Multi-job Merging Framework and Scheduling Optimization for Apache Flink

Hangxu Ji[1], Gang Wu[1(✉)], Yuhai Zhao[1], Ye Yuan[2], and Guoren Wang[2]

[1] School of Computer Science and Engineering, Northeastern University, Shenyang, China
wugang@mail.neu.edu.cn

[2] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

Abstract. With the popularization of big data technology, distributed computing systems are constantly evolving and maturing, making substantial contributions to the query and analysis of massive data. However, the insufficient utilization of system resources is an inherent problem of distributed computing engines. Particularly, when more jobs lead to execution blocking, the system schedules multiple jobs on a first-come-first-executed (FCFE) basis, even if there are still many remaining resources in the cluster. Therefore, the optimization of resource utilization is key to improving the efficiency of multi-job execution. We investigated the field of multi-job execution optimization, designed a multi-job merging framework and scheduling optimization algorithm, and implemented them in the latest generation of a distributed computing system, Apache Flink. In summary, the advantages of our work are highlighted as follows: (1) the framework enables Flink to support multi-job collection, merging and dynamic tuning of the execution sequence, and the selection of these functions are customizable. (2) with the multi-job merging and optimization, the total running time can be reduced by 31% compared with traditional sequential execution. (3) the multi-job scheduling optimization algorithm can bring 28% performance improvement, and in the average case can reduce the cluster idle resources by 61%.

Keywords: Multi-job merging · Scheduling optimization · Distributed computing · Flink

1 Introduction

The IT industry term "Big Data" has existed for more than a decade and is a household term. To provide improved support for massive data computing, researchers have developed various distributed computing systems and are constantly releasing new versions of them to improve the system performance and enrich system functions.

Apache Flink [2] is the latest generation of distributed computing systems and exhibits high throughput and low latency when processing massive data.

C. S. Jensen et al. (Eds.): DASFAA 2021, LNCS 12681, pp. 20–36, 2021.
https://doi.org/10.1007/978-3-030-73194-6_2

It can cache intermediate data and support incremental iteration using its own optimizer. Many experimental studies, optimization technologies, and application platforms based on Flink are emerging because of its numerous advantages. For example, in the early days of Flink's birth, most of the research focused on the comparison between Flink and Spark [10,11,15], and pointed out that Flink is more suitable for future data computing. With the popularity of Flink, recent researches include testing tools based on Flink [9], multi-query optimization technology [16], and recommender systems [5], etc.

However, almost the distributed computing systems exhibit insufficient utilization of the hardware resources. Although Flink maximizes resource utilization by introducing TaskSlot to isolate memory, idle resources also exist because of the low parallelism of some Operators during traditional sequential execution. Moreover, when a user submits multiple jobs, Flink only run them on a first-come-first-executed (FCFE) basis, which cannot make jobs share the Slots. In a worse-case scenario, if job A is executing and job B after it does not meet the execution conditions because of insufficient remaining resources, job C cannot be executed in advance even though job C after job B meets the execution conditions, causing severe wastage of resources. These FCFE strategies of running multiple jobs only ensure fairness at the job level, but are not desired by users. In most cases, users only desire the minimum total execution time for all jobs. The above problems can be solved by simultaneously executing multiple jobs and dynamically adjusting the job execution sequence so that jobs that meet the execution conditions can be executed in advance.

In this study, we review the problem of insufficient utilization of system resources due to the fact that Flink does not support simultaneous execution of multiple jobs and optimization of execution sequence, and then focus on the multi-job efficiency improvement brought about by increasing Slot occupancy rate. The basic idea is to make simultaneously executing through multi-job merging and dynamically adjusting the execution sequence through multi-job scheduling, and the contributions of this paper are summarized below.

(1) We propose a groundbreaking framework that can support multi-job merging and scheduling in Flink. It can collect and parse multiple jobs to be executed, and generate new job execution plans through two optimization methods of multi-job merging and scheduling, and submit them to Flink for execution.
(2) To simultaneously execute multiple jobs, we propose multi-job merging algorithms based on subgraph isomorphism and heuristic strategies to enable multiple jobs to share the Slots. Both two algorithms can improve the efficiency and adapt to different job scenarios during the experiment.
(3) To dynamically adjusting the job execution sequence, we propose multi-job scheduling algorithm based on maximum parallelism to make jobs that satisfy the remaining resources execute in advance. Experimental results demonstrate that the algorithm can enhance the efficiency and reduce free resources.

The remainder of this paper is organized into 5 sections. Section 2 introduces the Flink DAGs, Flink Slots, and summarizes the current contributions toward improving resource utilization in distributed computing. Section 3 introduces the multi-job collection and execution agent, including the components, implementation method, and function. Section 4 describes multi-job execution optimization algorithms, including merging optimization and scheduling optimization. Section 5 presents the performance evaluation with respect to the running time and the number of Slots idle. Section 6 presents a brief conclusion.

