Optimized Movie Rating Analysis with MapReduce

Student ID: 3089515
Course: Cluster Computing
Course Code: ITNPBD7
Coordinator: Dr David Cairns
Submission Date: 07/04/2023

Table Of Contents

1) Use of HDFS
1.1 Creating a directory
1.2 Loading Data
1.3 Submitting M/R job
1.4 Moving results to the local and Examining

2) Design
2.1 Design 1: Mapper, Combiner, and Reducer
2.2 Design 2: Mapper and Reducer Only
2.3 Comparison and Potential Improvements

3) Distributed Computation
3.1 Pros of using a Computational Cluster
3.2 Cons of using a Computational Cluster

4) Implementation
4.1 Brief
4.2 Python Files
4.3 Hadoop Output

1. Use of HDFS

The Hadoop Distributed File System (HDFS) is used for storing ,managing,analysing large data files across multiple machines in a distributed and fault-tolerant manner. In this project,I've used HDFS to store and analyse movie ratings data efficiently using the MapReduce framework.

1.1 Creating a directory in HDFS

To do a Hadoop M/R job we're to create a directory in HDFS, for this I've used the hdfs dfs -mkdir command. This command creates a new directory in the specified HDFS location.The following command was used to create a directory named "MoviesRatings" for the data:

```
hdfs dfs -mkdir /user/asa/MoviesRatings
```

1.2 Loading data into HDFS

Next,The movie ratings data was uploaded to our newly created directory in HDFS using the hdfs dfs -copyFromLocal command.This command copies the local file to the HDFS location specified.The following command was used to upload the "ratings.txt" file to the "MoviesRatings" directory:

```
hdfs dfs -copyFromLocal ratings.txt MoviesRatings
```

1.3 Submitting the MapReduce job

To submit the MapReduce job to the Hadoop cluster,I've used the hadoop jar command along with the Hadoop Streaming library,Specifying the input and output directories and the required mapper, combiner and reducer scripts.The following are the shell scripts that were used to run the MapReduce Jobs.

Shell Script for MapReduce job for selected years:

```
hadoop jar /home/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.0.jar \
-files mapper.py,combiner.py,reducer.py,years.txt \
-mapper mapper.py \
-combiner combiner.py \
-reducer reducer.py \
-input /user/asa/MoviesRatings/ratings.txt \
-output /user/asa/results
```

Shell Script for MapReduce job for All years:

```
hadoop jar /home/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.0.jar \
-files mapper.py,combiner.py,reducer.py,years.txt \
-mapper mapper.py \
-combiner combiner.py \
-reducer reducer.py \
-input /user/asa/MoviesRatings/ratings.txt \
-output /user/asa/results1
```

1.4 Moving results to the local and examining

To save and move the output files onto the local machine, I’ve used the hadoop fs -get command. This command retrieves the output files from HDFS and stores them in the specified local directory:

Get the results from HDFS for the selected set of years:

```
hadoop fs -get /user/asa/results/* selected_years_results/
```

Get the results from HDFS for the All set of years:

```
hadoop fs -get /user/asa/results1/* All_years_results/
```

2. Design

In this step the two different MapReduce design approaches one with a combiner and one without a combiner is explained and compared.

2.1 Design 1: Mapper, Combiner and Reducer

The mapper reads the input data and emits genre title pairs along with the corresponding rating and a count of 1 for each movie. The keys are the genre title pairs and the values are tuples containing the rating and the count. The combiner then groups the data by genre and title and calculates the sum and count of ratings for each movie,Reducing the total volume of data that I being shuffled across the network.

By using a combiner the amount of data that is transferred between the mappers and reducers are minimized,As an intermediate the results are pre aggregated before being sent to the reducers.Also here the reducer is set to only accept and aggregate movies with atleast a minimum of 15 individual ratings to them as a special condition to avoid incorrect analysis and to produce accurate results according to the dataset ratings.txt.This design results in a more efficient solution,As it reduces network traffic,processing time and resource utilization.The maximum number of different reducers that could be used in this design depends on the number of unique genre title pairs in the input data.

