International Journal of Research in Science and Technology http://www.ijrst.com

(IJRST) 2022, Vol. No. 12, Issue No. IV, Oct-Dec **e-ISSN: 2249-0604, p-ISSN: 2454-180X**

Benchmarking Apache Spark vs. Hadoop: Evaluating In-Memory and Disk-Based Processing Models for Big Data Analytics[1]

Ahmed Elgalb, George Samaan
Independent Researchers, Iowa, United States.

DOI:10.37648/ijrst.v12i04.008

Received: 12 September 2022; Accepted: 13 November 2022; Published: 13 December 2022

ABSTRACT

Apache Spark and Hadoop MapReduce are two of the most popular data processing paradigms for large-scale computing and each has its own model and philosophy of execution. Spark's in-memory model promises better execution for iterative, interactive, and streaming workloads, and Hadoop MapReduce's disk-based solution remains a staple of massive one-pass jobs. This paper presents an in-depth discussion of both frameworks based on studies and benchmarks published prior to 2022. By exploring their architectures, performance, fault tolerance, and compatibility with larger analytics stacks, it illustrates where each one is superior and where they are able to work together. It explains the cost of large-scale in-memory caching, why iterative machine learning algorithms get the most out of Spark's DAG architecture, and why some tasks still get better off using Hadoop's stable batch structure. Across many tables and examples, the paper illustrates a subtle point: Spark and Hadoop are not competing monoliths but complementary tools for distinct workload profiles that each are relevant to an array of real-world data engineering and analytics situations.

INTRODUCTION

Over the last 20 years, this rise in the number of massive datasets inspired the idea of creating distributed systems that can perform analytics and transformations on scales that were previously unknown. Most companies use these distributed processing platforms not only for storing massive amounts of data, but also for generating information that could be used to make financial, retail, research, governance, and so on decisions. Against this backdrop, Apache Hadoop and Apache Spark have emerged as the foundations of big data platforms. Hadoop MapReduce, developed from Google's original MapReduce, was a disk-based batch-processing approach that rapidly became the core of massive data analysis [1]. Eventually, however, iterative and interactive tasks – especially machine learning and near-real-time analytics – began challenging that disk-centric paradigm. Apache Spark stepped in to address this performance shortfall with in-memory computation which dramatically reduced latencies for repeatedly retrieving data [2], [3].

Spark's in-memory architecture has frequently been touted as a "game-changer," because of its RDD abstraction, which allows data partitions to be cached in RAM. This methodology avoids the disk reads and writes that were typical of Hadoop MapReduce's map–shuffle–reduce loops [2], [3]. Spark too provides a single engine for batch analytics, streaming, SQL, machine learning, and graphs processing, while Hadoop's more traditional approach uses a much wider but still somewhat fragmented suite of tools (Hive, Pig, Storm, etc.) to cover specialized tasks [4]. Spark has obvious advantages in iterative, interactive environments, but Hadoop MapReduce is good for single-pass tasks when memory is tight or your company is looking for a cheaper way to transform data at scale.

[1] *How to cite the article:* Elgalb A., Samaan G., Benchmarking Apache Spark vs. Hadoop: Evaluating In-Memory and Disk-Based Processing Models for Big Data Analytics; IJRST, Oct-Dec 2022, Vol 12, Issue 4, 43-52, DOI: http://doi.org/10.37648/ijrst.v12i04.008

Much of the literature that existed prior to 2022 was dedicated to assessing Spark and Hadoop with respect to performance, shuffle overhead, fault tolerance, utilization, and cost-effectiveness [5], [6]. Though most of these research emphasize Spark's superiority over MapReduce in iterative machine learning pipelines, they also confirm that MapReduce is capable of catching up in situations where there are no repeated reads of data. This post gathers those thoughts, covers the architecture behind both frameworks, and covers how actual deployments combine Spark and Hadoop into hybrid workflows. It also weighs cost-based factors, as in-memory computing can use cluster nodes that have high amounts of RAM, and this can be paid for with shorter runtimes, especially in the cloud pricing models [7], [8].

