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

1 Overview

This user manual provides detailed 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 four key tables (PATIENTS, ADMISSIONS, ICUSTAYS, DIAGNOSES_ICD) to enable batch analytics such as average length of stay, ICU readmission distribution, and mortality rates by demographic groups. The pipeline leverages Hadoop for distributed storage, Hive for SQL-based analytics, and Docker for containerized deployment.

1.1 Purpose

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

- Efficient storage and management of MIMIC-III data in Hadoop Distributed File System (HDFS).
- Batch analytics using HiveQL for insights into patient outcomes.
- 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 compatibility with Hadoop and Hive.
- Distributed storage and analytics using Hadoop and Hive.
- Containerized deployment for easy setup and reproducibility.

1.3 Key Components

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

2 System Requirements

2.1 Hardware

- **CPU**: 4 cores (6+ recommended for faster processing).
- **RAM**: 16 GB (32 GB recommended for large 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 and Hive.

3 Installation and Setup

3.1 Prerequisites

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

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
```

The environment size is approximately 5-6 GB. An internet connection is required for the initial setup.

3.4 Configuring HDFS and Hive

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

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

- **Hive**: Access the Hive server:

  ```
  docker exec -it hive-server bash
  hive
  ```

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

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

```
import pandas as pd
import numpy as np
import pyarrow as pa
import pyarrow.parquet as pq
from pyarrow import csv


# Configure paths
INPUT_PATH = r"PATH/to/MIMMIC_DATASET/ .CSV/" #put the path of mimic dataset
OUTPUT_DIR = r"PATH/to/CLEANED_dir/"                    #put the path to desired output dir

csv_files = [INPUT_PATH +'PATIENTS.csv', INPUT_PATH +'ADMISSIONS.csv', INPUT_PATH
+'ICUSTAYS.csv', INPUT_PATH +'DIAGNOSES_ICD.csv']

for file in csv_files: # Read CSV
    df = pd. read_csv(file)


# Data cleaning
    if file == INPUT_PATH + 'PATIENTS.csv':
        df.to_csv(OUTPUT_DIR+ 'patients.csv', index=False)
        df.to_parquet(OUTPUT_DIR + 'patients.parquet', engine='pyarrow', index=False)

    # Step 1: Read with explicit dtype conversion
        table = pq.read_table(OUTPUT_DIR + 'PATIENTS.parquet')
        schema = pa.schema([
        pa.field('row_id', pa.int64()),
        pa.field('subject_id', pa.int64()),
        pa.field('gender', pa.string()),
        pa.field('dob', pa.timestamp('ns')),
        pa.field('dod', pa.timestamp('ns')),
        pa.field('dod_hosp', pa.timestamp('ns')),
        pa.field('dod_ssn', pa.timestamp('ns')),
        pa.field('expire_flag', pa.int32())     ]) # CRITICAL: Force INT32


# Step 2: Write with hive-compatible settings
        pq.write_table(
        table.cast(schema),
        OUTPUT_DIR +'patients.parquet',
        version='2.6',
        use_dictionary=True,
        compression='SNAPPY')

    elif file == INPUT_PATH + 'ADMISSIONS.csv':
        #fill null text with unkonwn
        text_columns = ['language', 'religion', 'marital_status', 'ethnicity', 'diagnosis']
        df[text_columns] = df[text_columns].fillna('UNKNOWN')
        #convert to category
        cat_cols = ['admission_type', 'admission_location', 'discharge_location','insurance', 'language',
'religion', 'marital_status', 'ethnicity']
        df[cat_cols] = df[cat_cols].astype('category')
```

```
#convert to datetime
    time_cols = ['admittime', 'dischtime', 'deathtime', 'edregtime', 'edouttime']
    for col in time_cols:
        df[col] = pd.to_datetime(df[col], errors='coerce')

