Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.063047

REVIEW

A Survey of Spark Scheduling Strategy Optimization Techniques and Development Trends

Chuan Li and Xuanlin Wen*

Department of Computer Science and Technology, School of Computer Science, Xi'an University of Posts and Telecommunications, Xi'an, 710100, China

*Corresponding Author: Xuanlin Wen. Email: 731148488@stu.xupt.edu.cn

Received: 03 January 2025; Accepted: 06 March 2025; Published: 19 May 2025

ABSTRACT: Spark performs excellently in large-scale data-parallel computing and iterative processing. However, with the increase in data size and program complexity, the default scheduling strategy has difficulty meeting the demands of resource utilization and performance optimization. Scheduling strategy optimization, as a key direction for improving Spark's execution efficiency, has attracted widespread attention. This paper first introduces the basic theories of Spark, compares several default scheduling strategies, and discusses common scheduling performance evaluation indicators and factors affecting scheduling efficiency. Subsequently, existing scheduling optimization schemes are summarized based on three scheduling modes: load characteristics, cluster characteristics, and matching of both, and representative algorithms are analyzed in terms of performance indicators and applicable scenarios, comparing the advantages and disadvantages of different scheduling modes. The article also explores in detail the integration of Spark scheduling strategies with specific application scenarios and the challenges in production environments. Finally, the limitations of the existing schemes are analyzed, and prospects are envisioned.

KEYWORDS: Spark; scheduling optimization; load balancing; resource utilization; distributed computing

1 Introduction

With the rapid development and large-scale popularization of digital technology, the demand for large-scale data computation and analysis in various industrial fields has been growing exponentially, in which the efficiency of big data processing is one of the focuses of attention in industry and academia, and distributed strategies have been widely used in big data efficiency optimization. In applications with complex dependency-constrained processes and scenarios with dynamically changing data volumes, some distributed frameworks, such as MapReduce, Flink, and Spark, have shown remarkable adaptability and superiority, which split a large-scale data job into multiple computational tasks and schedule them to be executed on different nodes of a cluster in parallel [1]. Among them, Spark is based on in-memory computing, unifies batch and stream processing, has a vast ecosystem [2], and has become a crucial choice for big data processing.

Apache Spark is an open-source big data processing framework that abstracts datasets into resilient distributed datasets (RDDs), divides the computation process into several phases based on the dependencies between the RDDs, and further decomposes each phase into multiple tasks and assigns them to nodes to realize the parallel computation, and finally choosing to aggregate or directly store local results according to specific needs [3]. RDDs can be stored in memory for extended periods and can be directly accessed from

memory when reused, eliminating the need to recreate or retrieve them from disk [4]. This feature enables Spark to efficiently reuse recently created intermediate results, making it well-suited for iterative problems and scenarios with low latency requirements [5].

However, in recent years, the data size and program complexity have been rising, and the rapid update of hardware resources has made the phenomenon of the uneven computing power of cluster nodes more and more serious [6]. Spark framework is facing multi-faceted and multi-level optimization needs in terms of efficiency. Reference [7] summarizes and analyzes the current domestic and international research on Spark optimization techniques from five aspects: development principles, memory usage, configuration parameters, scheduling strategies, and the Shuffle process. However, the reference does not propose a more detailed classification strategy for a certain aspect of the Spark execution process based on different scenario characteristics and optimization objectives, such as classifying the storage and extraction optimization strategy for the initial data or classifying the scheduling strategy.

HDFS (Hadoop distributed file system) is often used as a distributed file system responsible for storing the data to be processed for Spark. Reference [8] summarizes the existing HDFS optimization techniques from three dimensions: file logical structure, hardware devices, and application load for the Spark data access phase. The optimized HDFS provides faster data read and write speeds, higher throughput, and fault tolerance for Spark. However, the Reference does not consider the dynamic resource allocation and load scheduling during job execution. Reference [9] analyzes the efficiency optimization strategy of Spark's general-purpose environment at the JVM level in terms of memory management, cluster collaboration, storage of data objects, GC algorithm, and hardware assistance. By analyzing the object survival time and resource utilization on JVM, we can judge the macro status of cluster resource allocation to a certain extent. However, the factors affecting resource utilization will change depending on the application, stage, and node, which is difficult for the underlying system to adapt to, and paying too much attention to the underlying principles of the complex system may reduce the attention to the actual application scenarios, weakening the mastery of the overall resource utilization of the cluster, utilization of the cluster as a whole.

The scheduling strategy is closely integrated with the actual application scenarios and has a significant impact on performance [7], which can be directly adjusted based on changes in business requirements and scenarios to guide program design and effectively coordinate the reasonable allocation of cluster resources from the macro level. Therefore, this paper analyzes the Reference on Spark scheduling strategy optimization in recent years and summarizes the scheduling strategy from three aspects, namely load-based, node-based, and based on the matching of the two, according to the different perspectives when the strategy is designed.

2 Basic Theory of Spark

This section provides an overview of Spark's basic runtime logic and a foundation for understanding Spark scheduling principles. It also compares the limitations of common Spark resource managers and their default scheduling algorithms. Finally, it analyzes how scheduling strategies affect the execution efficiency of Spark programs.

2.1 Basic Operation Logic of Spark

The entry point of a Spark [10] application is the Driver program, which is responsible for initializing the SparkContext object as a proxy for communication with the entire cluster. After the user submits the program to the Spark cluster through SparkContext, the SparkContext typically reads data from storage systems such as HDFS based on the specified file path. As shown in Fig. 1, the application reads the data, creates an initial Resilient Distributed Dataset (RDD), and splits it into multiple partitions. The RDD undergoes a series of transformation operations (e.g., map, reduce) to form a chain of RDD transformations, generating a Directed

Acyclic Graph (DAG) as a job. The DAG Scheduler primarily divides the entire job into multiple stages based on whether the RDD data undergoes re-partitioning (shuffle). Narrow dependency operations, such as map, filter, pass data from each partition of the parent RDD to a specific partition of the child RDD, or, through flatMap, map each input partition to multiple output partitions. The data remains local but is logically mapped to multiple sub-items. These operations do not trigger data re-partitioning and therefore will be assigned to a single stage. In contrast, wide dependency operations like reduceByKey, and join, cause data to be transferred from one partition to another, and this data movement triggers a shuffle. As a result, the scheduler will use this as a boundary to divide into new stages. Additionally, resetting the number of partitions and some data skew correction algorithms can also lead to shuffle operations. Each stage is further divided into multiple tasks, which are packaged into a TaskSet, with each task running on a single partition. The Task Scheduler receives the TaskSet and assigns the tasks to available Executors in the cluster according to scheduling policies such as FIFO. The Resource Manager manages and allocates computing resources in the cluster throughout this process.

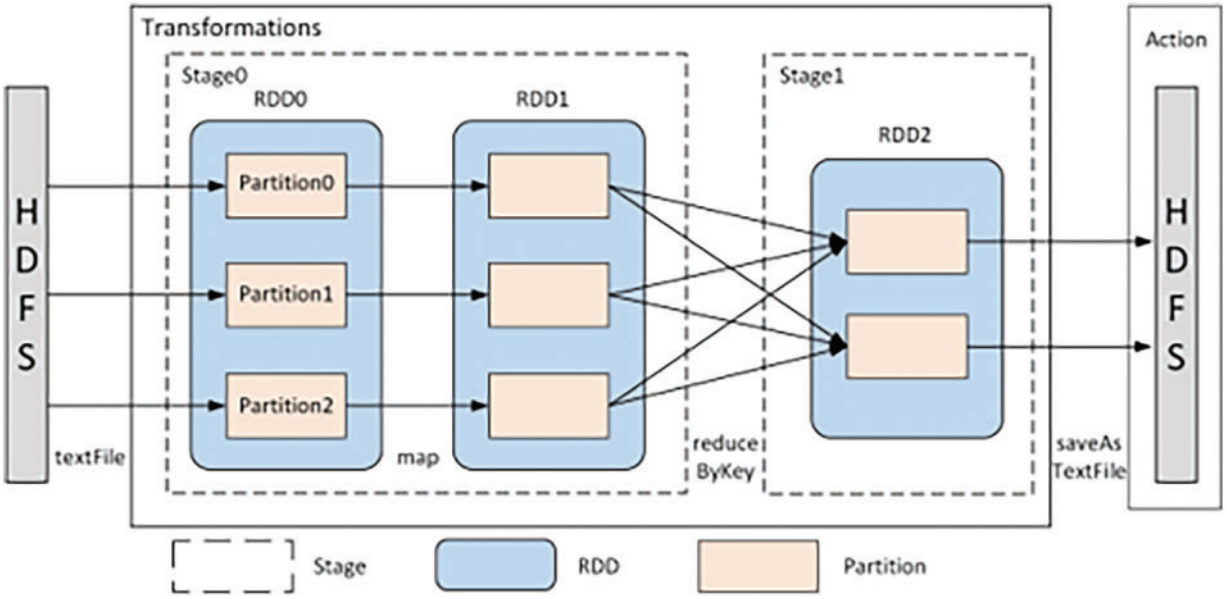

Figure 1: Spark basic execution logic diagram

2.2 Resource Manager and Its Default Scheduling Strategy

Resource managers can provide efficient and flexible resource management functions at multiple levels in a multi-job and multi-user environment. The resource managers supported by Spark include Standalone, YARN, and Mesos [11]. Among them, Standalone is the default manager for Spark, but due to its relatively poor performance in terms of resource management flexibility, multi-user support, and scheduling policies, it is more suitable for the simple configuration and management of small clusters. YARN is a general-purpose resource management system and scheduling platform that can dynamically and flexibly manage resources in a cluster. By supporting multiple scheduling strategies, such as FIFO, Capacity Scheduler, and Fair Scheduler, YARN can meet resource management requirements in different scenarios and supports the simultaneous operation of multiple users and multiple frameworks. Mesos dynamically allocates CPU, memory, and other resources among multiple frameworks through a resource offer mechanism and a fair resource allocation algorithm (e.g., Dominant Resource Fairness, DRF) to ensure efficient and fair resource utilization. While

similar to YARN in terms of architecture and functionality, Mesos offers greater versatility and is capable of managing resources for multiple frameworks and applications.

FIFO [12] is the simplest and most basic scheduling strategy, where the scheduler queues all jobs uniformly and schedules resources in the order of submission. If the current job does not occupy the entire cluster, the scheduler allocates these idle resources to subsequent jobs, achieving a certain degree of parallelism. FIFO is suitable for scenarios with limited resource demand and more balanced resource requests, but when jobs are heterogeneous, long jobs will occupy more resources and time. FIFO cannot adjust the order of resource allocation according to the priority or importance of the jobs, resulting in long waiting times for subsequent tasks.

Capacity [13] establishes multiple job queues with defined capacity ratios to plan resource allocation, and manages resources controllably through pre-defined resource capacities and queue management mechanisms, avoiding serious resource grabbing. However, there is some subjectivity in users' setting of capacity guarantees and limits, even though the system can flexibly use idle resources from other queues to some extent, the accuracy of the prediction is difficult to guarantee, which can easily lead to resource wastage, etc.

Fair [14] allows users to define queues based on job requirements. In addition to setting the minimum shares for each queue, it also sets queue weights based on user needs. The larger the weight, the more resources the queue will obtain under the same conditions. Therefore, jobs have fairness in resource acquisition. In a multi-tenant environment, jobs with different priorities or resource requirements can compete and fairly utilize resources. However, in a multi-resource environment, Fair may not handle the complex relationships between multiple resources well, leading to some unbalanced resource allocation.

DRF [15] derives a resource quota vector for each object by calculating the dominant resource requirements of a job or queue and allocates the resources proportionally or by priority. When the resource requirements of jobs vary significantly, DRF can ensure that the occupancy ratios of different resources remain fair. This process may have problems such as proportionality bias, reflecting the actual resource requirements not accurately enough, and mixing of multiple resource types that cannot fully and fairly satisfy the resource requirements, leading to room for improvement in the efficiency optimization of the framework.

Table 1 compares the default scheduling algorithms of the four managers. FIFO adopts a simple first-come, first-served strategy, suitable for scenarios with homogeneous workloads and simple resource demands. Both Capacity and Fair are multi-queue resource allocation strategies, with the key difference being that Capacity predefines the minimum and maximum queue capacities, making it suitable for scenarios requiring fixed resource reservations, while Fair dynamically adjusts queue weights based on fairness principles, making it suitable for priority scheduling scenarios with complex urgent tasks. DRF ensures fair use of multiple resources through "dominant resource" allocation, making it suitable for complex computing environments constrained by multiple resources such as CPU, memory, and bandwidth.

