7. Azure Databricks

Why do we need Databricks?

Let's say we need Apache Spark for our Data processing needs. For this, we would need to provision the machines, install spark and the necessary libraries, maintain the scaling and the availability of the machine. With Databricks the entire environment can be provisioned with just a few clicks.

Databricks would create the underlying compute infrastructure for you. In addition to the servers, it has it's own underlying file system which is an abstraction of the underlying storage layer. Would also install spark and other libraries.

In addition, there is a workspace provided where users can work with Notebooks and users can then collaborate on Notebooks.

Important concepts about Databricks

In databricks, we would create clusters in what we call a workspace. This cluster would have Spark and other components installed. Because it is a spark clusters, it would follow the same architecture of a master/slave nodes.

There are two types of clusters –

Interactive clusters	Job cluster
• Here we can analyze our data our data with the help of interactive notebooks • Multiple users can collaborate on the cluster	• If the only purpose is to run a job then we would use this. • When a job is to be run, databricks would start the cluster and when the job is complete, the cluster will be terminated

There are again two types of interactive clusters –

Standard Cluster	High concurrency cluster
• This is recommended if you are a single user • Here there is not fault isolation. If multiple users are using a cluster and one has a fault, it can impact the workloads of other users • Here the resources of the cluster might get allocated to a single workload • Had support for Python, R, Scala, SQL	• This is recommended for multiple users • We have fault isolation here • Here the resources of the cluster are effectively shared across different user workload • Has support for Python, R and SQL • Table access control, here you can grand the revoke access to data from Python and SQL

Creating a workspace and then creating a cluster

Go to create resources, search for Azure Databricks and click create.

- Give a name, workspace name, region, pricing tier as Trial. Leave everything as it is and click review and create.
- Once the resource is created, we can launch the workspace. We can create clusters, notebooks, spark table and a spark job using this workspace.

Creating a cluster – This cluster would have the underlying machines that would have spark installed. For the Data Engineering exam, the questions could either be about Spark Dataframes or about the Clusters in Databricks.

Now, since Azure Synapse and Databricks both give us the option of having internally manage spark clusters, which one would we choose?

When it comes to the spark pool in Azure Synapse, it is Microsoft's own implementation of Apache Spark to tie up the entire Spark with the Synapse ecosystem. But in the case of Azure databricks, the databricks team are responsible for the underlying spark engine. They are developing the entire ecosystem that is directed towards data engineering and data science in once complete package under Databricks.

One obvious difference between the two implementations is the version of spark that you get to use. In Databricks you get Spark 3+ where as in the Spark pool for Azure Synapse you would get just version 2.4.

Let's begin by clicking new cluster –

- Give the cluster a name, next cluster mode (important to know for the exam) as standard.

Cluster Mode has three options – high concurrency, standard, single node. When we have single users working on the cluster developing notebooks, then they can use the standard mode. When the standard mode is chosen there are some other options that are also available for it.

- *First, the ability to terminate the cluster after a certain number of minutes of activity.*
- *Next, we also have the capability of autoscaling. We have a minimum of 2 workers. Let's say there's a very large dataset that is being processed, the two nodes are not enough. So with the help of autoscaling databricks is able to spin up a new worker in the cluster to help with the job.*

High Concurrency cluster – allows you to have many users on the cluster. Terminate after option is not enabled. Lastly, there is single node cluster which we will choose This acts as both the driver node and the worker node.

- Select autoscaling and terminate after X number of minutes of inactivity
- Next we select the worker type. This would determine the memory and the cores of the machine that would be used as your worker and the driver node.

In databricks we get charged by the virtual machines that we provision and the databricks units based on the VM instance selected.

- Create the cluster with the single node cluster option.

Notebook with the cluster

We create a notebook from the left hand plus button.

