Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

Spark Sql Notebooks

- In the **Import Notebooks** pane, select **URL**, and paste in the following URL:
  - https://github.com/MicrosoftDocs/mslearn-read-and-write-data-using-azure-databricks/blob/master/DBC/04-Reading-Writing-Data.dbc?raw=true
  - It includes the following notebooks:
    - **01-Getting-Started**: This notebook gets you started with your Databricks workspace.
    - **02-Querying-Files**: This notebook contains exercises to help you query large data files and visualize your results.
    - **03-Joins-Aggregations**: You do basic aggregation and joins in this notebook.
    - **04-Accessing-Data**: This notebook lists the steps for accessing data from various sources by using Databricks.
    - **05-Querying-JSON**: In this notebook, you learn how to query JSON and hierarchical data with DataFrames.
    - **06-Data-Lakes**: This notebook lists exercises that show how to create an Azure Data Lake Storage Gen2 instance and use Databricks DataFrames to query and analyze this data.
    - **07-Azure-Data-Lake-Gen2**: In this notebook, you use Databricks to query and analyze data stores in Azure Data Lake Storage Gen2.
    - **08-Key-Vault-backed-secret-scopes**: This notebook lists the steps for configuring a Key Vault-backed secret scope. You'll create a Key Vault-backed secret scope and securely store in it usernames and passwords for a sample SQL database and an Azure Cosmos DB instance to be used in the following notebooks.
    - **09-SQL-Database-Connect-Using-Key-Vault**: In this notebook, you'll connect to a SQL database by using your Azure SQL username and password that you created and securely stored in the Key Vault-backed secret scope in the previous notebook.
    - **10-Cosmos-DB-Connect-Using-Key-Vault**: In this notebook, you'll connect to an Azure Cosmos DB instance by using the Azure Cosmos DB username and password that you previously created and securely stored in the Key Vault-backed secret scope.
    - **Exploratory-Data-Analysis**: This notebook is in the *Optional* subfolder. It includes a sample project for you to explore later.

02-Querying-Files

- **Querying files:** You can use DataFrames to query large data files. DataFrames are derived from data structures known as resilient distributed datasets (RDDs). RDDs and DataFrames are immutable distributed collections of data.
  - ***Resilient***: RDDs are fault tolerant, so if part of your operation fails, Spark quickly rebuilds any lost data.
  - ***Distributed***: RDDs are distributed across networked machines known as a cluster.
  - ***DataFrame***: A data structure where data is organized into named columns, like a table in a relational database, but with richer optimizations available.
- **Azure Key Vault-backed secret scopes:** Azure Databricks has two types of secret scopes:
  - Key Vault-backed
  - Databricks-backed
- These secret scopes allow you to securely store secrets, such as database-connection strings. If someone tries to output a secret to a notebook, the secret is replaced with **[REDACTED]**. This behavior helps prevent unauthorized users from viewing the secret or someone accidentally leaking the secret when displaying or sharing the notebook.
- note that you do not have to worry about how to order operations because the optimizier determines the optimal order of execution of the operations for you.
  - df.select(...).orderBy(...).filter(...) vs
  - df.filter(...).select(...).orderBy(...)

Spark Sql

SQL	DataFrame (Python)
`SELECT col_1 FROM myTable`	`df.select(col("col_1"))`
`DESCRIBE myTable`	`df.printSchema()`
`SELECT * FROM myTable WHERE col_1 > 0`	`df.filter(col("col_1") > 0)`
`..GROUP BY col_2`	`..groupBy(col("col_2"))`
`..ORDER BY col_2`	`..orderBy(col("col_2"))`
`..WHERE year(col_3) > 1990`	`..filter(year(col("col_3")) > 1990)`
`SELECT * FROM myTable LIMIT 10`	`df.limit(10)`
`display(myTable)` (text format)	`df.show()`
`display(myTable)` (html format)	`display(df)`

These two generate the same results:

```
%sql
SELECT count(*)
FROM PeopleDistinctNames
INNER JOIN SSADistinctNames ON firstName = ssaFirstName
```

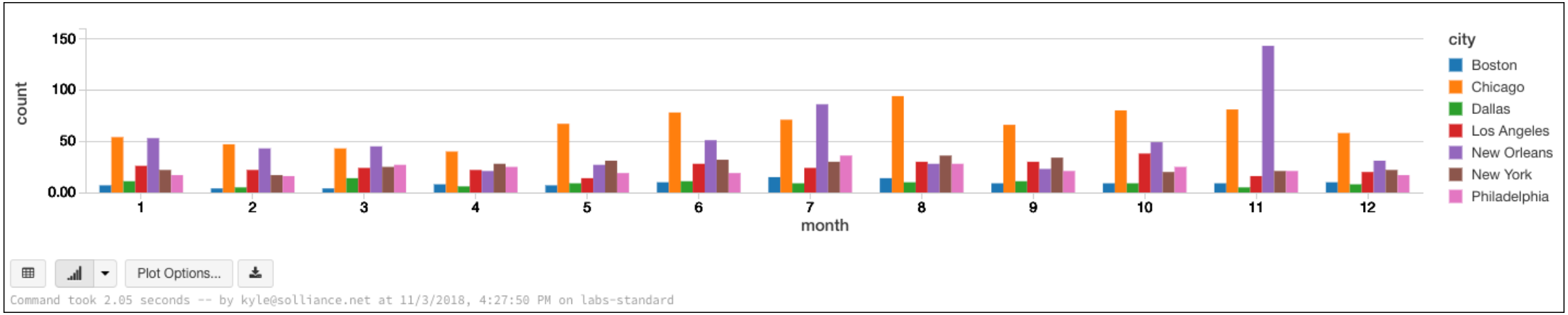
```
%sql
SELECT count(firstName)
FROM PeopleDistinctNames
WHERE firstName IN (
  SELECT ssaFirstName FROM SSADistinctNames
)
```

03-Joins-Aggregations

- Note: Sql queries can be run using spark.sql("") command:
  - **Spark.sql("select * from myTable")**
- .agg(count("*").alias("total"))
- Run linux commands using %fs:
  - %fs ls /mnt/training/dataframes/people-10m.parquet (See the contents of the file)
- Read Parquet:
  - peopleDF = spark.read.parquet("/mnt/training/dataframes/people-10m.parquet")
  - display(peopleDF)
- Create column on the fly:
  - .select(year("birthDate").alias("birthYear"))
- In DataFrames, **temporary views** are used to make the DataFrame available to SQL, and work with SQL syntax seamlessly.
- **Temporary View** gives you a name to query from SQL, but unlike a table it exists only for the duration of your Spark Session.
  - As a result, the temporary view will not carry over when you restart the cluster or switch to a new notebook.
  - It also won't show up in the Data button on the menu on the left side of a Databricks notebook which provides easy access to databases and tables.
  - **peopleDF.createOrReplaceTempView("People10M")**
- Test a value with the expected value:
  - **dbTest**("test1", expected_value, value)
  - Print("success: value is equal to the expected value")
  - or
  - dbTest("test2", Row(salary=2), results[0])

Plot

```
1 display(homicidesDf.select("city", "month").orderBy("city", "month").groupBy("city", "month").count())
```



Review Questions

- Q: How do you create a DataFrame object?
- A: An object is created by introducing a variable name and equating it to something like myDataFrameDF =.

- Q: Why do you chain methods (operations) myDataFrameDF.select().filter().groupBy()?
- A: To avoid the creation of temporary DataFrames as local variables.
  - For example, you could have written the above as: tempDF1 = myDataFrameDF.select(), tempDF2 = tempDF1.filter() and then tempDF2.groupBy().
  - This is syntatically equivalent, but, notice how you now have extra local variables.

- Q: What is the DataFrame syntax to create a temporary view?
- A: myDF.createOrReplaceTempView("MyTempView")

- Q: What is the equivalent of “SELECT count(*) from myTable”
- A: .agg(count(“*”).alias(“total”))

04-Accessing-Data

- The Databricks File System (DBFS) is the built-in alternative to the Hadoop Distributed File System (HDFS).
- **Parquet file**
  - A Parquet "file" is a collection of files (partitions) stored in a single directory.
  - The Parquet format offers features that make it the ideal choice for storing "big data" on distributed file systems.
- **Comma-separated-values (CSV)**
  - The file's delimiter; the default is ",".
  - Whether the file has a header or not; the default is **false**.
  - Whether or not to infer the schema; the default is **false**.
  - Take a look at the head of a csv file:
    - **%fs head** /mnt/training/bikeSharing/data-001/day.csv **--maxBytes=492**
  - **Read csv**
    - (spark
    - .read # Call the read method returning a DataFrame
    - .option("inferSchema","true") # Option to tell Spark to infer the schema
    - .option("header","true") # Option telling Spark that the file has a header
    - .csv("/mnt/training/bikeSharing/data-001/day.csv"))
- Take a look at the blobs already mounted to your DBFS:
  - %fs mounts

How to mount to Azure Blob file systems?

1 - Create a SAS URL

- Go to Azure portal,
- on select “storage accounts”,
- then “Shared access signature”,
- then “Generate SAS”.

2 – Run below Scala code to mount to azure blob

```
SasURL = "https://dbtraineastus2.blob.core.windows.net/?sv=2017-07-29&ss=b&srt=sco&sp=rl&se=2023-04-19T06:32:30Z&st=2018-04-18T22:32:30Z&spr=https&sig=BB%2FQzc0XHAH%2FarDQhKcpu49feb7llv3ZjnfViuI9IWo%3D"
indQuestionMark = SasURL.index('?')
SasKey = SasURL[indQuestionMark:len(SasURL)]
StorageAccount = "dbtraineastus2"
ContainerName = "training"
MountPoint = "/mnt/temp-training"

dbutils.fs.mount(
  source = "wasbs://%s@%s.blob.core.windows.net/" % (ContainerName, StorageAccount),
  mount_point = MountPoint,
  extra_configs = {"fs.azure.sas.%s.%s.blob.core.windows.net" % (ContainerName, StorageAccount) : "%s" % SasKey}
)
```

Review Questions

- Q: What is Azure Blob Store?
  - A: Blob Storage stores from hundreds to billions of objects such as unstructured data—images, videos, audio, documents easily and cost-effectively.
- Q: What is DBFS?
  - A: DBFS stands for Databricks File System. DBFS provides for the cloud what the Hadoop File System (HDFS) provides for local spark deployments. DBFS uses Azure Blob Store and makes it easy to access files by name.
- Q: Which is more efficient to query, a parquet file or a CSV file?
  - A: Parquet files are highly optimized binary formats for storing tables. The overhead is less than required to parse a CSV file. Parquet is the big data analogue to CSV as it is optimized, distributed, and more fault tolerant than CSV files.
- Q: What is the syntax for defining a DataFrame in Spark from an existing parquet file in DBFS?
  - A: Scala: val IPGeocodeDF = spark.read.parquet("dbfs:/mnt/training/ip-geocode.parquet")
  - Python: IPGeocodeDF = spark.read.parquet("dbfs:/mnt/training/ip-geocode.parquet")
- Q: What is the syntax for defining a DataFrame in Spark from an existing CSV file in DBFS using the first row in the CSV as the schema?
  - A: Scala: val myDF = spark.read.option("header","true").option("inferSchema","true").csv("dbfs:/mnt/training/myfile.csv")
  - Python: myDF = spark.read.option("header","true").option("inferSchema","true").csv("dbfs:/mnt/training/myfile.csv")

05-Querying-JSON

- JSON
  - a common file format used in big data applications and in data lakes (or large stores of diverse data).
  - File formats such as JSON arise out of a number of data needs. For instance, what if:
    - Your schema, or the structure of your data, **changes over time**?
    - You need **nested fields** like an array with many values or an array of arrays?
    - You don't know how you're going use your data yet, so you **don't want to spend time** creating relational tables?
  - The popularity of JSON is largely due to the fact that JSON allows for **nested, flexible schemas**.
- Add a column to an existing dataframe: .withColumn("new_col_name", values)
- Access to a nested column by "." object.
  - Json_df.select("dates.createdOn")
- Flatten a nested column:
  - Json_df.withColumn("publishedOn",col("dates.publishedOn"))
- Look at the size of the array columns using ***size***:
  - Json_df.select(***size***("authors").alias('number_of_authors'))
- Access the first element in the array column using ***[]***:
  - Json_df.select(col("authors")[0].alias("primaryAuthor")))
- Break the elements in an array to separate rows using ***explode***:
  - Json_df.select(explode(col("authors")).alias("author"))

Convert date string to date:

```
.select(
    F.to_date(F.date_format("dates.publishedOn", "yyyy-MM-dd")).alias("publishedOn")
)
```

Unit Test

- **from pyspark.sql import Row**

- # TEST – len of the output:
  - dbTest(“test1-count", 12, outputDF.count())
  - print("Tests passed!")
- # TEST – check some of the array entries:
  - results = outputDF.collect()
  - dbTest(“test2-array-0", Row(category=u'Announcements', total=72), results[0])
  - dbTest("test2-array-10", Row(category=u'Platform', total=4), results[10])
  - dbTest("test2-array-11", Row(category=u'Streaming', total=21), results[11])
  - print("Tests passed!")

Review Questions

- Q: What is the syntax for accessing nested columns?
  - A: Use the dot notation: select("dates.publishedOn")
- Q: What is the syntax for accessing the first element in an array?
  - A: Use the [subscript] notation: select("col(authors)[0]")
- Q: What is the syntax for expanding an array into multiple rows?
  - A: Use the explode method: select(explode(col("authors")).alias("Author"))

06-Data-Lakes
Data Warehouse vs Data Lakes

- **Data warehouse:** a central database that is a classic approach to querying data from different data types e.g. CSV, XML, JSON, etc.
  - Traditionally, data engineers must design the schema for the central database, extract the data from the various data sources, transform the data to fit the warehouse schema, and load it into the central database.
  - A data scientist can then query the data warehouse directly or query smaller data sets created to optimize specific types of queries.
  - The data warehouse approach works well, but requires a great deal of **up front effort to design and populate schemas**.
  - It also limits historical data, which is constrained to only the data that fits the warehouse's schema.
- **Data Lake** is an alternative approach which:
  - Is a storage repository that **cheap**ly stores a vast amount of raw data in its native format.
  - Consists of current and historical data dumps in various formats including XML, JSON, CSV, Parquet, etc.
  - May contain operational relational databases with live transactional data.
  - When querying the Data Lake, you will need to normalize data before extracting useful insights.
  - Data lakes reduce the upfront costs while still making data easier to query over time.
  - The normalized tables can later be loaded into a formal data warehouse through nightly batch jobs.
- **Spark** is ideal for querying **Data Lakes**.
  - Spark DataFrames can be used to read directly from raw files contained in a Data Lake and then execute queries to join and aggregate the data.
- Aggregate two dataframes:
  - df1.union(df2)
  - Df1.unionall(df2)
  - They must be identical dataframes structurally – i.e. same column names and types

Database vs Data Warehouse

- A database stores current data while a data warehouse stores historical data.
- A database often changes due to frequent updates done on it, and hence, it cannot be used for analysis or reaching decision.
- A general database is used for Online Transactional Processing while a data warehouse is used for Online Analytical Processing.

Review Questions

- **Q:** What is a Data Lake?
  - **A:** A Data Lake is a collection of data files gathered from various sources. Spark loads each file as a table and then executes queries by joining and aggregating these files.
- **Q:** What are advantages of Data Lakes over classic Data Warehouses?
  - **A:** Data Lakes allow for large amounts of data to be aggregated from many sources with minimal preparatory steps. Data Lakes also allow for very large files. Powerful query engines such as Spark can read the diverse collection of files and execute complex queries efficiently.
- **Q:** What are some advantages of Data Warehouses?
  - **A:** Data warehouses are neatly curated to ensure data from all sources fit a common schema. This makes them easy to query.
- **Q:** What's the best way to combine the advantages of Data Lakes and Data Warehouses?
  - **A:** Start with a Data Lake. As you query, you will discover cases where the data needs to be cleaned, combined, and made more accessible. Create periodic Spark jobs to read these raw sources and write new "golden" DataFrames that are cleaned and more easily queried.

Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

02-Querying-Files

Note: In **Natural Join**, If there is no condition specifies then it returns the rows based on the common column.

- alias a parquet file using a sql table name
- Use normal sql commands in spark:
  - %sql
  - SELECT firstName, middleName, lastName, birthDate
  - FROM People10M
  - WHERE year(birthDate) > 1990 AND gender = 'F'
- Look at the columns and their types:
  - %sql
  - **DESCRIBE** People10M
  - **Describe extended** People10M
- Create temporary table in sql:
  - %sql
  - **CREATE OR REPLACE TEMPORARY VIEW** TheDonnas **AS**
  - SELECT *
  - FROM People10M
  - WHERE firstName = 'Donna'
- Spark SQL queries tables that are backed by ***physical*** files.
- You can visualize the results of your queries with built-in Databricks graphs.

Create an unmanaged table using a path:

```
%sql
CREATE TABLE IF NOT EXISTS People10M
USING parquet
OPTIONS (
  path "/mnt/training/dataframes/people-10m.parquet"
)
```

Create unmanaged table from CSV file using path:

```
%sql

CREATE DATABASE IF NOT EXISTS Databricks;
USE Databricks;

CREATE TABLE IF NOT EXISTS AirlineFlight
USING CSV
OPTIONS (
  header="true",
  delimiter=",",
  inferSchema="true",
  path="dbfs:/mnt/training/asa/flights/small.csv"
);

CACHE TABLE AirlineFlight;

SELECT * FROM AirlineFlight;
```

Review

- Q: What is the prefix used in databricks cells to execute SQL queries?
  - %sql

- Q: How do temporary views differ from tables?
  - Tables are visible to all users, can be accessed from any notebook, and persist across server resets. Temporary views are only visible to the **current user**, in the **current notebook**, and are gone once the spark session ends.

- Q: What is the SQL syntax to create a temporary view?
  - CREATE OR REPLACE TEMPORARY VIEW <<ViewName>> AS <<Query>>

04-Accessing-Data

- Create a databricks table from a parquet file:
- How to access databricks tables?
  - “data” tab in the left hand side
  - Select the databricks
  - See the tables
- Create a databricks tables from a csv file:
- We can also create a table from a manual upload:
  - “data” tab in the left hand side
  - Select the folder you want to upload
  - Click “+” and upload the file and
  - select “Create table with UI”
- Create a databricks tables from a json file:

```
%sql
CREATE DATABASE IF NOT EXISTS databricks;

USE databricks;

CREATE TABLE IF NOT EXISTS IPGeocode
  USING parquet
  OPTIONS (
    path "dbfs:/mnt/training/ip-geocode.parquet"
  )
```

```
%sql
CREATE TABLE IF NOT EXISTS BikeSharingDay
  USING csv
  OPTIONS (
    path "/mnt/training/bikeSharing/data-001/day.csv",
    inferSchema "true",
    header "true"
  )
```

```
%sql
CREATE TABLE IF NOT EXISTS DatabricksBlog
  USING json
  OPTIONS (
    path "dbfs:/mnt/training/databricks-blog.json",
    inferSchema "true"
  )
```

Look at Mount Points

- Look at all mountPoints and their Sources:
  - **%fs mounts**
  - **Or**
  - **dbutils.fs.mounts()**

- To unmount a mounted directly:
  - **%fs unmount /mnt/temp-training**
  - **Or**
  - **dbutils.fs.unmount("/mnt/temp-training")**

Review

- Q: How can you create a new table?
  - Create new tables by either:
    - Uploading a new file using the Data tab on the left.
    - Mounting an existing file from DBFS and create a table from those existing files

- Q: What is the SQL syntax for defining a table in Spark from an existing parquet file in DBFS?
  - CREATE TABLE IF NOT EXISTS IPGeocode USING parquet OPTIONS (path "dbfs:/mnt/training/ip-geocode.parquet")

05-Querying-JSON

- Access to nested columns using “.”
- Flatten a nested column:
- Work with timestamps and dates:
- Get size and access elements in array columns:

- date_format(cast(dates.publishedOn as timestamp), "MMM dd, yyyy") as publishedOn

```
%sql
SELECT dates.createdOn, dates.publishedOn
FROM DatabricksBlog
```

```
%sql
CREATE OR REPLACE TEMPORARY VIEW DatabricksBlog2 AS
  SELECT *,
         cast(dates.publishedOn AS timestamp) AS publishedOn
  FROM DatabricksBlog
```

```
%sql
SELECT title,
       date_format(publishedOn, "MMM dd, yyyy") AS date,
       link
FROM DatabricksBlog2
WHERE year(publishedOn) = 2013
ORDER BY publishedOn
```

```
%sql
SELECT size(authors),
       authors
FROM DatabricksBlog

SELECT authors[0] AS primaryAuthor
FROM DatabricksBlog
```

Explode array elements within a single or multiple columns

- Explode array elements within a single column using “explode”:

```
%sql
SELECT title,
       authors,
       explode(authors) AS author,
       link
FROM DatabricksBlog
WHERE size(authors) > 1
ORDER BY title
```

- Explode array elements within multiple columns either using nested “explode” or “Lateral View”.

```
%sql
select title, author, category
from databricksblog
lateral view explode(authors) exploded_authors_view as author
lateral view explode(categories) exploded_categories_view as category
where title = 'Apache Spark 1.1: The State of Spark Streaming'
order by author, category
```

