

Getting started with Apache Spark on Azure Databricks

Apache Spark

Apache Spark™ is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics. In this tutorial, you will get familiar with the Spark UI, learn how to create Spark jobs, load data and work with Datasets, get familiar with Spark's DataFrames API, run machine learning algorithms, and understand the basic concepts behind Spark Streaming. This Spark environment you will use is Azure Databricks. Instead of worrying about spinning up and winding down clusters, maintaining clusters, maintaining code history, or Spark versions, Azure Databricks will take care of that for you, so you can start writing Spark queries instantly and focus on your data problems.

Microsoft Azure Databricks is built by the creators of Apache Spark and is the leading Spark-based analytics platform. It provides data science and data engineering teams with a fast, easy and collaborative Spark-based platform on Azure. It gives Azure users a single platform for Big Data processing and Machine Learning.

Azure Databricks is a “first party” Microsoft service, the result of a unique collaboration between the Microsoft and Databricks teams to provide Databricks' Apache Spark-based analytics service as an integral part of the Microsoft Azure platform. It is natively integrated with Microsoft Azure in a number of ways ranging from a single click start to a unified billing. Azure Databricks leverages Azure's security and seamlessly integrates with Azure services such as Azure Active Directory, SQL Data Warehouse, and Power BI. It also provides fine-grained user permissions, enabling secure access to Databricks notebooks, clusters, jobs and data.

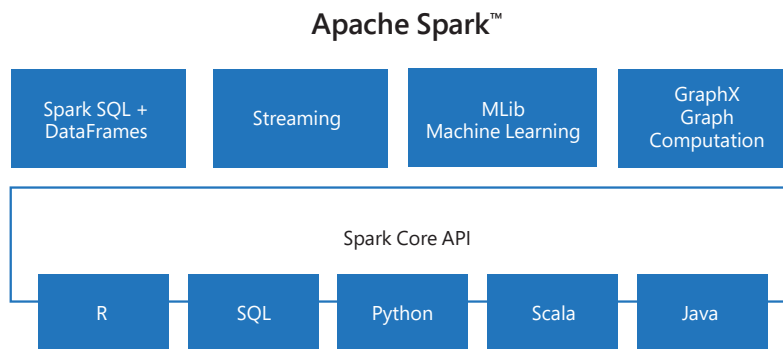
Azure Databricks brings teams together in an interactive workspace. From data gathering to model creation, Databricks notebooks are used to unify the process and instantly deploy to production. You can launch your new Spark environment with a single click, and integrate effortlessly with a wide variety of data stores and services such as Azure SQL Data Warehouse, Azure Cosmos DB, Azure Data Lake Store, Azure Blob storage, and Azure Event Hub.

Table of contents

Getting started with Spark	4	DataFrames	25
Setting up Azure Databricks	7	Machine learning.....	29
A quick start	11	Streaming.....	35
Datasets.....	16		

Getting started with Spark

Getting started with Spark



Spark SQL + DataFrames

Structured Data: Spark SQL

Many data scientists, analysts, and general business intelligence users rely on interactive SQL queries for exploring data. Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

Streaming

Streaming Analytics: Spark Streaming

Many applications need the ability to process and analyze not only batch data, but also streams of new data in real-time. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter.

MLlibMachine Learning

Machine Learning: MLlib

Machine learning has quickly emerged as a critical piece in mining Big Data for actionable insights. Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (up to 100x faster than MapReduce). The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

GraphXGraph Computation

Graph Computation: GraphX

GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

Spark Core API

General Execution: Spark Core

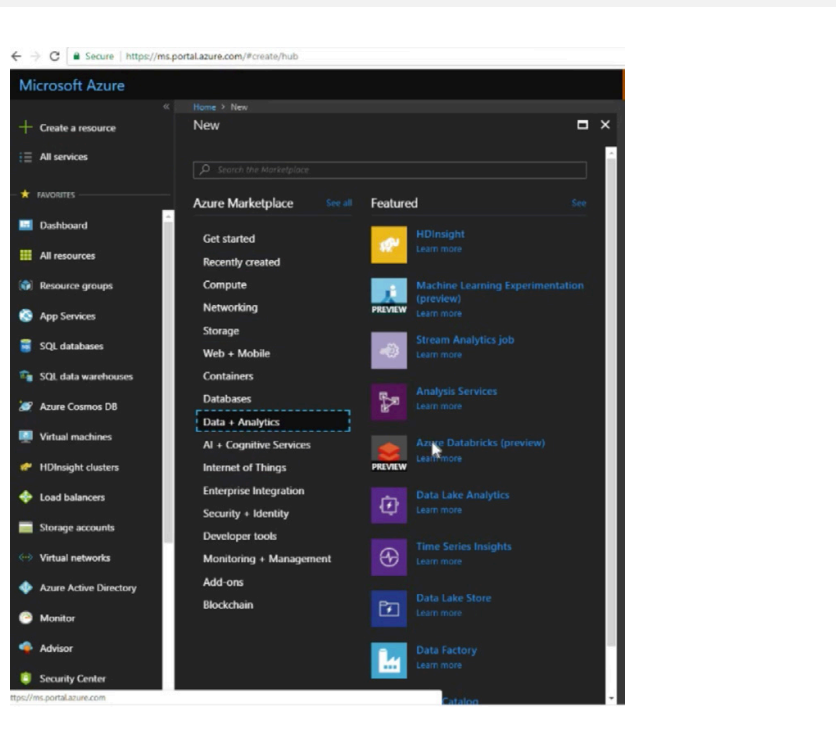
Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides in-memory computing capabilities to deliver speed, a generalized execution model to support a wide variety of applications, and Java, Scala, and Python APIs for ease of development.

