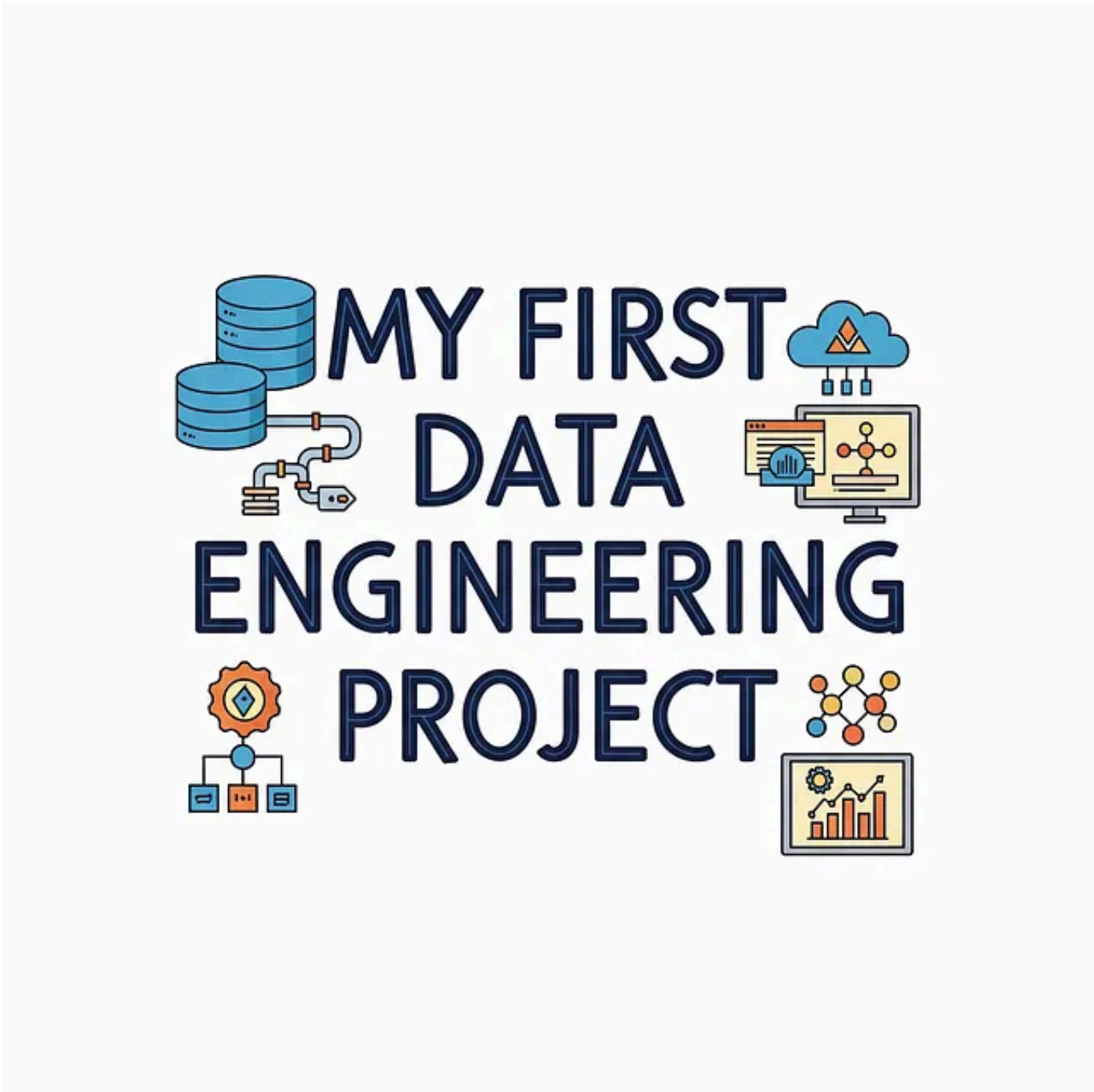


My First Data Engineering Project: Phase 1 — Migrating a Massive Data Warehouse from Oracle to Cloudera

Anurag Sharma 6 min read · Jun 10, 2025

Welcome to the first chapter of my data engineering journey! This blog dives into a pivotal project that marked my entry into the world of data engineering — a complex and exhilarating migration of a 300 TB data warehouse for one of the largest multi-investment companies in the US.

The goal? Transition their Oracle-based data warehouse to Cloudera's Hadoop Distributed File System (HDFS) using Sqoop, setting the stage for a modernized, scalable data ecosystem. Let's unpack the first part of this ambitious project, filled with challenges, learning curves, and triumphs.

The Project Context

Our client, a financial powerhouse managing diverse investment portfolios, relied heavily on data to drive decisions.