=

```
%sql
select title, author, explode(categories) as category
from(
  select title, explode(authors) as author, categories
  from databricksblog
  WHERE title = "Apache Spark 1.1: The State of Spark Streaming"
  )
order by author, category
```

summary

- Q: What is the syntax for accessing nested columns?
  - A: Use the dot notation: SELECT dates.publishedOn
- Q: What is the syntax for accessing the first element in an array?
  - A: Use the [subscript] notation: SELECT authors[0]
- Q: What is the syntax for expanding an array into multiple rows
  - A: Use the explode keyword, either:
    SELECT explode(authors) as Author or
    LATERAL VIEW explode(authors) exploded_authors_view AS author

06-Data-Lakes

- Accessing to DataLakes could be done through mounting, similar to mounting other azure storages.
- Combine the two datasets with the same schema with “union”.

```sql
%sql

CREATE OR REPLACE TEMPORARY VIEW CrimeDataNewYork
  USING parquet
  OPTIONS (
    path "dbfs:/mnt/training/crime-data-2016/Crime-Data-New-York-2016.parquet"
  )
```

```sql
%sql

CREATE OR REPLACE TEMPORARY VIEW HomicidesNewYork AS
  SELECT month(reportDate) AS month, offenseDescription AS offense
  FROM CrimeDataNewYork
  WHERE lower(offenseDescription) LIKE 'murder%' OR lower(offenseDescription) LIKE 'homicide%'
```

```sql
%sql

CREATE OR REPLACE TEMPORARY VIEW HomicidesBostonAndNewYork AS
  SELECT * FROM HomicidesNewYork
    UNION ALL
  SELECT * FROM HomicidesBoston
```

Add a column with constant value:

```sql
select month, robberies, 'Los Angeles' as city
from RobberiesByMonthLosAngeles
```

Double Union:

```sql
select * from HomicidesNewYork
Union all
select * from(
    select * from HomicidesBoston
    union all
    select * from HomicidesChicago
    )
```

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

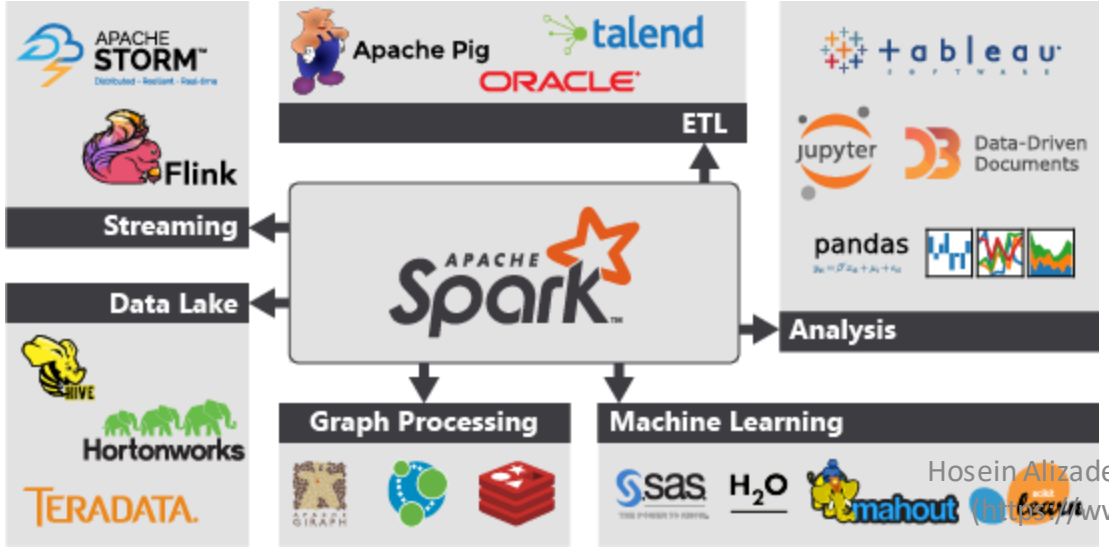
- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

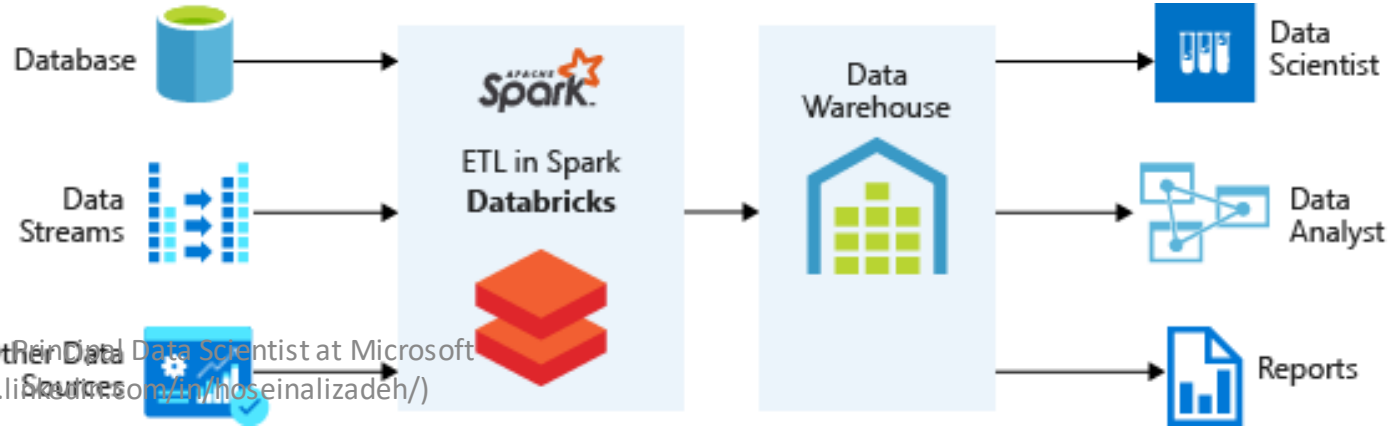
Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

ETL: Extract, Transform, and Load

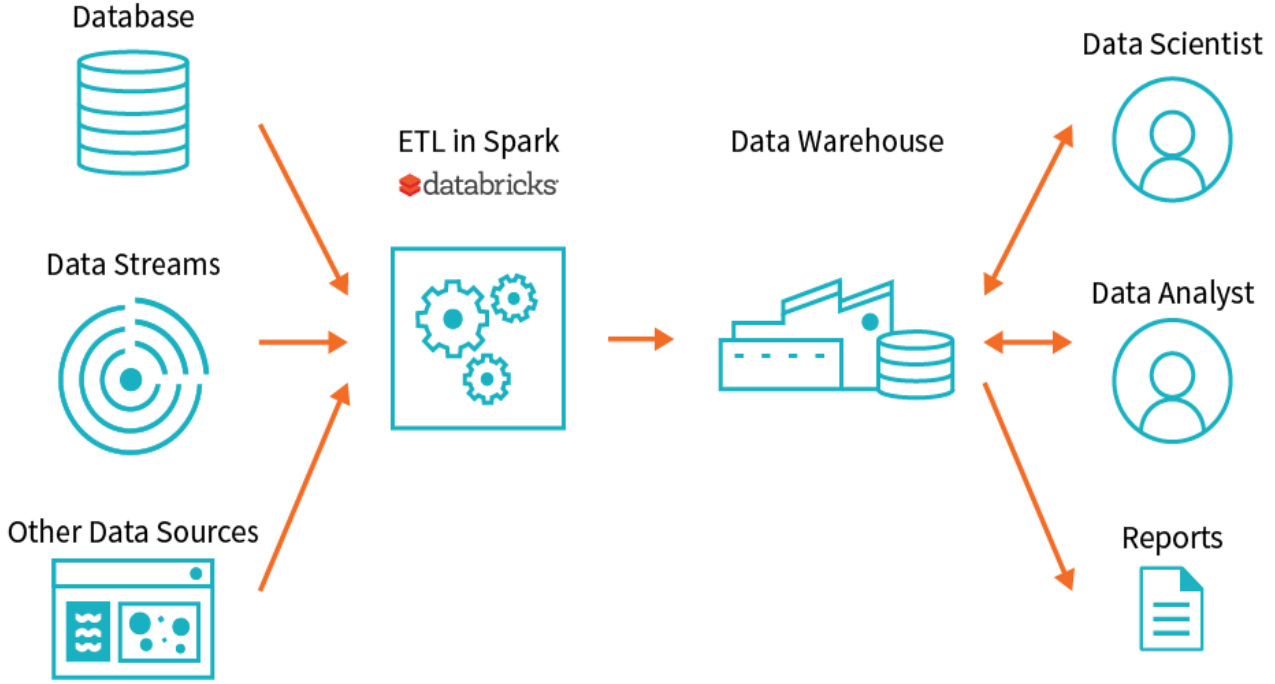
- Databricks and Spark offer a unified platform.
- Spark's unified platform is scalable to petabytes of data and clusters of thousands of nodes.
- Spark on Databricks decouples data storage from the compute and query engine.





ETL

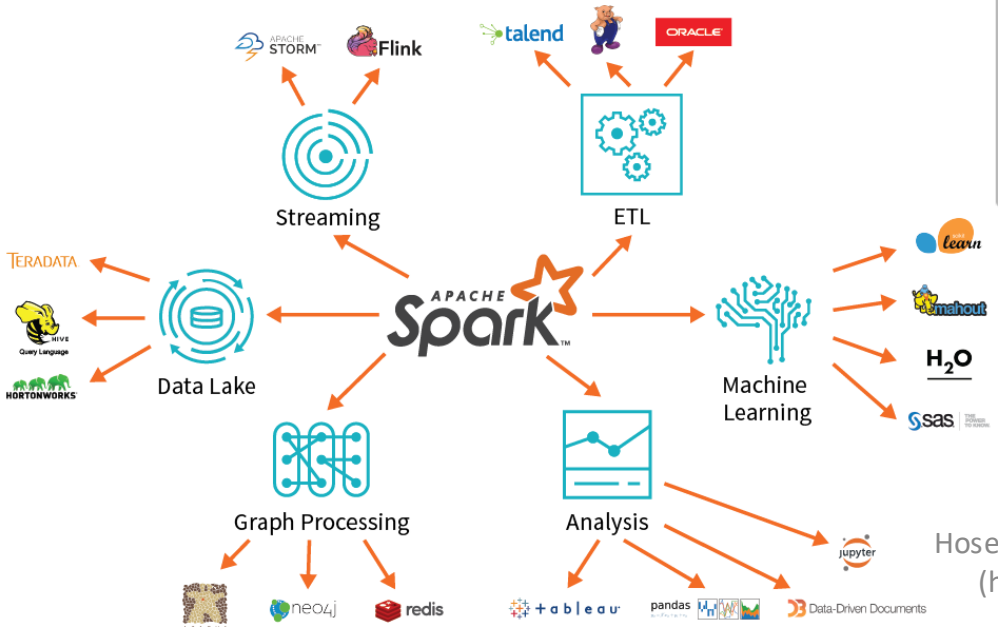
- **ETL concepts:**
  - Optimizing data formats and connections
  - Determining the ideal schema
  - Handling corrupt records
  - Automating workloads



ETL in Spark

```
path = "/mnt/training/EDGAR-Log-20170329/EDGAR-Log-20170329.csv"

logDF = (spark
  .read
  .option("header", True)
  .csv(path)
  .sample(withReplacement=False, fraction=0.3, seed=3) # using a sample to reduce data size
)
```

Read a sample of the csv file




```
(serverErrorDF
  .write
  .mode("overwrite") # overwrites a file if it already exists
  .parquet("/tmp/log20170329/serverErrorDF.parquet")
)
```

Write cleaned data back as Parquet

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

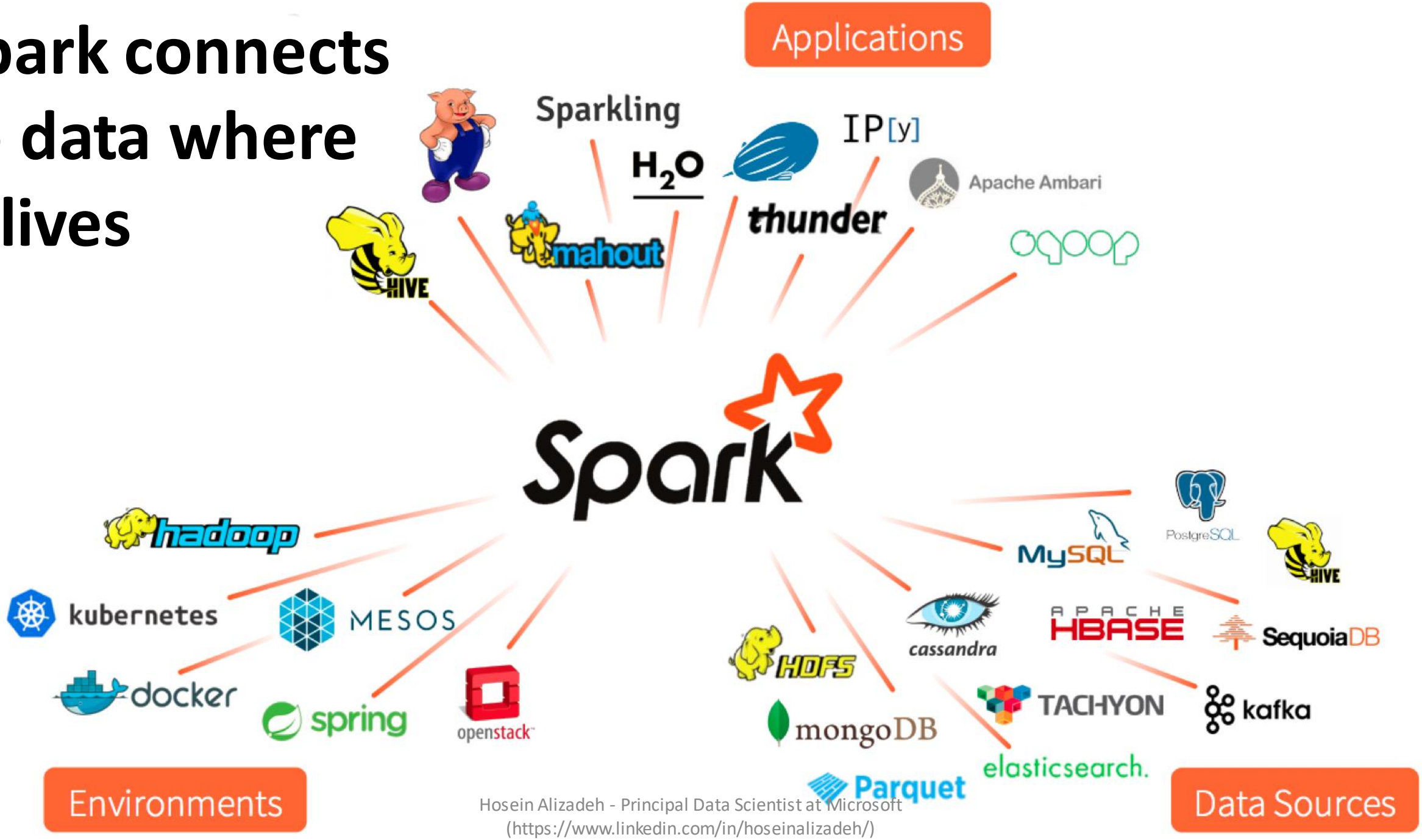
Review

- Question: What does ETL stand for and what are the stages of the process?
- Answer: ETL stands for extract-transform-load
  - Extract refers to ingesting data. Spark easily connects to data in a number of different sources.
  - Transform refers to applying structure, parsing fields, cleaning data, and/or computing statistics.
  - Load refers to loading data to its final destination, usually a database or data warehouse.

- Question: How does the Spark approach to ETL deal with devops issues such as updating a software version?
- Answer: By decoupling storage and compute, updating your Spark version is as easy as spinning up a new cluster. Your old code will easily connect to Azure Blob, or other storage. This also avoids the challenge of keeping a cluster always running, such as with Hadoop clusters.

- Question: How does the Spark approach to data applications differ from other solutions?
- Answer: Spark offers a unified solution to use cases that would otherwise need individual tools. For instance, Spark combines machine learning, ETL, stream processing, and several other solutions all with one technology.

Spark connects to data where it lives



Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Some dbutils functions

- to recursively remove all items from a directory:
  - **dbutils.fs.rm("/tmp/", True)**

- All clusters have storage available to them in:
  - **/tmp/**

- To remove all items recursively from a folder use:
  - %fs rm –r /folder/

03-Connecting-to-Azure-Blob-Storage

- Azure Blob Storage is the backbone of Databricks workflows.
  - By colocating data with Spark clusters, Databricks quickly reads from and writes to Azure Blob Storage in a distributed manner.
- The Databricks File System (DBFS), is a layer over Azure Blob Storage that allows you to mount Blob containers, making them available to other users in your workspace and persisting the data after a cluster is shut down.
- To define Azure Blob credentials, you need the following elements:
  - Storage account name
  - Container name
  - Mount point (how the mount will appear in DBFS)
  - Shared Access Signature (SAS) key
- Other connections work in much the same way, whether your data sits in Cassandra, Cosmos DB, Redshift, or another common data store. The general pattern is always:
  - Define the connection point
  - Define connection parameters such as access credentials
  - Add necessary options
  - After adhering to this, read data using `spark.read.options(<option key>, <option value>).<connection_type>(<endpoint>).`

How to mount a blob storage?

```python
#Define your Azure Blob credentials
STORAGE_ACCOUNT = "dbtraineastus2"
CONTAINER = "training"
MOUNT_POINT = "/mnt/training"
SAS_KEY = "?sv=2017-07-29&ss=b&srt=sco&sp=rl&se=2023-04-19T06:32:30Z&st=201

#Define the strings
source_str = "wasbs://{container}@{storage_acct}.blob.core.windows.net/".fo
conf_key = "fs.azure.sas.{container}.{storage_acct}.blob.core.windows.net".

#Mount
try:
  dbutils.fs.mount(
    source = source_str,
    mount_point = MOUNT_POINT,
    extra_configs = {conf_key: SAS_KEY}
  )
except Exception as e:
  print("ERROR: {} already mounted. Run previous cells to unmount first".format(MOUNT_POINT))
```

```python
STORAGE_ACCOUNT = "dbtraineastus2"
CONTAINER = "training"
SAS_KEY = "?sv=2017-07-29&ss=b&srt=sco&sp=rl&se=2023-04-19T06:32:30Z&st=2018-04-
mount_point = "/mnt/training"

source_string = f"wasbs://{CONTAINER}@{STORAGE_ACCOUNT}.blob.core.windows.net/"
conf_key = f"fs.azure.sas.{CONTAINER}.{STORAGE_ACCOUNT}.blob.core.windows.net"

try:
  dbutils.fs.unmount(mount_point)
except:
  print("okay")

dbutils.fs.mount(
  source = source_string,
  mount_point = mount_point,
  extra_configs = {conf_key:SAS_KEY}
)
```

How to read data?

```python
crimeDF = (spark.read
  .option("delimiter", "\t")
  .option("header", True)
  .option("timestampFormat", "mm/dd/yyyy hh:mm:ss a")
  .option("inferSchema", True)
  .csv("/mnt/training/Chicago-Crimes-2018.csv")
)
display(crimeDF)
```

timestampFormat

- spark.read
- .option("delimiter", "\t")
- .option("header", True)
- .option("inferSchema", True)
- **.option("timestampFormat", "MM/dd/yyyy hh:mm:ss a")**
- .csv("/mnt/training/Chicago-Crimes-2018.csv")

Review

- Question: What accounts for Spark's quick rise in popularity as an ETL tool?
  - Answer: Spark easily accesses data virtually anywhere it lives, and the scalable framework lowers the difficulties in building connectors to access data. Spark offers a unified API for connecting to data making reads from a CSV file, JSON data, or a database, to provide a few examples, nearly identical.
- Question: What is DBFS and why is it important?
  - Answer: The Databricks File System (DBFS) allows access to scalable, fast, and distributed storage backed by S3 or the Azure Blob Store.
- Question: How do you connect your Spark cluster to the Azure Blob?
  - Answer: By mounting it. Mounts require Azure credentials such as SAS keys and give access to a virtually infinite store for your data. One other option is to define your keys in a single notebook that only you have permission to access. Click the arrow next to a notebook in the Workspace tab to define access permissions.
- Question: How do you specify parameters when reading data?
  - Answer: Using .option() during your read allows you to pass key/value pairs specifying aspects of your read. For instance, options for reading CSV data include header, delimiter, and inferSchema.
- Question: What is the general design pattern for connecting to your data?
  - Answer: The general design pattern is as follows:
    - Define the connection point
    - Define connection parameters such as access credentials
    - Add necessary options such as for headers or parallelization

04-Connecting-to-JDBC
How to read from a database via JDBC?

This command is executed as a serial read through a single connection to the database.

