Dynamic-awareness-based Spark Execution Plan Selection and Shuffle Optimization Strategies

Congyang Wang, Junyang Yu, Han Li*, Yanhao Zhang, Dan Wang, Qingsong Xu and Haifeng Fei

***Abstract*—When Spark processes RDD data, it employs the default execution mode of distributing data to cluster nodes for processing and then aggregating results back to the Driver. This model entails bidirectional data transmission between cluster nodes and the Driver, resulting in two shuffle processes. This approach not only prolongs task execution time but also intensifies cluster resource competition and impairs system performance. To address this problem, this paper proposes a Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy (D2CS). This strategy integrates the Driver side into cluster computing resources and dynamically selects the optimal execution plan by sensing and analyzing data scale and cluster configuration, effectively avoiding resource contention and shortening task completion time. The strategy comprises three key components: first, extending the Spark SQL framework to enable Driver-side operator computation; second, designing a task execution time prediction algorithm to estimate operator execution times on both the Driver side and cluster nodes; third, determining the optimal execution plan through a time-cost model to ensure efficient cluster resource utilization. Additionally, to address the frequent disk spills caused by the serial aggregation-sorting process in Shuffle Read, this paper proposes a Parallel Shuffle Aggregation and Sorting Strategy (PASA), which shortens the execution pipeline and enhances system resource utilization by parallelizing these operations. Experimental results show that D2CS effectively alleviates cluster resource competition and significantly boosts task execution efficiency. Compared with Spark's default mechanism, PASA achieves performance improvements of 7.42%–39.88%, and compared with the Cherry and DBPM strategies, it achieves improvements of 2.95%–26%.**

***Index Terms*—Parallel Computing, Resource Competition, Shuffle, Execution Plan Selection.**

I. Introduction

With the rapid development of Internet technology, the demand for massive real-time data processing continues to grow. The increasing data scale and computational task complexity have imposed higher requirements on distributed computing frameworks. The early MapReduce [1] framework solved the problem of large-scale data processing, but its intermediate results were stored on disk, limited by disk I/O performance bottlenecks, which hindered further improvements in job execution efficiency. As a distributed framework based on in-memory computing, Spark demonstrates significant advantages in iterative scenarios like data mining and machine learning by reducing frequent external storage reads and writes, achieving higher execution efficiency than the traditional Hadoop framework. Current mainstream distributed frameworks (e.g., Storm [2], Spark [3], Flink [4]) support SQL operations for unified data analysis and management. For example, HiveSQL [5] can adapt to the underlying DAG execution models of MapReduce or Spark, while frameworks like Spark SQL and Flink SQL natively provide SQL interfaces. As shown in Figure 1, SQL operations are ultimately converted into parallel task execution supported by these frameworks. Therefore, optimizing the SQL execution plan is crucial for enhancing system performance.

For Spark SQL query optimization, various strategies have been proposed. [6] focuses on adaptive query optimization at runtime; [7] studies the dynamic selection mechanism of the JVM in SQL queries; [8-10] address data recomputation and skew optimization during SQL runtime; [11-12] integrate AI into the computing framework to enable online tuning of SQL runtime configurations. However, most of these optimization mechanisms overly rely on historical data or expert systems, leading to additional computational overhead and heavier burdens on cluster operations. To address this issue, this paper presents a lightweight optimization strategy that dynamically allocates computation tasks to the Driver side, reducing resource consumption in intermediate steps to shorten task completion time.

In distributed computing, the reasonable allocation of resources such as CPU, network, and memory is critical. Excessive resource allocation may lead to CPU idle time, which impacts the scheduling of other jobs in the cluster [11]. To address this, this paper proposes offloading some tasks to the Driver side for preprocessing, avoiding the overhead of data distribution and result aggregation in the shuffle process, as well as performance degradation caused by disk spills. Specifically, for routine data tasks, directly submitting them to the cluster for processing may incur network transmission and task scheduling overheads that exceed the local computation overhead on the Driver side, thereby prolonging task completion time. However, given the significant variations in data scale and cluster configuration parameters (e.g., number of Executors, node computational capacity, memory allocation), traditional fixed execution models struggle to balance efficiency and resource utilization. Therefore, this paper introduces a Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy (D2CS). By constructing a

Manuscript received November 11, 2024; revised May 10, 2025. This work is supported by the National Natural Science Foundation of China (No. 92367302 and No. 92467103), the subproject "New Industrial Internet Service Security System" under the NSFC Major Research Program (No. 92367302), and the Henan Provincial Science and Technology Key Project (No. 242102210046).

Congyang Wang is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: wangcongyang@henu.edu.cn).

Junyang Yu is a professor of the School of Software, Henan University, Kaifeng 475004, China (e-mail: jyyu@henu.edu.cn).

Han Li is an associate professor of School of Software, Henan University, Kaifeng 475004, China (corresponding author to provide phone: +86-13837853817; e-mail: lihan@henu.edu.cn).

Yanhao Zhang is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: zhangyanhao@henu.edu.cn).

Dan Wang is a graduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: wdan00@163.com).

Qingsong Xu is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: 2670993885@qq.com).

Haifeng Fei is a postgraduate student of School of Software, Henan University, Kaifeng 475004, China (e-mail: fhf@henu.edu.cn).

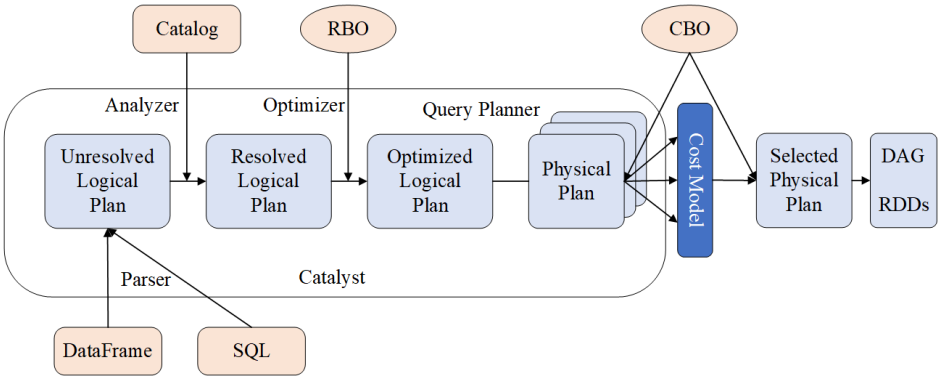

Fig. 1. Spark SQL Running Principles.

task execution time prediction model and a time-cost decision model, D2CS dynamically selects the execution path with minimal computational overhead, reducing the overheads of data distribution, intermediate file generation, and result aggregation during the shuffle process.

In addition, for the Shuffle Read phase in Spark's Sort Shuffle mechanism, the existing process employs a serial "aggregate-then-sort" mode. Insufficient memory allocation triggers multiple disk spills and frequent I/O operations, severely degrading performance. To address this, this paper proposes a Parallel Shuffle Aggregation and Sorting Strategy (PASA). By designing a parallel execution pipeline for sorting and aggregation, PASA optimizes the data processing workflow, reduces the risk of memory overflow, and enhances system resource utilization.

