

COMM017 - Big Data and Cloud Computing

Course Work 02

Patient Record Classification using MapReduce

Rivindi Kalasing Ekanayake

24012043

MSc Advanced Computer Science

Dr Hamzah AlZubi

Submission on 30th May 2025

Abstract

The design, development, and assessment of a Hadoop MapReduce-based cloud-deployed patient record classification system on a YARN-managed cluster are described in this paper. The project uses a simulation of an NHS-like setting to dynamically classify patient records according to patient ID, chief complaint (symptom), or NHS region in response to the growing amount of electronic healthcare data. The solution illustrates how distributed big data technologies can facilitate scalable and timely healthcare analytics, adhering to the guidelines of the coursework.

The solution, which was developed with Java and implemented on Microsoft Azure, supports medical data retrieval tasks by utilising the parallel processing capability of MapReduce and the flexibility of parameter-driven filtering. The project met its goals by demonstrating scalable categorisation, effective resource management with YARN, and modular design for reuse, despite difficulties with cloud deployment, runtime parameterisation, and output management. This illustrates a real-world use of big data analytics in the field of health informatics.

Table of Contents

Abstract	ii
Table of Contents	iii
List of Figures	v
Patient Record Classification using MapReduce	7
1. Introduction.....	7
1.1. Case Study Overview.....	7
1.2. Significance of the study.....	8
2. Literature Review: Leveraging Hadoop-Based Big Data Architectures for Scalable Healthcare Analytics	9
3. Implementation Desing.....	11
3.1. System Architecture	11
3.2. Objectives of the Implementation	11
3.3. Data set description.....	12
3.4. Implementation on Azure.....	12
4. Code explanation	14
4.1. Mapper Phase: PatientDataMapper.java	14
4.1.1. Imports and class declaration.....	15
4.1.2. Variable declaration	16
4.1.3. Setup Method	16
4.1.4. Map Method.....	16
4.1.5. Filtering logic.....	17
4.1.6. Overview of PatientDataMapper class.....	18
4.2. Reducer Phase: PatientDataReducer.java	18
4.2.1. Imports	19
4.2.2. Class declaration	19

4.2.3.	Reduce Method	20
4.2.4.	Overview of the PatientDataReducer class	20
4.3.	PatientDataDriver.java: Configuring and Starting the Job	21
4.3.1.	Imports	21
4.3.2.	Class declaration	22
4.3.3.	Main method	22
4.3.4.	Job configuration and Job setup.....	22
4.3.5.	Input/output path setup	23
4.3.6.	Overview of the PatientDataDriver class.....	24
4.4.	Mounting Local File with Azure Virtual Machine.....	24
4.5.	Run Hadoop in YARN	24
4.5.1.	Application of Hadoop and YARN in the Azure-Based System.....	25
4.6.	Output	26
5.	Potential Applications of Patient Record Classification using MapReduce	28
6.	Conclusion	30
7.	References.....	xxxi
8.	Appendix.....	xxxii
	PatientDataMapper Code	xxxii
	PatientDataReducer code	xxxiv
	PatientDataDriver Code	xxxiv

List of Figures

Figure 1-Azure virtual machine	13
Figure 2- create PatientMapper.java	14
Figure 3-PatientDataMapper code I.....	15
Figure 4-PatientDataMapper code II	15
Figure 5-Import libraries in DataMapper.....	15
Figure 6-PatientDataMapper class.....	15
Figure 7-Variables in DataMapper.....	16
Figure 8-setup method – DataMapper	16
Figure 9-Map method DataMapper	16
Figure 10-split CSV	17
Figure 11-filtering logic	17
Figure 12-Create PatientDataReducer	18
Figure 13-patientDataReducer code	19
Figure 14-import libraries to the DataReducer	19
Figure 15-DPatientDataReducer class	19
Figure 16-Reduce method in DataReducer.....	20
Figure 17-PatientDataDriver create	21
Figure 18-PatientDataDriver code	21
Figure 19-Import libraries to the DataDriver.....	21
Figure 20-DataDriver class	22
Figure 21-Main method of DataDriver	22
Figure 22-job setup	23
Figure 23-Input/Output path setup.....	23
Figure 24-MedicalFiles CSV uploading to the Azure bolb storage	24
Figure 25-Output download from Azure VM blob storage as the CSV file	24
Figure 26-Compile PatientDataDriver Class	24
Figure 27-create jar	25
Figure 28-Run jar file using hadoop	25
Figure 29-Run jar file using YARN	25
Figure 30-Virtual Machines on Azure.....	25
Figure 31-search for the Hypertension	26

Figure 32-Generated output file.....	26
Figure 33-Search for the Patient Id:8374926150.....	26
Figure 34-Generated output file.....	27
Figure 35-Search for the England.....	27
Figure 36-Generated output file.....	27

Patient Record Classification using MapReduce

1. Introduction

A significant and constantly growing volume of patient data characterises the current healthcare setting. The ability to effectively handle and evaluate this data is crucial for making timely decisions that can improve outcomes for patients. With such massive volumes of data, traditional data processing methods are insufficient, necessitating the use of state-of-the-art big data technologies. Hadoop has emerged as a fundamental framework for big data processing thanks to its ability to distribute data across multiple nodes and handle large-scale processing tasks efficiently. An essential component of the open-source platform, YARN (Yet Another Resource Negotiator) organises jobs and manages resources throughout a cluster, which is why data-intensive applications are frequently run on it.

This project develops a scalable method for filtering patient records from large CSV files according to particular criteria like patient ID, geography, or symptoms using the Hadoop YARN framework. Virtual machines hosted on Microsoft Azure are used for the execution, which makes use of distributed processing and resource management technologies with Hadoop and YARN integration. This setup ensures that healthcare organisations can meet their growing data needs while maintaining the solution's durability, flexibility, and adaptability.

1.1. Case Study Overview

This coursework simulates a real-world healthcare situation in which a growing number of electronic medical records (EMRs) requires strong big data technologies to classify patient data. Synthetic patient data, including NHS number, symptom descriptions (chief complaint), and NHS Trust regions, are included in the dataset. In order to illustrate scalable, parallel processing on a distributed system with YARN support, these records will be categorised using Hadoop MapReduce.

Traceability is aided by the patient ID classification, tracking of common health conditions is aided by the symptom classification, and localised health resource planning is facilitated by the region classification. This configuration is similar to real-world applications seen in NHS public health

analytics, regional hospital networks, and health monitoring systems (Rakesh Raja, Indrajit Mukherjee, and Bikash Kanti Sarkar, 2020).

1.2. Significance of the study

Categorising patient data is crucial for controlling regional healthcare loads, symptom tracking, and personalised care in general healthcare practice. High availability and distributed computing capabilities are introduced by using MapReduce for this operation, which shortens the time needed to analyse data while preserving correctness and consistency.

