Big Data Analytics
Lecture 6: Data Formats and the MapReduce Paradigm

Based on Slides by Dr. Tariq Mahmood

Fall 2025

Abstract

These notes delve into the fundamental concepts of data handling in Hadoop and provide a comprehensive introduction to the MapReduce programming model. We will explore the differences between structured, semi-structured, and unstructured data, and understand why Hadoop's "schema-on-read" approach is advantageous for big data. The core of the notes is a step-by-step analysis of the MapReduce paradigm, using a practical weather data analysis problem to illustrate its power, scalability, and implementation details in Java.

Contents

1 Data Formats and Hadoop's Approach

Big data comes in many forms. Understanding these data structures is key to choosing the right processing tools.

Unstructured Data This data does not have a predefined data model or internal structure. Examples include plain text, images, videos, and server logs. The 'error.log' and 'access.log' files are prime examples of unstructured text data.

Semi-structured Data This data does not conform to the formal structure of a relational database, but it contains tags or other markers to separate semantic elements. It is looser than structured data, and though a schema might exist, it is often used only as a guide. Examples include XML, JSON, and spreadsheets (like the one shown in slide 4).

1.1 Hadoop to the Rescue: Schema-on-Read

Traditional relational database management systems (RDBMS) use a **schema-on-write** approach. This means you must define a strict schema (table structure, data types) *before* you can load any data. This loading phase can be extremely costly and time-consuming, as the data must be parsed, validated, and indexed before it is available for querying.

Hadoop excels with unstructured and semi-structured data because it employs a **schema-on-read** approach.

- You can load data into Hadoop's Distributed File System (HDFS) as is, without any prior validation or structuring. In Hadoop, loading data is often just a fast file copy.
- The structure or schema is applied to the data at processing time (i.e., when you run a MapReduce job or a Hive query).
- This provides immense flexibility and avoids the costly data loading phase of an RDBMS, making it ideal for exploratory analysis and handling diverse data sources.

1.2 The Normalization Problem and Log Files

In database design, **normalization** is the process of organizing columns and tables to minimize data redundancy. For example, instead of writing out a full client hostname in every log entry, a normalized system might store the hostname once in a separate table and use a small integer ID to reference it in the log table.

- Normalization creates problems for Hadoop's design philosophy, which assumes it's possible to perform streaming reads and writes. To reconstruct the original record in a normalized system, you would need to perform lookups in other tables, which is a non-local operation that breaks the streaming paradigm.
- Web server logs are a perfect example of **non-normalized** data. The client hostname and other information are specified in full each time, even if the same client appears many times.
- Because of this self-contained, non-normalized nature, log files of all kinds are particularly well-suited to analysis with Hadoop.

2 The Power and Design of MapReduce

MapReduce is a programming model designed for parallel processing of vast amounts of data. Its power comes from two core design principles: linear scalability and fault tolerance.

2.1 Linear Scalability

MapReduce achieves scalability through partitioning. The input data is partitioned into chunks, and the functional primitives (map and reduce) can work in parallel on these separate partitions. This leads to predictable, linear scaling behavior:

- If you double the size of your input data while keeping your cluster size the same, a MapReduce job will run approximately twice as slowly.
- However, if you also **double the size of your cluster** (i.e., double the number of nodes), the job will run in roughly the **same amount of time as the original**.

This linear scalability is a powerful feature that is not generally true of complex SQL queries in traditional database systems, which can have unpredictable performance scaling.

2.2 Fault Tolerance and the Shared-Nothing Architecture

Coordinating distributed computation is challenging, especially when dealing with partial failures (i.e., some machines in the cluster failing while others continue to run).

- MapReduce **spares the programmer from having to think about failure**. The framework automatically detects failed tasks and reschedules them on healthy machines.
- This is possible because MapReduce employs a **shared-nothing architecture**. This means that tasks are independent and have no dependency on one another. Each task works on its own slice of data and does not communicate with other tasks.
- From the programmer's perspective, the order in which map or reduce tasks are executed doesn't matter, which greatly simplifies the development of distributed applications.

