PySpark Tutorial for Data Engineering

Table of Contents

1. What is Apache Spark?

Apache Spark is a unified analytics engine for large-scale data processing. It's designed to be fast (up to 100x faster than Hadoop MapReduce), easy to use, and capable of handling various workloads, including:

- **Batch Processing**: Processing large datasets in chunks
- **Stream Processing**: Real-time data processing
- **Machine Learning**: MLlib for scalable ML algorithms
- **Graph Processing**: GraphX for graph computations
- **SQL Analytics**: Spark SQL for structured data queries

Why Spark for Data Engineering?

1. **Speed**: In-memory computing and optimized execution engine
2. **Ease of Use**: High-level APIs in Python, Scala, Java, R
3. **Unified Platform**: Single platform for batch, streaming, ML, and analytics
4. **Fault Tolerance**: Automatic recovery from node failures
5. **Scalability**: Runs from laptops to thousands of machines

2. Key Concepts

1. Cluster Architecture

- **Driver Program**: Coordinates the Spark application
- **Cluster Manager**: Allocates resources (YARN, Mesos, Kubernetes, Standalone)
- **Executors**: Worker processes that run tasks and store data
- **Tasks**: Units of work sent to executors

2. Core Data Abstractions

- **RDD (Resilient Distributed Dataset)**: Low-level, immutable distributed collection
- **DataFrame**: Higher-level abstraction with schema, similar to a table
- **Dataset**: Type-safe version of DataFrame (mainly used in Scala/Java)

3. Lazy Evaluation

Transformations are not executed immediately but only when an action is called. This allows Spark to optimize the entire pipeline.

4. Transformations vs Actions

- **Transformations**: Operations that create new RDD/DataFrame (map, filter, join)
- **Actions**: Operations that trigger computation and return results (collect, count, save)

3. Setting Up PySpark

Local Installation

python
```
# Install using pip
pip install pyspark

# Or using conda
conda install pyspark

# Import PySpark
from pyspark.sql import SparkSession
from pyspark import SparkContext, SparkConf
```

Creating Spark Session

python

```python
from pyspark.sql import SparkSession

# Create Spark session (unified entry point)
spark = SparkSession.builder \
    .appName("MyDataEngineeringApp") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.executor.memory", "2g") \
    .config("spark.executor.cores", "2") \
    .getOrCreate()

# Get Spark Context from session
sc = spark.sparkContext
```

4. Core Components

SparkSession

The entry point for DataFrame and SQL functionality:

python
```python
# Most common way to start
spark = SparkSession.builder.appName("MyApp").getOrCreate()

# With specific configurations
spark = SparkSession.builder \
    .appName("DataEngineering") \
    .config("spark.sql.warehouse.dir", "/path/to/warehouse") \
    .enableHiveSupport() \
    .getOrCreate()
```

SparkContext

Lower-level functionality and RDD operations:

python
```python
sc = spark.sparkContext
# Set log level
sc.setLogLevel("WARN")
```

5. Working with RDDs

RDDs are the fundamental building blocks of Spark, though DataFrames are preferred for most use cases.

Creating RDDs

python
```
# From Python collections
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)

# From text files
text_rdd = sc.textFile("path/to/file.txt")

# From multiple files
files_rdd = sc.wholeTextFiles("path/to/directory/*")
```

Basic RDD Operations

python
```
# Transformations
filtered_rdd = rdd.filter(lambda x: x > 2)
mapped_rdd = rdd.map(lambda x: x * 2)
flat_mapped_rdd = text_rdd.flatMap(lambda line: line.split(" "))

# Actions
count = rdd.count()
first_element = rdd.first()
all_elements = rdd.collect()  # Be careful with large datasets!
sample = rdd.take(10)
```

6. DataFrames and Datasets

DataFrames are the preferred abstraction for structured data processing.

