6. Spark Pools in Synapse

Note I should emphasize that I am skipping sections about introduction to Python and Scala from the Udemy tutorial and also not documenting every thing about the introduction to Spark Dataframes since I have worked with them. I will be documenting Azure Specific Spark integrations.

If you need a good introduction to Spark without installing Spark, you can use Google colab and follow this tutorial- https://jacobcelestine.com/knowledge_repo/colab_and_pyspark/

Spark is a parallel processing framework that can be used in big-data analytical applications. Synapse brings spark into the picture your big data processing needs and can use spark pols to process the data that is stored in Azure.

In the spark architecture we have the driver node, that accepts the incoming spark jobs and then distributed the work on to the executors that would execute the task. Similar to the SQL Pool where we would have a control node which was responsible for the compute nodes.

To process data what we need to do first is create a spark pool.

Go to your Synapse and from the left, select spark pools and click new.

- Give the pool name, the node size (we pick the smallest one), autoscale (disabled for us), and the minimum number of nodes that is 3.

Note – This would be a serverless spark pool meaning we'd not be charged based on the creation of the pool, we'd be charged based on the underlying nodes starting up to process your job.

- So, in our case one node would be the driver node and the remaining two nodes would be the executors.
- On the next screen, the automatic pausing feature would help when the nodes are not being used for a certain amount of time, the nodes would be paused and it would save the compute costs. I selected 15 minutes, hit review and create.

Working with Notebooks

Go to your Synapse Studio, click on new and Notebook. Name your notebook on the left and select your spark pool. Select your language as Scala (we won't be doing heavy work in Scala and the work with spark is the same in Python.

We use the following code to create an RDD –

```
val data = Array(1, 2, 3, 4, 5)
// The parallelize method of the Spark context will create an RDD
val dist = sc.parallelize(data)
// To get the count of values in the RDD
dist.count()
// If you want to get the elements of the RDD
dist.collect()
```

In the above operation, after the sc.parallelize operation spark would not have created an RDD as this is what is called a lazy operation. It would only create the RDD when some action is asked of Spark on the RDD.

Working with Dataframes -

Like a table in a relational db.

To convert an existing RDD to a dataframe we would –

```
val df=dist.toDF()
```

To sort a dataframe based upon a certain column –

We create the dataframe and then sort it based on a certain column.

```
from pyspark.sql.functions import desc
from pyspark.sql.functions import col

courses = [(1,'AZ-900',10.99),(2,'DP-203',11.99) ,(3,'AZ-104',12.99)]
df = spark.createDataFrame(courses, ['Id', 'Name', 'Price'])
sorteddf=df.sort(col("Price").desc())
sorteddf.show()
```

Loading log.csv on the spark dataframe from out data lake

We list out the account name, container name and the relative path to the file. Save that information as the adls path. Access the data with the access key for the data lake.

We copy the first half of the code to a cell –

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

account_name = ""
container_name = ""
relative_path = ""
adls_path = 'abfss://%s@%s.dfs.core.windows.net/%s' % (container_name, account_name, relative_path)

spark.conf.set("fs.azure.account.auth.type.%s.dfs.core.windows.net" %account_name, "SharedKey")
spark.conf.set("fs.azure.account.key.%s.dfs.core.windows.net" %account_name ,"Key1 from access keys")

df1 = spark.read.option('header', 'true') \
                .option('delimiter', ',') \
                .csv(adls_path + '/Log.csv')

display(df1)
```

Aggregation functions using Group by

We can use this code as an example –

```
from pyspark.sql.functions import *

# .agg is a method that can be used to perform aggregation based on a column of data
# .orderBy helps to order by a particular column
newdf=(df1.groupBy("Operationname")
     .agg(count("Correlationid").alias("Total operations"))
     .orderBy(col("Total operations").desc()))

display(newdf)
```

Now this data can also be displayed in a visual manner by switching to the Chart tab. If the two keys have been identified by Azure it would show you a graph but if it hasn't then you can select the two keys by clicking the button on the top right.

Top right button –

If you want values that are not null, then you can use this code in a new cell –

```
filterdf=newdf.filter(col("Operationname").isNotNull())
```

```
display(filterdf)
```

Using SQL statements on your data

As in example we can use this code –

