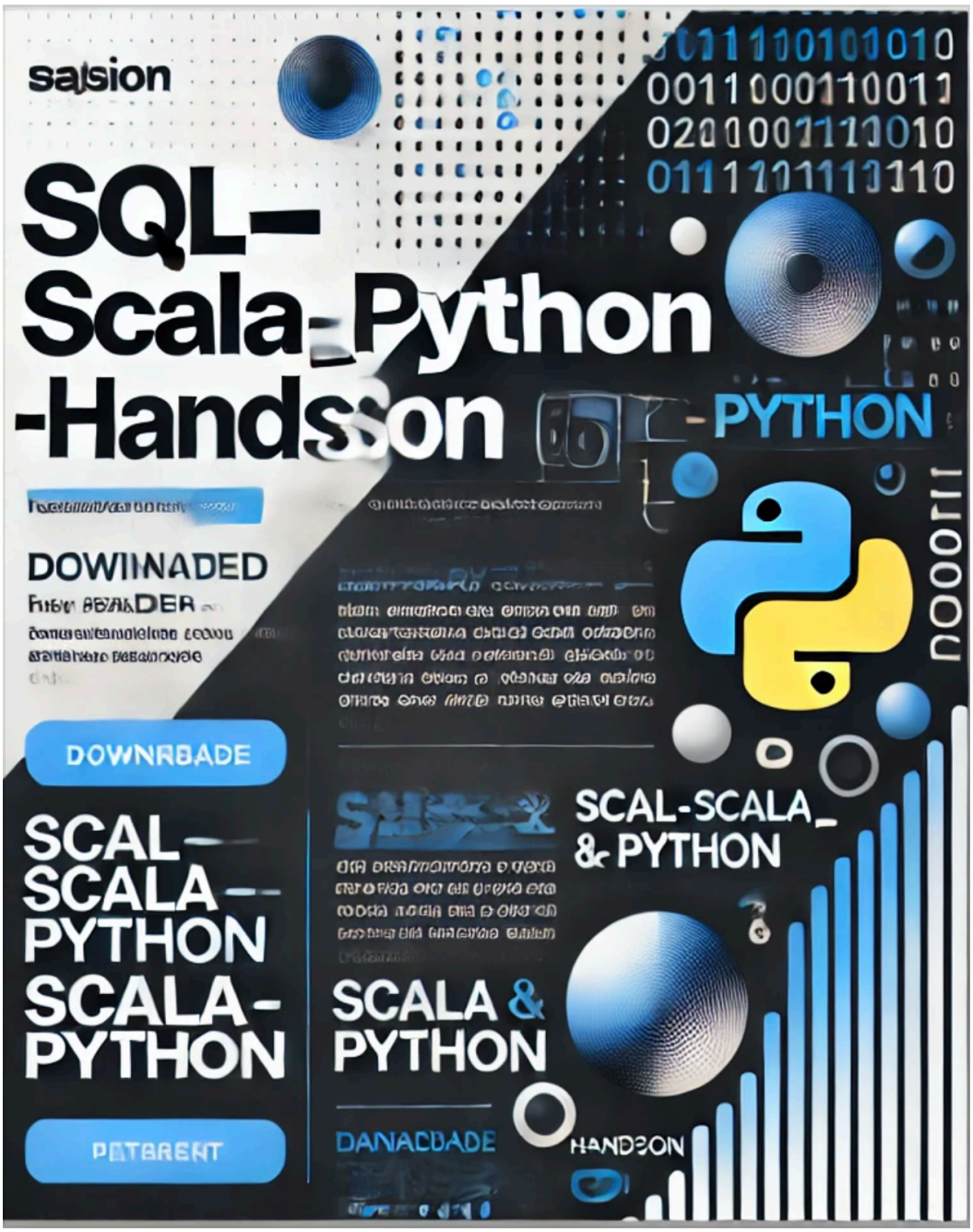
SQL-SCALA_PYTHON - HANDSON



package Pack

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql._
import org.apache.spark.sql.types._
```

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions._
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.catalyst.expressions.{DayOfMonth,
Month, Year}
import scala.io.Source



object objnove24 {


  def main(args:Array[String]):Unit={

    println("Hello Guyes")

    val conf = new
SparkConf().setAppName("first").setMaster("local[*]").set("spark.dri
ver.host", "localhost")
      .set("spark.driver.allowMultipleContexts", "true")

    val sc = new SparkContext(conf)

    sc.setLogLevel("ERROR")

    val spark = SparkSession.builder.getOrCreate()

    import spark.implicits._

val Data = Seq(
  (1, "Alice", "Johnson", "HR", 50000, "2021-05-10", "Female"),
  (2, "Bob", "Smith", "IT", 75000, "2020-08-15", "Male"),
  (3, "Charlie", "Brown", "Finance", 60000, "2019-03-20", "Male"),
  (4, "Diana", "Prince", "IT", 85000, "2021-01-10", "Female"),
  (5, "Eva", "Green", "Marketing", 45000, "2022-07-05", "Female"),
  (6, "Frank", "Adams", "Finance", 70000, "2020-12-11", "Male"),
  (7, "Grace", "Kelly", "HR", 52000, "2018-09-25", "Female"),
  (8, "Hank", "Miller", "IT", 90000, "2023-04-01", "Male"),
  (9, "Ivy", "Harper", "Finance", 58000, "2022-06-20", "Female"),
  (10, "Jack", "Daniels", "HR", 48000, "2021-11-15", "Male"),
  (11, "Kate", "Winslet", "Marketing", 53000, "2020-03-10",
"Female"),
```

```
  (12, "Liam", "Neeson", "Finance", 75000, "2023-01-20", "Male"),
  (13, "Mia", "Wallace", "HR", 55000, "2019-12-30", "Female"),
  (14, "Nathan", "Drake", "IT", 82000, "2018-02-14", "Male"),
  (15, "Olivia", "Newton", "Marketing", 46000, "2022-09-18",
"Female"),
  (2, "Bob", "Smith", "IT", 75000, "2020-08-15", "Male"), // Duplicate
row