Their infrastructure was split into two parts:

1. Operational Database (OLTP): This supported daily applications, handling real-time transactions for customer-facing systems.

2. Data Warehouse (DW): Built on Oracle DB, this powered Business Intelligence (BI) reporting through Oracle Business Intelligence Enterprise Edition (OBIEE). It stored historical data, aggregated metrics, and supported critical reporting for stakeholders.

The data warehouse was the heart of their BI operations, housing 300 terabytes of data, with an additional 30–50 GB of new data streaming in daily from transactional systems. The client wanted to modernize by moving the entire data warehouse to Cloudera's HDFS, leveraging Hadoop's scalability and cost-efficiency. The first milestone was to migrate all data using Sqoop jobs, ensuring seamless data loading and validation.

The Challenge: Migrating 300 TB to HDFS

Migrating 300 TB of data is no small feat. The Oracle data warehouse

contained 250–300 tables, each with its own schema, constraints, and data types. These tables supported a mix of full refreshes (complete data reloads) and incremental updates (daily deltas). Our task was to:

1. Create Hive Tables: Replicate the Oracle table structures in Hive, which would reside on HDFS.

2. Write Sqoop Jobs: Automate data extraction from Oracle to HDFS.

3. Schedule and Orchestrate: Use cron and Control-M to manage job execution.

4. Validate Data: Ensure data integrity and consistency post-migration.

The sheer volume of data, combined with the need for daily incremental loads, made this a high-stakes endeavor. Any misstep could disrupt the client's BI reporting, which was critical for their business operations.

Step 1: Converting Oracle DDLs to Hive DDLs

The first hurdle was creating 250–300 Hive tables that mirrored the Oracle schema. Manually writing Data Definition Language (DDL) scripts for each table would have been a nightmare — error-prone and time-consuming. To streamline this, we developed a **Python script** to automate the conversion of Oracle DDLs to Hive DDLs.

Here is how it worked:

• **Extraction:** We queried Oracle's metadata (e.g., USER_TABLES, USER_TAB_COLUMNS) to extract table structures, including column names, data types, and constraints.
• **Mapping:** We mapped Oracle data types to Hive equivalents (e.g., Oracle's VARCHAR2 to Hive's STRING, NUMBER to DECIMAL or INT). We also accounted for Hive's partitioning and storage formats (e.g., Parquet for efficient columnar storage).
• **Script Generation:** The Python script generated Hive DDL statements, incorporating table properties like partitioning (e.g., by date for incremental loads) and storage optimization.
• **Execution:** We ran the generated DDLs in a single batch using Hive's command-line interface, creating all tables on HDFS.
This automation saved weeks of manual effort and reduced human error. We validated the Hive schemas against Oracle to ensure accuracy, paying special attention to complex data types and primary key constraints.

Step 2: Writing Sqoop Jobs
With Hive tables ready, we turned to **Apache Sqoop**, the go-to tool for transferring data between relational databases and Hadoop. Sqoop's ability to parallelize data extraction made it ideal for handling 300 TB of data. We designed Sqoop jobs to handle both **full refreshes** (for static tables) and **incremental loads** (for tables updated daily).
Key considerations for Sqoop jobs:
• **Full Refresh Jobs:** For static tables, we used Sqoop's import command to extract entire tables from Oracle to HDFS. We optimized these jobs by:

◦ Splitting large tables using a primary key or index column to enable parallel processing (e.g., — split-by).
◦ Specifying Hive table mappings with — hive-import to load data directly into Hive.
◦ Using Parquet format for storage efficiency (— as-parquetfile).
• **Incremental Jobs:** For daily updates (30–50 GB), we used Sqoop's incremental mode (— incremental append or — incremental lastmodified), leveraging a timestamp or ID column to identify new or updated records.
• **Performance Tuning:** To avoid overloading the Oracle database, we fine-tuned parameters like — num-mappers (to control parallelism) and — fetch-size (to manage memory usage). We also used connection pooling to optimize database connections.
Each Sqoop job was scripted to handle a specific table, with error handling and logging to track successes and failures.

Step 3: Scheduling with Cron and Control-M
To ensure reliable execution, we scheduled Sqoop jobs using a combination of **cron** (for lightweight, time-based triggers) and **Control-M** (for enterprise-grade workflow orchestration).