Table 1: Comparative analysis of common spark scheduling strategies

Scheduling strategy	Algorithm description	Limitations	Suitable scenarios
FIFO	Allocates resources in the order of job submission	Long tasks consume large amounts of resources, making it difficult to handle sudden high-priority jobs	Scenarios with balanced resource demands and no obvious priority distinctions

(Continued)

Table 1 (continued)

Scheduling strategy	Algorithm description	Limitations	Suitable scenarios
Capacity	Allocates resources based on predefined resource capacities and multi-queue management	Inaccurate resource demand estimation or significant demand fluctuations may lead to resource idle	Multiple queues manage resources separately, and resource demands are stable
Fair	Allocates resources based on resource capacities and weight settings	Unbalanced resource allocation in heterogeneous clusters	Few types of resource demands with unstable requirements
DRF	Allocates resources based on the dominant resource demand ratio of each task	Struggles to respond promptly to drastic changes in resource demand	Scenarios with multiple stable resource demands

2.3 Common Performance Metrics for Evaluating Scheduling Strategies

In the Spark distributed computing framework, the performance metrics commonly used to evaluate the effectiveness of scheduling strategies include load execution time, resource utilization, system throughput, and task failure rate, as well as some derived metrics.

2.3.1 Load Execution Time

Spark load execution time includes job completion time, stage completion time, and task completion time, it is the most intuitive performance metric for evaluating scheduling strategy efficiency. Execution time is influenced by a combination of other performance metrics and external factors. When scheduling improves resource utilization and system throughput or reduces task failure rates, the execution time of different Spark workloads decreases to varying degrees, demonstrating the effectiveness of the scheduling strategy. Additionally, an important metric in distributed systems, speedup, can be calculated based on execution time and parallelism (number of nodes or processes).

In Spark, the SparkListener mechanism monitors certain execution details of a program. By recording and calculating the start and end timestamps of a workload, it determines the execution time, which is then displayed in the Spark UI or History Server.

2.3.2 Resource Utilization

Resource utilization reflects the efficiency of Spark's scheduling strategy in utilizing cluster resources, specifically whether the allocated resources are fully utilized. If scheduling fails to fully utilize cluster resources, it can lead to idle resources and waiting time, increasing unnecessary queuing time for workloads and thus affecting efficiency.

Resource utilization mainly includes the utilization rates of CPU, memory, and transmission channels (such as network bandwidth). CPU utilization is typically calculated as the proportion of Executor CPU Time to the total available CPU time for tasks, as collected by monitoring tools. Memory utilization is determined by the ratio of actually used memory to the total allocated memory. Transmission channel utilization is calculated based on the ratio of actual transmission rates, such as network bandwidth, to their theoretical maximum values. These runtime data can be found on the executor page and memory page of the Spark UI.

2.3.3 System Throughput

Spark system throughput is primarily used to measure the system's capability to complete operations such as data loading, communication, and caching within a unit of time, reflecting the overall processing capacity of the system. Throughput can be measured and calculated based on the amount of data processed per unit time, including key parameters such as input data volume, output data volume, and shuffle data volume. These metrics can be viewed on the Jobs page of the Spark Web UI or manually calculated from log files.

2.3.4 Task Failure Rate

During job execution, the ratio of failed tasks to the total number of tasks represents the task failure rate, which reflects the stability and fault tolerance of the scheduling strategy. A high task failure rate indicates unreasonable resource allocation or network congestion, which can significantly extend the execution time of the overall job or stage, and may even require resubmission. The measurement method is consistent with the aforementioned metrics, where failure counts are collected by monitoring tools and can be viewed on the Jobs page of the Spark UI or analyzed using external log analysis tools.

2.4 The Impact of Scheduling on Execution Efficiency

Scheduling affects the efficiency of Spark program execution from several aspects, involving Data Skew, task parallelism, execution order, data locality, node resource evaluation, and fault tolerance mechanism, leading to different degrees of resource underutilization, scheduling delays, and low throughput.

2.4.1 Data Skew

Data skew refers to the situation in Spark jobs where certain partitions contain significantly more data than others, leading to an uneven task load. This results in varying execution times across partitions, where some tasks may be completed earlier, leaving resources idle while waiting for other unfinished tasks. Additionally, data skew significantly increases network transmission and disk I/O overhead during the Shuffle phase. When an imbalanced distribution encounters wide dependency operations, data from overloaded partitions must be scheduled and transferred to other partitions in sequence, further increasing transmission costs.

2.4.2 Parallelism

Task Parallelism refers to the number of concurrent executions of a task, which defines the number of tasks to be scheduled and is based on the RDD's partitioning. When the number of tasks is increased, tasks can be processed by more Executors simultaneously, thus improving execution efficiency. However, Spark, as a parallel computing framework, has a parallelism bottleneck, which, to some extent, aligns with Amdahl's law as expressed in the following formula:

$$S = \frac{1}{1 - P + \frac{p}{N}} \tag{1}$$

where S is the speedup ratio when the parallelism is N, p is the proportion of the parallelizable part of the computation, and N is the number of processors or threads (number of nodes) for parallel computation. It should be noted that Spark's task parallelism is not equivalent to the number of nodes, but Amdahl's law discusses the impact of the non-parallelizable part of the program and the degree of parallelism on the speedup ratio. Non-parallelizable computation includes serial computation that relies on the results of the

previous task, data transmission in the network and coordination between nodes, and I/O operations in non-distributed storage systems [16], while a Spark task may contain all of the above operations. For example, when parallelism increases, although the degree of simultaneous data execution improves and smaller, denser data blocks reduce resource waiting time caused by imbalanced execution, the management overhead of scheduling and transmission also increases. This includes connection establishment, transmission control information, message acknowledgment, memory caching, and garbage collection. These additional overheads prevent execution efficiency from increasing indefinitely with parallelism, as they belong to the non-parallelizable computation part in the formula. Therefore, the relationship between the task parallelism, the number of nodes, and the efficiency conforms to the description of Amdahl's law to a certain extent after certain transformations and derivations in the understanding and application. This formula can be used to guide partitioning, predict the performance improvement limits of the system, and help researchers balance resource investment and performance returns to enhance resource utilization.

2.4.3 Execution Order

DAG is used to describe the structure and logical flow of the Spark execution program, where RDDs may have complex dependencies, especially in wide dependencies, where tasks depend on the output data of the preceding tasks, and if the execution of a preceding task is delayed, the downstream task must wait. The execution order may lead to unnecessary scheduling delays, reduce load waiting time or unnecessary lag time, and improve overall execution efficiency.

2.4.4 Data Locality

Data locality is one of the key principles of Spark scheduling, which means trying to schedule tasks to the node where the data resides or as close to it as possible. This is essentially related to network transmission overhead, unreasonable long-distance scheduling will consume a significant amount of time and network bandwidth, making it highly prone to performance bottlenecks. As shown in Fig. 2, Spark defines locality levels from near to far as follows: PROCESS_LOCAL, where tasks and data are in the same JVM process; NODE_LOCAL, where tasks and data are in the same node; RACK_LOCAL, where tasks and data are in the same rack; No_PREF, where locality is meaningless, and the data is stored in an external storage system such as HDFS within the same node; and ANY, where tasks can run on any node, often a cross-rack node. Taking NODE_LOCAL and RACK_LOCAL as examples, when a task executes on the same node, Spark typically uses the Netty network application framework for inter-process communication. In this case, no network transmission is involved; instead, data is transferred via the operating system's loopback interface or even shared memory. When task execution requires cross-node communication but remains within the same rack, communication is generally carried out over Ethernet. Due to limitations such as unstable network bandwidth, the overhead of the network protocol stack, and switch performance, the amount of data transmitted per unit time is lower than that of intra-node communication, resulting in lower throughput.

DAG Scheduler will choose the nearest scheduling target as much as possible to minimize the data transmission overhead, frequent long-distance transmission will reduce the system throughput. In addition, to realize data local processing as much as possible, Spark will adopt a delay scheduling strategy when there is no high-priority local task, expecting to find a task that meets the current locality level by waiting, and this process will lead to a certain waiting time. Therefore, locality directly affects system throughput and latency.

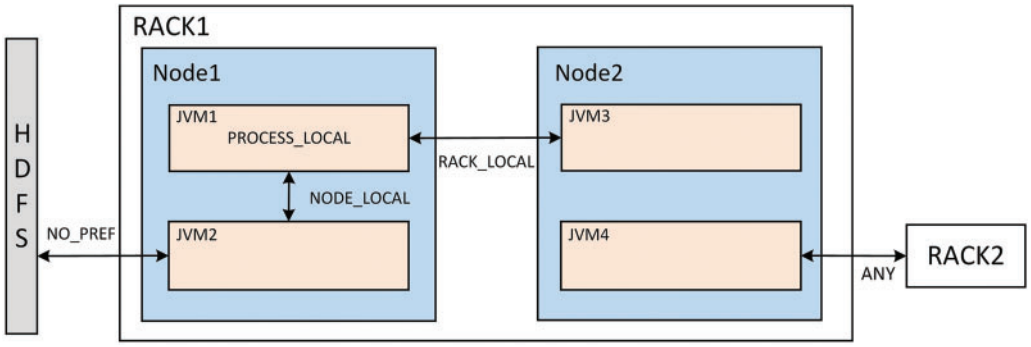

Figure 2: Spark locality diagram

2.4.5 Node Resource Evaluation

To alleviate data skew caused by operations such as initial partitioning, shuffle, or key-value imbalance, Spark schedules tasks to nodes with sufficient resources. The system periodically monitors the resource usage of each node and assigns tasks to the node with the most abundant resources. However, this model performs poorly in scenarios with complex task dependencies, dynamic node loads, or significant changes in computing power, often leading to severe task lag. As a result, some studies propose making long- and short-term predictions of resource and load changes for each node.

In addition to the number of resources, the resource characteristics of the node also affect the efficiency of task execution: CPU-intensive tasks require a large number of computational operations, and the target nodes for such tasks are usually multi-core processors or servers with powerful computational capabilities; memory-intensive tasks require a large amount of memory space for processing and storing data, and the tasks usually include large-scale datasets or tasks requiring frequent access and operation; I/O-intensive tasks rely on a large number of data read and write operations, such as database queries, large-scale log processing, and result saving. When designing the scheduling strategy, it is necessary to analyze the type of nodes and node characteristics of tasks in the application scenario, otherwise, the cluster resource utilization will be seriously affected.

2.4.6 Fault-Tolerance Mechanism

Spark's fault-tolerance mechanism is implemented in RDD and related mechanisms. First is the lineage mechanism, where Spark records RDD lineage (i.e., dependency) information. When partition data is lost or fails, the DAG Scheduler recalculates the failed task by replaying the RDD lineage from the starting data of that stage or the intermediate results persisted in memory (similar to an archive) and schedules it, allowing the program to continue executing normally [2]; however, task retransmission incurs additional time overhead and has difficulty handling abnormally slow tasks. To further enhance tolerance, Spark and the supporting storage system use speculative execution, where if certain tasks are found to be running abnormally slow, Spark launches several replica tasks on other idle nodes, and the result of the first completed replica is taken. When a node completes the task, the remaining unexecuted replicas are immediately released to reduce the number of task recalculations. The two fault tolerance mechanisms are shown in Fig. 3.

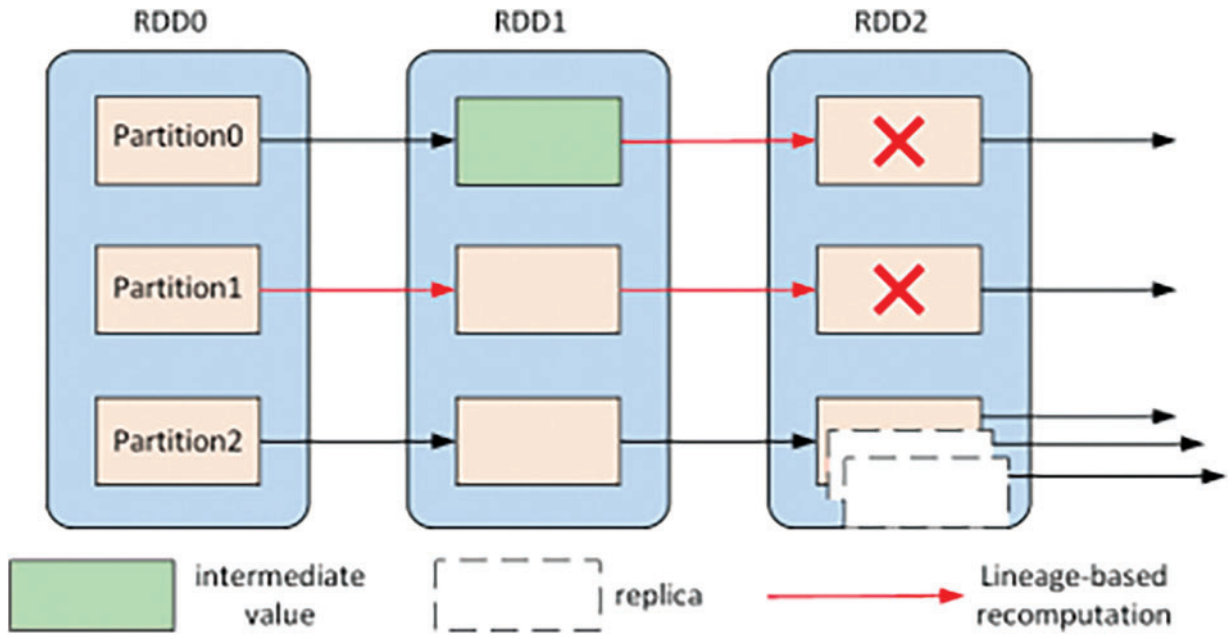

Figure 3: Spark lineage mechanism and speculative execution mechanism

While this mechanism improves reliability, it also introduces additional resource overhead. Firstly, two types of delays are introduced: one is the additional task waiting time caused by pedigree tracking during task recalculation, which becomes significant with long DAG chains; the other is the extra queuing time for replicas, as the redundant replicas increase the total queuing time. Secondly, recalculation and speculative execution occupy resources originally reserved for other queues or jobs. Launching additional processes consumes CPU time slices, memory, and communication channels, which may cause tasks originally scheduled in a certain order to be delayed or executed earlier due to resource competition, reducing execution efficiency. Additionally, persisting intermediate results in memory, while reducing computational load and improving efficiency, requires additional memory space for caching data, which may lead to more frequent GC. The design of fault-tolerance mechanisms should be handled with great care, as a well-designed mechanism can reduce task failure rates but could also increase unnecessary task waiting time.

