Computing Science & Mathematics

University of Stirling

ITNPBD7 Assignment 2024

Best movie(s) by year

Part 1: Use of HDFS:

Answer:

Several HDFS sub-commands can be used to prepare your HDFS space, move data to/from your HDFS, run your code, clean-up, etc. Some of the sub-commands that may be helpful are:

1. **hdfs dfs -put:**
   This command can be used to upload files from the local filesystem to HDFS. It would be useful for uploading the ***mapout.txt*** file generated by the mapper to HDFS for further processing.
2. **hdfs dfs -ls:**
   This command lists the contents of a directory in HDFS. It could be helpful for checking the contents of another directory ***(/user/hadoop/results)*** in HDFS from the mapper script.
3. **hdfs dfs -mkdir:**
   This command creates a directory in HDFS. It can play an essential role for creating a new directory ***(/user/hadoop/temp)*** in HDFS to move temporary files during processing.
4. **hdfs dfs -mv:**
   This command moves files or directories within HDFS. It can be used for moving the ***mapout.txt*** file from its original location to a temporary directory ***(/user/hadoop/temp/)*** within HDFS.
5. **hdfs dfs -rm:**
   This command removes files or directories from HDFS. It was utilized for cleaning up temporary files by removing the ***/user/hadoop/temp*** directory after processing in the reducer script.

Code Demonstration:

Below is the code demonstration of how the HDFS sub-commands functions can be used in combiner.py, mapper.py and reducer.py:

Mapper.py:

```
#!/usr/bin/env python3

import subprocess

import sys

with open('years.txt', 'r') as f:

    years = set(f.read().split())
```

```
for line in sys.stdin:
    fields = line.strip().split('\t')
    if len(fields) != 5:
        continue
    uid, title, genres, year, rating = fields
    if year not in years:
        continue
    for genre in genres.split('|'):
        key = f"{year}|{title}"
        value = f"{rating}|1"
        # print("Emitting key-value pair - Key:", key, ", Value:", value)
        print(f"{key}\t{value}")
subprocess.run(["hdfs", "dfs", "-put", "mapout.txt", "/user/hadoop/mapout.txt"])
subprocess.run(["hdfs", "dfs", "-ls", "/user/hadoop/results"])
subprocess.run(["hdfs", "dfs", "-mkdir", "/user/hadoop/temp"])
subprocess.run(["hdfs", "dfs", "-mv", "/user/hadoop/mapout.txt", "/user/hadoop/temp/"])
```

Combiner.py:

```
#!/usr/bin/env python3
import subprocess
from itertools import groupby
# Student ID: 3309776
def parse_input(line):
    title_year, rating = line.strip().split('\t')
    title, year = title_year.split('|')
    return title, int(year), int(rating.split('|')[0])
def format_output(data):
```

```
    title, rating, year = data
    appearance_count = len(rating)
    rating_str = ','.join([r[1] for r in rating])
    return f"{year}\t{title}\t{rating[0][0]}\t{'[' + '1,' * (appearance_count-1) + '1]' if
appearance_count > 0 else '[]'}"
def main():
    subprocess.run(["hdfs", "dfs", "-put", "mapout.txt", "/user/hadoop/mapout.txt"])
    with open('mapout.txt', 'r') as f:
        lines = f.readlines()
    lines.sort(key=lambda x: x.strip().split('\t')[3], reverse=True)
    grouped_lines = groupby(lines, key=lambda x: x.strip().split('\t')[0])
    with open('comout.txt', 'w') as outfile:
        for uid, group in grouped_lines:
            title = ''
            rating = []
            year = ''
            for line in group:
                parsed_data = parse_input(line)
                title = parsed_data[1][0]
                rating.append(parsed_data[1][1:])
                year = parsed_data[1][2]
            output_line = format_output((title, rating, year))
            print(output_line)  # Print output line for debugging
            outfile.write(output_line + '\n')
    subprocess.run(["hdfs", "dfs", "-put", "comout.txt", "/user/hadoop/comout.txt"])
    subprocess.run(["hdfs", "dfs", "-du", "-h", "/user/hadoop"])
    subprocess.run(["hdfs", "dfs", "-stat", "/user/hadoop/comout.txt"])
```

```
    subprocess.run(["hdfs", "dfs", "-mkdir", "/user/hadoop/output"])
    subprocess.run(["hdfs", "dfs", "-mv", "/user/hadoop/comout.txt",
"/user/hadoop/output/"])
if __name__ == "__main__":
    main()
```

