**SAS-to-PySpark migration** with **sample mappings** between key SAS functionalities and their PySpark equivalents.

I'll include common **data manipulation**, **ETL tasks**, and **analytical workflows**.

### **1. SAS DATA Step → PySpark DataFrames**

**The DATA Step in SAS is a core component for data transformation. In PySpark, you can replicate this functionality using the DataFrame API.**

#### **Example: Basic Data Step**

**SAS Code:**

**sas**

**data output;**

**set input;**

**where age > 18;**

**total\_income = salary + bonus;**

**run;**

**PySpark Equivalent:**

**python**

**from pyspark.sql import SparkSession**

**from pyspark.sql.functions import col, when**

**spark = SparkSession.builder.appName("Migration").getOrCreate()**

**# Load data into a DataFrame**

**df = spark.read.csv("input.csv", header=True, inferSchema=True)**

**# Transform data**

**output = df.filter(col("age") > 18).withColumn("total\_income", col("salary") + col("bonus"))**

**output.show()**

**Key Mappings:**

* **SAS SET statement → PySpark DataFrame creation using spark.read or an existing DataFrame.**
* **WHERE condition → PySpark filter() or where() function.**
* **Derived columns → withColumn() with transformations like col() or when().**

### **2. SAS PROC SQL → PySpark SQL Queries**

**PROC SQL in SAS allows SQL-based operations like filtering, joining, and aggregating. PySpark offers similar functionality via the spark.sql() interface or the DataFrame API.**

#### **Example: Filtering and Grouping**

**SAS Code:**

**sas**

**proc sql;**

**create table summary as**

**select region, sum(sales) as total\_sales**

**from sales\_data**

**where year = 2023**

**group by region;**

**quit;**

**PySpark Equivalent (DataFrame API):**

**python**

**from pyspark.sql.functions import sum**

**# Read sales data**

**df = spark.read.csv("sales\_data.csv", header=True, inferSchema=True)**

**# Filter and aggregate data**

**summary = df.filter(col("year") == 2023) \**

**.groupBy("region") \**

**.agg(sum("sales").alias("total\_sales"))**

**summary.show()**

**PySpark Equivalent (SQL Interface):**

**python**

**df.createOrReplaceTempView("sales\_data")**

**summary = spark.sql("""**

**SELECT region, SUM(sales) AS total\_sales**

**FROM sales\_data**

**WHERE year = 2023**

**GROUP BY region**

**""")**

**summary.show()**

**Key Mappings:**

* **PROC SQL SELECT → PySpark SQL query or DataFrame transformations.**
* **GROUP BY → groupBy() and aggregate functions like sum(), avg(), count().**
* **WHERE clause → Filter conditions (filter() or SQL syntax).**

### **3. SAS PROC SORT → PySpark sort()**

**PROC SORT in SAS sorts data by specific columns. In PySpark, the same functionality can be achieved using orderBy().**

#### **Example: Sorting Data**

**SAS Code:**

**sas**

**proc sort data=input out=sorted\_data;**

**by descending sales;**

**run;**

**PySpark Equivalent:**

**python**

**sorted\_data = df.orderBy(col("sales").desc())**

**sorted\_data.show()**

**Key Mappings:**

* **BY statement in PROC SORT → orderBy() with asc() or desc() in PySpark.**

### **4. SAS MACROS → Python Functions and UDFs**

**SAS Macros automate code execution. In PySpark, you can use Python functions or UDFs (User Defined Functions) for similar automation.**

#### **Example: Parameterized Filtering**

**SAS Macro Code:**

**sas**

**%macro filter\_data(condition);**

**data output;**

**set input;**

**where &condition.;**

**run;**

**%mend;**

**%filter\_data(sales > 1000);**

**PySpark Equivalent:**

**python**

**def filter\_data(df, condition):**

**return df.filter(condition)**

**output = filter\_data(df, "sales > 1000")**

**output.show()**

**Key Mappings:**

* **SAS Macro → Python functions for reusable code.**
* **For row-level operations, use UDFs.**

### **5. SAS PROC MEANS/SUMMARY → PySpark Aggregations**

**PROC MEANS generates summary statistics like mean, sum, and count. PySpark achieves this using aggregation functions.**

#### **Example: Summarizing Data**

**SAS Code:**

**sas**

**proc means data=input n mean sum max;**

**class region;**

**var sales;**

**run;**

**PySpark Equivalent:**

**python**

**from pyspark.sql.functions import mean, sum, count, max**

**summary = df.groupBy("region") \**

**.agg(count("sales").alias("n"),**

**mean("sales").alias("mean\_sales"),**

**sum("sales").alias("total\_sales"),**

**max("sales").alias("max\_sales"))**

**summary.show()**

**Key Mappings:**

* **PROC MEANS → groupBy() with aggregate functions like mean(), sum(), max().**

### **6. SAS DATA Step Loops → PySpark Transformations**

**Loops in SAS (e.g., DO loops) can often be replaced with vectorized operations in PySpark for efficiency.**

#### **Example: Row-by-Row Processing**

**SAS Code:**

**sas**

**data output;**

**set input;**

**do i = 1 to 5;**

**sales = sales \* i;**

**end;**

**run;**

**PySpark Equivalent:**

**python**

**from pyspark.sql.functions import expr**

**output = df.withColumn("sales\_transformed", expr("sales \* 5"))**

**output.show()**

**Key Mappings:**

* **Replace SAS loops with PySpark’s column operations for vectorized performance.**

### **Summary Table of Mappings**

| SAS Functionality | PySpark Equivalent | Key Notes |
| --- | --- | --- |
| DATA Step | PySpark DataFrames | Use withColumn() and filter() |
| PROC SQL | PySpark SQL / DataFrame API | spark.sql() or DataFrame syntax |
| PROC SORT | orderBy() | Supports ascending and descending |
| PROC MEANS/SUMMARY | groupBy() with aggregates | Mean, sum, count, etc. |
| Macros | Python Functions / UDFs | Modularize logic |
| DO Loops | Vectorized column operations | Avoid row-by-row processing |

### **Notes**

**Migrating from SAS to PySpark involves:**

1. **Analyzing existing code and identifying equivalent PySpark operations.**
2. **Ensuring code is optimized for parallel and distributed execution.**
3. **Providing thorough validation to match outputs.**
4. **Training teams to write clean, modular PySpark code.**