- The MapReduce-based classifier in this case study allows:
- classification of thousands of patient records in an efficient manner.
- Quick examination of symptom patterns to identify possible outbreaks.
- regional divisions to support resource allocation and planning for NHS Trust.
- infrastructure that is replicable and scalable that can be implemented in national health systems.

2. Literature Review: Leveraging Hadoop-Based Big Data Architectures for Scalable Healthcare Analytics

The use of big data technology in the medical field has been accelerated by the digitisation of healthcare data, which includes everything from administrative and operational datasets to electronic health records (EHRs) and diagnostic imaging. Enhancing disease prediction, optimising healthcare resource utilisation, personalising treatment strategies, and streamlining clinical workflows are all made possible by big data analytics.

certain studies emphasise that enormous-scale pattern identification across enormous and complicated datasets is made possible by big data frameworks (Blagoj Ristevski, Ming Chen, 2018). By revealing hidden patterns in genetic sequences, diagnostic measurements, and patient histories, these systems can help improve decision-making and lower diagnostic mistakes. The authors also point out that distributed computing models like MapReduce enable platforms like Apache Hadoop to handle the intake and processing of unstructured data, which is a prevalent feature in clinical settings. High throughput, parallel processing, and fault tolerance are provided by Hadoop's modular architecture, which consists of the Hadoop Distributed File System (HDFS), MapReduce, and YARN (Yet Another Resource Negotiator). According to Palanisamy and Thirunavukarasu (2019), this architecture is particularly helpful in public health settings where data processing across dispersed nodes must be reliable and efficient (Palanisamy, R. and Thirunavukarasu, 2017). Specifically, MapReduce makes it possible to classify medical data efficiently, including processing diagnostic reports and clinical notes.

In the United Kingdom, the National Health Service (NHS) has identified the strategic value of big data through programs such as the NHS Data Strategy. This framework encourages the creation of safe and interoperable digital infrastructure to aid research, policymaking, and operational enhancements (NHS, 2023). The Hadoop-based architecture simulated in this course reflects these national objectives by proving the viability of distributed processing for patient categorisation tasks.

Using distributed computing systems like Apache Hadoop, which enable effective storage and concurrent processing of large datasets, is a fundamental component of big data processing in the healthcare industry. In Hadoop, the MapReduce programming model is essential for turning unprocessed medical data into information that can be put to use. The Map phase of this approach

classifies and divides patient records, while the Reduce phase aggregates and analyses data along important dimensions such as disease kind, demographic characteristics, and geographic location. Applications in healthcare where structured dataset analysis and categorisation are crucial are a good fit for this methodology.

The theoretical underpinnings and real-world benefits of using big data frameworks, especially Hadoop-YARN, in healthcare informatics are thus highlighted across the literature. Advanced analytics like patient stratification, population health monitoring, and predictive diagnoses are made possible by these technologies, which provide scalable, fault-tolerant, and affordable solutions for handling large and varied health datasets. Building on these realisations, the current coursework uses a MapReduce-based architecture to create a healthcare record classification system (Nazari E, Shahriari MH, Tabesh H, 2019). This application shows how the concepts of distributed computing may be used to effectively classify patient records and produce structured outputs that are suited for clinical decision-making. The system demonstrates the feasibility of big data infrastructures in meeting the changing needs of clinical data analytics by mimicking a real-world scenario that is in line with NHS digital strategies.

3. Implementation Desing

3.1. System Architecture

The system architecture employs a classical MapReduce model tailored for structured medical data classification, designed to efficiently process and organize large-scale patient records. To provide distributed and fault-tolerant storage, CSV data is first ingested into the Hadoop Distributed File System (HDFS). Each patient record is parsed during the Map phase to produce composite keys that aid in data segmentation and classification based on the location, principal complaint, and NHS number. Hadoop then groups related records for processing by shuffling and sorting these key-value pairs. In order to generate organised, categorised outputs that are appropriate for analysis, such as counts, averages, or filtered subsets, the Reduce phase combines values under each key. In order to provide users with organised and useful information that is pertinent to particular medical conditions or demographics, the processed results are finally extracted from HDFS and assembled into downloadable CSV files in response to user requests.

3.2. Objectives of the Implementation

Building a scalable, effective, and distributed system for Hadoop MapReduce-based healthcare record classification in a YARN-managed environment is the main goal of this project. These goals complement the demands of real-world healthcare, where there is a large amount of data that is constantly expanding and necessitates prompt classification and analysis.

Efficiency and Scalability

Ensuring the system's scalability is one of the project's main goals. Continuously treating more patients and keeping more thorough records of clinical observations, prescriptions, and care plans result in ever-increasing datasets for healthcare organisations. Scalable processing and storage of such large datasets are made possible by Hadoop. The system can easily grow up by adding additional nodes to the Hadoop cluster, handling higher patient record processing volumes without sacrificing efficiency.

YARN's Resource Management

It is essential for coordinating the distribution of resources and the completion of tasks. It guarantees that the cluster's computational resources are used effectively. By dynamically allocating resources to different applications, YARN optimises system load while enabling many

jobs to perform concurrently. Even with high processing demands, our dynamic scheduling technique guarantees load balancing, equitable distribution, and few bottlenecks.

Analysis and Filtering of Data

Classifying patient data according to NHS number, symptoms (principal complaint), and NHS region is the system's primary role. Healthcare practitioners can retrieve specific patient information with the use of this type of filtering. For example, region-based filters help with trend analysis and resource allocation, while symptom-based filters can help identify outbreaks.

A MapReduce paradigm is used to implement this classification and filtering: the Mapper reads the CSV input and generates key-value pairs according to the classification type, which the Reducer then aggregates and outputs. This guarantees a scalable, effective, and distributed approach to handling massive amounts of medical data.

3.3. Data set description

The dataset used in this study, titled MedicalFiles.csv, includes structured fields such as PatientNHSNumber (which acts as a unique identification), Name, Age, Gender, Chief Complaint, History of Present Illness, NHS Trust Region. Chief_Complaint is used to classify patient symptoms, while NHS_Trust_Region facilitates the segmentation of data within regions. The dataset is subjected to preprocessing procedures before processing, which include standardising text fields (especially clinical descriptions), eliminating records that are incomplete or distorted, and verifying structural consistency among entries to make sure the CSV follows a common schema. In the MapReduce processing pipeline, these procedures are essential for enabling accurate classification and dependable parsing.

3.4. Implementation on Azure

Using a Microsoft Azure virtual machine for this project offers a number of operational and architectural benefits. Rapid Hadoop cluster deployment is made possible by Azure's incredibly dependable and adaptable cloud environment, which allows virtual machines to be provided rapidly. These virtual computers can be scaled to meet processing demands and provide high availability.

