Understanding join strategies in distributed systems

Yevgeniya Tyryshkina
National Research University Higher School of Economics
Moscow, Russia
tyryshkina_evgeniya@mail.ru

***Abstract*— This article discusses the features of distributed computing in systems based on MapReduce, analyzes join strategies in Apache Spark, gives basic recommendations for optimizing join operation, and proposes an algorithm of the join operation for distributed data warehouses, which increases the processing speed. Experimental data represents that the method becomes more effective when the volume of initial data increases. So, for 2 TB data, the join operation was performed ~ 37% faster, for 7 TB data already by ~ 47%.**

Keywords*— *MapReduce, distributed systems, Apache Spark, data warehouses, analytics

I. Introduction

The need for a new engine and programming model for data analysis has arisen due to the trend in computer programming that is common across technology industries and changes in the underlying hardware and computer applications. For a long time, the speed of computation has increased due to the increase in processor performance: every year new processors were able to perform more operations per second than before. As a result, applications became faster without the need for code changes. This trend has created a large and resilient ecosystem of applications running on only one processor, which has spurred the development of more powerful processors necessary to maintain scalability and data growth over time. This trend towards hardware expansion ended around 2005: hardware designers faced severe heat dissipation constraints, so they had to abandon the performance acceleration of one processor and switch to the promising direction of parallelizing central processor cores operating at the same speed. This meant for software developers that to make applications run faster, it was necessary to modify from the code, add parallelism, which laid the foundation for new programming models, such as, for example, Apache Spark. Another important factor in the development of such technologies is the constant decline in the cost of storage and data collection, which did not slow down along with the decrease in the growth of processor performance. Every 14 months, the cost of storing 1TB of data is reduced by almost 2 times, thus allowing organizations of all sizes to create analytical systems built on large data warehouses.

II. MapReduce Technology

A. The concept of MapReduce

MapReduce is a distributed model that was created for parallel computing and is used in big data technologies. The Mappreduce technology allows computations on very large datasets of several petabytes and works in computer clusters of hundreds of nodes. This technology was first introduced by Google. [1].

MapReduce can rightfully be called the main Big Data technology because it is initially focused on parallel computing in distributed clusters. The MapReduce technology is based on the principle of dividing information into many parts, these parts are distributed between individual nodes in a cluster, which allows parallel data processing. Development of MapReduce applications is quite simple to implement, since programs are automatically distributed among the cluster nodes. The executive system takes care of the implementation details. It is responsible for the separation of input data and exchange between computational nodes, distributes tasks between them, and in addition, automatic fault handling is supported. Thanks to this, programmers can easily and efficiently use the resources of distributed Big Data systems. MapRedutse technology is extremely universal and can be applied in completely different areas: from indexing web pages, calculating the amount of content on different hosts, reversing the graph of web links and building inverted indexes or implementing counters for requests to a web host to complex analytical tasks using machine learning technologies that require processing huge amounts of data, such as machine translation or clustering documents. Also, MapReduce is adapted for multiprocessor systems, voluntary computing, dynamic cloud and mobile environments [2].

B. Computational model

The concept of MapReduce is that the process of processing data stored in one or more files is divided into two phases: map and reduce [3]. The result is key-value pairs. Each phase has a key value as input and output. Intermediate results obtained after the completion of the map stage are moved between the nodes of the cluster. This movement is called shuffle. After that, the reduction phase begins. These steps can be repeated several times.

In more detail, the 5 stages of data processing are presented in Figure 1.