2 Background and Related Work

In this section, we first summarized some of the implementation principles in Flink, including the composition and generation process of Flink DAGs, the functions and advantages of Flink Slot, to verify the feasibility of our work. Then, the related work of distributed job generation optimization and scheduling optimization is explained, and the advantages and deficiencies of existing work are pointed out.

2.1 Flink DAGs

Flink uses DAGs (Directed Acyclic Graphs) to abstract operations, which are more able to express the data processing flow than the traditional MapReduce [7]. According to the job submission and deployment process, Flink DAGs mainly includes JobGraph and ExecutionGraph. Figure 1 depicts the process of generating Flink DAGs. First, the system create JobVertexIDs based on the Operators in the job, chains the Operators through the Optimizer, and then generates a JobGraph by adding JobEdges and setting attributes. JobGraph is composed of three basic elements: JobVertex, JobEdge and IntermediateDataSet,

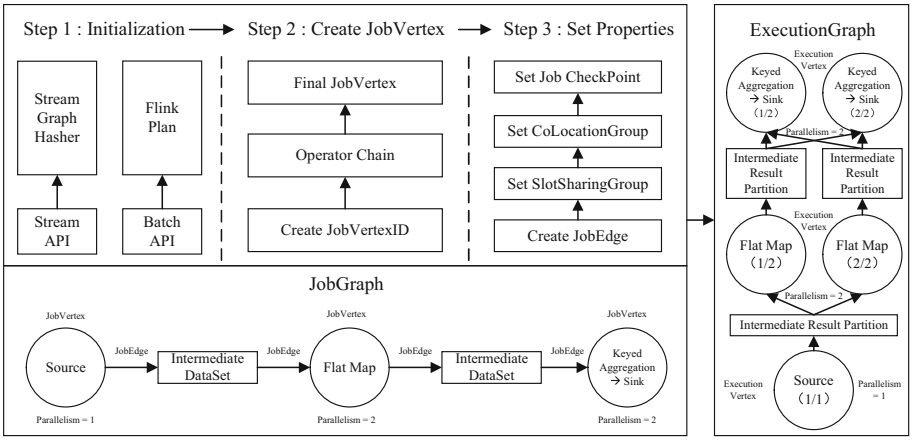


Fig. 1. The process of generating Flink DAGs (JobGraph and ExecutionGraph)

and contains all the contents of a job by assigning various attributes to these three elements. Finally, JobManager divides JobGraph into ExecutionVertex, IntermediateResultPartition and ExecutionEdge equal to its number according to the degree of parallelism to generate the final ExecutionGraph. Therefore, the research on JobGraph generation process is the core of Flink optimization, and it is also the focus of this work.

2.2 Flink Slot

In order to control the number of subtasks run by internal threads, Flink introduced TaskSlot as the minimum resource unit. The advantage of Slot is that it isolates memory resources, so jobs transmitted from JobMaster can be independently executed in different Slot, which can improve the utilization of cluster resources. As shown in Fig. 2, TaskManagers receive the task to be deployed from JobManager. If a TaskManager has four Slots, it will allocate 25% of memory for each Slot. One or more threads can be in each Slot, and threads in the same Slot share the same JVM. When subtasks belong to the same job, Flink also allows sharing Slot, which can not only quickly execute some tasks that consume less resources, but also logically remove redundant calculations that consume resources. It is precisely because of the existence of shared Slot that the multi-job merging and optimization techniques we will introduce in Sect. 4 are possible.

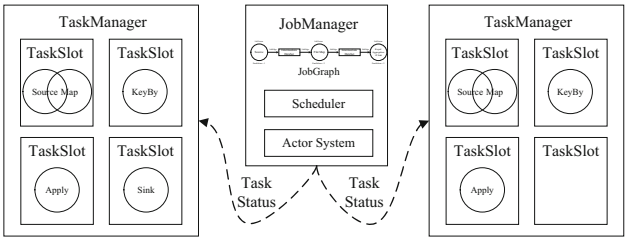


Fig. 2. Flink task deployment

2.3 Related Work

Current distributed computing systems, such as Spark [23] and Flink, are executed by converting complex programming logic into simple DAGs. The complex programming logic is mainly reflected in the user-defined function (UDF) in the Operators, so most of the research is to analyze UDF and construct optimization technology. Mainstream DAGs generation and optimization strategies include nested query decomposition technologies involving UDF [13], Operator reuse method [17,19], and Operator rearrangement algorithms [18], etc. In addition, part of the research is based on UDF code analysis to seek optimization opportunities [1,8,12,17]. In terms of distributed job scheduling optimization

and load balancing, each distributed computing system has its own scheduler as its core component [3,21]. At the same time, due to the increasing complexity of distributed operations and the continuous expansion of node scale, a large number of optimization technologies have been born. For example, in the research on Hadoop, researchers have proposed scheduling strategies based on job size [22], resource quantity [14], and deadline awareness [4]. In Spark based on memory computing, current research includes interference-aware job scheduling algorithm [21], job scheduling algorithm based on I/O efficiency [20], etc. Although the above works have improved the job efficiency, they are all oriented to a single job, without considering the mutual influence between multiple jobs.