The contributions of this paper are summarized as follows:

(1) To address the resource overhead incurred by intermediate steps during operator execution in the cluster, this paper proposes a Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy (D2CS). Using the Sort operator as a case study, we illustrate its implementation, which comprises three key technical components: first, designing an execution plan within the Spark SQL framework to enable Driver-side computation and expand localized processing paths for operators; second, constructing a task execution time prediction model that estimates operator execution times on both the Driver side and cluster nodes, leveraging historical task data and cluster configuration parameters; finally, dynamically selecting the optimal execution path through a time-cost decision model. This approach effectively reduces cluster resource contention, significantly shortens job completion time, and eliminates the redundant two-way shuffle overhead inherent in traditional models—especially for small-scale data scenarios.

(2) To address frequent disk spills caused by the serial "aggregate-then-sort" process in the Shuffle Read phase of Spark's Sort Shuffle mechanism under large-scale data scenarios, this paper proposes a Parallel Shuffle Aggregation and Sorting Strategy (PASA). By designing a parallel execution pipeline for aggregation and sorting, the strategy employs ordered data structures to achieve incremental ordered aggregation, thereby maximizing memory utilization and minimizing disk I/O overhead. Experimental results demonstrate that PASA significantly reduces disk I/O frequency and system energy consumption, achieving task performance improvements of 7.42%–39.88% compared with Spark's default mechanism.

The remainder of this paper is structured as follows: Section II introduces related research work. Section III details the system architecture and model design. Section IV presents experimental results and performance analysis. Section V concludes the paper.

II. Related Work

A. Spark SQL-Based Optimization Techniques

With the proliferation of large-scale datasets, traditional SQL query processing techniques face performance bottlenecks [13]. [14] reveals that in the early single-threaded Volcano model, the overhead from frequent invocations of the next() method is substantial. Vectorized execution techniques mitigate such overhead by processing data in batches, significantly enhancing CPU cache utilization.

The advancement of parallel computing models has driven SQL execution optimization. The concurrent execution of the SQL execution tree primarily involves intra-operator and inter-operator parallelism: the former enables a single operator to process multiple data partitions through data partitioning, a common approach in stream-processing systems; the latter utilizes Pipeline technology to fuse successive operators into execution units via Operator Fusion, reducing function calls and accelerating query execution. In the hybrid distributed-local parallelism scheme [15], broadcast replaces traditional shuffle for data transfer in Exchange operators with small output result sets, effectively minimizing communication overhead.

In existing research, [11] proposes an approach for the automatic online adjustment of Spark SQL configurations; [16] applies the successive halving algorithm to Spark SQL configuration optimization, demonstrating excellent performance in machine-learning hyperparameter tuning; [17] dynamically loads data partitions using RDF data statistics and user query loads to optimize Spark SQL processing. These studies primarily enhance performance through auxiliary mechanisms or "space-for-time" strategies but generally overlook the rational use of Driver-side computing resources. In the traditional model, operators distribute data to cluster nodes for processing and then aggregate results back to the Driver, creating bi-directional data transmission. This often leads to cluster communication and scheduling overhead exceeding the direct computation costs on the Driver side in small-data scenarios, exacerbating cluster resource contention and affecting the scheduling of other jobs.

Different from existing work, this paper integrates the Driver side into the distributed computing architecture and proposes a Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy (D2CS) by extending the Spark physical execution plan. D2CS dynamically selects execution plans by sensing data scale and cluster configuration, aiming to minimize task completion time. This approach achieves efficient computing resource allocation and overcomes the inefficiencies of traditional fixed execution modes.

B. Spark Shuffle Read Optimization Techniques

In Spark's Sort-based shuffle mechanism, the serial execution of aggregation followed by sorting creates data processing bottlenecks: when the data scale exceeds memory capacity, partial data must be spilled to disk to free space, with frequent disk I/O significantly degrading cluster performance. To address this, researchers have proposed cache-aware algorithms [18-21] and efficient aggregation algorithms [22-24]. These approaches typically split datasets into cache-sized chunks, sort, aggregate, and cache each chunk before merging results to reduce memory pressure.

[25] introduces the Spark vectorized execution engine (VEE), which reduces random memory access frequency through batch data reordering in memory to improve query performance. [26] filters invalid data early to reduce shuffle overhead in Join operations but applies only to specific scenarios. [27] employs a hardware offloading approach to migrate tasks to dedicated hardware, yet this incurs high costs. [28] accelerates processing by restructuring shuffle control and data planes, though it introduces a new management architecture and relies on specific hardware configurations. [29] proposes a dynamic partition adjustment strategy to optimize shuffle efficiency but adds extra decision-making overhead. [30] designs a Reduce-phase Push optimization strategy for small shuffle blocks, though its generalization remains limited.

While the above methods enhance shuffle performance, they often suffer from hardware dependencies, increased system complexity, or narrow applicability. This paper proposes a Parallel Shuffle Aggregation and Sorting Strategy (PASA), which enables the parallel execution of aggregation and sorting through pipeline restructuring. This eliminates multiple disk spills during serial processing, reduces data overflow instances via efficient memory management, and shortens I/O time. PASA offers a lightweight solution for large-scale data processing, improving system resource utilization without additional hardware costs.

III. System Architecture and Strategy Implementation

During the execution of Spark operators, data is typically distributed from the Driver side to cluster nodes for processing, with results aggregated back to the Driver. In this model, if network transmission and cluster scheduling overheads exceed the direct computation costs on the Driver side, resource waste and performance degradation occur. To address this, we propose a Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy (D2CS). D2CS constructs a time-cost decision model to dynamically select the optimal execution path through real-time evaluation of data scale, cluster resource configurations (e.g., number of Executors, node computational capacity, network bandwidth), and task characteristics. Additionally, to resolve the pipeline bottleneck caused by the serial "aggregate-then-sort" process in the Shuffle Read phase, we introduce a Parallel Shuffle Aggregation and Sorting Strategy (PASA). PASA restructures the processing pipeline by enabling parallel execution of aggregation and sorting, eliminating multiple disk spills and reducing I/O overhead through efficient memory management. The overall architecture of the strategy is shown in Figure 2, achieving fine-grained computing resource allocation and deep performance tuning of the shuffle process through the collaborative optimization of D2CS and PASA.

A. Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy

This subsection designs the Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy (D2CS) based on a quantitative analysis of data transmission and computation costs. By modeling the relationship between data scale, cluster resource parameters, and task execution time, we construct a time-cost decision model to dynamically select optimal execution plans. This approach minimizes job completion time while reducing cluster resource contention, reserving hardware resources for multi-task scheduling.

When a user submits an application to a Spark cluster, the client first creates a Driver process and registers it with the master node. SparkContext requests resources via a cluster manager (e.g., Mesos, Kubernetes, or Standalone) to launch multiple Executors as distributed computing units. Spark abstracts RDD transformation relationships into a directed acyclic graph (DAG), which is divided into ShuffleMapStage and ResultStage based on narrow and wide dependencies. TaskScheduler generates TaskSet according to stage dependencies and distributes them to the thread pools of Executor nodes for execution, completing the full scheduling process for distributed computing.

To implement D2CS, this paper extends three core components within Spark's native execution framework to ensure compatibility with the existing architecture:

(1) Driver-side Execution Plan: Design a physical execution plan for driver-side computation, omitting shuffle data distribution and result aggregation for lightweight operators to complete computations directly on the Driver side. This avoids network transmission and scheduling overheads between cluster nodes. The module complements the native distributed execution plan by extending Spark's SparkPlan abstract class and adding localized execution nodes (e.g., DriverPlan).

(2) Cluster Status Collection Module: Deployed on the Driver side to collect real-time cluster task submission information, including configuration parameters such as the number of Executors, node computational capacity (represented by VCPU cores), memory allocation, network bandwidth, and historical task execution time data.

