Thomas FAVOLI

README – DATASTORE :

This document provides important information and guidelines for using the "datastore" program. Please read this carefully to ensure smooth operation and efficient handling of data.

Setting Parameters:

Before running the "datastore" program, make sure to set the following parameters in the file:

- **HASH_SIZE:** This parameter defines the size of the hash table used by the program. It is recommended to increase this value as the size of the file increases to optimize performance.

- **MAX_SIZE:** The "discount_count" endpoint relies on this parameter, which should be adjusted if the count exceeds 100,000. Increasing it as needed will prevent potential data overflow issues.

- **FILE_SIZE:** This parameter represents the number of lines in the input file. Including this parameter is necessary to manage memory allocation, especially on limited storage space (which was the case with my computer. I had to limit the size of the input).

- **MINIMUM_YEAR:** Specify the year corresponding to the lowest year in the input file. This information is crucial for processing data accurately.

- **Filename:** Provide the path to the desired input file that contains the data points for the program.

PS : I suppose that there are only 14 different events from the data I received.

Running the Program:

Follow the steps below to compile and run the "datastore" program:

1. Compile the program using GCC:

gcc -o datastore datastore.c

2. Execute the compiled program:

./datastore

Program Commands:

While running the "datastore" program, you can use the following commands:

/q ? : To quit and exit the program.

`exists` - whether a data point for this user_id and event_type exists
- `event` - mandatory
- `user_id` - mandatory

/exists?event=LEFT_SWIPE&user_id=9734233544

/exists?event=MOUSE_DOWN&user_id=9734233544

`count` - simply count the number of events after filtering
- `date_from` - mandatory
- `date_to` - mandatory
- `event` - optional (if not desired still write na)
- `user_id` - optional (if not desired it is not necessary to write anything).

/count?date_from=2022-01-07T00:26:27&date_to=2022-01-07T03:03:38

/count?date_from=2022-01-07T00:26:27&date_to=2022-01-07T03:03:38&event=RESIZE

/count?date_from=2022-01-07T00:26:27&date_to=2022-01-07T03:03:38&event=TAP

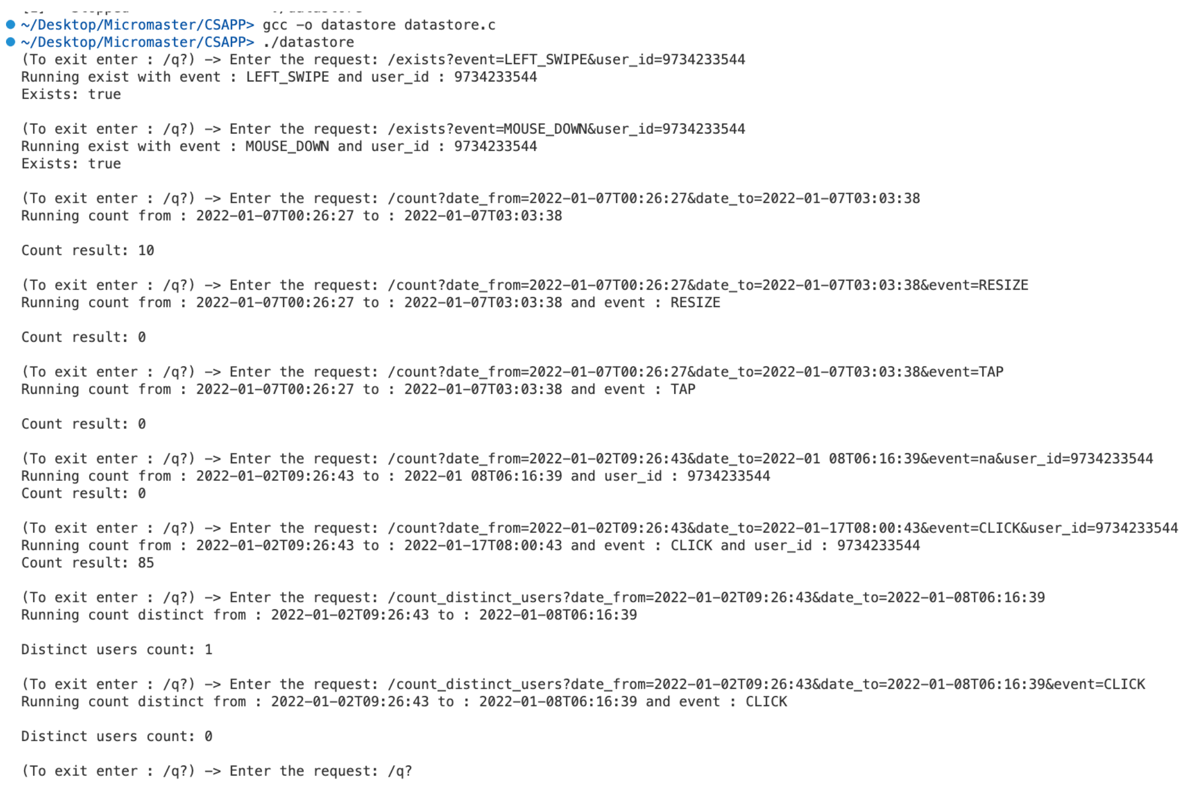
/count?date_from=2022-01-02T09:26:43&date_to=2022-01 08T06:16:39&event=na&user_id=9734233544

