**Some Sources:**

<https://people.apache.org/~pwendell/spark-nightly/spark-master-docs/latest/sql-programming-guide.html>

Electronics Retail Company, "Initech", with brick & mortar locations as well as an online marketplace

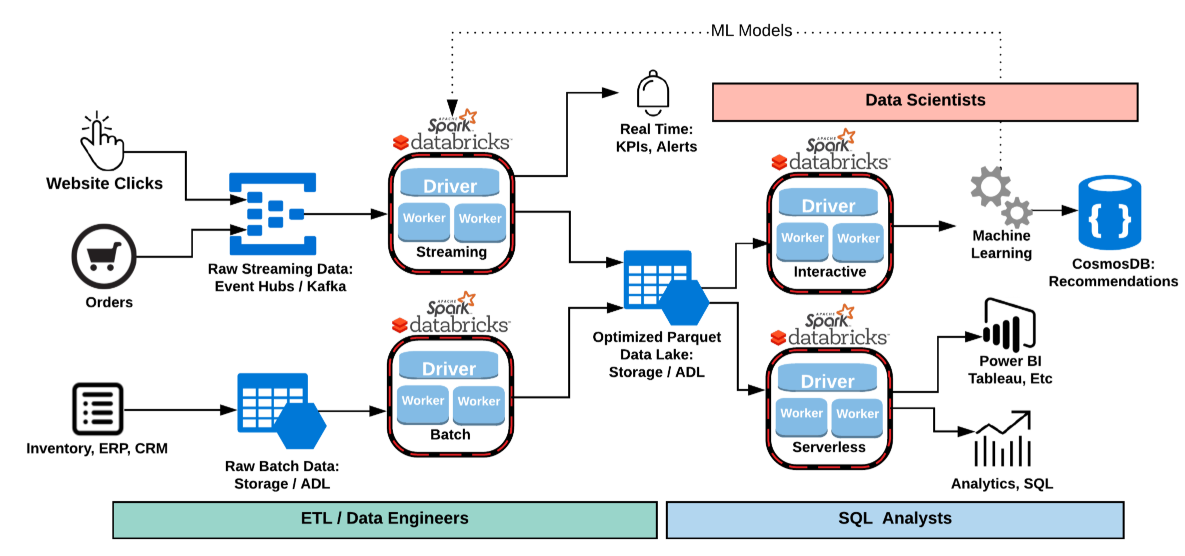
%md

### Business Challenges

\* Total revenues have been down 10% YoY

\* Online sales growing, but at a slower rate than Amazon and others. Only ~2% YoY, they are looking to grow ~8-10% YoY

\* Company wide initiative to increase online traffic and conversion rate to meet that ~8-10% YoY growth target

