635. Design Log Storage System

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 `i` 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:

```
1  def put(self, id: int, timestamp: str) -> None:
2      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:

```
1  def retrieve(self, start: str, end: str, granularity: str) -> List[int]:
2      i = self.d[granularity]  # Find the index up to which we should compare timestamps
3      return [id for id, ts in self.logs if start[:i] <= ts[:i] <= end[:i]]
```

The use of dictionary `d` to store the indices for each granularity level allows us to easily retrieve the correct substring length to compare. For example, for granularity `Year`, at `Year` I 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 numerical 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 = LogSystem()
2  logSystem.put(1, "2017:01:01:23:59:59")
3  logSystem.put(2, "2017:01:01:22:59:59")
4  logSystem.put(3, "2016:01:01:00:00:00")
```

In our logs data structure, we now have:

```
1  [(1, "2017:01:01:23:59:59"), (2, "2017:01:01:22:59:59"), (3, "2016:01:01:00:00:00")]
```

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

The `d` 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:

- 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".
- For log (2, "2017:01:01:22:59:59"), the truncated timestamp is also "2017:01", which is within the range.
- For log (3, "2016:01:01:00:00:00"), the truncated timestamp is "2016:01", 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

```python
1  class LogSystem:
2      def __init__(self):
3          # Initialize logs list to hold all the log entries as tuples (id, timestamp)
4          self.logs = []
5
6          # Dictionary to map the granularity level to the index of timestamp string
7          self.granularity_to_index = {
8              "Year": 4,    # Index where the Year value ends in the timestamp
9              "Month": 7,   # Index where the Month value ends in the timestamp
10             "Day": 10,    # Index where the Day value ends in the timestamp
11             "Hour": 13,   # Index where the Hour value ends in the timestamp
12             "Minute": 16, # Index where the Minute value ends in the timestamp
13             "Second": 19, # Index where the Second value ends in the timestamp
14         }
15
16     def put(self, log_id: int, timestamp: str) -> None:
17         # Store the log with its id and timestamp
18         self.logs.append((log_id, timestamp))
19
20     def retrieve(self, start: str, end: str, granularity: str) -> list:
21         # Find the index up to which we will compare timestamps based on the granularity
22         index = self.granularity_to_index[granularity]
23
24         # Retrieve log_ids of logs where timestamp is within the start and end range,
25         # sliced according to the granularity level
26         return [log_id for log_id, ts in self.logs if start[:index] <= ts[:index] <= end[:index]]
27
28 # Example of how to use the LogSystem class
29 # d = LogSystem()
30 # d.put(1, timestamp)
31 # results = d.retrieve(start, end, granularity)
```

Java Solution

```java
1  import java.util.ArrayList;
2  import java.util.HashMap;
3  import java.util.List;
4  import java.util.Map;
5
6  // Class representing a system to store and retrieve logs based on time granularity
7  class LogSystem {
8
9      // A list to store all logs
10     private List<String[]> logs = new ArrayList<>();
11
12     // A map to store the length of the timestamp string
13     // based on the granularity
14     private Map<String, Integer> granularityMap = new HashMap<>();
15
16     // Constructor for initializing the granularity map
17     public LogSystem() {
18         granularityMap.put("Year", 4);
19         granularityMap.put("Month", 7);
20         granularityMap.put("Day", 10);
21         granularityMap.put("Hour", 13);
22         granularityMap.put("Minute", 16);
23         granularityMap.put("Second", 19);
24     }
25
26     // Method to store an individual log entry
27     public void put(int id, String timestamp) {
28         logEntries.add(new LogEntry(id, timestamp));
29     }
30
31     // Method to retrieve log entries within the start and end
32     // timestamp based on the given granularity
33     public List<Integer> retrieve(String start, String end, String granularity) {
34         List<Integer> result = new ArrayList<>();
35         int granularityLength = granularityMap.get(granularity);
36
37         // Extract the relevant portions of the start and end timestamps
38         // based on the granularity
39         String startPrefix = start.substring(0, granularityLength);
40         String endPrefix = end.substring(0, granularityLength);
41
42         // Loop over all log entries to find the logs within the time range
43         for (LogEntry log : logEntries) {
44             // Extract the relevant portion of the log's timestamp
45             String logTimestampPrefix = log.getTimestamp().substring(0, granularityLength);
46
47             // Check if the log's timestamp is within the time range
48             if (startPrefix.compareTo(logTimestampPrefix) <= 0 && logTimestampPrefix.compareTo(endPrefix) <= 0) {
49                 result.add(log.getId());
50             }
51         }
52         return result;
53     }
54 }
55
56 // Class representing an individual log entry
57 class LogEntry {
58     private int id;            // The ID of the log entry
59     private String timestamp;  // The timestamp of the log entry
60
61     // Constructor for log entry
62     public LogEntry(int id, String timestamp) {
63         this.id = id;
64         this.timestamp = timestamp;
65     }
66
67     // Getter for ID
68     public int getId() {
69         return id;
70     }
71
72     // Getter for timestamp
73     public String getTimestamp() {
74         return timestamp;
75     }
76 }
77
78 /**
79  * The LogSystem class can be used as follows:
80  * LogSystem logSystem = new LogSystem();
81  * logSystem.put(id, timestamp); // To store a log entry
82  * List<Integer> result = logSystem.retrieve(start, end, granularity); // To retrieve log entries
83  */
```

