MIMIC-III Big Data Pipeline User Manual: Healthcare Analytics with Hadoop, Hive, and MapReduce

Ahmed Mohamed Srour

May 2025

Prepared by: Ahmed Mohamed Srour

Contents

1 Overview

This user manual provides comprehensive instructions for setting up, operating, and maintaining a big data pipeline for healthcare analytics using the MIMIC-III Clinical Database Demo v1.4. The pipeline processes key tables (PATIENTS, ADMISSIONS, ICUSTAYS, DIAGNOSES_ICD) to enable batch analytics, such as calculating average length of stay, ICU readmission distribution, and mortality rates by demographic groups. It leverages Hadoop for distributed storage, Hive for SQL-based analytics, MapReduce for custom processing, and Docker for containerized deployment.

1.1 Purpose

The pipeline automates the extraction, transformation, and loading (ETL) of healthcare data to:

- Store and manage MIMIC-III data efficiently in Hadoop Distributed File System (HDFS).
- Perform batch analytics using HiveQL and MapReduce for insights into patient outcomes.
- Enable scalable processing of structured healthcare data in Parquet format.

1.2 Scope

The pipeline processes a subset of the MIMIC-III demo dataset (100 patients) and supports:

- Data cleaning and conversion to Parquet for Hadoop and Hive compatibility.
- Distributed storage and analytics using Hadoop, Hive, and MapReduce.
- Containerized deployment for simplified setup and reproducibility.

1.3 Key Components

- **Hadoop**: Distributed storage and processing via HDFS and MapReduce.
- **Hive**: SQL-based querying for batch analytics.
- **MapReduce**: Custom processing for advanced analytics tasks.
- **Docker**: Containerized environment for Hadoop, Spark, and Hive.
- **Python**: Scripts for data preprocessing and cleaning.
- **Git**: Version control for code and configurations.

2 System Requirements

2.1 Hardware

- **CPU**: Minimum 4 cores (6+ recommended for faster processing).
- **RAM**: 16 GB (32 GB recommended for larger datasets).
- **Storage**: 50 GB free disk space for Docker images, HDFS, and data files.

2.2 Software

- **Operating System**: Linux (Ubuntu 20.04+), macOS, or Windows 10/11 with WSL2.
- **Docker**: Docker Desktop or Engine (version 20.10+).
- **Docker Compose**: Version 1.29+.
- **Git**: Version 2.25+ for repository cloning.
- **Python**: Version 3.8+ with `pandas` and `pyarrow` for data preprocessing.
- **Java**: OpenJDK 8 or 11 for Hadoop, Hive, and MapReduce.

3 Installation and Setup

3.1 Prerequisites

- Install **Docker**: Refer to the Docker Get Started Guide.
- Install **Docker Compose**: See Compose Installation.
- Install **Git**: Follow Git Installation.
- Install **Python**: Ensure Python 3.8+ is installed with `pip`.
- Install **Java**: Ensure OpenJDK 8 or 11 is installed.

3.2 Cloning the Repository

Clone the Dockerized Hadoop, Spark, and Hive environment:

```
git clone https://github.com/Marcel-Jan/docker-hadoop-spark.git
cd docker-hadoop-spark
```

3.3 Starting Docker Containers

Launch the containers using Docker Compose:

```
docker-compose up -d
```

Verify containers are running (e.g., `namenode`, `datanode`, `hive-server`):

```
docker ps
```

Note: The environment requires approximately 5-6 GB of storage. An internet connection is needed for initial setup.

