981. Time Based Key-Value Store

Medium Design Hash Table String Binary Search

Problem Description

The problem is about creating a data structure that allows storing multiple values for the same key but at different timestamps, similar to a versioning system. The `TimeMap` should support two operations:

1. `set`: This operation stores the key with the value at the given timestamp.
2. `get`: This operation retrieves the value associated with a key at a specific timestamp. The catch here is that if the exact timestamp doesn't exist for that key, we need to provide the value with the closest previous timestamp. If there are no earlier timestamps, it should return an empty string.

To understand it better, imagine you are building a history tracking system for document edits. Each time someone edits a document, you record the new version of the document with a timestamp. Later, someone might want to see the version at a particular time. If a version is not available at the exact time requested, you show the latest version before the requested time.

Intuition

The key to solving this problem lies in efficiently managing the history of each key's values along with their timestamps. The intuition is to use a hash map to store keys but with a twist: each key has a list of pairs, each containing a timestamp and the corresponding value at that timestamp.

Given this structure, implementing the `set` function becomes straightforward: simply append the `(timestamp, value)` pair to the list associated with the key.

However, the challenge is in the `get` function, where you're asked to retrieve a value that is the closest to, but not greater than, the given timestamp. This is where binary search comes into play. The list of (timestamp, value) pairs for each key can be considered as sorted by timestamp. Using binary search, we that the closest previous timestamp to the given timestamp without going over it.

The binary search is implemented using the `bisect_right` function from Python's `bisect` module, which returns the index where an element should be inserted to maintain the sorted order. We search for a tuple where the first element is the given timestamp, and the second element is a dummy character chosen to be larger than any possible value (in this case, `chr(127)` which is the last ASCII character). If `bisect_right` provides a non-zero index, we step one index back and return the associated value. If the list is empty or the index is 0, it means there are no valid timestamps before the given timestamp, and as per problem description, we return an empty string.

Solution Approach

The solution is implemented with the following key ideas:

1. **Use of Defaultdict**: The `TimeMap` class leverages a defaultdict with lists as its default value. This is a choice of data structure from the `collections` module in Python which helps in automatically initializing a list for each new key without checking for the key's existence. The structure `self.ktv` is a dictionary where each key has a list of `(timestamp, value)` tuples.
2. **Storing Values with TimeStamps**:
   - The `set` method appends a tuple consisting of the `timestamp` and the `value` to the list associated with the `key`. Since `set` is always called with increasing timestamp order, the list of values for each key naturally remains sorted by timestamp.
3. **Retrieving Values**:
   - The `get` method uses a binary search algorithm to quickly find the correct value for a given timestamp.
   - It checks whether the `key` exists in `self.ktv`. If not, it returns an empty string since there are no entries for that `key`.
   - If the key exists, a binary search is performed using the `bisect_right` function, which finds the insertion point in the list of values to maintain the sorted order.
   - The target of the binary search is a tuple where the first element is the `timestamp` and the second is a very high-value character. `chr(127)` works as the dummy character because `chr(127)` will always be greater than any string value, ensuring `bisect_right` returns the position for the latest timestamp that is not greater than the given timestamp.
   - If `bisect_right` returns zero, it means there were no timestamps less than or equal to the `timestamp` that we passed, so we return an empty string.
   - Otherwise, we subtract one from the index that `bisect_right` returned to find the largest timestamp less than or equal to the `timestamp` requested. Then we return the value associated with this timestamp.

Here's the encapsulated code structure:

- Initialization: `def __init__(self): self.ktv = defaultdict(list)`
- set: `def set(self, key: str, value: str, timestamp: int) -> None: self.ktv[key].append((timestamp, value))`
- get:

```python
def get(self, key: str, timestamp: int) -> str:
    if key not in self.ktv:
        return ''
    tv = self.ktv[key]
    i = bisect_right(tv, (timestamp, chr(127)))
    return tv[i - 1][1] if i else ''
```

Overall, the use of binary search ensures that the retrieval of values is efficient, even when the number of timestamps grows large. It allows the `get` operation to have a time complexity of O(log n), where n is the number of timestamps associated with a key, making the solution scalable.

Example Walkthrough

Let's walk through the process with a simple example to illustrate the solution approach.

Suppose we are using the `TimeMap` to record the status of a project at different times.