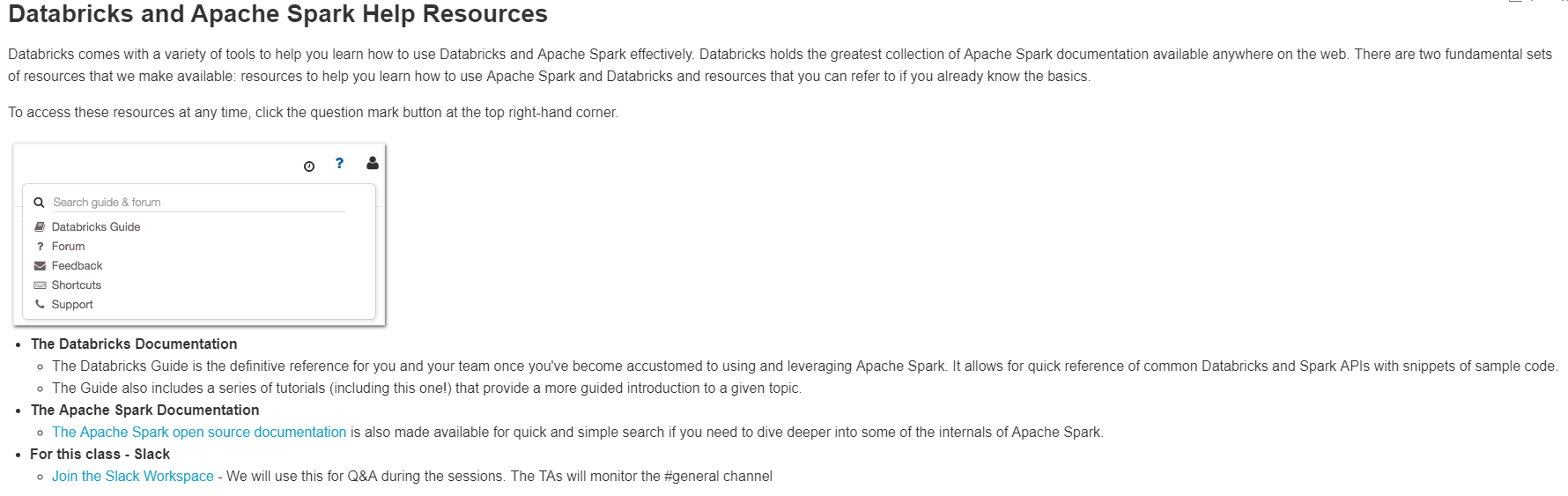


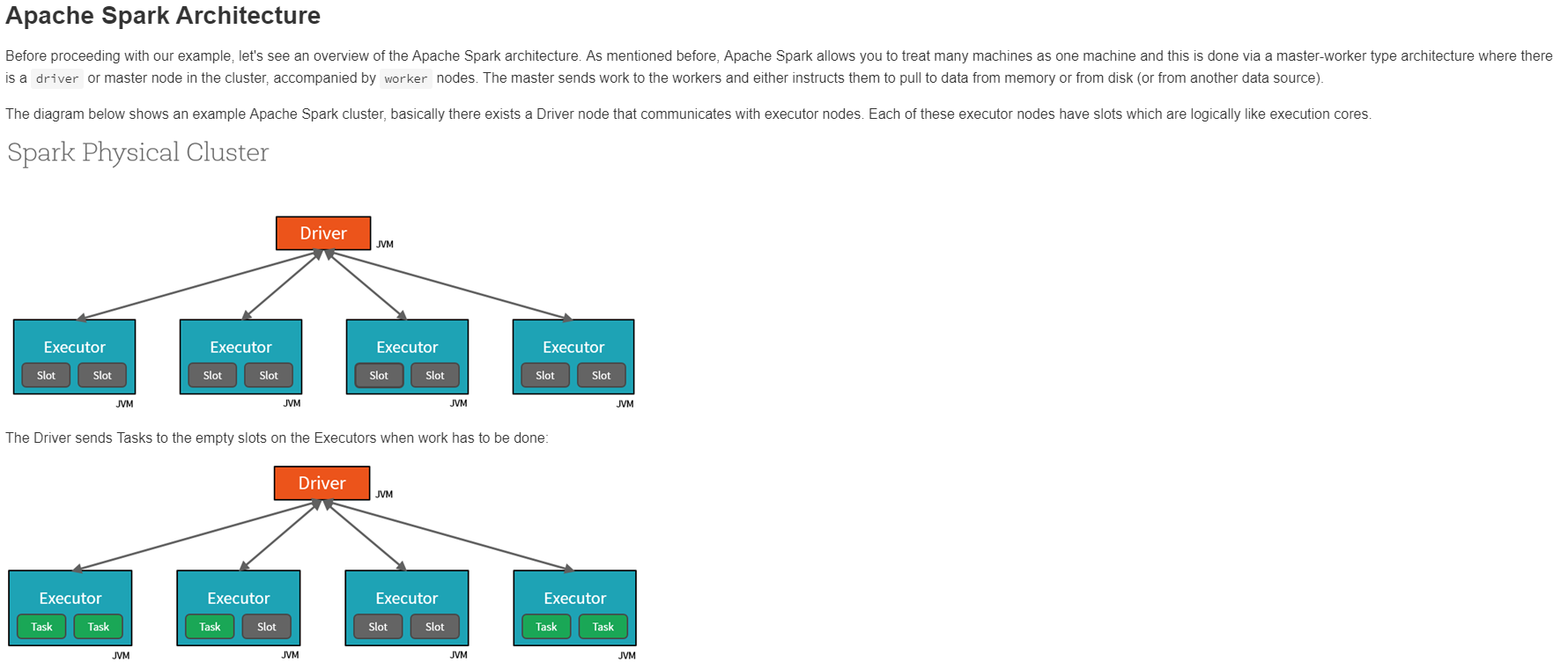
**What is Azure Databricks?:Azure Databricks is Unified Analytics Platform for Data Engineers, Data Scientists, and Analysis**

## Databricks Terminology

Databricks has key concepts that are worth understanding. You'll notice that many of these line up with the links and icons that you'll see on the left side. These together define the fundamental tools that Databricks provides to you as an end user. They are available both in the web application UI as well as the REST API.