  (7, "Grace", "Kelly", "HR", 52000, "2018-09-25", "Female") //
Duplicate row
)
// Convert to DataFrame
val df =
Data.toDF("EMPID","FNAME","LNAME","DEPARTMENT","SALARY","D
OJ","GENDER")


    // Show the DataFrame
    df.show()

    df.createOrReplaceTempView("emp")

//SELECT:

    println("SELECT in SQL")

    spark.sql("select * from emp").show()
    spark.sql("select empid,name from emp").show()

    println("SPARK  SCALA")

    df.select("*").show()

    df.select("empid","name").show()

//DISTINCT:


   println("IN  SQL")

    spark.sql(" select distinct empid ,name from emp order by
empid").show()
```

```
    println("IN DSL (DataFrame API) using  SCALA")


df.select("empid","name").distinct().orderBy("empid").show()
```

//WHERE:

```
 println("IN  SQL")

spark.sql("select * from emp where empid=2").show()

    spark.sql("select * from emp where empid>4").show()

    spark.sql("select name,salary  from emp where
salary>60000").show()

    spark.sql("select * from emp where empid in (5,12)").show()



    println("IN DSL (DataFrame API) using  SCALA")

    df.filter($"empid"===2).show()

    df.filter($"empid">2).show()

    df.filter($"salary">60000).select("empid","salary").show()

    df.filter($"empid".isin(5,12)).show()

    df.where($"empid">7).show()
```

//ORDER BY:

```
 println("IN  SQL")

spark.sql("select * from emp order by salary asc").show()

    spark.sql("select empid, name ,salary as ordersalary from emp
```

```
order by salary desc").show()



   println("IN DSL (DataFrame API) using  SCALA")

   df.sort($"salary".asc).show()
   df.select($"empid",$"name",
$"salary".as("ordersalary")).orderBy("salary").show()
```

//LIMIT:

```
 println("IN  SQL")

spark.sql("select * from emp limit “3).show()


   println("IN DSL (DataFrame API) using  SCALA")

   df.select(“*”).limit(2).show()
```