- At time 1, the status is "Started"
- At time 3, the status updates to "In Progress"
- At time 5, the status finally changes to "Completed"

We would perform the following operations on our `TimeMap`:

1. `set("projectStatus", "Started", 1)`
2. `set("projectStatus", "In Progress", 3)`
3. `set("projectStatus", "Completed", 5)`

Now, our `TimeMap` data structure, `self.ktv`, has an entry for "projectStatus", which is a list of timestamp and value tuples like the following:

```
"projectStatus": [(1, "Started"), (3, "In Progress"), (5, "Completed")]
```

Next, let's retrieve the status of the project at different times using the `get` operation:

- `get("projectStatus", 2)`: This should return "Started" since it's the latest status before time 2.
- `get("projectStatus", 4)`: This should return "In Progress" for the same reason.
- `get("projectStatus", 5)`: This should return "Completed" as it matches the timestamp exactly.
- `get("projectStatus", 0)`: This should return an empty string since there are no statuses recorded before time 1.

Let's walk through the first `get` operation step-by-step:

1. Look up "projectStatus" in `self.ktv` and find the list `[(1, "Started"), (3, "In Progress"), (5, "Completed")]`.
2. Using `bisect_right`, find the index where we would place `(2, chr(127))` to maintain sorted order.
   - The sorted position would be index 1 (between `(1, "Started")` and `(3, "In Progress")`).
3. Subtract one from the index: `1 - 1 = 0`.
4. We return the value associated with this index, so `("projectStatus", 2)` returns `"Started"`.

Now, imagine checking for `get("projectStatus", 0)`:

1. Look up "projectStatus" and find the same list as above.
2. With `bisect_right`, we find the index where `(0, chr(127))` would fit, which gives us index 0 because 0 is less than any existing timestamps in the list.
3. Since the index is 0, it would be incorrect to subtract 1 to find the previous timestamp, as this would lead to a negative index. Instead, we return an empty string, as there's no status before time 1.

Therefore, `("projectStatus", 0)` returns an empty string, which aligns with the functionality described in the problem. This example confirms that the binary search method provides an efficient and effective way to retrieve the correct value, complying with the problem's requirements.

Python Solution

```python
from collections import defaultdict
from bisect import bisect_right

class TimeMap:
    def __init__(self):
        # Initialize a dictionary to store the lists of (timestamp, value) tuples for each key.
        self.key_time_value = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        # Append the (timestamp, value) tuple to the list corresponding to the key.
        self.key_time_value[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        # If the key does not exist in the dictionary, return an empty string.
        if key not in self.key_time_value:
            return ''

        # Retrieve the list of (timestamp, value) tuples for the given key.
        time_value_pairs = self.key_time_value[key]

        # Use bisect_right to find the index where the timestamp would be inserted
        # to maintain the list in sorted order. Since the list is sorted by timestamp,
        # and we want to find the largest timestamp less than or equal to the given timestamp,
        # we use (timestamp, chr(127)) as a "high" value to ensure we can locate
        # timestamps that are equal to the given one.
        index = bisect_right(time_value_pairs, (timestamp, chr(127)))

        # If index is 0, there is no timestamp less than or equal to the given timestamp,
        # so we return an empty string. If not, we return the value corresponding to
        # the timestamp immediately before the insertion point, which would be index - 1.
        return time_value_pairs[index - 1][1] if index else ''

# Your TimeMap object will be instantiated and called as such:
# obj = TimeMap()
# obj.set(key,value,timestamp)
# param_2 = obj.get(key,timestamp)
```

Java Solution

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

class TimeMap {
    // Using a Map where each key is a string and its value is a TreeMap that associates timestamps with values
    private Map<String, TreeMap<Integer, String>> keyTimeValueMap = new HashMap<>();

    /**
     * Initializes the TimeMap object.
     */
    public TimeMap() {
        // Constructor does not need to initialize anything since the HashMap is already initialized.
    }

    /**
     * Stores the key with the value along with the given timestamp.
     *
     * @param key       the key to store
     * @param value     the value associated with the key
     * @param timestamp the timestamp at which the value is set for the key
     */
    public void set(String key, String value, int timestamp) {
        // Compute if absent will insert a new TreeMap if the key is not already present
        // It then puts the given timestamp and value into the TreeMap for the given key
        keyTimeValueMap.computeIfAbsent(key, k -> new TreeMap<>()).put(timestamp, value);
    }