3 Framework Structure

3.1 Model

We propose a framework, which is an Agent implemented between the Flink Client and Flink JobManager, and capable of supporting multi-job merging and scheduling optimization. The ultimate goal of the framework is to generate optimized Flink DAGs. The composition of this framework is illustrated in Fig. 3. In the following, we will introduce Collector, Parser and Generator respectively. The Optimizer will be described in detail in Sect. 4.

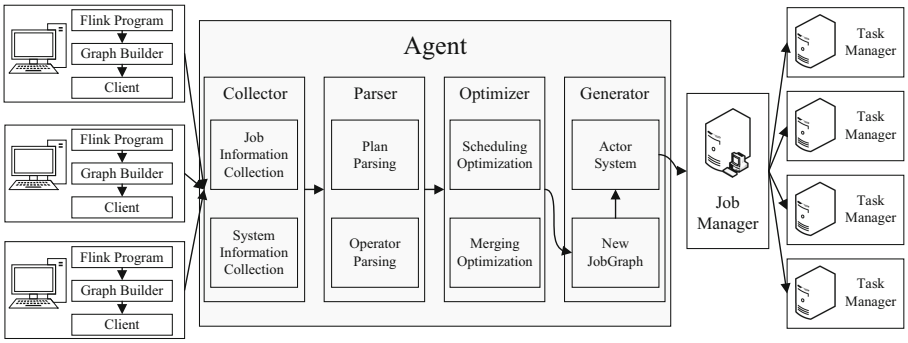


Fig. 3. The structure of the framework

Collector. In order to improve the multi-job efficiency by improving the utilization of system resources, the Optimizer must receive job information and system information as the data to be analyzed. Therefore, the Agent first provides a Collector, which collects the following information:

- **Jar Files:** As mentioned above, Flink abstracts computational logic in the form of DAGs. DAGs contain important information such as operators and UDFs, which are encapsulated in the Jar files.

- **Flink Plan:** The Flink Plan contains the deployment and strategy for job execution, mainly including the following information: GenricDataSinkBase in a collective form, which contains data terminal attributes; Cache files in the form of a HashMap, where the key is the name of the file and the value is the path to the file; ExecutionConfig is the configuration information for executing a job, which includes the adopted execution policy and the recovery method for the failed restart.
- **System Resources:** These mainly include the number of CPU cores and the memory size of each node in the cluster, as well as specific information in the Flink configuration file, including parallelism, Slot quantity, etc.

Parser. Because there are nested composite attributes inside Flink Plan and Operator, they cannot be directly converted to transferable byte streams, so a Parser is built in the Agent to serialize them. Algorithm 1 shows the process of serialization. First, gets the Sinks collection in Flink Plan (lines 1–3), and then assign the Operator and the user code encapsulation properties in it to the new object (lines 4–5). Finally, CatchFile, ExecutionConfig, JobID and the user code encapsulation properties are serialized respectively (lines 6–9). The final result is written on the Agent (line 10). At this point, the Agent has obtained all the information the Optimizer needs.

Algorithm 1: Serialization

Input: Flink Plan
Output: Job information of byte stream type

```
1  get DataSinkBase and GenericDataSinkBase from Plan;
2  get Operator op from GenericDataSinkBase;
3  read input Operator from DataSinkBase;
4  if UserCodeWraper is not null then
5  |  Set UserCodeWraper to OperatorEx;
6  Serialize(CatchFile); Serialize(ExecutionConfig); Serialize(OperatorEx);
7  if length > Buffer_Size then
8  |  add Buffer_Size;
9  Serialize(JobID);
10 writeToAgent;
```

Generator. The Generator first receives the optimization strategy sent from the Optimizer, including the jobs that can be merged and the optimal execution sequence of the jobs. The multi-job JobGraph is then generated by calling the implemented multiJobGraphGenerator. Since the multi-job scheduling optimization is an optimization in job execution order, when the jobs do not need to be merged, JobGraphGenerator in Flink is directly called by the Generator

to generate JobGraph. Finally, similar to the functionality of the Flink Client, the Actor System is responsible for submitting jobs to the cluster for execution.

3.2 Advantages

In the traditional Flink, the Client submits a job to the JobManager, and the JobManager schedules tasks to each TaskManager for execution. Then, the TaskManager reports heartbeat and statistics to the JobManager. When a user submits multiple jobs, Flink schedules each job on a FCFE basis. The proposed framework establishes multi-job collection, merging and scheduling optimization functions between Client and JobManager, without modifying and deleting Flink's own source code, which has two advantages. Firstly, since the framework and Flink are completely independent of each other, no matter which version is updated, they will not be affected by each other. In addition, the framework adds a switch to Flink, which allows users to choose whether to turn on the multi-job merging and scheduling optimization functions, because the traditional FCFE process ensures fairness at job level, which is also what some users require.

4 Multi-job Merging and Scheduling

This section introduces the two modules in Optimizer in detail, including two multi-job merging and optimization algorithms, and a multi-job scheduling optimization algorithm.