- We select the default language as Scala and cluster as cluster203

We start with a simple code –

```
// Lab - Simple notebook
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()
```

Dataframes on the cluster

We have a sequence with some information, we then are creating a RDD with it and then converting it to a Dataframe.

```
val data=Seq((1,"DP-203",9.99),(1,"AI-102",10.99),(1,"AZ-204",11.99))

// We can then create an RDD from the sequence
val dataRDD=sc.parallelize(data)

// From the output of the RDD , you will see the data types are being automatically inferred
```

```
// Then we can convert the RDD to a data frame
val df=dataRDD.toDF()
display(df)
```

We get the following output. Note that the column headers are autogenerated –

	_1	_2	_3
1	1	DP-203	9.99
2	1	AI-102	10.99
3	1	AZ-204	11.99

Now we are going to define a schema for our data. We are using StructType to have different StructTypes which would be the fields for your dataset. And if we are creating a dataframe from our sequence, we have to use the Row type while creating the sequence.

The code that we would use is –

```
// The import statement is needed to reference the Types in the StructField class
import org.apache.spark.sql.types._

// Reference for the data types -
https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/sql/types/DataTypes.html
val schema = StructType( Array(
                StructField("Course ID", IntegerType),
                StructField("Course Name", StringType),
                StructField("Course price", DoubleType),
            ))

// Here our data needs to be of the Row type
val dataRows=Seq(
    Row(1,"DP-203",9.99),
    Row(2,"AI-102",10.99),
    Row(3,"AZ-204",11.99))

val newdf=spark.createDataFrame(sc.parallelize(dataRows),schema)
display(newdf)
```

To sort the newdf by the course price in descending order –

```
display(newdf.sort(newdf.col("Course price").desc))
// OR Can also use this way of selecting columns
display( newdf.sort($"Course price".desc)
```

Filtering based on the where condition –

```
import org.apache.spark.sql.functions._
val filterdf=newdf.where($"Course Name"==="DP-203")
display(filterdf)
// OR
display(newdf.select($"Course Name" === "DP-203"))
```

Aggregating –

```
val priceavg=newdf.agg("Course price"->"avg")
display(priceavg)
```

Reading the Log.csv file

Delete all the cells that are open.

Now click on the File button at the top of the notebook and click upload Data. Now we can upload our log.csv file onto Azure databricks. It has an underlying filesystem in place. So if you want to work with files locally, you can do so by uploading them. The databricks can also connect to the storage account and also create mount points onto those storage accounts but we'll do that later.

So we select he Log.csv file to upload. Upon going to next, we see the location of the file on the left and the syntax to create a dataframe with the file on the right. So we copy the command on the right and paste it into a new cell.

So the command for me was –

val df1 = spark.read.format("csv").load("dbfs:/FileStore/shared_uploads/ray.aroun@live.com/Log.csv")

We can then display the dataframe to see that the column headers are also inside the dataframe as values.

	_c0	_c1	_c2	_c3	_c4	_c5	_c6	_c7
1	Id	Correlationid	Operationname	Status	Eventcategory	Level	Time	Subscription
2	1	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Succeeded	Administrative	Informational	2021-06-15T04:44:38.223Z	20c6eec9-2d80
3	2	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Started	Administrative	Informational	2021-06-15T04:44:21.547Z	20c6eec9-2d80
4	3	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Accepted	Administrative	Informational	2021-06-15T04:44:21.702Z	20c6eec9-2d80
5	4	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Succeeded	Administrative	Informational	2021-06-15T04:44:31.332Z	20c6eec9-2d80
6	5	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Started	Administrative	Informational	2021-06-15T04:44:12.533Z	20c6eec9-2d80
7	6	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Accepted	Administrative	Informational	2021-06-15T04:44:16.038Z	20c6eec9-2d80

Truncated results, showing first 1000 rows.

So a way to fix it is to read the file with the .option("header","true") before the .load()
Now upload displaying the new dataframe with the header option as true, we see the column headers.