These chapters take a closer look at the evolution of each model, exploring its architectural differences, typical performance results, and how each model fits into the larger big data ecosystem. A number of tables provide in-depth comparisons of things like shuffle overhead, memory footprint, streaming, fault tolerance, and concurrency management. The final two sections of this paper propose that Spark and Hadoop are still complimentary offerings that perform well in a specific situation. Rather than seeing them as the obvious enemy, many big data enterprises combine Hadoop's robust data storage and batch ETL features with Spark's speed and flexibility for iterative or interactive analytics.

HISTORICAL CONTEXT AND RELATED WORK

The introduction of Hadoop MapReduce as an open source platform provided organizations with a way to address the challenge of big data without investing in complex, high-end hardware. Because Hadoop relied on HDFS as a distributed storage mechanism, it could fragment big data files into blocks that were replicated between nodes, which provided fault tolerance and high availability [1]. The programming language MapReduce performed very well for very low latency batch tasks — like daily log aggregation, big data ETL, and massive indexing of pages. But once tasks required more than a few iterations, or nearly real-time responses, writing the intermediate output to disk became a significant bottleneck.

Spark was developed as an outgrowth of research at UC Berkeley's AMPLab to combat these overheads. The in-memory abstraction it used, the RDD, offered lineage-based fault tolerance — instead of replicating data from one node to another in memory, Spark could recompile partitions that were lost by reversing the transformations that created them [2]. As a performance benchmark, iterative machine learning functions, like logistic regression or k-means clustering, ran much faster if their intermediate output remained in memory. High throughput workflows that ran the same dataset over and over again were the fastest [3]. They also praised Spark's DAG scheduler, which aggregates multiple transformations into smaller shuffle boundaries, compared with the rigid map–shuffle–reduce steps in Hadoop's initial approach.

Together with Spark innovations, Hadoop was trying to adapt. YARN (Yet Another Resource Negotiator) separated the resource manager layer from MapReduce to allow for multiple engines, such as Spark and Tez, to run on the same cluster [4]. Tez added DAG-based processing to Hive queries and thus increased Hadoop's performance for interactive and iterative work, though a majority remained more interested in Spark's in-memory speedups [5], [9].

By 2020-2021, dozens of comparative research papers, some from academia, others from enterprise labs, had made Spark known for handling iterative data analytics more efficiently [5], [6], [7]. Meanwhile, most of these studies also noted that when the size of data is so large that memory caching provides no longer significant benefits or single-pass transformations are more frequent, Hadoop is still the right tool. In real life scenarios, usage patterns are similar: enterprises execute Hadoop MapReduce jobs on a nightly or weekly basis for batch ETL that's stable, while Spark gets used as an intermediate storage for data analysis or near real-time streaming. The combined approach uses Hadoop to store the data (HDFS) and Spark to perform accelerated or iterative tasks.

ARCHITECTURE AND EXECUTION MODELS

Hadoop MapReduce

Hadoop MapReduce's architecture is disk oriented. The input data is read from HDFS and put into map jobs that split or churn individual blocks. The intermediate key-value pairs generated are stored on local disk, and shuffled back to reducers that complete the aggregations or joins. Finally, the minimized output is returned in HDFS [1]. This closed map–reduce pipeline is easy to visualise, but can be inefficient for data flows that require many iterations. For every iteration, the output of the reduce step must be written to memory and read over again if processing is required.

With respect to fault tolerance, Hadoop utilizes data replication in HDFS (by default it is set to three) to prevent data loss in the event that a node fails [1]. In the event that one of the nodes containing a data block crashes, the replica is found on another node and the map or reduce operations are moved forward. This approach is very robust for large linear batch operations, but can cause additional data overhead.

With YARN, the new Hadoop 2.x architecture further generalized resource allocation, enabling non-MapReduce applications to be run on the same cluster [4]. Even so, the original MapReduce engine is still primarily disk-centric and the outputs from each step are typically read from disk. This configuration is reliable, proven, and easy to maintain in the context of batch ETL, although repeated disk I/O can become a performance bottleneck in interactive or complex pipelines.

Apache Spark

Spark's defining feature is the Resilient Distributed Dataset that caches data partitions in memory for the entire duration of a job or over multiple computation steps. Spark's DAG scheduler creates a schedule in which transformations using the same data are combined or optimized, minimizing shuffles [2], [3]. This reduces the time spent writing intermediate key-value pairs to disk for workflows that loop through the dataset (e.g., iterative machine learning or graph algorithms).