4.1 Multi-job Merging

Multi-job merging is suitable for situations where the cluster nodes are not large and the maximum parallelism of a single job reaches or approaches the maximum parallelism of the cluster. When the user submits multiple jobs and expects the shortest total execution time of the them, the multi-job merging and optimization module will be turned on. For jobs that can be merged, Optimizer merges the execution plans of these jobs into one execution plan, and the internal

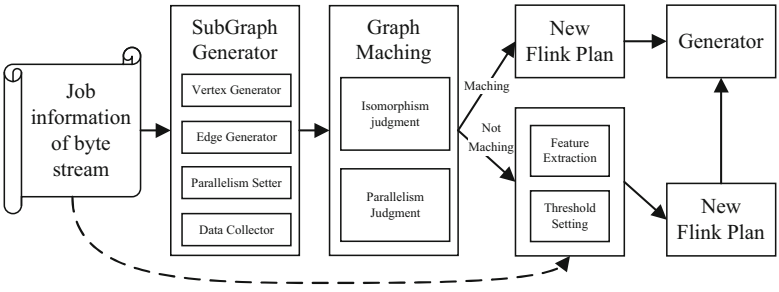


Fig. 4. Multi-job merging module

structure of these execution plans still guarantees their connection sequence. In addition, the user can choose whether to enable the function and the maximum number of jobs to be merged. As shown in Fig. 4, the module first obtains the job information, and uses the subgraph isomorphism algorithm to select jobs with higher similarity to merge. For dissimilar jobs, a heuristic method is used to set thresholds to determine the merged jobs.

Problem Description. Flink Slot uses a memory isolation method to allocate and reclaim resources, which greatly facilitates the management of cluster memory resources, but when users submit multiple jobs, memory resources cannot be shared between jobs. A job in the execution process has all its allocated resources, and the resources will not be recycled until the end of the job. Especially when the parallelism of some Operators is too low, the waste of resources will be more obvious. Therefore, the proposed multi-job merging algorithm aims at merging multiple jobs so that they can use Slot resources together, thereby improving the resource utilization.

Subgraph-Based Merging. In order to find out which jobs are merged first, we give a definition as follow:

- **Definition 1: Job Isomorphism.** A job can be represented by triples: $job = (G, P, D)$, where G is the JobGraph, P represents the maximum degree of parallelism, and D is the amount of data input. G_{sub} is a subgraph generated based on the key information in JobGraph. If $job1.G_{sub}$ and $job2.G_{sub}$ are isomorphic, $job1.P = job2.P$, and $job1.D \approx job2.D$, we determine that the two jobs are isomorphic and can be merged first.

The algorithm first choose the isomorphic jobs with similar data size for merging as much as possible, because they have similar task deployment and data deployment in TaskManager. It will make the merged jobs have fewer thread switching during execution, with better utilization of system resources. Since Flink uses JobGraphs to abstract jobs, the subgraph isomorphism algorithm can be used to judge the similarity of jobs. We only need to find a classic exact subgraph isomorphism algorithm to solve it because the vertex scale of JobGraphs is very small. We choose VF2 [6] among a large number of classic exact subgraph isomorphism algorithms, because it is more suitable for solving subgraph isomorphism of directed graphs and has a smaller space consumption.

Algorithm 2 shows the process of subgraph-based merging. First, select a job from the jobs to be executed (line 1), and use the reverse traversal method to generate a subgraph by reading its job information (line 2). Then use VF2 to perform subgraph isomorphism calculation, find all mergeable jobs, and add them to a collection (lines 3–7). If there are jobs that can be merged, the job merge algorithm will be executed and the merged job information will be deleted (lines 8–10). Finally, generate the new Flink Plan (line 11).

Algorithm 2: Subgraph-based Merging

Input: Job information of byte stream type from Parser
Output: The merged Flink Plan

```
for job in jobs do
    generate subgraph for job;
    collection.add(job);
    rjobs = jobs.remove(job);
    for j in rjobs do
        if isomorphism(j, job) then
            collection.add(j);
    if collection.size ≥ 2 then
        mergeJob(j, job);
        jobs = jobs.remove(job and collection);
write(new Plan);
```

Heuristic-Based Merging. For jobs with different structures and input data sizes, we propose a heuristic-based merging strategy to find the jobs with the highest "similarity" to merge. Through a large number of experimental results, we selected the jobs with better results after merging, and performed feature extraction on them. The following are some definitions:

- **Definition 2: Job Similarity.** The value obtained by weighting the ratio of the feature parameters between the two jobs extracted, the specific parameters will be introduced later.
- **Definition 3: GlobalOperator.** GlobalOperator refers to operators that need to obtain data from other nodes for processing in a multi-node cluster, such as Join and Reduce.