* **Workspaces**
  + Workspaces allow you to organize all the work that you are doing on Databricks. Like a folder structure in your computer, it allows you to save **notebooks** and **libraries** and share them with other users. Workspaces are not connected to data and should not be used to store data. They're simply for you to store the **notebooks** and **libraries** that you use to operate on and manipulate your data with.
* **Notebooks**
  + Notebooks are a set of any number of cells that allow you to execute commands. Cells hold code in any of the following languages: Scala, Python, R, SQL, or Markdown. Notebooks have a default language, but each cell can have a language override to another language. This is done by including %[language name] at the top of the cell. For instance %python. We'll see this feature shortly.
  + Notebooks need to be connected to a **cluster** in order to be able to execute commands however they are not permanently tied to a cluster. This allows notebooks to be shared via the web or downloaded onto your local machine.
  + **Dashboards**
    - **Dashboards** can be created from **notebooks** as a way of displaying the output of cells without the code that generates them.
  + **Notebooks** can also be scheduled as **jobs** in one click either to run a data pipeline, update a machine learning model, or update a dashboard.
* **Libraries**
  + Libraries are packages or modules that provide additional functionality that you need to solve your business problems. These may be custom written Scala or Java jars; python eggs or custom written packages. You can write and upload these manually or you may install them directly via package management utilities like pypi or maven.
* **Tables / SQL**
  + Tables are structured data that you and your team will use for analysis. Tables can exist in several places. Tables can be stored in cloud storage, they can be stored on the cluster that you're currently using, or they can be cached in memory. [For more about tables see the documentation](https://docs.cloud.databricks.com/docs/latest/databricks_guide/index.html#02%20Product%20Overview/07%20Tables.html).
* **Clusters**
  + Clusters are groups of computers that you treat as a single computer. In Databricks, this means that you can effectively treat 20 computers as you might treat one computer. Clusters allow you to execute code from **notebooks** or **libraries** on set of data. That data may be raw data located on cloud storage or structured data that you uploaded as a **table** to the cluster you are working on.
  + It is important to note that clusters have access controls to control who has access to each cluster.
* **Jobs**
  + Jobs are the tool by which you can schedule execution to occur either on an already existing **cluster** or a cluster of its own. These can be **notebooks** as well as jars or python scripts. They can be created either manually or via the REST API.

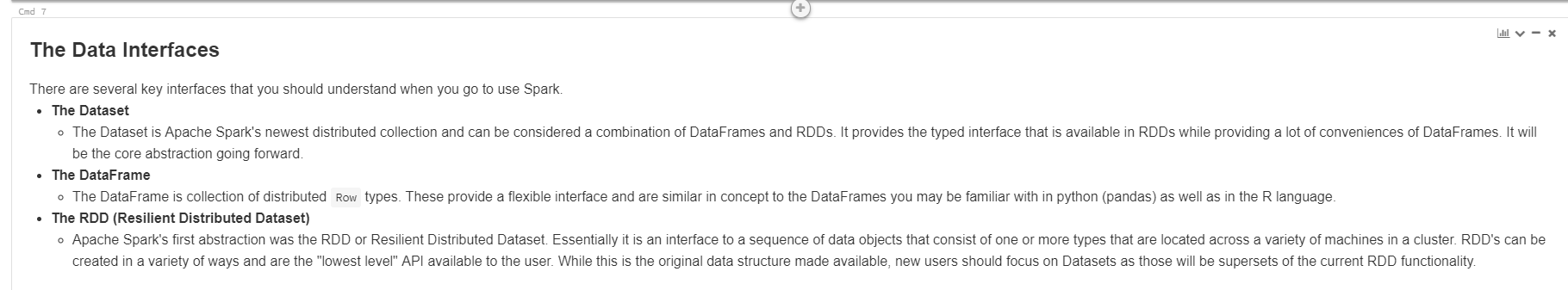




### The Data Interfaces

There are several key interfaces that you should understand when you go to use Spark.

* **The Dataset**
  + The Dataset is Apache Spark's newest distributed collection and can be considered a combination of DataFrames and RDDs. It provides the typed interface that is available in RDDs while providing a lot of conveniences of DataFrames. It will be the core abstraction going forward.
* **The DataFrame**
  + The DataFrame is collection of distributed Row types. These provide a flexible interface and are similar in concept to the DataFrames you may be familiar with in python (pandas) as well as in the R language.
* **The RDD (Resilient Distributed Dataset)**
  + Apache Spark's first abstraction was the RDD or Resilient Distributed Dataset. Essentially it is an interface to a sequence of data objects that consist of one or more types that are located across a variety of machines in a cluster. RDD's can be created in a variety of ways and are the "lowest level" API available to the user. While this is the original data structure made available, new users should focus on Datasets as those will be supersets of the current RDD functionality.



Writing like above one in Azure Databricks is given bellow

%md ### The Data Interfaces

There are several key interfaces that you should understand when you go to use Spark.

- \*\*\*\*The Dataset\*\*\*\*

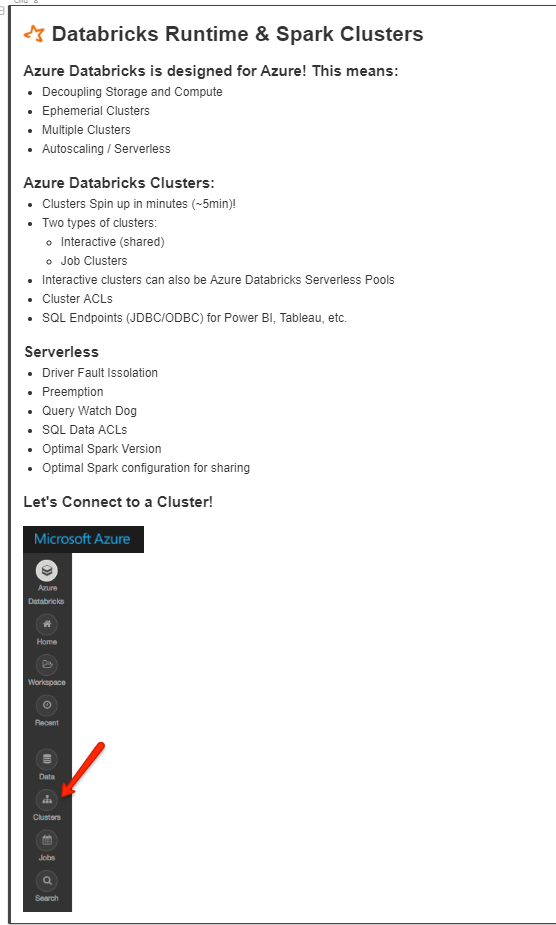
- The Dataset is Apache Spark's newest distributed collection and can be considered a combination of DataFrames and RDDs. It provides the typed interface that is available in RDDs while providing a lot of conveniences of DataFrames. It will be the core abstraction going forward.

