Exploring Apache Spark and Spark SQL in Microsoft Azure Databricks

Introduction

This class introduces students to Apache Spark on Azure with Databricks. It helps student to understand the value proposition of Apache Spark over other Big Data technologies like Hadoop. They should understand the similarities between Hadoop & Spark, their differences and respective nuances. They should be able to decide when to use what and why for a given business use case in a typical enterprise environment.

Azure specific highlights of Apache Spark

Source: https://docs.microsoft.com/en-us/azure/azure-databricks/what-is-azure-databricks

#1 Ease creation

You can create a new Spark cluster in minutes without the complexity usually associated with infrastructure

#2 Ease of use

Spark cluster in Databricks give you access to a notebook interface. You can use these notebooks for interactive data processing and visualization.

#3 REST APIs

Spark clusters in Databricks allows you to connect to a RESP API and work with the data you produced during your analysis. On top of that, Databricks provides an interface allowing you to schedule and monitor your Spark jobs.

#4 Support for Azure Data Lake Storage

Spark clusters in Databricks can use Azure Data Lake Storage as both the primary storage or additional storage.

#5 Integration with Azure services

Spark cluster in Databricks comes with a connector to Azure Event Hubs. You can build streaming applications using the Event Hubs, in addition to Apache Kafka, which is already available as part of Spark.

#6 Support for ML Server

Databricks ML Flow platform speeds up ML development and deployment. Also, the notebooks support using Python which is the de facto language for ML life cycle.

#7 Integration with Azure DevOps

In a real world utilization of Databricks and Spark, CI/CD processes become important. As part of Azure, Databricks works nicely with AzureDevops and allow you to manage code versioning as well as Continuous delivery.

#8 Scalability

Databricks allows you to seamlessly replace smaller cluster by bigger cluster and re-attach your notebooks to the new cluster in minutes. Price will increase with cluster usage and size.

Main highlights of Spark SQL

Source: http://spark.apache.org/sql/

#1 Integrated - Seamlessly mix SQL queries with Spark programs. Spark SQL lets users query structured data inside Spark programs, using either SQL or a familiar DataFrame API. Usable in Java, Scala, Python and R.

#2 Uniform Data Access - Connect to any data source the same way. DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. Users can even join data across these sources.

#3 Hive Compatibility - Run unmodified Hive queries on existing data.Spark SQL reuses the Hive frontend and metastore, giving users full compatibility with existing Hive data, queries, and UDFs.

#4 Standard Connectivity - Connect through JDBC or ODBC.A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.

Takeaways

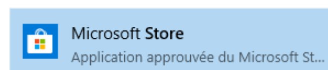
- Provision an Databricks Spark Cluster.
- Access data from Azure storage container and create Dataframe.
- Understand joins, functions and user defined functions.
- Connect your Databricks Spark Cluster with Power BI Visualization.

Prerequisites

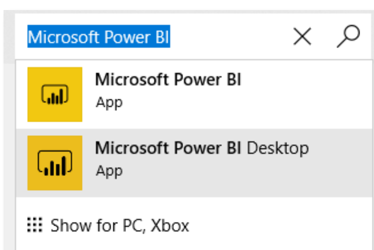
a) An Azure subscription. See here.

a) Microsoft Power BI Desktop See here

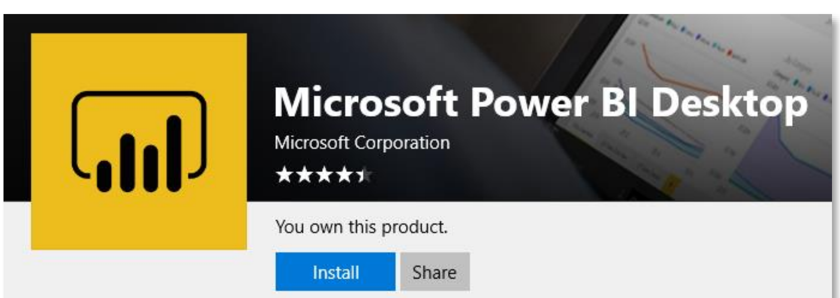
1) Launch the Microsoft Store (from windows 10)



2) In the Search bar, type Microsoft Power BI Desktop and select Microsoft Power Bi Desktop.



3) Click on **Install**



Section 1 - Prepare Cluster and dataset

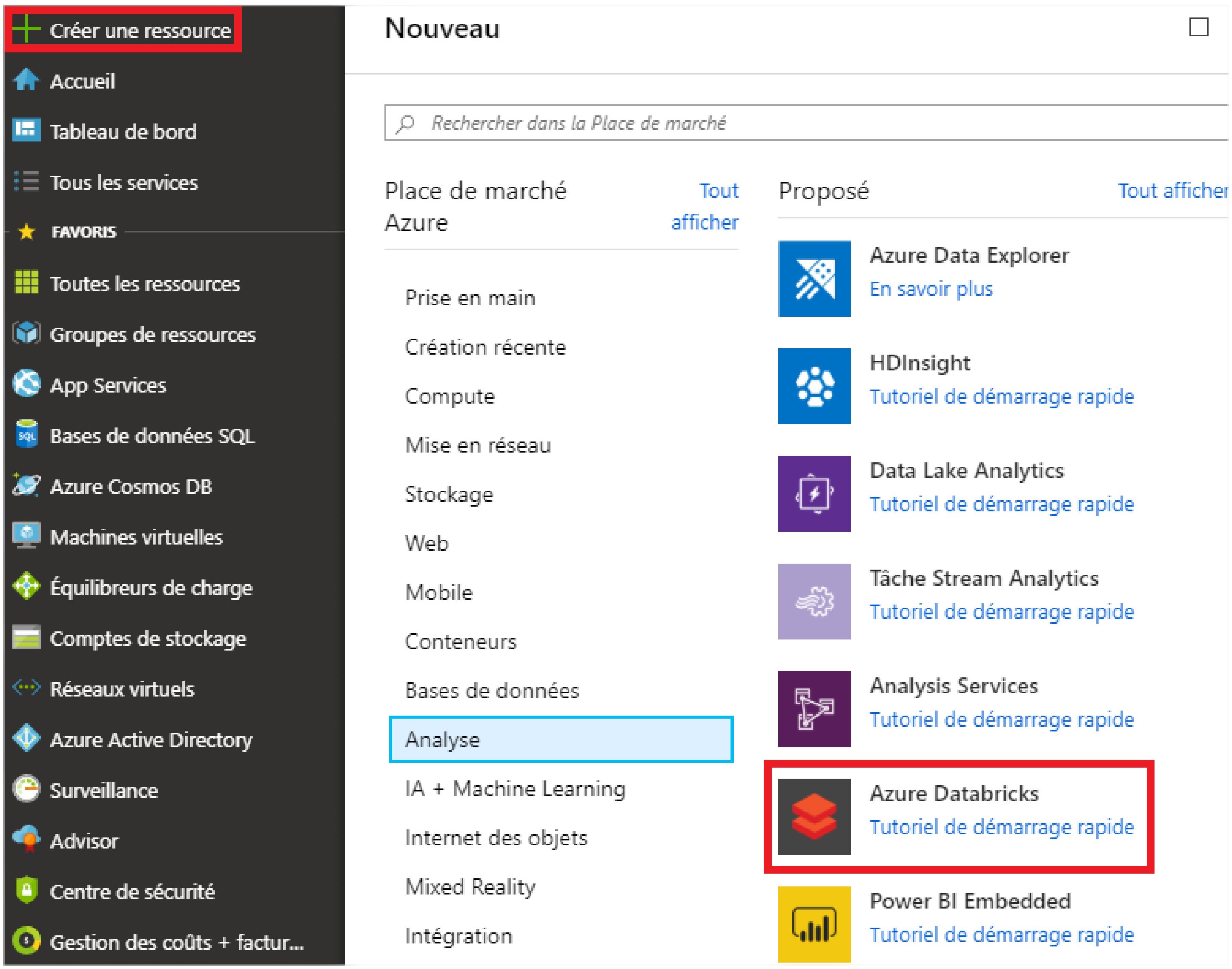
Provision an Azure Databricks cluster

Access Azure Portal

1. Sign in to the Azure portal.

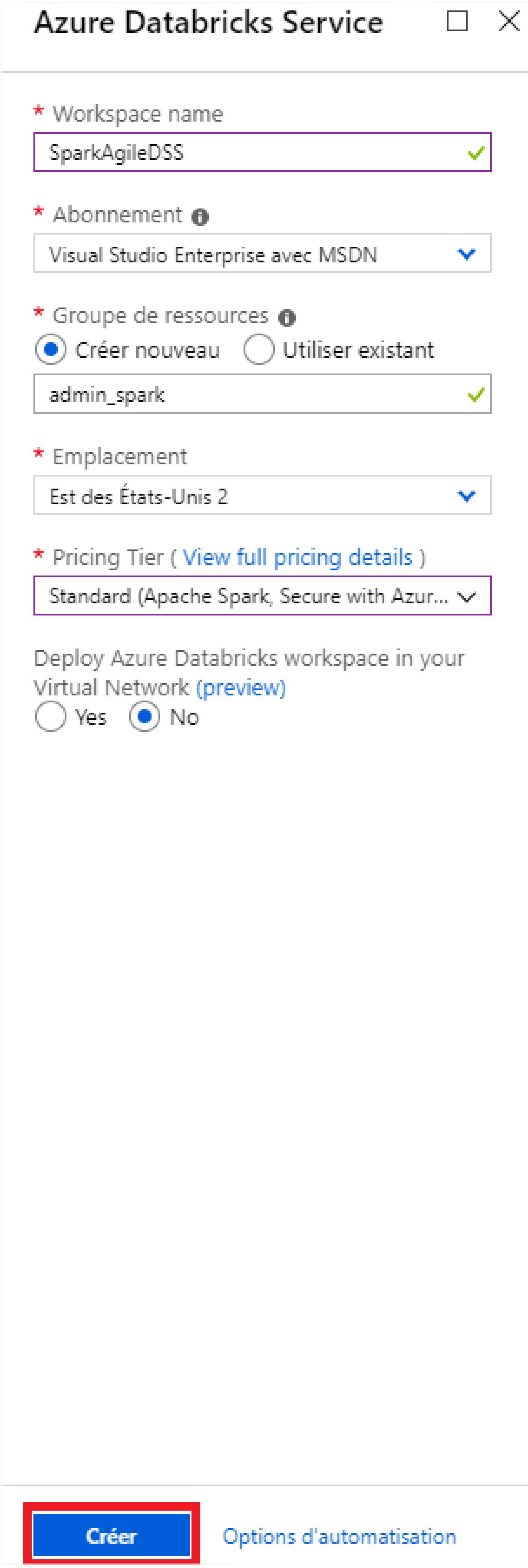
Create Azure Databricks cluster

1. Click Create new resource, click Analysis, and then click Azure Databricks.



Provide Cluster Details

1. In the Azure Databricks Service blade, enter an available **Workspace Name.** Note that it cannot include “Microsoft” or “MS”.



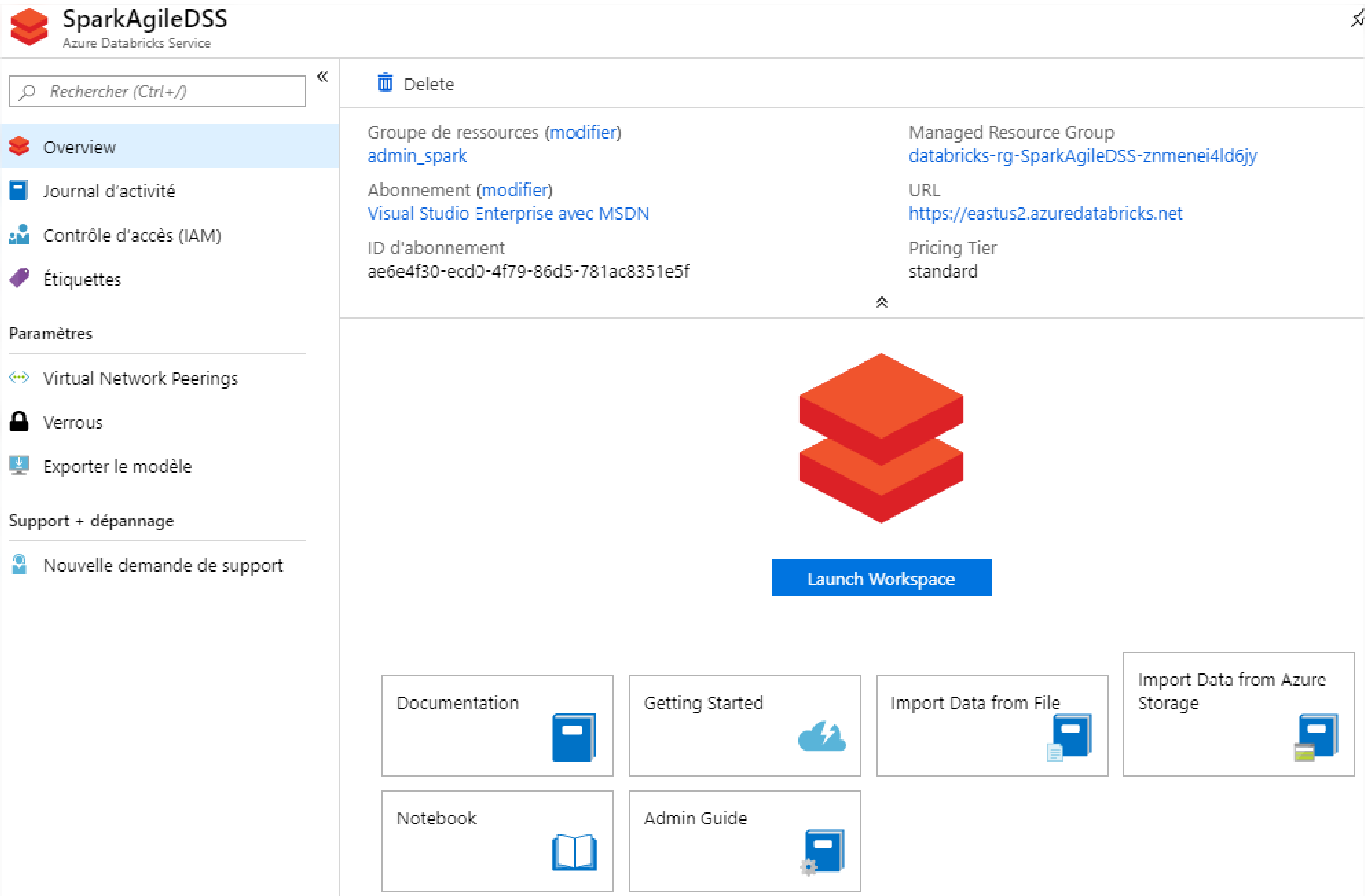
A green check mark appears beside the cluster name if it is available.

2. For **Subscription**, if you have more than one subscription, click the Subscription entry to select the Azure subscription to use for the cluster.

Provision cluster

1. Click **Create** button to finalize cluster creation. This may take 5 minutes.

This creates the cluster and adds a tile for it to the **Startboard** of your Azure portal.

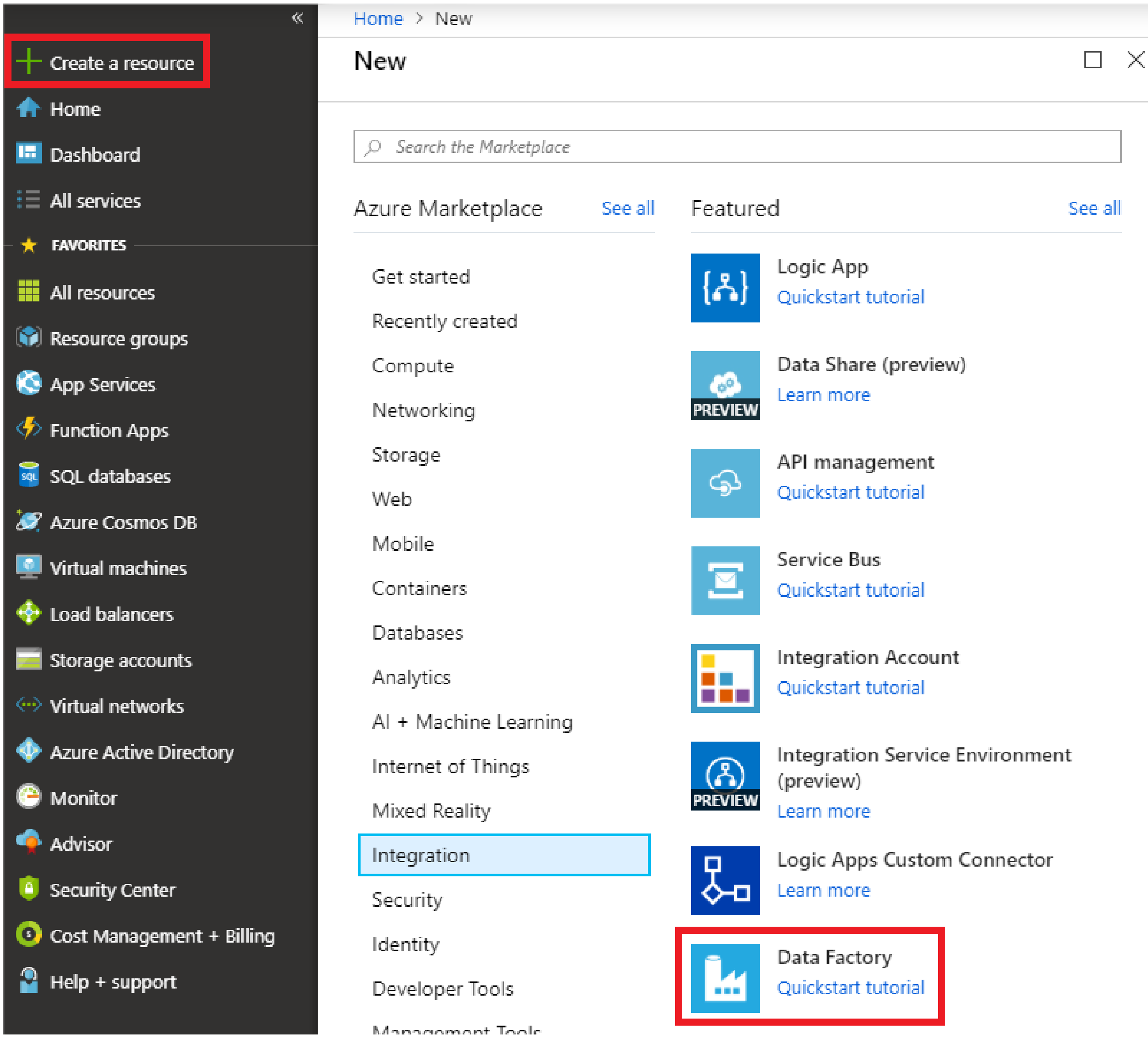


Load datasets files to storage account.

