SQL QUERY EXECUTION OPTIMIZATION ON SPARK SQL

G. Mozhaiskii, V. Korkhov, I. Gankevich

Saint Petersburg State University, 13B Universitetskaya Emb., Saint Petersburg, 199034, Russia

E-mail: i.gankevich@spbu.ru

Spark and Hadoop ecosystem includes a wide variety of different components and can be integrated with any tool required for Big Data nowadays. From release-to-release developers of these frameworks optimize the inner work of components and make their usage more flexible and elaborate. Nevertheless, since inventing MapReduce as a programming model and the first Hadoop releases data skew has been the main problem of distributed data processing. Data skew leads to performance degradation, i.e., slowdown of application execution due to idling while waiting for the resources to become available. The newest Spark framework versions allow handling this situation easily from the box. However, there is no opportunity to upgrade versions of tools and appropriate logic in the case of corporate environments with multiple large-scale projects development of which was started years ago. In this article we consider approaches to execution optimization of SQL query in case of data skew on concrete example with HDFS and Spark SQL 2.3.2 version usage.

Keywords: Big Data, Spark SQL, distributed data processing, HDFS

1. Introduction

The main purpose of this work is to optimize the execution time of certain SQL query (fig. 1), which uses SQL JOIN operation and filters the result set by specified parameters, performed on large amount of distributed data with minor source code changes. Data contain records about events in multiple tables that we join and correlate, multiple event tables can be combined by keys and occurrence order.

Required tools to use are Apache Spark framework version 2.3.2 and its SQL module. The data represented in Apache Parquet files format with SNAPPY[1] compression and stored in HDFS.

```
SELECT
    U.field01,
    U.field02,
    U.field03,
    I.field03 AS I_DATE,
    U.field04
  FROM UE U
    LEFT JOIN (SELECT field02,field01,field03,
                      (LAG(field03,1,0) OVER win) AS previous_date,
                      (LEAD(field03,1,0) OVER win) AS next_date
               FROM IntraFreq
               WINDOW win AS (PARTITION BY field02, field01 ORDER BY field03))
I
    ON U.field02 = I.field02 AND
       U.field01 = I.field01 AND
       ABS(U.field03-I.field03) < ABS(U.field03-I.previous_date) AND
       ABS(U.field03-I.field03) < ABS(U.field03-I.next_date)
```

Figure 1. SQL query execution of which we optimized.

The query is executed on Apache YARN cluster using the following configuration:

- 4 Spark executor instances;
- Spark submit mode is *cluster*;
- Spark driver has 4 GB of RAM, 1 CPU kernel;
- Spark executor has 16 GB RAM, 4 CPU kernels.

2. Data representation

The data were read using *Dask* python3 library than converted to *Pandas* data frames and finally read by *PySpark*. The dimension of the first table is 92 797 197×10 and of the second one is 11 268 684×10. There is a significant difference in the dimensions of the data tables. It makes a direct impact on the performance of SQL JOIN operation in case of data distribution.

3. Spark analytics

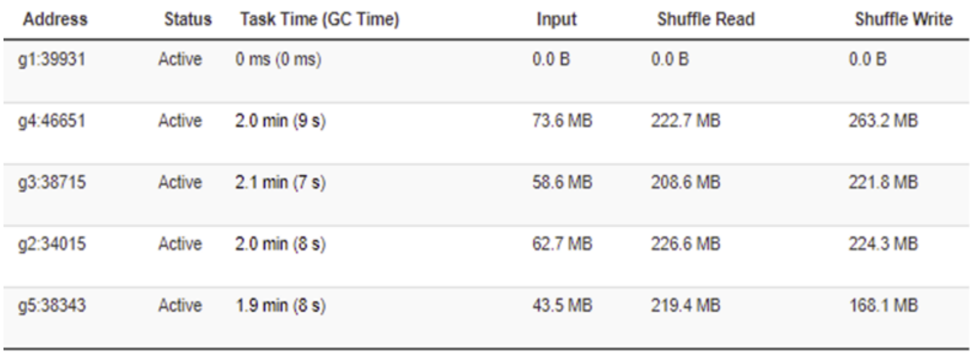
The examination of the query execution process provided by Spark analytics module gives the following results: the data extraction process takes 7.6 seconds and leads to unbalanced shuffle read-write of 877.4 MB (fig.2). That means that the certain part of computational time is spent on sending data over the network from one cluster node to another which means slowdown of the whole application execution flow. The query execution time is 31.65 seconds, and the total time is around 39.26 seconds.

The join algorithm which Spark uses is *SortMergeJoin* which shows bad performance on tables with significant difference in dimensions.