```
df1.createOrReplaceTempView("logdata")

sql_1=spark.sql("SELECT Operationname, count(Operationname) FROM logdata GROUP BY Operationname")
sql_1.show()
```

You can also use direct SQL based language to query the data –

We can either put %%sql before our code for the notebook to execute the cell as a Spark SQL cell or we just just select Spark SQL from the top right.

Example –

```
%%sql
SELECT Operationname, count(Operationname) FROM logdata GROUP BY Operationname
```

Reading data from the data lake using the inbuild connector present in Azure Synapse to copy the data to Dedicated SQL pool

This method currently only works with scala.

- We begin by importing the following libraries

```
import com.microsoft.spark.sqlanalytics.utils.Constants
import org.apache.spark.sql.SqlAnalyticsConnector._
```

- Now we create the schema for the data that we are about to read

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._


val dataSchema = StructType(Array(
    StructField("Id", IntegerType, true),
    StructField("Correlationid", StringType, true),
    StructField("Operationname", StringType, true),
    StructField("Status", StringType, true),
    StructField("Eventcategory", StringType, true),
    StructField("Level", StringType, true),
    StructField("Time", TimestampType, true),
    StructField("Subscription", StringType, true),
    StructField("Eventinitiatedby", StringType, true),
    StructField("Resourcetype", StringType, true),
    StructField("Resourcegroup", StringType, true)))
```

- Next, we used the existing linked connection to our data lake and read the data –

```
val df =
spark.read.format("csv").option("header","true").schema(dataSchema).load("abfss://data@datalake2000.dfs.core.windo
ws.net/raw/Log.csv")
df.printSchema()
```

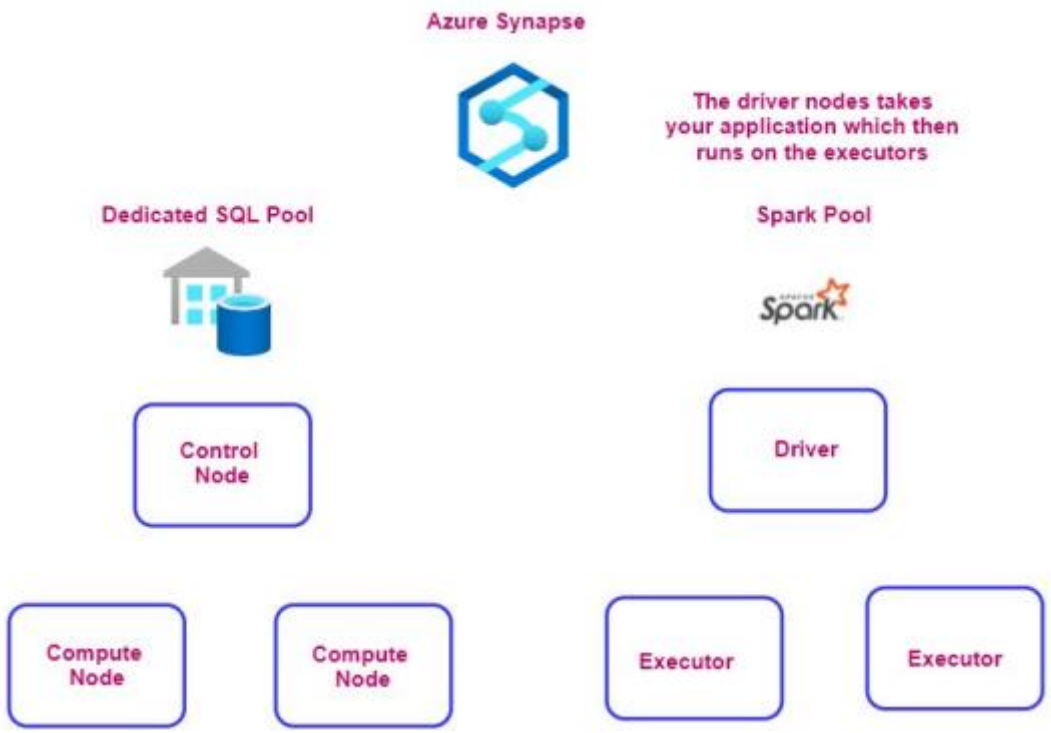
- If the above doesn't work, your data lake is not linked to your Azure Synapse and you you need to link it.

