# Resource Efficiency Optimization for Big Data Mining Algorithms under Multi-MapReduce Job Coordination: Postprint

**Authors:** Liao Bin, Zhang Tao, Yu Jiong, Huang Jinglai, Guo Binglei, Liu Yan

**Date:** 2019-04-01T00:00:00+00:00

## Abstract

Since any MapReduce job must independently perform a series of complex operations such as task scheduling and resource allocation, this results in substantial redundant disk I/O and repeated resource allocation operations among multiple collaborative MapReduce jobs within the same algorithm, leading to low resource utilization efficiency during computation. Big data mining algorithms are typically partitioned into multiple collaborative MapReduce jobs; taking the ItemBased algorithm as an example, this paper analyzes the resource efficiency issues in big data mining algorithms under multi-MapReduce job collaboration, and proposes a DistributedCache-based ItemBased algorithm that utilizes DistributedCache to cache I/O data among multiple MapReduce Jobs, thereby breaking the independence limitations between jobs and reducing waiting latency between Map and Reduce tasks. Experimental results demonstrate that DistributedCache can improve data reading speed for MapReduce jobs, and the algorithm reconstructed using DistributedCache significantly reduces waiting latency between Map and Reduce tasks, achieving more than a threefold improvement in resource efficiency.

## Full Text

## Preamble

### Resource Efficiency Optimization for Big Data Mining Algorithms in Multi-MapReduce Collaboration Scenarios

Liao Bin[1]†, Zhang Tao[2], Yu Jiong[2], Huang Jinglai[1], Guo Binglei[2], Liu Yan[3]
(1. College of Statistics & Information, Xinjiang University of Finance & Economics, Urumqi 830012, China;
2. School of Information Science & Engineering, Xinjiang University, Urumqi

830008, China;
3. School of Software, Tsinghua University, Beijing 100084, China)

**Abstract:** Since any MapReduce job must independently perform a series of complex operations such as task scheduling and resource allocation, multiple MapReduce jobs coordinated by the same algorithm exhibit substantial redundant disk I/O and repeated resource application operations, leading to inefficient resource utilization during computation. Big data mining algorithms are typically decomposed into multiple collaborative MapReduce jobs. Taking the ItemBased algorithm as an example, this paper analyzes the resource efficiency issues in big data mining algorithms under multi-MapReduce job collaboration. We propose a DistributedCache-based ItemBased algorithm that leverages DistributedCache to cache I/O data between multiple MapReduce jobs, breaking the isolation barrier between jobs and reducing waiting latency between Map and Reduce tasks. Experimental results demonstrate that DistributedCache can improve data reading speed for MapReduce jobs, and the reconstructed algorithm significantly reduces waiting latency between Map and Reduce tasks, achieving over threefold improvement in resource efficiency.

**Keywords:** MapReduce optimization; ItemBased algorithm; memory file system; I/O efficiency; resource optimization

---

## 0 Introduction

According to a 2015 report from IDC (Internet Data Center) [**?**], the total data generated worldwide in 2015 approached 10 ZB, with projections reaching 44 ZB by 2020. This explosive data growth presents both opportunities and challenges for the IT industry, as data generation has shifted from passive to active modes, marking the advent of the big data era. The scale effect of big data has led to continuously rising costs in storage, computation, analysis, and management, making high-efficiency, low-cost big data computing technologies a focal point for academic and industrial research.

Since Google published papers on the distributed storage system GFS [**?**] and the MapReduce computation model [**?**] in 2003, MapReduce has gradually become the universal underlying computation framework for Hadoop, Spark, Pig, Hbase, Hive, and other big data systems. Compared to previous parallel computation models (e.g., PRAM, MPI), MapReduce fundamentally differs by adhering to the "move computation to data" principle rather than "move data to computation." The core essence of this principle involves scheduling computation programs to nodes closest to the data (locally) to reduce data transmission during data-intensive job execution. However, when a MapReduce job is decomposed into multiple tasks by the scheduling system, these tasks are not isolated entities. Their collaborative execution requires extensive disk I/O and network transmission operations (the MapReduce task execution flow is shown in Figure 1 [Figure 1: see original paper]), such as Split, RecordReader, Partition,

and Shuffler operations, all of which involve transferring intermediate results across machines and storing them on disk as input for subsequent pipeline operations. Particularly for big data mining algorithms implemented based on the MapReduce model, their high complexity typically necessitates collaboration among multiple MapReduce jobs. For instance, the PageRank algorithm under MapReduce is decomposed into four jobs, while the common Bayes algorithm is split into ten jobs. However, resource scheduling and management among these multiple MapReduce jobs under the same algorithm are not coordinated, resulting in severe redundant disk I/O and repeated resource application operations that cause low algorithm resource utilization efficiency \cite{4~6}.