C++ Solution

```cpp
1  #include <vector>
2  #include <string>
3  #include <unordered_map>
4
5  using std::vector;
6  using std::string;
7  using std::unordered_map;
8  using std::pair;
9
10 class LogSystem {
11 public:
12     LogSystem() {
13         granularityMap_["Year"] = 4;    // Year granularity ends at the 4th character index (yyyy)
14         granularityMap_["Month"] = 7;   // Month granularity ends at the 7th character index (yyyy-MM)
15         granularityMap_["Day"] = 10;    // Day granularity ends at the 10th character index (yyyy-MM-dd)
16         granularityMap_["Hour"] = 13;   // Hour granularity ends at the 13th character index (yyyy-MM-dd:HH)
17         granularityMap_["Minute"] = 16; // Minute granularity ends at the 16th character index (yyyy-MM-dd:HH:mm)
18         granularityMap_["Second"] = 19; // Second granularity ends at the 19th character index (yyyy-MM-dd:HH:mm:ss)
19     }
20
21     // Stores a log entry with a unique ID and timestamp.
22     void put(int id, string timestamp) {
23         logs_.emplace_back(id, timestamp);
24     }
25
26     // Retrieves IDs of log entries that have timestamps between the range [start, end],
27     // considering the specified granularity.
28     vector<int> retrieve(string start, string end, string granularity) {
29         vector<int> result;
30         int granularityLength = granularityMap_[granularity]; // Get the length of the timestamp corresponding to the granularity level.
31
32         // Iterate through all stored log entries.
33         for (const auto& log : logs_) {
34             // Extract the portion of the current log to match granularity.
35             auto timestampSubstr = log.second.substr(0, granularityLength);
36
37             // If the log's timestamp is within the range, add its ID to the result.
38             if (timestampSubstr >= start.substr(0, granularityLength) &&
39                 timestampSubstr <= end.substr(0, granularityLength)) {
40                 result.emplace_back(log.first);
41             }
42         }
43         return result;
44     }
45
46 private:
47     unordered_map<string, int> granularityMap_; // Vector storing log entries as <ID, timestamp> pairs.
48     vector<pair<int, string>> logs_; // Maps granularity strings to substring index limits.
49 };
```

Typescript Solution

```typescript
1  // TypeScript code for a log system using global variables and functions
2
3  // Define available granularities and corresponding character index limits in a timestamp
4  const granularityMap: { [key: string]: number } = {
5      "Year": 4,    // Year granularity ends at the 4th character index (yyyy)
6      "Month": 7,   // Month granularity ends at the 7th character index (yyyy-MM)
7      "Day": 10,    // Day granularity ends at the 10th character index (yyyy-MM-dd)
8      "Hour": 13,   // Hour granularity ends at the 13th character index (yyyy-MM-dd:HH)
9      "Minute": 16, // Minute granularity ends at the 16th character index (yyyy-MM-dd:HH:mm)
10     "Second": 19, // Second granularity ends at the 19th character index (yyyy-MM-dd:HH:mm:ss)
11 };
12
13 // Initialize an array to store log entries as [id, timestamp] tuples
14 let logs: [id: number, timestamp: string][] = [];
15
16 // Function to store a log entry with an unique ID and timestamp
17 function put(id: number, timestamp: string): void {
18     logs.push([id, timestamp]);
19 }
20
21 // Function to retrieve IDs of log entries that have timestamps between the range [start, end],
22 // considering the specified granularity
23 function retrieve(start: string, end: string, granularity: string): number[] {
24     const result: number[] = [];
25     const granularityLength: number = granularityMap[granularity]; // Index denotes the granularity level
26
27     // Iterate through all log entries
28     for (const [id, timestamp] of logs) {
29         // Extract the relevant part of the current log to match granularity
30         const timestampSubstr: string = timestamp.substr(0, granularityLength);
31
32         // If the log's timestamp is within the range, add its ID to the result
33         if (timestampSubstr >= start.substr(0, granularityLength) &&
34             timestampSubstr <= end.substr(0, granularityLength)) {
35             result.push(id);
36         }
37     }
38     return result;
39 }
```

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 logs in the system and s is the length of the timestamp substring being compared. The complexity of the comparison operation is assumed to be $O(s)$ since the length of the substrings 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 $O(n)$.