This works well for small data sets; at scale, parallel reads are necessary for optimal performance.

```
#Define your database connection criteria
jdbcHostname = "server1.databricks.training"
jdbcPort = 5432
jdbcDatabase = "training"

jdbcUrl = "jdbc:postgresql://{0}:{1}/{2}".format(jdbcHostname, jdbcPort, jdbcDatabase)

#Define connection properties
connectionProps = {
  "user": "readonly",
  "password": "readonly"
}

#Read from the database
accountDF = spark.read.jdbc(url=jdbcUrl, table="Account", properties=connectionProps)
display(accountDF)
```

Review

- Java Database Connectivity (JDBC) is an API that defines database connections in Java environments.
  - ***Spark is written in Scala***, which runs on the Java Virtual Machine (JVM).
  - This makes JDBC the preferred method for connecting to data whenever possible.
  - Hadoop, Hive, and MySQL all run on Java and easily interface with Spark clusters.
  - To define database connection criteria, you need the ***hostname, port, and database name***.

- **Question:** What is JDBC?
  - **Answer:** JDBC stands for Java Database Connectivity – and is a Java API for connecting to databases such as MySQL, Hive, etc.
- **Question:** How does Spark read from a JDBC connection by default?
  - **Answer:** With a ***serial*** read. With additional specifications, Spark conducts a faster, parallel read. Parallel reads take full advantage of Spark's distributed architecture.
- **Question:** What is the general design pattern for connecting to your data?
  - **Answer:** The general design patter is as follows:
    - Define the connection point
    - Define connection parameters such as access credentials
    - Add necessary options such as for headers or parallelization

Schemas

- **Schemas** are at the heart of data structures in Spark.
  - **A schema describes the structure of your data by naming columns and declaring the type of data in that column.**
  - Rigorously enforcing schemas leads to significant performance optimizations and reliability of code.
- Why is open source Spark so fast, and why is Databricks Runtime even faster?
  - First, Spark runs first in **memory** rather than reading and writing to disk.
  - Second, using **DataFrames allows Spark to optimize the execution** of your queries because it knows what your data looks like.
- **Schemas with Semi-Structured JSON Data**
  - **Tabular data**, such as that found in CSV files, has a formal structure where each observation has a value (even if it's a NULL).
  - **Semi-structured data** does not need to conform to a formal data model. Instead, a given feature may appear zero, once, or many times for a given observation.
  - Semi-structured data storage works well with **hierarchical data** and with schemas that may **evolve over time**. One of the most common forms of semi-structured data is JSON data, which consists of **attribute-value** pairs.
- **User-Defined Schemas**
  - Challenges with inferred schemas include:
    - Schema inference means Spark scans all of your data, creating an extra job, which can affect performance
    - Consider providing alternative data types (for example, change a Long to an Integer)
    - Consider throwing out certain fields in the data, to read only the data of interest
  - To define schemas, build a StructType composed of StructFields.

- **Print the first few lines of a file:**
  - **%fs head** /mnt/training/zips.json --maxBytes=492

05-Applying-Schemas-to-JSON-Data

Primitive and Nonprimitive Types

The Spark `types` package provides the building blocks for constructing schemas.

A primitive type contains the data itself. The most common primitive types include:

Numeric	General	Time
FloatType	StringType	TimestampType
IntegerType	BooleanType	DateType
DoubleType	NullType	
LongType		
ShortType		

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

zipsSchema2 = StructType([
  StructField("city", StringType(), True),
  StructField("pop", IntegerType(), True)
])
```

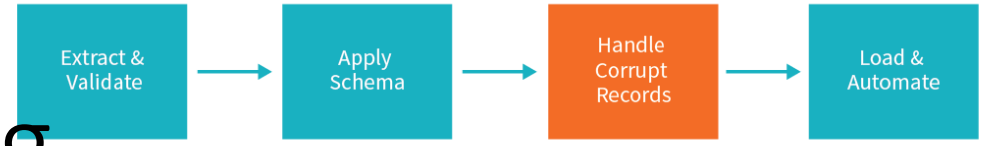
- Non-primitive types or composite types.
  - Technically, non-primitive types contain **references to memory locations** and not the data itself.
  - Nonprimitive types are the **composite of several primitive types** such as an Array of the primitive type Integer.
  - The two most common composite types are **ArrayType** and **MapType**.
    - These types allow for a given field to contain an arbitrary number of elements in either an Array/List or Map/Dictionary form.

- To time an inline command: **%timeit *command***
- See the schema fields of a datafrmae: **[i for i in df.schema]**

Review

- Question: What are two ways to attain a schema from data?
  - Answer:
    - Allow Spark to infer a schema from data (recommended first step) or
    - Provide a user defined schema
- Question: Why should you define your own schema?
  - Answer: Benefits of user defined schemas include:
    - Avoiding the extra scan of your data needed to infer the schema
    - Providing alternative data types
    - Parsing only the fields you need
- Question: Why is JSON a common format in big data pipelines?
  - Answer: Semi-structured data works well with hierarchical data and where schemas need to evolve over time. It also easily contains composite data types such as arrays and maps.
- Question: By default, how are corrupt records dealt with using spark.read.json()?
  - Answer: They appear in a column called _corrupt_record. These are the records that Spark can't read (e.g. when characters are missing from a JSON string).

06-Corrupt-Record-Handling



- ETL pipelines need robust solutions to handle corrupt data. Corrupt data includes:
  - Missing information
  - Incomplete information
  - Schema mismatch
  - Differing formats or data types
  - User errors when writing data producers

- There are three different options:
  - PERMISSIVE, DROPMALFORMED, FAILFAST

- Databricks Runtime has a built-in feature that saves corrupt records to a given end point. To use this, set the badRecordsPath.
  - This is a preferred design pattern since it persists the corrupt records for later analysis even after the cluster shuts down.

ParseMode	Behavior
PERMISSIVE	Includes corrupt records in a "_corrupt_record" column (by default)
DROPMALFORMED	Ignores all corrupted records
FAILFAST	Throws an exception when it meets corrupted records

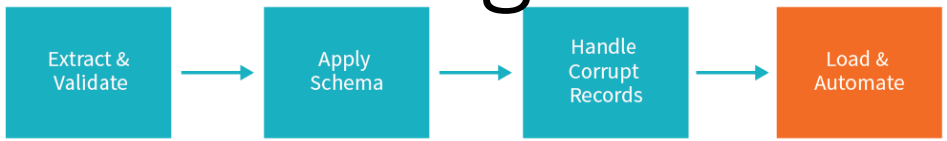
```
data = """{"a": 1, "b":2, "c":3}|{"a": 1, "b":2, "c":3}|{"a": 1, "b, "c":10}""".split('|')

corruptDF = (spark.read
  .option("mode", "PERMISSIVE")
  .option("columnNameOfCorruptRecord", "_corrupt_record")
  .json(sc.parallelize(data))
)
display(corruptDF)
```

```
data = """{"a": 1, "b":2, "c":3}|{"a": 1, "b":2, "c":3}|{"a": 1, "b, "c":10}""".split('|')

df = (spark.read
      .option("badRecordsPath", "/tmp/corruptRecords")
      .json(sc.parallelize(data))
)

c = spark.read.json("/tmp/corruptRecords/*/*/*")
display(c)
```

Review

- Question: By default, how are corrupt records dealt with using spark.read.json()?
  - Answer: They appear in a column called ”_corrupt_record”.
- Question: How can a query persist corrupt records in separate destination?
  - Answer: The Databricks feature badRecordsPath allows a query to save corrupt records to a given end point for the pipeline engineer to investigate corruption issues.
- **Q:** Where can I get more information on dealing with corrupt records?
  - **A:** Check out the Spark Summit talk on Exceptions are the Norm: Dealing with Bad Actors in ETL

07-Loading-Data-and-Productionalizing



- Productionalization:
  - **Parquet is a best practice for loading data from ETL jobs**
  - **Scheduling jobs**
  - Incorporating notebooks into production workflows
- BLOB stores like S3 or Azure Blob are the data storage option of choice on Databricks.
- Parquet is the storage format of choice.
  - When writing data to DBFS, the best practice is to use Parquet
  - Apache Parquet is a highly efficient, column-oriented data format that shows massive performance increases over other options such as CSV.
  - Parquet compresses data repeated in each column and preserves the schema from a write.
- Use camelCase best practice for spark dataframe columns

```
cols = crimeDF.columns
titleCols = [''.join(j for j in i.title() if not j.isspace()) for i in cols]
camelCols = [column[0].lower()+column[1:] for column in titleCols]

crimeRenamedColsDF = crimeDF.toDF(*camelCols)
display(crimeRenamedColsDF)
```

Write in Parquet

- Write to Parquet by calling the following method on a DataFrame: .write.parquet("mnt/.parquet").
  - Specify the write mode (for example, overwrite or append) using .mode().
  - Write to /tmp/, a directory backed by the Azure Blob or S3 available to all Datatabricks clusters.
  - If your /tmp/ directory is full, clear contents using **%fs rm -r /tmp/**.
  - Parquet is distributed so each partition of data in the cluster writes to its own "part“
  - Use the *repartition* DataFrame method to repartition the data to limit the number of separate parts.
- **Automate by Scheduling a**

```
crimeRenamedColsDF.repartition(1).write.mode("overwrite").parquet("/tmp/crimeRepartitioned.parquet")
```

  - Scheduling a job allows you to perform a batch process at a regular interval.
  - Schedule email updates for successful completion and error logs.
  - Jobs is not available for Community Edition users.

Review

- **Question:** What is the recommended storage format to use with Spark?
  - **Answer:** Apache Parquet is a highly optimized solution for data storage and is the recommended option for storage where possible.
  - In addition to offering benefits like compression, it's distributed, so a given partition of data writes to its own file, enabling parallel reads and writes.
  - Formats like CSV are prone to corruption since a single missing comma could corrupt the data. Also, the data cannot be parallelized.
- **Question:** How do you schedule a regularly occurring task in Databricks?
  - **Answer:** The Jobs tab of a Databricks notebook or the new Jobs API allows for job automation.
- **Q:** Where can I get more information on scheduling jobs on Databricks?
  - **A:** Check out the Databricks documentation on Scheduling Jobs on Databricks
- **Q:** How can I schedule complex jobs, such as those involving dependencies between jobs?
  - **A:** There are two options for complex jobs. The easiest solution is Notebook Workflows, which involves using one notebook that triggers the execution of other notebooks. For more complexity, Databricks integrates with the open source workflow scheduler Apache Airflow.
- **Q:** How do I perform spark-submit jobs?
  - **A:** Spark-submit is the process for running Spark jobs in the open source implementation of Spark. Jobs and the jobs API are a robust option offered in the Databricks environment. You can also launch spark-submit jobs through the jobs UI as well.
- **Extra Practice:** Apply what you learned in this module by completing the optional Parsing Nested Data exercise.

Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

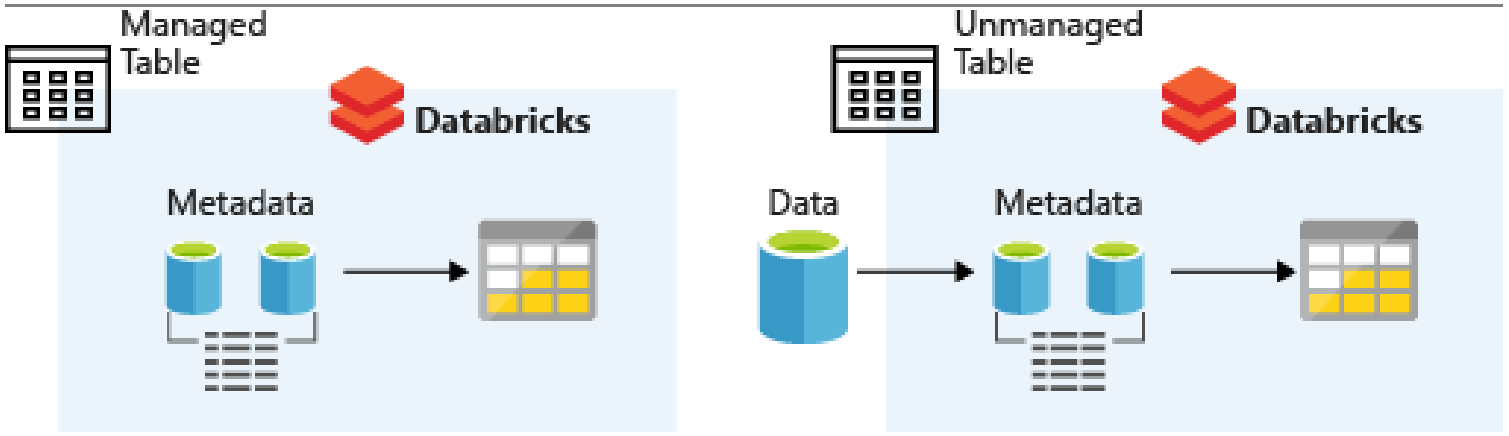
Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

Perform advanced data transformation in Azure Databricks

- The goal of ETL is to transform raw data to populate a data model.
  - The most common models are:
    - relational models
    - snowflake (or star) schemas
    - query-first modeling
- Common transformations include:
  - Normalizing values
  - Imputing null or missing data
  - Deduplicating data
  - Performing database rollups
  - Exploding arrays
  - Pivoting DataFrames
- Advanced ETL processes include:
  - data transformation by using custom and advanced user-defined functions (UDFs),
  - managing complex tables,
  - and loading data into multiple databases simultaneously

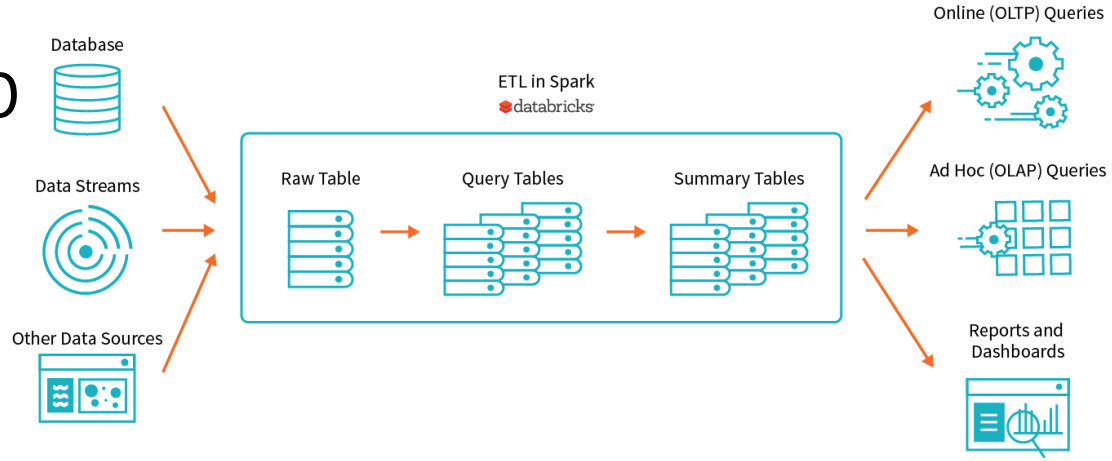
Tables



- **Table management**
  - **Managed tables:**
    - A **managed table** is a table that manages both the actual data and the metadata.
    - In this case, a DROP TABLE command removes both the metadata for the table and the data itself.
  - **Unmanaged tables:**
    - Unmanaged tables manage the metadata from a table, while the actual data is managed separately and is often backed by a blob store such as Azure Blob storage.
    - Dropping an unmanaged table drops only the metadata that is associated with the table while the data remains in place.

- **Import Advanced-ETL Notebooks**: https://github.com/MicrosoftDocs/mslearn-perform-advanced-data-transformation-in-azure-databricks/blob/master/DBC/05.2-Advanced-ETL.dbc?raw=true

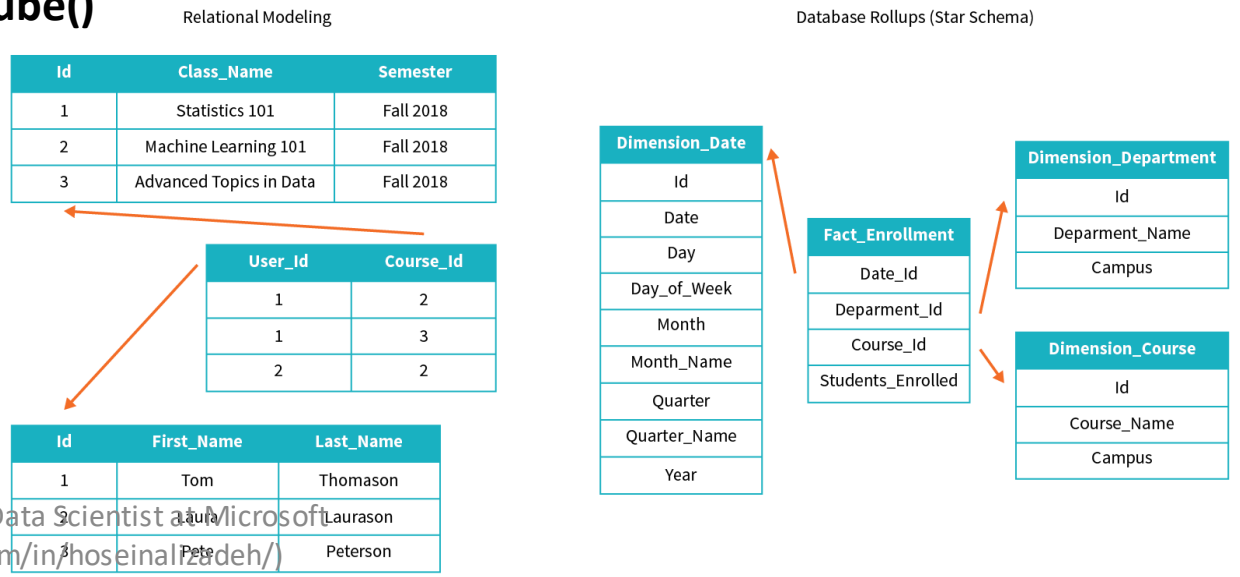
01-Course-Overview-and-Setup



- **Raw, Query, and Summary Tables**
  - **Raw tables:** This refers to data that arrives in the pipeline, conforms to a schema, and does not include any sort of parsing or aggregation.
  - **Query tables:**
    - Raw tables are then parsed into query-ready tables, known as **query tables.**
    - Query tables might populate a relational data model for ad hoc (OLAP) or online (OLTP) queries, but generally do not include any sort of aggregation such as daily averages.
    - Put another way, query tables are cleaned and filtered data.
  - **Summary tables:**
    - They are business level aggregates often used for reporting and dashboarding.
    - This includes aggregations such as daily active website visitors.

02-Common-Transformations

- The goal of transformations in ETL is to transform raw data in order to populate a data model.
- The most common models are **relational models** and **snowflake (or star) schemas**
- Built-in functions offer a range of performant options to manipulate data. This includes options familiar to:
  - SQL users such as .select() and .groupBy()
  - Python, Scala and R users such as max() and sum()
  - **Data warehousing options such as rollup() and cube()**



Create Spark Dataframe

- Create dataframe including null values:

```
corruptDF = spark.createDataFrame([
  (11, 66, 5),
  (12, 68, None),
  (1, None, 6),
  (2, 72, 7)],
  ["hour", "temperature", "wind"]
)

display(corruptDF)
```

▸ (4) Spark Jobs

▸ corruptDF: pyspark.sql.dataframe.DataFrame = [hour: long, temperature: long ... 1 more fields]

	hour	temperature	wind
1	11	66	5
2	12	68	null
3	1	null	6
4	2	72	7

Showing all 4 rows.

Transform

- Normalizing the data:

Retrieve max value:

Normalizing data:

```
from pyspark.sql.functions import col, max, min

colMin = integerDF.select(min("id")).first()[0]
colMax = integerDF.select(max("id")).first()[0]

normalizedIntegerDF = (integerDF
  .withColumn("normalizedValue", (col("id") - colMin) / (colMax - colMin) )
)

display(normalizedIntegerDF)
```

- Strategies for dealing with Null values include:
  - **Dropping these records**:
    - Works when you do not need to use the information for downstream workloads
    - Drop any records that have null value: **df = df.dropna("any") ⇔ df = df.na.drop()**
    - Drop if null values in specific columns: **df = df.filter("(columnA is not null) AND (columnB is not null)")**
  - **Adding a placeholder (e.g. -1)**:
    - Allows you to see missing data later on without violating a schema: **df = df.na.fill(-1)**
  - **Basic imputing**:
    - Allows you to have a "best guess" of what the data could have been, often by using the mean of non-missing data
      - hourMean = corruptDF.na.drop().select(F.mean(F.col('hour'))).first()[0]
      - windMean = corruptDF.na.drop().select(F.mean(F.col('wind'))).first()[0]
      - corruptImputedDF = corruptDF.na.fill({'hour':hourMean, 'wind':windMean})
  - **Advanced imputing**:
    - Determines the "best guess" of what data should be using more advanced strategies such as clustering machine learning algorithms or oversampling techniques

Transform

- **Duplicate data** comes in many forms:
  - The simple case involves records that are *complete duplicates* of another record.
    - **df.dropDuplicates()**
  - The more complex cases involve duplicates that are not complete matches, such as
    - matches on one or two columns => **df.dropDuplicates(["id","favorite_color"])**
    - "fuzzy" matches that account for formatting differences or other non-exact matches.
- **Other Helpful Data Manipulation Functions**

Function	Use
`explode()`	Returns a new row for each element in the given array or map
`pivot()`	Pivots a column of the current DataFrame and perform the specified aggregation
`cube()`	Create a multi-dimensional cube for the current DataFrame using the specified columns, so we can run aggregation on them
`rollup()`	Create a multi-dimensional rollup for the current DataFrame using the specified columns, so we can run aggregation on them

Pivot & Rollup

	country	group	province	total
1	Iran	men	north	5
2	Iran	women	north	4
3	Iran	kids	north	8
4	AU	men	north	11
5	AU	women	north	9
6	AU	kids	north	6
7	Iran	men	south	5

df.groupBy('country').pivot('group').sum().show()

