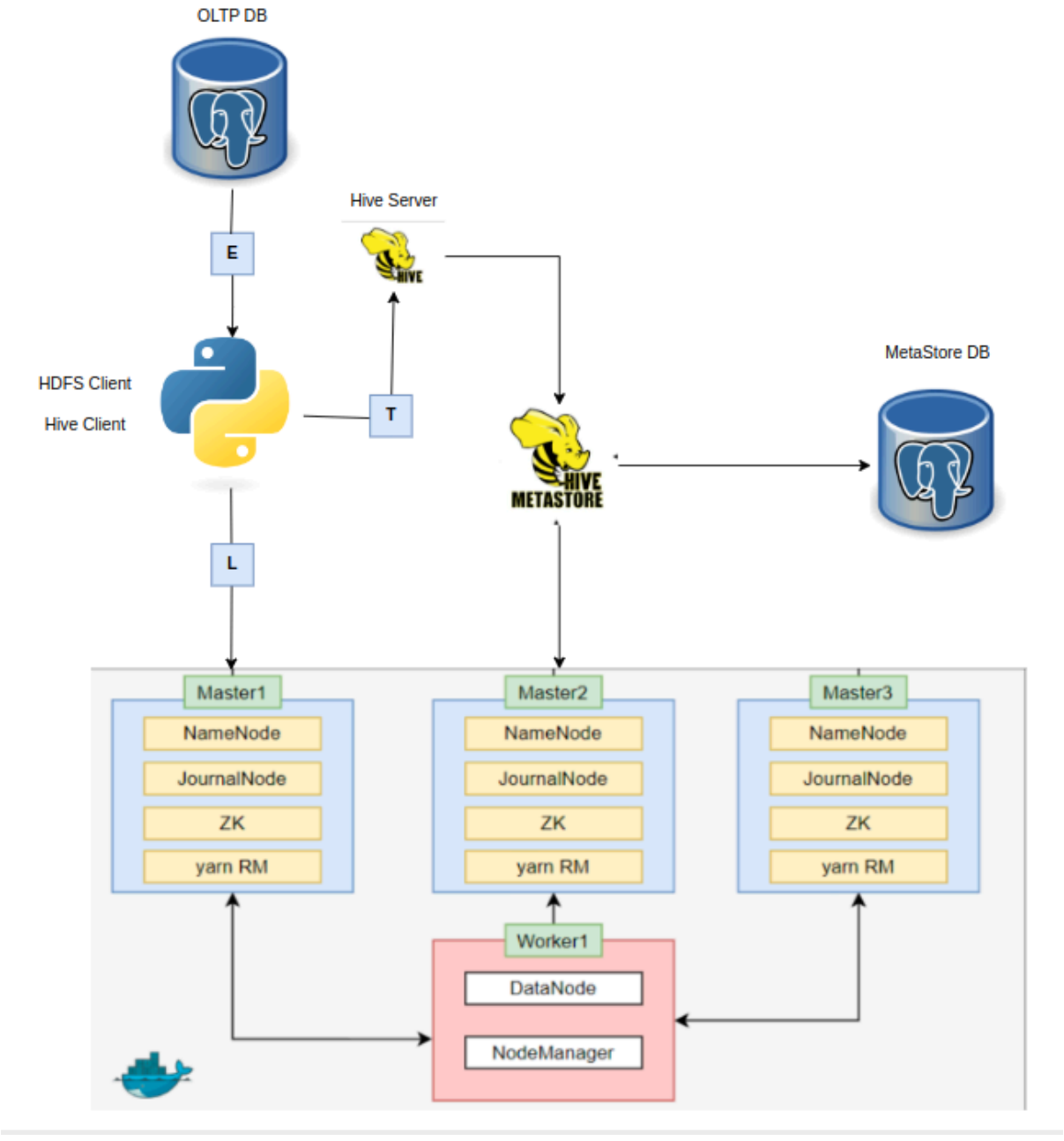
Hive Project



Phase 1:

Architecture Summary

- **Hive Metastore** backed by **PostgreSQL**
- **HiveServer2** for query execution
- **Docker** to containerize services
- **Postgres** for MetaStore
- **Postgres** for OLTP
- **Python** for ELT
- **Hadoop cluster** (Namenode HA using Zookeeper and 1 Datanode)

Implementation Overview

To optimize space and improve maintainability, I first refactored the Dockerfile into a multi-stage build. This helped reduce the final image size by separating the build environment from the runtime image.

I then created a `docker-compose.yml` configuration that defines all services involved: PostgreSQL, Hive Metastore, HiveServer2, Hadoop NameNodes and DataNodes, and Zookeeper. Each service has its respective volumes mounted to persist data across container restarts and facilitate debugging and state inspection.

An `entrypoint.sh` script was added to the Hive containers to automate schema initialization and service startup. This script is executed automatically when the containers launch, thanks to the `entrypoint` directive in the Docker Compose configuration.

Ports were published explicitly for all necessary services:

- PostgreSQL: internal
- Hive Metastore: internal
- HiveServer2: `10000, 10002`
- Zookeeper: internal
- Hadoop NameNode 1: 9870, 8088 for NN and RM web UI
- Hadoop NameNode 2: 9871, 8089 for NN and RM web UI
- Hadoop NameNode 3: 9872, 8090 for NN and RM web UI
- Hadoop DataNode: internal

Each service also mounts relevant configuration files and storage directories, such as `core-site.xml`, `hdfs-site.xml`, `hive-site.xml`, and HDFS volumes

Phase 2:

Objectives

The second phase of the project focused on migrating the data warehouse from Amazon Redshift to Hive and implementing a scalable ELT pipeline with advanced data modeling and performance optimization.