/count?date_from=2022-01-02T09:26:43&date_to=2022-01-17T08:00:43&event=CLICK&user_id=9734233544

`count_distinct_users` - get the number of unique user_ids recorded
- Possible params:
  - `date_from` - mandatory
  - `date_to` - mandatory
  - `event` - optional (if not desired it is not necessary to write anything).

/count_distinct_users?date_from=2022-01-02T09:26:43&date_to=2022-01-08T06:16:39

/count_distinct_users?date_from=2022-01-02T09:26:43&date_to=2022-01-08T06:16:39&event=CLICK



```
● ~/Desktop/Micromaster/CSAPP> gcc -o datastore datastore.c
● ~/Desktop/Micromaster/CSAPP> ./datastore
  (To exit enter : /q?) -> Enter the request: /exists?event=LEFT_SWIPE&user_id=9734233544
  Running exist with event : LEFT_SWIPE and user_id : 9734233544
  Exists: true

  (To exit enter : /q?) -> Enter the request: /exists?event=MOUSE_DOWN&user_id=9734233544
  Running exist with event : MOUSE_DOWN and user_id : 9734233544
  Exists: true

  (To exit enter : /q?) -> Enter the request: /count?date_from=2022-01-07T00:26:27&date_to=2022-01-07T03:03:38
  Running count from : 2022-01-07T00:26:27 to : 2022-01-07T03:03:38

  Count result: 10

  (To exit enter : /q?) -> Enter the request: /count?date_from=2022-01-07T00:26:27&date_to=2022-01-07T03:03:38&event=RESIZE
  Running count from : 2022-01-07T00:26:27 to : 2022-01-07T03:03:38 and event : RESIZE

  Count result: 0

  (To exit enter : /q?) -> Enter the request: /count?date_from=2022-01-07T00:26:27&date_to=2022-01-07T03:03:38&event=TAP
  Running count from : 2022-01-07T00:26:27 to : 2022-01-07T03:03:38 and event : TAP

  Count result: 0

  (To exit enter : /q?) -> Enter the request: /count?date_from=2022-01-02T09:26:43&date_to=2022-01 08T06:16:39&event=na&user_id=9734233544
  Running count from : 2022-01-02T09:26:43 to : 2022-01 08T06:16:39 and user_id : 9734233544
  Count result: 0

  (To exit enter : /q?) -> Enter the request: /count?date_from=2022-01-02T09:26:43&date_to=2022-01-17T08:00:43&event=CLICK&user_id=9734233544
  Running count from : 2022-01-02T09:26:43 to : 2022-01-17T08:00:43 and event : CLICK and user_id : 9734233544
  Count result: 85

  (To exit enter : /q?) -> Enter the request: /count_distinct_users?date_from=2022-01-02T09:26:43&date_to=2022-01-08T06:16:39
  Running count distinct from : 2022-01-02T09:26:43 to : 2022-01-08T06:16:39

  Distinct users count: 1

  (To exit enter : /q?) -> Enter the request: /count_distinct_users?date_from=2022-01-02T09:26:43&date_to=2022-01-08T06:16:39&event=CLICK
  Running count distinct from : 2022-01-02T09:26:43 to : 2022-01-08T06:16:39 and event : CLICK

  Distinct users count: 0

  (To exit enter : /q?) -> Enter the request: /q?
```

<u>Program conception :</u>

We are faced with a substantial amount of data that requires analysis, and our focus will be on implementing two key operations: insertion and search (endpoints).

Deletion operations are unnecessary since the 'insert' method will ensure that only unique Nodes (characterized by a distinct combination of user_id, timestamp, and event) are inserted into the structure.

Consequently, each unique event will be represented as a node within a linked list. The linked list proves to be convenient for efficient insertion and sequential addition for specific dates with various possible aggregations.

To optimize time during comparisons, we convert the timestamp and event values to long data types.