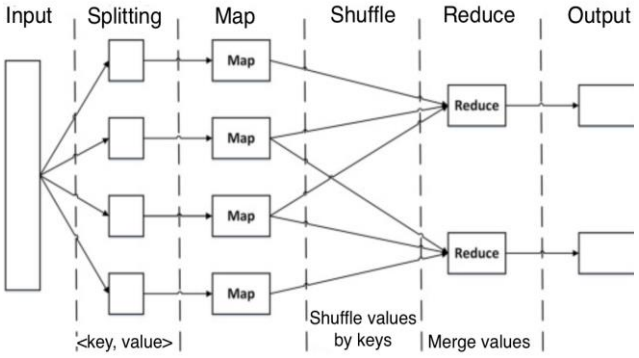

Fig. 1. MapReduce Computational model

- Map - preprocessing input data as a large list of values. At the same time, the main node is responsible for dividing the key-value list into several parts and transferring it to the worker nodes for

978-1-6654-3941-1 @2021 IEEE

execution. The task of each worker node is to perform a transformation operation to its local data and save the obtained result.

- Shuffle, when worker nodes redistribute data based on the keys previously generated by the Map function so that all the data for one key resides on one worker node;
- Reduce is an operation for reducing the list of key-values obtained as a result of the Map operation, which is performed in parallel on each worker node for each group, combined by a key. The result of the operation is collected by the main node. The master node receives intermediate responses from worker nodes and transmits them to free nodes for the next step.

These steps are repeated a number of times, depending on the problem, and the final result is a solution to the original problem.

The procedure of data transformation of (key, value) pairs is shown as follows:

Map: (k1, v1) → List (k2, v2)

Reduce: (k2, List (v2)) → List (k3, v3)

III. Apache Spark

Apache Spark is a complete computing system that provides a set of libraries for performing data processing in distributed cluster systems. Currently, among the open source tools used to solve such problems, spark is recognized as the most actively developing. The pace of development, high efficiency and ease of use makes this tool very useful for any specialist interested in solving computational problems in big data.

Development can be done in several widely used programming languages such as Python, Java, Scala, and R. Tasks can be run from a laptop or compute cluster of thousands of nodes. Spark can solve a wide range of tasks from SKL and machine learning to streaming data processing. This makes Apache Spark a convenient start-up system, flowing into big data processing on an incredibly

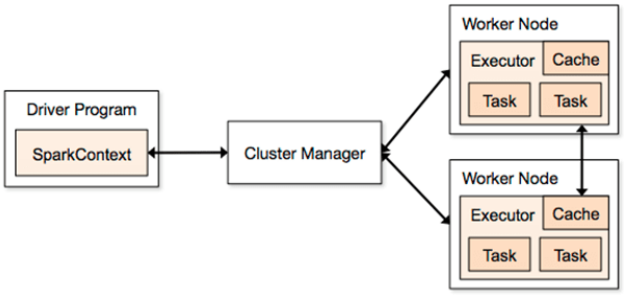

Fig. 2. Apache Spark architecture[4]

huge scale.

A Spark application consists of a single control process, called a driver, and a set of executing processes called executors, that are dispersed across the cluster nodes [4].

The driver provides high-level control of the workflow. Executors ensure that work is done in the form of tasks, as well as storing any data that the user wants to cache. The driver and executors work throughout the entire program execution process, while resources for the executors are dynamically allocated. Each performer is capable of performing several tasks in parallel. The deployment of these processes to the cluster is handled by the cluster manager (YARN, Mesos, or Spark Standalone), but the driver and executors are present in every Spark application.

IV. Join strategies

Traditionally, joins are classified according to semantic characteristics into cross join, inner join, outer join and semi-join. In this paper, we consider the possibility of speeding up a particular case of a join operation: a left outer join , that is, all values from the left set are included in the final selection, regardless of whether there are key matches in the right set. If a matching identifier is found in the corresponding table, it is returned or a null value is added.

The following algorithms are distinguished according to the different ways of implementing the join: joins with nested loops, joins based on sorting, and joins based on the hash. Also, some tools allow the use of secondary data structures such as secondary indexes, join indexes, bitmap indexes, Bloom filters. Such secondary structures are created to further improve the basic join algorithms. Although these well-known traditional methods of data processing used in relational DBMSs are adapting to the new conditions of distributed computing, such approaches are rarely applied. Approaches with building additional indexes are often ineffective since it can take too much time to rebuild the indexes during the computation process, which completely covers the resulting speedup on the join.