Recommendation algorithms represent one of the most widely applied categories of big data mining algorithms. This paper investigates the resource efficiency issues among multiple MapReduce jobs in big data mining algorithms, using the ItemBased collaborative filtering algorithm as a case study. Our contributions include: (1) analyzing the resource efficiency problems of the ItemBased algorithm in MapReduce environments; (2) proposing a unified caching mechanism for I/O data among multiple MapReduce jobs coordinated by the same algorithm to reduce redundant I/O operations and optimize resource utilization efficiency; and (3) validating the resource efficiency improvements through comparative experiments.

---

## 1 Related Research

Most research on big data mining algorithms, particularly recommendation algorithms, targets improving recommendation quality in specific application scenarios (e.g., e-commerce, news recommendation, video recommendation). Primary evaluation metrics include Precision, Recall, F-Measure, E-Measure, and Average Precision. Few studies address execution environment optimization or resource utilization efficiency. However, as data volumes grow exponentially and the MapReduce parallel computation model becomes increasingly prevalent, algorithm development has shifted from single-machine to distributed models, drawing growing attention to computational and resource efficiency from both academia and industry.

References [**?**, **?**] attempted to leverage MapReduce parallelism to improve the computational efficiency of User-Based collaborative filtering algorithms, implementing them on Hadoop and further enhancing efficiency by flexibly configuring the number of map and reduce tasks. Schelter et al. [**?**] addressed the poor scalability of similarity-based neighborhood collaborative filtering algorithms when facing massive data growth by reimplementing the algorithm based on MapReduce and validating computational efficiency improvements through tests on 700 million Yahoo music data records. Reference [**?**] introduced MinHash clustering, Probabilistic Latent Semantic Indexing (PLSI), and Covisitation counting techniques into Google News' collaborative filtering algorithm,

where MinHash probabilistic clustering applies to scenarios with low precision requirements, and further improved computational speed using MapReduce and Bigtable technologies. Reference [**?**] divided the core computation steps of collaborative filtering algorithms into four MapReduce jobs and proposed reducing data transmission during algorithm execution through data partitioning strategies, demonstrating effective improvement in network resource utilization efficiency for ItemBased recommendation algorithms. However, whether UserBased or ItemBased collaborative filtering algorithms, MapReduce-based implementations require multiple collaborative jobs with substantial redundant I/O and repeated resource application operations between jobs, leaving significant room for resource utilization optimization.

Current approaches to address the low resource efficiency of MapReduce data mining algorithms involve platform migration, such as moving algorithms from Hadoop to Spark [**?**] to leverage Spark' s in-memory and iterative computation advantages for efficiency gains. While effective, platform migration presents two major issues: (a) **Migration Cost:** Moving algorithms from Hadoop to Spark requires rewriting the entire business code according to new APIs and syntax due to the language shift from Java to Scala [**?**], imposing high learning and development costs on R&D personnel and significant system migration and deployment costs on administrators. (b) **Stability Issues:** After platform migration, the new platform' s lack of long-term stability testing can affect the quality of service for upper-layer applications.

In contrast to previous work, this paper: (1) analyzes resource efficiency defects of the ItemBased recommendation algorithm in MapReduce platforms; (2) proposes unified caching of I/O data among multiple MapReduce jobs coordinated by the same algorithm to reduce redundant I/O operations and optimize resource utilization efficiency; and (3) unlike platform migration solutions, avoids redeveloping business code, saving migration and deployment costs while ensuring system stability.

---

## 2 Resource Efficiency Defects in Multi-MapReduce Job Collaboration

Since item-to-item similarities are relatively stable, the ItemBased recommendation algorithm is more suitable for offline computation and more widely applied than other algorithms (e.g., popularity-based, content-based, model-based algorithms). The core idea of ItemBased algorithms is "like attracts like," assuming that items of interest to users must be similar to items they have rated highly. The algorithm first calculates user preferences for items, then computes item similarities based on these preferences, and finally identifies the top-N items most similar to each item. Table 1 shows the input parameters for the ItemBased recommendation algorithm in the Mahout machine learning library on the Hadoop platform.

The ItemBased algorithm primarily consists of four stages: (a) preparing User-Item and Item-Item matrices; (b) calculating similarities between User-Item and Item-Item matrices; (c) partialMultiply—aggregating the similarity matrix by identical items as keys to prepare for subsequent matrix multiplication; and (d) computing recommendation vectors. These four steps can be further decomposed into nine MapReduce jobs, with specific job stages and execution sequences shown in Table 2 .

