Spark Partition Strategy Based on Genetic Mutation Algorithm

Shuo Wang(✉)

Beijing University of Posts and Telecommunications, Beijing 100089, China
goumang25@163.com

Abstract. Spark is a general computing engine in the big data field. The shuffle phase is an important part of ensuring Spark's operation. However, when the default partitioning strategy is used for partitioning in this phase, tasks with a large amount of data will cause data skew and cause calculation delays. Aiming at the problem of data skew, this paper proposes a partitioning strategy based on genetic mutation algorithm (GAPartition), sampling and predicting the data volume of each Task, and then using genetic mutation algorithm to map tasks and partitions into mathematical models, re-partitioning, and finally constructing the evaluation model of the degree of cluster tilt. Through experiments, the calculation time of Spark's default HashPartition and RangePartition partitions at different degrees of tilt is compared. The experimental results show that GAPartition can effectively solve the problem of data tilt and shorten the execution time of tasks in the cluster.

Keywords: Spark · Data skew · Load balancing · Partition strategy

1 Introduction

In recent years, the Internet has entered an era of explosive data growth. The scale of data grows exponentially at any time. Big data computing engines represented by Mapreduce, Spark, and Flink have emerged. Spark continues the design idea of MapReduce divide and conquer, but is based on memory computing. And other features, so that the calculation speed is 100 times faster than MapReduce. Compared with Flink, it has richer SQL support and a more active community. So Spark is currently the most popular big data computing engine.

Data skew is a common phenomenon in data processing. In the SparkShuffle stage, the same Key on each node should be pulled to the same task for processing. At this time, the Key and Value may be unbalanced. If the amount of data processed by a task far exceeds other tasks, it wills Data skew occurs, and data skew will cause calculation delays and reduce the calculation efficiency of Spark.

In response to the problem of data skew in the Spark Shuffle stage, in the industry, solutions such as increasing the number of partitions of Shuffle, filtering the skewed keys, and two-stage aggregation are often used to deal with the problem, but the above methods are not modified from the perspective of partitions, and the default Hash is used. Partitions may still cause great tilt. In academia, Liu et al. [1]. Proposed a partitioning

Y. Pei et al. (Eds.): IC 2022, LNEE 935, pp. 300–311, 2022.
https://doi.org/10.1007/978-981-19-4132-0_36

algorithm for data skew in the Reduce stage of data flow. These data are regarded as candidate samples. Predict the characteristics of the intermediate data based on the sampled samples to generate a reference table to guide the distribution of the next batch of data evenly. YAN Yi-fei et al. [2] pre-partitioned items based on the most adaptive hash algorithm, and finally partitioned the full amount of data according to the pre-partition table, which effectively solved the problem of key skew and value skew. Zeyu He et al. [4] proposed a dynamic execution optimization method to balance the workload between tasks. During the aggregation process, tasks in smaller partitions can steal and process data from tasks in larger partitions, avoiding sampling and reallocation overhead.

This paper proposes a new partitioning method. Firstly, the data size corresponding to each key is predicted by the pool sampling algorithm, and then the load balancing problem is mapped to the mathematical model of the genetic algorithm, and the allocation of keys to partitions is abstracted as a gene, and a feasible solution The abstract is a chromosome, and its fitness matrix is constructed. The roulette algorithm retains good genes. Through the processes of crossover, duplication, and mutation, the optimal chromosome is the optimal solution to the problem [3].

2 Data Skew in the Shuffle Phase

There is a dependency between Spark RDDs, and the dependency chain will be broken at the wide dependency, and the RDD will be divided into multiple stages. The connection between the stages is Shuffle. Operator operations such as ReduceByKey, GroupByKey, Join, etc. all produce Shuffle. Spark continues the idea of Mapreduce divide and conquer. In the Shuffle stage, there are also Map and Reduce ends. Shuffle connects the two ends to partition the data on the Map. As in the above several operator operations, all values corresponding to each key need to be aggregated into for a total value, the same key should be put into the same partition and passed to Reduce for aggregation during Shuffle.