Setting up Azure Databricks

Setting up Azure Databricks

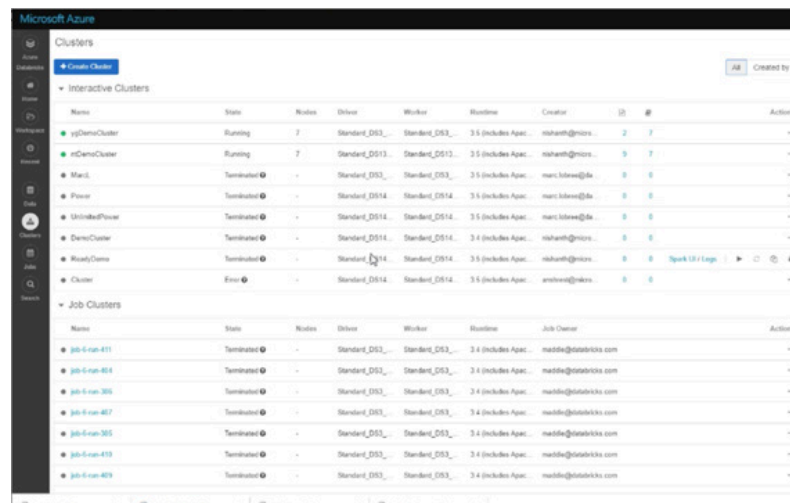
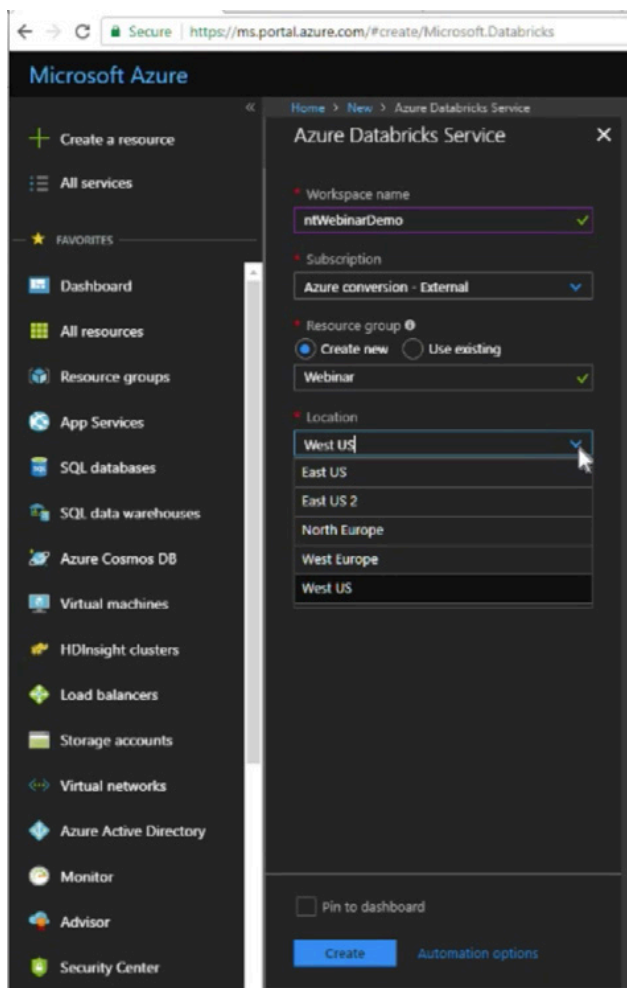
To get started, set up your Azure Databricks account [here](#).

If you do not already have an Azure account, you can get a trial account to get started. Once you have entered the Azure Portal, you can select Azure Databricks under the Data + Analytics section.



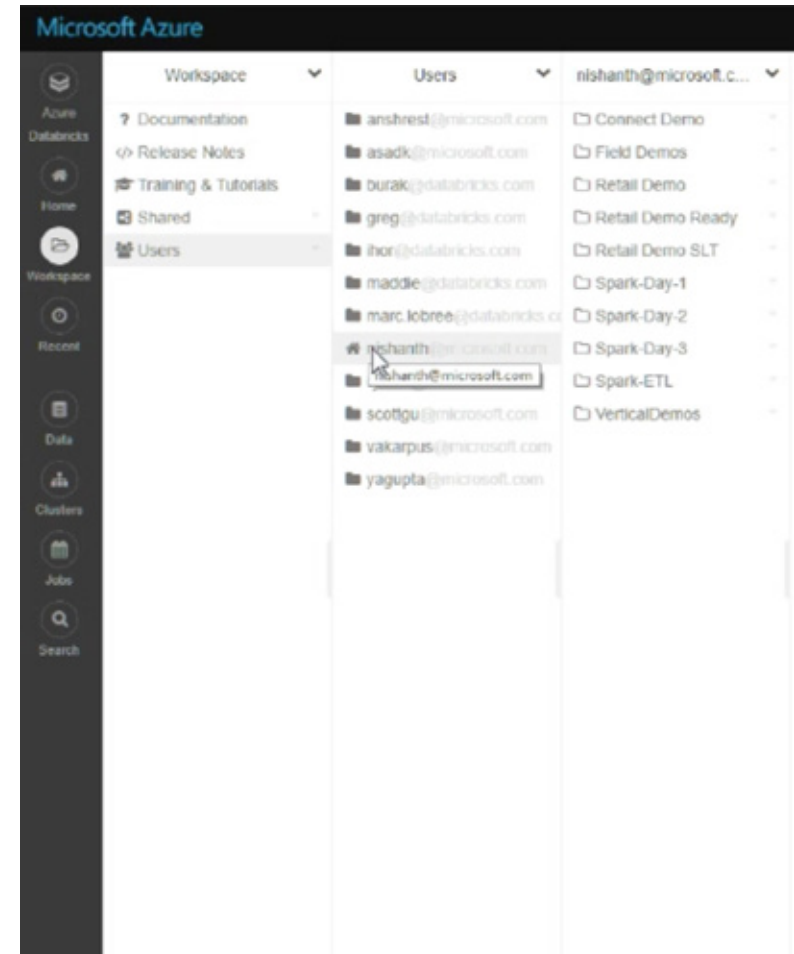
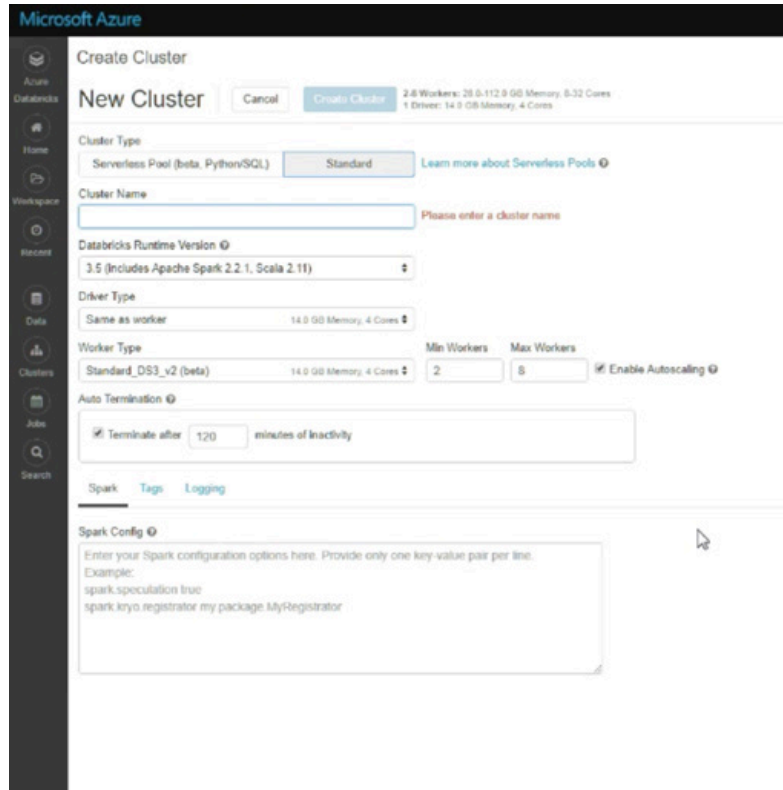
You can easily set up your workspace within the Azure Databricks service.

Once you are in the Azure Databricks Workspace, you can Create a Cluster.



And then configure that cluster. Using Databricks Serverless and choosing Autoscaling, you will not have to spin up and manage clusters – Databricks will take care of that for you.

Once you are up and running you will be able to import Notebooks.



A quick start

Overview

To access all the code examples in this stage, please import the [Quick Start using Python](#) or [Quick Start using Scala](#) notebooks.

This module allows you to quickly start using Apache Spark. We will be using Azure Databricks so you can focus on the programming examples instead of spinning up and maintaining clusters and notebook infrastructure. As this is a quick start, we will be discussing the various concepts briefly so you can complete your end-to-end examples. In the “Additional Resources” section and other modules of this guide, you will have an opportunity to go deeper with the topic of your choice.