Spark relies on a lineage-based fault tolerance. Whenever a partition is lost Spark uses its list of transformations to reconstruct that partition from the dataset (or from an earlier step [2]). This is different from Hadoop's use of physically replicating blocks of data on disk. Lineage-based recovery is good for temporary computations, but it can be costly if more than one partition is failing at the same time or the lineage chain is long. Users can thus rely on storing essential RDDs on disk (or at checkpoints) to avoid unnecessary recomputation in large jobs.

Spark runs standalone with its own cluster manager, or it can reside on top of YARN or Mesos [4], [10]. For a large number of users, YARN in a cluster with Hadoop provides a powerful way to combine data storage (HDFS) and in-memory analytics (Spark) in one cluster. But Spark's performance largely hinges on memory availability; if the dataset is not in memory, then Spark's advantage decreases.

PERFORMANCE: ITERATIVE TASKS VS. SINGLE-PASS BATCH JOBS

Many benchmarks and real-life articles highlight workload profiles when deciding to use Spark or Hadoop MapReduce. But Spark's unique strength appears in iterative operations, where you use the same partitions over and over again:

1. **Machine Learning Pipelines**: Algorithms like k-means clustering, logistic regression, or collaborative filtering often require multiple passes over the entire dataset to refine model parameters. Spark cuts down latency significantly, since intermediate states remain cached between iterations, rather than being written to disk [2], [3].
2. **Interactive Analytics**: When data scientists explore large datasets, repeatedly applying transformations, joins, or filters, Spark's in-memory engine supports fluid, low-latency queries, making interactive notebooks and iterative exploration feasible. Hadoop's disk-based approach would slow such iterative queries.

3. **Graph Processing**: Iterative graph algorithms like PageRank repeatedly update ranks for nodes across multiple rounds. Storing the graph in memory obviates reloading it from disk after each round, yielding substantial speedups [2].

Hadoop can stay competitive for tasks that need to crawl every dataset just once. Large batch transformations, like nightly data aggregations or log parsing usually happen all at once. Spark caches data only once, which will not provide any significant performance benefit since each record only gets read once [4]. The advantage of Hadoop in such a case is a lesser memory footprint and a predictable, well-formed, easier to use ecosystem for linear ETL pipelines.

The following table provides a comparison of the two frameworks in the iterative versus single-pass job roles:

Job Type	**Spark**	**Hadoop MapReduce**
Iterative ML or Graph Workloads	Typically outperforms due to cached data reuse	Suffers from repeated disk read-write overhead
Interactive Data Exploration	Lower-latency queries, memory-based transformations	High latency for repeated queries
Single-Pass ETL or Batch	May be overkill if data is not reused frequently	Often competitive or sufficient for linear tasks

Research done prior to 2022 often reveals these disparities via benchmarks that quantify job times, shuffle data volumes, and CPU usage [5], [6]. In aggregate, these studies support the view that Spark's memory-based engine thrives on iterative or asynchronous data manipulation while Hadoop can be maintained for simpler, large-scale transformations.

SHUFFLE, SCHEDULING, AND FAULT TOLERANCE IN DEPTH

Shuffle Overhead and DAG Optimization

Shuffle overhead (cost of redistributing intermediate data among nodes) is often a large component of distributed computing performance. The MapReduce's code requires a shuffle step after the map step that is read by the reduce jobs [1]. Spark's DAG optimizer can trim or combine shuffle steps across multiple transformations, decreasing the amount of time data writes to disk or is carried over the network [3]. Yet Spark has to shuffle data even when transformations such as groupBy or reduceByKey require changing data partitions. The difference is that Spark can join stages together and store intermediate results in memory instead of repeated disk writes like MapReduce does.

But if Spark is processing a dataset larger than the available RAM of the cluster, it might perform spill-to-disk operations, giving you a performance that is similar to a disk system. This scale makes Spark a little less useful, especially for environments with tens or hundreds of terabytes, where optimal partitioning or additional hardware is essential.

Scheduling and Concurrency

Hadoop's MapReduce tasks get slots in YARN and every map/reduce task uses a memory and CPU space [4]. On YARN, Spark submits driver and executor processes, each with multiple tasks slots. Spark is better at assigning workloads, but it also holds resources for the entire application lifecycle, which might cause conflicts with other workloads if not managed well. This can be a problem in multi-tenant clusters where multiple teams are concurrently running Spark apps, and can create memory pressure and queue issues.