However, the linked list is less suitable for checking the uniqueness of an event or for efficiently performing the exist method.
To address this, I propose utilizing a hash table with separate chaining. Each node (line in the data file) will be associated with two hashes: the hash_exist, which represents the entry in the hash table, and a unique hash for determining the uniqueness and existence of a node.

```c
struct node {
    long user_id;          // Data: Represents a user ID (assuming it's a long integer).
    long timestamp;        // Data: Represents a timestamp (assuming it's a long integer).
    int event;             // Data: Represents an event (assuming it's an integer).
    int hash_unique;       // Data: Represents a value for hash table unique purposes (assuming it's an integer).
    int hash_exist;        // Data: Represents a value for hash table existence purposes (assuming it's an integer).
    struct node *next;     // Pointer to the next node in the linked list.
};
```

It is worth noting that I opted for a straightforward hash function, as the modulo operation provided uniformly distributed inputs across the hash table.

Furthermore, for improved traversal, each node in the separate chaining approach will be cast into a unique hash_node.

```c
struct hash_node {
    int hash_unique;            // Data: Represents a unique value for hash table purposes (assuming it's an integer).
    struct node* node_ptr;     // Pointer to a node in the linked list associated with this hash entry.
    struct hash_node *next;    // Pointer to the next node in the hash table bucket (linked list of hash nodes).
};
```

The main function is responsible for initializing both the linked list and the hash table.

Subsequently, data processing commences. To process the data, we first verify the uniqueness of a node using the hash_table. If it is unique, we proceed to insert it into both the linked list and the hash table.

The run function efficiently handles input parsing and dispatches the data to the appropriate method.

Methods such as count and distinct count perform traversals on the linked list, while the exist method utilizes the hash table to optimize processing time.

Finally, to release the allocated memory resources, we perform deallocation for both the linked list and the hash table.

In conclusion, the combination of a linked list for nodes and a hash table with chaining offers an optimal trade-off between space-time efficiency. This data structure choice ensures effective handling of our data analysis requirements.

<u>Testing :</u>

During testing, I carefully selected specific variables to assess the functionality of the program. Please note that the actual test function is not included in the C file but was used externally to evaluate the program's behavior.

When conducting the test, I considered the following:

- **Data Structure Choice:** I evaluated the efficiency of the chosen data structure, which involves a combination of linked list and hash table with separate chaining. The linked list allows for sequential data addition, while the hash table facilitates quick checks for uniqueness and existence of events.

- **Handling Unique Data:** As the 'insert' method ensures only unique nodes (comprising a unique combination of user_id, timestamp, and event) are inserted, I verified if the program successfully identified and handled duplicates, avoiding redundant data.

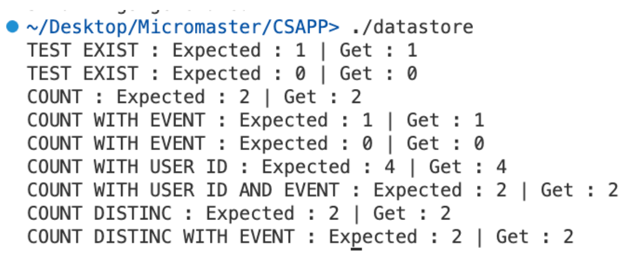
- **Hash Function and Hash Table Performance:** I examined the performance of the hash function and the hash table with separate chaining. I assessed whether the chosen hash function provided an even spread of inputs across the hash table, minimizing collisions and improving search time.

- **Data Processing:** During the test, I focused on data processing efficiency. I checked how well the program managed to insert unique data points into the linked list and hash table simultaneously.

- **Aggregation Queries:** I evaluated the program's ability to respond to aggregation queries effectively. Specifically, I assessed the efficiency of methods like count and distinct count, which required traversing the linked list.

- **Existence Check:** To ensure optimized search operations, I verified how the program utilized the hash table for the existence check, effectively saving processing time.

- **Memory Management:** I examined whether the program handled memory efficiently by freeing memory after data processing was complete, avoiding memory leaks.



```
● ~/Desktop/Micromaster/CSAPP> ./datastore
  TEST EXIST : Expected : 1 | Get : 1
  TEST EXIST : Expected : 0 | Get : 0
  COUNT : Expected : 2 | Get : 2
  COUNT WITH EVENT : Expected : 1 | Get : 1
  COUNT WITH EVENT : Expected : 0 | Get : 0
  COUNT WITH USER ID : Expected : 4 | Get : 4
  COUNT WITH USER ID AND EVENT : Expected : 2 | Get : 2
  COUNT DISTINC : Expected : 2 | Get : 2
  COUNT DISTINC WITH EVENT : Expected : 2 | Get : 2
```