3 Case Study: Mining NCDC Weather Data

To understand MapReduce in detail, we will analyze a large dataset of global weather records from the National Climatic Data Center (NCDC). This data is a good candidate for MapReduce because it is voluminous, semi-structured, and record-oriented.

3.1 The Problem

The data files are organized by date and weather station. The core question we want to answer is: **What was the highest recorded global temperature for each year in the dataset?**

3.2 The Data Format

The NCDC data is in a fixed-width format. Each line represents a single reading from a weather station. Key fields for our analysis include:

- Columns 16-19: Year
- Columns 88-92: Air Temperature (in tenths of degrees Celsius)
- Column 93: Quality code (indicates if the reading is valid)

A value of '+9999' for temperature indicates a missing reading.

3.3 Traditional Single-Machine Approach

Before MapReduce, this problem would typically be solved on a single machine using command-line tools like 'awk'. The following script illustrates this approach:

```
#!/usr/bin/env bash
for year in all/*
do
  # Print the year, followed by a tab
  echo -ne `basename $year .gz`"\t"
  # Unzip the year's file and pipe it to awk
  gunzip -c $year | \
  awk '{
    temp = substr($0, 88, 5) + 0; # Extract temperature
    q = substr($0, 93, 1);       # Extract quality code
```

```
    # If temp is valid and higher than current max, update max
    if (temp != 9999 && q ~ /[01459]/ && temp > max) max = temp
  }
  END { print max }' # At the end of the file, print the max temp
done
```

Listing 1: max_temperature.sh: A single-machine solution

This script works, but it processes one year at a time, sequentially. To speed it up, we need to run parts of it in parallel.

3.4 The Need for a Framework like MapReduce

One could try to parallelize the script by processing different years in different processes. However, this has limitations:

1. **Dividing Work:** Dividing the work into equal-sized pieces isn't always easy (e.g., files for different years have different sizes). A better approach is to split the total input into fixed-size chunks.
2. **Combining Results:** Combining results from independent processes can be delicate. If you use the fixed-size chunk approach, data for a single year will likely be split across several chunks. This means you need an additional processing step to combine the intermediate results for each year.
3. **Single Machine Bottleneck:** Ultimately, you are still limited by the processing capacity of a single machine. If the best time you can achieve with all your processors is 20 minutes, you can't make it go any faster.

This is where MapReduce comes in. It provides a formal framework for splitting the input, processing the chunks in parallel across a distributed cluster, and handling the combination of results.

4 The MapReduce Model in Detail

MapReduce breaks processing into two distinct phases: the **map phase** and the **reduce phase**. The core data structure used for input and output in both phases is the **key-value pair**.

4.1 The Map Phase: Data Preparation and Filtering

The map phase takes raw data as input and prepares it for the reducer.

- **Input:** By default, the input to the map function is a key-value pair where the key is the line offset within the file (which we usually ignore) and the value is the line of text itself.
- **Function:** The map function's job is to extract the relevant information and emit a new key-value pair. For our weather example, we pull out the year and the air temperature. The year becomes the **output key**, and the temperature becomes the **output value**.
- **Filtering:** The map function is also a good place to drop bad records. We filter out temperatures that are missing (‘9999‘), suspect, or erroneous based on the quality code.

Example Map Transformation:

```
-- Input to Mapper --
(0,   0067...1950...N9+00001...)
(106, 0043...1950...N9+00221...)
(212, 0043...1950...N9-00111...)
(318, 0043...1949...N9+01111...)

-- Output from Mapper --
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
```

4.2 The Shuffle and Sort Phase (Framework Magic)

Between the map and reduce phases, the MapReduce framework performs a critical step known as the "shuffle and sort."

- The framework collects all the key-value pairs emitted by all the map tasks.
- It then **sorts and groups** these pairs by key.
- This means all values associated with the same key are brought together in a single list.

