Historical Min/Max Range-Sum Queries

Aseem R. Baranwal and Trevor Clokie

School of Computer Science
University of Waterloo

Abstract. In this project we start with a survey of the recent literature on array range queries. Then, we formulate a specific problem of answering historical min/max range sum queries on arrays. We present a solution to this problem involving persistent segment trees.

Keywords: Range queries · sum · persistence.

1 Introduction

Range-sum queries are one of the most interesting problems in the data-structure design space. They have numerous theoretical and practical applications in computational geometry and graphics. For example, finding the most intensely coloured parts of an image may utilize 2-D range-sum (density) queries on the image-matrix. We present a classification of range queries based on the work on Skala [9] as part of the survey, and also talk about the range min/max queries and range median queries. This serves as a precursor to our project and provides us with few ideas to explore. In Section 4, we design our problem formally, extending from an initially non-persistent version of the problem. We also show a reduction of this problem to a more general range-sum query problem on arrays. Next, we propose ideas for solving the problem in Section 5. We also attempt to optimize the solution and give further potential ideas.

2 Survey of recent work

We start with the survey by Skala [9], exploring the classification of array range queries and looking at two specific problems.

2.1 Classification of range queries

Since this problem space is so vast, we consider Skala's classification scheme for array range queries to categorize them. Some kinds of array range queries may include finding a particular element in a given range, such as the maximum, minimum, median, or mode. Other array range queries might return a value for the range that is not necessarily found in the range, such as the sum, mean, variance, or density[1] of all elements in the range.

[1] The density of a range is the number or proportion of elements in the range that are non-zero, non-empty, or otherwise non-trivial.

1. **Weight queries**: These are based on the actual values of the elements. Some examples of weight queries include minimum, maximum, median, and sum range queries.
2. **Colour queries**: These are concerned with only the existence of elements, and not the values or frequencies thereof. Examples of colour queries may include density range queries and counting range queries.
3. **Frequency queries**: Output of these queries depend not on the actual values of elements, but on the number of times they occur in a given range. An example of this would be the mode range query.

We note that there may be an overlap between these types, but consideration of this fact is unnecessary for our purposes. One example of such a query might be: *how many positive integers with value greater than their frequency are present in a given range?*

3 Solutions to specific problems

There are some generally well-known and preliminary techniques for solving range queries. If the values of the elements are large, but the number of elements is small, then *Rank-Space Reduction* [6] is a technique that lets the core data structure deal with values proportional to the number of values. This gives solutions with output-sensitive complexity, which is usually desired. Another useful data-structure known as *Wavelet Trees* [8] are helpful with succinct range queries (rank and select) on vectors with non-binary alphabets. In this section, we discuss some pre-existing solutions to two well-studied problems, which will help us to formulate our problem and work on its solution.

3.1 Range max/min queries

First, we examine the range maximum query, and its equivalent, the range minimum query. This problem involves returning the index of the element with the maximum value in a given range. As this is one of the simplest range queries, it is one of the most well-studied problems in this area. One good solution to this problem makes use of range query trees; also commonly referred to as segment trees [1]. The core idea with segment trees is to view the complete array as an interval, and split it recursively into half, with nodes storing information about all sub-intervals, up to the leaves which store information about individual elements. A query interval is then a combination of a logarithmic number of nodes in this tree.

3.2 Range median queries

The most notable solution to the range median problem was given by Beat Gfeller and Peter Sanders [7]. Their core idea is to reduce the range median problem to a logarithmic number of range counting problems with a lazy pre-processing.

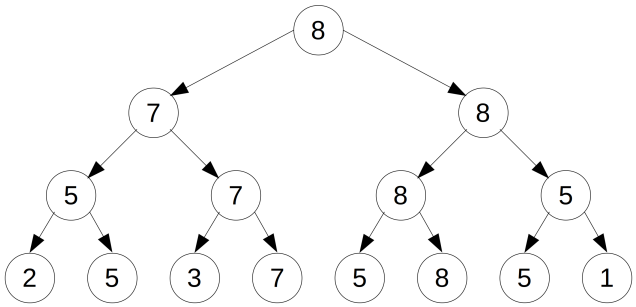


Fig. 1. An example of a segment tree for solving range max queries. Each internal node has weight equal to the maximum of its two child nodes' weights.

