# Introduction to Apache Spark

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size.

**Provides:**

- Development APIs
  - Batch processing
  - Interactive queries
  - Real-time analytics
  - Machine learning
  - Graph processing

## Apache Spark vs. Apache Hadoop

Hadoop MapReduce is a programming model for processing big data sets with a parallel, distributed algorithm. With each step, MapReduce reads data from the cluster, performs operations, and writes the results back to HDFS. Because each step requires a disk read, and write, MapReduce jobs are slower due to the latency of disk I/O.

Spark was created to address the limitations of MapReduce. Spark does processing in-memory, reducing the number of steps in a job, and by reusing data across multiple parallel operations. With Spark, only one-step is needed where data is read into memory, operations performed, and the results written back.

## Resilient Distributed Dataset (RDD)

Data is kept in RAM (random access memory) instead of the slower disk drives. Fault Tolerance tracks data lineage information to allow for rebuilding lost data automatically on failure. Immutability simply rules out lots of potential problems due to various updates from varying threads at once. Having Immutable data is safer to share across processes.

### Lazy Evaluation

Transformation do not compute the results as and when stated.

### In-Memory Computation

Data is kept in RAM (random access memory) instead of the slower disk drives.

### Fault Tolerance

Tracks data lineage information to allow for rebuilding lost data automatically on failure.

### Immutability

Immutability simply rules out lots of potential problems due to various updates from varying threads at once. Having Immutable data is safer to share across processes.

## Two ways to apply operations on RDDs

- **Transformation**
  - These are the operations, which are applied on a RDD to create a new RDD. Filter, groupBy and map are the examples of transformations.
  - Because each step requires a disk read, and write, MapReduce jobs are slower due to the latency of disk I/O.

- **Wide Transformations**
  - Here, all elements required to compute the records in that single partition may live in many of the partitions of the parent RDD. These use groupByKey() and reduceByKey().

- **Narrow Transformations**
  - In this type, all the elements which are required to compute the records in a single partition live in that single partition.

### Create Dataframes

- **count()**, **collect()**, **take(n)**,** count value()**, **reduce()**, **fold()**, **aggregate()**, **foreach()**.
### DDL (cont)

<table>
<thead>
<tr>
<th>via CSV</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>df = spark.read.option(&quot;header&quot;, True) .csv(&quot;/tmp/resources/simple-zipcodes.csv&quot;)</td>
<td>Used to partition the large dataset (DataFrame) into smaller files based on one or multiple columns while writing to disk</td>
</tr>
</tbody>
</table>

If you have a header with column names on your input file, you need to explicitly specify True

```python
df = spark.read.csv("path1,path2,-path3")
df = spark.read.csv("Folder path")
```

Using the read.csv() method you can also read multiple csv files, just pass all file names by separating comma as a path

Using nullValues option you can specify the string in a CSV to consider as null. For example, if you want to consider a date column with a value "1900-01-01" set null on DataFrame.

```python
df.write.option("header", True) .partitionBy("state") .mode("overwrite") .csv("/tmp/zipcodes-state")
```

PySpark splits the records based on the partition column and stores each partition data into a sub-directory. If we have a total of 6 different states hence, it creates 6 directories

```python
df.write.option("header", True) .partitionBy("state","city") .mode("overwrite") .csv("/tmp/zipcodes-state")
```

It creates a folder hierarchy for each partition; we have mentioned the first partition as state followed by city hence, it creates a city folder inside the state folder (one folder for each city in a state).