[1] https://github.com/google/snappy

SortMergeJoin is composed of 2 steps. The first step is to sort the datasets and the second operation is to merge the sorted data in the partition by iterating over the elements and according to the join key join the rows having the same value. It is default algorithm that Spark uses for joins, but it is not suitable for our data, because the partitions should be co-located. Otherwise, there will be shuffle operations to redistribute the rows that are joined as it has a pre-requirement that all rows having the same value for the join key should be stored in the same partition.

Moreover, the partitions size per node are too large, the recommended size should be between 30MB and 100MB per execution node according to [1].



Address	Status	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
g1:39931	Active	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
g4:46651	Active	2.0 min (9 s)	73.6 MB	222.7 MB	263.2 MB
g3:38715	Active	2.1 min (7 s)	58.6 MB	208.6 MB	221.8 MB
g2:34015	Active	2.0 min (8 s)	62.7 MB	226.6 MB	224.3 MB
g5:38343	Active	1.9 min (8 s)	43.5 MB	219.4 MB	168.1 MB

Figure 2. The execution statistic per node before optimization.

4. Optimization

To optimize time efficiency of SQL query (fig. 1) next approaches were used: *BroadcastHashJoin*, *spark.shuffle.partitions* parameter tuning and *bucketing*.

According to [2] the *MAPJOIN* (or *BROADCAST*) hint shows good performance boost in case of similar data features. The *BroadcastHashJoin* (*MAPJOIN*) will spread one of the tables to all executor nodes, so exchange stage execution time will be reduced. That approach (fig. 3) will save time spent previously on data routing over the network and reduce shuffle read-write volume to 111.5MB, but the problem with non-uniform data distribution for processing between execution nodes is still actual.

```sql
SELECT /*+ BROADCAST(I) */
    U.field01,
    U.field02,
    U.field03,
    I.field03 AS I_DATE,
    U.field04
  FROM UE U
    LEFT JOIN (SELECT field02,field01,field03,
                      (LAG(field03,1,0) OVER win) AS previous_date,
                      (LEAD(field03,1,0) OVER win) AS next_date
               FROM IntraFreq
               WINDOW win AS (PARTITION BY field02, field01 ORDER BY field03))
    ON U.field02 = I.field02 AND
       U.field01 = I.field01 AND
       ABS(U.field03-I.field03) < ABS(U.field03-I.previous_date) AND
       ABS(U.field03-I.field03) < ABS(U.field03-I.next_date)
```

Figure 3. SQL query optimized using *BroadcastHashJoin*.

The *spark.shuffle.partitions* parameter according to [1] should give uniform data distribution across executor nodes and its size should be between 30MB and 100MB. Empirically the parameter value was set to 35. That provides uniform shuffle read-write around 35MB per execution node.

Nevertheless, the exchange stage is still present while reading the table which will be broadcasted. To reduce this stage bucketing can be used [3]. Bucketing partitions the data and prevents data shuffling. The data is allocated to a predefined number of buckets based on the value of one or more data columns. At the application startup time, data processing module calculates the hash values for specified columns and based on it places the data in one of the buckets. Bucketing provides following advantage - each bucket contains equal size of data, map-side joins perform better compared to a non-bucketed table.

To implement bucketing the way of parquet file reading should be changed (fig. 4).

```
CREATE TABLE IntraFreq
      USING PARQUET
      CLUSTERED BY (field01)
            SORTED BY (field02 ASC, field01 ASC, field03 ASC)
      INTO $partitions_number buckets
AS select field01, field02, field03 from parquet.`$path`
```

Figure 4. The way to read parquet file to use bucketing.

5. Results and future

SQL query (fig. 1) optimization techniques and corresponding results presented in table 1 and figure 5.

Nevertheless, these optimization approaches are applicable only when the size of the broadcasted table is lower than the amount of memory allocated for Spark driver and executors. In the future, we plan to implement other optimizations including custom partitioner and custom shuffle manager.

Table 1. The optimization techniques and their effect on query execution time

Set up	**“SELECT” query time**	**Data Reding time**	**Total time**
SortMergeJoin	~31.65 sec	~7.6 sec	~40.38 sec
BroadcastHashJoin + *spark.shuffle.partitions* = 35 + *bucketing*	~11.9 sec	~19.2 sec	~31.28 sec