In a distributed environment, the additional concept of connection graph topology is introduced. Graph topology reflects how data partitions will be processed in a distributed system and this depends on several of the following factors: [5]:

• partitioning scheme that characterizes data partitioning in the system into partitions. The partitioning scheme consists of a partitioning function (for example, hash, range, or arbitrary partitioning), a key, and a partition counter.

• data exchange operators that change the partitioning scheme of the dataset and include initial partitioning, redistribution, full merge, partial redistribution, and partial merge (Figure 3).

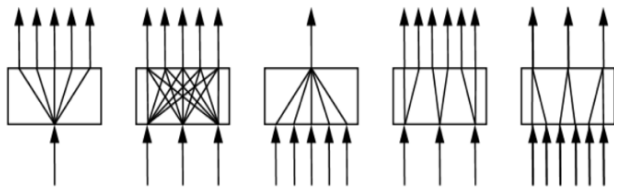

Fig. 3. Types of data exchange topologies (initial partitioning, redistribution, full merge, partial redistribution and partial merge) [5]

• merge schemes that modify data exchange statements to provide certain additional properties within a section (for example, order).

• partitioning policies, which determine whether partitions can be duplicated across multiple worker nodes, and include duplicate distribution and non-duplication distribution.

The basic idea of partitioning is quite simple - instead of storing data in one piece, we will divide it into several independent parts. All parts retain the primary key from the original part, so any data can be accessed quickly enough.

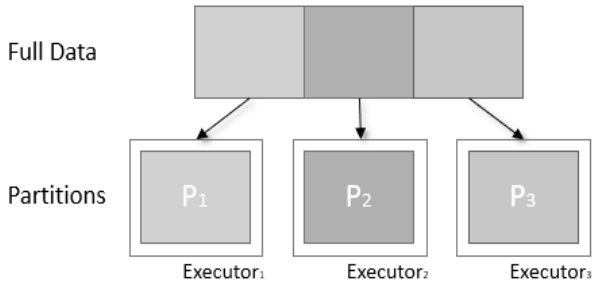

Fig. 4. The scheme of ideal partitioning, in which the data is evenly divided into partitions and each partition (P) is entirely located on one executor (Executor)

In order to choose the correct strategy for combining data, Spark needs to perform the stage of building a physical plan for the query execution.

A. Broadcast join.

The most effective way to combine data is when one side of the combination is small. This criterion can be influenced by changing the spark.sql.autoBroadcastJoinThreshold parameter in SQLConf. In this case, the small side is copied to all executive nodes, where it becomes possible to connect to the main table locally.

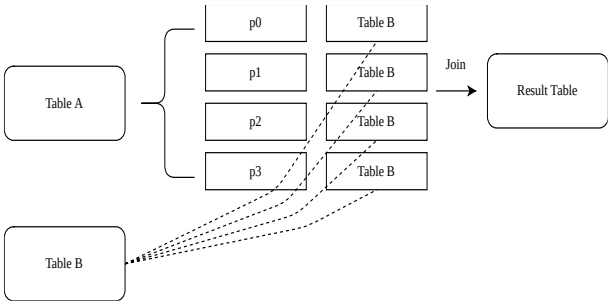

Fig. 5. Broadcast join

Keep in mind that there is a maximum size limit for a smaller table when it is still possible to use a broadcast join. Spark recently increased this size from 2GB to 8GB. In addition to the size of the data, keep in mind that in the case of an outer join, the spark may try to apply the sort-merge strategy and, probably, you will need to explicitly disable the spark.sql.join.preferSortMergeJoin config, and also at the programmatic level specify which table should be used as broadcast as follows [6]:

largedataframe.join (broadcast (smalldataframe))

B. Sort merge join.

This is the default method if the keys for the join can be sorted. Of the features, it can be noted that, unlike the previous method, code generation optimization for operating is available only for inner join

Input datasets must be sorted by the columns involved in the join condition. The connection is carried out in one scan (pass through) each of the input tables. That is, the same string is read only once, which gives an advantage over joining with nested loops.