- Next, we write the data onto the dedicated sql pool –

```
import com.microsoft.spark.sqlanalytics.utils.Constants
import org.apache.spark.sql.SqlAnalyticsConnector._
df.write.sqlanalytics("DATABASE NAME.dbo.NAME OF THE TABLE", Constants.INTERNAL)
```

- There are a few constraints about the above command to work –
  1. The table should not exist in the dedicated sql pool
  2. The transfer of data would happen in the context of the Azure Admin user. The user must be added as an Azure AD user in the Azure Synapse and the same user must have the Storage Blob Contributor Role on the storage account attached to the Synapse workspace
- Data shows up on the dedicated sql pool

The general architecture of the Azure Synapse –



So we can see both the dedicate sql pool and the spark pool work in a similar manner. With the spark pool however, the spark instances are created when you connect to the spark pool, create a session and then run a job. Another spark instance can be created only if there are spare nodes remaining.

Creating Spark tables

We can create tables and store them in the metastore of the Spark pool (whatever that is). An example of how to do it is –

```
val df = spark.read.sqlanalytics("newpool.dbo.logdata")
df.write.mode("overwrite").saveAsTable("logdatainternal")
```

Read it as –

```
%%sql
SELECT * FROM logdatainternal
```

This is useful if you want to have temporary tables in place to do analysis. Ideally the tables would be stored in the dedicated sql pool. But the benefit of it is the metastore is also shared with the **serverless sql pool** as well.

Go to your Synapse studio and open up the sql editor. Select Built-in for the connect to option to connect to the serveless sql and default as the database to refer to the metastore. And run the above query again and the data would show up.

Creating tables and databases

The spark tables are parquet backed tables. We can begin by creating the database

```
%%sql
CREATE DATABASE internaldb
```

- Then create the table

```
CREATE TABLE internaldb.customer(Id int,name varchar(200)) USING Parquet
```

- Insert data like

```
INSERT INTO internaldb.customer VALUES(1,'UserA')
```

- Query it like

```
SELECT * FROM internaldb.customer
```

- We can also read our log.csv file into a dataframe and write that into a table

```
%%pyspark
df = spark.read.load('abfss://data@DATA LAKE.dfs.core.windows.net/raw/Log.csv', format='csv'
, header=True
)
df.write.mode("overwrite").saveAsTable("internaldb.logdatanew")
```

- To delete the database, you'd have to delete the tables and then the database

Reading JSON files

The main thing to note is we are using the explode function while displaying the array because one of the json objects has multiple values like Courses in following json file.

The json file looks like this –

[

{"customerid":1,"customername":"UserA","registered":true,"courses":["AZ-900","AZ-500","AZ-303"]},

{"customerid":2,"customername":"UserB","registered":true,"courses":["AZ-104","AZ-500","DP-200"]}

]

The courses column has multiple

```
%%spark
import org.apache.spark.sql.functions._
val df =
spark.read.format("json").load("abfss://data@DATALAKE.dfs.core.windows.net/raw/customer/customer_a
rr.json")
val
newdf=df.select(col("customerid"),col("customername"),col("registered"),explode(col("courses")))
```

```
display(newdf)
```

Similarly, if there are objects inside other objects we would use the same approach. For a json file that looks like this –

{"customerid":1,"customername":"UserA","registered":true,"courses":["AZ-900","AZ-500","AZ-303"],"details" :
{"mobile":"111-1112","city":"CityA"}}

{"customerid":2,"customername":"UserB","registered":true,"courses":["AZ-104","AZ-500","DP-200"],"details":{"mobile":"333-1112","city":"CityB"}}

In the above file, details object has mobile and city objects. So we would read it like –

```
%%spark
import org.apache.spark.sql.functions._
val df =
spark.read.format("json").load("abfss://data@datalake2000.dfs.core.windows.net/raw/customer/customer_obj.json")
val
newdf=df.select(col("customerid"),col("customername"),col("registered"),explode(col("courses")),col("details.city"),col("details.mobile"))
display(newdf)
```