Creating DataFrames

python
```
# From Python data structures
data = [("Alice", 25), ("Bob", 30), ("Charlie", 35)]
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)

# From RDD
rdd = sc.parallelize(data)
```

```python
df = spark.createDataFrame(rdd, columns)

# From files (more common in data engineering)
df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
df = spark.read.json("path/to/file.json")
df = spark.read.parquet("path/to/file.parquet")
```

DataFrame Schema

python
```python
# Print schema
df.printSchema()

# Get schema programmatically
schema = df.schema
print(schema)

# Define schema explicitly
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.createDataFrame(data, schema)
```

Basic DataFrame Operations

python
```python
# Show data
df.show()
df.show(20, truncate=False)

# Get basic info
df.count()
df.columns
df.dtypes
df.describe().show()

# Select columns
df.select("name", "age").show()
df.select("*").show()
```

```python
# Add/rename columns
from pyspark.sql.functions import col, lit
df = df.withColumn("age_plus_10", col("age") + 10)
df = df.withColumnRenamed("name", "full_name")

# Drop columns
df = df.drop("age_plus_10")
```

7. Data Sources and I/O

Reading Data

python

```python
# CSV with options
df = spark.read.option("header", "true") \
    .option("inferSchema", "true") \
    .option("delimiter", ",") \
    .csv("path/to/file.csv")

# JSON
df = spark.read.json("path/to/file.json")
# Multi-line JSON
df = spark.read.option("multiline", "true").json("path/to/file.json")

# Parquet (columnar format, very efficient)
df = spark.read.parquet("path/to/file.parquet")

# Database (JDBC)
df = spark.read.format("jdbc") \
    .option("url", "jdbc:postgresql://localhost/test") \
    .option("dbtable", "employees") \
    .option("user", "username") \
    .option("password", "password") \
    .load()

# Delta Lake (if available)
df = spark.read.format("delta").load("path/to/delta/table")
```

Writing Data

python

```python
# CSV
df.write.mode("overwrite").option("header", "true").csv("path/to/output")
```

```python
# Parquet (recommended for data lakes)
df.write.mode("append").parquet("path/to/output")

# JSON
df.write.mode("overwrite").json("path/to/output")

# Database
df.write.format("jdbc") \
    .option("url", "jdbc:postgresql://localhost/test") \
    .option("dbtable", "employees") \
    .option("user", "username") \
    .option("password", "password") \
    .mode("append") \
    .save()

# Partitioned writes (important for performance)
df.write.partitionBy("year", "month").parquet("path/to/partitioned/data")
```

Write Modes

- **overwrite**: Replace existing data
- **append**: Add to existing data
- **ignore**: Skip if data exists
- **error**: Fail if data exists (default)

8. Data Transformations

Filtering

python

```python
# Simple filters
df.filter(col("age") > 25).show()
df.filter("age > 25").show()  # SQL-style string
df.where(col("age") > 25).show()  # where is alias for filter

# Multiple conditions
df.filter((col("age") > 25) & (col("name").startswith("A"))).show()
df.filter((col("age") > 25) | (col("age") < 20)).show()

# Null checks
df.filter(col("name").isNotNull()).show()
df.filter(col("name").isNull()).show()
```

Selecting and Projecting

python
```
from pyspark.sql.functions import *

# Select specific columns
df.select("name", "age").show()

# Select with expressions
df.select(col("name"),
        col("age"),
        (col("age") + 5).alias("age_plus_5")).show()

# Select with functions
df.select(upper(col("name")).alias("upper_name"),
        round(col("salary"), 2).alias("rounded_salary")).show()
```

Adding and Modifying Columns

python
```
# Add new column
df = df.withColumn("age_category",
                when(col("age") < 30, "Young")
                .when(col("age") < 50, "Middle")
                .otherwise("Senior"))

# Modify existing column
df = df.withColumn("age", col("age") + 1)

# Add current timestamp
df = df.withColumn("processed_at", current_timestamp())

# Add literal values
df = df.withColumn("country", lit("USA"))
```