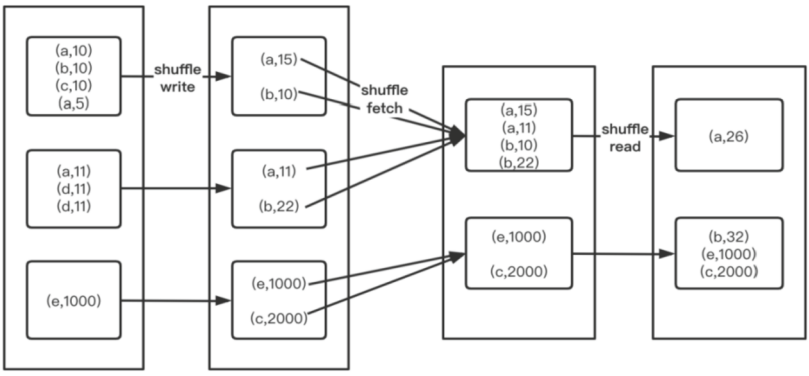

Fig. 1. The cube built by the practical instance

The partitioning process depends on the partitioning strategy. The main function is to determine the number of Reducers in the Shuffle process and to which Reducer the data

on the Mapper side should be allocated [5]. Spark's partitioner is usually HashPartitioner, which is based on hash implementation. For each Key, divide its HashCode by the number of partitions to take the remainder. The remainder is the partition ID corresponding to this key, which ensures that the same Key is in the same partition.. However, when there are a large number of different keys with the same remainder, the data in this partition will be much larger than other partitions. As shown in Fig. 1, in extreme cases, after the Key modulus, the keys with large data volume such as e and c may have the same modulus and will be divided into the same partition, while the keys with small data volume such as a and b will be divided To the same partition, although the same key is guaranteed to be in the same partition, it leads to data skew. The execution time of ReduceTack incoming from this partition will increase sharply, and even OOM will occur, and the execution time of Spark jobs is the longest. Long task decision, so data skew will seriously reduce the computing efficiency of Spark.

3 Algorithm Description

3.1 Data Sampling

The pool sampling algorithm is used to predict the proportion of different keys in the overall data. When the total amount of data is unknown, the algorithm only traverses the data (O(N)) to complete the equal probability sampling, and the key proportion of the sample is the whole The key proportion of the data [6]. As shown in Algorithm 1.

Algorithm 1 pool samplingsampling
Input: Sampling capacity k; Data Key collection Data
Output: map of the proportion of each key

```
map=Map(key,num)
i <- 0
for i = 0 -> Data.length-1 do
   if   i ≤ k then
      map(Data[i]) + 1 //key corresponding value+1
   else
      a = rand(0,i) //Random number between 0 to i
         if   a < k   then
            map(Data[a]) - 1 //key corresponding value-1
            map(Data[i]) + 1
         end if
   end if
end for
for each value in map
   value <- value/Data.length
end for
```

After sampling, the sample proportion information and total space can be obtained. With this information, the characteristics of the overall data can be approximated. In order to carry out the subsequent partition strategy.

3.2 Genetic Mutation Algorithm

Before using the genetic algorithm, we must first clarify the problem to be solved. What we are facing is to allocate N Tasks to M partitions for processing. The data volume of each task is known and the number of partitions is known. We need to use an allocation method to make the data volume of all partitions tend to be even.

The genetic mutation algorithm obtains the optimal solution of the problem through the process of crossover, mutation, and replication of multiple bands. Simulates the thought of survival of the fittest in nature. First, we must map the problem into a mathematical model of genetic algorithm.

Concept 1: Genes and chromosomes
As shown in the figure below, a solution to the problem is called a "chromosome". Multiple elements constitute a chromosome, then the elements are called "genes" on the chromosome. In the above question, the chromosome is expressed as assigning task t to partition p for processing. For example, {2,1,1,0,2} means that Task-0 is assigned to Partition-2, and Task-1 is assigned to Partition-1, Task-2 is assigned to Partition-1… (Fig. 2)

Fig. 2. Genes and chromosomes

Concept 2: Task size matrix
Indicates the amount of data added by task t to partition p

$$\text{taksMatix}[t]\left[p\right] = \text{task}[t] t \in (0, N), p \in (0, M)$$

Concept 3: Fitness function
Fitness is used to evaluate the pros and cons of a chromosome. High-quality chromosomes will be retained in the next iteration, and inferior ones will be eliminated. In this problem, the fitness is the largest amount of data in each partition. Expressed as:

$$\text{Fitness}_i = \max(\text{chromosomes}_i[t]) t \in (0, N), i \in (0, \text{chronum})$$

Chronum is the number of chromosomes.

