Micro Spark

Xi Han, Wei Fang, Yi Xu
1024Spark
{xhan14, wfang2, yxu66}@dons.usfca.edu
May 19, 2015

Abstract

We implement a micro Spark that can act almost the same as real Spark. Our micro Spark can serve user script and support interactive shell mode. We borrow the framework and the abstraction called resilient distributed datasets (RDD) from Spark. An RDD is a read-only collection of objects. Operations are composed of RDD transformation and action. By analyzing the RDD lineage, a directed acyclic graph (DAG) is generated to describe the relation of stages. In addition, fault tolerance is achieved and we introduce a new API called TopByKey.

1 Introduction

Distributed computing frameworks on clusters of unreliable machines has become popular that automatically provide locality-aware scheduling, fault tolerance and load balancing. MapReduce [1] and Spark [2] are widely adopted for large-scale data analytics. These systems allow user to write scripts to execute parallel computation without worry about work distribution and fault tolerance. In this paper, we focus on the implementation of Spark prototype that working in the similar framework and behavior with Spark. RDD achieves fault tolerance through the notion of lineage and stage. In the end of stage, result is written to reliable disk. If a slave is lost in the stage, our system reassigns this task to another slave.

This paper is organized as follows. Section 2 describes implementation of Micro Spark RDDs, DAG Scheduler, Stage, Task Scheduler. Section 3 shows some example jobs. Section 4 describes our extension TopByKey. Section 5 presents some benchmarks. Section 6 makes some concluding remarks and gives directions for future work.

2 Implementation

2.1 RDD

There are two kinds of RDDs: transformation and action. Because computation in Micro Spark is lazy. Our system will commit no computation when going through transformation RDDs until it encounters a action RDD.

Transformations	map(mapFunc) filter(filterFunc) flatMap(flatMapFunc) groupByKey(partitioner) reduceByKey(partitioner) partialSortByKey(key, reverse) mapValues(func) join(RDD, partitioner)
Action	collect() count() topByKey(n, key, reverse)

Table 1. Transformations and actions available on RDDs in Micro Spark

2.2 DAG Scheduler

Whenever a user runs an action (e.g. collect), the scheduler examines that RDD's lineage graph to build a DAG of stages to execute, as illustrated in Figure 1. Boxes with solid outlines are RDDs. Partitions are shaded rectangles. There are two kinds of dependencies between parent RDD and child RDD: narrow dependency and wide dependency. Dependency is decided by the flow of data. If data from one partition of parent RDD only goes to one child partition. The dependency is narrow. In Figure 1, map and union belong to narrow dependency. GroupBy and join belong to wide dependency. DAG analyzer always scans the graph from the end of RDD and starts a new stage. Whenever it

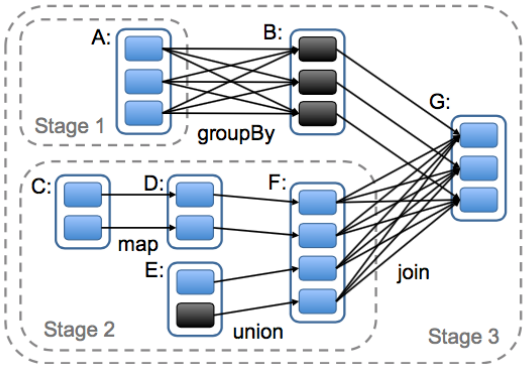

Figure 1. Example of DAG of stages

encounters a wide dependency, end the existing stage and start a new stage.

If all RDDs have been visited, we get a stages graph. The stages are organized in a list. Task scheduler will follow this list to submit stages. In the end of stage, Micro Spark writes result to a reliable disk. Name format of intermediate file is `shuffle_shuffleID_partitionID_targetPartitionID`. The child partition can follow this rule to find the desired input data. This approach simplifies the process, and recompute less when there is a failure in the previous stage. There is no way to have failure in parent stage affect the child stage. However, saving intermediate data to disk costs more time than keeping data in memory. This can be improved by caching the result in the final RDD of previous stage. This is a tradeoff between recovery time and process time.

