Solution: Logger Rate Limiter

Let's solve the Logger Rate Limiter problem using the Hash Map pattern.

We'll cover the following

Statement

For the given stream of message requests and their timestamps as input, you must implement a logger rate limiter system that decides whether the current message request is displayed. The decision depends on whether the same message has already been displayed in the last S seconds. If yes, then the decision is FALSE, as this message is considered a duplicate. Otherwise, the decision is TRUE.

> **Note:** Several message requests, though received at different timestamps, may carry identical messages.

Constraint:

- Timestamps are in ascending order.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The problem we're trying to solve here implies that there's a queue of messages with timestamps attached to them. Using this information, we can use a queue data structure to process the incoming message requests.

In addition to using a queue, we can use a set data structure to efficiently identify and remove duplicates.

Our naive approach is an event-driven algorithm, where every incoming message prompts us to identify and remove all those messages from the queue whose timestamps are more than S seconds older than the timestamp of the new message. Whenever we remove a message from the queue, we also remove it from the set.

After performing this time limit expiry check, we can be certain that *none* of the messages in the queue and in the set are more than S seconds older than the new message. Now, if this new message is present in the set,

it's a duplicate and we return FALSE.

Otherwise, we add it to the message queue and to the message set, and return TRUE.

The time and space complexity of this approach are $O(n)$, where n is the size of the message queue at any given point in time.

Optimized approach using hash maps

We need to know if a message already exists and keep track of its time limit. When thinking about such problems where two associated values need to be checked, we can use a hash map.

We can use all incoming messages as keys and their respective time limits as values. This will help us eliminate duplicates and respect the time limit of S seconds as well.

Here is how we'll implement our algorithm using hash maps:

1. Initialize a hash map.
2. When a request arrives, check if it's a new request (the message is not among the keys stored in the hash map) or a repeated request (an entry for this message already exists in the hash map). If it's a new request, accept it and add it to the hash map.
3. If it's a repeated request, compare the difference between the timestamp of the incoming request and the timestamp of the previous request with the same message. If this difference is greater than the time limit, S, accept it and update the timestamp for that specific message in the hash map. Otherwise, reject it.

Let's say we have a few message requests and their timestamps, as shown below. The time limit, ***S***, has been set as **8**. Let's check which of the following requests should be approved and which ones will be rejected.

requests	time
good morning	1
hello world	5
good morning	6
good morning	7
hello world	15

1 of 6

Let's look at the code for this solution below:

Java

```
class RequestLogger {

    // initailization of requests hash map
    private HashMap<String, Integer> requests;
    int limit;

    public RequestLogger(int timeLimit) {
        requests = new HashMap<String, Integer> ();
        limit = timeLimit;
    }
```

```
11
12      // function to accept and deny message requests
13
14      public boolean messageRequestDecision(int timestamp, String request) {
15          // checking whether the specific request exists in
16          // the hash map or not
17          if (!this.requests.containsKey(request)) {
18              this.requests.put(request, timestamp);
19              return true;
20          }
21          // if it exists, check whether its time duration
22          // lies within the defined timestamp
23          if (timestamp - this.requests.get(request) >= limit) {
24              // store this new request in the hash map, and return true
25              this.requests.put(request, timestamp);
26              return true;
27          } else {
28              // the request already exists within the timestamp
```

Solution summary

Let's summarize our optimized algorithm:

1. After initializing a hash map, whenever a request arrives, we check whether it's a new request or a repeated request after the assigned time limit
2. If the request meets either of the conditions mentioned in the above step, we accept and update the entry associated with that specific request in the hash map. Otherwise, reject the request and return the final decision.

Time complexity

The decision function checks whether a message has already been encountered, and if so, how long ago it was encountered. Thanks to the use of hash maps, both operations are completed in constant time—therefore, the time complexity of the decision function is $O(1)$.

Space complexity

The space complexity of the algorithm is $O(n)$, where n is the number of incoming requests that we store.

Back
Logger Rate Limiter

Next
Next Greater Element

Mark as Completed