String Operations

python
```
from pyspark.sql.functions import *

# String functions
df.select(
    upper(col("name")),
    lower(col("name")),
```

```
    length(col("name")),
    substring(col("name"), 1, 3),
    trim(col("name")),
    regexp_replace(col("name"), "pattern", "replacement")
).show()

# String splitting
df = df.withColumn("name_parts", split(col("full_name"), " "))
df = df.withColumn("first_name", col("name_parts").getItem(0))
```

Date and Time Operations

python

```
from pyspark.sql.functions import *

# Current date/time
df = df.withColumn("current_date", current_date())
df = df.withColumn("current_timestamp", current_timestamp())

# Date parsing and formatting
df = df.withColumn("parsed_date", to_date(col("date_string"), "yyyy-MM-dd"))
df = df.withColumn("formatted_date", date_format(col("date_col"), "MM/dd/yyyy"))

# Date arithmetic
df = df.withColumn("date_plus_30", date_add(col("date_col"), 30))
df = df.withColumn("days_diff", datediff(col("end_date"), col("start_date")))

# Extract date parts
df = df.withColumn("year", year(col("date_col")))
df = df.withColumn("month", month(col("date_col")))
df = df.withColumn("day_of_week", dayofweek(col("date_col")))
```

9. Data Aggregations

Basic Aggregations

python

```
from pyspark.sql.functions import *

# Simple aggregations
df.agg(
    count("*").alias("total_count"),
    sum("salary").alias("total_salary"),
```

```
    avg("salary").alias("avg_salary"),
    min("age").alias("min_age"),
    max("age").alias("max_age"),
    stddev("salary").alias("salary_stddev")
).show()

# Group by aggregations
df.groupBy("department").agg(
    count("*").alias("employee_count"),
    avg("salary").alias("avg_salary"),
    max("salary").alias("max_salary")
).show()

# Multiple group by columns
df.groupBy("department", "location").agg(
    count("*").alias("count"),
    sum("salary").alias("total_salary")
).show()
```

Advanced Aggregations

python

```
# Collect list/set
df.groupBy("department").agg(
    collect_list("employee_name").alias("employees"),
    collect_set("skill").alias("unique_skills")
).show()

# Percentiles and quantiles
df.agg(
    expr("percentile_approx(salary, 0.5)").alias("median_salary"),
    expr("percentile_approx(salary, array(0.25, 0.5, 0.75))").alias("quartiles")
).show()

# Count distinct
df.agg(countDistinct("department").alias("unique_departments")).show()
```

Pivot Operations

python

```
# Pivot table
pivot_df = df.groupBy("year").pivot("department").agg(sum("sales"))
pivot_df.show()
```

```python
# Unpivot (melt) - more complex, requires stack function
unpivot_df = df.select(
    "id", "name",
    expr("stack(3, 'Q1', Q1, 'Q2', Q2, 'Q3', Q3) as (quarter, sales)")
).where("sales is not null")
```

10. Joins and Unions

Inner Joins

python

```python
# Inner join (default)
result = df1.join(df2, df1.id == df2.id, "inner")
result = df1.join(df2, "id")  # If column names are the same

# Multiple conditions
result = df1.join(df2,
                  (df1.id == df2.id) & (df1.date == df2.date),
                  "inner")
```

Different Join Types

python

```python
# Left outer join
result = df1.join(df2, "id", "left_outer")

# Right outer join
result = df1.join(df2, "id", "right_outer")

# Full outer join
result = df1.join(df2, "id", "outer")

# Cross join (Cartesian product)
result = df1.crossJoin(df2)

# Anti join (rows in df1 not in df2)
result = df1.join(df2, "id", "left_anti")

# Semi join (rows in df1 that have matches in df2)
result = df1.join(df2, "id", "left_semi")
```

Handling Join Column Name Conflicts

```python
# When both DataFrames have columns with same names
df1_aliased = df1.alias("a")
df2_aliased = df2.alias("b")

result = df1_aliased.join(df2_aliased,
                          col("a.id") == col("b.id"),
                          "inner") \
                    .select("a.id", "a.name", "b.salary")
```

Unions

```python
# Union (includes duplicates)
result = df1.union(df2)

# Union by name (matches columns by name, not position)
result = df1.unionByName(df2)

# Remove duplicates after union
result = df1.union(df2).distinct()
```

11. Window Functions

Window functions operate on a group of rows and return a single value for each row.

