Week5_Lab01_Simonsen

June 9, 2024

[ ]:
```
import pyspark.sql.functions as F

#Scale up executor memory from 1g
from pyspark.sql import SparkSession

spark = (SparkSession.builder
         .master("local[*]")
         .config("spark.executor.memory", "2g")
         .appName("Week5_Lab01_Simonsen")
         .getOrCreate()

        )
```

[ ]:
```
sc = spark.sparkContext
```

[ ]:
```
spark.conf.get('spark.executor.memory')
```

[ ]:
```
'2g'
```

[ ]:
```
# 1)        Use Spark to read in the airport-codes-na and departuredelays␣
 ↪datasets.

airport_codes_file = "dbfs:/FileStore/Merrimack/Week_5/airport_codes_na.txt"
codes_df = (spark.read
            .format("text")
            .option("delimiter", "\t")
            .option("header","true")
            .csv(airport_codes_file)
        )

codes_df.createOrReplaceTempView("codes_na")
```

[ ]:
```
spark.sql("SELECT * FROM codes_na LIMIT 10").show()
```

```
+-----------+-----+-------+----+
|       City|State|Country|IATA|
+-----------+-----+-------+----+
| Abbotsford|   BC| Canada| YXX|
```

```
|   Aberdeen|   SD|    USA| ABR|
|    Abilene|   TX|    USA| ABI|
|      Akron|   OH|    USA| CAK|
|    Alamosa|   CO|    USA| ALS|
|     Albany|   GA|    USA| ABY|
|     Albany|   NY|    USA| ALB|
|Albuquerque|   NM|    USA| ABQ|
| Alexandria|   LA|    USA| AEX|
|  Allentown|   PA|    USA| ABE|
+-----------+-----+-------+----+
```

[ ]:
```
departure_delays_file = "dbfs:/FileStore/Merrimack/Week_5/departuredelays.csv"

dep_delay_df = (spark.read.format("csv")
                .option("inferSchema", "true")
                .option("header", "true")
                .load(departure_delays_file)
            )

dep_delay_df = (dep_delay_df
          .withColumn("delay", F.expr("CAST(delay as INT) as delay"))
          .withColumn("distance", F.expr("CAST(distance as INT) as distance"))
          )

dep_delay_df.createOrReplaceTempView("dep_delay_na")
```

[ ]:
```
spark.sql("SELECT * FROM dep_delay_na LIMIT 10").show()
```

```
+-------+-----+--------+------+-----------+
|   date|delay|distance|origin|destination|
+-------+-----+--------+------+-----------+
|1011245|    6|     602|   ABE|        ATL|
|1020600|   -8|     369|   ABE|        DTW|
|1021245|   -2|     602|   ABE|        ATL|
|1020605|   -4|     602|   ABE|        ATL|
|1031245|   -4|     602|   ABE|        ATL|
|1030605|    0|     602|   ABE|        ATL|
|1041243|   10|     602|   ABE|        ATL|
|1040605|   28|     602|   ABE|        ATL|
|1051245|   88|     602|   ABE|        ATL|
|1050605|    9|     602|   ABE|        ATL|
+-------+-----+--------+------+-----------+
```

[ ]:
```
# 2)        Join the airport codes dataset to the departuredelays dataset.

dep_delay_df.join(codes_df, dep_delay_df['origin'] == codes_df['IATA'], 'inner')\
```

```
    .show(5, False)
```

```
+-------+-----+--------+------+-----------+---------+-----+-------+----+
|date   |delay|distance|origin|destination|City     |State|Country|IATA|
+-------+-----+--------+------+-----------+---------+-----+-------+----+
|1011245|6    |602     |ABE   |ATL        |Allentown|PA   |USA    |ABE |
|1020600|-8   |369     |ABE   |DTW        |Allentown|PA   |USA    |ABE |
|1021245|-2   |602     |ABE   |ATL        |Allentown|PA   |USA    |ABE |
|1020605|-4   |602     |ABE   |ATL        |Allentown|PA   |USA    |ABE |
|1031245|-4   |602     |ABE   |ATL        |Allentown|PA   |USA    |ABE |
+-------+-----+--------+------+-----------+---------+-----+-------+----+
only showing top 5 rows
```