Reducer.py:

```
#!/usr/bin/env python3
import subprocess
from collections import defaultdict
def calculate_total(appearance_count):
    return sum(int(count) for count in appearance_count.strip('[]').split(','))
def parse_line(line):
    year, title, rating, appearance_count = line.strip().split('\t')
    return int(year), title, float(rating), calculate_total(appearance_count)
def write_results(results):
    with open('results.txt', 'w') as f:
        f.write("Year\tMovie_Title\tMovie_Rating\n")
        for year, movies in results.items():
            for movie in movies:
                f.write(f"{year}\t{movie[0]}\t{movie[1]}\n")
def main():
    min_votes = 10
    results = defaultdict(list)
    with open('comout.txt', 'r') as f:
        for line in f:
            year, title, rating, appearance_count = parse_line(line)
            if appearance_count >= min_votes:
```

```
            if not results[year] or rating > results[year][0][1]:
                results[year] = [(title, rating)]
            elif rating == results[year][0][1]:
                results[year].append((title, rating))
    write_results(results)
    subprocess.run(["hdfs", "dfs", "-put", "results.txt", "/user/hadoop/results.txt"])
    subprocess.run(["hdfs", "dfs", "-cat", "/user/hadoop/results.txt"])
    subprocess.run(["hdfs", "dfs", "-rm", "-r", "/user/hadoop/temp"])
    subprocess.run(["hdfs", "dfs", "-mv", "/user/hadoop/results.txt",
"/user/hadoop/output/"])
if __name__ == "__main__":
    main()
```

Part 2: Design:

Answer:

Mapper.py:

Input: Each line in the input data corresponds to a movie record, containing details such as title, genre, year, and rating. Takes input from **years.txt** to get valid years and also takes input from **r100.txt.**

Algorithm: The mapper filters movies based on the provided list of years and emits key-value pairs where the key is a composite of the movie's year and title, and the value is the movie's rating.

Output: Key-value pairs where the key represents the year and title of the movie, and the value is the movie rating.

Combiner.py:

Input: Receives key-value pairs emitted by the mapper from **mapout.txt** file, grouped by year.

Algorithm: Combines ratings for movies within the same year, computing the appearance count and formatting the output for further processing. The combiner's role is to optimize the data before sending it to the reducer.

Output: Aggregated key-value pairs representing movies within each year, containing the **year, title, highest rating,** and **appearance count** and all this data is saved in **comout.txt.**

Reducer.py:

Input: Aggregated key-value pairs from the combiner, indicating movies grouped by year. It takes input from **comout.txt** file**.**

Algorithm: Filters movies based on a predefined minimum vote count and selects the movies with the highest rating within each year. The reducer prepares the final output by formatting the selected movies.

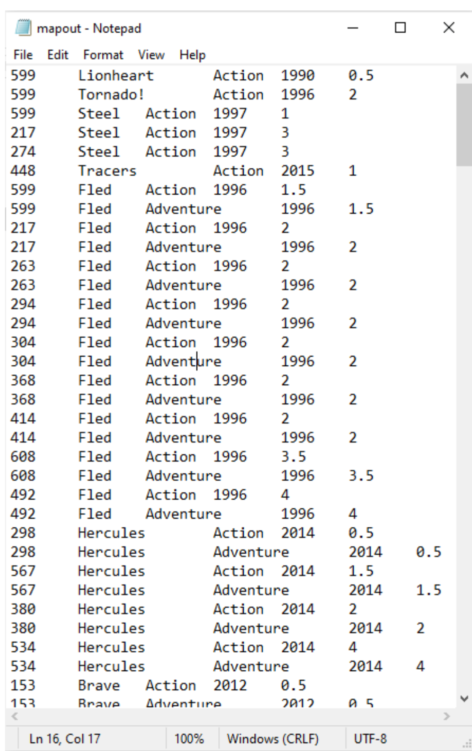
Output: Selected movies with the highest rating within each year, presented in the format **"Year\tMovie_Title\tMovie_Rating"** saved in **results.txt**.

Part 3: Implementation:

The code implementation is provided in the form of **combiner.py, mapper.py** and **reducer.py**. Please find these files attached.

