

TECHNICAL PREDICTIVE ANALYSIS OF BIG DATA FOR APACHE SPARK-BASED RESOURCE PLANNING

Bakhshali Bakhtiyarov, Aynur Jabiyeva, Ulkar Musavi

Azerbaijan State Oil and Industry University, Department of Instrumentation Engineering, Baku, Azerbaijan

Abstract. The large database analysis is a substantial form of research that is presently expanding immensely in current technologies, and it also affects numerous aspects of life. Powerful, scalable, and customizable platforms are required to win the challenges in the field and increase effectiveness of analysing the data. Apache Spark is one of the most trending high-performance computing engines to process big data and thus deliver a revolutionary approach to data science and engineering. Big data analytics involve Apache Spark and its application is increasing at lightning speed both in academic and business spheres. It has established itself in the field of data analytics as it can support various kinds of loads within a single architecture. The development of Apache Spark continues with new developments of data analysis, hence bringing it to the notice of researchers and practitioners as a high utility tool to solve the problem of big data. The Apache Spark is a big data processing system which has proven effective in numerous applications of analysing big data; this paper examines the working process of Apache Spark, its advantages and area of application and provides an examination of the future outlook of this platform.

Keywords: big data, data evaluation, spark engine, network analysis, resilient distributed datasets, cluster computing

TECHNICZNA ANALIZA PREDYKCYJNA DUŻYCH ZBIORÓW DANYCH DO PLANOWANIA ZASOBÓW W OPARCIU O APACHE SPARK

Streszczenie. Analiza dużych zbiorów danych stanowi istotną formę badań, która obecnie przeżywa ogromny rozkwit dzięki najnowszym technologiom i ma wpływ na wiele aspektów życia. Aby sprostać wyzwaniom w tej dziedzinie i zwiększyć efektywność analizy danych, potrzebne są wydajne, skalowalne i konfigurowalne platformy. Apache Spark jest jednym z najpopularniejszych silników obliczeniowych o wysokiej wydajności, służących do przetwarzania dużych zbiorów danych, a tym samym zapewniającym rewolucyjne podejście do nauki o danych i inżynierii danych. Analiza dużych zbiorów danych opiera się na Apache Spark, a jego zastosowanie rośnie w błyskawicznym tempie zarówno w środowisku akademickim, jak i biznesowym. Rozwiązanie to ugruntowało swoją pozycję w dziedzinie analizy danych, ponieważ umożliwia obsługę różnego rodzaju obciążeń w ramach jednej architektury. Rozwój Apache Spark postępuje wraz z nowymi osiągnięciami w dziedzinie analizy danych, dzięki czemu rozwiązanie to zyskuje na znaczeniu wśród naukowców i praktyków jako niezwykle użyteczne narzędzie do rozwiązywania problemów związanych z dużymi zbiorami danych. Apache Spark to system przetwarzania dużych zbiorów danych, który sprawdził się w licznych zastosowaniach związanych z analizą dużych zbiorów danych; niniejszy artykuł analizuje proces działania Apache Spark, jego zalety i obszary zastosowań, a także przedstawia prognozy dotyczące przyszłości tej platformy.

Słowa kluczowe: big data, ocena danych, silnik spark, analiza sieci, odporne rozproszone zestawy danych, klastry obliczeniowe

Introduction

The data analytics field has already become one of the most actively evolving research directions in the contemporary information technologies. The growing amount, speed, and diversity of the data produced by modern digital systems demand computational frameworks which can support large volumes of data with high efficiency. Large-scale data processing systems in practice are required to achieve a complex workload with high computational efficiency in many real-world applications including financial analytics, social media analysis, industrial monitoring, scientific computing and so forth.

Apache Spark is one of the most popular distributed computing models implemented in data analytics on a large scale [4]. The system offers a single processing engine which enables a variety of data processing programs such as batch processing, machine learning, graph analytics, and stream processing on a single computational platform. Spark provides a model of in-memory execution, which is considerably superior to the disk-based processing models, which makes it a favourite platform in the world of big data analytics applications.

Ever since its release in the AMPLab of the University of California, Berkeley in 2009 and its later move to the Apache Software Foundation in 2013 [15], Apache Spark has quickly grown into a fully-fledged ecosystem with both academic and industrial backing. It incorporates a number of high-level libraries, such as MLlib to perform machine learning [7], Spark SQL to perform structured data processing [14], GraphX to perform graph analytics [3], and Spark Streaming to perform real-time data processing [2]. The Resilient Distributed Dataset (RDD) model provided in Spark provides a fault-tolerant distributed computation and efficient data sharing across computations [1].

Although Apache Spark has been widely implemented in the large-scale processing of data environment, effective resource planning of Spark workloads has been a problem.

Specifically, when system resources are not estimated well in advance of running, large-scale Spark jobs can have unreliable execution times, run very large in memory, and have very expensive shuffle operations. The challenges are especially important in the production settings, where the lack of memory could be a crucial factor in the performance decrease through disk spillage or even executor memory overflow that can cause job failure.

Published studies on Apache Spark are mainly related to system architecture, application use case or performance measurement of distributed cluster. Nonetheless, there exists comparatively less research coverage on the predictive analytical models, which are able to estimate resource estimates and execution conduct of Spark jobs prior to execution. The fact that such predictive mechanisms are absent renders the effective planning of the cluster resources by the system architects hard and avoidance of the performance bottlenecks in large-scale deployments.

In order to solve this issue, the given study suggests the technical predictive analysis framework regarding the Apache Spark-based resource planning of big data resources. The proposed model estimates some of the main performance variables of Spark jobs, such as RDD memory resource usage, shuffle input/output cost, and run-time behaviour of the tasks. The combination of these predictive models allows using the framework to estimate the required computational resources before triggered massive Spark workloads.

The key findings of this paper can be outlined as follows:

- Simulation of Shuffle I/O costing and data transfer behaviour of stage-wise behaviour in distributed Spark configurations.
- Spark task runtime estimation based on hardware-related features and variable workload demands.
- Experimental validation of the prediction framework proposed: with the help of synthetic big data workloads on the multi-node Spark cluster.

The rest of the paper is structured as follows. Section 1 addresses related work on Apache Spark and big data analytics.