3.4 Configuring HDFS and Hive

- **HDFS**: Access the Hadoop NameNode container:

```
docker exec -it namenode bash
```

- **Hive**: Access the Hive server and connect to Beeline:

```
docker exec -it hive-server bash
beeline -u jdbc:hive2://localhost:10000
```

4 Data Preprocessing

4.1 Downloading the MIMIC-III Demo Dataset

Download the MIMIC-III Clinical Database Demo v1.4 from PhysioNet. Extract the following CSV files:

- PATIENTS.csv
- ADMISSIONS.csv
- ICUSTAYS.csv
- DIAGNOSES_ICD.csv

4.2 Data Cleaning and Conversion to Parquet

Clean the dataset to ensure compatibility with Parquet and Hive, including data type conversions and timestamp alignment. Use the following Python script:

```
import pandas as pd
import pyarrow.parquet as pq
import pyarrow as pa
from datetime import datetime

# Define input CSV files
csv_files = ['PATIENTS.csv', 'ADMISSIONS.csv', 'ICUSTAYS.csv', 'DIAGNOSES_ICD.csv']

for file in csv_files:
    # Read CSV
    df = pd.read_csv(file)
    # Data cleaning
    if file == 'PATIENTS.csv':
        # Convert date columns to datetime
        for col in ['DOB', 'DOD', 'DOD_HOSP', 'DOD_SSN']:
            if col in df.columns:
                df[col] = pd.to_datetime(df[col], errors='coerce')
        # Ensure EXPIRE_FLAG is integer
        df['EXPIRE_FLAG'] = df['EXPIRE_FLAG'].astype('int32')
    elif file == 'ADMISSIONS.csv':
        # Convert date columns to datetime
```

```
        for col in ['ADMITTIME', 'DISCHTIME', 'DEATHTIME', 'EDREGTIME', 'EDOUTTIME']:
            if col in df.columns:
                df[col] = pd.to_datetime(df[col], errors='coerce')
        # Ensure IDs are integers
        df['HADM_ID'] = df['HADM_ID'].astype('int32')
        df['SUBJECT_ID'] = df['SUBJECT_ID'].astype('int32')
    elif file == 'ICUSTAYS.csv':
        # Convert date columns to datetime
        for col in ['INTIME', 'OUTTIME']:
            if col in df.columns:
                df[col] = pd.to_datetime(df[col], errors='coerce')
        # Ensure IDs are integers
        for col in ['ICUSTAY_ID', 'HADM_ID', 'SUBJECT_ID']:
            df[col] = df[col].astype('int32')
    elif file == 'DIAGNOSES_ICD.csv':
        # Ensure ICD9_CODE is string and handle missing values
        df['ICD9_CODE'] = df['ICD9_CODE'].fillna('UNKNOWN').astype(str)
        # Ensure IDs and SEQ_NUM are integers
        for col in ['HADM_ID', 'SUBJECT_ID', 'SEQ_NUM']:
            df[col] = df[col].astype('int32')
    # Convert to Parquet
    table = pa.Table.from_pandas(df)
    pq.write_table(table, file.replace('.csv', '.parquet'))
```

Run the script:

```
pip install pandas pyarrow
python convert_to_parquet.py
```

4.3 Loading Parquet Files into HDFS

Copy the Parquet files to HDFS:

```
docker exec -it namenode bash
hdfs dfs -mkdir /mimic_data
hdfs dfs -put /path/to/parquet/*.parquet /mimic_data/
```

Verify files in HDFS:

```
hdfs dfs -ls /mimic_data
```

5 Operating the Pipeline

5.1 Starting the Pipeline

Ensure Docker containers are running:

```
docker-compose up -d
```

5.2 Creating Hive Tables

Access the Hive server and create tables for the Parquet files:

```
docker exec -it hive-server bash
beeline -u jdbc:hive2://localhost:10000
```

Example Hive table creation scripts:

- **PATIENTS**:

```
CREATE EXTERNAL TABLE patients (
    subject_id INT,
    gender STRING,
    dob TIMESTAMP,
    dod TIMESTAMP,
    dod_hosp TIMESTAMP,
    dod_ssn TIMESTAMP,
    expire_flag INT
)
STORED AS PARQUET
LOCATION '/mimic_data/PATIENTS.parquet';
```