Since each recursive search list encountered in their method is a sub-list of the previous list, we can leverage a standard technique to search for the same key in all these lists. This technique is called *Fractional Cascading*, given by Chazelle and Guibas [3].

Later in 2011, Brodal et al. [2] gave an improvement on this method where they reduced the range counting problems to rank computations in bit-arrays. In doing so, the recursive sub-problems are compressed in such a way that ensures that the bit-arrays remain sufficiently dense at all times. This improvement led us to think more on if we can optimize the range-sum queries historically for bit-vectors as a special case.

4 Formulating the historical-max range-sum problem

4.1 Ideas from solutions in Section 3

Let us start with a very basic version of the problem for numeric arrays.

Problem 1. Let A be an array consisting of n numbers. There are two types of operations that we want to support efficiently on the array.

1. `QUERY`(l, r): Report the sum of elements in the range $[l \dots r]$, and
2. `UPDATE`(i, x): Update the value of A_i to x, for some $i \geq 0$.

This problem is easily solvable using segment-trees, following along the ideas of a typical RMQ solution that we have already seen in Section 3.1. We achieve $O(\log n)$ time for both query and update operations. To make this problem more interesting, we **add persistence** to the format of the queries.

4.2 Adding persistence

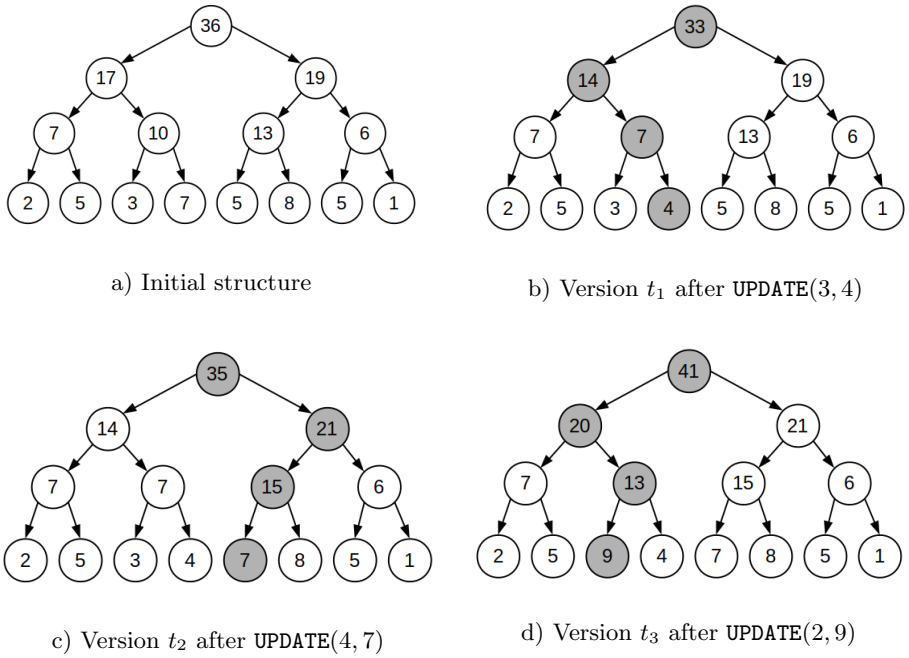
Problem 2. Given an array A of n elements, design a data structure to support the following operations efficiently.

1. `QUERY`(l, r, v): Report the sum of elements in the range $[l \dots r]$ that was seen after the v^{th} update operation, and
2. `UPDATE`(i, x): Update the value of A_i to x. This creates a new version instead of simply updating the nodes.

A segment-tree satisfies the conditions required by a data structure to be made persistence according to Driscoll et al. [5]. Below we present the design for the same. Recall that all nodes of a segment-tree store information about intervals. For the range-sum problem, let us consider the example below.

Example 1. Let $A = \{2, 5, 3, 7, 5, 8, 5, 1\}$ be the initial array that we operate on. The pre-processed segment tree for this array is given by Figure 2a. This is the initial version of the segment tree and we denote it by t_0. The root stores the sum of the entire array while the leaves denote individual elements of the array. We denote the i^{th} segment tree Now let us see how the tree changes when we receive update operations. Figure 2b shows the node values updated for the operation `UPDATE`$(3, 4)$. Subsequent figures 2c and 2d depict similar modifications for operations `UPDATE`$(4, 7)$, and `UPDATE`$(2, 9)$ respectively.