[ ]:
```
# 3)        What type of Spark join would be best for this? Do you need to set␣
 ↪any configuration parameters to complete the join?

print(sc.defaultParallelism)
print(spark.conf.get('spark.executor.memory'))

'''
The spark join that woud be best for this would be a Broadcast Hash Join. Given␣
 ↪that the airport-codes-na dataset is relatively small in comparison to the␣
 ↪departuredelays dataset, this type of join makes sense. By using this type of␣
 ↪join, spark broadcasts the small dataframe (airport-codes-na) to each worker,␣
 ↪where it is then joined to the larger dataframe (departuredelays) at each␣
 ↪worker. This eliminates the need for complex shuffles, and is much less of an␣
 ↪expensive operation than say, a shuffle-sort merge join.

For this type of a join, I did not need to set any configuration parameters. Per␣
 ↪the above and in checking my executors tab within the Spark UI, I have minimal␣
 ↪tasks (currently 10 at the time of writing this) that I am running on 1␣
 ↪executor, which is also the driver. I have 4 cores allocated to the driver.␣
 ↪So, in this case, the number of cores multipled by number of executors is 4,␣
 ↪which is already set to the correct value in listing the default.parallelism␣
 ↪value above. If I noticed my jobs were creating "resource-hungry" workloads by␣
 ↪creating a lot of executors, I could set the cluster value for spark.
 ↪dynamicAllocation.maxExecutors to say, 5 for example.
'''
```

```
4
2g
```

[ ]:
```
'\nThe spark join that woud be best for this would be a Broadcast Hash Join.
Given that the airport-codes-na dataset is relatively small in comparison to the
departuredelays dataset, this type of join makes sense. By using this type of
join, spark broadcasts the small dataframe (airport-codes-na) to each worker,
where it is then joined to the larger dataframe (departuredelays) at each
```

worker. This eliminates the need for complex shuffles, and is much less of an expensive operation than say, a shuffle-sort merge join. \n\nFor this type of a join, I did not need to set any configuration parameters. Per the above and in checking my executors tab within the Spark UI, I have minimal tasks (currently 10 at the time of writing this) that I am running on 1 executor, which is also the driver. I have 4 cores allocated to the driver. So, in this case, the number of cores multipled by number of executors is 4, which is already set to the correct value in listing the default.parallelism value above. If I noticed my jobs were creating "resource-hungry" workloads by creating a lot of executors, I could set the cluster value for spark.dynamicAllocation.maxExecutors to say, 5 for example.\n'

[ ]:
```
# 4) What would a logical partition for this dataset be? You can use methods␣
 ↪like aggregation and count to determine if there are any logical partitions.
(dep_delay_df
    .groupBy("Origin")
    .count()
    .orderBy("count", ascending=False)
    .show(n=10, truncate=False)
)

#Based on my code, origin would be a logical partition for this dataset.
```

```
+------+-----+
|Origin|count|
+------+-----+
|ATL   |91484|
|DFW   |68482|
|ORD   |64228|
|LAX   |54086|
|DEN   |53148|
|IAH   |43361|
|PHX   |40155|
|SFO   |39483|
|LAS   |33107|
|CLT   |28402|
+------+-----+
only showing top 10 rows
```

[ ]:
```
#5)        Write the combined data back to your data directory using the␣
 ↪partition key you determined in the previous step.
# a.        Write the data in a directory using parquet.
# b.        Write the data in a directory using ORC.
# c.        What are the benefits/disadvantages of using ORC or Parquet for␣
 ↪reading/writing data?
```

```
#a
path = "dbfs:/FileStore/Merrimack/Week_5/parquet_data"

(dep_delay_df
 .repartition("origin")
 .write.format('parquet')
 .partitionBy('origin')
 .mode('overwrite')
 .option("header", "true")
 .save(path)
)

#b
path_orc = "dbfs:/FileStore/Merrimack/Week_5/orc_data"

(dep_delay_df
 .repartition("origin")
 .write.format('orc')
 .partitionBy('origin')
 .mode('overwrite')
 .option("header", "true")
 .save(path_orc)
)

#c
'''
ORC: One of the benefits to using ORC is the compression to shrink file size. Of␣
 ↪all the most commonly used data storage formats (Parquet, ORC, AVRO, Text/
 ↪CSV), ORC uses the best compression algorithm. Additionally, ORC is commonly␣
 ↪known to perform very within Hadoop/Hive envrionments with large amount of␣
 ↪data stored on disk. However, this also could be viewed as a drawback to using␣
 ↪ORC. Many organizations are tranisitioning to a cloud-based data architecture,␣
 ↪and ORC does not generally perform as well as Parquet in a Spark environment.␣
 ↪Additionally, the ORC does not offer the schema resolution flexibility of␣
 ↪AVRO, and it is generally a bit more difficult to change and adjust the schema␣
 ↪within the data.
```

```
Parquet: Similar to ORC files, Parquet also uses good compression algorithms in␣
 ↪comparison across all commonly used data storage formats. While Parquet does␣
 ↪not have quite as good of file compression as ORC, it generally performs best␣
 ↪in Spark envrionments. Given the rise in popularity of Spark because of its␣
 ↪ability to utilize memory-based computation as opposed to disk, this is␣
 ↪appealing for many organizations. However, outside of the Spark envrionment,␣
 ↪Parquet data storage formatting may not be the best performing option.␣
 ↪Additionally, Spark also does not offer the schema resolution flexibility that␣
 ↪AVRO does. All things considered, so long as Parquet continues to perform best␣
 ↪in Spark of all the commonly used data storage formats, it will continue to be␣
 ↪a widely popular and commonly used option.
'''
```