Writing your first Apache Spark Job

To write your first Apache Spark Job using Azure Databricks, you will write your code in the cells of your Azure Databricks notebook. In this example, we will be using Python. For more information, you can also reference the [Apache Spark Quick Start Guide](#) and the Azure Databricks Documentation. The purpose of this quick start is showcase RDD’s (Resilient Distributed Datasets) operations so that you will be able to understand the Spark UI when debugging or trying to understand the tasks being undertaken.

When running this first command, we are reviewing a folder within the Databricks File System (an optimized version of Azure Blob Storage) which contains your files.

```
# Take a look at the file system
%fs ls /databricks-datasets/samples/docs/
```

path

dbfs:/databricks-datasets/samples/docs/README.md

In the next command, you will use the Spark Context to read the README.md text file.

```
# Setup the textFile RDD to read the README.md file
# Note this is lazy
textFile = sc.textFile("/databricks-datasets/samples/docs/README.md")
```

And then you can count the lines of this text file by running the command.

```
# Perform a count against the README.md file
textFile.count()

> # When performing an action (like a count) this is when the textFile is read
# Click on [View] to see the stages and executors
textFile.count()

Out[34]: 82
```

One thing you may have noticed is that the first command, reading the textFile via the Spark Context (sc), did not generate any output while the second command (performing the count) did. The reason for this is because RDDs have actions (which returns values) as well as transformations (which returns pointers to new RDDs). The first command was a transformation while the second one was an action. This is important because when Spark performs its calculations, it will not execute any of the transformations until an action occurs. This allows Spark to optimize (e.g. run a filter prior to a join) for performance instead of following the commands serially.

Apache Spark DAG

To see what is happening when you run the count() command, you can see the jobs and stages within the Spark Web UI. You can access this directly from the Databricks notebook so you do not need to change your context as you are debugging your Spark job.

As you can see from the below **Jobs** view, when performing the action count() it also includes the previous transformation to access the text file.

Details for Job 227

Status: SUCCEEDED

Job Group: 6315769790877914010_8378925948751892694_289c1bd99b994ab3a5b8d5628182afe7

Completed Stages: 1

- Event Timeline
- DAG Visualization

Stage 502

```

    graph TD
      A[Stage 502] --> B[textFile]
      B --> C[count]
  
```

Completed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total
502	6315769790877914010	# When performing an action (like a count) this... count at <python-input-34-270fe185824b>:3	2015/12/14 00:16:41	1 s	2/2

Details for Stage 502 (Attempt 0)

Total Time Across All Tasks: 2 s

Locality Level Summary: Process local: 2

Input Size / Records: 3.8 KB / 82

DAG Visualization

```

    graph TD
      A[Stage 502] --> B["/mnt/tardis6/docs/README.md [1330] textFile at NativeMethodAccessorImpl.java:-2"]
      B --> C["MapPartitionsRDD [1331] textFile at NativeMethodAccessorImpl.java:-2"]
      C --> D["PythonRDD [1333] count at <python-input-34-270fe185824b>:3"]
  
```

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.6 s	0.6 s	1 s	1 s	1 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Input Size / Records	1967.0 B / 38	1967.0 B / 38	1967.0 B / 44	1967.0 B / 44	1967.0 B / 44

What is happening under the covers becomes more apparent when reviewing the **Stages** view from the Spark UI (also directly accessible within your Databricks notebook). As you can see from the DAG visualization below, prior to the PythonRDD [1333] count() step, Spark will perform the task of accessing the file ([1330] textFile) and running MapPartitionsRDD [1331] textFile.

RDDs, Datasets, and DataFrames

As noted in the previous section, RDDs have actions which return values and transformations which return points to new RDDs. Transformations are lazy and executed when an action is run. Some examples include:

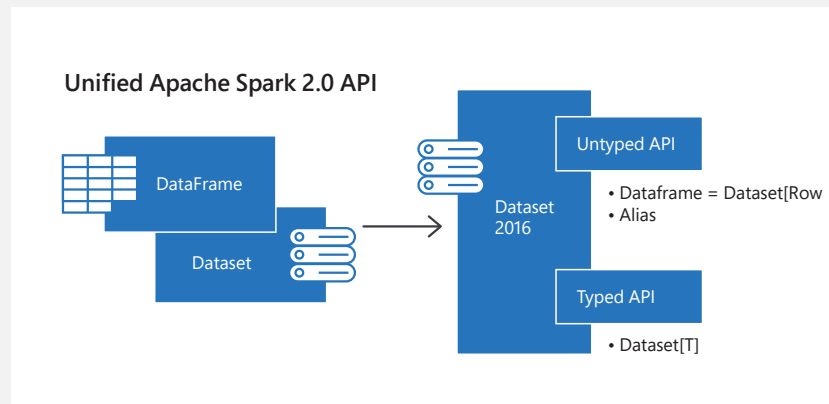
Transformations: `map()`, `flatMap()`, `filter()`, `mapPartitions()`, `mapPartitionsWithIndex()`, `sample()`, `union()`, `distinct()`, `groupByKey()`, `reduceByKey()`, `sortByKey()`, `join()`, `cogroup()`, `pipe()`, `coalesce()`, `repartition()`, `partitionBy()`, ...

Actions: `reduce()`, `collect()`, `count()`, `first()`, `take()`, `takeSample()`, `takeOrdered()`, `saveAsTextFile()`, `saveAsSequenceFile()`, `saveAsObjectFile()`, `countByKey()`, `foreach()`, ...

In many scenarios, especially with the performance optimizations embedded in DataFrames and Datasets, it will not be necessary to work with RDDs. But it is important to bring this up because:

- RDDs are the underlying infrastructure that allows Spark to run so fast (in-memory distribution) and provide data lineage.
- If you are diving into more advanced components of Spark, it may be necessary to utilize RDDs.
- All the DAG visualizations within the Spark UI reference RDDs.

Saying this, when developing Spark applications, you will typically use DataFrames and Datasets. As of Apache Spark 2.0, the DataFrame and Dataset APIs are merged together; a DataFrame is the Dataset Untyped API while what was known as a Dataset is the Dataset Typed API.



Datasets

Overview

To access all the code examples in this stage, please import the [Examining IoT Device Using Datasets](#) notebook.

The Apache Spark Dataset API provides a type-safe, object-oriented programming interface. In other words, in Spark 2.0 `DataFrame` and `Datasets` are unified as explained in the previous section ‘`RDDs, Datasets and DataFrames`,’ and `DataFrame` is an alias for an untyped `Dataset [Row]`. Like `DataFrames`, `Datasets` take advantage of Spark’s Catalyst optimizer by exposing expressions and data fields to a query planner. Beyond Catalyst’s optimizer, `Datasets` also leverage Tungsten’s fast in-memory encoding. They extend these benefits with compile-time type safety—meaning production applications can be checked for errors before they are ran—and they also allow direct operations over user-defined classes, as you will see in a couple of simple examples below. Lastly, the Dataset API offers a high-level domain specific language operations like `sum()`, `avg()`, `join()`, `select()`, `groupBy()`, making the code a lot easier to express, read, and write.

In this section, you will learn two ways to create `Datasets`: dynamically creating a data and reading from JSON file using Spark Session. Additionally, through simple and short examples, you will learn about Dataset API operations on the `Dataset`, issue SQL queries and visualize data. For learning purposes, we use a small IoT Device dataset; however, there is no reason why you can’t use a large dataset.

