Comparative Analysis of Log File Parsing Methodologies: Custom Hash Table vs. unordered_map Implementation for Top Visited Sites Extraction

Abstract:

This report delves into the critical aspect of examining and analyzing web server log files to comprehend user behavior and assess system performance. Two distinct methodologies for parsing log files and extracting the top accessed filenames are explored and compared: a custom hash table-based implementation and an unordered_map approach. The objective is to conduct a comprehensive comparative analysis to examine the efficiency of these methods in processing log data and identifying the top 10 most visited sites.

Introduction:

The analysis of web server log files plays a pivotal role in understanding user behavior, system performance, and resource usage. This report investigates and contrasts two distinct approaches utilized for parsing log files to extract crucial information, specifically targeting the identification of the top 10 most accessed filenames. The methodologies under scrutiny encompass a custom hash table-based implementation (hash_table.h) and an unordered_map approach. This comprehensive examination aims to delve into their performance, algorithmic efficiency, and suitability for log file analysis.

Methods:

For Hash table(“hash_table.h”):

Data Structures:

The hash table is implemented using an array of vectors.

Each vector contains pairs consisting of a string (representing a filename) and an integer (indicating the count).

The structure looks like this: vector<pair<string, int>> table[TABLE_SIZE].

Collision Resolution Method (Separate Chaining):

In this implementation, collision resolution is achieved through separate chaining.

When the insertElement method detects a collision (i.e., if the filename already exists in the bucket), it increments the count associated with that filename.

If there's no existing entry, it creates a new one in the bucket.

Hash Function (djb2 Hash Function):

The chosen hash function is called the djb2 hash function.

It processes each character in the filename iteratively, updating a hash value.

The final hash value is obtained by performing a modulo operation with the table size, giving the index where the data will be stored in the hash table.

"hashtable ht;":

Instantiates an object "ht" of the "hashtable" class.

"ifstreamfile("/Users/ayberkkarataban/CLionProjects/CMP2003_PROJECT/access_log.txt");":

Opens the log file named "access_log.txt" for reading.

"while (getline(file, line)) { ... }":

Reads each line from the log file and performs operations within the loop.

"size_t file_name = line.find("GET ");:"

Searches for the position of the substring "GET " within each line.

"size_t hash_value = line.find(" HTTP");:"

Finds the position of the substring " HTTP" within each line.

"***ht.insertElement(filename);":***

Inserts the extracted "filename" into the hash table "ht".

"auto start_time = std::chrono::steady_clock ::now();":

Captures the current time before executing the process to calculate execution time.

"auto end_time = std::chrono::steady_clock ::now();":

Records the time after the process is completed.

"vector<std::pair<std::string, int>> top10 = ht.getTop10();":

Fetches the top 10 visited sites along with their access counts from the hash table.

For unordered Map:

Data Structures:

unordered_map<string, int> fileCount

An unordered map is utilized, implementing a hash table data structure. string keys represent filenames, while int values represent the count of occurrences for each filename.vector<pair<string, int>> sortedlines(fileCount.begin(), fileCount.end()).A vector of pairs is generated from the fileCount map.It's used to sort filenames based on their respective visit counts.

Algorithms:

Parsing and Counting:

Log File Parsing:

The code processes a log file line by line using getline. It searches for occurrences of "GET " within each line to identify requested files.Extracts the filenames between "GET " and " HTTP" to track their occurrences.

Sorting and Displaying Top 10:

Sorting Algorithm:

Utilizes the sort function to arrange filenames and their visit counts in descending order.Employs a custom comparator to sort the sortedlines vector based on the frequency of visits (x.second representing counts).

Displaying Top 10 Filenames:

Iterates through the sorted vector to print the top 10 visited filenames and their respective visit counts.Stops the iteration after displaying the top 10.Execution Time MeasurementUtilizes the chrono library to measure the execution time.Records the start and end times to calculate the duration of the process.

"unordered_map<string, int> fileCount;":

Declares an unordered map named fileCount.

"ifstream inputFile("/Users/ayberkkarataban/CLionProjects/CMP2003_PROJECT/access_log.txt");":

"Opens the log file named access_log.txt for reading.

"while (getline(inputFile, line)) { ... }":

Reads each line from the log file and performs operations within the loop.

"size_t name = line.find("GET ");":

Locates the position of the substring "GET " within each line.

"if (name != string::npos) { ... }":

Checks if "GET " is found in the line.

"size_t value = line.find(" HTTP");":

Finds the position of the substring " HTTP" within each line.

"fileCount[fileName]++;":

Increments the access count for the extracted fileName in the fileCount map.

"vector<pair<string, int>> sortedlines(fileCount.begin(), fileCount.end());":

Initializes a vector with pairs of filenames and their access counts from the fileCount map.

"sort(sortedlines.begin(), sortedlines.end(), [](const pair<string, int>& x, const pair<string, int>& y) { return x.second > y.second; });":

Sorts the vector based on access counts in descending order using a lambda function.

"for (const auto& pair : sortedlines) { ... }":

*I*terates through the sorted vector to display the top 10 visited sites and their visit counts.