```
+-------+----+---+-----+
|country|kids|men|women|
+-------+----+---+-----+
|     AU|  12| 22|   18|
|   Iran|  16| 10|    8|
+-------+----+---+-----+
```

```
(df.rollup('country', 'group')
 .agg(F.sum('total').alias('new_total'))
 .orderBy('new_total', ascending=0)
 .show())
```

```
+-------+-----+---------+
|country|group|new_total|
+-------+-----+---------+
|   null| null|       86|
|     AU| null|       52|
|   Iran| null|       34|
|     AU|  men|       22|
|     AU|women|       18|
|   Iran| kids|       16|
|     AU| kids|       12|
|   Iran|  men|       10|
|   Iran|women|        8|
+-------+-----+---------+
```

Note: null values in Rollup means all values of that column are considered.

Rollup is very similar to groupBy.

- groupBy does the job over the values of the column
- rollup does the job on values as well as presenting the ground aggregation over all values within that column representing with null value.

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Regular Expression

- Remove '-' from values of column 'ssn'.
  - eg '946-12-3406' -> '946123406'
  - **.withColumn('ssnNums', F.regexp_replace(F.col('ssn'), '-', ''))**

```
df4 = (df3
 .withColumn('transformedName', F.regexp_extract(F.col('name'), 'Alizadeh', 0))
 .withColumn('replacedName', F.regexp_replace(F.col('name'), ' ', ', '))
)
df4.show()
```

▸ (5) Spark Jobs

▸ df4: pyspark.sql.dataframe.DataFrame = [name: string, id: long ... 4 more fields]

```
+---------------+---+---------+--------+--------------+----------------+
|           name| id|firstName|lastName|transformedName|    replacedName|
+---------------+---+---------+--------+--------------+----------------+
|Hosein Alizadeh|  1|   Hosein|Alizadeh|       Alizadeh|Hosein, Alizadeh|
|    Megan Ganji|  2|    Megan|   Ganji|               |    Megan, Ganji|
|Ariana Alizadeh|  3|   Ariana|Alizadeh|       Alizadeh|Ariana, Alizadeh|
|    Nane Vahedi|  4|     Nane|  Vahedi|               |    Nane, Vahedi|
|  Hadi Alizadeh|  5|      NaN|     NaN|       Alizadeh|  Hadi, Alizadeh|
+---------------+---+---------+--------+--------------+----------------+
```

Replacing all colors with 'COLOR'

```
from pyspark.sql.functions import regexp_replace
regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
df.select(
  regexp_replace(col("Description"), regex_string, "COLOR").alias("color_clean"),
  col("Description")).show(2)
```

```
+--------------------+--------------------+
|         color_clean|         Description|
+--------------------+--------------------+
|COLOR HANGING HEA...|WHITE HANGING HEA...|
| COLOR METAL LANTERN| WHITE METAL LANTERN|
+--------------------+--------------------+
```

Extracting a pattern string from a column

```
from pyspark.sql.functions import regexp_extract
extract_str = "(BLACK|WHITE|RED|GREEN|BLUE)"
df.select(
     regexp_extract(col("Description"), extract_str, 1).alias("color_clean"),
     col("Description")).show(2)
```

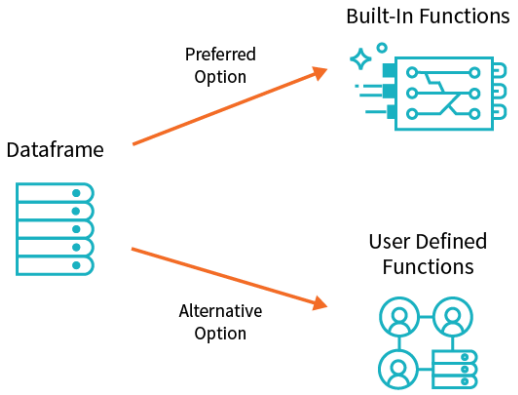
```
+-------------+--------------------+
|  color_clean|         Description|
+-------------+--------------------+
|        WHITE|WHITE HANGING HEA...|
|        WHITE| WHITE METAL LANTERN|
+-------------+--------------------+
```

Review

- **Question:** What built-in functions are available in Spark?
  - **Answer:** Built-in functions include SQL functions, common programming language primitives, and data warehousing specific functions. See the Spark API Docs for more details. (Python or Scala).
- **Question:** What's the best way to handle null values?
  - **Answer:** The answer depends largely on what you hope to do with your data moving forward.
  - You can drop null values or impute them with several different techniques. For instance, clustering your data to fill null values with the values of nearby neighbors often gives more insight to machine learning models than using a simple mean.
- **Question:** What are potential challenges of deduplicating data and imputing null values?
  - **Answer:** Challenges include knowing which is the correct record to keep and how to define logic that applies to the root cause of your situation. This decision-making process depends largely on how removing or imputing data will affect downstream operations like database queries and machine learning workloads.
  - Knowing the end application of the data helps determine the best strategy to use.
- **Q:** How can I do ACID transactions with Spark?
  - **A:** ACID compliance refers to a set of properties of database transactions that guarantee the validity of your data. Databricks Delta is an ACID compliant solution to transactionality with Spark workloads.
- **Q:** How can I handle more complex conditional logic in Spark?
  - **A:** You can handle more complex if/then conditional logic using the `when()` function and its `.otherwise()` method.
- **Q:** How can I handle data warehousing functions like rollups?
  - **A:** Spark allows for **rollups and cubes, which are common in star schemas**, using the `rollup()` and `cube()` functions.

03-User-Defined-Functions

- UDFs provide custom, generalizable code that you can apply to ETL workloads when the built-in functions in Spark aren't sufficient.
- Why should we use spark built-in functions as much as possible:
  - First, *built-in functions are finely tuned* so they run faster than less efficient code provided by the user.
  - Secondly, the Catalyst Optimizer (Spark's optimization engine) knows the objective of built-in functions so it can *optimize the execution of your code by changing the execution order of your tasks.*
- User Defined Functions (UDFs) are useful:
  - when you need to define logic specific to your use case and
  - when you need to encapsulate that solution for reuse.
- UDFs are generally more performant in Scala than Python.
  - Because Spark must spin up a Python interpreter on every executor to run the function, which causes a substantial performance bottleneck.
  - **UDFs take a function or lambda and make it available for Spark to use**.
- Register the function as a UDF by designating the following:
  - A name for access in Python (manualSplitPythonUDF)
  - A name for access in SQL (manualSplitSQLUDF)
  - The function itself (manual_split)
  - The return type for the function (StringType)
    - **manualSplitPythonUDF = spark.udf.register("manualSplitSQLUDF", manual_split, StringType())**



UDFs

- Create a dataframe of 100k values with a string to index using hash function sha1:

```
df = (spark.range(1, 100000)
      .withColumn('random', rand(123).cast(T.StringType()))
      .withColumn('hash', F.sha1(F.col('random')))
     )
```

- Lambda is an inline method of defining functions:

```
lambda x: x + 1
```

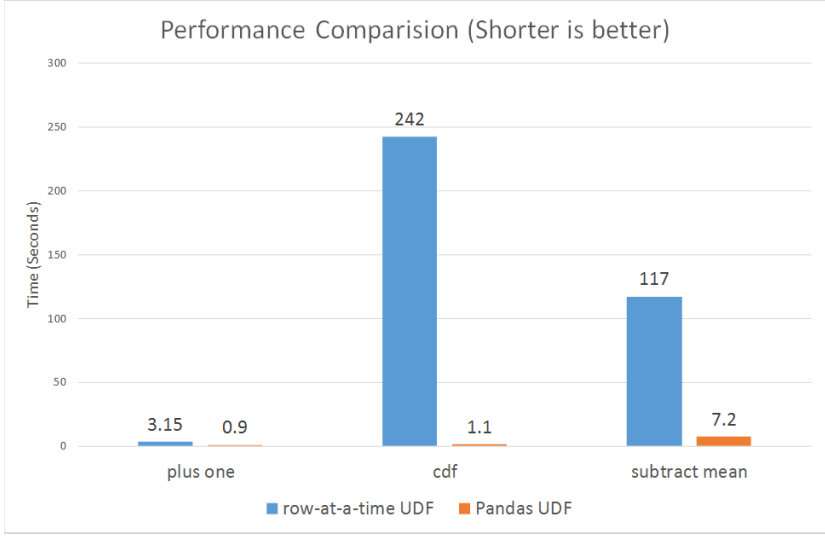
- Define a lambda function and register it as UDF all in one line:

```
plusOneUDF = spark.udf.register('plusOneUDF', lambda x:x+1, T.FloatType())

df2 = randomFloatsDF.withColumn('random_float_plusOne', plusOneUDF(F.col('random_float')))
display(df2)
```

Review

- **Question:** What are the performance tradeoffs between UDFs and built-in functions? When should I use each?
  - **Answer:** Built-in functions are normally faster than UDFs and should be used when possible. UDFs should be used when specific use cases arise that aren't addressed by built-in functions.
- **Question:** How can I use UDFs?
  - **Answer:** UDFs can be used in any Spark API. They can be registered for use in SQL and can otherwise be used in Scala, Python, R, and Java.
- **Question:** Why are built-in functions faster?
  - **Answer:** Reasons include:
    - The **catalyst optimizer knows** how to optimize built-in functions
    - They are **written in highly optimized Scala**
    - There is **no serialization cost at the time of running** a built-in function
- **Question:** Can UDFs have multiple column inputs and outputs?
  - **Answer:** Yes, UDFs can have **multiple column inputs** and **complex outputs**. This is covered in the following lesson.
- **Q:** Where can I find out more about UDFs?
  - **A:** Take a look at the Databricks documentation for more details

04-Advanced-UDFs



- Advanced UDFs
  - UDFs can take multiple column inputs.
  - UDFs with Complex Output
    - While UDFs **CANNOT** return multiple columns, they can return complex, named types that are easily accessible. This approach is especially helpful in ETL workloads that need to clean complex and challenging data structures.
  - Vectorized pandas UDFs
    - Vectorized UDFs in Python Pandas called Pandas UDFs.
    - This alleviates some of the serialization and invocation overhead of conventional Python UDFs.
    - While there are many types of these UDFs, this walkthrough focuses on *scalar UDFs*.
      - This is an ideal solution for Data Scientists needing performant UDFs written in Python.
    - available in Spark 2.3.
    - Use the decorator syntax to designate a Pandas UDF. The input and outputs are both Pandas series of doubles.

```
def manual_add(x, y): return x + y
manualAddPythonUDF = spark.udf.register("manualAddSQLUDF", manual_add, IntegerType())
df.select("*", manualAddPythonUDF("col1", "col2").alias("sum"))
```

```
mathOperationsSchema = StructType([
  StructField("sum", FloatType(), True),
  StructField("multiplication", FloatType(), True),
  StructField("division", FloatType(), True)
])
def manual_math(x, y): return (float(x + y), float(x * y), x / float(y))
manualMathPythonUDF = spark.udf.register("manualMathSQLUDF", manual_math, mathOperationsSchema)
display(integerDF.select("*", manualMathPythonUDF("col1", "col2").alias("sum")))
```

```
%python
from pyspark.sql.functions import pandas_udf, PandasUDFType
@pandas_udf('double', PandasUDFType.SCALAR)
def pandas_plus_one(v): return v + 1
display(df.withColumn('id_transformed', pandas_plus_one("id")))
```

Review

- Question: How do UDFs handle multiple column inputs and complex outputs?
  - Answer: UDFs allow for multiple column inputs. Complex outputs can be designated with the use of a defined schema encapsulate in a StructType() or a Scala case class.

- Question: How can I do vectorized UDFs in Python and are they as performant as built-in functions?
  - Answer: Spark 2.3 includes the use of vectorized UDFs using Pandas syntax. Even though they are vectorized, these UDFs will not be as performant built-in functions, though they will be more performant than non-vectorized Python UDFs.

- **Q:** Where can I find out more about UDFs?
  - **A:** Take a look at the Databricks documentation for more details
- **Q:** Where can I find out more about vectorized UDFs in Python?
  - **A:** Take a look at the Databricks blog for more details
- **Q:** Where can I find out more about User Defined Agregate Funcations?
  - **A:** Take a look at the Databricks documentation for more details

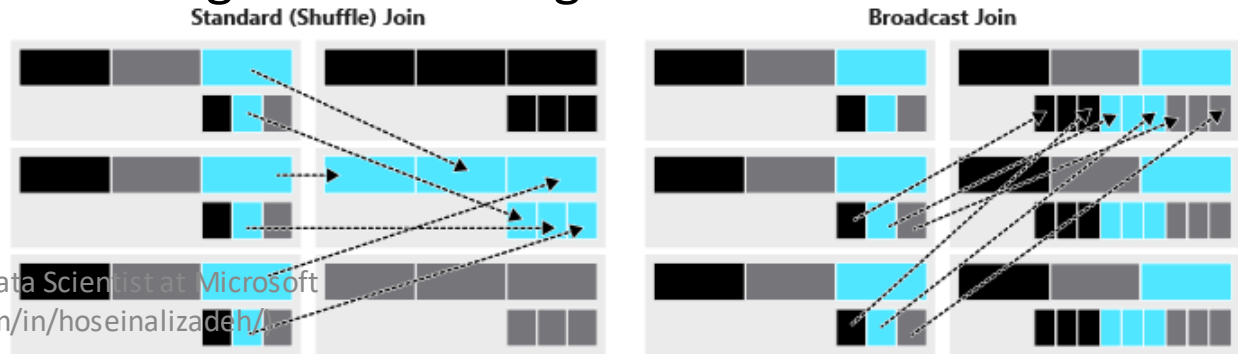
05-Joins-and-Lookup-Tables

- **Shuffle and Broadcast Joins**
  - Traditional databases join tables by pairing values on a given column which is computationally expensive (because of row-wise comparison).
  - **A standard (or shuffle) join** moves all the data on the cluster for each table to a given node on the cluster which is expensive. (because of row-wise comparisons + data transfer across a network).
  - **Broadcast join** remedies this situation if one DataFrame is sufficiently small. A broadcast join duplicates the smaller of the two DataFrames on each node of the cluster, avoiding the cost of shuffling the bigger DataFrame.
    - There are two ways of telling Spark to explicitly broadcast tables.
      - The first is to change the Spark configuration, which affects all operations.
      - The second is to declare it using the `broadcast()` function in the `functions` package
        - from pyspark.sql.functions import broadcast
        - **pageviewsDF.join(broadcast(labelsDF), "dow")**.explain()
  - By default Spark uses Broadcast join if the smaller table is less than **10MB**.
    - **spark.conf.get("spark.sql.autoBroadcastJoinThreshold")**
  - To disables "autoBrocdcastJoinThreshold", set it to -1:
    - **spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)**

Joins

- **Joins and lookup tables**
  - **Standard Join**
    - A standard (or shuffle) join moves all the data on the cluster for each table to a specific node on the cluster.
    - A standard join uses considerable computation to perform row-wise comparisons and to transfer data across a network, which is often the biggest performance bottleneck of distributed systems.
  - **Broadcast join**
    - A broadcast join fixes this situation when one dataframe is sufficiently small.
    - A broadcast join duplicates the smaller dataframe on each node of the cluster, which avoids the cost of shuffling the bigger dataframe.

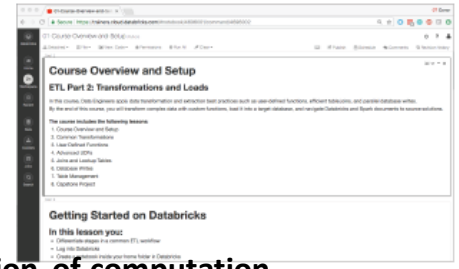
- look at the physical plan used to return a query:
  - **df.explain()**
- look at the broadcast threshold by accessing the configuration settings:



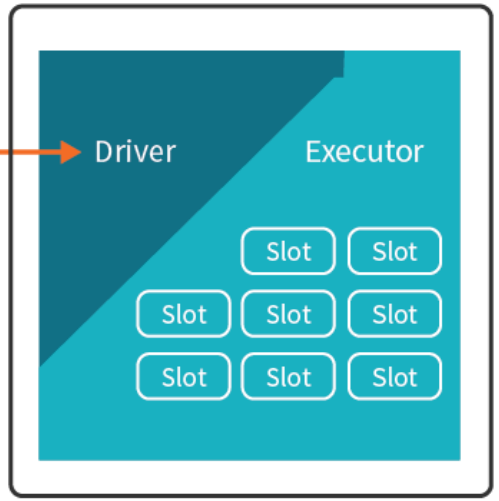
Review

- **Question:** Why are joins expensive operations?
  - **Answer:** Joins perform many row-wise comparisons, making the cost associated with joining tables grow with the size of the data in the tables.
- **Question:** What is the difference between a shuffle and broadcast join? How does Spark manage these differences?
  - **Answer:** A shuffle join shuffles data between nodes in a cluster. By contrast, a broadcast join moves the smaller of two DataFrames to where the larger DataFrame sits, minimizing the overall data transfer.
  - By default, Spark performs a broadcast join if the total number of records is below 10MB. The threshold can be manually specified, or you can manually specify that a broadcast join should take place. Since the automatic determination of whether a shuffle join should take place is by number of records, this could mean that wide data would take up significantly more space per record and should therefore be specified manually.
- **Question:** What is a lookup table?
  - **Answer:** A lookup table is small table often used for referencing commonly used data such as mapping cities to countries.
- **Q:** Where can I get more information on optimizing table joins where data skew is an issue?
  - **A:** Check out the Databricks documentation on Skew Join Optimization

06-Database-Writes



Your Notebook



- A **partition refers to the distribution of data** while a **slot/core/threat refers to the distribution of computation**.
  - A partition is a portion of your total data set.
  - A slot/core is a resource available for the execution of computation in parallel.
  - The **number of partitions should be a multiple of the number of cores**.
    - For instance, with 5 partitions and 8 slots, 3 of the slots will be underutilized.
    - With 9 partitions and 8 slots, a job will take twice as long as it waits for the extra partition to finish.
- **Managing Partitions**
  - In the context of JDBC database writes, **the number of partitions determine the number of connections used to push data through the JDBC API.**
  - There are two ways to control this parallelism:
  - View the number of partitions by:
    - changing the DataFrame into an RDD
    - `How many partitions? Use .getNumPartitions()`
      - partitions = wikiDF.rdd.getNumPartitions()
    - To **increase the number of partitions to 16, use repartition()**
      - repartitionedWikiDF = wikiDF.repartition(16)
    - To **reduce the number of partitions to 2, use coalesce()**
      - coalescedWikiDF = repartitionedWikiDF.coalesce(2)
    - **Spark uses a default value of 200 partitions**.
      - Get and set any number of different configuration settings in this manner: **spark.conf.get("spark.sql.shuffle.partitions")**
      - Adjust the number of partitions with: **spark.conf.set("spark.sql.shuffle.partitions", "8")**
        - **Note: This changes the number of partitions after a shuffle operation such as orderBy().**
  - When writing to a database, the number of active connections to the database is determined by the ***number of partitions of the DataFrame***.
    - Repartition and write: wikiDF.repartition(10).write.mode("overwrite").parquet("/tmp/wiki.parquet")

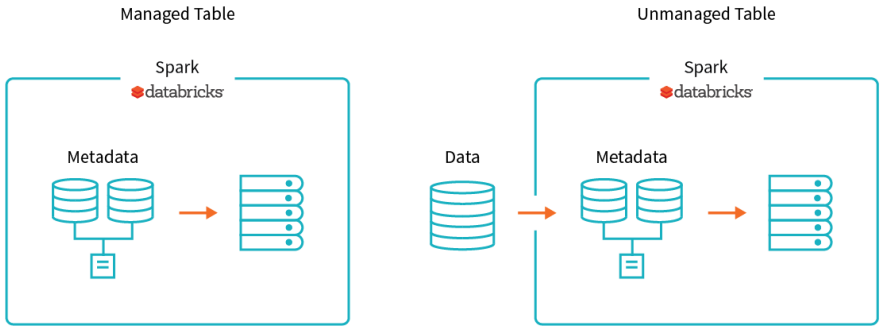
Function	Transformation Type	Use	Evenly distributes data across partitions?
`.coalesce(n)`	narrow (does not shuffle data)	reduce the number of partitions	no
`.repartition(n)`	wide (includes a shuffle operation)	increase the number of partitions	yes

- "***Upsert***" inserts a record into a database if it doesn't already exist and updates the existing record if it does.
  - **Upserts are not supported in core Spark.** You can only append or overwrite.
  - Databricks offers a data management system called **Databricks Delta that does allow for upserts** and other transactional functionality. See the Databricks Delta docs for more information.

Review

- Question: How do you determine the number of connections to the database you write to?
  - Answer: Spark makes one connection for each partition in your data. Increasing the number of partitions increases the database connections.
- Question: How do you increase and decrease the number of partitions in your data?
  - .repartitions(n) increases the number of partitions in your data. It can also decrease the number of partitions, but since this is a wide operation it should be used sparingly.
  - .coalesce(n) decreases the number of partitions in your data.
- Question: How can you change the default number of partitions?
  - Answer: Changing the configuration parameter spark.sql.shuffle.partitions will alter the default number of partitions (in spark.conf.set command).
- **Q:** Where can I find more information on reading the Spark UI?
  - **A:** Check out the Databricks blog on Understanding your Spark Application through Visualization

07-Table-Management



- **Optimization of Data Storage with Managed and Unmanaged Tables**
  - A **managed (Internal) table** is a table that manages both the data itself as well as the metadata.
    - In this case, a `DROP TABLE` command removes both the metadata for the table as well as the data.
    - Managed tables allow access to data using the ***Spark SQL API***.
      - Write to a managed table:
        - **df.write.mode("overwrite").saveAsTable("myTable")**
  - **Unmanaged (External) tables** manage the metadata from a table such as the schema and data location, but the data itself sits in a different location, often backed by a blob store like the Azure Blob Storage.
    - Dropping an unmanaged table drops only the metadata associated with the table while the data itself remains in place.
    - Write to an unmanaged table by adding an `.option()` that includes a path.
      - **df.write.mode("overwrite") .option('path', '/tmp/myTableUnmanaged').saveAsTable("myTable")**
  - To describe the contents of the table use either **"describe"** or **"describe extended"**:
    - %sql
    - Describe extended table_name
  - "**External**" table is the same as "**Unmanaged**" table

Review

- **Question:** What happens to the original data when I delete a managed table? What about an unmanaged table?
  - **Answer:** Deleting a managed table deletes both the metadata and the data itself. Deleting an unmanaged table does not delete the original data.
- **Question:** What is a ***metastore***?
  - **Answer:** A metastore is a **repository of metadata** such as the location of where data is and the schema information. A metastore does not include the data itself.
- **Q:** Where can I find out more about connecting to my own metastore?
  - **A:** Take a look at the Databricks documentation for more details

ETL Process Review

- **Extraction:** By using Java Database Connectivity (JDBC), you can virtually connect to any data store, including Azure Blob storage. Databricks supports connections to multiple database types, including:
  - Traditional databases, like PostgreSQL, SQL Server, and MySQL.
  - Message brokers, like Kafka and Kinesis.
  - Distributed databases, like Cassandra and Redshift.
  - Data warehouses, like Hive and Azure Cosmos DB.
  - File types, like CSV, Parquet, and Avro.
- **Data validation:** One aspect of ETL jobs is to validate that the data is what you expect. The data-validation step primarily includes determining:
  - Approximately the expected number of records.
  - Whether the expected fields are present.
  - That there are no unexpected missing values.
- **Transformation:** This step generally includes applying structure and a schema to your data so that you can transform it into the desired format. Schemas can be applied to tabular data, such as data found in CSV files or relational databases, or to semistructured data, such as JSON data.
- **Corrupt record handling:** Handling bad data is also an important part of the entire ETL process. The built-in functions of Databricks allow you to handle corrupt data, such as:
  - Missing and incomplete information.
  - Schema mismatches.
  - Differing formats or data types.
  - User errors when creating data producers.
- **Loading data:** A common and highly effective design pattern in the Databricks and Spark ecosystem involves loading structured data back to the Databricks File System (DBFS) as a Parquet file. Databricks also allows you to put code into production by scheduling notebooks for regular execution.

Write a UDF to Parse and Clean URLs

