2722. Join Two Arrays by ID

Medium

Leetcode Link

Problem Description

The problem provides two arrays, `arr1` and `arr2`, each containing objects that have an `id` field with an integer value. The goal is to merge these arrays into a single array `joinedArray` in such a way that `joinedArray` has the combined contents of both `arr1` and `arr2`, with each object having a unique `id`. If an `id` exists in only one array, the corresponding object is included in `joinedArray` without changes. If the same `id` appears in both arrays, then the resultant object in `joinedArray` should have its properties merged; if a property exists only in one object, it is directly taken over, but if a property is present in both, the value from the object in `arr2` must overwrite the value from `arr1`. Finally, `joinedArray` is sorted in ascending order by the `id` key.

Intuition

To solve this problem, we need to find an efficient way to merge the objects based on their `id`. A `Map` data structure is suitable for this task because it allows quick access and insertion of key-value pairs where the key is unique (the `id` in our case). The following steps exemplify the approach:

1. Create a new `Map`, and populate it with objects from `arr1`, using `id` as the key. This enables us to quickly locate any object based on its `id`.

2. Go through each object in `arr2`, check if an object with the same `id` already exists in the `Map`:

   - If it does, merge the existing object with the object from `arr2`. Object destructuring (`{ ...d.get(x.id), ...x }`) facilitates this by copying properties from both objects into a new object, with properties from `x` (the object from `arr2`) having priority in case of any conflicts.

   - If it does not, simply add the object from `arr2` to the `Map`.

3. Convert the `Map` values into an array using `[...d.values()]`, which ensures that each id is represented by a single, merged object.

4. Sort the resulting array by `id` in ascending order.

This method ensures that we honor the conditions for merging and respect the values from `arr2` in cases of overlap, all while preparing the `joinedArray` to be returned with the correct order of `id` values.

Solution Approach

The solution approach is straightforward and efficient, leveraging the JavaScript `Map` object to handle merging and ensuring unique `id` values. Here's a walkthrough of the implementation:

1. Initialize a new `Map` and populate it with the `id` and corresponding object from `arr1`:

```
const d = new Map(arr1.map(x => [x.id, x]));
```

In this line, `arr1.map(x => [x.id, x])` effectively prepares an array of `[id, object]` pairs that can be accepted by the `Map` constructor, establishing a direct mapping between each `id` and its respective object.

2. Iterate through `arr2` and merge or add objects as necessary:

```
arr2.forEach(x => {
    if (d.has(x.id)) {
        d.set(x.id, { ...d.get(x.id), ...x });
    } else {
        d.set(x.id, x);
    }
});
```

In this snippet, `d.has(x.id)` checks if the current `id` from `arr2` is already present in the map `d`. If it is, the objects from `arr1` and `arr2` are merged with object spread syntax `{ ...d.get(x.id), ...x }`, where properties from `x` (from `arr2`) can overwrite those from `d.get(x.id)` (from `arr1`) in case of duplication. If the `id` is not present, the `id` and object are added to the map as a new entry.

3. Transform the `Map` into an array and sort the objects by `id`:

```
return [...d.values()].sort((a, b) => a.id - b.id);
```

Here, `[...d.values()]` transforms the `Map` values (our merged objects) into an array. The `sort` function is used to sort the array in ascending order based on the `id`. The comparator `(a, b) => a.id - b.id` ensures a numerical sort rather than a lexicographic one, which is crucial as `id`s are integers.

This approach elegantly solves the problem by constructing a `Map`, handling the merging logic through conditions and object spreading, and then returning the sorted array of unique objects. By using `Map`, we can efficiently look up and decide how to handle each object from `arr2`, making the algorithm both straightforward in logic and practical in terms of computation complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above.

Suppose `arr1` and `arr2` are as follows:

```
const arr1 = [{ id: 1, name: 'John', age: 25 }, { id: 2, name: 'Jane' }];
const arr2 = [{ id: 2, city: 'New York' }, { id: 1, age: 26 }, { id: 3, name: 'Kyle' }];
```

We want to merge these arrays into `joinedArray` by following the solution's steps.

1. Initialize a new `Map` and populate it with the `id` and corresponding object from `arr1`:

   ```
   const d = new Map(arr1.map(x => [x.id, x]));
   // Map content after initialization:
   // 1 => { id: 1, name: 'John', age: 25 }
   // 2 => { id: 2, name: 'Jane' }
   ```

   We start with `arr1`, turning it into a `Map` where each object is keyed by its `id`.