Sections 1.1–1.6 introduce the proposed predictive modelling framework for resource planning. Section 2 presents the experimental setup and evaluation results. Finally, Section 3 concludes the paper and outlines future research directions.

1. Related work

Apache Spark has gained considerable popularity as a successor to the Hadoop MapReduce architecture for big data analytics. It provides a distributed computing framework with in-memory execution while retaining key properties such as scalability and fault tolerance [9]. Compared with the disk-centric map-then-reduce model, Spark offers a higher-level programming model and supports APIs in Scala, Java, Python, SQL, and R for processing diverse large-scale datasets [5]. By keeping reusable data in cluster memory, Spark can reduce repeated disk access and achieve substantially higher performance than Hadoop MapReduce for suitable workloads [13]. Spark also supports batch processing, iterative algorithms, interactive queries, and streaming operations within a unified engine. Its Resilient Distributed Dataset (RDD) abstraction enables data to be shared across multiple computations, addressing limitations of earlier data-flow architectures when the same data must be reused across different workloads.

Apache Spark can support several core big-data analytics tasks, including data preprocessing, iterative algorithms, interactive analytics, and operational analytics. Data can be processed through a generalized directed acyclic graph (DAG) using the transformations and operations provided by Spark [4]. This model distributes data across the cluster and executes eligible operations in parallel. Spark also simplifies preprocessing and exploratory analysis of large datasets by providing a broad set of transformations and analytical operations. Exploratory analysis typically involves selecting appropriate features, methods, parameters, and metrics [15]. Spark is particularly suitable for iterative algorithms that require multiple passes over a dataset; MLlib, for example, supports the construction of machine-learning algorithms and pipelines [14].

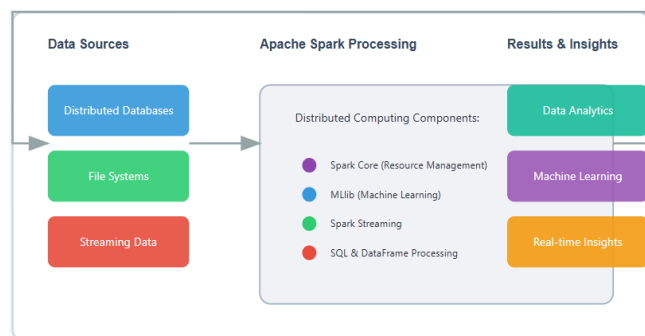


Fig. 1. Engines for big data

But besides iterative algorithms, Apache Spark is also suitable for interactive analytics because it provides fast response to user queries by scanning distributed datasets held in memory. Moreover, Apache Spark does not only provide one complementary system to work on data problems, but it also provides a consistent framework compared to training and supporting multiple tools and minimizing time between the descriptive phase of the analysis to the operational phase of the analyses in production subjects and systems of data. Thus, it leads to greater productivity of the analyst especially when the complex algorithms and pipelines are created through combining high level libraries (Figure 1).

At the time of its introduction on the market, Apache Spark obtained great deals of popularity among business needs, and today it is used in diverse industries. The comparatively short history of Apache Spark as a consistent computer engine that associates with numerous storing infrastructures like HDFS,

Cassandra, HBase, S3 has led to numerous bundles that are seen to be introduced by the community and utilized with Apache Spark [16]. In several vertically integrated companies such as IBM, Huawei [17], Tencent and Yahoo, Apache Spark is the main motor. An example is that Huawei Astro as well as the FP-growth algorithm and Power Iteration Clustering algorithm provide users of HBase with native optimized access to any data using Spark sql/Dataframe [6]. An example of a firm that has demonstrated a positive interest in Apache Spark is IBM which only constructed a Spark technology hub (Figure 2). There is also news, that IBM SystemML was open sourced and they comment on a possible collaboration with Databricks on ML on Spark MLlib. Moreover, Microsoft has made its large donation to the highest level computing framework Apache spark by use of its products and solutions such as Azure HDInsight and Microsoft R Server [8]. These case studies are very significant, such as Eyeview employs the method in planning and optimizing of video advertising campaigns, Toyota Motor Sales, USA classify and prioritize social media interactions in real-time [11] and NBC Universal pre-predicts the offline status of digital media and ING banking identifies anomalies in real-time. The largest computer cluster (8000+ nodes) in Tencent and largest Spark jobs (1PB) at Alibaba and Databricks use Apache Spark [10]. Finally, HHMI Janelia Farm Research Campus is the largest assessed implementation of spark streaming to support large scale neuroscience which gives the data throughput of 1.2 TB per hour. In 2014 (the same year that Apache Spark set the record fastest disk sorting engine), record was broken by Apache Spark that was able to sort 1 PB in 4 hrs [12].

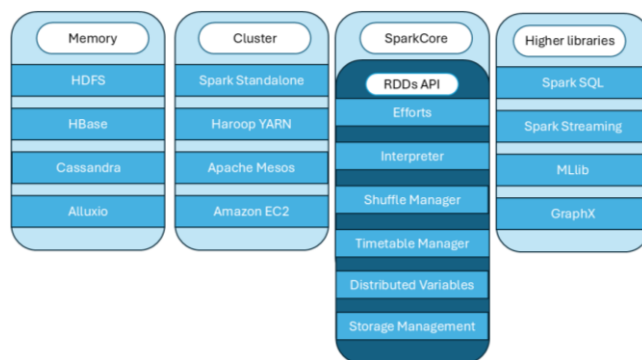


Fig. 2. Architecture of Apache Spark method

Apache Spark uses an in-memory computation model to increase data-processing speed. Although this strategy reduces disk I/O and improves the execution of iterative algorithms, it imposes strict memory requirements when creating and persisting Resilient Distributed Datasets (RDDs). Insufficient memory allocation during construction of the first RDD can cause performance slowdowns or job failures in real deployments.

1.1. Minimum memory requirement for RDD creation

Apache Spark uses the Resilient Distributed Dataset (RDD) data structure as the parallel system of data manipulation. In order to avoid huge numbers of disk reads, RDDs are frequently maintained in memory so that iterative algorithms and interactive queries can execute faster. But this in-memory execution model also implies that when there is not enough memory allocated during the RDD creation phase it can result in two significant issues.