//COUNT:

```
println("IN  SQL")

// 1. Select all rows
spark.sql("select * from emp").show()
// 2. Count total rows
spark.sql("select count(*) from emp").show()
// 3. Count distinct departments
spark.sql("select count( distinct department) from emp      ").show()
// 4. Count rows where salary > 70000
spark.sql("select count(*) from emp where salary>70000").show()
// 5. Group by department and count employees in each department
spark.sql("select  department, count(*) as personindep from emp
group by department").show()
```

```
println("IN DSL (DataFrame API) using  SCALA")


// 1. Select all rows
df.show()

// 2. Count total rows
println(s"Total Rows: ${df.count()}")

// 3. Count distinct departments
df.select(countDistinct("DEPARTMENT").as("distinct_departments"))
.show()

// 4. Count rows where salary > 70000
df.filter(col("SALARY") > 70000)
  .select(count("*").as("count_salary_gt_70000"))
  .show()

// 5. Group by department and count employees in each department
df.groupBy("DEPARTMENT")
  .agg(count("*").as("personindep"))
  .show()
```

//SUM:

```
println("IN  SQL")


spark.sql("select sum(salary) from emp").show

spark.sql("select DEPARTMENT, sum(salary) as deptwisesalary from
emp group by DEPARTMENT ").show()
```

println("IN DSL (DataFrame API) using SCALA")

df.select(*sum*("salary")).show()

df.agg(*sum*("salary")).show()

df.groupBy("department").agg(*sum*("salary").as("deptwisesalary")).show()

//AVG

println("IN SQL")

spark.sql("select avg(salary) from emp").show

spark.sql("select DEPARTMENT, avg(salary) as deptwisesalary from emp group by DEPARTMENT ").show()

println("IN DSL (DataFrame API) using SCALA")

df.select(*avg*(“salary")).show()

df.agg(*avg*(“salary")).show()

df.groupBy("department").agg(*avg*(“salary").as("deptwisesalary")).show()

//MAX/MIN

//CONCAT

println("IN SQL")

```
spark.sql(" select concat(fname ,' - ' , lname) as name from
emp").show()


println("IN DSL (DataFrame API) using  SCALA")


df.select(concat(col("fname"), lit("  + "),
col("lname"))as("name")).show()
```

//TRIM— VIDEO pending

```
 println("IN  SQL")


spark.sql(" select  fname, lname, rtrim(department) from
emp").show()
spark.sql(" select  fname, lname, ltrim(department) from
emp").show()


    println("IN DSL (DataFrame API) using  SCALA")

    df.select("department").show()

    df.select(trim(col("department"))).show()
    df.select(ltrim(col("department"))).show()
    df.select(rtrim(col("department"))).show()
```

//SUBSTRING

```
//select SUBSTRING(string, start, length) FROM table;
//df.select(substring(col("string"),start, length))


   println("IN  SQL")

spark.sql("select substring( DOJ, 1, 4) as DOJY  from emp").show()
```

```
    println("IN DSL (DataFrame API) using  SCALA")

    df.select(substring(col("DOJ"),1,4).as("DOJY")).show()


//CURDATE,
//NOW,
//CURTIME


 println("IN  SQL")


spark.sql("SELECT NOW() AS current_datetime from emp").show()

    println("IN DSL (DataFrame API) using  SCALA")


    df.select(current_date()).show()
    df.select(current_timestamp()).show()
    df.select(unix_timestamp()).show()

    df.withColumn("current_time", date_format(current_timestamp(),
"HH:mm:ss")).show()


//CONVERT
//CAST


println("IN  SQL")

spark.sql("describe emp").show()

 spark.sql("select  cast(DOJ as date) from emp").show()


    println("IN DSL (DataFrame API) using  SCALA")

    df.printSchema()
```

```
    val df1=df.select(col("DOJ").cast("date"))

    df1.printSchema()
```