2.4.7 Multi-Factor Collaborative Optimization Effect

To enhance efficiency, some researchers adopt a combination of multiple optimization strategies to achieve multi-factor collaborative optimization. In some cases, the overall effect can exceed the simple sum of the effects of each strategy applied individually. Certain combinations of the aforementioned factors can complement and reinforce each other.

For example, parallelism and node resource evaluation have a natural complementary relationship. Parallelism determines the number of task splits. If the parallelism is too high, the data volume of each task block becomes small, leading to intense resource competition and high scheduling and transmission costs. If the parallelism is too low, the data volume of each task block becomes large, potentially resulting in the underutilization of cluster resources. Node resource evaluation aims to dynamically assess the resource status of each node in the cluster, thereby providing a basis for task allocation. When these two factors are optimized collaboratively, the number of task splits can be adjusted based on the remaining resources of nodes to avoid resource waste, and the parallelism can be determined based on the bandwidth resources of

nodes to balance scheduling overhead and data movement costs. Parallelism and fault tolerance also have significant potential for collaborative optimization. Under high parallelism, the data volume of task blocks is small, and the computational consumption along the DAG chain of lineage relationships is reduced. When speculative execution is performed, task replicas do not occupy excessive node resources. On the other hand, when important intermediate values need to be solidified, the larger data volume of tasks under low parallelism can effectively retain a significant amount of useful intermediate values, avoiding frequent read and write operations.

Additionally, collaborative optimization involving dual or even triple strategies targeting other factors can also lead to significant improvements in Spark scheduling performance. For instance, combining locality and node resource evaluation can balance data transmission costs while avoiding resource contention on nodes. Combining parallelism, node resource evaluation, and fault tolerance can allocate more resources and replicas to high-priority tasks, preventing resource shortages and ensuring task execution while enhancing fault tolerance.

3 Research Progress on Spark Scheduling Strategy Optimization

To improve program execution time, Spark researchers generally optimize the performance metrics for one or more influencing factors, so it is difficult to directly generalize the Reference from this. Spark monitors the load status through the DAG Scheduler and Task Scheduler and schedules the load based on predetermined policies. The computing resource status of cluster nodes is monitored and managed by the resource manager, which evaluates the node's arithmetic power and schedules resources based on factors such as remaining resources or resource consumption trends. Therefore, scheduling strategies can be summarized from three models based on load characteristics, node characteristics, and matching of the two, which helps researchers and developers to understand the performance advantages, applicable scenarios, and potential shortcomings of different scheduling strategies based on actual scenarios and modules.

3.1 Scheduling Strategy Optimization Based on Load Characteristics

The nodes of homogeneous clusters have similar arithmetic power, so the focus of such scheduling strategy optimization is to accurately evaluate the characteristics of loads such as jobs and tasks, including obtaining accurate data volumes and execution times, dynamic prediction, and analyzing the linkages between loads.

3.1.1 Scheduling Optimization Based on Job Characteristics

A job represents the overall computational logic of an application submitted by a user, often consisting of one or more stages. Its characteristics focus more on comprehensively describing and planning the resource requirements, execution order, and time constraints of large work units. Therefore, job scheduling takes a macroscopic view of resource allocation, considering factors such as fairness, capacity limits, multi-user isolation, and data characteristic isolation to determine the priority, resource quotas, and data processing strategies for different jobs based on business attributes. For example, Fair and Capacity allocate specific resources to jobs from different users or set job priorities based on the value density of different jobs [14], ensuring that certain critical jobs receive higher priority or a larger share of resources.

To optimize the scheduling efficiency of jobs, Reference [17] employs a greedy strategy and a one-step look-ahead strategy, using the shortest job first (SJF) strategy for scheduling when the candidate set of jobs has the same scheduling gain. The proposed PAS algorithm introduces a timeout mechanism to ensure that jobs are prioritized for execution after the waiting time exceeds the predicted value. However, this strategy

may increase meaningless waiting time for long jobs when job lengths vary greatly, or cause delays for high-priority long jobs. Reference [18] defines the computational cost of wide dependencies, resource vacancy rate, and overflow write probability, and demonstrates that task parallelism can impact the execution time of a job. The designed parallelism deduction algorithm (PDA) iterates in each stage to optimize the solution by constructing multiple base datasets, including total data, execution area reservation ratio, operation closure set, and resource table, among others. While the algorithm improves resource utilization, the computation of complex formulas may increase system execution overhead. Reference [19] considers the imbalance and spatial heterogeneity of large-scale air quality data, and during partitioning, divides the data based on spatiotemporal features to improve the prediction efficiency of the distributed random forest algorithm for air quality.

3.1.2 Scheduling Optimization Based on Task Characteristics

A task is the smallest execution unit obtained by subdividing a job, typically corresponding to a partition of the data. It has a finer granularity and a smaller scope, with characteristics that focus more on the specific operational details and execution properties of small-scale data. Task scheduling optimization strategies aim to improve execution efficiency within a single stage, primarily focusing on data locality, fault tolerance, parallelism, and load balancing. The goal is to meticulously plan resource allocation during execution, prevent slow or failed tasks, and optimize transmission costs in a structured manner.

At the task scheduling level, Reference [20] addresses the issue by transforming it into a Minimum Weighted Bipartite Matching (MWBM) problem to minimize overall communication costs. A location-aware actuator assignment strategy is also proposed to further enhance data locality and reduce the amount of data that needs to be transferred over the network. Reference [21] provides a foundation for preemptive scheduling of prioritized tasks based on the overall deadline specified by the user when submitting the application and utilizes Docker containers for fine-grained dynamic allocation of computing resources. The container later compensates computational resources based on the task's preemption progress and expected progress, thereby improving the overall task completion time. Reference [22] firstly abstracts the task of financial foreign exchange market monitoring indicators into a market-level directed acyclic graph (M-DAG) and prunes it while caching important nodes, making full use of the key nodes to reduce the waste of resources during scheduling. Secondly, Reference proposes a market-level resource dynamic allocation strategy (MYARN) based on dimensions such as transaction mode, product type, and transaction behaviour, which divides the market into multiple sub-markets according to the complexity of the task and allocates computing resources proportionally to the computational complexity among the sub-markets, while the default YARN fair scheduling algorithm is used within the sub-market. Targeted multi-level scheduling reduces costs while improving the resource utilization rate of scheduling. Reference [23] abstracts the scheduling problem of hyperspectral image (HSI) classification tasks into an integer programming model, determines the optimal node mapping, and influences the number of partitions based on the number of subtasks, task dependencies, and communication overheads of the divisible task. It further utilizes the improved quantum heuristic algorithm (QEA) to find the optimal solution and determine the best task-to-partition (PE) mapping. Reference [24], similar to Reference [23], formulates a multi-objective optimal scheduling model by considering the total energy consumption of computational tasks as the optimization objective.

3.1.3 Comprehensive Scheduling Optimization Based on Load

There is a strong correlation between the various levels of Spark, and a more accurate evaluation model can be obtained by comprehensively considering the connections between different levels of load when

analyzing the factors affecting load characteristics. Reference [25] estimated the resource requirements of a task based on the resource usage of tasks at the same stage, designed a semi-predictive task scheduler called COBRA, and developed a task scheduling algorithm by setting task waiting time thresholds and resource utilization thresholds. Reference [26] proposed the xSpark manager, which calculates the minimum number of CPU cores required for a stage based on its execution time, the number of programs expected in the job, and the deadline determined by the amount of data written by the parent node in the DAG. The study also proposed resource scheduling strategies such as earliest deadline priority, proportional allocation of resource requirements, and performance metrics priority, enabling users to independently select the appropriate strategy based on their goals, whether minimizing deadline violations or minimizing resource usage. Reference [27] optimizes cache management from task dependencies by calculating the caching and prefetching priorities of RDDs based on their unprocessed workloads and caching urgency (CU), which depends on the completion time and resource requirements of the previous stage. It further adjusts the task scheduling order by calculating CU as well as the resource utilization of the set of stages containing long-running tasks, aiming to maximize the cache hit rate and mitigate computational resource fragmentation.

The Spark task scheduling algorithm based on data skew and deadline constraints proposed in Reference [28] consists of three parts: the stage ordering component, which sorts tasks based on the degree of data skew, skew rate, and data volume; the task scheduling component, which classifies tasks into skewed, small, and normal tasks, prioritizing the fastest VMs to execute skewed tasks or those with the largest data volume; and finally, the study improves scheduling economy and efficiency by merging fragmented time slots on VMs and filling idle time slots on VMs. Reference [29] integrates the PAC monitoring framework into the Spark core. By analyzing decompression key indicators, input data characteristics, and resource usage, it identifies factors that affect the performance of Spark compression algorithms, including compression and decompression speed, compression ratio, input data type and size, and resource utilization. Optimizing the integration of Zlib into Spark can significantly reduce I/O overhead, thereby indirectly improving scheduling efficiency. Reference [30] proposes a new system, Tripod, based on Tez, which enhances the job scheduling and data cache synchronization capabilities in the framework. The cache-driven scheduling algorithm in Tripod optimizes the execution order by fully utilizing the bandwidth resource idle periods of non-input stages in the DAG for prefetching input data. At the same time, a novel cache strategy, CAP, is introduced, which promptly caches task data blocks that cannot be prefetched and identifies and reduces cache misses on the critical path to shorten the DAG execution time.

3.1.4 Comparison of Representative Algorithms

A comparison of the three representative strategies is shown in Table 2. As a job scheduling strategy, Reference [17] focuses on overall job optimization, which adopts a more macroscopic approach compared to task scheduling and load-integrated scheduling strategies. However, in scenarios with non-uniform job lengths, the scheduling latency increases significantly due to the larger job volume. Reference [23] focuses on fine-grained task optimization, which is more granular compared to job optimization and is better suited to handling complex task dependencies. However, the algorithm has high complexity, leading to additional runtime overhead. Reference [25] combines the advantages of job and task scheduling and achieves multi-dimensional optimization through dynamic resource demand estimation and threshold setting. Compared to pure job scheduling or task scheduling strategies, this approach achieves a better balance between resource utilization and scheduling efficiency. However, the model design is highly dependent on the accuracy of the analysis of the linkages between loads and requires adjustment according to the specific environment.

Table 2: Comparison of load-based scheduling optimization strategies

Load scheduling strategy	Reference	Influencing factors	Performance evaluation metrics	Applicable scenarios
Job scheduling optimization	[17]	Execution order	Improves resource utilization to a certain extent, reduces latency	Batch processing tasks where job lengths are relatively uniform
Task scheduling optimization	[23]	Parallelism, execution order	Significantly improves resource utilization	Scenarios with uneven task lengths and complex structures, where tasks can be further decomposed
Integrated scheduling optimization	[25]	Execution order	Dynamically improves resource utilization and reduces latency	Large-scale parallel computing environments and real-time task scheduling with uneven loads

3.2 Scheduling Strategy Optimization Based on Node Characteristics

In heterogeneous environments, cluster nodes vary in computing power, storage performance, and transmission capability, as well as their changing trends. Scheduling strategies that disregard node characteristics can result in the underutilization of high-performance nodes and the overloading of low-performance nodes. Researchers typically begin by constructing an accurate model to describe node characteristics. For example, node characteristics can be categorized into static and dynamic attributes: resources such as the number of CPU cores, memory size, and network bandwidth are considered static attributes, while CPU utilization, memory usage, and system throughput are classified as dynamic attributes. Alternatively, node characteristics can be divided into computing resources, storage resources, and transmission capability. Once the model is established, concrete and accurate data are required for materialization. Researchers utilize systems like Spark Metrics to monitor real-time node metrics, including CPU utilization, memory usage, task execution time, and GC time. For resource information that is difficult to collect directly, calculations and derivations can be employed. For instance, fluctuating network bandwidth is challenging to measure precisely in real-time but can be estimated indirectly through indicators such as the average system throughput over a period. Finally, based on the characteristics of the collected node resource data, researchers select appropriate evaluation methods for resource allocation optimization. For static resources, decision trees, clustering, and other techniques can be used to assess node priority and other scheduling factors. For dynamic resources, reinforcement learning or deep learning based on time series analysis can be viable choices. The flexible application of these methods enables an effective evaluation of node characteristics to optimize resource allocation in Spark.