2.2 Design 2: Mapper and Reducer

The mapper in this design is the same as in the Design 1,Emitting genre title pairs along with the corresponding rating and count of 1 for each movie.However,without a combiner the intermediate output is directly shuffled and sent to the reducers.This increases the volume of data moved across the network as the mappers emit data for every single movie record which can potentially lead to higher network traffic, processing time and resource utilization.

The reducer takes the intermediate key value pairs from the mappers groups them by the key,calculates the average rating for each genre title pair and identifies the highest rated movie for each genre.The maximum number of different reducers that could be used in this design is also determined by the number of unique genre title pairs from the input data.

2.3 Comparison and Potential Improvements

In this specific program the absence of a combiner may not have a significant impact on performance since the dataset is relatively small.However for a larger dataset using a combiner can lead to significant improvements in efficiency by reducing the data volume shuffled between mappers and reducers.

Both the designs can still be optimized further.For example,the use of efficient data structures and algorithms can help improve the overall performance of the MapReduce job. Also the use of partitioners can help distribute the data more evenly among reducers,Which in turn can help prevent workload imbalances.

In summary,While the current solution is effective for the task at hand there are still room for improvements. If a more optimal solution is desired further investigation into alternative approaches,Such as implementing a custom partitioner or utilizing more efficient algorithms may help in yield of better performance.

3. Distributed Computation

In this step we consider the implications of using a distributed computation cluster such as Condor instead of Hadoop for processing a Hypothetical petabyte sized dataset.

3.1 Pros of using a computational cluster

Flexible resource allocation: Condor's resource management system can efficiently allocate tasks to underutilized machines in a cluster,This dynamic allocation leads to an overall better resource utilization compared to Hadoop,Which relies on a fixed number of mappers and reducers.

Scalability: Condor supports the ability to scale across multiple machines and manage a large number of tasks with ease.This scalability is especially important when dealing with petabyte scale datasets.

Fault tolerance: Condor provides fault tolerance by employing mechanisms such as checkpointing and process migration.These features ensures that tasks can recover from failures and continue execution that is similar to Hadoop's fault tolerance capabilities.

3.2 Cons of using a computational cluster

No built-in support for MapReduce: Unlike Hadoop,Condor does not have native support for the MapReduce paradigm.This lack of support may require additional development effort to create a custom solution for processing the data using a MapReduce like workflow.

Less integrated with data storage: Hadoop has the Hadoop Distributed File System (HDFS) as an integrated distributed file system specifically designed to work with MapReduce jobs. In contrast,Condor would require additional setup and configuration for managing large scale distributed data storage,Which might involve integrating it with external storage systems.

Considering these factors,Hadoop seems to be a more suitable choice for processing a petabyte sized dataset, as it provides an integrated solution for data storage and processing with native support for MapReduce jobs. Hadoop's architecture is designed to handle large-scale data processing tasks efficiently with features like data locality and parallel processing.

On the other hand,Using a computational cluster like Condor would require implementing a custom MapReduce like solution and handling large scale distributed data storage management. While Condor's flexibility and scalability are advantageous these benefits may be outweighed by the additional development effort required and the complexity involved in building a custom solution.

In conclusion,When analyzing a petabyte sized dataset,Hadoop offers a more suitable architecture,As it provides an integrated platform for data storage and processing with built in support for the MapReduce paradigm. Although Condor has its own advantages such as the flexible resource allocation and scalability,The additional development effort required to create a custom data processing solution and manage distributed data storage may outweigh these benefits.

4. Implementation

4.1 Brief

In this section, The implementation of the MapReduce solution is explained which includes using 2 designs,Design 1 Mapper,Combiner,Reducer,Where a combiner is used for efficiency improvements and Design 2 Mapper,Reducer.Also the use of schell script to run the job.

