# 1. Initialize Spark Session

spark = SparkSession.builder.appName("ComplexDataFrameExample").getOrCreate()

- Creates a Spark session with the application name Complex Data Frame Example.

This session is the entry point to use the PySpark API.

# 2. Define Schema for Nested Data

schema = StructType([

StructField("id", IntegerType(), True),

StructField("name", StringType(), True),

StructField("tags", ArrayType(StringType()), True)

])

- Defines a schema for a DataFrame. It has three fields: `id` (integer), `name` (string), and `tags` (an array of strings).

- The third argument (`True`) allows `null` values in the respective fields.

# 3. Sample Data

data = [

(1, 'Alice', ['HR', 'Recruitment']),

(2, 'Bob', ['Engineering', 'Software']),

(3, 'Catherine', ['HR', 'Training'])

]

- Creates a list of tuples where each tuple represents a row of data with `id`, `name`, and an array of `tags`.

# 4. Create DataFrame

df\_complex = spark.createDataFrame(data, schema)

- Creates a DataFrame from the `data` and the defined `schema`.

# 5. Show DataFrame

df\_complex.show(truncate=False)

- Displays the DataFrame without truncating the content of columns, showing the entire content of each row.

# 6. Sample Data for Salaries

data = [

(1, 'Alice', 'HR', 70000),

(2, 'Bob', 'Engineering', 80000),

(3, 'Catherine', 'HR', 75000),

(4, 'David', 'Engineering', 95000),

(5, 'Eva', 'Marketing', 60000)

]

- Creates a list of tuples, each containing an `id`, `name`, `department`, and `salary`.

# 7. Define Schema for Salaries

schema = StructType([

StructField("id", IntegerType(), True),

StructField("name", StringType(), True),

StructField("department", StringType(), True),

StructField("salary", IntegerType(), True)

])

- Defines a schema for the `salaries` DataFrame, including fields for `id`, `name`, `department`, and `salary`.

# 8. Create Salaries DataFrame

df\_salaries = spark.createDataFrame(data, schema)

- Creates a DataFrame from the salary `data` and its associated `schema`.

# 9. Perform Aggregations

df\_salaries.groupBy("department").agg(

avg("salary").alias("average\_salary"),

count("id").alias("employee\_count"),

max("salary").alias("max\_salary")

).show()

- Groups the DataFrame by `department`, performs aggregations (average salary, count of employees, and maximum salary), and shows the result.

- `alias` is used to rename the columns for the result.

# 10. Register DataFrame as SQL Temporary View

df\_salaries.createOrReplaceTempView("salaries")

- Creates a temporary SQL view called `salaries` from the `df\_salaries` DataFrame. This allows running SQL queries on it.

# 11. Run SQL Query

result = spark.sql("""

SELECT department, AVG(salary) AS average\_salary

FROM salaries

GROUP BY department

""")

- Executes a SQL query that selects `department` and calculates the average salary, grouped by `department`.

# 12. Show SQL Result

result.show()

- Displays the result of the SQL query.

# 13. DataFrame Transformations

df\_transformed = df\_salaries.withColumn("adjusted\_salary", col("salary") \* 1.1)

- Creates a new column `adjusted\_salary` in the DataFrame by multiplying the `salary` column by 1.1 (for a 10% raise).

df\_transformed = df\_transformed.withColumn("high\_earner", expr("CASE WHEN salary > 80000 THEN 'Yes' ELSE 'No' END"))

- Adds a new column `high\_earner` with a conditional expression that checks if the salary is greater than 80,000. If true, it assigns 'Yes'; otherwise, 'No'.

df\_transformed.show()

- Displays the transformed DataFrame.

# 14. Handle Missing Data

data = [

(1, 'Alice', 70000),

(2, None, 80000),

(3, 'Catherine', None),

(4, 'David', 95000)

]

- Creates a new dataset with some missing values for `name` and `salary`.

schema = StructType([

StructField("id", IntegerType(), True),

StructField("name", StringType(), True),

StructField("salary", IntegerType(), True)

])

- Defines a schema with fields for `id`, `name`, and `salary`.

df\_missing = spark.createDataFrame(data, schema)

- Creates a DataFrame with the missing values.

df\_filled = df\_missing.fillna({'name': 'Unknown', 'salary': 0})

- Fills missing values: sets the default value for missing `name` as 'Unknown' and for missing `salary` as 0.

df\_dropped = df\_missing.dropna()

- Drops rows with any missing values.

df\_filled.show()

df\_dropped.show()

- Displays the DataFrames after filling and dropping missing values.

# 15. Save and Load Data

df\_salaries.write.parquet("/dbfs/FileStore/tables/salaries.parquet")

- Saves the `df\_salaries` DataFrame as a Parquet file (a columnar storage format) at the specified path.

df\_loaded = spark.read.parquet("/dbfs/FileStore/tables/salaries.parquet")

- Loads the DataFrame from the saved Parquet file.

df\_loaded.show()

- Displays the loaded DataFrame.

# 16. Databricks Visualization

df\_salaries.createOrReplaceTempView("salaries")

- Registers the `df\_salaries` DataFrame as a temporary SQL view again.

result = spark.sql("""

SELECT department, AVG(salary) AS average\_salary

FROM salaries

GROUP BY department

""")

- Executes the same SQL query to calculate the average salary per department.

display(result)

- Displays the result using Databricks visualization tools (only works in Databricks).

# 17. Handle Large DataSets

df\_salaries.cache()

- Caches the DataFrame in memory to optimize performance for repeated actions.

df\_salaries.write.partitionBy("department").parquet("/dbfs/FileStore/tables/salaries\_partitioned.parquet")

- Partitions the DataFrame by `department` before saving it as a Parquet file, which helps optimize performance when querying large datasets.

df\_salaries.unpersist()

- Removes the cached DataFrame from memory, freeing up resources.