A Tale of Spark Job in Production

Introduction

In this blog I documented some of the optimizations added while developing a Data Pipeline based on **Apache Spark**. There were many references/blogs followed to achieve the desired optimizations and I am far from documenting all of them but here I mentioned few notable ones.

For this Transformation job we had the following challenges and requirements:

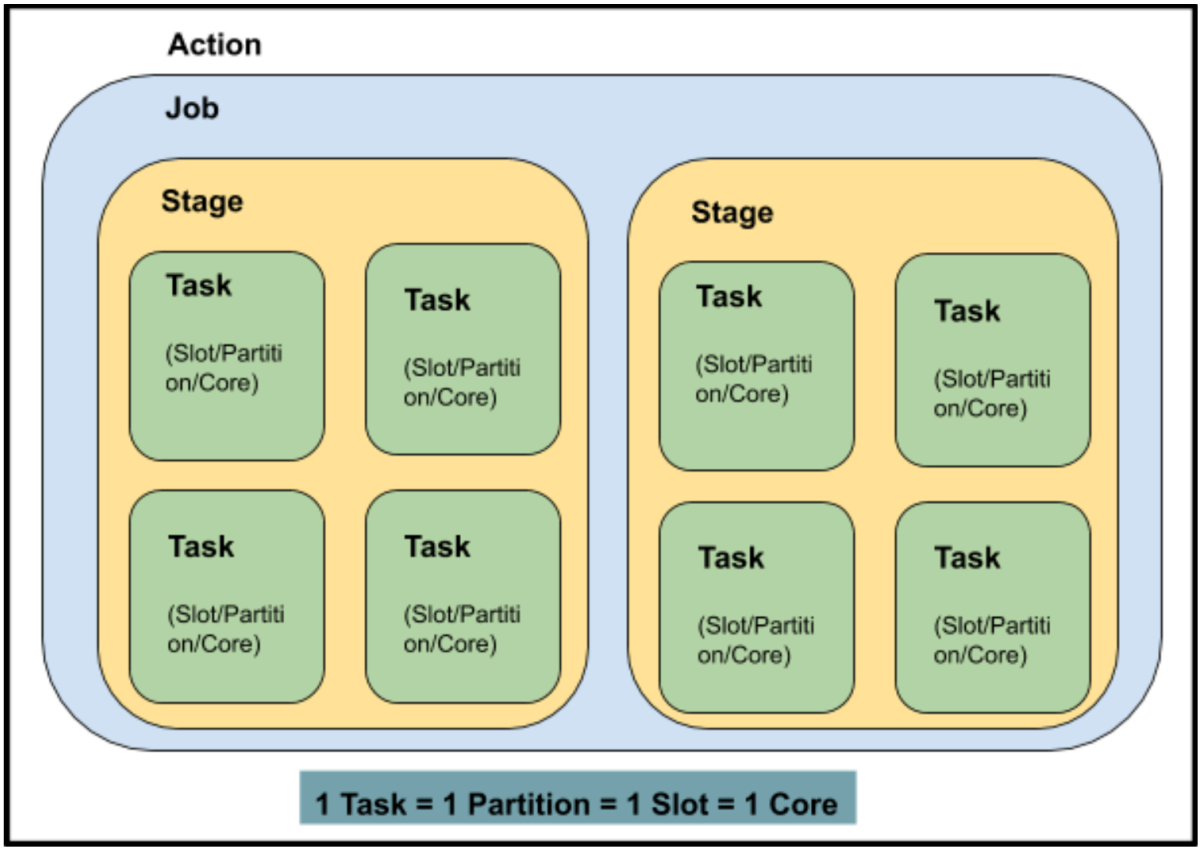
- Data
    - 6 Months of sample data (each file size ~ 1 GB each)
    - After transformation, we needed to save them in Avro + Parquet format.
- **Spark 2.4.4** deployed on **Kubernetes** (our internal Infrastructure)
- Expected Processing Time within 3 Hours including all http calls
- Scalable
- Where is the Notebook !!!

