Chapter 17

Spark SQL Shuffled Hash Join

Spark official documentation does not provide any documentation regarding shuffled hash join. What does that mean? That means there is no function contract, neither functional nor other aspects such as performance, when to use, and when to avoid. That leaves the data engineer with informal references such as the reference I will use now [5]:
« The Shuffle Hash Join is one of the join algorithms used in Apache Spark to combine data from two different DataFrames or datasets. It's designed to perform efficient joins by partitioning and hashing the data. »

The above quoted information from that reference, contradicts the following information from another reference [5]:
« According to SPARK-11675 Shuffled Hash Join was removed in Spark 1.6 and the reason was:
"…I think we should just standardize on sort merge join for large joins for now, and create better implementations of hash joins if needed in the future".

…Shuffled Hash Join was reintroduced in Spark 2.0 according to SPARK-13977.

…It's worth mentioning the PR for SPARK-13977 which points that Shuffled Hash Join was removed in favor of Sort Merge Join which is faster and more robust. »

17.1 The Catastrophic Abstraction

The only honest description that I can use after the above text is: abstraction should never be like that. Let us go back to the original idea of software abstraction in [50], and in section 15.3. Specifically figure 15.3. Spark sql hash join breaks all rules of correct abstraction by not providing documentation at all, but at the same time, providing configuration that is 'passed by' in the formal documentation. Moreover, and on the other level, spark suggests, in his formal documentation, spark suggest to fine-tune spark SQL join for optimal performance! while spark sql join is not scalable as we proved in this book [1].

I have seen many experienced data engineers asking, young candidates in interviews, asking questions such as:
How to speedup spark join, when you have two very large tables?

And they force the poor young engineer to enter into an endless discussion

[1] *As the kind reader knows we speek about optimal performance in the context of speedup, which is the measure of scalability. For non-scalable applications, we do not advise users to try to change things to improve speedup, because it does not exist.*

of spark configurations and hints.
They do that because spark official documentation does that and neglect the world-wide-accepted fact shown figure 15.3.

Regardless of our opinion, hash-join still exist in spark, as an option. I will benchmark spark ShuffledHashJoin in the next section 17.2.

17.2 Benchmark Spark SQL Shuffled Hash Join

We will repeat usecase-two from section 14.1 for shuffled hash join. Input tables are presented in listing 14.1. Scala spark code is modified to include the shuffled hash join hint as presented in listing 17.1.

Listing 17.1: *Scala spark code for shuffled hash join benchmark.*

```
def testDataFrameJoin()(implicit session: SparkSession) = {
    myPrintln("testDataFrameJoin ... SHUFFLE_HASH ...")
    val trainDF = session.read.parquet("/user/warehouse/large/train.parquet")
    val trainWithUuidDF =
            session
                .read
                .parquet("/user/warehouse/large/trainWithUuid.parquet")
    val columnsNamesPostfixed: List[String] =
        trainWithUuidDF.columns.map{columnName =>
                    s"${columnName}_wUuid")
                }.toList
    val trainWithUuidDfRenamed =
            trainWithUuidDF.toDF(columnsNamesPostfixed: _*)
    val resultDF = trainDF.hint("SHUFFLE_HASH").join(trainWithUuidDfRenamed,
      trainDF("weight") === trainWithUuidDfRenamed("weight_wUuid") &&
      trainDF("feature_00") === trainWithUuidDfRenamed("feature_00_wUuid"),
      "inner")
    resultDF.write.save("/user/warehouse/large/result.parquet")
}
```

17.3 Usecase Running & Results

We execute the benchmark using spark-submit in listing 17.2. Results are shown in table 17.1.

Listing 17.2: *Spark submit command for shuffled hash join benchmark.*

```
spark-submit --class fr.hp.cluster.performance.SortLoadForSpark \
    --master yarn \
    --deploy-mode client \
    --num-executors 1 \
    --conf spark.sql.join.preferSortMergeJoin=false \
        --conf spark.sql.autoBroadcastJoinThreshold=2 \
        --conf spark.sql.shuffledHashJoinFactor=1 \
    /home/unified-hp-user/DATA/jars/hp-spark-performance-assembly-1.0.jar
```

Number of executors	Time elapsed in minutes sort merge join	Time elapsed in minutes hash join
1	Fail	Fail
4	47	44
6	41	42
12	42	38
16	41	40

Table 17.1: *Results for shuffled hash join benchmark.*

Results presented in table 17.1 show that elapsed time benchmark for both SortMergeJoin and ShuffledHashJoin are close. There is no performance edge

in-favor-of neither of them. However, SortMergeJoin is the default algorithm by Spark. So we recommend using it. We recommend not touching the spark configuration related ninetieth to hints nor to other extra configurations seen in listing 17.2.

Another important remark is the fail case for one executor. Please see figure 17.1. This is clearly either the OOM problem or the OOS problem that have been extensively explained throughout of this book. Please see section 23.4.

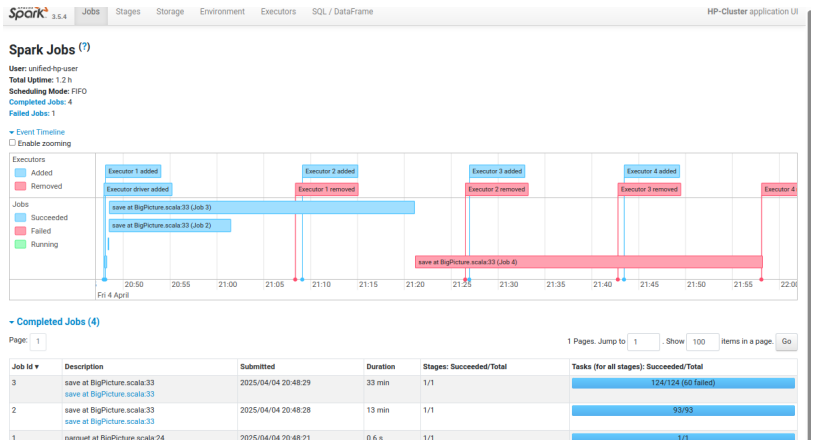

Figure 17.1: *Execution-fail shuffled hash join for one executor. OOM problem or OOS problem? I will let the kind reader answer this question.*

We do not think there is neither feasibility usecase nor performance usecase that will make shuffled hash join a must-use approach. As result, we think that the data engineer should not involve in understanding these three different SQL join algorithms. Also, the data engineer should not provide hints in relation to these algorithms, nor configurations. The only configuration the data engineer should provide in this context is the number of execu-

tors. We should calculate the minimum number of executors as directed in section 30.3.

17.4 Discussion

In this part we have extensively presented SPARK SQL library. We have shown how to avoid OOM problem clearly. During this road, we benchmarked and analyzed the three spark SQL join strategies and concluded that we should not go into their details. Spark SQL library is a valuable library. We all use it and we all appreciate it. We should always appreciate the fact that SQL transformations are not-scalable yet feasible. Feasibility has a golden value as we demonstrated in many places in this book.

Finally, we conclude from this part that spark is auto-configured. We should not break its valuable optimized execution plan that he generate it automatically. If we want to best benefit from spark, we should let him manage his internal details and resources, and we focus on the functional aspects of our application using the novel spark performance model that we present in this book.