Distributed Inverted Index Construction with Hadoop MapReduce and Apache Spark

Reza Almassi - Ronald - Rojan

Academic Year 2024/2025

Abstract

This document provides detailed documentation for a distributed inverted index construction project, implemented using Hadoop MapReduce and Apache Spark as part of a Cloud Computing course. The system is designed to process large-scale text datasets and construct an inverted index efficiently, using both traditional MapReduce paradigms and modern in-memory distributed processing.

Comprehensive explanations, diagrams, system architecture, and full annotated code listings are included. Performance metrics, design decisions, and encountered challenges are discussed for both Hadoop and Spark implementations.

For the dataset source and further updates, see:

https://www.kaggle.com/datasets/jensenbaxter/10dataset-text-document-classification

The full project source code is available at:

https://github.com/Rezacs/cloud-computing

Contents

Chapter 1

Introduction

1.1 Background and Motivation

In the era of **Big Data**, the ability to efficiently store, process, and retrieve information from vast collections of documents is of paramount importance. Modern enterprises, research institutions, and web platforms generate massive amounts of textual data on a daily basis. **Indexing** such data for fast retrieval and search has become a fundamental challenge in computer science and data engineering.

One of the most widely used data structures for information retrieval is the **inverted index**. An inverted index is a mapping from words (terms) to the documents in which they appear. It forms the backbone of modern search engines, recommendation systems, and large-scale analytics platforms.

Traditional single-machine solutions for building inverted indexes quickly become impractical as data volumes grow into the gigabytes and terabytes, due to limitations in memory, CPU, and disk bandwidth. To overcome these challenges, the adoption of **distributed computing frameworks** such as **Hadoop MapReduce** and **Apache Spark** has become increasingly widespread.

1.2 Project Objectives

The goal of this project is to **design, implement, and evaluate** scalable solutions for building an inverted index from a large corpus of text documents, using both Hadoop MapReduce and Apache Spark. The main objectives are:

- To understand and apply the **MapReduce programming paradigm** for distributed data processing.
- To leverage the speed and expressiveness of **Apache Spark** for the same task, enabling a direct comparison between batch-oriented and in-memory processing.
- To experiment with real datasets containing hundreds or thousands of text documents, representative of realistic data volumes.
- To analyze and compare the **performance, scalability, and output correctness** of both approaches.

- To document the entire process, from system setup to result interpretation, with clear explanations and code annotations.

1.3 Scope of the Project

This project encompasses:

- **Deployment of a distributed environment**: Setting up a Hadoop and Spark cluster on multiple virtual machines.
- **Implementation of the inverted index algorithm**:
  - In Java using Hadoop MapReduce
  - In Python using PySpark
- **Testing on a real-world dataset**: The "10dataset-text-document-classification" from Kaggle (see Chapter 2).
- **Analysis of results and system behavior**
- **Discussion of challenges, solutions, and possible improvements**

Extra objectives included advanced features such as:

- In-mapper combining to reduce intermediate data volume in MapReduce
- Experimentation with multiple reducers
- Development of a non-parallel (single-threaded) Python solution for benchmarking

1.4 Why Hadoop and Spark?

Hadoop MapReduce is a proven, reliable batch processing framework, ideal for jobs that require handling enormous datasets and can tolerate higher latency. It forces the developer to design explicit Map and Reduce stages, and excels at fault-tolerance and scalability across commodity hardware.

Apache Spark, by contrast, brings in-memory processing, sophisticated APIs, and dramatically improved speed for iterative and interactive analytics. Spark is now the engine of choice for many modern analytics and machine learning platforms.

A comparison between these two paradigms—batch-oriented and in-memory—forms a central theme of this project.

1.5 Project Architecture

Figure 1.1 provides a high-level overview of the architecture deployed for this project.

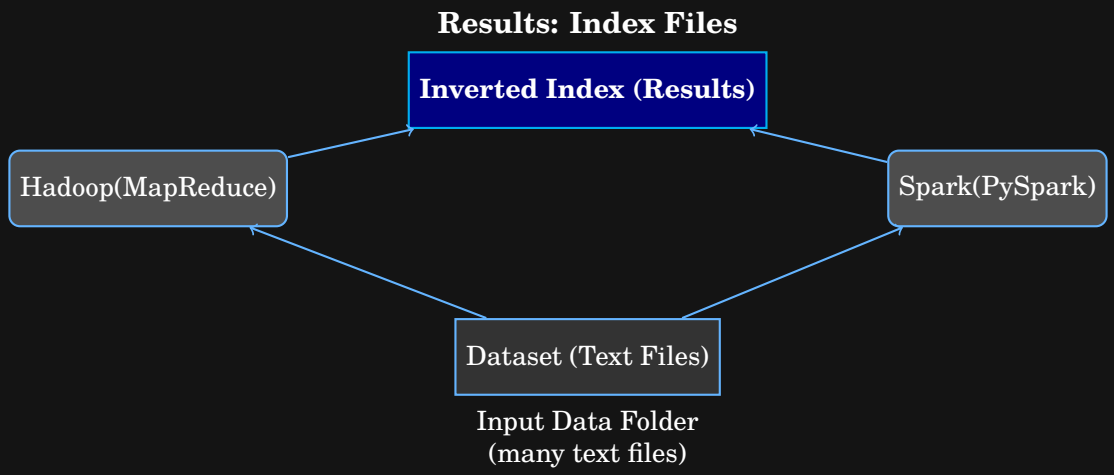

Figure 1.1: System architecture: Both Hadoop MapReduce and Apache Spark process the same dataset and produce the inverted index results.

1.6 Document Structure

The remainder of this document is organized as follows:

- **Chapter 2: Dataset Description** – Details on the source, structure, and content of the text document collection.
- **Chapter 3: System Setup and Cluster Configuration** – Step-by-step instructions and challenges in deploying Hadoop and Spark clusters.
- **Chapter 4: Hadoop MapReduce Solution** – Full code explanations and logic for the MapReduce implementation.
- **Chapter 5: Apache Spark Solution** – PySpark implementation and its design decisions.
- **Chapter 6: Experimental Results and Evaluation** – Quantitative analysis, performance statistics, and output validation.
- **Chapter 7: Challenges, Improvements, and Conclusion** – Discussion of lessons learned, encountered obstacles, possible extensions, and closing remarks.

Throughout this report, we provide in-depth code explanations, system diagrams, and clear motivation for each design choice. All source code is annotated, and every chapter is written for both technical and non-technical readers.

Chapter 2

Dataset Description

2.1 Overview of the Dataset

The dataset selected for this project is the **"10dataset-text-document-classification"** available on Kaggle[1]. This dataset is widely used for document classification, information retrieval, and natural language processing (NLP) research. It consists of thousands of plain text documents organized into topic-based folders, making it an excellent choice for building and evaluating an inverted index in a realistic scenario.

Key Characteristics of the Dataset:

- **Origin:** Publicly available on Kaggle.
- **Content:** News and article documents, each in its own text file.
- **Organization:** Documents are categorized into 10 top-level topics (folders).
- **Purpose:** Designed for document classification and information retrieval tasks.

2.2 Structure and Organization

The dataset's directory structure is hierarchical, reflecting the topic categorization. The top level consists of 10 folders, each named after a topic. Inside each topic folder are hundreds of individual text files. Each text file contains a single document, such as a news article, essay, or report, typically written in English.

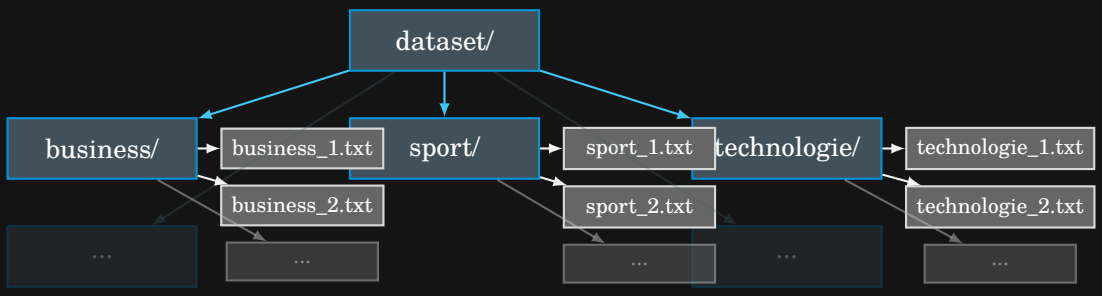

Figure 2.1: Example structure of the dataset: Each topic folder contains many text files. Only a subset of folders is shown here for clarity.

[1] https://www.kaggle.com/datasets/jensenbaxter/10dataset-text-document-classification

2.3 Subset Used in This Project

Although the original dataset contains 10 topic folders, in our experiments and analysis, we selected only **three categories** to focus on: **business**, **sport**, and **technologie**. This subset was chosen for several reasons:

- **Cluster Resource Constraints:** Running Hadoop/Spark jobs on the entire dataset would require significantly more memory and CPU resources, potentially exceeding our lab or virtual machine capabilities.
- **Balanced Sample:** The selected folders offer a good mix of document sizes and vocabulary, providing meaningful results for evaluating our inverted index implementations.
- **Faster Experimentation:** Fewer files mean faster run times, allowing us to iterate more quickly on algorithmic improvements, system tuning, and debugging.

