Big Data Analytics with Hadoop & Apache Spark

Contents

Introduction to Big Data

Big Data refers to datasets that are too large, too fast, or too complex for traditional data-processing tools.

The famous **5 V's**:

- **Volume** – massive amounts of data (GB $\rightarrow$ TB $\rightarrow$ PB).
- **Velocity** – fast incoming data (real-time streams).
- **Variety** – structured, semi-structured, unstructured.
- **Veracity** – noise, uncertainty in data.
- **Value** – extracting useful insights.

Big Data exists when *your current tools are insufficient* to store or process the data in a reasonable time.

Hadoop Ecosystem

What is Hadoop?

Hadoop is an open-source ecosystem enabling distributed storage and processing of massive datasets.

Main components:

- **HDFS** – distributed storage
- **MapReduce** – batch computation model
- **YARN** – cluster resource manager

HDFS Architecture

HDFS stores files by splitting them into large blocks (default: 128MB) distributed across multiple machines.

Key parts:

- **NameNode** – stores metadata (filesystem tree).
- **DataNode** – stores actual data blocks.
- **Replication** – default 3 copies per block.

MapReduce Model

MapReduce is a two-step programming model: **Map** $\rightarrow$ **Shuffle** $\rightarrow$ **Reduce**

Example: Word Count

- Map: Output (word, 1)
- Reduce: Sum counts for each word

Apache Spark

Why Spark is Faster

Spark keeps data **in-memory** and executes workflows using an optimized **DAG** engine — often up to 100$\times$ faster than classic MapReduce.

Advantages:

- In-memory computing
- Lazy evaluation
- Rich APIs (DataFrames, SQL, MLlib, GraphX)
- Works with HDFS, S3, Kafka

Spark Programming Model

Types of operations:

- **Transformations** (lazy): `map`, `filter`, `groupBy`
- **Actions**: `count()`, `collect()`, `show()`

PySpark Examples (Python)

Creating a Spark Session

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("BigDataNotes") \
    .master("local[*]") \
    .getOrCreate()
```

Reading CSV Data

```
df = spark.read.csv("data/people.csv",
                    header=True, inferSchema=True)

df.show()
df.printSchema()
```

Filtering and Selecting Data

```
adults = df.filter(df.age > 30).select("name", "age")
adults.show()
```

Group By Aggregation

```
from pyspark.sql import functions as F

city_counts = df.groupBy("city").agg(
    F.count("*").alias("population")
)

city_counts.show()
```

Word Count with RDDs

```
text = spark.sparkContext.textFile("data/book.txt")

counts = (text.flatMap(lambda line: line.split())
              .map(lambda w: (w.lower(), 1))
              .reduceByKey(lambda a, b: a + b))

counts.take(10)
```

Spark SQL Example

```
df.createOrReplaceTempView("people")

result = spark.sql("""
    SELECT city, AVG(age) as avg_age
    FROM people
    GROUP BY city
""")

result.show()
```

Summary

Hadoop = distributed storage + batch processing **Spark** = fast, in-memory distributed computation Together they form the backbone of modern Big Data systems.

Additional Pedagogical Notes and Code Examples

This section contains extra explanations and many code examples in PySpark to help you learn by doing.

Big Data and HDFS in Practice

Idea: Instead of one "super" machine, we use many cheap machines that work together and store data in HDFS.

Conceptual steps when storing a large file in HDFS:

1. File is split into blocks (e.g. 128MB).
2. Each block is replicated (default 3 copies).
3. Blocks are distributed across several DataNodes.
4. NameNode keeps metadata: which file has which blocks and on which nodes.

Typical HDFS shell commands:

```
# List root directory in HDFS
hdfs dfs -ls /

# Make a directory in HDFS
hdfs dfs -mkdir -p /user/gianna/data

# Upload a local file into HDFS
hdfs dfs -put people.csv /user/gianna/data/

# Show the file content from HDFS
hdfs dfs -cat /user/gianna/data/people.csv

# Remove a file
hdfs dfs -rm /user/gianna/data/people.csv
```

Working with RDDs (Low-Level API)

RDD = Resilient Distributed Dataset. It is a distributed list of objects that you can transform with `map`, `filter`, `reduceByKey`, etc.