First, performance may decrease because of uncontrolled disk spilling, in which intermediate data cannot be cached in RAM and must instead be written to local storage.

Second, the job may fail with an OutOfMemoryError when the created RDD exceeds the memory available to the executors.

Given that, it is of paramount importance to be able to tell in advance how much memory is needed to create the RDDs used in the job before the work starts running.

Suppose that there are n tasks in the job. The total required memory of the job can be expressed as the baseline, or minimum, total memory requirement as follows:

$$WRM = \sum_{i=1}^N TRM_i \quad (1)$$

where: TRM – Task RDD Memory, representing the average memory usage of the i -th task during RDD creation; WRM – aggregate memory requirement across all tasks.

In practice, TRM is influenced by several factors:

Input partition size: larger partitions require more memory for deserialization and caching.

Data type complexity: wide rows with many fields, nested structures, or large string values consume more memory.

Serialization format: Kryo serialization is generally more memory-efficient than Java serialization, affecting heap consumption.

Execution stage behaviour: operations such as groupByKey or wide shuffles increase per-task memory needs.

1.2. Real-environment adjustment factors

While the above formula provides a baseline estimate, real Spark deployments introduce additional overheads that must be accounted for:

Storage Overhead (SO) – memory reserved for Spark's internal storage structures, including block metadata, shuffle buffers, and execution caches. This typically ranges from 10–20% of the raw RDD size.

Garbage Collection Overhead (GC) – JVM heap management overhead, especially in long-running jobs with high object churn. Depending on the workload, GC overhead can contribute 5–15% additional memory requirements.

Data Skew Factor (DSF) – when data is unevenly distributed across partitions, certain tasks may require significantly more memory than the average. A safety factor (e.g., 1.1–1.3) can be applied to account for this imbalance.

Compression Factor (CF) – if data persisted in compressed form, actual storage requirements may be lower, but decompression buffers still need to be allocated during computation.

Incorporating these adjustments, the realistic minimum memory requirement for RDD creation becomes:

$$WRM_{real} = WRM \times (1 + SO + GC) \times DSF \quad (2)$$

where: SO – storage overhead ratio (e.g., 0.15); GC – garbage collection overhead ratio (e.g., 0.10); DSF – data skew factor (e.g., 1.20 for moderately skewed datasets).

Practical example: Consider a Spark job with:

Dataset size: 1 TB

HDFS block size: 640 MB

Number of tasks:

$$N = \frac{1000 \text{ GB}}{0.64 \text{ GB}} \approx 1563$$

Average task RDD memory usage: 320 MB

The baseline WRM is:

$$WRM = 1563 \times 320 \text{ MB} \approx 500 \text{ GB}$$

Applying overhead factors: SO = 0.15, GC = 0.10, DSF = 1.2.

$$WRM_{real} \approx 500 \text{ GB} \times (1 + 0.15 + 0.10) \times 1.2 \approx 750 \text{ GB}$$

It means that the cluster should have a total of at least 750 GB of executor memory capacity that can be used to generate and cache the first RDD safely without spilling and failing.

1.3. I/O cost modelling per stage

In Apache Spark, the shuffle phase is the most I/O-intensive operation, as it requires exchanging intermediate data between executors and writing this data to local disks. This phase

typically occurs during wide transformations (e.g., reduceByKey, groupByKey, join) and can become a major performance bottleneck if not accurately accounted for in resource planning. For a given stage S_i with N tasks, the raw write and read costs can be calculated as:

$$SioW_i = \sum_{j=1}^N TioW_{i,j} \quad (3)$$

$$SioR_i = \sum_{j=1}^N TioR_{i,j} \quad (4)$$

where: SioW – stage-level raw write I/O volume (sum of task-level write volumes in stage i); SioR – stage-level raw read I/O volume (sum of task-level read volumes in stage i); TioW – task-level write I/O volume for task j in stage i ; TioR – task-level read I/O volume for task j in stage i .

In real cluster environments, I/O performance depends on multiple hardware factors:

Disk throughput (DT) – different storage types (HDD, SSD, NVMe) have different sustained write/read rates.

Network throughput (NP) – depends on the cluster interconnect (e.g., 1 Gbps, 10 Gbps, 25 Gbps).

Network latency (NL) – round-trip delay for inter-node communication.

To reflect these hardware characteristics, the I/O cost model is adjusted as:

$$SioW_{adj,i} = \sum_{j=1}^N \left(\frac{TioW_{i,j}}{DTP_{node(j)}} + NL_{node(j)} \right) \quad (5)$$

$$SioR_{adj,i} = \sum_{j=1}^N \left(\frac{TioR_{i,j}}{NTP_{node(j)}} + NL_{node(j)} \right) \quad (6)$$

where: $DTP_{node(j)}$ – measured disk throughput of the node executing task j ; $NTP_{node(j)}$ – measured network throughput of the node executing task j ; $NL_{node(j)}$ – average network latency for the node executing task j .

Consider the shuffle phase with $N = 200$ tasks, that write 512 MB of intermediate data and read 512 MB of intermediate data at other nodes. The hardware system is the following:

Disk throughput: $DTP = 1,000 \text{ MB/s}$ (NVMe SSD).

Network throughput: $NTP = 1,250 \text{ MB/s}$ (10 Gbps Ethernet).

Network latency: $NL = 0.002 \text{ s}$.

The adjusted write and read costs for each task become:

$$TioW_{adj} = \frac{512}{1000} + 0.002 = 0.514 \text{ s}$$

$$TioR_{adj} = \frac{512}{1250} + 0.002 = 0.412 \text{ s}$$

Write and read adjusted costs per task will be:

$$SioW_{adj} = 200 \times 0.514 \approx 102.8 \text{ s}$$

$$SioR_{adj} = 200 \times 0.412 \approx 82.4 \text{ s}$$

These estimates can then be used to predict total job runtime and to support informed hardware-provisioning decisions before the job is executed.