2.3 Task Scheduler

One RDD contains certain number of partitions. The number of partitions can be defined by user in RDD API parameter. The source of data is textFile RDD. First, we split a file or files by byte to partitions. Second, each partition reads data and always discard the content before first newline in this partition (except the first partition) and fetch the content before first newline in the next partition. Data flows in one partition of the stage. But in the end of stage, or in the other words wide dependency, data is shuffled by HashPartitioner. Finally write the shuffle data to intermediate files. In Figure 2, one narrow partition chain is converted to a task and assigned to one worker. Task information assigned to the worker includes 1. RDDs in this stage; 2. ShuffleId; 3. Partitioner that next shuffled RDD will use; 4. User functions. If there is no available worker, our system will create new process in driver that can ssh to other worker and create a worker process in worker. The available workers list is in slaves file.

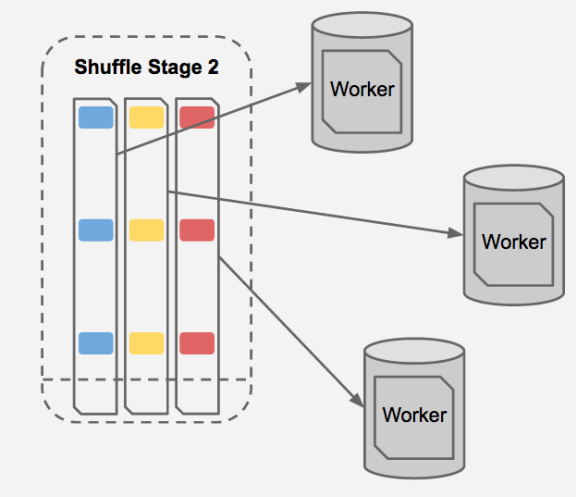

Figure 2. Task Scheduler

For fault tolerance part in stage, if the worker fails in which one task is running, driver can detect the failure and reassign this task to another worker.

3 Example

We now show some sample Micro Spark programs.

3.1 Wordcount

Suppose that we wish to count the number of words in this document. This can be implemented by starting with a file dataset object as follows:

Listing 1 Word Count

```
lines = spark.textFile("wordcount.txt", 4)
words = lines.flatMap(lambda line: line.split(" "))
wordDict = words.map(lambda word: (word, 1))
counts = wordDict.reduceByKey(HashPartitioner(5))
print spark.collect(counts)
```

We first create a distributed dataset called lines that represents the file as a collection of lines. We transform this dataset to create the set of words, and then map each word to 1 and add up these ones using reduce. Because words and wordDict are lazy RDDs, they are never materialized. Until calling collect, driver starts to execute DAG scheduler and task scheduler and assign task to worker. Each worker does textFile, flatMap, Map operations to this partition data. After shuffling and reducing the data, worker returns the result to the driver. The driver appends and displays the result to the user.

3.2 PageRank

Suppose that we wish to do PageRank algorithm.

Listing 2 Page Rank

```
def computeContribs(urls, rank):
    """Calculates URL contributions to the rank of other URLs."""
    num_urls = len(urls)
    for url in urls:
        yield (url, rank / num_urls)

def parseNeighbors(urls):
    """Parses a urls pair string into urls pair."""
    parts = urls.split(' ')
    return parts[0], parts[1]

print "Start page ranking"
lines = spark.textFile("pagerank.txt")
links = lines.map(lambda urls: parseNeighbors(urls))
                        .groupByKey(HashPartitioner())
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))

# Calculates and updates URL ranks
# continuously using PageRank algorithm.
contribs = links.join(ranks, HashPartitioner())
                .flatMap(lambda url_urls_rank:
                computeContribs(url_urls_rank[1][0]
                , url_urls_rank[1][1]))
ranks = contribs.reduceByKey(HashPartitioner())
                .mapValues(lambda rank: rank * 0.85 + 0.15)

for iteration in range(10):
    # Calculates URL contributions to the rank of other URLs.
    contribs = links.join(ranks, HashPartitioner())
    .flatMap(lambda url_urls_rank:
    computeContribs(url_urls_rank[1][0], url_urls_rank[1][1]))
    #Re-calculates URL ranks based on neighbor contributions.
    ranks = contribs.reduceByKey(HashPartitioner())
                    .mapValues(lambda rank: rank * 0.85 + 0.15)

# Collects all URL ranks and dump them to console.
for (link, rank) in spark.collect(ranks):
    print("%s has rank: %s." % (link, rank))
```