Azure is perfect for organisations like healthcare organisations that operate in multiple places

because of its globally distributed infrastructure, which facilitates wide geographic access and collaboration. In this case, using Azure guarantees safe, remote access to important datasets and processing workflows while simultaneously lessening the strain on hardware and infrastructure. Because of this, Azure is a wise option for deploying healthcare analytics applications that require dispersed data access, scalability, and durability. Hadoop was set up in pseudo-distributed mode on Ubuntu-based virtual machines that were provisioned using Microsoft Azure. The source dataset (MedicalFiles.csv) was uploaded remotely to Azure Blob Storage and then moved to the Hadoop Distributed File System (HDFS) for processing. This cloud-based architecture exhibits scalability and flexibility appropriate for practical health informatics settings.

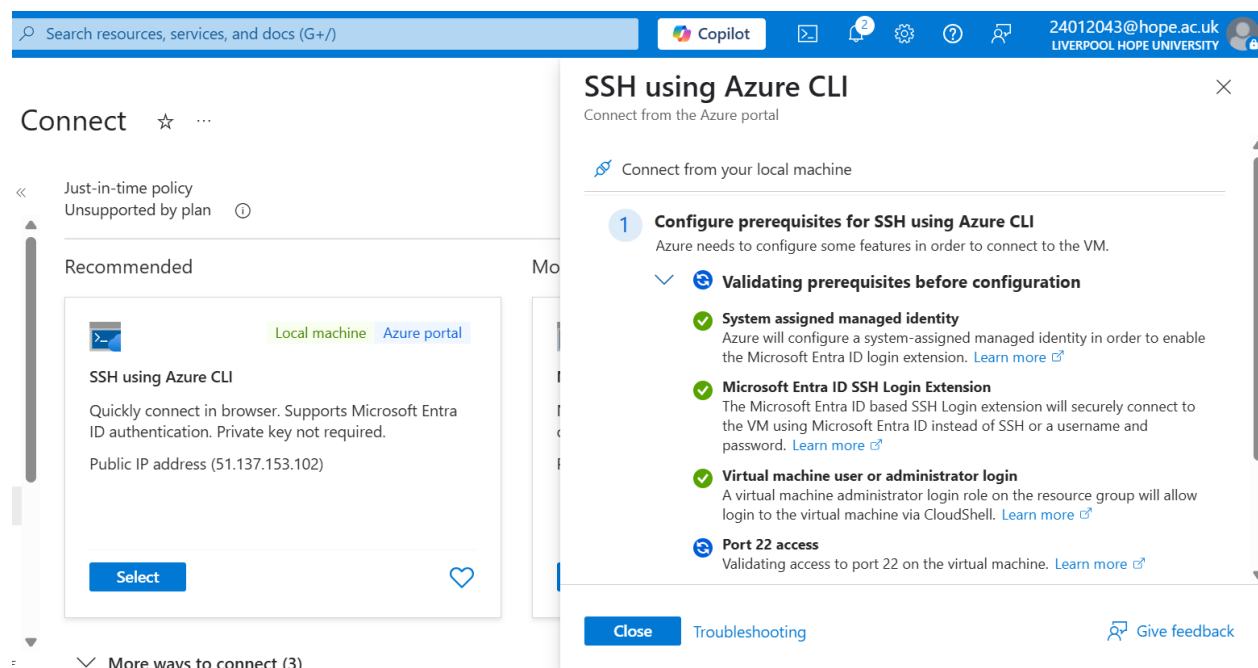


Figure 1-Azure virtual machine

4. Code Explanation

The technical implementation of the patient classification system utilising Hadoop MapReduce in a virtual environment hosted by Azure is covered in this section. The scalable patient data classifier on Hadoop YARN is implemented in Java. The Mapper class, Reducer class, and Driver class are the three primary components of this system, each plays a crucial part in processing and filtering patient data from sizable CSV files according to a certain set of criteria.

Three essential files generate up the Java implementation, and each one serves a distinct function in the MapReduce process:

Mapper Phase: PatientDataMapper.java

In accordance with user-specified filtering logic, this class is in charge of processing each line of the input file and producing key-value pairs. Runtime settings that are given through Hadoop's configuration object govern the logic.

Reducer Phase: PatientDataReducer.java

Each record is written to the output by this class after it receives grouped key-value pairs from the mapper.

PatientDataDriver.java: Configuring and Starting the Job

The MapReduce job is started and configured by the main class. In addition to improving scalability and maintainability, this segmented modular structure facilitates reusable and adaptable logic for upcoming categorisation filters.

4.1. Mapper Phase: PatientDataMapper.java

Each line of a “MedicalFiles” CSV file is read by the mapper, which then filters the records according to a specified filter type and value (such as patient ID, symptom, or area). The header line is initially delivered exactly as is. It then divides each data line into fields, determines if the record satisfies the filter condition, and, if so, outputs the filtered record.

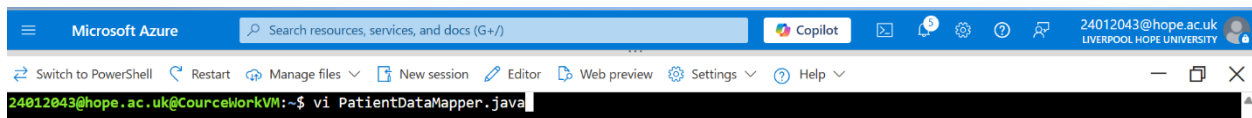


Figure 2- create PatientMapper.java

```

Switch to PowerShell Restart Manage files New session Editor Web preview Settings Help
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

public class PatientDataMapper extends Mapper<LongWritable, Text, Text, Text> {

    private String filterType;
    private String filterValue;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        // Read filter type and filter value from context configuration
        filterType = context.getConfiguration().get("filterType");
        filterValue = context.getConfiguration().get("filterValue");
    }

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        if (key.get() == 0 && line.contains("PatientNHSNumber")) {
            // Skip the header line and emit it for reducers
            context.write(new Text("header"), new Text(line));
            return;
        }

        // Split the line into fields considering quotes and commas
        String[] fields = line.split("(?=[^\"']*\"|^\".*\"|'[^']*'|'.*').*", -1);
        String record = String.join(",", fields);
    }
}

```

Figure 3-PatientDataMapper code I

```

// Emit the record based on the filter criteria
switch (filterType) {
    case "patient_id":
        if (fields[0].equals(filterValue)) {
            context.write(new Text(filterValue), new Text(record));
        }
        break;
    case "symptom":
        if (fields[7].contains(filterValue)) {
            context.write(new Text(filterValue), new Text(record));
        }
        break;
    case "region":
        if (fields[11].equals(filterValue)) {
            context.write(new Text(filterValue), new Text(record));
        }
        break;
}
}
}
-- INSERT -- 51 1 Bot

```

Figure 4-PatientDataMapper code II

4.1.1. Imports and class declaration

```

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

```

Figure 5-Import libraries in DataMapper

- Hadoop's writable data types utilities are provided by org.apache.hadoop.io.*; * denotes all libraries.
- Hadoop.mapreduce.org.apache.For MapReduce tasks, Mapper is the default Mapper class.
- To manage input/output exceptions, import java.io.IOException.

```

public class PatientDataMapper extends Mapper<LongWritable, Text, Text, Text> {

```

Figure 6-PatientDataMapper class

- PatientDataMapper class extends Mapper class and Mapper class generics the LongWritable type as the input key type, and input and output values and value key types as Text.

