Towards Resource Efficiency: Practical Insights into Large-Scale Spark Workloads at ByteDance

Yixin Wu* ByteDance Inc.
Xiuqi Huang* Shanghai Jiao Tong University
Zhongjia Wei ByteDance Inc.
Hang Cheng ByteDance Inc.
Chaohui Xin ByteDance Inc.
Zuzhi Chen ByteDance Inc.
Binbin Chen ByteDance Inc.
Yufei Wu ByteDance Inc.
Hao Wang ByteDance Inc.
Tieying Zhang ByteDance Inc.
Rui Shi† ByteDance Inc.
Xiaofeng Gao Shanghai Jiao Tong University
Yuming Liang ByteDance Inc.
Pengwei Zhao ByteDance Inc.
Guihai Chen Shanghai Jiao Tong University

ABSTRACT

At ByteDance, where we execute over a million Spark jobs and handle 500PB of shuffled data daily, ensuring resource efficiency is paramount for cost savings. However, achieving optimization of resource efficiency in large-scale production environments poses significant challenges. Drawing from our practical experiences, we have identified three key issues critical to addressing resource efficiency in real-world production settings: ① slow I/Os leading to excessive CPU and memory idleness, ② coarse-grained resource control causing wastage, and ③ sub-optimal job configurations resulting in low utilization. To tackle these issues, we propose a resource efficiency governance framework for Spark workloads. Specifically, ① we devise the multi-mechanism shuffle services, including Enhanced External Shuffle Service (ESS) and Cloud Shuffle Service (CSS), where CSS employs a push-based approach to enhance I/O efficiency through sequential reading. ② We modify the Spark configuration parameter protocol, allowing for fine-grained resource control by introducing several new parameters such as milliCores and memoryBurst, as well as supporting operators with additional spill modes. ③ We design a two-stage configuration auto-tuning method, comprising rule-based and algorithm-based tuning, providing more reliable Spark configuration optimizations. By deploying these techniques on millions of Spark jobs in production over the last two years, we have achieved over 22% CPU utilization increase, 5% memory utilization increase, and 10% shuffle block time ratio decrease, effectively saving millions of CPU cores and petabytes of memory daily.

PVLDB Reference Format:
Yixin Wu, Xiuqi Huang, Zhongjia Wei, Hang Cheng, Chaohui Xin, Zuzhi Chen, Binbin Chen, Yufei Wu, Hao Wang, Tieying Zhang, Rui Shi, Xiaofeng Gao, Yuming Liang, Pengwei Zhao, and Guihai Chen. Towards Resource Efficiency: Practical Insights into Large-Scale Spark Workloads at ByteDance. PVLDB, 17(12): 3759 - 3771, 2024.
doi:10.14778/3685800.3685804

*Yixin Wu and Xiuqi Huang contributed equally to this work.
†Dr. Rui Shi is the corresponding author, shirui@bytedance.com.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.

1 INTRODUCTION

At ByteDance, Apache Spark [10] is the most widely used compute engine for large-scale data processing, with more than 1.7 million Spark jobs executed daily by various teams across the company. Despite several prior attempts [21, 35, 37, 41] to optimize Spark workloads, such large-scale and diverse applications at ByteDance bring unique and complex challenges to resource efficiency.

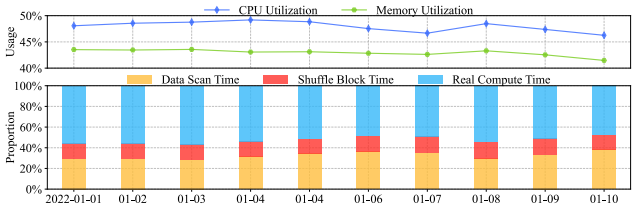

Figure 1: An Example of Production Resource Efficiency- It shows ByteDance's resource utilization for millions of Spark jobs in the first 10 days of 2022. Data scan and shuffle block time consume more than 45% of the total compute time. The average CPU utilization is 47.98% and the memory utilization is 42.95%.

In Figure 1, we show the resource utilization and time proportion of ByteDance production workloads before implementing resource efficiency enhancements, where CPU and memory utilization remains in a low range. Primary factors that impact resource efficiency include Hadoop Distribute File System (HDFS) slowness, shuffle fetch failures [34], coarse-grained resource control [38] and sub-optimal job configurations [24]. This highlights the primary directions for our work towards resource efficiency, including reducing slow I/Os, refining resource control, and tuning configuration parameters. However, previous methods [7, 15, 21, 24] are not sufficient to handle the large-scale Spark workloads at ByteDance, as the following special challenges need to be addressed.

① **Expensive I/O costs.** Spark's data scan and shuffle operations are both resource-intensive and time-consuming. On the one hand, when reading remote data from HDFS, waiting for I/O operations causes certain periods of CPU and memory idleness. On the other hand, Spark's External Shuffle Service (ESS) shares

doi:10.14778/3685800.3685804

the disk resources with other computing processes on the same node, which may result in fetch failures due to high disk pressure. Moreover, as a single ESS process serves all of the intermediate shuffle data on a compute node, the abnormality of a single job can potentially exacerbate faults and impact other shuffle tasks on the same node. Although some approaches [15, 32, 34, 42] have been proposed to improve shuffle efficiency, they cannot meet stability and performance needs at our scale.

② **Coarse-grained resource control.** With the explosive growth in Spark workloads, there is an urgent need to further enhance the resource efficiency of production clusters by reducing both resource allocation and actual usage. Previous methods are mostly focused on choosing server specifications to match jobs' demands [7, 37] or combining resource utilization and cost as the optimization goal [5], which is hard to handle resource requirements variation of different stages. Although Spark provides stage-level resource settings using ResourceProfile [8], adoption is hindered by the required changes to user code and the lack of support for SQL. Besides, Spark's minimum granularity of resource allocation is one CPU core, that a task is allocated at least one core, potentially resulting in inefficient CPU and memory utilization.

③ **Sub-optimal Spark configuration.** Confronted with diverse business needs, manually setting the appropriate parameters for Spark jobs is extremely time-consuming, given the varying characteristics and resource demands of Spark applications. In large-scale production clusters, job interference, bandwidth fluctuations, and workload changes further increase the difficulty for automatic configuration tuning methods to adapt to various applications and a dynamic production environment. However, the majority of configuration tuning methods focus on performance optimizations [4, 22, 24, 41, 44], with relatively fewer approaches considering resource efficiency [5, 21], particularly rare [35] enabling Spark's dynamic allocation feature [11].

Our Methodologies. We design a resource efficiency governance framework for Spark workloads. This framework is designed to enhance the stability, performance, and resource utilization of Spark jobs through a series of techniques implemented from the bottom up. Among them, there are three main techniques to solve the above challenges. ① We provide multi-mechanism shuffle services to improve the stability of shuffle and reduce I/O delay. ② We design a fine-grained resource control mechanism to accurately adjust job resource allocations according to their actual usage. ③ We devise a two-stage configuration auto-tuning method to provide appropriate parameters for various jobs. These three techniques work in tandem to improve the overall resource efficiency of Spark workloads. In particular, the multi-mechanism shuffle services free up idle CPU and wasted memory caused by slow shuffles, which are then leveraged by fine-grained resource control and two-stage configuration tuning.

Contribution. For large-scale Spark workload, we summarize four key contributions are as follows:

- Based on the characteristics of ByteDance production clusters, we design the multi-mechanism shuffle services which include Enhanced ESS with request throttling and executor rolling, as well as a push-based Cloud Shuffle Service (CSS). This design improves shuffle stability and efficiency, significantly reducing shuffle fetch failures and shuffle block time. (Sec. 3)
- We enable fine-grained resource control by modifying underlying Spark core modules by introducing new CPU and memory allocation parameters. Also, we support additional spill modes for Spark operators to reduce memory footprint and out-of-memory (OOM) failures. (Sec. 4)
- We establish an end-to-end online tuning pipeline, which employs a two-stage configuration auto-tuning method combining both rule-based and algorithm-based tuning. This method is most effective for enhancing CPU and memory utilization in production environments while prioritizing stability. (Sec. 5)
- These techniques have been widely applied across ByteDance production clusters, yielding a significant improvement in resource efficiency. Over 1.7 million Spark jobs, we have improved CPU utilization from 48% to over 70% and memory utilization from 43% to 50%. During the month of March 2024, we have optimized more than 530,000 jobs, reducing the average job execution time by 11.1 minutes, with over 1 million CPU cores and 4.6 PB memory saved daily. (Sec. 6)

2 OVERVIEW AND SYSTEM DESIGN

In this section, we provide an overview of Spark at ByteDance and our proposed resource efficiency governance framework.

2.1 Overview of Spark at ByteDance

Figure 2 illustrates the lifecycle of a Spark application. Upon a user's submission, a driver initializes and interprets the submitted application into multiple jobs, and generates a Directed Acyclic Graph (DAG) for each job. Each DAG, consisting of various stages requiring data shuffling in between, is scheduled by the DAGScheduler. Each stage consists of parallel tasks performing identical functions, all of which are scheduled to execute on executors. Both executors and ESS run on containers allocated in the clusters managed by Yodel (YARN on Gödel [40]). Typically, the active tasks interact with the HDFS for data scanning. Below, we provide detailed background information pertinent to the Spark jobs at ByteDance.

