362. Design Hit Counter

Medium Design Queue Array Hash Table Binary Search

Leetcode Link

Problem Description

The problem at hand is designing a system that keeps track of the number of hits (or actions/events) that occur on a resource in the last 5 minutes. Importantly, the system is expected to handle hits in a chronological order, meaning that the timestamps for these hits are always increasing. The design should consist of a hit counter that can accept a new hit at a specific timestamp, with timestamps measured in seconds, and retrieve the total count of hits that occurred in the past 5 minutes from any given timestamp.

To summarize, the `HitCounter` class should provide two primary functionalities:

1. `hit(int timestamp)`: Record a new hit at the given timestamp.
2. `getHits(int timestamp)`: Fetch the number of hits in the last 300 seconds (5 minutes) from the given timestamp.

This counter could be used, for example, on a website to track the number of hits to a server, on an application to monitor the number of occurrences of a particular event, or any similar use-case where such time-bound counting is required.

Intuition

When considering how to design the `HitCounter`, one must realize that while we need to keep track of hits, storing every single hit with its timestamp is not efficient, especially as time progresses. We only care about hits in the last 5 minutes, not beyond that. Therefore, we must handle hits in such a way that avoids unnecessary memory use and ensures efficient retrieval.

The intuition behind the solution is to use a data structure that can record the hits while allowing fast insertion and look-up. A Python `Counter` object can handle the accumulation of hits well since it uses a hash table under the hood. Each hit is associated with a timestamp, and this timestamp acts as key in the `Counter`. The corresponding value is the number of hits that occurred at that timestamp.

The `hit` method updates the counter for the given timestamp. Multiple hits at the same timestamp simply increment the count for that timestamp.

For the `getHits` method, the goal is to calculate the sum of all the hits that happened from `(timestamp - 300)` to `timestamp`, because these represent the hits within the last 5 minutes. The method iterates through all items in the `Counter`, but only sums those where the timestamp of the hit is within the last 300 seconds from the given `timestamp`.

In practice, the `Counter` can potentially grow large if hits aren't removed after they become irrelevant (older than 5 minutes). To improve the implementation, one might periodically clean the `Counter` by removing old hits from it, to keep the memory footprint small and the `getHits` method efficient. However, the provided solution does not implement this optimization, and it's something users of the class would need to handle separately if needed.

Solution Approach

The implementation of the `HitCounter` class uses two primary methods as part of its API: `hit` and `getHits`.

The `Counter` data structure from Python's standard library collection is employed here to keep a tally of the hits. A `Counter` is essentially a dictionary or a hash map where keys are the items to be counted and the values are the counts. This is especially useful for our use case because it allows constant-time operations for storing and updating the number of hits per timestamp.

Let's delve into the implementation details of both methods:

###Hit Method When the `hit` method is called with a timestamp, it performs the following operations:

1. It checks if the timestamp is already present as a key in the `Counter`.
2. If it is, it increments the count associated with that timestamp.
3. If not, it creates a new entry with the timestamp as the key and sets the count to 1. Here's the snippet of the code that handles the hit:

```
self.counter[timestamp] += 1
```

The `+=` operation here is the Pythonic way to increment the value for a given key in a `Counter`. It's efficient because it does not require a search for the key before incrementation as the `Counter` handles that internally.

###GetHits Method The `getHits` method retrieves the count of hits within the past 5 minutes from a given timestamp and it operates as follows:

1. It initializes a variable to hold the sum of hits.
2. It iterates over all items in the counter.
3. For each item, it checks if the timestamp of that item is within the past 300 seconds of the given current timestamp.
4. If it is, it adds the count of hits for that timestamp to the sum.
5. After iterating through all the items, it returns the sum. The following snippet from the code explains this logic:

```
return sum(t for t, v in self.counter.items() if t + 300 > timestamp)
```

Here, we use a list comprehension to iterate over the items of the `Counter`, summing up the values for the keys that meet the time condition (i.e., within the last 300 seconds). This method is straightforward and functional, but as mentioned earlier, it can become inefficient if the `Counter` grows too large due to old entries not being removed.