2. Iterate through `arr2` and merge or add objects as necessary:

   ```
   arr2.forEach(x => {
       if (d.has(x.id)) {
           d.set(x.id, { ...d.get(x.id), ...x });
       } else {
           d.set(x.id, x);
       }
   });

   // Map content after processing arr2:
   // 1 => { id: 1, name: 'John', age: 26 }
   // 2 => { id: 2, name: 'Jane', city: 'New York' }
   // 3 => { id: 3, name: 'Kyle' }
   ```

   As we process `arr2`, we check if an `id` is already in the map:

   - For `id: 2`, we find it in the map and merge the object with `{ city: 'New York' }`. Jane now also has a `city` property.
   - For `id: 1`, we merge and update John's `age` to 26.
   - `id: 3` is new, so we add `{ id: 3, name: 'Kyle' }` to the map.

3. Transform the `Map` into an array and sort the objects by `id`:

   ```
   return [...d.values()].sort((a, b) => a.id - b.id);
   // Resulting joinedArray:
   // [{ id: 1, name: 'John', age: 26 },
   //  { id: 2, name: 'Jane', city: 'New York' },
   //  { id: 3, name: 'Kyle' }]
   ```

   Finally, we convert the `Map` back into an array of values and sort this array by `id`. This gives us the correctly merged and ordered `joinedArray`.

Through this example, we've seen the effectiveness of using a `Map` to identify unique objects and merge them when necessary, ensuring that `arr2` has precedence in properties, and finished by sorting the `joinedArray` in ascending order by `id`.

Python Solution

```python
def join(array1, array2):
    # Create a dictionary to hold merged objects,
    # using 'id' as the key for fast access.
    merged_data = {}

    # Process the first array and map each object's 'id' to itself.
    for element in array1:
        merged_data[element['id']] = element

    # Iterate through the second array.
    for element in array2:
        # If the 'id' already exists in the dictionary, merge the current object
        # with the existing one by updating the dictionary at this 'id' key.
        if element['id'] in merged_data:
            existing_element = merged_data[element['id']]
            # Merge existing_element with element. In case of conflicting keys,
            # the values from element will update those from existing_element.
            merged_data[element['id']] = {**existing_element, **element}
        else:
            # If the 'id' is new, add the current object to the dictionary.
            merged_data[element['id']] = element

    # Convert the merged data back to a list, then sort by 'id' and return.
    # Sorting is done by using a lambda function that extracts the 'id' for comparison.
    return sorted(merged_data.values(), key=lambda x: x['id'])

```

Java Solution

```java
import java.util.*;
import java.util.stream.Collectors;

public class ArrayJoiner {

    /**
     * Joins two lists of objects based on their 'id' property, merges objects with the
     * same 'id' from both lists, and includes all unique objects. The resulting list is
     * sorted by the 'id' property in ascending order.
     *
     * @param list1 The first list of objects with 'id' property
     * @param list2 The second list of objects with 'id' property
     * @return A sorted list of merged objects
     */
    public List<Map<String, Object>> join(List<Map<String, Object>> list1, List<Map<String, Object>> list2) {
        // Create a Map to hold merged objects with the 'id' as the key
        Map<Integer, Map<String, Object>> mergedData = new HashMap<>();

        // Process the first list and map each object's 'id' to the object itself
        for (Map<String, Object> element : list1) {
            mergedData.put((Integer) element.get("id"), element);
        }

        // Iterate through the second list
        for (Map<String, Object> element : list2) {
            Integer id = (Integer) element.get("id");
            // If the 'id' already exists in the map, merge the existing object with the current one
            if (mergedData.containsKey(id)) {
                Map<String, Object> existingElement = mergedData.get(id);
                // Combine all keys from both maps, preferring the second element's value if a key collision occurs
                Map<String, Object> combinedElement = new HashMap<>(existingElement);
                combinedElement.putAll(element);
                mergedData.put(id, combinedElement);
            } else {
                // If the 'id' is new, add the current object to the map
                mergedData.put(id, element);
            }
        }

        // Convert the merged map to a list and sort it by 'id' in ascending order
        return mergedData.values().stream()
                .sorted(Comparator.comparingInt(element -> (Integer) element.get("id")))
                .collect(Collectors.toList());
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <map>
#include <algorithm>

// Function to join two vectors of objects based on their 'id' property.
// It merges objects with the same 'id' and includes all unique objects from both vectors.
// The function also sorts the resulting vector by the 'id' property in ascending order.
std::vector<std::map<std::string, int>> Join(
    const std::vector<std::map<std::string, int>>& array1,
    const std::vector<std::map<std::string, int>>& array2) {

    // Create a map to hold merged objects, using 'id' as the key.
    std::map<int, std::map<std::string, int>> mergedData;

    // Process the first vector and map each object's 'id' to itself.
    for (const auto& element : array1) {
        mergedData[element.at("id")] = element;
    }

    // Iterate through the second vector.
    for (const auto& element : array2) {
        // If the 'id' already exists in the map, merge the existing object with the current one.
        if (mergedData.find(element.at("id")) != mergedData.end()) {
            for (const auto& pair : element) {
                mergedData[element.at("id")][pair.first] = pair.second;
            }
        } else {
            // If the 'id' is new, add the current object to the map.
            mergedData[element.at("id")] = element;
        }
    }

    // Create a vector to hold the merged objects for sorting.
    std::vector<std::map<std::string, int>> mergedVector;

    // Extract values from the map and push them into the vector.
    for (const auto& pair : mergedData) {
        mergedVector.push_back(pair.second);
    }

    // Sort the vector by the 'id' in ascending order.
    std::sort(mergedVector.begin(), mergedVector.end(),
        [](const std::map<std::string, int>& a, const std::map<std::string, int>& b) {
            return a.at("id") < b.at("id");
        });

    return mergedVector;
}