    /**
     * Retrieves the value for the key at the given timestamp or the last value set before the timestamp.
     *
     * @param key       the key to retrieve the value for
     * @param timestamp the timestamp to retrieve the value at
     * @return the value set for the key at the given timestamp or the closest previous timestamp
     */
    public String get(String key, int timestamp) {
        // If the key does not exist, return an empty string
        if (!keyTimeValueMap.containsKey(key)) {
            return "";
        }
        // Retrieve the TreeMap for the given key
        TreeMap<Integer, String> timeValueMap = keyTimeValueMap.get(key);
        // Find the greatest timestamp less than or equal to the given timestamp
        Integer closestTimestamp = timeValueMap.floorKey(timestamp);
        // If there is no timestamp that is less than or equal to the given timestamp, return an empty string
        // Otherwise, return the associated value
        return closestTimestamp == null ? "" : timeValueMap.get(closestTimestamp);
    }
}
```

C++ Solution

```cpp
#include <string>
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

class TimeMap {
public:
    /** Initialize your data structure here. */
    TimeMap() {
        // Constructor remains empty as there's no initialization needed beyond that of member variables.
    }

    /**
    * Stores the key with its value along with the timestamp in the map.
    * The values associated with each key are stored in a vector of pairs, sorted by their timestamps.
    * @param key The string key of the value.
    * @param value The value to associate with the key.
    * @param timestamp The time at which the value was set.
    */
    void set(string key, string value, int timestamp) {
        // Appending the pair <timestamp, value> to the vector associated with the key.
        keyTimeValueMap[key].emplace_back(timestamp, value);
    }

    /**
    * Retrieves a value associated with the key at a particular timestamp.
    * If the specific timestamp doesn't exist, the value at the most recent timestamp before that is returned.
    * @param key The key whose value needs to be retrieved.
    * @param timestamp The timestamp at which the value was needed.
    * @return The value at the given timestamp or the last known value before the given timestamp.
    */
    string get(string key, int timestamp) {
        // Reference to the vector of pairs for the given key.
        auto& pairs = keyTimeValueMap[key];

        // Constructing a dummy pair up to the current timestamp to use in searching.
        pair<int, string> dummyPair = {timestamp, string({127})};  // 127 is used as the end of the range character.

        // Finding the upper bound, which will point at the first element that is greater than the dummy pair.
        auto it = upper_bound(pairs.begin(), pairs.end(), dummyPair);

        // If the iterator is at the beginning, there are no elements less than or equal to the timestamp,
        // hence return an empty string. Otherwise, decrement iterator to get to the element which has
        // the latest time less than or equal to the given timestamp.
        return it == pairs.begin() ? "" : (prev(it))->second;
    }

private:
    // Map that associates each key with a vector of pairs, where each pair consists of a timestamp
    // and the value associated with that timestamp.
    unordered_map<string, vector<pair<int, string>>> keyTimeValueMap;
};

/**
 * Your TimeMap object will be instantiated and called as such:
 * TimeMap* obj = new TimeMap();
 * obj->set(key, value, timestamp);
 * string val = obj->get(key, timestamp);
 */
```

Typescript Solution

```typescript
type KeyValuePair = { timestamp: number; value: string };
const keyTimeValueMap: Record<string, KeyValuePair[]> = {};

function set(key: string, value: string, timestamp: number): void {
    if (!keyTimeValueMap[key]) {
        keyTimeValueMap[key] = [];
    }
    // Append the pair {timestamp, value} to the array associated with the key.
    keyTimeValueMap[key].push({ timestamp, value });
}

function get(key: string, timestamp: number): string {
    const pairs = keyTimeValueMap[key];
    if (!pairs) {
        return '';
    }
    // Use binary search to find the smallest index i such that pairs[i].timestamp is greater than the timestamp.
    let low = 0, high = pairs.length;
    while (low < high) {
        const mid = Math.floor((low + high) / 2);
        if (pairs[mid].timestamp <= timestamp) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    // If we found an element whose timestamp is greater than our target or there are no
    // elements with a timestamp <= target, 'low' will be the length of the array or 0, respectively.
    if (low === 0) {
        return '';
    }
    // Return the last known value before the given timestamp.
    return pairs[low - 1].value;
}
```

Time and Space Complexity

Time Complexity

The `set` method has a time complexity of `O(1)` for each operation, as it appends the `(timestamp, value)` tuple to the list corresponding to the key in the `defaultdict`.

The `get` method's time complexity is `O(log N)` for each operation, where `N` is the number of entries associated with the specific key. This is because it performs a binary search (`bisect_right`) to find the position where the given `timestamp` would fit in the sorted order of timestamps stored.

Space Complexity

The space complexity is `O(K + T)`, where `K` is the number of unique keys and `T` is the total number of `set` calls or the total number of timestamp and value pairs. This is because all values with their corresponding timestamps for all keys are stored in the `defaultdict`.