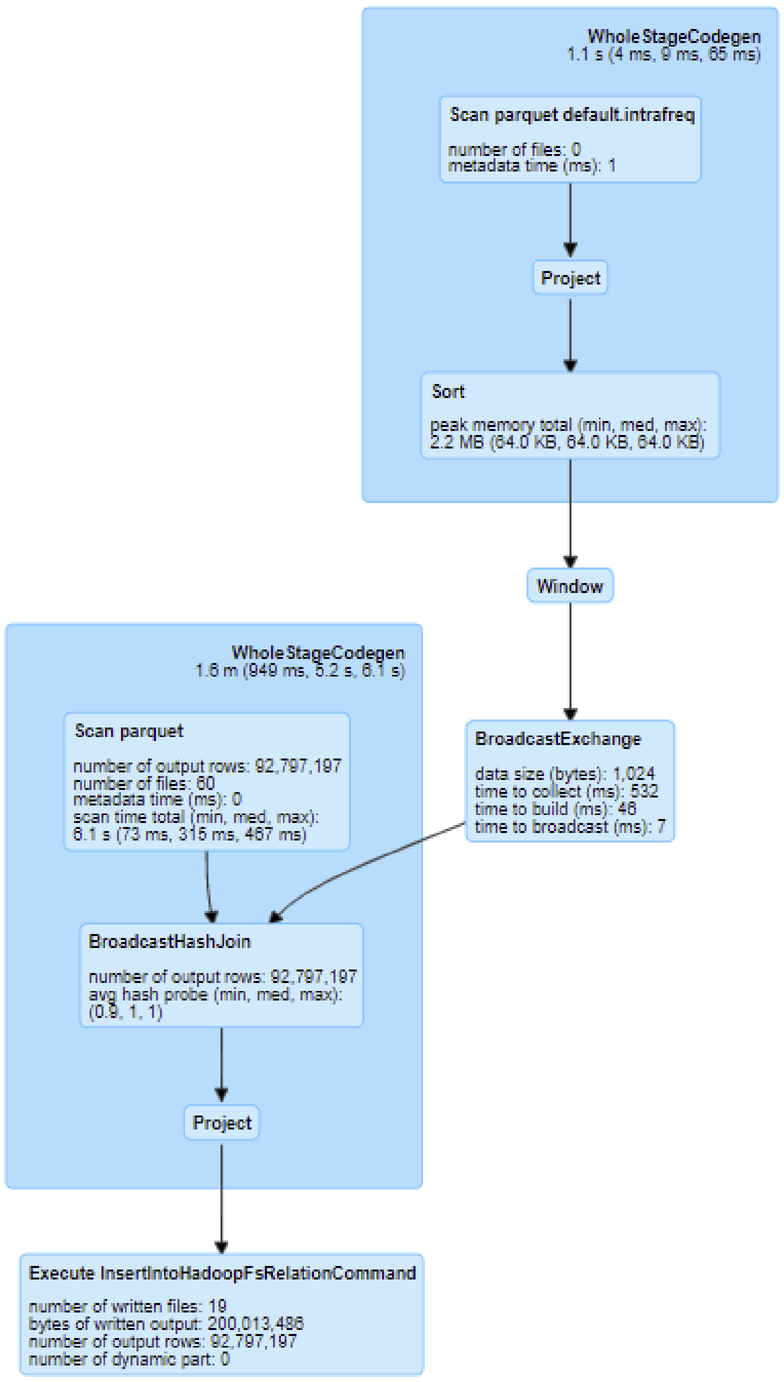

WholeStageCodegen
1.1 s (4 ms, 9 ms, 65 ms)
Scan parquet default.intrafreq
number of files: 0
metadata time (ms): 1
Project
Sort
peak memory total (min, med, max):
2.2 MB (64.0 KB, 64.0 KB, 64.0 KB)
Window
WholeStageCodegen
1.6 m (949 ms, 5.2 s, 6.1 s)
Scan parquet
number of output rows: 92,797,197
number of files: 60
metadata time (ms): 0
scan time total (min, med, max):
6.1 s (73 ms, 315 ms, 467 ms)
BroadcastExchange
data size (bytes): 1,024
time to collect (ms): 532
time to build (ms): 46
time to broadcast (ms): 7
BroadcastHashJoin
number of output rows: 92,797,197
avg hash probe (min, med, max):
(0.9, 1, 1)
Project
Execute InsertIntoHadoopFsRelationCommand
number of written files: 19
bytes of written output: 200,013,486
number of output rows: 92,797,197
number of dynamic part: 0

Figure 5. *DAG* after optimizations.

References

[1] A. Yudovin. Essential Optimization Methods to Make Apache Spark Work Faster // ALTOROS. – 2019. – URL: https://www.altoros.com/research-papers/essential-optimization-methods-to-make-apache-spark-work-faster/.

[2] Ji X. et al. Query Execution Optimization in Spark SQL // Scientific Programming. – 2020. – Vol. 2020.

[3] Pawan Singh Negi. Bucketing in Spark: Spark job optimization using Bucketing // Clairvoyant Blog. – 2021. – URL: https://blog.clairvoyantsoft.com/bucketing-in-spark-878d2e02140f.