	Id	Correlationid	Operationname	Status	Eventcategory	Level	Time	Subscriptio
1	1	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Succeeded	Administrative	Informational	2021-06-15T04:44:38.223Z	20c6eec9-2c
2	2	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Started	Administrative	Informational	2021-06-15T04:44:21.547Z	20c6eec9-2c
3	3	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Accepted	Administrative	Informational	2021-06-15T04:44:21.702Z	20c6eec9-2c
4	4	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Succeeded	Administrative	Informational	2021-06-15T04:44:31.332Z	20c6eec9-2c
5	5	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Started	Administrative	Informational	2021-06-15T04:44:12.533Z	20c6eec9-2c
6	6	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Accepted	Administrative	Informational	2021-06-15T04:44:16.038Z	20c6eec9-2c
7	7	08cd2e19-477c-4ecc-83a6-575b9ce265e3	Pause SQL Analytics pools.	Succeeded	Administrative	Informational	2021-06-14T17:57:02.240Z	20c6eec9-2c

Truncated results, showing first 1000 rows.

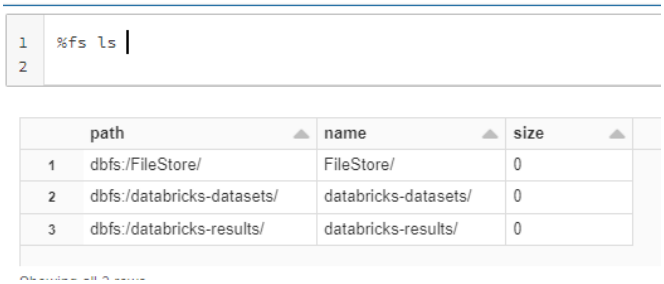
The Databricks file system

This is an abstraction layer on top of the scalable object storage. So to interact with this storage we have the databricks file system. We can use directories and other file semantics. These files would also persist even if the cluster is terminated. The default storage location is the DBFS root.

There are some predefined root locations –

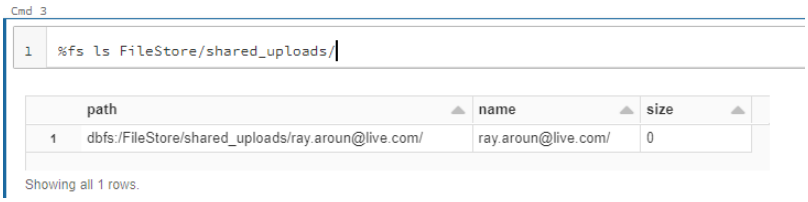
- /FileStore : imported data files, generated plots, uploaded libraries
- /databricks-datasets : sample public datasets
- /user/hive/warehouse : data and metadata for non-external Hiive tables

To look at the database file system, we can actually use some commands in the notebook cells. To list the contents of the filesystem we can simply type – %fs ls



	path	name	size
1	dbfs:/FileStore/	FileStore/	0
2	dbfs:/databricks-datasets/	databricks-datasets/	0
3	dbfs:/databricks-results/	databricks-results/	0

This would show the three root locations in the file system and then you can keep changing the folder in front of the ls command to see the contents of the location you have put.



We also can create a new folder in the desired location using the command – %fs mkdirs location/FOLDER_NAME

Using the SQL API on the dataframe

We are reusing the dataframe the above which had the correct headers.

When we read the data without a particular schema all the columns are read as strings.

```
1 df2.printSchema

root
 |-- Id: string (nullable = true)
 |-- Correlationid: string (nullable = true)
 |-- Operationname: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- Eventcategory: string (nullable = true)
 |-- Level: string (nullable = true)
 |-- Time: string (nullable = true)
 |-- Subscription: string (nullable = true)
 |-- Eventinitiatedby: string (nullable = true)
 |-- Resourcetype: string (nullable = true)
 |-- Resourcegroup: string (nullable = true)
```

We can re-read the file with the option Map("inferSchema" -> "true" , "header" -> "true"). This is the complete command –

```
val df2 = spark.read.format("csv").options(Map("inferSchema"->"true","header"->"true"))load("dbfs:/// Log.csv")
```

Now we can see the schema is according to the types of the values present in the column.