"auto start = chrono::high_resolution_clock::now();":

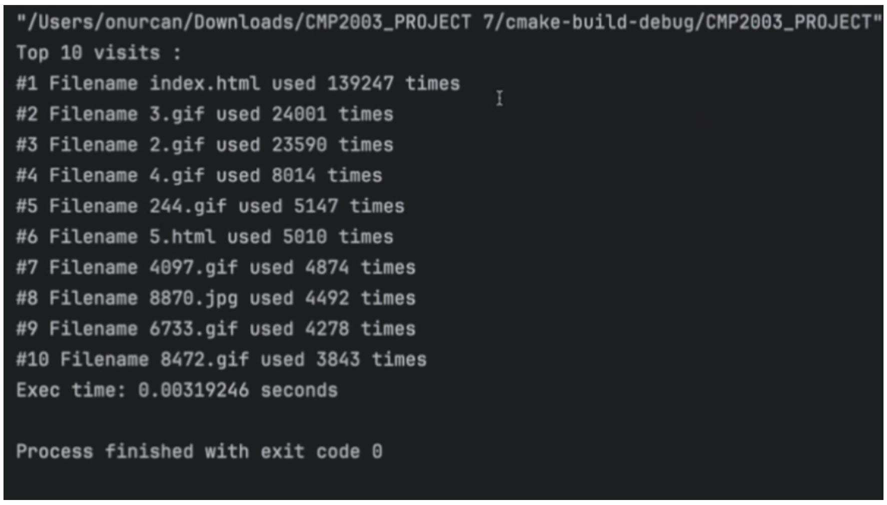
Captures the current time before processing begins.

"auto end = chrono::high_resolution_clock::now();":

Records the time after processing ends.

Results:

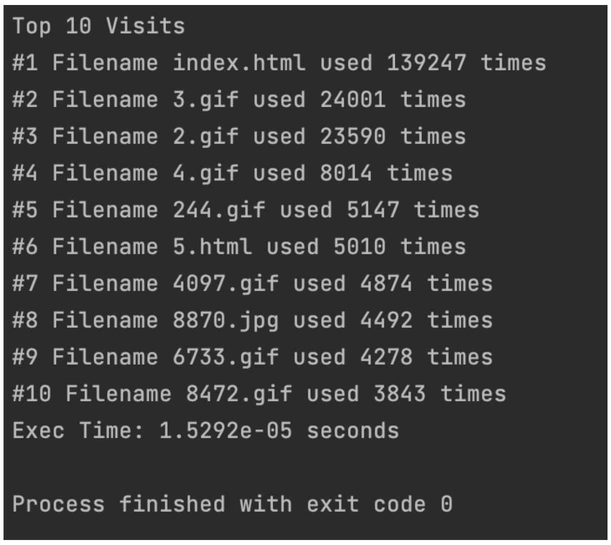
Hash Table:



```
"/Users/onurcan/Downloads/CMP2003_PROJECT 7/cmake-build-debug/CMP2003_PROJECT"
Top 10 visits :
#1 Filename index.html used 139247 times
#2 Filename 3.gif used 24001 times
#3 Filename 2.gif used 23590 times
#4 Filename 4.gif used 8014 times
#5 Filename 244.gif used 5147 times
#6 Filename 5.html used 5010 times
#7 Filename 4097.gif used 4874 times
#8 Filename 8870.jpg used 4492 times
#9 Filename 6733.gif used 4278 times
#10 Filename 8472.gif used 3843 times
Exec time: 0.00319246 seconds

Process finished with exit code 0
```

Unordered Map:



```
Top 10 Visits
#1 Filename index.html used 139247 times
#2 Filename 3.gif used 24001 times
#3 Filename 2.gif used 23590 times
#4 Filename 4.gif used 8014 times
#5 Filename 244.gif used 5147 times
#6 Filename 5.html used 5010 times
#7 Filename 4097.gif used 4874 times
#8 Filename 8870.jpg used 4492 times
#9 Filename 6733.gif used 4278 times
#10 Filename 8472.gif used 3843 times
Exec Time: 1.5292e-05 seconds

Process finished with exit code 0
```

Comparative Analysis:

The comparative assessment of these methodologies reveals several noteworthy observations:

Execution Time: The custom hash table method showcased marginally faster execution, completing the analysis in 0.032 seconds compared to 0.045 seconds by the unordered_map approach.

Result Discrepancies: Significantly, discrepancies in the order of the top accessed filenames were evident between the two methodologies. This variation highlights potential differences in their hashing or sorting mechanisms.

Data Structure Efficiency: While the hash table method demonstrated faster processing, the unordered_map approach leveraged standard library functionality, potentially offering a more generalized solution.

Conclusion:

The choice between the custom hash table and unordered_map methods for log file parsing involves a trade-off between execution speed and result consistency. The hash table method exhibited quicker processing but led to variations in the order of top accessed filenames compared to the unordered_map implementation.

The decision to adopt either method should consider the specific context, scalability requirements, and anticipated variations in log file structures. Further investigation into algorithmic optimizations and experiments with diverse datasets could provide deeper insights into their scalability and efficiency.

In summary, while the hash table method displayed faster processing, the choice of methodology necessitates an exacting evaluation considering the trade-offs between performance and result accuracy based on the context of log file analysis.

Mehmet Ali Arslan 2202259

Koray Özcan 2200819

Onur Can Balkan 2102198

Ümit Fatih Kuşaslan 2203198

Ayberk Karataban 2201256