- \*\*\*\*The DataFrame\*\*\*\*

- The DataFrame is collection of distributed `Row` types. These provide a flexible interface and are similar in concept to the DataFrames you may be familiar with in python (pandas) as well as in the R language.

- \*\*\*\*The RDD (Resilient Distributed Dataset)\*\*\*\*

- Apache Spark's first abstraction was the RDD or Resilient Distributed Dataset. Essentially it is an interface to a sequence of data objects that consist of one or more types that are located across a variety of machines in a cluster. RDD's can be created in a variety of ways and are the "lowest level" API available to the user. While this is the original data structure made available, new users should focus on Datasets as those will be supersets of the current RDD functionality.



Writing like above one in Azure Databricks is given bellow

%md

##![Spark Logo Tiny](https://kpistoropen.blob.core.windows.net/collateral/roadshow/logo\_spark\_tiny.png) Databricks Runtime & Spark Clusters

####Azure Databricks is designed for Azure! This means:

\* Decoupling Storage and Compute

\* Ephemerial Clusters

\* Multiple Clusters

\* Autoscaling / Serverless

####Azure Databricks Clusters:

\* Clusters Spin up in minutes (~5min)!

\* Two types of clusters:

\* Interactive (shared)

\* Job Clusters

\* Interactive clusters can also be Azure Databricks Serverless Pools

\* Cluster ACLs

\* SQL Endpoints (JDBC/ODBC) for Power BI, Tableau, etc.

####Serverless

\* Driver Fault Issolation

\* Preemption

\* Query Watch Dog

\* SQL Data ACLs

\* Optimal Spark Version

\* Optimal Spark configuration for sharing

####Let's Connect to a Cluster!

![Clusters](<https://kpistoropen.blob.core.windows.net/collateral/roadshow/clusters.png>)

**Databricks Mount Points:**

* Connect to our Azure Storage Account - <https://docs.azuredatabricks.net/spark/latest/data-sources/azure/azure-storage.html>
* Connect to our Azure Data Lake - <https://docs.azuredatabricks.net/spark/latest/data-sources/azure/azure-datalake.html>

dbutils.help() == to know diff packages under dbutils

# We have already mounted this directory, so no need to do it again.

# This is what the code looks like:

dbutils.fs.mount(

source = "wasbs://source@adbworkshops.blob.core.windows.net/",

mount\_point = "/mnt/training-sources/",

extra\_configs = {"fs.azure.sas.source.adbworkshops.blob.core.windows.net": "SAS-KEY"})

%fs head /mnt/training-sources/initech/productsCsv/product.csv == used to see top values of the file present in the path

1. Start with the file **dbfs:/mnt/training-sources/initech/productsCsv/product.csv**, a file containing product details.
2. Read in the data and assign it to a DataFrame named **productDF**.
3. Run the last cell to verify that the data was loaded correctly and to print its schema.

**Azure Data Bricks Using Python**

spark.conf.set("spark.sql.shuffle.partitions", 4)

%fs head /mnt/training-sources/initech/productsCsv/product.csv

# A reference to our csv file , below is the path where the file is located

csv\_file = "/mnt/training-sources/initech/productsCsv/"

temp\_df = (spark.read # The DataFrameReader

#.option("delimiter", "\t") This is how we could pass in a Tab or other delimiter.

.csv(csv\_file) # Creates a DataFrame from CSV after reading in the file

)

display(temp\_df)

## Verify Your Work

Run the following cell to verify that your DataFrame was created properly.

%python

# TEST

product\_df.printSchema()

columns = product\_df.dtypes

assert len(columns) == 8, "Expected 8 columns but found " + str(len(columns))

assert columns[0][0] == "product\_id", "Expected column 0 to be \"product\_id\" but found \"" + columns[0][0] + "\"."

assert columns[0][1] == "int", "Expected column 0 to be of type \"int\" but found \"" + columns[0][1] + "\"."

assert columns[1][0] == "category", "Expected column 1 to be \"category\" but found \"" + columns[1][0] + "\"."

assert columns[1][1] == "string", "Expected column 1 to be of type \"string\" but found \"" + columns[1][1] + "\"."

assert columns[2][0] == "brand", "Expected column 2 to be \"brand\" but found \"" + columns[2][0] + "\"."

