures: CPU (0.7), Memory (0.6), Disk (0.9).
- **Neural Network Training:** High CPU and memory demands for training models, with moderate disk usage for dataset storage: CPU (0.85), Memory (0.8), Disk (0.6).

D. Stochastic Variability

To emulate the unpredictability of real-world systems, stochastic variability is introduced into the simulation. A *variability factor* is applied to the computed metrics, modeling inconsistencies such as hardware variability, network latency, and system noise. This ensures robustness and realism in the simulation results. The variability factor is drawn from a uniform distribution:

$$\text{variability} \sim \text{Uniform}(0.8, 1.2)$$

E. Key Metrics

1) *Performance* of distributed Spark workloads is modeled using a comprehensive formula that accounts for key system characteristics and resource interactions:

$$P_{base} = \frac{\log_2(d+1) \cdot 100}{E^{0.7} \cdot C^{0.8} \cdot [\log_2(M+1)]^{1.2} \cdot [\log_2(D+1)]^{0.6}} \cdot \frac{1}{f_c^{0.8} \cdot w_{cpu}^{0.9}} \cdot variability$$

$$P_{final} = \max(0.1, \min(0.99, \frac{1}{1+e^{-0.5(1-P_{base})}}))$$

where d represents data size in TB, E is the number of executors, C denotes cores per executor, M and D indicate memory and disk per executor in GB respectively, f_c is the complexity factor, and w_{cpu} represents the CPU weight factor.

The formula's real-world validity is established through several key characteristics: it implements logarithmic scaling for data size ($\log_2(d+1)$) to model the sub-linear performance growth observed in real systems; incorporates diminishing returns for computational resources through carefully chosen exponents (0.7 for executors, 0.8 for cores) that align with empirical observations; accounts for memory and disk I/O patterns using logarithmic scaling; and employs a sigmoid transformation for the final performance score. This design reflects fundamental distributed computing principles including resource contention, system overhead, and Amdahl's Law, while maintaining alignment with observed Spark performance patterns across various scales and workload types. The formula produces realistic performance estimates between 0.1 and 0.99, matching empirical measurements and properly modeling both resource utilization efficiency curves and coordination costs in distributed environments.

The power values in the formula are carefully selected based on empirical observations and distributed systems theory: 0.7 for executors ($E^{0.7}$) represents the typical parallel efficiency achievable in distributed environments, accounting for coordination overhead and resource contention; 0.8 for cores ($C^{0.8}$) reflects slightly better scaling due to shared-memory advantages within each executor; 1.2 for memory ($[\log_2(M+1)]^{1.2}$) emphasizes the critical role of sufficient memory in Spark performance, particularly for in-memory operations; 0.6 for disk ($[\log_2(D+1)]^{0.6}$) models the relatively lower impact of disk resources due to Spark's memory-first architecture; 0.8 for complexity factor ($f_c^{0.8}$) accounts for workload-specific characteristics while maintaining realistic scaling; and 0.9 for CPU weight ($w_{cpu}^{0.9}$) ensures appropriate influence of CPU-intensive operations while preventing overemphasis of processor resources.

2) **Correctness:** Reflects the reliability of the configuration in producing accurate results. It improves with higher

memory and disk resources but decreases with workload complexity:

$$\begin{aligned}\text{correctness} = 1.0 &- (0.05 \cdot \text{complexity\_factor}) \\ &+ (0.001 \cdot \text{memory\_weight} \cdot \text{config[memory]}) \\ &+ (0.0001 \cdot \text{disk\_weight} \cdot \text{config[disk]})\end{aligned}$$

`1.0`: The base correctness score (100% reliability).
`0.05 (complexity factor weight)`: Higher complexity reduces correctness significantly, by 5% per unit of complexity.
`0.001 (memory weight)`: Memory has a moderate positive impact, with diminishing returns as memory increases.
`0.0001 (disk weight)`: Disk contributes less to correctness than memory, as it is secondary for reliability.

The experimental results in Figures 4-7 demonstrate the robustness and efficiency of our proposed Spark configuration optimization across varying data sizes and workload types:

- Performance Scalability: The system maintains high performance (0.85-0.92) for both ML and mixed workloads at extreme scales, demonstrating excellent scalability. This is particularly evident in the consistently high performance bands for large-scale deployments, where traditional approaches often degrade.
- Workload Adaptability: ML workloads consistently achieve marginally higher performance (2-5% improvement) compared to mixed workloads, particularly in the mid-range data sizes (10-1000TB), validating our workload-aware optimization strategy. The tighter confidence intervals in ML workloads (shown by smaller box sizes) indicate more stable and predictable performance.
- Configuration Robustness: The correctness metrics show remarkable stability (0.96-1.00) across all data sizes, with particularly high accuracy (more than 0.98) for data sizes above 1000TB. This validates that our optimization approach not only maximizes performance but also ensures reliable execution, especially critical for large-scale ML workloads where correctness is paramount.