At ByteDance, clusters are categorized into two types: dedicated and mixed. Dedicated clusters, equipped with solid-state disks (SSD), offer stable resources for high-priority jobs. Despite SSDs offering improved I/O performance, maintaining shuffle stability in large-scale workloads still remains challenging. Mixed clusters, on the other hand, share disk resources with various services, such as online services and HDFS. The sharing leads to increased competition for disk I/Os and capacity, which exacerbates shuffle stability issues.

Gödel, a resource management and scheduling system based on Kubernetes [3], is deployed across the aforementioned clusters, offering a unified computing infrastructure and resource pool. Prior to Gödel's deployment, cluster resources were managed by YARN. To facilitate the smooth transition of Spark from YARN to Kubernetes, Yodel was developed, providing a YARN-compatible interface atop Gödel. These Yodel clusters, with tens of millions of CPU cores, are responsible for processing large-scale Spark workloads.

With over 1.7 million daily Spark applications, of which 75% are periodic jobs, optimizing Spark configurations to improve utilization and performance is crucial for our company. However,

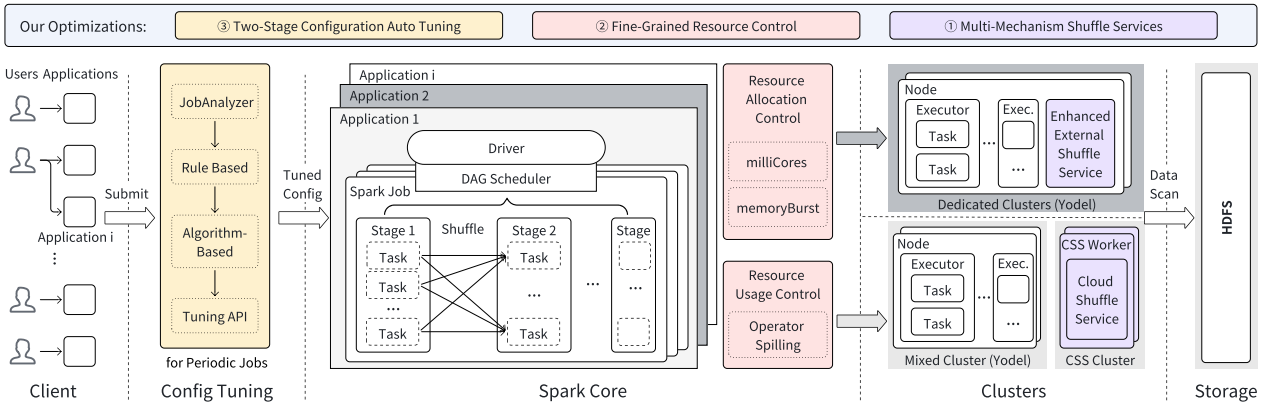

Figure 2: Design of Resource Efficiency Governance Framework - It shows three optimizations we proposed for enhancing resource efficiency within Spark, including multi-mechanism shuffle services (purple, Sec. 3), fine-grained resource control (pink, Sec. 4), and two-stage configuration auto-tuning (orange, Sec. 5). These optimizations are implemented in the lifecycle (white) of Spark applications.

manually tuning the jobs poses a significant challenge due to the vast number of jobs and a limited understanding of Spark among most users. Furthermore, the iterative tries of tuning to achieve optimal configurations result in resource wastage. Consequently, there is a strong demand for auto-tuning solutions for these periodical jobs by leveraging historical metrics.

2.2 System Design for Resource Efficiency

As mentioned in Sec. 1, there are three main challenges that affect resource efficiency when running Spark jobs, including expensive I/O costs, coarse-grained resource control, and sub-optimal Spark configuration. To meet these challenges, we have built a resource efficiency governance framework as shown in Fig. 2, including multi-mechanism shuffle services, fine-grained resource control, and two-stage configuration auto-tuning.

Multi-Mechanism Shuffle Services. Shuffle consumes a lot of time during the execution of Spark jobs [45] and is an efficiency and stability bottleneck for large-scale Spark clusters [14]. At ByteDance, the amount of daily shuffle data exceeds 500 PB, with individual jobs exceeding hundreds of TB. At this scale, increasing shuffle stability and efficiency results in large amounts of resource savings. According to the two types of clusters, we provide multi-mechanism shuffle services including Enhanced ESS and CSS. Enhanced ESS leverages request throttling and executor rolling strategies to reduce cases of shuffle fetch failures and retries when using ESS in dedicated clusters. The CSS is a remote shuffle service that decouples shuffle performance and stability from limited disk I/O and space in mixed clusters. Therefore, we improve the performance and stability of Spark shuffle on these clusters.

Fine-grained Resource Control. The resource utilization of Spark applications is affected by user-specified resource requirements and actual resource usage during application execution. While Spark tasks across different stages can have large variations in CPU and memory needs, they are all executed on executors under the identical resource configuration. Hence, users often set their resource requirements based on the most resource-intensive stages, leading to excessive resource wastage in other stages. With the introduction of ResourceProfiles in Spark v3.1.1 [8], users can specify the resource requirements of tasks and executors at the stage level. However, this feature lacks support for SQL and heavily relies on users' manual tweaks, therefore can not be applied widely at large scale. To this end, by taking advantage of Yodel's elastic scaling mechanism, we enhanced the Spark Core module to introduce the support for milliCores and memoryBurst, enabling finer resource allocation and improving resource utilization. For the actual usage of resources, we have added some new spill modes with configuration parameters for operator spill, which can further reduce the maximum memory usage and OOM failures.

Two-Stage Configuration Auto Tuning. Spark configuration parameter settings significantly influence resource efficiency, particularly in large-scale workloads. Sub-optimal job configurations can lead to substantial resource wastage. We devise an end-to-end auto-tuning pipeline, relieving job owners of the burden of studying and tuning a large number of Spark configurations. Our auto-tuning pipeline employs a two-stage auto-tuning method that leverages metrics collecting and aggregating from applications to deliver tuned configurations tailored to each job's requirements. These configurations encompass parameters related to shuffle operations, alongside newly introduced parameters by fine-grained resource control. The rule-based tuning draws upon the extensive expertise of Spark experts to devise heuristic rules from performance metrics, facilitating quick job configuration tuning. The configurations will be iteratively tuned with each job execution. When jobs occur OOM failures, these jobs would temporarily disable tuning rules, and use the algorithm-based tuning to explore configurations with better resource efficiency.

3 MULTI-MECHANISM SHUFFLE SERVICES

Shuffle involves expensive all-to-all data transfer between tasks of adjacent stages. Remote fetch failure and retries due to shuffle instability can cause significant resource wastage and low CPU utilization or stage retries. We propose multi-mechanism shuffle services that specifically address shuffle issues of dedicated clusters

and mixed clusters. For dedicated clusters, we deploy Enhanced ESS, which improves ESS with request throttling and executor rolling. For mixed clusters, we develop our own push-based CSS that enables shuffle partitions to be pushed to and merged by remote servers. The Enhanced ESS and CSS improve the stability of shuffle services and reduce the shuffle block time.

3.1 Enhanced External Shuffle Service (ESS)

Enhanced ESS is used in dedicated clusters where disk resources are mainly SSDs which provide much higher disk I/Os and throughput. We equip ESS with two capabilities: request throttling and executor rolling, aimed at improving stability. Request throttling serves to prevent shuffle service overload by excessive requests from abnormal applications. Executor rolling promotes a more uniform distribution of shuffle data across nodes within the cluster.

Request Throttling. This feature prevents a single application from exhausting the resources of an ESS node and affecting the shuffle performance of other applications in the cluster. Applications with an excessively large number of shuffle tasks produce numerous small chunks during shuffle write, subsequently leading to a large number of fetch requests to the ESS nodes during the shuffle read stage. Such cases result in millions of shuffle requests to be queued on an ESS node, with an average chunk size of less than 20KB, causing high random I/Os on the node. As a result, the impacted ESS nodes are unable to handle shuffle requests from other applications due to some misconfigured applications.

Shuffle block fetch latency L is used as a metric to identify shuffle congestion. Every 5 seconds, ESS reads the recently collected shuffle latency statistics on the node to obtain the overall fetch latency L_{ess}. When L_{ess} exceeds a latency threshold L_θ of 10 seconds, request throttling will be triggered on the ESS node. Then, for each application, an allocated chuck fetch rate $\hat{fr_a}$ is calculated based on the maximum overall chuck fetch rate fr_{ess} within the last 5 minutes of the ESS node.

$$\hat{fr_a} = \frac{\max_T^{T-\Delta T} fr_{ess}}{(1+\log N) \times p} \tag{1}$$

Among Eq. 1, T denotes the current time, while $\Delta T = 5$ minutes signifies the time interval. The $p \in [1, 2, 3, 4, 5]$ denotes the application priority code, where 1 is of the highest priority. N is the number of applications fetching shuffle chunks on the ESS node. We establish an exponential relationship between fr_a and N, based on production experience.

Applications with an observed chuck fetch rate fr_a exceeding its allocated chuck fetch rate $\hat{fr_a}$ will have their allowed queued chunks limited at $\hat{qc}$. We use the largest fr_a within the last 5 minutes, multiplied by the latency threshold L_θ, to calculate $\hat{qc}$ as shown in Eq. 2.

$$\hat{qc} = \max_T^{T-\Delta T} fr_a \times L_\theta \times \lambda_p \tag{2}$$

With $L_\theta = 10$ seconds, the calculated $\hat{qc}$ is the expected queued chunks the application is able to handle within 10 seconds.