Stand-alone Spark clusters or Mesos-based implementations work the same way but can have different dynamic resource allocation mechanisms. Whether Spark or Hadoop is better in a concurrency situation depends on how tasks are overlapping, the memory usage of each application, and scheduling requirements. Some organisations use Spark clusters for more sophisticated analytics in order to minimise clashes with legacy Hadoop batch jobs.

Fault Tolerance Strategies

HDFS replication underpins Hadoop data storage fault tolerance [1]. When a node is unavailable, the system assigns an instance with the same block of data to another node. MapReduce jobs running on the crashed node are also re-extended. Spark's algorithm leverages RDD tree rather than replicating in memory. To lose a partition leads to recomputation from previous levels, which is efficient if partial or incremental changes are minimal. If more than one node crashes or the lineage graph is complex, Spark might have to reprocess large portions of the data. Thus Spark and Hadoop take failure well, but approach it in a different way. In batch-driven applications, replication is easy to justify, but lineage-based recovery is best for temporary or short-term projects.

STREAMING AND REAL-TIME ANALYTICS

Real-time analytics constitutes a prominent factor in deciding between Spark and Hadoop. Spark introduced Spark Streaming, a micro-batch framework that processes new data arriving in short time windows (e.g., one second), effectively bridging batch and streaming paradigms with a consistent API [3]. By using RDDs for each small batch, Spark Streaming allows near real-time data processing with latencies in the sub-second to multi-second range, sufficient for many enterprise monitoring or IoT use cases. Integration with Kafka or Flume streamlines ingesting data from logs, sensors, and message queues.

Hadoop MapReduce itself lacks a native streaming model. Early streaming scenarios in the Hadoop ecosystem employed tools like Storm or Samza, each providing record-at-a-time processing. These external tools integrate with HDFS, but the flow from stream ingestion to batch jobs can be more fragmented [10]. While some parts of the Hadoop ecosystem, like Kafka, can feed data into real-time pipelines, the MapReduce engine remains disk-based and is not typically used for immediate streaming transformations. In contexts where micro-batch or near real-time analytics is a priority, Spark often stands out as an all-in-one solution with both batch and streaming capabilities. Nonetheless, specialized streaming engines might be more appropriate if ultra-low latency (milliseconds) is required, since Spark's micro-batch intervals can introduce slight delays.

ECOSYSTEM TOOLS AND INTEGRATION

Broader Hadoop Ecosystem

The term "Hadoop" often refers not only to MapReduce, but to a wide range of complementary applications and projects. Hive offers a SQL-like interface for converting queries to MapReduce (or Tez) jobs, Pig offers a data flow language, HBase supports fast NoSQL data storage, and Oozie supports workflows [4], [9]. Gradually, Apache Tez offset some of the brute force overhead of MapReduce by providing a DAG-based flow that accelerates Hive queries and bridges the gap with Spark's performance. But this improvement still involves a disk-based intermediate output mechanism, albeit one that is more pipelined than the old MapReduce. Such changes are a sign that Hadoop isn't a one-time thing, but it has been attempting to overcome some of its previous performance limitations.

Spark's Unified Stack

Spark's architecture wants to merge many workloads into one engine. Spark SQL — Structured queries; Spark Streaming — Micro-batch, MLlib — Machine Learning, and GraphX — Graph analytics [3]. These modules all implement the same runtime, so an RDD or DataFrame datastore can be reused for incremental transformations, SQL queries, or streaming windows. This integration can ease the creation of code and alleviate the burden of data format conversions. Data scientists often choose Spark because they can code in Python, Scala or Java and leverage a single platform to perform batch and streaming operations.

On real-world deployments, Spark jobs will read or write from HDFS, essentially using Hadoop's file system for storage but Spark for in-memory execution. This integration enables companies who already own large Hadoop clusters to leverage Spark modules for advanced analytics. Some teams opt instead for private Spark clusters with short-term or cloud-based provisioning, and rely on S3 or another object store as the data-in-motion backend.

