DAA CASE STUDY

2211CS020470 AIML-SIGMA

Overview

An e-commerce platform needs to implement a sorting feature for products based on various attributes like price, rating, and name. With millions of products in the catalog, ensuring that the sorting operation is efficient and scalable is crucial. Sorting operations, especially when dealing with such large datasets, must be chosen carefully based on the data characteristics and the trade-offs in terms of time and space complexities.

In this section, we will discuss the time and space complexities of commonly used sorting algorithms (Quick Sort and Merge Sort), and how the characteristics of the data (such as price range, rating precision, and name lengths) influence the choice of sorting algorithms.

1. Time and Space Complexities of Commonly Used Sorting Algorithms

Quick Sort:

- **Time Complexity:**
  - Best Case: O(n log n) when the pivot is well-chosen (e.g., balanced partitions).
  - Average Case: O(n log n), which occurs when the pivot splits the array into reasonably balanced partitions on average.
  - Worst Case: O(n^2), which happens when the pivot is the smallest or largest element (e.g., when the array is already sorted or nearly sorted).

- **Space Complexity:**
  - Auxiliary Space: O(log n) for the recursive stack in the best and average cases.
  - Worst-case space complexity can be O(n) if the recursion is very deep (i.e., the pivot is always the smallest or largest element).

Merge Sort:

- **Time Complexity:**
  - Best Case: O(n log n), as the algorithm always divides the array in half, regardless of the order of elements.
  - Average Case: O(n log n), since each merge step operates in linear time, and the total number of merge steps is logarithmic.
  - Worst Case: O(n log n), the algorithm performs the same amount of work in the worst case as in the average case.

- **Space Complexity:**
  - Auxiliary Space: O(n) because the algorithm requires extra space to store the merged arrays, in addition to the space used by the recursion stack.

2. Impact of Data Characteristics on Sorting Algorithm Choice

The characteristics of the data, such as the range of prices, lengths of product names, and the precision of ratings, can significantly impact the selection of the appropriate sorting algorithm. Let's explore how each of these attributes affects the decision:

- **Price Range:**
  - Large Range: If the range of prices is large (e.g., ranging from $1 to $100,000), comparison-based algorithms like Quick Sort and Merge Sort are generally the best choice because they efficiently handle wide numerical ranges.
    - Quick Sort would generally perform better due to its lower auxiliary space requirements, but it may suffer from worst-case performance if the pivot selection is poor.
    - Merge Sort is more stable and guarantees O(n log n) time complexity, which is useful if stability is important.

- **Product Name Lengths:**
  - Short Strings: If product names are relatively short, traditional sorting algorithms like Quick Sort and Merge Sort work efficiently.
  - Long Strings: For longer product names or when dealing with lexicographical sorting, algorithms that compare strings character by character (like Merge Sort) are effective. Radix Sort could be considered if the names have a fixed length, as it may provide linear time complexity, but this requires additional memory for the auxiliary array.

- **Rating Precision:**
  - Decimal Numbers: Ratings are often decimal values (e.g., 4.5 stars). Quick Sort and Merge Sort can handle these values as they operate in a similar way for floating-point numbers as they do for integers. If ratings have high precision (e.g., to the hundredth decimal), sorting algorithms like Merge Sort might be more stable in case of tie-breaking scenarios.

- **Multi-level Sorting:**
  When sorting by multiple attributes (e.g., first by price, then by rating, and then by name), a stable sorting algorithm like Merge Sort is often chosen to maintain the relative order between products with the same value for one attribute but a different value for another. This ensures that products sorted by price are still sorted by rating or name as needed.

C++ Code Example:

Quick Sort Example in C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Quick Sort helper function
int partition(vector<pair<string, double>> &arr, int low, int high) {
    double pivot = arr[high].second;  // price as the key for sorting
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j].second <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Quick Sort function
void quickSort(vector<pair<string, double>> &arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<pair<string, double>> products = {{"Product A", 20.99}, {"Product B", 10.50},
{"Product C", 25.00}};

    quickSort(products, 0, products.size() - 1);

    cout << "Sorted Products by Price (Quick Sort):" << endl;
    for (auto &product : products) {
        cout << product.first << " - $" << product.second << endl;
    }
```

```
    return 0;
}
```

Merge Sort Example in C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Merge function for Merge Sort
void merge(vector<pair<string, double>> &arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<pair<string, double>> leftArr(n1), rightArr(n2);

    for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
    for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i].second <= rightArr[j].second) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// Merge Sort function
```

```
void mergeSort(vector<pair<string, double>> &arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    vector<pair<string, double>> products = {{"Product A", 20.99}, {"Product B", 10.50},
{"Product C", 25.00}};

    mergeSort(products, 0, products.size() - 1);

    cout << "Sorted Products by Price (Merge Sort):" << endl;
    for (auto &product : products) {
        cout << product.first << " - $" << product.second << endl;
    }
    return 0;
}
```