4.3 The Reduce Phase: Aggregation

The reduce phase performs the actual aggregation or summary.

- **Input:** The reduce function receives the output from the shuffle and sort phase. Its input is a key and a list of all the values associated with that key.
- **Function:** The reducer's job is to iterate through the list of values and produce a single output value (or multiple values). For our example, the reducer iterates through the list of temperatures for a given year and finds the maximum value.
- **Output:** The final output is a set of key-value pairs representing the result of the aggregation.

Example Reduce Transformation:

```
-- Input to Reducer (after shuffle & sort) --
(1949, [111, 78])
(1950, [0, 22, -11])

-- Output from Reducer --
(1949, 111)
(1950, 22)
```

5 Implementing a MapReduce Job in Java

Let's look at the Java code to implement the maximum temperature job.

5.1 The Mapper

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

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);

    int airTemperature;
    if (line.charAt(87) == '+') { // Handle leading '+' sign
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }

    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
```

```
  }
}
```

Listing 2: MaxTemperatureMapper.java

Key Points: The class extends ‘Mapper‘ with generic types defining the input key, input value, output key, and output value types. The ‘map‘ method contains the logic to parse the line and ‘context.write()‘ emits the ‘(year, temp)‘ pair.

5.2 The Reducer

```
public class MaxTemperatureReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

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

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

Listing 3: MaxTemperatureReducer.java

Key Points: The class extends ‘Reducer‘. The input types (‘Text‘, ‘IntWritable‘) must match the output types of the mapper. The ‘reduce‘ method receives a key (‘year‘) and an ‘Iterable‘ of all associated temperatures, finds the maximum, and writes the final ‘(year, $\max_t emp$)‘*pair*.

5.3 The Driver Program

The driver is the main program that configures and launches the MapReduce job.

```
public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input> <output>");
      System.exit(-1);
    }

    Job job = new Job();
    // Identifies the JAR file containing the job code.
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

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

    // Specify the Mapper and Reducer classes
    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    // Specify the final output types for the job
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

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

```
28   }
29 }
```

Listing 4: MaxTemperature Driver Program

5.3.1 Driver Configuration Details

- **Job Object:** A ‘Job‘ object forms the complete specification of the work and gives control over how it is run.
- **JAR File:** The ‘setJarByClass()‘ method tells Hadoop where to find the job’s code. Hadoop packages the code into a JAR file and distributes it around the cluster.
- **Input/Output Paths:** ‘FileInputFormat.addInputPath()‘ specifies the input, which can be a single file, a directory (all files in it will be processed), or a file pattern. It can be called multiple times for multiple input sources. ‘FileOutputFormat.setOutputPath()‘ specifies the output directory. **Important:** This directory must not exist before running the job; this is a safety precaution to prevent accidental data loss from overwriting the results of a previous long-running job.
- **Mapper/Reducer Types:** ‘setMapperClass()‘ and ‘setReducerClass()‘ specify the classes to use.
- **Output Types:** ‘setOutputKeyClass()‘ and ‘setOutputValueClass()‘ control the final output types of the *reduce function*. These must match what the Reducer class produces. The map output types default to these same types. If a mapper produces different types than the reducer (e.g., ‘(Text, IntWritable)‘ from map and ‘(Text, DoubleWritable)‘ from reduce), you must explicitly set the map output types using ‘setMapOutputKeyClass()‘ and ‘setMapOutputValueClass()‘.
- **Job Submission:** ‘job.waitForCompletion(true)‘ submits the job and blocks until it finishes. The boolean argument controls verbose output to the console. The return value indicates success or failure.

References

[1] Apache Hadoop Documentation. https://hadoop.apache.org/

[2] Dean, Jeffrey, and Sanjay Ghemawat. ”MapReduce: simplified data processing on large clusters.” *Communications of the ACM* 51, no. 1 (2008): 107-113.