(3) Time-Cost Decision Engine: Based on collected cluster status and task characteristics, a dual-end execution time prediction model is constructed to estimate operator processing

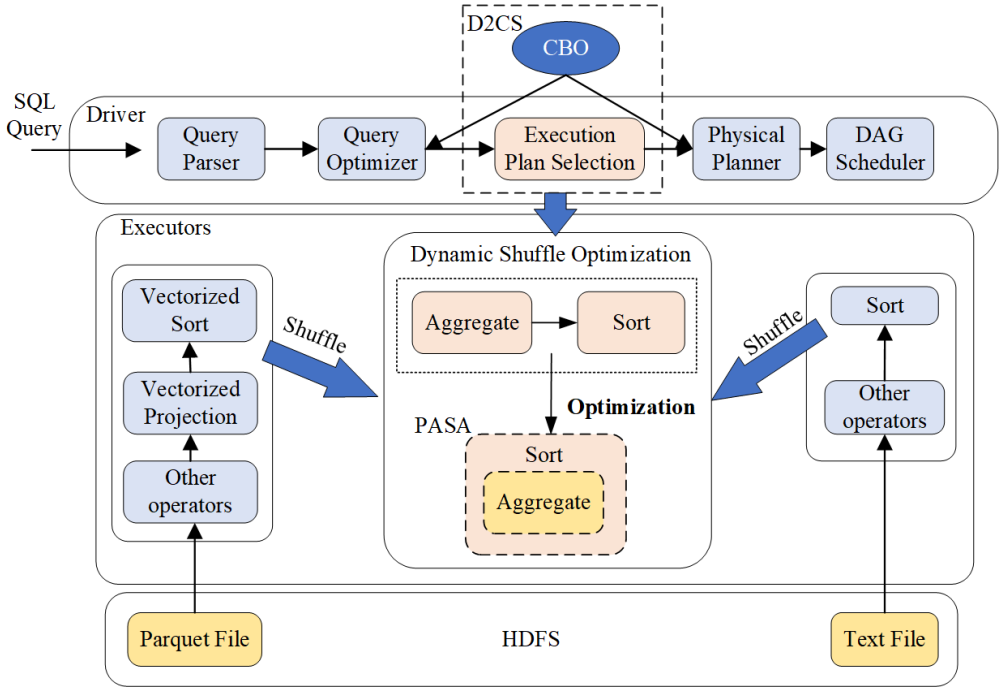

Fig. 2. System Architecture.

times on the Driver side and cluster nodes, including data transmission, computation, and shuffle overhead. A time-cost comparison model dynamically selects the execution plan with the lowest time cost.

D2CS employs a plug-in architecture: the status collection and decision engine modules are embedded directly in the Driver side without altering cluster node logic. The Driver-side execution plan is implemented by extending Spark SQL's physical execution plan generator, requiring only minimal modifications to Spark's core code to ensure system compatibility and maintainability.

B. Algorithm Implementation of Data-Scale and Configuration-Aware Dynamic Execution Plan Selection Strategy

The D2CS strategy first quantifies the input data scale and integrates real-time collected cluster resource configurations (e.g., number of Executors, node computational capacity). Using a Long Short-Term Memory (LSTM) network, it constructs a time-cost prediction model to separately model the localized computation time on the Driver side and the distributed processing time on cluster nodes. The strategy dynamically selects the execution path with the minimum total execution cost: if the cluster processing time cost exceeds that of Driver-side execution, a localized physical plan is triggered; otherwise, the traditional distributed execution plan is adopted.

1) Task Information Statistics Algorithm

Task data size is a core parameter for estimating data transmission and computation times. During program execution, the number of data records is obtained via Spark's DataFrame interface, and the physical execution plan is parsed using the Spark REST API to extract input/output data characteristics of operators in each stage. The detailed implementation is outlined in Algorithm 1.

As a key variable for estimating Driver-side data loading time, localized computation time, and cluster task completion time, the quantification of data scale directly impacts the decision accuracy of the D2CS strategy. Cluster task execution times are influenced by nonlinear factors such as data skew, resource contention, and network latency—complex dependencies that traditional static models struggle to capture accurately. D2CS addresses this challenge by training an LSTM model with historical task data extracted via Algorithm 1, enabling precise prediction of both cluster node processing times and Driver-side computation times. Additionally, it estimates Driver-side data fetching time using historical average network bandwidth, constructing a multi-dimensional time-cost evaluation system that integrates these metrics for holistic cost assessment.

Algorithm 1 REST API-Based Task Information Extraction Algorithm

Input: Spark application ID: applicationId, Spark REST API address: apiUrl.
Output: List of SQL relation information: SqlRelation-Info[].
applicationUrl ← apiUrl + applicationId + "/sql"
dataSize ← DataFrame.size()
applicationData ← HTTP.GET(applicationUrl)
applicationName ← applicationData.getName()
duration ← applicationData.getDuration()
nodesInfo ← parse(applicationData.getNodeArray())
planDescription ← applicationData.getPlanDescription()
edges ← applicationData.getEdges()
return SqlRelationInfo[dataSize, duration, nodesInfo, edges, planDescription]

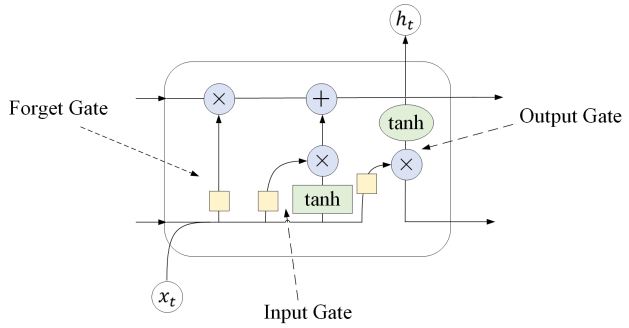

Fig. 3. LSTM Neural Network Structure.

2) Task Completion Time Prediction Algorithm Based on LSTM

Task execution times in cluster environments are influenced by multi-dimensional dynamic factors—data scale, resource configurations, network conditions, and data distribution—whose complex nonlinear relationships make them difficult for traditional statistical models to capture. To address this, we construct an LSTM-based time prediction model that learns from historical task data (features including data scale, number of Executors, CPU cores, and actual execution time) to accurately predict both Driver-side localized computation times and cluster distributed processing times. The LSTM network architecture is tailored to our specific data characteristics and prediction requirements, comprising multiple layers with configurable numbers of LSTM units. These units effectively capture long-term dependencies in time-series data, enabling the model to leverage historical patterns in resource allocation and task behavior. This capability enhances prediction accuracy for both localized and distributed execution paths, providing a robust foundation for the D2CS strategy's dynamic decision-making.

The input layer receives raw data, while the number of hidden layers and units per layer are tuned via experiments for optimal performance. Each LSTM unit contains forgetting, input, and output gates (Figure 3), enabling the network to determine when to retain or discard information—critical for capturing temporal dynamics in data during prediction. The output layer converts hidden layer outputs into final predictions: task execution times.

To validate the accuracy of the machine learning model for predicting task execution times, comparative experiments were conducted with classic machine learning models. The dataset was randomly split into a 70% training set and 30% test set. Root Mean Square Error (RMSE, Equation (1)) served as the model evaluation criterion, where n is the number of samples, y_i denotes the observed value, and $\hat{y}_i$ represents the model's predicted value:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2} \tag{1}$$

As training progresses, RMSE on both the training and test sets decreases, indicating high prediction accuracy and effective model learning from the dataset. Final results for all models are presented in Table I.