4.1.2. Variable declaration

```
private String filterType;
private String filterValue;
```

Figure 7-Variables in DataMapper

- “filterType” and “filterValue” variables to store the filter criteria obtained from the job’s configuration. FilterType keeps the type of filter to apply and filtervalue keeps the specific value to filter on.

4.1.3. Setup Method

```
@Override
protected void setup(Context context) throws IOException, InterruptedException {
    // Read filter type and filter value from context configuration
    filterType = context.getConfiguration().get("filterType");
    filterValue = context.getConfiguration().get("filterValue");
}
```

Figure 8-setup method – DataMapper

- setup() runs once before the map method. It retrieves configuration parameters define the filtering criteria, from the Hadoop job context and the Mapper will use for each record.

4.1.4. Map Method

```
@Override
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
    String line = value.toString();
    if (key.get() == 0 && line.contains("PatientNHSNumber")) {
        // Skip the header line and emit it for reducers
        context.write(new Text("header"), new Text(line));
        return;
    }
}
```

Figure 9-Map method DataMapper

- This method reads and processes each line of the input file and converts the input line to a Java String.

key: the byte offset of the line.

value: the line content as a Text object.

- Checks if this is the first line (`key.get() == 0`) and contains the header "PatientNHSNumber".
- If yes, outputs the header with the key "header" and skips further processing of this line.

```
// Split the line into fields considering quotes and commas
String[] fields = line.split("(?=(?:[^\"]*"\"[^\"]*"\"?)*[^\"]*$)", -1);
String record = String.join(",", fields);
```

Figure 10-split CSV

- Creates fields out of the CSV line. Commas inside quotes are ignored because the regex `("(?:\"(?:[^\"]|\\\"|\\\\\\\\)*\"|\"(?:[^\"]|\\\\\\\\)*\")")` is used to split by commas outside quotes.
- Empty fields are continuously trailed by the -1 argument.
- Merges the fields once more to create a CSV string, which is identical to the original line but normalised. If the record matches the filter, this is the one that will be output.

4.1.5. Filtering logic

```
// Emit the record based on the filter criteria
switch (filterType) {
    case "patient_id":
        if (fields[0].equals(filterValue)) {
            context.write(new Text(filterValue), new Text(record));
        }
        break;
    case "symptom":
        if (fields[7].contains(filterValue)) {
            context.write(new Text(filterValue), new Text(record));
        }
        break;
    case "region":
        if (fields[11].equals(filterValue)) {
            context.write(new Text(filterValue), new Text(record));
        }
        break;
}
```

Figure 11-filtering logic

- Several filters feature implements using a switch case on filterType:
 - o Patient_id: Verifies that the filter value precisely matches the first field (index 0).
 - o Symptom: Verifies whether the filter value (substring match) is present in the eighth field (index 7).

- Region: Verifies that the twelfth field (index 11) precisely corresponds to the filter value.
- It outputs the key as the filter value and the value as the CSV record if a record matches the filter.

4.1.6. Overview of PatientDataMapper class

The PatientDataMapper class is an integral component of the Hadoop MapReduce framework, designed to efficiently process and filter large datasets. It extends the Mapper class provided by the Hadoop library, allowing it to process key-value pairs as input and output. The key objective of this class is to read each line of the input CSV file, parse its content, and emit key-value pairs based on user-defined filtering criteria.

The class begins by declaring two instance variables, filterType and filterValue, which store the criteria used to filter records. These values are initialized in the setup method, which is called once at the beginning of the map task. The core functionality of the PatientDataMapper class resides in the map method, which is invoked for each line of the input file.

Based on the filter criteria specified by filterType, the method emits key-value pairs. For example, if the filterType is "patient_id", it checks if the first field (PatientNHSNumber) matches the filterValue and emits the record accordingly. This design not only optimizes the processing of large datasets but also provides flexibility in defining various filtering criteria, making it highly adaptable to different use cases.

4.2. Reducer Phase: PatientDataReducer.java

The 'PatientDataReducer' is a Hadoop Reducer class that handles filtered patient records received from the Mapper. The process involves aggregating records based on identifiers such as 'patient_id', 'symptom', or 'region', and thereafter outputting each record without a key, utilising 'NullWritable' for the key representation.

```
24012043@hope.ac.uk@CourseWorkVM:~$ vi PatientDataReducer.java
```

Figure 12-Create PatientDataReducer

```

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

public class PatientDataReducer extends Reducer<Text, Text, NullWritable, Text> {

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        for (Text value : values) {
            context.write(NullWritable.get(), value);
        }
    }
}

```

Figure 13-patientDataReducer code

4.2.1. Imports

```

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;

```

Figure 14-import libraries to the DataReducer

These import the Hadoop classes and the IOException class.

- org.apache.hadoop.io.* provides access to Hadoop's data types, including Text and NullWritable.
- org.apache.hadoop.mapreduce.Reducer is the base class of reducers.
- IOException handles input/output errors that occur during file operations.

4.2.2. Class declaration

```

public class PatientDataReducer extends Reducer<Text, Text, NullWritable, Text> {

```

Figure 15-DPatientDataReducer class

This defines the Reducer class which extends by the Hadoop Reducer class by defining following generic types.

- Input Key: Text (e.g., filter value such as symptom or region).
- Input Value: Text (patient records).
- Output Key: NullWritable (No key is written).
- Output Value: Text (Write the entire patient record to the output).

4.2.3. Reduce Method

```
@Override
public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    for (Text value : values) {
        context.write(NullWritable.get(), value);
    }
}
```

Figure 16-Reduce method in DataReducer

The `reduce()` method is used once for each unique key in the Mapper output. It receives:

- key: The group key (for example, "Flu" when filtering by symptom).
- Values: All of the records linked with that key.
- Context: Used to write the output.

