Avraham (Avi) Amon

Data Structures

HW 3

9/23/2016

If I were designing both the regional and central databases, I would design them to be as efficient as possible. In other words, the structures I would use correspond to the actions that the databases will do the most.

In the regional databases, I would implement arrays. The reason for this is because we would constantly want to be able to access information in them. We don’t want to delete anything at all, but we would want to add. But because we don’t care about sorting the transactions (in the regional DBs at least), we can constantly add elements without having to worry about shifting the indices of the transactions. Basically, I would create an arbitrarily large array and assign the transaction to the index with the first null value. Once all null values are taken up, which would happen every once in a while, we simply double the array, shift everything over, and continue going as usual. We are doing random access more, as a customer can call at any time asking for their purchase, so that is more important. Adding elements would be O(1) time because we are just assigning a value to an index, but plus O(n) in the worst case that we need to double the array size, and accessing information would also be O(1) as we would be able to search for the transaction from the index. In order to prevent altering, you can just set up a conditional statement that would prevent assigning a value to the array if it doesn’t have null value unless you enter a key which the manager would have. Then, altering would just be O(1) time as the user would just be changing a value at an index in the array, i.e., assigning a new value to that index. At the end of the day, I would copy all of the values corresponding to that day’s transactions from each region into a single list, the reason for I will explain in the next paragraph. The time would be O(n) as it would be going through 3 separate loops, one for each region, and assigning the transaction stored there to be an element (its own node) in a list. Because no loops are nested, as we are going through each region one at a time, the time would be O(3n) which is just O(n).

At corporate headquarters, I would implement an array of lists, where each list corresponds to a customer, and each node on in the list would be a reference to a single transaction in the regional databases. First, every morning, the product line managers at HQ will take the new list that was created by the previous day’s transaction and loop through it to review it. The time will be O(n), and this would be true regardless of whether or not we used an array or a list. But next, the managers at HQ will do a number of things. First, they will combine all transactions from a single customer and store it in their own database. They will also be doing some deleting of the component transactions, and they will store all transactions by a single customer together in a single place. The DB implemented at HQ is an array where each index refers to a customer, and at that index is stored a list. That way, we loop through the combined list we received each day from the regions, and at each node, we add the transaction to the list at each customer’s index. We then delete component transactions. Because we only want to retrieve information occasionally, since we know the customer we want, we can use his index to access the outer array, and then we’d loop through the list stored at the index until we find the transaction we are looking for. We won’t be searching so much for a specific transaction so using a list inside is more useful as it allows us to constantly add transactions at O(1) time each day without having to worry about resizing it like for an array. As for the outer array, we won’t be adding nearly as many customers as we will transactions so it makes sense to use an array for quick random access to be able to add elements to the list. If we run out of room in the array, we do the same trick we did in the regional databases, and just make a new array of 2 times the size of the current one. Since we can get the customer index, adding elements to the inner list will be O(1) time as we are just adding nodes. Deleting is also O(1) as we are just getting rid of a node from the list. Retrieval will be O(n) times as the random access is O(1) and looping through a list is O(n).

The new summary/roll-up view of the day’s transaction will be stored in one of the indices of the array at HQ, and it will also be a list. So in all but one of the indices of the HQ array database, each one will be a list representing a customer, and the other one will be a list of the aggregate data and IDs. Both retrieval and creating the aggregate data would take O(n) time as we’d have to loop through the elements once to create the aggregate list, and then finding an element in a list is O(n) time.

Every week, they can drill into the details of this new list and loop through to find the aggregate transaction needed. That would take O(n) time as even though we have the index for the list (we will presumably keep track of the one list stored in the array that is different than the others), we’d still have to loop through the list to find the element we want. We can then undo an aggregate by retrieving the component from the regional DB. This would take O(n) time to go through the list to get the ID, but O(1) time to get it from the regional DB as once we have the ID, and since the regions are using arrays, we have fast access. Reinserting is also fast since we have the index for the array at HQ, and adding to a list is just O(1) time. The same is true about deleting. It too is O(1) time.

For creation of a new summary roll up, it should take O(n) time, even if he only wants certain data. We would access our summary list in the array, and loop through it once, only adding to a new list the things we want. Since adding to a list and accessing the array where we have the index is only O(1) time, the only time consuming part would be the O(n) traversal through the list.