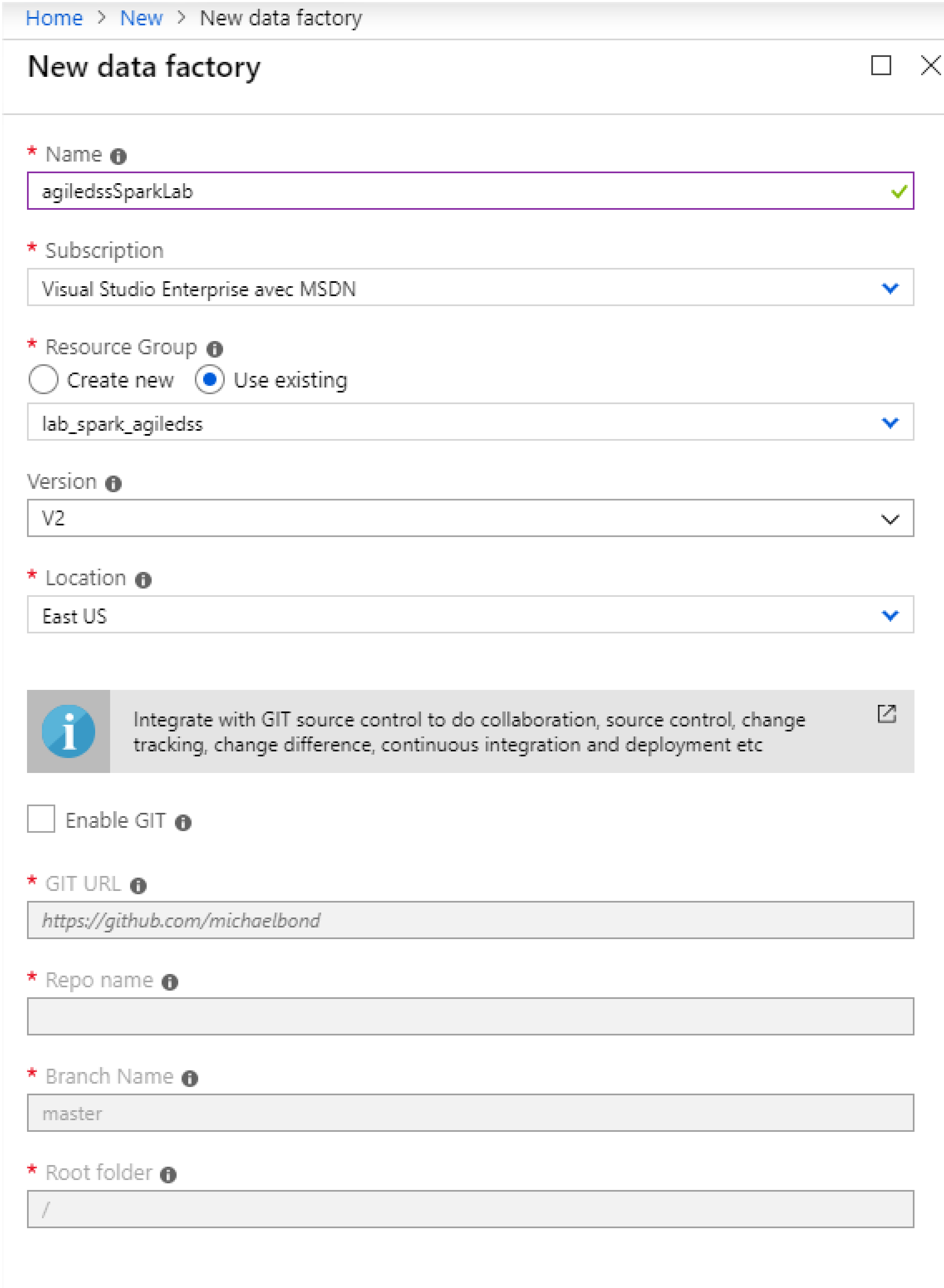
In this section, you'll copy the files required for the lab to the storage account previously created. You'll copy the files between two storage accounts with the help of Data Factory.

To copy the files, follow the below steps.

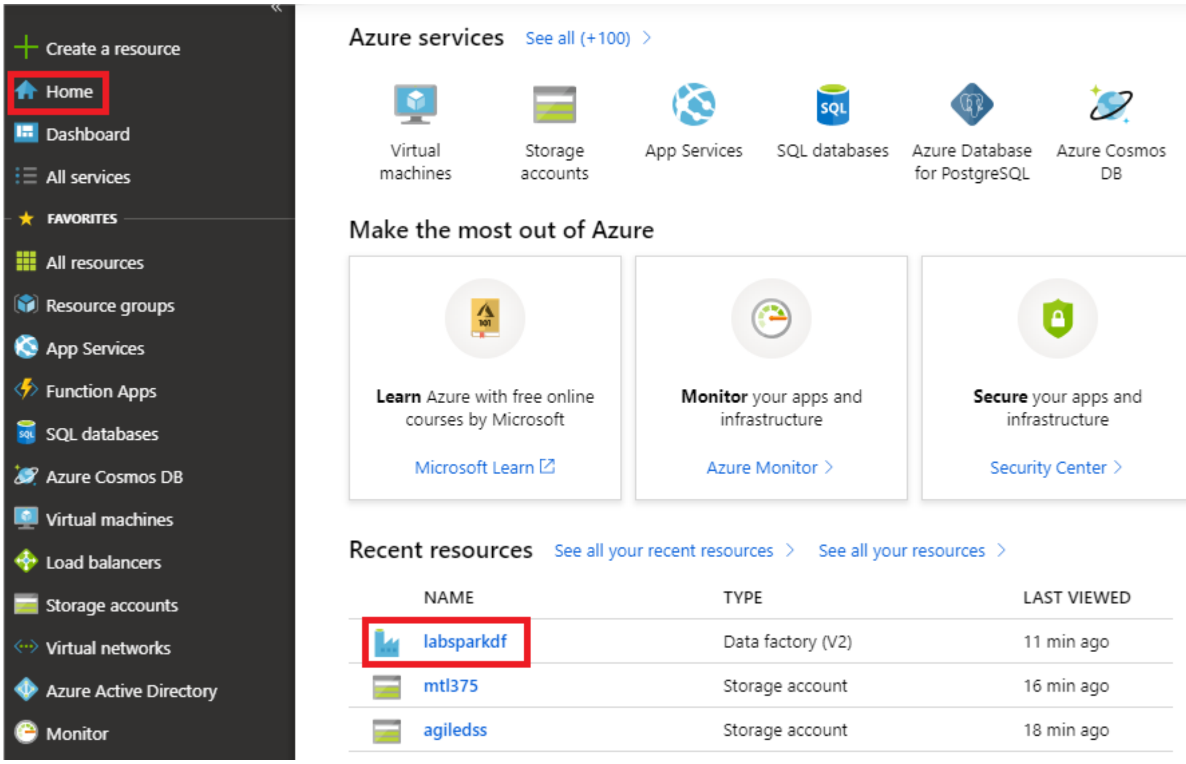
1. Create a Data Factory instance from Azure Portal.



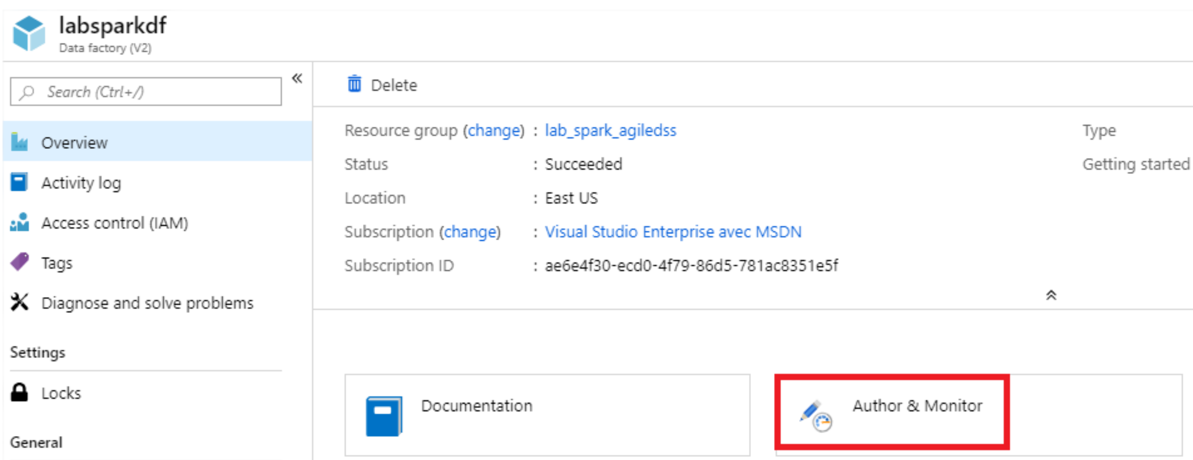
2. Fill the required information to create the new **Data Factory**. Enter a name you will easily recognize. Choose your subscription, created with your account and include the **Data Factory** in the resource group we created with the Databricks cluster. We will use the V2. Make sure you choose the same geographic area so you won't be charge for getting data out of the data center.



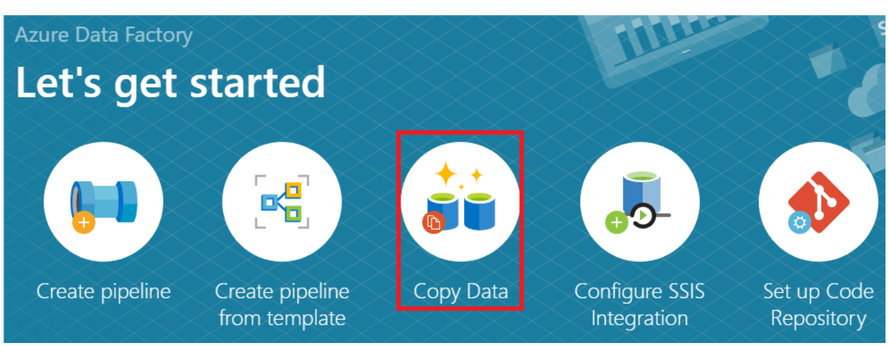
3. In your Portal Home, select your **Data Factory**.



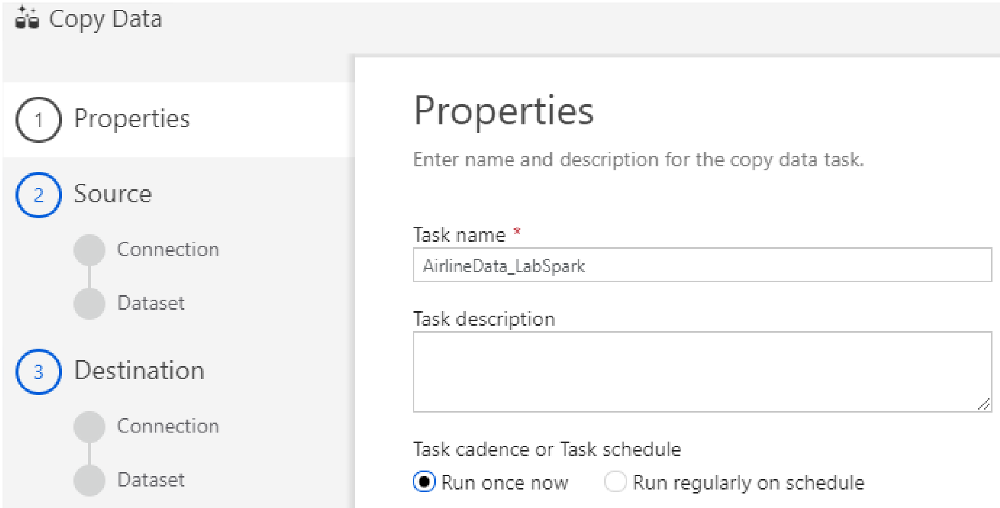
4. Once inside the **Data Factory** interface, click on **Author and Monitor**.



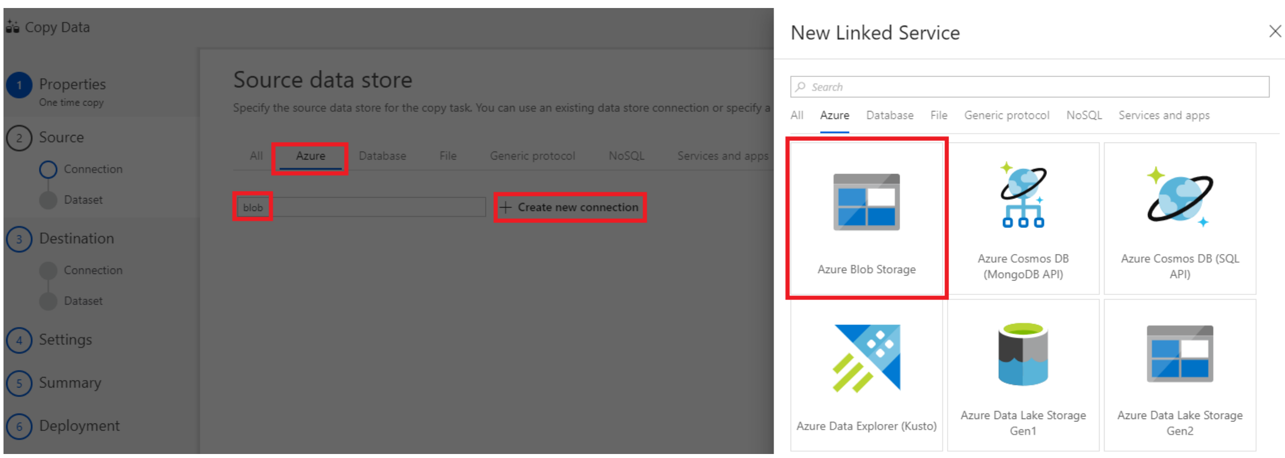
5. Click on Copy Data.



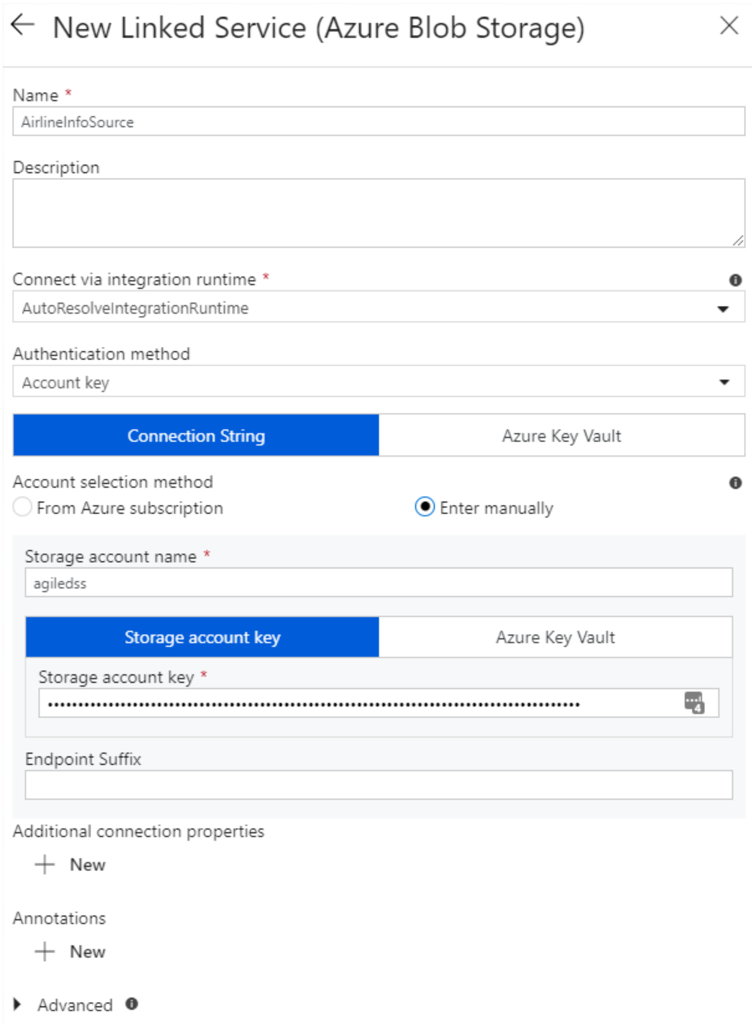
6. Give a name to the pipeline. We will only run it once. Then click **Next**.



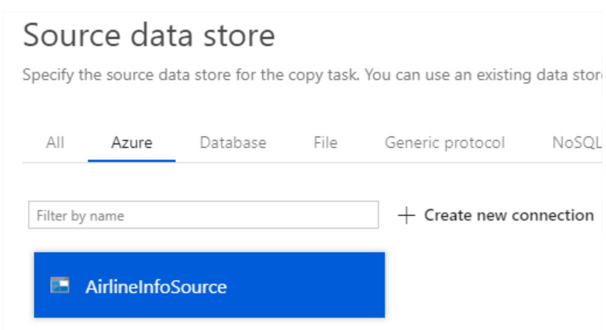
7. Select the **Azure** tab, then search for **blob**, then click on **Create new connection**. A new window will appear on the top of where you can select **Azure Blob Storage**. Once this is done, click on **Continue** at the bottom right of the screen.



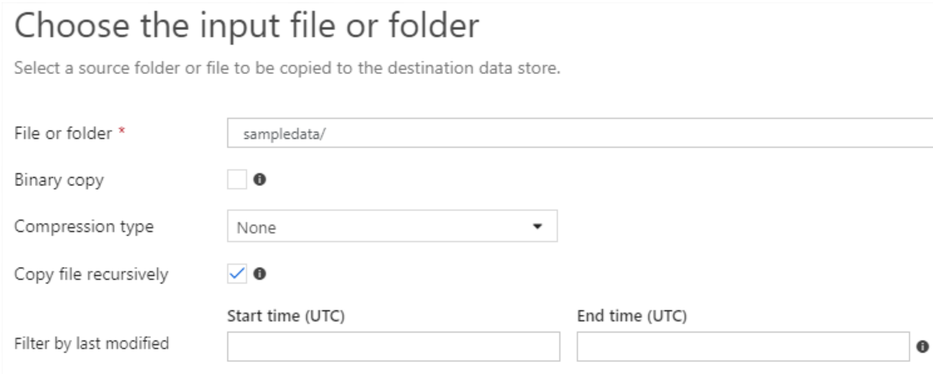
8. Give a name to your source, then select the default Runtime. In the **Authentication method** box, select **Account key**, then **Connection String**, then **Enter manually**. In the fileds below, the **Storage account name** is agiledss and the key is : 6YJEwcCQZarYJAYwcWj5l/kGs/A0evANjeqE7UE/Kfb0ig3c603z4AF9PfdVsWAWoSg8Pcj23T6GwkhoOi+bLw==. Test the connection and click **Finish.**



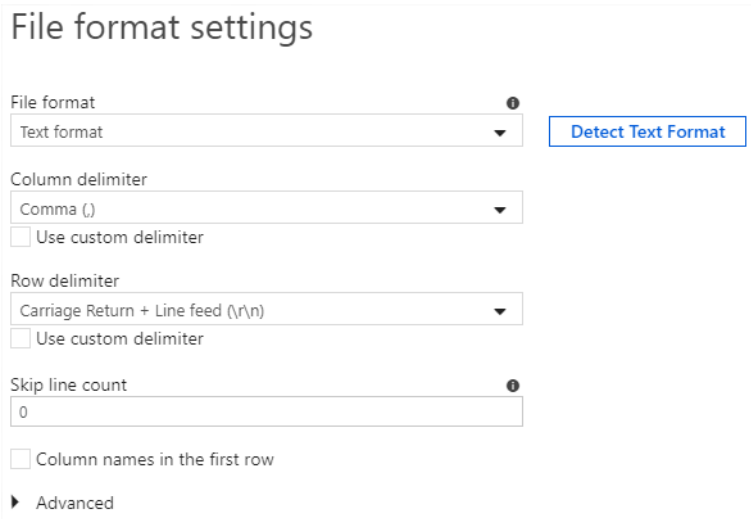
9. Clear the filter and select the newly created Data Source, then click Next.