Time and Space requirement : As clearly observed, we only need to change a logarithmic number of nodes for each update operation. Querying a range for the sum is done in the usual way mentioned in Section 3.1, which also takes logarithmic time. The only additional resource we are using is space. Apart from the original values, we require $O(\log n)$ new nodes for each update operation. We also maintain an array of pointers to all versions of the structure, requiring $O(u)$ space, where u is the number of update operations. We depict the actual persistent data structure for Example 1 in Figure 3 to clarify this.



a) Initial structure

b) Version t_1 after `UPDATE`$(3, 4)$

c) Version t_2 after `UPDATE`$(4, 7)$

d) Version t_3 after `UPDATE`$(2, 9)$

Fig. 2. Segment-tree operating on the array in Example 1

– Time Complexity: $O(\log n)$.
– Space Complexity: $O(n + u \log n)$, u = number of update operations.

We represent the initial version with t_0, and we represent the version after v^{th} update operation by t_v. The crucial thing to note here is that we do not copy the whole structure for an update operation. Instead, we only create new nodes for those intervals that are modified because of the update. For intervals that are not affected, we simply point to the existing nodes from previous versions.

4.3 Defining the problem

Now that we have a solution to the problem of finding the sum of a range for a given version of the data structure, we are ready to propose the final modification that will create an unsolved problem for us to work on. Apart from finding the sum of a range for a given version, we are also interested in the maximum sum that the range has ever witnessed across all versions. We call this the *Historical-Max-Sum* range-query problem. Formally, we define the problem below.

Problem 3. Given an array A consisting of integer elements, design a data structure to efficiently support the following operations:

1. `SUM`(v, l, r): Report the sum of elements in the range $[l \dots r]$ in version v.

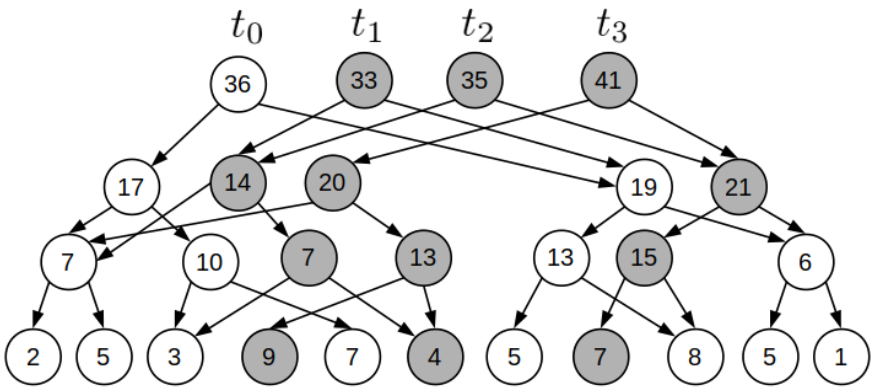


Fig. 3. The persistent structure with all pointers

2. `MAX-SUM`(l, r): Report the highest sum of elements that the range $[l \dots r]$ has had across all versions.
3. `UPDATE`(i, e): Modify the i^{th} element, $A_i = e$.

The challenge :
Suppose we create a persistent structure for the elements. Consider the case where the query interval spans multiple nodes. Clearly, it is not straightforward to compute the historically-maximum sum because each node's individual maximum sum may belong to different versions, while we require the sum from a single version. This is the hardest part and the core of our problem, and this is what most of the researching time is spent on. After going through several bad ideas, we think one of our ideas is worth exploring.

5 Ideas for solutions

We propose a natural solution which is what one could start with, and then present an improvement that consumes only a constant times more space using additional pointers in our structure. Further, we propose an algorithm which we claim to be a substantial optimization, but we are not yet able to provide the complete proof. We would like to work on this problem beyond the scope of this project to completely finish the solution with proof of complexity. In theory, at least a better probabilistic time complexity may be derived for the proposed method.

5.1 Designing the solution

A very obvious direction to think in is using the existing persistent segment tree structure that we know of for version based range-sum queries. Let us design the model with which we will work. **Each node in the tree stores** 4 **items**:

- `left`: A pointer to the left child of this node.
- `right`: A pointer to the right child of this node.
- `sum`: The sum of elements in the interval represented by this node.
- `next`: A pointer to the next version of this interval node.