```
def getDomain(URL):
  import re
  pattern = re.compile(r"https?://(www\.)?([^/#?]+).*$")
  match = pattern.search(URL)
  return match.group(2)

URL = "https://www.databricks.com/"
print("The string {} matched {}".format(URL, getDomain(URL)))

getDomainUDF = spark.udf.register('getDomainUDF', getDomain,
T.StringType())

Df = tweetDF.select(getDomainUDF(F.col('URL')).alias('parsedURL'))
```

```
urls = [
  "https://www.databricks.com/",
  "https://databricks.com/",
  "https://databricks.com/training-overview/training-self-paced",
  "http://www.databricks.com/",
  "http://databricks.com/",
  "http://databricks.com/training-overview/training-self-paced",
  "http://www.apache.org/",
  "http://spark.apache.org/docs/latest/"
]

for url in urls:
  print(getDomain(url))
```

```
databricks.com
databricks.com
databricks.com
databricks.com
databricks.com
databricks.com
apache.org
spark.apache.org
```

Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

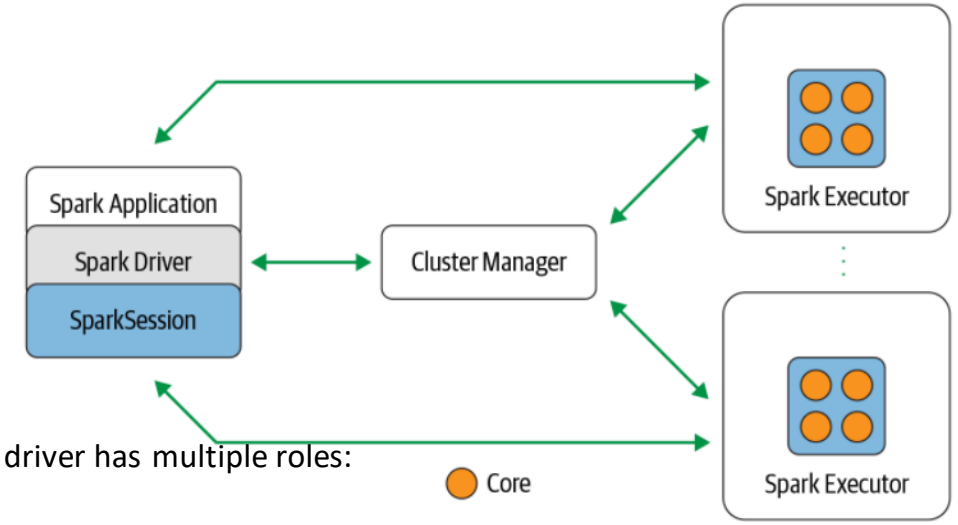
Azure Databricks

- Azure Databricks is a fully-managed version of the open-source Apache Spark analytics and data processing engine. Azure Databricks is an enterprise-grade and secure cloud-based big data and machine learning platform.

- Databricks provides a notebook-oriented Apache Spark as-a-service workspace environment, making it easy to manage clusters and explore data interactively.

- The supported Databricks notebook format is the **DBC** file type.
- **Chrome** and **Firefox** are the recommended browsers to use with Databricks.

Spark's Distributed Execution



- SPARK DRIVER
  - As the part of the Spark application responsible for instantiating a SparkSession, the Spark driver has multiple roles:
    - it communicates with the cluster manager;
    - it requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs);
    - it transforms all the Spark operations into DAG computations, schedules them, and distributes their execution as tasks across the Spark executors.
    - Once the resources are allocated, it communicates directly with the executors.
- CLUSTER MANAGER
  - The cluster manager is responsible for managing and allocating resources for the cluster of nodes on which your Spark application runs. Currently, Spark supports four cluster managers:
    - the built-in standalone cluster manager,
    - Apache Hadoop YARN,
    - Apache Mesos,
    - Kubernetes.
- SPARK EXECUTOR
  - A Spark executor runs on each worker node in the cluster. The executors communicate with the driver program and are responsible for executing tasks on the workers.
  - In most deployment modes, **only a single executor runs per node**.
- DEPLOYMENT MODES
  - An attractive feature of Spark is its support for many deployment modes, enabling Spark to run in different configurations and environments. Because the cluster manager is agnostic to where it runs (as long as it can manage Spark's executors and fulfill resource requests), Spark can be deployed in some of the most popular environments—such as Apache Hadoop YARN and Kubernetes—and can operate in different modes.

Spark Deployment Modes

Table 1-1. Cheat sheet for Spark deployment modes

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

DISTRIBUTED DATA AND PARTITIONS

- Partitioning allows for efficient parallelism. A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth. That is, each executor's core is assigned its own data partition to work on (see Figure 1-6).

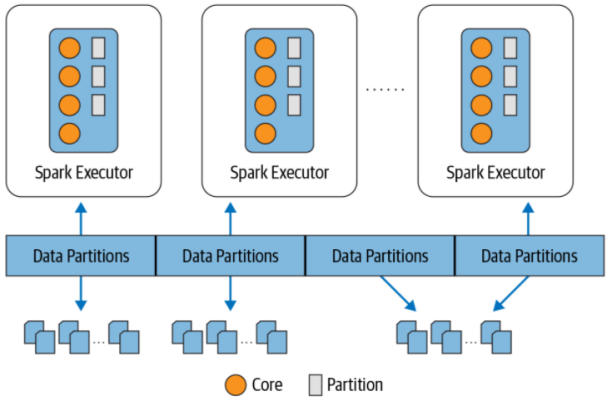


Figure 1-6. Each executor's core gets a partition of data to work on

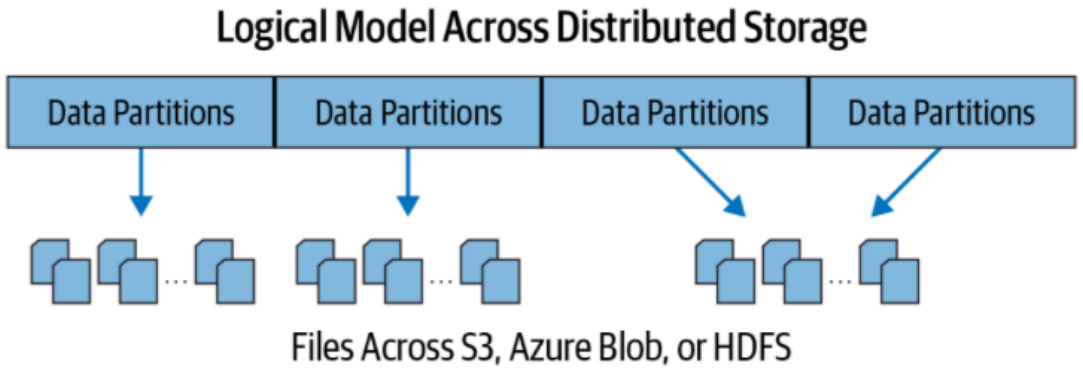
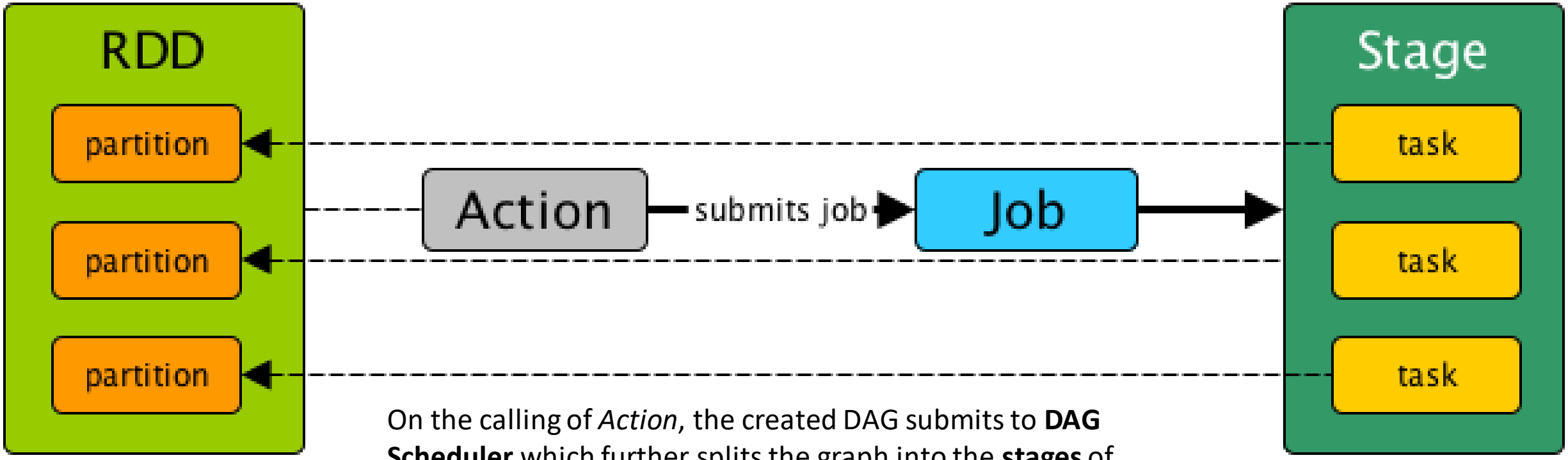


Figure 1-5. Data is distributed across physical machines

Jobs, stage and tasks

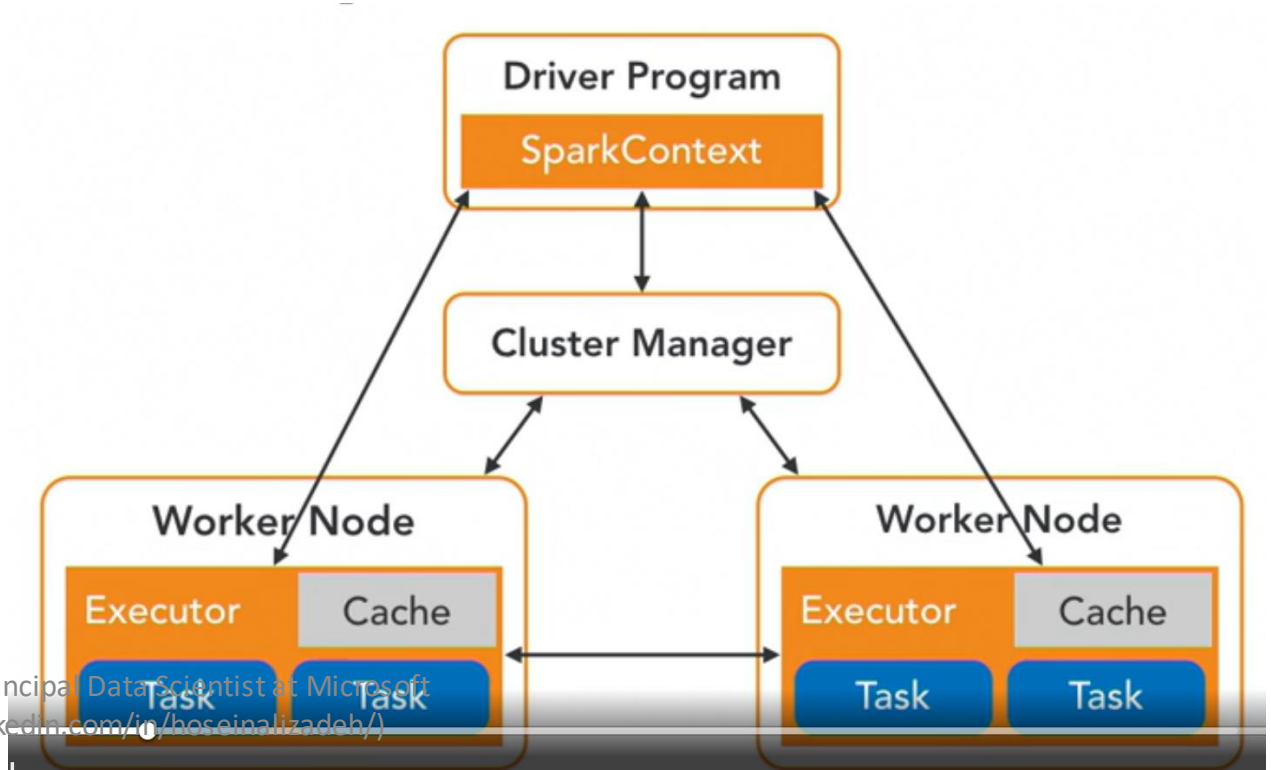
- https://data-flair.training/blogs/spark-stage/



On the calling of *Action*, the created DAG submits to **DAG Scheduler** which further splits the graph into the **stages** of the **task**.
Each **stage** is comprised of **tasks**, based on the partitions of the RDD, which will perform same computation in parallel.

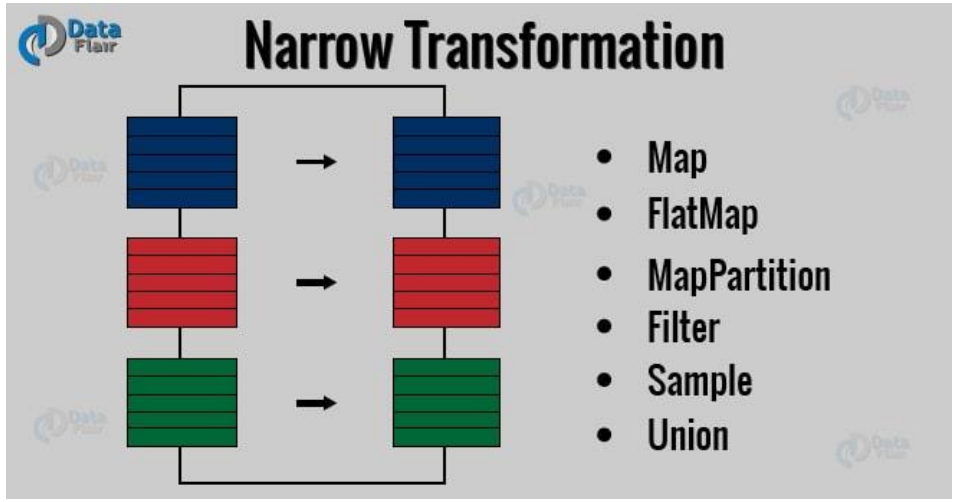
What is Driver and Executor?

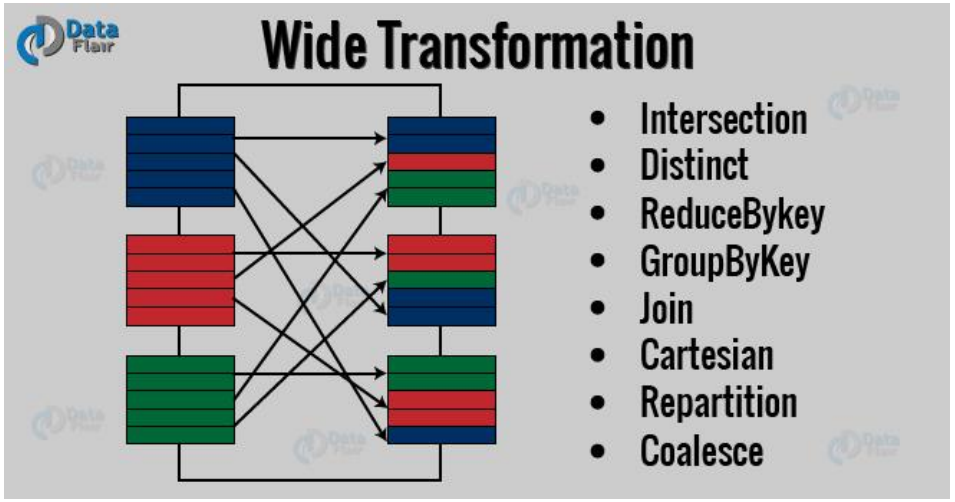
- https://data-flair.training/blogs/spark-executor/



Operation (Transformation & Action)

- https://data-flair.training/blogs/spark-rdd-operations-transformations-actions



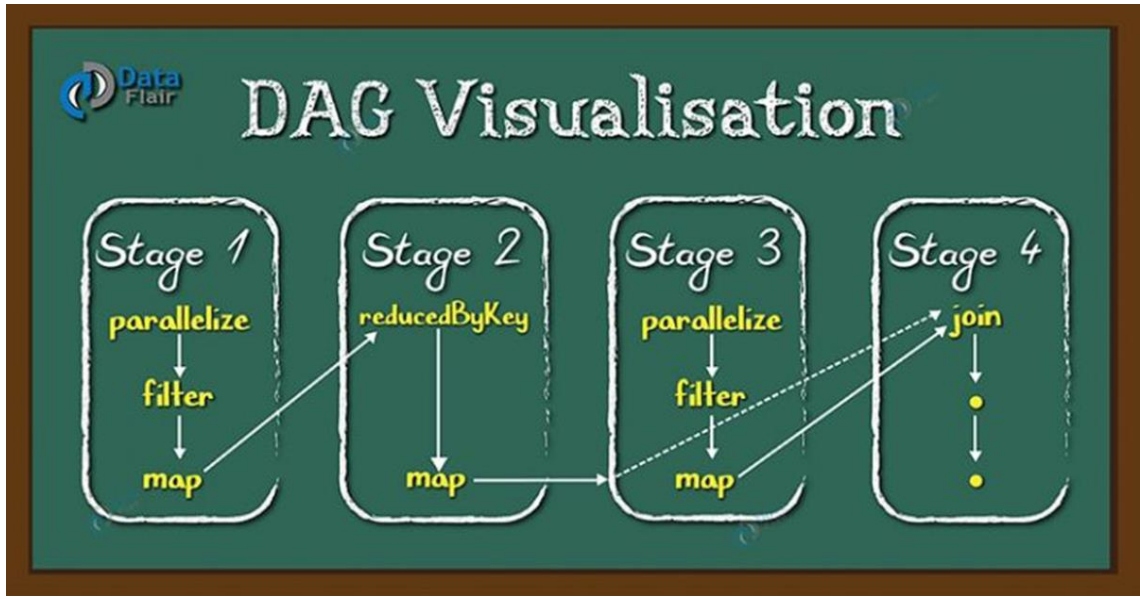


ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap
- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct
- takeOrdered
- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

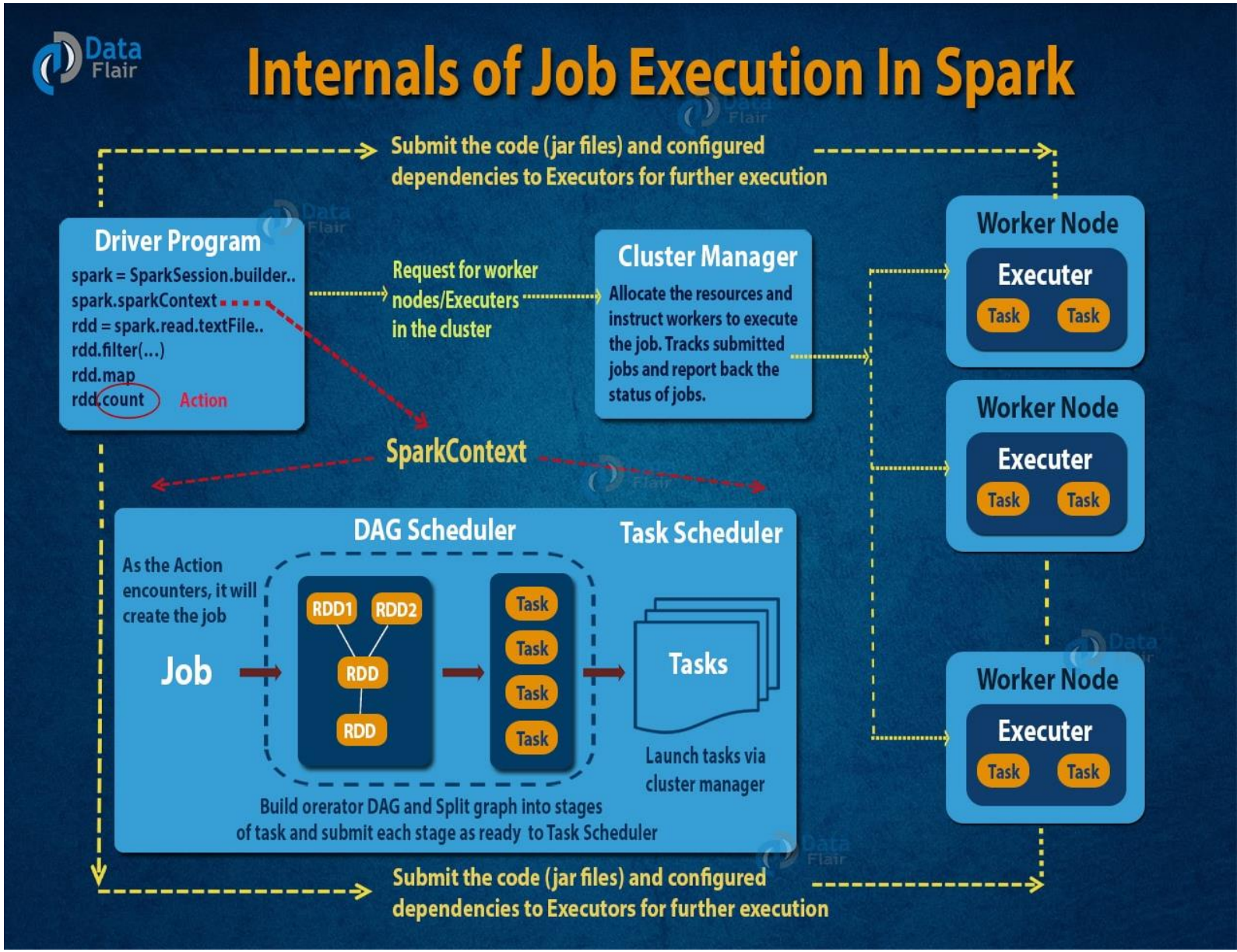
What is **DAG** (Directed Acyclic Graph)

- DAG in Apache Spark is a set of **Vertices and Edges**, where vertices represent the RDDs and the edges represent the operation to be applied on RDD. In Spark DAG, every edge directs from earlier to later in the sequence.
- It is a strict generalization of **MapReduce** model. DAG operations can do better global optimization than other systems like MapReduce.



How DAG works?

- The interpreter is the first layer, using a Scala interpreter.
- When we call an **Action** on Spark RDD, Spark submits the operator graph to the **DAG Scheduler.**
- Divide the operators into **stages** of the task in the DAG Scheduler. A stage contains task based on the partition of the input data. The DAG scheduler pipelines operators together. For example, map operators schedule in a single stage.
- The stages pass on to the **Task Scheduler**. It launches task through cluster manager. The dependencies of stages are unknown to the task scheduler.
- The **Workers** execute the task on the slave.



Hosein Alizadeh - Principal Data Scientist at Microsoft (https://www.linkedin.com/in/hoseinalizadeh/)

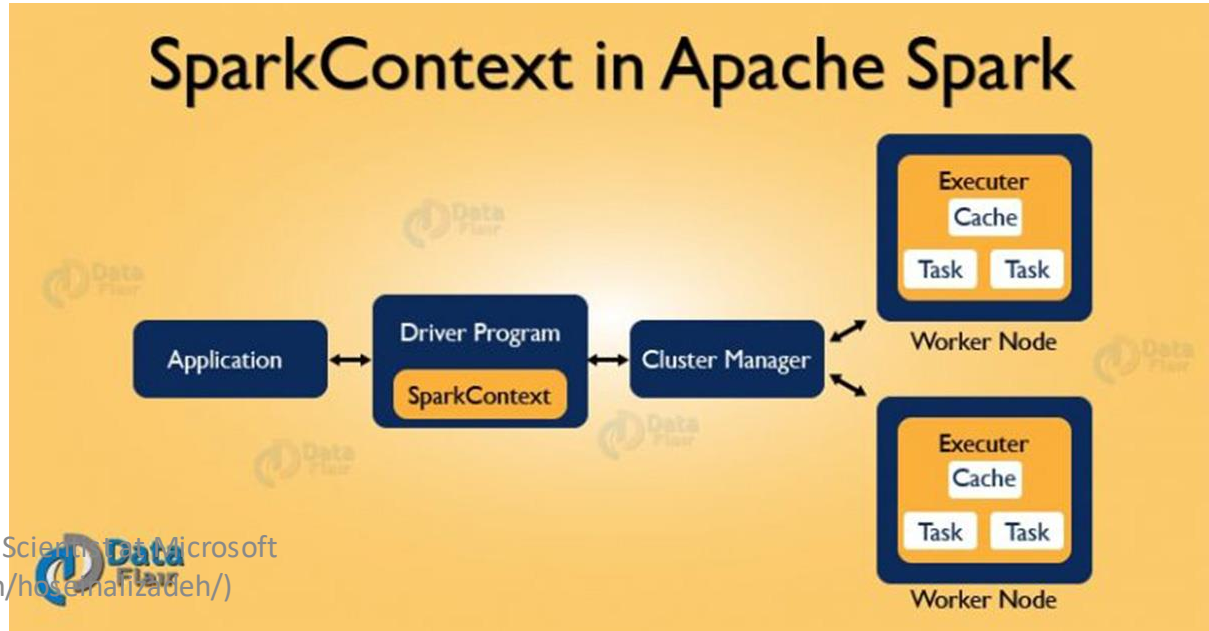
Sparkling Features of Spark RDD

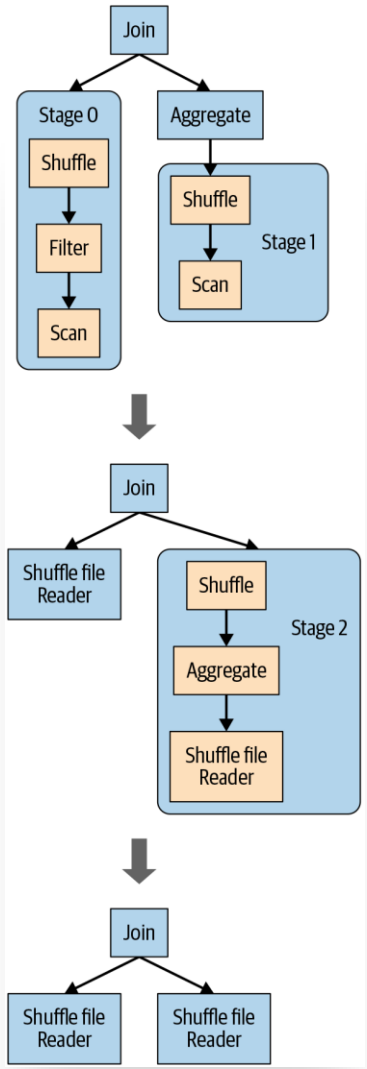
- **In-memory computation**
  - Basically, while storing data in RDD, data is stored in memory for as long as you want to store. It improves the performance by an order of magnitudes by keeping the data in memory.
- **Lazy Evaluation**
  - Spark Lazy Evaluation means the data inside RDDs are not evaluated on the go. Basically, ***only after an action triggers*** all the changes or the computation is performed. Therefore, it limits how much work it has to do.
- **Immutability**
  - Immutability means once we create an RDD, we can not manipulate it. Moreover, we can create a new RDD by performing any transformation. Also, *we achieve consistency through immutability*.

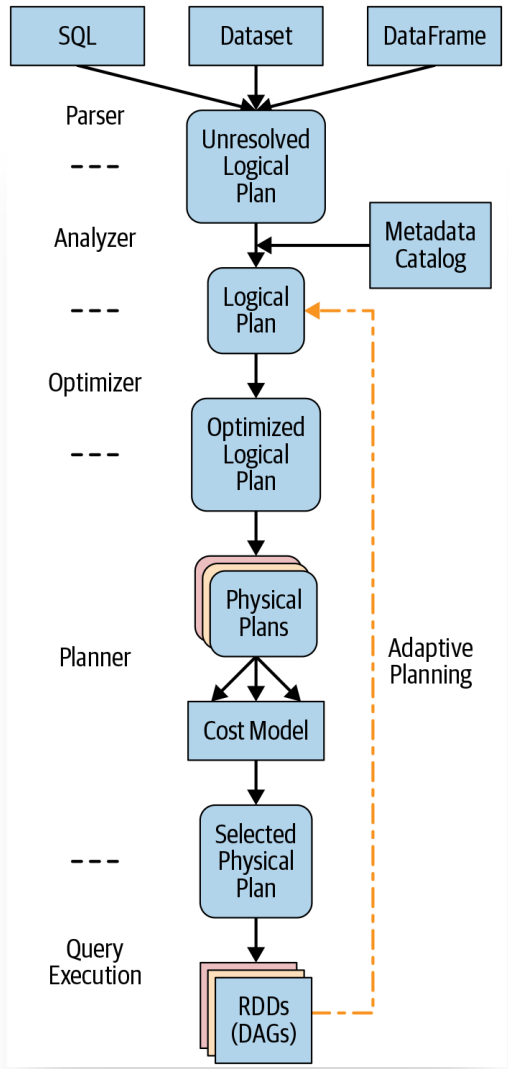
Saprk Context

- https://data-flair.training/blogs/learn-apache-spark-sparkcontext/
- SparkContext is the entry point of Spark functionality.
- **Only one SparkContext may be active per JVM node**.

- Spark driver application generates SparkContext which allows Spark Application to access Spark Cluster with the help of Resource Manager.

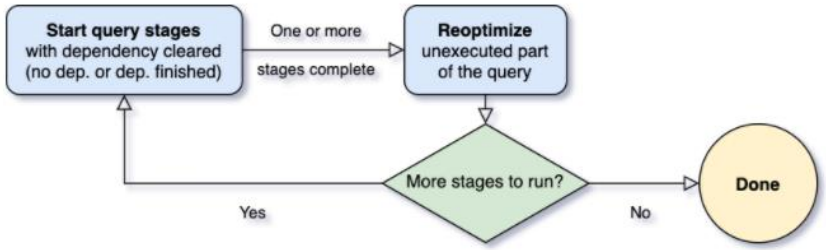






Adaptive Query Execution

- AQE reoptimizes and **adjusts query plans based on runtime statistics** collected in the process of query execution. It attempts to do the following at runtime:
  - Reduce the number of reducers in the shuffle stage by decreasing the number of shuffle partitions.
  - Optimize the physical execution plan of the query, for example by converting a SortMergeJoin into a BroadcastHashJoin where appropriate.
  - Handle data skew during a join.
- Spark operations in a query are pipelined and executed in parallel processes, but a shuffle or broadcast exchange breaks this pipeline, because the output of one stage is needed as input to the next stage. These **breaking points are called materialization points** in a query stage, and they present an opportunity to reoptimize and reexamine the query, as illustrated in the left Figure.



Fault Tolerance by Structured Streaming which uses Checkpointing

- The most important operational concern for a streaming application is **failure recovery**.
- **Structured Streaming** allows you to recover an application by just restarting it. To do this, you must configure the application to **use checkpointing** and **write-ahead logs**, both of which are handled automatically by the engine.
  - 1) you must configure a query **to write to a *checkpoint location* on a reliable file system** (like HDFS, S3).
  - 2) Structured Streaming will then **periodically save**:
    - a) **all relevant progress information** (for instance, the range of offsets processed in a given trigger)
    - b) the **current intermediate state values** to the checkpoint location.