Concept 4: Cross
There will be multiple iterations in a genetic algorithm, and one iteration becomes an evolution. In each evolution, the two chromosomes of the parent (one is called the parent chromosome and the other is called the mother chromosome) "cross" to produce the new generation of chromosomes (Fig. 3).

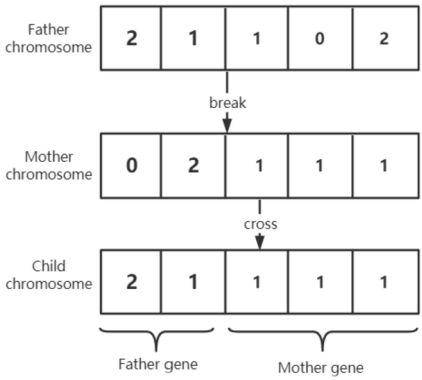

Fig. 3. Cross

As shown in the figure, the cross will select a random position,cut off the two chromosomes of the parent at that position, and splice them together to generate a new chromosome. This new chromosome is a fusion of parental chromosomal genes.

In order to retain the excellent genes of the previous generation, the best quality chromosomes must be selected as parents for each evolution [7]. The higher the fitness, the better the quality, and the chromosomes with higher fitness probability are easier to be selected. It is generally implemented through the roulette algorithm.

$$\text{selectionProbability}_i = \text{Fitness}_i / \sum_{k=0}^{\text{chronum}} \binom{n}{k} \text{Fitness}_k \in (0, \text{chronum})$$

Algorithm 2 Roulette Algorithm
Input: chromosome fitness
Output: Selected chromosome

```
j <- 0;
r <-Random(0,1);
for selected = 0 -> chronum-1 do
   j = j + selectionProbability[selected]
   if(r<=j)
      return chromosome[selected]
   end if
end for
```

Concept 5: Variation

Although the cross can retain good genes, the genes of the new generation are still the parent genes. This can only ensure that after N times of evolution, the calculation result is closer to the local optimal solution, and the global optimal solution will never be obtained. Randomly select several genes on the new chromosome, and then randomly modify the value of the gene to introduce new genes into the existing chromosomes, which is called mutation.

Algorithm 3 mutation
Input: chromosome matrix newChromosomeMatrix
Output: The chromosome matrix of the new generation after mutation

```
variation(newChromosomeMatrix) {
// Randomly find a chromosome
  chromosomeIndex = random(0, chromosomeNum-1);
      // Randomly find a task
        taskIndex = random(0, taskNum-1);
      // Randomly find a node
        nodeIndex = random(0, nodeNum-1);
      newChromosomeMatrix[chromosomeIndex][taskIndex] = nodeIndex;
      return newChromosomeMatrix;
}
```

Concept 6: Copy
In each evolution, the most adaptable chromosomes in the previous generation need to be directly copied to the next generation intact.

Assuming that there are N chromosomes in each evolution, and M chromosomes are directly copied, then N-M chromosomes come from cross.

Algorithm 4 Copy
Input: the parent chromosome matrix chromosomeMatrix and adaptability, the new chromosome matrix newChromosomeMatrix, and the number of copies reserved
Output: The new generation chromosome matrix after copying.

```
copy(chromosomeMatrix, newChromosomeMatrix) {
   chrIndexs = maxN(adaptability, l); //Select the index of the l chromosome with the highest fitness
    for i = 0 ->chrIndexs.length
       var chromosome = chromosomeMatrix[chrIndexs[i]];
       newChromosomeMatrix.push(chromosome);
    end for
    return newChromosomeMatrix;
}
```

The overall flow chart of the genetic mutation algorithm is as follows (Fig. 4)