Periodically every 30 seconds, the throttling limits $\hat{qc}$ will be recalculated based on the latest fr_a, and the current L_{ess} will be compared with the maximum observed L_{ess}. If the latency continues to deteriorate, $\hat{qc}$ will be reduced by a ratio λ_p, proportional to the job priority, resulting in lower limits for lower priority applications. λ_p is initially 1, and when the overall fetch latency L_{ess} returns to normal, λ_p will be reset to the initial value and applications that are previously limited will remain limited for up to 30 minutes since the initial time of the initial throttling, to prevent impacting other applications.

Executor Rolling. Through statistical analysis of historical applications with slow shuffles, we have observed that there is a high correlation between slow shuffle reads and the volume of shuffle data written on nodes. Specifically, the nodes with the top 5 shuffle write volumes contribute to half of the slow reads, and the top 2 nodes account for 35% of the slow reads. Therefore, ensuring even distribution of shuffle write data on ESS nodes can significantly improve shuffle performance.

An executor rolling strategy is introduced to limit the total shuffle write size of each executor. When a task ends, the driver accumulates the shuffle write size reported by each executor. When an executor's shuffle write reaches a specified threshold, the executor is released and a new executor will be requested subsequently. At ByteDance, the threshold for triggering executor rolling is set to 20GB, which is the 95th percentile (P95) of all executor shuffle write sizes measured previously. Additionally, scheduling policies are implemented on the scheduler to provide soft constraints for distributing containers evenly across the cluster.

The uneven distribution of shuffle write data on ESS hosts can be attributed mostly to two reasons. Firstly the shuffle write sizes of different applications are highly skewed, leading to some applications with extremely large shuffle write sizes, leaving large shuffle data on the nodes that they run on. Also due to resource constraints, some applications may only use a fraction of their requested executors at earlier stages, leading to uneven distribution of shuffle data on those allocated executors. The executor rolling approach is helpful in both cases. For the first case, taking advantage of ByteDance's large cluster scale, data can be distributed across more nodes in the cluster through the strategy. For example, a large application previously using 2,000 executors can have its shuffle data spread more evenly across more than 2,000 nodes using the rolling strategy. For the second one, by releasing and acquiring new executors on different nodes, the shuffle data can be distributed more evenly.

3.2 Cloud Shuffle Service (CSS)

The shuffle stability issues due to fetch failure and slow shuffle read I/Os in mixed resource clusters are usually much more severe than those in dedicated clusters. To this end, we have developed CSS [1], a push-based remote shuffle service, to eliminate the dependence of jobs on local disks. CSS allows sequential fetch during shuffle read, while disaggregating the compute and storage nodes for shuffle, therefore improving the performance and reliability of Spark shuffle on mixed clusters.

The components of a typical CSS cluster are shown in Fig. 3. (1) The Cluster Manager is responsible for resource allocation and maintaining the status of registered workers and applications. Stateful information can be persisted on Zookeeper or external storage to achieve high availability and fault tolerance. (2) CSS Workers accept incoming data from shuffle applications and support two

[1] CSS Open-Source Codebase: https://github.com/bytedance/CloudShuffleService

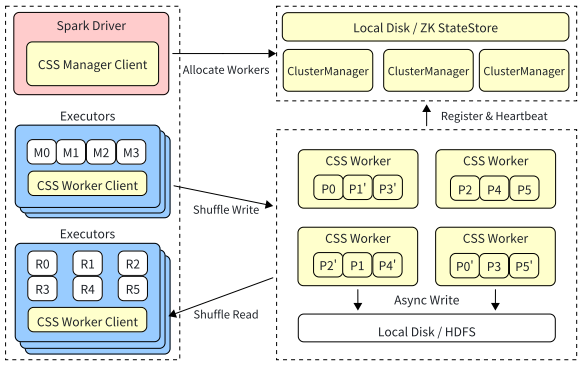

Figure 3: Overview of Cloud Shuffle Service

data persistence modes, namely HDFS and local disk storage. Shuffle data of the same partition is written to one primary worker and one backup worker to achieve fault tolerance. A CSS cluster can be scaled by adding more active workers. (3) The CSS Manager Client is a shuffle application lifecycle manager running on the Spark Driver. It is responsible for reporting the heartbeats of each running application and ongoing shuffles. It also sends requests for workers from the Cluster Manager and tracks progress and metadata information for each shuffle stage, including the status and location of each shuffle partition. (4) CSS Worker Client implements Spark's ShuffleManager interface and is used by shuffle tasks running on executors to write and subsequently fetch data from the CSS workers. During shuffle write stages, each map task writes output shuffle data of the same partition to the same pair of CSS workers. During shuffle read stages, each reducer task can sequentially fetch its corresponding partition data in large chunks from any of the CSS workers and switch over to the other when encountering failures or slowness.

CSS has the following key features that contribute to its performance and stability improvements in mixed clusters.

(1) Unlike ESS, CSS shuffle tasks push data of the same partition to the same group of remote servers. The servers buffer received data in memory for each partition before flushing them to disk in chunks, resulting in merged data files for partition data. Subsequent tasks can read the data files for each partition sequentially, which can significantly reduce the I/O cost compared to random I/Os.

(2) Partition Groups can be used to allocate multiple partitions on the same group of remote servers. This allows tasks to push shuffle data from multiple partitions in a batch during shuffle write, reducing the number of I/O requests during shuffle write.

(3) Fast in-memory data replication is used to increase shuffle fault tolerance at a lower cost. Due to the merging of task partitions with CSS workers, data loss can be costly to retry as it requires all tasks that write to the partition to retry during the stage retry. Data replication reduces the chance of shuffle stage failure when data lost on a CSS worker occurs. During shuffle write, each partition data P pushed by shuffle tasks also has a replica P' pushed to another worker simultaneously. CSS increases the speed of data replication by returning after successful in-memory writes and subsequent asynchronous disk flushing, allowing tasks to continue processing data push without waiting for disk flushing after each write request. This mechanism has proved to be very effective in increasing shuffle write speed with a low failure rate in production.

(4) Load balancing through the Cluster Manager, which periodically collects statistics and performance metrics reported by each CSS worker. During requests from applications to allocate CSS worker resources, the Cluster Manager allocates workers with lower shuffle push or fetch I/Os, increasing the overall utilization and performance of the cluster.

4 FINE-GRAINED RESOURCE CONTROL

We enhanced Spark by modifying its configuration parameters and interaction protocols for fine-grained resource control to better leverage Yodel's capabilities. For resource allocation, with the executor elastic scaling mechanism provided by Yodel, we add milliCores and memoryBurst parameters to align resource settings more closely with average utilization. Besides, the introduction of milliCores facilitates the decoupling of resource requests and task parallelism. For actual resource usage, we introduce new spill modes to Spark operators with relevant parameters, making data spilling to disk more flexible and reducing the maximum memory usage. These two mechanisms can significantly reduce OOM failures while improving resource efficiency through fine-grained resource control over both allocation and actual usage.

4.1 Resource Allocation Control

As mentioned in Sec. 2, resource requirements of Spark applications vary greatly at different stages. Spark v3.1.1 introduces the ResourceProfile feature, enabling users to specify executor resource requirements at the stage level through executor reallocation among different stages. However, this functionality is currently limited to RDDs and does not extend to SQL. Reallocating executors represents a significant expense that can adversely affect performance. Additionally, the utilization of this feature requires modifying user codes with the RDD.withResources and ResourceProfileBuilder APIs, rendering it less appropriate for large-scale applications.

In contrast to utilizing ResourceProfile, our solution avoids the reallocation of executors and is configurable without code changes. Leveraging Kubernetes resource management, specifically regarding requests and limits, we can control container resource utilization through request and limit parameters for CPU and memory, enabling more elasticity in executor resources. If a node where the container is running has enough resources available, it's possible (and allowed) for a container to use more resources than the requested resources. However, a container is not allowed to use more than its resource limit. While configuring Spark applications' resource setting, the request parameter is determined according to historical average CPU utilization, and the limit parameter is based on historical peak CPU utilization. To facilitate this, we adapted the resource protocol of Yodel, incorporating support for related Spark configurations to configure request and limit settings for actual scheduled containers' CPU and memory.

For CPU resources, the minimum granularity of CPU per task is one core in Spark, during I/O waiting periods, available CPU resources will be underutilized. To address this issue and decouple

Table 1: Supporting Operators with New Spill Modes for Resource Usage Control

Operator	Supported Spill Modes	Configuration Parameters	Example Values
Shuffle Write / Sort	Force Spill by Number of Records	`spark.shuffle.spill.numElementsForceSpillThreshold`	1000000
	Force Spill by Memory Used	`spark.{shuffle, unsafe.sorter}.spill.recordsSizeForceSpillThreshold`	512m, 1g, 2g
	Force Spill by Fraction of Memory Used	`spark.{shuffle, unsafe.sorter}.spill.recordsSizeForceSpillFraction`	0.1, 1.0
	Allow Spill by Memory Used	`spark.{shuffle, unsafe.sorter}.spill.recordsSizeAllowSpillThreshold`	512m, 1g, 2g
	Allow Spill by Fraction of Memory Used	`spark.{shuffle, unsafe.sorter}.spill.recordsSizeAllowSpillFraction`	0.1, 1.0
Shuffle Read	Control memory when fetching data	`spark.executor.memoryDirect`	4096m, 8192m
Aggregate	Force Spill by Memory Used	`spark.sql.TungstenAggregate.forceFallbackBytes`	512m, 1g, 2g
	Force Spill by Fraction of Memory Used	`spark.sql.TungstenAggregate.forceFallbackFraction`	0.1, 1.0

resource allocation from task parallelism, we introduced a new configuration parameter called `spark.executor.milliCores`. This parameter is solely used to specify the CPU request for the executor, taking precedence over `spark.executor.cores`, while task parallelism is still determined by the values of `spark.executor.cores`. Besides, `spark.executor.milliCores` supports a granular unit of 1/1000 CPU core, allowing for more precise resource requirements settings. Hence, we can increase CPU utilization by reducing CPU requests while maintaining task parallelism. Subsequently, to optimize application performance following the reduction in CPU requests, the CPU limit is typically set to a multiple of `spark.executor.milliCores`. The specific multiplier for each cluster can be configured and is empirically set to 2 without jeopardizing the overall users' experience.