The Spark job has the following steps:

- Loading CSV files from S3
- Aggregating data over 1-Minute & 5-Minutes windows
- Processing data by calling an HTTP API
- Writing data to S3 in Avro & Parquet

Background on Spark Transformations

At its core, Apache Spark creates multiple Jobs, which are triggered by either narrow/wide transformations aka actions. Each job comprises of multiple stages and each stage comprises of multiple tasks. A task is the lowest level of abstraction in Spark, and the only part that interacts with the hardware directly. So, a Task lives inside an Executor **jvm**. Below is a representation of a Spark Action (or Transformation).



Representation of Spark Transformation

Determining Number of Partitions and Maximum Partition Size

After loading the entire input as dataframe, we checked the number of partitions -

```
dataframe.rdd.getNumPartitions
```

We found out that our partition counts were lower and were not fully utilizing the available cores. So boosting the parallelism was a natural choice which we did by reducing the input partition size.

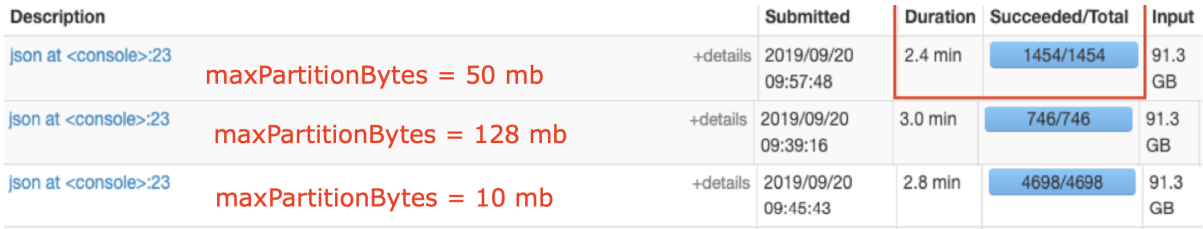
There are other reasons one might want to manipulate the input partition size. For example data generation in downstreaming like explode function (smaller initial partition can keep more space for

new data), complicated nested structure, Inefficient source data, compression/decompression and others.

Spark's default input partition size is **128mb** and it provides a nice mutable configuration to change the input partition bytes using ***spark.sql.files.maxPartitionBytes*****.** *So, we decided to lower than maxPartitionBytes size and that will lead us to use all the available cores properly and size of each partition was small enough to do some data generation-based operations.*

After trying out with different partitionBytes size, we found out that PartitionBytes size of 20 mb was performing the best and it finally raised the total rdd partitions count from 40 to 560.