3.2.1 References on Scheduling Optimization Based on Node Characteristics

Reference [31] divided the computational power metrics of nodes into static metrics (e.g., CPU cores, memory size) and dynamic metrics (e.g., CPU load and residual rate, task queue length, memory, and disk residual rate). A linear weighting model for node priority was established to ensure a reasonable balance between static and dynamic metrics in the evaluation of node priority. The Spark Dynamic Adaptive Scheduling Algorithm (SDASA) assigns tasks based on real-time node priorities, ensuring that nodes

with higher priorities receive tasks first. The utilization-aware resource allocation method proposed in the Reference [32] can identify the causes of insufficient resource utilization based on monitoring components and inference modules and elastically adjust the number of actuators according to real-time resource usage. When performing iterative computation in a multi-tenant cloud environment, iSpark characterizes virtual nodes based on I/O rate and CPU steal time, extending the two-dimensional resource constraints to three dimensions to evaluate node scheduling priority.

With the development of machine learning technology and improvements in hardware computational power, the prediction of dynamic indicators has become increasingly accurate and adaptable. Reference [33] focuses on performance metrics such as CPU, memory, and bandwidth, and uses quadratic smoothing to predict the average load value of nodes in future cycles. Weighting is achieved by constructing a judgment matrix to measure the relative importance between different factors through hierarchical analysis, while the entropy method is applied to further adjust the weights of the metrics. Based on the predicted node load value and processing capacity, nodes are labeled as high or low load and parallel migration is used to transfer data to low-load nodes. Neural network models can reduce subjective bias and improve the consistency and accuracy of the assessment. Therefore, Reference [34] uses the indicator weights obtained from AHP as the output of the neural network, node dynamic performance indicators as the inputs, and the judgment matrix obtained through expert scoring as the expectation to train and iterate the model. However, the model also has certain limitations, such as the need for a large number of training samples, dependency on initial weight values, and the difficulty in ensuring the objectivity of expectations, as well as the complexity of the model structure. Reference [35] classified nodes into new computing nodes, computing nodes with the least load, and computing nodes with the maximum probability of completing the task, and found that directing tasks more toward new and less-loaded nodes helps improve the performance of the spatio-temporal adaptive reflectivity fusion model.

3.2.2 Comparison of Representative Algorithms

As shown in Table 3, Reference [31] assigns static and dynamic indicators to nodes through a manually designed linear weighting model to improve system resource utilization. This method is suitable for environments with relatively fixed resource demands but cannot meet the requirements of complex dynamic change models. Reference [33] utilizes feature analysis and machine learning methods for load prediction and weight adjustment, making it suitable for cluster environments with significant load changes and more complex models. Reference [34] further combines AHP weights and neural networks to achieve dynamic and intelligent evaluation of node priorities, significantly improving resource utilization and system throughput with higher generalization capabilities. However, it faces challenges such as model complexity, high computational resource requirements, and additional time consumption for sample training.

Table 3: Comparison of node-based scheduling optimization strategies

Reference	Influencing factors	Applicable scenarios	CPU cores	Memory size	CPU utilization	Memory utilization	Bandwidth utilization
[31]	Resource evaluation	Scenarios with relatively stable resource types and demands	√	√	√	√	
[33]	Resource evaluation	Cluster environments with significant dynamic changes in node resources			√	√	√
[34]	Resource evaluation	Complex scenarios with dynamic resource demands and high task parallelism	√	√	√	√	

3.3 Scheduling Strategy Optimization Based on Load and Node Matching

Under circumstances where both the load and nodes exhibit heterogeneity, scheduling based solely on the characteristics of either one makes it difficult to accurately allocate loads to suitable nodes, resulting in resource waste. The performance of the Spark system largely depends on the accurate calculation and comprehensive configuration of parameters [36]. This is a highly dynamic and complex process, where scheduling is influenced by multiple factors such as task type, node resources, and data distribution. In addition to constructing more advanced traditional evaluation models, researchers leverage machine learning to analyze historical scheduling data and real-time system states, uncovering potential correlations and patterns between load prediction and resource allocation. This is not merely about predicting future load fluctuations to pre-allocate resources or forecasting changes in remaining node resources to adjust allocations, but rather about achieving a comprehensive matching of both characteristics. For example, clustering methods can be used to intelligently group nodes, enabling batch scheduling of specific workloads and allocation of designated resources based on the characteristics of sub-node groups. Alternatively, neural networks can be employed to learn from load data and resource adjustment records, revealing the causal relationship between load variations and resource allocation, thereby providing more precise guidance for scheduling strategies. Therefore, to more effectively address the challenges posed by load and node heterogeneity, it is essential to comprehensively consider the matching patterns between loads and resources and design an integrated scheduling mechanism.

3.3.1 Traditional Model-Driven Load-Node Matching

Reference [37] classified jobs into CPU-intensive and memory-intensive types. It applied the Analytic Hierarchy Process (AHP) to construct judgment matrices for the static and dynamic variables of nodes, building an initial weight model. During the experimental process, the weights were adjusted separately based on the performance of different types of jobs. Finally, the real-time performance-oriented priority of each node in the cluster was calculated, and jobs were assigned according to the priority. Reference [38] proposed a cross-layer optimization system named Clio, which defined an interference factor to quantify the impact of resource competition among multiple tasks on the same node on performance. At the same time, it adopted a boundary key partitioning algorithm, dynamically adjusting the allocation of boundary key partitions based on the predicted execution time of tasks to minimize execution time differences between tasks and mitigate the impact of lagging tasks. Reference [39] established models for node computational power, node resource utilization, data bucket skewness, and task memory requirements. It further designed three-stage optimization objectives: the parallelism estimation algorithm predicts the parallelism for the next stage by analyzing the data volume from the previous computation stage and the availability of node resources; the data skew correction algorithm identifies tasks with memory requirements exceeding node memory, splits and redistributes such tasks to construct data buckets, thereby improving execution efficiency; the heterogeneous node task allocation algorithm assigns tasks based on the computational power of nodes and the size of data buckets, improving parallelism and execution efficiency. Reference [40] divided Spark nodes into large and small nodes, defining resource availability for nodes and resource demand for jobs. The scheduler also considered the job deadline as a factor influencing priority ranking.

Reference [41] used the computational load of each task and the computational capacity of GPU devices as scheduling factors, estimating the runtime of different tasks on different devices. Based on this, a time-cost matrix was constructed to calculate the upper and lower bounds of execution time. Reference [42] calculates dynamic thresholds based on the current cluster load and performance evaluation, adjusting the number of replicas to balance performance and fault tolerance. Tasks are categorized into two types: CPU-sensitive and disk-sensitive. The corresponding tasks are executed on suitable nodes based on the

resource utilization thresholds of the nodes and the task types. Reference [43] proposes the SimCost simulation prediction model to optimize Spark scheduling and resource allocation. It uses the Monte Carlo method and a cost model to predict job execution time. The cost model includes multiple factors such as parameter modules, resource usage time, shuffle events, etc., to determine the configuration of factors that result in optimal cost-effectiveness, thereby improving scheduling efficiency. Reference [44] addresses the issue of heterogeneous communication costs at various scheduling stages and models the communication cost of tasks transmitting data to executors. This leads to an optimal executor allocation problem aimed at minimizing total communication costs, where the search for the optimal solution involves finding the best executor start nodes. A greedy descent heuristic algorithm is proposed to quickly find a set of high-quality solutions.

3.3.2 Machine Learning-Driven Load-Node Matching

To more precisely allocate loads to corresponding computing nodes, some studies have employed machine learning techniques for fine-grained classification and feature analysis of loads. Reference [45] used K-means clustering to group jobs based on factors such as data characteristics, job execution time, and communication requirements, then submitted the clustered jobs to nodes with similar characteristics, reducing communication time between clusters. Meanwhile, it improved performance through methods like task grouping, thread pool management, and execution parameter optimization, compensating for the communication overhead. Reference [46] developed a task-scheduling system called Stargazer, which consists of a time inference component and a task-scheduling component. The time inference component uses an LSTM model to estimate task completion time and makes timely scheduling adjustments for tasks with predicted excessive completion times. The task scheduling component determines whether tasks are local or non-local based on the scheduling model and dispatches non-local tasks to other nodes for processing. Reference [47] proposed a Dynamic Memory-Aware Task Scheduler (DMATS) for Spark, which considers memory and network I/O as dynamic resource evaluation variables for nodes. Task characteristics include task locality, input data size, memory resources required for execution, and blocking factor. DMATS first estimates the initial adaptive task concurrency using variables other than memory, then applies the AIMD algorithm to adjust concurrency based on real-time feedback on memory usage ratios.

Reference [48], leveraging reinforcement learning and a collaborative locality goodness measurement algorithm, improves task scheduling efficiency by scheduling repeatedly appearing data. It evaluates the collaborative locality goodness of jobs based on computational resources such as CPU utilization and I/O wait time and uses a gradient gambling algorithm to adjust the probability distribution of job selection. Reference [49] determines the levels of sub-tasks in the workflow through forward and backward breadth-first search, then uses GBDT to predict the execution time of sub-tasks. By combining the parallel application of Directed Acyclic Graph (DAG) and critical path algorithms, the critical path of the workflow is identified, and the sub-tasks on this path are given the highest priority. Different amounts of resources are allocated to each level based on this priority. Reference [50] optimizes scheduling strategies to reduce costs. It combines Sparrow Search Algorithm (SSA) and Extreme Gradient Boosting (XGBoost) to predict resource demands for real-time task flows in heterogeneous clusters. The designed CEBFD placement method starts executors on reasonable nodes to reduce costs. To ensure efficiency, a Service Level Agreement (SLA) based on job deadlines is also proposed, which comprehensively improves service quality. Reference [51] proposed the Hawkeye system, which uses a reinforcement learning model to analyze the current running speed of tasks and their expected execution time, quickly identifying lagging tasks. The system dynamically adjusts the lagging task identification threshold and speculative execution trigger threshold using reinforcement learning to improve recognition accuracy. Subsequently, based on historical data, it intelligently evaluates

and selects nodes with better performance and fewer lagging tasks to execute speculative tasks (assign task replicas).

3.3.3 Comparison of Representative Algorithms

As shown in Table 4, all three references aim to improve resource utilization, reduce scheduling latency, and increase system throughput, thereby comprehensively enhancing program execution time. The differences lie in their approaches. Reference [39] relies on manually designed models, which have limited predictive capabilities when the scenario changes. While precise model construction can accurately predict the overall computational power of nodes, the prediction for highly dynamic computing resources such as memory is relatively coarse, making it suitable for scenarios where the factors affecting cluster computing resources are relatively less complex.

Table 4: Comparison of scheduling optimization strategies based on load-node matching

Reference	Influencing factors	Differences	Applicable scenarios
[39]	Execution order, data locality, node resource evaluation	Limited predictive capability, highly targeted	Heterogeneous computing environments with fixed characteristics and tasks with data skew
[46]	Data locality, node resource evaluation, execution order	Additional training resource consumption, strong long-term prediction capability	Environments, where task execution time is predictable and intelligent scheduling is required
[51]	Fault-tolerance mechanism, node resource evaluation	High algorithm complexity, significant system load, high fault tolerance	Complex cluster environments requiring high resource utilization and low scheduling latency

Reference [46] requires a large amount of training data and computational resources, without the need for detailed analysis of the underlying principles of resource changes. It focuses on predicting task execution time and determining task locality from a macro perspective, with strong long-term prediction capabilities. However, it struggles to handle rapid fluctuations in cluster resources and incurs additional time consumption for training. Reference [51] learns optimal strategies through interaction with the environment, enabling dynamic adjustments to task replica scheduling decisions. Despite the high algorithmic complexity, significant system load, and the need for extensive interaction data and computational resources during the training process, it effectively balances fault tolerance and additional load during program execution and demonstrates excellent performance in dynamic environments.

3.4 Comparison of Three Scheduling Strategy Models

The characteristics of three different scheduling modes can be analyzed from dimensions such as evaluation complexity and flexibility. Evaluation complexity includes data collection overhead, data accuracy, as well as constraint conditions, and feasible solution space, while flexibility can be divided into scenario adaptability and applicable scenarios. The comparison results are shown in Table 5.