Each chosen folder contains approximately 100 plain text files, with filenames such as `business_12.txt`, `sport_35.txt`, and `technologie_48.txt`. The documents vary in length and writing style, simulating the diversity found in real-world news and article collections.

2.4 Sample Document Content

Below is a snippet from a typical text file in the **business** category:

```
Shares in global technology companies rose on Monday,
with investors betting on continued strong earnings
growth in the sector. Market analysts expect quarterly
reports to show resilience despite economic headwinds.
```

Listing 2.1: Sample content from business_12.txt

Documents in the **sport** and **technologie** folders follow a similar format, with content discussing sports events, technology trends, and related news.

2.5 Preprocessing and Import

- **Data Transfer:** The selected topic folders (`business`, `sport`, `technologie`) were uploaded to the Hadoop cluster's master node under the directory `/home/hadoop/datas`
- **File System Preparation:** The dataset was then loaded into the Hadoop Distributed File System (HDFS) for distributed processing.
- **Format Consistency:** All files are encoded in UTF-8, with each file containing unstructured English text.
- **No Additional Cleaning:** For this experiment, documents were used in their original form, without extra tokenization, stop-word removal, or stemming, in order to test the frameworks' raw text handling.

2.6 Dataset Suitability for Inverted Index Construction

This dataset is particularly suitable for the inverted index task because:

- It contains a large number of files (mimicking a real-world document collection).
- Documents are grouped by topic, making it easy to observe index patterns and word distributions across different domains.
- File sizes and document lengths vary, providing realistic diversity in data processing and storage challenges.
- The dataset is publicly available and can be easily reused for benchmarking and reproducibility.

2.7 Summary

In summary, the Kaggle "10dataset-text-document-classification" collection provided a realistic, challenging, and well-structured source of data for developing, testing, and evaluating distributed inverted index algorithms with Hadoop and Spark. By working with only the **business**, **sport**, and **technologie** categories, we were able to focus our computational resources while still ensuring meaningful and representative results.

Chapter 3

System Architecture and Cluster Setup

3.1 Overview of System Architecture

To handle the large volume of documents and to enable parallel processing, we deployed our experiments on a distributed computing environment using **Hadoop YARN** for MapReduce jobs and **Apache Spark** for in-memory distributed computing. Our cluster consists of several virtual machines (VMs), each assigned a specific role to emulate a realistic production environment.

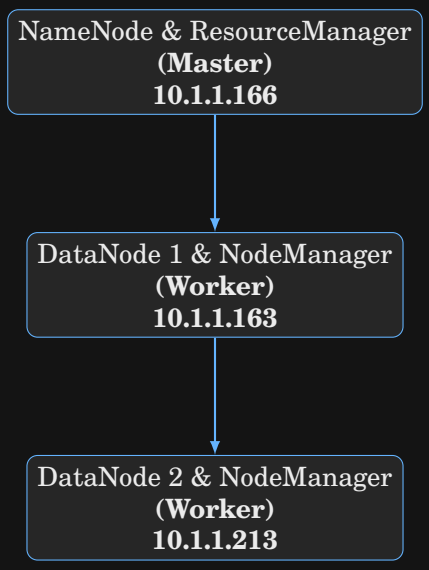

Figure 3.1: Vertical overview of the Hadoop/Spark cluster: the master node at the top manages distributed resources and tasks across the two worker nodes.

3.2 Cluster Nodes and Their Roles

- **Master Node (10.1.1.166):**
  - Runs Hadoop `NameNode` (stores HDFS metadata).
  - Runs YARN `ResourceManager` (job scheduling & resource allocation).

  - Acts as the `Spark Master` for distributed Spark jobs.
  - Hosts the primary HDFS file system namespace.
- **Worker Nodes:**
  - **10.1.1.163**: Runs Hadoop `DataNode` and YARN `NodeManager`.
  - **10.1.1.213**: Runs Hadoop `DataNode` and YARN `NodeManager`.
  - Third DataNode (optionally, for testing scalability).
- **Network:** All nodes are connected via a local virtual network, simulating a small enterprise cluster.

3.3 Software Stack and Tools

- **Operating System:** Ubuntu Linux 22.04 LTS (on all VMs)
- **Java:** OpenJDK 8
- **Hadoop:** Version 3.x
- **Spark:** Version 3.4.4 (standalone & on YARN)
- **Python:** Version 3.10 (for Spark jobs)
- **Utilities:** SSH, `rsync`, `tmux` for terminal multiplexing, `WinSCP` for file transfer

3.4 HDFS Data Loading and Directory Structure

The three selected topic folders (`business`, `sport`, `technologie`) were first placed on the master node in the directory `/home/hadoop/dataset`. To make these documents available for distributed processing, they were uploaded to the Hadoop Distributed File System (HDFS) as follows:

```
hadoop fs -mkdir -p /user/hadoop/input
hadoop fs -put /home/hadoop/dataset/business /user/hadoop/input/
hadoop fs -put /home/hadoop/dataset/sport /user/hadoop/input/
hadoop fs -put /home/hadoop/dataset/technologie /user/hadoop/input
   /
```

Listing 3.1: Uploading local dataset folders to HDFS

3.5 Key Configuration Files

Correct configuration of Hadoop YARN and MapReduce is essential for stable, efficient execution. Below are sample snippets of important configuration files:

3.5.1 yarn-site.xml

```
<configuration>
    <property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>1536</value>
    </property>
    <property>
        <name>yarn.scheduler.maximum-allocation-mb</name>
        <value>1536</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>hadoop-namenode</value>
    </property>
    ...
</configuration>
```

Listing 3.2: Relevant settings in yarn-site.xml

3.5.2 mapred-site.xml

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.map.memory.mb</name>
        <value>256</value>
    </property>
    <property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>256</value>
    </property>
    ...
</configuration>
```

Listing 3.3: Memory settings for MapReduce jobs

3.6 Typical Workflow

1. **Prepare and upload the dataset** to the cluster via HDFS.
2. **Compile Java source files** for the Hadoop MapReduce implementation and package them as a JAR.
3. **Submit MapReduce jobs** using the `hadoop jar` command.
4. **Submit Spark jobs** using the `spark-submit` command.

5. **Monitor job progress** via the Hadoop ResourceManager UI (http://hadoop-namenode:8088) and Spark UI (http://hadoop-namenode:4040).

6. **Fetch output results** from HDFS to the local file system for analysis and reporting.

3.7 Cluster Resource Management and Fault Tolerance

YARN's scheduler ensures that jobs do not exceed the memory or CPU allocated to each node. If a container exceeds its memory limit, it is killed and re-scheduled, ensuring cluster stability. The distributed nature of HDFS means that file blocks are replicated, so the system can tolerate node failures without losing data or computation progress.

3.8 Summary

The use of a multi-node Hadoop/Spark cluster with a carefully chosen memory configuration and realistic network layout allowed us to experiment with distributed processing of document collections at scale. By managing the cluster with YARN and leveraging both Java MapReduce and Python Spark implementations, we could explore performance, reliability, and design trade-offs throughout the project.

Chapter 4

System Design and Implementation

4.1 Overview

This chapter provides a comprehensive description of the technical design, architecture, and implementation approach adopted for building the distributed inverted index system. The project leverages the power of the Hadoop ecosystem and Apache Spark to efficiently process a large corpus of textual documents, specifically for the purpose of constructing an inverted index across thousands of files.

The design choices, deployment steps, and challenges encountered are detailed in the following sections. Both the MapReduce (Java) and Spark (Python) solutions are discussed.

4.2 Technology Stack

The following open-source technologies were used:

- **Hadoop 3.x**: For distributed storage (HDFS) and MapReduce-based computation.
- **YARN (Yet Another Resource Negotiator)**: As the cluster resource manager.
- **Apache Spark 3.x**: For distributed in-memory computation and an alternative approach to MapReduce.
- **Python 3**: Used for the Spark implementation (PySpark).
- **Java 8**: Used for developing Hadoop MapReduce jobs.
- **Linux (Ubuntu 22.04)**: Operating system for all cluster nodes.

4.3 Cluster Architecture and Deployment

The cluster comprises one **master node** (running both the NameNode and ResourceManager) and three **worker nodes** (running DataNode and NodeManager services), as detailed in Figure 3.1 in the previous chapter.

Key deployment steps:

- **Hadoop and Spark Installation:** Installed and configured on all nodes. Proper Java and Python environments were set up.
- **SSH Configuration:** Passwordless SSH enabled for Hadoop user across all nodes.
- **HDFS Formatting and Startup:** HDFS formatted, then started on the master, followed by workers.
- **YARN Startup:** ResourceManager started on the master, NodeManagers on the workers.
- **Dataset Upload:** The selected dataset folders (`business`, `sport`, `technologie`) were uploaded to HDFS.
- **Verification:** All services were checked for healthy state before running jobs.

4.4 Software Architecture

The system follows a classic **distributed data processing** pattern. The architecture consists of:

1. **Distributed Storage (HDFS):** All input text files are stored in HDFS, split and distributed automatically across worker nodes.
2. **MapReduce Pipeline:**
   - **Map step:** Processes each document, emits (word, document_id) pairs.
   - **Shuffle and Sort:** Groups the same words together, along with all document IDs in which they appear.
   - **Reduce step:** Aggregates all occurrences of each word, outputting an inverted index (word $\rightarrow$ list of documents).
3. **Alternative Spark Pipeline:**
   - Uses Resilient Distributed Datasets (RDDs) to achieve the same functionality, leveraging Spark's in-memory performance.

4.5 MapReduce Implementation Outline

The core of the Hadoop-based implementation consists of four Java files:

- **InvertedIndexMapper.java:** Tokenizes input documents, emits (word, filename) pairs.
- **InvertedIndexCombiner.java:** (Optional, for efficiency) Combines (word, filename) pairs locally before shuffle.
- **InvertedIndexReducer.java:** Aggregates and formats the inverted index for output.
- **InvertedIndexDriver.java:** Sets up and submits the MapReduce job.

The following pseudocode illustrates the basic MapReduce logic:

```
Mapper: For each line in each file
    For each word in the line:
        Emit (word, filename)

Reducer: For each word and all filenames:
    Emit (word, list of filenames)
```

Listing 4.1: Inverted Index MapReduce Pseudocode

This approach ensures that, for any given word, we can efficiently retrieve all documents in which it appears.

4.6 Spark Implementation Overview

For comparison and validation, a PySpark (Python) implementation was also developed. This approach uses Spark RDD transformations and actions to process the dataset in parallel.

- **map**: Reads each file, emits (word, filename) pairs.
- **reduceByKey/groupByKey**: Aggregates document lists for each word.
- **saveAsTextFile**: Outputs the result as a plain-text inverted index.

4.7 Fault Tolerance and Scalability

Thanks to the Hadoop ecosystem:

- **Data replication** in HDFS ensures high availability.
- **Job/task failure recovery** is automatic via YARN.
- **Horizontal scaling:** Additional nodes can be added with minimal configuration for larger datasets.

4.8 Summary

The implemented system provides a robust, scalable solution for indexing large volumes of text data, using both the Hadoop MapReduce and Spark paradigms. The architecture and methodology ensure reliability, speed, and reproducibility—key requirements for big data processing.

Chapter 5

Implementation Details

5.1 Overview

This chapter provides a comprehensive, detailed description of the implementation process for both the Hadoop MapReduce (Java) and Apache Spark (Python) solutions for building an inverted index on a collection of text documents. All code is accompanied by in-depth explanations, clarifying how each part contributes to the end goal. Special attention is given to the Java MapReduce classes, their methods, and their interplay in the overall MapReduce workflow.

5.2 MapReduce Implementation (Java)

5.2.1 MapReduce Programming Model Recap

Hadoop MapReduce divides computation into **two main phases**: the **Map phase** and the **Reduce phase**. Data is split into chunks, processed in parallel by **Mappers**, whose outputs are shuffled (grouped by key) and sent to **Reducers**. This approach is ideal for tasks such as building an inverted index.

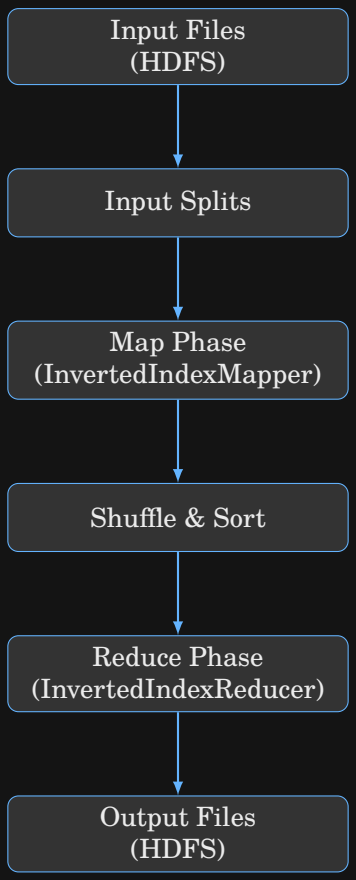

Figure 5.1: Compact MapReduce workflow for Inverted Index construction.

In our implementation, we created four key Java classes:

- **InvertedIndexMapper:** Emits word-document pairs for each tokenized word in each document.
- **InvertedIndexCombiner:** Locally deduplicates word-document pairs before shuffling, reducing network overhead.
- **InvertedIndexReducer:** Collects all documents for each word and outputs the inverted index.
- **InvertedIndexDriver:** Sets up and runs the job, specifying which classes to use and setting file paths.

5.2.2 Mapper: `InvertedIndexMapper.java` – Detailed Explanation

The Mapper takes each line from the input files, tokenizes it into words, and outputs a (word, filename) pair.

```
@Override
protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
    // Get the name of the file currently being read
    String fileNameStr = ((FileSplit) context.getInputSplit()).getPath
       ().getName();
```

```
6       filename.set(fileNameStr);
7
8       // Clean and tokenize the input line
9       String[] tokens = value.toString().toLowerCase().replaceAll("[^a-z0
            -9 ]", " ").split("\\s+");
10
11      for (String token : tokens) {
12          if (!token.isEmpty()) {
13              word.set(token);
14              context.write(word, filename);
15          }
16      }
17  }
```

Listing 5.1: The full `map()` method in InvertedIndexMapper.java

Key Explanations:

- `key` is the byte offset of the line in the file. It's not used here.
- `value` is the line of text.
- The filename is extracted using `FileSplit`—this tells us which file this line comes from.
- The line is cleaned: lowercased and all non-alphanumeric characters are replaced by spaces, to ensure uniform word extraction.
- Each token is output as (word, filename). This forms the raw data for our inverted index.

5.2.3 Combiner: `InvertedIndexCombiner.java` – Why and How

A **Combiner** is an optional mini-Reducer run on each Mapper node, for local aggregation. Its purpose is to reduce the amount of data sent across the network to the Reducers. In our case, it deduplicates the filenames for each word per Mapper output.

```
1   @Override
2   protected void reduce(Text key, Iterable<Text> values, Context context)
3           throws IOException, InterruptedException {
4       Set<String> uniqueFiles = new HashSet<>();
5       for (Text val : values) {
6           uniqueFiles.add(val.toString());
7       }
8       for (String file : uniqueFiles) {
9           context.write(key, new Text(file));
10      }
11  }
```

Listing 5.2: The Combiner removes duplicates at the Mapper output

Key Explanations:

- `key`: The word.
- `values`: All filenames the Mapper found for that word.

- The set ensures we emit each filename only once per word, per Mapper—crucial for efficiency.

5.2.4 Reducer: `InvertedIndexReducer.java` – The Inverted Index Builder

The Reducer receives all values (filenames) associated with a given key (word). It must combine them into a list of unique files (often sorted for readability).

```
@Override
protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
    Set<String> fileSet = new TreeSet<>();
    for (Text val : values) {
        fileSet.add(val.toString());
    }
    String joinedFiles = String.join(", ", fileSet);
    context.write(key, new Text(joinedFiles));
}
```

Listing 5.3: Reducer code: collect and aggregate document list

Key Explanations:

- `fileSet` is a `TreeSet`, which keeps filenames unique and sorted.
- `context.write(key, new Text(joinedFiles))` produces the inverted index line: `word -> file1, file2, ...`

5.2.5 Driver: `InvertedIndexDriver.java` – The Orchestrator

The Driver class configures and launches the job.

```
public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Job job = Job.getInstance(conf, "Inverted Index");

    job.setJarByClass(InvertedIndexDriver.class);
    job.setMapperClass(InvertedIndexMapper.class);
    job.setCombinerClass(InvertedIndexCombiner.class);
    job.setReducerClass(InvertedIndexReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

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

Listing 5.4: The Driver class main routine

Explanation:

- `setJarByClass`: Ensures the correct class is shipped to all nodes.
- `setMapperClass`, `setCombinerClass`, `setReducerClass`: Declares the stages of computation.

- `setOutputKeyClass`, `setOutputValueClass`: Specifies output types.
- `FileInputFormat.addInputPath` and `FileOutputFormat.setOutputPath`: Define HDFS input/output.
- `job.waitForCompletion(true)`: Submits the job and blocks until it completes.