bmc

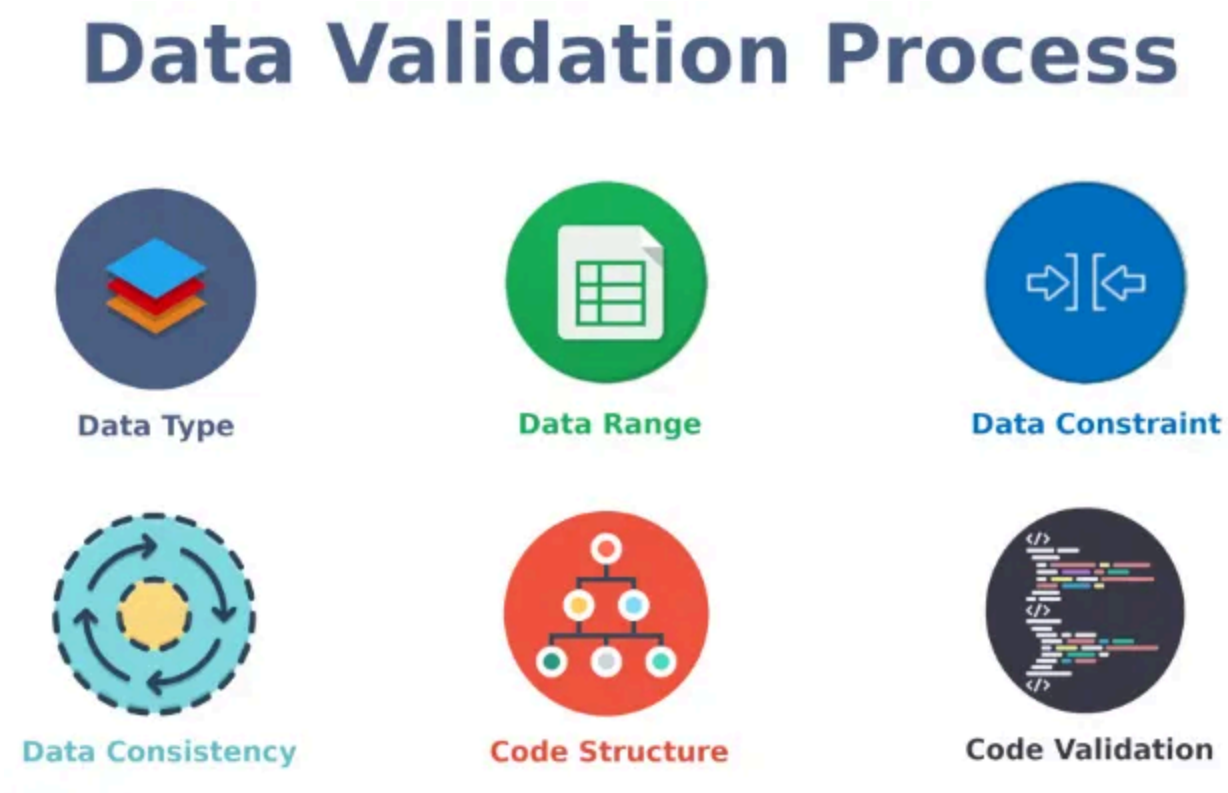
Control-M

This hybrid approach allowed us to:

• **Cron:** Handle simple, recurring jobs (e.g., daily incremental loads at 2 AM).

• **Control-M:** Manage complex dependencies, such as ensuring full refresh jobs are completed before incremental ones. Control-M also provided monitoring, alerting, and retry mechanisms for failed jobs.

We created detailed workflows in Control-M, mapping dependencies between Sqoop jobs and downstream validation tasks. This ensured a smooth pipeline, even when dealing with 30–50 GB of daily data.



Step 4: Data Loading and Validation

The final piece of this milestone was loading data into Hive and validating its integrity. After Sqoop jobs transferred data to HDFS, we:

• **Loaded Data into Hive:** Used Hive's LOAD DATA command for Parquet files or relied on Sqoop's — hive-import to populate tables directly.

• **Validated Data:**

◦ **Row Counts:** Compared row counts between Oracle and Hive tables to ensure completeness.

◦ **Data Sampling:** Ran queries to verify data consistency, checking key columns and aggregations.

◦ **Checksums:** For critical tables, we computed checksums (e.g., using MD5 on key fields) to confirm data integrity.

◦ **Business Logic Checks:** Validated that key metrics (e.g., total investments, portfolio values) matched between Oracle and Hive.

We automated much of the validation using Python scripts, which generated reports for the client, highlighting any discrepancies. This rigorous process ensured zero data loss and maintained trust in the migrated data.

Challenges and Lessons Learned

This phase wasn't without hurdles. The sheer scale of 300 TB stretched our cluster's capacity, requiring careful resource allocation. Network bottlenecks occasionally slowed Sqoop transfers, which we mitigated by optimizing bandwidth and scheduling jobs during off-peak hours. Additionally, some Oracle tables had complex constraints (e.g., nested types), which Hive didn't support natively, so we flattened these structures during DDL conversion.

The biggest lesson? **Automation is your friend.** The Python script for DDL conversion and automated validation scripts saved countless hours. Planning for scalability — through partitioning, parallel processing, and scheduling — was critical to handling daily data inflows.