A single MapReduce job is decomposed into multiple map and reduce tasks by the task scheduler. Based on input parameter settings, input data is read from HDFS or other file systems and fed into corresponding Map tasks. After Map computation completes, data is sent to designated Reduce tasks through Shuffle and Sort operations on <K,V> key-value pairs, with Reduce tasks receiving input in the format <key, Iterator>. Finally, Reduce results are written to the file path specified by the output parameter. Due to the independence of MapReduce jobs—where any job must independently perform task scheduling, resource allocation, HDFS data reading, computation, data shuffling, and result output—a single ItemBased recommendation algorithm execution requires up to 18 HDFS read and write operations. These high-frequency data I/O operations reduce MapReduce cluster resource utilization and degrade overall algorithm performance. Moreover, since disk I/O represents the performance bottleneck in MapReduce clusters, high-frequency disk I/O resource application and release operations create resource contention, causing waiting phenomena between Map and Reduce tasks that further reduce computational efficiency. This analysis demonstrates that optimizing ItemBased algorithm performance in MapReduce environments cannot focus solely on the algorithm itself; the redundant disk I/O and repeated resource application operations caused by job independence are the primary factors degrading resource utilization and execution efficiency.

---

### 2.1 ItemBased Algorithm MapReduce Job Resource Optimization

**2.1.1 Efficiency Analysis of Multi-Job ItemBased Algorithm Execution** As shown in Table 2, the ItemBased recommendation algorithm splits a single computation into four stages—PreparePreferenceMatrixJob, RowSimilarityJob, partialMultiply, and RecommenderJob—comprising nine MapReduce jobs including ItemIDIndexMapper-Reducer.

Let the total execution time of an algorithm on a MapReduce platform be T, with the computation process divided into N MapReduce jobs, where each job' s completion time is T_i. When one MapReduce job completes, resource preparation is required before the next job executes. Let the resource preparation time between jobs i and i+1 be P_i. The algorithm' s total completion time is the sum of map, shuffler, and reduce execution times.

From equations (1) to (3), the total algorithm execution time can be derived as shown in equation (4). In equation (4), the map, shuffler, and reduce sub-stage

execution times are closely related to factors such as task quantity, node computational capacity, resource status, and data volume. However, with identical task quantities, data volumes, and node capacities, resource status becomes the primary factor affecting job execution time. Due to independence among different MapReduce jobs, even jobs belonging to the same algorithm suffer from severe disk I/O redundancy and repeated resource application operations during computation, severely reducing MapReduce algorithm resource utilization efficiency. According to equation (4), theoretically improving resource utilization efficiency (collaboration efficiency) between MapReduce jobs and breaking job independence can enhance computational efficiency for complex MapReduce algorithms.

**2.2 Multi-MapReduce Job Resource Optimization Methods** The defect analysis in Section 2 and theoretical analysis in Section 3.1 demonstrate that improving resource collaboration efficiency between MapReduce jobs and eliminating repeated resource operations caused by job independence can enhance resource utilization. Two primary optimization approaches exist: (a) utilizing the built-in DistributedCache mechanism in the MapReduce framework to break resource sharing barriers between jobs, where the caching system distributes shared data to each compute node' s memory to improve computational efficiency; and (b) introducing a memory-centric virtual distributed storage system above disk-based distributed files (e.g., HDFS) to load shared data resources from HDFS into distributed shared memory, thereby reducing disk I/O invocations.

Method (a) is algorithm-specific and highly flexible but requires partial code refactoring and deployment modifications. Method (b) deploys a new distributed in-memory file system into production, potentially impacting overall system stability but eliminating algorithm code modifications. For the individual ItemBased algorithm, this paper adopts method (a) to improve resource utilization efficiency. Algorithm 1 presents the DistributedCache-based ItemBased algorithm steps.

**Algorithm 1: DistributedCache-Based ItemBased Algorithm**
**INPUT:** Parameter1: <userID, itemID, score> user rating matrix, Parameter2: InputURL.
**OUTPUT:** Parameter1: <userID, itemIDS> recommendation results, Parameter2: OutputURL.