COST CONSIDERATIONS AND RESOURCE OPTIMIZATION

Although performance is a primary factor for iterative computations, maintaining a Spark cluster with high memory usage can be very expensive, especially on pay-as-you-go clouds [7], [8]. A Spark application that runs half the time of a similar Hadoop job might not necessarily be less expensive if the instances you need cost twice or three times as much RAM. It is a workload dependent ratio between the length of work and the cost of resources. If iterative analytics or repeated scans are the heaviest drivers through the pipeline, the accelerations offered by Spark can make room for more advanced (and hence more expensive) hardware. On the other hand, if tasks are mostly single pass, the memory premium associated with in-memory data structures may not be sufficient to make up for the benefit.

Administrators in on-premises environments also consider the overhead of creating nodes with large RAM footprints and more powerful networks, which can be critical to making the most of Spark's abilities. Hadoop, on the other hand, can be efficiently used on clusters with low RAM, especially on mass-market hardware with large disks. Some organizations use a combination, leaving a portion of the cluster for Spark workloads requiring more memory and parallelism, but keeping Hadoop MapReduce up and running for more frequent nightly ETL processes without the advantages of iterative caching.

This combination of accelerated analytics and expense means that Spark and Hadoop aren't all "cheaper." Each situation requires sizing carefully. Spark may decrease the amount of hours required to complete large jobs, but memory-intensive nodes or cluster configurations could create overhead, particularly if the cluster is heavily underused in the off-peak.

REAL-WORLD DEPLOYMENTS AND HYBRID STRATEGIES

Organizations rarely adopt one structure solely in isolation from the other. Many use Hadoop as a persistent data lake where raw logs, sensor output, or transactional archive is stored on HDFS. They deploy periodic MapReduce or Hive jobs to cleanse and compress these data sets. Spark (either executed on the same cluster via YARN or on a separate dedicated machine) then runs machine learning, interactive data exploration, or streaming analytics [4], [5]. The multi-tiered approach takes advantage of Hadoop's durability and batching capabilities and uses Spark's performance for specific higher-level processing.

One bank could use Spark to run iterative risk modeling on daily triggered trade data in a manner that dramatically reduces the time required to run this process over thousands of historical data chunks. Meanwhile, regular batch jobs for nightly data cleansing and compliance reporting could be executed on Hadoop, minimizing memory consumption for simple linear tasks. For example, an enterprise shopping site could implement Spark Streaming for real-time clickstream processing, which would feed near-real-time dashboards or recommendation engines, but continue to use MapReduce for monthly or quarterly archive jobs that pull together general performance data.

Such real-world examples aspire to a complementary pattern. Far from entirely supplanting Hadoop, Spark is often an add-on to handle complex analytics and time-sensitive workloads. In multi-tenant clusters, resource control is an issue as big memory Spark jobs are in conflict with huge Hadoop MapReduce jobs when they are simultaneously running under YARN. Some companies avoid these conflicts by giving Spark dedicated subsets of nodes or committing resource-hungry jobs at specific times.

EXPANDED COMPARATIVE INSIGHTS: A TABULAR OVERVIEW

The table below attempts to integrate the various aspects of Spark vs. Hadoop under a more comprehensive lens. It builds on earlier comparative points but includes broader considerations, from concurrency management to the typical user base for each framework.

Dimension	**Apache Spark**	**Hadoop MapReduce**
Execution Model	In-memory DAG-based transformations; merges multiple steps to reduce shuffle	Disk-centric map–shuffle–reduce; each phase writes intermediate data to local disk
Iterative ML & Graph	Demonstrates large speedups through data caching, DAG fusion	Bottlenecked by repeated data reload from disk after each iteration
Fault Tolerance	Lineage-based; recomputes lost partitions, usually more efficient for ephemeral transformations	Replication-based via HDFS; tasks restart on different nodes holding replica blocks
Streaming	Spark Streaming with micro-batches integrated into a unified engine	No direct streaming in MapReduce; alternative frameworks like Storm or Samza used for real-time
Resource Management	Standalone, Mesos, or YARN; memory tuning crucial for best performance	Typically managed by YARN; simpler memory usage but can be less flexible in iterative scenarios
Memory Overhead	High; benefits vanish if cluster insufficiently provisioned	Lower overall memory demands, oriented around commodity hardware
Shuffle Optimization	Fused stages, in-memory partial aggregations reduce shuffle cost	Mandatory shuffle after map tasks, dependent on local disk I/O and sorting
Use Case Spectrum	Iterative analytics, near real-time processing, interactive data science workflows	Single-pass large-scale ETL, archiving, and stable batch operations