assert columns[2][1] == "string", "Expected column 2 to be of type \"string\" but found \"" + columns[2][1] + "\"."

print("Congratulations, all tests passed!\n")

Specifying the Schema for DF:

# Required for StructField, StringType, IntegerType, etc.

from pyspark.sql.types import \*

csv\_schema = StructType([

StructField("product\_id", LongType(), True),

StructField("category", StringType(), True), # TO-DO : you must complete the rest

<<FILL-IN>>

])

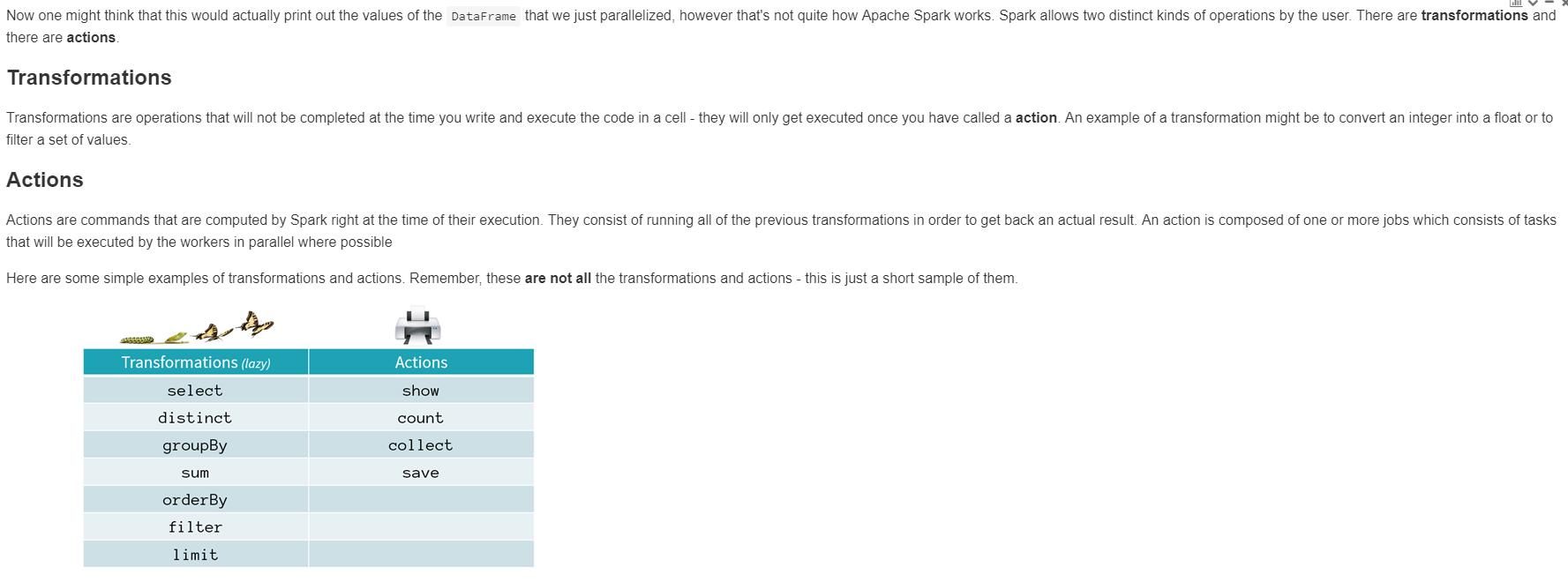
product\_df = (spark.read # The DataFrameReader

.option('header', 'true') # Ignore line #1 - it's a header

.schema(csv\_schema) # Use the specified schema

.csv(csv\_file) # Creates a DataFrame from CSV after reading in the file

)



### Challenges

* Larger Data
* Faster data and decisions - seconds, minutes, hours not days or weeks after it is created
* Streaming Pipelines can be hard
* Realtime Dashboards and alerts - for the holiday season, promotional campaigns, track falling or rising trends

### Azure Databricks Solutions

* Deploy Event Hubs with a click of button
* Connect Azure Databricks with a click of a button
* Easy streaming pipelines almost the same as batch - SQL, Python, Scala, Java & R
* Make this data avialable on Storage or ADL to end users in minutes not days or weeks.

### Why Initech Needs Streaming

* Sales up or down (rolling 24 hours, 1 hour), to identify trends that are good or bad
* Holidays and promotions - how are the performing in real time

## What is Structured Streaming?

Data is appended to the Input Table every trigger interval. For instance, if the trigger interval is 1 second, then new data is appended to the Input Table every seconds. (The trigger interval is analogous to the batch interval in the legacy RDD-based Streaming API.)

spark.conf.set("spark.sql.shuffle.partitions", 4)

## Part-1: Create Streaming DataFrame

#schema for our streaming DataFrame

from pyspark.sql.types import \*

schema = StructType([ \

StructField("orderUUID", StringType(), True), \

StructField("productId", IntegerType(), True), \

StructField("userId", IntegerType(), True), \

StructField("quantity", IntegerType(), True), \

StructField("discount", DoubleType(), True), \

StructField("orderTimestamp", TimestampType(), True)])

#streaming DataFrame reader for data on Azure Storage

streaming\_df = spark.readStream \

.schema(schema) \

.option("maxFilesPerTrigger", 1) \

.csv("dbfs:/mnt/training-sources/initech/streaming/orders/data/part-\*")

## What can we do with this Streaming DataFrame?

If you run the following cell, you'll get a continuously updating display of the number of records read from the stream so far. Note that we're just calling display() on our DataFrame, exactly as if it were a DataFrame reading from a static data source.