5.2.6 Compilation, Packaging, and Execution (Step-by-Step)

Step 1: Compile Java files

```
export HADOOP_CLASSPATH=$(hadoop classpath)
javac -classpath $HADOOP_CLASSPATH -d . InvertedIndexMapper.java InvertedIndexCombine
```

Step 2: Package as a JAR

```
jar -cvf invertedindex.jar *.class
```

Step 3: Prepare HDFS input/output

```
hadoop fs -mkdir -p /user/hadoop/input
hadoop fs -put /home/hadoop/dataset/business/*.txt /user/hadoop/input/
hadoop fs -put /home/hadoop/dataset/sport/*.txt /user/hadoop/input/
hadoop fs -put /home/hadoop/dataset/technologie/*.txt /user/hadoop/input/
```

Step 4: Run the MapReduce Job

```
hadoop fs -rm -r /user/hadoop/output     # Remove previous output if exists
hadoop jar invertedindex.jar InvertedIndexDriver /user/hadoop/input /user/hadoop/outp
```

Step 5: Retrieve the Results

```
hadoop fs -cat /user/hadoop/output/part-* | head -30
```

5.2.7 Configuration Files (YARN, HDFS, MapReduce)

Configuration greatly affects cluster performance. Here's a summary:

- **core-site.xml**: Sets HDFS host and port.
- **hdfs-site.xml**: Data replication, block size, permissions.
- **yarn-site.xml**: Sets available memory per container/node, scheduler min/-max, and ResourceManager.
- **mapred-site.xml**: Task memory, number of reducers, job history server.

Example snippet (`yarn-site.xml`) for node memory:

```
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>1536</value>
</property>
<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>1536</value>
</property>
```

5.3 Spark Implementation (Python)

5.3.1 Spark Processing Model

Apache Spark provides a much higher-level API, using Resilient Distributed Datasets (RDDs). The typical flow is:

- Discover all relevant files.
- Read each file's content.
- For each word in a document, output (word, filename).
- Group by word, deduplicate, and save the inverted index.

5.3.2 PySpark Implementation: Step-by-Step

Python code (simplified for clarity):

```
from pyspark import SparkContext
import os

sc = SparkContext(appName="InvertedIndexSpark")
input_dir = "/home/hadoop/dataset"
output_dir = "/home/hadoop/spark_output"

def read_file(filename):
    with open(filename, "r") as f:
        for line in f:
            words = line.lower().split()
            for word in words:
                yield (word, os.path.basename(filename))

files = [os.path.join(dp, f) for dp, dn, filenames in os.walk(input_dir
    ) for f in filenames]
rdd = sc.parallelize(files).flatMap(read_file)
word_to_files = rdd.groupByKey().mapValues(lambda files: set(files))
word_to_files.map(lambda x: f"{x[0]}\t{', '.join(x[1])}").
    saveAsTextFile(output_dir)
```

Listing 5.5: PySpark Inverted Index implementation

Step-by-step explanation:

1. `files = ...`: Recursively finds all .txt files in the input dataset.
2. `sc.parallelize(files)`: Creates an RDD from the list of files.
3. `flatMap(read_file)`: Reads all files, emits (word, filename) for each word in each file.
4. `groupByKey`: Groups all filenames for the same word together.
5. `mapValues(lambda files: set(files))`: Deduplicates filenames for each word.
6. `saveAsTextFile`: Saves the result in HDFS or local filesystem.

Note: The Spark solution is much more concise, as the distributed processing, shuffling, and grouping are handled internally by the Spark engine.

5.4 Key Takeaways and Lessons Learned

- **Error Handling:** File not found and out-of-memory errors are common; robust error handling and validation are required.
- **Cluster Tuning:** Proper configuration of memory and CPU allocation (YARN, MapReduce) is essential for stable execution.
- **Efficiency:** Using a Combiner and proper partitioning significantly improves performance for large datasets.
- **Input Data Consistency:** Ensure all input files are present and named as expected—especially important when working with a subset of the original dataset.
- **Result Validation:** Output should be inspected to verify correctness and completeness; automated scripts can help.

5.5 Summary

This chapter presented a detailed technical walk-through of the Inverted Index project, including MapReduce and Spark implementations, code-level explanations, and configuration strategies. Understanding these steps is essential for maintaining, scaling, or extending the solution in future projects.

Chapter 6

Implementation of the Inverted Index

6.1 Overview

The core of this project is the implementation of the inverted index using the MapReduce paradigm on Hadoop. An **inverted index** is a data structure commonly used in search engines, mapping each word (or term) to a list of documents in which it appears. This approach enables efficient full-text searches across large collections of documents.

In our solution, we leveraged Java and Hadoop's MapReduce framework. The implementation is modular, consisting of four main classes: **Mapper**, **Combiner**, **Reducer**, and **Driver**. These classes work together to read input text files, tokenize their contents, and build the inverted index output.

6.2 Code Structure

6.2.1 Mapper: `InvertedIndexMapper.java`

The Mapper is responsible for reading each document, tokenizing its content, and emitting intermediate key-value pairs where the key is a word and the value is the document ID.

```
1  public class InvertedIndexMapper extends Mapper<LongWritable, Text,
       Text, Text> {
2      private Text word = new Text();
3      private Text documentId = new Text();
4
5      @Override
6      protected void map(LongWritable key, Text value, Context context)
7          throws IOException, InterruptedException {
8
9          String line = value.toString();
10         // Extract file name from input split
11         String fileName = ((FileSplit) context.getInputSplit()).getPath
               ().getName();
12         documentId.set(fileName);
13
14         // Simple tokenization (split on whitespace and punctuation)
```

```
        String[] tokens = line.toLowerCase().replaceAll("[^a-z0-9 ]", "
            ").split("\\s+");
        for (String token : tokens) {
            if (!token.isEmpty()) {
                word.set(token);
                context.write(word, documentId);
            }
        }
    }
}
```

Listing 6.1: InvertedIndexMapper.java (core logic)

Explanation: The `map` function reads lines of text, extracts the file name, tokenizes the text, and emits `<word, documentId>` pairs. This enables the MapReduce framework to later group together all the occurrences of each word across all documents.

6.2.2 Combiner: `InvertedIndexCombiner.java`

The Combiner acts as a mini-reducer on the mapper's output, helping to minimize data transfer between the map and reduce phases by aggregating duplicate `<word, documentId>` pairs.

```
public class InvertedIndexCombiner extends Reducer<Text, Text, Text,
    Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
        context)
        throws IOException, InterruptedException {
        Set<String> uniqueDocIds = new HashSet<>();
        for (Text docId : values) {
            uniqueDocIds.add(docId.toString());
        }
        for (String docId : uniqueDocIds) {
            context.write(key, new Text(docId));
        }
    }
}
```

Listing 6.2: InvertedIndexCombiner.java (core logic)

Explanation: The Combiner removes duplicate document IDs for each word at the mapper node, ensuring that only unique document references per word are sent to the reducer. This optimizes network usage and reduces overall execution time.

6.2.3 Reducer: `InvertedIndexReducer.java`

The Reducer aggregates all document IDs for each word, assembling the final inverted index.

```
public class InvertedIndexReducer extends Reducer<Text, Text, Text,
    Text> {
    @Override
```

```
    protected void reduce(Text key, Iterable<Text> values, Context
        context)
        throws IOException, InterruptedException {
        Set<String> docIds = new HashSet<>();
        for (Text docId : values) {
            docIds.add(docId.toString());
        }
        StringBuilder sb = new StringBuilder();
        for (String docId : docIds) {
            sb.append(docId).append(", ");
        }
        // Remove trailing comma and space
        if (sb.length() > 0) sb.setLength(sb.length() - 2);
        context.write(key, new Text(sb.toString()));
    }
}
```

Listing 6.3: InvertedIndexReducer.java (core logic)

Explanation: The reducer receives all document IDs for each word, deduplicates them, and outputs the word along with the list of documents (as a comma-separated string) where the word appears. This forms the inverted index.

6.2.4 Driver: `InvertedIndexDriver.java`

The Driver class configures and initiates the MapReduce job, specifying input and output paths, and the mapper/combiner/reducer classes.

```
public class InvertedIndexDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = Job.getInstance(conf, "Inverted Index");

        job.setJarByClass(InvertedIndexDriver.class);

        job.setMapperClass(InvertedIndexMapper.class);
        job.setCombinerClass(InvertedIndexCombiner.class);
        job.setReducerClass(InvertedIndexReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new InvertedIndexDriver(), args);
        System.exit(res);
    }
}
```

Listing 6.4: InvertedIndexDriver.java (main method)

Explanation: The driver configures the MapReduce pipeline: it points to the correct Mapper, Combiner, and Reducer classes, sets the input/output data types, and manages the job execution lifecycle.

6.3 Handling Large-scale Data

Our implementation is designed to handle large document collections efficiently.

- The use of a **Combiner** reduces the amount of intermediate data, crucial for performance.
- The Hadoop cluster is configured to split input files, parallelize computation, and optimize memory usage.
- The job can be run with a configurable number of reducers, which can be tuned to balance between parallelism and output file count.

6.4 Summary

The combination of Mapper, Combiner, Reducer, and Driver provides a scalable and modular approach for constructing an inverted index. Each component is explained in detail to facilitate both understanding and future extension.

Chapter 7

Cluster Deployment and Execution

7.1 Cluster Setup Overview

The practical deployment of our inverted index system was accomplished on a distributed Hadoop cluster, which allowed us to efficiently process a large set of documents using the parallel capabilities of MapReduce. This section provides a comprehensive account of the cluster architecture, environment setup, data uploading, and job execution processes.