Typical Cost Dynamics	Potentially higher if large memory nodes are required; offset if iteration speeds reduce job duration	Usually stable for batch tasks; may become time-consuming for iterative or interactive queries
Ecosystem Integration	Unified engine (SQL, streaming, MLlib), easy data sharing via RDD/DataFrame cache	Broad ecosystem (Hive, Pig, HBase, Tez, Oozie), requires more bridging for streaming and advanced analytics

That table highlights the central story from previous research: Spark is all about iterative performance and real-time scalability, while Hadoop MapReduce is best suited for continuous, one-step batch processing, and runs faster on commodity hardware. Most manufacturing processes use both to strike the right balance between price and efficiency.

FUTURE TRENDS AND EVOLVING TECHNOLOGIES

Spark and Hadoop were continually refined and expanded. Hadoop's introduction of Tez, the migration to YARN as a general resource manager, and the inclusion of Hive on Tez all made interactive queries run more efficiently, which mitigated some of the latency from old-school MapReduce [9]. Spark, meanwhile, has improved its DAG scheduler and also worked with container orchestration platforms like Kubernetes, expanding its deployment capabilities [3].

Managing the cluster in the cloud is standard now, abstracting out much of the cluster configuration from end users. Such services as Amazon EMR, Google Dataproc, Azure HDInsight provide Spark and Hadoop on a pay-as-you-go basis, automatically scaling resources according to workload requirements [7], [8]. In such services, raw performance becomes sometimes a matter of cost and simplicity. In most cases, users specialize in transformations of data using upper level interfaces (Spark SQL, Hive, etc. and ask the cloud provider to maximize the cluster's internal resources.

Also, scientists are trying to use high-end hardware accelerators, like GPUs or even special FPGAs, to accelerate ML workloads. Spark has made some attempts to bring GPU scheduling into deep learning, but Hadoop's old MapReduce model is primarily disk- and CPU-bound [3]. HPC solutions typically use Spark-like models for incremental computation, though niche HPC systems can provide even lower latency and more direct GPU support.

Despite these evolutions, the fundamental difference between in-memory DAG and disk-based batch flow still influences design. Spark's streaming micro-batching is still an ideal choice for near real-time analytics; for sub-second or millisecond-latency streaming, users might opt for Flink or Storm. Hadoop's consistency across data lakes keeps it in use for general archiving and batch processing work — and is often the main storage layer that Spark can connect to.

CASE STUDIES AND OBSERVED BEST PRACTICES

Examples from the pre-2022 literature point to large scale technology firms, financial institutions, and research consortiums implementing Spark and Hadoop in their overall data plans. One example: An enterprise with a longstanding Hadoop infrastructure adopts Spark in small doses for new workloads, like iterative data science or near real-time analytics [4], [5]. They hold on to Hadoop for steady-state batch operations that are not cached in memory (such as monthly aggregator jobs that parse logs and store aggregated metrics).

In the financial industry, Spark allows you to detect fraud in real time by running streaming analytics on Kafka transaction data. That same data, after initial real-time checks, could be replicated to HDFS for full-batch MapReduce analysis to create compliance summaries or monthly risk reports. By coordinating both frameworks via YARN or independent clusters, banks benefit from low latency detection while utilizing Hadoop for low-cost archiving.

Some online stores emphasize the alignment between Spark and streamed structured data. Customer behavior reports come in daily; Spark executes micro-batches to deliver product suggestions or targeted offers almost real-time. Big-picture inventory or historical sales data, in the meantime, are stored in Hadoop and periodically parsed for big-picture patterns or offline model training. If Spark has in-memory capabilities, interactive or iterative queries can run at speeds that make dashboards and recommendations viable.

Best practices for such deployments are typically focused around resource tuning, partitioning, and workload management. Practitioners frequently suggest:

Best practices from these deployments typically revolve around resource tuning, partition strategies, and workload scheduling. Practitioners frequently suggest:

- Allocating ample memory to Spark executors for tasks requiring data caching.
- Employing partitioning schemes that align with shuffle boundaries and data distribution, reducing data skew and network overhead.
- Splitting repetitive or time-sensitive workloads (where Spark shines) from stable batch processes (handled well by Hadoop), possibly scheduling them in different YARN queues or times.
- Monitoring cluster performance with real-time metrics, adjusting memory sizes, shuffle concurrency, and parallelism settings in Spark to reduce disk spills and optimize DAG execution.