To stop the continuous update, just cancel the query.

display(streaming\_df)

## Part-2: Transform Streaming DataFrame

### It's just a DataFrame: We can use normal DataFrame transformations on our streaming DataFrame. For example, let's group the number of orders by productid

from pyspark.sql.functions import \*

top\_products = streaming\_df.groupBy("productId").agg(sum(col("quantity")).alias("total\_units\_by\_product")).orderBy(desc("total\_units\_by\_product"))

* Call display on top\_products
* Turn the streaming table into a streaming bar chart

display(top\_products)

## Part-3: Streaming Joins

Load the product lookup data from Azure Storage

product\_lookup = spark.read.parquet("/mnt/training-sources/initech/productsFull/")

joined\_df = streaming\_df.join(product\_lookup, "ProductID")

display(joined\_df)

## Part-4: Calculate a Streaming Dashboard - Revenue by Product Name

## top\_products = joined\_df.groupBy("Name").agg(sum(col("quantity")\*col("StandardCost")).alias("total\_revenue\_by\_product")).orderBy(desc("total\_revenue\_by\_product"))

## display(top\_products)

#### Azure Databricks for Serverless SQL & BI Queries for Data at Scale



To Create URL in Azure DataBricks like above

%md

##![Spark Logo Tiny](https://kpistoropen.blob.core.windows.net/collateral/roadshow/logo\_spark\_tiny.png)[Azure Databricks & Power BI Integration Docs](<https://docs.azuredatabricks.net/user-guide/bi/power-bi.html>)

**Machine Learning using Azure Data Bricks**

# Providing Product Recommendations: One of the most common uses of big data is to predict what users want. This allows Google to show you relevant ads, Amazon to recommend relevant products, and Netflix to recommend movies that you might like. This lab will demonstrate how we can use Apache Spark to recommend products to a user.

We will start with some basic techniques, and then use the SparkML library's Alternating Least Squares method to make more sophisticated predictions. Here are the SparkML [Python docs](https://spark.apache.org/docs/latest/api/python/pyspark.ml.html) and the [Scala docs](https://spark.apache.org/docs/latest/api/scala/#org.apache.spark.ml.package).

For this lesson, we will use around 900,000 historical product ratings from our company Initech.

In this lab:

* Part 0: Exploratory Analysis
* Part 1: Collaborative Filtering
* Part 2: Analysis

## Part 0: Exploratory Analysis

Let's start by taking a look at our data. It's already mounted in /mnt/training-msft/ratings.parquet table for us. Exploratory analysis should answer questions such as:

* How many observations do I have?
* What are the features?
* Do I have missing values?
* What do summary statistics (e.g. mean and variance) tell me about my data?

Start by importing the data. Bind it to productRatings by running the cell below

**product\_ratings = spark.read.parquet("dbfs:/mnt/training-sources/initech/productRatings/")**

**reading products from path**

**product\_df = spark.read.parquet("**dbfs:/mnt/training-sources/initech/productsShort/**")**

## Part 1: Collaborative Filtering

The image below (from [Wikipedia](https://en.wikipedia.org/?title=Collaborative_filtering)) shows an example of predicting of the user's rating using collaborative filtering. At first, people rate different items (like videos, products, articles, images, games). After that, the system is making predictions about a user's rating for an item, which the user has not rated yet. These predictions are built upon the existing ratings of other users, who have similar ratings with the active user. For instance, in the image below the system has made a prediction, that the active user will not like the video.

<https://courses.edx.org/c4x/BerkeleyX/CS100.1x/asset/Collaborative_filtering.gif>

**#We'll hold out 60% for training, 20% of our data for validation, and leave 20% for testing**

**seed = 1800009193L**

**(training\_df, validation\_df, test\_df) = product\_ratings.randomSplit([.6, .2, .2], seed=seed)**

### My Ratings

* Fill in your ratings for the above product\_df
* Pick 5-10 product ids to rate
* Choose your ratings be 1-5

**my\_user\_id = 0**

**my\_rated\_products = [**

**(1, my\_user\_id, 5), # Replace with your ratings.**

**(2, my\_user\_id, 5), (3, my\_user\_id, 5), (4, my\_user\_id, 5), (6, my\_user\_id, 1), (7, my\_user\_id, 1),**

**(9, my\_user\_id, 1), (9, my\_user\_id, 1), (9, my\_user\_id, 1), ]**

**my\_ratings\_df = spark.createDataFrame(my\_rated\_products, ['product\_id','user\_id','rating'])**

Join your ratings with the product\_df to see your ratings with the product metadata

**display(my\_ratings\_df.join(product\_df, ['product\_id']))**

Union your ratings with the trainingDF to see your ratings with the product metadata

**training\_with\_my\_ratings\_DF = training\_df.union(my\_ratings\_df)**

### **Alternating Least Squares**

In this part, we will use the Apache Spark ML Pipeline implementation of Alternating Least Squares, [ALS (Python)](http://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.recommendation.ALS) or [ALS (Scala)](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.recommendation.ALS). ALS takes a training dataset (DataFrame) and several parameters that control the model creation process.

