# A DAG Refactor Based Automatic Execution Optimization Mechanism for Spark

Hang Zhao[1], Yu Rao[1], Donghua Li[1], Jie Tang[1](✉), and Shaoshan Liu[2]

[1] South China University of Technology University, Guangzhou 510641, People's Republic of China
cstangjie@scut.edu.cn
[2] PerceptIn, Fremont, USA

**Abstract.** In today's big data era, traditional disk-based MapReduce big data framework encountered bottlenecks due to its lower memory utilization and inefficient orchestration of complex tasks. With the advantage of fully use memory resources, Spark provides a lot of data manipulate operators and use DAG to express the dependences. Spark split entire job to multi-stage according to DAG and schedule them in a distributed execution environment, which better adapted to the new characteristic of big data processing. However, Spark didn't consider the resource requirement of different operators and schedule them indiscriminately, which could cause load imbalances on different nodes in the cluster and cause some node become bottlenecks due to its extraordinary resource consumption. In the past, solve this problem need developers to have a lot of experience of Spark and write code sophisticated. In this paper, we proposed a DAG refactor based automatic execution optimization mechanism for Spark. The experimental results show that the DAG refactor mechanism can greatly improve Spark performance by up to 8.8X without misinterpretation of original program semantics.

**keywords:** Big data · Spark · Semantic DAG · DAG refactor

## 1 Introduction

With the development of information technology, massive data has been generated every day [1]. Traditional big data processing framework, such as Hadoop, use disk to store intermediate data, always encounter disk I/O bottleneck. Spark use RDD (Resilient Distributed Dataset) to store intermediate data and use Linages to archive fault-tolerate [2], which could archive significant performance improvement compared to Hadoop. Therefore, a lot of applications have implemented in Spark, such as Deep Learning [3], smart city [4, 5], and automatically vehicle.

Spark provides a lot of data manipulate operators and use DAG (Directed Acyclic Graph) to express the dependences, then Spark split entire job to multi-stage according to DAG and schedule them in a distributed execution environment. However, extensive experimentations show that different operators have different running characteristic, while Spark didn't consider the resource requirement of different operators and

© IFIP International Federation for Information Processing 2019
Published by Springer Nature Switzerland AG 2019
X. Tang et al. (Eds.): NPC 2019, LNCS 11783, pp. 338–344, 2019.
https://doi.org/10.1007/978-3-030-30709-7_30

schedule them indiscriminately. In this paper, we proposed a DAG refactor based automatic execution optimization mechanism for Spark. This mechanism could reconstruct the DAG of original program automatically into another structure with higher execution efficiency. With the automatic DAG refactor, the overall system resource utilization can be effectively improved and task execution time can be greatly reduced.

## 2 DAG in Spark

As shown in Fig. 1, in Spark, *DAGScheduler* divides the Job into several stages according to the wide or narrow depends of RDD. The *DAGScheduler* packages each stage into a *TaskSet* and hands it over to *TaskScheduler*, which will dispatch task to *Executor*s. During scheduling, the *SchedulerBackend* is responsible for providing available resources.

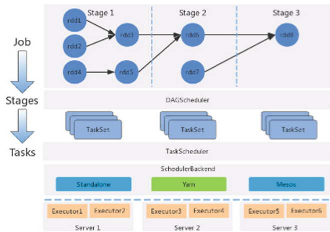


Fig. 1. The architecture of Spark.

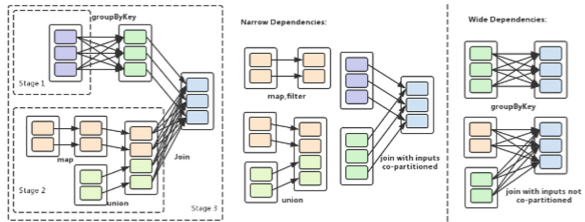


Fig. 2. Wide vs narrow dependencies

Spark provides a rich set of data manipulation operators to build complex processing logic, them can be divided into two categories: (1) Transformation operator, mainly used to describe the conversion relationship between RDD, such as map, filter, and union in the Fig. 2. (2) Action operator, which will trigger Spark to submit job, such as groupByKey and join in Fig. 2.

## 3 Observation on Spark Operator

Spark offers great flexibility to application developers by its rich operators set. However, there still lacks theoretical and experimental research on Spark operators. In this paper, we explore different characteristics of operators in Spark through a large number of experiments and get the observation below: (1) Spark operators can be classified into computation intensive operators and Shuffle intensive operators according to the characteristics of operators. (2) Performance of application varies greatly when different operators contributed to the same semantic. (3) Performance of application varies greatly when execution sequence of operators changes. (4) Data volume decides the execution performance and usage of each operator.

# 4 Automatic DAG Refactor Mechanism

In scheduling, Spark only considers the narrow-dependency or wide-dependency of operators in stages division. It is prone to overlook different resource requirements and runtime feature of operators. Thus resulted schedule decisions can not fully mine the in-memory computing potential. In this paper, we propose a Spark automatic optimization framework based on DAG refactor to take care of such sophisticated work and make execution more efficient automatically.