    #convert to boolean
    df['hospital_expire_flag'] = df['hospital_expire_flag'].astype(bool)
    df['has_chartevents_data'] = df['has_chartevents_data'].astype(bool)
    #Converts all values to uppercase and removes leading/trailing whitespace to avoid mismatches
    df['ethnicity'] = df['ethnicity'].str.upper().str.strip()
    #multiple granular or noisy categories into a smaller set of standardized groups
    df['ethnicity'] = df['ethnicity'].replace({
    'WHITE': 'WHITE',
    'WHITE - OTHER EUROPEAN': 'WHITE',
    'WHITE - EASTERN EUROPEAN': 'WHITE',
    'WHITE - BRAZILIAN': 'WHITE',
    'WHITE - RUSSIAN': 'WHITE',
    'BLACK/AFRICAN AMERICAN': 'BLACK',
    'BLACK/CARIBBEAN ISLAND': 'BLACK',
    'ASIAN': 'ASIAN',
    'HISPANIC OR LATINO': 'HISPANIC',
    'HISPANIC/LATINO - PUERTO RICAN': 'HISPANIC',
    'UNKNOWN/NOT SPECIFIED': 'UNKNOWN',
    'UNABLE TO OBTAIN': 'UNKNOWN'})
    # correct mairtal status
    df['marital_status'] = df['marital_status'].str.upper().str.strip()
    df['marital_status'] = df['marital_status'].replace({
    'UNKNOWN (DEFAULT)': 'UNKNOWN',
    'LIFE PARTNER': 'PARTNER'})
    #calc how long patients stayed at hosiptal?
    df['los_days_hos'] = (df['dischtime'] - df['admittime']).dt.days
    df['los_days_hos'] = df['los_days_hos'].round().astype(int)

    #Converts all values to uppercase and removes leading/trailing whitespace to avoid mismatches
    df['diagnosis'] = df['diagnosis'].str.upper().str.strip()
    # just extract date to be easy to group by date only
    df['admit_date'] = df['admittime'].dt.date
    df['discharge_date'] = df['dischtime'].dt.date
    df['admit_date'] = pd.to_datetime(df['admit_date'], errors='coerce')
    df['discharge_date'] = pd.to_datetime(df['discharge_date'], errors='coerce')
    #convert to csv after cleaned
    df.to_csv(OUTPUT_DIR+ 'Admissions.csv', index=False)
    #convert to parquet
    df.to_parquet(OUTPUT_DIR + 'admissions.parquet', engine='pyarrow', index=False)
```

```
table = pq.read_table(OUTPUT_DIR + 'admissions.parquet')
    schema = pa.schema([
    pa.field('row_id', pa.int64()),
    pa.field('subject_id', pa.int64()),
    pa.field('hadm_id', pa.int64()),
    pa.field('admittime', pa.timestamp('ns')),
    pa.field('dischtime', pa.timestamp('ns')),
    pa.field('deathtime', pa.timestamp('ns')),
    pa.field('admission_type', pa.dictionary(pa.int8(), pa.string())),
    pa.field('admission_location', pa.dictionary(pa.int8(), pa.string())),
    pa.field('discharge_location', pa.dictionary(pa.int8(), pa.string())),
    pa.field('insurance', pa.dictionary(pa.int8(), pa.string())),
    pa.field('language', pa.dictionary(pa.int8(), pa.string())),
    pa.field('religion', pa.dictionary(pa.int8(), pa.string())),
    pa.field('marital_status', pa.dictionary(pa.int8(), pa.string())),
    pa.field('ethnicity', pa.dictionary(pa.int8(), pa.string())),
    pa.field('edregtime', pa.timestamp('ns')),
    pa.field('edouttime', pa.timestamp('ns')),
    pa.field('diagnosis', pa.string()),
    pa.field('hospital_expire_flag', pa.bool_()),
    pa.field('has_chartevents_data', pa.bool_()),
    pa.field('los_days_hos', pa.int64()),
    pa.field('admit_date', pa.timestamp('ns')),
    pa.field('discharge_date', pa.timestamp('ns')) ])


    pq.write_table(
    table.cast(schema),
    OUTPUT_DIR + 'admissions.parquet',
    version='2.6',
    use_dictionary=True,
    compression='SNAPPY',
    )

  elif file == INPUT_PATH + 'ICUSTAYS.csv':
    df.drop(columns=['dbsource','first_careunit','last_careunit','first_wardid','last_wardid'],inplace=True)
    df['los_hours'] = df['los'] * 24
    df['los_hours'] = df['los'].round().astype(int)
    df.drop(columns=['los'],inplace=True)
     #convert to csv after cleaned
    df.to_csv(OUTPUT_DIR+ 'ICUSTAYS.csv', index=False)
    #convert to parquet
    df.to_parquet(OUTPUT_DIR + 'ICUSTAYS.parquet', engine='pyarrow', index=False)
    table = pq.read_table(OUTPUT_DIR + 'ICUSTAYS.parquet')

    schema = pa.schema([
    ('row_id', pa.int64()),
    ('subject_id', pa.int64()),
    ('hadm_id', pa.int64()),
    ('icustay_id', pa.int64()),
    ('intime', pa.timestamp('ns')),
    ('outtime', pa.timestamp('ns')),
    ('los_hours',pa.int64()) ])

    pq.write_table(table.cast(schema),
    OUTPUT_DIR +'ICUSTAYS.parquet',
    version='2.6',
    use_dictionary=True,
    compression='SNAPPY' )
```