Table 1. Parameters of job feature

Definition	Formula	Description	Threshold
Task size ratio	$\boldsymbol{F} = \frac{size(m)}{size(n)}$ (1)	The ratio of the total data size processed by two different jobs (m is larger)	[1, 1.8]
DAG depth ratio	$\boldsymbol{D} = \frac{dept(m)}{dept(n)}$ (2)	The ratio of the length of the longest Operator chain in the JobGraph of the two jobs (m is larger)	[1, 2]
GlobalOperator ratio	$\boldsymbol{G} = \frac{gol(m)}{gol(n)}$ (3)	The ratio of the number of GlobalOperator in the two jobs (m is larger)	[1, 1.5]
Parallelism ratio	$\boldsymbol{P} = \frac{parallelism(m)}{parallelism(n)}$ (4)	The ratio of the parallelism of the two jobs (m is larger)	[1, 2]
DAG similarity	$\boldsymbol{S} = \sqrt{\sum_{i=1,j\leq i}^{n}(M_{ij} - N_{ij})^2}$ (5)	Euclidean distance of JobGraph of the two jobs	/

- **Definition 4: LocalOperator.** LocalOperator refers to operators that do not need to obtain data from other nodes for processing, but only process local node data, such as Map and Filter.

Table 1 shows the job feature parameters that need to be extracted and have a greater impact on the efficiency of the merged job execution, Through a large number of experiments, the threshold range of the parameters that meet the combined conditions is estimated. In the case of a small cluster node, the parallelism setting of most jobs adopts the default value, which is equal to the maximum parallelism of Flink. At the same time, we also found that if the parallelism is the same and the thresholds of $\boldsymbol{F}$, $\boldsymbol{D}$, and $\boldsymbol{G}$ are all within the threshold range, the merged jobs can bring satisfactory performance improvement. Since subgraph-based merging has filtered out most similar jobs, the number of jobs to be merged is not large and most of them have larger differences. Therefore, in heuristic-based merging, two jobs are selected for merging.

Algorithm 3 shows the process of heuristic-based merging. First, select one of the submitted jobs to compare with other jobs (lines 1–3). If the jobs' parallelism is the same, merge the two jobs by calculating the threshold (lines 4–5) and selecting the job with the highest score (lines 6–9). Then, perform the same threshold calculation and scoring operations as above in the remaining jobs with different parallelism (lines 10–16). Finally merge the jobs that meet the conditions (line 17) and generate the new Flink Plan (line 18).

4.2 Multi-job Scheduling

Multi-job scheduling is suitable for the large scale of cluster nodes so that the parallelism setting of jobs is less than the maximum parallelism of Flink. Through the scheduling optimization strategy, the system can not only improve operating efficiency by making full use of resources, but also maintain a balanced state of cluster load.

Problem Description. The upper limit of Flink's parallelism is the total number of Slots, which means that the total parallelism of running jobs must be less than or equal to the total number of Slots. When the parallelism of a job to be submitted is greater than the number of remaining Slots, the job will be returned. If the total number of Slots in Flink is n, the parallelism of a job being executed is $0.5n$, and the parallelism of the next job to be executed is $0.6n$, then half of the cluster resources will be idle because the job cannot be submitted. If the parallelism of the job being executed and the job to be submitted are both small, the resource usage of the cluster will be higher. The above situation will make the resource usage of the cluster unstable, which neither guarantees the cluster to run Flink jobs efficiently for long time, nor can it maintain a stable load.

Algorithm 3: Heuristic-based Merging

Input: Job information of byte stream type that does not satisfy the subgraph isomorphism condition
Output: The merged Flink Plan

```
1  for job in jobs do
2      rjobs = jobs.remove(job);
3      {for j in rjobs do
4          if job.parallelism == j.parallelism then
5              calculate(F); calculate(D); calculate(G);
6              if meet the threshold then
7                  score = F × 0.8 + D × 0.5 + G × 0.3;
8      Find j with the largest score;
9      mergeJob(j, job); jobs = jobs.remove(job);
10 for job in jobs do
11     rjobs = jobs.remove(j and job);
12     for j in rjobs do
13         calculate(F); calculate(D); calculate(G); calculate(P);
14         if meet the threshold then
15             calculate(S);
16     Find j with the smallest S;
17     mergeJob(j, job); jobs = jobs.remove(j and job);
18 write(new Plan);
```

Scheduling Based on Maximum Parallelism. Our solution is to give priority to the execution of the job with the highest degree of parallelism that meets the remaining resources of the system, so as to avoid system resource idleness to the greatest extent. In addition, try to make long-running and short-running jobs run at the same time to avoid excessive thread switching caused by intensive short jobs at a certain time. Therefore, the algorithm first extracts the characteristics of each job, and divides it into three groups of long, medium, and short jobs through KMeans clustering algorithm, and finally uses a round-robin scheduling method to submit to the cluster the job with the highest degree of parallelism that meets the execution conditions in each group.