Here I've considered two designs for the MapReduce solution:

Design 1: Mapper, Combiner and Reducer

Design 2: Mapper and Reducer Only

We chose to implement Design 1,Which consists a mapper, combiner and a reducer. This design is favorable because it offers better efficiency by minimizing the data shuffled between mappers and reducers.

To test the implementation i've utilized the simhadoop.sh script in a terminal on the local machine. This script emulates the behavior of a MapReduce job on Hadoop and generates intermediary files for the mapper output (mapout5.txt), combiner output (comout5.txt) and the reducer output (results5.txt).

Modifying the provided Python code:

The provided Python code was a simple word count program that needed significant modifications to meet the desired requirements. I started by modifying the mapper to read input data and emit genre title pairs along with the corresponding rating and a count of 1 for each movie. This modification was essential for calculating the average rating for each movie.

Developing a custom Combiner:

Once the mapper and reducer were working correctly,The focus was on improving the solution by creating a custom combiner.The custom combiner groups the data by genre and title and calculates the sum and count of ratings for each movie. This step reduces the amount of data shuffled between mappers and reducers leading to an overall better performance.

Testing with simhadoop.sh:

Using the simhadoop.sh script the code was tested in the terminal on my local machine. The script emulates the MapReduce process on Hadoop and generates intermediary files (mapout5.txt, comout5.txt and results5.txt) for the mapper output, combiner output and reducer output respectively.

Final submission and output:

After ensuring the implementation's correctness and efficiency,The final code was submitted on the Hadoop server to analyse the ratings.txt file.Test was run for both the full and an empty version of the years.txt file. The results are submitted alongside the Python code.

4.2 Python Files

Design 1(Mapper,Combiner,Reducer)

Mapper:

```
#!/usr/bin/env python

import sys

def load_years(filename):
    years = set()
    with open(filename) as f:
        for line in f:
            years.update(line.strip().split())
    return years

years = load_years("years.txt")
all_years = not bool(years)

for line in sys.stdin:
    uid, title, genres, year, rating = line.strip().split('\t')
    if all_years or year in years:
        for genre in genres.split('|'):
            print(f"{genre}\t{title}|{rating}\t1")
```

Combiner:

```
#!/usr/bin/env python

import sys

current_genre = None
current_title_ratings = {}

for line in sys.stdin:
    genre, title_rating, count = line.strip().split('\t')
    title, rating = title_rating.split('|')
    rating, count = float(rating), int(count)

    if current_genre == genre:
        if title in current_title_ratings:
            current_title_ratings[title].append(rating)
        else:
            current_title_ratings[title] = [rating]
    else:
        if current_genre:
            for title, ratings in current_title_ratings.items():
                avg_rating = sum(ratings) / len(ratings)
                print(f"{current_genre}\t{title}|{avg_rating}\t{len(ratings)}")
        current_genre = genre
        current_title_ratings = {title: [rating]}

if current_genre:
    for title, ratings in current_title_ratings.items():
        avg_rating = sum(ratings) / len(ratings)
        print(f"{current_genre}\t{title}|{avg_rating}\t{len(ratings)}")
```

Reducer:

```
#!/usr/bin/env python

import sys

current_genre = None
current_title = None
current_sum = 0
current_count = 0
highest_avg_rating = 0
MIN_RATING_COUNT = 15
```

```
for line in sys.stdin:
    genre, title_rating, rating_count = line.strip().split('\t')
    title, avg_rating = title_rating.split('|')
    avg_rating = float(avg_rating)
    rating_count = int(rating_count)

    if current_genre == genre:
        if rating_count >= MIN_RATING_COUNT:
            current_sum += avg_rating * rating_count
            current_count += rating_count

            if avg_rating > highest_avg_rating:
                highest_avg_rating = avg_rating
                current_title = title
    else:
        if current_genre and current_title:
            print(f"{current_genre}\t{current_title}\t{highest_avg_rating:.1f}")

        current_genre = genre
        if rating_count >= MIN_RATING_COUNT:
            current_title = title
            current_sum = avg_rating * rating_count
            current_count = rating_count
            highest_avg_rating = avg_rating
        else:
            current_title = None
            current_sum = 0
            current_count = 0
            highest_avg_rating = 0

if current_genre and current_title:
    print(f"{current_genre}\t{current_title}\t{highest_avg_rating:.1f}")
```

Design 2 (Mapper,Reducer)

Mapper:

The mapper will remain the same,Parsing the input and outputting the genre, title, and rating.