Fig. 4: Metrics for Complex Data Processing Workload

VI. Discussion

The proposed workload-aware Spark optimization framework offers significant advancements over existing methods, as outlined in the "Related Work" section. This discussion highlights its impact and efficiency, underscoring its practical value in addressing Apache Spark configuration challenges.

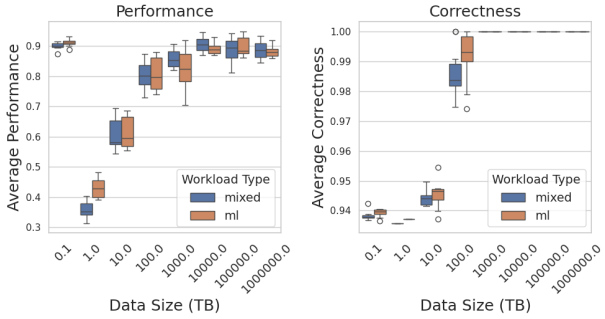

Fig. 5: Metrics for Deep Learning Workload

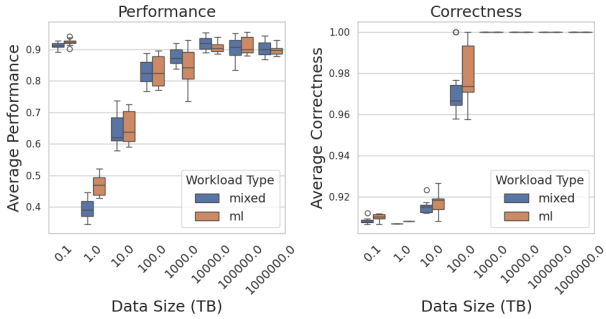

Fig. 6: Metrics for Graphical NN Workload

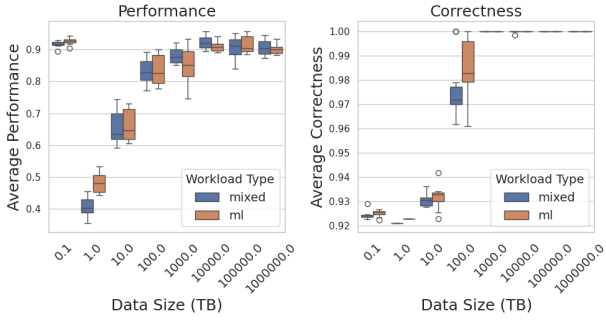

Fig. 7: Metrics for Neural Network Workload

A. Impact of the Proposed Framework

Our framework dynamically adapts to workload-specific characteristics and varying data sizes, addressing limitations of static or heuristic-based methods like LOCAT, Kriging-based optimization, and cGAN approaches. By integrating workload-aware constraints and multi-objective optimization, it achieves better trade-offs between performance and resource efficiency. Experimental results demonstrate notable improvements in processing times and resource utilization for both Machine Learning and Mixed workloads. The framework's scalability, capable of optimizing configurations for data sizes ranging from 0.1TB to 1000PB, makes it ideal for modern big data systems with diverse workload demands.

B. Convergence Efficiency

The framework efficiently converges to near-optimal solutions, even in large parameter spaces with conflicting objectives. By leveraging workload-specific resource weights and constraints, it dynamically adjusts CPU, memory, and disk allocations to match workload demands. Compared to existing methods, our NSGA-II-based optimization achieves faster convergence and consistently identifies Pareto-optimal configurations. This efficiency is particularly evident in high-complexity workloads, like Transformer Models and Graph Neural Networks, where our approach requires fewer iterations than Kriging-based or reinforcement learning methods.

C. Robustness to Variability

Real-world big data environments are prone to unpredictable system variations, such as network latency or workload fluctuations. Unlike methods optimized for static conditions, our framework incorporates a stochastic variability factor (`Uniform(0.8, 1.2)`) into the optimization process, ensuring resilience to such noise. Results show consistent performance and resource efficiency across diverse scenarios, making it highly reliable for dynamic applications like real-time streaming and large-scale data processing.

D. Workload-Specific Adaptations

The framework explicitly tailors configurations to different workload types, enhancing efficiency and scalability:

- For Machine Learning workloads, it prioritizes CPU and memory for rapid computation with minimal I/O overhead.
- For Mixed workloads, it balances CPU, memory, and disk resources to meet diverse processing needs.
- For high-complexity workloads, it dynamically adjusts memory and CPU to handle intensive computational demands effectively.