10. Select the browser **sampledata**, then click **Next**.

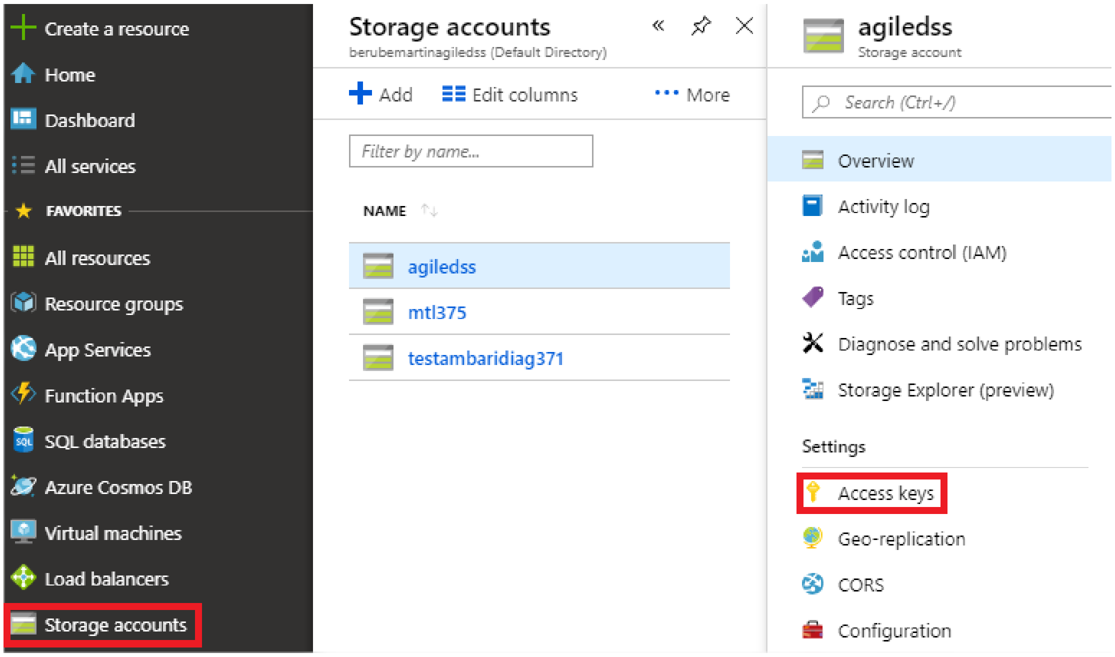


11. On the next screen, click **Detect Text Format**, then **Next**.

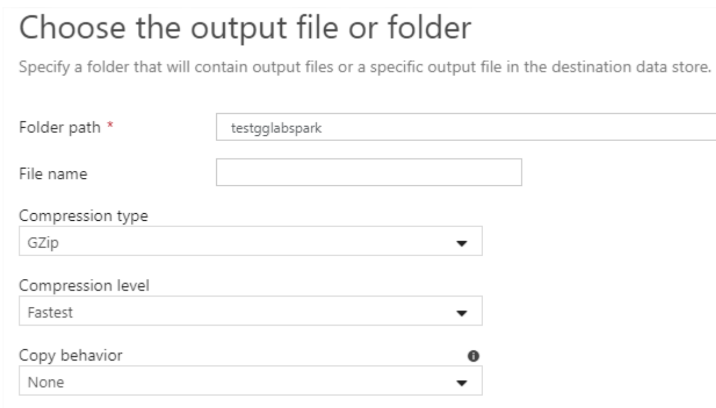


12. We will now create a new connection to deposit the data in the Container you create. Click on **Create new connection**, then select **Azure Blob Storage** just like we did at Step 7.

13. Enter a name for your destination and select the same elements as Step 8 until **Enter manually**. To the remaining fields you will need to go to you Azure Portal window, click on **Storage Accounts**, then select the one you created. Finally, under **Access keys**, copy one of the key strings (any of the 2 ending with "==" should do). Paste it in the **Storage account key** and fill your **Storage account name** with the from you container account. Test connection, then **Finish**.

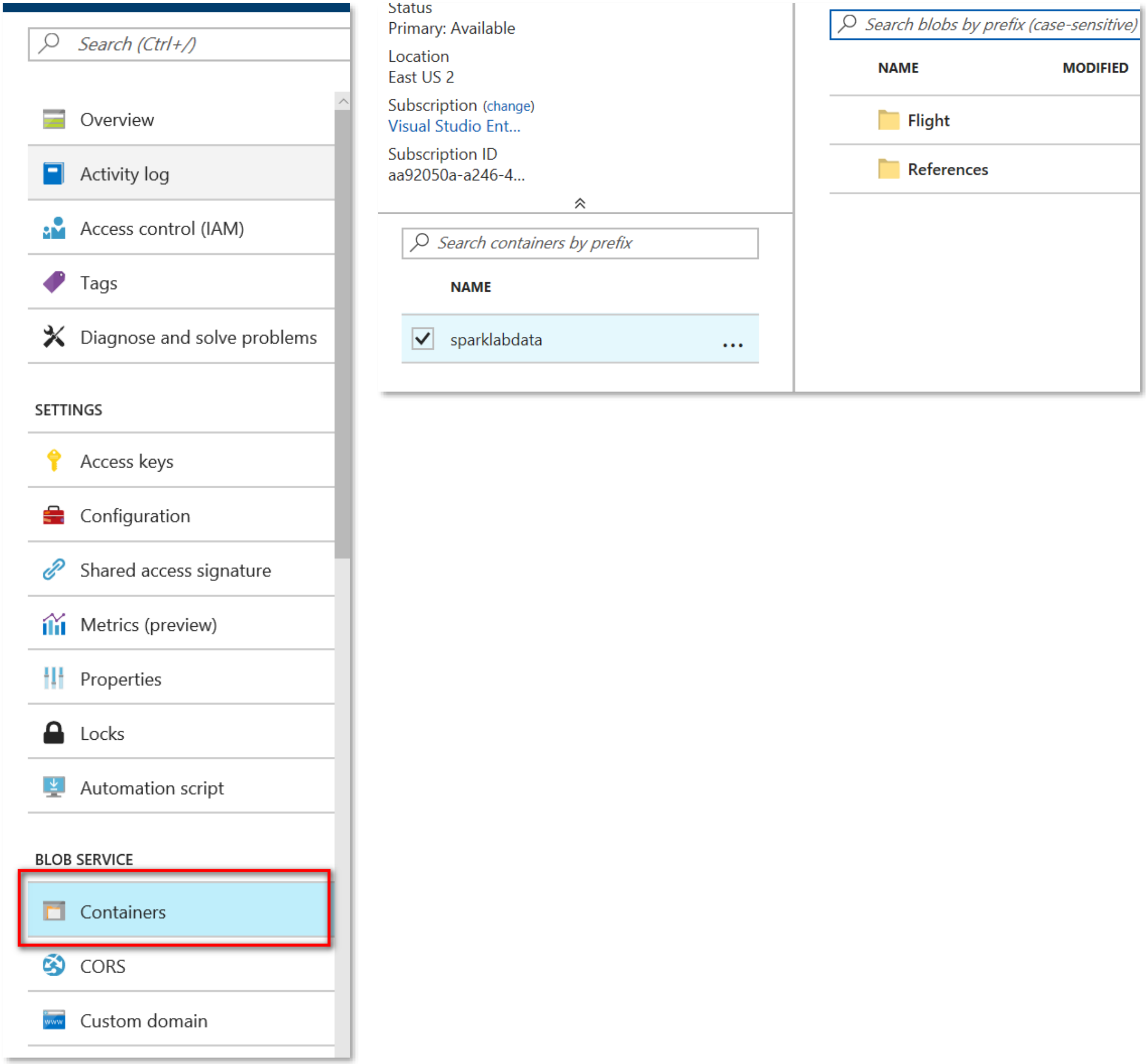


14. Choose a Folder where you want to save the data or create one. Compress it to Gzip and Fastest. Leave the format to default **Text format**, it's the csv type we want.



15. Click **Next** to Settings and Summary. In the Deployment screen, click on **Monitor**. This should be quick and once it's completed, go check in your **Container** to make sure the files have been copied.

16. In Azure portal, navigate to your storage account, then Containers below 'BLOB SERVICE' (see following screenshot), and verify that a new container 'sparklabdata' has been created, containing all the resources:

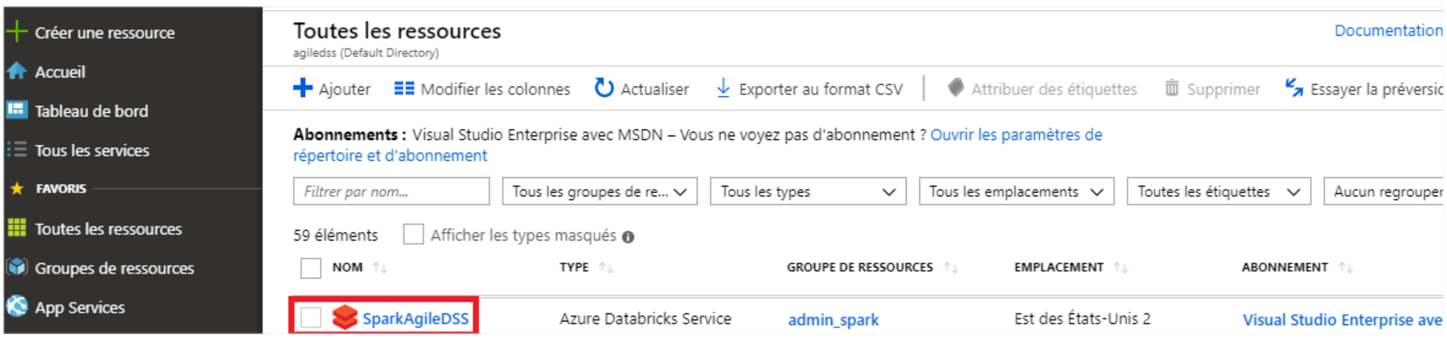

Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
SETTINGS
Access keys
Configuration
Shared access signature
Metrics (preview)
Properties
Locks
Automation script
BLOB SERVICE
Containers
CORS
Custom domain
Status
Primary: Available
Location
East US 2
Subscription (change)
Visual Studio Ent...
Subscription ID
aa92050a-a246-4...
Search containers by prefix
NAME
sparklabdata
Search blobs by prefix (case-sensitive)
NAME
MODIFIED
Flight
References

Section 2 - Spark SQL and Dataframe

Access data from Azure storage container and Create Data frame.

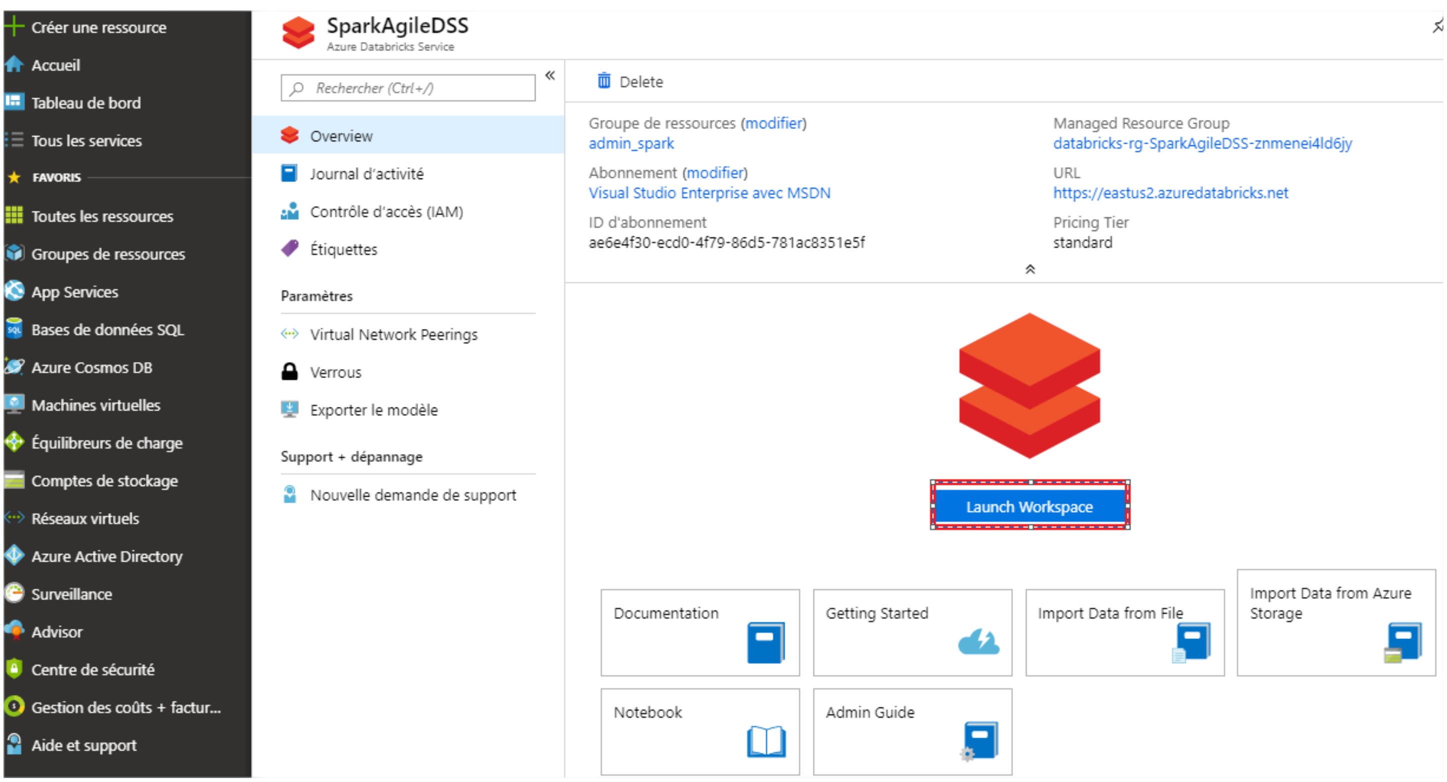
Access Azure

1. Sign in to the **Azure portal**.
2. Click tile for your Spark Service.



Launch Notebook

1. Click on **Launch Workspace** tile present on the Cluster Blade.

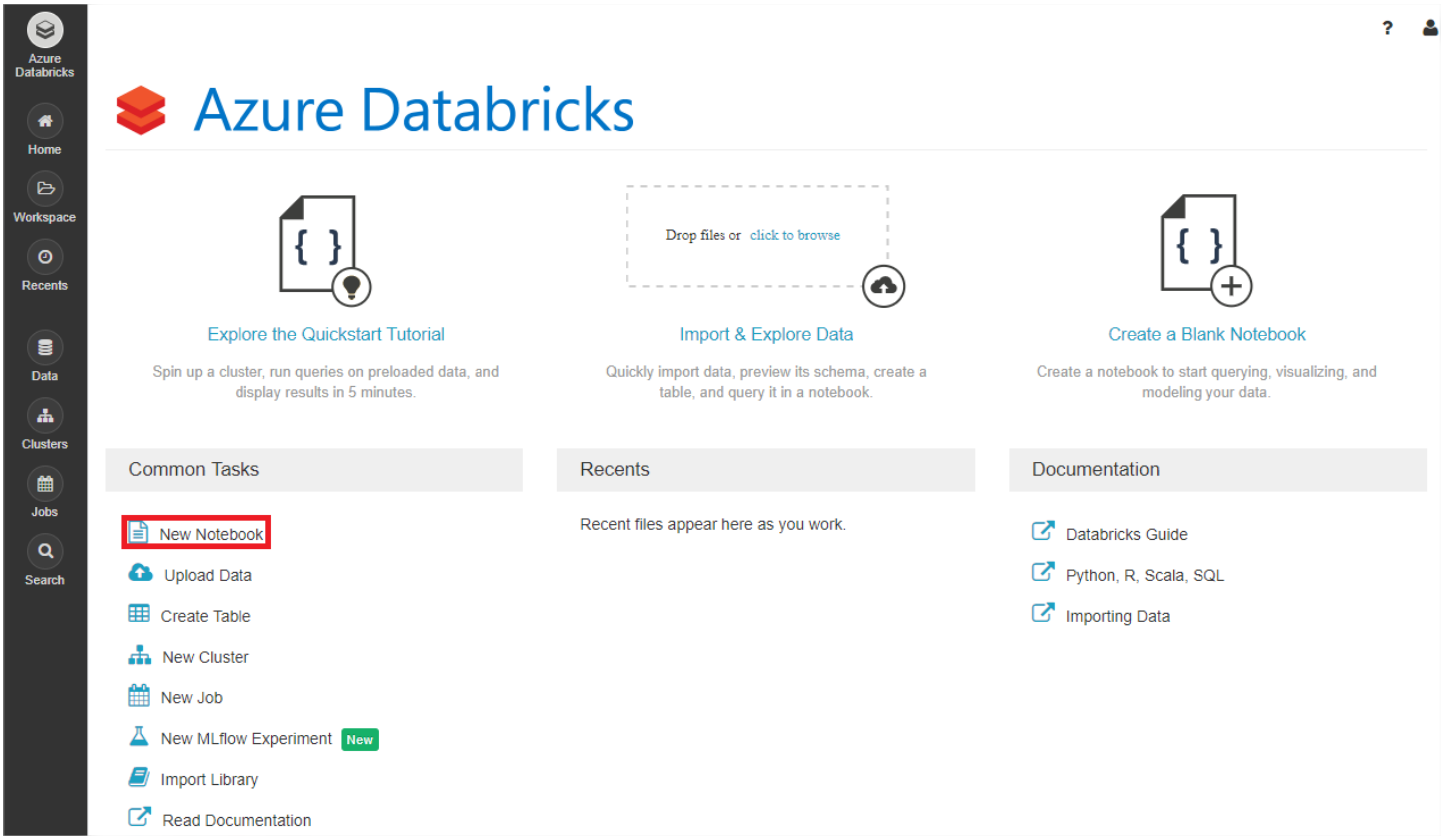


When prompted, use the admin credential of your Spark Cluster.

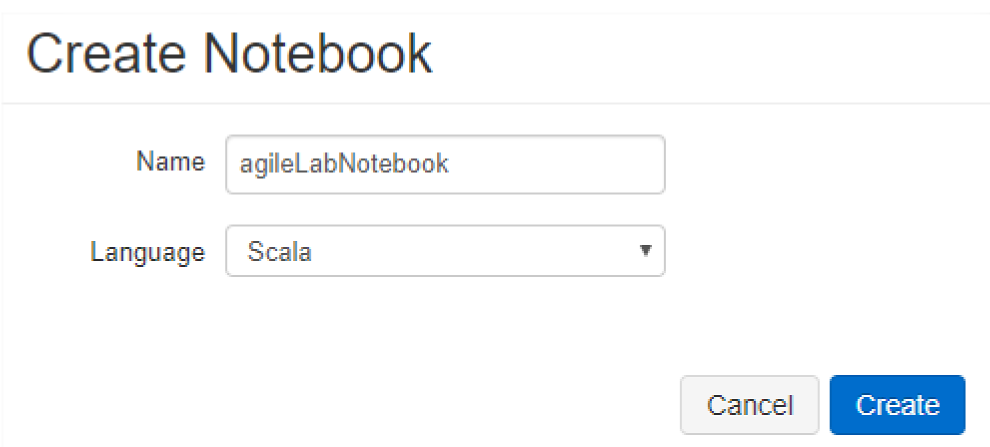
Create a new Notebook

If prompted, enter the admin credentials for the Spark cluster.

Azure Databricks Notebook will open.



1. Click **New Notebook** button in the middle left or upper right of the screen.
2. Select **Scala** as the language, from the dropdown.
3. Give a name to the note



Create Spark and SQL context

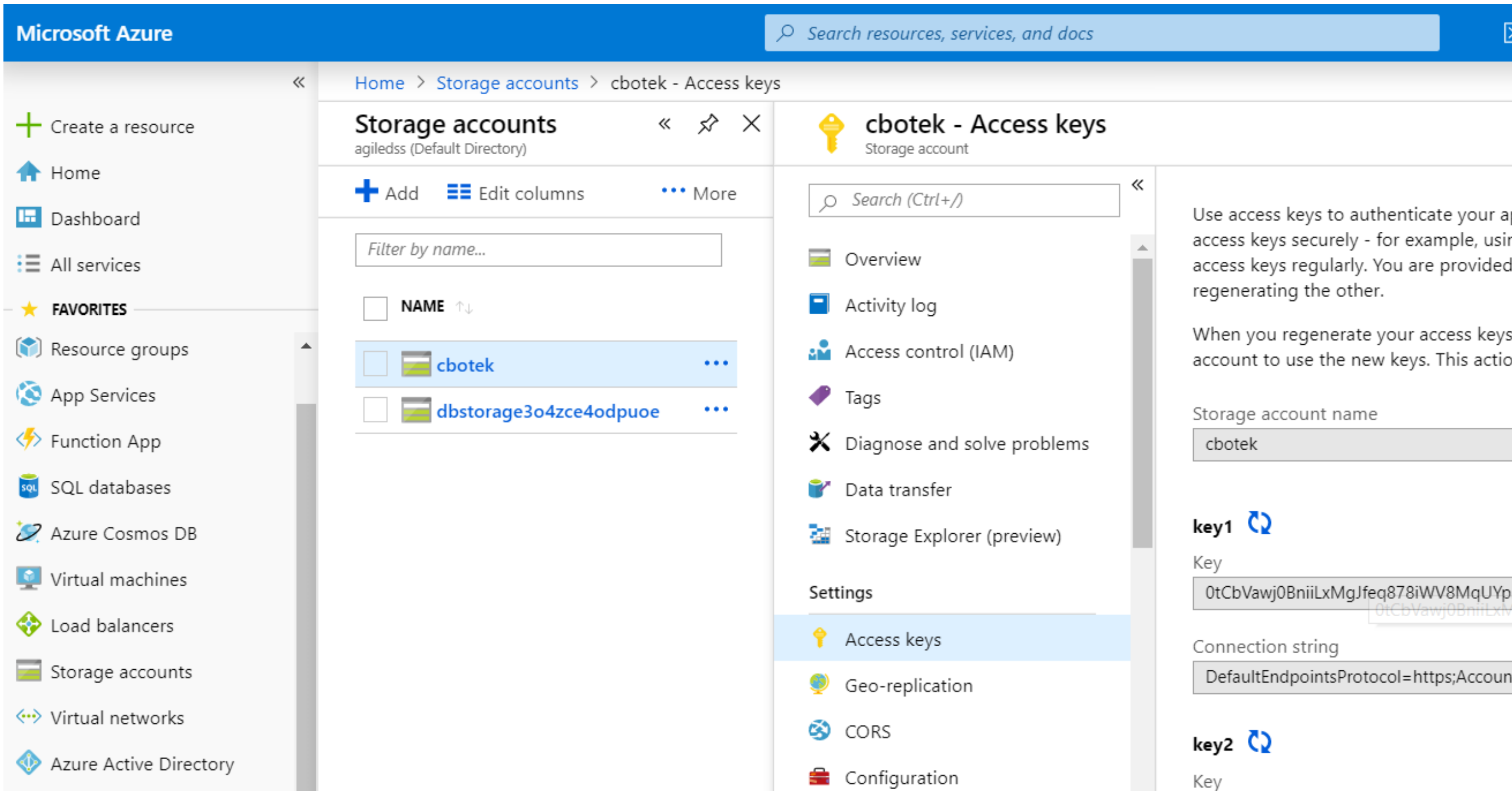
Starting from Spark2.0 there is no need to import and start SparkContext and SQLContext!