Creating or Loading Sample Data

There are two easy ways to have your structured data accessible and process it using Dataset APIs within a notebook. First, for primitive types in examples or demos, you can create them within a Scala or Python notebook or in your sample Spark application. For example, here’s a way to create a `Dataset` of 100 integers in a notebook.

Note that in Spark 2.0, the `SparkContext` is subsumed by `SparkSession`, a single point of entry, called `spark`. Going forward, you can use this handle in your driver or notebook cell, as shown below, in which we create 100 integers as `Dataset[Long]`.

```
// range of 100 numbers to create a Dataset.
val range100 = spark.range(100)
range100.collect()
```

```
▶ (1) Spark Jobs
range100: org.apache.spark.sql.Dataset[Long] = [id: bigint]
res0: Array[Long] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99)
Command took 4.01s
```

Second, the more common way is to read a data file from an external data sources, such HDFS, S3, NoSQL, RDBMS, or local filesystem. Spark supports multiple formats : JSON, CSV, Text, Parquet, ORC etc. To read a JSON file, you can simply use the SparkSession handle spark.

```
// read a JSON file from a location mounted on a DBFS mount point
// Note that we are using the new entry point in Spark 2.0 called spark
val jsonData = spark.read.json("/databricks-datasets/data/people/person.json")
```

At the time of reading the JSON file, Spark does not know the structure of your data—how you want to organize your data into a typed-specific JVM object. It attempts to infer the schema from the JSON file and creates a DataFrame = Dataset[Row] of generic Row objects.

Alternatively, to convert your DataFrame into a Dataset reflecting a Scala class object, you define a domain specific Scala case class, followed by explicitly converting into that type, as shown below.

```
// First, define a case class that represents our type-specific Scala JVM
Object
case class Person (email: String, iq: Long, name: String)

// Read the JSON file, convert the DataFrames into a type-specific JVM Scala
object // Person. Note that at this stage Spark, upon reading JSON, created
a generic
// DataFrame = Dataset[Rows]. By explicitly converting DataFrame into
Dataset
// results in a type-specific rows or collection of objects of type Person
val ds = spark.read.json("/databricks-datasets/data/people/person.json").
as[Person]
```

In a second example, we do something similar with IoT devices state information captured in a JSON file: define a case class and read the JSON file from the FileStore, and convert the DataFrame = Dataset[DeviceIoTData].

There are a couple of reasons why you want to convert a DataFrame into a type-specific JVM objects. First, after an explicit conversion, for all relational and query expressions using Dataset API, you get compile-type safety. For example, if you use a filter operation using the wrong data type, Spark will detect mismatch types and issue a compile error rather than an execution runtime error, resulting in catching errors earlier. Second, the Dataset API provides highorder methods making code much easier to read and develop.

In the following section, Processing and Visualizing a Dataset, you will notice how the use of Dataset typed objects make the code much easier to express and read.

As above with Person example, here we create a case class that encapsulates our Scala object.

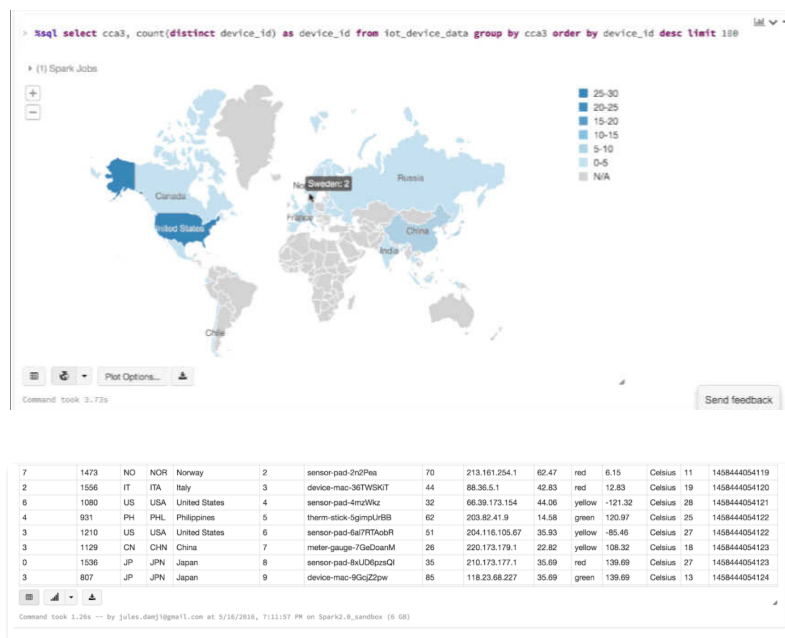
```
// define a case class that represents our Device data.
case class DeviceIoTData (
  battery_level: Long,
  c02_level: Long,
  cca2: String,
  cca3: String,
  cn: String,
  device_id: Long,
  device_name: String,
  humidity: Long,
  ip: String,
  latitude: Double,
  longitude: Double,
  scale: String,
  temp: Long,
  timestamp: Long
)
// fetch the JSON device information uploaded into the Filestore
val jsonFile = "/databricks-datasets/data/iot/iot_devices.json"

// read the json file and create the dataset from the case class DeviceIoTData
// ds is now a collection of JVM Scala objects DeviceIoTData
val ds = spark.read.json(jsonFile).as[DeviceIoTData]
```

Viewing a Dataset

To view this data in a tabular format, instead of exporting this data out to a third party tool, you can use the Databricks display() command. That is, once you have loaded the JSON data and converted into a Dataset for your type-specific collection of JVM objects, you can view them as you would view a DataFrame, by using either display() or using standard Spark commands, such as take(), foreach(), and println() API calls.

```
// display the dataset table just read in from the JSON file
display(ds)
```



```
// Using the standard Spark commands, take() and foreach(), print the first
// 10 rows of the Datasets.
ds.take(10).foreach(println(_))
```

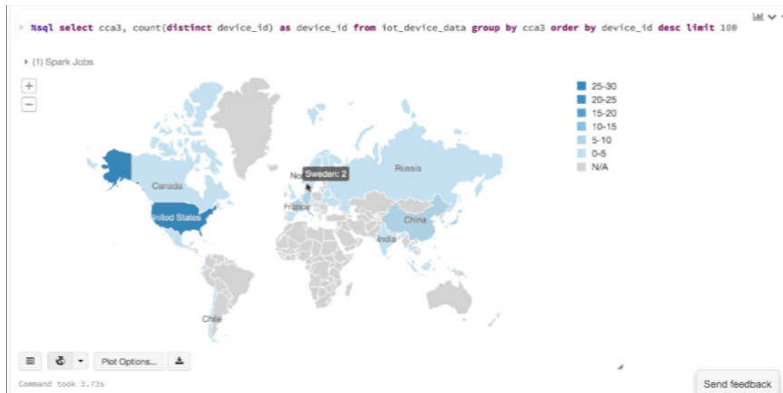
▶ (1) Spark Jobs

```
DeviceIoTData(8,868,US,USA,United States,1,meter-gauge-1xbYRYcj,51,68.161.225.1,38.
DeviceIoTData(7,1473,NO,NOR,Norway,2,sensor-pad-2n2Pea,70,213.161.254.1,62.47,red,8
DeviceIoTData(2,1556,IT,ITA,Italy,3,device-mac-36TWSKiT,44,88.36.5.1,42.83,red,12.8
DeviceIoTData(6,1080,US,USA,United States,4,sensor-pad-4mzWkz,32,66.39.173.154,44.0
DeviceIoTData(4,931,PH,PHL,Philippines,5,therm-stick-5gimpUrBB,62,203.82.41.9,14.50
DeviceIoTData(3,1210,US,USA,United States,6,sensor-pad-6aL7RTAobR,51,204.116.105.6)
DeviceIoTData(3,1129,CN,CHN,China,7,meter-gauge-7GeDoanM,26,220.173.179.1,22.82,yel
DeviceIoTData(0,1536,JP,JPN,Japan,8,sensor-pad-8xUD6pzsQI,35,210.173.177.1,35.69,ro
DeviceIoTData(3,807,JP,JPN,Japan,9,device-mac-9GcjZ2pw,85,118.23.68.227,35.69,green
DeviceIoTData(7,1470,US,USA,United States,10,sensor-pad-10BsywSYUF,56,208.109.163.0
Command took 1.18s -- by jules.damji@gmail.com at 5/16/2016, 7:12:05 PM on Spark2.0_sandbox (6 GB)
```

