100 Most Popular Words(using MapReduce)

Programming Assignment 2

COEN 242 Big Data (Spring 2023)

Instructor Prof. Kishore Pusukuri
Santa Clara University

Shweta Deshmukh (1650082) Xiaoxiao Jiang (1651657)

THE PROBLEM STATEMENT:

1. Determine 100 most frequent/repeated words in the given dataset using MapReduce.
2. Determine 100 most frequent/repeated words in the given dataset considering only the words having more than 6 characters using MapReduce.

Input:

1. Use Previously used datasets
2. For testing you could use the smaller datasets(50MB or 300MB dataset)
3. **But finally the algorithm should be implemented on the largest dataset (i.e.16GB dataset).**

Experiment Environment

- **Hardware:** Macbook M1 Pro, 8-core processor, 512GB disk space, 16GB memory
- **Software:** Hadoop 3.3.4, AdoptOpenJDK 8

Code Design

This Java program leverages Hadoop's MapReduce framework to solve the problem of finding the 100 most frequent words in a large dataset, while only considering words that are longer than 6 characters. The code can be broken down as follows:

Main Class: WordCount

This is the main driver class for the MapReduce job. It sets up the job's configuration, including input and output paths, the Mapper, Combiner, and Reducer classes, and the output key and value types. It also specifies the location of the "stopwords" file and the number of most frequent words to find (100).

Mapper Class: TokenizerMapper

The Mapper class tokenizes the input lines into individual words. It converts all words to lowercase, and ignores any words that are in the "stopwords" set or are shorter than 7 characters. Each word is output with a count of 1.

In the setup method, it reads a file of "stopwords" from the Hadoop Distributed File System (HDFS) into a HashSet for quick lookup during the map phase. This ensures that any common, uninformative words (like 'the', 'and', 'is', etc.) are ignored in the word frequency analysis.

Reducer(Combiner) Class: IntSumReducer

The IntSumReducer class sums up the counts for each word. It is also used as a combiner class in the MapReduce job. Combiners are an optimization in MapReduce that perform a local reduce on the mapper's output, which can significantly reduce the amount of data transferred between the mappers and reducers.

Reducer Class: TopKReducer

The TopKReducer class finds the 100 most frequent words out of all the words. It maintains a priority queue (min-heap) of size 100. For each word, it adds the word to the queue and if the size of the queue exceeds 100, it removes the least frequent word. This ensures that the queue always contains the 100 most frequent words seen so far.

In the cleanup method, it writes the words in the queue to the context, which are then written to the output file.

This design ensures that the MapReduce job can efficiently process large datasets that may not fit entirely in memory. By using a combiner and a priority queue, it minimizes the amount of data that needs to be transferred and stored, which can greatly speed up the job and reduce the memory requirements.

Summary

Overall, this program demonstrates how to perform a common big data analysis task (finding top 100 frequent words) using Hadoop's MapReduce. It uses techniques such as tokenizing input data in the Mapper, using a combiner for local aggregation, managing in-memory data structures in the Reducer, and controlling job configuration parameters. It also shows how to handle auxiliary data (stopwords) by loading it into memory in the Mapper's setup method. This design enables the program to efficiently process large datasets that might not fit entirely into memory, making it a scalable solution for big data processing tasks.

Tuning & Analysis

The purpose of this part was to evaluate the impact of different configurations on the performance of MapReduce tasks. The experiment was conducted using a **16GB dataset**, as specified in the documentation, which emphasizes the importance of implementing the algorithm on the largest dataset available