Create data frame from data stored in azure blob storage

A first block of code is already created for you at the top of the screen:

Every time you run a job in Databricks, your web browser window title will show a (Busy) status alongside the notebook title.

1. Spark needs access to our Blob Storage in order to work with the data:



For this, we need to go back on the Azure portal:

- click on **Storage accounts** on the left hand side
- click on the storage account name we just put the data with **Artifactory**
- Then click on **Access Keys**
- Copy the key under **Key1**

Now back to Databricks, we paste the following command in the first cell of the notebook while replacing

```
spark.conf.set(
  "fs.azure.account.key.<storage_account_name>.blob.core.windows.net",
  "<key>")
```

Now Execute the cell with CTRL+SHIFT+ENTER or the play button on the right hand side of the cell

```
Cmd 1

spark.conf.set(
  "fs.azure.account.key.cbotek.blob.core.windows.net",
  "0tCbVawj0BniiLxMgJfeq878iWV8MqUYp3klz76+67wvtUOKDShSRS4MCclv/PYQQrZNNxcj+l7sk6BUBdkcYA==")
```

2. Paste the following snippet in below empty cell, do not forget to replace <container_name> (should be sparklabdata)
   and <storage_account_name>.

```
# Define dataset azure path
Airportspath
="wasb://<container_name>@<storage_account_name>.blob.core.windows.net/References/A
irports.csv"

# Obtain dataframe
val airports = spark.read.csv(Airportspath)

# show first 20 lines
airports.show()
```

3. Press SHIFT + ENTER. Or Press Play button from tool bar to execute the code inside cell.

```
Cmd 2

val airports = spark.read.csv("wasbs://datalake@cbotek.blob.core.windows.net/References/Airports.csv")
airports.show()

▸ (2) Spark Jobs
▸ airports: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 12 more fields]
+---+--------------------+-----------------+----------------+---+----+-------------------+--------------------+----+---+---
|_c0|                 _c1|              _c2|             _c3|_c4| _c5|                _c6|                 _c7| _c8|_c9|_c1
+---+--------------------+-----------------+----------------+---+----+-------------------+--------------------+----+---+---
|  1|      Goroka Airport|           Goroka|Papua New Guinea|GKA|AYGA|-6.081689834590001|       145.391998291|5282| 10|
|  2|      Madang Airport|           Madang|Papua New Guinea|MAG|AYMD|     -5.20707988739|       145.789001465|  20| 10|
```

4. Output of above code execution will be as shown below, meaning Spark application correctly started:

Spark Application Id: application_1554229124335_0004
Spark WebUI: http://hn1-sparka.shz1afxuo4we1mdd4ugdg021xg.cx.internal.cloudapp.net:8088/proxy/application_1554229124335_0004/
Took 52 sec. Last updated by anonymous at April 02 2019, 3:11:52 PM.

5. Verify "airports" data type, it should be "DataFrame". You can paste this code in an empty cell and run it:

```
type(airports)
```

```
<class 'pyspark.sql.dataframe.DataFrame'>
```

```
#try airports alone
airports
```

DataFrame operations, explore the data

Execute following operations on DataFrame created earlier and observe the output. Use empty cells in the notebook to execute these operations.

- Do the same thing for another dataset

```
Cmd 5
val root = "wasbs://datalake@cbotek.blob.core.windows.net"
val flightPerf = spark.read.option("header", "true").csv(s"${root}/Flight/*/*")

flightPerf.show()

Cancel  Running command...
(1) Spark Jobs
```

:

```
flightPerf.count()
```

Output (This will take several minutes):

```
Cmd 6
flightPerf.count()

(1) Spark Jobs
res9: Long = 161783211
1
```

- Sample the data by selecting few years:

```
Cmd 7
import org.apache.spark.sql.functions._

val flightPerfSample = flightPerf.filter(col("Year").isin("2017", "2016", "2015", "2014"))

flightPerfSample: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Year: string, Quarter: string ... 108 more fields]
import org.apache.spark.sql.functions._
flightPerfSample: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Year: string, Quarter: string ... 108 more fields]
```

- Look at the data structure:

```
flightPerfSample.printSchema()
```

Output:

```
Cmd 8

1  flightPerfSample.printSchema()

root
 |-- Year: string (nullable = true)
 |-- Quarter: string (nullable = true)
 |-- Month: string (nullable = true)
 |-- DayofMonth: string (nullable = true)
 |-- DayOfWeek: string (nullable = true)
 |-- FlightDate: string (nullable = true)
 |-- UniqueCarrier: string (nullable = true)
 |-- AirlineID: string (nullable = true)
 |-- Carrier: string (nullable = true)
 |-- TailNum: string (nullable = true)
 |-- FlightNum: string (nullable = true)
 |-- OriginAirportID: string (nullable = true)
 |-- OriginAirportSeqID: string (nullable = true)
 |-- OriginCityMarketID: string (nullable = true)
 |-- Origin: string (nullable = true)
 |-- OriginCityName: string (nullable = true)
 |-- OriginState: string (nullable = true)
 |-- OriginStateFips: string (nullable = true)
 |-- OriginStateName: string (nullable = true)
 |-- OriginWac: string (nullable = true)
```

…

We can see that our dataset has quite a lot of columns!

- Let's display our dataset:

```
flightPerfSample.show()
```

Output… not very readable with our dataset…

```
Cmd 9

1  flightPerfSample.show()

▸ (1) Spark Jobs

|Year|Quarter|Month|DayofMonth|DayOfWeek|FlightDate|UniqueCarrier|AirlineID|Carrier|TailNum|FlightNum|OriginAirportID|OriginAirportSeqID|OriginCityMark
ginState|OriginStateFips|OriginStateName|OriginWac|DestAirportID|DestAirportSeqID|DestCityMarketID|Dest|      DestCityName|DestState|DestStateFips|  Dest
|DepDelay|DepDelayMinutes|DepDel15|DepartureDelayGroups|DepTimeBlk|TaxiOut|WheelsOff|WheelsOn|TaxiIn|CRSArrTime|ArrTime|ArrDelay|ArrDelayMinutes|ArrDe
celled|CancellationCode|Diverted|CRSElapsedTime|ActualElapsedTime|AirTime|Flights|Distance|DistanceGroup|CarrierDelay|WeatherDelay|NASDelay|SecurityDe
talAddGTime|LongestAddGTime|DivAirportLandings|DivReachedDest|DivActualElapsedTime|DivArrDelay|DivDistance|Div1Airport|Div1AirportID|Div1AirportSeqID|
stGTime|Div1WheelsOff|Div1TailNum|Div2Airport|Div2AirportID|Div2AirportSeqID|Div2WheelsOn|Div2TotalGTime|Div2LongestGTime|Div2WheelsOff|Div2TailNum|Di
qID|Div3WheelsOn|Div3TotalGTime|Div3LongestGTime|Div3WheelsOff|Div3TailNum|Div4Airport|Div4AirportID|Div4AirportSeqID|Div4WheelsOn|Div4TotalGTime|Div4
m|Div5Airport|Div5AirportID|Div5AirportSeqID|Div5WheelsOn|Div5TotalGTime|Div5LongestGTime|Div5WheelsOff|Div5TailNum|_c109|
```

- We can select specific columns:

```
flightPerfSample.select("AirlineID","FlightDate")show()
```

Output:

```
Cmd 11

flightPerfSample.select("AirlineID","FlightDate").show()
```

```
▸ (1) Spark Jobs
+---------+----------+
|AirlineID|FlightDate|
+---------+----------+
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
|    20355|2014-07-19|
+---------+----------+
only showing top 20 rows
```

- Apply some filter and show only 1 row:

```
flightPerfSample.select("origin", "dest").filter(col("AirlineID") ===
19805).show(1)
```

```
+------+----+
|Origin|Dest|
+------+----+
|   JFK| LAX|
+------+----+
only showing top 1 row
```

- We can also rename the output columns:

```
flightPerfSample.select(col("origin").as("FROM"),
col("dest").as("TO")).filter(col("AirlineID") === 19805).show(1))
```

Running SQL Queries

- To register the DataFrame as SQL table copy below code in empty cell and execute it

```
flightPerfSample.createOrReplaceTempView("flights")
```

- Then we can work with SQL query using the table we just created

```
spark.sql("show tables").show()
```

```
Cmd 16

spark.sql("show tables").show()
```

```
+--------+---------+-----------+
|database|tableName|isTemporary|
+--------+---------+-----------+
|        |  flights|       true|
+--------+---------+-----------+
```

- Execute below SQL query and show 10 first lines using the methods we saw above

```
SELECT AirlineId, Year, AirTime, CancellationCode, Cancelled, DepDelay, Dest, Distance, Origin, UniqueCarrier FROM flights
```

Output:

Cmd 17

```
spark.sql("SELECT AirlineId, Year, AirTime, CancellationCode, Cancelled, DepDelay, Dest, Distance, Origin, UniqueCarrier FROM flights").show()
```

(1) Spark Jobs

AirlineId	Year	AirTime	CancellationCode	Cancelled	DepDelay	Dest	Distance	Origin	UniqueCarrier
20355	2014	239.00	null	0.00	11.00	CLT	2125.00	LAX	US
20355	2014	335.00	null	0.00	-8.00	PHX	2979.00	LIH	US
20355	2014	375.00	null	0.00	0.00	LIH	2979.00	PHX	US
20355	2014	340.00	null	0.00	-7.00	PHX	2979.00	LIH	US
20355	2014	319.00	null	0.00	-7.00	SFO	2521.00	PHL	US
20355	2014	131.00	null	0.00	94.00	PHL	1013.00	MIA	US
20355	2014	163.00	null	0.00	1.00	MIA	1013.00	PHL	US
20355	2014	242.00	null	0.00	-1.00	CLT	2125.00	LAX	US
20355	2014	255.00	null	0.00	1.00	LAS	1916.00	CLT	US
20355	2014	217.00	null	0.00	-3.00	CLT	1916.00	LAS	US
20355	2014	270.00	null	0.00	-4.00	LAX	2125.00	CLT	US
20355	2014	136.00	null	0.00	111.00	PHL	1013.00	MIA	US
20355	2014	136.00	null	0.00	18.00	MIA	1013.00	PHL	US
20355	2014	108.00	null	0.00	-12.00	DCA	814.00	TPA	US
20355	2014	130.00	null	0.00	-3.00	MSP	930.00	CLT	US
20355	2014	90.00	null	0.00	-6.00	CLT	590.00	PBI	US
20355	2014	67.00	null	0.00	2.00	PHL	449.00	CLT	US
20355	2014	74.00	null	0.00	32.00	CLT	449.00	PHL	US
20355	2014	300.00	null	0.00	-5.00	PHL	2521.00	SFO	US
20355	2014	315.00	null	0.00	18.00	SFO	2521.00	PHL	US

only showing top 20 rows

- Let's find out how many rows we have per year:

```
SELECT count(*), Year FROM flights GROUP BY Year ORDER BY Year
```

Output:

(1) Spark Jobs

Year	count
2014	5819811
2015	5819079
2016	5617658
2017	5210416

- Verify that the counts are similar here:

```
flightPerfSample.groupBy("YEAR").count().sort("YEAR").show()
```

Notice that 2017 has significantly less flights, and it makes sense because the data is not complete. But what is the last month?

Perform operations on data frames to analyze the data

Use some analytic functions

Some useful functions:

- **groupBy(*cols)**: Groups the DataFrame using specified columns, in order to run aggregation on them.
- **count():** Returns the number of rows in DataFrame.
- **collect():** Returns all records as list of row.
- **orderBy(*cols, ascending=True/False):** Returns a new DataFrame sorted by the specified columns.
- **avg(*args):** Computes average values for each numeric column for each group.
- **sum(*args):** Computes sum for each numeric column for each group.

- Get the number of arrival flights by state in 2014

```
flightPerfSample.filter(col("Year") ===
2014).groupBy("DestStateName").count().show()
```

Output:

```
+--------------------+------+
|       DestStateName| count|
+--------------------+------+
|                Utah|112078|
|              Hawaii| 96499|
| U.S. Virgin Islands|  5123|
|           Minnesota|113085|
|U.S. Pacific Trus...|   479|
|                Ohio| 82027|
|              Oregon| 65458|
|            Arkansas| 27904|
|               Texas|717767|
|        North Dakota| 15250|
|        Pennsylvania|109574|
|         Connecticut| 21780|
|            Nebraska| 22837|
|             Vermont|  4197|
|              Nevada|153365|
|         Puerto Rico| 28114|
|          Washington|121792|
|            Illinois|391833|
|            Oklahoma| 40491|
|            Delaware|   711|
+--------------------+------+
only showing top 20 rows
```

- **Try by yourself**: Select top 5 States from previous output

- **Try by yourself**: For those 5 states, calculate the number of flights variation (in %), year over year (from 2014 to 2015, and 2015 to 2016).

Here is the desired output:

```
▸ (6) Spark Jobs
▸ res: org.apache.spark.sql.DataFrame = [DestStateName: string, 2014: long ... 6 more fields]
+-------------+------+------+------+------+------------------+-------------------+-------------------+
|DestStateName|  2014|  2015|  2016|  2017|      2014_to_2015|       2015_to_2016|       2016_to_2017|
+-------------+------+------+------+------+------------------+-------------------+-------------------+
|        Texas|717767|688031|578440|511105|-4.142848584568526|-15.928206723243576| -11.64079247631561|
|     Illinois|391833|418264|340426|334513| 6.745475751149073|-18.609777556758416|-1.7369413617056182|
|      Georgia|386052|394412|398518|348187|2.1655113818863896| 1.0410433759622038|-12.629542454795015|
|      Florida|428056|449248|447168|416851| 4.950754106939286|-0.4629959398817647| -6.779778517246314|
|   California|739170|707762|727407|692216|-4.249090195760104|  2.775650571802373|-4.8378693083789415|
+-------------+------+------+------+------+------------------+-------------------+-------------------+

res: org.apache.spark.sql.DataFrame = [DestStateName: string, 2014: bigint ... 6 more fields]
```

There are multiple ways of achieving this, for example:

- We could filter the dataset in order to have only the states we found in the last query
- Next we can group the data per **DestinationStateName** and pivot per **Year**
- Then we can count the number of rows
- And finally compute the difference between 2014 and 2015, 2015 and 2016

Bonus: Can you try to do this with a window fonction ?

Learn how to JOIN dataset

- Load another dataset containing the Cancellation References

Cmd 23

```
val root = "wasbs://datalake@cbotek.blob.core.windows.net"
val refAnnulations = spark.read.option("header", "true").csv(s"${root}/References/RefAnnulations.csv")
refAnnulations.show()
```

▸ (2) Spark Jobs
▸ refAnnulations: org.apache.spark.sql.DataFrame = [Code: string, Description: string]

```
+----+--------------------+
|Code|         Description|
+----+--------------------+
|   A|             Carrier|
|   B|             Weather|
|   C|National Air System|
|   D|            Security|
+----+--------------------+

root: String = wasbs://datalake@cbotek.blob.core.windows.net
refAnnulations: org.apache.spark.sql.DataFrame = [Code: string, Description: string]
```

- Show top 5 origin cities having the most flight cancellation

```
flightPerfSample
.filter(col("Cancelled") === 1)
.groupBy("OriginCityName", "CancellationCode")
.count()
.orderBy(desc("count"))
.join(refAnnulations, col("CancellationCode") === col("Code")).show()
```

Output:

```
+--------------------+----------------+-----+----+-------------------+
|      OriginCityName|CancellationCode|count|Code|        Description|
+--------------------+----------------+-----+----+-------------------+
|         Chicago, IL|               B|20367|   B|            Weather|
|Dallas/Fort Worth...|               B|11357|   B|            Weather|
|        New York, NY|               B|10672|   B|            Weather|
|         Chicago, IL|               C| 9822|   C|National Air System|
|         Atlanta, GA|               B| 9703|   B|            Weather|
|         Houston, TX|               B| 9401|   B|            Weather|
|          Newark, NJ|               C| 7153|   C|National Air System|
|         Chicago, IL|               A| 6534|   A|            Carrier|
|          Denver, CO|               B| 6528|   B|            Weather|
|        New York, NY|               C| 6228|   C|National Air System|
|Dallas/Fort Worth...|               A| 5500|   A|            Carrier|
|   San Francisco, CA|               B| 5421|   B|            Weather|
|          Boston, MA|               B| 5404|   B|            Weather|
|      Washington, DC|               B| 5375|   B|            Weather|
|        New York, NY|               A| 5303|   A|            Carrier|
|          Newark, NJ|               B| 4995|   B|            Weather|
|     Los Angeles, CA|               A| 4834|   A|            Carrier|
|       Baltimore, MD|               B| 4215|   B|            Weather|
|         Atlanta, GA|               A| 4112|   A|            Carrier|
|         Orlando, FL|               B| 4093|   B|            Weather|
+--------------------+----------------+-----+----+-------------------+
```

Data type conversion and statistical functions

One of the main advantage of PySpark/Scala over SQL is the access to a ton of libraries, for statistical purpose and matrix calculation for example.

- As a simple example, calculate the correlation coefficient between the AIR_TIME and DISTANCE. For that we can use the function “corr”, taking in arguments 2 columns of a dataframe (using the Pearson method).

  - Let’s try this:

```
flightPerfSample.stat.corr("AirTime", "Distance")
```

Output:

```
java.lang.IllegalArgumentException: requirement failed: Currently correlation calculation for columns with dataType string not supported.
	at scala.Predef$.require(Predef.scala:224)
	at org.apache.spark.sql.execution.stat.StatFunctions$$anonfun$collectStatisticalData$3.apply(StatFunctions.scala:159)
	at org.apache.spark.sql.execution.stat.StatFunctions$$anonfun$collectStatisticalData$3.apply(StatFunctions.scala:157)
	at scala.collection.immutable.List.foreach(List.scala:392)
	at org.apache.spark.sql.execution.stat.StatFunctions$.collectStatisticalData(StatFunctions.scala:157)
	at org.apache.spark.sql.execution.stat.StatFunctions$.pearsonCorrelation(StatFunctions.scala:109)
	at org.apache.spark.sql.DataFrameStatFunctions.corr(DataFrameStatFunctions.scala:160)
	at org.apache.spark.sql.DataFrameStatFunctions.corr(DataFrameStatFunctions.scala:180)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:61)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:63)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:65)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:67)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:69)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:71)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:73)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:75)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:77)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:79)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:81)
	at line5af61aaa5dba49f68051686d69538d60134.$read$$iw$$iw$$iw$$iw.<init>(command-1987597012917638:83)
```

Oops… corr function is based on numeric values, and it looks like Spark is not automatically converting our strings into numeric values.