In this study, the LSTM network architecture consists of three layers: the input layer, hidden layer, and output layer.

TABLE I
RMSE OF THE MODELS

Type	RMSE
Bayesian Neural Networks	4.86
Support Vector Regression	6.86
K-nearest Neighbor	5.35
LSTM	4.69

The input layer processes the input data, x_t. The hidden layer contains one or more LSTM units, each incorporating a forget gate, an input gate, and an output gate to control information flow. The output layer processes the hidden layer's output and generates the final predicted value. The forgetting gate formula is defined as:

$$f_t = \text{sigmoid}\left(W_f\left[h_{t-1}, x_t\right] + b_f\right) \tag{2}$$

where f_t represents the forgetting gate output, w_f is the forgetting gate weight matrix, h_{t-1} denotes the hidden state from the previous time step, x_t is the current input at time t, and b_f is the bias term.The input gate operations involve two key computations:

$$\widetilde{c_t} = \tanh\left(W_c\left[h_{t-1}, x_t\right] + b_c\right) \tag{3}$$

$$i_t = \text{sigmoid}\left(W_i\left[h_{t-1}, x_t\right] + b_i\right) \tag{4}$$

Here, $\widetilde{c_t}$ is the candidate cell state update, w_c and w_i are the weight matrices for the candidate state and input gate, respectively, b_c and b_i are their corresponding bias terms, i_t represents the input gate output, and $\tanh$ denotes the hyperbolic tangent function.The cell state update equation is:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \widetilde{c_t} \tag{5}$$

where f_t is the forgetting gate output, c_{t-1} is the cell state at the previous time step, i_t is the input gate value, and $\widetilde{c_t}$ is the candidate state.The output gate operations are defined as:

$$o_t = \text{sigmoid}\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \tag{6}$$

$$h_t = o_t \cdot \tanh(c_t) \tag{7}$$

where W_o is the output gate weight matrix, b_o is the bias term, c_t is the current cell state, and o_t represents the output gate output.The final output layer maps the hidden state h_t to the predicted Sort task execution time:

$$\hat{y_t} = W_{\text{out}} h_t + b_{\text{out}} \tag{8}$$

where W_{out} and b_{out} are the output layer weight matrix and bias term, respectively.Experimental results show that small training sets lead to prediction inaccuracies, while excessively large sets increase training overhead. To balance performance, we leverage the past ten days of historical data as the training set, performing daily offline training. Despite external variables impacting CPU sorting time and cluster task completion time, the proposed model achieves robust prediction performance through systematic evaluation.

Experimental results show that the proposed LSTM model achieves superior performance, validating the effectiveness of our feature extraction approach. Among compared models, LSTM outperforms others with an RMSE of 4.69 on the dataset, highlighting its optimal ability to fit nonlinear time-series data. Consequently, LSTM is selected as the task time prediction model for D2CS, providing a high-precision foundation for time-cost evaluation.

3) Dynamic Execution Plan Selection Algorithm Based on Time Cost

Based on task data size estimation and dual-end (Driver side, Cluster side) execution time forecasting, this subsection designs a dynamic execution plan selection algorithm.

To estimate Driver-side data acquisition time, we calculate the average bandwidth B_w as a baseline parameter by collecting network bandwidth data from similar operator tasks over the past ten days. Using the task data size D_w obtained from Algorithm 1, the data transmission time for a single worker node is expressed as:

$$t_n^w = \frac{D_w}{B_w} \tag{9}$$

The core logic of the dynamic selection algorithm is to compare Driver-side and cluster-side time costs and select the execution path with the minimum time cost. The algorithm takes the task data size D_w and cluster configuration parameters (e.g., number of Executors, node computational capacity) as input and outputs the optimal execution plan. The detailed implementation is outlined in Algorithm 2.

Algorithm 2 Dynamic Execution Plan Selection Algorithm

Input: Data size D_w, Cluster configuration parameter set: CS_info
Output: Optimal Execution Plan
$t_{cloud} \leftarrow LSTM(D_w, information)$
$t_{driver\_fetch} \leftarrow t_n^w$ with eq(9)
$t_{driver\_sort} \leftarrow LSTM(D_w)$
$t_{driver} \leftarrow t_{driver\_fetch}$ + t_{driver_sort}
if $t_{\text{driver}} > t_{\text{cloud}}$ **then**
  "$SortExec$"
else
  "$DriverSortExec$"
end if

C. Parallel Shuffle Aggregation and Sorting Strategy

To address frequent disk spills caused by the serial "aggregate-then-sort" processing pipeline in Spark's Shuffle Read phase, this subsection proposes the Parallel Shuffle Aggregation and Sorting Strategy (PASA). By restructuring the data processing pipeline, PASA achieves efficient memory utilization, reduces I/O overhead, and enhances large-scale data processing efficiency.

In Spark distributed computing, Shuffle is a core component for cross-stage data redistribution, triggered by wide dependencies in RDDs—where a single partition of a child RDD depends on multiple partitions of the parent RDD. The Shuffle process consists of two phases (Figure 4):

In the Shuffle Write phase, Map tasks partition intermediate data by key, reduce data volume through pre-aggregation,

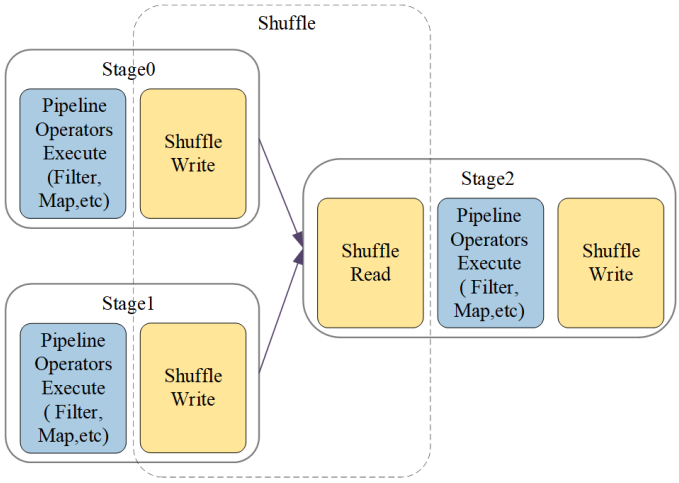

Fig. 4. Two Phases of Spark Shuffle.

and write it to local disk files according to sorting rules. During Shuffle Read, Reduce tasks fetch corresponding partitioned data from each node and perform aggregation (e.g., summation, deduplication) and sorting operations in sequence. Spark's current serial processing model operates as follows: it first uses the ExternalAppendOnlyMap data structure for aggregation, triggering a disk spill if memory is insufficient. After aggregation, the result set is sorted, with another disk spill occurring if data volume exceeds memory thresholds. The PASA strategy advances the sorting process to execute concurrently with aggregation. This concurrent approach shortens the execution pipeline, reduces memory usage, enhances cache efficiency, and minimizes disk spills by integrating sorting and aggregation—eliminating the need for separate spill operations during each stage.

D. Algorithm Implementation of the Parallel Shuffle Aggregation and Sorting Strategy

As depicted in Figure 5, Reduce tasks continuously fetch data from Map task partition files, using $fagg()$ for incremental aggregation as data is received—a process that aggregates while fetching. Once aggregation is complete, the data is loaded into an array, sorted by key, and the sorted results are either output or passed to subsequent operations.