[ ]:
```
'\nORC: One of the benefits to using ORC is the compression to shrink file size.
Of all the most commonly used data storage formats (Parquet, ORC, AVRO,
Text/CSV), ORC uses the best compression algorithm. Additionally, ORC is
commonly known to perform very within Hadoop/Hive envrionments with large amount
of data stored on disk. However, this also could be viewed as a drawback to
using ORC. Many organizations are tranisitioning to a cloud-based data
architecture, and ORC does not generally perform as well as Parquet in a Spark
environment. Additionally, the ORC does not offer the schema resolution
flexibility of AVRO, and it is generally a bit more difficult to change and
adjust the schema within the data.\n\nParquet: Similar to ORC files, Parquet
also uses good compression algorithms in comparison across all commonly used
data storage formats. While Parquet does not have quite as good of file
compression as ORC, it generally performs best in Spark envrionments. Given the
rise in popularity of Spark because of its ability to utilize memory-based
computation as opposed to disk, this is appealing for many organizations.
However, outside of the Spark envrionment, Parquet data storage formatting may
not be the best performing option. Additionally, Spark also does not offer the
schema resolution flexibility that AVRO does. All things considered, so long as
Parquet continues to perform best in Spark of all the commonly used data storage
formats, it will continue to be a widely popular and commonly used option.\n'
```

[ ]:
```
# 6)        What if any configuration changes did you set to tune your job?
# a.        As this process scales up, what parameters do you think you would␣
 ↪need to optimize your job?

'''
I modified the executor memory to 2GB instead of the default value of 1GB␣
 ↪assigned. Per the above, the other configuration values I checked such as␣
 ↪default parallelism and number of executors seemed appropriately configured␣
 ↪for the job and tasks I was running.
```

```
As this process scales up, I would expect to have to adjust certain parameters␣
 ↪to optimize my job. First, I would expect to adjust the min and max number of␣
 ↪executors to be sure I am appropriately scaling for the job. More than likely,␣
 ↪I would want to set the spark.dynamicAllocation.minExecutors to 2 to be sure I␣
 ↪at least have 2 workers executing a minimum. I would also want to set the␣
 ↪spark.dynamicAllocation.maxExecutors to a higher value, say 20 at the most, to␣
 ↪avoid Databricks allocating an infinite number of workers to the job, which␣
 ↪would be very costly. Setting the spark.dynamicAllocation.executorIdleTimeout␣
 ↪to a value of a few minutes, potentially 2, would also be a good idea to avoid␣
 ↪idle workers running without tasks. A lot of this may also depend on how many␣
 ↪pertitions I'm bringing in, and how resource intensive the jobs I'm running␣
 ↪are (i.e. aggregations, calculations, etc.). Finally, it may be necessary to␣
 ↪increase values if shuffles are occuring, such as increasing the spark.shuffle.
 ↪unsafe.file.output.buffer to adjust for merging files, increasing the value of␣
 ↪spark.shuffle.registration.timeout to increase the time until timeout, and␣
 ↪also increaseing the spark.shuffle.file.buffer to allow Spark to buffer more␣
 ↪before writing data back to disk.
'''
```

[ ]:
```
"\nI modified the executor memory to 2GB instead of the default value of 1GB
assigned. Per the above, the other configuration values I checked such as
default parallelism and number of executors seemed appropriately configured for
the job and tasks I was running. \n\nAs this process scales up, I would expect
to have to adjust certain parameters to optimize my job. First, I would expect
to adjust the min and max number of executors to be sure I am appropriately
scaling for the job. More than likely, I would want to set the
spark.dynamicAllocation.minExecutors to 2 to be sure I at least have 2 workers
executing a minimum. I would also want to set the
spark.dynamicAllocation.maxExecutors to a higher value, say 20 at the most, to
avoid Databricks allocating an infinite number of workers to the job, which
would be very costly. Setting the spark.dynamicAllocation.executorIdleTimeout to
a value of a few minutes, potentially 2, would also be a good idea to avoid idle
workers running without tasks. A lot of this may also depend on how many
pertitions I'm bringing in, and how resource intensive the jobs I'm running are
(i.e. aggregations, calculations, etc.). Finally, it may be necessary to
increase values if shuffles are occuring, such as increasing the
spark.shuffle.unsafe.file.output.buffer to adjust for merging files, increasing
the value of spark.shuffle.registration.timeout to increase the time until
timeout, and also increaseing the spark.shuffle.file.buffer to allow Spark to
buffer more before writing data back to disk.\n"
```