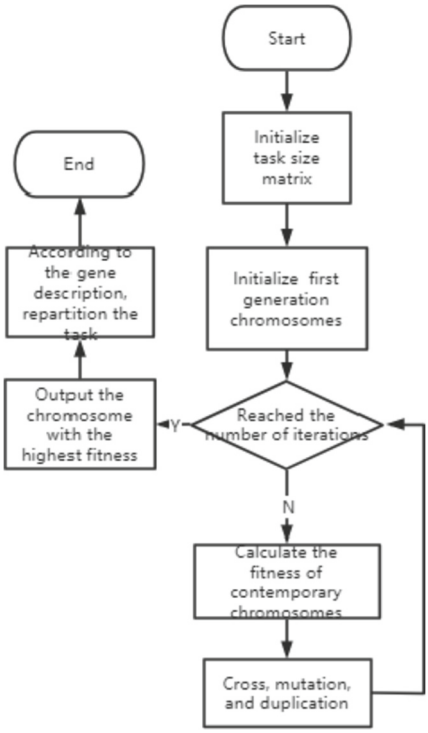

Fig. 4. The overall flow

3.3 GAPartition

GAPartition occurs in the partitioning stage of Spark's Groupbykey and Reducebykey. The algorithm is divided into four parts as a whole. As shown in Algorithm 3, the distribution of the keys of the computing task Data is first predicted by pool sampling. Each key corresponds to a task, and then Task and user-defined partition number M are used as input parameters to execute genetic mutation algorithm [8]. Finally, according to the partition number marked by the genetic mutation algorithm, by rewriting the partition method of Spark, GAPartition is realized to partition the tasks. End the Shuffle phase of Spark.

Algorithm 5 GAPartition
Input: partition number M, calculation task Data
Output: partition result of each task
Task(key,num)=PondSampling(Data,k)//Pool sampling algorithm,where k is the sampling amount
for t = 0 -> Task.size-1 do
for p = 0 -> M do
taskMatix[t][p] <- Task[t]//Task t assigned to p, the value is the sampling size of the task
end for
end for
chromosomeMatrix = init();//Initialize the first generation of chromosomes
// Iterative propagation
for itIndex=1 -> iteratorNum do
Time(chromosomeMatrix);// Calculate the task processing time of the current chromosome
calAdaptability(chromosomeMatrix);
newChromosomeMatrix = cross(chromosomeMatrix);// cross
newChromosomeMatrix = variation(newChromosomeMatrix);// mutation
newChromosomeMatrix = copy(chromosomeMatrix, newChromosomeMatrix);// copy
end for
After the iteration, select the chromosome with the highest fitness in the last generation chromosomeMatrix[best] as result.
for t = 0 -> chromosomeMatrix[best].size-1 do
Task[chromosomeMatrix[best][t]] partition number is t
end for

4 Comparative Experiment

The experimental environment is a Spark cluster built up with 5 virtual machines, and the configuration of each node is shown in Table 1. By default, a Slave node corresponds to a partition, and there are five partitions in total.

Table 1. Cluster configuration table

Node type	Number	CPU	Memory	System
Master	1	2core 2. 6GHz	8G	Hadoop2.7,Spark2.3,Centos7
Slave	5	1core 2. 6GHz	4G	

In this experiment, the dispersion coefficient is used to calculate the degree of data dispersion, which can evaluate the overall tilt of the job after partition. P is the number of partitions, and S is the amount of data. It can be calculated that the average capacity of each partition is

$$S_{mean} = S_{totle}/P$$

The total size of each partition is O, and b is the Taskid assigned to the partition.

$$O_j = \sum_{i=1}^{b} Task_i$$

The dispersion coefficient D is

$$D = \frac{\sqrt{\sum_{j=1}^{p} \left(O_j - S_{mean}\right)^2}}{S_{mean}}$$

When D is smaller, it means that the amount of data stored in each partition is more similar, and the data slope is smaller; otherwise, it means that the amount of data in each bucket is unbalanced.

The wordcount task is used to evaluate the performance of the partitioning scheme at different degrees of tilt, and four sets of data with key tilts of 0.2, 0.4, 0.6, and 0.8 are generated for experiments by comparing Spark's default partitioning method Hash partition.