Processing and Visualizing a Dataset

An additional benefit of using the Azure Databricks `display()` command is that you can quickly view this data with a number of embedded visualizations. For example, in a new cell, you can issue SQL queries and click on the map to see the data. But first, you must save your dataset, `ds`, as a temporary table.

```
// registering your Dataset as a temporary table to which you can issue SQL
queries
ds.createOrReplaceTempView("iot_device_data")
```



```
// filter out all devices whose temperature exceed 25 degrees and generate
// another Dataset with three fields that of interest and then display
// the mapped Dataset
val dsTemp = ds.filter(d => d.temp > 25).map(d => (d.temp, d.device_name,
d.cca3))
display(dsTemp)
```

Like RDD, Dataset has transformations and actions methods. Most importantly are the high-level domain specific operations such as `sum()`, `select()`, `avg()`, `join()`, and `union()` that are absent in RDDs. For more information, look at the [Scala Dataset API](#).

Let's look at a few handy ones in action. In the example below, we use `filter()`, `map()`, `groupBy()`, and `avg()`, all higher-level methods, to create another Dataset, with only fields that we wish to view. What's noteworthy is that we access the attributes we want to filter by their names as defined in the case class. That is, we use the dot notation to access individual fields. As such, it makes code easy to read and write.

(1) Spark Jobs

_1	_2	_3	_4
34	meter-gauge-1xbYRYcj	1	USA
28	sensor-pad-4mzWkz	4	USA
27	sensor-pad-6a7RTAcobR	6	USA
27	sensor-pad-8xUDpazQI	8	JPN
26	sensor-pad-10BsywSYUF	10	USA
31	meter-gauge-17zb8FghhI	17	USA
31	sensor-pad-18XULNBKv	18	CHN
29	meter-gauge-19eg1Bp1CO	19	USA
30	device-mac-21qz2h	21	AUT

```
// Apply higher-level Dataset API methods such as groupBy() and avg().
// Filter temperatures > 25, along with their corresponding
// devices' humidity, compute averages, groupBy cca3 country codes,
// and display the results, using table and bar charts
val dsAvgTmp = ds.filter(d => {d.temp > 25}).map(d => (d.temp, d.humidity,
d.cca3)).groupBy($"_3").avg()
```

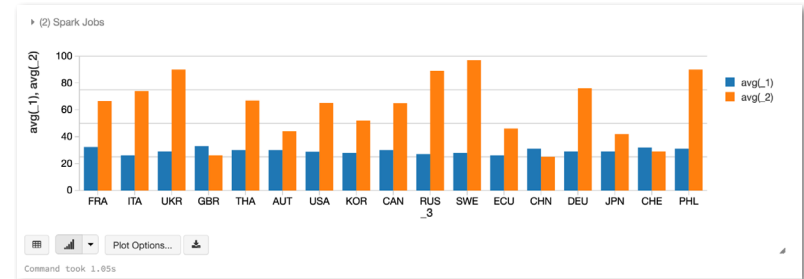
```
// display averages as a table, grouped by the country
display(dsAvgTmp)
```

(2) Spark Jobs

_3	avg(_1)	avg(_2)
FRA	32.33333333333333	66.66666666666667
ITA	26	74
UKR	29	90
GBR	33	26
THA	30	67
AUT	30	44
USA	28.76923076923077	65.23076923076923
KOR	28	52
CAN	30	65

Command took 1.65s

```
// display the averages as bar graphs, grouped by the country
display(dsAvgTmp)
```



```
// Select individual fields using the Dataset method select()
// where battery_level is greater than 6. Note this high-level
// domain specific language API reads like a SQL query
display(ds.select($"battery_level", $"c02_level", $"device_name")
where($"battery_level" > 6).sort($"c02_level"))
```

(1) Spark Jobs

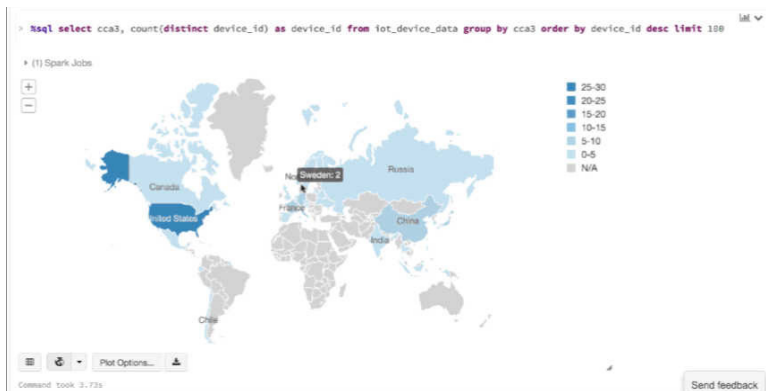
battery_level	c02_level	device_name
8	857	sensor-pad-46MiQ33UDaaa
8	868	meter-gauge-1xbYRYcj
8	934	meter-gauge-712JgErD0zVw
7	940	sensor-pad-34F1Jubre3B
9	986	sensor-pad-48jt4eL
7	997	therm-stick-55kEAHLqWn0
7	1131	therm-stick-35Lg804z
7	1155	sensor-pad-20gFNFBgqr
7	1160	meter-gauge-61NehO8Msi

Command took 0.27s

Below is an example showing how quickly you can go from table to map to charts using Datasets and Azure Databricks `display()` command.

Having saved the Dataset of DeviceIoTData as a temporary table, you can issue SQL queries to it.

```
%sql select cca3, count (distinct device_id) as device_id from  
iot_device_data group  
by cca3 order by device_id desc limit 100
```



DataFrames

Overview

To access all the code examples in this stage, please import the [Population vs. Price DataFrames notebook](#).

Apache Spark DataFrames were created to run Spark programs faster from both a developer and an execution perspective. With less code to write and less data to read, the Catalyst optimizer solves common problems efficiently and faster using DataFrame functions (e.g. select columns, filtering, joining different data sources, aggregation, etc.). DataFrames also allow you to seamlessly intermix operations with custom SQL, Python, Java, R, or Scala code.

Accessing the sample data

The easiest way to work with DataFrames is to access an example dataset. We have made a number of datasets available in the `/databricks-datasets` folder which is accessible within the Databricks platform. For example, to access the file that compares city population vs. median sale prices of homes, you can access the file `/databricks-datasets/samples/population-vs-price/data_geo.csv`.