The memory resource used by Spark executors can be divided into two parts: JVM memory and memory overhead, which are determined by the parameters `spark.executor.memory` and `spark.executor.memoryOverhead`, respectively. In the default on-heap memory mode, the majority of execution memory resides in the JVM memory. Given that the JVM lacks elasticity in managing this part of memory, configuring JVM memory carries a high risk of causing out-of-memory exceptions. Conversely, configuring overhead memory poses lower risks since its use is more flexible. Similar to CPU, we added `spark.executor.memoryBurst.ratio` to specify the ratio that Spark uses to decrease the request setting for the overhead memory portion of the total executor memory while keeping the limit setting unchanged. We can consider that the actual memory overhead δ allocated to executors after memory overhead optimization is equal to:

$$\delta = x_{memO} - \min\{x_{mem} + x_{memO} \times (x_{memB} - 1), x_{memO}\} \quad (3)$$

where x_{memO}, x_{mem}, and x_{memB} represent the corresponding `spark.executor.memoryOverhead`, `spark.executor.memory`, and `spark.executor.memoryBurst.ratio` parameters. The setting of `spark.executor.memoryBurst.ratio` considers the balance between stability and memory utilization. By default, the value is set to 1.2, which improves memory utilization while preventing the job from OOM failures.

4.2 Resource Usage Control

In Spark, the MemoryManager monitors the memory usage of task operators while an executor is running. Each task operator can request and utilize memory up to its allocated limit. The strategy, also known as "lazy spill", only triggers spill operations when the task's available memory becomes insufficient. Under certain circumstances, operators may spill their in-memory data to disk. Alternatively, when a task requests memory from the MemoryManager and faces memory scarcity, the MemoryManager may instruct other operators supporting spilling to free up memory by spilling their data to disk. However, Spark's memory management using this strategy is relatively coarse, which is a significant cause of OOM failures.

In comparison to memory, there is ample space available in disks to accommodate spilled data. To this end, We have improved Spark's memory management by introducing new spill modes for underlying Spark operators, such as sort and aggregation, enabling fine-grained resource usage control. Specifically, in addition to Spark's existing "Force Spill by Number of Records" mode, we have introduced four additional spill modes. The meanings of new spill modes are as follows:

(1) Force Spill by Memory Used: Forces this operator to spill data to disk when its number of records in memory exceeds the specified amount.
(2) Force Spill by Fraction of Memory Used: Forces this operator to spill data to disk when the amount of execution memory used is greater than the specified fraction of the maximum execution memory.
(3) Allow Spill by Memory Used: The operator can be triggered to spill by other operators when the amount of execution memory used is greater than the specified amount.
(4) Allow Spill by Fraction of Memory Used: The operator can be triggered to spill by other operators when the amount of execution memory used is greater than the specified fraction of the maximum execution memory.

Table 1 shows the spill operators and currently supported modes. Both shuffle write and sort operators support one of the original and all four new spill modes, with similar related parameters. The memory of the shuffle read operator is directly controlled when fetching data, while the aggregate operator adds two force spill modes. With improvements made to spill, we can accurately control the memory used by Spark operators, ensuring that memory is allocated and released more efficiently.

5 TWO-STAGE CONFIG AUTO TUNING

The configuration parameters of Spark jobs have a significant influence over resource utilization. However, in production environments, conducting parameter-tuning experiments for each job is impractical, especially with methods [19, 24, 37] that incur additional execution costs. Thus, we have established an online tuning pipeline specifically for periodical jobs that takes full advantage of running metrics. Instead of pre-tuning parameters, we start with

default or user-defined parameters (usually sub-optimal) and record the running metrics of each job. These metrics are then analyzed to improve the parameters for the next execution of the job. To achieve quick and stable convergence of online tuning, we have developed a two-stage configuration auto-tuning method as shown in Fig. 4. The first stage is rule-based tuning, leveraging manually crafted rules by Spark experts, thus circumventing inefficient exploration. The second stage involves algorithm-based tuning, wherein the Bayesian optimization algorithm is refined for stability. It aims to find better parameters while minimizing the probability of OOM failures in production.

5.1 Online Tuning Pipeline

Figure 4 illustrates the workflow of our online tuning pipeline featuring the two-stage configuration auto-tuning method. It consists of four components: (1) Tuning API is responsible for interacting with the data platform and end users, which allows users to monitor and select tuning options of CPU, memory, and shuffle. (2) JobAnalyzer analyzes Eventlog data during Spark job execution and other data from clusters to generate job running metrics in real time. (3) Rule-Based Tuning is composed of several heuristic rules, which take the running metrics of jobs as input and generate tuned parameters based on the relationships of the heuristic rules in the rule tree. (4) Algorithm-Based Tuning is specially optimized for the stability requirements of online parameter tuning. It explores parameters with better performance based on the running metrics of historical parameters.

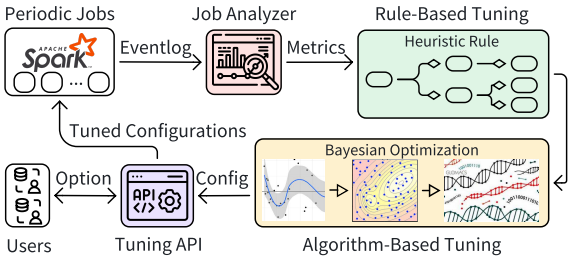

Figure 4: Overview of Two-Stage Config Auto Tuning

Users can flexibly enable or disable configuration auto-tuning via the Tuning API. When auto-tuning is activated for a periodical job, the Tuning API receives the job and initiates the entire tuning process. Before each job execution thereafter, the Tuning API retrieves the latest tuned configurations.

The Spark Eventlog is generated continuously throughout the job execution process. The JobAnalyzer analyzes these Eventlogs and aggregates them into real-time metrics. There are more than 200 running metrics in total, covering various dimensions such as application, executor, and stages. Table 2 presents key metrics.

Our two-stage configuration auto-tuning method combines rule-based and algorithm-based tuning. Both leverage real-time metrics obtained from JobAnalyzer to calculate tuned parameters for jobs. These metrics and parameters are stored for subsequent use by the Tuning API, enabling it to update parameters for future job execution. This iterative process allows for the continuous optimization of configurations for resource efficiency.

5.2 Rule-Based Tuning

Rule-based tuning can quickly improve the configuration for a multitude of production jobs. After Job Analyzer aggregates metrics for each job, we obtain a rough profile of the job, such as how many tasks it has, how much data it inputs, what the average and maximum CPU utilization is, etc. Leveraging these metrics, rule-based tuning employs heuristic rules to adjust configuration parameters for the job. In ByteDance production environments, with an increasing number of heuristic rules, we use a rule tree to describe the relationship among rules. These rules are typically categorized into CPU, memory, and shuffle groups, delineated by the parameters they target, as illustrated in Table 3.

The basic heuristic rule for CPU and memory tuning can be summarized as follows: when the average and maximum utilization are low, the corresponding resource application amount is reduced while maintaining constant job concurrency. Conversely, when the utilization is too high, the resource application amount is increased. To better handle various corner cases, we have incorporated additional metrics to provide a better understanding of the job's current running status. For example, the "executor_duty_factor" metric helps differentiate between low CPU utilization due to long idle executors or low utilization of the job thread itself.

As mentioned in Sec. 3, a cause of shuffle problems is the presence of a large number of random I/Os in ESS. By using optimized shuffle-related parameters, the number of random I/Os can be effectively reduced. Our heuristic rules observe the number of partitions in each stage of the job. If the number of partitions significantly exceeds the number of cores the job can apply for, it is considered unnecessary concurrency. By adjusting parameters such as `spark.sql.files.maxPartitionBytes` and `spark.sql.adaptive.maxNumPostShufflePartitions`, the number of partitions in the read stage and shuffle stage can be controlled, thereby avoiding the generation of excessive small shuffle files. Similarly, additional metrics are used to assess the impact of shuffle-related parameter tuning. For example, memory spill data size in stages is measured to predict whether an increase in the amount of data processed by a single task will result in more severe performance degradation.

All heuristic rules are designed in pairs, comprising both increasing and decreasing rules for each parameter. These symmetric tuning conditions and directions ensure convergence in most tuning rules. Furthermore, there is an independent effect evaluation and offline analysis outside the rules to follow up on non-convergence and negative tuning. In cases of negative tuning, the system automatically stops the related tuning rules or switches to algorithm-based tuning to minimize the impact on online production.