The process we will use for determining the best model is as follows:

1. Pick a set of model parameters. The most important parameter to model is the rank, which is the number of columns in the Users matrix (green in the diagram above) or the number of rows in the Products matrix (blue in the diagram above). In general, a lower rank will mean higher error on the training dataset, but a high rank may lead to [overfitting](https://en.wikipedia.org/wiki/Overfitting). We will train models with a rank of 2 using the trainingDF dataset.
2. Set the appropriate parameters on the ALS object:
   * The "User" column will be set to the values in our user\_id DataFrame column.
   * The "Item" column will be set to the values in our product\_id DataFrame column.
   * The "Rating" column will be set to the values in our rating DataFrame column.
   * We'll be using a regularization parameter of 0.1.

**Note**: Read the documentation for the ALS class **carefully**. It will help you accomplish this step.

1. Have the ALS output transformation (i.e., the result of ALS.fit()) produce a new column called "prediction" that contains the predicted value.
2. Create multiple models using ALS.fit(), one for each of our rank values. We'll fit against the training data set (trainingDF).
3. We'll run our prediction against our validation data set (validationDF) and check the error.
4. Use .setColdStartStrategy("drop") so that the model can deal with missing values.

**from pyspark.ml.recommendation import ALS**

**# Let's initialize our ALS learner**

**als = ALS()**

**# Now we set the parameters for the method**

**(als.setPredictionCol("prediction")**

**.setUserCol("user\_id")**

**.setItemCol("product\_id")**

**.setRatingCol("rating")**

**.setMaxIter(5)**

**.setSeed(seed)**

**.setRegParam(0.1)**

**.setRank(2)**

**.setColdStartStrategy("drop")**

**)**

**model = als.fit(training\_with\_my\_ratings\_DF) #fill in with training\_with\_my\_ratings\_DF**

**# Run the model to create a prediction. Predict against the validationDF.**

**predict\_df = model.transform(validation\_df)**

**display(predict\_df)**

## Part 2: Your Recommendations:

Let's look at what ALS recommended for your user based on your ratings

#Filter the predictions DF for your user id something like "user\_id = ID"

predictions = model.recommendForAllUsers(10)

my\_predictions = predictions.filter("user\_id = 0")

display(my\_predictions)

from pyspark.sql.functions import \*

my\_recs = my\_predictions.select("user\_id", explode("recommendations").alias("recommendations")).select("user\_id", "recommendations.product\_id", "recommendations.rating")

display(my\_recs)

from pyspark.sql.functions import \*

my\_recs = my\_predictions.select("user\_id", explode("recommendations").alias("recommendations")).select("user\_id", "recommendations.product\_id", "recommendations.rating").join(product\_df, ['product\_id'])

**display(my\_recs)**

**Azure Data Bricks Using SQL**

**SET spark.sql.shuffle.partitions = 4**

### Step #1 - Create a SparkSQL table

* Change the name of the table to your\_name\_products

**CREATE OR REPLACE TEMPORARY VIEW products**

**USING CSV**

**OPTIONS (path "/mnt/training-sources/initech/productsCsv/")**

**SELECT \* FROM products**

### Step #2 - Create a SparkSQL table that infers the schema

* Change the name of the table to your name
* Add the extra OPTIONS - hint:
  + header
  + inferSchema

**CREATE OR REPLACE TEMPORARY VIEW products**

**USING CSV**

**OPTIONS (path "/mnt/training-sources/initech/productsCsv/",**

**header “True”, inferSchema “True” )**

**Specifying the Schema while reading**

**CREATE OR REPLACE TEMPORARY VIEW products (**

**product\_id int,**

**--TO-D0--**

**USING CSV**

**OPTIONS (path "/mnt/training-sources/initech/productsCsv/",**

**--TO-D0-- )**

**Streaming**

## Part-1: Create Streaming DataFrame

**%python**

**#schema for our streaming DataFrame**

**from pyspark.sql.types import \***

**schema = StructType([ \**

**StructField("orderUUID", StringType(), True), \**

**StructField("productId", IntegerType(), True), \**

**StructField("userId", IntegerType(), True), \**

**StructField("quantity", IntegerType(), True), \**

**StructField("discount", DoubleType(), True), \**

**StructField("orderTimestamp", TimestampType(), True)])**

**streaming\_df = spark.readStream \**

**.schema(schema) \**

**.option("maxFilesPerTrigger", 1) \**

**.csv("dbfs:/mnt/training-sources/initech/streaming/orders/data/part-\*")**

**streaming\_df.createOrReplaceTempView("orders")**

**SELECT \* FROM orders**

## Part-2: Transform Streaming Table

### It's just a SparkSQL Table

We can use normal Spark SQL transformations on our streaming DataFrame. For example, let's group the number of orders by productId