## 4.1 System Design

The automatic DAG refactor mechanism proposed is shown as Fig. 3. The mechanism can reconstruct DAG by modification of RDD dependency and the user-defined execution function. It mainly includes a general DAG refactor module and an extensible DAG refactor rule library.

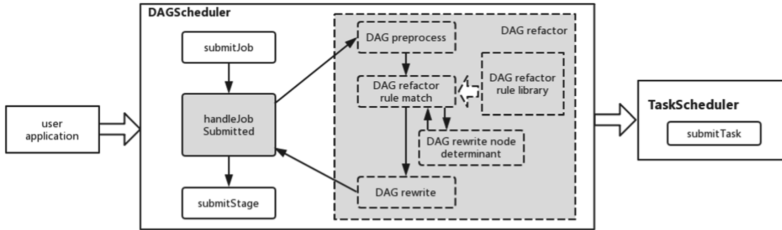


**Fig. 3.** System architecture diagram

## 4.2 DAG Refactor Rules Library

In this paper, we extracted the characteristics of different operators in Spark running process through a large number of experiments. Then, conclude the replacement rules of Spark operator, and form a Spark operator replacement rule library through the analysis of characteristics of Spark operator and semantic analysis of DAG. All replacement rules are shown in Table 1.

**Table 1.** Replacement rules

| Rules | Scope of application |
|---|---|
| map -> mapPartitions | User function overhead too large |
| foreach -> foreachPartitions | User function overhead too large |
| groupByKey + map -> reduceByKey | Shuffle data too large |
| groupByKey + mapPartitions -> reduceByKey | Shuffle data too large |
| reduce -> treeReduce/treeAggregate | Driver side performance bottleneck |
| aggregate -> treeAggregate | Driver side performance bottleneck |
| reduce-side join -> map-side join | Has a broadcastable table |
| map + filter -> filter + map | Data reduction after filter |
| filter -> filter + coalesce | Data skew occurs after Filter |
| union + distinct -> distinct + union + distinct | Very much duplicate data |

# 5 Implementation in Spark

## 5.1 DAG Refactor

The implementation of proposed DAG refactor mechanism mainly by modifying the function *handleJobSubmitted* in the *DAGScheduler* to handle job submission, and the job submitted by the user can be extracted. Then call the *DAGRefactor* component to refactor user job, form a refactored job, and finally replace the original job with refactored job, and then continue to execute by the Spark.

## 5.2 DAGRefactor Design and Implementation

The DAGRefactor class diagram is shown in Fig. 4. Origin_job and refactored_job store the original job and refactored job after refactor respectively; adjacency_table and inverse_adjacency_table are intermediate variables of running process, which are used to store the adjacency table and inverse adjacency table of DAG; rule_list lists definable refactor rules, it is convenient to add more refactor schemes later.

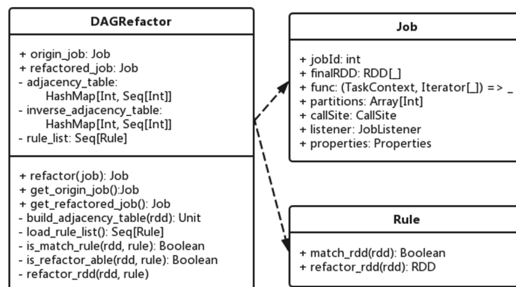


**Fig. 4.** DAGRefactor class diagram

Depending on their functionality, DAGRefactor provides modules such as DAG Analyse, Rules Match, Refactored Check, DAG Rewrite, and DAG Refactor Rules Library, and provides separate function interfaces for different modules.

# 6 Experiments and Evaluation

## 6.1 Evaluation Environment

By running Spark in real environment, the improvement of DAG refactor strategy proposed in this paper can be evaluated and analyzed. Spark is based on its 2.3.0 version and we set 4 Spark worker instances, each have 4 CPU cores and 6 GB memory.

### 6.2 SQL

*Spark-SQL* is a typical scenario of *foreach* operator. First, we made experiment by using *foreach* operation to inserting 1,000,000 rows and 100,00 rows respectively. Next, we empty the database and proposed framework make refactor by using *foreachPartitions* operator. Table 2 shows experimental results, compared with *foreach* implementation, *foreachPartitions* gives 8.8X speedup at best meanwhile consumes less bandwidth, and no data loss.

**Table 2.** Comparison of *foreach* and *foreachPartitions*

| | Executor * core | Time (s) | CPU | Bandwidth | Data loss |
|---|---|---|---|---|---|
| foreach (1000K) | 4 * 4 | 457.494 | 10% | 0–50 Mbps | 79% |
| foreachPartitions (1000K) | 4 * 4 | 52.089 | 5% | 15 Mbps | 0% |
| foreach (100K) | 2 * 1 | 260.193 | 5% | 6 Mbps | 0% |
| foreachPartitions (100K) | 2 * 1 | 54.054 | 5% | 2 Mbps | 0% |

### 6.3 Data Aggregation

Data aggregate summarizes all records of RDDs in two phases. As shown in Fig. 5, 30 s later, the limited computing resources of driver results in slow execution and diver becomes a performance bottleneck, resulting in total execution time of up to 55 s.