5.3 Algorithm-Based Tuning

To address jobs that cannot be effectively optimized through the heuristic rule-based tuning, we have developed an algorithm-based tuning method using Bayesian optimization. First, We make a formal definition of Spark configuration tuning, which can be regarded as a black-box optimization problem:

$$\hat{x} \leftarrow \arg\min_{x} f(x) \tag{4}$$

where $x \in R^n$ represents the Spark parameters for tuning, as listed in Tab. 3, and $f(x) \in R$ represents the parameter evaluation. The

Table 2: Spark Job Running Metrics

Catalogs	Key Metrics	Descriptions	Units
Application	status	Job status (SUCCEEDED/KILLED/FAILED)	-
	app_total_duration_ms	Total runtime of all tasks	ms
	app_total_tasks	Total number of tasks	-
	avg_{tasks, input}_run_time	Average runtime of {all tasks, input tasks}	min
	app_total_input_{tasks, bytes}	Total input {tasks, data size}	-
	app_total_shuffle_read_{tasks,bytes}	Total shuffle read {tasks, size}	-
Executor	executor_mem_alloc_total	Total memory allocated amount of all executors	G · h
	executor_mem_max_alloc_total	Maximum memory allocated amount of all executors in last 3 days	G
	executor_mem_usage_{avg, max}	{Average, Maximum} memory utilization of executors	-
	executor_cpu_usage_{avg ,max}	{Average, Maximum} CPU utilization of executors	-
	avg_executor_alloc_rate_{3d, 7d}	Maximum executor allocated rate in last {3, 7} days	-
	executor_duty_factor	Load rate of executors	
Stage	stage_avg_tasks_runtime_max	Maximum average runtime of all stages	ms
	stage_run_time_max_avg_input_memory_bytes_spilled	Average memory spill of tasks in the slowest stage	G
	stage_max_avg_{tasks, input}_run_time	Average runtime of the largest {task, input task} in stages	min
	stage_avg_input_task_runtime_max	Maximum runtime of the average runtime of input tasks in stages	min
	stage_total_input_tasks_max	Maximum number of tasks in the input stages	-
	stage_total_input_tasks_max_avg_runtime	Average runtime of the input stage with the largest number of tasks	min
	stage_avg_input_task_runtime_max_avg_memory_bytes_spilled	Average amount of memory spill data in the slowest input stage	G
	stage_avg_shuffle_read_task_runtime_max	Maximum runtime of the average runtime of tasks in shuffle read stages	min
	stage_shuffle_read_bytes_max	Maximum shuffle read size in shuffle read stages	G
	stage_run_time_max_total_shuffle_read_tasks	Number of tasks in the slowest stage	-

Table 3: Tuned Configuration Parameters and Value Ranges

Types	Configuration Parameters	Ranges
CPU	spark.dynamicAllocation.minExecutor	1 ~ 5
	spark.dynamicAllocation.initialExecutors	1 ~ 5
	spark.dynamicAllocation.maxExecutor	5 ~ 2000
	spark.executor.instances	1 ~ 2000
	spark.executor.cores	1 ~ 4
	spark.executor.milliCores	spark.executor.cores × (125 ~ 2000)
Memory	spark.executor.memory	4g ~ 64g
	spark.executor.memoryOverhead	1g ~ 4g
	spark.executor.memoryBurst.enabled	True/False
	spark.executor.memoryBurst.ratio	1 ~ 1.4
Shuffle	spark.sql.files.maxPartitionBytes	64M ~ 16G
	spark.sql.adaptive.maxNumPostShufflePartitions	500 ~ 100000

goal is to find the configuration $\hat{x}$ that minimizes the objective function. To pursue the improvement of utilization, we define the evaluation indicator as the reciprocal of the product of CPU and memory utilization. Additionally, in cases where job performance is exceptionally poor, resulting in issues like OOM failures or excessively long execution time, an additional substantial penalty will be imposed. Note that f is an unknown function that only allows a limited number of evaluations. As the tuning is performed online, each evaluation is actually a single execution of a periodical Spark job in production.

Based on Bayesian Optimization, we have developed an algorithm that effectively utilizes the evaluation results of historical parameters to recommend better ones. This algorithm aims to improve the efficiency and effectiveness of the tuning process by leveraging the information gained from previous evaluations. The algorithm follows these steps:

(1) Fit a surrogate model: We choose the Gaussian Process (GP) as a surrogate model to approximate unknown function f, thereby the evaluation results of unseen parameters can be inferred. Let $X \in R^{k\times n}$ be the k sets of historical parameters, $Y \in R^k$ be the historical evaluation results, and K be the covariance function, then the evaluation result of parameter x obeys the Gaussian distribution $N(\mu(x), \sigma^2(x))$, where the predictive expectation and variance are calculated as Eq. 5:

$$\begin{aligned}\mu(x) =& K(x,X)K^{-1}(X,X)Y \\ \sigma^2(x) =& K(x,x) - K(x,X)\cdot K^{-1}(X,X)\cdot K(X,x)\end{aligned} \tag{5}$$

(2) Minimize an acquisition function: The Expected Improvement (EI), as defined in Eq. 6, is usually adopted as an acquisition function in Bayesian Optimization. It measures the mathematical expectation of evaluation result improvement that the unseen parameter x can bring. The EI achieves a good trade-off between exploitation (μ) and exploration (σ^2), enabling effective comparisons among different unseen parameters.

$$\begin{aligned}EI(x) =& \max(\mu(x) - f(\hat{x}), 0) + \sigma(x)\psi(\frac{\mu(x)-f(\hat{x})}{\sigma(x)}) \\ & - |\mu(x) - f(\hat{x})|\Phi(\frac{\mu(x)-f(\hat{x})}{\sigma(x)}\end{aligned} \tag{6}$$

where ψ is the standard normal probability density function, Φ is the standard normal cumulative distribution function, and $\hat{x}$ is the best parameter in historical runs.

Considering that online tuning has a strong aversion to risks [21, 39, 43], we restrict the optimization of the acquisition function in a region with limited variance, so that the tuning trajectory can be smoothed, i.e., parameters far away from historical parameters will not be explored immediately. Therefore, the recommended parameters x^* for the next run are formulated as below:

$$\begin{aligned}x^* \leftarrow & \arg\min_x EI(x) \\ & \text{s.t. } \sigma^2(x) \le \mu^2(x)\end{aligned} \tag{7}$$

Different from traditional works using the gradient descent method [21] or quasi-Newton's method [2, 20] to optimize the acquisition function, which may fall into a local optimum, we employ the genetic algorithm to solve the Eq. 7. This approach allows for a more efficient and targeted exploration of the parameter space, improving resource utilization of Spark jobs while ensuring stability.

6 EVALUATIONS

In this section, we validate the effectiveness of these techniques using ByteDance production workloads. We will analyze and answer the following questions from the perspectives of stability, performance, and resource utilization:

- How much performance improved by shuffle services?
- How does resource control help stability and utilization?
- How many resources can be saved by configuration tuning?
- What are the advantages of these techniques in production?

6.1 Enhanced ESS and CSS Evaluations

This section evaluates the effects brought by multi-mechanism shuffle services, including evaluating the stability of Enhanced ESS through production workloads and evaluating the performance of CSS using the TPC-DS benchmark [30].

Enhanced ESS. The request throttling is evaluated through its performance in production clusters. Figure 5 shows the effect of request throttling on a job that sends massive shuffle requests to the ESS node over a period of time. When the ESS server's shuffle latency starts deteriorating due to increasing shuffle requests, request throttling takes effect for the job contributing the most shuffle requests, reducing the number of subsequent shuffle requests sent by the job. Within a few minutes, the ESS can finish serving its queued requests, and shuffle latency is back to normal shortly.

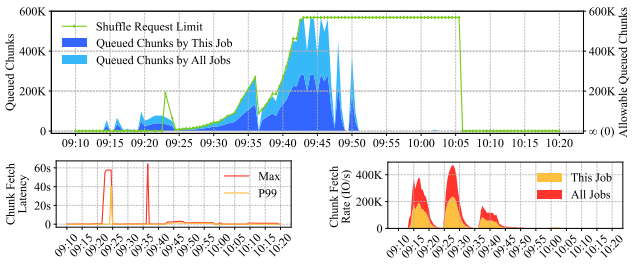

Figure 5: Example of Shuffle Request Throttling

The executor rolling is also evaluated on a product cluster. As shown in Fig. 6, we compare the disk usage before and after the executor rolling is launched. Here, "Max Disk Used" represents the maximum shuffle data size on all physical machines in a day in the cluster. "Avg Disk Used" represents the average shuffle data size. "Disk Used P50", "Disk Used P90", and "Disk Used P99" represent the 50th, 90th, and 99th percentiles of the shuffle data size, respectively. We select the disk usage data for one day in January 2023 before and one day in January 2024 after the executor rolling launch. With the growth of the business, the median disk used for shuffle on each physical machine has increased from 0.7TB to 1.2TB, the average has increased from 1.8TB to 2.6TB, and the maximum has decreased from 48T to 23T. The 99th percentile has decreased slightly. Therefore, it can be concluded that after the launch of executor rolling, the usage of disks is more evenly distributed, avoiding large amounts of shuffle data written to a few executors.

CSS. This experiment is conducted on a 40-node cluster featuring Intel Xeon Gold 6130 CPUs @ 2.10GHz, equipped with 64GB * 16 DRAM, 16 13T HDDs, and 2 * 25GB network interface cards

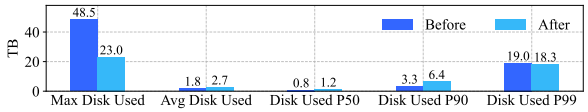

Figure 6: Executor Rolling Balances Disk Usage

to ensure network transmission efficiency. The dynamic executor allocation is enabled and the executor resource configurations are the same to ensure similar resource allocation from the cluster for all shuffle services. The CSS remote shuffle cluster is deployed on a 9-node cluster with Intel Xeon Gold 6230 CPUs @ 2.10GHz, 64GB * 16 DRAM, 12 13T HDDs, and 2 * 25GB network interface cards.