Reducer:

In this case the reducer this version of the reducer now calculates the average rating using the sum and count of ratings for each title and it still checks whether a movie has at least 15 ratings.

```
#!/usr/bin/env python

import sys

current_genre = None
current_title = None
current_sum = 0
current_count = 0
highest_avg_rating = 0
MIN_RATING_COUNT = 15

for line in sys.stdin:
    genre, title_rating, rating_count = line.strip().split('\t')
    title, rating = title_rating.split('|')
    rating = float(rating)
    rating_count = int(rating_count)

    if current_genre == genre:
        if current_title == title:
            current_sum += rating
            current_count += rating_count
        else:
            if current_count >= MIN_RATING_COUNT:
                avg_rating = current_sum / current_count
                if avg_rating > highest_avg_rating:
                    highest_avg_rating = avg_rating
                    highest_rated_title = current_title

            current_title = title
            current_sum = rating
            current_count = rating_count
    else:
        if current_genre and current_title and current_count >= MIN_RATING_COUNT:
            print(f"{current_genre}\t{highest_rated_title}\t{highest_avg_rating:.1f}")

        current_genre = genre
        current_title = title
        current_sum = rating
        current_count = rating_count
        highest_avg_rating = 0

if current_genre and current_title and current_count >= MIN_RATING_COUNT:
    print(f"{current_genre}\t{current_title}\t{highest_avg_rating:.1f}")
```

4.3 Hadoop Outputs

Output for selected years:

```
Action	Fight Club	4.3
Adventure	Spirited Away (Sen to Chihiro no kamikakushi)	4.2
Animation	Ghost in the Shell (Kôkaku kidôtai)	4.1
Children	Toy Story 3	4.1
Comedy	Pulp Fiction	4.2
Crime	Shawshank Redemption, The	4.4
Documentary	Hoop Dreams	4.3
Drama	Shawshank Redemption, The	4.4
Fantasy	Spirited Away (Sen to Chihiro no kamikakushi)	4.2
Film-Noir	Miller's Crossing	4.2
Horror	Silence of the Lambs, The	4.2
IMAX	Dark Knight, The	4.2
Musical	Dancer in the Dark	4.0
Mystery	Usual Suspects, The	4.2
Romance	Amelie (Fabuleux destin d'Amélie Poulain, Le)	4.2
Sci-Fi	Matrix, The	4.2
Thriller	Fight Club	4.3
War	Schindler's List	4.2
Western	Lone Star	4.2
```

Output for All years:

```
Action	Once Upon a Time in the West (C'era una volta il West)	4.3
Adventure	Lawrence of Arabia	4.3
Animation	Grave of the Fireflies (Hotaru no haka)	4.2
Children	Toy Story 3	4.1
Comedy	Philadelphia Story, The	4.3
Crime	Shawshank Redemption, The	4.4
Documentary	Hoop Dreams	4.3
Drama	Streetcar Named Desire, A	4.5
Fantasy	Princess Bride, The	4.2
Film-Noir	Sunset Blvd. (a.k.a. Sunset Boulevard)	4.3
Horror	Rosemary's Baby	4.2
IMAX	Dark Knight, The	4.2
Musical	Singin' in the Rain	4.1
Mystery	Rear Window	4.3
Romance	Sunset Blvd. (a.k.a. Sunset Boulevard)	4.3
Sci-Fi	Logan	4.3
Thriller	Fight Club	4.3
War	Ran	4.4
```

Western	Once Upon a Time in the West (C'era una volta il West)	4.3