  - Manually cast the data and assign result into a new dataframe:

```
import org.apache.spark.sql.types._
val newFlightPerfSample = flightPerfSample.select(col("AirTime").cast(FloatType),
$"Distance" cast "float")
```

  - Try again the correlation calculation

```
newFlightPerfSample.stat.corr("AIR_TIME","DISTANCE")
```

Output:

```
import org.apache.spark.sql.types._
newFlightPerfSample: org.apache.spark.sql.DataFrame = [AirTime: float, Distance: float]
res51: Double = 0.9615025690006345
```

Nearly perfect correlation (coefficient is always between -1 and 1), but you already probably guessed it, as this correlation is quite obvious...

Visualize the results

Try by yourself: Find out the State destination with the bigger difference in 2016, in term of number of flights, between 2 months (variation in %).

- To resolve this, first you can build a temp table containing the count of flights by MONTH / DEST_STATE_NAME
- From here you can calculate the variation in % with a window function. Here is how we create a window. We will use this to compute the variation

```
val windowSpec = Window.partitionBy("DestStateName").orderBy("Month")
```

Output:

```
+-------------+-----+-----+--------------+
|DestStateName|Month|count|month_to_month|
+-------------+-----+-----+--------------+
|California   |3    |59449|6504          |
|California   |10   |62267|6042          |
|Florida      |3    |43998|5702          |
|Texas        |3    |49840|4644          |
|Georgia      |3    |34146|4382          |
|Florida      |12   |40383|4201          |
|Illinois     |10   |29801|3817          |
|Illinois     |3    |27950|3428          |
|Georgia      |10   |34004|3092          |
|Florida      |11   |36182|2920          |
|California   |5    |61471|2889          |
|Colorado     |3    |21613|2874          |
|Arizona      |3    |16280|2348          |
|New York     |3    |22043|2211          |
|California   |7    |65622|2086          |
|California   |6    |63536|2065          |
|Illinois     |5    |29954|2033          |
|California   |12   |61339|1934          |
|Michigan     |10   |13477|1765          |
|Texas        |5    |49070|1706          |
+-------------+-----+-----+--------------+
only showing top 20 rows
```

Spoiler

```
val import org.apache.spark.sql.expressions._
import org.apache.spark.sql.functions._

val countPerMonth = flightPerf.filter(col("YEAR") ===
2016).groupBy("DestStateName", "Month").count()

val windowSpec = Window.partitionBy("DestStateName").orderBy("Month")

val monthToMonthDiff =
countPerMonth
```

```
.withColumn("month_to_month", $"count" - lag($"count", 1).over(windowSpec))
.filter($"month_to_month".isNotNull)
.orderBy($"month_to_month".desc)


monthToMonthDiff.show(false)
```

- Our winner should be California. Let’s visualize the month trend for this state (here we assume that a temp table “flightMonthTable” has been created, containing the count of flights by MONTH, YEAR and DEST_STATE_NM):

```
countPerMonth.filter($"DestStateName" === "California").orderBy($"Month").show(12)
```

Output:

```
+-------------+-----+-----+
|DestStateName|Month|count|
+-------------+-----+-----+
|   California|    1|56225|
|   California|   10|62267|
|   California|   11|59405|
|   California|   12|61339|
|   California|    2|52945|
|   California|    3|59449|
|   California|    4|58582|
|   California|    5|61471|
|   California|    6|63536|
|   California|    7|65622|
|   California|    8|66303|
|   California|    9|60263|
+-------------+-----+-----+
```

Section 3 - Power BI on Spark With Databricks

To Design a Power BI report based on Spark, we need to persist our data into a Hive table.

Dataframe to HIVE

- Create a new notebook
- Create the hive table with this data: the number of flights and average delay (DEP_DELAY) by destination state for each departure city. To be more representative we will only consider the flights having a delay > 1 hour.
    - Re-create the flight sample dataframe from the previous part

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

spark.conf.set(
  "fs.azure.account.key.cbotek.blob.core.windows.net",

"0tCbVawj0BniiLxMgJfeq878iWV8MqUYp3klz76+67wvtUOKDShSRS4MCclv/PYQQrZNNxcj+l7sk6BUBd
kcYA==")
val root = "wasbs://datalake@cbotek.blob.core.windows.net"

val refAnnulations = spark.read.option("header",
"true").csv(s"${root}/References/RefAnnulations.csv")

val airports = spark.read.csv(s"${root}/References/Airports.csv")

val routes = spark.read.option("header",
"false").csv(s"${root}/References/Routes.csv")

val data2014 = spark.read.option("header", "true").csv(s"${root}/Flight/2014/*")
val data2015 = spark.read.option("header", "true").csv(s"${root}/Flight/2015/*")
val data2016 = spark.read.option("header", "true").csv(s"${root}/Flight/2016/*")
val data2017 = spark.read.option("header", "true").csv(s"${root}/Flight/2017/*")
val flightPerfSample = data2014.union(data2015).union(data2016).union(data2017)
```

- Build our query and assign it to a new dataframe

```
# Register a temp table
flightPerfSample.registerTempTable("departureDelays")

# New dataframe
val AvgDelay =
spark
.sql("SELECT  OriginCityName, DestStateName, 'United States' as Country,
AVG(DepDelay) as AverageDelay, COUNT(*) as DelayFrequency FROM departureDelays
WHERE DepDelay > 60 GROUP BY OriginCityName, DestStateName")
AvgDelay.createOrReplaceTempView("avgDelay")
```

- You can check if the table was created successfully by calling ‘show tables’

```
spark.sql(“show tables”).show()
```

- At this point our analysis table is temporary

Cmd 4

```
spark.sql("SHOW TABLES").show()
```

```
+--------+---------------+-----------+
|database|      tableName|isTemporary|
+--------+---------------+-----------+
|        |       avgdelay|       true|
|        |departuredelays|       true|
+--------+---------------+-----------+
```

- In order to create a table in Hive we need to execute the line below:

```
spark.sql("create table DestinationStateAverageDelayAnalysis as select * from
avgDelay")
```

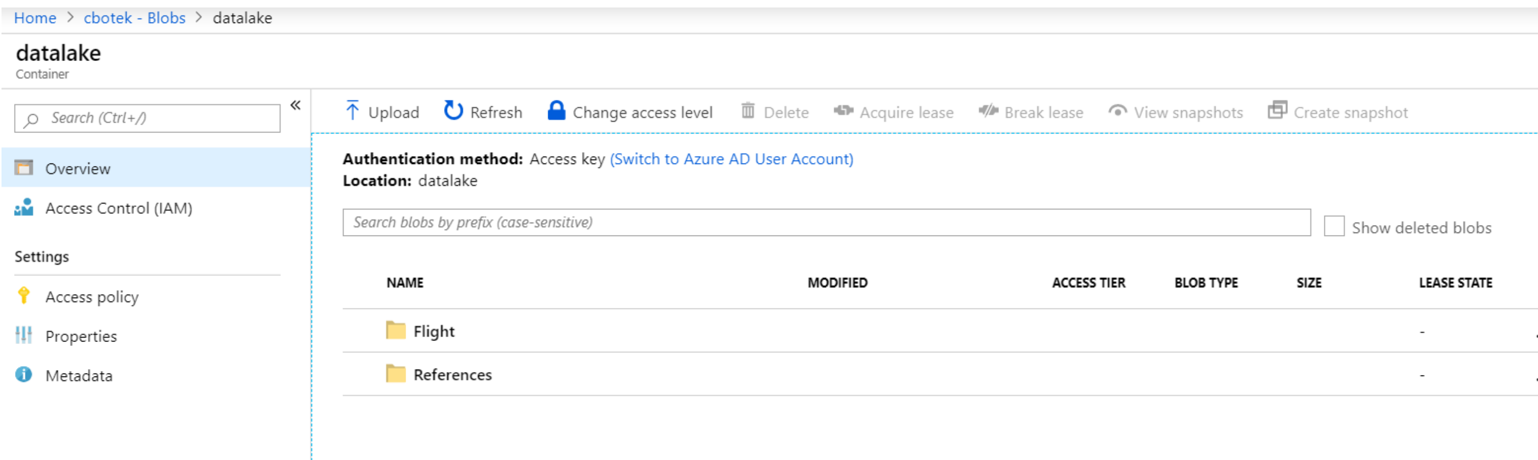
- Let's see what's the difference now when we execute show tables again

```
spark.sql("SHOW TABLES").show()
```

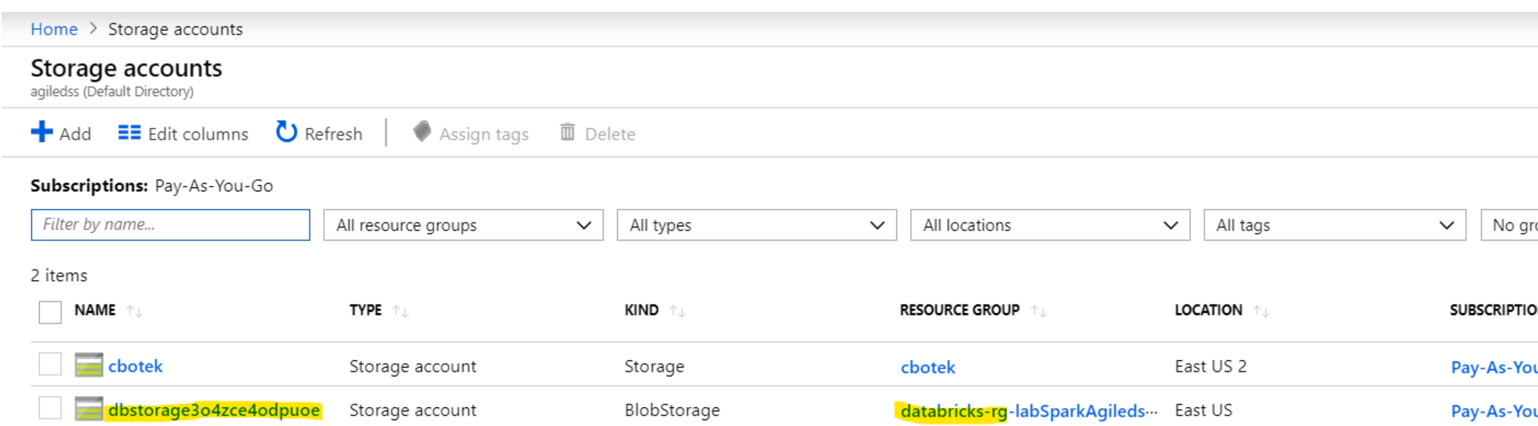
Please compare your result and make sure you all have something similar