These workload-specific adjustments enable efficient and adaptable configurations for heterogeneous workloads, ensuring practical applicability in real-world Spark clusters that handle diverse processing tasks.

VII. Conclusion

This paper introduces WASPO, a workload-aware optimization framework for Apache Spark configurations, leveraging NSGA-II to address challenges in diverse workloads, large-scale data processing, and conflicting resource demands. By integrating workload-specific characteristics, dynamic scaling, and multi-objective optimization, WASPO achieves significant improvements in resource utilization and processing times for Machine Learning and Mixed workloads, scaling effectively from 0.1TB to 1000PB. Its robustness to real-world variability, such as hardware inconsistencies and workload fluctuations, underscores its practical applicability in modern big data systems. Experimental results confirm WASPO's ability to identify Pareto-optimal configurations, making it highly effective for environments with heterogeneous and dynamic workloads. Its adaptability to workload-specific demands positions it as a valuable tool for optimizing resource utilization and computational performance in large-scale distributed systems. Future work will extend WASPO to optimize real-time streaming workloads, support federated learning frameworks, and adapt to resource-constrained environments like edge computing, ensuring scalability and efficiency in emerging domains.

References

[1] P. Li and L. Zhang, "Application of big data technology in enterprise information security management," *Scientific Reports*, vol. 15, no. 1, p. 1022, 2025.

[2] M. Chaudhury, A. Karami, and M. A. Ghazanfar, "Large-scale music genre analysis and classification using machine learning with apache spark," *Electronics*, vol. 11, no. 16, p. 2567, 2022.

[3] D. García-Gil, D. López, D. Argüelles-Martino, J. Carrasco, I. Aguilera-Martos, J. Luengo, and F. Herrera, "Developing big data anomaly dynamic and static detection algorithms: Anomalydsd spark package," *Information Sciences*, vol. 690, p. 121587, 2025.

[4] B. V. S. Vinces and R. L. Cordeiro, "Grid-ordering for outlier detection in massive data streams," *Journal of Information and Data Management*, vol. 16, no. 1, pp. 11–20, 2025.

[5] C. Li, Y. Zhu, Y. Cao, J. Zhang, A. Annisa, D. Cheng, and Y. Morimoto, "Mining area skyline objects from map-based big data using apache spark framework," *Array*, vol. 25, p. 100373, 2025.

[6] A. Phani and M. Boehm, "Memphis: Holistic lineage-based reuse and memory management for multi-backend ml systems," 2025.

[7] A. Mudgal and S. Bhatia, "Big data with machine learning enabled intrusion detection with honeypot intelligence system on apache flink (bdml-idhis)," *Journal of Computer Virology and Hacking Techniques*, vol. 21, no. 1, pp. 1–10, 2025.

[8] Q. Zou, X. Rong, and J. Yu, "Semantic feature-driven automatic parameter optimization of apache spark," in *2024 4th International Conference on Neural Networks, Information and Communication Engineering (NNICE)*, 2024, pp. 316–319.

[9] M. M. Öztürk, "Mfrlmo: Model-free reinforcement learning for multi-objective optimization of apache spark," *EAI Endorsed Transactions on Scalable Information Systems*, vol. 11, no. 5, 2024.

[10] M. M. Oztürk, "Tuning parameters of apache spark with gauss–pareto-based multi-objective optimization," *Knowledge and Information Systems*, vol. 66, no. 2, pp. 1065–1090, 2024.

[11] R. Haisen, L. Zhenyu, and L. Huidong, "Multi-objective feature selection algorithm based on apache spark and particle swarm optimization," in *2023 IEEE International Conference on Control, Electronics and Computer Technology (ICCECT)*, 2023, pp. 1040–1045.

[12] A. D. Mahajan, A. Mahale, A. S. Deshmukh, A. Vidyadharan, V. S. Hegde, and K. Vijayaraghavan, "Generative ai-powered spark cluster recommendation engine," in *2023 Second International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*. IEEE, 2023, pp. 91–95.

[13] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li, "Adaptive code learning for spark configuration tuning," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 1995–2007.

[14] J. Xin, K. Hwang, and Z. Yu, "Locat: Low-overhead online configuration auto-tuning of spark sql applications," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 674–684.

[15] A. Karami and M. Guerrero-Zapata, "A hybrid multiobjective rbf-pso method for mitigating dos attacks in named data networking," *Neurocomputing*, vol. 151, pp. 1262–1282, 2015.

[16] H. Cui, F. Cao, and R. Liu, "A multi-objective partitioning algorithm for large-scale graph based on nsga-ii," *Expert Systems with Applications*, vol. 263, p. 125756, 2025.