Algorithm 4 describes the process of scheduling optimization. First, extract the features of each job (line 1), including the amount of data, the number of GlobalOperators, the degree of parallelism of each Operator, and the DAG depth. A monitor is placed to monitor whether the job to be executed is empty (line 2). If there are jobs to be submitted, the number of Slot remaining in

Algorithm 4: Multi-job Scheduling Based on Maximum Parallelism

Input: Jobs information of byte stream type from Parser
Output: Flink Plan for the next job to be executed

```
1  Extract the features of each job;
2  while jobs.size ≠ null do
3      slot = the number of remaining Slots;
4      for i = 1 to 3 do
5          job[i] = k_means(features);
6      i = (i++) % 3;
7      for job in job[i] do
8          if job.parallelism > slot then
9              continue;
10         Find the job with maximum parallelism;
11     if job.exsist then
12         job.execute();
13         jobs = jobs.remove(job);
14     continue;
```

the system will be obtained (line 3). According to the extracted job features, the KMeans clustering algorithm is used to divide the job into three groups: long-time running, medium-time running, and short-time running (lines 4–5). Then find a job by group, the job that satisfy the remaining Slot number of the system and have the greatest degree of parallelism is selected (lines 6–10). If such a job exists, it is submitted to the cluster for execution and removed from the queue of jobs to be executed (lines 11–13). Finally, regardless of whether a job is submitted for execution, the search for jobs that can be submitted for execution will continue according to the above criteria (line 14).

5 Evaluation Results

In this section, we describe the performance evaluation of the proposed multi-job merging algorithms and the scheduling optimization algorithm in our framework. The data sets is used to test the running time and the number of Slots occupied.

5.1 Experimental Setup

We run experiments on a 7-nodes OMNISKY cluster (1 JobManager & 6 TaskManagers), and all nodes are connected with 10-Gigabit Ethernet. Each node has two Intel Xeon Silver 4210 CPUs @ 2.20 GHz (10 cores × 2 threads, 20

TaskNumberSlots), 128 GB memory, and 1 TB SSD. Hadoop version 2.7.0 (for storing data on HDFS) and Flink version 1.8.0 are chosen as the experimental environment, and their configuration files are configured according to the hardware environment as mentioned above.

We select three distributed jobs to run the experiment, namely WordCount, Table Join and KMeans, from the perspectives of the type of Operators included in the job and whether the job includes iterative tasks. All running time measurements include the generation time of new JobGraphs. We use a large number of real-world data sets and generated data sets to evaluate the experimental results, and all the experiments are tested more than 10 times. The specific information are as follows:

- **WordCount:** The WordCount job contains almost only LocalOperators, and there is not too much data exchange between TaskManagers. We select the text from the literary work Hamlet as the data set and manually expand it to 500 MB–5 GB.
- **Table Join:** As Join is a GlobalOperator in Flink, it needs to obtain intermediate data from each TaskManager, so a large number of data exchanges between nodes will occur in Table Join job, especially in multi-Table Join. We choose multiple relational tables generated by the big data test benchmark TPC-H, with the size range 500 MB–10 GB.
- **KMeans:** It is a clustering job with iterative tasks in which both LocalOperators and GlobalOperators are iterated. We use the UCI standard data set Wine, and manually expand the number of data samples to reach 500 MB–1 GB.

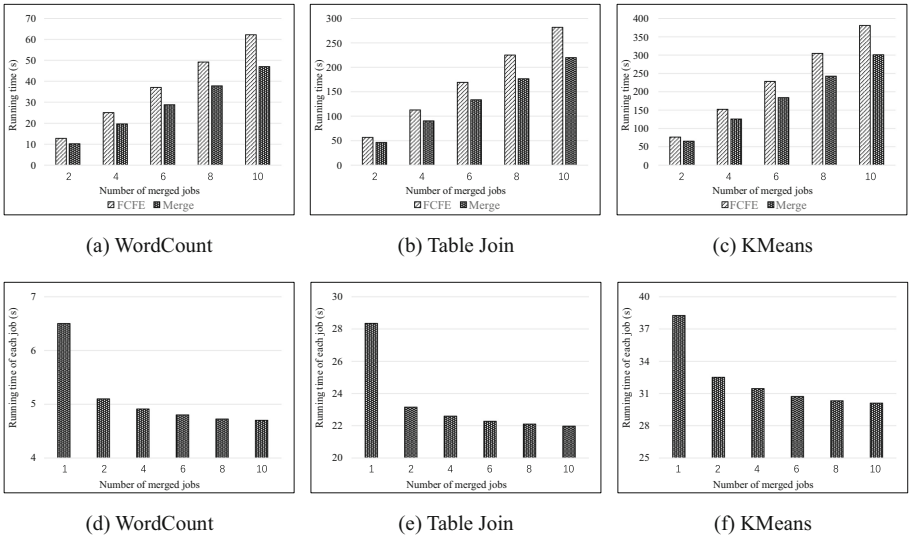
Table 2. Effect of merging two identical jobs on data sets of different sizes