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 1024 * 1024 * 20)
```



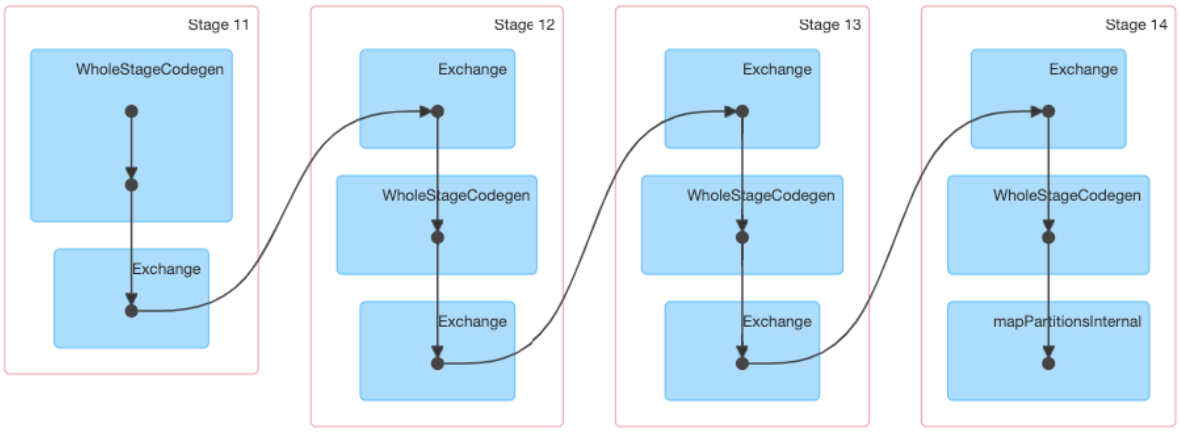
☝ *above you see 3 ways of loading the same data using different maxPartitionByte size. The Duration is the total time it took to load the full data (lower is better).*
A very low partitionByte size of 10 mb didn't improve the performance (2.8 Minutes Duration), but keeping it at 50 mb was the best (2.4 Minutes Duration), based on the available resources.

Setting the optimal *partitionByte as 60mb in our case* decreased the source data reading from 2.8 minutes to 1.8 minutes.

Determine the Number of Shuffle Partitions

As there were no changes made except partitionByte config in Spark, aggregation was performing poorly because of Spark Shuffling and that needed to be changed..

Shuffling is the process of data redistribution between stages or partitions. Certain operations within Spark trigger an event known as the shuffle. *The shuffle is Spark's mechanism for redistributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.*
The most significant optimization in Spark is to reduce Shuffling.



Above is one of the stages in our aggregation transformation showing data exchanges between executors/workers.
WholeStageCodegen is an adaption of a modern compiler technique to transform an entire query into a single function. The paper on WholeStageCodegen (☝image) is found here.
Exchange(☝image) here refers to Shuffle Exchange, which sends data across the network, which is the reason why Exchange is not using whole-stage code generation.

By default Spark Shuffle has 200 partitions. As far as I am aware there is no logic behind this number, and finding the best number of partitions proved to be one of the most significant steps for optimizing the pipeline for us.

Following is one of the equations which is recommended by community to calculate the number of shuffle partition-

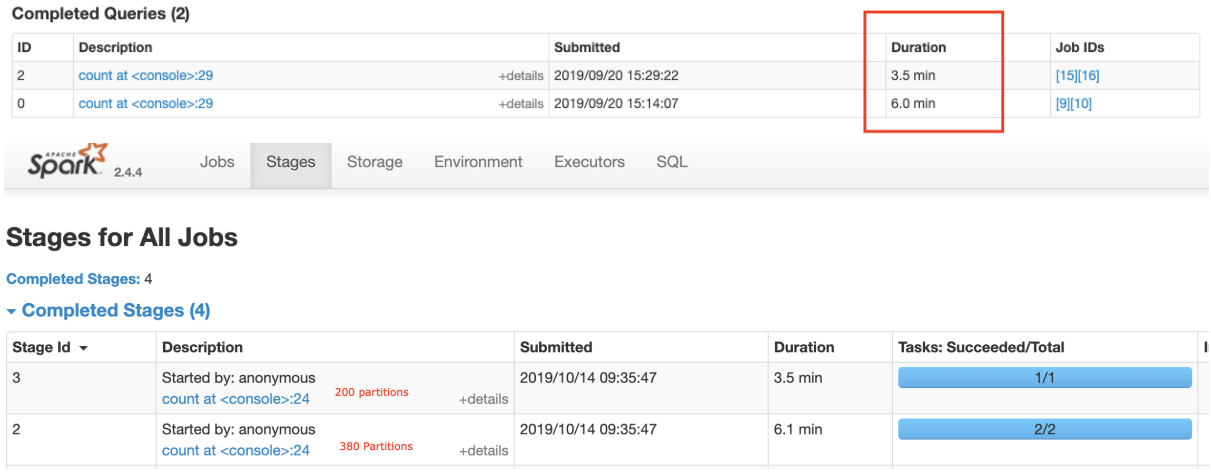
```
Shuffle Partition Count = Total Data Size / Target Size
Expected Partitions = Shuffle Partition Count / Total Cores Available

// Shuffle Partition should be the direct factor of Expected Partitions
Shuffle Partition Count = Shuffle Partition Count - Expected Partitions

// Special Case Only if Total Cores available is greater than Shuffle //
// Partition Count
Shuffle Partition Count = Total Cores Available
```

After finding the right number of shuffle partitions, which can be changed in the runtime by the following configuration, our aggregation step improved by almost 2x.