For each value (that is, a patient record) in the group:

- It transfers the record to the output.
- Uses `NullWritable.get()` to skip the key in the output (just values are written to the file).

4.2.4. Overview of the PatientDataReducer class

The `PatientDataReducer` class is a crucial component of the MapReduce process in the Hadoop framework, ensuring the summarization of the output of the Mapper class. This class extends the `Reducer` class, which aggregates filtered data to shape it and prepare it for the final output. The `PatientDataReducer` class receives intermediate key/value pairs emitted by the Mapper class, pre-processes them, and generates a final set of output values. The class defines the `reduce` method, which is abstract and overridden for every unique key that the mappers emit. The `reduce` method is passed a key of type `Text` and an iterable collection of `Text` values, which are records corresponding to the same key generated by the mappers. The class iterates over the collection of values associated with the key, emitting the value for each value by calling the `context.get()` method. This ensures that all records corresponding to the filter criteria are included in the final output. The overall design of the `PatientDataReducer` class is concise and compelling, reducing computational overhead and ensuring immediate results. The class is flexible enough to handle

most serialized filtered data, and the filtering logic is contained in the PatientDataMapper class, making the overall MapReduce job modular and easy to maintain.

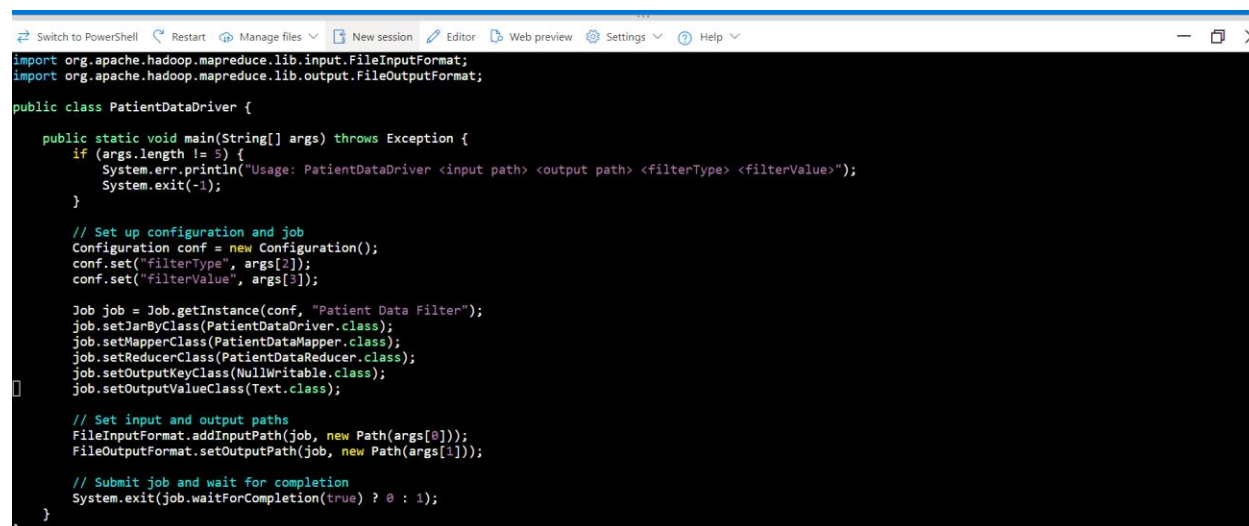
4.3. PatientDataDriver.java: Configuring and Starting the Job

The PatientDataDriver class is the software that runs a Hadoop MapReduce job and filters patient records depending on particular criteria such as patient_id, ailment, or location. It serves as the entry point and controls how the job is done in the cluster. This class

- Reads filter parameters and input/output paths from the command line.
- Configures the MapReduce task with the PatientDataMapper and PatientDataReducer classes.
- Submits the job to Hadoop and awaits completion.

```
24012043@hope.ac.uk@CourseWorkVM:~$ vi PatientDataDriver.java
```

Figure 17-PatientDataDriver create



```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class PatientDataDriver {

    public static void main(String[] args) throws Exception {
        if (args.length != 5) {
            System.err.println("Usage: PatientDataDriver <input path> <output path> <filterType> <filterValue>");
            System.exit(-1);
        }

        // Set up configuration and job
        Configuration conf = new Configuration();
        conf.set("filterType", args[2]);
        conf.set("filterValue", args[3]);

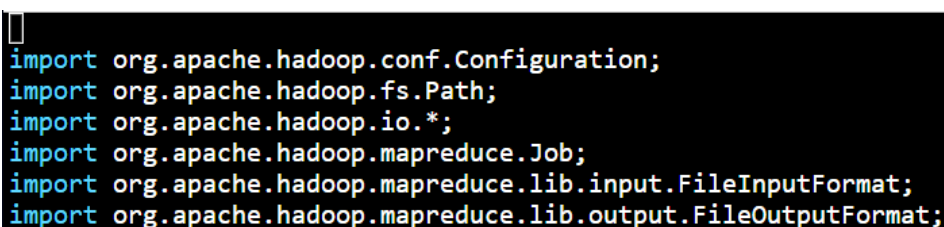
        Job job = Job.getInstance(conf, "Patient Data Filter");
        job.setJarByClass(PatientDataDriver.class);
        job.setMapperClass(PatientDataMapper.class);
        job.setReducerClass(PatientDataReducer.class);
        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Text.class);

        // Set input and output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // Submit job and wait for completion
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Figure 18-PatientDataDriver code

4.3.1. Imports



```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

Figure 19-Import libraries to the DataDriver

- org.apache.hadoop.conf.Configuration: Provides configuration for Hadoop jobs.
- org.apache.hadoop.fs.Path: Hadoop's filesystem path representation.
- org.apache.hadoop.io.*: Provides Hadoop's I/O utilities.
- org.apache.hadoop.mapreduce.Job: Main class for configuring and running a MapReduce job.
- org.apache.hadoop.mapreduce.lib.input.FileInputFormat: Class for specifying the input path for the job.
- org.apache.hadoop.mapreduce.lib.output.FileOutputFormat: Class for specifying the output path for the job.

4.3.2. Class declaration

```
public class PatientDataDriver {
```

Figure 20-DataDriver class

Declares the PatientDataDriver class that contains the main method – the program's entry point.

4.3.3. Main method

The main method starts the application and can throw exceptions that occur during job setup or execution.

```
public static void main(String[] args) throws Exception {
    if (args.length != 5) {
        System.err.println("Usage: PatientDataDriver <input path> <output path> <filterType> <filterValue>");
        System.exit(-1);
    }
}
```

Figure 21-Main method of DataDriver

The argument check ensures that the program is run with exactly four parameters (four also can use as the parameter since only checking 4 arguments). Println is use to handle the error instructions and exits if any parameters are missing.