In the traditional Shuffle Read phase—where aggregation precedes sorting—intermediate data may spill to disk if memory is insufficient during computation. When memory cannot store all data, part of it is written to disk to free space, which increases task completion time due to slower disk access compared to memory. Critically, because aggregation and sorting are performed sequentially, this serial pipeline often triggers two separate disk spills: one during aggregation when memory is exhausted, and another during sorting when the result set exceeds memory capacity.

The entire aggregation-and-sorting process can be modeled as follows:

(1) Amount of Overwritten Data

In the serial aggregation-then-sorting pipeline, when the aggregated data volume D exceeds available memory M, the spilled data volume during aggregation is:

$$S' = D - M \tag{10}$$

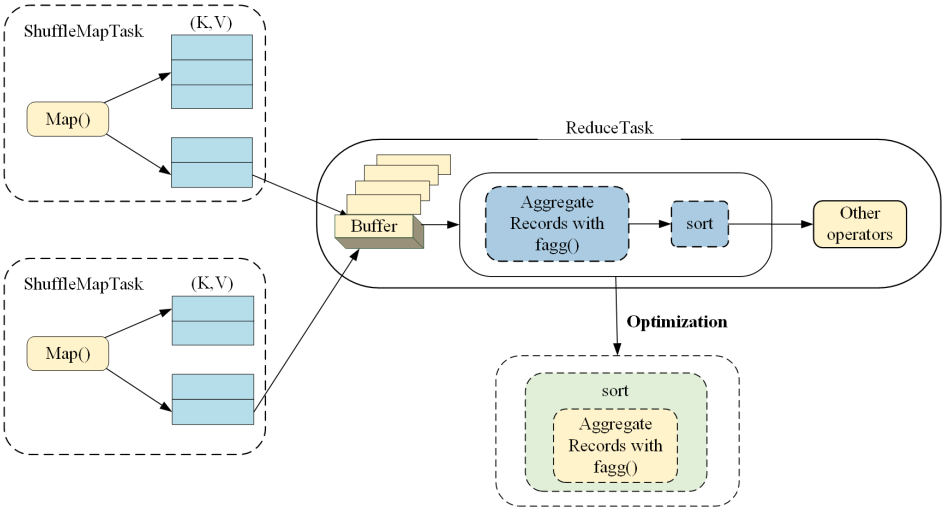

Fig. 5. Optimized Shuffle Read Process.

Subsequently, the sorting process operates on the aggregated results. Since sorting requires additional memory for intermediate results, a second spill occurs if the sorted data also exceeds M. The total spilled data volume from both stages is:

$$S = 2 \times S' \tag{11}$$

(2) Number of Spill Events

Spill events occur when available memory is insufficient to store data. Spark writes to separate disk files in units of 10,000 data records, where the average size of 10,000 records is B. The number of spills during aggregation is:

$$N' = \frac{S}{B} = \frac{D - M}{B} \tag{12}$$

Here, N' represents both the number of spill events and the count of generated spill files. During the subsequent sorting phase, the same memory constraint applies, leading to a total spill count of:

$$N = 2 \times N' \tag{13}$$

(3) Task Completion Time Calculation

The task completion time for the aggregation-sorting process includes both memory processing time and disk I/O time.

The memory processing time Tm depends mainly on the rate of data processing in memory Rmem and the number of spill events to be processed. The specific calculations are as follows:

$$T_m = \frac{M}{R_{\text{mem}}} \times (N + 2) \tag{14}$$

Where N is the total spill count, the plus 2 in this count accounts for the final in-memory processing without spills in both the aggregation and sorting phases. The disk spill time of the process can be expressed as:

$$T_d^w = \frac{B}{R_{\text{disk}}} \times N = \frac{S}{R_{\text{disk}}} \tag{15}$$

For each spill of data S' during aggregation, the sorting phase requires reading these spills from disk, incurring additional disk read overhead:

$$T_d^r = \frac{S'}{R_{\text{disk}}} \times N' \tag{16}$$

Combining memory processing and disk I/O times, the total task completion time is:

$$T_{\text{total}} = T_m + T_d^w + T_d^r \tag{17}$$

$$= \frac{S'}{R_{\text{disk}}} \times N' + \frac{S}{R_{\text{disk}}} + \frac{M}{R_{\text{mem}}} \times (N + 2) \tag{18}$$

The goal of PASA is to minimize the total time by reducing the spill count N, spilled data volume S, and disk read overhead, achieving comprehensive optimization of computation time, memory usage, and disk I/O efficiency. As outlined in Algorithm 3, PASA's core innovation is integrating sorting with aggregation: using the TimSort algorithm [31] to perform in-memory local aggregation for tuples with the same key during sorting. For ordered segments that exceed the memory threshold and spill to disk, a multi-way merge algorithm is applied for final sorting and aggregation, thus streamlining the pipeline to avoid full data redistribution.

IV. Experiment

A. Experiment Setting

We extend Spark 3.0.0's physical execution plan generator and ShuffleReader interface, embedding the D2CS and PASA strategies as plug-ins into the native framework to ensure compatibility with existing cluster scheduling mechanisms. Experiments are conducted on a distributed cluster comprising 6 physical servers: 1 Master node (responsible for resource scheduling) and 5 Worker nodes (handling computation tasks). Cluster hardware configurations are listed in Table II. Spark uses the Standalone deployment mode, with core parameters shown in Table III, ensuring a controlled experimental environment through uniform configuration.

B. D2CS Experimental Validation

To validate the dynamic optimization performance of the D2CS strategy, we integrate it into the Spark 3.0.0 framework and design comparative experiments for the Sort operator. Experiments employ different application submission configurations (Table III), covering small-scale (100 records) to large-scale (2.5 million records) data scenarios. Datasets in Parquet format are generated in 10,000-record increments.

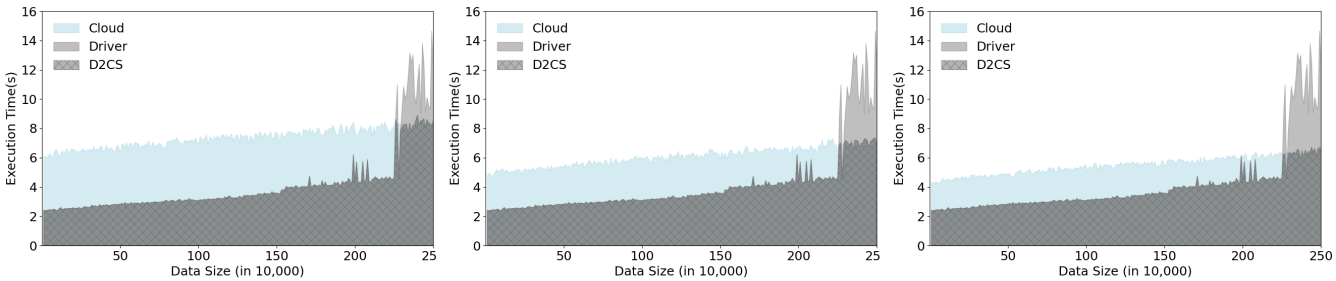

Fig. 6. Execution Time Comparison of Execution Plans Under Different Configurations.