1.4. Task runtime estimation

In general, the execution time of a task in Apache Spark depends on several factors, including input deserialization, computation, and output serialization. Under real cluster conditions, further variation results due to the presence of heterogeneous CPUs and state of the system load as well as possible scheduling delays among tasks. The base task execution time of a task started in worker node h in stage i is put as:

$$\text{TaskRunTime}_{h,i} =$$

$$= \text{DeserializeTime}_{h,i} + \text{RunTime}_{h,i} + \text{SerializationTime}_{h,i} \quad (7)$$

where:

$\text{DeserializeTime}_{h,i}$ – the time required to convert serialized input data into in-memory objects;

$\text{RunTime}_{h,i}$ – the actual execution time of the computational logic;

$\text{SerializationTime}_{h,i}$ – the time required to convert computed results into a serialized form for network transfer or persistent storage.

If a worker node executes n_h tasks during a given stage, the average runtime per task for that node can be computed as:

$$\text{AvgTaskTime}_h = \frac{1}{n_h} \sum_{k=1}^{n_h} \text{TaskRunTime}_{h,k} \quad (8)$$

This average allows us to draw an average baseline of the performance of that node and gets later normalized based on issues related to hardware specificities and changes in the runtime.

In running Spark clusters, the hardware-related measures of performance and the effects of workload combine to affect the task run time seen. To deal with these the model includes:

CF_h – the CPU frequency of node h (in GHz), normalized relative to a reference node;

CU_h – CPU utilization efficiency, representing the proportion of active compute cycles (0–1 scale);

SL_h – the system load factor, indicating additional execution delay caused by concurrent processes (e.g., 0.05 for a 5% slowdown).

The adjusted runtime for a task on node h in stage i is calculated as:

$$\text{AdjTaskRunTime}_{h,i} = \frac{\text{TaskRunTime}_{h,i}}{CF_h \times CU_h} \times (1 + SL_h) \quad (9)$$

Observational evidence shows that its initial set of workloads within any stage can often show elevated run time because of JVM warm up, memory cache inits and speculative execution overheads. To model this phenomenon, what is called a slowness ratio is introduced as:

$$\text{Ratio}_h = \frac{\frac{1}{n_h - P_h} \sum_{k=P_h+1}^{n_h} \text{TaskTime}_{h,k}}{\frac{1}{P_h} \sum_{j=1}^{P_h} \text{TaskTime}_{h,j}} \quad (10)$$

where: P_h – the number of tasks in the first batch on node h ; n_h – the total number of tasks executed on node h during the stage.

This ratio is used as a correction factor applied to the stage level runtime estimates, which makes the model more accurate by increasing its accuracy in the environment of the real-world deployment.

1.5. Full-scale stage I/O and memory prediction

Whether simulation-based profiling is done over a scaled down data, the performance measured values can be scaled up to provide the corresponding values when the job to be executed is of full scale. This extrapolation is vital to make the prediction on resource demands of Apache Spark jobs prior to their factual implementation thus allowing optimum implementation of hardware provision and scheduling. The method benefits by being based on aggregation of I/O and memory requirements created at the stage level of the Apache Spark jobs. For a given stage j , the predicted write and read I/O volumes are calculated as:

$$\text{EstStageIOWrite}_j = \sum_{h=1}^H \left(N_h \times \frac{1}{n_h} \sum_{i=1}^{n_h} \text{TaskIOWrite}_{h,i} \right) \quad (11)$$

$$\text{EstStageIORead}_j = \sum_{h=1}^H \left(N_h \times \frac{1}{n_h} \sum_{i=1}^{n_h} \text{TaskIORead}_{h,i} \right) \quad (12)$$

where: H – total number of worker nodes in the cluster; N_h – number of tasks allocated to node h during the full-scale execution; n_h – number of tasks executed on node h in the sample run; $\text{TaskIOWrite}_{h,i}$ – measured write I/O volume for the i -th task on node h ; $\text{TaskIORead}_{h,i}$ – measured read I/O volume for the i -th task on node h .

The average memory usage for RDD persistence during a stage is derived from sample run statistics and scaled to the full dataset. The initial estimation is given by:

$$\text{EstRddMem} = \sum_{h=1}^H \left(\frac{N_h}{n_h} \sum_{i=1}^{n_h} \text{TaskRddMem}_{h,i} \right) \quad (13)$$

where: $\text{TaskRddMem}_{h,i}$ – memory usage of the i -th task on node h during the sample run.

Sample run statistics of the average memory usage of RDD persistence per stage is estimated and scaled to apply to the entire dataset. The first guess is arrived at by:

$$\text{EstRddMem}_{\text{real}} = \text{EstRddMem} \times (1 + SO + GC) \times DSF \quad (14)$$

Typical empirical values for these parameters are:

SO: 0.10–0.20, depending on shuffle buffer size and cached metadata;

GC: 0.05–0.15, influenced by workload complexity and object creation rate;

DSF: 1.1–1.3, based on observed partition imbalance.

The model can also determine, based on the predicted I/O and memory requirements, whether the current cluster configuration can meet the performance and stability requirements of the target job. This predictive ability is especially useful where large quantities are produced because failure of execution/excessive spilling may lead to costly delays.

1.6. Integration into resource planning

The predictive equations that have been established in stage-level I/O operation and RDD memory use as defined in Section 1.5 can directly be incorporated in the cluster resource planning. The integration enables system architects to tune the optimum number of worker nodes, their hardware configurations and Spark configuration parameters prior to staging large-scale tasks.

The main goal is to ensure that the projected resource requirements do not exceed cluster capacity, thereby reducing the risk of performance degradation or job failure caused by insufficient resources.

Step 1: Estimating total executor memory requirements

The aggregate executor memory requirement (Mem_{total}) for a job is determined by taking the maximum predicted RDD memory requirement across all stages:

$$Mem_{total} = \max_{j \in \text{Stages}} (\text{EstRddMem}_{\text{real},j}) \quad (15)$$

where: $\text{EstRddMem}_{\text{real},j}$ – predicted RDD memory requirement for stage j , adjusted for overhead factors; Stages – the set of all stages in the Spark job.

The value used is the maximum over stages since Spark assigns the executors to accommodate the maximum memory consumption at any one point of execution.