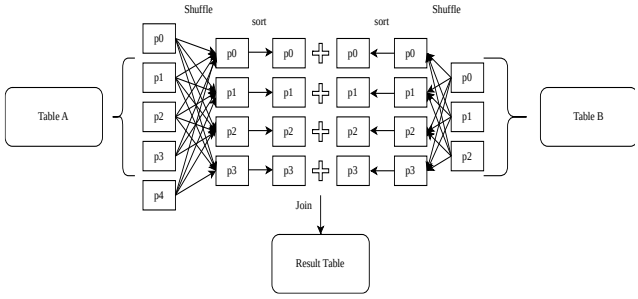

Fig. 6. Sort merge join

C. BroadcastNestedLoopJoin u CartesianProduct

There are situations when it is impossible to directly compare two sets of data by key or there are no keys, then the join strategy is selected based on the size of the tables, or CartesianProduct.

D. Shuffle hash join.

If the keys cannot be sorted, or the default sort-merge join selection setting is disabled, Catalyst tries to use a shuffle hash join.

In addition to checking for settings, it is also checked that Spark has enough memory to build a local hash map for one partition (the total number of partitions is set by the spark.sql.shuffle.partitions setting)[7].

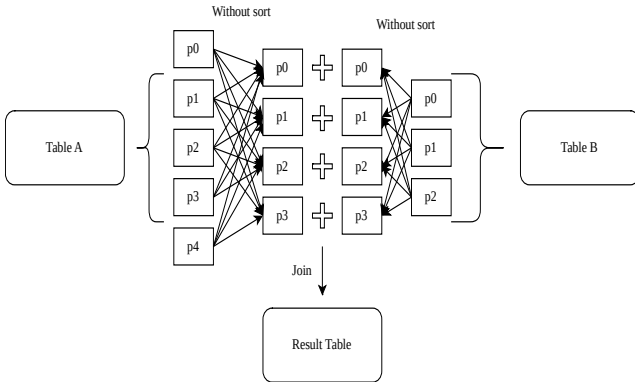

Fig. 7. Shuffle Hash Join

V. ALGORITHM FOR ACCELERATING JOIN OPERATION

In this section of the article, we will propose an algorithm of the join operation for distributed data warehouses, which increases the processing speed.

The proposed algorithm assumes the use of data partitioning before merging. To work with partitions, the mapPartitions () or mapPartitionsWithIndex () functions are used, which converts each partition of the source dataset into several result elements (possibly none). Specific partitioning rules can be set by creating your customPartitioner class. It is important not to forget that the size of the partitions should be approximately the same to avoid the problem of data skew[8].

Then, in the process of processing RDD2, you can do repartition using the same algorithm as in RDD1 (customPartitioner), and then inside the

mapPartitionsWithIndex () function, refer to the index to a specific file of the first set and load it into memory directly using the map data structure. Mutable Maps are implemented as hash-tables, with m(...) lookups and m(...) = ... updates being efficient O(1) operations.

The key advantage that this approach provides is the complete absence of the need to redistribute the data that lies on the nodes to create an RDD since the second RDD is not created at all, which means there is no need to physically redistribute all the data to the machines where the calculations will take place.

```
val data = RDD2.partitionBy(customPartitioner)
val res = data.mapPartitionsWithIndex((index, partition) = {
val conf = confBroadcast.value.value
val partitionRDD1 = readFromHDFS(conf, path + index)
val partitionData = partitionRDD1.map {
case Array(k, v) => k -> v.toLong;
case _ => “” -> “”}
}.toMap

val newPartition = partition.map(
        record => {(record , partitionData(record._1))})
new Partition
})
```

The main feature of this approach is that each performer will be able to independently load data into the RAM from any other machine in the cluster, without the participation of the driver machine, which is a kind of "bottleneck". Also, the block of data that is needed at the moment for this particular performer for those partitions on which he is currently performing calculations will be loaded, which allows a part of the data set to be loaded into memory promptly, and not entirely. In a sense, it is an imitation of a "merge join" behavior, but in our case, there is no need to sort the rows, which also gives a serious gain in execution speed. And the great thing is that this approach ensures that there are no cache misses, which greatly affects the operation's speed.