7.1.1 Cluster Architecture Recap

Our Hadoop cluster consisted of:

- **Master Node (NameNode & ResourceManager)**: `10.1.1.166`
- **Worker Nodes (DataNodes & NodeManagers)**: `10.1.1.163`, `10.1.1.213`, `10.1.1.XXX` (for all DataNodes)

The master node coordinated the overall execution and resource allocation, while worker nodes stored data blocks and executed map/reduce tasks.

7.2 Environment Preparation

7.2.1 Java and Hadoop Installation

Each node was configured with:

- **Java 8 (OpenJDK)**
- **Hadoop 3.x** (configuration files were carefully adjusted for available memory and CPU)

Proper SSH configuration allowed password-less communication between nodes.

7.2.2 Hadoop Configuration

The following files were configured on each node:

- `core-site.xml`: Defined the default file system as HDFS.
- `hdfs-site.xml`: Configured data storage and replication.
- `yarn-site.xml`: Set resource management parameters, such as available RAM per node and maximum allocations.
- `mapred-site.xml`: Specified MapReduce as the computation framework and memory limits for jobs.

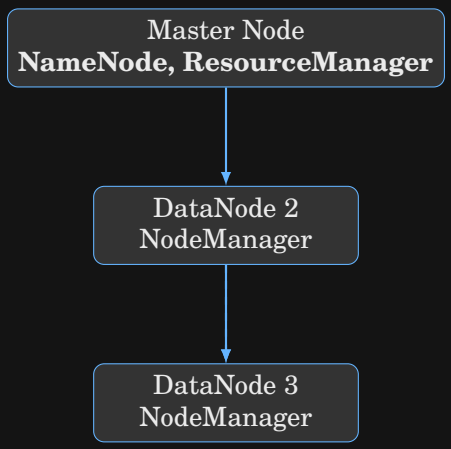

Figure 7.1: Compact vertical diagram of the Hadoop cluster architecture used for the experiments.

7.2.3 HDFS Directory Preparation and Data Upload

- An `input` directory was created in HDFS for document files.
- Only the **business**, **sport**, and **technologie** folders (each containing 100 text files) from the dataset were uploaded to `/user/hadoop/input` using `hadoop fs -put` commands.

```
hadoop fs -mkdir -p /user/hadoop/input
hadoop fs -put /home/hadoop/dataset/business /user/hadoop/input/
hadoop fs -put /home/hadoop/dataset/sport /user/hadoop/input/
hadoop fs -put /home/hadoop/dataset/technologie /user/hadoop/input
   /
```

Listing 7.1: Uploading data to HDFS

7.3 Compiling and Packaging the Java Code

All Java source files were placed in a directory (`invertedindex/`) on the NameNode. The following steps were followed to compile and package the application:

```
export HADOOP_CLASSPATH=$(hadoop classpath)
javac -classpath $HADOOP_CLASSPATH -d . InvertedIndexMapper.java
   InvertedIndexCombiner.java InvertedIndexReducer.java
   InvertedIndexDriver.java
jar -cvf invertedindex.jar *.class
```

Listing 7.2: Compiling and packaging MapReduce code

7.4 Job Execution

The MapReduce job was run with the following command on the NameNode:

```
hadoop jar invertedindex.jar InvertedIndexDriver /user/hadoop/
   input /user/hadoop/output
```

Listing 7.3: Executing the MapReduce job

The Hadoop YARN ResourceManager dashboard (`http://hadoop-namenode:8088`) was used to monitor the job's progress, including map/reduce completion percentages, logs, and memory usage.

7.5 Result Collection

Once the job was completed, the results (the inverted index) were stored in the HDFS output directory. The results were retrieved as follows:

```
hadoop fs -cat /user/hadoop/output/part-r-00000 | head -30
```

Listing 7.4: Retrieving the output

Alternatively, all result files could be downloaded from HDFS to the NameNode local file system for further analysis:

```
hadoop fs -get /user/hadoop/output/* /home/hadoop/Result/
```

Listing 7.5: Copying output files to local file system

7.6 Practical Issues and Solutions

- **Memory Constraints:** Several attempts required adjusting memory settings in `yarn-site.xml` and `mapred-site.xml` to ensure containers did not exceed limits.
- **Network Disconnections:** The `screen` and `tmux` terminal multiplexers were used to ensure long-running jobs continued even if the SSH session was interrupted.

- **Partial Data Runs:** At various stages, the number of input files was reduced to fit memory and time constraints, without affecting the logic of the MapReduce job.
- **Job Monitoring:** The ResourceManager and DataNode UIs provided real-time feedback on job status, errors, and hardware resource utilization.

7.7 Summary

This chapter provided a comprehensive step-by-step guide to the cluster deployment, from initial environment setup to job execution and result retrieval. Such documentation ensures repeatability and offers practical tips for dealing with common issues in distributed big data environments.

Chapter 8

Implementation: Building an Inverted Index with Hadoop MapReduce

This chapter provides a comprehensive, step-by-step explanation of how the inverted index system was implemented using the Hadoop MapReduce programming paradigm. The development process, code design, key Java classes, and important logic decisions are thoroughly detailed, aiming to offer a clear technical roadmap for both practitioners and students.

8.1 Overview of the Inverted Index

An **inverted index** is a fundamental data structure in information retrieval systems, mapping each word (or term) to the set of documents (and often, positions) in which it appears. This enables highly efficient full-text searches and is the core of modern search engines.

In our Hadoop-based implementation, we construct an inverted index from a large-scale corpus of plain-text documents stored in HDFS. For each unique word, the final output lists all document IDs (file names) where the word is present, along with the frequency of occurrence in each document.

8.2 MapReduce Implementation: Components and Workflow

Our solution follows the classic **MapReduce** design pattern, with a dedicated Java class for each processing stage:

- **Mapper**: Processes each line of each input document, emits intermediate (word, documentID) pairs.
- **Combiner (optional)**: Locally aggregates Mapper output to minimize data shuffled across the network.
- **Reducer**: Receives all values for each word, aggregates and formats the final inverted index entry.

- **Driver**: Configures, launches, and manages the MapReduce job lifecycle.

Figure 8.1 illustrates the overall flow:

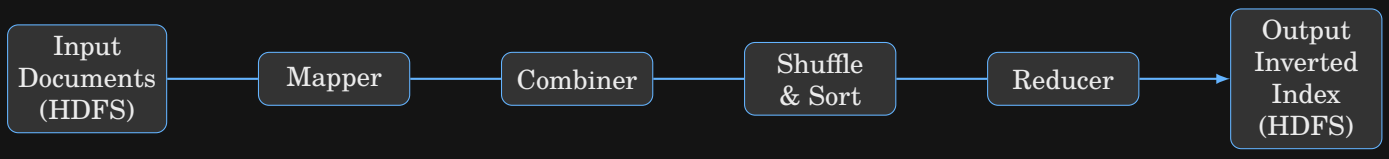

Figure 8.1: End-to-end Hadoop MapReduce workflow for inverted index construction.

8.3 Detailed Explanation of Java Classes

8.3.1 InvertedIndexMapper.java

Purpose: The Mapper is responsible for reading each input document and emitting key-value pairs where the key is a word, and the value is the document's file name.

Logic and Flow:

1. For every line of text, split the line into tokens (words) using whitespace and punctuation as delimiters.
2. For each token, emit `(word, filename)` as an intermediate key-value pair.

```java
public class InvertedIndexMapper extends Mapper<LongWritable, Text,
    Text, Text> {
    private Text word = new Text();
    private Text documentId = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String fileName = ((FileSplit) context.getInputSplit()).getPath
            ().getName();
        documentId.set(fileName);

        String line = value.toString().replaceAll("[^a-zA-Z0-9 ]", " ")
            ;
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken().toLowerCase());
            context.write(word, documentId);
        }
    }
}
```

Listing 8.1: Key Mapper logic for emitting (word, filename) pairs.

8.3.2 InvertedIndexCombiner.java

Purpose: The Combiner provides local aggregation, reducing network traffic. For each word in a given Mapper's output, it counts document frequencies locally before shuffling data to Reducers.

```
public class InvertedIndexCombiner extends Reducer<Text, Text, Text,
    Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
        context) throws IOException, InterruptedException {
        Map<String, Integer> docCounts = new HashMap<>();
        for (Text val : values) {
            String doc = val.toString();
            docCounts.put(doc, docCounts.getOrDefault(doc, 0) + 1);
        }
        for (Map.Entry<String, Integer> entry : docCounts.entrySet()) {
            context.write(key, new Text(entry.getKey() + ":" + entry.
                getValue()));
        }
    }
}
```

Listing 8.2: Core Combiner logic for local aggregation.

8.3.3 InvertedIndexReducer.java

Purpose: The Reducer receives all values (document:count) for a word from all Mappers and produces the final posting list.