```
spark.conf.set("spark.sql.shuffle.partitions", 380)
```



Using mapPartitions for backend API call

We used mapPartition of the Spark dataframe class, and then called the serialized streaming api class. As mapPartitions runs separately on each partition (block) of the RDD, this is a non shuffling operation.
https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations

```
dataframe.mapPartitions(partition => Streaming Api Class(partition)).toDF
```

mapPartitions converts each partition of the source RDD into multiple elements of the result. Each of the partitions fit into memory easily so we need not redistribute the data across network again. We reduced the number of partitions using coalesce to control the concurrency of the calls to the external API.

Saving Data in Avro Format

As a final step we wanted to save the results as Avro. With the newly supported avro api in Spark, I assumed it will be straightforward but I was deploying on Kubernetes and was using Spark 2.4.3. The very first error I encountered when I tried to save the final result is

```
java.lang.UnsupportedOperationException: No Encoder found for java.time.Instant
```

After researching further I realized the fix is already available, but only from Spark 3.0 :-(.
https://issues.apache.org/jira/browse/SPARK-27222

As we couldn't **reuse** the already generated avro files using Kafka's schema registry, we generated new avro schemas for this job. In the new generated avro files all the fields were same except Timestamp, as timestamp is not supported, we changed it to Long. ~~This is in principle exactly the same, but only encoder related changes, and that brought some behaviour change in our code as well.~~
we will reset the avro after Spark 3.0, if we don't forget :-(.

In the end, how hard will it be to save the data by partition of Date. Well it turned out to be very very slow.

Writing Partitioned Data

Getting into more details about Spark writing, we realized assuming *Spark writing is Parallelized* is incorrect. Checking the source code of Spark writer clarified it better - https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/datasources/FileFormatWriter.scala#L85

So we found few ways which might improve spark writing (especially by partitions) -

1. Make sure to use all of the cores to write, so always keep track on the total size of your write and average file size.

```
Number of cores = 50
Total File Size to write = 100 GB
Expected File Partition Size = 10 GB
Actual Core usage = 100 GB/ 10 GB = 10 Cores
Unused Cores = 50 - 10 = 40 Cores
```

Bigger file size can be optimized using controlled partitions and records per file.

2. If there is Shuffle involved during write, reduce it. Default is 200, so one should reduce it further. Following is an example of setting partitions to 16.

```
spark.conf.set("spark.sql.shuffle.partitions", 16)
```

To ensure for not extending the number of records per file, use configuration

```
spark.conf.set("spark.sql.files.maxrecordsperfile”, 100000)
```

Use Version 2 of FileOutputCommitter

```
spark.conf.set("spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version", 2)
```

3. Try to use repartition with partitionBy, which keep one partition per file. Adding a stage barrier like *localCheckpoint* may help, but it's *optional* and it can be *costly* sometimes.

```
df.localCheckpoint
.repartition($"year",$"month", $"day")
.write
.format("avro")
.mode("overwrite")
.partitionBy("year","month", "day").save("s3a://path/to/save/partition")
```

4. To optimize writing further, we tried parallelism. This is code heavy way where we created list of filters and we iterate over the array which made the entire write operation parallel. ***This method performs better than standard filter only if one has many shuffle partitions***.

```
// Define Scala ForkJoin
val taskSupport = new ForkJoinTaskSupport(new ForkJoinPool(N))
case class PartitionBucket(year:Int, month: Int, day :Int)

// Create Array for Filter Condition
val buffer = ArrayBuffer[PartitionBucket]()
Range(1,13).toArray.foreach(month => {
        Range(1,31).toArray.foreach(day => parts.append(2019,month,day))
})
val partitionByMonth = buffer.toArray.par
partitionByMonth.tasksupport = taskSupport

// Iterate through the list to parallelize the entire operations and save parallely for each partition
partitionByMonth.foreach(part => {
        df2.filter('year === part.year && 'month === part.month && 'day === part.day)
.write.mode("append").partitionBy("year","month", "day").parquet("/path/to/save")
})
```

Finally, the spark job was done and all tests were passing along with integration tests. Then there was CI/CD and deployment in Kubernetes which is by itself another experience to share .

Reg *Where is the Notebook !!!* , I managed to setup Zeppelin on top of Kubernetes but that by itself needs a complete blog, so I will skip that topic here and will write a new blog for that 🙂.