Just like above we can use the filter functionality –

```
display(df2.filter(df2("Status")==="Succeeded"))
```

Lastly, the groupby statement –

```
display(df2.groupBy(df2("Status")).count())
```

Note – There is also a visualization functionality present for the results the tables that are displayed. We can access this with the bar chart button on the bottom left of the result. Upon clicking we can see a graph generated and we can also tinker around with what values to display using the plot options.

Date functions –

With the following command we are displaying the year, month and the day of the year of the time column that we have.

```
display(df2.select(year(col("time")),month(col("time")),dayofyear(col("time"))))
```

We can also give aliases to our output to have meaningful headers –

```
display(df2.select(year(col("time")).alias("Year"),month(col("time")).alias("Month"),dayofyear(col("time")).alias(
"Day of Year")))
```

We can also convert the date to a particular format using the to_date fucntion–

```
display(df2.select(to_date(col("time"),"dd-mm-yyyy").alias("Date")))
```

Filtering out null values from your results –

To get null value result set –

```
display(df2.filter(col("Resource group").isNull))
```

To remove null values from the results –

```
display(df2.filter(col("Resource group").isNotNull))
```

We can also use high level language inside filter –

display(df2.filter("Resourcegroup is not NULL"))

Reading Parquet Files

We first upload the parquet files that we had used earlier. After uploading, I copy the locations of the files that was provided to use with Spark.

After uploading I use the * operator to read all the parquet files together.

```
val dfparquet = spark.read.format("parquet").load("dbfs:/FileStore/shared_uploads/test/*")
display(dfparquet)
```

```
// .options(Map("inferSchema"->"true","header"->"true"))
val dfparquet = spark.read.format("parquet").load("dbfs:/FileStore/shared_uploads/test/*")
display(dfparquet)
```

(2) Spark Jobs

dfparquet: org.apache.spark.sql.DataFrame = [Id: integer, Correlationid: string ... 9 more fields]

	Id	Correlationid	Operationname	Status	Eventcategory	Level	Time	Subsc
1	1	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Succeeded	Administrative	Informational	2021-06-15T04:44:38.223+0000	20c6ee
2	2	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Started	Administrative	Informational	2021-06-15T04:44:21.547+0000	20c6ee
3	3	66641e13-d19f-4ce5-aafd-9d5d7befa557	Delete SQL database	Accepted	Administrative	Informational	2021-06-15T04:44:21.702+0000	20c6ee
4	4	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Succeeded	Administrative	Informational	2021-06-15T04:44:31.332+0000	20c6ee
5	5	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Started	Administrative	Informational	2021-06-15T04:44:12.533+0000	20c6ee
6	6	e2958162-93d9-4643-a847-82cf25c49930	Delete SqlPools	Accepted	Administrative	Informational	2021-06-15T04:44:16.038+0000	20c6ee
7	7	08cd2e19-477c-4ecc-83a6-575b9ce265e3	Pause SQL Analytics pools	Succeeded	Administrative	Informational	2021-06-14T17:57:02.240+0000	20c6ee

Truncated results, showing first 1000 rows.

Here the column headers are inferred without giving any option and even the schema is inferred properly.

Reading JSON files

Same procedure, upload these files and read them with the “json” format. We would try to see how to read json file where there are multiple values inside an object and there are multiple objects inside an object.

We can read using the same technique as reading a regular csv file but the data would not be displayed properly.

```
val objdf =
spark.read.format("json").load("dbfs:/FileStore/shared_uploads/ray.aroun@live.com/customer_obj.jso
n")
val arrdf =
spark.read.format("json").load("dbfs:/FileStore/shared_uploads/ray.aroun@live.com/customer_arr.jso
n")
```

objdf is showing up as –

```
display(objdf)
```

▸ (1) Spark Jobs

	courses	customerid	customername	details	registered
1	▸ ["AZ-900", "AZ-500", "AZ-303"]	1	UserA	▸ {"city": "CityA", "mobile": "111-1112"}	true
2	▸ ["AZ-104", "AZ-500", "DP-	2	UserB	▸ {"city": "CityB", "mobile": "333-	true

showing all 2 rows.