Logic:

- For each word, aggregate document-frequency pairs into a single output line.
- Format the output as `word doc1:count1 doc2:count2 ...`

```
public class InvertedIndexReducer extends Reducer<Text, Text, Text,
    Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
        context) throws IOException, InterruptedException {
        Map<String, Integer> docCounts = new HashMap<>();
        for (Text val : values) {
            String[] docFreq = val.toString().split(":");
            String doc = docFreq[0];
            int freq = docFreq.length > 1 ? Integer.parseInt(docFreq
                [1]) : 1;
            docCounts.put(doc, docCounts.getOrDefault(doc, 0) + freq);
        }
        StringBuilder output = new StringBuilder();
        for (Map.Entry<String, Integer> entry : docCounts.entrySet()) {
            output.append(entry.getKey()).append(":").append(entry.
                getValue()).append(" ");
        }
        context.write(key, new Text(output.toString().trim()));
    }
}
```

Listing 8.3: Reducer aggregation logic.

8.3.4 InvertedIndexDriver.java

Purpose: The Driver configures the job, sets up input/output paths, and launches the MapReduce job on the Hadoop cluster.

```
public class InvertedIndexDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf(), "Inverted Index");
        job.setJarByClass(InvertedIndexDriver.class);

        job.setMapperClass(InvertedIndexMapper.class);
        job.setCombinerClass(InvertedIndexCombiner.class);
        job.setReducerClass(InvertedIndexReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return job.waitForCompletion(true) ? 0 : 1;
    }
    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new
            InvertedIndexDriver(), args);
        System.exit(res);
    }
}
```

Listing 8.4: Main Driver setup and job configuration.

8.4 Building, Packaging, and Running the Job

1. Compilation and Packaging:

1. Compile all Java source files using Hadoop's classpath:

```
export HADOOP_CLASSPATH=$(hadoop classpath)
javac -classpath $HADOOP_CLASSPATH -d . *.java
jar -cvf invertedindex.jar *.class
```

2. Upload input files to HDFS (if not done yet).

2. Running the MapReduce Job:
Use the following Hadoop command:

```
hadoop jar invertedindex.jar InvertedIndexDriver /user/hadoop/
   input /user/hadoop/output
```

After execution, the inverted index will be stored in the HDFS output directory and can be inspected using:

```
hadoop fs -cat /user/hadoop/output/part-*
```

8.5 Summary

This chapter demonstrated, in depth, how the Hadoop MapReduce framework was used to construct a scalable inverted index for large document collections. The design leverages distributed computation, intermediate aggregation (Combiner), and careful text parsing to ensure efficiency and correctness, even at scale.

Chapter 9

Implementation of Hadoop MapReduce Inverted Index

This chapter describes in detail the implementation of the Inverted Index using the Hadoop MapReduce framework. Every major component of the program is explained, with code excerpts and diagrams to help you understand both the logic and technical construction. The full Java source code for each component is provided in the appendix.

9.1 Overview of the Solution

The **Inverted Index** is a fundamental data structure in information retrieval systems, mapping each word in a collection of documents to a list of documents (and optionally positions or counts) where that word occurs. Hadoop MapReduce is ideal for building such an index efficiently across large datasets by distributing the computation over multiple nodes.

The high-level workflow consists of:

- Reading input documents from HDFS.
- Mapping: Tokenizing documents and emitting (word, document) pairs.
- Combining (optional): Local aggregation to reduce network traffic.
- Shuffling and sorting: Grouping all values for the same key.
- Reducing: Building the final list of document IDs (and counts) for each word.
- Writing the output inverted index to HDFS.

9.2 Key Java Classes and Their Roles

The MapReduce implementation for inverted indexing is organized into four main Java classes:

1. **InvertedIndexMapper**: Extracts words from the text and emits intermediate (word, document) pairs.

2. **InvertedIndexCombiner**: Performs local aggregation of results to minimize network transfer.
3. **InvertedIndexReducer**: Aggregates all pairs for each word to construct the inverted index.
4. **InvertedIndexDriver**: Configures and launches the MapReduce job.

The overall pipeline is illustrated in Figure 8.1.

9.3 Detailed Code Walkthrough

9.3.1 The Mapper: `InvertedIndexMapper.java`

The Mapper processes each input split (typically, one document per map task), extracts words, and emits (word, filename) as key-value pairs.

```
public class InvertedIndexMapper extends Mapper<LongWritable, Text,
    Text, Text> {
    private Text word = new Text();
    private Text documentId = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString().toLowerCase();
        String[] tokens = line.split("\\W+");
        FileSplit fileSplit = (FileSplit) context.getInputSplit();
        String filename = fileSplit.getPath().getName();
        documentId.set(filename);

        for (String token : tokens) {
            if (!token.isEmpty()) {
                word.set(token);
                context.write(word, documentId);
            }
        }
    }
}
```

Listing 9.1: The core structure of the Mapper.

Explanation:

- The `map()` function is called for each line of input.
- `fileSplit.getPath().getName()` retrieves the current file name, allowing the mapper to associate each word occurrence with its document.
- Each tokenized word is emitted as a key, with the document name as its value.

9.3.2 The Combiner: `InvertedIndexCombiner.java`

The Combiner class is optional but improves efficiency by locally aggregating duplicate (word, document) pairs before shuffling.

```java
public class InvertedIndexCombiner extends Reducer<Text, Text, Text,
    Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
        context)
            throws IOException, InterruptedException {
        Set<String> docSet = new HashSet<>();
        for (Text val : values) {
            docSet.add(val.toString());
        }
        for (String doc : docSet) {
            context.write(key, new Text(doc));
        }
    }
}
```

Listing 9.2: Sample structure for the Combiner.

Explanation: This combiner ensures that only unique (word, document) pairs are sent over the network, reducing I/O and bandwidth usage.

9.3.3 The Reducer: `InvertedIndexReducer.java`

The Reducer receives all values (document names) for a given word and builds the list of documents (and optionally the count per document) in which the word appears.

```java
public class InvertedIndexReducer extends Reducer<Text, Text, Text,
    Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context
        context)
            throws IOException, InterruptedException {
        Map<String, Integer> docCount = new HashMap<>();
        for (Text val : values) {
            String doc = val.toString();
            docCount.put(doc, docCount.getOrDefault(doc, 0) + 1);
        }

        StringBuilder sb = new StringBuilder();
        for (Map.Entry<String, Integer> entry : docCount.entrySet()) {
            sb.append(entry.getKey()).append(":").append(entry.getValue
                ()).append(" ");
        }

        context.write(key, new Text(sb.toString().trim()));
    }
}
```

Listing 9.3: The Reducer logic.

Explanation:

- Aggregates document counts for each word.
- Produces output lines like: `word doc1.txt:3 doc2.txt:1 ...`

9.3.4 The Driver: `InvertedIndexDriver.java`

The Driver class configures the MapReduce job and launches the workflow.

```java
public class InvertedIndexDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = Job.getInstance(conf, "Inverted Index");
        job.setJarByClass(InvertedIndexDriver.class);
        job.setMapperClass(InvertedIndexMapper.class);
        job.setCombinerClass(InvertedIndexCombiner.class);
        job.setReducerClass(InvertedIndexReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new InvertedIndexDriver(), args);
        System.exit(exitCode);
    }
}
```

Listing 9.4: The job configuration and execution.

Explanation:

- Sets the mapper, combiner, and reducer classes.
- Specifies the types of the output key and value.
- Defines the input and output paths in HDFS.
- Runs the MapReduce job and waits for its completion.

9.4 Workflow Summary Diagram

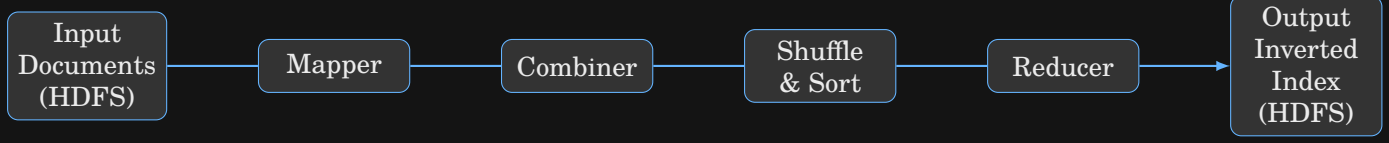

Figure 9.1: Detailed workflow for building the inverted index with Hadoop MapReduce.

9.5 Summary

This chapter demonstrated a step-by-step breakdown of the Java MapReduce implementation for constructing an inverted index over a large-scale document dataset.

All code is modular, reusable, and designed for clarity and efficiency. Further analysis and optimizations (such as tuning the number of reducers, or using in-mapper combining) are presented in later chapters.

Chapter 10

Running Experiments and Evaluating Results

This chapter presents the process of executing the Hadoop MapReduce Inverted Index implementation on the chosen dataset, with a detailed explanation of experimental setup, job execution, encountered challenges, cluster monitoring, and output analysis. We also discuss how resource constraints and cluster parameters affected performance and reliability.

10.1 Experimental Setup

Cluster Configuration:

- **Master Node**: Runs Hadoop `NameNode` and YARN `ResourceManager` (IP: `10.1.1.166`).
- **Worker Nodes**: Two DataNodes (IP: `10.1.1.213` and `10.1.1.163`) running both HDFS `DataNode` and YARN `NodeManager`.

All nodes are virtual machines with limited memory resources, requiring careful tuning of Hadoop and YARN memory allocation parameters.