In our comparative analysis, we assess the execution time and resource utilization of three shuffle services - ESS, Magnet (Spark 3.2's push-based shuffle service [34]), and CSS - using a 1TB TPC-DS benchmark [30]. Figure 7 illustrates that CSS improves the speed by over 10% for certain SQLs compared to both ESS and Magnet, with significant performance enhancements observed in nearly 30% of the queries. Table 4 attests to this improvement, showing CSS reduces total execution time by 0.4 hours relative to ESS and 1.3 hours against Magnet. Moreover, CPU and memory allocation and usage are notably decreased with Magnet and CSS.

Both CSS and Magnet leverage a push and merge-based shuffle strategy. Queries 14a, 14b, 48, 62, and 68, characterized by large shuffle write stages followed by smaller read stages, benefit most from this approach. While both exhibit improvements, CSS outperforms Magnet by buffering reducer partitions in memory and flushing data blocks when exceeding block size thresholds. Its efficient in-memory ack mechanism mitigates push overhead and minimizes shuffle write times, thus enhancing overall performance. Besides, Magnet shows performance degradation compared to ESS due to its additional waiting time incurred during merge result reception and block merging at map task completion, which is particularly disadvantageous when dealing with smaller shuffle data sizes.

6.2 Resource Control Evaluations

This section evaluates the effects of fine-grained resource control. However, it is essential to note that the efficacy of these features hinges on the configuration parameters' appropriate set and tuned. Consequently, the optimization outcomes combine the two-stage configuration tuning method. These features provide the ability to configure fine-grained resources, and the two-stage configuration tuning method maximizes the benefits of these features.

Resource Allocation Control. Before the implementation of the `milliCores`, we enhanced the average CPU utilization of approximately 240,000 jobs to 56.31% by employing shuffle optimization and rule-based configuration tuning. On 2023-08-16, the milliCoresis was introduced in grayscale alongside corresponding parameter tuning as a new feature. We have selected a batch of production jobs from 2023-07-01 to 2024-02-24, with the total number increased from 211,816 to 360,720. Concurrently, as shown in Fig. 8, the number of enabled jobs rises from 0 to 350,108 during the same period. Remarkably, as the proportion of enabled jobs surged, the average

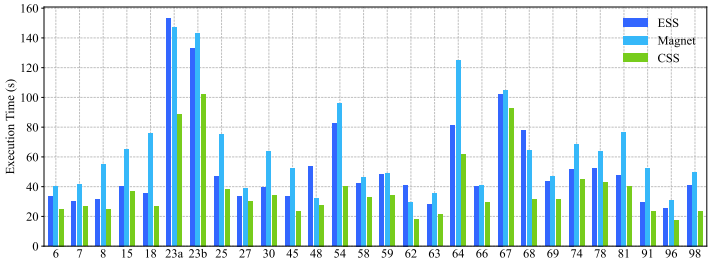

Figure 7: Execution Time Comparison of Shuffle Services in 1TB TPC-DS

Table 4: Resource Comparison

		ESS	Magnet	CSS
Total Execution Time (h)		3.69	4.59	3.29
Total Impro Ratio (%)		-	-9.90%	+10.93%
CPU Allocated (Cores)	Max	1240	1190	1080
	Avg	463	464	443
CPU Used (Cores)	Max	819	589	623
	Avg	94.4	79.8	97.6
Memory Allocated (TB)	Max	4.83	4.86	4.22
	Avg	1.82	1.82	1.74
Memory Used (TB)	Max	2.95	1.99	2.39
	Avg	0.85	0.72	0.80

CPU utilization gradually climbed to 94.8%. This highlights the critical role of the milliCores in improving resource efficiency.

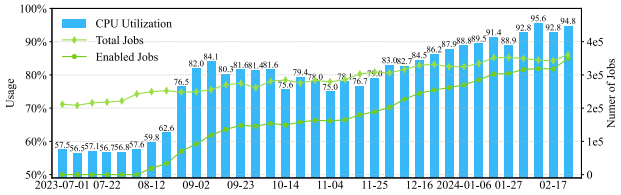

Figure 8: CPU Utilization Improvements with milliCores

As shown in Fig. 9, we quantify the reduction in memory allocation by the implementation of the memoryBrust. This feature reduces users' memory allocation by reducing the value of memory overhead while ensuring stability. We use Eq. 3, multiplied by each job's duration to calculate the benefits in memory allocation. This feature was launched in October 2023, and since then, the number of enabled jobs has progressively increased from 3,753 to 474,692. By the end of January 2024, ByteDance had achieved a daily savings of 55,130 TB · hour in memory allocation. These findings underscore the effectiveness of the `memoryBurst` feature in reducing memory demands and saving resources.

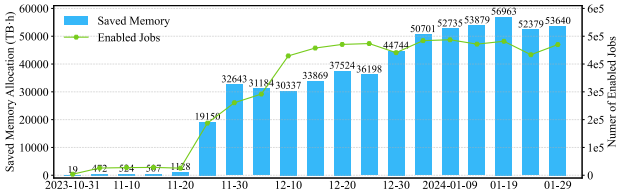

Figure 9: Memory Savings with memoryBurst

Resource Usage Control. Spill optimization is widely used in production jobs, we analyze several typical jobs for improvements brought by the shuffle write, shuffle read, and sort operators. For instance, in a data warehousing job with heavy shuffle write operations, the spill otimization results in a 55.52% reduction in allocated memory (from 3.26TB to 1.45TB) and a 56.20% decrease in actual memory usage (from 1.21TB to 0.53TB). In another case, considering a shuffle-read-intensive job, spill optimization leads to a significant decrease in the number of OOM tasks within a stage, dropping from 7482 to 27, alongside a reduction in execution time from 29 minutes to 11 minutes. Moreover, memory used can see a substantial decline of 65% (from 23.1TB to 8.16TB). Notably, the shuffle operators' spill capability evolves from passive spilling when container memory reaches full capacity to active spilling when the memory used hits a specific threshold, resulting in smoother fluctuations with increased spill frequency but consistent shuffle performance. This transition notably enhances memory resource utilization. Lastly, taking a sort job as an example, the memory allocation decreases from 330TB to 214TB, with actual memory used dropping from 300TB to 129TB, constituting a 57% reduction. The job duration remains relatively stable, changing from 2.1h to 2.2h. Additionally, disk spill occurrences shifted from being triggered solely upon memory reaching full capacity to a regular triggering mechanism once the memory threshold of 1GB is reached.

6.3 Configuration Tuning Evaluations

Both the rule-based and algorithm-based tuning methods are deployed in ByteDance production environments. In this section, we present various results associated with resource utilization by our tuning methods, where utilization is calculated by dividing the total CPU or memory used by the total CPU or memory allocated.

Rule-Based Tuning. The rule-based tuning has undergone several iterations and optimizations throughout the online process. Initially, it primarily manages job resource allocation by adjusting CPU parameters such as `spark.dynamicAllocation.maxExecutor`, `spark.vcore.boost.ratio`, `spark.executor.cores`, and memory parameters like `spark.executor.memory` and `spark.executor.memoryOverhead`. As depicted in Fig. 10, during the first period stage 2022-07 to 2023-08 CPU utilization has risen from 51.4% before 2023-03 to 59.4% after 2023-03 on average. Similarly, the memory utilization rate has increased from 43.2% to 46.3%.

Subsequently, with the implements of the milliCores (refer to Fig. 8) and the memoryBrust (refer to Fig. 9) features, the rule-based tuning maximizes the benefits of these additions by adjusting corresponding parameters like `spark.executor.milliCores`, `spark.executor. memoryBurst.enabled`, and `spark.executor.memoryBurst.ratio`. It is evident that after 2023-08 and 2023-10, there has been a notable improvement in CPU utilization during the second period and memory utilization during the third period, correlating with the increase in tuned jobs. With the refinement of the rules, the CPU utilization of all tuned jobs reached 90.0%, and memory utilization reached 52.7%, encompassing nearly one-third of production jobs and yielding significant improvements.

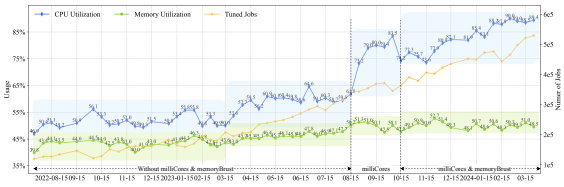

Figure 10: Rule-Based Tuning

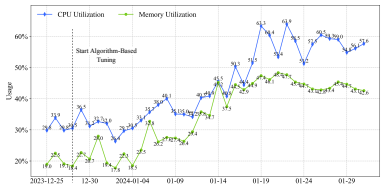

Figure 11: Algorithm-Based Tuning

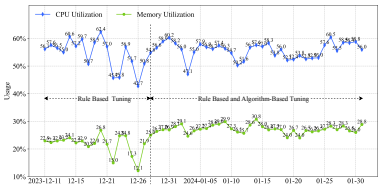

Figure 12: Two-Stage Tuning

Algorithm-Based Tuning. The algorithm-based tuning serves as a supplement to rule-based tuning in production environments. For some online Spark jobs, rule-based tuning may not occur due to stability considerations or the rules may not cover the current state of the job. As a result, the job may not be tuned or the tuning effect may not be significant. In such cases, these jobs are sent to the algorithm-based tuning to further improve resource utilization. We set the maximum iteration as 25 times for algorithm-based tuning of online jobs in production.