- In a failure scenario, you simply need to:
  - 1) **restart your application**,
  - 2) **point to the same checkpoint location**,
  - 3) Then it will automatically recover its state and start processing data where it left off.
- You do not have to manually manage this state on behalf of the application—Structured Streaming does it for you.

Garbage Collection

- Spark runs on the Java Virtual Machine. Because Spark can store large amounts of data in memory, it has a major reliance on Java's memory management and garbage collection (GC). Therefore, garbage collection (GC) can be a major issue that can affect many Spark applications.

- Common symptoms of excessive GC in Spark are:
  - Slowness of application
  - Executor heartbeat timeout
  - GC overhead limit exceeded errors.

- ADDRESSING GARBAGE COLLECTION ISSUES:
  - DATA STRUCTURES: If using RDD based applications, use data structures with fewer objects. For example, use an array instead of a list.
  - SPECIALIZED DATA STRUCTURES that optimize memory usage of primitive types
  - STORING DATA OFF-HEAP:
  - BUILT-IN VS USER DEFINED FUNCTIONS (UDFS)
  - BE STINGY ABOUT OBJECT CREATION

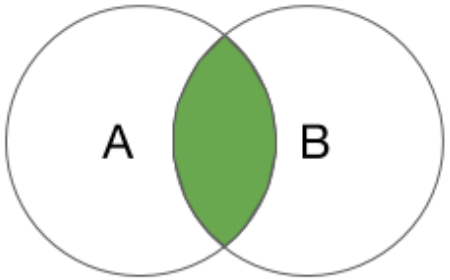
Review

- 1) After we submit a job before an action is called...what ever transformation are put in the code before an action is called on RDD. RDD will have history of lineage. that is which is the parent RDD and what are transformation has occurred to create this RDD and its dependency. this is called lineage (logical execution plan)
- 2) When an action is called on RDD, the lineage will be converted into DAG (Physical execution plan).
- 3) DAG (Physical execution plan) will be submitted to DAG Scheduler which in turn will split the DAG into Stages.
- 4) Each stage will have list of tasks.
- 5) Each task will run in an executor (One executor will run one task on one partition)
- Is it the responsibility of Catalyst optimizer that converts the RDD lineage into the best optimized execution plan as DAG?
  - Catalyst Optimizer (Spark's optimization engine) can *optimize the execution of your code by changing the execution order of your tasks.*
- It the responsibility of Tungsten encodes that convert the Scala code into bytecode? May be!!!

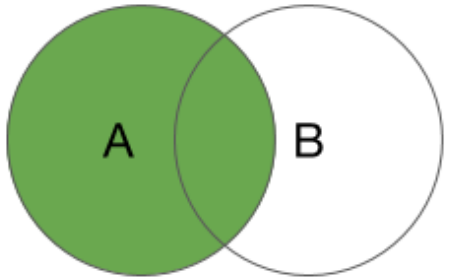
RDD Persistence (Cache)

- Spark RDD persistence is an optimization technique in which saves the result of RDD evaluation. Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.
- We can make persisted RDD through **cache()** and **persist()** methods. When we use the cache() method we can store all the RDD in-memory. We can persist the RDD in memory and use it efficiently across parallel operations.
- The difference between cache() and persist() is that:
  - Using cache() the default storage level is **MEMORY_ONLY.**
    - **df.cache().count()**
  - Using persist() we can use various storage levels.
    - **df.persist().count()**
    - **From pyspark import StorgaeLevel as SL**
    - **df.persist(SL.MEMORY_AND_DISK)**
  - Note: Both cache and persist are lazy operations, thus have to do an action after them to execute them.
  - Note: You can manually remove it using unpersist() method. i.e. **df.unpersist()**
  - By default, both cache() and persist() use MEMORY_ONLY StorageLevel, unless it is specifically mentioned in persist method.
- RDD Persistence is a key tool for an interactive algorithm. Because, when we persist RDD each node stores any partition of it that it computes in memory and makes it reusable for future use. This process speeds up the further computation ten times.

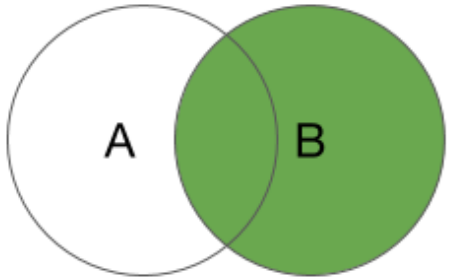
Join in dataframe



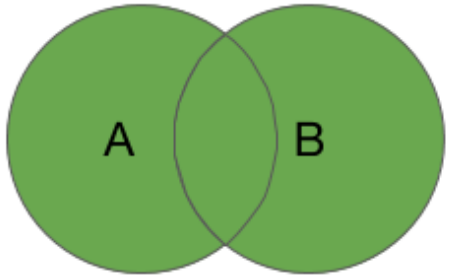
INNER JOIN



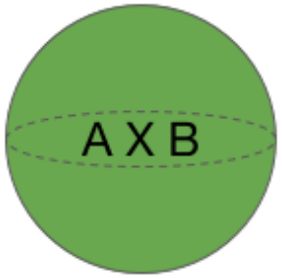
LEFT OUTER JOIN



RIGHT OUTER JOIN



FULL OUTER JOIN



CARTESIAN (CROSS) JOIN

Spark RDDs Operations

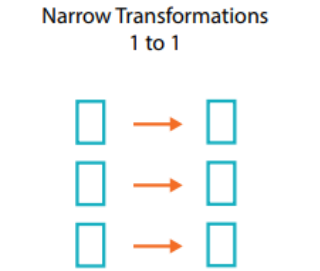
- **Resilient Distributed Dataset – RDD**:
  - It is the fundamental unit of data in Spark. Basically, it is ***a distributed collection of elements across cluster nodes***. Also performs parallel operations. Moreover, Spark ***RDDs are immutable*** in nature.

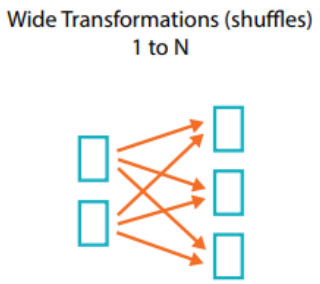
- There are 3 ways to create Spark RDDs:
  - **i. Parallelized collections:** By invoking parallelize method in the driver program, we can create parallelized collections.
  - **ii. External datasets:** One can create Spark RDDs, by calling a textFile method. Hence, this method takes URL of the file and reads it as a collection of lines.
  - **iii. Existing RDDs:** Moreover, we can create new RDD in spark, by applying transformation operation on existing RDDs.

- There are two types of operations, which Spark RDDs supports:
  - **i. Transformation Operations:** *It creates a new Spark RDD from the existing one*. Moreover, it passes the dataset to the function and returns new dataset. There are two types of RDD transformations:
    - **Narrow transformation** (e.g. map, filter, drop, coalesce, etc.). ***Narrow transformation*** **does not require the shuffling** of data across a partition, the **narrow transformations will group into single stage**.
    - **Wide transformation** (e.g. groupBy, repartition, distnct, reduceByKey). **In *wide transformation* the data shuffles**. Hence, **wide transformation results in stage boundaries**.
  - **ii. Action Operations:** (eg display, count, collect, first, show, take, head) In Apache Spark, *action returns final result to driver program or write it to the external* data store.

- *Transformations are **LAZY** ⇔ Actions are **EAGER***

- **RDD Linage**:
  - **Each RDD maintains a pointer to one or more parent** along with metadata about what type of relationship it has with the parent. For example, if we call ***val b=a.map()*** **on an RDD, the RDD *b* keeps a reference to its parent RDD *a***, that's an **RDD lineage**.





LAZY versus *EAGER*

- *Transformations are **LAZY** ⇔ Actions are **EAGER***

- Benefits of laziness:
  - Not forced to load all data at step #1
    - Technically impossible with **REALLY** large datasets.
  - Easier to parallelize operations
    - N different transformations can be processed on a single data element, on a single thread, on a single machine.
  - Optimizations can be applied prior to code compilation

Databricks Certified Associate Developer for Apache Spark 3.0 with Python 3

- DataFrames
- Spark SQL
- ETL Part 1
- ETL Part 2
- Spark Architecture
- Other Coding Notes

Download latest Databricks material here:
https://github.com/MicrosoftDocs/mslearn_databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

Magic command

To create a markdown cell you need to use a **"magic"** command which start with `%`.

The following provides the list of supported magics:

- `%python` - Allows you to execute **Python** code in the cell.
- `%r` - Allows you to execute **R** code in the cell.
- `%scala` - Allows you to execute **Scala** code in the cell.
- `%sql` - Allows you to execute **SQL** statements in the cell.
- `sh` - Allows you to execute **Bash Shell** commmands and code in the cell.
- `fs` - Allows you to execute **Databricks Filesystem** commands in the cell.
- `md` - Allows you to render **Markdown** syntax as formatted content in the cell.
- `run` - Allows you to **run another notebook** from a cell in the current notebook.

%run "./Includes/Classroom-Setup"

Other commands:

- Randomly divide a df to two parts:
- trainDF, testDF = urlTrendsDF.randomSplit([0.8,0.2], seed = 123)
- Filter names that contains “TX”:
- df.filter(F.col("NAME").like("%TX%"))

Spark

Description	Command
Create a Spark Dataframe	df = spark.range(1000).toDF("number")
Look at the columns	df.columns
Look at the schema	df.schema
Select a column	df.select("number").show(15)
Filter	df.where("number > 10 and number < 14").show()
Sample from dataframe	df.sample(withReplacement=False, fraction=0.2, seed=123).show(10)
Add a new column to a dataframe	df.withColumn(“plusOne", col(“total") + 1)
Convert Spark Dataframe to Pandas	pdf = df.toPandas()
Access to pandas entries	pdf.loc[0, 'ColumnA']
Slicing by index	pdf.loc[3:7:2]
Boolean mask	pdf[pdf['ColumnB']=='red']
Boolean mask multiple conditions	pdf[(pdf['ColumnB']=='red') & (pdf['ColumnA']==0.4)]

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Python For Data Science *Cheat Sheet*

PySpark - SQL Basics

Learn Python for data science Interactively at www.DataCamp.com

PySpark & Spark SQL

Spark SQL is Apache Spark's module for working with structured data.

Initializing SparkSession

A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files.