Achievements

- Migrated historical and operational data from Amazon Redshift to Hive tables.
- Designed and implemented **ELT pipelines** that perform **incremental loading** of only changed or new records.
- Implemented **Slowly Changing Dimension Type 2 (SCD2)** using both:
    - **ACID-compliant transactional Hive tables**.
    - **Non-ACID append-only tables** with manual version tracking.
- Enabled **Apache Tez** as the default Hive execution engine for faster query planning and execution.
- Activated **vectorized query execution** to boost performance for analytical workloads.

Key Steps

1. Extraction from Redshift

Data was extracted from Redshift using the `UNLOAD` command to export to S3 in compressed Parquet format. This data was then transferred into HDFS using a PySpark job.

2. Table Design

Hive tables were created with:

- Partitioning by date and dimension keys.
- Bucketing for large fact tables.
- ORC file format for ACID support.

3. Incremental Load

- A staging layer loads daily delta data using last modified timestamps.
- Final load stage uses Hive SQL to merge into main tables.

4. SCD2 Logic

- **ACID tables** used MERGE statements to update current rows and insert new versions.
- **Non-ACID tables** were managed using a full overwrite strategy. For each load, the entire non-ACID table was rebuilt from the staging data to reflect the latest state. This approach avoids the complexity of row-level updates, which are unsupported in non-transactional tables. This method ensures that the non-ACID tables always reflect the most recent complete snapshot of the dimension.

Data Warehouse Design Transition

As part of the migration from a traditional data warehouse (Redshift) to a distributed Hadoop-based warehouse (Hive on HDFS), I reassessed the data modeling approach. While the initial model followed a classical **star schema**, it became clear that this design did not leverage the full performance potential of distributed systems like Hive.

1. Schema Optimization

Star schemas are optimal for relational databases, but in a distributed environment like Hive, frequent joins across multiple dimensions and fact tables can be costly. To improve performance and reduce shuffle operations, I:

- Flattened the star schema into **denormalized wide tables** (also referred to as **big tables**).
- Partitioned these large tables based on **business-relevant attributes**, such as `reservation_date_key`, `loyalty tier`, or `Farebase`, depending on common query patterns.

This redesign ensures better scan efficiency and parallelism in Hive's distributed query engine.

2. Handling Slowly Changing Dimensions (SCD)

Some dimensions, such as passenger, require history tracking for accurate historical reporting. For these:

- I created **dedicated dimension history tables**.
- Each SCD2 dimension table includes `start_date`, `end_date` fields.

- For performance, SCD handling was done differently for ACID vs non-ACID tables, as previously described.

Accumulating Fact Table Design (Planned - Not Implemented)

Although not yet implemented, I designed a model for **accumulating snapshot fact tables** to track lifecycle processes, such as **flight activity** and **customer complaint resolution**. These tables:

- Follow an **append-only** strategy to simplify data handling in a distributed environment.
- Include multiple status date fields (e.g., `opened_date`, `in_progress_date`, `resolved_date`) to track progression.
- Use a **versioning approach** to capture state transitions over time.

For example, a complaint record may span several status changes. Rather than overwriting rows, each status change appends a new version, preserving the timeline of lifecycle events. This design ensures transparency and supports auditability across customer-related processes.

ELT Pipeline Setup

To automate and modularize the ELT process, I customized the Hive container image by installing **Python** inside it. This enabled me to execute PySpark-style extraction and HQL-based transformations within the same environment.

1. Python-Based Extraction

- A Python script (`main.py`) was created to connect to the source data (postgres OLTP), process it
- The script writes data in CSV format for optimized writing performance.

2. External Table Creation and Transformation

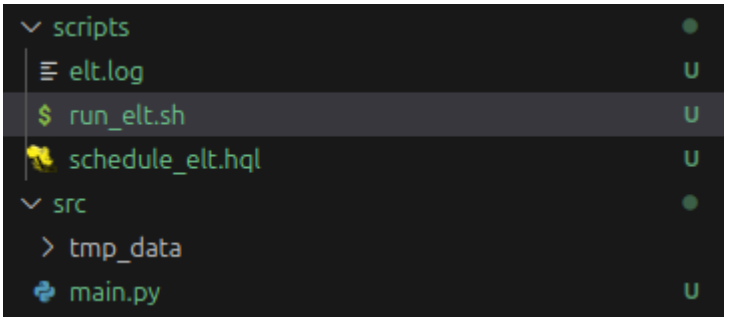
- Using HQL scripts (`schedule_elt.hql`), I defined **external tables** that point to the staged data in HDFS.
- These scripts then perform the necessary transformations, joins, and insertions into the final partitioned fact and dimension tables.

3. Orchestration via Cron

- On the Docker host machine, a `crontab` entry was configured to automate the pipeline.

- The cron job executes a Bash wrapper script (run_elt.sh) via `docker exec`, which:
    - Runs the Python script to extract.
    - Stage the data into hdfs (external tables)
    - Executes the HQL transformation script using Beeline.

```
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
0 12 * * * docker exec -d hive_elt /home/hadoop/scripts/run_elt.sh
```



GitHub Link:

https://github.com/otifi3/hive_cluster