```
elif file == INPUT_PATH + 'DIAGNOSES_ICD.csv':
        df.to_parquet(OUTPUT_DIR +'diagnoses_icd.parquet',engine= 'pyarrow',index=False )
        # Step 1: Read with explicit dtype conversion
        table = pq.read_table(OUTPUT_DIR + 'diagnoses_icd.parquet')
        schema = pa.schema([
        pa.field('row_id', pa.int64()),
        pa.field('subject_id', pa.int64()),
        pa.field('hadm_id', pa.int64()),
        pa.field('seq_num', pa.int64()),
        pa.field('icd9_code', pa.string()) ])

# Step 2: Write with hive-compatible settings
        pq.write_table(
        table.cast(schema),
        OUTPUT_DIR +'diagnoses_icd.parquet',
        version='2.6',
        use_dictionary=True,
        compression='SNAPPY')
```

Run the script:

```
pip install pandas numpy pyarrow
python clean.py
```

Loading Parquet Files into HDFS

Copy the Parquet files to HDFS:

```
docker cp PATIENTS.parquet namenode:/tmp/

docker exec -it namenode bash

hdfs dfs -mkdir -p /user/hive/warehouse/patients_data/

hdfs dfs -put /tmp/PATIENTS.parquet /user/hive/warehouse/patients_data/

# do for all files
```

Verify files in HDFS:

```
hdfs dfs -ls /user/hive/warehouse
```

5 Operating the Pipeline

5.1 Creating Hive Tables

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

```
docker exec -it hive-server bash
 hive
```

Example Hive table creation scripts:

PATIENTS:

```
CREATE EXTERNAL TABLE patients (
  row_id BIGINT,
  subject_id BIGINT,
  gender STRING,
  dob TIMESTAMP,
  dod TIMESTAMP,
  dod_hosp TIMESTAMP,
  dod_ssn TIMESTAMP,
  expire_flag INT
)
STORED AS PARQUET
LOCATION '/user/hive/warehouse/patients_data/'
TBLPROPERTIES (
  'parquet.compress'='SNAPPY',
  'parquet.validation'='STRICT'
);
```

ADMISSIONS:

```
CREATE EXTERNAL TABLE ADMISSIONS (
    row_id BIGINT,
    subject_id BIGINT,
    hadm_id BIGINT,
    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 BOOLEAN,
    has_chartevents_data BOOLEAN,
    los_days_hos BIGINT
    admit_date TIMESTAMP,
    discharge_date TIMESTAMP
)
STORED AS PARQUET
LOCATION '/user/hive/warehouse/admission_data/'
TBLPROPERTIES (
  'parquet.compress'='SNAPPY',
  'parquet.validation'='STRICT'
);
```

ICUSTAYS:

```
CREATE EXTERNAL TABLE ICUSTAYS (
    row_id BIGINT,
    subject_id BIGINT,
    hadm_id BIGINT,
    icustay_id BIGINT,
    intime TIMESTAMP,
    outtime TIMESTAMP,
    los_hours BIGINT
)
STORED AS PARQUET
LOCATION '/user/hive/warehouse/icustays_data/'
TBLPROPERTIES (
  'parquet.compress'='SNAPPY',
  'parquet.validation'='STRICT'
);
```

DIAGNOSES_ICD:

```
create external table diagnoses_icd (
     row_id BIGINT,
     subject_id BIGINT,
     hadm_id BIGINT,
     seq_num BIGINT,
     icd9_code STRING
     )
     STORED AS PARQUET
     LOCATION '/user/hive/warehouse/patients_data/
     TBLPROPERTIES (
     'parquet.compress'='SNAPPY'
     'parquer.validation'=STRICT'
     );
```

5.2 Running Hive Analytics

Run HiveQL queries for batch analytics. Example queries:

5.2.1 Average Length of Stay per Diagnosis

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

5.2.2 Distribution of ICU Readmissions

```
SELECT
   readmit_flag,
   COUNT(*) AS num_patients
FROM (
   SELECT
      subject_id,
      hadm_id,
      COUNT(icustay_id) AS icu_visits,
      CASE
         WHEN COUNT(icustay_id) > 1 THEN 'Readmitted'
         ELSE 'Single Stay'
      END AS readmit_flag
   FROM
      icustays
   GROUP BY
      subject_id, hadm_id
```

5.2.3 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;
```

5.3 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.