Table 5: Comparison of the three scheduling modes

Scheduling mode	Collection overhead	Accuracy	Constraints and feasible solution space	Adaptability	Applicable scenarios
Based on load characteristics	Low	Accurate	Load constraints: parallelism, priority, limited solution space	Low	High task heterogeneity, homogeneous or similar node resources
Based on node characteristics	Relatively high	Inaccurate	Resource constraints: multiple conditions such as CPU/memory utilization, larger solution space	Relatively high	Similar task characteristics, high heterogeneity in node resources
Combined load-node matching	High	Inaccurate	Must satisfy both task and node constraints, significantly increased solution space	High	High task heterogeneity, high node resource heterogeneity, suitable for most scenarios

In the load-based scheduling optimization mode, tasks and collected data are derived from precisely abstracted applications and datasets provided by users and developers, resulting in low collection difficulty. The system can directly extract and use the data from storage systems and memory. When evaluating load characteristics, the constraints are primarily related to task parallelism and priority, with the solution space generally being relatively limited. Due to the neglect of node heterogeneity, this model is not suitable for large-scale heterogeneous clusters. Compared to algorithms that comprehensively consider load and node heterogeneity, resource utilization during execution tends to be more unstable. This paper conducts a quantitative analysis of resource utilization for three reference algorithms tested on the HiBench standard dataset. The experimental results of resource utilization during the mid-peak period in each reference are collected, and the variance is calculated to reflect the fluctuation differences in resource utilization. The middle 50% of the indicators are selected to avoid interference from fluctuations at the start and end, and the variance more accurately reflects the range of data changes.

As shown in Table 6, PAS [17] is based on load characteristics and does not consider node heterogeneity. The fluctuation range of CPU resource utilization is much larger compared to the algorithms of DMATS [47] and the reference [42]. PAS lacks adaptability to nodes with different computational capabilities. At the same time, in the comparison of execution time metrics in the reference, PAS's execution time is close to that of DRF, which has a strong adaptability to heterogeneous clusters, further confirming that PAS has a low adaptability to heterogeneous clusters. In terms of memory utilization, the stability of the scheduling algorithms in the latter two is only slightly higher than PAS, possibly because memory is typically a large-capacity resource, and fluctuations are less likely to cause significant differences. When memory usage does not reach its limit, the caching mechanism may make memory utilization very stable, so the optimization effect is not significant. Due to interference from uncontrollable factors such as device resource differences, environmental differences, and data volume, it is difficult to directly compare execution times. All three algorithm experiments were compared with Spark's default scheduling strategy, FIFO. Therefore, this paper uses Spark's default scheduling strategy as a baseline and compares the execution time ratio of the optimized algorithm with the execution time of Spark's default strategy as a reference for the algorithm's optimization effect.

Table 6: Quantitative analysis of resource utilization rates of representative algorithms

Algorithm	Scheduling mode	CPU utilization variance	Memory utilization variance	Execution time ratio
PAS [17]	Based on load characteristics	0.0255	0.0017	0.904
DMATS [47]	Combined load-node matching	0.0191	0.0012	0.79
Algorithm in reference [42]	Combined load-node matching	0.0020	0.0015	0.77

The scheduling optimization mode that considers node characteristics requires the collection of various data, such as CPU, memory, and network bandwidth, to evaluate node computational power and suitability. This requires monitors to collect dynamic data, which has limited accuracy and is difficult to predict. The numerous constraints further increase the solution space. When combining both load and node characteristics, in addition to the basic constraints, the interactions between the constraints of both must also be considered, significantly expanding the solution space. The evaluation overhead may become unacceptable for both the system and the users. Therefore, although the scheduling mode that considers node characteristics can adapt to complex cluster environments, it is still necessary to balance the evaluation complexity of scheduling criteria with the adaptability of the model when optimizing the scheduling strategy.

3.5 Integration of Spark Scheduling Strategies with Specific Scenarios

3.5.1 Existing Scheduling Optimization Based on Specific Scenarios

Current Spark scheduling research focuses on the construction and optimization of general models, as shown in Fig. 4.

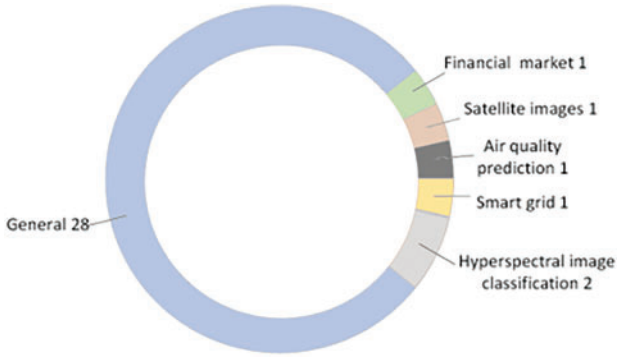

Figure 4: Number of application scenarios in spark scheduling strategies

While general models have broader applicability, their performance in practical scenarios often faces limitations, making it challenging to meet the specific requirements of diverse and complex application scenarios.

In Section 2, some studies propose more adaptive and efficient scheduling methods by deeply analyzing performance bottlenecks and task characteristics in specific scenarios. Reference [19] analyzed the characteristics of air quality data collected hourly by monitoring stations with uneven local distribution, partitioning the data based on temporal and spatial attributes to address the imbalance and spatial heterogeneity of large-scale air quality data, ensuring balanced computations across the system. Reference [22] focused on three monitoring indicator calculation tasks in the financial foreign exchange market, constructing a DAG for the indicator calculation tasks and caching important results that are widely relied upon by subsequent tasks and computationally expensive to avoid redundant calculations. Additionally, based on different business scenarios and computational complexity, jobs were divided into different "markets", with distinct resource management frameworks applied both between and within markets to adapt to varying resource fluctuation characteristics. By reducing redundant task scheduling and allocating resources appropriately based on task characteristics, not only is the scheduling resource utilization significantly improved, but costs are also effectively controlled. References [23,24] divided pixels in hyperspectral image classification tasks into sub-pixel, pixel, and super-pixel levels, extracting and processing multiple features. For example, during super-pixel feature extraction, the original HSI dataset was decomposed into multiple spatial domain partitions, while at the sub-pixel level, N-FINDR was used to extract endmembers. Tasks were finely categorized into serial and parallel, or computation-intensive and data-intensive tasks, which not only improved Spark's parallel acceleration efficiency but also ensured classification accuracy. Reference [35] designed experiments based on the characteristics of resource-constrained remote sensing big data to simulate the computing environment, making the improved Spark framework more targeted. Reference [45] addressed the characteristics of smart grid big data by clustering data based on transformer IDs, enabling parallel training and testing within each transformer group, which reduced the number of iterations required for model training and testing.

3.5.2 Exploration of Scheduling Optimization Based on Specific Scenarios

These studies provide an initial demonstration of how to optimize Spark's scheduling strategy by incorporating the characteristics of specific application scenarios. This section will continue to explore in detail the optimization details of combining scheduling strategies with specific scenario characteristics and the process of practical application, using scenarios such as user review clustering, power quality monitoring, healthcare, and finance as examples.

User review mining [52] scenarios have characteristics such as large data volume, diverse text content, and imbalanced data representation. Simply parallelizing data blocks and scheduling them using general methods may achieve balanced data distribution but could overlook the differences in the importance of the data. For example, the key small samples of low-frequency features, such as unique feedback on new products or potential market trend signals, are critical for business decisions. Uniformly partitioning data blocks may lead to insufficient resource allocation for key samples like highly-rated reviews or authoritative users' suggestions for product improvements. To ensure that key small sample data from low-frequency features are not overlooked, researchers can first perform pre-clustering on the feature dataset using Canopy, creating a new ordered distance dataset. After dividing it into multiple subsets based on the features and their weights, the data is stored in HDFS. Then, Spark creates tasks with identifiers for computational complexity, data volume, and business-related features, which are targeted and assigned to nodes with different characteristics. For instance, tasks containing key small sample data and having higher computational complexity can be allocated more CPU resources to ensure each task receives the appropriate resource support.

Power quality monitoring [53] scenario data exhibits more distinct characteristics due to differences in regional electricity usage structures and industrial activities. Traditional fixed scheduling strategies are prone

to overlooking regional time differences and task diversity. Key regional tasks may experience delays due to insufficient resources during execution, leading to power quality issues that cannot be detected and addressed promptly. By collecting and analyzing historical data, and establishing a time-series analysis model, the scheduler can predict the timing and trend of regional events. For example, in industrial areas where large equipment is concentrated and operates 24 h a day, data activity is high during weekdays, causing voltage fluctuations. Continuing to rely on locality-first scheduling might lead to insufficient resources, resulting in additional waiting time for tasks. However, during this period, most residents in residential areas are away, and the power monitoring system's computational resources are idle. The weight of cross-rack scheduling can be appropriately increased, and tasks with lower real-time requirements can be transferred to the residential area's power system for processing. When the electricity usage peak in the residential area arrives at night, the number of long-distance transmissions can be reduced. Additionally, this period will vary with the seasons, and the plan can be adjusted in advance using a time-series analysis model.

Map data mining [54] contains richer spatiotemporal factors. After dividing the area according to the grid map, not only can the temporal patterns of data flows in different regions be analyzed to establish time series analysis models, but also a spatiotemporal data flow model with multidimensional features can be constructed by combining the number and importance of regional facilities. For example, factors such as population density, traffic accessibility, and key traffic routes with specific topologies can be considered to capture their characteristics at different times. During school dismissal times, data can be stored locally or in areas with fewer nearby data to mitigate data skew during computational peaks, or tasks can be scheduled to nearby region racks during this period. Heuristic algorithms can then be used to calculate the optimal multi-level locality weights.

The Internet of Things big data scenario [55] involves strict event dependencies and sequencing. In the prediction of atmospheric pressure and temperature data, the data involves multiple task steps such as collection, transmission, modeling, and prediction. Critical path identification methods and task priorities can be designed to ensure the execution order, avoiding resources being excessively occupied by the training module, which could cause LoRa to stagnate when processing sensor data. At the same time, different sensor data sizes must be considered, and a multi-level caching structure is used to assist in designing the task processing priority order.

In the healthcare scenario [56], epidemic outbreak prediction relies on multi-source real-time data, including social media, public health records, and environmental factors. Spark scheduling strategies should prioritize processing real-time data flowing in from social media or decompose it into small batches for quick processing, to avoid delays in real-time data processing caused by resource waiting. For data with lower update frequencies, such as public health records and environmental factors, fixed resources can be allocated for stable processing. Furthermore, genomic computation tasks have significant differences in complexity. Whole genome association studies involve 3 billion base pairs and their association with diseases or traits, requiring large amounts of data and involving multiple complex situations such as population stratification, co-expression, and environmental synergy. In contrast, single-gene expression quantification analyzes only the expression level of a single gene, with a simple data source and analysis method. Spark can allocate tasks with different data volumes to compute nodes with different resources in batches, preventing data skew.

In the financial field [57], the information system relationship models are diverse, and there are significant differences in metadata. FIBO is a standardized global ontology that provides universal metadata definitions for the financial sector, while ISLO is a metadata structure for information systems of specific companies. Similar to the aforementioned fields, tasks can be tagged according to the metadata of FIBO and ISLO. For example, financial indicator calculations based on the FIBO standard can be classified as general tasks important for overall decision-making and assigned the highest priority. For ISLO tasks, priority can

be sorted based on data volume and business logic. Moreover, the differences between the two can also play an important role in fault tolerance. When handling cross-system and cross-company computational tasks, the FIBO standard can quickly identify the same type of data across different data sources, clearly map out the sources and transformation paths of different data, and identify important results that need to be reused. This can help avoid redundant computation and data confusion in RDD lineage tracking, thereby improving computational efficiency.

From the above content, it can be seen that in different application scenarios, many aspects such as initial data distribution, node resource changes, and so on, have distinct characteristics. By deeply analyzing specific application scenarios, potential optimization directions can be further identified, thus improving the Spark scheduling strategy in a more specific and fine-grained manner.

3.6 Challenges in Implementing Scheduling Strategies in Production

Despite the significant theoretical advancements and notable performance improvements achieved by existing scheduling optimization strategies, their implementation in real-world production environments still faces numerous challenges.

3.6.1 Initial Data Skew

Data sources in production environments are highly diverse, ranging from scattered personal uploads to large-scale datasets collected in bulk by organizations. Due to the lack of unified standards and rigorous preprocessing, these datasets often exhibit high randomness and unpredictable skewness. If preprocessing is performed in advance, the overhead of computing and storing statistical distributions of large-scale data can be prohibitively expensive. On the other hand, applying post-processing adjustments poses major challenges to correction models, as they must dynamically adapt to the characteristics of data skewness, and the same model may perform inconsistently across different datasets, or even fail. Furthermore, as the program executes, the data distribution may dynamically change, adding to the complexity of handling skewness.

The existing solutions generally first identify skewed tasks and then handle them specifically. The handling method can be as shown in reference [28], where after identification, no re-partitioning is performed. Instead, skewed tasks are executed first, with a large amount of resources allocated to them during resource-abundant phases to avoid normal tasks from competing for resources, while also avoiding the high overhead caused by large-scale data re-distribution. However, in scenarios with extremely severe skew, data re-distribution, as shown in reference [39], is required. By decomposing and reconstructing severely skewed large tasks, data can be evenly distributed across nodes, greatly improving resource utilization and optimizing locality.

