Intro:

**Distributed systems**:

- more than one machine (many set of cores), with one master node. (unlike local with limited core and, 1 unit ram and storage)

- can access computational resources across a no. of machines connected through a network.

- much easier to scale out. (Than to ‘scale up’ a single machine.)

- include fault tolerance.

**BIG DATA**:

- process data over a distributed network of machines.

- provides scale up and fault tolerance.

**Hadoop**:

Graphical user interface, text, application

Description automatically generated

"**HDFS**": Hadoop distributed File System.

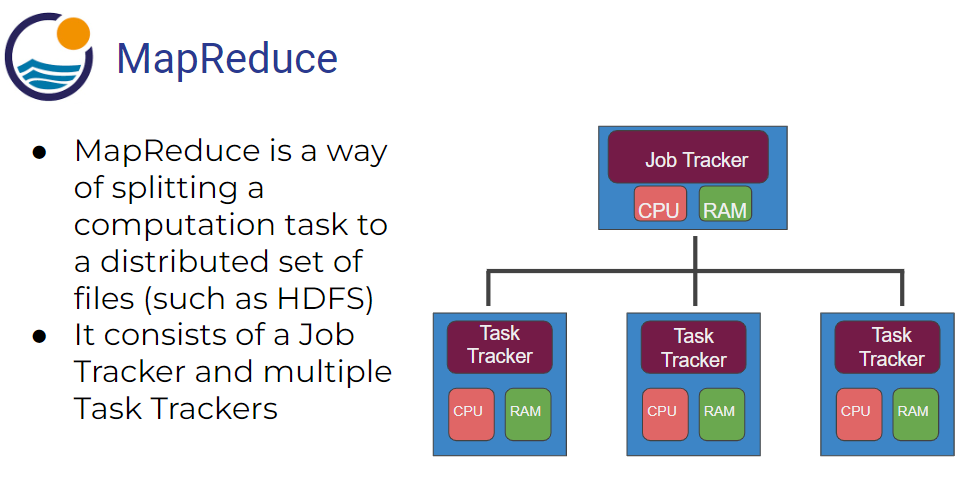
Diagram

Description automatically generated

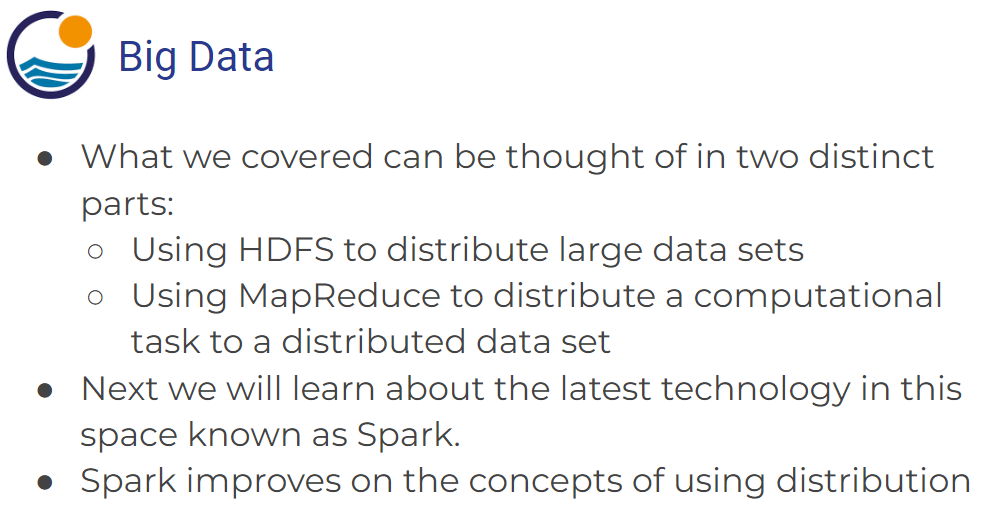
**MapReduce**:

- Allows computation on the HDFS data.

- Hadoop uses it.



**Hadoop Ecosystem**:



**SPARK Intro**:

- Spark: is a big data technology, a data processing framework. Address all limitations of MapReduce.

- It does not have its own file system like Hadoop HDFS, it supports most of all popular file systems.