Like we used the explode function in the Synapse Spark Pool json file example, we can also do the same here.

```
val newjson=objdf.select(col("customerid"),col("customername"),col("registered"),explode(col("courses")))
display(newjson)
```

```
1  val newjson=objdf.select(col("customerid"),col("customername"),col("registered"),explode(col("courses")))
2
3  display(newjson)
```

▸ (1) Spark Jobs

▸ newjson: org.apache.spark.sql.DataFrame = [customerid: long, customername: string ... 2 more fields]

	customerid	customername	registered	col
1	1	UserA	true	AZ-900
2	1	UserA	true	AZ-500
3	1	UserA	true	AZ-303
4	2	UserB	true	AZ-104
5	2	UserB	true	AZ-500
6	2	UserB	true	DP-200

Similarly, the other file has objects inside an object so after exploding the object, we would mention the name of the columns inside.

Structured Streaming Data

We begin by understanding that is the type of data that would be streamed. We upload the PT1H.json file that we had used earlier. Copy the read statement generated and display the dataframe. Now we know the data can be read in a dataframe. Now we can proceed to read data from the event hubs and then store the data on the dedicated sql pool.

While learning about streaming we had created a dbhub event hub which was receiving diagnostic information from our adventureworks database. We would be reusing the same event hub for the purpose for receiving streaming data.

- To make this happen we would need an external library to be installed on to the cluster that we created. Open up the cluster from the compute section. Click on the libraries tab and click install new.
  When the new pop-up opens, click the maven tab and click search packages. From the drop down menu next to the search box, select Maven Central and search for azure-eventhubs-spark. From the results select the library with the Artifact Id azure-eventhubs-spark_2.21 and install.

Libraries can be installed for python, java, scala or R. There are different modes in which you can install libraries –

- *Workspace libraries – they serve as the local repository to create cluster installed libraries*
- *Cluster libraries – are installed on the cluster and are available for all the notebooks running on the cluster (we are doing this)*

- Once the library is installed, go back to the notebook click on the cluster name at the top and select detach and reattach option

- We use the following script –

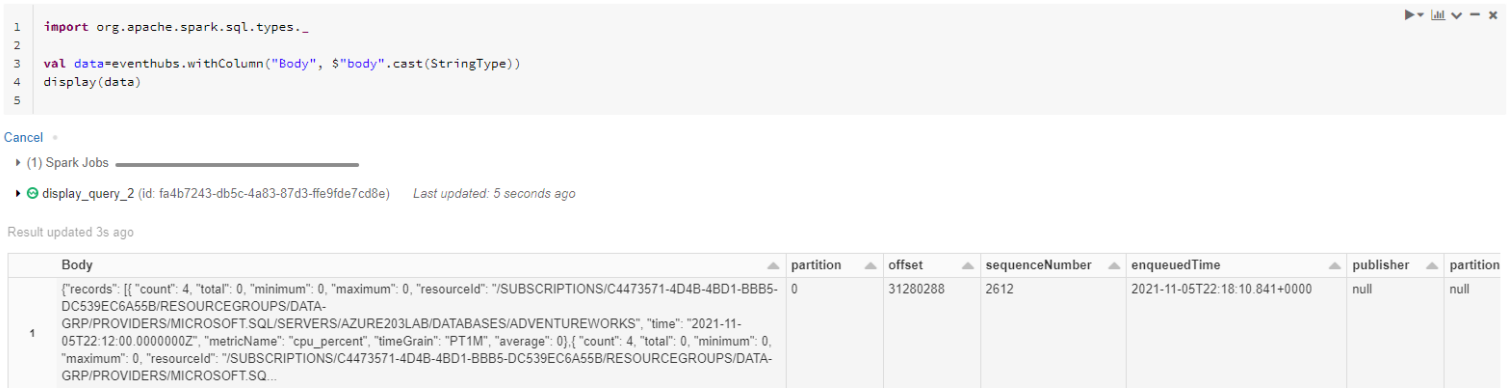
```
import org.apache.spark.eventhubs._
val connectionString =
"Endpoint=sb://appnamespace45667.servicebus.windows.net/;SharedAccessKeyName=Listen;SharedAccessKey=iKRqLLEp0vJnRDd4YKR2DtydWyaXM0SdelYZXKyRd7Y=;EntityPath=dbhub"
val eventHubsConf = EventHubsConf(connectionString)

val eventhubs = spark.readStream
  .format("eventhubs")
  .options(eventHubsConf.toMap)
  .load()
```