5.3.1 Prerequisites

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

5.3.2 Installation and Setup

Access the namenode container:

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

Update the package list and install Java

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

Verify Java installation

```
java -version
```

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
```

5.3.3 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> {
```

```java
        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();
```

```java
        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);
    }
}
```

5.3.4 Compiling and Packaging the MapReduce Job

Create a directory for compiled classes

```
mkdir -p avg_classes
```

Set the Hadoop classpath

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

Compile the Java file

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

Package the compiled classes into a JAR file

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

5.3.5 Running the MapReduce Job

```
hadoop jar average_age.jar AverageAgeMR \
  /user/hive/warehouse/age_data
```

6 Troubleshooting

This section addresses common issues and their resolutions for the MIMIC-III big data pipeline.

6.1 Docker Issues

1. **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>` (e.g., `namenode`).
     - 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`.
2. **Cannot Access Hive Server**
   - **Error**: Connection refused on `jdbc:hive2://localhost:10000`.
   - **Solution**:
     - Verify Hive server is running: `docker ps | grep hive-server`.
     - Check Hive logs: `docker logs hive-server`.
     - Ensure network connectivity within Docker network: `docker network ls`.

6.2 HDFS Issues

1. **Failed to Upload Parquet Files to HDFS**
   - **Error**: `hdfs dfs -put` fails with permission or connection errors.
   - **Solution**:
     - Verify HDFS is running: `hdfs dfs admin -report`.
     - Check permissions: `hdfs dfs -chmod -R 777 / user/hive/warehouse/`.
     - Ensure NameNode is accessible: `telnet namenode 9000`.

2. **Corrupted Parquet Files**
   - **Error**: Hive queries fail with Parquet schema mismatch or corruption errors.
   - **Solution**:
     - Re-run the convert_to_parquet.py script to regenerate Parquet files.
     - Verify data types in Python script match Hive table schema (e.g., TIMESTAMP for date columns, INT for IDs).
     - Check for missing or malformed data in CSV files before conversion.

6.3 Hive Issues

1. **Query Fails with Schema Mismatch**
   - **Error**: Column type mismatch or Invalid column type when running Hive queries.
   - **Solution**:
     - Ensure Hive table schema matches Parquet file schema (e.g., TIMESTAMP for admittime, INT for subject_id).
     - Re-run convert_to_parquet.py with correct data type conversions.
     - Drop and recreate Hive table: DROP TABLE patients; CREATE EXTERNAL TABLE patients ....
2. **Slow Query Performance**
   - **Error**: Hive queries take too long to execute.
   - **Solution**:
     - Partition tables by subject_id or hadm_id for large datasets.
     - Increase Hive memory: Update hive-site.xml with higher hive.tez.container.size
     - Optimize joins by filtering data early in queries.

6.4 Data Cleaning Issues

1. **Timestamp Parsing Errors**
   - **Error**: Invalid timestamp format during Parquet conversion.
   - **Solution**:
     - Ensure pd.to_datetime in convert_to_parquet.py uses errors='coerce' to handle invalid timestamps.
     - Inspect CSV files for inconsistent date formats and standardize them before conversion.
2. **Data Type Mismatch in Hive**
   - **Error**: Hive queries fail due to type mismatch (e.g., STRING vs. INT).
   - **Solution**:
     - Verify data types in convert_to_parquet.py (e.g., int32 for IDs, string for ICD9_CODE).
     - Update Hive table schema to match Parquet file types.

6.5 General Troubleshooting Tips

- **Check Logs**: Use docker logs <container-name> for detailed error messages.
- **Restart Services**: docker-compose restart to resolve transient issues.
- **Update Images**: docker-compose pull to ensure 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 the following components:

- **Hadoop**: Stores MIMIC-III data in HDFS and supports MapReduce for processing.
- **Hive**: Executes SQL-based analytics on Parquet files.
- **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 (convert data types, align timestamps) and convert to Parquet using Python.
3. **Loading**: Upload Parquet files to HDFS.
4. **Analytics**: Create Hive tables and run batch queries for analytics.

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) →
4. Hive Analytics (e.g., length of stay, readmissions).

8 Appendices

8.1 Glossary

- **Hadoop**: An open-source framework for distributed storage and processing.
- **HDFS**: Hadoop Distributed File System, used for storing large datasets.
- **Hive**: A data warehouse infrastructure for SQL-based querying on Hadoop.
- **Parquet**: A columnar storage format optimized for big data processing.
- **HiveQL**: SQL-like query language for Hive.
- **Docker**: A platform for containerizing applications.
- **MIMIC-III**: A freely accessible critical care database.