```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession \
        .builder \
        .appName("Python Spark SQL basic example") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()
```

Creating DataFrames

From RDDs

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

Infer Schema

```
>>> sc = spark.sparkContext
>>> lines = sc.textFile("people.txt")
>>> parts = lines.map(lambda l: l.split(","))
>>> people = parts.map(lambda p: Row(name=p[0],age=int(p[1])))
>>> peopledf = spark.createDataFrame(people)
```

Specify Schema

```
>>> people = parts.map(lambda p: Row(name=p[0],
                                     age=int(p[1].strip())))
>>> schemaString = "name age"
>>> fields = [StructField(field_name, StringType(), True) for
field_name in schemaString.split()]
>>> schema = StructType(fields)
>>> spark.createDataFrame(people, schema).show()
+--------+---+
|    name|age|
+--------+---+
|    Mine| 28|
|   Filip| 29|
|Jonathan| 30|
+--------+---+
```

From Spark Data Sources

JSON

```
>>> df = spark.read.json("customer.json")
>>> df.show()
+--------------------+---+---------+--------+--------------------+
|             address|age|firstName |lastName|         phoneNumber|
+--------------------+---+---------+--------+--------------------+
|[New York,10021,N...| 25|     John|   Smith|[[212 555-1234,ho...|
|[New York,10021,N...| 21|     Jane|     Doe|[[322 888-1234,ho...|
+--------------------+---+---------+--------+--------------------+
>>> df2 = spark.read.load("people.json", format="json")
```

Parquet files

```
>>> df3 = spark.read.load("users.parquet")
```

TXT files

```
>>> df4 = spark.read.text("people.txt")
```

Duplicate Values

```
>>> df = df.dropDuplicates()
```

Queries

```
>>> from pyspark.sql import functions as F
```

Select

```
>>> df.select("firstName").show()
```
Show all entries in `firstName` column

```
>>> df.select("firstName","lastName") \
      .show()
>>> df.select("firstName",
              "age",
               explode("phoneNumber") \
              .alias("contactInfo")) \
      .select("contactInfo.type",
              "firstName",
              "age") \
      .show()
```
Show all entries in `firstName`, `age` and `type`

```
>>> df.select(df["firstName"],df["age"]+ 1)
      .show()
>>> df.select(df['age'] > 24).show()
```
Show all entries in `firstName` and `age`, add `1` to the entries of `age`
Show all entries where `age` `>24`

When

```
>>> df.select("firstName",
               F.when(df.age > 30, 1) \
              .otherwise(0)) \
      .show()
>>> df[df.firstName.isin("Jane","Boris")]
                .collect()
```
Show `firstName` and 0 or 1 depending on `age` `>30`

Show `firstName` if in the given options

Like

```
>>> df.select("firstName",
              df.lastName.like("Smith")) \
      .show()
```
Show `firstName`, and `lastName` is `TRUE` if `lastName` is like `Smith`

Startswith - Endswith

```
>>> df.select("firstName",
              df.lastName \
                .startswith("Sm")) \
      .show()
>>> df.select(df.lastName.endswith("th")) \
      .show()
```
Show `firstName`, and `TRUE` if `lastName` starts with `Sm`

Show last names ending in `th`

Substring

```
>>> df.select(df.firstName.substr(1, 3) \
                       .alias("name")) \
      .collect()
```
Return substrings of `firstName`

Between

```
>>> df.select(df.age.between(22, 24)) \
      .show()
```
Show `age`: values are `TRUE` if between `22` and `24`

Add, Update & Remove Columns

Adding Columns

```
>>> df = df.withColumn('city',df.address.city) \
           .withColumn('postalCode',df.address.postalCode) \
           .withColumn('state',df.address.state) \
           .withColumn('streetAddress',df.address.streetAddress) \
           .withColumn('telePhoneNumber',
                       explode(df.phoneNumber.number)) \
           .withColumn('telePhoneType',
                       explode(df.phoneNumber.type))
```

Updating Columns

```
>>> df = df.withColumnRenamed('telePhoneNumber', 'phoneNumber')
```

Removing Columns

```
>>> df = df.drop("address", "phoneNumber")
>>> df = df.drop(df.address).drop(df.phoneNumber)
```

GroupBy

```
>>> df.groupBy("age")\
      .count() \
      .show()
```
Group by `age`, count the members in the groups

Filter

```
>>> df.filter(df["age"]>24).show()
```
Filter entries of `age`, only keep those records of which the values are `>24`

Sort

```
>>> peopledf.sort(peopledf.age.desc()).collect()
>>> df.sort("age", ascending=False).collect()
>>> df.orderBy(["age","city"],ascending=[0,1])\
      .collect()
```

Missing & Replacing Values

```
>>> df.na.fill(50).show()
>>> df.na.drop().show()
>>> df.na \
      .replace(10, 20) \
      .show()
```
Replace null values
Return new `df` omitting rows with null values
Return new `df` replacing one value with another

Repartitioning

```
>>> df.repartition(10)\
      .rdd \
      .getNumPartitions()
>>> df.coalesce(1).rdd.getNumPartitions()
```
`df` with 10 partitions

`df` with 1 partition

Running SQL Queries Programmatically

Registering DataFrames as Views

```
>>> peopledf.createGlobalTempView("people")
>>> df.createTempView("customer")
>>> df.createOrReplaceTempView("customer")
```

Query Views

```
>>> df5 = spark.sql("SELECT * FROM customer").show()
>>> peopledf2 = spark.sql("SELECT * FROM global_temp.people")\
                 .show()
```

Output

Data Structures

```
>>> rdd1 = df.rdd
>>> df.toJSON().first()
>>> df.toPandas()
```
Convert `df` into an RDD
Convert `df` into a RDD of string
Return the contents of `df` as Pandas `DataFrame`

Write & Save to Files

```
>>> df.select("firstName", "city")\
      .write \
      .save("nameAndCity.parquet")
>>> df.select("firstName", "age") \
      .write \
      .save("namesAndAges.json",format="json")
```

Stopping SparkSession

```
>>> spark.stop()
```

Inspect Data

```
>>> df.dtypes
>>> df.show()
>>> df.head()
>>> df.first()
>>> df.take(2)
>>> df.schema
```
Return `df` column names and data types
Display the content of `df`
Return first `n` rows
Return first row
Return the first `n` rows
Return the schema of `df`

```
>>> df.describe().show()
>>> df.columns
>>> df.count()
>>> df.distinct().count()
>>> df.printSchema()
>>> df.explain()
```
Compute summary statistics
Return the columns of `df`
Count the number of rows in `df`
Count the number of distinct rows in `df`
Print the schema of `df`
Print the (logical and physical) plans

Convert String to timestamp

- Option 1: using F.to_timestamp
  - **df.select(F.to_timestamp(F.col("publishedOn")).alias("publishedOn").show()**
- Option 2: Using cast('timestamp')
  - **df.select(F.col("publishedOn").cast('timestamp')).alias("publishedOn")**
- Option 3: using unix_timestamp and cast
  - **df.select(F.unix_timestamp(F.col("capturedAt"), "yyyy-MM-dd'T'HH:mm:ss").cast("timestamp"))** **->** "2015-04-19T05:56:25"
  - **Note:** unix_timestamp converts time string with given pattern (see SimpleDateFormat) to Unix time stamp (in seconds), return null if fail.

- Using substr:
  - **F.col('string_column').substr(5,10)**

- The created_at column is formatted as a non-standard timestamp in string format.
- Parse the timestamp column using unix_timestamp and cast the result as Timestamp
  - Note: the "EEE" in "EEE MMM dd HH:mm:ss ZZZZZ yyyy" is not supported in Spark 3.

  - "Mon Jul 18 09:48:55 +0.000 2019"
  - **F.unix_timestamp(F.col('created_at'), "EEE MMM dd HH:mm:ss ZZZZZ yyyy").cast(F.TimestampType().alias('createdAt')**

Convert date string to date:

.select(**F.to_date(F.date_format("dates.publishedOn", "yyyy-MM-dd"))**.alias("publishedOn"))

```
%sql
date_format(cast(dates.publishedOn as timestamp), "MMM dd, yyyy") as publishedOn
```

```
spark.read
    .option("delimiter", "\t")
    .option("header", True)
    .option("inferSchema", True)
    .option("timestampFormat", "MM/dd/yyyy hh:mm:ss a")
    .csv("/mnt/training/Chicago-Crimes-2018.csv")
```

```
%sql
CREATE OR REPLACE TEMPORARY VIEW DatabricksBlog2 AS
  SELECT *,
         cast(dates.publishedOn AS timestamp) AS publishedOn
  FROM DatabricksBlog
```

```
%sql
SELECT title,
       date_format(publishedOn, "MMM dd, yyyy") AS date,
       link
FROM DatabricksBlog2
WHERE year(publishedOn) = 2013
ORDER BY publishedOn
```

02-Querying-Files

- Give count() an alias: use “agg”
  - .groupBy("firstName")
  - .agg(count("firstName").alias("nameCount"))
  - .orderBy("nameCount", ascending = False)

OR

.count().withColumnRenamed("count","total")

- Take the first rows of a dataframe:
  - df.head(3) → Return first n rows
  - df.take(3) → Return the first n rows as array
  - df.**limit**(3) → Return the first n rows as a dataframe
  - df.first() → Return first row
  - df.take(n)[2] → Return the third row from the first n rows
  - df[2] → return the third column of the dataframe

Coding Notes

- How many of the first names in peopleDF appear in ssaDF data files?
- peopleDF.withColumnRenamed("firstName", "firstNamePeople").join(ssaDF, col("firstNamePeople") == col("firstName")).select("firstName").distinct().count()
- OR
- peopleDF.join(ssaDF, "firstName").select("firstName").distinct().count()

- Change the values of column “Salary” inplace:
  - peopleDF.withColumn("salary", abs(col("salary")))

- Notes: Some of the functions in “pyspark.sql.functions” require “ColumnName” and others “Column”.
  - Those like “avg” that require “columnName” could be used like, select(avg(“Salary”))
  - But those like “abs” that require “column” must be used with “col” function like, select(abs(col(“Salary”)))

- As a refinement, assume all salaries under $20,000 represent bad rows and filter them out. Additionally, categorize each person's salary into $10K groups.
  - peopleWithFixedSalariesDF.filter("salary >= 20000").withColumn("Salary10k", round(col("salary")/10000))

- Add a column with constant value:
  - from pyspark.sql.functions import lit
  - .withColumn("city", lit("Los Angeles")
  - %sql: Select “Los Angeles” as city

Challenge Exercise

- Find the most popular first name for girls in 1885, 1915, 1945, 1975, and 2005
  - Step1: df_filtered = filter by “gender=‘F’” and “year=1885 OR year=1915, …”
  - Step2: df_max = df_filtered.groupBy(“year”).agg(max(‘total’).alias(‘total’)
  - Step3: df_joined = df_filtered.join(df_max, [‘year’, ‘total’]

Split & access elements in dataframe

- Split a text column of “name” (e.g “Hosein Alizadeh”) to two columns of “firstName” and “lastName” (e.g “Hosein” and “Alizadeh”)
  - df.withColumn('firstName', F.split(F.col('name'), ' ')[0])
  - .withColumn('lastName', F.split(F.col('name'), ' ')[1])

- Take the first 3 rows of a dataframe as array:
  - df.take(3)

- Access to elements of Spark dataframe:
  - first column of first row: **df.first()[0]**
  - First column of 3rd row: **df.take(3)[2][0]**
  - 7th column of 3rd row: **df.take(3)[2][6]**

- Read data and drop the row if three specific columns were null:
  - df.filter(“(col1 is null) AND (col2 is null) AND (col3 is null)”)

Add a new row to spark df:

```
+---------------+---+---------+--------+
|           name| id|firstName|lastName|
+---------------+---+---------+--------+
|Hosein Alizadeh|  1|   Hosein|Alizadeh|
|    Megan Ganji|  2|    Megan|   Ganji|
|Ariana Alizadeh|  3|   Ariana|Alizadeh|
+---------------+---+---------+--------+
```

```
dfnew = spark.createDataFrame([('Nane Vahedi', 4, 'Nane','Vahedi')], df.columns)
df2 = df.unionAll(dfnew)
df2.show()
```

```
+---------------+---+---------+--------+
|           name| id|firstName|lastName|
+---------------+---+---------+--------+
|Hosein Alizadeh|  1|   Hosein|Alizadeh|
|    Megan Ganji|  2|    Megan|   Ganji|
|Ariana Alizadeh|  3|   Ariana|Alizadeh|
|    Nane Vahedi|  4|     Nane|  Vahedi|
+---------------+---+---------+--------+
```

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Appendix

Spark ML

Apache Spark is used to Train and evaluate machine learning models

```
from pyspark.sql.functions import col, floor, translate, round
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, OneHotEncoder
from pyspark.ml.regression import LinearRegression

inputDF = (spark.read.table("AirlineFlight")
  .withColumn("HourOfDay", floor(col("CRSDepTime")/100))
  .withColumn("DepDelay", translate(col("DepDelay"), "NA", "0").cast("integer")))

(trainingDF, testDF) = inputDF.randomSplit([0.80, 0.20], seed=999)

pipeline = Pipeline(stages=[
    OneHotEncoder(inputCol="HourOfDay", outputCol="HourVector"),
    VectorAssembler(inputCols=["HourVector"], outputCol="Features"),
    LinearRegression(featuresCol="Features", labelCol="DepDelay", predictionCol="DepDelayPredicted", regParam=0.0)
  ])

model = pipeline.fit(trainingDF)
resultDF = model.transform(testDF)

displayDF = resultDF.select("Year", "Month", "DayOfMonth", "CRSDepTime", "UniqueCarrier", "FlightNum", "DepDelay", round("DepDelayPredicted", 2).alias("DepDelayPredicted"))
display(displayDF)
```