```python
# Create RDD from a Python list
numbers = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

# Map: multiply each number by 10
times_ten = numbers.map(lambda x: x * 10)

# Filter: keep only even numbers
even_numbers = times_ten.filter(lambda x: x % 2 == 0)

print("Even numbers:", even_numbers.collect())
```

Word count with RDDs (similar to MapReduce):

```python
text_rdd = spark.sparkContext.textFile("data/book.txt")

wc = (text_rdd
      .flatMap(lambda line: line.split())
      .map(lambda w: (w.lower(), 1))
      .reduceByKey(lambda a, b: a + b))

# Show first 20 (word, count) pairs
for word, count in wc.take(20):
    print(word, count)
```

Creating DataFrames in Different Ways

```
from pyspark.sql import Row

# 1) From a list of Python dicts
data = [
    {"name": "Alice", "age": 30, "city": "Brussels"},
    {"name": "Bob",   "age": 25, "city": "Paris"},
    {"name": "Chloe", "age": 35, "city": "Brussels"},
]

df1 = spark.createDataFrame(data)
df1.show()

# 2) From a list of Rows
rows = [
    Row(name="David", age=40, city="Madrid"),
    Row(name="Eva",   age=28, city="Athens")
]
df2 = spark.createDataFrame(rows)
df2.show()

# 3) From CSV file with inferred schema
df3 = (spark.read
            .option("header", True)
            .option("inferSchema", True)
            .csv("data/people.csv"))
df3.show()
df3.printSchema()
```

DataFrame: Select, Filter, New Columns

```
from pyspark.sql import functions as F

df = df3  # assuming df3 from previous example

# Select a subset of columns
df.select("name", "age").show()

# Filter rows: age > 30
df.filter(df.age > 30).show()

# Multiple conditions (AND / OR)
df.filter((df.age > 25) & (df.city == "Brussels")).show()

# Add a new column (computed)
df2 = df.withColumn("age_in_10_years", df.age + 10)
df2.show()

# Rename a column
df_renamed = df2.withColumnRenamed("age", "current_age")
df_renamed.show()
```

GroupBy & Aggregations (More Examples)

```
from pyspark.sql import functions as F

# Count how many people per city
df.groupBy("city").agg(
    F.count("*").alias("n_people")
).show()

# Average and max age per city
df.groupBy("city").agg(
    F.avg("age").alias("avg_age"),
    F.max("age").alias("max_age")
).show()

# Filter on aggregated result (e.g. cities with avg_age > 30)
agg = df.groupBy("city").agg(
    F.avg("age").alias("avg_age")
)

agg.filter(agg.avg_age > 30).show()
```

Joins Between DataFrames

Imagine two datasets:

- `customers(id, name, city)`
- `orders(order_id, customer_id, amount)`

```
customers = spark.read.csv("data/customers.csv",
                           header=True, inferSchema=True)
orders = spark.read.csv("data/orders.csv",
                        header=True, inferSchema=True)

# Inner join on customer id
joined = orders.join(customers,
                     orders.customer_id == customers.id,
                     "inner")

joined.select("order_id", "name", "amount", "city").show()

# Left join: keep all orders even if no customer match
left_join = orders.join(customers,
                        orders.customer_id == customers.id,
                        "left")

left_join.show()
```

User-Defined Functions (UDFs)

Use a UDF when a transformation cannot easily be expressed with built-in functions.

```python
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType

# Python function
def categorize_age(age):
    if age is None:
        return "unknown"
    if age < 18:
        return "child"
    elif age < 30:
        return "young adult"
    elif age < 60:
        return "adult"
    else:
        return "senior"

# Convert Python function to UDF
age_category_udf = udf(categorize_age, StringType())

df_with_cat = df.withColumn("age_category",
                            age_category_udf(col("age")))

df_with_cat.select("name", "age", "age_category").show()
```

Window Functions (Running Totals, Rankings)

Window functions let you compute things like "running sum", "rank per group", "previous value", etc.

Example: sales per day and running total per city.

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

sales = spark.read.csv("data/daily_sales.csv",
                       header=True, inferSchema=True)
# Columns: date, city, amount

w = Window.partitionBy("city").orderBy("date")

sales_with_running = sales.withColumn(
    "running_total",
    F.sum("amount").over(w)
)

sales_with_running.show()
```

Spark SQL: More Complex Queries

```
df.createOrReplaceTempView("people")

# Cities with at least 2 people and average age > 30
result = spark.sql("""
    SELECT city,
           COUNT(*) AS n_people,
           AVG(age) AS avg_age
    FROM people
    GROUP BY city
    HAVING COUNT(*) >= 2 AND AVG(age) > 30
    ORDER BY avg_age DESC
""")

result.show()
```

Machine Learning with MLlib: Classification

Workflow idea: **DataFrame** → **features vector** → **model fit** → **predictions**.