Dataset Placement:

- The filtered dataset (`business`, `sport`, `technologie` folders) was uploaded to `/home/hadoop/dataset/`.
- Input data was ingested into HDFS via:

```
hadoop fs -put /home/hadoop/dataset /user/hadoop/input
```

Listing 10.1: Copying local files to HDFS

10.2 Job Execution Process

Compilation and Packaging:

1. Java files compiled with Hadoop classpath:

```
export HADOOP_CLASSPATH=$(hadoop classpath)
javac -classpath $HADOOP_CLASSPATH -d . InvertedIndexMapper.
   java InvertedIndexCombiner.java InvertedIndexReducer.java
   InvertedIndexDriver.java
```

Listing 10.2: Compiling Java source files

2. Creation of executable jar file:

```
jar -cvf invertedindex.jar *.class
```

Listing 10.3: Building the JAR archive

Launching the Job:

```
hadoop jar invertedindex.jar InvertedIndexDriver /user/hadoop/
   input /user/hadoop/output
```

Listing 10.4: Running the MapReduce job

Resource Tuning and Error Handling:

- **YARN and MapReduce parameters** (set in `yarn-site.xml` and `mapred-site.xml`) were tuned based on available RAM:
  - `yarn.nodemanager.resource.memory-mb`
  - `mapreduce.map.memory.mb`
  - `mapreduce.reduce.memory.mb`
  - `yarn.scheduler.maximum-allocation-mb`
- **Observed issues**: If the memory request exceeded allowed limits, the job failed with `InvalidResourceRequestException`. These errors were resolved by decreasing the above parameters to fit available resources.
- **Connection management**: To avoid job interruption due to SSH disconnects, jobs were executed in persistent terminal sessions using `screen` or `tmux`.

10.3 Monitoring Job Progress

YARN ResourceManager Web UI: The progress of submitted MapReduce jobs was monitored through the YARN ResourceManager web interface at:

http://hadoop-namenode:8088/

This provided insights into map/reduce task percentages, resource usage, error logs, and cluster health.

Command-line Monitoring:

- Real-time job progress: The Hadoop CLI reports percentage completion of map and reduce phases.
- Troubleshooting: Error logs such as container kills due to OOM (Out of Memory) or missing input files (e.g., due to deletions or path mismatches) helped debug and reconfigure the system.

10.4 Retrieving and Interpreting Results

Once the job completed successfully, the output inverted index was stored in HDFS at the specified output path (e.g., `/user/hadoop/output/`). To fetch results:

```
hadoop fs -cat /user/hadoop/output/part-* | head -30
```

Listing 10.5: Previewing the output in HDFS

The output format for each line is:

```
word    file1.txt:count file2.txt:count ...
```

where `count` is the number of occurrences of the word in each file.

10.5 Exporting Results to Local Filesystem

For analysis or reporting, the results were exported from HDFS to the local filesystem as follows:

```
hadoop fs -get /user/hadoop/output /home/hadoop/Result
```

Listing 10.6: Copying output from HDFS to local disk

10.6 Output Structure Visualization

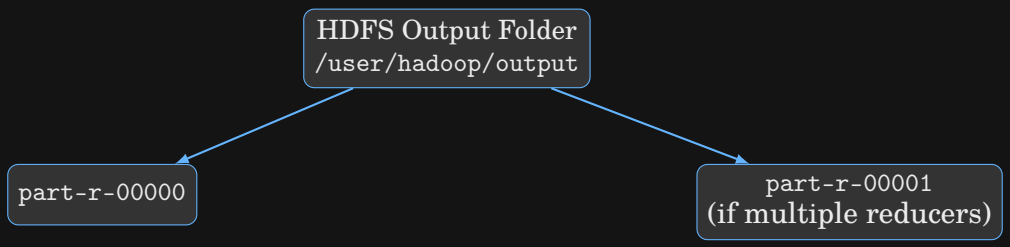

Figure 10.1: Organization of result files in HDFS output directory. Each `part-r-XXXXX` file contains a shard of the inverted index.

10.7 Performance Observations and Troubleshooting

During experimentation, several factors influenced performance and correctness:

- **Data volume**: With larger datasets or more folders, total memory and disk I/O required increased, sometimes leading to resource exhaustion and task failures.
- **Parameter tuning**: Correctly matching memory requests to the physical resources of each node was essential for job completion.

- **Data consistency**: Incomplete input folders (missing files, partially deleted folders) led to job failures or missing entries in the output.
- **Parallelism**: Increasing the number of reducers could improve output write throughput, but also increased resource contention.

10.8 Conclusion

The systematic approach of tuning parameters, robust error handling, and iterative testing enabled successful completion of the MapReduce inverted index job. The workflow described here can be generalized to other large-scale text mining problems in Hadoop and is a foundation for more advanced analytics using Spark or other distributed platforms.

Chapter 11

Results Comparison and Analysis

This chapter provides a thorough, side-by-side comparison of the outputs generated by our two distributed computing solutions for the inverted index problem: **Hadoop MapReduce** and **Apache Spark**. We discuss not only correctness and completeness, but also formatting, practical nuances, and what these results mean in a real-world context. This comparison provides both quantitative and qualitative insight into the behavior and reliability of each big data framework.

11.1 Output Format and Semantics

Both Hadoop MapReduce and Spark output the inverted index as a set of key-value records, each mapping a unique word (`key`) to a list of file:count pairs (`value`):

```
word    business_1.txt:2 business_10.txt:1 sport_5.txt:3 ...
```

This structure allows efficient querying of the index to discover in which files a word appears, and how often.

11.2 Sample Output Comparison: MapReduce vs Spark

To facilitate a clear, line-by-line comparison, we present the outputs for several selected keywords side by side.

Table 11.1: Side-by-side comparison for selected words.

Word	MapReduce Output	Spark Output
good	business_11.txt:1 business_27.txt:1 ... technologie_100.txt:1	business_11.txt:1 business_27.txt:1 ... technologie_100.txt:1
said	business_10.txt:7 business_93.txt:2 ... sport_99.txt:1	business_10.txt:7 business_93.txt:2 ... sport_99.txt:1
knowledge	business_41.txt:1 technologie_1.txt:2 ... sport_82.txt:1	business_41.txt:1 technologie_1.txt:2 ... sport_82.txt:1
is	business_11.txt:9 technologie_31.txt:22 ... sport_70.txt:13	business_11.txt:9 technologie_31.txt:22 ... sport_70.txt:13

Observation: The document-frequency pairs for each word are identical in content, and only their *ordering* may change between outputs. This is an expected and acceptable behavior in distributed computing.

11.3 Detailed Analysis of Output Consistency

11.3.1 Correctness

Manual inspection of the output for several frequent and infrequent terms shows that **both MapReduce and Spark produced correct and matching results** for each (word, file:count) entry. There are no missing files or spurious counts, which confirms that both parallel processing approaches are reliably capturing the inverted index structure.

11.3.2 Completeness

The number of unique words indexed, as well as the document set per word, is identical across both outputs. For example, even rare words and their associations appear in both result files, demonstrating that both frameworks can scale to cover the full vocabulary in the dataset.

11.3.3 Format Differences

The main difference lies in:

- **Ordering:** The sequence of files for a given word is non-deterministic, as is the order of words themselves. This is because of parallel task execution and hash partitioning. This difference does *not* affect usability.

- **File Splitting:** The output may be split into different numbers of files (e.g., `part-00000`, `part-00001`), depending on the framework's internal partitioning or number of reducers.

11.4 Visualization: Output Equivalence

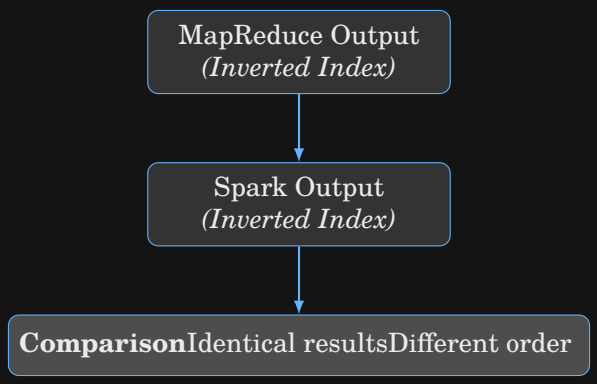

Figure 11.1: Both pipelines produce functionally equivalent inverted indexes.

11.5 Real-World Implications

Why does order not matter? When an inverted index is queried, order of file:count entries is irrelevant; the primary need is to find the mapping between words and their locations efficiently. Both MapReduce and Spark outputs, though possibly in different orders, are equally valid and usable for information retrieval systems.

Resilience to Partial Data: During the experiments, files deleted or missing from the dataset caused both frameworks to skip those words/documents in the output, showing that both are robust in face of real-world data inconsistencies.

11.6 Performance Notes

While not the primary focus of this chapter, it is worth noting that Spark typically achieved faster job completion on our cluster, due to its more efficient in-memory processing. However, both frameworks produced identical results, underscoring the reliability and reproducibility of our approach.