//IF -
//CASE

IF(condition, value_if_true, value_if_false)
//df.select(when(condition,value1)\.otherwise(value2))

```
  println("IN  SQL")

spark.sql("select empid, fname,lname,department, if(salary>70000,
'high','low') as salary_status from emp").show()

spark.sql("select empid, department, case when salary>70000 then
'high' else 'low' end as salary_status from emp").show()

 spark.sql("select  empid,  salary,CASE  WHEN salary > 80000 THEN
'Very High'      WHEN salary > 50000 THEN 'High'   ELSE 'Low' END
AS salary_category FROM emp").show()


    println("IN DSL (DataFrame API) using  SCALA")



df.select(  when($"salary">70000,"High").otherwise("Low").as("Salar
y_status")).show()


    df.withColumn("salary_status",
when($"salary">70000,"High").otherwise("Low")).show()


    // Add "salary_category" column
     df.withColumn("salary_category",
      when($"salary" > 70000, "Very High")
```

```
      .when($"salary" > 50000, "High")
      .otherwise("Low")
  ).show()



  df.withColumn("salary_category", expr(
    "CASE WHEN salary > 70000 THEN 'Very High' " +
      "WHEN salary > 50000 THEN 'High' " +
      "ELSE 'Low' END"
  ))
    .show()
```

//COALESCE

//The COALESCE function is a powerful and widely-used function in both **SQL** and **Apache Spark (Scala DSL)**. It is used to handle **NULL** values by returning the first
//non-NULL value from a list of expressions.

```
//SELECT COALESCE(column1,column2, column3) FROM table;
//df.select(coalesce("column1","column2", "column3"))
```

```
/*
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

val spark = SparkSession.builder()
  .appName("COALESCE Example")
  .master("local[*]")
  .getOrCreate()

import spark.implicits._

// Sample Data
val data = Seq(
```

```
  (101, null, 500, 1000),
  (102, null, null, 800),
  (103, null, null, null)
)

val df = data.toDF("emp_id", "bonus1", "bonus2", "bonus3")

df.show()

val result = df.withColumn("total_bonus", coalesce($"bonus1",
$"bonus2", $"bonus3", lit(0)))

result.show()

*/

/*
SELECT emp_id,
       COALESCE(bonus1, bonus2, bonus3, 0) AS total_bonus
FROM employees;


SELECT emp_id,
       COALESCE(phone_number, 'Not Provided') AS phone
FROM customers;


*/
```

//JOIN

```
/*

SELECT table1.column1, table2.column2
FROM table1
JOIN table2
ON table1.column_name = table2.column_name;

SELECT e.emp_id, e.name, d.dept_name
```

```
FROM employees e
INNER JOIN departments d
ON e.dept_id = d.dept_id;


df1.join(df2, df1("column") === df2("column"), "join_type")


import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

val spark = SparkSession.builder()
  .appName("Join Example")
  .master("local[*]")
  .getOrCreate()

import spark.implicits._

// Sample Data for employees
val employees = Seq(
  (1, "Alice", 101),
  (2, "Bob", 102),
  (3, "Charlie", 103)
).toDF("emp_id", "name", "dept_id")

// Sample Data for departments
val departments = Seq(
  (101, "HR"),
  (102, "IT"),
  (104, "Finance")
).toDF("dept_id", "dept_name")

// Show input data
employees.show()
departments.show()


*/


//PIVOT
```

```
df.groupBy("pivot_column")\
.pivot("column").agg(agg_function)
```