Basic Window Operations

```python
from pyspark.sql.window import Window
from pyspark.sql.functions import *

# Define window specification
window_spec = Window.partitionBy("department").orderBy("salary")

# Row number
df = df.withColumn("row_number", row_number().over(window_spec))

# Rank functions
df = df.withColumn("rank", rank().over(window_spec))
df = df.withColumn("dense_rank", dense_rank().over(window_spec))

# Lag and Lead
```

```
df = df.withColumn("prev_salary", lag("salary", 1).over(window_spec))
df = df.withColumn("next_salary", lead("salary", 1).over(window_spec))
```

Aggregate Window Functions

```
python
# Running totals and averages
window_unbounded = Window.partitionBy("department") \
                         .orderBy("date") \
                         .rowsBetween(Window.unboundedPreceding, Window.currentRow)

df = df.withColumn("running_total", sum("amount").over(window_unbounded))
df = df.withColumn("running_avg", avg("amount").over(window_unbounded))

# Moving averages
window_moving = Window.partitionBy("department") \
                      .orderBy("date") \
                      .rowsBetween(-2, 0)  # Previous 2 rows + current


df = df.withColumn("moving_avg_3", avg("amount").over(window_moving))
```

Percentile Functions

```
python
# Percentile rank
df = df.withColumn("percentile_rank",
                   percent_rank().over(window_spec))

# N-tile (quartiles, deciles, etc.)
df = df.withColumn("quartile", ntile(4).over(window_spec))
```

12. Performance Optimization

Caching and Persistence

```
python
# Cache in memory
df.cache()
df.persist()

# Specify storage level
from pyspark import StorageLevel
df.persist(StorageLevel.MEMORY_AND_DISK)
```

```
df.persist(StorageLevel.MEMORY_ONLY_SER)

# Remove from cache
df.unpersist()
```

Partitioning

```
python
# Check current partitions
print(f"Number of partitions: {df.rdd.getNumPartitions()}")

# Repartition (can increase or decrease)
df_repartitioned = df.repartition(10)
df_repartitioned = df.repartition("department")  # By column

# Coalesce (only decrease partitions, more efficient)
df_coalesced = df.coalesce(5)
```

Broadcast Joins

```
python
from pyspark.sql.functions import broadcast

# For small DataFrames that fit in memory
large_df = spark.read.parquet("large_dataset")
small_df = spark.read.parquet("small_lookup_table")

# Broadcast the small DataFrame
result = large_df.join(broadcast(small_df), "key")
```

Bucketing

```
python
# Write data with bucketing (for repeated joins)
df.write \
  .bucketBy(10, "user_id") \
  .sortBy("timestamp") \
  .saveAsTable("bucketed_table")
```

13. Spark SQL

You can use SQL syntax directly with DataFrames:

Creating Temporary Views

python

```
# Create temporary view
df.createOrReplaceTempView("employees")

# Run SQL queries
sql_result = spark.sql("""
    SELECT department,
           AVG(salary) as avg_salary,
           COUNT(*) as employee_count
    FROM employees
    WHERE age > 25
    GROUP BY department
    ORDER BY avg_salary DESC
""")

sql_result.show()
```

Complex SQL Operations

python

```
# Window functions in SQL
spark.sql("""
    SELECT employee_name,
           department,
           salary,
           ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as
rank,
           salary - LAG(salary, 1) OVER (PARTITION BY department ORDER BY salary) as
salary_diff
    FROM employees
""").show()

# Common Table Expressions (CTEs)
spark.sql("""
    WITH dept_stats AS (
        SELECT department,
               AVG(salary) as avg_salary,
               COUNT(*) as emp_count
        FROM employees
        GROUP BY department
    )
    SELECT e.employee_name,
           e.department,
```

```
        e.salary,
        d.avg_salary,
        e.salary - d.avg_salary as salary_diff_from_avg
    FROM employees e
    JOIN dept_stats d ON e.department = d.department
""").show()
```

