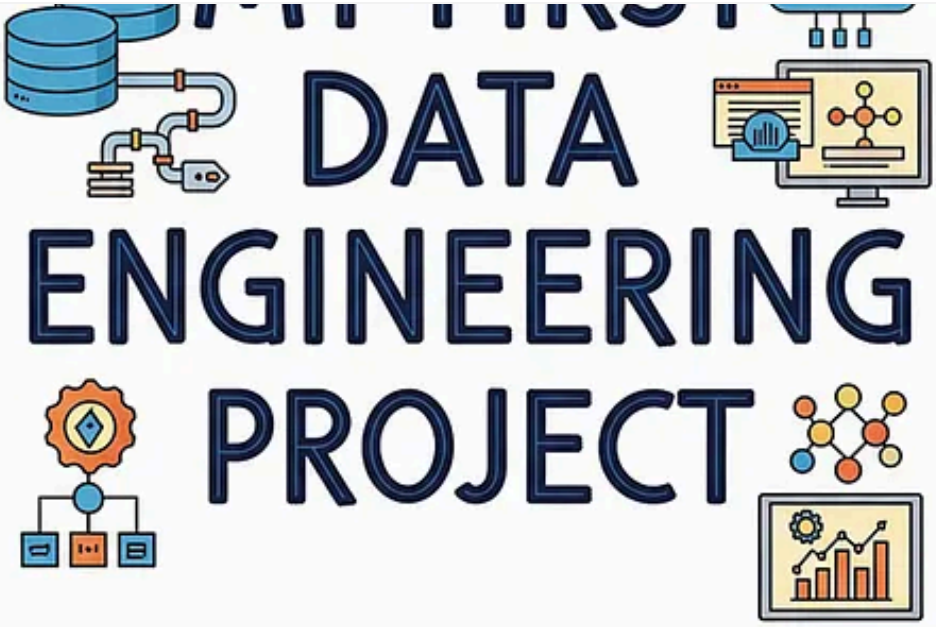

My First Data Engineering Project: Phase 2— Migrating DataStage ETL Jobs to PySpark

Anurag Sharma 5 min read · Jun 11, 2025

1 1

The second chapter of my data engineering journey! The second phase of a transformative project for a major US-based multi-investment company, where I played a pivotal role in modernizing their data infrastructure. After successfully migrating a 300 TB Oracle data warehouse to Cloudera's HDFS (covered in the blog mentioned below), the next challenge was to convert 112 DataStage ETL jobs to PySpark. This phase was a critical step in replacing legacy ETL processes with a scalable, modern framework. Let's explore the journey, my responsibilities, and the lessons learned.

https://medium.com/@one.step.analytics.on.data/my-first-data-engineering-project-phase-1-migrating-a-massive-data-warehouse-from-oracle-to-02c2b6391ddc

The Project Context

Our client's data ecosystem was split into two parts: an operational Oracle database (OLTP) for real-time transactional applications and a data warehouse for BI reporting via OBIEE.

The DataStage ETL jobs were integral to the OLTP side, performing transformations to:

• Populate data on the company's customer-facing website.
• Generate small CSV files for downstream systems.
• Create report tables that are displayed online to customers.

Unlike the data warehouse's massive 300 TB scale, the OLTP database handled smaller, transactional datasets. The goal was to migrate these DataStage jobs — a total of 112 — to PySpark, reading from and writing to the Oracle database using Spark's scalable processing capabilities. This was no small task, as Spark's ecosystem in its early stages posed unique challenges, particularly in replicating complex business logic.

My Roles and Responsibilities

• **Job Analysis and Mapping:** Analyzed DataStage jobs to understand their transformations, business logic, and dependencies.
• **PySpark Development:** Designed and developed PySpark scripts to replicate DataStage functionality, reading from and writing to Oracle DB.
• **UDF Creation:** Wrote custom User-Defined Functions (UDFs) to handle complex business logic not natively supported by Spark.
• **Performance Optimization:** Tuned PySpark jobs for efficiency, managing resource allocation, and connection handling.
• **Testing and Validation:** Validated PySpark outputs against DataStage results

to ensure accuracy and business continuity.

• **Documentation and Collaboration:** Documented the migration process and collaborated with business analysts to confirm logic accuracy.

The Challenge: From DataStage to PySpark

Migrating DataStage jobs to PySpark required translating proprietary ETL workflows into Spark's distributed computing framework. DataStage's visual interface and built-in transformation components didn't always map cleanly to PySpark's programmatic approach. Additionally, Spark's relative immaturity at the time meant limited out-of-the-box functions for complex business logic, forcing us to get creative with UDFs. The dataset size was smaller than the data warehouse, but the jobs demanded precision to ensure customer-facing outputs (e.g., website data, CSV files, report tables) remained accurate.