Algorithm 3 Shuffle Read Optimization Algorithm Based on Parallel Sorting Aggregation

Input: Cross-node data block collection: B, Aggregation function: $f_{\text{agg}} : (V_1, V_2) \to V_3$, Sorting rule: $O : K \times K \to \{-1, 0, 1\}$, Memory threshold: θ
Output: Global ordered aggregation result: $R\{(k, c)\}$, where $c = f_{\text{agg}}(V_1, V_2, \ldots, V_m)$
stream ← OpenShuffleBlockStream(B) // Fetch data
deserialized_iter ← DeserializeToKeyValue(stream)
sorter ← CreateExternalSorter(f_{agg}, O) // Initialize
while there exists (k, v) in deserialized_iter **do**
  sorter.insert(k, v) // Insert data into sorter
  if sorter.memory_usage $> \theta$ **then**
    sorter.spill() // Spill blocks to disk
  end if
end while
R ← sorter.merge() // Merge all spilled blocks
return R // Return the global result
Function sorter.insert(k, v):
buffer ← DynamicArray $< (K, C) >$ // Initialize buffer
// Find insertion position via binary search based on sorting rule O
pos ← BinarySearch(buffer, k, O)
if buffer[pos].key $== k$ **then** // Existing key
  buffer[pos].value ← f_{agg}(buffer[pos].value, v)
else // New key: insert while maintaining sorted order
  buffer.insert(pos, (k, v)) // Insert new key-value pair
end if
if buffer.size $>$ TimSort.RunLength **then**
  Run ← TimSort.StableSort(buffer) //Create sorted run
  Spill(Run) // Spill the sorted run to disk
  buffer.clear() // Clear buffer
end if
Function sorter.merge():
MinHeap ← PriorityQueue $< (K, C) >$ (compare $= O$)
for each spilled run R_i **do** // Load the first record
  MinHeap.push(R_i.next())
end for
while MinHeap is not empty **do** // Merge all spilled runs
  $(k_{\text{current}}, c_{\text{current}})$ ← MinHeap.pop()
  while MinHeap.top().key $== k_{\text{current}}$ **do**
    $c_{\text{current}} \leftarrow f_{\text{agg}}(c_{\text{current}}$, MinHeap.pop().value)
  end while
end while

TABLE II
CLUSTER CONFIGURATION

Type	Configuration
CPU	Intel ® Xeon ® Platinum8160 CPU ® 2.10GHz
RAM	80GB
Hard disk	500GB
Environment	Centos7.0, Spark3.0.0, Hadoop 2.6.0
Digital meter	2500W, 10A
JDK	JDK 1.8

TABLE III
CONFIGURATION OF SPARK PARAMETERS

Type	Configuration
Executor cores	4, 2, 1
Executor memory	8GB, 4GB, 2GB
Executor number	6

TABLE IV
CONFIGURATION OF SPARK PARAMETERS

Type	Configuration
Executor cores	4
Executor memory	8GB
Executor number	6

Each experiment is run 10 times, with the average execution time serving as the performance metric.

The effectiveness of the D2CS dynamic selection strategy is validated by comparing the execution times of the Sort operator under the Driver-side execution plan (DriverSort) and cluster-side execution plan (CloudSort). To account for heterogeneous node hardware configurations in different cluster environments, we test D2CS adaptability by varying Executor CPU cores and memory capacity. Experimental results are shown in Figure 6.As indicated, DriverSort significantly outperforms CloudSort when data scale is below 2.3 million records; CloudSort becomes more efficient beyond this threshold. This is because small-scale data processing in the cluster incurs two Shuffle processes—data distribution to Executors and result aggregation back to the Driver—where network and scheduling overheads exceed the CPU cost of Driver-side local sorting. Conversely, when data scale surpasses 2.3 million records, Driver-side computational resources become a bottleneck. Here, the parallel computing advantages of the distributed cluster outweigh Shuffle overheads, making CloudSort more efficient.

Unlike Spark's default cluster and Driver-side execution, the D2CS strategy dynamically selects the optimal execu-

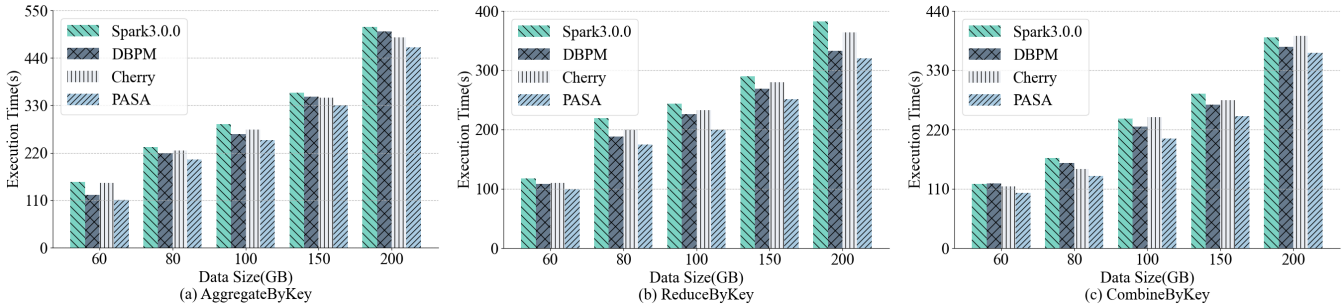

Fig. 7. Execution Time of Different Strategies for Various Operators.

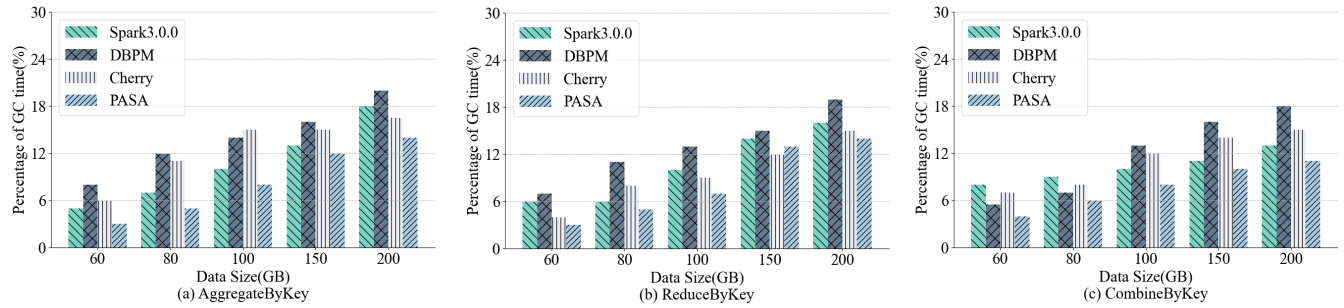

Fig. 8. GC Time Share of Different Strategies.