```
+--------+------------------------------------+-----------+
|database|tableName                           |isTemporary|
+--------+------------------------------------+-----------+
|default |destinationstateaveragedelayanalysis|false      |
|        |avgdelay                            |true       |
|        |departuredelays                     |true       |
+--------+------------------------------------+-----------+
```

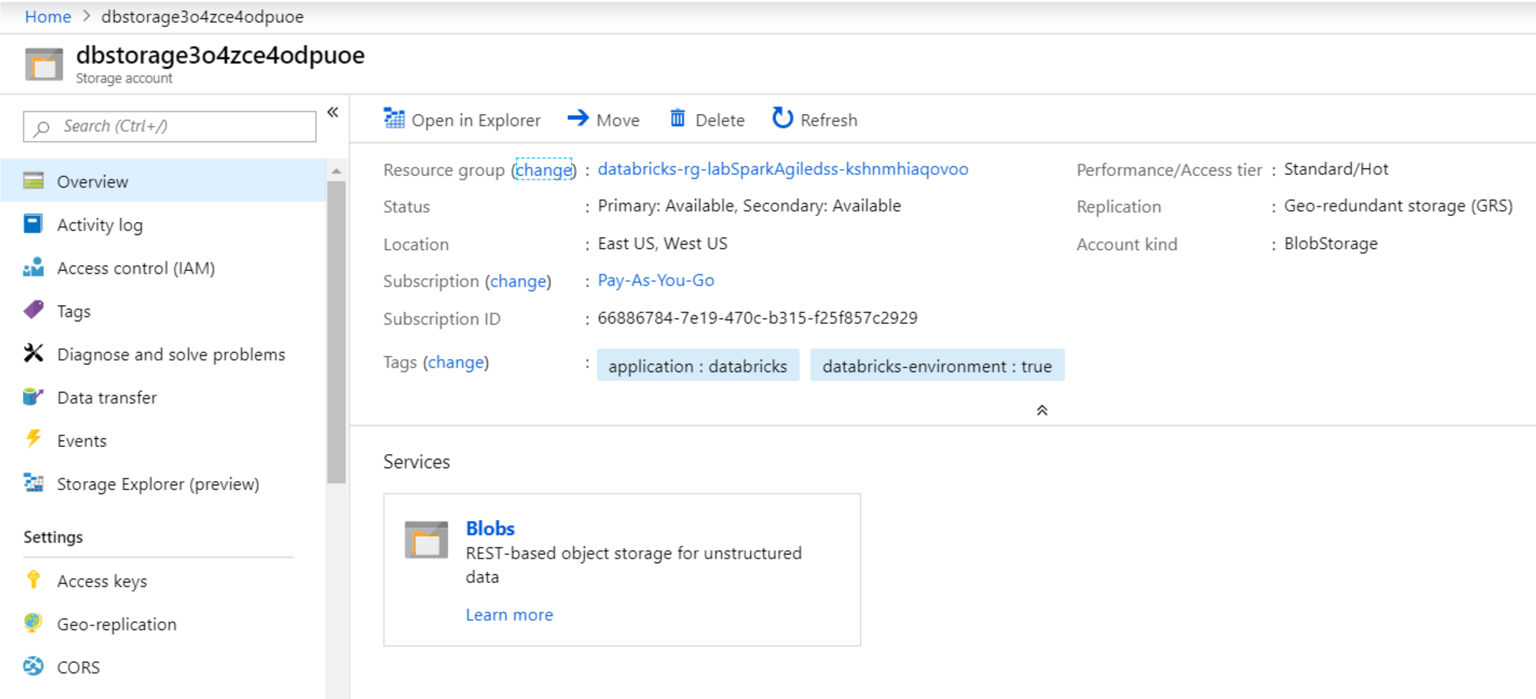
- At this point if we go back to our blob storage we do not see any differences. So where does this table was save exactly?



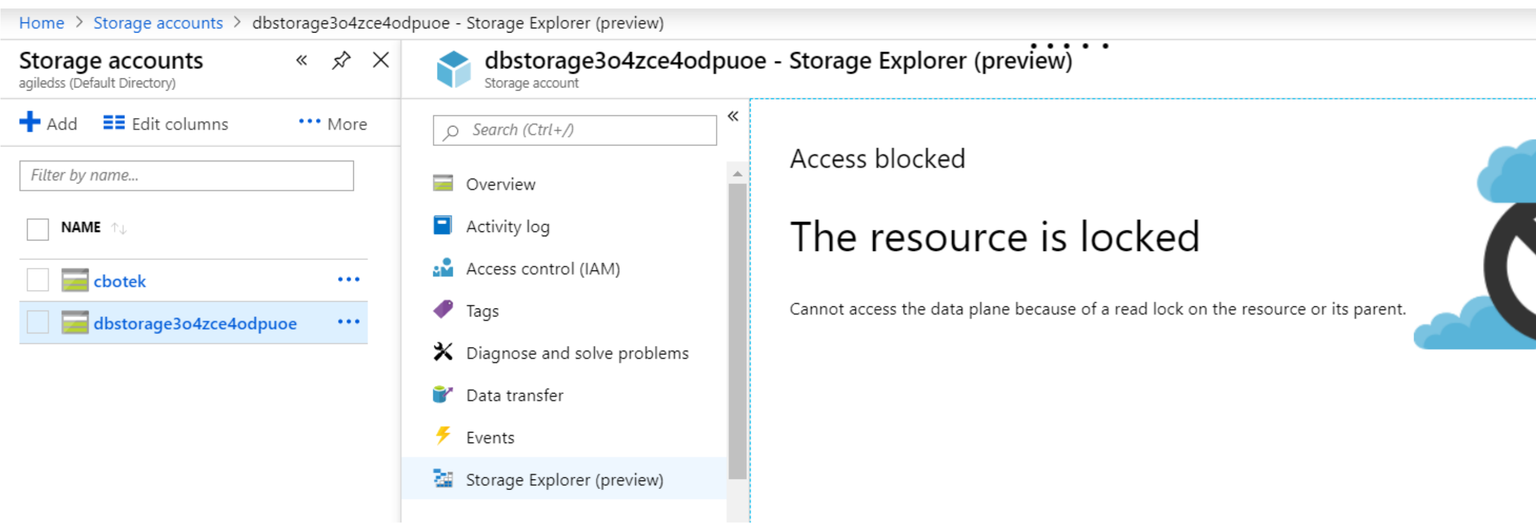
- Databricks is storing its meta data on a different blob storage which we cannot access
- If we go back to the azure portal you should see a blob storage with a name similar to mine:



If we click on it we can look at the details



But we cannot access the files



- Now go back to the notebook tab, and type the following command to query your table

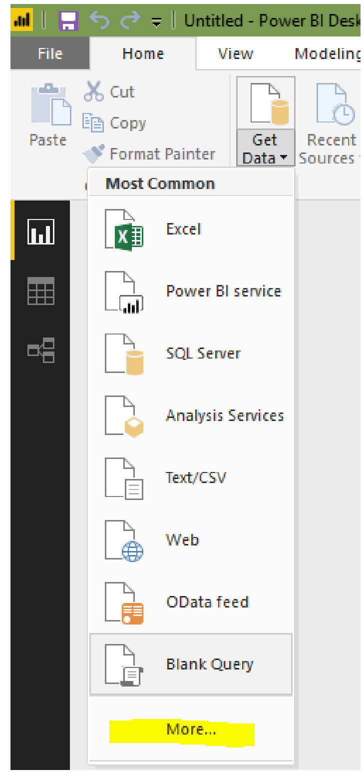
```
spark.sql("Select * from destinationStateAverageDelayAnalysis Limit 5").show(false)
```

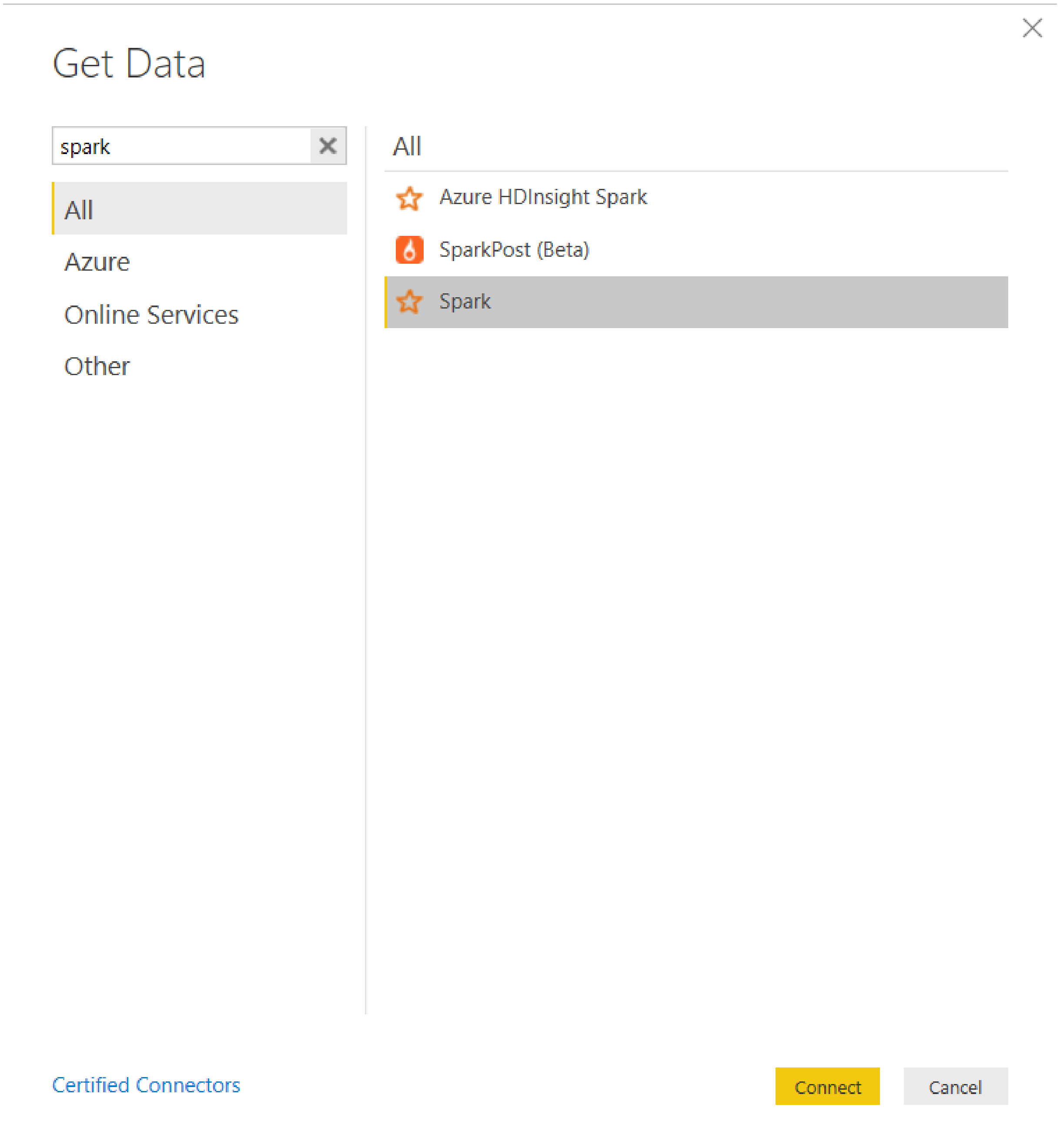
```
+--------------+--------------+-------------+------------------+--------------+
|OriginCityName|DestStateName |Country      |AverageDelay      |DelayFrequency|
+--------------+--------------+-------------+------------------+--------------+
|Sacramento, CA|North Carolina|United States|112.52272727272727|44            |
|Chicago, IL   |Massachusetts |United States|118.01713673687969|2801          |
|Baltimore, MD |New York      |United States|112.8171466845278 |1493          |
|Tampa, FL     |Indiana       |United States|118.55932203389831|236           |
|Pittsburgh, PA|Tennessee     |United States|105.38461538461539|39            |
+--------------+--------------+-------------+------------------+--------------+
```

Connect an Azure Databricks Spark Datasource

In this exercise, you'll connect Power BI to the previous hive table.

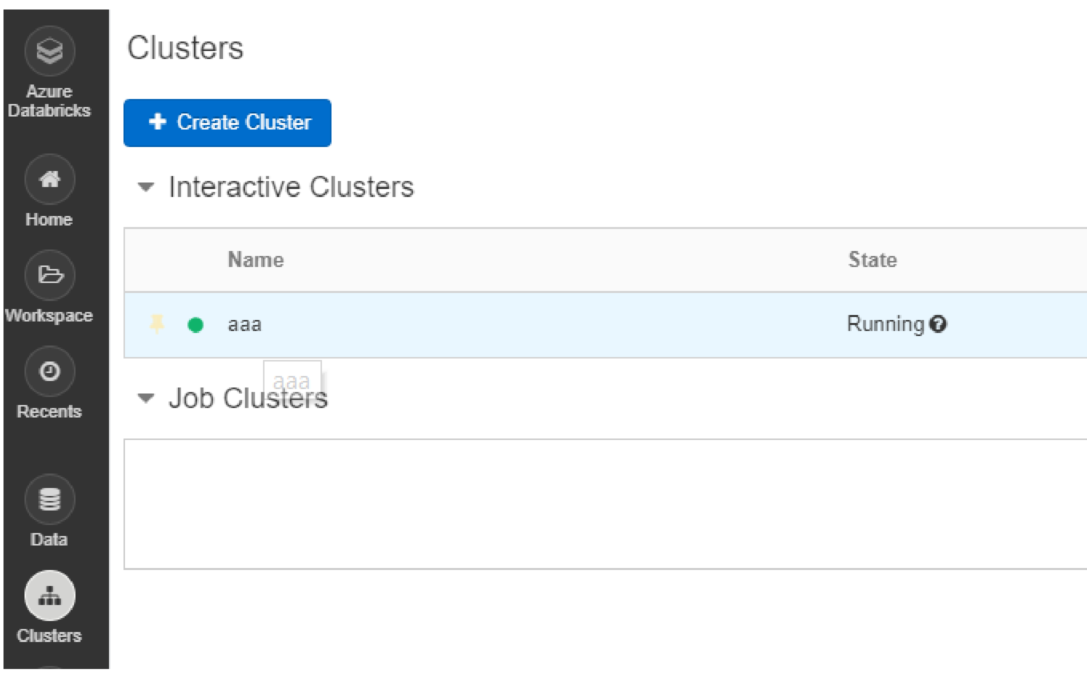
- Open you Microsoft Power BI Desktop application
- With a new report, inside the **Home** tab, expand the **Get Datasource** menu and select the **More…** option
- In the **Get Data** dialog window, on the left side, select **Spark**.



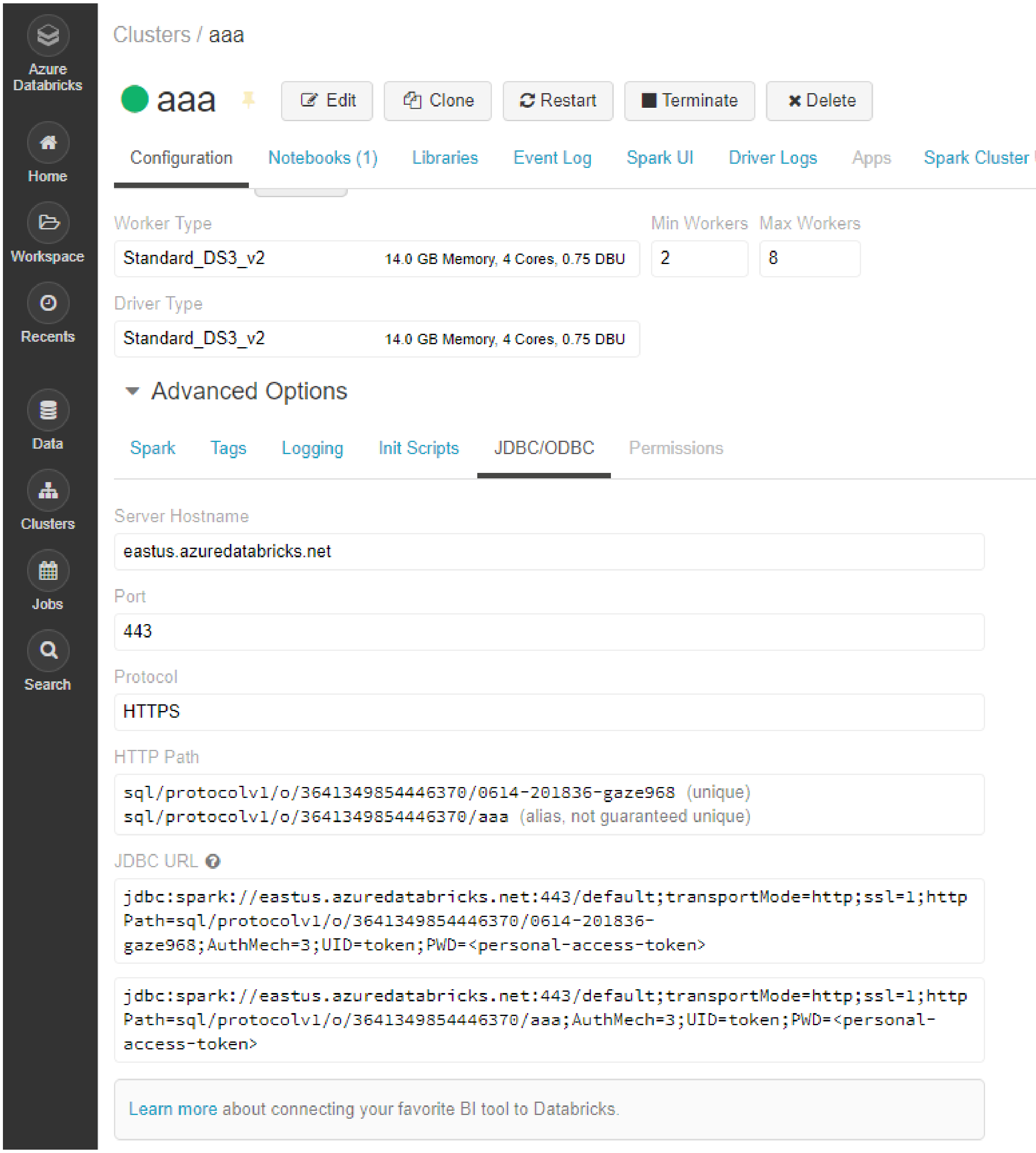


- Click “**Connect**”

- In order to find these informations, let’s go back to Databricks and click on the left hand side on **Clusters**



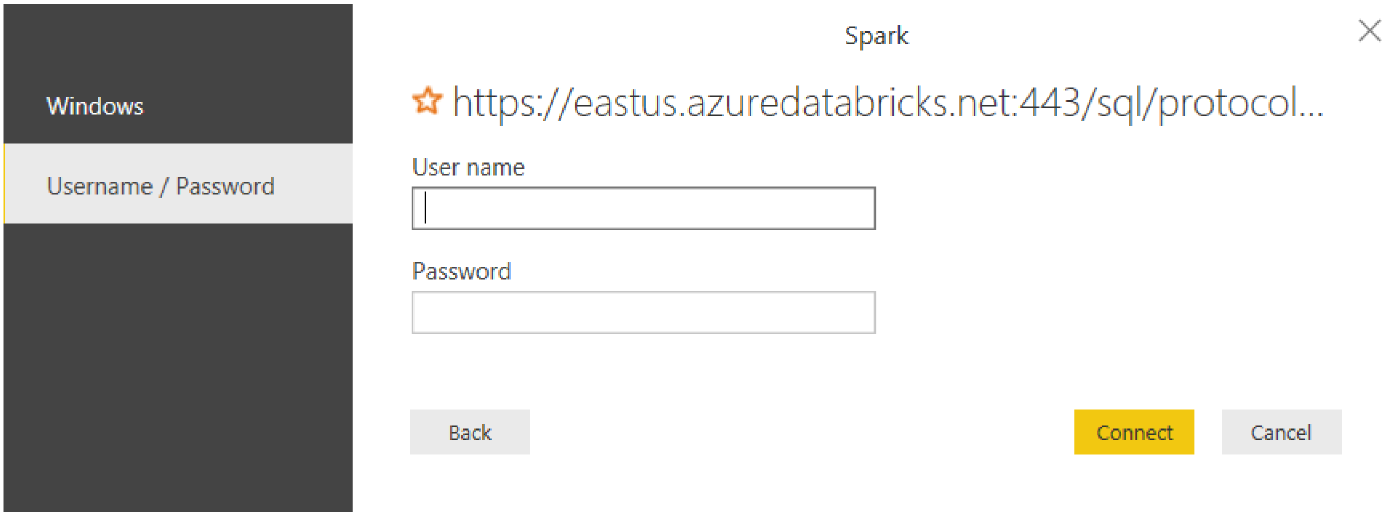
- Then click on your cluster



Extract the base url: `eastus.azuredatabricks.net:443` and add your unique HTTP Path: `sql/protocolv1/o/3641349854446370/0614-201836-gaze968`

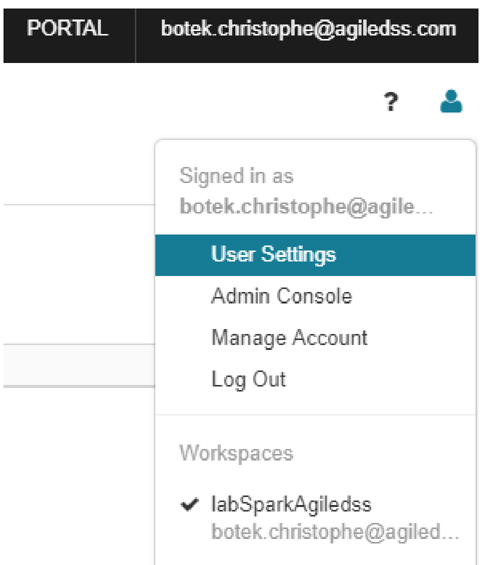
Here is the final url to put in Power BI:
`https://eastus.azuredatabricks.net:443/sql/protocolv1/o/3641349854446370/0614-201836-gaze968`

- Copy the url and paste it in Power BI, then click on connect

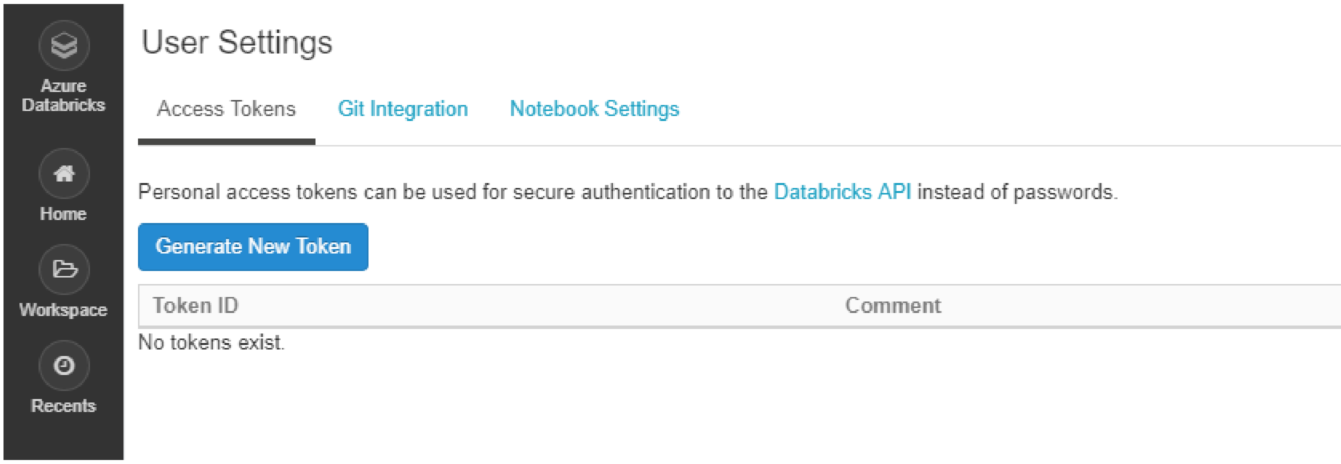


Make sure you see this page or ask for help ☺

- Now let's resolve the username/password in order to connect to our cluster. Go back to Databricks and click on the top right corner, **User settings**

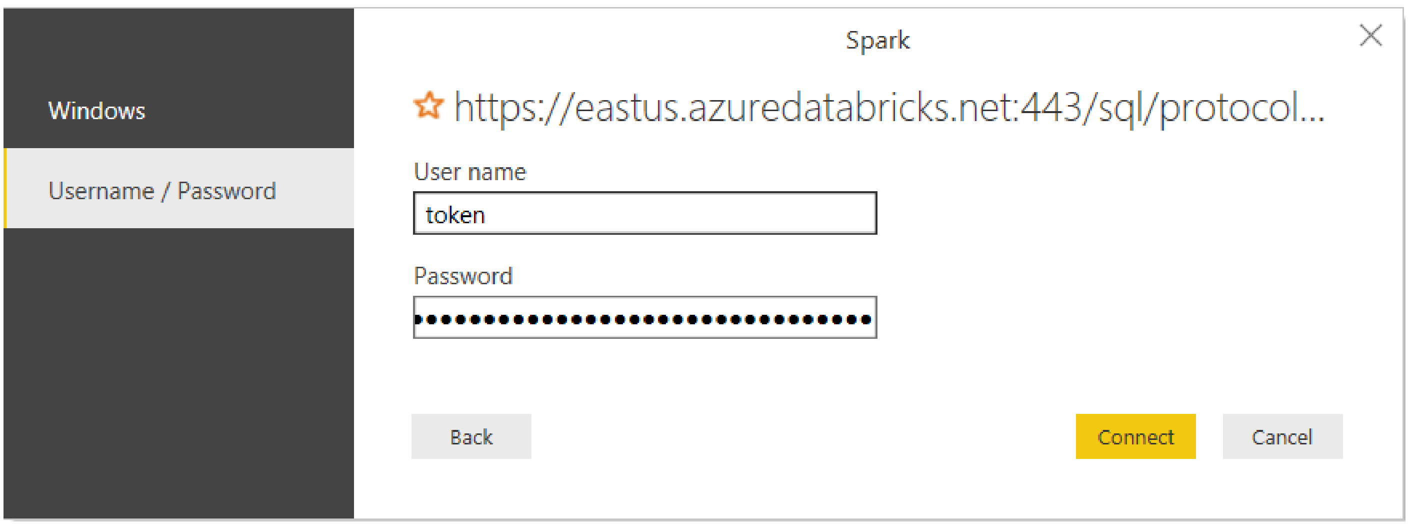


You should see this



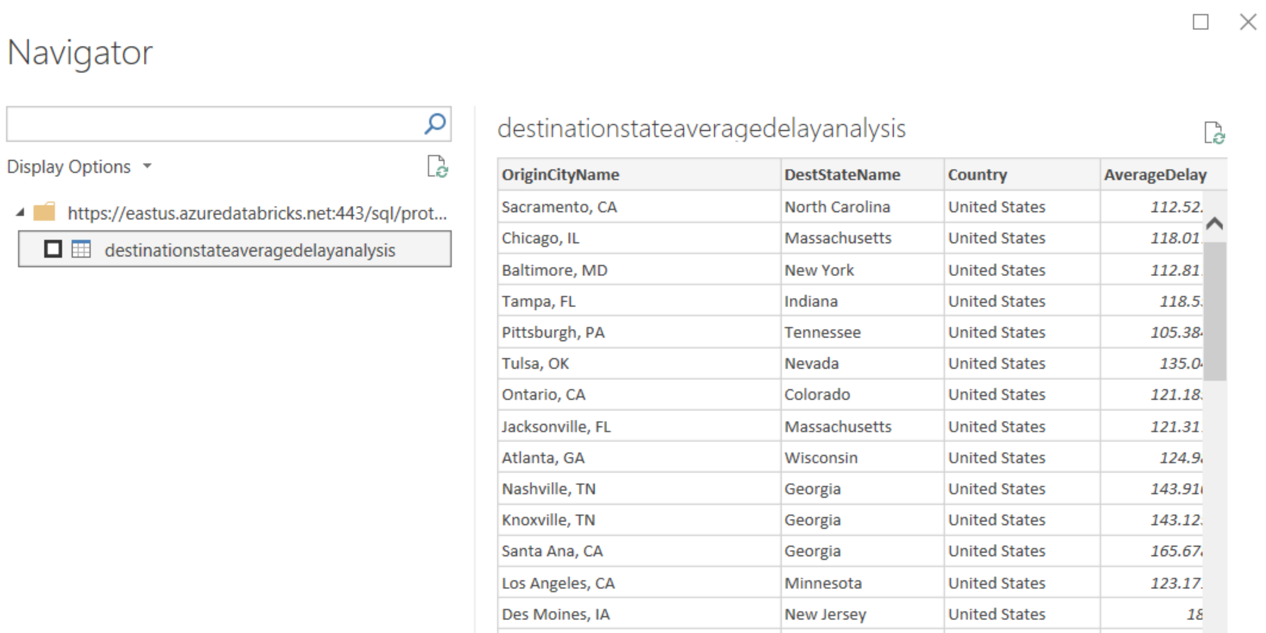
- Now click on **Generate New Token**, then ok, and then copy the token

- Now we can go back to Power BI, paste the token in the password field and enter 'token' as username. Like so:



- Click "**Connect**".

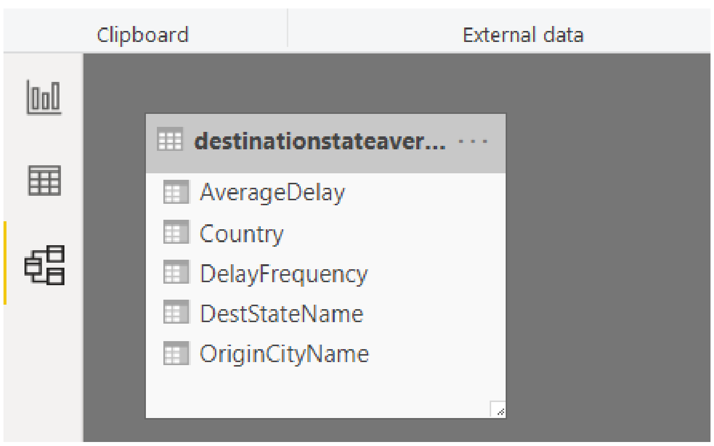
- In the Navigator dialog window, expand the HIVE database, and then expand **<your_cluster_name>**.azuredatabricks.net
- Make sure you see the Hive table we created earlier.



- Click **Load**.
- Explore your data model in the diagram tab at the left.

The data will be loaded into the Power BI Desktop file.

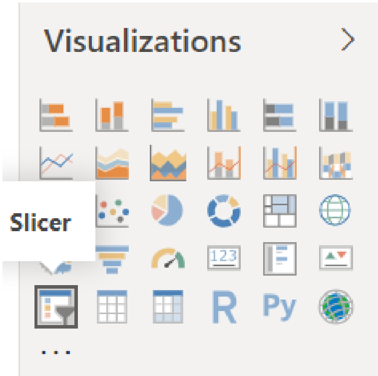
Once loaded, in the **Queries** pane (located at the left), select the query to review the data from the Hive table.



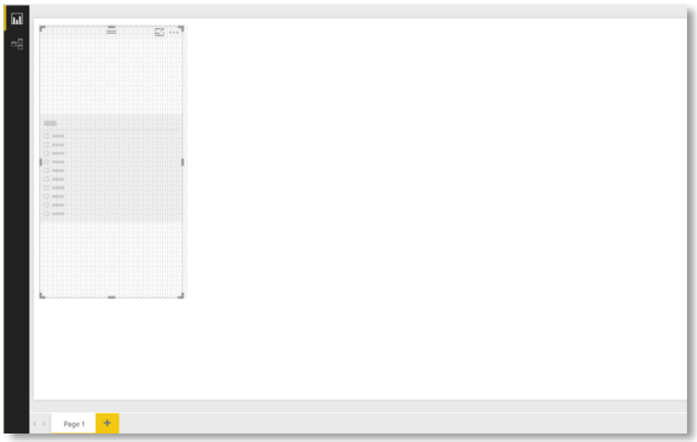
Designing the Power BI report

In this exercise, you will design an interactive report based on the hive table.

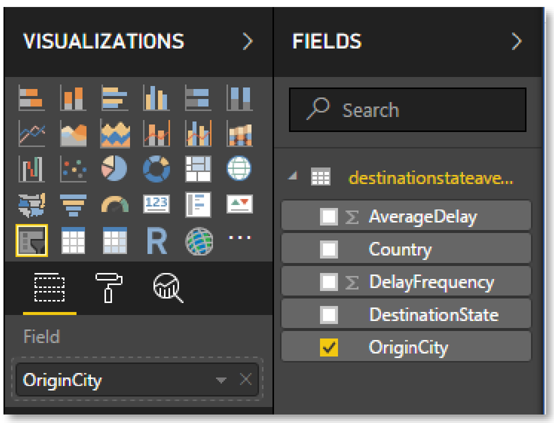
- Go to the report pane
- To add a Segment from inside the Visualization pane, click the **Slicer** icon



- Reposition and resize the visualization based on the following diagram.

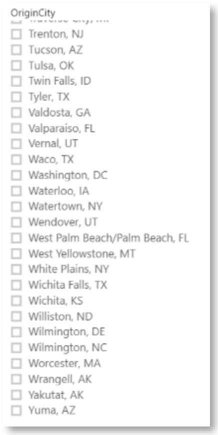


- In the Fields pane (located at the right), Expand the **destinationStateAverageDelayAnalysis** table.



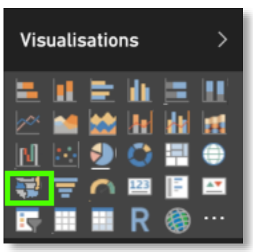
- From the Fields pane, inside the expanded table, check the **OriginCity** field.

Verify that the visualization looks like the following



- To add a Map, from inside the Visualization pane, click on the **Filled Map** icon.

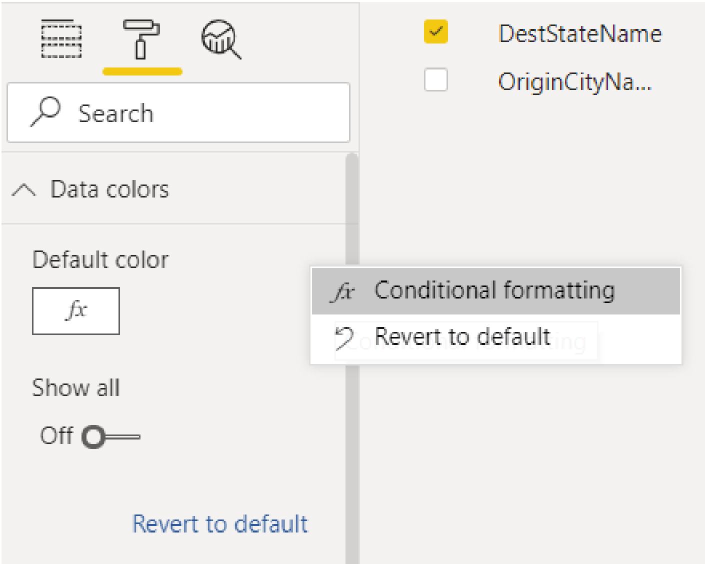
Tips: you can hover the cursor over each icon to reveal a tooltip describing the type of visualization.



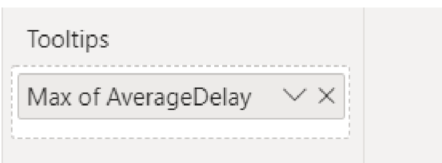
- Reposition and resize the map visualization based on the following diagram.

- From the Fields pane, inside the expanded table, drag the **DestinationState** to Emplacement property and repeat the operation with the **Country** bellow the **DestinationState**.

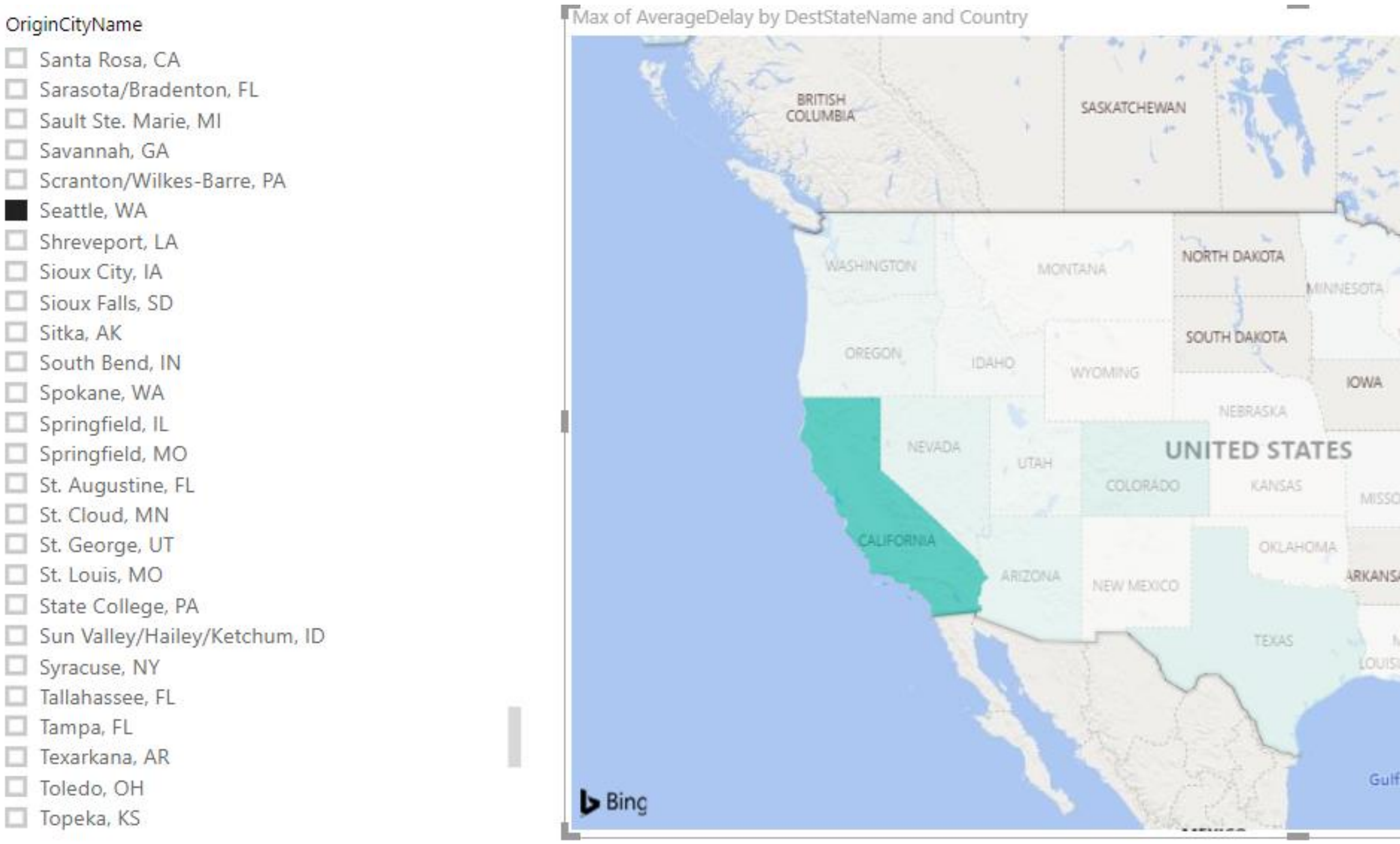
- From the Format pane, click on Data Colors and then on Conditional Formatting. Choose de range of color you like the most to represent the minimum and the maximum values.



- From the Fields pane, from inside the expanded table, drag the **AverageDelay**, to the Tool Tips property. You can play a little with the different calculation offered. I selected the Maximum of Average delay:



- Verify that the visualization looks like the following



User Define Function

In this final exercise you will create a new Hive table, and connect a Power BI visualization on it to display the traffic flow. We will use this exercise to introduce to you the RDD API and the user define functions.

- Go back to the previous notebook and paste the following in order to create a path to the airports CSV file