We will use the `spark-csv` package from [Spark Packages](#) (a community index of packages for Apache Spark) to quickly import the data, specify that a header exists, and infer the schema.

Note, the `spark-csv` package is embedded into Spark 2.0.

```
# Use the Spark CSV datasource with options specifying:
# - First line of file is a header
# - Automatically infer the schema of the data
data = sqlContext.read.format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load("/databricks-datasets/samples/population-vs-price/data_geo.csv")

data.cache() # Cache data for faster reuse
data = data.dropna() # drop rows with missing values
```

```
# Register table so it is accessible via SQL Context
# For Apache Spark = 2.0
data.createOrReplaceTempView("data_geo")
```

Viewing the DataFrame

Now that you have created the data DataFrame, you can quickly access the data using standard Spark commands such as `take()`. For example, you can use the command `data.take(10)` to view the first ten rows of the data DataFrame.

```
> data.take(10)
```

```
↳ (1) Spark Jobs
```

```
Out[3]:
```

```
[Row(2014 rank=101, City=u'Birmingham', State=u'Alabama', State Code=u'AL', 2014 Population est
Row(2014 rank=125, City=u'Huntsville', State=u'Alabama', State Code=u'AL', 2014 Population est
Row(2014 rank=122, City=u'Mobile', State=u'Alabama', State Code=u'AL', 2014 Population est
Row(2014 rank=114, City=u'Montgomery', State=u'Alabama', State Code=u'AL', 2014 Population est
Row(2014 rank=64, City=u'Anchorage[19]', State=u'Alaska', State Code=u'AK', 2014 Population est
Row(2014 rank=78, City=u'Chandler', State=u'Arizona', State Code=u'AZ', 2014 Population est
Row(2014 rank=86, City=u'Gilbert[20]', State=u'Arizona', State Code=u'AZ', 2014 Population est
Row(2014 rank=88, City=u'Glendale', State=u'Arizona', State Code=u'AZ', 2014 Population est
Row(2014 rank=38, City=u'Mesa', State=u'Arizona', State Code=u'AZ', 2014 Population est
Row(2014 rank=148, City=u'Peoria', State=u'Arizona', State Code=u'AZ', 2014 Population est
```

```
Command took 0.12s
```

To view this data in a tabular format, instead of exporting this data out to a third party tool, you can use the `display()` command within Azure Databricks.

```
> display(data)
```

↳ (2) Spark Jobs

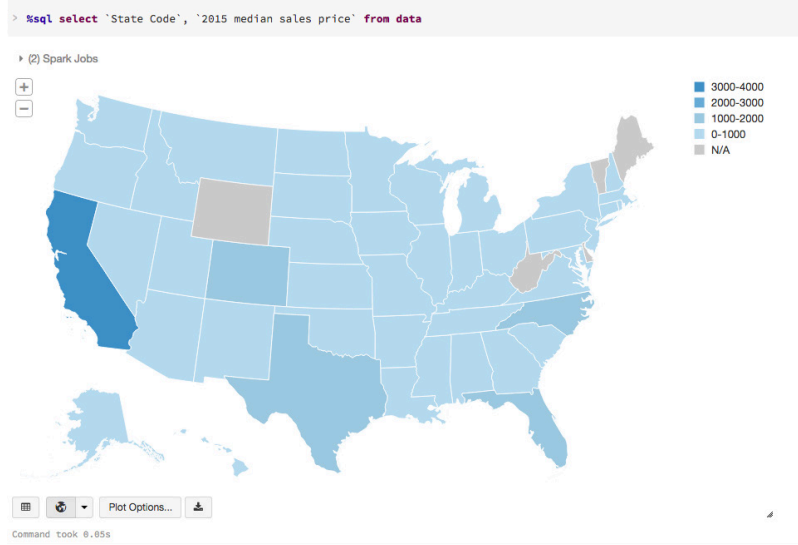
2014 rank	City	State	State Code	2014 Population estimate
101	Birmingham	Alabama	AL	212247
125	Huntsville	Alabama	AL	188226
122	Mobile	Alabama	AL	194675
114	Montgomery	Alabama	AL	200481
64	Anchorage[19]	Alaska	AK	301010
78	Chandler	Arizona	AZ	254276
86	Gilbert[20]	Arizona	AZ	239277
88	Glendale	Arizona	AZ	237517
38	Mesa	Arizona	AZ	464704
148	Peoria	Arizona	AZ	115470

Command took 1.83s

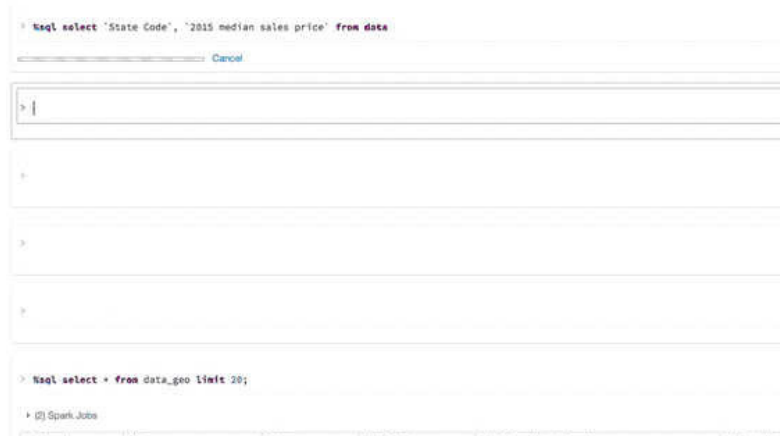
Visualizing your DataFrame

An additional benefit of using the Azure Databricks `display()` command is that you can quickly view this data with a number of embedded visualizations. For example, in a new cell, you can specify the following SQL query and click on the map.

```
%sql select 'State Code', '2015 median sales price' from data
```



Below is an example showing how quickly you can go from table to map using DataFrames and the Azure Databricks `display()` command.



Machine learning

Overview

To access all the code examples in this stage, please import the [Population vs. Price Linear Regression](#) notebook. As organizations create more diverse and more user-focused data products and services, there is a growing need for machine learning, which can be used to develop personalizations, recommendations, and predictive insights. Apache Spark's Machine Learning Library (MLlib) allows data scientists to focus on their data problems and models instead of solving the complexities surrounding distributed data (such as infrastructure, configurations, and so on).

Accessing the sample data

The easiest way to work with DataFrames is to access an example dataset. We have made a number of datasets available in the `/databricks-datasets` folder which is accessible from Azure Databricks. For example, to access the file that compares city population vs. median sale prices of homes, you can access the file `/databricks-datasets/samples/population-vs-price/data_geo.csv`.

We will use the `spark-csv` package from [Spark Packages](#) (a community index of packages for Apache Spark) to quickly import the data, specify that a header exists, and infer the schema.