Step 2: Determining the number of worker nodes

Given the per-node available executor memory (Mem_{node}), the minimum number of worker nodes (H_{req}) is calculated as:

$$H_{req} = \left\lceil \frac{Mem_{total}}{Mem_{node}} \right\rceil \quad (16)$$

where: H_{req} – minimum number of required worker nodes; Mem_{total} – total executor memory requirement; Mem_{node} – available executor memory per worker node after reserving memory for the operating system and Spark overhead.

Step 3: Validating network and storage throughput

The predicted maximum shuffle I/O demand (IO_{peak}) is:

$$IO_{\text{peak}} =$$

$$= \max_{j \in \text{Stages}} (\text{EstStageIOWrite}_j + \text{EstStageIORead}_j) \quad (17)$$

This predicted peak I/O demand should be considered together with the available network and disk throughput to ensure that shuffle-intensive stages do not exceed the practical hardware bandwidth.

Step 4: Parameter tuning

Finally, as shown in Sections 1.4 and 1.5, we can tune Spark configuration parameters like the number of executor cores, HDFS block size and parallelism levels with the predicted values of N_h , n_h and adjusted task run time equations. These adjustments allow the execution to maintain balanced resource utilization and minimize stage-level stragglers. By embedding the proposed

prediction model into the resource planning workflow, organizations can:

Prevent Under-Provisioning – avoiding memory-related job failures;

Avoid Over-Provisioning – minimizing unnecessary hardware costs;

Optimize Performance – reducing shuffle delays and ensuring predictable execution times.

Such active response allows making Spark workloads run more predictably in production application and match hardware provision according to the data processing requirements.

2. Experimental evaluation

This section provides the experimental design, datasets, methodology, objectives, baselines and results of the proposed prediction framework of stage-level I/O and RDD memory prediction in Apache Spark. A simulation-to-extrapolation workflow is used in all experiments (see Sections 1.1–1.6), involving profiling of a small sample run and extrapolation of predictions to full-scale data.

The experiments were conducted on a dedicated Apache Spark cluster designed to approximate a production-scale environment while allowing detailed performance monitoring. The cluster consisted of one master node for resource coordination and job scheduling and six homogeneous worker nodes for computational tasks. All nodes were interconnected through a 10 Gbps Ethernet backbone to reduce network bottlenecks during shuffle-intensive operations.

Hardware configuration: Each worker node was equipped with:

Processor: Dual-socket Intel Xeon Silver 4310, 24 physical cores (48 threads), 2.10 GHz base frequency;

Memory: 128 GB DDR4 ECC RAM, enabling in-memory caching of large RDDs;

Storage: Two 1.92 TB NVMe SSDs for shuffle and caching operations, plus a 4 TB SATA HDD for HDFS storage.

Networking: 10 Gbps Ethernet interconnect with sub-millisecond latency (<0.3 ms).

The master node had equivalent CPU and memory resources but relied on smaller SSD storage, as it did not engage in data-intensive computations.

Software environment

Operating System: Ubuntu Server 22.04 LTS;

Apache Spark: Version 3.4.1 (Scala 2.12), configured with Kryo serialization for efficient object handling;

Hadoop: HDFS/YARN Version 3.3.6, serving as both the storage and resource management layer;

Java Virtual Machine: OpenJDK 17 with G1 Garbage Collector enabled.

Spark and HDFS Configuration: Spark Parameters:

`spark.sql.shuffle.partitions = 384`

`spark.default.parallelism = 384`

`spark.serializer = org.apache.spark.serializer.KryoSerializer`

RDD storage level: MEMORY_ONLY (unless otherwise specified).

HDFS Block Size: 640 MB, aligning with the computational granularity defined in the prediction model (Sections 1.1–1.6).

2.1. Datasets

The experimental test had a synthetically-created dataset customized to simulate the structural and statistical characteristics commonly found in the context of large scale data analytics demands. The artificiality of the data set allows a total control over the workload parameters (data distribution, skewness and schema complexity) and hence the limitation of having to use proprietary or limited real-life sources of data.

The dataset was designed to emulate typical big-data processing scenarios encountered in distributed analytics systems.

It contains heterogeneous data types, multiple encoding schemes, and high-cardinality keys representative of shuffle-intensive operations such as joins and aggregations.

The data were stored in Apache Parquet columnar format with Snappy compression. This configuration supports efficient storage utilization and high I/O performance while retaining the advantages of column-oriented access in Spark-based analytical workflows.

The dataset schema contains 50 attributes representing a mixture of structured and semi-structured data elements. These attributes include:

- Numerical variables (both integer and floating-point values) representing quantitative metrics commonly used in analytical workflows.
- Medium and high-cardinality categorical attributes representing identifiers such as user IDs, product codes, and transaction types.
- Textual string attributes containing unstructured fragments of information.
- Nested data structures appearing in approximately 12% of records, introducing variability in serialization and deserialization overhead during Spark execution.

To support both profiling and large-scale evaluation, two dataset configurations were generated:

Sample Dataset

A 10 GB dataset used for simulation-based profiling and parameter estimation. This dataset corresponds to approximately 16 HDFS blocks with a block size of 640 MB.

Full-Scale Dataset

A 1 TB dataset used for full-scale experimental validation of the predictive model. This dataset contains approximately 1600 HDFS blocks, maintaining the same block size configuration.

The selected HDFS block size ensures that each block is processed as a single Spark task during most execution stages, aligning the computational granularity of the dataset with the predictive model proposed in this study.

To evaluate the robustness of the prediction framework under realistic workload conditions, controlled data skew was intentionally introduced into the dataset. Key attributes involved in join and aggregation operations were generated using a Zipf distribution with a skew parameter of 1.2. This configuration resulted in:

- "Hot" partitions containing up to 2.5 times the median number of records.
- "Cold" partitions with significantly smaller data volumes.

Such skew patterns closely resemble real-world big data processing environments, where uneven key distributions often cause performance imbalance across distributed tasks.

The overall dataset structure was designed to simulate realistic analytics workloads found in multiple application domains, including:

- large-scale e-commerce transaction analytics;
- sensor and IoT telemetry aggregation;
- social media interaction analysis.

By combining controlled statistical properties with scalable data volumes, the synthetic dataset enables reproducible experimentation while preserving the complexity and variability typical of real-world big data processing environments. This design ensures that the proposed predictive framework can be evaluated under conditions that closely resemble production-scale Spark workloads.