### Queries

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

# Select Columns
df.select("firstName").show()
df.select("firstName","lastName") .show()

# split multiple array column
data into rows
df2 = df.select(df.name,explode(df.subjectandID))

# Show all entries where age >24
df.select(df['age'] > 24).show()

# Show name and 0 or 1 depending on age > or < than 30
df.select("Name", F.when(df.age > 30, 1).otherwise(0)) .show()

# Show firstName if in the given options
df[df.firstName.isin("Jane","Boris")).collect()

# Show firstName, and lastName if lastName is Smith.
df.select("firstName", df.lastName.like("Smith")) .show()
```
### Queries (cont)

- `df.select("firstName", df.lastName.like("%Sm")
  .show()`)  
  *Show firstName, and TRUE if lastName starts with Sm*

- `df.sel ect("fi rstName",
  df.lastNa me.l ike("%Sm")).show()`  
  *Show firstName, and TRUE if lastName starts with Sm*

- `df.sel ect(df.lastNa me.endswi tch("th")).show()`  
  *Show last names ending in th*

- `df.sel ect(df.firstNa me.substr(1, 3)
  .aliass("name"))
  .collect()`)  
  *Return substrings of firstName*

- `df.select(df.age.between(22, 24))
  .show()`)  
  *Show values where age is between 22 and 24*

- `df.select(df.\["firstName"], df["-age"]+ 1), .show()`  
  *Show all entries in firstName and age + 1*

### DML (cont)

- `df= df.na.drop(\(\text{how} = \text{any}, \text{thresh} = 2\))`  
  *To drop null values we use the drop() function with the \(\text{how}\) attribute.

  - `thresh: \text{default None if specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.}

  - `subset: \text{optional optional list of column names to consider.}`

- `df.na.fill(50)`)  
  *To fill nulls*

### Creating a Session

```python
import pyspark  # importing the module
from pyspark.sql import SparkSession

# creating a session
session = SparkSession.builder.appName('First App').getOrCreate()

# calling the session variable
```

### DML (cont)

- `unionDF = df.union(df2)`  
  *returns the new DataFrame with all rows from two Dataframes regardless of duplicate data.*

- `use the use the distinct() function to return just one record when duplicate exists().()`  
  *use the distinct() function to return just one record when duplicate exists.*

- `union() method of the DataFrame is used to merge two DataFrame's of the same structure/schema.*

- `to fill df.na.fill(50)`)  
  *To fill nulls*

- `from pyspark.sql import SparkSession`  
  *importing the SparkSession module*

- `import pyspark # importing the module
  from pyspark.sql import SparkSession
  # creating a session
  session = SparkSession.builder.appName('First App').getOrCreate()`  
  *calling the session variable

- `https://readable.com`  
  *Sponsored by Readable.com Measure your website readability!*

---

**Published 17th January, 2022.**  
**Last updated 28th February, 2022.**  
**Page 3 of 4.**
### Creating delta tables

- Define the input and output formats and paths and the table name.
- ```
read_format = 'delta'
write_format = 'delta'
load_path = '/databricks-datasets/learning-spark-v2/people/people-10m.delta'
save_path = '/tmp/delta/people-10m'
table_name = 'default.people10m'
```  
- Load the data from its source.
- ```
people = spark.read.format(read_format).load(load_path)
```  
- Write the data to its target.
- ```
people.write.format(write_format).save(save_path)
```  
- Create the table.
- ```
spark.sql("CREATE TABLE " + table_name + " USING DELTA LOCATION " + save_path + ")"
```  
### Data preprocessing (cont)

#### Existing column name to use for
(not necessary if the new column has nothing to do with the existing column)
- ```
# adding columns in dataframe
data = data.withColumn("Age_after_3_y", data["Age"]*3)
to change data type
You would also need cast() along with withColumn().
```  
#### The below statement changes the datatype from String to Integer for the salary column.
- ```
df.withColumn("salary",col("salary").cast("Integer")).show()
```  
#### Change a value
- Pass an existing column name as a first argument
- and a column as the value to be assigned as a second argument
- ```
df.withColumn("salary",col("salary")*100).show()
```  
#### Drop
- ```
df.drop("salary")
```  
#### Adding columns - `df.withColumn("newCol", newVal)`
#### Changing data types - `df.withColumn("newCol",col("OldCol").cast("NewDT")).show()`
#### Changing Values - `df.withColumn("oldCol", col("oldCol") operation)`
#### Dropping = `withColumnRenamed`
#### Renaming = `withColumnRenamed`

### Spark SQL

- ```
spark.sql(select * from tablename)
```