WordCount			Table join			KMeans		
Size	Running time		Size	Running time		Size	Running time	
	FCFE	Merge		FCFE	Merge		FCFE	Merge
500 MB & 500 MB	12.8 s	**10.2 s**	1 GB & 1 GB	56.7 s	**46.3 s**	500 MB & 500 MB	76.5 s	**65.0 s**
1.5 GB & 1.5 GB	33.5 s	**26.6 s**	1.2 GB & 1.2 GB	79.2 s	**65.0 s**	700 MB & 700 MB	121.6 s	**105.5 s**
3 GB & 3 GB	62.6 s	**48.2 s**	1.5 GB & 1.5 GB	122.0 s	**99.8 s**	850 MB & 850 MB	155.7 s	**131.1 s**
5 GB & 5 GB	110.7 s	**84.5 s**	2 GB & 2 GB	213.5 s	**170.8 s**	1 GB & 1 GB	212.4 s	**177.7 s**
500 MB & 5 GB	68.6 s	**61.2 s**	1 GB & 2 GB	132.1 s	**121.8 s**	500 MB & 1 GB	142.7 s	**128.1 s**
3 GB & 5 GB	85.5 s	**72.0 s**	1.5 GB & 2 GB	160.9 s	**136.4 s**	700 MB & 1 GB	165.0 s	**144.7 s**

5.2 Testing of Multi-job Merging

We first test the merging effect of two identical jobs under different scale data sets, and the specific results are shown in Table 2. The first 3 rows are the effects of merging data sets of the same size (sorted by data set size), and the rest are the effects of merging data sets of different sizes. It can be found that the

three types of jobs can bring about efficiency improvement after merging and executing. Among them, WordCount job has better improvement effect than the other two, with the improvement efficiency reaching 31% in the best case, because both Table Join and KMeans have a large number of data exchanges between nodes during execution. In addition, the efficiency of merged jobs with the same size data set is better, because when the data set sizes are different, the smaller job will complete the data processing in advance, and the efficiency improvement brought by sharing resources between the two jobs cannot be maintained for a long time.



(a) WordCount (b) Table Join (c) KMeans

(d) WordCount (e) Table Join (f) KMeans

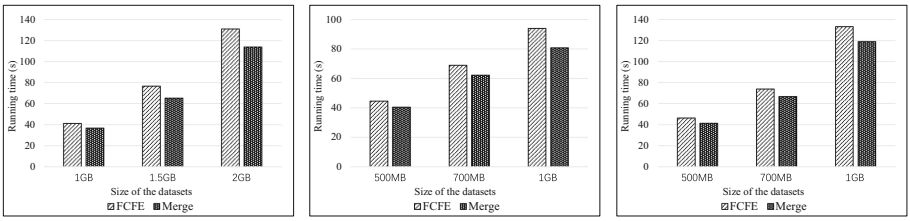
Fig. 5. Effect of the number of merged jobs on efficiency

Next we show the effect of the number of merged jobs on efficiency, which is shown in Fig. 5. We use the same job to measure the experimental results, and we can see that the more jobs are merged, the more obvious the efficiency improvement, because when the number of jobs is large, the extraction of CPU and memory resources will be more sufficient.

For merging different types of jobs, we use data sets of the same size to evaluate efficiency. As shown in Fig. 6, merging different types of jobs can still bring good performance improvement, ranging from 15% to 21%.

5.3 Testing of Scheduling Optimization

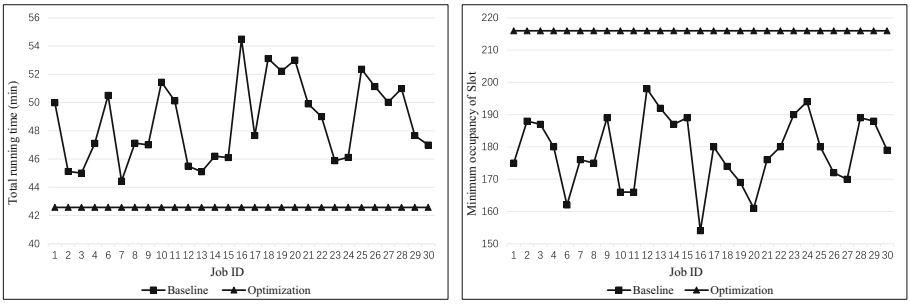
Finally, we show the effect of the multi-job scheduling optimization algorithm. According to the hardware environment described in Sect. 5.1, the maximum parallelism of Flink is set to 240, which is the same as the total number of CPU



(a) WordCount & Table Join (b) WordCount & KMeans (c) Table Join & KMeans

Fig. 6. Effect of merging different jobs

cores in 6 TaskManagers. We generated 10 jobs in WordCount, Table Join and KMeans respectively, and randomly set the degree of parallelism for these 30 jobs, ranging from 50 to 180. Then 30 submissions are randomly generated for these 30 jobs, and the total execution time of these 30 submissions is measured and compared with the scheduling optimization method proposed by us.



(a) Total running time (b) Minimum occupancy of Slot

Fig. 7. Effect of different Job execution sequences

Figure 7(a) shows the running time of each set of jobs by random submission. It can be seen that the total time of executing jobs in the sequence after scheduling optimization is shorter than the total time of executing jobs in the 30 randomly generated sequences, and the performance improvement of 28% can be achieved in the best case. Slot occupancy is shown in Fig. 7(b). Since the Slot usage of the last executed job may be too low due to the job parallelism being set too small, we only measure the Slot usage when the first 25 jobs are executed. It can be found that when jobs are executed in an unoptimized order, Slot usage will be too low for a certain period of time during each execution of the jobs. On average, executing jobs in the optimized order will reduce cluster idle resources by 61%.

