635. Design Log Storage System

Medium | Design | Hash Table | String | Ordered Set

Leetcode Link

Problem Description

The given LeetCode problem involves creating a log system that can store log entries with a unique ID and timestamp, and then retrieve logs that fall within a given time range based on varying levels of granularity. A timestamp in this problem is given in the string format `"Year:Month:Day:Hour:Minute:Second"`, where each part of the date-time is zero-padded. The system needs to support two operations:

- **`put(int id, string timestamp)`:** This function takes a log entry's unique ID and its timestamp, then stores it in the system.
- **`retrieve(string start, string end, string granularity)`:** This method returns the list of log IDs that have timestamps within the inclusive range specified by `start` and `end`. The granularity parameter determines how precise the time range should be, ranging from the year down to the second. The granularity truncates the timestamp to the specified level of detail, and only that level of detail is considered when filtering logs.

For example, if the granularity is "Day", then the logs are filtered based on their year, month, and day, without considering hours, minutes, or seconds.

Intuition

The intuition behind the solution involves storing the log entries efficiently so that they can be easily retrieved based on time range queries with varying granularities. To retrieve the correct logs, we need to compare only the parts of the timestamps that are relevant to the specified granularity. The solution defines a dictionary `granularity_map` that contains the cutoff indices for each granularity. These indices are used to truncate the timestamp strings to the required precision.

The retrieval function takes advantage of Python's string comparison, which can be used directly on the truncated timestamp strings. Since the timestamps are fixed-length and zero-padded, lexicographical comparison of the strings works equivalently to comparing the numerical values.

Here's the thought process for arriving at the solution approach:

1. Store all logs in a list as they are received. There's no need to sort them since we can filter them during retrieval.
2. Define a dictionary that maps each granularity to the index up to which the timestamp should be considered. For instance, if the granularity is "Year", we only consider the first four characters of the timestamp.
3. When retrieving logs within a specific range, determine the cutoff index for the timestamps based on granularity. Use slicing to truncate the `start`, `end`, and current log timestamp to the relevant precision.
4. Iterate through all stored logs, compare the truncated timestamp of each log to the truncated `start` and `end` timestamps, and filter accordingly.
5. Return the list of IDs of the logs that satisfy the range and granularity criteria.

By following this approach, we ensure that we can efficiently retrieve the correct log entries without the need for complex date-time parsing or conversion.

Solution Approach

The implementation of the `LogSystem` class involves two key methods: `put` and `retrieve`.

The `put` method is straightforward. Each call to `put` adds a tuple consisting of the `id` and `timestamp` to the logs list:

```
def put(self, id: int, timestamp: str) -> None:
    self.logs.append((id, timestamp))
```

Here we don't need to worry about sorting or the position of the log entry because the retrieval will handle the search based on the timestamp.

The `retrieve` method does the main work of filtering the log entries according to the provided time range and granularity:

```
def retrieve(self, start: str, end: str, granularity: str) -> List[int]:
    # Truncate start, end, and log timestamps to the specified granularity
    return [id for id, ts in self.logs if start[:i] <= ts[:i] <= end[:i]]
```

The use of dictionary `i` to store the indices for each granularity level allows us to easily retrieve the correct substring length to compare. For example, for granularity `Year`, `d['Year']` would be 4, so `start[:4]` would give us the year component of the `start` timestamp.

A comprehensive breakdown of the algorithm used:

1. Upon initialization (`__init__`), an empty list `logs` is created to store the log entries. A dictionary `d` is also initialized to map granularity to corresponding substring indices.
2. The `put` method simply appends the provided log entry to the `logs` list without any additional processing.
3. The `retrieve` method calculates the index `i` up to which the timestamps should be considered, based on the granularity. It uses list comprehension to iterate through all the stored logs and includes the ones that match the range condition.
4. For each log, the comparison is done by slicing the timestamps of the logs and the provided `start` and `end` parameters up to index `i`. This way, we only compare the parts of the timestamps that matter for the specified granularity.
5. We rely on Python's string comparison to compare the sliced timestamps. Since timestamps are zero-padded and have a consistent format, this works out as effectively as numeric comparison.
6. The method finally returns a list of IDs whose timestamps fall within the specified range.

This implementation is efficient as it separates the storage and querying of log entries. The dictionary `d` effectively eliminates complex parsing or time conversion by using string slicing based on predefined indices corresponding to each granularity.

Example Walkthrough

Let's say we instantiate a `LogSystem` and perform a series of `put` and `retrieve` operations as per the solution approach described.

First, we add a few logs to the system with varying timestamps:

```
1 LogSystem ls = LogSystem()
2 ls.put(1, "2017:01:01:23:59:59")
3 ls.put(2, "2017:01:02:23:59:59")
4 ls.put(3, "2017:02:01:23:59:59")
```

In our logs data structure, we now have:

```
1 [(1, "2017:01:01:23:59:59"), (2, "2017:01:02:23:59:59"), (3, "2017:02:01:23:59:59")]
```

No sorting is needed; we just store the logs as they come.