Example: simple logistic regression.

```
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

# Suppose df has: label (0/1), age, income, balance
data = spark.read.csv("data/bank.csv",
                      header=True, inferSchema=True)

# Features into one vector column
assembler = VectorAssembler(
    inputCols=["age", "income", "balance"],
    outputCol="features"
)

# Model
lr = LogisticRegression(featuresCol="features",
                        labelCol="label")

pipeline = Pipeline(stages=[assembler, lr])

train, test = data.randomSplit([0.7, 0.3], seed=42)

model = pipeline.fit(train)

predictions = model.transform(test)

predictions.select("age", "income", "balance",
                   "label", "prediction", "probability").show(20,
                      truncate=False)
```

MLlib: Regression Example

```python
from pyspark.ml.regression import LinearRegression

housing = spark.read.csv("data/housing.csv",
                         header=True, inferSchema=True)
# Example columns: price, rooms, size, distance_to_center

assembler = VectorAssembler(
    inputCols=["rooms", "size", "distance_to_center"],
    outputCol="features"
)

housing_vec = assembler.transform(housing)

train, test = housing_vec.randomSplit([0.8, 0.2], seed=1)

lr = LinearRegression(featuresCol="features",
                      labelCol="price")

lr_model = lr.fit(train)

print("Coefficients:", lr_model.coefficients)
print("Intercept:", lr_model.intercept)

pred = lr_model.transform(test)
pred.select("rooms", "size", "distance_to_center",
            "price", "prediction").show(10)
```

MLlib: K-Means Clustering

```
from pyspark.ml.clustering import KMeans

points = spark.read.csv("data/points2d.csv",
                        header=True, inferSchema=True)
# Columns: x, y

assembler = VectorAssembler(
    inputCols=["x", "y"],
    outputCol="features"
)

points_vec = assembler.transform(points)

kmeans = KMeans(k=3, seed=1)
model = kmeans.fit(points_vec)

centers = model.clusterCenters()
print("Cluster centers:")
for c in centers:
    print(c)

clustered = model.transform(points_vec)
clustered.select("x", "y", "prediction").show(20)
```

Spark Structured Streaming: Simple Example

Structured Streaming treats streaming data as an unbounded table. We write queries that are continuously updated.

Example: word count from a socket.

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

# In a terminal, run: nc -lk 9999
# Then type lines of text.

lines = (spark.readStream
              .format("socket")
              .option("host", "localhost")
              .option("port", 9999)
              .load())

words = lines.select(F.explode(F.split(lines.value, " ")).alias("
   word"))

word_counts = words.groupBy("word").count()

query = (word_counts.writeStream
                    .outputMode("complete")
                    .format("console")
                    .start())

query.awaitTermination()
```

Mini ETL Pipeline Example

ETL = Extract → Transform → Load. Spark is often used to build ETL pipelines on Big Data.

Goal: Read raw CSV of transactions, clean data, aggregate by customer, and write result as Parquet.

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

# 1. Extract
raw = spark.read.csv("data/transactions.csv",
                     header=True, inferSchema=True)
# Columns: customer_id, date, amount, country

# 2. Transform
clean = (raw
         .filter(raw.amount.isNotNull())
         .filter(raw.customer_id.isNotNull()))

aggregated = (clean
              .groupBy("customer_id")
              .agg(
                  F.count("*").alias("n_tx"),
                  F.sum("amount").alias("total_spent"),
                  F.avg("amount").alias("avg_ticket")
              ))

# 3. Load (save)
(aggregated.write
          .mode("overwrite")
          .parquet("output/customers_aggregated"))

# You can later read:
result = spark.read.parquet("output/customers_aggregated")
result.show()
```

Try to re-run the ETL pipeline with different filters or new columns. You learn Spark best by experimenting and breaking things yourself.