3.6.2 Severe Cluster Heterogeneity

Except for dedicated high-performance computing centers or large-scale enterprise server clusters, most multi-user production environments suffer from severe cluster heterogeneity. The lack of uniform hardware models, the mixed usage of old and new devices, and the dynamic fluctuations in node load make resource utilization efficiency difficult to predict and control. Additionally, the uncertainty in user behavior and regional variations in workloads make load prediction extremely challenging, rendering real-time scheduling adaptation difficult. If a per-node resource evaluation and prediction model is used, it can incur significant computational overhead, especially when deployed on low-performance nodes, potentially offsetting the benefits of optimization.

In extreme heterogeneous scenarios where each node in the cluster has different characteristics, using a sub-cluster division strategy based on distinct features is insufficient to assess the computational power of each node. The neural network model shown in reference [34] obtains the computational power of individual nodes through dynamic performance indicators of the nodes, and although it can also handle extreme heterogeneity, it has problems such as complex models, high computational resource consumption, and difficulty in ensuring the objectivity of labels. The lightweight time-series analysis-based model, represented by reference [33], can predict the future computational power trends of each node based on a small amount of historical data, but it is challenging to capture complex causal relationships.

3.6.3 Lack of Historical Data

Machine learning models rely on large-scale historical data samples for training, but in new domains and with new users, the lack of historical data is a common issue. Prediction models without sufficient historical data support struggle to accurately capture dynamic data characteristics, leading to unstable optimization results. Moreover, designing models from scratch based on scenario characteristics is not only time-consuming but also costly, making it less cost-effective in the short term.

3.6.4 Security and Privacy

In multi-user environments, privacy protection requirements may prevent certain users from sharing critical security- and business-sensitive data for centralized processing, thereby increasing the complexity of global optimization. Techniques such as secure multi-party computation, including fully homomorphic encryption (FHE) [58], significantly increase computational overhead far beyond that of raw data processing. Additionally, most existing FHE libraries are implemented in C++, while the Spark framework is based on Scala, necessitating additional serialization and conversion processes at the data structure interaction layer, further increasing processing latency and difficulty.

Beyond the challenges outlined above, the deployment of Spark scheduling strategies in production environments often encounters unforeseen issues. These challenges demand that scheduling optimization methods possess greater adaptability, scalability, and efficiency to provide viable solutions for complex environments.

4 Future Research Directions

Existing Spark task scheduling algorithms have made significant progress in areas such as resource utilization, alleviating scheduling delays, and fault recovery. However, with the rapid increase in cluster scale and complexity in emerging scenarios, there is still room for optimization and innovation in Spark scheduling algorithms.

4.1 Applications in Emerging Fields

Although existing general Spark scheduling models consider heterogeneous characteristics such as cluster computing and network resources to some extent, the significant variations in data characteristics, task priorities, and local cluster resource demands across different scenarios make it difficult for a universal model to meet the scheduling optimization needs of all use cases.

In future research, scheduling strategies can be optimized from the perspective of "domain awareness", adapting to different industries or scenarios by considering data distribution, workload characteristics, and node distribution patterns. For example, in the smart grid scenario [59], the cluster structure is complex, and as new nodes are added, their operating states and characteristics change dynamically. Neural networks

or knowledge graphs can be leveraged to assess the regional operational characteristics of power systems in real-time, enabling Spark to dynamically schedule workloads to nodes with more available resources or lower failure rates, thereby improving efficiency. In the Supply Chain Finance scenario [60], upstream and downstream enterprises exhibit significant differences in support policies, resource boundaries, and geographic locations, leading to varying levels of digitalization. Clustering methods can be used to analyze the correlation between enterprise digitalization levels and their geographic distribution, such as regional bandwidth patterns. Based on this analysis, Spark can formulate a more flexible data locality strategy, balancing transmission distance and computational resource allocation.

Therefore, when applying Spark to emerging scenarios, it is essential to incorporate research findings and practical insights from the specific domain to determine the key optimization directions for Spark's scheduling strategy.

4.2 Execution Time Prediction for Distributed System Loads

Load execution time prediction is crucial for optimizing Spark scheduling strategies, and many studies have adopted deadline-based scheduling priority models. However, existing solutions over-rely on historical data, which may be unavailable in real-world production environments, making it difficult to accurately capture changing patterns and make reliable predictions. At the same time, some machine learning models can provide accurate predictions, but they introduce additional training time, significantly increasing scheduling delays. In scenarios with high demands for efficiency and low cost, low-cost prediction models are needed.

Time-series analysis-based models are a type of statistical or machine-learning model that does not rely on large amounts of training data. They can summarize patterns by understanding the changes in data over time, enabling the prediction of future load execution times. Common models include ARIMA and the exponential smoothing method mentioned in Section 3.2, which model the data using previous observations and estimate parameters using simple mathematical formulas to predict the load execution time within a certain period. Compared to machine learning models that need to capture causal relationships, time-series analysis-based models require fewer computational steps.

Traditional static execution time prediction formulas (such as Amdahl's Law, Gustafson's Law, etc.) are a more resource-efficient prediction method. They perform polynomial extrapolation based on resource configurations such as the number of nodes, node computational power, and parallelism, fitting a specific nonlinear formula to represent the relationship between load execution time and resource amount, with the resource amount as the input parameter to predict execution time. However, these formulas are becoming less suitable for the current complex environments, and some researchers have begun exploring low-cost distributed execution time prediction formulas that are better suited to current big data environments. Reference [61] employs a two-dimensional plane heat transfer model, using the boundary size growth rate to measure the communication cost between nodes and constructing a relationship between the number of nodes and program execution time in the Spark framework. During fitting tests on various programs, the coefficient of determination (R^2) for almost all programs exceeded 0.9. When determining whether the execution time of subsequent loads will exceed the deadline, one can first determine whether the load is a simple count or a different program such as machine learning, and then use the corresponding polynomial calculation to predict the load execution time. Additionally, in Reference [62], the authors demonstrated through multi-feature model experiments that execution time prediction models designed based on program characteristics such as computational complexity have higher accuracy when using extrapolation methods compared to machine learning models. This indicates that, in scheduling models requiring task execution time prediction, the relationship between other specific characteristics of the program, such as the number

of partitions, heap memory allocation, etc., and load execution time can be explored. By deriving and fitting different prediction formulas, prediction accuracy can be improved, inference time can be shortened, and prediction costs can be minimized as much as possible.

4.3 Relationships between Loads

Current research mostly focuses on static, single-level load scheduling optimization. Even the integrated studies listed in Section 3.1.3 only analyze the relationships between stages and tasks or between stages and jobs. While these studies propose effective optimization solutions at their respective levels, they do not fully explore the interconnections among jobs, stages, and tasks. As a result, scheduling strategies may achieve only local optimization, without sufficiently investigating whether Spark scheduling strategies have the potential to balance and coordinate all three levels for global optimization.

Tez [63] is a DAG-based computational framework in the Apache ecosystem that optimizes resource utilization and execution order through classical graph processing methods such as topological sorting and path analysis. It also incorporates reinforcement learning and online learning to optimize allocation and scheduling from an overall performance perspective. However, Tez's optimization analysis mainly focuses on multiple stages and tasks within a single job and does not explicitly address multi-job, multi-stage, multi-task optimization. Systematically constructing a multi-user, multi-level association model can help enhance the adaptability and robustness of scheduling algorithms in dynamic scenarios of multi-job coordination. Nevertheless, from the principles and advantages of Tez, it efficiently utilizes resources and flexibly handles task dependencies. After optimization, it can be integrated with Spark to be applied in multi-job environments. For example, meteorological monitoring, analysis, and early warning correspond to multiple jobs, and each job contains multiple stages such as data collection, cleaning, modeling, and analysis. Based on the original model of single-job stage tasks, additional dependency relationships and priority evaluation modules for meteorological jobs can be added to avoid consuming too many resources during single-job optimization, which could sacrifice the overall meteorological business efficiency.

4.4 Insights from Other Distributed Domains

Compared with other distributed parallel computing fields, Spark's scheduling strategies have the following limitations: there is a lack of fine-grained modeling for different levels of network transmission distances, making it difficult to precisely define transmission overhead, and it is insufficiently adaptable to mobile devices; the connection between stage division and program functional modules is not deeply explored, making it difficult to adaptively decompose and lightweight based on the program structure in multi-level mobile devices when the job size and stage gap are significant; in the DAG of complex programs, Spark lacks a priority scheduling mechanism for tasks with complex and tight dependencies.

Spark scheduling optimization can draw inspiration from other distributed parallel computing domains. For example, based on research in edge computing [64], Spark can refine locality parameters to differentiate network overhead at varying levels, such as different transmission distances or rack hierarchies. It can also be deployed on highly mobile IoT devices [65], by integrating task offloading and hierarchical decomposition techniques, Spark can deeply explore the relationships between stage partitioning and program functional modules, enabling lightweight computation. For priority scheduling in complex dependencies, Spark can draw inspiration from the critical path method in high-performance computing [66] and the scheduling algorithms in multi-core embedded systems [67], providing a more fine-grained preemptive execution order optimization approach based on data dependencies and computational complexity.

4.5 Integration of Spark with Blockchain

Spark faces the risk of Driver central node failure, which can cause the entire job to be immediately interrupted, requiring the standby Driver to resubmit the job. At the same time, cross-rack transmissions are often unavoidable, leading to a significant increase in system transmission overhead. Therefore, Spark can be optimized by integrating with blockchain [68]. In the event of a Driver failure, a new Driver node can be selected from the existing cluster through a consensus mechanism, similar to the view change mechanism in consortium blockchains for re-electing a Leader [69].

To reduce transmission overhead, smart contracts can be used to dynamically allocate computational tasks to appropriate nodes, transmitting the required data indices to the nodes. The original metadata transmission work is performed by the blockchain system during the initial data synchronization, which not only improves efficiency but also ensures data security. Reference [70] stores the hash values of structured and unstructured data in agricultural data are stored in the blockchain, and unstructured data is stored in HDFS, avoiding the efficiency issues associated with centralized storage. When using Spark to reorganize and integrate structured data, data can be directly extracted from the blockchain based on the index, reducing unnecessary transmission costs.

4.6 SDN Empowered Spark Scheduling

SDN [71] technology, with its unique architectural concepts and strong flexibility, has become a cutting-edge topic and is very popular in cloud computing task scheduling. Fig. 5 shows the co-occurrence relationship between the keywords SDN, Cloud Computing, and Scheduling in 797 SDN-related journal papers from the past three years in the Web of Science, indicating that the application of SDN in cloud computing task scheduling is one of the current hot research directions. However, as a research area highly similar to cloud-computing scheduling, Spark scheduling has a certain gap in the research on the integration of the two.

4.6.1 Core Concepts of SDN

SDN [72] is a new network paradigm designed to overcome the limitations of traditional networks. It separates the control attributes of each independent decision-making decentralized network device (such as routers) from the data transmission attributes, leaving only the data transmission attributes, while the control device implements centralized network management, giving the network architecture a global network perspective. The SDN architecture is divided into three layers: the data plane, composed of transmission devices lacking intrinsic control or decision-making software; the control plane, responsible for monitoring the data plane information and transmitting application-level decisions; and the application plane, which is responsible for formulating and adjusting task scheduling and resource allocation based on the information provided by the control plane.

The SDN infrastructure has a certain degree of universality and achieves communication and collaboration between planes through standardized interfaces. Therefore, under different optimization requirements, it can integrate various software to make more accurate scheduling decisions. For example, to find the optimal task scheduling solution, Reference [73] defines formulas such as the expected computation time of tasks at each node in the scheduling model of the fog IoT system. The AWOA algorithm, which combines AO and WOA, seeks the optimal solution and integrates AWOA into the SDN framework to optimize the service quality of IoTFS. As can be seen, SDN can integrate with advanced technologies such as heuristic algorithms to improve overall network performance.