**SELECT**

**sum(quantity) AS total\_units\_by\_product,**

**productId**

**FROM orders**

**GROUP BY productId**

**ORDER BY total\_units\_by\_product DESC**

## Part-3: Streaming Joins

### Streaming Joins

Grouping by unkown product IDs is not that that exciting. Let's join the stream with the product lookup data set

**CREATE OR REPLACE TEMPORARY VIEW products**

**USING parquet**

**OPTIONS ("path" "/mnt/training-sources/initech/productsFull/")**

**ML in Azure Data Bricks**

**CREATE OR REPLACE TEMP VIEW ratings**

**USING PARQUET**

**OPTIONS ("path" "dbfs:/mnt/training-sources/initech/productRatings/")**

**SELECT \* FROM ratings**

Take a count of the data using the count() in the SQL query

**SELECT count(\*) FROM ratings**

**CREATE OR REPLACE TEMP VIEW products**

**USING PARQUET**

**OPTIONS ("path" "dbfs:/mnt/training-sources/initech/productsShort/")**

## Part 1: Collaborative Filtering

**%python**

**product\_ratings = table("ratings")**

**seed = 1800009193L**

**(training\_df, validation\_df, test\_df) = product\_ratings.randomSplit([.6, .2, .2], seed=seed)**

### My Ratings

* Fill in your ratings for the above product\_df
* Pick 5-10 product ids to rate
* Choose your ratings be 1-5

**%python**

**my\_user\_id = 0**

**my\_rated\_products = [ (1, my\_user\_id, 5), # Replace with your ratings.**

**(2, my\_user\_id, 5), (3, my\_user\_id, 5), (4, my\_user\_id, 5), (6, my\_user\_id, 1), (7, my\_user\_id, 1),**

**(9, my\_user\_id, 1), (9, my\_user\_id, 1), (9, my\_user\_id, 1), ]**

**spark.createDataFrame(my\_rated\_products, ['product\_id','user\_id','rating']).createOrReplaceTempView("my\_ratings")**

**SELECT \* FROM my\_ratings JOIN products ON (my\_ratings.product\_id = products.product\_id)**

### Union my\_ratings table with existing ratings table

* use SQL's UNION ALL operator

**CREATE OR REPLACE TEMP VIEW ratings\_with\_my\_ratings AS**

**SELECT \* FROM my\_ratings UNION ALL SELECT \* FROM ratings**

### **Alternating Least Squares**

**%python**

**from pyspark.ml.recommendation import ALS**

**# Let's initialize our ALS learner**

**als = ALS()**

**# Now we set the parameters for the method**

**(als.setPredictionCol("prediction") .setUserCol("user\_id") .setItemCol("product\_id") .setRatingCol("rating")**

**.setMaxIter(5) .setSeed(seed) .setRegParam(0.1) .setRank(2) .setColdStartStrategy("drop") )**

**%python**

**training\_with\_my\_ratings\_DF = table("ratings\_with\_my\_ratings")**

**model = als.fit(training\_with\_my\_ratings\_DF)**

**# Run the model to create a prediction. Predict against the validationDF.**

**model.transform(validation\_df).createOrReplaceTempView("validation")**

**SELECT \* FROM validation**

## Part 2: Your Recommendations:

Let's look at what ALS recommended for your user based on your ratings

%python

model.recommendForAllUsers(10).createOrReplaceTempView("predictions")

SELECT \* FROM predictions

### Find your predictions:

Filter for your user\_id

SELECT \* FROM predictions WHERE user\_id = 0

SELECT user\_id, product.product\_id, product.rating FROM predictions LATERAL VIEW explode(recommendations) AS product WHERE user\_id = 0

### Join your recommendations with the products table

* Make the recommendations human readable by joining with the lookup
* Join on product\_id

SELECT \* FROM (SELECT user\_id, product.product\_id, product.rating FROM predictions

LATERAL VIEW explode(recommendations) AS product WHERE user\_id = 0) as pred JOIN products

ON (pred.product\_id = products.product\_id)

**Some Other Missing works on above things**

**Using Python -1**

**# A reference to our csv file**

**csv\_file = "/mnt/training-sources/initech/productsCsv/"**

**temp\_df = (spark.read # The DataFrameReader**

**#.option("delimiter", "\t") This is how we could pass in a Tab or other delimiter.**

**.csv(csv\_file) # Creates a DataFrame from CSV after reading in the file )**

## Read CSV using Spark's built in CSV reader to Infer the Schema

**csv\_file = "dbfs:/mnt/training-sources/initech/productsCsv/product.csv"**

**product\_df = (spark.read # The DataFrameReader**

**.option("header", "true") # Use first line of all files as header**

**.option("inferSchema", "true") # Automatically infer data types**

**.csv(csv\_file) # Creates a DataFrame from CSV after reading in the file**

**)**