- For the connection string we go to the dbhub event hub and go to shared access policies. Add a new policy with manage permission. Copy the connection string.
- Upon displaying the eventhubs dataframe, it would be empty at first but it would start to populate little by little as the events start coming in but the
- Cancel display query when finished working.
- Now we get the string contents of the data doesn't make sense

```
import org.apache.spark.sql.types._
val data=eventhubs.withColumn("Body", $"body".cast(StringType))
display(data)
```

- We see the contents of the stream in the body part of the dataframe



- Next objective is to display the contents of the stream properly. The complete code starting of the connection string is as follows-

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

val connectionString = ""
val eventHubsConf = EventHubsConf(connectionString).setStartingPosition(EventPosition.fromStartOfStream)
```

- Next we use the function get_json_object which extracts the json object from a json string. We are doing this to get the body part of the message which are first being converted to a string

```
val eventhubs = spark.readStream
  .format("eventhubs")
  .options(eventHubsConf.toMap)
  .load()
  .select(get_json_object(($"body").cast("string"), "$.records").alias("records"))
```

- Next we are setting a max number of records that we want to access which is 30 in our case and then storing that into jsonElements variable. Next, we explode the jsonElements variable to get the individual elements that we have and store into a dataframe.

```
val maxMetrics = 30
val jsonElements = (0 until maxMetrics).map(i => get_json_object($"records", s"$$[$i]"))
val newDF = eventhubs
```

```
  .withColumn("records", explode(array(jsonElements: _*))) // Here _* is a special expression in spark to get each element of the array
  .where(!isnull($"records"))
```

- Next we create a schema for our dataframe, again create a new dataframe with it using the from_json function which parses a column containing a json string

```
val dataSchema = new StructType()
        .add("count", IntegerType)
        .add("total", IntegerType)
        .add("minimum", IntegerType)
        .add("maximum", IntegerType)
        .add("resourceId", StringType)
        .add("time", StringType)
        .add("metricName", StringType)
        .add("timeGrain", StringType)
        .add("average", IntegerType)

val df=newDF.withColumn("records",from_json(col("records"),dataSchema))