Overall, the pattern used in the solution is quite effective for this specific problem. However, in a real-world scenario where time efficiency for the `getHits` method becomes crucial, further optimization could be considered, like cleaning up old timestamps at regular intervals or after a certain number of hits to maintain the `counter` size.

Example Walkthrough

Let's walk through a small example to illustrate how the `HitCounter` operates. Let's assume the following sequence of hits and queries:

1. A hit at timestamp 10.
2. A hit at timestamp 20.
3. A hit at timestamp 30.
4. A query for the number of hits at timestamp 40.

Now, let's see how the `HitCounter` will handle these events:

Recording hits

- We call `hit(10)`. The `counter` is updated to `{10: 1}`.
- We call `hit(20)`. The `counter` is updated to `{10: 1, 20: 1}`.
- We call `hit(30)`. The `counter` is updated to `{10: 1, 20: 1, 30: 1}`.

At each hit, the counter records the timestamp with a count. If there were multiple hits at the same timestamp, the count would increase accordingly; however, in this example, there is only one hit per timestamp.

Query for hits

- Next, we call `getHits(40)`. We need to sum counts of hits happening from `40 - 300 = -260` (but since a timestamp cannot be negative, we effectively start at 0) to 40.
- The `counter` has three timestamps entries: 10, 20, and 30, all of which fall within the last 300 seconds (5 minutes) from timestamp 40.
- Thus, the `getHits(40)` sums up the count of hits at all current timestamps and returns `1 + 1 + 1 = 3`.

The count of hits within the last 5 minutes from timestamp 40 is three, as there are no hits outside the 5-minute window yet. With this approach, querying for hits at any point provides the total number of hits in the preceding 5 minutes in an efficient manner, taking into account only the relevant timestamps.

Python Solution

```
from collections import Counter

class HitCounter:
    def __init__(self):
        """
        Initialize the HitCounter with a Counter to keep track of timestamp occurrences.
        """
        self.hits = Counter()

    def hit(self, timestamp: int) -> None:
        """
        Record a hit at a given timestamp.
        Each hit increments the count for the specific timestamp.

        :param timestamp: The current timestamp (in seconds granularity).
        """
        self.hits[timestamp] += 1

    def get_hits(self, timestamp: int) -> int:
        """
        Retrieve the number of hits in the past 5 minutes (300 seconds) from the current timestamp.

        :param timestamp: The timestamp at which to get the number of hits.
        :return: Total number of hits in the last 5 minutes.
        """
        # Filter out the timestamps that are older than 5 minutes and sum their hit counts.
        return sum(count for time, count in self.hits.items() if time > timestamp - 300)

# Example usage
hit_counter = HitCounter()
# hit_counter.hit(timestamp)
# total_hits = hit_counter.get_hits(timestamp)
```

Java Solution

```
import java.util.HashMap;
import java.util.Map;

public class HitCounter {
    // Use a map to store the count of hits for each timestamp
    private Map<Integer, Integer> hitCounts;

    /** Constructor to initialize the HitCounter object. */
    public HitCounter() {
        hitCounts = new HashMap<>(); // Initialize the map for storing hits
    }

    /**
     * Record a hit at the given timestamp.
     *
     * @param timestamp - The current timestamp in seconds granularity.
     */
    public void hit(int timestamp) {
        // Increment the hit count for the given timestamp, if it exists; otherwise, set it to 1
        hitCounts.put(timestamp, hitCounts.getOrDefault(timestamp, 0) + 1);
    }

    /**
     * Return the number of hits in the past 5 minutes.
     *
     * @param timestamp - The current timestamp in seconds granularity.
     * @return the total number of hits in the last 5 minutes.
     */
    public int getHits(int timestamp) {
        int hits = 0; // Variable to accumulate the number of hits

        // Iterate through all entries in the map
        for (Map.Entry<Integer, Integer> entry : hitCounts.entrySet()) {
            // Check if the entry's timestamp is within the past 5 minutes from the given current timestamp
            if (timestamp - entry.getKey() < 300) { // 300 seconds equals 5 minutes
                // Sum up the hits that are within the past 5 minutes
                hits += entry.getValue();
            }
        }
        // Return the total number of hits
        return hits;
    }
}

// Usage example
// HitCounter hitCounter = new HitCounter();
// hitCounter.hit(timestamp);
// int numberOfHits = hitCounter.getHits(timestamp);
```