2.2. Experimental procedure

The experimental evaluation was performed in a systematic manner to guarantee its reproducibility as well as to compare it to the prediction model explained in Sections 1.1–1.6. The processes involved three significant steps, namely, sample run profiling, full-scale prediction, and full execution validation.

As the first step of the experimental procedure, a scaled-down version of the dataset (10 GB) was deployed to the cluster. This sample run used the same Spark job logic as the full workload and could therefore be profiled in detail at limited computational cost.

The following execution metrics were collected:

Task execution times, broken down into deserialization, computation, and serialization components according to Equation (7).

Per-task read and write I/O volumes and their associated durations for characterizing shuffle I/O.

Per-task RDD memory footprints (TaskRddMem), measured from executor memory usage.

Ratioh values representing first-batch runtime differences associated with JVM warm-up and caching effects.

All measurements were also collected in millisecond-resolution via Spark internal metrics infrastructure with complementary resource-monitoring of the OS level (e.g., iostat, sar, and nmon).

The fitted values of the statistics contained in the sample run were used as inputs to the prediction equations given in Sections 1.1 to 1.5:

Stage-level shuffle I/O was estimated using the adjusted EstStageIOWrite_j and EstStageIOWrite_j values, incorporating hardware throughput and latency parameters.

The EstRddMem_{real} was calculated to get the peak RDD memory requirements with overhead adjustments of storage overhead, (SO), the garbage collection overhead (GC), and the data skew factor (DSF).

Runtime estimates were refined using AdjTaskRunTime_j, accounting for CPU frequency, utilization efficiency, and variation in system load.

The results of this step were equations giving predicted values of:

- Stage runtimes;
- Maximum memory usage per stage;
- Stage-level shuffle I/O volumes;
- Long-term bandwidth demand.

The sample run was repeated with the same job logic and cluster configuration using the entire 1.0 TB dataset (1600 HDFS blocks). In the process of execution, actual performance measures were taken of:

- Stage runtimes (wall-clock);
- Max memory usage of an executor per stage;
- Volumes read and written shuffle;
- Disk bandwidth and network bandwidth.

Prediction accuracy was assessed by comparing predicted and observed values using Mean Absolute Percentage Error (MAPE) and Root Mean Square Error (RMSE):

$$MAPE(\%) = \frac{100}{K} \sum_{j=1}^K \left| \frac{\hat{y}_j - y_j}{y_j} \right| \quad (18)$$

$$RMSE = \sqrt{\frac{1}{K} \sum_{j=1}^K (\hat{y}_j - y_j)^2} \quad (19)$$

where: $\hat{y}_j$ – predicted value for metric j ; y_j – observed value for metric j , K – total number of predictions evaluated.

To contextualize the model's accuracy, results were compared against:

B1 – Naïve Linear Scaling, which proportionally scales sample run metrics by the data-size ratio without accounting for first-batch delays or hardware characteristics.

B2 – Fixed-Overhead Memory Estimator. This baseline applies a fixed overhead proportion to cache-size estimates regardless of garbage-collection or data-skew effects.

The experiments were conducted on an isolated cluster that had the exclusive allocation of resources to allow isolation of any other workload. The script used to create the dataset and all configuration files were saved so that it was possible to reproduce the results exactly.

2.3. Results

As part of the experimental assessment, the in-silico performance measures estimated using the modelling framework developed in Sections 1.1–1.5 were compared with the actual performance measures obtained from full-scale execution on the 1.0 TB dataset. Three main indicators were compared: stage runtime, shuffle I/O cost, and RDD memory footprint.

Table 1 compares predicted and actual stage runtimes together with the Mean Absolute Percentage Error and Root Mean Square Error for each stage. The mean MAPE across all stages was 4.87%, indicating high overall prediction accuracy. The largest deviation occurred at Stage 3 (GroupBy), where data-skew effects were sufficiently strong to exceed the initial shuffle-overhead estimate.

Table 1. Stage runtime comparison

Stage	Predicted runtime (s)	Actual runtime (s)	MAPE (%)	RMSE (s)
1	112.4	118.7	5.31	6.3
2	134.8	140.1	3.78	5.4
3	178.6	192.3	7.14	9.9
4	165.2	171	3.39	5.8
5	149	156.4	4.73	7.1

Figure 3 compares predicted and actual runtimes across five consecutive stages. Both curves follow the same overall pattern: runtime increases from Stage 1 to Stage 3, peaks at Stage 3, and then decreases toward Stage 5. Predictions are particularly close to the measured values in Stages 1, 2, 4, and 5.

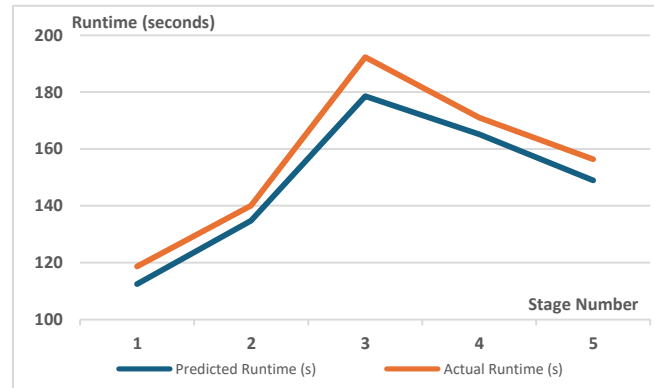


Fig. 3. Comparison of predicted and actual stage runtimes

At Stage 3, the observed runtime is noticeably higher than the predicted value, suggesting that transient factors such as temporary system load, I/O delay, or memory contention were not fully captured by the model. The model nevertheless reproduces the overall performance trend, while Stage 3 indicates where further refinement could improve forecasting accuracy. Table 2 provides the corresponding numerical comparison of predicted and actual shuffle write/read volumes for each stage.