Figure 11 shows the online performance of jobs tuned by algorithm-based tuning. There are a total of approximately 3000 jobs, and algorithm-based tuning has taken over these jobs from 2023-12-28. Before the algorithm's intervention, due to the limitations of rule-based tuning, the utilization of this batch of jobs was not high, with average CPU and memory utilization hovering around 31% and 21%, respectively. After the algorithm's takeover, the CPU and memory utilization of these jobs gradually improved and eventually stabilized at 58% and 45% respectively.

A Two-Stage Tuning Case. Figure 12 shows the utilization changes of all jobs in a project, with approximately 5% of jobs being taken over by algorithm-based tuning after 2023-12-28. Therefore, the utilization of the first half of the curve represents the result of the project using only rule-based tuning, while the latter half represents the combined results of rule-based tuning for some jobs and algorithm-based tuning for a few jobs. It can be observed that there is little change in CPU utilization between using rules only and employing a two-stage combination. This is because the CPU utilization of this batch of jobs is already relatively high when using rules. However, in terms of memory utilization, the former is around 21% while the latter is around 26%, showing a significant improvement. This is because the proportion of memory usage is relatively high for the 5% of jobs, and after further optimization with the algorithm, the memory utilization of the project significantly improved.

Algorithm-based tuning can further improve the results obtained from rule-based tuning. However, due to its time-consuming nature and slower tuning speed, algorithm-based tuning needs to be used with rule-based tuning to achieve better results in production.

6.4 Overall Tuning Performance

These techniques have undergone extensive iteration, optimization, and deployment. In this section, we will use data over two years to conduct a statistical analysis of these techniques in enhancing the resource efficiency of large-scale Spark workloads at ByteDance.

Figure 13 illustrates the improvement in resource efficiency across all Spark jobs spanning from 2022 to 2023. Regarding CPU utilization, iterative optimizations of rule-based configuration tuning have led to an increase from 48.5% in 2022 to 60% in 2023,

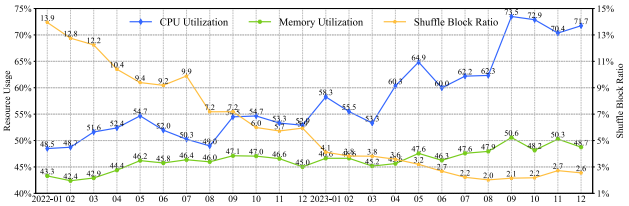

Figure 13: Overall Resource Efficiency Improvements

subsequently increased to over 70% with the introduction of the milliCoresfeature. Similarly, memory utilization has increased from 43.3% in 2022 to 46% through rule-based configuration tuning, further bolstered to nearly 50% with the `memoryBrust` feature. The continuous iteration of rule-based configuration tuning and the launch of algorithm-based configuration tuning, as well as the expansion of the enabled job scope, are still ongoing. The shuffle block ratio, representing the proportion of time waiting for shuffle, initially stood at approximately 14% in 2022-01, reduced to about 4%-6% through enhancements in shuffle speed and stability via Enhanced ESS and CSS, and drove down to 2% by subsequent parameter tuning.

Table 5: Resources Saved by Optimized Jobs

Month	# of Users	# of Jobs	# of Jobs with CPU Util. ≥ 60%	# of Job with MEM Util. ≥ 50%	Saved CPU Alloc. (core · day)	Saved MEM Alloc. (PB · day)	Avg. Job Exec. Time (min)	Avg. Job Exec. Time ↓ (min)
2023-03	9657	255639	151218	103615	695588	14.7	28.9	19.1
2023-04	9856	226733	163538	96838	597411	12.0	27.8	17.1
2023-05	10148	292357	180306	122761	718819	13.7	26.9	18.3
2023-06	10689	341912	182876	106604	701394	9.2	28.3	15.0
2023-07	11031	363587	195663	102353	557278	8.1	28.2	13.9
2023-08	11734	441064	275763	123287	597583	9.5	28.9	11.0
2023-09	12548	523290	294565	125514	938821	7.4	28.8	10.5
2023-10	12765	478943	258175	112404	865428	9.6	27.1	11.5
2023-11	13092	489371	261811	131240	811081	10.3	30.6	6.4
2023-12	13478	502325	280444	143940	897858	7.2	29.2	6.9
2024-01	13771	545687	292727	151612	930637	5.7	27.3	8.3
2024-02	14004	533445	308475	143541	1049231	4.6	24.7	11.1

= "The Number", Util. = "Utilization", MEM = "Memory", Alloc. = "Allocation", Avg. = "Average", Exec. = "Execution", ↓ = "Reduction"

As shown in Tab. 5, we calculate the average number of optimized jobs and resource allocation savings per day for each month from 2023-03 to 2024-02. Notably, our techniques have yielded substantial effects, with the number of service users exceeding 9,657 to 14,004 and optimized jobs surging from 255,639 to 533,445. Correspondingly, the daily average of jobs with CPU utilization exceeding

60% increased from 151,218 in 2023-03 to more than 300,000 in 2024-02, while jobs with memory utilization surpassing 50% have also reached around 150,000. The daily savings in CPU and memory resource allocation have peaked at 1,049,231 core · day and 4.6 PB · day respectively. Besides, the average job execution time has also dropped significantly, reaching an 11.1-minute reduction in 2024-02, accounting for 31% of the average job execution time before. Meanwhile, it can also be seen that the resource benefits fluctuate in different months.

7 LESSONS LEARNED

We briefly summarize the experience and lessons learned from deploying the resource efficiency governance framework in production since 2022. Below is a representative list.

Choosing appropriate shuffle services. In our production environments, we extensively deploy Enhanced ESS in dedicated clusters and CSS in mixed clusters. As the disk resources in dedicated clusters are SSDs, disk I/O is not a bottleneck for shuffle. In this context, we have observed that leveraging Enhanced ESS effectively meets the requirements of our users. Therefore, it is not necessary to switch all jobs to a remote shuffle service.

Decoupling CPU and memory optimization. Initially, to enhance CPU utilization, we pursued a strategy involving utilizing a single core to handle more tasks and capitalize on idle CPU resources. However, this strategy caused a higher probability of OOM failures in some production clusters, due to concurrent tasks requiring increased memory allocation. Consequently, we have introduced milliCores to optimize CPU utilization while still maintaining task parallelism. This allows the configuration tuning of CPU and memory parameters more independently.

Minimizing user-visible parameters. We have discussed using the request and limit parameters to enhance resource utilization while ensuring execution speed. However, we ultimately decided not to expose CPU and memory limit parameters to users. If users are allowed to freely set the limit, the actual amount of resources consumed could become uncontrollable. Additionally, when a large number of such jobs run simultaneously, it could lead to excessive machine utilization and severely degrading job performance.

8 RELATED WORK

Resource efficiency involves the coordinated work of many components. Thus, we discuss the related work for the following aspects.

Shuffle Mechanism. In large-scale spark workloads, fast I/O support is one of the basic necessities [18]. Many solutions [25, 34] about shuffle mechanisms focus on optimizing disk I/O and pipeline. Sailfish [32] is centered around aggregating intermediate data to improve performance by batching disk I/O. Riffle [42] merges fragmented intermediate shuffle files into larger block files, converting small, random disk I/O requests into large, sequential ones. iShuffle [15] presents a shuffle-on-write operation to push map output data to nodes proactively. Magnet [34] and Apache Celeborn [9] are remote push and merge-based shuffle services. Our enhancements alleviate the I/O bottleneck and improve resource efficiency by optimizing ESS, CSS, and tuning relevant parameters.

Resource Optimization. Maroulis et al. [26] orchestrate the execution order of Spark applications and tune the computing nodes' CPU frequencies to minimize energy consumption. Islam et al. [17] use reinforcement learning to place the executors of jobs while leveraging VM instance prices. AutoExecutor [33] predicts query run times to determine the optimal number of executors, guiding the right-sizing of resources. SimCost [5, 6] introduces resource time representing both the utilization and cost to assist in making cost-effective decisions on resource configuration. Asemen et al. [1] propose MILP and LP models to predict runtime in different machine types with a cost-aware resource recommendation algorithm to select the best configuration. The techniques we propose delve into the Spark engine with finer granularity, enabling control over resource allocation and usage. Concurrently, we integrate parameter tuning methods to optimize resource parameters, larger than the types or range optimized by [1, 5, 7].

Configuration Tuning. The goal of most research is performance optimization, that is, reducing the execution time of spark jobs. Following the survey [16], they can be categorized into six types including rule-based approach [12], cost modeling approach [4, 13, 22, 36],simulated-based approach [31], experiment-driven approach [29, 44], machine learning approach [7, 23, 27] and adaptive approach [24, 41]. Some methods consider multi-objectives, such as using cloud-level parameters to control server costs [7, 37], tuning MLlib parameters to optimize memory consumption and accuracy [28], and Bayesian optimization (BO) based solutions [21, 35] for multiple targets. However, these methods typically do not enable Spark's dynamic allocation [11] and are only suitable for a fixed number of executors in smaller-scale clusters. Even Li et al. [21] have production job tuning evaluations that have not turned on this feature. We enable this unignorable feature coupled with our unique parameters, making tuning more effective.