4.3.4. Job configuration and Job setup

Creates a Hadoop Configuration object and assigns two custom configuration parameters:

- filterType: the type of filtering to perform (for example, patient_id, symptom, region).
- FilterValue: the value to match.

These are later obtained in the Mapper via context.getConfiguration().

```
// Set up configuration and job
Configuration conf = new Configuration();
conf.set("filterType", args[2]);
conf.set("filterValue", args[3]);

Job job = Job.getInstance(conf, "Patient Data Filter");
job.setJarByClass(PatientDataDriver.class);
job.setMapperClass(PatientDataMapper.class);
job.setReducerClass(PatientDataReducer.class);
job.setOutputKeyClass(NullWritable.class);
job.setOutputValueClass(Text.class);
```

Figure 22-job setup

- Job job = Job.getInstance(conf, "Patient Data Filter"): Creates a new job instance with the given configuration.
- job.setJarByClass(PatientDataDriver.class): Sets the jar containing the job's classes.
- job.setMapperClass(PatientDataMapper.class): Sets the mapper class.
- job.setReducerClass(PatientDataReducer.class): Sets the reducer class.
- job.setOutputKeyClass(NullWritable.class): Sets the output key class.
- job.setOutputValueClass(Text.class): Sets the output value class.

4.3.5. Input/output path setup

Sets the input file path (from args[0]) and output directory (from args[1]).

- FileInputFormat.addInputPath(job, new Path(args[0])): Sets the input path.
- FileOutputFormat.setOutputPath(job, new Path(args[1])): Sets the output path.

```
// Set input and output paths
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// Submit job and wait for completion
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Figure 23-Input/Output path setup

Lastly, Submits the job to the Hadoop cluster and waits until it finishes. Returns 0 if the job succeeds, otherwise 1.

4.3.6. Overview of the PatientDataDriver class

The Hadoop MapReduce job is launched from the PatientDataDriver class, which orchestrates the Mapper and Reducer classes for data filtering and aggregation. It configures and begins the job, describes input and output paths, sets up required configurations, and submits the task to the Hadoop YARN framework as the driver class. Essential Hadoop packages are imported at the start of the class.

The main function of the program is to validate input arguments and set up Hadoop configuration. The Mapper class filters records according to user preferences. A new instance of the job is created and passed along with a meaningful job name. The jar file containing the mapper, reducer, and driver classes is set, ensuring they can be found and used by Hadoop while the job is running.

The main way to set up Hadoop involves creating an instance of the Configuration class and setting the filter type and value. The Mapper class sorts information based on user preferences. A new Job instance is created, and the driver, mapper, and reducer classes are set.

When calling the job, the PatientDataDriver class tells the Mapper and Reducer classes to use, as well as the output key and value. FileInputFormat is used to set up the job's input and output files, and the main method calls job to send the job to the Hadoop YARN system. The job is completed and returns a status code, either indicating success or failure.

4.4. Mounting Local File with Azure Virtual Machine

```
24012043@hope.ac.uk@CourseWorkVM:~$ az storage blob upload --csb100320035bc47a86 MedicalFiles.csv
```

Figure 24-MedicalFiles CSV uploading to the Azure bolb storage

```
24012043@hope.ac.uk@CourseWorkVM:~$ az storage blob download csb100320035bc47a86 MedicalFiles.csv
```

Figure 25-Output download from Azure VM blob storage as the CSV file

4.5. Run Hadoop in YARN

```
24012043@hope.ac.uk@CourseWorkVM:~$ hadoop com.sun.tools.javac.Main PatientDataDriver.java
```

Figure 26-Compile PatientDataDriver Class

```
24012043@hope.ac.uk@CourseworkVM:~$ jar cf mc.jar PatientDataDriver*.class
```

Figure 27-create jar

```
24012043@hope.ac.uk@CourseworkVM:~$ hadoop jar PatientDataJob.jar testz.PatientDataDriver
```

Figure 28-Run jar file using hadoop

```
24012043@hope.ac.uk@CourseworkVM:~$ yarn jar PatientDataJob.jar
```

Figure 29-Run jar file using YARN

4.5.1. Application of Hadoop and YARN in the Azure-Based System

To allow scalable and distributed data processing, Apache Hadoop and YARN are deployed on a cloud-based infrastructure powered by Microsoft Azure Virtual Machines (VMs). The system architecture consists of two Linux-based virtual machines (VMs): CourseworkVM, which serves as the principal node for operating the Hadoop ecosystem, and LinuxMachine, which acts as a supporting node in the configuration.

Name	Type	Location	Resource Group	Subscription	Last accessed
CourseWorkVM	Virtual machine	UK West	BigData_HopeUni	Azure for Students	1 minute ago
LinuxMachine	Virtual machine	West Europe	BigData_HopeUni	Azure for Students	27 days ago
CourseWorkVM-ip	Public IP address	UK West	BigData_HopeUni	Azure for Students	27 days ago
BigData_HopeUni	Resource group	West Europe	BigData_HopeUni	Azure for Students	1 month ago

Figure 30-Virtual Machines on Azure

Core Hadoop daemons such as NameNode, ResourceManager, and JobHistoryServer are housed in the CourseworkVM, whereas LinuxMachine manages the DataNode and NodeManager services. Using the Hadoop command-line interface, the PatientDataJob JAR file is executed from CourseworkVM to start the job submission process. The Mapper and Reducer logic for search functionality over the MedicalFile dataset stored in HDFS is specified by the PatientDataDriver class, which is described within the JAR. After a job is submitted, the ResourceManager approves the request and launches a fresh instance of ApplicationMaster to oversee the job's lifecycle. After that, YARN works with the NodeManagers on LinuxMachine and CourseworkVM to start

containers that carry out map and reduce operations concurrently. Mappers create intermediate data, which is then sorted and shuffled before reducers process it to create the final output.

4.6.Output

In this program, allow users to perform classification of available data set in three categories as the Symptom, patient Id and also the region of the patient.