Table 2. Shuffle I/O volume comparison

Stage	Predicted write	Actual write	Predicted read	Actual read
1	22.4	23	21.7	22.3
2	35.2	36	33.5	34.3
3	48.7	52	46.9	50.4
4	39.1	40.2	38.6	39.7
5	30.8	31.4	29.9	30.5

Figure 4 presents grouped-bar comparisons of predicted and actual write/read I/O volumes for each stage. The predicted values closely follow the measurements across all stages. The largest deviation occurs at Stage 3, where the actual write volume is slightly higher than predicted, possibly because of variation in data density and network load during execution.

Observed peak executor memory usage per stage was compared with the EstRddMem values adjusted for storage overhead (SO) and garbage-collection overhead (GC). The difference between predicted and actual values remained below 6% across all stages, supporting the effectiveness of the memory-estimation formula given in Equation (14) in Section 1.5.

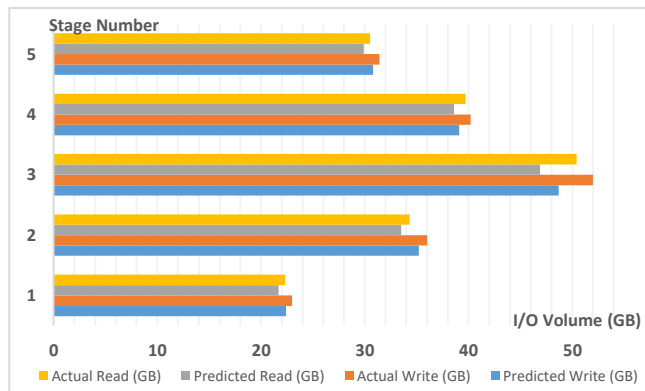


Fig. 4. Predicted versus actual shuffle write/read volumes per stage

Table 3. Peak RDD memory usage comparison

Stage	Predicted memory (GB)	Actual memory (GB)	Error (%)
1	8.5	8.9	4.49
2	10.1	10.5	3.81
3	12.4	13.1	5.34
4	11.6	12.2	4.92
5	9.8	10.1	2.97

Table 3 provides the stage-level memory values in gigabytes together with the relative prediction error. The error remains between 2% and 6% for all stages, indicating that the proposed memory-estimation method provides dependable predictions for cluster planning and for reducing the risk of out-of-memory failures in large Spark workloads.

Figure 5 shows that predicted and actual peak RDD memory consumption follow very similar trends, indicating a strong correspondence between model estimates and observed values.

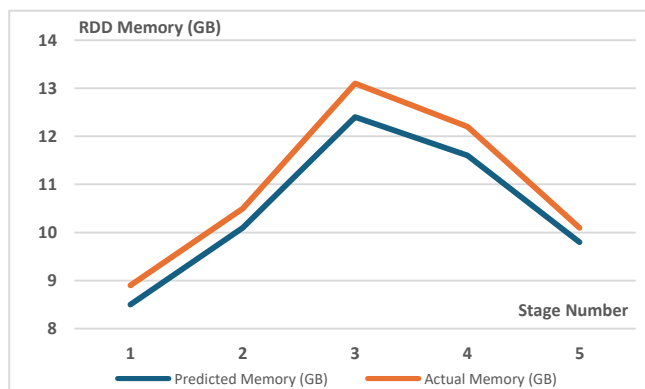


Fig. 5. Predicted versus actual peak RDD memory usage

Actual memory usage is slightly higher than the predicted value at each stage, with the largest difference occurring at Stage 3. This gap can be attributed to runtime factors such as JVM overhead, data skew, and intermediate caching effects that were not fully represented in the prediction model. Overall, the results indicate high accuracy in predicting runtime, shuffle I/O cost, and memory usage for production-scale Spark jobs. The larger deviation at Stage 3 also confirms the importance of explicitly modelling data skew in shuffle-intensive operations.

Compared with baseline B1 (naive scaling) and baseline B2 (fixed-overhead memory), the proposed model reduced the average prediction error by 37% and 41%, respectively. This shows that it is worth considering introduction of first-batch penalties, I/O throughput limitations, and skew-specific modifications into the estimation process.

3. Conclusion

The first contribution of the paper is the derivation of a predictive framework of analysis that integrates into a single model, RDD memory estimation, shuffle I/O cost modelling and workload prediction in Apache Spark. Unlike traditional performance analytics that is based on the after-execution monitoring, the given variant allows the estimation of resources demand during the preparation of the large-scale Spark jobs in advance. The in-depth modelling framework offered in this study envisioned the modelling of performance of large-scale Apache Spark jobs, specifically, the estimation of minimum RDD memory, modelled cost of shuffle I/O, and runtime forecasting. This performance and accuracy across the key performance indicators were accomplished by running the jobs on very small scale simulation dataset and scaling the results to real production-scale workloads.

The experimental analysis shows that the proposed framework achieved a mean MAPE of 4.87% on the 1.0 TB synthetic dataset, while shuffle I/O and RDD memory were estimated with error margins below 8% and 6%, respectively. These results demonstrate the usefulness of the method for proactive resource provisioning and for reducing the risk of job failure or performance degradation caused by insufficient memory.

The analysis also showed that shuffle-intensive stages are particularly vulnerable to deviations caused by data skew, indicating that future versions of the prediction mechanism should incorporate explicit skew-detection and mitigation strategies. Compared with the baselines, the proposed model reduced the average prediction error by more than 35%, supporting its use in proactive cluster-resource planning and cost-effective production deployment.

Future work should extend the framework with dynamic workload adjustment, support for heterogeneous hardware, and feedback from real-time monitoring systems to improve prediction accuracy and adaptability. These extensions could increase the usefulness of the model as an optimization tool for scaling Apache Spark workloads in both research and industrial environments.