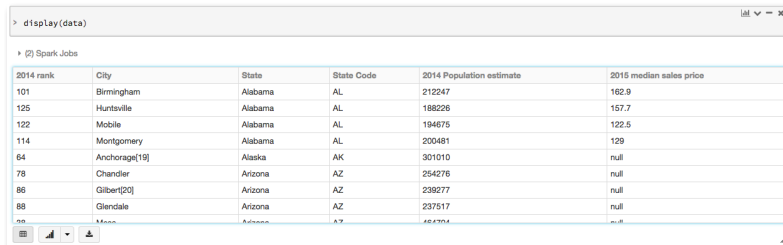
Note, the `spark-csv` package is embedded into Spark 2.0.

```
# Use the Spark CSV datasource with options specifying:
# - First line of file is a header
# - Automatically infer the schema of the data
data = sqlContext.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/databricks-datasets/samples/population-vs-price/data_geo.csv")

data.cache() # Cache data for faster reuse
data = data.dropna() # drop rows with missing values

# Register table so it is accessible via SQL Context
# For Apache Spark = 2.0
data.createOrReplaceTempView("data_geo")
```

To view this data in a tabular format, instead of exporting this data out to a third party tool, you can use the `display()` command within [Databricks](#).



```
> display(data)
```

2014 rank	City	State	State Code	2014 Population estimate	2015 median sales price
101	Birmingham	Alabama	AL	212247	162.9
125	Huntsville	Alabama	AL	168226	157.7
122	Mobile	Alabama	AL	194675	122.5
114	Montgomery	Alabama	AL	200481	129
64	Anchorage[19]	Alaska	AK	301010	null
78	Chandler	Arizona	AZ	254276	null
86	Gilbert[20]	Arizona	AZ	239277	null
88	Glendale	Arizona	AZ	237517	null
no.					

Prepare and visualize data for ML algorithms

In supervised learning—such as a regression algorithm—you typically will define a label and a set of features. In our linear regression example, the label is the 2015 median sales price while the feature is the 2014 Population Estimate. That is, we are trying to use the feature (population) to predict the label (sales price). To simplify the creation of features within Python Spark MLlib, we use `LabeledPoint` to convert the feature (population) to a Vector type.

```
# convenience for specifying schema
from pyspark.mllib.regression import LabeledPoint

data = data.select("2014 Population estimate", "2015 median sales price")
    .map(lambda r: LabeledPoint(r[1], [r[0]]))
    .toDF()
display(data)
```

features	label
▶ {"type":1,"size":1,"indices":[],"values":[212247]}	162.9
▶ {"type":1,"size":1,"indices":[],"values":[188226]}	157.7
▶ {"type":1,"size":1,"indices":[],"values":[194675]}	122.5
▶ {"type":1,"size":1,"indices":[],"values":[200481]}	129
▶ {"type":1,"size":1,"indices":[],"values":[1537058]}	206.1
▶ {"type":1,"size":1,"indices":[],"values":[527972]}	178.1
▶ {"type":1,"size":1,"indices":[],"values":[62722]}	121.2

Executing Linear Regression Model

In this section, we will execute two different linear regression models using different regularization parameters and determine its efficacy. That is, how well do either of these two models predict the sales price (label) based on the population (feature).

Building the model

```
# Import LinearRegression class
from pyspark.ml.regression import LinearRegression

# Define LinearRegression algorithm
lr = LinearRegression()

# Fit 2 models, using different regularization parameters
modelA = lr.fit(data, {lr.regParam:0.0})
modelB = lr.fit(data, {lr.regParam:100.0})
```

Using the model, we can also make predictions by using the transform() function which adds a new column of predictions. For example, the code below takes the first model (modelA) and shows you both the label (original sales price) and prediction (predicted sales price) based on the features (population).


```
# Make predictions
predictionsA = modelA.transform(data)
display(predictionsA)
```

features	label	prediction
> {"type":1,"size":1,"indices":[],"values":[212247]}	162.9	199.31676595846622
> {"type":1,"size":1,"indices":[],"values":[188226]}	157.7	198.40882267887176
> {"type":1,"size":1,"indices":[],"values":[194675]}	122.5	198.65258131548575
> {"type":1,"size":1,"indices":[],"values":[200481]}	129	198.8720359044423
> {"type":1,"size":1,"indices":[],"values":[1537058]}	206.1	249.39183544694856
> {"type":1,"size":1,"indices":[],"values":[527972]}	178.1	211.2505069330287
> {"type":1,"size":1,"indices":[],"values":[197706]}	131.8	198.76714674075743
> {"type":1,"size":1,"indices":[],"values":[346997]}	685.7	204.41003255541705
> {"type":1,"size":1,"indices":[],"values":[200964]}	124.7	220.707071856408

Evaluating the Model

To evaluate the regression analysis, we will calculate the root mean square error using the RegressionEvaluator. Below is the pySpark code for evaluating the two models and their output.

```
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(metricName="rmse")
RMSE = evaluator.evaluate(predictionsA)
print("ModelA: Root Mean Squared Error = " + str(RMSE))

# ModelA: Root Mean Squared Error = 128.602026843
predictionsB = modelB.transform(data)
RMSE = evaluator.evaluate(predictionsB)
print("ModelB: Root Mean Squared Error = " + str(RMSE))

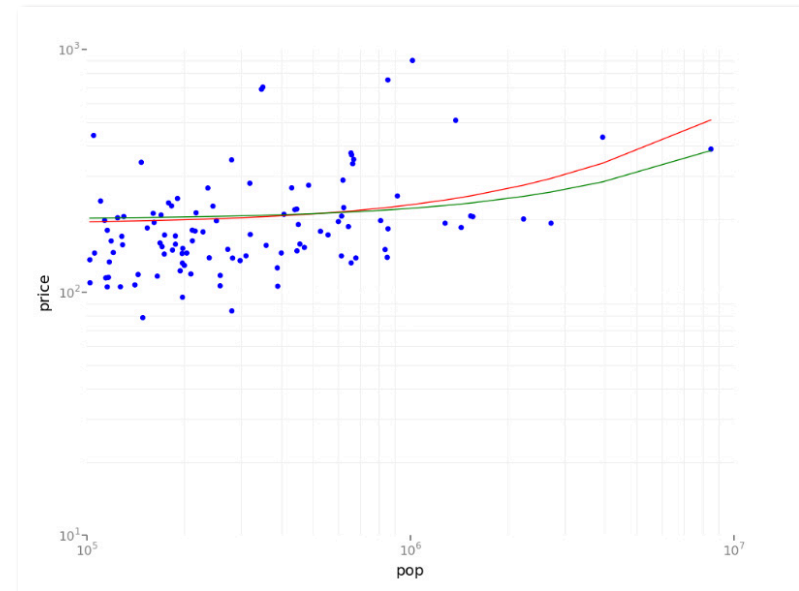
# ModelB: Root Mean Squared Error = 129.496300193
```

As is typical for many machine learning algorithms, you will want to visualize the scatterplot. Azure Databricks supports Python pandas and ggplot, the code below creates a linear regression plot using Python Pandas DataFrame (pydf) and ggplot to display the scatterplot and the two regression models.

```
# Import numpy, pandas, and ggplot
import numpy as np
from pandas import *
from ggplot import *

# Create Python DataFrame
pop = data.map(lambda p: (p.features[0])).collect()
price = data.map(lambda p: (p.label)).collect()
predA = predictionsA.select("prediction").map(lambda r: r[0]).collect()
predB = predictionsB.select("prediction").map(lambda r: r[0]).collect()

pydf = DataFrame({'pop':pop,'price':price,'predA':predA, 'predB':predB})
```