- **ADMISSIONS**:

```
CREATE EXTERNAL TABLE admissions (
    subject_id INT,
    hadm_id INT,
    admittime TIMESTAMP,
    dischtime TIMESTAMP,
    deathtime TIMESTAMP,
    admission_type STRING,
    admission_location STRING,
    discharge_location STRING,
    insurance STRING,
    language STRING,
    religion STRING,
    marital_status STRING,
    ethnicity STRING,
    edregtime TIMESTAMP,
    edouttime TIMESTAMP,
    diagnosis STRING,
    hospital_expire_flag INT
)
STORED AS PARQUET
LOCATION '/mimic_data/ADMISSIONS.parquet';
```

- **ICUSTAYS**:

```
CREATE EXTERNAL TABLE icustays (
    subject_id INT,
    hadm_id INT,
    icustay_id INT,
    dbsource STRING,
    first_careunit STRING,
```

```
    last_careunit STRING,
    first_wardid INT,
    last_wardid INT,
    intime TIMESTAMP,
    outtime TIMESTAMP,
    los DOUBLE
)
STORED AS PARQUET
LOCATION '/mimic_data/ICUSTAYS.parquet';
```

- **DIAGNOSES_ICD**:

```
CREATE EXTERNAL TABLE diagnoses_icd (
    subject_id INT,
    hadm_id INT,
    seq_num INT,
    icd9_code STRING
)
STORED AS PARQUET
LOCATION '/mimic_data/DIAGNOSES_ICD.parquet';
```

5.3 Running Hive Analytics

Run HiveQL queries for batch analytics. Example queries:

- **Average Length of Stay per Diagnosis**:

```
SELECT d.icd9_code, AVG(DATEDIFF(a.dischtime, a.admittime)) AS avg_los
FROM admissions a
JOIN diagnoses_icd d ON a.hadm_id = d.hadm_id
GROUP BY d.icd9_code;
```

- **Distribution of ICU Readmissions**:

```
SELECT subject_id, COUNT(DISTINCT hadm_id) AS readmission_count
FROM icustays
GROUP BY subject_id
HAVING readmission_count > 1;
```

- **Mortality Rates by Demographic Groups**:

```
SELECT p.gender,
    COUNT(CASE WHEN p.dod IS NOT NULL THEN 1 END) / CAST(COUNT(*) AS
       DOUBLE) AS mortality_rate
FROM patients p
JOIN admissions a ON p.subject_id = a.subject_id
GROUP BY p.gender;
```

5.4 Running MapReduce Jobs

This section describes how to implement and run a MapReduce job to calculate the average age of patients in the MIMIC-III dataset using the `PATIENTS.csv` file.

5.4.1 Prerequisites

- Ensure Java (OpenJDK 8 or 11) is installed in the `namenode` container.
- Verify that the `PATIENTS.csv` file is available for processing.

5.4.2 Installation and Setup

1. Access the `namenode` container:

```
docker exec -it namenode bash
```

2. Update the package list and install Java:

```
apt update
apt install -y openjdk-8-jdk
```

3. Verify Java installation:

```
java -version
```

4. Install `nano` if not already present:

```
apt update
echo "deb http://archive.debian.org/debian stretch main" > /etc/apt/sources.list
echo "deb http://archive.debian.org/debian-security stretch/updates main" >> /etc/apt/sources.list
apt-get update
apt-get install nano
```

5.4.3 Creating the MapReduce Program

Create a Java program to calculate the average age of patients:

```
nano AverageAge.java
```