```
val airportsPath = s"${root}/References/Airports.csv"
```

- Instantiate a dataframe as a textfile this time

```
val airportsDf = spark.read.text(airportsPath)

airportsDf.show(false)
```

Output:

```
+-----------------------------------------------------------------------------------------------------------------------------------------------
|value
+-----------------------------------------------------------------------------------------------------------------------------------------------
|1,"Goroka Airport","Goroka","Papua New Guinea","GKA","AYGA",-6.081689834590001,145.391998291,5282,10,"U","Pacific/Port_Moresby","airp
|2,"Madang Airport","Madang","Papua New Guinea","MAG","AYMD",-5.20707988739,145.789001465,20,10,"U","Pacific/Port_Moresby","airport","
|3,"Mount Hagen Kagamuga Airport","Mount Hagen","Papua New Guinea","HGU","AYMH",-5.826789855957031,144.29600524902344,5388,10,"U","Pac
|4,"Nadzab Airport","Nadzab","Papua New Guinea","LAE","AYNZ",-6.569803,146.725977,239,10,"U","Pacific/Port_Moresby","airport","OurAirp
|5,"Port Moresby Jacksons International Airport","Port Moresby","Papua New Guinea","POM","AYPY",-9.443380355834961,147.22000122070312,
|6,"Wewak International Airport","Wewak","Papua New Guinea","WWK","AYWK",-3.58383011818,143.669006348,19,10,"U","Pacific/Port_Moresby"
|7,"Narsarsuaq Airport","Narssarssuaq","Greenland","UAK","BGBW",61.1604995728,-45.4259986877,112,-3,"E","America/Godthab","airport","O
|8,"Godthaab / Nuuk Airport","Godthaab","Greenland","GOH","BGGH",64.19090271,-51.6781005859,283,-3,"E","America/Godthab","airport","Ou
|9,"Kangerlussuaq Airport","Sondrestrom","Greenland","SFJ","BGSF",67.0122218992,-50.7116031647,165,-3,"E","America/Godthab","airport",
|10,"Thule Air Base","Thule","Greenland","THU","BGTL",76.5311965942,-68.7032012939,251,-4,"E","America/Thule","airport","OurAirports"
|11,"Akureyri Airport","Akureyri","Iceland","AEY","BIAR",65.66000366210938,-18.07270050048828,6,0,"N","Atlantic/Reykjavik","airport","
|12,"Egilsstaðir Airport","Egilsstadir","Iceland","EGS","BIEG",65.2833023071289,-14.401399612426758,76,0,"N","Atlantic/Reykjavik","air
```

- We will need to create a User Defined function in order to split each line into an array, trim the data and remove the double quotes

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

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

val clean: String => Array[String] = _.split(",").map(_.replace("\"", "").trim())

val cleanUDF = udf(clean)

val airportsDfCleaned = airportsDf.withColumn("value", cleanUDF(col("value")))

airportsDfCleaned.show(false)
```

This is a bit better but we will need to get each value into a separated column.

```
+-----------------------------------------------------------------------------------------------------------------------------------------------
|value
+-----------------------------------------------------------------------------------------------------------------------------------------------
|[1, Goroka Airport, Goroka, Papua New Guinea, GKA, AYGA, -6.081689834590001, 145.391998291, 5282, 10, U, Pacific/Port_Moresby, ai
|[2, Madang Airport, Madang, Papua New Guinea, MAG, AYMD, -5.20707988739, 145.789001465, 20, 10, U, Pacific/Port_Moresby, airport,
|[3, Mount Hagen Kagamuga Airport, Mount Hagen, Papua New Guinea, HGU, AYMH, -5.826789855957031, 144.29600524902344, 5388, 10, U,
|[4, Nadzab Airport, Nadzab, Papua New Guinea, LAE, AYNZ, -6.569803, 146.725977, 239, 10, U, Pacific/Port_Moresby, airport, OurAir
|[5, Port Moresby Jacksons International Airport, Port Moresby, Papua New Guinea, POM, AYPY, -9.443380355834961, 147.2200012207031
|[6, Wewak International Airport, Wewak, Papua New Guinea, WWK, AYWK, -3.58383011818, 143.669006348, 19, 10, U, Pacific/Port_Mores
|[7, Narsarsuaq Airport, Narssarssuaq, Greenland, UAK, BGBW, 61.1604995728, -45.4259986877, 112, -3, E, America/Godthab, airport,
|[8, Godthaab / Nuuk Airport, Godthaab, Greenland, GOH, BGGH, 64.19090271, -51.6781005859, 283, -3, E, America/Godthab, airport, O
|[9, Kangerlussuaq Airport, Sondrestrom, Greenland, SFJ, BGSF, 67.0122218992, -50.7116031647, 165, -3, E, America/Godthab, airport
|[10, Thule Air Base, Thule, Greenland, THU, BGTL, 76.5311965942, -68.7032012939, 251, -4, E, America/Thule, airport, OurAirports]
|[11, Akureyri Airport, Akureyri, Iceland, AEY, BIAR, 65.66000366210938, -18.07270050048828, 6, 0, N, Atlantic/Reykjavik, airport,
|[12, Egilsstaðir Airport, Egilsstadir, Iceland, EGS, BIEG, 65.2833023071289, -14.401399612426758, 76, 0, N, Atlantic/Reykjavik, a
```

- In order to do this, we can figure out the number of values in each line and ask spark to create a column for each index:

```
//In our case we counted 15 different values for each line

val airportsDfSplitted = airportsDfCleaned.select((0 until 14).map(i => col("value")(i).alias(s"col_$i")): _*)

airportsDfSplitted.show()
```

Output:

▸ (1) Spark Jobs

▸ airportsDfSplitted: org.apache.spark.sql.DataFrame = [col_0: string, col_1: string ... 12 more fields]

col_0	col_1	col_2	col_3	col_4	col_5	col_6	col_7	col_8	col_9	col_10	
1	Goroka Airport	Goroka	Papua New Guinea	GKA	AYGA	-6.081689834590001	145.391998291	5282	10	U	P
2	Madang Airport	Madang	Papua New Guinea	MAG	AYMD	-5.20707988739	145.789001465	20	10	U	P
3	Mount Hagen Kagam...	Mount Hagen	Papua New Guinea	HGU	AYMH	-5.826789855957031	144.29600524902344	5388	10	U	P
4	Nadzab Airport	Nadzab	Papua New Guinea	LAE	AYNZ	-6.569803	146.725977	239	10	U	P
5	Port Moresby Jack...	Port Moresby	Papua New Guinea	POM	AYPY	-9.443380355834961	147.22000122070312	146	10	U	P
6	Wewak Internation...	Wewak	Papua New Guinea	WWK	AYWK	-3.58383011818	143.669006348	19	10	U	P
7	Narsarsuaq Airport	Narssarssuaq	Greenland	UAK	BGBW	61.1604995728	-45.4259986877	112	-3	E	
8	Godthaab / Nuuk A...	Godthaab	Greenland	GOH	BGGH	64.19090271	-51.6781005859	283	-3	E	
9	Kangerlussuaq Air...	Sondrestrom	Greenland	SFJ	BGSF	67.0122218992	-50.7116031647	165	-3	E	
10	Thule Air Base	Thule	Greenland	THU	BGTL	76.5311965942	-68.7032012939	251	-4	E	
11	Akureyri Airport	Akureyri	Iceland	AEY	BIAR	65.66000366210938	-18.07270050048828	6	0	N	
12	Egilsstaðir Airport	Egilsstadir	Iceland	EGS	BIEG	65.2833023071289	-14.401399612426758	76	0	N	

- Last thing we need to do is to apply a schema to this dataframe

```
val schema =

StructType(List(
  StructField("AirportId", StringType, true),
  StructField("Name", StringType, true),
  StructField("City", StringType, true),
  StructField("Country", StringType, true),
  StructField("IATA", StringType, true),
  StructField("ICAO", StringType, true),
  StructField("Latitude", StringType, true),
  StructField("Longitude", StringType, true),
  StructField("Altitude", StringType, true),
  StructField("Timezone", StringType, true),
  StructField("DST", StringType, true),
  StructField("TzDatabase", StringType, true),
  StructField("Type", StringType, true),
  StructField("Source", StringType, true)))

val airportsWithSchema = airportsDfSplitted.sqlContext.createDataFrame(airportsDfSplitted.rdd, schema)
airportsWithSchema.show(false)
```

- Create temporary view based on the two DataFrames

```
// Creates a temporary view based on the DataFrame
airportsWithSchema.createOrReplaceTempView("airports_na")
flightPerfSample.createOrReplaceTempView("departureDelays")
```