// Next we need to ensure there are multiple columns for each property of the JSON object
val finalDF=df.select(col("records.*"))
display(finalDF)
```

- Run the cell

We get the stream in our dataframe in a clean manner –

	count	total	minimum	maximum	resourceId	time	metricName
1	4	0	0	0	/SUBSCRIPTIONS/C4473571-4D4B-4BD1-BBB5-DC539EC6A55B/RESOURCEGROUPS/DATA-GRP/PROVIDERS/MICROSOFT.SQL/SERVERS/AZURE203LAB/DATABASES/ADVENTUREWORKS	2021-10-30T00:12:00.0000000Z	cpu_percent
2	4	0	0	0	/SUBSCRIPTIONS/C4473571-4D4B-4BD1-BBB5-DC539EC6A55B/RESOURCEGROUPS/DATA-GRP/PROVIDERS/MICROSOFT.SQL/SERVERS/AZURE203LAB/DATABASES/ADVENTUREWORKS	2021-10-30T00:13:00.0000000Z	cpu_percent
3	4	0	0	0	/SUBSCRIPTIONS/C4473571-4D4B-4BD1-BBB5-DC539EC6A55B/RESOURCEGROUPS/DATA-GRP/PROVIDERS/MICROSOFT.SQL/SERVERS/AZURE203LAB/DATABASES/ADVENTUREWORKS	2021-10-30T00:14:00.0000000Z	cpu_percent

Getting data from your data lake gen2

We would need to make use of Azure Key Vault. We will store the access key as a secret in the key vault service and then create a databricks scoped secret to access the key value.

- Go to all resources and search Key vault and create
- Select your subscription, resource group, unique key vault name, days to retain the deleted vaults as 7 and create.
- Once it is created, go to it and select secrets from the left and click Generate / import
- Give it a name and the key obtained from going to access keys of our data lake would be the value here and create

Next we need to create a databricks scoped secret

- Go to your databricks workspace, copy the url of the workspace from the homepage and paste it in a new tab with /#secrets/createScope added to the URL – so mine looks like - https://adb-461202112312313.1.azuredatabricks.net/#secrets/createScope
- Enter the scope key, copy the vault url form the vault home page and paste into DNS name field, for resource id go to the properties for your vault and copy it from there and create.

Note – I faced a verification error was the message while creating the scope key said the it was not able to locate the DNS. The error was fixed after about 10 minutes of creating the vault had passed. So probably if you get the same error and everything looks right then probably wait.

- Create a new notebook with scala and paste the following code. The scope name is the scope name that you just created, the key would be the name of the secret that you created

```
spark.conf.set(
    "fs.azure.account.key.DATA LAKE NAME.dfs.core.windows.net",
```

```
    dbutils.secrets.get(scope="data-lake-key",key="datalakekey"))

val df = spark.read.format("csv").option("header","true").load("abfss://data@DATA LAKE
NAME.dfs.core.windows.net/raw/Log.csv")

display(df)
```

- After running the cell, you can see the dataframe has the contents of the log.csv file.

Writing Data into the dedicated SQL pool

We will write the above data that we read into the dedicated sql pool. First we delete the contents of your logdata table from the dedicated sql pool.

- Building on the code from the above notebook, we select all the columns into a new dataframe. We are converting the Time column to timestamp before sending to Syanpse

```
var nd = df.withColumn("Time3" , to_timestamp($"Time"))
var nd1 = nd.drop("Time")
var nd2 = nd1.withColumnRenamed("Time3", "Time")
val dfcorrect=nd2.select(col("Id"), col("Correlationid"), col("Operationname"), col("Status"),
col("Eventcategory"), col("Level"), col("Time"), col("Subscription"), col("Eventinitiatedby"),
col("Resourcetype"), col("Resourcegroup"))
```

- We also need to have a temporary staging area as well. So here I'm saying it can use the synapse container as the temporary staging area

```
val tablename="logdata"
val tmpdir="abfss://synapse@datalake2000.dfs.core.windows.net/ "
```

- Next we are creating a connection to our synapse

```
val connection =
"jdbc:sqlserver://azuresynapsedp203.sql.azuresynapse.net:1433;database=dp203;user=sqladminuser;password=PASSWORD;e
ncrypt=true;trustServerCertificate=false;"
```

- Using the write function we are writing the data to an external data store. Append would add new columns to already existing data.

```
dfcorrect.write
  .mode("append") // Here we are saying to append to the table
  .format("com.databricks.spark.sqldw")
  .option("url", connection)
  .option("tempDir", tmpdir) // For transfering to Azure Synapse, we need temporary storage for the staging data
  .option("forwardSparkAzureStorageCredentials", "true")
  .option("dbTable", tablename)
  .save()
```

Data now shows up in the logdata table.

Now that we are able to move data that is stored on the data lake onto the dedicated sql pool, we can now stream the data to the dedicated sql pool.

Streaming and saving / writing data in the dedicated sql pool

For this part we are using the dblog table to save the data that is being streamed. Delete it and recreate it –