VI. Experimental results

Experimental calculations were performed on text data compressed with the GZIP algorithm. Several tests of the new merge approach have been performed and compared to the standard sort-merge join used in Spark SQL [9] (Figure 8).

To carry out the calculations, 128 performers with 6 cores and 16Gb of memory for performers and 8G for the driver were involved.

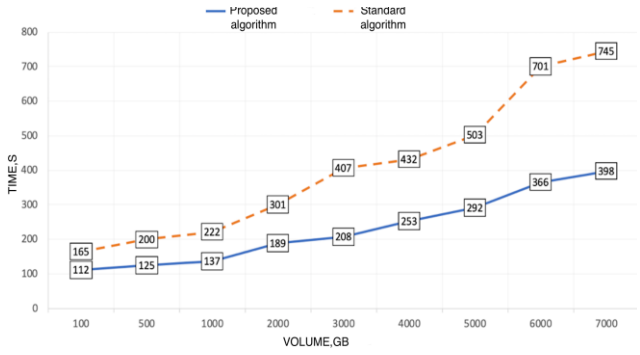

Fig. 8. Comparison of the calculation execution time relative to the amount of input data for the algorithm proposed in work (solid line) and in standard Spark SQL (dashed line)

The measurements were carried out by standard spark.time() over the volumes from 100 GB to 7Tb of text unstructured data. Each measurement was performed 5 times, and the execution time was averaged.

VII. CONCLUSION

Thus, the results of the experiments showed that the algorithm works more efficiently, the larger the input data. It can be seen that for 2Tb data the join operation was performed ~37% faster than the proposed algorithm in Spark SQL, for data of size 7Tb it was already ~47%. Today, Spark is the most popular and shows a high query execution speed compared to other products for distributed computing, therefore, such acceleration can be considered significant.

Many tasks have yet to be solved and implemented using this algorithm. For example, we have now considered only one type of join operation - left (right) join. In the future, it is necessary to implement internal join. Also, the problem of data skew is not solved, it is necessary to implement automatic skew detection and the creation of additional keys. Nevertheless, this algorithm can be useful in many computational tasks, since it is quite stable and shows a high execution speed.

Acknowledgment

The author would like to thank the Basic Research Program of the National Research University Higher School of Economics for their support.

References

1. K. Shim. MapReduce algorithms for big data analysis. VLDB Endowment, 2012, vol. 5, no. 12, pp. 2016–2017. doi:10.14778/2367502.2367563.

2. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Communications of the ACM, 2008, vol. 51, no. 1, pp. 107–113. doi:10.1145/1327452.1327492.

3. J. Dittrich, J. Quian´e-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). PVLDB, 2010, vol. 3, no. 1-2, pp. 515-529. doi: 10.14778/1920841.1920908.

4. Holden Karau, Rachel Warren. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark, O’Reilly, 2017, p. 10.

5. Nicolas Bruno, YongChul Kwon, Mingchuan Wu. Advanced Join Strategies for Large-Scale Distributed Computation // Proceedings of the VLDB EndowmentAugust, 2014.

6. R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. [In Proc. of the “2013 ACM SIGMOD International Conference on Management of Data”]. New York, 2013. doi:10.1145/2463676.2465288.

7. J. Dittrich, J. Quian´e-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). PVLDB, 2010, vol. 3, no. 1-2, pp. 515-529. doi: 10.14778/1920841.1920908.

8. Y. C. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. [In Proc. of “Open Cirrus Summit”]. Moscow, 2011.

9. Holden Karau, Rachel Warren. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark, O’Reilly, 2017, p. 10.