//PIVOT in SQL and Spark DSL with Examples
//The PIVOT operation is used to **transform rows into columns** in SQL and Spark (DSL). It is often used to aggregate and restructure data for better analysis and
//presentation.

```
/*


// Sample Data
val data = Seq(
  ("East", "A", 100),
  ("East", "B", 150),
  ("West", "A", 200),
  ("West", "B", 250)
)

val df = data.toDF("region", "product", "sales")


df.createOrReplaceTempView("pvt")

// Show Input DataFrame
df.show()

println("in SQL how to do??")



spark.sql("select * from pvt").show()

println("in DSL how to do?")

df.select("*").show()
```

```
df.groupBy("region").pivot("product",
Seq("A","B")).agg(sum("sales")).show()




*/


/*
    val df = Seq(
      (1, "id","1001"),
      (1, "name","adi"),
      (2, "id","1002"),
      (2, "name","vas")
    ).toDF("pid","keys","values")



    println("INPUT")
    df.show()



    println("OUTPUT")

  df.groupBy("pid")
      .pivot("keys")
      .agg(first("values")).show()


println("OUTPUT2")

//df.groupBy("pivot_column")|
    //.pivot("column").agg(agg_function)

    df.groupBy("pid").pivot("keys",
Seq("id","name")).agg(first("values")).show()

println("OUTPUT 3")
```

```
    df.groupBy("pid").pivot("keys").agg(first("values")).show()


*/

/*

  // Input data
  val data = Seq(
    ("A", "Laptop", 1000),
    ("A", "Phone", 800),
    ("A", "Laptop", 1500),
    ("B", "Laptop", 1200),
    ("B", "Phone", 600),
    ("B", "Phone", 1000)


  )

  val df = data.toDF("Category", "Product", "Sales")

  println(" INPUT TABLE")

  df.show()

  println( "Pivot operation with SUM aggregation")

df.groupBy("Category")
    .pivot("Product", Seq("Laptop", "Phone"))
    .agg(sum("Sales")).show()

  println("Pivot operation with AVG aggregation")

  df.groupBy("Category").pivot("Product").agg(avg("sales")).show()

  println("Pivot operation with first aggregation")

  df.groupBy("Category").pivot("Product").agg(first("sales")).show()

  println("Pivot operation with max aggregation")

  df.groupBy("Category").pivot("Product").agg(max("sales")).show()
```

```
  println("Pivot operation with min aggregation")

  df.groupBy("Category").pivot("Product").agg(min("sales")).show()

  println("Pivot operation with count() aggregation")

df.groupBy("Category").pivot("Product").agg(count("sales")).show()

  println("Pivot operation with collect_list() aggregation")

df.groupBy("Category").pivot("Product").agg(collect_list("sales")).sh
ow()

*/
```

//RANK()
//DENSE_RANK()
//ROW_NUMBER()

```
/*

// Input data

    val data = Seq(
      ("Alice", 90),
      ("Bob", 80),
      ("Cathy", 80),
      ("David", 70)
    )
      val df=data.toDF("name", "marks")

    df.createOrReplaceTempView("std")
```

```
    df.show()

    println("SQL")

    spark.sql("select name ,marks, rank() over ( order by marks) as
RANK from std").show()


    spark.sql(" select name, marks ,dense_rank() over(order by marks)
as DENSE_RANK from std").show()

    spark.sql("select name,marks, row_number() over(order by marks )
as row_number from std").show()

println("DSL")


//df.select("name",
rank().over(Window.orderBy($"marks".desc)).alias("rank"))

    val windowSpec = Window.orderBy($"marks".desc)

    val result = df
      .withColumn("rank", rank().over(windowSpec))
      .withColumn("dense_rank", dense_rank().over(windowSpec))
      .withColumn("row_number", row_number().over(windowSpec))

    result.show()



    df.withColumn("rank",
rank().over(Window.orderBy($"marks".desc)))
      .withColumn("dense_rank",
dense_rank().over(Window.orderBy($"marks".desc)))
      .withColumn("row_number",
row_number().over(Window.orderBy($"marks".desc))).show()


*/
```

//CTE

//What is a CTE in SQL: A **CTE (Common Table Expression)** is a temporary result set that is defined within the execution scope of a single SQL query. It makes
//complex queries more readable by allowing you to define reusable intermediate results.

/*

```
    // Input data
    val data = Seq(
      (1, "North", 1000),
      (2, "South", 2000),
      (3, "North", 1500),
      (4, "South", 2500)
    )

    val df=data.toDF("id", "region", "sales")


    df.createOrReplaceTempView("tmp")


    println("IN SQL")

spark.sql(" select * from tmp").show()


    println(" IN SQL")

    spark.sql(" with ctable as (select region, sum(sales) as total_sale from tmp group by region) select region, total_sale from ctable where total_sale>3000").show()
```

```
    println(" IN DSL")

    val
df1=df.groupBy("region").agg(sum("sales").as("total_sale")).filter($"t
otal_sale">3000)

    df1.show()

    //val df2=df1.filter($"total_sale">3000)

   // df2.show()

*/
```

PREPARED BY Mantu Kumar Deka , Please add me in LinkedIn

Also Subscribe my YouTube channel : MKD Mixture