C++ Solution

```
#include <unordered_map>

class HitCounter {
    // Use an unordered map to store the count of hits for each timestamp
private:
    std::unordered_map<int, int> hitCounts;

public:
    /** Constructor to initialize the HitCounter object. */
    HitCounter() {
        // The map is automatically initialized when the object is created
    }

    /**
     * Record a hit at the given timestamp.
     *
     * @param timestamp - The current timestamp in seconds granularity.
     */
    void hit(int timestamp) {
        // Increment the hit count for the given timestamp
        // If it does not exist, it is inserted into the map with a count of 0, then incremented
        hitCounts[timestamp]++;
    }

    /**
     * Return the number of hits in the past 5 minutes.
     *
     * @param timestamp - The current timestamp in seconds granularity.
     * @return the total number of hits in the last 5 minutes.
     */
    int getHits(int timestamp) {
        int hits = 0; // Variable to accumulate the number of hits

        // Iterate through all entries in the map
        for (auto &entry : hitCounts) {
            // Check if the entry's timestamp is within the past 5 minutes from the current timestamp
            if (timestamp - entry.first < 300) { // 300 seconds equals 5 minutes
                // Sum up the hits that are within the past 5 minutes
                hits += entry.second;
            }
        }
        // Return the total number of hits
        return hits;
    }
};

// Usage example
// HitCounter hitCounter;
// hitCounter.hit(timestamp);
// int numberOfHits = hitCounter.getHits(timestamp);
```

Typescript Solution

```
// A map to store the count of hits for each timestamp
let hitCounts: Map<number, number> = new Map<number, number>();

/**
 * Record a hit at the given timestamp.
 *
 * @param timestamp - The current timestamp in seconds granularity.
 */
function hit(timestamp: number): void {
    // Increment the hit count for the given timestamp, or set it to 1 if not present
    const currentCount = hitCounts.get(timestamp) || 0;
    hitCounts.set(timestamp, currentCount + 1);
}

/**
 * Return the number of hits in the past 5 minutes.
 *
 * @param timestamp - The current timestamp in seconds granularity.
 * @return the total number of hits in the last 5 minutes.
 */
function getHits(timestamp: number): number {
    let hits = 0; // Variable to accumulate the number of hits

    // Iterate through all key-value pairs in the map
    hitCounts.forEach((count, time) => {
        // Check if the time is within the past 5 minutes from the current timestamp
        if (timestamp - time < 300) { // 300 seconds equals 5 minutes
            hits += count; // Add up the hits within the past 5 minutes
        }
    });

    // Return the total number of hits
    return hits;
}

// Usage example
// hit(timestamp);
// let numberOfHits = getHits(timestamp);
```

Time and Space Complexity

Time Complexity

`init` (\mathcal{O}(1)): Initializing the counter data structure is a constant time operation.

`hit` (\mathcal{O}(1)): Incrementing the counter for a given timestamp is a constant time operation assuming that the underlying data structure has \(O(1)\) access time. The `Counter` in Python often provides \(O(1)\) time complexity for insertion and access.

`getHits` (\mathcal{O}(N)): This operation involves iterating over all entries in the counter and summing the values for timestamps within the last 5 minutes. In the worst case scenario, \(N\) represents the number of unique timestamps that have been recorded. The complexity becomes \(O(N)\) when we need to iterate through all timestamps. However, depending on the use case, if timestamps are in a dense range and we get hits at almost every second, the actual time complexity may approach \(O(300)\) since we are interested only in the hits in the last 5 minutes (300 seconds). Nevertheless, the worst-case scenario remains \(O(N)\).

Space Complexity

The space complexity is \(\mathcal{O}(N)\), where \(N\) is the number of unique timestamps recorded. This is because the counter needs to store each individual timestamp's hit count. In a scenario where the system runs indefinitely, the counter can grow unbounded as more unique timestamps are recorded. The memory usage becomes a function of the number of unique timestamps, thus causing a linear growth in space complexity.