- Can use Hadoop Distributed File System (HDFS), HBase, Cassandra, Amazon S3, Amazon Redshift, Couchbase, etc.

- All types data processing of Hadoop can be done in spark.

- It runs programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

- Considered a Flexible alternative to MapReduce.

- Don't require the files to be stored in HDFS (unlike Hadoop).

- \*A Framework for dealing with big (large) data, on distributed systems.

\*\*\* MLlib and PySpark are separate non-exclusive parts of the Spark big data domain?

**Spark *vs* MapReduce**:

- MapReduce requires files to be stored in HDFS, Spark does not.

- Spark also can perform operations up to 100x faster than MapReduce.

- MapReduce writes most data to disk after each map and reduce operation.

- Spark keeps most of the data in memory after each transformation.

- Spark can spill over to disk if the memory is filled.

\*\* Its spark vs MapReduce (not Hadoop, really).

**Spark RDDs** (Resilient Distributed Dataset):

- RDD has 4 main features: 1) Distributed Collection of Data, 2) Fault-tolerant, 3) Parallel operation - partitioned, 4) Ability to use many data sources.

- Resilient Distributed Dataset (RDDs) are immutable, lazily evaluated, and cacheable.

-

- There are two types of Spark operations: 1) Transformations, 2) Actions.

- 'Transformations' are basically a recipe to follow. 'Actions' actually perform what the recipe says to do and returns something back.

- Similarly, a lot of times you will write a "method" call, but won’t see anything as a result until you call the "action".

- \*\*With the release of Spark 2.0, Spark is moving towards a **DataFrame based syntax**, but keep in mind that the way files are being distributed can still be thought of as RDDs, it is just the typed-out syntax that is changing.

Text

Description automatically generated

Table

Description automatically generated

==============================================================

SPARK SETUP:

**Reason to use Linux based systems**:

- Realistically Spark won’t be running on a single machine, it will run on a cluster on a service, like AWS. These ‘cluster services’ will pretty much always be a Linux based system.

- Many options available: 1) Ubuntu+Spark+Python on VirtualBox; 2) Amazon EC2 with Python and Spark; 3) Databricks Notebook System; and 4) AWS EMR Notebook; etc.

-

**Environment Setup options:**

**Databricks** – no real installation required. Have its own storage format known as DBFS. (Table format).

- Create account, verify email, login.

- \*\*Create a cluster. Add name and pick Apache spark version.

- Create Notebook and attach cluster.

**Upload data to Databricks**:

- ‘Tables’ on the left is where you store the data.

- Click on ‘create table’ inside ‘tables’.

- Add the data source. (can add – file, s3 bucket, or sql db)

- upload table,

- preview table, add name, delimiter, select if first row is header or not.

- select/check data types for columns.

Access the table in the workspace/notebook:

To pull the table needs a bit of sql context code in notebook. {*maybe outdated?*}

(would need to import sqlContext module in non-Databricks environment.)

Call data into DataFrame:

=====================================================================

old

A picture containing timeline

Description automatically generated

GETTING TO SPARK FUNDAMENTALS:

**Spark DataFrames**:

- \*With the release of Spark 2.0, Spark is moving towards a 'DataFrame' based syntax.

- DataFrame is like pandas DF, excel sheets, with columns and rows.

- Spark DataFrames hold data in a column and row format. Each column represents some feature or variable. Each row represents an individual data point.

- Spark DataFrames can input and output data from a wide variety of sources.

- At the end of the transformation calls, we can either show or collect the results to display or for some final processing.

-

Works very much like pandas.

Start a spark session.

Read from a file.

Custom Schema Creation for bigger imported tables:

**Grab data from the DF:**

‘Select’ (*df.select()*) vs ‘Grab’ data from a DF table.

Select: can return a DF with a single column, (instead of a column), which gives more flexibility.

Can select multiple columns.

Use of “.show()” always to actually see the data elements (instead of their abstract definition).

Df.head(n) grabs ‘n’ rows from the DF.

Df has lots of objects available like rows and column objects.

\* Spark can read from distributed data source and map to distributed computing.

*Create new column*: df.withColumn(‘name’, df[‘column’]) – add column or replace an existing column.