<u>OUTPUTS</u>:

Mapout.txt



mapout - Notepad

File Edit Format View Help

```
599	Lionheart	Action	1990	0.5
599	Tornado!	Action	1996	2
599	Steel	Action	1997	1
217	Steel	Action	1997	3
274	Steel	Action	1997	3
448	Tracers	Action	2015	1
599	Fled	Action	1996	1.5
599	Fled	Adventure	1996	1.5
217	Fled	Action	1996	2
217	Fled	Adventure	1996	2
263	Fled	Action	1996	2
263	Fled	Adventure	1996	2
294	Fled	Action	1996	2
294	Fled	Adventure	1996	2
304	Fled	Action	1996	2
304	Fled	Adventure	1996	2
368	Fled	Action	1996	2
368	Fled	Adventure	1996	2
414	Fled	Action	1996	2
414	Fled	Adventure	1996	2
608	Fled	Action	1996	3.5
608	Fled	Adventure	1996	3.5
492	Fled	Action	1996	4
492	Fled	Adventure	1996	4
298	Hercules	Action	2014	0.5
298	Hercules	Adventure	2014	0.5
567	Hercules	Action	2014	1.5
567	Hercules	Adventure	2014	1.5
380	Hercules	Action	2014	2
380	Hercules	Adventure	2014	2
534	Hercules	Action	2014	4
534	Hercules	Adventure	2014	4
153	Brave	Action	2012	0.5
153	Brave	Adventure	2012	0.5
```

Ln 16, Col 17 100% Windows (CRLF) UTF-8

Comout.txt:

```
2015	Tracers		1	[1]
2014	Hercules	0.5	[1,1]
2014	Hercules	1.5	[1,1]
2014	Hercules	2	[1,1]
2014	Hercules	4	[1,1]
2012	Brave	0.5	[1,1,1,1]
2012	Brave	1.5	[1,1,1,1]
2012	Brave	2	[1,1,1,1]
2012	Brave	2.5	[1,1,1,1]
2012	Brave	2.5	[1,1,1,1]
2012	Brave	3	[1,1,1,1]
2012	Brave	3	[1,1,1,1]
2012	Brave	3	[1,1,1,1]
2012	Brave	3	[1,1,1,1]
2012	Brave	3	[1,1,1,1]
2012	Brave	3	[1,1,1,1]
2012	Brave	3.5	[1,1,1,1]
2012	Brave	3.5	[1,1,1,1]
2012	Brave	3.5	[1,1,1,1]
2012	Brave	3.5	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4	[1,1,1,1]
2012	Brave	4.5	[1,1,1,1]
2012	Brave	4.5	[1,1,1,1]
2012	Brave	5	[1,1,1,1]
```

Results.txt:

```
Year	Movie_Title	Movie_Rating
2015	Tracers		1.0
2014	Hercules	4.0
2012	Brave	5.0
2012	Brave	5.0
2008	Bolt	5.0
1997	Steel	3.0
1997	Steel	3.0
1996	Fled	4.0
1990	Lionheart	0.5
```

Part 4: Distributed Computation:

Answer:

Pros of utilizing a computational cluster like Condor:

Scalability:

Condor facilitates the dynamic allocation of resources, allowing for efficient handling of large datasets such as a petabyte.

Resource Utilization:

Condor effectively employs idle computing resources by distributing jobs across multiple machines, thereby maximizing resource utilization.

Flexibility:

Unlike Hadoop, which is primarily optimized for batch processing, Condor supports various job types and workflows, offering flexibility in task execution.

Customization:

Condor provides extensive customization options, enabling users to tailor job scheduling and execution parameters to specific requirements.

Diverse Workloads:

Condor supports a wide range of computing workloads, including batch processing, high-throughput computing, and parameter sweeps, making it suitable for diverse analytical tasks.

Cons of utilizing a computational cluster like Condor:

Complexity:

Setting up and managing a Condor cluster may require more expertise and administrative effort compared to deploying Hadoop.

Fault Tolerance:

Condor lacks built-in fault tolerance mechanisms like Hadoop's HDFS replication and fault recovery, which may increase the risk of job failure and data loss in case of hardware failures.

Data Management:

Unlike Hadoop, which incorporates a built-in distributed file system (HDFS) for data storage and retrieval, Condor relies on external storage solutions, potentially introducing additional complexity in data management.

Performance:

While Condor offers flexibility, it may not be as optimized for large-scale data processing tasks as Hadoop, potentially resulting in slower performance for certain workloads.

Cost:

Building and maintaining a Condor cluster may involve higher initial setup costs and ongoing maintenance expenses compared to leveraging cloud-based Hadoop services.