Migrating SAS to PySpark: A Technical Guide

Key Technical Steps for SAS-to-PySpark Migration

1. **Assess and Understand Existing SAS Code**
    - Identify all **SAS processes**: ETL pipelines, data transformations, statistical procedures, and reporting workflows.
    - Document dependencies on datasets, macros, and libraries.
    - Prioritize SAS scripts based on complexity and usage.

Example:
sas

```
/* Sample SAS Code */

data output;

  set input;

  if age > 18;

run;
```

○

2. **Map SAS Functions and Procedures to PySpark**
    - Use **PySpark equivalents** for SAS operations such as `PROC SORT`, `PROC SUMMARY`, and `DATA STEP`.
    - Common transformations like filtering, aggregating, and joining data can be rewritten using PySpark's **DataFrame API**.

Example Migration:
SAS Code (Filtering):
sas

```
proc sql;

  create table adults as
  select * from input where age > 18;

quit;
```

PySpark Equivalent:
python

```
from pyspark.sql import SparkSession



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

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

adults = df.filter(df["age"] > 18)

adults.show()
```

   -

3. **Handle SAS Macros and Automation**
   - SAS macros used for automation and parameterization can be replaced using **Python functions** and PySpark dynamic query building.
   - PySpark's **UDFs (User Defined Functions)** provide similar capabilities.

Example: **SAS Macro:**
sas

```
%macro filter_data(condition);

  data output;

    set input;

    where &condition.;

  run;

%mend;
```

Python Function with PySpark:
python

```
def filter_data(df, condition):
    return df.filter(condition)


adults = filter_data(df, "age > 18")
adults.show()
```

○

4. **Optimize Data Processing with PySpark**
   - Convert row-by-row SAS processing to PySpark's **distributed processing** using DataFrames or RDDs.
   - Optimize joins and aggregations using **Spark SQL** and partitioning techniques.
   - Utilize **broadcast joins** for small datasets to reduce shuffle operations.
5. **Statistical and Machine Learning Procedures**
   - Replace SAS **PROC REG, PROC GLM, PROC LOGISTIC** with PySpark's **MLlib** library for machine learning models.

Example: **SAS Logistic Regression:**
sas

```
proc logistic data=input;
  model target(event='1') = age income;
run;
```

PySpark Logistic Regression:
python

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import VectorAssembler


assembler = VectorAssembler(inputCols=["age", "income"],
outputCol="features")
```

```
data = assembler.transform(df).select("features", "target")



lr = LogisticRegression(labelCol="target")

model = lr.fit(data)

model.summary.predictions.show()
```

   -

6. **Testing and Validation**
   - Validate PySpark outputs against SAS outputs to ensure accuracy.
   - Use sample datasets to compare data quality, metrics, and performance.

Best Practices for SAS-to-PySpark Migration

1. **Incremental Migration**: Migrate SAS code **module-by-module** rather than all at once.
2. **Documentation**: Document mapping of SAS features to PySpark equivalents.
3. **Training and Support**: Provide upskilling for teams transitioning from SAS to PySpark.
4. **Leverage PySpark Features**: Use DataFrames and PySpark optimizations for better performance.
5. **Parallel Execution**: Test on **distributed clusters** like Apache Spark on AWS EMR, Databricks, or Azure Synapse Analytics.

Summary

Migrating SAS to PySpark involves:

- Understanding the existing SAS workflows.
- Rewriting data transformations, automations, and statistical processes using PySpark's **DataFrame API** and **MLlib**.
- Ensuring the migration achieves better scalability, performance, and maintainability.

By following these steps, organizations can modernize their data pipelines and make the most of PySpark’s distributed processing capabilities.