```
Enter the type of data filtering (patient_id, symptom, region):
symptom
Enter the symptom to filter:
Hypertension,
Filtered data has been exported to FilteredRecords22.csv.
```

Figure 31-search for the Hypertension

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	PatientNH	Name	Age	Gender	Date_of_A	Medical_R	Medical_H	Chief_Complaint	History_of	Physical_E	Assessmer	NHS_Trust	Region					
2	8.37E+09	John Smith	45	Male	Apr-15	2024	1.23E+08	Hypertension, Type	Persistent	Intermittent	Blood Pres	1. Monitor	East_of_England					
3	9.7E+09	Michelle G	50	Female	Dec-12	2024	8.77E+08	Hypertension, Dyslip	Fatigue an	Complainii	Blood Pres	1. Screen f	East_of_England					
4	2.95E+09	Steven Wil	70	Male	Mar-05	2025	8.77E+08	Hypertension, Type	Chest pain	Complainii	Blood Pres	1. Adminis	North_East_and_Yorkshire					
5	6.13E+09	Amanda TI	25	Female	Apr-10	2025	5.43E+08	Major Depressive Di	Depressior	Reports pe	Vital signs	1. Initiate	North_West					
6	9.36E+09	Jennifer Ki	40	Female	Oct-20	2025	98765432	Hypothyroidism, De	Fatigue an	Complainii	Thyroid gl	1. Initiate	North_East_and_Yorkshire					
7	7.39E+09	Daniel Sco	55	Male	Nov-15	2025	5.43E+08	Hypertension, Type	Fatigue an	Complainii	Blood Pres	1. Order cc	North_West					
8	5.84E+09	Mark John	42	Male	Jan-15	2025	1.36E+08	Hypertension, Type	Shortness	Experienci	Blood Pres	1. Adminis	North_East_and_Yorkshire					
9	4.26E+09	Rachel Sm	47	Female	May-10	2025	6.74E+08	Hypertension, Osteo	Joint pain	Complainii	Limited rai	1. Prescrib	North_West					
10	7.59E+09	Julia Marti	55	Female	Jul-20	2025	8.95E+08	Hypothyroidism, De	Fatigue an	Complainii	Thyroid gl	1. Initiate	South_West					
11	5.72E+09	Thomas Jc	55	Male	Oct-25	2025	8.75E+08	Hypertension, Type	Fatigue an	Complainii	Blood Pres	1. Order cc	East_of_England					
12	4.86E+09	Natalie Mz	50	Female	Jan-15	2026	6.54E+08	Hypothyroidism, De	Fatigue an	Complainii	Thyroid gl	1. Initiate	North_West					
13	1.57E+09	Hannah Br	38	Female	Apr-10	2026	3.69E+08	Hypertension, Type	Chest pain	Experienci	Blood Pres	1. Adminis	London					
14	7.31E+09	Owen Whi	47	Male	Aug-25	2026	3.21E+08	Hypertension, Osteo	Joint pain	Complainii	Limited rai	1. Prescrib	North_West					
15																		

Figure 32-Generated output file

```
Enter the type of data filtering (patient_id, symptom, region):
patient_id
Enter the patient id to filter:8374926150
Filtered data has been exported to FilteredRecords22.csv
```

Figure 33-Search for the Patient Id:8374926150

PatientNH	Name	Age	Gender	Date_of_A	Medical_R	Medical_H	Chief_Con	History_of	Physical_E	Assessmer	NHS_Trust	Region
8.37E+09	John Smith	45	Male	Apr-15	2024	1.23E+08	Hypertensi	Persistent	Intermitte	Blood Pres	1. Monitor blood pressure close	East_of_England

Figure 34-Generated output file