1. For i = 0 to readData(InputURL).blocksize-1 do
2. block = MEMORY.load(InputURL) // Read file block
3. DistributedCache.addCacheFile(block) // Load file block into memory
4. End for

5. For each job in List list
6. MapReduce.start().getJob(list)
7. context.setCacheFiles(PreparePreferenceMatrixJob)
8. context.getCacheFiles()
9. End for
10. For each job in List list
11. MapReduce.start().getJob(list)
12. context.setCacheFiles(RowSimilarityJob)
13. context.getCacheFiles()
14. End for
15. MapReduce.start().getJob(partialMultiply)
16. context.setCacheFiles(partialMultiply)
17. context.getCacheFiles()
18. context.setCacheFiles(PartialMultiplyMapper-Reducer)
19. context.getCacheFiles()
20. MapReduce.start().getJob(PartialMultiplyMapper-Reducer)
21. Return Job.addCacheFile(new File( “OutputURL” ))

Lines 1-4 load input data into distributed cache based on the InputURL parameter. Lines 5-9 execute all MapReduce tasks in the PreparePreferenceMatrixJob stage, enabling in-memory data sharing among the ItemIDIndexMapper-Reducer, ToItemPrefsMapper-Reducer, and ToItemVectorsMapper-Reducer jobs. Lines 10-14 execute all MapReduce tasks in the RowSimilarityJob stage, where DistributedCache enables in-memory data sharing among the CountObservationsMapper-Reducer, VectorNormMapper-Reducer, CooccurrencesMapper-Reducer, and UnsymmetrifyMapper-Reducer jobs. Since the partialMultiply and RecommenderJob stages consist of single MapReduce jobs, lines 15-20 implement in-memory read/write operations for the partialMultiply and PartialMultiplyMapper-Reducer jobs. Finally, line 21 writes the algorithm’ s computation results to memory-level files.

## 3 Experimental Evaluation

### 3.1 Experimental Environment Configuration

To compare the execution efficiency and resource utilization of the ItemBased algorithm under different test environments, we designed two experiments: Experiment 1 tests data throughput under native MapReduce and DistributedCache in-memory caching conditions; Experiment 2 compares the ItemBased recommendation algorithm performance between native MapReduce and DistributedCache in-memory caching conditions. The experimental cluster comprises 11 nodes: 1 master node and 10 worker nodes. The cluster environment is described in Table 3 .

### 3.2 Experiment 1: MapReduce Job Data Throughput Comparison Test

To test the data throughput difference between native MapReduce and DistributedCache in-memory caching, this experiment concurrently executed 10 MapReduce jobs reading text files, thereby constraining the performance bottleneck to I/O operations. The experimental dataset consisted of 40 GB text files. HDFS and DistributedCache block sizes were configured at two different settings: 512 MB and 1 GB. HDFS block distribution followed the default rack-aware policy. Three test runs were conducted for each configuration.

**Table 4** shows the test results for 512 MB block size. When block size was increased from 512 MB to 1 GB by configuring dfs.block.size in HDFS and the corresponding parameter in DistributedCache, the test results are shown in **Table 5** .

The results in Tables 4 and 5 demonstrate that DistributedCache improves MapReduce job data reading speed compared to native HDFS across all metrics: total application completion time, total job completion time, average map task completion time, and shuffler running time. Notably, comparing Tables 4 and 5 reveals that the 1 GB block size achieves greater efficiency improvement than 512 MB. This occurs because larger block sizes reduce the number of Map tasks for a fixed dataset size, decreasing resource contention pressure in clusters with fixed resource quantities and consequently increasing Map task reading speed. Thus, larger block sizes yield more pronounced performance improvements.

### 3.3 Experiment 2: ItemBased Algorithm Performance Comparison

The ItemBased recommendation algorithm implementation used in experiments was sourced from Mahout version 0.11.1, with the algorithm entry class being `org.apache.mahout.cf.taste.hadoop.item.RecommenderJob`. The command for running with native HDFS as the storage target was:

```
./hadoop jar /home/hadoop/mahout0.11.1/mahout-examples-0.11.1-job.jar \
org.apache.mahout.cf.taste.hadoop.item.RecommenderJob \
-i /mahout/itemcf/inputdata \
-o /mahout/itemcf/result \
-s SIMILARITY_LOGLIKELIHOOD \
--tempDir /mahout/itemcf/temp1
```

When using DistributedCache as the storage system, the ItemBased algorithm was refactored according to Algorithm 1. The refactored execution command differs from HDFS:

```
./hadoop jar /home/hadoop/mahout0.11.1/mahout-examples-0.11.1-job.jar \
org.apache.mahout.cf.taste.hadoop.item.RecommenderJob \
-i distributedCache://ubuntu201:23456/ratings/ratings.data \
-o distributedCache://ubuntu201:23456/ratings/result \
-s SIMILARITY_LOGLIKELIHOOD \
--tempDir distributedCache://ubuntu201:19998/ratings/temp
```

The experiments used a user rating matrix containing 20 million <userID, itemID, score> tuples. Three experimental runs were averaged to produce the HDFS versus DistributedCache resource efficiency comparison in **Table 6** .