4.1 Sampling Ratio Selection Experiment

Since GApartition involves sampling, we use separate sampling rates of 5%, 10%, and 20% for sampling. The most suitable sampling ratio is selected by comparing with the real ratio with a key slope of 0.7. The final sampling result is as shown in the figure below. It can be seen that when the sampling ratio is 20%, it is the closest to the true proportion, so we choose the sampling ratio of 20% for the next experiment (Fig. 5).

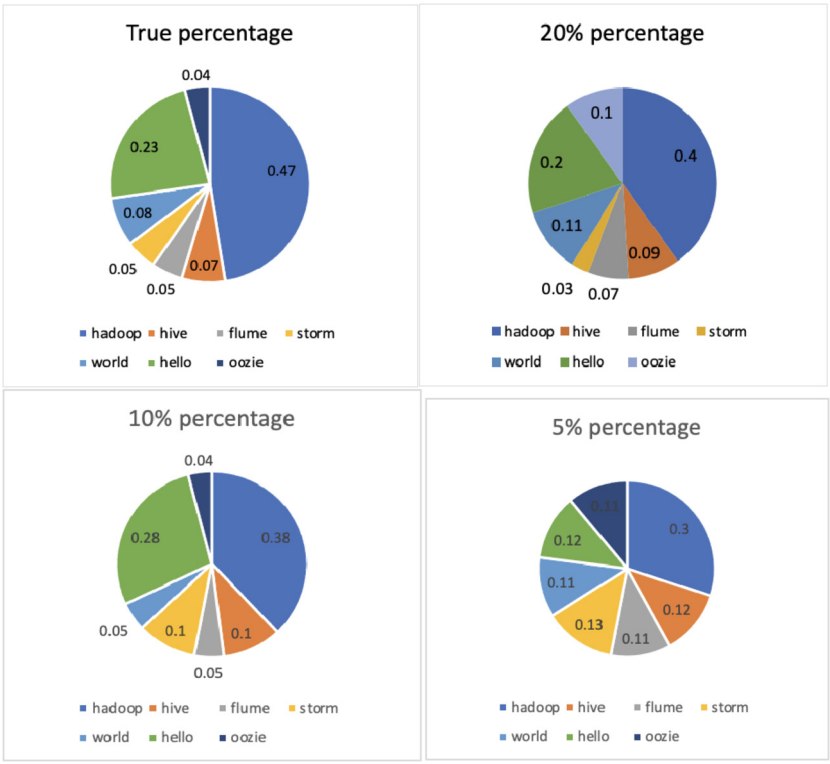

Fig. 5. The result of sampling ratio selection experiment

4.2 Execution Time Comparison Experiment

The figure shows the execution time of the Wordcount task under Hashpartition and GApartition in the case of a 20% sampling ratio, different Key slopes, and the same amount of data. When the slope of the key is 0.2, the speed of Hashpartition is higher than that of GApartition, because GApartition has one more sampling step, which increases the execution time. However, as the slope increases, the time consumption of sampling is negligible compared with the time of task calculation. At a slope of 0.8, the execution time is shortened by 12% compared to Hashpartition. It can be seen that compared to Hashpartition, GApartition can effectively deal with the task delay caused by key tilt (Fig. 6).

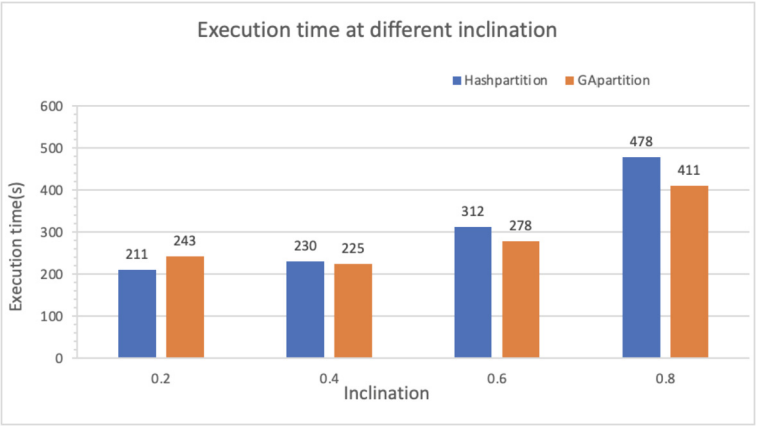

Fig. 6. The result of execution time

4.3 Partition Tilt Comparison Experiment

Using the dispersion coefficient D as an indicator, comparing the overall tilt of GApartition and Hashpartition, the data set sizes are 0.5G, 1G, 1.5G, 2G, and 2.5G, respectively, and their Key tilts are the same (Fig. 7).