Paste the following script into `AverageAge.java`:

```java
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class AverageAge {

    public static class AgeMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
```

```
        private final static Text word = new Text("age");
        private final static SimpleDateFormat format = new
            SimpleDateFormat("yyyy-MM-dd", Locale.ENGLISH);

        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String[] fields = value.toString().split(",");
            if (fields[0].equals("row_id") || fields.length < 5) return;
                // skip header or bad lines

            try {
                Date dob = format.parse(fields[3]);
                Date dod = fields[4].isEmpty() ? format.parse("2200-01-01"
                    ) : format.parse(fields[4]);
                long age = (dod.getTime() - dob.getTime()) / (1000L * 60 *
                     60 * 24 * 365);
                if (age > 0 && age < 150) // sanity check
                    context.write(word, new IntWritable((int) age));
            } catch (Exception e) {
                // Ignore malformed dates
            }
        }
    }

    public static class AvgReducer extends Reducer<Text, IntWritable, Text
        , DoubleWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context
             context) throws IOException, InterruptedException {
            int sum = 0, count = 0;
            for (IntWritable val : values) {
                sum += val.get();
                count++;
            }
            if (count != 0)
                context.write(new Text("Average Age"), new DoubleWritable
                    ((double) sum / count));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "average age");
        job.setJarByClass(AverageAge.class);
        job.setMapperClass(AgeMapper.class);
        job.setReducerClass(AvgReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Save and exit `nano`.

5.4.4 Compiling and Packaging the MapReduce Job

1. Create a directory for compiled classes:

```
mkdir -p avg_classes
```

2. Set the Hadoop classpath:

```
export HADOOP_CLASSPATH=$(hadoop classpath)
```

3. Compile the Java file:

```
javac -classpath $HADOOP_CLASSPATH -d avg_classes AverageAge.java
```

4. Package the compiled classes into a JAR file:

```
jar -cvf avg.jar -C avg_classes/ .
```

5.4.5 Transferring Data to HDFS

1. Copy the `PATIENTS.csv` file to the `namenode` container (from the host machine):

```
docker cp /path/to/Patients.csv namenode:/root/
```

2. Create a directory in HDFS:

```
hdfs dfs -mkdir -p /user/root/mimic
```

3. Upload the CSV file to HDFS:

```
hdfs dfs -put /root/Patients.csv /user/root/mimic/
```

4. Verify the file in HDFS:

```
hdfs dfs -ls /user/root/mimic/
```

5.4.6 Running the MapReduce Job

Run the MapReduce job to calculate the average age:

```
hadoop jar avg.jar AverageAge /user/root/mimic/Patients.csv /user/root/
    output_avg
```

Note: If you rerun the job, delete the previous output directory:

```
hdfs dfs -rm -r /user/root/output_avg
```

5.4.7 Viewing the Output

Display the result:

