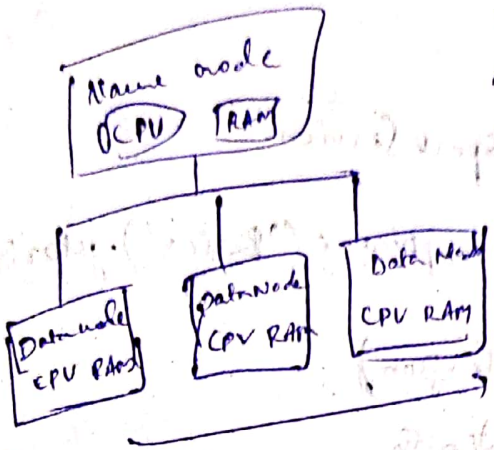
Spark: (It is handle to Big Data) use to

→ Hadoop Distributed File System (HDFS) (Distribute large dataset)

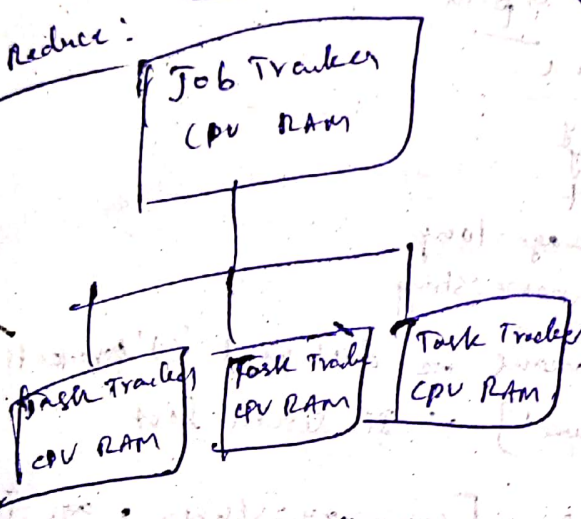
↓
uses because of fault tolerance

↓
If we use only local machine, if data is lost then we loose all data. Coming to HDFS, it distributes data to all processes and maintain availability of data



(Distribute a Computational task to a distributed dataset).

Map Reduce:



Spark vs MapReduce?

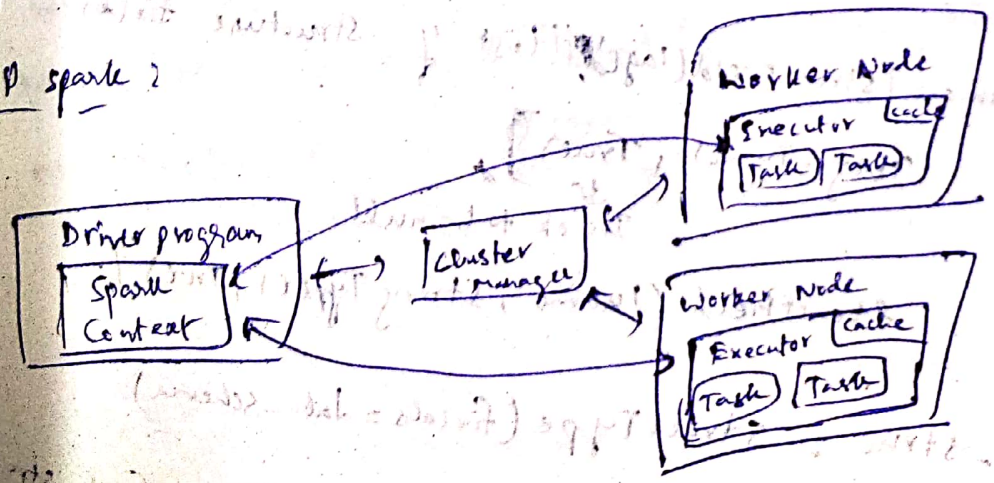
↓

100x faster than mapReduce.

→ Spark keeps most of the data in memory

→ It writes most of data to disk after each map and reduce operations

RDD spark?



Python + Spark

Column → Some feature / variable
row → Individual datapoint

Spark dataframes Basics:

- from pyspark.sql import SparkSession
- spark = SparkSession.builder.appName('Basics').getOrCreate()

df = spark.read.json('people.json') // path.

df.show() //

age	name
30	Andy
20	Ravi

df.printSchema() //
root
|-age : long
|-name : string

df.columns // attribute. means we dont need () brackets
['age', 'name'] → It return list

df.describe() // Dataframe[summary : string, age : int]

df.describe().show()

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

data_schema = [StructField('age', IntegerType(), True), // list of structure fields
StructField('name', StringType(), True)]
↓ Its ok to be null.

final_struc = StructType(fields = data_schema)

df = spark.read.json('people.json', schema = final_struc)

df.printSchema() //
root
|-- age : integer
|-- name : string

type(df['age']) # pyspark.sql.column.Column

df.select('age') # DataFrame[age: int] → It returns object

df.select('age').show() # age
null
30
19
→ It returns dataframe.

df.head(2)[0] # Row(age=None, name='Tony')

df.select(['age', 'name']) # multiple columns.

df.withColumn('newage', df['age']).show() # Creating new Column.

↳ # age name newage
null Tony null
20 Ruby 20

df.withColumn('double_age', df['age']*2).show()

↳ It doubles the age

df.withColumnRenamed('age', 'my_new_age').show()

SQL?

df.createOrReplaceTempView('people') # Registered as Sql Temp view

results = spark.sql("select * from people")

results.show() # table

new_results = spark.sql("select * from people where age=30")

new_results.show() # age name
30 Andy

df = spark.read.csv('appl_stock.csv', inferSchema=True, header=True)

df.head(3) // first three rows

df.filter("Close < 500").show()
↑ Column name.

- df.filter("Close < 500").select(['Open', 'Close']).show()
  // it gives the columns of open close where close < 500.

- df.filter(df['Close'] < 500).show()

- df.filter((df['Close'] < 200) & (df['Open'] > 200)).show()

result=
- df.filter(df['Low'] == 197).collect()
  // it returns a list

- row = result[0]

- row.asDict() // converts list to dict

- row.asDict()['Volume']
  ↓ one of the column name

groupby & Aggregate operations:

- df.groupBy("Company").mean().show()
  ↑ Column name
  sum()
  max()
  min()
  count()

- df.agg({'Sales': 'sum'}).show() // sum(Sales) 870
  ↓ Column name; ↓ function name (max, min) etc

- group_data = df.groupBy("Company")
  group_data.agg({'Sales': 'max'}).show()

Company	max(Sales)
Apple	350
google	858

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

- df.select(avg('sales')).show() // [avg(sales) / 33.33] (Average_Sales)
  alias('Average_Sales')

- df.orderBy(df['sales'].desc()).show() → Descending order

- df.orderBy("sales").show() → Ascending order

Missing Data:

~~df.dropna()~~ df.na.drop().show() (thresh=2 (means, if row has 2 null values, then drop that row))

- df.na.drop(how='all').show()
  'any' → (By default it set as 'any' means If row has any null value then drop it)

- df.na.drop(subset=['sales']).show()

- df.na.fill('FILL VALUE').show()
  0 → (It fills in string type
  It fills in int type.

- df.na.fill('No name', subset=['Name']).show()
  ⇒ (If name has null value, then No name is filled)

Dates & Time Stamps:

- from pyspark.sql.functions import dayofmonth, hour, dayofyear, month, year, weekofyear, format_number, date_format

- df.select(dayofmonth(df['Date'])).show()
  hour
  month

- df.withColumn("Year", year(df['Date'])).show()

- new df.groupBy("Year").mean().select(["Year", "avg(close)"]).show()