```

Typescript Solution

```typescript
// Function to join two arrays of objects based on their 'id' property.
// It merges objects with the same 'id' and includes all unique objects from both arrays.
// The function also sorts the resulting array by the 'id' property in ascending order.
function join(array1: any[], array2: any[]): any[] {
    // Create a Map to hold merged objects, using 'id' as the key.
    const mergedData = new Map<number, any>();

    // Process the first array and map each object's 'id' to itself.
    array1.forEach(element => mergedData.set(element.id, element));

    // Iterate through the second array.
    array2.forEach(element => {
        // If the 'id' already exists in the map, merge the existing object with the current one.
        if (mergedData.has(element.id)) {
            const existingElement = mergedData.get(element.id);
            mergedData.set(element.id, { ...existingElement, ...element });
        } else {
            // If the 'id' is new, add the current object to the map.
            mergedData.set(element.id, element);
        }
    });

    // Return a sorted array of the merged objects, based on their 'id'.
    return Array.from(mergedData.values()).sort((elementA, elementB) => elementA.id - elementB.id);
}

```

Time and Space Complexity

Time Complexity:

1. The time complexity for creating the Map `d` from `arr1` is `O(n)`, where `n` is the number of elements in `arr1`. This involves iterating over `arr1` and inserting each element into the Map.

2. The time complexity for the `forEach` loop over `arr2` is `O(m)`, where `m` is the number of elements in `arr2`. Inside this loop, checking for the existence of an element with `has` and updating or setting with `set` is `O(1)` because Maps in TypeScript/JavaScript typically provide these operations with constant time complexity.

3. The spread operator `...` used in the merge `{ ...d.get(x.id), ...x }` has a time complexity that is linear to the number of properties in the objects being merged. Since this is inside the loop, its impact depends on the size of objects; if we assume they have `k` properties on average, this operation would have a complexity of `O(k)` every time it is executed.

4. The `sort` function has a worst-case time complexity of `O(p log(p))`, where `p` is the number of elements in the resulting array which can be at most `n + m`.

Overall, the time complexity would be `O(n) + O(m) + O(mk) + O((n+m) log(n+m))`. Assuming `k` is not very large and can be considered nearly constant, we can simplify this to `O((n+m) log(n+m))`.

Space Complexity:

1. The space complexity for the Map `d` involves storing up to `n+m` elements, giving a space complexity of `O(n+m)`.

2. If the merge `{ ...d.get(x.id), ...x }` creates new objects, this happens `m` times at most, but does not increase the overall number of keys in the final map, so the space complexity remains `O(n+m)` for the Map itself.

3. The array returned by `[...d.values()]` will contain at most `n+m` elements, so this is `O(n+m)`.

Given these considerations, the overall space complexity of the function is `O(n+m)`.