```
display(
  resultDF
    .groupBy("HourOfDay")
    .avg("DepDelay", "DepDelayPredicted")
    .toDF("HourOfDay", "Actual", "Predicted")
    .orderBy("HourOfDay")
)
```

Azure Data Factory

- Azure Data Factory is a cloud-based data **integration** service.
  - It lets you **create data-driven workflows** in the cloud for **orchestrating** and **automating data movement** and data **transformation**.
  - You can use Azure Data Factory to **load data into SQL Data Warehouse.**
  - Azure Data Factory is a **data ingestion** and **transformation** service that allows you to load raw data from over **70** different on-premises or cloud sources.
  - Data Factory supports **data workflow pipelines**. These pipelines are a logical group of tasks and activities that allows end-to-end data-processing scenarios.
- **Select the destination:**
  - On **Table mapping** tab, you'll set the mapping between the source and destination tables.
  - On the **Schema mapping** tab, you'll set the mapping between the source and destination columns.
  - On the **Settings** tab, you'll specify actions to take if there are incompatible rows between the source and the destination.
  - The **Summary** tab summarizes all the details that you've entered. You can also change the values on this tab.
  - The **Deployment** tab provides the status of various deployment activities. You'll see the status update for each of the configured steps for the new pipeline as it deploys.

Load Data by PolyBase

- **PolyBase** is a technology that accesses data outside of a database via the **T-SQL language**.
- In Azure SQL Data Warehouse, you can import and export data to and from Azure **Blob** storage and Azure **Data Lake** Store.
- Use HADOOP when the external data source is Hadoop or Azure Blob storage for Hadoop
- ***Reject options***. You can specify reject parameters that determine how PolyBase will handle dirty records it retrieves from the external data source.
  - A data record is considered ***dirty record*** if its data types or the number of columns don't match the column definitions of the external table.

Connect to SQL DW via Azure Databricks

- You can access SQL Data Warehouse from Azure Databricks by using the SQL DW connector.
- SQL Data Warehouse connector is a data source implementation for Apache Spark that uses Azure Blob storage + PolyBase in SQL DW to transfer large volumes of data efficiently between an Azure Databricks cluster and a SQL Data Warehouse instance.
- Both the Azure Databricks cluster and the SQL Data Warehouse instance access a common Blob storage container to exchange data.
  - In Azure Databricks, Spark jobs are triggered by the SQL DW connector to read data/write data to the Blob storage container.
  - On the SQL DW side, data loading and unloading operations performed by PolyBase are triggered by the SQL DW connector through JDBC.
- ***dwtemp*** is the common container that the Azure Databricks cluster and the SQL Data Warehouse instance will use.

Summary

1. Which of the following is a feature of Azure SQL Data Warehouse that can help you save in compute costs when you don't need to run any queries for a while?
    1. Scale down the data warehouse units (DWUs).
    2. Delete your SQL Data Warehouse instance and restore it later from backup.
    3. **Pause the SQL Data Warehouse instance.**
2. What's the recommended way to save data if you plan to run several queries against one SQL Data Warehouse table?
    1. **Save the data in a format like Parquet.**
    2. Use the SQL Data Warehouse connector.
    3. Save data directly to Azure Blob storage.
3. What are the two prerequisites for connecting Azure Databricks with SQL Data Warehouse that apply to the SQL Data Warehouse instance?
    1. **Create a database master key and configure the firewall to enable Azure services to connect.**
    2. Use a correctly formatted ConnectionString and create a database master key.
    3. Add the client IP address to the firewall's allowed IP addresses list and use the correctly formatted ConnectionString.

Data Factory

- Data ingestion and transformation by Data Factory and Databricks involves the following steps:
  - **Create an Azure storage account.** You'll use this storage account to store your ingested and transformed data.
  - **Create a Data Factory instance.** After you set up your storage account, create your Data Factory instance by using the Azure portal.
  - **Create a data workflow pipeline.** To create the pipeline, copy data from your source by using a copy activity in Data Factory. A copy activity allows you to copy data from different on-premises and cloud sources.
  - **Add a Databricks notebook to the pipeline.** This notebook contains the code to transform and clean the raw data as required.
  - **Analyze the data.** Now that your data is cleaned up, use Databricks notebooks to further train the data or analyze it to output the required results.

Summary

1. What's the purpose of linked services in Azure Data Factory?
    1. **To represent a data store or a compute resource that can host execution of an activity.**
    2. To represent a processing step in a pipeline.
    3. To link data stores or computer resources together for the movement of data between resources.
2. Can parameters be passed into an Azure Databricks notebook from Azure Data Factory?
    1. **Yes**
    2. No
3. Databricks activities (notebook, JAR, Python) in Azure Data Factory will fail if the target cluster in Azure Databricks isn't running when the cluster is called by Data Factory.
    1. **No**
    2. Yes

- **Linked services define the connection information needed for Data Factory to connect to external resources.**
- **The target notebook must include widgets configured with the necessary parameter names. The values can then be passed as parameters from a Databricks notebook activity in Data Factory.**
- **If the target cluster is stopped, Databricks will start the cluster before attempting to execute the notebook, JAR, or Python file. This situation will result in a longer execution time because the cluster must start, but the activity will still execute as expected.**

Summary

- **Q:** What makes Spark different than Hadoop?
  - **A:** Spark on Databricks performs 10-2000x faster than Hadoop Map-Reduce. It does this by providing a high-level query API which allows Spark to highly optimize the internal execution without adding complexity for the user.
  - Internally, Spark employs a large number of optimizations such as pipelining related tasks together into a single operation, communicating in memory, using just-in-time code generation, query optimization, efficient tabular memory (Tungsten), caching, and more.
- **Q:** What are the visualization options in Databricks?
  - **A:** Databricks provides a wide variety of built-in visualizations. Databricks also supports a variety of 3rd party visualization libraries, including d3.js, matplotlib, ggplot, and plotly.
- **Q:** Where can I learn more about DBFS?
  - **A:** See the document Databricks File System - DBFS.
- **Q:** Where can I find a list of the machine learning algorithms supported by Spark?
  - **A:** The Spark documentation for Machine Learning describes the algorithms for classification, regression, clustering, recommendations (ALS), neural networks, and more. The documentation doesn't provide a single consolidated list, but by browsing through the Spark MLLib documentation you can find the supported algorithms. Additionally, 3rd party libraries provide even more algorithms and capabilities.
- **Q:** Where can I learn more about stream processing in Spark?
  - **A:** See the Structured Streaming Programming Guide.
- **Q:** Where can I learn more about GraphFrames?
  - **A:** See the GraphFrames Overview. The Databricks blog has an example which uses d3 to perform visualizations of GraphFrame data.

Review Question

- **Question:** Which of the following are good applications for Apache Spark? (Select all that apply.)
  - Querying, exploring, and analyzing very large files and data sets
  - Joining data lakes
  - Machine learning and predictive analytics
  - Processing streaming data
  - Graph analytics
  - Overnight batch processing of very large files

Azure Data Lake Storage - ADLS Gen2

- There are three supported methods for connecting Databricks to ADLS Gen2:
  - Direct access with a Shared Key.
  - Direct access with OAuth.
  - Mounting using OAuth.
- A fundamental part of ADLS Gen2 is the addition of a hierarchical namespace to Blob storage.
  - The hierarchical namespace organizes objects/files into a hierarchy of directories for efficient data access.
  - A common object store naming convention uses slashes in the name to mimic a hierarchical directory structure.
  - Operations such as renaming or deleting a directory become single atomic metadata operations on the directory rather than enumerating and processing all objects that share the name prefix of the directory.
  - **Important**: When the hierarchical namespace is enabled, Azure Blob Storage APIs are not available, which means you cannot use the `wasb` or `wasbs` scheme to access the `blob.core.windows.net` endpoint.
  - Before you can access the hierarchical namespace in your ADLS Gen2 account, you must initialize a file system.

Connect Databricks and PowerBI - Review

- You can connect your Databricks cluster to Power BI to create visualization
- To expand your options for creating useful visualizations, Databricks supports the third-party Matplotlib libraryns for your data.

1. To use Power BI with DataBricks tables, which kind of connection do you need to establish first?

JDBC
ODBC
REST API direct connection

2. In notebooks, which function can be used for visualizing data in any language?

show()
visualize()
display()

3. Matplotlib is a library that's included with Databricks and doesn't need to be imported as a third-party library.

Correct
Incorrect

Databricks Notebooks: https://github.com/MicrosoftDocs/mslearn-create-data-visualizations-using-azure-databricks-and-power-bi/blob/master/DBC/07-Visualization.dbc?raw=true
Note: 04-Matplotlib has some interesting visualization using matplotlip including pandas com, seaborns heatmap, and pandas bar

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

ETL in databricks

- Import url:
  - https://github.com/MicrosoftDocs/mslearn-perform-basic-data-transformation-in-azure-databricks/blob/master/DBC/05.1-Basic-ETL.dbc?raw=true
- Including**:**
  - **01-Course-Overview-and-Setup**: This notebook gets you started with your Databricks workspace.
  - **02-ETL-Process-Overview**: This notebook contains exercises to help you query large data files and visualize your results.
  - **03-Connecting-to-Azure-Blob-Storage**: You do basic data aggregation and joins in this notebook.
  - **04-Connecting-to-JDBC**: This notebook lists the steps for accessing data from various sources by using Databricks.
  - **05-Applying-Schemas-to-JSON**: In this notebook, you learn how to query JSON and hierarchical data with DataFrames.
  - **06-Corrupt-Record-Handling**: This notebook lists exercises that help you create an Azure Data Lake Storage Gen2 storage account and use Databricks DataFrames to query and analyze this data.
  - **07-Loading-Data-and-Productionalizing**: Here you use Databricks to query and analyze data stores in Azure Data Lake Storage Gen2.
  - **Parsing-Nested-Data**: This notebook is in the *Optional* subfolder. It includes a sample project you can explore later.

Knowledge Check

1. How do you specify parameters when you're reading data?

Using .parameter() during your read allows you to pass key/value pairs that specify aspects of your read.

Using .option() during your read allows you to pass key/value pairs that specify aspects of your read.

Using .option() during your read allows you to pass key/value pairs that specify aspects of your read. For instance, options for reading CSV data include header, delimiter, and inferSchema.

Using .keys() during your read allows you to pass key/value pairs that specify aspects of your read.

2. How do you connect your Spark cluster to Azure Blob storage?

By calling the .connect() function on the Spark cluster.

By mounting it.

Correct. Mounts require Azure credentials, such as shared access signature (SAS) keys.

By calling the .connect() function on the Azure Blob storage account.

3. By default, how are corrupt records dealt with when you're using spark.read.json()?

They appear in a column called "_corrupt_record".

Correct. Corrupt records are the ones that Spark can't read (for example, when characters are missing from a JSON string).

They get deleted automatically.

They throw an exception and exit the read operation.

Create Data Pipelines by using Databricks Delta

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

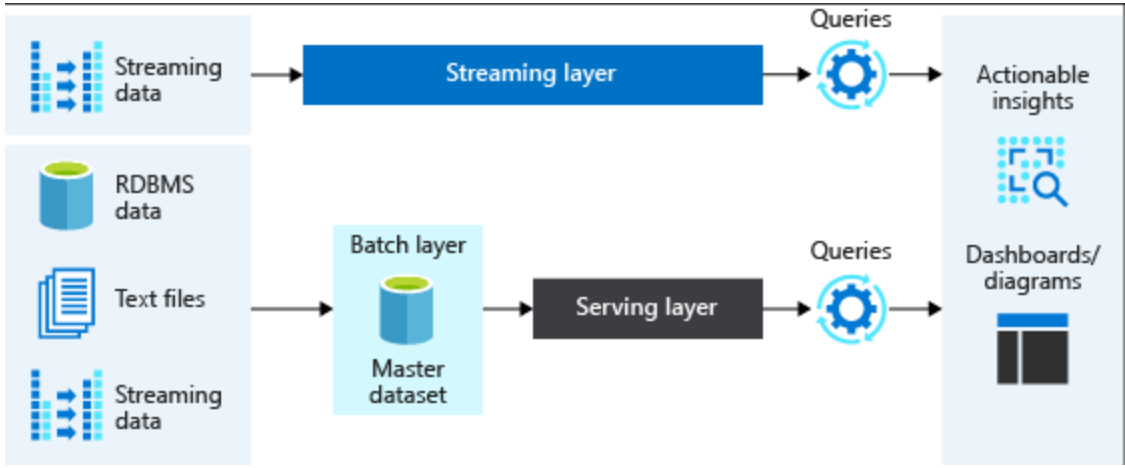
Databricks Delta

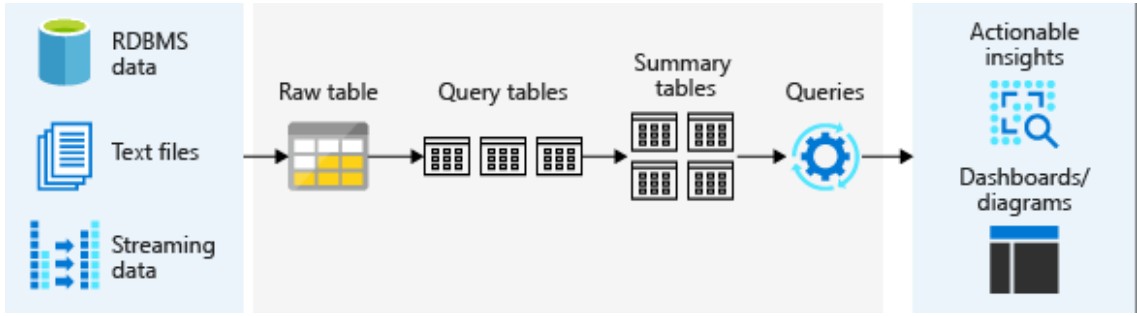
- **Databricks Delta**
  - **Databricks Delta** is a transactional storage layer designed specifically to work with Apache Spark and Databricks File System (DBFS).
  - At the core of Databricks Delta is an *optimized Spark table*. It stores your data as Apache Parquet files in DBFS and maintains a transaction log that efficiently tracks changes to the table.
  - It is a Spark table with built-in reliability and performance optimizations.
  - Read and write data that's stored in Databricks Delta by using Apache Spark SQL batch and streaming APIs.
  - Databricks Delta provides the following functionality:
    - **ACID transactions**: Multiple writers can simultaneously modify a dataset and see consistent views.
    - **DELETE, UPDATE, and UPSERT operations**: Writers can modify a dataset without interfering with jobs that read the dataset.
    - **Automatic file management**: Data access speeds up because data is organized into large files that can be read efficiently.
    - **Statistics and data skipping**: Reads are 10 to 100 times faster because statistics are tracked about the data in each file. This tracking lets Delta avoid reading irrelevant information.

- **data lake**
  - A **data lake** is a storage repository that inexpensively stores a vast amount of raw data, both current and historical, in native formats such as XML, JSON, CSV, and Parquet. It may contain operational relational databases with live transactional data.
  - To extract meaningful information from a data lake, you must solve problems such as:
    - Schema enforcement when new tables are introduced.
    - Table repairs when any new data is inserted into the data lake.
    - Frequent refreshes of metadata.
    - Bottlenecks of small file sizes for distributed computations.
    - Difficulty sorting data by an index if data is spread across many files and partitioned.

Databricks Delta vs. Lambda Architectures

- **Lambda Architecture**
  - The Lambda Architecture is a big-data processing architecture that combines both ***batch*** and ***real-time*** processing methods.
  - It features an append-only, immutable data source that serves as the system of record.
  - Time-stamped events are appended to existing events, and nothing is overwritten. Data is implicitly ordered by time of arrival.
  - There are really two pipelines here; hence the name **Lambda** Architecture.
    - one batch and
    - one streaming
- **Databricks Delta Architecture**
  - The Databricks Delta architecture is a vast improvement upon the traditional Lambda Architecture.
  - Text files, RDBMS data, and streaming data are collected into a *raw table*, also known as a *bronze table* at Databricks. A raw table is then parsed into *query tables*, also known as *silver tables* at Databricks. Query tables may be joined with dimension tables.
  - *Summary tables*, also known as *gold tables* at Databricks, are business-level aggregates. They're often used for reporting, dashboards, and aggregations such as daily active website users.





01-Introducing-Delta

- **Question:** What is Databricks Delta?
  - **Answer:** Databricks Delta is a mechanism of effectively managing the flow of data (**data pipeline**) to and from a **Data Lake**.
- **Question:** What are some of the pain points of existing data pipelines?
  - **Answer:**
    - Introduction of new tables requires schema creation
    - Whenever any new data is inserted into the data lake, table repairs are required
    - Metadata must be frequently refreshed
    - Small file sizes become a bottleneck for distributed computations
    - If data is sorted by a particular index (i.e. eventTime), it is very difficult to re-sort the data by a different index (i.e. userID)
- **Q:** Where can I find documentation on Databricks Delta?
  - **A:** See Databricks Delta Guide.

02-Create

- Q: What is the Databricks Delta command to display metadata?
  - A: Metadata is displayed through **DESCRIBE DETAIL** tableName.
- Q: Where does the schema for a Databricks Delta data set reside?
  - A: The table name, path, database info are stored in **Hive metastore**, the actual schema is stored in the **_delta_logs directory**.
- Q: What is the general rule about partitioning and the cardinality of a set?
  - A: We should partition on sets that are of **small cardinality to avoid penalties** incurred with managing large quantities of partition info meta-data.
- Q: What is schema-on-read?
  - A: It stems from Hive and roughly means: **the schema for a data set is unknown until you perform a read operation**.
- Q: How does this problem manifest in Databricks assuming a parquet based data lake?
  - A: It shows up as missing data upon load into a table in Databricks.
- Q: How do you remedy this problem in Databricks above?
  - A: To remedy, you **repair the table using MSCK REPAIR TABLE** or **switch to Databricks Delta**!

03-Append

- Q: What parameter do you need to add to an existing dataset in a Delta table?
  - A: **df.write...mode("append").save("..")**

- Q: What's the difference between .mode("append") and .mode("overwrite") ?
  - A: append atomically adds new data to an existing Databricks Delta table and overwrite atomically replaces all of the data in a table.

- Q: I've just repaired myTable using MSCK REPAIR TABLE myTable; How do I verify that the repair worked ?
  - A: SELECT **count(*)** FROM myTable and make sure the count is what I expected

- Q: In exercise 2, why did we use .withColumn(.. cast(rand(5) ..) i.e. pass a seed to the rand() function ?
  - A: In order to ensure we get the SAME set of pseudo-random numbers every time, on every cluster.

04-Upsert

- Q: What does it mean to UPSERT?
  - A: To UPSERT is to either INSERT a row, or if the row already exists, UPDATE the row.

- Q: What happens if you try to UPSERT in a parquet-based data set?
  - A: That's **not possible** due to the schema-on-read paradigm, you will get an error until you refresh the table.

- Q: What is schema-on-read?
  - A: It stems from Hive and roughly means: the schema for a data set is unknown until you perform a read operation.

- Q: How to you perform UPSERT in a Databricks Delta dataset?
  - A: Using the **MERGE INTO *my-table* USING *data-to-upsert***.

- Q: What is the caveat to USING data-to-upsert?
  - A: Your source data has ALL the data you want to replace: in other words, you create a new dataframe that has the source data you want to replace in the Databricks Delta table.

05-Streaming

- **Q:** Why is Databricks Delta so important for a data lake that incorporates streaming data?
  - **A:** Frequent meta data refreshes, table repairs and accumulation of small files on a secondly- or minutely-basis!
- **Q:** What happens if you shut off your stream before it has fully initialized and started and you try to `CREATE TABLE .. USING DELTA` ?
  - **A:** You will get this: `Error in SQL statement: AnalysisException: The user specified schema is empty;.`
- **Q:** When you do a write stream command, what does this option do `outputMode("append")` ?
  - **A:** This option takes on the following values and their respective meanings:
    - **append**: add only new records to output sink
    - **complete**: rewrite full output - applicable to aggregations operations
    - **update**: update changed records in place
- **Q:** What happens if you do not specify `option("checkpointLocation", pointer-to-checkpoint directory)`?
  - **A:** When the streaming job stops, you lose all state around your streaming job and upon restart, you start from scratch.
- **Q:** How do you view the list of active streams?
  - **A:** Invoke **`spark.streams.active`**
- **Q:** How do you verify whether `streamingQuery` is running (boolean output)?
  - **A:** Invoke **`spark.streams.get(streamingQuery.id).isActive`**

06-Optimization

- Q: Why are many small files problematic when doing queries on data backed by these?
  - A: If there are many files, some of whom may not be co-located the principal sources of slowdown are
    - network latency
    - (volume of) file metatadata
- Q: What do OPTIMIZE and VACUUM do?
  - OPTIMIZE creates the larger file from a collection of smaller files and
  - VACUUM deletes the invalid small files that were used in compaction.

- Q: What size files does OPTIMIZE compact to and why that value?
  - A: Small files are compacted to around **1GB**; this value was determined by the Spark optimization team as a good compromise between speed and performance.

- Q: What should one be careful of when using VACUUM?
  - A: Don't set a retention interval shorter than **seven days** because old snapshots and uncommitted files can still be in use by concurrent readers or writers to the table.

- Q: What does ZORDER do?
  - A: It is a technique to colocate related information in the same set of files.

07-Architecture

- **Q:** What is the difference between Lambda and Databricks Delta architecture?
  - **A:** The principal difference is that with Databricks Delta architecture, output queries can be performed on streaming and historical data at the same time. In Lambda architecture, streaming and historical data are treated as two separate branches feeding output queries.
- **Q:** What is role of raw (bronze) tables?
  - **A:** Raw tables capture streaming and historical data into a permanent record (streaming data tends to disappear after a short while). Though, it's generally hard to query.
- **Q:** What is role of query (silver) tables?
  - **A:** Query tables consist of normalized raw data that is easier to query.
- **Q:** What is role of summary (gold) tables?
  - **A:** Summary tables contain aggregated key business metrics that are queried frequently, but the silver queries themselves would take too long.

Review Check

1. What is the Databricks Delta command to display metadata?
MSCK DETAIL *tablename*
DESCRIBE DETAIL *tableName*
SHOW SCHEMA *tablename*

2. You just repaired myTable by using MSCK REPAIR TABLE *myTable*. How do you verify that the repair worked?
Use SELECT count(*) FROM *myTable* and make sure the count is what was expected.
Execute DESCRIBE DETAIL *myTable*.
display(*mytable*)

3. How do you perform UPSERT in a Databricks Delta dataset?
Use UPSERT INTO *my-table*.
Use UPSERT INTO *my-table* /MERGE.
Use MERGE INTO *my-table* USING *data-to-upsert*.

4. How do you view the list of active streams?
Invoke **spark.streams.active**.
Invoke **spark.streams.show**.
Invoke **spark.view.active**.

5. What size does OPTIMIZE compact small files to?
Around 100 MB.
Around 1 GB.
Around 500 MB.

Perform
Data Engineering with Azure Databricks

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hoseinalizadeh/)

Databricks Notebooks: https://github.com/MicrosoftDocs/mslearn-streaming-in-azure-databricks/blob/master/DBC/08-Streaming.dbc?raw=true

Streaming

- **Apache Spark Structured Streaming**
  - Apache Spark Structured Streaming is a fast, scalable, and fault-tolerant stream processing API. You can use it to perform analytics on your streaming data in near real time.
  - With Structured Streaming, you can use SQL queries to process streaming data in the same way that you would process static data. The API continuously increments and updates the final data.
- **Event Hubs and Spark Structured Streaming**
  - Azure Event Hubs is a scalable real-time data ingestion service that processes millions of data in a matter of seconds. It can receive large amounts of data from multiple sources and stream the prepared data to Azure Data Lake or Azure Blob storage.
  - Azure Event Hubs can be integrated with Spark Structured Streaming to perform processing of messages in near real time. You can query and analyze the processed data as it comes by using a Structured Streaming query and Spark SQL.
- **Streaming concepts**
  - Examples of such data include
    - bank card transactions,
    - Internet of Things (IoT) device data, and
    - video game play events.
  - Data coming from a stream is typically not ordered in any way.
  - A streaming system consists of:
    - Input sources such as Kafka, Azure event hubs, files on a distributed system, or TCP-IP sockets
    - Sinks such as Kafka, Azure event hubs, various file formats, forEach sinks, console sinks, or memory sinks
  - The problems of traditional data pipelines are exacerbated in streaming:
    - frequent metadata refreshes,
    - table repairs and small files accumulate every minute or second.
    - Many small files result because data might be streamed in at low volumes with short triggers.
  - Databricks Delta is uniquely designed to address these needs.

- 1. When you do a write stream command, what does the outputMode("append") option do?
  - The append mode allows records to be updated and changed in place.
  - **The append outputMode allows records to be added to the output sink.**
- 2. In Spark Structured Streaming, what method should you use to read streaming data into a DataFrame?
  - **spark.readStream**
  - spark.read

Create data visualizations by using Azure Databricks and Power BI

Databricks Notebooks: https://github.com/MicrosoftDocs/mslearn-create-data-visualizations-using-azure-databricks-and-power-bi/blob/master/DBC/07-Visualization.dbc?raw=true
Note: 04-Matplotlib has some interesting visualization using matplotlip including pandas com, seaborns heatmap, and pandas bar

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)

Connect Databricks and PowerBI - Review

- You can connect your Databricks cluster to Power BI to create visualization
- To expand your options for creating useful visualizations, Databricks supports the third-party Matplotlib libraryns for your data.

1. To use Power BI with DataBricks tables, which kind of connection do you need to establish first?

JDBC
ODBC
REST API direct connection

2. In notebooks, which function can be used for visualizing data in any language?

show()
visualize()
display()

3. Matplotlib is a library that's included with Databricks and doesn't need to be imported as a third-party library.

Correct
Incorrect

Databricks Notebooks: https://github.com/MicrosoftDocs/mslearn-create-data-visualizations-using-azure-databricks-and-power-bi/blob/master/DBC/07-Visualization.dbc?raw=true
Note: 04-Matplotlib has some interesting visualization using matplotlip including pandas com, seaborns heatmap, and pandas bar

Hosein Alizadeh - Principal Data Scientist at Microsoft
(https://www.linkedin.com/in/hosseinalizadeh/)