14. Streaming with PySpark

Basic Streaming

python

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Create streaming DataFrame
streaming_df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "input-topic") \
    .load()

# Process the stream
processed_df = streaming_df \
    .selectExpr("CAST(value AS STRING) as json_string") \
    .select(from_json("json_string", schema).alias("data")) \
    .select("data.*")

# Write stream
query = processed_df \
    .writeStream \
    .format("console") \
    .outputMode("append") \
    .trigger(processingTime="10 seconds") \
    .start()


query.awaitTermination()
```

Windowed Aggregations in Streaming

python

```python
# Windowed aggregations
```

```
windowed_counts = streaming_df \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window("timestamp", "5 minutes"),
        "user_id"
    ) \
    .count()
```

15. Best Practices for Data Engineering

1. Data Format Selection

- Use **Parquet** for analytical workloads (columnar, compressed)
- Use **Delta Lake** for ACID transactions and versioning
- Use **Avro** for schema evolution in streaming
- Avoid CSV for large-scale production workloads

2. Partitioning Strategy

python

```
# Partition by commonly filtered columns
df.write.partitionBy("year", "month", "day").parquet("path/to/data")

# Avoid over-partitioning (small files problem)
# Aim for 100MB-1GB per partition
```

3. Schema Management

python

```
# Always define schemas explicitly for production
from pyspark.sql.types import *

schema = StructType([
    StructField("id", LongType(), False),
    StructField("name", StringType(), True),
    StructField("timestamp", TimestampType(), True)
])

df = spark.read.schema(schema).json("path/to/data")
```

4. Error Handling

python

```python
# Handle corrupt records
df = spark.read \
    .option("mode", "PERMISSIVE") \
    .option("columnNameOfCorruptRecord", "_corrupt_record") \
    .json("path/to/data")

# Filter out corrupt records
clean_df = df.filter(col("_corrupt_record").isNull())
corrupt_df = df.filter(col("_corrupt_record").isNotNull())
```

5. Resource Management

python

```python
# Configure Spark appropriately
spark = SparkSession.builder \
    .appName("DataEngineering") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
    .config("spark.executor.memory", "4g") \
    .config("spark.executor.cores", "4") \
    .config("spark.sql.files.maxPartitionBytes", "128MB") \
    .getOrCreate()
```

6. Data Quality Checks

python

```python
# Data quality validations
def validate_data(df):
    # Check for nulls in required columns
    null_counts = df.select([
        count(when(col(c).isNull(), c)).alias(c)
        for c in df.columns
    ])

    # Check for duplicates
    total_count = df.count()
    distinct_count = df.distinct().count()

    if total_count != distinct_count:
        print(f"Warning: Found {total_count - distinct_count} duplicate records")

    return null_counts

# Data profiling
```

```
df.describe().show()

df.select([countDistinct(c).alias(c) for c in df.columns]).show()
```

7. Testing Strategy

python

```
# Unit testing with small datasets
def test_transformation():
    test_data = [("Alice", 25), ("Bob", 30)]
    test_df = spark.createDataFrame(test_data, ["name", "age"])

    result_df = my_transformation_function(test_df)

    assert result_df.count() == 2
    assert "age_category" in result_df.columns

# Integration testing with sample production data
def test_with_sample_data():
    sample_df = spark.read.parquet("sample_data_path")
    result_df = full_pipeline(sample_df)

    # Validate results
    assert result_df.count() > 0
    assert not any([row._corrupt_record for row in result_df.collect()])
```

This tutorial covers the essential concepts and practical examples you'll need for data engineering with PySpark. The key is to start with small datasets, understand the concepts, and gradually work with larger, more complex data processing pipelines.