- this is not an inplace operation, so original df don’t change. To do that you need to assign it to new variable (creates new df).

- can add various numerical operations (+, \*, /, etc.) to whole columns here as well.

Rename a column – df.withColumnRenamed(‘old\_name’, ‘new\_name’) .show()

Use sql to directly interact with the DF:

To use, register the dataFrame as a temporary sql temporary view:

df.createOrReplaceTempView(“table\_name”)

now can pass sql queries as string:

results = spark.sql(“SELECT \* FROM table\_name WHERE age=30”)

results.show()

**Spark DF basic Operations**:

Start with Import spark session (always):

From pyspark.sql import SparkSession

Spark = SparkSession.builder.appName(‘app\_name’).getOrCreate()

\*Csv file reading offers the option to infer schema. (not with json?)

df = spark.read.csv(‘filename.csv’, inferSchema=True, header=True)

df.printSchema()

**Filter Data based on conditions**:

\*Spark DF are built on top of spark SQL platform.

Using SQL example (sql syntax): (closing price < $500)

df.filter(“column\_name < 500”).show()

pick a single column after: df.filter(“column\_name < 500”).select(‘column\_name’).show()

Python use cases:

df.filter(df[‘column\_name’] < 500).show()

df.filter(df[‘column\_name’] < 500).select(‘column\_name’)show()

Filter on multiple conditions (\*Need to use Boolean operators):

df.filter( (df[‘column\_name’] < 200) & ~(df[‘column\_name’]>200) ).show()

\*Use brackets to separate the 2 conditions that use the Boolean operator.

Specific row instance:

df.filter(df[‘column\_name’] == 197).show()

“collect” the result data to use later (gives out a list):

result = df.filter(df[‘column\_name’] == 197).collect()

\*That row in the list can be converted to a Dict:

row = result[0]

row.asDict()

row.asDict()[‘item\_name’]

**GroupBy and Aggregate Operations (Functions)**:

Start with Import spark session (always):

From pyspark.sql import SparkSession

Spark = SparkSession.builder.appName(‘app\_name’).getOrCreate()

\* Group rows together based on some column value. Eg: group sales by the date they occurred.

\* Aggregate operator: Combine multiple rows data into a single output (can apply on a grouped set from above).

df.groupBy(‘column\_name’)

\*Can run a variety of methods of of this. (**sum, max, min, mean, count** (no. of rows),)

df.groupBy(‘column\_name’).mean()

\*Not all methods need a groupBy call first. Can just call generalized.agg method that aggregates across all rows in df. **(takes a dict ({}) in input**.)

##

df.agg({‘column\_name’: ‘operation\_name’}).show()

Alternate syntax:

group\_data = df.groupBy(“column\_name”)

group\_data.agg({‘column\_name’: ‘operation\_name’}).show()

* Does the same thing as above.
* A generalized approach, its useful in loops, etc. (passing var names).

**Import functions from spark**:

from pyspark.sql.functions import countDistinct, avg, stddev

\*combine them with select():

df.select( countDistinct(‘column\_name’) ).show()

use a different notation:

df.select( avg(‘column\_name’).alias(‘alias\_name’) ).show()

**Format the Data itself**:

df.select( stddev(‘column\_name’)).show()

numbers formatting:

from pyspark.sql.functions import format\_number

fix the new column name first:

sales\_std = df.select( stddev(‘column\_name’).alias(‘alias\_name’))

sales\_std.show()

fix the significant digits:

sales\_std.select(format\_number(‘column\_name’, num\_of\_significantdigits)).show()

\*have to call alias() at very end to keep the column name fixed:

sales\_std.select(format\_number(‘column\_name’, num\_of\_significantdigits).alias(‘alias\_name’) ).show()

**Order and Sort things**:

Order by a column (ascending):

df.orderBy(‘column\_name’).show()

for descending order (pass the actual column, not just its name):

df.orderBy(df[‘column\_name’].desc() ).show()

**Missing data**:

Handle missing data options:

1. Keep it as null,
2. Drop the entire row,
3. Fill with some values.

Spark session:

From pyspark.sql import SparkSession

Spark = SparkSession.builder.appName(‘app\_name’).getOrCreate()

Drop missing data Approach:

df.na.drop().show()

specify ‘threshold’ argument (drop only rows with at least 2 null values):

df.na.drop(thresh=2).show()

‘how’ parameter (=’any’, ‘all’):

df.na.drop(how=’all’).show()

‘Subset’ argument (subset columns lit):

df.na.drop(subset=[‘column\_name’]).show()

Fill the missing values:

df.na.fill(‘string\_val’).show()

\*’fill’ matches the column data type to the passed value data type and only fills for matching columns.