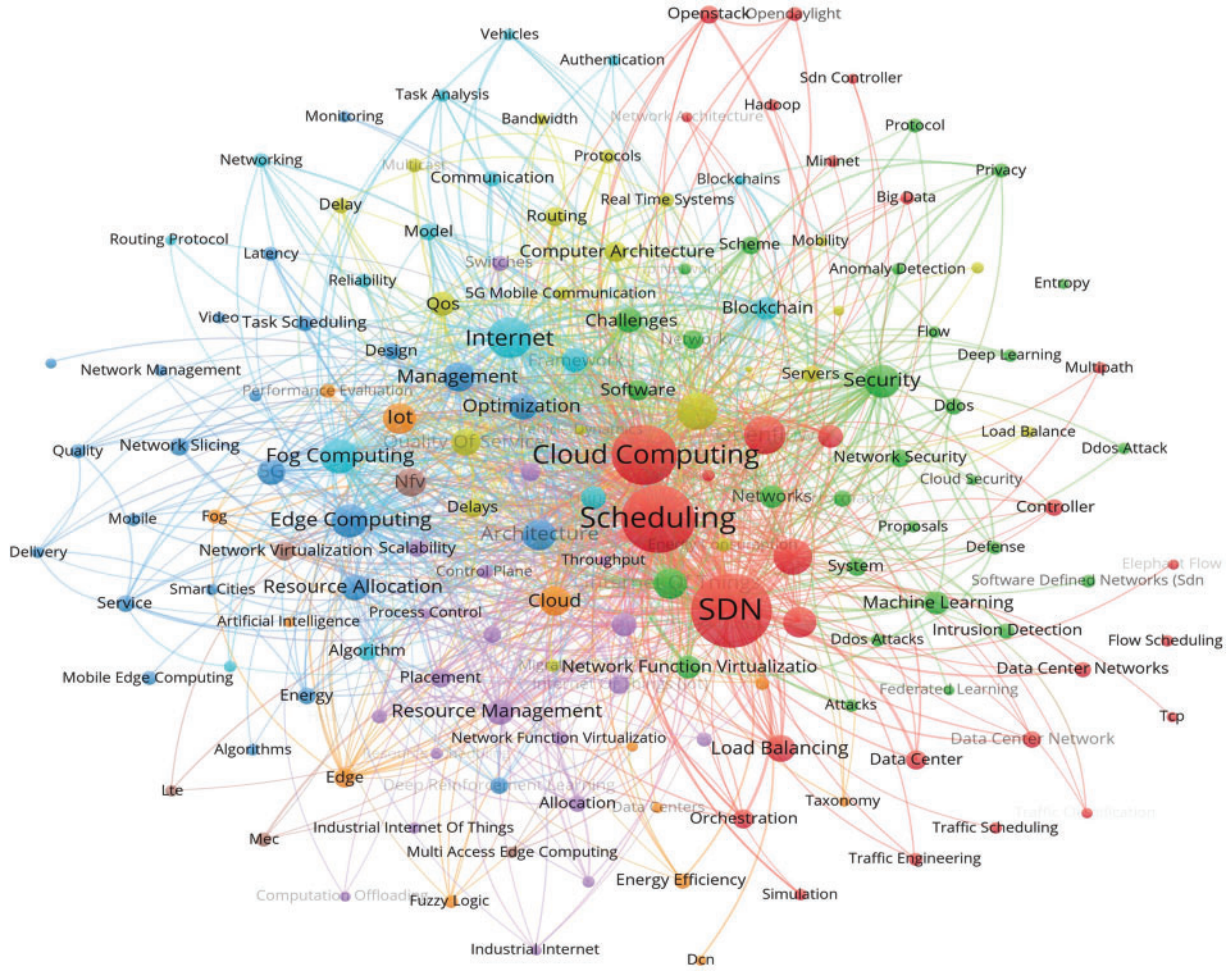

Figure 5: Co-occurrence network of research hotspots on SDN and cloud computing task scheduling

4.6.2 How SDN Empowers Spark Scheduling

Spark task scheduling involves data transmission along different paths. Existing locality mechanisms lack dynamic optimization of overall network topology information. Unreasonable cross-rack scheduling and local scheduling can increase network latency and bandwidth consumption due to external complex factors, thereby reducing scheduling efficiency.

The integration of SDN with Spark can directly optimize the locality mechanism. By constructing models based on network bandwidth, latency, and other metrics, along with dynamic network topology information, SDN, in combination with heuristic algorithms, graph algorithms, or reinforcement learning methods, can be integrated into Spark's resource manager and scheduler. This enables Spark to more accurately determine the locality levels between nodes, with cross-rack scheduling receiving more granular evaluations. For example, when a rack experiences severe attacks and its network bandwidth becomes highly unstable, the SDN control core, integrated with heuristic algorithms, can promptly assess whether the abstract topology of multi-rack clusters needs to be changed and choose adjacent racks instead.

Moreover, it is not just the locality mechanism—SDN's characteristics can also indirectly affect other factors influencing Spark's scheduling efficiency. For instance, when a node is mitigating data skew but overcorrecting, resulting in excessive network traffic, SDN can limit data re-partitioning to balance resource

utilization and transmission overhead. In adaptive dynamic parallelism scheduling algorithms, SDN can evaluate network transmission quality to decide on increasing or decreasing parallelism. When quality is poor, parallelism can be appropriately increased to prevent task scheduling failures and the need for retransmitting large amounts of data. In terms of resource evaluation, SDN can combine time-series models to more flexibly and accurately assess the trend of node network bandwidth resource changes.

4.7 Insights from Classic Cluster Management Systems

4.7.1 Insights from Borg for Spark Scheduling

The resource allocation method in Spark based on computational or lightweight model predictions can adapt to most scenarios. However, in scenarios where the resource demands of tasks exhibit highly complex variations and latency, strictly allocating resources based on predictions may lead to performance bottlenecks. Borg's [74] oversubscription resource mechanism has significant advantages in dealing with complex and dramatic changes in scenarios. It allows a certain degree of over-allocation of resources based on resource demand prediction and analysis, thereby improving resource utilization. Spark can combine advanced technologies like reinforcement learning to build multi-factor prediction models in complex environments, allowing hierarchical scheduling on nodes with more abundant resources. For example, in epidemic prediction scenarios, if reinforcement learning assesses factors such as repeated IP network comments, increased local hospital visits, and a surge in traffic during holidays, and determines that an epidemic might break out in a certain area, local evaluation tasks can allow the over-allocation of nodes for scheduling.

4.7.2 Insights from Kubernetes for Spark Scheduling

Currently, the vast majority of Spark scheduling mechanisms primarily allocate resources based on task or node priorities. If priority calculations and sorting are performed for each node, it will generate significant computational overhead, and this method lacks flexibility at the node level, making it difficult to handle special scheduling requirements. The container scheduling system Kubernetes [75], derived from Borg, can implement affinity and anti-affinity scheduling strategies or taints and tolerations strategies using Pods as scheduling units, isolating specific types of tasks from certain nodes, and achieving goal-oriented scheduling in a multi-user environment. Spark can combine reinforcement learning to batch plan task scheduling from the perspectives of task adaptation, privacy protection, local cluster stability, and other factors, optimizing resource allocation. For example, users can restrict tasks involving sensitive information to a specific sub-cluster for regular scheduling, or, if reinforcement learning assesses a node's stability as low and the pattern is difficult to discern, this node can be shielded and wait for it to return to normal operation.

5 Conclusion

With the development of information technology, the complexity and scale of big data environments have reached unprecedented levels. As one of the popular distributed parallel computing platforms, optimizing the performance of Spark has become a key research focus at present. Among these, scheduling strategy optimization plays a significant role by optimizing resource allocation from the perspective of clusters, analyzing performance bottlenecks at a macro level, and closely integrating with the characteristics of real-world application scenarios, making it a topic worth further investigation.

This paper first provides a brief introduction to the basics of Spark, introduces common Spark scheduling efficiency evaluation metrics, compares several common Spark scheduling strategies, and analyzes how scheduling affects program execution efficiency. Then, it details and compares the optimization approaches

of existing Spark scheduling strategies from three perspectives: load, nodes, and the matching of the two. Several representative algorithms are discussed to highlight the connections and differences between different strategy characteristics. Subsequently, the advantages and disadvantages of the three scheduling modes are compared, along with an exploration of how Spark improves scheduling strategies for specific application scenarios. Finally, the limitations of existing research are analyzed, and the future is envisioned from aspects such as applications in emerging fields, prediction of execution time for distributed-system loads, relationships between loads, inspirations from other distributed fields, integration with blockchain, SDN-enabled Spark scheduling, and inspirations from classic cluster management systems.

Acknowledgement: We sincerely thank Professor Wei Liu from Xidian University for his valuable suggestions during the research process.

Funding Statement: This work was supported in part by the Key Research and Development Program of Shaanxi under Grant 2023-ZDLGY-34. The author Li C. received funding for this research. The funder's website is available at https://kjt.shaanxi.gov.cn/.

Author Contributions: The authors confirm their contribution to the paper as follows: Conceptualization, Chuan Li and Xuanlin Wen; methodology, Chuan Li; formal analysis, Xuanlin Wen; investigation, Xuanlin Wen; resources, Chuan Li and Xuanlin Wen; data curation, Xuanlin Wen; writing—original draft preparation, Xuanlin Wen; writing—review and editing, Chuan Li and Xuanlin Wen; visualization, Xuanlin Wen; supervision, Chuan Li; project administration, Chuan Li. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Wu X, Ji S. Comparative study on MapReduce and spark for big data analytics. J Softw. 2018;29(6):1770–91. (In Chinese). doi:10.13328/j.cnki.jos.005557.
2. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, et al. Apache spark: a unified engine for big data processing. Commun ACM. 2016;59(11):56–65. doi:10.1145/2934664.
3. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, et al., editors. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12); 2012 Apr 25–27; San Jose, CA, USA. p. 15–28.
4. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10); 2010 Jun 22; Boston, MA, USA.
5. Aziz K, Zaidouni D, Bellafkih M. Leveraging resource management for efficient performance of Apache Spark. J Big Data. 2019;6(1):78. doi:10.1186/s40537-019-0240-1.
6. Tang S, He B, Yu C, Li Y, Li K. A survey on spark ecosystem: big data processing infrastructure, machine learning, and applications. IEEE Trans Knowl Data Eng. 2022;34(1):71–91. doi:10.1109/TKDE.2020.2975652.
7. Liao H, Huang S, Xu J. Survey on performance optimization technologies for spark. Comput Sci. 2018;45(7):7–15,37. (In Chinese). doi:10.11896/j.issn.1002-137X.2018.07.002.
8. Jin GD, Bian HQ, Chen YG, Du XY. Survey on storage and optimization techniques of HDFS. J Softw. 2020;31(1):137–61. (In Chinese). doi:10.13328/j.cnki.jos.005872.
9. Wang Y, Zeng H, Xu L, Wang W, Wei J, Huang T. Survey on JVM optimization for big data processing frameworks. J Softw. 2021;34(1):463–88. (In Chinese). doi:10.13328/j.cnki.jos.006502.
10. Zaharia M. An architecture for fast and general data processing on large clusters. Berkeley, CA, USA: Morgan & Claypool; 2016.

11. Xian G. Parallel machine learning algorithm using fine-grained-mode spark on a mesos big data cloud computing software framework for mobile robotic intelligent fault recognition. IEEE Access. 2020;8:131885–900. doi:10.1109/ACCESS.2020.3007499.
12. Xu Y, Liu L, Ding Z. DAG-aware joint task scheduling and cache management in spark clusters. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2020 May 18–22; New Orleans, LA, USA. p. 378–87. doi:10.1109/ipdps47924.2020.00047.
13. Wang G, Xu J, Liu R, Huang S. A hard real-time scheduler for spark on YARN. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID); 2018 May 1–4; Washington, DC, USA. p. 645–52. doi:10.1109/CCGRID.2018.00096.
14. Neciu LF, Pop F, Apostol ES, Truica CO. Efficient real-time earliest deadline first based scheduling for Apache spark. In: 2021 20th International Symposium on Parallel and Distributed Computing (ISPDC); 2021 Jul 28–30; Cluj-Napoca, Romania. p. 97–104. doi:10.1109/ispdc52870.2021.9521640.
15. Saha P, Beltre A, Govindaraju M. Exploring the fairness and resource distribution in an Apache mesos environment. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD); 2018 Jul 2–7; San Francisco, CA, USA. p. 434–41. doi:10.1109/CLOUD.2018.00061.
16. Li W, Yao Z. Multicore architecture speedup computation based on Amdahl's law and Rent's rule. Dianzi Xuebao Acta Electron Sin. 2012;40(2):230–4. (In Chinese). doi:10.3969/j.issn.0372-2112.2012.02.004.
17. Li Y, Li T, Shen P, Hao L, Yang J, Zhang Z, et al. PAS: performance-aware job scheduling for big data processing systems. Secur Commun Netw. 2022;2022(2):8598305. doi:10.1155/2022/8598305.
18. Bian C, Yu J, Xiu YR. Parallelism deduction algorithm for spark. J Univ Electron Sci Technol China. 2019;48(4):567–74. (In Chinese). doi:10.3969/j.issn.1001-0548.2019.04.014.
19. Asgari M, Yang W, Farnaghi M. Spatiotemporal data partitioning for distributed random forest algorithm: air quality prediction using imbalanced big spatiotemporal data on spark distributed framework. Environ Technol Innov. 2022;27(2):102776. doi:10.1016/j.eti.2022.102776.
20. Fu Z, Tang Z, Yang L, Liu C. An optimal locality-aware task scheduling algorithm based on bipartite graph modelling for spark applications. IEEE Trans Parallel Distrib Syst. 2020;31(10):2406–20. doi:10.1109/TPDS.2020.2992073.
21. Wu R, Zhang Z, Jia Y, Qiao H. Adaptive scheduling strategy based on deadline under cloud platform. J Comput Appl. 2023;43(1):176. (In Chinese). doi:10.11772/j.issn.1001-9081.2021112018.
22. Cheng W, Wang Z, Zhou Y, Guo Y, Zhao J. Design of distributed computing framework for foreign exchange market monitoring. J Comput Appl. 2020;40(1):173. (In Chinese). doi:10.11772/j.issn.1001-9081.2019061002.
23. Wu Z, Sun J, Zhang Y, Zhu Y, Li J, Plaza A, et al. Scheduling-guided automatic processing of massive hyperspectral image classification on cloud computing architectures. IEEE Trans Cybern. 2021;51(7):3588–601. doi:10.1109/TCYB.2020.3026673.
24. Sun J, Li H, Zhang Y, Xu Y, Zhu Y, Zang Q, et al. Multiobjective task scheduling for energy-efficient cloud implementation of hyperspectral image classification. IEEE J Sel Top Appl Earth Obs Remote Sens. 2020;14:587–600. doi:10.1109/JSTARS.2020.3036896.
25. Zhang X, Qian Z, Zhang S, Li X, Wang X, Lu S. Semi-clairvoyant scheduling in data analytics systems. IEEE Trans Comput. 2019;68(9):1376–89. doi:10.1109/TC.2019.2906193.
26. Baresi L, Leva A, Quattrocchi G. Fine-grained dynamic resource allocation for big-data applications. IEEE Trans Softw Eng. 2021;47(8):1668–82. doi:10.1109/TSE.2019.2931537.
27. Zhao Y, Dong J, Liu H, Wu J, Liu Y. Performance improvement of DAG-aware task scheduling algorithms with efficient cache management in spark. Electronics. 2021;10(16):1874. doi:10.3390/electronics10161874.
28. Gu H, Li X, Lu Z. Scheduling spark tasks with data skew and deadline constraints. IEEE Access. 2020;9:2793–804. doi:10.1109/ACCESS.2020.3040719.
29. Zhu C, Han B, Li G. PAC: a monitoring framework for performance analysis of compression algorithms in Spark. Future Gener Comput Syst. 2024;157(5):237–49. doi:10.1016/j.future.2024.02.009.