Without changing other variables, proposed framework refactor *aggregate* operator into *treeAggregate* operator. As shown in Fig. 6, *treeAggregate* operator adopted a tree-like aggregation strategy in the second phase, thus keeping CPU utilization at a high level consistently. It took only 27 s to complete all tasks and reduced execution time by 51%.

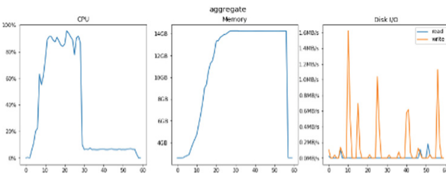


**Fig. 5.** *aggregate* resource consumption

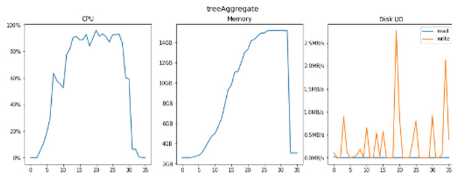


**Fig. 6.** *treeAggregate* resource consumption

### 6.4 Merge and Deduplication

We tested the refactor improvement of merging operations and deduplication operations of two RDDs, i.e. *A.union(B).distinct()* is rewritten to *A.distinct().union(B.distinct()).distinct()*. Figures 7 and 8 shows the resource consumption respectively. It can be seen that by refactor, total running time of program is reduced from 51 s to 47 s, and performance is improved by 7.8%.

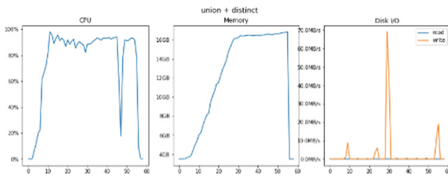


**Fig. 7.** *union* + *distinct* resource consumption

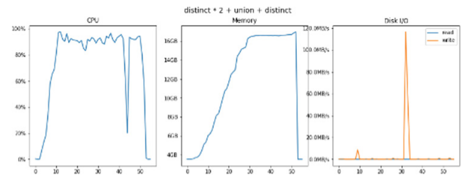


**Fig. 8.** *distinct* + *union* + *distinct* resource consumption

## 7 Relation Works

Spark provides rich operators and uses them to organize computational logic, but research on Spark operators is still relatively rare. [6] studied the input-output ratio of different operators, to estimate the size of intermediate data in computing process. [7] talked about incremental calculation, studied the difference between different operators when data increments. According to the difficulty of operator multiplexing, its divided operators into two types: indirect multiplexing which can be directly multiplexed and deduced by predicate. At the same time, implemented FQ-Tree-based reusable fragment matching and DAG refactor. For scheduling shuffle class operators, [8] analyzed memory scheduling algorithm in Spark Shuffle phase. Considering the un-balanced memory requirements of different task, fair memory allocation scheduling algorithm can not meet the demand well, proposed an adaptive scheduling algorithm that could dynamically adjust the memory allocation of tasks based on overflow historically.

## 8 Conclusions

In this paper, the different characteristics of different operators in Spark are studied by experiment. With this observation, we design and implement a DAG refactor based automatic execution optimization mechanism for Spark. With a large number of experimental analysis of operators in Spark, we summarize several rules for DAG refactor, which can directly optimize the calculation of related operators. Experiments show that the proposed DAG refactor based automatic execution optimization mechanism can improve Spark performance up to 8.8X by DAG refactor without destroying original program semantics.

## References

1. Pempek, T.A., Yermolayeva, Y.A., Calvert, S.L.: College students' social networking experiences on Facebook. J. Appl. Dev. Psychol. **30**(3), 227–238 (2009)
2. Zaharia, M., Chowdhury, M., Das, T., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Usenix Conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)
3. Hamilton, M., Raghunathan, S., Matiach, I., et al.: MMLSpark: Unifying Machine Learning Ecosystems at Massive Scales. arXiv preprint arXiv:1810.08744 (2018)

4. Agafonov, A., Yumaganov, A.: Short-term traffic flow forecasting using a distributed spatial-temporal k nearest neighbors model. In: 2018 IEEE International Conference on Computational Science and Engineering (CSE), pp. 91–98. IEEE (2018)
5. Nasiri, H., Nasehi, S., Goudarzi, M.: A survey of distributed stream processing systems for smart city data analytics. In: Proceedings of the International Conference on Smart Cities and Internet of Things, p. 12. ACM (2018)
6. Bae, J., Jang, H., Jin, W., et al.: Jointly optimizing task granularity and concurrency for in-memory mapreduce frameworks. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 130–140. IEEE (2017)
7. KanJing: The research of key techniques of incremental computing for DAG-based framework. Beijing University of Technology (2017)
8. Chen, Y.: Analysis and optimization of memory scheduling algorithm of spark shuffle. Zhejiang University (2016)