To keep track of roots of all versions, we maintain an array: $version$, and $version[i]$ stores the pointer to the root of the i^{th} version. We will start with describing the algorithm for pre-processing this structure for a given array. Algorithm 1 is the corresponding procedure. It is called initially with parameters $low = 1$ and $high = n$. After the algorithm finishes, the root of the tree is the pointer to the initial version of our persistent structure.

Algorithm 1: PREPROCESS$(A, root, low, high)$

Data:
A: The array of bits.
$root$: pointer to the root node.
low: lower end of the interval.
$high$: higher end of the interval.

Result:
Creates a global persistent structure to be utilized while performing updates and answering queries.

```
1  if low = high then
2  |  // This is the leaf node which denotes a single element.
3  |  root.sum = A[low];
4  |  root.left = root.right = NULL;
5  |  return;
6  end
7  mid := ⌊(low + high)/2⌋;
8  root.left := new Node();
9  root.right := new Node();
10 // Recursively pre-process child nodes.
11 PREPROCESS(A, root.left, low, mid);
12 PREPROCESS(A, root.right, mid + 1, high);
13 root.sum = root.left.sum + root.right.sum;
```

Space/Time analysis for pre-processing

- **Time complexity**: We start from the root, and recursively build the tree in a bottom-up approach. At each recursion level, we split the problem in two halves and process them individually to combine the result later. Hence the time complexity is given by:

$$T_p(n) = 2T_p\left(\frac{n}{2}\right) + O(1) = O(n)$$

– **Space requirement**: Each node takes up constant space, and the number of nodes (starting from leaves) is given by:

$$S_p(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 = O(n)$$

The update operation is performed using Algorithm 2 as provided. Note that for each update, we only need to traverse from the root to that elemental node, hence we do not touch more than $O(\log n)$ nodes in any update. For every update, the procedure is called with $low = 1$ and $high = n$. it is definite that the sum of the whole array will need to be updated, hence we start with the root. Depending on the location of the index to be updated, we recurse into the left or right child until the leaf (element which is updated) is reached. We maintain unchanged values from the previous version simply by pointing to the corresponding nodes of the previous version (as done in standard persistent segment trees). Also note that the **pointer field *next*** (assigned in line 19) is used to help us directly access versions of a given interval node, which will be useful when querying for historically maximum sum.

Space/Time analysis for update operation

– **Time complexity**: We recurse to the leaf which is to be updated. In a bottom-up approach, we create new nodes for all interval nodes that need to be updated in the path from that leaf up to the root. Since the height of the data structure is $O(\log n)$, we spend logarithmic time for this operation.

$$T_u(n) = T_u\left(\frac{n}{2}\right) + O(1) = O(\log n)$$

– **Space requirement**: Each update operation creates logarithmic number of new nodes for the interval nodes that lie in the path from the updated leaf to the root, and hence additional space for each update is given by:

$$S_u(n) = O(\log n)$$

Next, we have the standard Algorithm 3, defined below which gives the range-sum of a specific version. The sum procedure takes as input a range $[l \ldots r]$ and the root of the specific version tree, and outputs the sum of the queried range in that version of the data structure.

Algorithm 2: UPDATE($prev, root, index, value, low, high$)

Data:
$prev$: Pointer to root node of the previous version.
$root$: Pointer to root node of the current version.
$index$: $A[index]$ will be updated.
$value$: New value for $A[index]$.
low: lower end of the interval
$high$: higher end of the interval

Result:
Creates a new version of the structure after the update.

```
1  if low = high then
2  |  // We have reached the leaf node.
3  |  root.sum := value;
4  |  return;
5  end
6  mid := ⌊(low + high)/2⌋;
7  if index ≤ mid then
8  |  root.right := prev.right;
9  |  root.left := new Node();
10 |  UPDATE(prev.left, root.left, index, value, low, mid);
11 |  return;
12 else
13 |  root.left := prev.left;
14 |  root.right := new Node();
15 |  UPDATE(prev.right, root.right, index, value, mid + 1, high);
16 |  return;
17 end
18 root.sum := root.left.sum + root.right.sum;
19 prev.next := root;
```

Algorithm 3: RANGE-SUM($root, l, r, low, high$)

Data:
$root$: Pointer to root of the tree (of specific version).
l: left end of the query interval.
r: right end of the query interval.
low: lower end of the node interval.
$high$: higher end of the node interval.