Visualizing the Model

```
# Create scatter plot and two regression models (scaling exponential) using
ggplot
p = ggplot(pydf, aes('pop','price')) +
  geom_point(color='blue') +
  geom_line(pydf, aes('pop','predA'), color='red') +
  geom_line(pydf, aes('pop','predB'), color='green') +
  scale_x_log10() + scale_y_log10()
display(p)
```

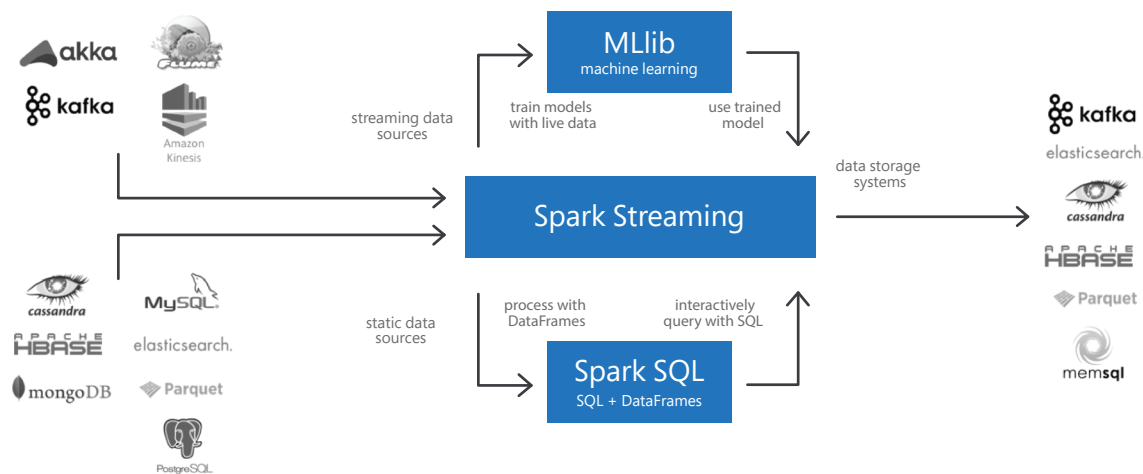
Streaming

Overview

To access all the code examples in this stage, please import the Streaming Wordcount notebook. To help introduce Apache Spark Streaming, we will be going through the Streaming Wordcount example – the “Hello World” example of Spark Streaming which counts words on 1-second batches of streaming data. It uses an in-memory string generator as a dummy source for streaming data. Please refer to the Streaming Wordcount notebook to execute this streaming job as this guide will focus on the primary coding components.

Apache Spark Streaming Concepts

Apache Spark Streaming is a scalable fault-tolerant streaming processing system. As part of Apache Spark™, it integrates with MLlib, SQL, DataFrames, and GraphX. As for Spark 2.0, we will also release Structured Streaming so you can work with Streaming DataFrames.



Sensors, IoT devices, social networks, and online transactions are all generating data that needs to be monitored constantly and acted upon quickly. As a result, the need for large-scale, real-time stream processing is more evident than ever before. There are four broad ways Spark Streaming is being used today:

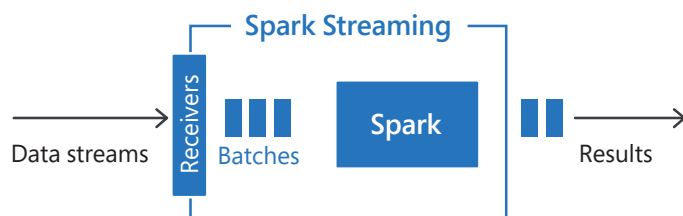
- Streaming ETL — Data is continuously cleaned and aggregated before being pushed into data stores.
- Triggers — Anomalous behavior is detected in real-time and further downstream actions are triggered accordingly. E.g. unusual behavior of sensor devices generating actions.
- Data enrichment — Live data is enriched with more information by joining it with a static dataset allowing for a more complete real-time analysis.
- Complex sessions and continuous learning — Events related to a live session (e.g. user activity after logging into a website or application) are grouped together and analyzed. In some cases, the session information is used to continuously update machine learning models.

In general, Spark Streaming works by having a set of receivers that receive data streams and chop them up into little batches. Spark then processes these batches and pushes out the results.

StreamingContext

Define the function that sets up the StreamingContext

As noted in the previous section, Spark Streaming requires two components: a receiver and a function that creates and sets up the streaming computation. For this Streaming Word Count example in this guide, we will focus on the function as this is the primary logic. Please reference the Streaming Word Count notebook to review the custom receiver as the dummy source.



```
// This is the dummy source implemented as a custom receiver. No need to understand
this.
import scala.util.Random
import org.apache.spark.streaming.receiver._

class DummySource(ratePerSec: Int) extends Receiver[String](StorageLevel.MEMORY_AND_
DISK_2) {
  ...
}

//
// This is the function that creates and sets up the streaming computation
//
var newContextCreated = false // Flag to detect whether new context was created
or not

// Function to create a new StreamingContext and set it up
def creatingFunc(): StreamingContext = {

  // Create a StreamingContext
  val ssc = new StreamingContext(sc, Seconds(batchIntervalSeconds))

  // Create a stream that generates 1000 lines per second
  val stream = ssc.receiverStream(new DummySource(eventsPerSecond))

  // Split the lines into words, and then do word count
  val wordStream = stream.flatMap { _.split(" ") }
  val wordCountStream = wordStream.map(word => (word, 1)).reduceByKey(_ + _)

  // Create temp table at every batch interval
  // For Apache Spark = 2.0
  // rdd.toDF("word", "count").createOrReplaceTempView("batch_word_count")
  wordCountStream.foreachRDD { rdd =>
    rdd.toDF("word", "count").createOrReplaceTempView("batch_word_count")
  }

  stream.foreachRDD { rdd =>
    System.out.println("# events = " + rdd.count())
    System.out.println("t " + rdd.take(10).mkString(", ") + " ...")
  }

  ssc.remember(Minutes(1)) // To make sure data is not deleted by the time we
query it interactively

  println("Creating function called to create new StreamingContext")
  newContextCreated = true
  ssc
}
```

Start Streaming Job: Stop existing StreamingContext if any and start/restart the new one

Here we are going to use the configurations at the top of the notebook to decide whether to stop any existing StreamingContext, and start a new one, or recover one from existing checkpoints.

```
// Stop any existing StreamingContext
if (stopActiveContext) {
  StreamingContext.getActive.foreach { _.stop(stopSparkContext = false) }
}

// Get or create a streaming context
val ssc = StreamingContext.getActiveOrCreate(creatingFunc)
if (newContextCreated) {
  println("New context created from currently defined creating function")
} else {
  println("Existing context running or recovered from checkpoint, may not be
  running currently defined creating function")
}

// Start the streaming context in the background.
ssc.start()

// This is to ensure that we wait for some time before the background
streaming job starts. This will put this cell on hold for 5 times the
batchIntervalSeconds.
ssc.awaitTerminationOrTimeout(batchIntervalSeconds * 5 * 1000)
```

Interactive Querying

As you can see from the example below, the below query will change every time you execute it to reflect the current word count based on the input stream of data.

word	count
6	85
0	80
9	106
3	88
4	95
7	90
a	895
l	895
one	895

Once you are done, just execute the statement below to stop the streaming context.

```
StreamingContext.getActive.foreach { _.stop(stopSparkContext = false) }
```

In closing

We hope you found this tutorial helpful in getting started on Spark. If you have further questions, be sure to visit azure.com/databricks.