```
hdfs dfs -cat /user/root/output_avg/part-r-00000
```

6 Troubleshooting

6.1 Docker Issues

- **Containers Fail to Start**:
  - **Error**: Containers exit immediately or fail to initialize.
  - **Solution**:
    * Check container status: `docker ps -a`.
    * View logs: `docker logs <container-name>`.
    * Ensure sufficient memory (16 GB minimum) and no port conflicts (e.g., HDFS ports 9000, 9870; Hive port 10000).
    * Restart containers: `docker-compose down && docker-compose up -d`.
- **Cannot Access Hive Server**:
  - **Error**: Connection refused on `jdbc:hive2://localhost:10000`.
  - **Solution**:
    * Verify Hive server is running: `docker ps | grep hive-server`.
    * Check logs: `docker logs hive-server`.
    * Ensure network connectivity: `docker network ls`.

6.2 HDFS Issues

- **Failed to Upload Parquet Files**:
  - **Error**: `hdfs dfs -put` fails with permission or connection errors.
  - **Solution**:
    * Verify HDFS is running: `hdfs dfsadmin -report`.
  * Check permissions: `hdfs dfs -chmod -R 777 /mimic`$_{d}ata.EnsureNameNodeisaccessible:$ `telnet namenode 9000`.
- **Corrupted Parquet Files**:
  - **Error**: Hive queries fail with schema mismatch or corruption errors.
  - **Solution**:
    * Regenerate Parquet files using `convert`$_{t}o_{p}arquet.py.EnsuredatatypesmatchHivetableschema(e.g.,$
    * Inspect CSV files for malformed data before conversion.

6.3 Hive Issues

- **Query Fails with Schema Mismatch**:
  - **Error**: Column type mismatch or invalid column type.
  - **Solution**:
    * Ensure Hive table schema matches Parquet file schema.
    * Regenerate Parquet files with correct data types.

  * Drop and recreate table: `DROP TABLE patients; CREATE EXTERNAL TABLE patients....`

- **Slow Query Performance**:
  - **Error**: Queries take too long.
  - **Solution**:
  * Partition tables by `subject`$_{i}dor$`hadm`$_{i}d.IncreaseHivememoryin$`hive-site.xml`(`hive.tez.container.`

6.4 Data Cleaning Issues

- **Timestamp Parsing Errors**:
  - **Error**: Invalid timestamp format during Parquet conversion.
  - **Solution**:
  * Use `errors='coerce'` in `pd.to`$_{d}atetimein$`convert`$_{t}o_{p}arquet.py.StandardizedateformatsinCSVfilesb$

- **Data Type Mismatch in Hive**:
  - **Error**: Queries fail due to type mismatch (e.g., STRING vs. INT).
  - **Solution**:
  * Verify data types in `convert`$_{t}o_{p}arquet.py(e.g.,$ `int32`$forIDs,$ `string`for `ICD9`$_{C}ODE).UpdateHivetable$

6.5 General Troubleshooting Tips

- **Check Logs**: Use `docker logs <container-name>` for detailed errors.
- **Restart Services**: Run `docker-compose restart` for transient issues.
- **Update Images**: Run `docker-compose pull` for the latest Docker images.
- **Contact Support**: For unresolved issues, create an issue in the private Git repository or contact the project owner.

7 Technical Specifications

7.1 Architecture

The pipeline uses a containerized architecture with:

- **Hadoop**: Stores MIMIC-III data in HDFS and supports MapReduce.
- **Hive**: Executes SQL-based analytics on Parquet files.
- **MapReduce**: Performs custom analytics, such as calculating average patient age.
- **Docker**: Runs Hadoop, Hive, and supporting services in isolated containers.

7.2 Data Flow

1. **Extraction**: Download MIMIC-III demo CSV files from PhysioNet.
2. **Transformation**: Clean data (data types, timestamps) and convert to Parquet using Python.
3. **Loading**: Upload Parquet files to HDFS.
4. **Analytics**: Create Hive tables, run HiveQL queries, or execute MapReduce jobs.

7.3 Data Flow Diagram

1. MIMIC-III CSV Files → (Python Cleaning and Conversion) →
2. Parquet Files → (HDFS Upload) →
3. HDFS Storage → (Hive Table Creation or MapReduce Job) →
4. Analytics (e.g., length of stay, readmissions, average age).

8 Backup and Recovery

8.1 Data Backup Procedures

- **HDFS Backup**:
  - Export HDFS data to local storage:

```
hdfs dfs -get /mimic_data /backup/mimic_data_$(date +%F)
```

  - Schedule backups using a cron job or manual script.

- **Hive Metadata Backup**:
  - Export Hive table definitions:

```
docker exec hive-server hive -e 'SHOW CREATE TABLE patients;' > /
   backup/patients_ddl.sql
```

  - Repeat for other tables.

8.2 Data Recovery Procedures

- **HDFS Recovery**:
  - Restore data to HDFS:

```
hdfs dfs -put /backup/mimic_data_<date> /mimic_data
```

- **Hive Recovery**:
  - Recreate tables using backed-up DDL scripts:

```
docker exec hive-server hive -f /backup/patients_ddl.sql
```

8.3 Maintenance of Backup Process

- Schedule regular HDFS backups (e.g., weekly).
- Verify backups by restoring to a test environment.
- Define a retention policy (e.g., keep backups for 30 days).

9 Appendices

9.1 Glossary

- **Hadoop**: Open-source framework for distributed storage and processing.
- **HDFS**: Hadoop Distributed File System for large datasets.
- **Hive**: Data warehouse infrastructure for SQL-based querying on Hadoop.
- **MapReduce**: Programming model for processing large datasets in parallel.
- **Parquet**: Columnar storage format optimized for big data.
- **HiveQL**: SQL-like query language for Hive.
- **Docker**: Platform for containerizing applications.
- **MIMIC-III**: Freely accessible critical care database.