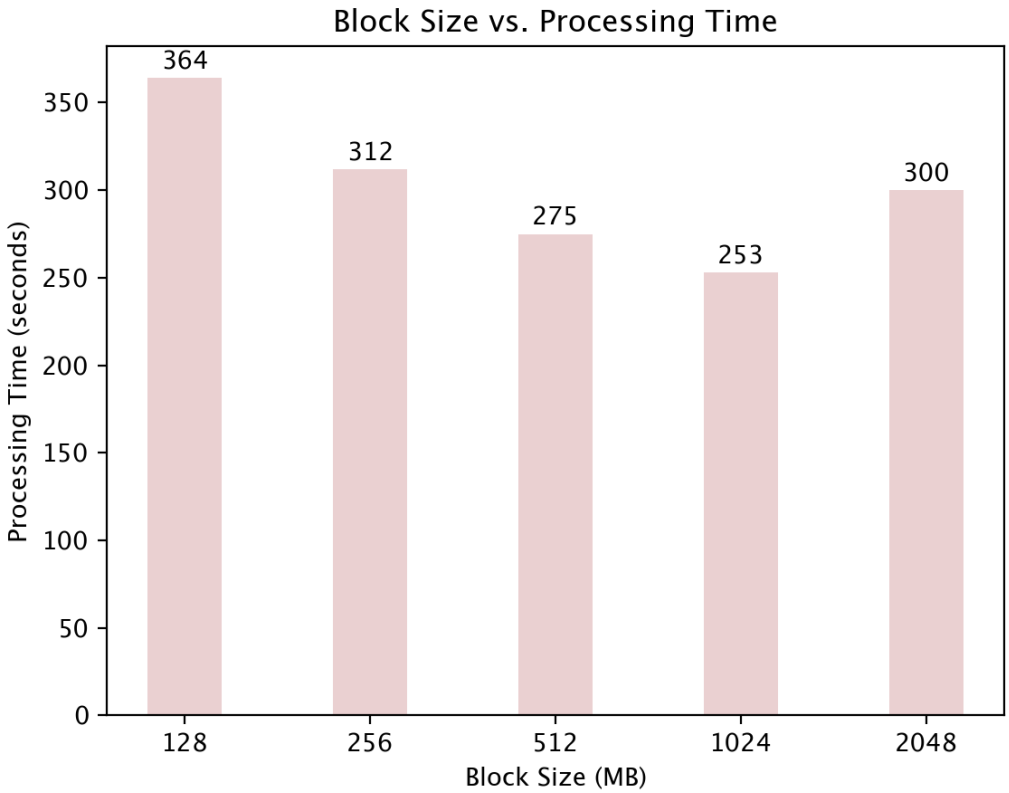
The following figure demonstrates the optimal runtime obtained for different dataset sizes.

Comparison of Processing Times for Datasets of Different Sizes

DataSet Size	Processing Time (sec)
300MB	22
2.5GB	65
16GB	253

Note: See details in output file.

Tuning and Analysis of Different Block Sizes (Based on a 16GB Dataset)



Block size plays a significant role in MapReduce performance. It determines the size of data blocks that are processed and distributed across nodes. A larger block size reduces the number of blocks and can potentially improve performance by reducing disk I/O and network overhead. However, excessively large block sizes may result in resource contention and increased processing time due to memory limitations and potential data skew.

In this experiment, five block sizes were tested: 128MB, 256MB, 512MB, 1GB, and 2GB. The objective was to identify the optimal block size for the MapReduce tasks in terms of processing time.

The results revealed that as the block size increased from 128MB to 1GB, the processing time consistently decreased. This improvement can be attributed to reduced overhead in handling a smaller number of larger blocks. The efficiency of data transfer and disk I/O operations improved, resulting in shorter processing times for these block sizes.

However, upon comparing the 1GB block size to the 2GB block size, a slight increase in processing time was observed. This can be attributed to the available resources on the machine, taking into account that the computer has 16GB of memory. However, after accounting for application overhead, the actual usable memory typically amounts to around 10GB. Furthermore, the machine has 8 cores available for processing.

Given these specifications, when processing a 16GB file with a block size of 2GB, only 5 of the available cores can be effectively utilized. Each Mapper and Reducer task handles a data block, and in this scenario, there would be 8 Mapper tasks running simultaneously, each processing a 2GB data block. However, due to the limited number of available cores (5 cores), only 5 Mapper tasks can be processed concurrently, while the remaining 3 tasks would have to wait for available cores. Consequently, this limitation hinders the parallelism and overall processing speed of the tasks, resulting in a longer processing time.

To summarize, the experiment conducted on a 16GB dataset demonstrated that increasing the block size from 128MB to 1GB generally reduced processing time. However, when the block size was further increased to 2GB, the processing time increased. The improvements in processing time for smaller block sizes can be attributed to reduced overhead, while the increase for the 2GB block size can be attributed to resource limitations and increased disk I/O.