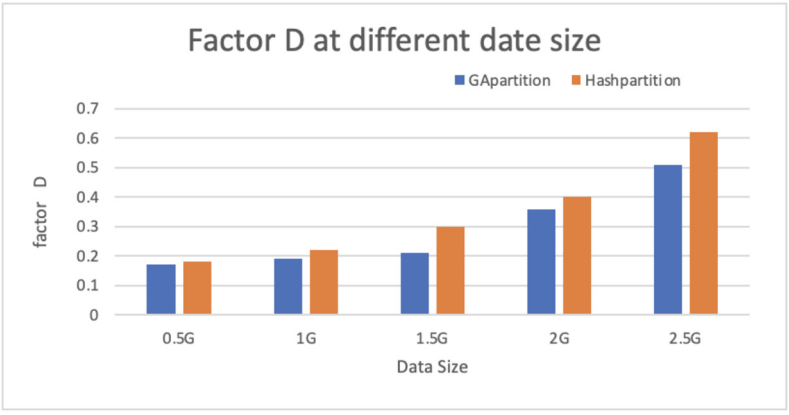

Fig. 7. The result of Partition tilt comparison experiment

The experimental results are shown in the figure. When the amount of data is less than 1G, the dispersion coefficient D of GApartition and Hashpartition is not much different; when the amount of data exceeds 1G, the D of GApartition rises more slowly, which has obvious advantages compared with Hashpartition. It shows that GApartition can effectively alleviate the data skew of the partition. Meet the expectations of designing GApartition.

5 Conclusion

Aiming at the problem of data skew caused by the default Hashpartition in the SparkShuffle stage, the article proposes a new partitioning strategy GApartition, which is based on the genetic mutation algorithm, according to the defined fitness function, through sampling, crossover, mutation, copy and other operations to get the task to Optimal allocation of partitions. Finally, according to the dispersion coefficient of the data tilt, the comparison experiment can verify that GApartition is universal and efficient, and it can reduce the data tilt and the task delay caused by it to a limited extent [9].

References

1. Yan, Y., Wang, Z., Qiu, X., Wang, J.: A shuffle partition optimization scheme based on data skew model in spark. J. Beijing Univ. Posts Telecommun. **43**(02), 116–121 (2020)
2. Liu, G., Zhu, X., Wang, J., Guo, D., Bao, W., Guo, H.: SP-partitioner: a novel partition method to handle intermediate data skew in spark streaming. Future Gener. Comput. Syst. **86** (2018)
3. Gu, H., Li, X., Lu, Z.: Scheduling spark tasks with data skew and deadline constraints. IEEE Access
4. He, Z., Huang, Q., Li, Z., Weng, C.: Handling data skew for aggregation in spark SQL using task stealing. Int. J. Parallel Prog. **48**(6), 941–956 (2020). https://doi.org/10.1007/s10766-020-00657-z
5. Tang, Z., Zhang, X., Li, K., Li, K.: An intermediate data placement algorithm for load balancing in spark computing environment. Future Gener. Comput. Syst. **78** (2018)
6. Tang, Z., Zeng, A., Zhang, X., Yang, L., Li, K.: Dynamic memory-aware scheduling in spark computing environment. J. Parallel Distrib. Comput. **141** (2020)
7. Zvara, Z., et al.: System-aware dynamic partitioning for batch and streaming workloads (2021)
8. O'Driscoll, A., Daugelaite, J., Sleator , R.D.: 'Big data', Hadoop and cloud computing in genomics J. Biomed. Inform. **46**(5) (2013)
9. Cao, S., Haihong, E., Song, M., Zhang, K.: Optimization of data distribution strategy in theta-join process based on spark. Algorithms, Computing and Systems (2018)