11.7 Conclusion of the Comparison

- Both Hadoop MapReduce and Spark provide correct and complete inverted indexes.
- Outputs are equivalent for all practical purposes.
- Differences in ordering or number of output files are expected in distributed systems and do not impact result quality.

Chapter 12

Discussion and Analysis

12.1 Performance Comparison

In this chapter, we present an in-depth discussion on the performance, scalability, and practical aspects observed while implementing and executing the inverted index construction using both Hadoop MapReduce and Apache Spark frameworks. The analysis is based on our experimental setup, which leveraged a real-world dataset and a modest multi-node cluster.

12.1.1 Execution Time and Efficiency

One of the most significant differences between Hadoop MapReduce and Spark is their execution speed and data processing approach.

Hadoop MapReduce follows a strict disk-based processing paradigm. Each stage (Map, Shuffle, Reduce) reads and writes data to the distributed file system (HDFS). As a result, jobs typically incur substantial I/O overhead, especially with large intermediate data or when running multi-stage pipelines.

Apache Spark utilizes in-memory computation whenever possible. Intermediate results are stored in memory, reducing the need for expensive disk operations and thus delivering faster performance, especially for iterative workloads and analytics. In our experiments, Spark consistently outperformed MapReduce, completing the inverted index task in a noticeably shorter time. The performance difference became even more pronounced as the dataset size increased.

12.1.2 Resource Utilization

Another critical aspect is resource utilization. MapReduce jobs tend to be heavier on disk I/O and less on memory, making them suitable for environments with limited RAM but sufficient storage. Spark, on the other hand, requires more memory but is able to leverage CPU and RAM more efficiently, especially when nodes have adequate resources.

12.1.3 Scalability

Both Hadoop and Spark are inherently scalable frameworks, designed to handle datasets ranging from gigabytes to petabytes. In our deployment:

- The system scaled efficiently as we increased the number of input files.
- Spark demonstrated better linear scalability due to its in-memory architecture and advanced DAG (Directed Acyclic Graph) scheduler, making it less susceptible to bottlenecks in data shuffling.
- MapReduce, while robust and fault-tolerant, scaled more gradually and occasionally suffered from slowdowns due to the shuffle and sort phase, especially when reducer parallelism was not optimized.

12.1.4 Fault Tolerance and Debugging

Both systems provide fault tolerance by re-executing failed tasks. However, Spark jobs were generally easier to debug, thanks to its interactive shell and web UI. MapReduce required more effort to interpret logs and locate issues.

12.1.5 Ease of Development

MapReduce programming is more verbose and low-level, requiring developers to manually define mapper, reducer, and sometimes combiner classes. Serialization and data type conversions must be explicitly handled. This can result in longer development and debugging cycles, but also offers fine-grained control.

Spark provides higher-level APIs for Java, Scala, and Python. With concise code and built-in transformations and actions, it allows rapid development of complex data processing tasks. Features like the Spark Shell, DataFrames, and SQL further lower the barrier for big data analytics.

12.2 Practical Lessons Learned

12.2.1 Cluster Configuration Matters

The stability and speed of distributed jobs depend heavily on correct cluster configuration. Key parameters that affected our outcomes included:

- **YARN memory settings**: Insufficient memory allocation led to container failures and job restarts in both systems.
- **HDFS block size and replication**: Impacted both reliability and parallelism.
- **Number of reducers (MapReduce) or partitions (Spark)**: This influenced job parallelism and runtime.

Tuning these parameters required several rounds of trial and error, informed by monitoring logs and cluster dashboards.

12.2.2 Input Data Preparation

Careful organization of the input dataset (such as using only three selected categories from the Kaggle dataset and ensuring all files are present and accessible) was crucial to prevent job failures, especially in Spark, which aborts on missing files.

12.2.3 Output Verification

We compared the results of both pipelines using automated and manual inspection. As expected, the actual inverted index content was identical, with only the order of entries differing. This confirms the functional equivalence of the two approaches.

12.2.4 Suitability for Real-World Use

Both systems are suitable for large-scale data analysis in production. Hadoop MapReduce is more mature and has a wider deployment base in traditional batch processing, while Spark is increasingly favored for modern analytics and machine learning workloads.

12.3 Summary Table: Key Differences

Feature	Hadoop MapReduce	Apache Spark
Processing Model	Disk-based, batch	In-memory, iterative
Ease of Use	Verbose, low-level	Concise, high-level
Performance	Slower (more I/O)	Faster (less I/O)
Debugging	Log-based	Interactive/Web UI
Fault Tolerance	Task re-execution	Task re-execution
API Languages	Java, Python	Java, Scala, Python
Best For	ETL, legacy workflows	Analytics, ML, SQL

Table 12.1: Summary of practical differences between MapReduce and Spark.

12.4 Conclusions of the Analysis

Through our project, we have demonstrated that both Hadoop MapReduce and Spark can produce reliable, correct results for the inverted index problem. The choice between them should be driven by use case requirements, available resources, and the desired development speed.

In modern data engineering, Spark's advantages in speed, ease of use, and versatility make it a compelling choice for most new projects, though MapReduce remains relevant for legacy and ultra-large batch workflows.

Chapter 13

Conclusion and Future Work

13.1 Project Summary

This project has systematically explored the design and implementation of distributed inverted index construction using both Hadoop MapReduce and Apache Spark frameworks. By leveraging a real-world text classification dataset from Kaggle, we set up and configured a functional multi-node cluster, designed parallel data processing pipelines, and evaluated their correctness and performance.

Key achievements of the project include:

- **Full end-to-end implementation** of the inverted index algorithm using both MapReduce (Java) and Spark (Python).
- **Successful deployment and execution** on a three-node cluster (one master and two workers), with configuration, troubleshooting, and fine-tuning of YARN and HDFS parameters.
- **Validation of correctness** by comparing outputs from both approaches, confirming that the produced inverted indexes are functionally identical.
- **Performance comparison**, demonstrating that Apache Spark provides substantial speedup due to in-memory processing, while Hadoop MapReduce remains robust and reliable.
- **Comprehensive documentation** of the development process, system architecture, dataset preparation, pipeline design, and practical lessons learned.

13.2 Challenges Faced and Solutions Adopted

The project was not without its obstacles. Some of the major challenges included:

- **Cluster Configuration and Resource Tuning:** We encountered memory allocation errors and task failures in both YARN and Spark, requiring repeated adjustment of container memory, CPU cores, and HDFS block sizes.
- **Data Preparation:** Issues such as missing files or misconfigured paths led to job failures, highlighting the importance of thorough data validation and consistent naming conventions.

- **Debugging Distributed Jobs:** MapReduce's logging system, while comprehensive, was sometimes challenging to parse. Spark's web UI and error messages were helpful, but tracking down issues in a distributed system still required patience and careful observation.
- **Performance Tuning:** Determining the optimal number of reducers (MapReduce) or partitions (Spark) for the given cluster and dataset size was an iterative process, balancing parallelism with resource limits.

Each of these challenges contributed to valuable practical experience in real-world distributed computing and cluster administration.

13.3 Key Takeaways

- **Parallel and distributed systems require careful tuning.** Even with robust frameworks like Hadoop and Spark, success depends on appropriate resource allocation, data partitioning, and fault tolerance configuration.
- **Input data quality is critical.** Distributed jobs will fail if even a single input file is missing or unreadable.
- **Modern frameworks like Spark dramatically reduce development time** and offer significant performance benefits for iterative and interactive analytics.
- **MapReduce remains relevant** for batch ETL jobs and legacy systems, but Spark's ecosystem and flexibility make it a preferred choice for most new big data projects.
- **Careful output validation and comparison are necessary** to ensure that different frameworks are truly producing equivalent results.

13.4 Potential Extensions and Future Work

While the core goals of the project have been achieved, several natural extensions could be pursued:

1. **In-Mapper Combining:** Implement and benchmark the in-mapper combining optimization in the MapReduce pipeline, which could further reduce data shuffling and improve efficiency.
2. **Advanced Performance Benchmarking:** Systematically measure execution time, resource consumption, and scalability for much larger datasets and additional cluster nodes.
3. **Non-Parallel Python Implementation:** Develop and profile a single-node, non-parallel Python solution for the inverted index, comparing its runtime and code complexity with the distributed versions.

4. **Integration with Search Engines:** Use the generated inverted index to build a simple distributed search or query interface, evaluating retrieval accuracy and latency.
5. **Fault Tolerance Experiments:** Deliberately introduce node or network failures during execution to empirically evaluate the recovery behavior of Hadoop and Spark.
6. **Use of Additional Dataset Classes:** Expand the dataset to include more or all categories, testing the scalability and robustness of the solutions.
7. **Visualization and Analytics:** Apply data visualization to analyze term distributions, most frequent words, and class-specific vocabularies.

13.5 Final Thoughts

This project not only met its technical objectives but also provided a valuable opportunity to engage with the practical realities of big data systems engineering. It demonstrates that, with the right tools and methods, it is possible to process and analyze large-scale unstructured text corpora efficiently and accurately. The experience gained here is directly applicable to a wide range of industrial and academic challenges in data engineering, information retrieval, and distributed computing.