30. Tong Y, Liu J, Wang H, He M, Zhou K, He R, et al. DAG-aware harmonizing job scheduling and data caching for disaggregated analytics frameworks. Future Gener Comput Syst. 2024;156(16):116–29. doi:10.1016/j.future.2024.03.005.
31. Hu Y, Sheng X, Mao J. Task scheduling optimization in Spark environment with unbalanced resources. Comput Eng Sci. 2020;42(2):203. (In Chinese). doi:10.3969/j.issn.1007-130X.2020.02.003.
32. Cheng D, Wang Y, Dai D. Dynamic resource provisioning for iterative workloads on Apache spark. IEEE Trans Cloud Comput. 2023;11(1):639–52. doi:10.1109/TCC.2021.3108043.
33. Meng L, Yang Y, Huang X, Lian L. Dynamic data partition based on node load. Comput Syst Appl. 2021;30(12):299–307. (In Chinese). doi:10.15888/j.cnki.csa.008224.
34. Hu Y, Qiu Y, Mao J. Node priority optimization in distributed heterogeneous clusters. J Natl Univ Def Technol. 2022;44(5):102–13. (In Chinese). doi:10.11887/j.cn.202205011.
35. Huang W, Zhou J, Zhang D. On-the-fly fusion of remotely-sensed big data using an elastic computing paradigm with a containerized spark engine on Kubernetes. Sensors. 2021;21(9):2971. doi:10.3390/s21092971.
36. Ahmed N, Barczak ALC, Susnjak T, Rashid MA. A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench. J Big Data. 2020;7(1):110. doi:10.1186/s40537-020-00388-5.
37. Hu YH, Wu YC, Zhu ZD, Li XX. Heterogeneous cluster resource allocation algorithm considering application and node characteristics. J Comput Eng Appl. 2022;58(18):327–34. (In Chinese). doi:10.3778/j.issn.1002-8331.2102-0275.
38. Zhao L, Li Y, Fogelman-Soulié F, Li K. A holistic cross-layer optimization approach for mitigating stragglers in in-memory data processing. J Syst Archit. 2020;111(1):101801. doi:10.1016/j.sysarc.2020.101801.
39. Bian C, Xiu WR, Yu J. A data skew correction scheduling strategy of heterogeneous Spark cluster. Comput Eng Sci. 2022;44(4):620–30. (In Chinese). doi:10.3969/j.issn.1007-130X.2022.04.006.
40. Islam MT, Srirama SN, Karunasekera S, Buyya R. Cost-efficient dynamic scheduling of big data applications in Apache spark on cloud. J Syst Softw. 2020;162(7):110515. doi:10.1016/j.jss.2019.110515.
41. Jing H, Qin B, Jiang X, Xia H. Distributed parallel task scheduling on spark-gpu framework for oceanographic geospatial data processing. Period Ocean Univ China. 2018;48:180–6. (In Chinese). doi:10.16441/j.cnki.hdxb.20180032.
42. Li C, Liu J, Li W, Luo Y. Adaptive priority-based data placement and multi-task scheduling in geo-distributed cloud systems. Knowl Based Syst. 2021;224(4):107050. doi:10.1016/j.knosys.2021.107050.
43. Chen Y, Hoque MA, Xu P, Lu J, Tarkoma S. SimCost: cost-effective resource provision prediction and recommendation for spark workloads. Distrib Parallel Databases. 2024;42(1):73–102. doi:10.1007/s10619-023-07436-y.
44. Fu Z, He M, Yi Y, Tang Z. Improving data locality of tasks by executor allocation in spark computing environment. IEEE Trans Cloud Comput. 2024;12(3):876–88. doi:10.1109/TCC.2024.3406041.
45. Zainab A, Ghrayeb A, Abu-Rub H, Refaat SS, Bouhali O. Distributed tree-based machine learning for short-term load forecasting with Apache spark. IEEE Access. 2021;9:57372–84. doi:10.1109/ACCESS.2021.3072609.
46. Du H, Zhang K, Xiang Q. Stargazer: toward efficient data analytics scheduling *via* task completion time inference. Comput Electr Eng. 2021;92(8):107092. doi:10.1016/j.compeleceng.2021.107092.
47. Tang Z, Zeng A, Zhang X, Yang L, Li K. Dynamic memory-aware scheduling in spark computing environment. J Parallel Distrib Comput. 2020;141(4):10–22. doi:10.1016/j.jpdc.2020.03.010.
48. Thamsen L, Beilharz J, Tran VT, Nedelkoski S, Kao O. Mary, Hugo, and Hugo*: learning to schedule distributed data-parallel processing jobs on shared clusters. Concurr Comput Pract Exp. 2021;33(18):e5823. doi:10.1002/cpe.5823.
49. Hu Y, Qiu Y, Mao J. Workflows scheduling powered by execution time prediction model. J Natl Univ Def Technol. 2024;46(5):228–38. (In Chinese). doi:10.11887/j.cn.202405024.
50. Li H, Luo W, Xie W, Ye H, Duan X. Adaptive scheduling framework of streaming applications based on resource demand prediction with hybrid algorithms. J Grid Comput. 2024;22(1):39. doi:10.1007/s10723-024-09756-4.
51. Du H, Zhang S. Hawkeye: adaptive straggler identification on heterogeneous spark cluster with reinforcement learning. IEEE Access. 2020;8:57822–32. doi:10.1109/ACCESS.2020.2982320.

52. Wu J. Distributed intelligent optimization of e-commerce user purchase data mining using spark framework. Informatica. 2024;48(20):29–40.
53. Arif A, Javaid N, Aldegheishem A, Alrajeh N. Big data analytics for identifying electricity theft using machine learning approaches in microgrids for smart communities. Concurr Comput Pract Exp. 2021;33(17):e6316. doi:10.1002/cpe.6316.
54. Li C, Zhu Y, Cao Y, Zhang J, Annisa A, Cheng D, et al. Mining area skyline objects from map-based big data using Apache Spark framework. Array. 2025;25(1):100373. doi:10.1016/j.array.2024.100373.
55. Fernández-Gómez AM, Gutiérrez-Avilés D, Troncoso A, Martínez-Álvarez F. A new Apache Spark-based framework for big data streaming forecasting in IoT networks. J Supercomput. 2023;79(10):11078–100. doi:10.1007/s11227-023-05100-x.
56. Shrotriya L, Sharma K, Parashar D, Mishra K, Rawat SS, Pagare H. Apache spark in healthcare: advancing data-driven innovations and better patient care. Int J Adv Comput Sci Appl. 2023;14(6):608–16. doi:10.14569/issn.2156-5570.
57. Fikri N, Rida M, Abghour N, Moussaid K, El Omri A. An adaptive and real-time based architecture for financial data integration. J Big Data. 2019;6(1):97. doi:10.1186/s40537-019-0260-x.
58. Bian S, Mao R, Zhu Y, Fu Y, Zhang Z, Ding L, et al. A survey on software-hardware acceleration for fully homomorphic encryption. J Electron Inf Technol. 2024;46(5):1790–805. (In Chinese). doi:10.11999/JEIT230448.
59. Si C, Xu S, Wan C, Chen D, Cui W, Zhao J. Electric load clustering in smart grid: methodologies, applications, and future trends. J Mod Power Syst Clean Energy. 2021;9(2):237–52. doi:10.35833/MPCE.2020.000472.
60. Maheshwari S, Gautam P, Jaggi CK. Role of big data analytics in supply chain management: current trends and future perspectives. Int J Prod Res. 2021;59(6):1875–900. doi:10.1080/00207543.2020.1793011.
61. Ahmed N, Barczak ALC, Rashid MA, Susnjak T. A parallelization model for performance characterization of Spark Big Data jobs on Hadoop clusters. J Big Data. 2021;8(1):107. doi:10.1186/s40537-021-00499-7.
62. Ahmed N, Barczak ALC, Rashid MA, Susnjak T. Runtime prediction of big data jobs: performance comparison of machine learning algorithms and analytical models. J Big Data. 2022;9(1):67. doi:10.1186/s40537-022-00623-1.
63. Singh R, Kaur PJ. Analyzing performance of Apache Tez and MapReduce with hadoop multinode cluster on Amazon cloud. J Big Data. 2016;3(1):19. doi:10.1186/s40537-016-0051-6.
64. Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge computing: vision and challenges. IEEE Internet Things J. 2016;3(5):637–46. doi:10.1109/JIOT.2016.2579198.
65. Zhang Y, Liang Y, Yin M, Quan H, Wang T, Jia W. Survey on the methods of computation offloading in mobile edge computing. J Comput Sci. 2021;44:2406–30.
66. Liao X, Xiao N. Emerging high-performance computing systems and technology. Sci Sin-Inf. 2016;46(9):1175–210. doi:10.1360/N112016-00147.
67. Chen G, Guan N, Lü MS, Wang Y. State-of-the-art survey of real-time multicore system. J Softw. 2018;29(7):2152–76. (In Chinese). doi:10.13328/j.cnki.jos.005580.
68. Rao IS, Mat Kiah ML, Hameed MM, Memon ZA. Scalability of blockchain: a comprehensive review and future research direction. Clust Comput. 2024;27(5):5547–70. doi:10.1007/s10586-023-04257-7.
69. Zhong W, Yang C, Liang W, Cai J, Chen L, Liao J, et al. Byzantine fault-tolerant consensus algorithms: a survey. Electronics. 2023;12(18):3801. doi:10.3390/electronics12183801.
70. Rao Y, Wu DL, Shi YL. Design and implementation of trusted fusion and sharing model of agricultural data based on Blockchain. J Anhui Agric Univ. 2023;50(3):550–6. (In Chinese). doi:10.13610/j.cnki.1672-352x.20230625.004.
71. Sellami B, Hakiri A, Ben Yahia S, Berthou P. Energy-aware task scheduling and offloading using deep reinforcement learning in SDN-enabled IoT network. Comput Netw. 2022;210(3):108957. doi:10.1016/j.comnet.2022.108957.
72. Mahdizadeh M, Montazerolghaem A, Jamshidi K. Task scheduling and load balancing in SDN-based cloud computing: a review of relevant research. J Eng Res. 2024;2024(12):1–15. doi:10.1016/j.jer.2024.11.002.
73. Salehnia T, Montazerolghaem A, Mirjalili S, Khayyambashi MR, Abualigah L. SDN-based optimal task scheduling method in Fog-IoT network using combination of AO and WOA. In: Handbook of whale optimization algorithm. Amsterdam, The Netherlands: Elsevier; 2024. p. 109–28.

74. Tirmazi M, Barker A, Deng N, Haque ME, Qin ZG, Hand S, et al., editors. Borg: the next generation. In: Proceedings of the Fifteenth European Conference on Computer Systems; 2020 Apr 27–30; Heraklion, Greece. p. 1–14. doi:10.1145/3342195.3387517.
75. Carrión C. Kubernetes scheduling: taxonomy, ongoing issues and challenges. ACM Comput Surv. 2023;55(7):1–37. doi:10.1145/3539606.