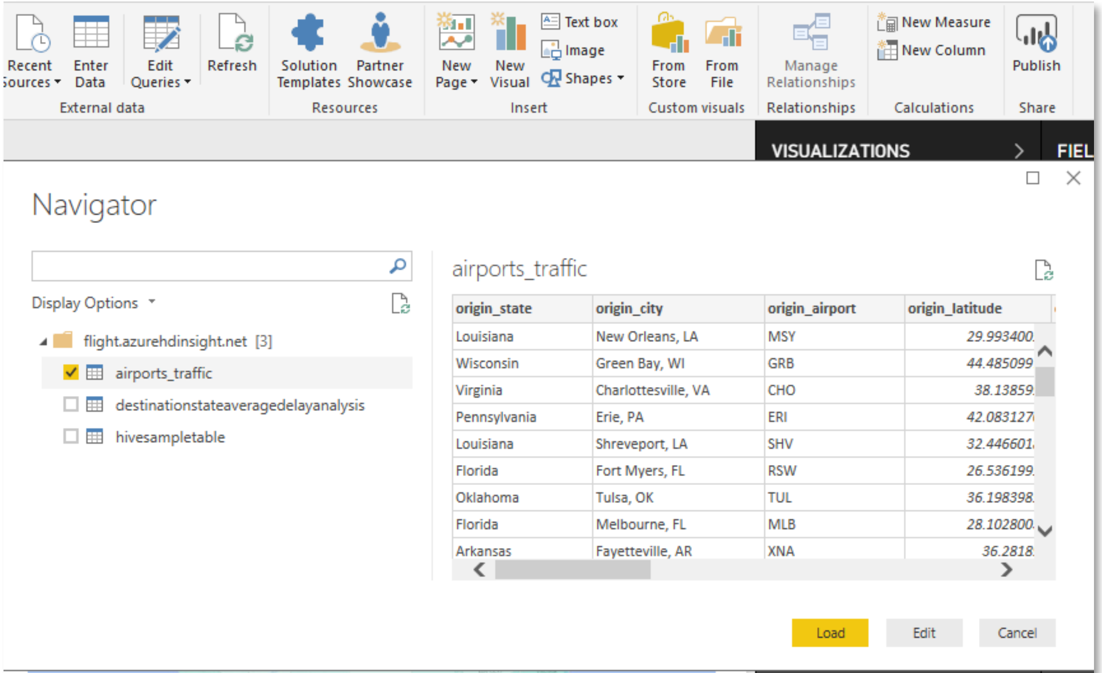
- Do the projection of Flights with the enrichment of the Latitude and Longitude of each Airport's location.

```
// We need to rename the columns and select the ones interesting to our analysis
val flights = flightPerfSample
.select(
  $"OriginStateName".as("origin_state"),
  $"OriginCityName".as("origin_city"),
  $"Origin".as("origin_airport"),
  $"DestStateName".as("destination_state"),
  $"DestCityName".as("destination_city"),
  $"Dest".as("destination_airport"),
  $"DepDelay".as("dep_delay"))

//We also need to cast longitude and latitude as double
val airports = airportsWithSchema
.withColumn("Latitude", $"Latitude".cast(DoubleType))
.withColumn("Longitude", $"Longitude".cast(DoubleType))

//then we can proceed with the aggregation
val airport_traffic = flights
.groupBy("origin_state", "origin_city", "origin_airport", "destination_state", "destination_city", "destination_airport")
.agg(count("*").as("FlightCount"), avg("dep_delay").as("dep_delay"))
.join(airports.select($"IATA", $"Latitude".as("origin_latitude"), $"Longitude".as("origin_longitude")), $"origin_airport" === $"IATA", "left")
.drop("IATA")
.join(airports.select($"IATA", $"Latitude".as("des_latitude"), $"Longitude".as("des_longitude")), $"destination_airport" === $"IATA", "left")

//and finaly we can save the result as a non temporary table
airport_traffic.write.saveAsTable("airports_traffic"))

//check if the table was saved correctly
spark.sql("show tables").show(false)
```

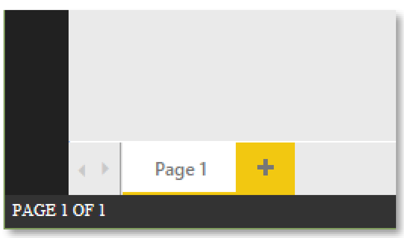
Output:

```
1  spark.sql("show tables").show(false)

+--------+------------------------------------+-----------+
|database|tableName                           |isTemporary|
+--------+------------------------------------+-----------+
|default |airports_traffic                    |false      |
|default |destinationstateaveragedelayanalysis|false      |
|        |airports_na                         |true       |
|        |departuredelays                     |true       |
+--------+------------------------------------+-----------+
```

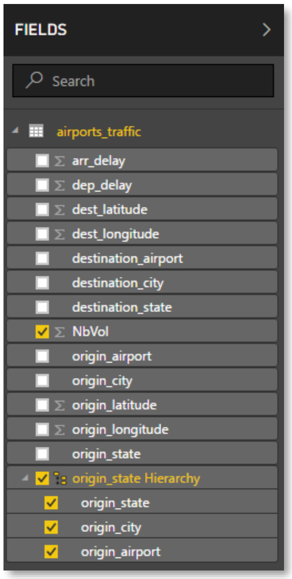
- Return on the Microsoft Power BI Desktop and click on the **Recent Sources** icon in the **Home** ribbon.
- Select **spark clustername** sources, check the new **airports_traffic**, and push the **Load** button.



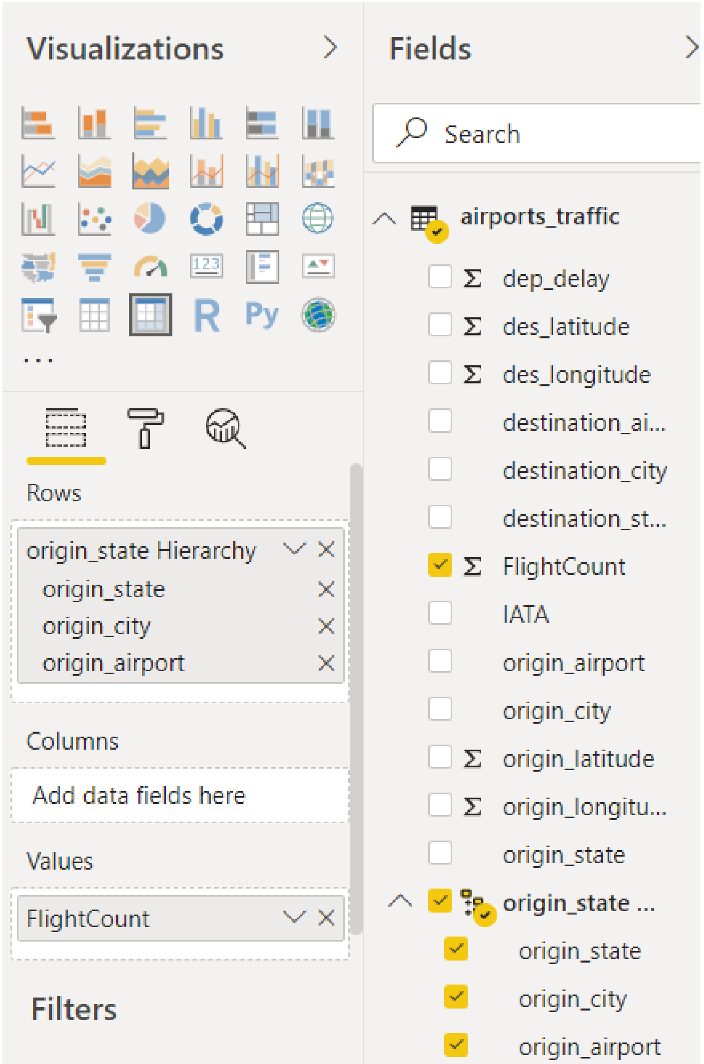
- On your report you can observe the new table in the Fields panel named **airport_traffic**, add a **new page** in the bottom of the report and click on **+**

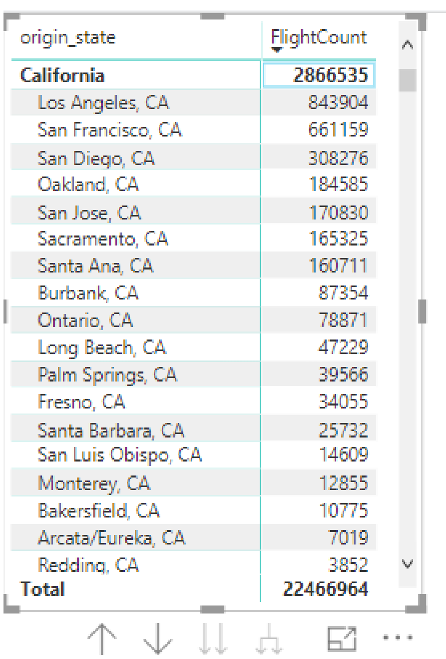


- On your page 2, refactor your airports_traffic field panel: create a new hierarchy, drag and drop the **origin_city** on the **origin_state**, a new field named **origin_state Hierarchy** will be created, continue and add the **origin_airport** by drag and drop.



- Add a matrice visualization, and add the origin_state_hierarchy as row and FlightCount as Value
  a. Sort the matrice by **FlightCount** decreasing

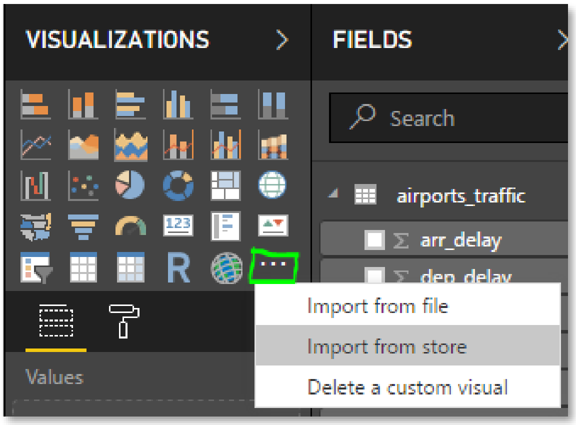




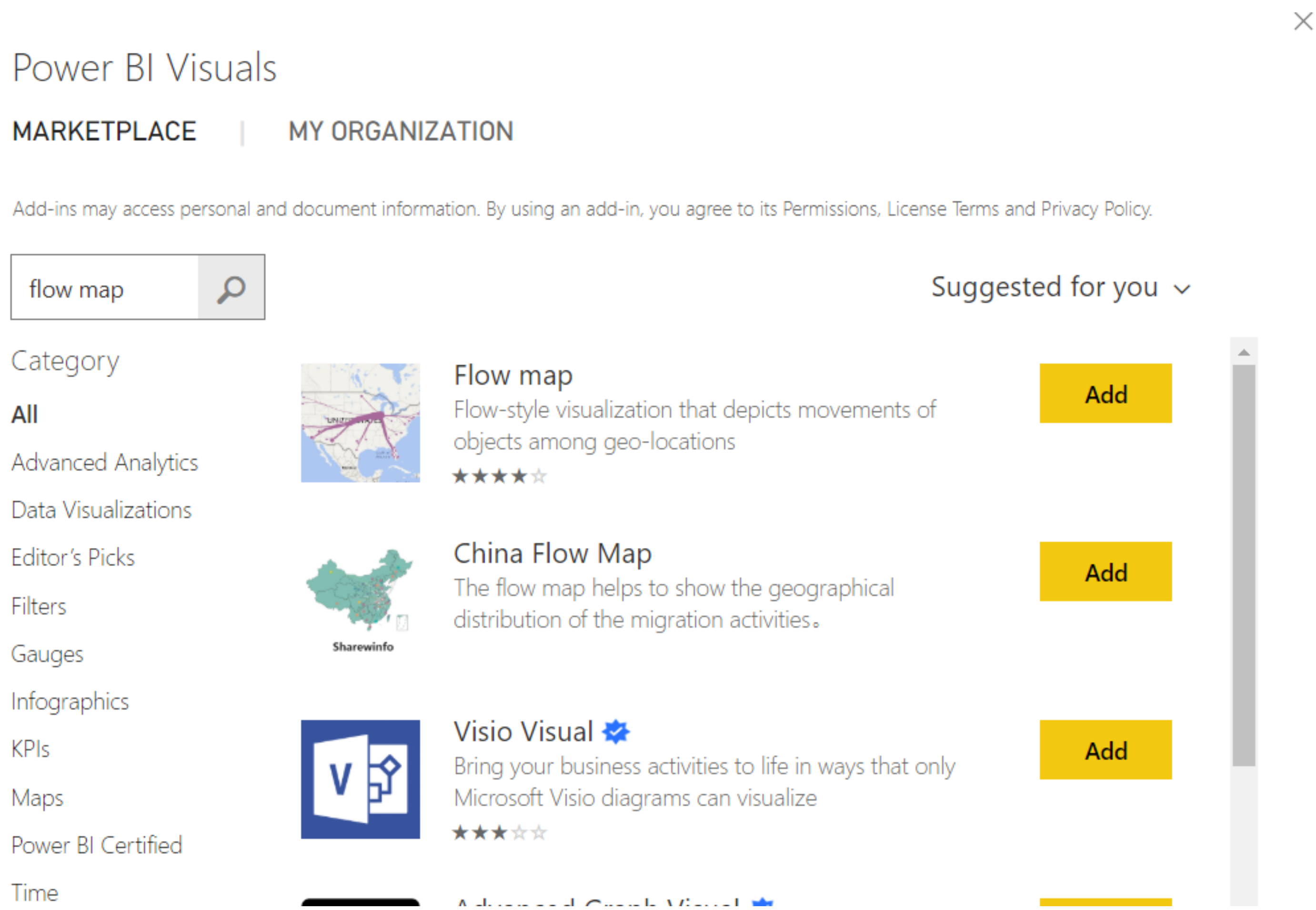
origin_state	FlightCount
California	**2866535**
Los Angeles, CA	843904
San Francisco, CA	661159
San Diego, CA	308276
Oakland, CA	184585
San Jose, CA	170830
Sacramento, CA	165325
Santa Ana, CA	160711
Burbank, CA	87354
Ontario, CA	78871
Long Beach, CA	47229
Palm Springs, CA	39566
Fresno, CA	34055
Santa Barbara, CA	25732
San Luis Obispo, CA	14609
Monterey, CA	12855
Bakersfield, CA	10775
Arcata/Eureka, CA	7019
Redding, CA	3852
Total	**22466964**

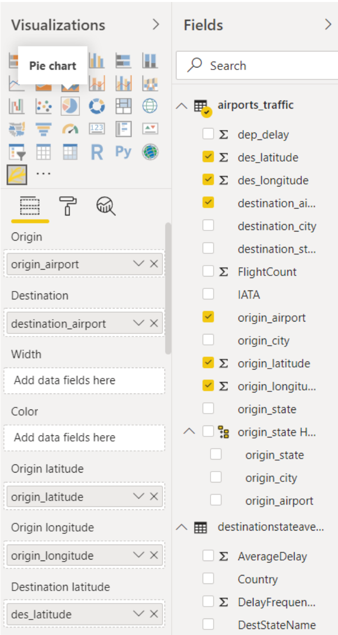
b. Tips: you can expand the next level or only the next level on selected item, click on the **FlightCount** column to sort by the highest number of flight.

- Add a new visualization from the store :



- Select the … and select Import from store.
- When the Power BI Custom Visuals Store open, select the **Maps** category and choose the **Flow map** and Add.





- Add on the **map flow properties** and **place fields** as the snapshot

a. Drag & drop the field **origin_airport** to the Map flow's **Origin** property

b. Drag & drop the field **destination_airport** to the Map flow's **Destination** property

c. Drag & drop the field **flightCount** to the Map flow's **Value** property

d. Drag & drop the field average of **origin_latitude** to **Origin latitude**

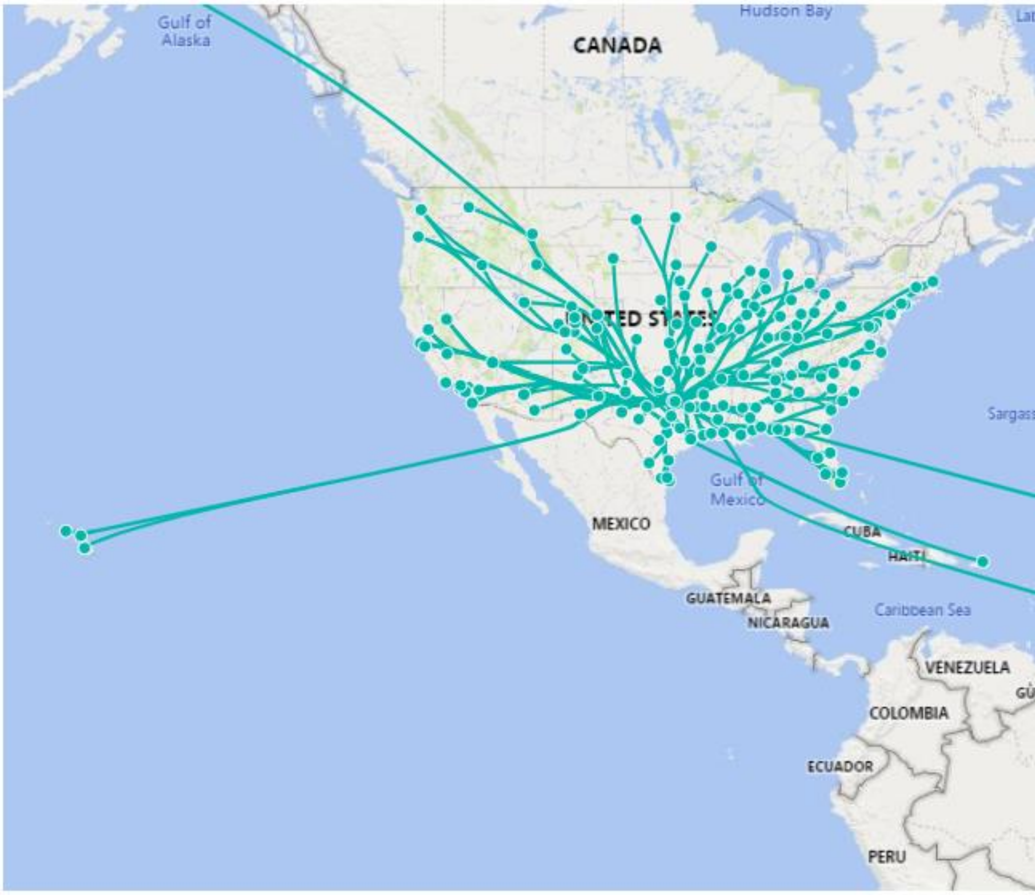
e. Drag & drop the field average of **origin_longitude** to **Origin longitude**

f. Drag & drop the field average of **dest_latitude** to **Destination latitude**

g. Drag & drop the field average of **dest_longitude** to **Destination longitude**

- Select an **origin_city** in the **matrice**. You should have something similar to this:

origin_state	FlightCount
California	2866535
Texas	2495260
Florida	1741372
Georgia	1527215
Illinois	1484963
New York	1009830
Colorado	954763
Arizona	699453
North Carolina	641966
Nevada	634033
Virginia	594539
Michigan	580694
Washington	544861
Minnesota	510395
Massachusetts	471634
New Jersey	465356
Utah	438512
Pennsylvania	429721
Missouri	408542
Hawaii	392036
Maryland	373662
Tennessee	323110
Ohio	297825
Louisiana	269717
Oregon	262624
Wisconsin	210167
Indiana	159413
Kentucky	145258
Alaska	141217
Oklahoma	132806
South Carolina	121319
Puerto Rico	111393
Alabama	105262
Total	22466964



Disclaimer: Once you have completed the lab, to reduce costs associated with your Azure subscription, you may want to delete your clusters!!!!

Terms of use

By using this hands-on lab, you agree to the following terms:

The technology/functionality described in this hands-on lab is provided by agileDSS for purposes of obtaining your feedback and to provide you with a learning experience. You may only use the hands-on lab to evaluate such technology features and functionality and provide feedback to Microsoft. You may not use it for any other purpose. You may not modify, copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell this hands-on lab or any portion thereof.

COPYING OR REPRODUCTION OF THE HANDS-ON LAB (OR ANY PORTION OF IT) TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THIS HANDS-ON LAB PROVIDES CERTAIN SOFTWARE TECHNOLOGY/PRODUCT FEATURES AND FUNCTIONALITY, INCLUDING POTENTIAL NEW FEATURES AND CONCEPTS, IN A SIMULATED ENVIRONMENT WITHOUT COMPLEX SET-UP OR INSTALLATION FOR THE PURPOSE DESCRIBED ABOVE. THE TECHNOLOGY/CONCEPTS REPRESENTED IN THIS HANDS-ON LAB MAY NOT REPRESENT FULL FEATURE FUNCTIONALITY AND MAY NOT WORK THE WAY A FINAL VERSION MAY WORK. WE ALSO MAY NOT RELEASE A FINAL VERSION OF SUCH FEATURES OR CONCEPTS. YOUR EXPERIENCE WITH USING SUCH FEATURES AND FUNCTIONALITY IN A PHYSICAL ENVIRONMENT MAY ALSO BE DIFFERENT.