Suppose we want to retrieve logs with timestamps in January 2017; we can call the `retrieve` method accordingly:

```
1 results = ls.retrieve("2017:01:01:00:00:00", "2017:01:31:23:59:59", "Month")
```

The `i` dictionary in the `LogSystem` has an entry `"Month": 7`, indicating that we should compare the timestamps up to the seventh character for a "Month" granularity, which corresponds to the year and month components.

The `retrieve` method will:

1. Determine the index `i` for month granularity, which is 7. Thus, we will compare timestamps up to `"2017:01"`.
2. Iterate through the logs:

```
1 - For log (1, "2017:01:01:23:59:59"), the truncated timestamp is "2017:01" which is within the range "2017:01" to "2017:01".
2 - For log (2, "2017:01:02:23:59:59"), the truncated timestamp is also "2017:01", which is within the range.
3 - For log (3, "2017:02:01:23:59:59"), the truncated timestamp is "2017:02", which is outside the range.
```

3. Create a list of the IDs where the condition holds true. In this case, logs 1 and 2 match, while log 3 does not.

As a result, `results` will have the value `[1, 2]`, meaning log entries with IDs 1 and 2 are the ones that fall within the desired time range and granularity.

This example demonstrates how the implementation allows us to efficiently filter logs based on a specified granularity and time range without complex parsing.

Python Solution

```
class LogSystem:
    def __init__(self):
        # Initialize logs list to hold all the log entries as tuples (id, timestamp)
        self.logs = []

        # Dictionary to map the granularity level to the index of timestamp string
        self.granularity_to_index = {
            "Year": 4,   # Index where the Year value ends in the timestamp
            "Month": 7,  # Index where the Month value ends in the timestamp
            "Day": 10,   # Index where the Day value ends in the timestamp
            "Hour": 13,  # Index where the Hour value ends in the timestamp
            "Minute": 16,# Index where the Minute value ends in the timestamp
            "Second": 19 # Index where the Second value ends in the timestamp
        }

    def put(self, log_id: int, timestamp: str) -> None:
        # Store the log with its id and timestamp
        self.logs.append((log_id, timestamp))

    def retrieve(self, start: str, end: str, granularity: str) -> list:
        # Find the index up to which we will compare timestamps based on the granularity
        index = self.granularity_to_index[granularity]

        # Retrieve log_ids of logs where timestamp is within the start and end range,
        # sliced according to the granularity index
        return [log_id for log_id, ts in self.logs if start[:index] <= ts[:index] <= end[:index]]

# Example of how to use the LogSystem class
# obj = LogSystem()
# obj.put(log_id, timestamp)
# result = obj.retrieve(start, end, granularity)
```

Java Solution

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

// Class representing a system to store and retrieve logs based on time granularity
class LogSystem {

    // A list to store all logs
    private List<LogEntry> logEntries = new ArrayList<>();

    // A map to store the length of the timestamp string
    // for each granularity level
    private Map<String, Integer> granularityMap = new HashMap<>();

    // Constructor for initializing the granularity map
    public LogSystem() {
        granularityMap.put("Year", 4);
        granularityMap.put("Month", 7);
        granularityMap.put("Day", 10);
        granularityMap.put("Hour", 13);
        granularityMap.put("Minute", 16);
        granularityMap.put("Second", 19);
    }

    // Method to store an individual log entry
    public void put(int id, String timestamp) {
        logEntries.add(new LogEntry(id, timestamp));
    }

    // Method to retrieve log entries within the start and end
    // timestamps, considering the specified granularity
    public List<Integer> retrieve(String start, String end, String granularity) {
        List<Integer> result = new ArrayList<>();
        int granularityLength = granularityMap.get(granularity);

        // Extract the relevant portions of the start and end timestamps
        // based on the granularity
        String startPrefix = start.substring(0, granularityLength);
        String endPrefix = end.substring(0, granularityLength);

        // Loop over all log entries to find the logs within the time range
        for (LogEntry logEntry : logEntries) {
            // Extract the relevant portion of the log's timestamp
            String logTimestampPrefix = logEntry.getTimestamp().substring(0, granularityLength);

            // Check if the log's timestamp is within the range
            if (startPrefix.compareTo(logTimestampPrefix) <= 0 && logTimestampPrefix.compareTo(endPrefix) <= 0) {
                result.add(logEntry.getId());
            }
        }
        return result;
    }
}

// Class representing an individual log entry
class LogEntry {
    private int id; // The ID of the log entry
    private String timestamp; // The timestamp of the log entry

    // Constructor for log entry
    public LogEntry(int id, String timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    // Getter for ID
    public int getId() {
        return id;
    }

    // Getter for timestamp
    public String getTimestamp() {
        return timestamp;
    }
}

/**
 * The LogSystem class can be used as follows:
 * LogSystem logSystem = new LogSystem();
 * logSystem.put(id, timestamp); // to store a log entry
 * List<Integer> results = logSystem.retrieve(start, end, granularity); // to retrieve log entries
 */
```

C++ Solution