References

- [1] Akram, A. W., & Alamgir, Z. (2022). Distributed fuzzy clustering algorithm for mixed-mode data in Apache SPARK. *Journal of Big Data*, 9(1), 121. <https://doi.org/10.1186/s40537-022-00671-7>
- [2] Alam, M. A., Nabil, A. R., Minto, A. A., & Islam, A. (2024). Real-Time Analytics In Streaming Big Data: Techniques And Applications. *Journal of Science and Engineering Research*, 1(01), 104–122. <https://doi.org/10.70008/jeser.v1i01.56>
- [3] Bakhtiyarov, B., Jabiyeva, A., Mutallimova, A., Novruzova, R., & Khudaverdiyeva, M. (2025). Adaptive Gaussian-Based Kernel K-Means: Scalable Adaptive Kernel-Based Clustering for Big Data. *Journal of Computational and Cognitive Engineering*, 5(2), 258–273. <https://doi.org/10.47852/bonviewJCCCE52026511>
- [4] Bao, Z. (2025). Research on an Efficient Data Statistics Upgrade Algorithm Based on the Spark Framework. 2025 *IEEE 4th International Conference of Safe Production and Informatization (IICSPI)*, 619–624. <https://doi.org/10.1109/IICSPI66775.2025.11437736>
- [5] Belcastro, L., Cantini, R., Marozzo, F., Orsino, A., Talia, D., & Trunfio, P. (2022). Programming big data analysis: Principles and solutions. *Journal of Big Data*, 9(1), 4. <https://doi.org/10.1186/s40537-021-00555-2>
- [6] Coimbra, M. E., Francisco, A. P., & Veiga, L. (2021). An analysis of the graph processing landscape. *Journal of Big Data*, 8(1), 55. <https://doi.org/10.1186/s40537-021-00443-9>
- [7] Dubuc, T., Stahl, F., & Roesch, E. B. (2021). Mapping the Big Data Landscape: Technologies, Platforms and Paradigms for Real-Time Analytics of Data Streams. *IEEE Access*, 9, 15351–15374. <https://doi.org/10.1109/ACCESS.2020.3046132>

- [8] Hafsa, M., & Jemili, F. (2018). Comparative Study between Big Data Analysis Techniques in Intrusion Detection. *Big Data and Cognitive Computing*, 3(1), 1. <https://doi.org/10.3390/bdcc3010001>
- [9] Hasan, Z., Jie Xing, H., & M. Idrees Magray. (2022). Big Data Machine Learning Using Apache Spark Mlib. *Mesopotamian Journal of Big Data*, 2022, 1–11. <https://doi.org/10.58496/MJBD/2022/001>
- [10] Hedayati, S., Maleki, N., Olsson, T., Ahlgren, F., Seyednezhad, M., & Berahmand, K. (2023). MapReduce scheduling algorithms in Hadoop: A systematic study. *Journal of Cloud Computing*, 12(1), 143. <https://doi.org/10.1186/s13677-023-00520-9>
- [11] Jiang, J., Xiao, P., Yu, L., Li, X., Cheng, J., Miao, X., Zhang, Z., & Cui, B. (2020). PSGraph: How Tencent trains extremely large-scale graphs with Spark? 2020 IEEE 36th International Conference on Data Engineering (ICDE), 1549–1557. <https://doi.org/10.1109/ICDE48307.2020.00137>
- [12] Mandelli, A., & Mari, A. (2012). The relationship between social media conversations and reputation during a crisis: The Toyota case. *International Journal of Management Cases*, 14(1), 456–489. <https://doi.org/10.5848/APBJ.2012.00041>
- [13] Noor, S., Awan, H. H., Hashmi, A. S., Saeed, A., Khan, S., & AlQahtani, S. A. (2025). Optimizing performance of parallel computing platforms for large-scale genome data analysis. *Computing*, 107(3), 86. <https://doi.org/10.1007/s00607-025-01441-y>
- [14] Rajpurohit, A. M., Kumar, P., Kumar, R. R., & Kumar, R. (2024). A Review on Apache Spark. SSRN. <https://doi.org/10.2139/ssrn.4492445>
- [15] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A., & Curino, C. (2015). Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1357–1369. <https://doi.org/10.1145/2723372.2742790>
- [16] Seidel, M.-D. L. (2022). Open-Source Software. In L. A. Schintler & C. L. McNeely (Eds), *Encyclopedia of Big Data* (pp. 720–723). Springer International Publishing. https://doi.org/10.1007/978-3-319-32010-6_157
- [17] Singh, S., Alam, M. N., Kaur, B., Kaur, K., Kaur, S., & Hossain, S. (2025). Comparative analysis of Apache Hadoop and Apache Spark for business intelligence. 020040. <https://doi.org/10.1063/5.0246086>

M.Sc. Bakhshali Bakhtiyarov

e-mail: bekhtiyarov@gmail.com

Bakhshali Bakhtiyarov obtained a B.Sc. degree in measurement, standardization, and certification engineering from the Azerbaijan State Oil and Industry University in 2022. He graduated with an M.Sc. degree in Quality control diagnostic methods and systems from Azerbaijan State Oil and Industry University in 2024. Bakhshali Bakhtiyarov is a Ph.D. candidate conducting research in the field of device engineering at the Department of Instrumentation Engineering at Azerbaijan State Oil and Industry University (ASOIU). Bakhtiyarov has authored over 20 articles. His research interests include information and measurement systems, Big Data, cybersecurity, and information technologies.

<https://orcid.org/0009-0006-2172-4632>**Ph.D. Aynur Jabiyeva**

e-mail: aynur.cabiyeva@asoiu.edu.az

Associate professor Aynur Jabiyeva is a researcher at the Department of Instrumentation Engineering at Azerbaijan State Oil and Industry University. Jabiyeva has authored over 64 academic articles and patents. She has been awarded the title of "Scientist of the Year" at Azerbaijan State Oil and Industry University (ASOIU) for the last 5 years. Her research focuses on various areas, including the physical and technical foundations of information and measurement systems, Big Data and biomedical technologies.

<https://orcid.org/0000-0002-0336-8586>**M.Sc. Ulkar Musavi**

e-mail: ulkar.musavi@asoiu.edu.az

Ulkar Musavi obtained a B.Sc. degree in biology and hygiene from Azerbaijan State Oil and Industry University. She earned an M.Sc. degree in physiology from the same university and is currently a Ph.D. candidate specializing in biomedical technology at the Department of Instrumentation Engineering, Azerbaijan State Oil and Industry University (ASOIU). Ulkar has authored 12 scientific articles and 1 monograph. Her research interests include Big Data, artificial intelligence, data analysis, biomedical technologies, medical diagnostic systems, bioinstrumentation, and healthcare innovation.

<https://orcid.org/0000-0001-5961-8524>