Pass integer to fill: df.na.fill(0).show()

Fill in for particular columns with the passed value, using ‘subset’ argument:

df.na.fill(‘string\_val’ , subset=[‘column\_name’]).show()

fill it with ‘mean’ value:

from pyspark.sql.functions import mean

mean\_val = df.select(mean(df[‘column\_name’])).collect()

mean\_sales = mean\_val[0][0]

df.na.fill(mean\_sales, [‘column\_name’]).show()

\*\* The Fill or Drop approach depends on case by case, no general rule, Algo or Objective will influence the decision.

**Dates and Timestamps**:

Import spark session.

Create a spark session.

Read the csv file in a df. (working with apple stock csv file)

from pyspark.sql.functions import (dayofmonth, hour, dayofyear, month,

year, weekofyear, format\_number, date\_format)

\*\*For any of pyspark.sql.functions: just do df.select() and call the function on the actual column(s), using the “[]” notation. Eg:

df.select(dayofmonth(df[‘column\_name’])).show()

* Gives a column of the day number for each element.

Hour (beginning of day expected): {^should return all zero here.}

df.select(hour(df[‘column\_name’])).show()

Average closing price per year:

Pick the year on the date column - # df.select(year(df[‘Date’])).show()

Create a new df adding a new column from that:

newdf = df.withColumn(“year”, year(df[‘Date’]))

pick the particular columns (with their new names) out after grouping:

result = newdf.groupBy(“year”).mean().select([“year”, ‘”avg(close)”])

Column Rename:

New = result.withColumnRenamed(“avg(Close)”, “Average closing price”)

New.select([‘year’, format\_number(“Average closing price”, 2).alias(“avg close”)]).show()

===================================

\*\*\* MLlib and PySpark are separate non-exclusive parts of the Spark big data domain?

**Intro to ML with MLib**:

**ML Use cases**:

Graphical user interface, text, application

Description automatically generated

**ML Process**:

Diagram

Description automatically generated

**ML (python + spark) with MLib Library**:

- Spark’s MLlib is mainly designed for Supervised and Unsupervised Learning tasks, with most of its algorithms falling under those two categories.

- **Supervised**: Methods like classification, regression, prediction and gradient boosting, supervised learning uses patterns to predict the values. used in applications where historical data predicts likely future events.

- **Unsupervised**: The goal is to explore the data and find some structure within. system is not told the "right answer". Techniques include self-organizing maps, nearest-neighbor mapping, k-means clustering and singular value decomposition (SVD). One issue is that it can be difficult to evaluate results of an unsupervised model.

**ML with SPARK (using MLlib)**:

- Spark has its own MLlib (ML Library) for Machine Learning. MLlib utilizes the Spark 2.0 DataFrame syntax.

- For MLlib, you need to format your data so that eventually it has just one or two columns:

* Features, Labels (Supervised), or
* Features (Unsupervised)

- Its big upside is that this exact same syntax (after above data processing) works with distributed data, which is no small feat for what is going on “under the hood”!

- Documentation examples are always with nicely formatted data.

-\* Learn to turn the regular/real dataset into features column.

\* *API docs* (deep dive into how fun works) are different from Spark *Programming Guide* documentation.

===========================================

**Linear Regression**:

Evaluate regression models.

======================================================================

- Spark SQL, DataFrames and Datasets Guide: <https://spark.apache.org/docs/latest/sql-programming-guide.html>

- Machine Learning Library (MLlib) Guide: <https://spark.apache.org/docs/latest/ml-guide.html>

**PySpark**: - an API of Apache Spark-distributed processing system.

ML in big data.

Spark Streaming