As shown in Table 6, when using HDFS, the algorithm' s total resource waiting latency was 181 seconds; with DistributedCache, this decreased to 39 seconds. The DistributedCache-refactored algorithm dramatically improved job resource utilization efficiency, reducing total resource waiting latency from 181 seconds to 39 seconds—a 364.1% improvement in resource efficiency.

**Figure 2 [Figure 2: see original paper]** and **Figure 3 [Figure 3: see original paper]** illustrate disk I/O usage under different storage conditions. Figure 2 shows disk I/O when using HDFS, while Figure 3 shows usage with DistributedCache. Comparing the figures reveals that DistributedCache maintains nearly idle disk I/O during job execution, with high disk I/O consumption occurring only when writing final results to HDFS. This validates the significant reduction in resource preparation time observed experimentally. By replacing intensive disk I/O with fast in-memory I/O, DistributedCache optimizes resource utilization efficiency during job execution, substantially reduces resource preparation time, and improves overall operational efficiency.

---

## 4 Conclusion

As MapReduce becomes the standard for parallel computing, resource utilization efficiency issues persist. Particularly for complex big data mining algorithms requiring multiple collaborative MapReduce jobs, severe redundant disk I/O and repeated resource application operations result in poor execution time, resource utilization, and energy efficiency. This paper identifies MapReduce job independence as the root cause, creating severe disk I/O redundancy and repeated

resource application operations among jobs of the same algorithm, thereby degrading execution efficiency.

We analyzed resource efficiency defects of the ItemBased algorithm in MapReduce environments and proposed using DistributedCache to cache intermediate data during computation, reducing redundant I/O operations and optimizing resource utilization efficiency. Two experimental groups demonstrated that DistributedCache replaces intensive disk I/O with fast in-memory I/O, optimizing resource utilization efficiency, substantially reducing resource preparation time, and improving overall resource efficiency by over threefold.

Future work will investigate DistributedCache resource usage and scheduling when cluster load pressure is high, specifically when multiple big data mining algorithms run concurrently in the cluster.

---

## References

[1] The digital universe in 2020: big data, bigger digital shadows, and biggest growth in the far east [EB/OL]. [2018-3-15]. http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf.

[2] Ghemawat S, Gobioff H, Leung S T. The google file system [C]// Proc of the 19th ACM Symposium on Operating System Principles. New York: ACM Press, 2003: 29-43.

[3] Chen C, Lin J, Kuo S. MapReduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems [J]. IEEE Trans on Cloud Computing, 2018, 6 (1): 127-140.

[4] Liao Bin, Zhang Tao, Yu Jiong, et al. Energy consumption modeling and optimization analysis for MapReduce [J]. Journal of Computer Research and Development, 2016, 53 (9): 2107-2131.

[5] Wu Qian, Wang Linping, Luo Xiangzhou, et al. Top-k high utility pattern mining algorithm based on MapReduce [J]. Application Research of Computers, 2017, 34 (10): 2897-2900.

[6] Liao Bin, Zhang Tao, Yu Jiong, et al. Temperature aware energy-efficient task scheduling strategies for mapreduce [J]. Journal on Communications, 2016, 37 (1): 61-75.

[7] Zhao Zhidan, Shang Mingsheng. User-based collaborative-filtering recommendation algorithms on Hadoop [C]// Proc of International Conference on Knowledge Discovery and Data Mining. Piscataway, NJ: IEEE Press, 2010: 478-481.

[8] Ma M M, Wang S P. Research of user-based collaborative filtering recommendation algorithm based on Hadoop [C]// Proc of International Conference

on Computer Information Systems and Industrial Applications. New York: Atlantis, 2015: 63-66.

[9] Schelter S, Boden C, Markl V. Scalable similarity-based neighborhood methods with MapReduce [C]// Proc of ACM Conference on Recommender Systems. New York: ACM Press, 2012: 163-170.

[10] Das A S, Datar M, Garg A, et al. Google news personalization: scalable online collaborative filtering [C]// Proc of International Conference on World Wide Web. New York: ACM Press, 2007: 271-280.

[11] Jiang J, Lu J, Zhang G, et al. Scaling-Up Item-Based Collaborative Filtering Recommendation Algorithm Based on Hadoop [C]// Proc of IEEE World Congress on Services. Piscataway, NJ: IEEE Press, 2011.

[12] Liao Bin, Zhang Tao, Guo Binglei, et al. Performance optimization of Item-Based recommendation algorithm based on Spark [J]. Journal of Computer Applications, 2017, 37 (7): 1900-1905.

*Note: Figure translations are in progress. See original paper for figures.*