```

Enter the type of data filtering (patient_id, symptom, region):
region
Enter the symptom to filter:
East_of_England,
Filtered data has been exported to FilteredRecords22.csv.

```

Figure 35-Search for the England

PatientNH	Name	Age	Gender	Date_of_A	Medical_R	Medical_H	Chief_Complaint	History_of	Physical_E	Assessmer	NHS_Trust	Region
8.37E+09	John Smith	45	Male	Apr-15	2024	1.23E+08	Hypertension, Type	Persistent	Intermitte	Blood Pres	1. Monitor	England
9.7E+09	Michelle G	50	Female	Dec-12	2024	8.77E+08	Hypertension, Dyslip	Fatigue an	Complaini	Blood Pres	1. Screen f	East_of_England
7.53E+09	Richard Ba	62	Male	Jul-25	2025	6.54E+08	Chronic Kidney Dise	Fatigue an	Complaini	Blood Pres	1. Order c	East_of_England
6.95E+09	David Clar	65	Male	Jun-15	2025	7.49E+08	Chronic Obstructive	Exacerbat	Presenting	Respirator	1. Adminis	East_of_England
5.72E+09	Thomas Jo	55	Male	Oct-25	2025	8.75E+08	Hypertension, Type	Fatigue an	Complaini	Blood Pres	1. Order c	East_of_England
8.95E+09	Avery Wils	42	Female	Mar-20	2026	2.47E+08	Asthma, Allergic Rhi	Shortness	Presenting	Respirator	1. Adminis	East_of_England

Figure 36-Generated output file

The MedicalFile dataset is successfully processed and filtered by the created Hadoop-YARN-based big data system using a custom MapReduce job that is implemented in Java. The application enables customised data retrieval from a big dataset by letting users apply certain filtering criteria based on Patient ID, Symptoms, and Region.

Only the matched records that meet the specified search parameters are included in the final output, which is produced as an Excel file called FilterRecord22.xlsx. Above screenshots demonstrate the outcome of the filtering process and to confirm that the system operates well in a real-world test environment.

5. Potential Applications of Patient Record Classification using MapReduce

The versatile patient data classifier created using Hadoop YARN could be used in a variety of healthcare contexts. This system can assist with a variety of critical healthcare jobs and projects since it can quickly process massive amounts of patient data and filter outcomes based on various criteria (Blagoj Ristevski, Ming Chen, 2018). Here are a few of the most essential applications for this scalable method. Based on the literature examined and the technical implementation presented in this research, the suggested categorisation system has a range of effective applications across several layers of healthcare and public health management.

Clinical Decision Support Systems (CDSS): use distributed MapReduce processing to classify patient symptoms in real-time, helping physicians uncover patterns and trends. Clustering cases based on common symptoms such as chest discomfort, shortness of breath, or fever, for example, might help to inform triage decisions and optimise care prioritisation in emergency and outpatient settings. Such insights can warn medical teams to impending health issues, increasing responsiveness and allowing for more accurate differential diagnosis.

Regional Health Surveillance and Epidemiological Monitoring: The system uses fields like NHS_Trust_Region to aggregate clinical data across regions. This helps public health professionals undertake spatiotemporal analysis to predict disease outbreaks, track the frequency of chronic illnesses, and uncover healthcare inequities between areas. It also improves pandemic preparedness by allowing authorities to detect symptom clusters that indicate infectious disease spread, leading prompt intervention actions like targeted testing or containment zones.

Strategic Health Policy and Resource Allocation: The system's ability to create organised, aggregated data is crucial for health service planning at both local and national levels. These data can be included into NHS decision-making dashboards to help with resource allocation, such as forecasting ICU bed demand, directing mobile testing units, and monitoring vaccination distribution methods. By recognising regional symptom and case load patterns, healthcare planners can better link policy responses with real-time field data.

Medical Research and Predictive Analytics: The technology can be repurposed by academic institutions, clinical researchers, and health data scientists for secondary medical research and predictive analytics purposes. The structured outputs of symptom-based classification can be used

to train machine learning models for predictive diagnostics, patient risk stratification, and outcome predicting. Furthermore, retrospective examination of anonymised categorised data can aid in longitudinal studies of illness development, treatment efficacy, and health-care utilisation.

Public Health Portals and National Informatics Integration: The system's architecture, based on scalable big data concepts, allows for integration with national systems like NHS Digital. Its outputs can be configured for API-based interaction with existing health information exchanges, automating regional data analytics, daily case tracking, and real-time health monitoring. As health systems continue to digitise, such modular technologies will be critical for developing responsive and intelligent national health data infrastructures.

6. Conclusion

This study successfully constructed a scalable and efficient patient record categorisation system based on the MapReduce programming paradigm in a Hadoop-YARN framework installed on a Microsoft Azure virtual machine. The system showcases the use of distributed computing technologies to manage large-scale structured healthcare datasets. The application uses a traditional MapReduce pipeline, which includes data ingestion, mapping, shuffling, reduction, and structured output, to identify patient symptoms and geographical health care locations. These classifications are eventually produced in structured CSV format for later usage in analytics and policy contexts. The findings of this project substantiate existing literature that emphasises the transformative role of big data technologies in healthcare. Prior studies have shown that frameworks such as Hadoop enable fault-tolerant, parallelised data processing which is particularly advantageous when dealing with the scale, heterogeneity, and complexity of medical datasets (Blagoj Ristevski, Ming Chen, 2018) (Palanisamy, R. and Thirunavukarasu, 2017).

The application also has several practical implications. By using symptom patterns to identify high-priority cases, it can act as a core module for clinical decision support systems. By combining data at the NHS Trust level, it facilitates regional epidemiological surveillance and allows for focused interventions during epidemics or public health emergencies. It also provides structured data outputs that may be incorporated into national health data infrastructure or NHS dashboards for evidence-based decision-making, making it relevant for healthcare resource planning and policy creation. By producing labelled data that can be used to train predictive machine learning models or perform retrospective evaluations of patient care pathways, the system also provides research opportunities.

In summary, this study shows that big data technologies may solve urgent issues in clinical data processing and health informatics when properly designed and implemented on cloud infrastructures. The system presents a prototype that supports the digital transformation initiatives pursued by national health systems like the NHS in addition to validating the operational viability of employing Hadoop-YARN for healthcare classification jobs. Future research could concentrate on integrating real-time data ingestion from hospital information systems, extending the system to support unstructured clinical narratives using natural language processing (NLP), and improving classification outputs using machine learning-driven techniques to improve clinical accuracy and insight generation.

7. References

Blagoj Ristevski, Ming Chen, 2018. Big Data Analytics in Medicine and Healthcare. *Journal of Integrative Bioinformatics*, Volume 15(3), pp. 1-7.

M. S. Roobini and M. Lakshmi, 2019. Application of Big Data for Medical DataAnalysis Using Hadoop Environment. *Springer Nature Switzerland*, Volume 26, p. pp. 1128–1135.

Mohammed Fakherldin¹, Ibrahim Aaker Targio Hashem^{2*}, Abdullah Alzuabi³, Faiz Alotaibi, 2018. Performance Evaluation of Hadoop in Cloud for Big Data. *International Journal of Engineering & Technology*, 7 ((4.15)), pp. 16-18.

Nazari E, Shahriari MH, Tabesh H, 2019. Big Data Analysis in Healthcare: Apache Hadoop, Apache Spark and Apache Flink. *Frontiers in Health Informatics*, 8(1), p. e14.

NHS, 2023. *National Health Service-UK*. [Online] Available at: <https://www.nhs.uk/> [Accessed 12 05 2025].

Palanisamy, R. and Thirunavukarasu, 2017. Implications of Big Data Analytics in developing Healthcare Frameworks – A review. *Journal of King Saud University - Computer and Information Sciences*, Volume 31(4), p. pp.415–425..

Rakesh Raja, Indrajit Mukherjee, and Bikash Kanti Sarkar, 2020. A Systematic Review of Healthcare Big Data. *Scientific Programming*, Volume 2020, p. 15 pages.

8. Appendix

PatientDataMapper Code

```
import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class PatientDataMapper extends Mapper<LongWritable, Text, Text, Text> {

    private String filterType;

    private String filterValue;

    @Override

    protected void setup(Context context) throws IOException, InterruptedException {

        filterType = context.getConfiguration().get("filterType");

        filterValue = context.getConfiguration().get("filterValue");

    }

    @Override

    public void map(LongWritable key, Text value, Context context) throws IOException,

    InterruptedException {

        String line = value.toString();

        if (key.get() == 0 && line.contains("PatientNHSNumber")) {
```

```

context.write(new Text("header"), new Text(line));

return;

}

String[] fields = line.split("(?=(?:[^\"]*"\"[^\"]*"")*\"[^\"]*$)", -1);

String record = String.join(",", fields);

switch (filterType) {

    case "patient_id":

        if (fields[0].equals(filterValue)) {

            context.write(new Text(filterValue), new Text(record));

        }

        break;

    case "symptom":

        if (fields[7].contains(filterValue)) {

            context.write(new Text(filterValue), new Text(record));

        }

        break;

    case "region":

        if (fields[11].equals(filterValue)) {

            context.write(new Text(filterValue), new Text(record));

        }

}

```

```
        break;
    }
}
}
```

PatientDataReducer code

```
import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class PatientDataReducer extends Reducer<Text, Text, NullWritable, Text> {

    @Override

    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
        InterruptedException {

        for (Text value : values) {

            context.write(NullWritable.get(), value);

        }

    }

}
```

PatientDataDriver Code

```
import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.*;
```

```

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class PatientDataDriver {

    public static void main(String[] args) throws Exception {

        if (args.length != 5) {

            System.err.println("Usage: PatientDataDriver <input path> <output path> <filterType>
<filterValue>");

            System.exit(-1);

        }

        Configuration conf = new Configuration();

        conf.set("filterType", args[2]);

        conf.set("filterValue", args[3]);

        Job job = Job.getInstance(conf, "Patient Data Filter");

        job.setJarByClass(PatientDataDriver.class);

        job.setMapperClass(PatientDataMapper.class);

        job.setReducerClass(PatientDataReducer.class);

        job.setOutputKeyClass(NullWritable.class);

        job.setOutputValueClass(Text.class);

```

```
FileInputFormat.addInputPath(job, new Path(args[0]));  
  
FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
System.exit(job.waitForCompletion(true) ? 0 : 1);  
  
}  
  
}
```