ComputeContribs and parseNeighbors are helper functions. The algorithm is executed as follows:

1. Read data from file
2. Map the data to links pair and rank pair
3. Set the initial weight to 1
4. Calculates URL contributions
5. Recalculates URL ranks based on neighbor contributions
6. Continuously using PageRank algorithm
7. Collect the result

In the script, join, groupByKey and reduceByKey are ShuffledRDDs with wide dependency. Other RDDs are MappedRDD with narrow dependency.

3.3 TopByKey

Suppose that we wish to do TopByKey. More details about TopByKey would be introduced in Section 4.

Listing 3 TopByKey

```
data = spark.textFile("sort.txt", 4)
words = data.flatMap(lambda line: line.split(" "))
print spark.topByKey(words, 5, lambda x:x.split(",")[1], true)
```

4 Extension

Our extension in Micro Spark is a new API, topByKey. We think of a scenario we only care about the top n result. But Spark doesn't have topByKey interface. Our topByKey can save amounts of time in sorting. Because we only maintain a fix-sized(n) heap for each partition. Those heaps are ordered by user-defined key function. In the end, we collect result from these heaps, sort the n*num_partition data, and finally get the top n result.

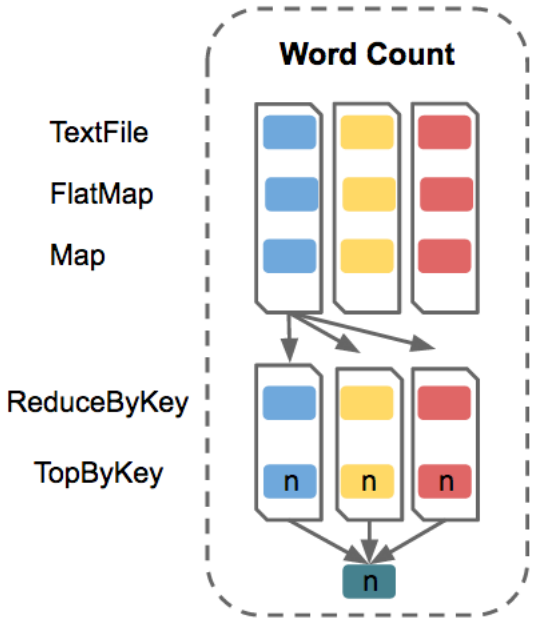

Figure 3. TopByKey

5 Benchmark

In order to show the advantage of topByKey, we implement the partial sorting interface. That means we keep data in order only for one partition. This should be faster than

complete sorting. We can see from Figure **??**, our topByKey can outperform partial sort by 4x in getting the top 10 words in alphabetical order. In order to remove initialization time, we start a simple sort in the beginning of script and then run the topByKey and partial sorting.

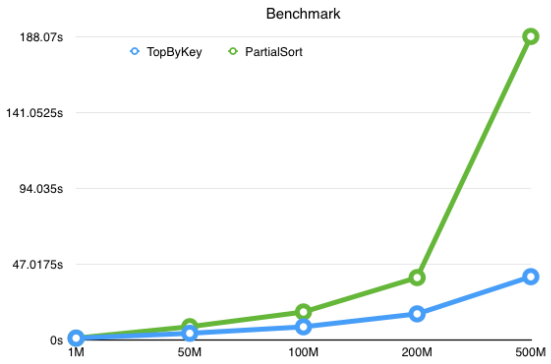

Figure 4. Benchmark

6 Conclusions

This paper presents a general mechanism of Micro Spark which achieves lazy evaluation, work distribution and fault tolerance. We provide sufficient APIs for wordcount, PageRank and interactive queries on log files. In addition, two submit ways are the same as Spark which are spark-submit and spark-shell. Our extension shows better performance over the original sorting and take(n).

In future work, we can focus on these areas:

1. Implement more APIs such as real sortByKey, cache and range partitioner
2. Improve process speed by caching result in memory of worker between stages

References

[1] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.