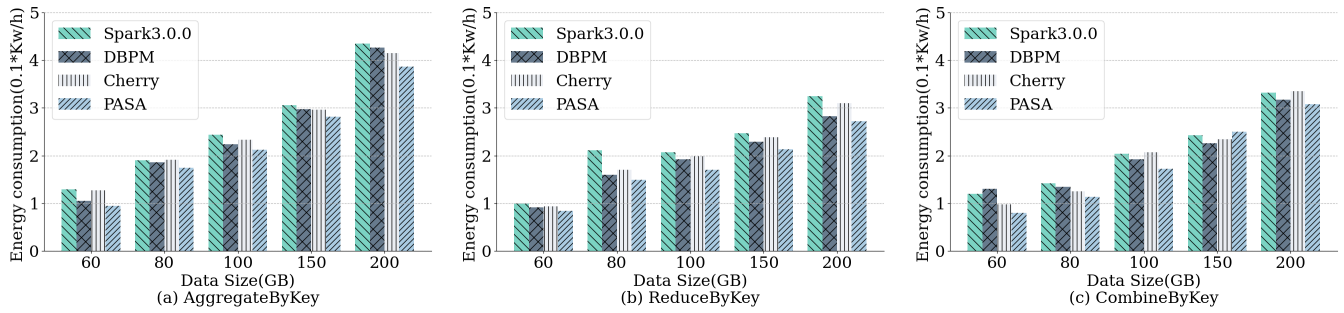

Fig. 9. Server Energy Consumption of Different Strategies.

tion path by real-time sensing of data scale and cluster configuration parameters: directly enabling localized Driver-side execution to avoid shuffle overhead when the data volume is small, and taking advantage of the cluster parallel computing capability when the data volume is large. The task completion time comparison under different configurations is shown in Table V. Compared with Spark's default cluster execution mechanism, D2CS reduces the computation time from 865 s to 63 s with 1 core and 2 GB memory per Executor (a performance improvement of 92.7%), reduces the computation time from 573 s to 93 s with 2 cores and 4 GB memory per Executor (an 83.8% improvement), and reduces the computation time from 444 s to 109 s with 4 cores and 8 GB memory per Executor (a 75.5% improvement).

TABLE V
TASK COMPLETION TIME COMPARISON UNDER DIFFERENT EXECUTOR CONFIGURATIONS

Executor Configuration	**Spark Default**	**D2CS**	**Performance Improvement Rate**
Each executor has 1 core and 2GB memory	865s	63s	92.7%
Each executor has 2 cores and 4GB memory	573s	93s	83.8%
Each executor has 4 cores and 8GB memory	444s	109s	75.5%

C. PASA Experimental Validation

To validate the optimization effect of the PASA strategy in the Shuffle Read phase, this study selects three representative strategies—Spark native mechanism, Cherry [30], and DBPM [29]—for systematic comparison. The experiments use the HiBench benchmark suite to generate WordCount datasets with multi-level data sizes (60 GB, 80 GB, 100 GB, 150 GB, 200 GB) ranging from 60 GB to 200 GB. Spark cluster parameters are configured as shown in Table IV.

The experimental scheme covers Spark's three core aggregation operators (ReduceByKey, AggregateByKey, CombineByKey) with dictionary-order sorting. Results in Figure 7 show that compared with Spark's native strategy, PASA achieves execution time improvements of 7.42%–39.88%. When compared with Cherry and DBPM, PASA reduces execution time by 2.95%–15.19% and 4.89%–26%, respectively. As data scale increases, PASA's advantages intensify,

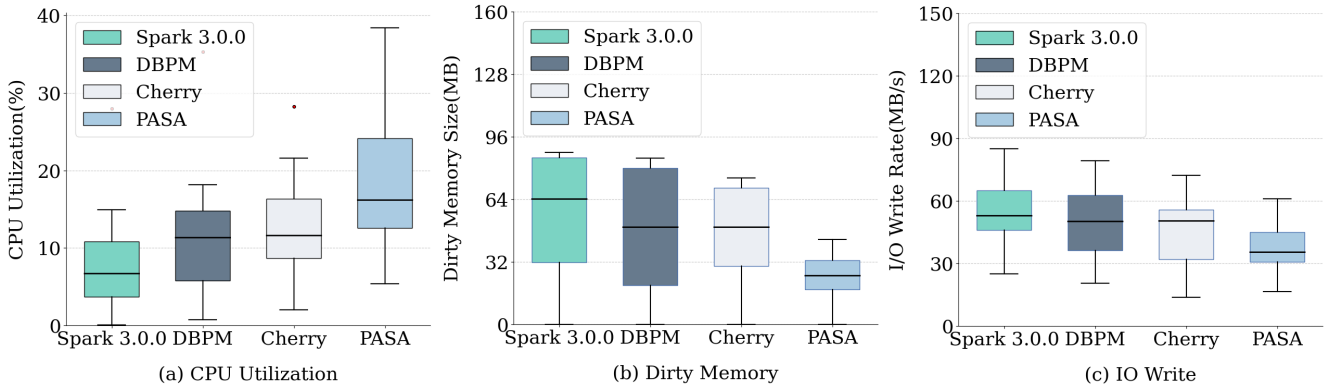

Fig. 10. CPU Usage, Memory Dirty Pages, and Disk I/O Status of Different Strategies.

reflecting its efficiency and stability in large-scale scenarios. The Cherry strategy restructures network exchange for Shuffle blocks but retains the serial aggregation-sorting pipeline in Shuffle Read, leading to high latency for massive data. DBPM balances data partitions to avoid stragglers but incurs overhead from sampling and partitioning decisions. By contrast, PASA leverages Spark's native Shuffle framework, shortening the pipeline through parallelized aggregation and sorting without introducing complex mechanisms—ensuring compatibility while achieving lightweight optimization.

To deeply analyze the performance improvement mechanism, we integrated node_exporter and graphite_exporter with Prometheus to build a multi-dimensional monitoring system for collecting node resource utilization and energy consumption metrics.

Figure 8 illustrates the trend of garbage collection (GC) time share as data size increases. As data scale grows, the GC ratio and elapsed time increase for all strategies. The DBPM strategy incurs higher memory overhead due to additional data structure maintenance, leading to a significant surge in GC time share. Although Cherry uses a push-based data prefetching mechanism to achieve interleaved memory utilization, its efficiency declines with larger datasets, causing a steep rise in GC ratio. By contrast, PASA employs ordered data structures for incremental aggregation, reducing memory fragmentation and invalid data storage. Under high load, PASA's GC ratio increase is only 48%–77.7% of the native strategy, significantly optimizing memory utilization efficiency.

Energy consumption is also a key concern for large-scale data centers, and Figure 9 shows the energy consumption of each strategy under different data scales. Compared with the Spark default strategy, PASA achieves global energy savings of 7.42%–29.44%; when compared with DBPM and Cherry strategies, PASA reduces energy consumption by 4.94%–15.15% and 4.8%–25.6%, respectively. This benefit stems from PASA's control over disk I/O and efficient utilization of CPU resources, which shortens task completion time while reducing overall system energy consumption—thereby lowering operating costs and environmental impact, and demonstrating its significant advantages in green computing and cost optimization.

Figure 10 shows the comparison of key system metrics collected. By integrating user-defined aggregation functions during sorting, PASA increases average CPU utilization by 5.33%–6.74%. Additionally, by merging the aggregation and sorting processes, PASA significantly reduces data written to disk, decreasing memory dirty page size by 33.2%–55.3% and disk I/O usage by 19.86%–34.76%. These improvements effectively mitigate resource contention issues in multi-tasking environments.

The above experiments demonstrate that the PASA strategy, by restructuring the Shuffle processing pipeline, achieves synergistic optimization of computational efficiency, memory management, energy consumption control, and system stability—all without increasing system complexity. This approach provides an efficient and sustainable solution for large-scale data processing.