Step-by-Step Migration Process:

1. Job Analysis and Documentation:

◦ Reverse-engineering each DataStage job, analyzing its stages (e.g., source, transform, target) using DataStage's job designer.

◦ Documented inputs (Oracle tables), transformations (e.g., joins, aggregations, filters), and outputs (website tables, CSV files, report tables).

◦ Identified dependencies between jobs to ensure the correct execution order in PySpark.

2. Mapping Transformations to PySpark:

◦ For each job, converting DataStage transformations into PySpark DataFrame operations. Common mappings included:

▪ **Joins:** DataStage's join stages became df.join() operations.

▪ **Aggregations:** Group-by operations in DataStage were converted to df.groupBy().agg().

▪ **Filters:** DataStage's filter stages mapped to df.filter().

◦ Connected to Oracle DB using Spark's JDBC connector (spark.read.jdbc for reading, df.write.jdbc for writing).

3. Writing Custom UDFs for Business Logic:

◦ Some DataStage transformations involved complex business logic not easily replicated with Spark's standard functions.

Examples included:

▪ **Date Formatting:** Converting Oracle date formats (e.g., DD-MON-YYYY) to specific string formats for website display.

▪ **Financial Calculations:** Computing weighted portfolio returns based on investment type and date ranges.

▪ **String Manipulations:** Parsing and concatenating customer names or addresses for report tables.

◦ I wrote UDFs in Python to handle these cases. For example:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

 # UDF for custom date formatting
 def format_date(date_str):
 # Convert DD-MON-YYYY to YYYY-MM-DD
   try:
   from datetime import datetime
   return datetime.strptime(date_str, '%d-%b-%Y').strftime('%Y-%m-%d')
   except:
```

```
    return None

    format_date_udf = udf(format_date, StringType())
    df = df.withColumn('formatted_date', format_date_udf(df['date_column']))

  Another UDF handled portfolio return calculations:
  def calculate_return(amount, weight, period):
    # Simplified example of weighted return
    return amount * weight / period if period > 0 else 0.0
    calculate_return_udf = udf(calculate_return, DoubleType())
    df = df.withColumn('portfolio_return', calculate_return_udf(df['amount'], df[
```

4. Job Development and Modularization:

◦ I structured PySpark scripts modularly, creating reusable functions for common tasks (e.g., reading from Oracle, writing CSVs).

◦ Each of the 112 jobs was implemented as a PySpark script, parameterized to handle different tables and transformations.

◦ Used Spark's SparkSession to manage connections and optimize resource usage.

5. Performance Optimization:

◦ Tuned Spark configurations (e.g., spark.executor.memory, spark.sql.shuffle.partitions) to handle the transactional dataset efficiently.

◦ Minimized database load by caching frequently accessed tables and using predicate pushdown in JDBC queries.

◦ Parallelized job execution where possible, leveraging Spark's distributed processing.

6. Testing and Validation:

◦ Ran PySpark jobs in a test environment, comparing outputs (website tables, CSVs, report tables) against DataStage results.

◦ Validated row counts, key metrics (e.g., portfolio values), and data formats using automated Python scripts.

◦ Conducted end-to-end testing with business analysts to ensure customer-facing outputs met requirements.

7. Deployment and Scheduling:

◦ Deployed PySpark scripts to the production cluster, integrating them with the existing Control-M scheduler.

◦ Ensured job dependencies were preserved, scheduling jobs to run after upstream data availability.

Challenges and Solutions:

Spark's immaturity posed challenges, particularly with complex transformations. DataStage's built-in components (e.g., for string parsing or financial calculations) lacked direct PySpark equivalents, necessitating UDFs. These UDFs, while powerful, introduced performance overhead, which I mitigated by:

• Optimizing UDF logic for efficiency (e.g., avoiding nested loops).

• Using native Spark functions wherever possible (e.g., pyspark.sql.functions for simple transformations).

• Caching intermediate DataFrames to reduce recomputation.

Another challenge was ensuring zero disruption to customer-facing outputs.

Lessons Learned:

• **UDFs Are a Double-Edged Sword:** While essential for complex logic, UDFs can degrade performance. Balancing UDFs with native Spark functions is key.

• **Modular Code Pays Off:** Structuring PySpark scripts as reusable modules simplified maintenance and scaling.

• **Collaboration is Crucial:** Working with analysts ensured business logic was preserved, reinforcing the value of clear communication.