6 Conclusion and Discussion

In this paper, we propose the groundbreaking framework that support multi-job merging and scheduling. Based on these two functions, optimization strategies are proposed to improve the efficiency of multi-job by making full use of the cluster resources. In order to verify the effectiveness of the proposed algorithms, we conduct many experiments to prove the superiority of our work. Since Flink is a "unify batch & streaming" system, a particularly interesting direction for future work is to extend our proposed framework and optimization algorithms to the streaming jobs, which can improve the function of the framework.

Acknowledgments. This research was supported by the National Key R&D Program of China under Grant No. 2018YFB1004402; and the NSFC under Grant No. 61872072, 61772124, 61932004, 61732003, and 61729201; and the Fundamental Research Funds for the Central Universities under Grant No. N2016009 and N181605012.

References

1. Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: Proceedings of the International Conference on Data Engineering, pp. 1151–1162 (2011)
2. Carbone, P., et al.: Apache flink: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**, 28–38 (2015)
3. Chakraborty, R., Majumdar, S.: A priority based resource scheduling technique for multitenant storm clusters. In: International Symposium on Performance Evaluation of Computer and Telecommunication Systems, pp. 1–6 (2016)
4. Cheng, D., Rao, J., Jiang, C., Zhou, X.: Resource and deadline-aware job scheduling in dynamic Hadoop clusters. In: IEEE International Parallel and Distributed Processing Symposium, pp. 956–965 (2015)
5. Ciobanu, A., Lommatzsch, A.: Development of a news recommender system based on apache flink, vol. 1609, pp. 606–617 (2016)
6. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**, 1367–1372 (2004)
7. Dean, J., Ghemawat, S.: MapReduce. Commun. ACM **51**(1), 107–113 (2008)
8. Eaman, J., Cafarella, M.J., Christopher, R.: Automatic optimization for MapReduce programs. Proc. VLDB Endow. (2011)
9. Espinosa, C.V., Martin-Martin, E., Riesco, A., Rodriguez-Hortala, J.: FlinkCheck: property-based testing for apache flink. IEEE Access **99**, 1–1 (2019)
10. Falkenthal, M., et al.: OpenTOSCA for the 4th industrial revolution: automating the provisioning of analytics tools based on apache flink, pp. 179–180 (2016)
11. Garca-Gil, D., Ramrez-Gallego, S., Garca, S., Herrera, F.: A comparison on scalability for batch big data processing on apache spark and apache flink. Big Data Anal. **2** (2017)
12. Hueske, F., Krettek, A., Tzoumas, K.: Enabling operator reordering in data flow programs through static code analysis. In: XLDI (2013)
13. Kougka, G., Gounaris, A.: Declarative expression and optimization of data-intensive flows. In: Bellatreche, L., Mohania, M.K. (eds.) DaWaK 2013. LNCS, vol. 8057, pp. 13–25. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40131-2_2

14. Pandey, V., Saini, P.: An energy-efficient greedy MapReduce scheduler for heterogeneous Hadoop YARN cluster. In: Mondal, A., Gupta, H., Srivastava, J., Reddy, P.K., Somayajulu, D.V.L.N. (eds.) BDA 2018. LNCS, vol. 11297, pp. 282–291. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04780-1_19
15. Perera, S., Perera, A., Hakimzadeh, K.: Reproducible experiments for comparing apache flink and apache spark on public clouds. arXiv:1610.04493 (2016)
16. Radhya, S., Khafagy, M.H., Omara, F.A.: Big data multi-query optimisation with apache flink. Int. J. Web Eng. Technol. **13**(1), 78 (2018)
17. Rumi, G., Colella, C., Ardagna, D.: Optimization techniques within the Hadoop eco-system: a survey. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 437–444 (2015)
18. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Optimizing analytic data flows for multiple execution engines. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 829–840 (2012)
19. Tian, H., Zhu, Y., Wu, Y., Bressan, S., Dobbie, G.: Anomaly detection and identification scheme for VM live migration in cloud infrastructure. Future Gener. Comput. Syst. **56**, 736–745 (2016)
20. Tinghui, H., Yuliang, W., Zhen, W., Gengshen, C.: Spark I/O performance optimization based on memory and file sharing mechanism. Comput. Eng. (2017)
21. Wang, K., Khan, M.M.H., Nguyen, N., Gokhale, S.: Design and implementation of an analytical framework for interference aware job scheduling on apache spark platform. Cluster Comput. **22**, 2223–2237 (2019). https://doi.org/10.1007/s10586-017-1466-3
22. Yao, Y., Tai, J., Sheng, B., Mi, N.: LsPS: a job size-based scheduler for efficient task assignments in Hadoop. IEEE Trans. Cloud Comput. **3**, 411–424 (2015)
23. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**, 56–65 (2016)