9 CONCLUSIONS

In this paper, we propose a series of techniques aimed at enhancing resource efficiency for Spark workloads, particularly focusing on stability, performance, and utilization in large-scale production environments. Throughout the complete lifecycle of Spark jobs, we present multi-mechanism shuffle services adapted to complex production environments, employ fine-grained resource control to minimize resource allocation and usage and utilize a quick and stable two-stage configuration auto-tuning method. Through various experiments and production metrics, our proposed techniques demonstrate substantial resource savings with millions of CPU cores and petabytes of memory daily. In the future, we plan to further refine and expand these technologies, such as improving HDFS performance, incorporating more tuning algorithms, etc.

ACKNOWLEDGMENTS

We would like to thank all those who have contributed to the design and development of our resource efficiency governance framework including Yadong Zhang, Xin Gao, Xuewei Lin, Lei Liu, Chang Liu, Zilong Zhou, Jiahuan He, Mourang Han, Cheng Peng, Fan Zeng, Min Zhang, Bo Li, Xiaofei Li, Huixiang Wang, Wei Wang, Shaofei Yang, and Dongyang Wang. This work was supported in part by National Natural Science Foundation of China [U23A20309] and ByteDance Research Project [CT20211123001686].

REFERENCES

[1] Mohammad-Mohsen Aseman-Manzar, Soroush Karimian-Aliabadi, Reza Entezari-Maleki, Bernhard Egger, and Ali Movaghar. 2022. Cost-aware Resource Recommendation for DAG-based Big Data Workflows: An Apache Spark Case Study. *IEEE Transactions on Services Computing (TSC)* 16, 3 (2022), 1726–1737.

[2] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 21524–21538.

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Communications of the ACM (CACM)* 59, 5 (2016), 50–57.

[4] Yuxing Chen, Peter Goetsch, Mohammad Ashraful Hoque, Jiaheng Lu, and Sasu Tarkoma. 2022. d-Simplexed: Adaptive Delaunay Triangulation for Performance Modeling and Prediction on Big Data Analytics. *IEEE Trans. Big Data* 8, 2 (2022), 458–469.

[5] Yuxing Chen, Mohammad A Hoque, Pengfei Xu, Jiaheng Lu, and Sasu Tarkoma. 2024. SimCost: Cost-Effective Resource Provision Prediction and Recommendation for Spark Workloads. *Distributed and Parallel Databases* 42, 1 (2024), 73–102.

[6] Yuxing Chen, Jiaheng Lu, Chen Chen, Mohammad Hoque, and Sasu Tarkoma. 2019. Cost-Effective Resource Provisioning for Spark Workloads. In *ACM International Conference on Information and Knowledge Management (CIKM)*. ACM, 2477–2480.

[7] Guoli Cheng, Shi Ying, and Bingming Wang. 2021. Tuning Configuration of Apache Spark on Public Clouds by Combining Multi-Objective Optimization and Performance Prediction Model. *Journal of Systems and Software* 180 (2021), 111028.

[8] Apache Software Foundation. 2022. SPARK-27495. https://issues.apache.org/jira/browse/SPARK-27495 SPIP: Support stage level resource configuration and scheduling.

[9] Apache Software Foundation. 2024. Apache Celeborn. https://celeborn.apache.org/ Celeborn is an intermediate data service for big data compute engines.

[10] Apache Software Foundation. 2024. Apache Spark. http://spark.apache.org Spark is a unified engine for large-scale data analytics.

[11] Apache Software Foundation. 2024. Job Scheduling Spark 351 Documentation. https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload.

[12] Apache Software Foundation. 2024. Tuning Spark 351 Documentation. https://spark.apache.org/docs/latest/tuning.html A short guide for tuning Spark.

[13] Enrico Gallinucci and Matteo Golfarelli. 2019. SparkTune: Tuning Spark SQL through Query Cost Modeling. In *International Conference on Extending Database Technology (EDBT)*. OpenProceedings, 546–549.

[14] Rong Gu, Xu Huang, Haipeng Dai, Xiaoyu Geng, Xiaofei Chen, Yihua Huang, Fu Xiao, and Guihai Chen. 2022. Efficient. Scalable and Robust Data Shuffle Service for Distributed MapReduce Computing on Cloud. In *HPCC/DSS/SmartCity/DependSys*. IEEE, 337–346.

[15] Yanfei Guo, Jia Rao, Dazhao Cheng, and Xiaobo Zhou. 2016. iShuffle: Improving Hadoop Performance with Shuffle-On-Write. *IEEE Transactions on Parallel and Distributed Systems(TPDS)* 28, 6 (2016), 1649–1662.

[16] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–37.

[17] Muhammed Tawfiqul Islam, Shanika Karunasekera, and Rajkumar Buyya. 2021. Performance and Cost-Efficient Spark Job Scheduling Based on Deep Reinforcement Learning in Cloud Computing Environments. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 7 (2021), 1695–1710.

[18] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K DK Panda. 2016. Efficient Data Access Strategies for Hadoop and Spark on HPC Cluster with Heterogeneous Storage. In *IEEE International Conference on Big Data (Big Data)*. IEEE, 223–232.

[19] Md Muhib Khan and Weikuan Yu. 2021. Robotune: High-Dimensional Configuration Tuning for Cluster-Based Data Analytics. In *International Conference on Parallel Processing (ICPP)*. ACM, 1–10.

[20] Shibo Li, Robert Kirby, and Shandian Zhe. 2021. Batch Multi-Fidelity Bayesian Optimization with Deep Auto-Regressive Networks. *Advances in Neural Information Processing Systems (NeurIPS)* 34 (2021), 25463–25475.

[21] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. 2023. Towards General and Efficient Online Tuning for Spark. *Proceedings of the VLDB Endowment (VLDB)* 16, 12 (2023), 3570–3583.

[22] Yuhao Li and Benjamin C Lee. 2022. Phronesis: Efficient Performance Modeling for High-dimensional Configuration Tuning. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–26.

[23] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. 2022. A Resource-Aware Deep Cost Model for Big Data Query Processing. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 885–897.

[24] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. Adaptive Code Learning for Spark Configuration Tuning. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1995–2007.

[25] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. 2023. Exoshuffle: An Extensible Shuffle Architecture. In *ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 564–577.

[26] Stathis Maroulis, Nikos Zacheilas, and Vana Kalogeraki. 2017. A Framework for Efficient Energy Scheduling of Spark Workloads. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2614–2615.

[27] Dimitra Nikitopoulou, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. 2021. Performance Analysis and Auto-Tuning for Spark In-Memory Analytics. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 76–81.

[28] M Maruf Öztürk. 2024. Tuning parameters of Apache Spark with Gauss–Pareto-based multi-objective optimization. *Knowledge and Information Systems* 66, 2 (2024), 1065–1090.

[29] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. 2016. Spark Parameter Tuning via Trial-and-Error. In *Advances in Big Data*. 226–237.

[30] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *The International Conference on Very Large Data Bases*. ACM, 1138–1149.

[31] David Buchaca Prats, Felipe Albuquerque Portella, Carlos HA Costa, and Josep Lluis Berral. 2020. You Only Run Once: Spark Auto-Tuning From a Single Run. *IEEE Transactions on Network and Service Management (TNSM)* 17, 4 (2020), 2039–2051.

[32] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: A Framework for Large Scale Data Processing. In *ACM Symposium on Cloud Computing (SOCC)*. ACM, 1–14.

[33] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. 2021. AutoExecutor: Predictive Parallelism for Spark SQL Queries. *Proceedings of the VLDB Endowment (VLDB)* 14, 12 (2021), 2855–2858.

[34] Min Shen, Ye Zhou, and Chandni Singh. 2020. Magnet: Push-based Shuffle Service for Large-scale Data Processing. *Proceedings of the VLDB Endowment (VLDB)* 13, 12 (2020), 3382–3395.

[35] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An Online Spark SQL Tuning Service via Generalized Transfer Learning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 4800–4812.

[36] Rekha Singhal and Praveen Singh. 2017. Performance Assurance Model for Applications on SPARK Platform. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 131–146.

[37] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant J. Shenoy. 2021. Spark-based Cloud Data Analytics using Multi-Objective Optimization. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 396–407.

[38] Yixin Song, Junyang Yu, JinJiang Wang, and Xin He. 2023. Memory Management Optimization Strategy in Spark Framework based on Less Contention. *The Journal of Supercomputing* 79, 2 (2023), 1504–1525.

[39] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proceedings of the VLDB Endowment (VLDB)* 12, 10 (2019), 1221–1234.

[40] Wu Xiang, Yakun Li, Yuquan Ren, Fan Jiang, Chaohui Xin, Varun Gupta, Chao Xiang, Xinyi Song, Meng Liu, Bing Li, Kaiyang Shao, Chen Xu, Wei Shao, Yuqi Fu, Wilson Wang, Cong Xu, Wei Xu, Caixue Lin, Rui Shi, and Yuming Liang. 2023. Gödel: Unified Large-Scale Resource Management and Scheduling at ByteDance. In *ACM Symposium on Cloud Computing (SOCC)*. ACM, 308–323.

[41] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 674–684.

[42] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. 2018. Riffle: Optimized Shuffle Service for Large-Scale Data Analytics. In *European Conference on Computer Systems (EuroSys)*. ACM, 1–15.

[43] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2102–2114.

[44] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *ACM Symposium on Cloud Computing (SOCC)*. ACM, 338–350.

[45] Chen Zou, Hui Zhang, Andrew A Chien, and Yang-Seok Ki. 2021. PSACS: Highly-Parallel Shuffle Accelerator on Computational Storage. In *IEEE International Conference on Computer Design (ICCD)*. IEEE, 480–487.