```
#include <vector>
#include <string>
#include <unordered_map>

using std::vector;
using std::string;
using std::unordered_map;
using std::pair;

class LogSystem {
public:
    LogSystem() {
        granularityMap_["Year"] = 4;   // Year granularity ends at the 4th character index (yyyy)
        granularityMap_["Month"] = 7;  // Month granularity ends at the 7th character index (yyyy-MM)
        granularityMap_["Day"] = 10;   // Day granularity ends at the 10th character index (yyyy-MM-dd)
        granularityMap_["Hour"] = 13;  // Hour granularity ends at the 13th character index (yyyy-MM-dd:HH)
        granularityMap_["Minute"] = 16; // Minute granularity ends at the 16th character index (yyyy-MM-dd:HH:mm)
        granularityMap_["Second"] = 19; // Second granularity ends at the 19th character index (yyyy-MM-dd:HH:mm:ss)
    }

    // Stores a log entry with a unique ID and timestamp.
    void put(int id, string timestamp) {
        logs_.emplace_back(id, timestamp);
    }

    // Retrieves IDs of log entries that have timestamps between the range [start, end],
    // considering the specified granularity.
    vector<int> retrieve(string start, string end, string granularity) {
        int index = granularityMap_[granularity]; // Index determines the granularity level.
        int startSubstr = start.substr(0, index); // Substrings of start and end based on the granularity to compare the timestamps.
        int endSubstr = end.substr(0, index);
        auto res = vector<int>();

        // Iterate through all stored logs to find matching entries.
        for (const auto& [id, ts] : logs_) {
            // Substring the timestamp of the current log to match granularity.
            auto timestampSubstr = ts.substr(0, index);

            // If the log's timestamp is within the range, add its ID to the result.
            if (startSubstr <= timestampSubstr && timestampSubstr <= endSubstr) {
                res.emplace_back(logEntry.first);
            }
        }
        return res;
    }

private:
    vector<pair<int, string>> logs_; // Vector storing log entries as <ID, timestamp> pairs.
    unordered_map<string, int> granularityMap_; // Maps granularity strings to substring index limits.
};
```

Typescript Solution

```
// Typescript code for a log system using global variables and functions

// Define available granularities and corresponding character index limits in a timestamp
const granularityMap: { key: string[]; number: number } = {
    "Year": 4,   // Year granularity ends at the 4th character index (yyyy)
    "Month": 7,  // Month granularity ends at the 7th character index (yyyy-MM)
    "Day": 10,   // Day granularity ends at the 10th character index (yyyy-MM-dd)
    "Hour": 13,  // Hour granularity ends at the 13th character index (yyyy-MM-dd:HH)
    "Minute": 16,// Minute granularity ends at the 16th character index (yyyy-MM-dd:HH:mm)
    "Second": 19 // Second granularity ends at the 19th character index (yyyy-MM-dd:HH:mm:ss)
};

// Initialize an array to store log entries as [id, timestamp] objects
let logs: [id: number, timestamp: string][] = [];

// Function to store a log entry with a unique ID and timestamp
function put(id: number, timestamp: string): void {
    logs.push([id, timestamp]);
}

// Function to retrieve IDs of log entries that have timestamps between the range [start, end],
// considering the specified granularity.
function retrieve(start: string, end: string, granularity: string): number[] {
    const result: number[] = []; // Array to store the matching log entry IDs
    const index = granularityMap[granularity]; // Index determines the granularity level.

    // Substrings of start and end based on the granularity to compare timestamps
    const startSubstr = start.substring(0, index);
    const endSubstr = end.substring(0, index);

    // Iterate through all stored logs to find matching entries
    for (const [id, ts] of logs) {
        // Substring the timestamp of the current log to match granularity
        const timestampSubstr = ts.substring(0, index);

        // If the log's timestamp is within the range, add its ID to the result
        if (startSubstr <= timestampSubstr && timestampSubstr <= endSubstr) {
            result.push(logEntry.id);
        }
    }
    return result;
}
```

Time and Space Complexity

Time Complexity

`put` method: The `put` operation has a time complexity of $O(1)$ since it only involves appending a tuple (ID and timestamp) to the end of the `self.logs` list which is a constant time operation.

`retrieve` method: The `retrieve` operation involves iterating over each log entry in `self.logs` and comparing the substrings of their timestamps. The time complexity is $O(n * s)$ where n is the number of log entries and s is the length of the timestamp substring being compared. The complexity of the comparison operation is assumed to be $O(1)$ since the length of the substring depends on the granularity and doesn't change with the size of `self.logs`.

Space Complexity

Overall: The space complexity of the LogSystem class is $O(n)$, where n is the number of log entries stored in `self.logs`. Each log entry requires a fixed amount of space for the integer ID and a fixed-length string for the timestamp, resulting in space directly proportional to the number of entries.

`put` method: For the `put` method, it only appends the log data without needing any extra space besides the `self.logs`, so the space complexity incurred by this method is $O(1)$ in addition to the overall $O(n)$ space needed for storing the logs themselves.

`retrieve` method: The `retrieve` method creates a new list of IDs that match the criteria, the space complexity of which is $O(k)$ where k is the number of matching log entries returned. Since k can be at most n in the worst case, the worst-case space complexity for `retrieve` can also be $O(n)$.