These operational insights confirm that it is rarely about Spark fully displacing Hadoop or vice versa. Instead, it is about harnessing each framework's strengths and mitigating their weaknesses.

CONCLUSION

Apache Spark and Hadoop MapReduce, which are equally important in the big data era, are different in terms of execution philosophy: Spark's in-memory DAG-style engine emphasizes minimal disk I/O and fast iterations, while Hadoop's disk-oriented MapReduce promises durability and ease of deployment for massive, one-step batching. Studies prior to 2022, on the whole, confirm that Spark is best suited for iterative machine learning and near real-time data streaming because it keeps partitions of data cached in memory between iterations [2], [3]. Hadoop on the other hand fits organizations very well when tasks are more linear, data is greater than achievable in-memory caching, and batch jobs are stable enough to run one time [1], [4]. This follows a traditional enterprise pattern as most companies store data in HDFS, perform daily ETL via MapReduce, and leverage Spark for complex analytics, streaming, or machine learning.

This paper has included several sections describing how the different characteristics of each system appear in terms of fault tolerance, shuffle overhead, concurrency, streaming integration, and cost. Spark's lineage-driven recovery is handy for temporary computations, but HDFS replication offers a more streamlined way to recover very large batch workloads. Spark micro-batching supports almost real time processing, and Hadoop streams via external resources (Storm or Samza). While Spark's memory heavy execution typically requires high end hardware or dedicated instances in the cloud, the benefits of its iterative performance can make up for that under appropriate conditions [5], [7], [8]. Where companies aren't getting the advantage of iterative caching, Hadoop's memory footprint and pipeline ease is up for grabs.

In practice, the models have somewhat merged. Hadoop introduced YARN and Tez, for more customisable DAG processing and resource pooling, while Spark sits seamlessly on top of YARN, using HDFS to store the data. These boundaries get even fuzzier when cloud vendors now provide managed big data solutions that perform cluster provisioning, meaning that there is less of the user's effort required to configure Spark or MapReduce individually. However, the fundamental contrast (Spark's in-memory cache vs Hadoop's disk-based flow) remains the driving force behind architectural choices.

The overall lesson here is that Spark or Hadoop MapReduce does not have a single edge. Rather, each fits certain workflow patterns. Spark performs efficiently for interactive or iterative analytics that produce immediate insights, making it a key tool for data science, dashboards and iterative changes. In large-scale data sets requiring predictable, single-pass ETL, Hadoop remains cost-effective and manageable. Real-world data ecosystems combine both frameworks to support the array of analytics workloads that exist across entire data lifecycles. Knowing how these dynamics play out, users can run Spark and Hadoop together, taking advantage of memory-based performance when desired and keeping massive disk-based batch execution when that's sufficient. This holistic approach will ensure that businesses get the most out of each tool's own strengths in tackling the increasing seas of emerging big data.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 15–28.

[3] M. Zaharia, P. Wendell, T. Das, and A. Dave, "Spark: A unified analytics engine for large-scale data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[4] D. Borthakur, J. S. Sarma, O. O'Malley, S. Radia, B. Reed, and K. Shah, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ACM, 2013, p. 5.

[5] P. Faris, N. Yusof, and M. Othman, "Performance analysis of big data in Hadoop," in *Proceedings of the 2018 Third International Conference on Informatics and Computing (ICIC)*, IEEE, 2018, pp. 1–5.

[6] A. K. Tiwari, "Comparative study of big data computing and storage tools: A review," in *2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech)*, IEEE, 2016, pp. 106–110.

[7] M. A. Ferrag, L. Maglaras, S. Moschoyiannis, and H. Janicke, "Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study," *Journal of Information Security and Applications*, vol. 50, p. 102419, 2020.

[8] V. Behzadan and W. Hsu, "Adversarial exploitation of policy learning in autonomous systems," in *Proceedings of the 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2017, pp. 281–288.

[9] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.

[10] L. A. Maglaras, J. Jiang, M. A. Ferrag, Z. Xia, and H. Janicke, "Cybersecurity solutions for the Internet of Things: A survey," *Transfers in Internet and Information Systems*, vol. 13, no. 1, pp. 1–20, 2019.