V. Conclusion

In Spark distributed computing, the traditional fixed execution model lacks dynamic awareness of data size and cluster resource configuration, leading to computational inefficiency for small-data tasks due to double Shuffle overhead and frequent disk spills caused by the serial aggregation-sorting pipeline in the Shuffle Read phase for large datasets. To address these challenges, this paper proposes a Data-size and Cluster-configuration-aware dynamic execution plan Selection strategy (D2CS) and a Parallel Aggregation-Sorting Optimization strategy (PASA). Specifically, D2CS extends Spark's physical execution plan to design a lightweight execution path adapted to the Driver-side, dynamically predicts the execution time for both Driver-side and cluster-side via an LSTM model, and selects the execution plan with minimal total overhead using a time-cost model. Taking the Sort operator as an example, this strategy avoids redundant cluster transmission and scheduling overhead in small-data scenarios, significantly reducing resource contention and execution time. The PASA strategy restructures the Shuffle Read processing pipeline by executing aggregation and sorting operations in parallel, transforming the serial model into incremental ordered aggregation to achieve efficient memory utilization through ordered data structures—particularly enhancing data processing efficiency in large-scale scenarios. Based on Spark 3.0.0, we implement D2CS and PASA, and experimental results demonstrate that our strategies effectively shorten task completion time and significantly improve system resource utilization.

Data and Code Availability

The data and code supporting this study are available in the GitHub repository at https://github.com/wcy666103/spark-paper

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[2] Apache Storm-Free and open source distributed realtime computation system. https://storm.apache.org

[3] Apache Spark-Unified engine for large-scale data analytics. https://spark.apache.org

[4] Apache Flink-Stateful Computations over Data Streams. https://flink.apache.org

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a Warehousing Solution over a MapReduce Framework," Proceedings of the VLDB Endowment, vol. 2, no. 2, pp. 1626–1629, 2009.

[6] Y. Zhao and R. Chen, "Spark SQL Query Optimization Based on Runtime Statistics Collection," in 2021 IEEE 6th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), IEEE, pp. 250–255, 2021.

[7] T. Chiba, T. Yoshimura, M. Horie and H. Horii, "Towards Selecting Best Combination of SQL-on-Hadoop Systems and JVMs," in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, pp. 245–255, 2018.

[8] A. Roy, A. Jindal, P. Gomatam, X. Ouyang, A. Gosalia, N. Ravi, S. Mann, and P. Jain, "SparkCruise: Workload Optimization in Managed Spark Clusters at Microsoft," Proceedings of the VLDB Endowment, vol. 14, no. 12, pp. 3122–3134, 2021.

[9] Z. He, Q. Huang, Z. Li, and C. Weng, "Handling Data Skew for Aggregation in Spark SQL Using Task Stealing," International Journal of Parallel Programming, vol. 48, pp. 941–956, 2020.

[10] X. Ji, M. X. Zhao, M. Y. Zhai, and Q. X. Wu, "Query Execution Optimization in Spark SQL," Scientific Programming, vol. 2020, no. 1, Article ID 6364752, 2020.

[11] J. Xin, K. Hwang, and Z. Yu, "LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications," in Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), Association for Computing Machinery, New York, NY, USA, pp. 674–684, 2022.

[12] Y. Shen, X. Y. Ren, Y. P. Lu, H. J. Jiang, H. Y. Xu, D. Peng, Y. Li, W. T. Zhang, and B. Cui, "Rover: An Online Spark SQL Tuning Service via Transfer Learning," in Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 4000–4812, 2023.

[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark".Proceedings of the 2015 ACM SIGMOD international conference on management of data.: 1383-1394, 2015.

[14] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in Cidr, vol. 5, pp. 225–237, 2005.

[15] W. Roediger, T. Muehlbauer, A. Kemper, and T. Neumann, "High-speed Query Processing over High-speed Networks," arXiv preprint arXiv:1502.07169, 2015.

[16] D. Jang, H. Yoon, K. Jung, and Y. D. Chung, "QHB+: Accelerated Configuration Optimization for Automated Performance Tuning of Spark SQL Applications," IEEE Access, vol. 12, pp. 60138–60148, 2024.

[17] K. Yamasaki and T. Amagasa, "RDF Data Partitioning for Efficient SPARQL Query Processing with Spark SQL," in International Conference on Information Integration and Web Intelligence, Cham, Switzerland: Springer Nature, 2023, pp. 92–106.

[18] B. Chandramouli and J. Goldstein, "Patience is a Virtue: Revisiting Merge and Sort on Modern Processors," in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 731–742, 2014.

[19] A. LaMarca and R. E. Ladner, "The Influence of Caches on the Performance of Sorting," Journal of Algorithms, vol. 31, no. 1, pp. 66–104, 1999.

[20] C. Nyberg, T. Barclay, Z. Cvetanovic, "AlphaSort: a Cache-Sensitive Parallel External Sort," Morgan Kaufmann Publishers Inc., 1998.

[21] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal, "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures," in Proceedings 17th International Conference on Data Engineering, IEEE, pp. 567–574, 2001.

[22] J. Cieslewicz and K. A. Ross, "Adaptive Aggregation on Chip Multiprocessors," in Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 339–350, 2007.

[23] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber, "Cache-Efficient Aggregation: Hashing Is Sorting," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15), Association for Computing Machinery, New York, NY, USA, pp. 1123–1136,2015.

[24] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable Aggregation on Multicore Processors," in Proceedings of the Seventh International Workshop on Data Management on New Hardware, pp. 1–9, 2011.

[25] Y. Shen, J. Song, and D. Jiang, "Using Vectorized Execution to Improve SQL Query Performance on Spark," in Proceedings of the 50th International Conference on Parallel Processing, pp. 1 - 11, 2021.

[26] X. Wu and Y. He, "Optimization of the Join between Large Tables in the Spark Distributed Framework," Applied Sciences, vol. 13, no. 10, p. 6257, 2023.

[27] J. Lin, T. Ji, X. Hao, H. Cha, Y. Le, X. Yu, and A. Akella, "Towards Accelerating Data Intensive Application's Shuffle Process Using SmartNICs," Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 7, no. 2, pp. 1–23, 2023.

[28] F. S. Luan, S. Wang, S. Yagati, S. Kim, K. Lien, I. Ong, T. Hong, S. Cho, E. Liang, and I. Stoica, "Exoshuffle: An Extensible Shuffle Architecture," in Proceedings of the ACM SIGCOMM 2023 Conference, 2023, pp. 564–577.

[29] C. Li, Q. Cai, and Y. Luo, "Data balancing-based intermediate data partitioning and check point-based cache recovery in Spark environment," The Journal of Supercomputing, vol. 78, no. 3, pp. 3561 - 3604, 2022.

[30] N. Nikitas, I. Konstantinou, V. Kalogeraki, and N. Koziris, "Cherry: A Distributed Task-Aware Shuffle Service for Serverless Analytics," in 2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, pp. 120–130, 2021.

[31] A. S. Sabah, S. S. Abu-Naser, Y. E. Helles, R. F. Abdallatif, F. Y. A. Abu Samra, A. H. Abu Taha, N. M. Massa, and A. A. Hamouda, "Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics," International Journal of Academic Engineering Research (IJAER), vol. 7, no. 6, pp. 76–84, 2023.

Congyang Wang born in 2000, received a bachelor's degree in software engineering from Henan University in 2022, where he graduated from the School of Software. He is currently a graduate student in the School of Software at Henan University. His research interests include distributed computing and big data processing.He has published three SCI journal paper and won several national awards in computer-related competitions. He is a CCF (China Computer Federation) student member and an Alibaba Cloud Open Source Pioneer. Since 2022, he has completed two industrial projects and holds two patents. As an Apache Contributor, he has made technical contributions to open-source projects in the big data field.