```
CREATE TABLE [dbo].[dblog]
(
  [count] [bigint], [total] [bigint], [minimum] [bigint], [maximum] [bigint],
  [resourceId] [varchar](1000), [time] datetime, [metricName] [varchar](500), [timeGrain] [varchar](100),
  [average] [bigint]
)
```

```
WITH(  DISTRIBUTION = ROUND_ROBIN,HEAP  )
```

Now back to the original notebook where we had written code to get streaming data, we make some changes.

- We add the configuration so that spark can authorize itself

```
spark.conf.set(
    "fs.azure.account.key.DATA LAKE NAME.dfs.core.windows.net",
    dbutils.secrets.get(scope="data-lake-key",key="datalakekey"))
```

- The schema for the table is change for integer types to long types

```
val dataSchema = new StructType()
        .add("count", LongType)
        .add("total", LongType)
        .add("minimum", LongType)
        .add("maximum", LongType)
        .add("resourceId", StringType)
        .add("time", DataTypes.DateType)
        .add("metricName", StringType)
        .add("timeGrain", StringType)
        .add("average", LongType)
```

- After the finalDF is being created, we reuse code from the previous part of writing to Synapse. We give the tablename dblog, we give the staging area in our data lake, the connection string

```
val tablename="logdata"
val tmpdir="abfss://synapse@datalake2000.dfs.core.windows.net/ "
val connection =
"jdbc:sqlserver://azuresynapsedp203.sql.azuresynapse.net:1433;database=dp203;user=sqladminuser;password=PASSWORD;e
ncrypt=true;trustServerCertificate=false;"
```

- Lastly, we use the writestream function to write the stream on to our dblog table

```
finalDF.writeStream// Here we need to change the function as writeStream
  .format("com.databricks.spark.sqldw")
  .option("url", connection)
  .option("tempDir", tmpdir) // staging area
  .option("forwardSparkAzureStorageCredentials", "true")
  .option("dbTable", tablename)
  .option("checkpointLocation","/tmp_location") // We need to mention a checkpoint location
  /*
  The checkpoint helps to resume a query from where it left off, if the query fails for any reason
  in the middle of processing data.
  Each query should have a different checkpoint location
  */
  .start()
```

- We see the data has started to show up in our Synapse tables

If this is too much to put together, this is what the final file looks like.

Azure Active Directory Credential Passthrough

Here the user working on the notebook would dbe able to access data in the azure data lake without having access to the access keys and probably using the vault and scoped credentials to access the data lake storage.

To test this feature out we will create a new storage account.

- Select the regular options for subscription, resource group, unique name, LRS. On Networking enable hierarchical namespace.

- Next, we would need to upload a file and give the required permissions to it. We create a data container and upload the log.csv file in it.
- Now we need to give permissions to the admin account. Even though we are the admin account, we would need to give the necessary permissions.
  - Now we go to the access control for the new data lake, click add and add a role assignment.
  - Select the reader role and go next
  - From select members, select the admin user
  - Review and assign
  - We have to **add another role**
  - Select storage blob data reader
  - Select the admin
  - Review and assign
- Now go to the Azure Storage Explorer and log in. right click on the container of your new data lake and click Manage Access Control lists.
  - Search for your user. I used just used my name to search. Click Add.
  - On the returning screen select Access and Read check boxes and hit okay
  - It would say successfullt saved permission for “data/”
  - Right click on the container again and select propagate access control list and hit okay
- Next we need to create a new cluster. Terminate your existing cluster and create a new single node cluster.
  - In the advanced options, select the credential passthrough for user-level data access and select the user
  - Create the cluster
- Create a new notebook with the cluster
- Enter the following piece of code. Replace databricks with your container name and newdatalake1000 with your data lake name –

```
val df =
spark.read.format("csv").option("header","true").load("abfss://databricks@newdatalake1000.dfs.core.windows.net/Log
.csv")
display(df)
```

- You should see the data show up

With this we have not used any access keys at all. No keys used from the data lake, no keys created on the cluster. We are being authorized based on the credentials of the User defined in the Active Directory. So if the same user is running the notebook that user would be able to access to the data lake.