Result:
Returns sum of elements in range $[l \dots r]$ for the version of the tree given by the $root$ pointer.

```
1 if l > high or low > r or low > high then
2 |  return 0;
3 end
4 mid := ⌊(low + high)/2⌋;
5 return
6   RANGE-SUM(root.left, l, r, low, mid) +
7   RANGE-SUM(root.right, l, r, mid + 1, high);
```

Note that we recurse to the bottom of the tree for at most two times, because all nodes in between will completely be a part of the query interval. We can visualize this via an extension of Example 1, with *next* pointers chaining all the versions of each interval node.

Space/Time analysis for version query

- **Time complexity**: At each level of the tree, at most two nodes are recursively expanded in the query algorithm. Thus, at each level at most 4 nodes are processed. The height of the tree is $\log n$, hence total time consumed is:

$$T_v = O(4 \log n) = O(\log n)$$

- **Space requirement**: There are no extra space requirements for queries to be answered. If there are a total of U update operations being performed, the complete structure takes $O(U \log n)$ space for the new nodes.

For the final type of query, we are given just the range end-points l and r, and the goal is to report the maximum sum that this range has seen across all versions. For instance, let us compare this problem with the regular range-max problem. Historical range-sum is much harder than historical range-max because the range-max problem has this really nice property for any two sets of numbers S_1 and S_2:

$$\max(S_1 \cup S_2)) \in \{\max(S_1), \max(S_2)\}.$$

Furthermore, this property is invariant across versions, meaning that it holds even if the two sets belong to different versions entirely. For our range-sum case, this is not true, hence there seems to be no direct, sub-linear way to compute

the historically maximum sum. We can solve the problem in time linear to the number of updates that have been performed up to now (we call this U). Then computing the maximum sum is similar to computing the version specific sum.

Naive Solution : The most preliminary solution is to query for the interval in all versions and take the maximum of them. This takes $4U \log n$ steps in the worst case. A slightly better approach is to use the `next` pointer fields of our node structure.

Slightly better solution : Determining the historically maximum sum is done as follows.

1. Gather all interval nodes $X_i$in the original version that collectively form the query interval $[l \dots r]$. This takes at most $4 \log n$ steps.
2. For each of the nodes X_i, maximize the value of $\sum_i X_i[j]$ over j = version stamp of the data structure by traversing through the `next` pointer field. This takes $U \log n$ time.

5.2 Further improvements

Clearly, we see that the constant factor 4 is reduced to 1 in the second approach above. But it is still very bad. How can we do better than linear time? To think in this direction, we notice a very subtle characteristic of the operations that our data structure has to support, which is that all nodes that combine to form a query interval are **mutually exclusive**, i.e. their representative intervals never overlap.

Defining "candidate" versions : The property above seem quite promising. To shed more light, consider a query interval $[l \dots r]$. Let this interval be a combination of k nodes $X_1, X_2, \dots, X_k$. We can now say that **each update operation on the data structure will update at most one of these nodes**, because they are non-overlapping. Hence, we only keep track of versions where the value of any node X_i decreases in the immediately next version (call these *candidate* versions). This can be done by maintaining additional data about these versions and the value held by the node in that version. This extra data is stored only in the initial version of the tree.

Now, a simple iteration over these *candidate* historically-maximum versions will give us the required solution. For versions v_i stored as the candidates for node X_i, we compute the following maximization:

$$\text{HistMaxSum} = \max_v \Big(\sum_i X_i[v] \Big)$$

5.3 Future work

Our claim is that the method proposed in Section 5.2 is sub-linear, because: we are cutting down on a lot of time by disregarding any versions where the update operation increased a node value. As an adversary, we are not yet able to design a case where this takes linear time.

To sketch the proof, we should perhaps start looking at the number of candidate versions $|v|_i$ that each node X_i can have. If we can prove a bound for $\sum |v_i|$, then our result will be immediate from the method described above.

We would also like to mention a paper by Paul Dietz [4] on fully persistent arrays might be of interest in this regard. It provides a view on persistence beyond the works of Driscoll et al. who worked on persistence with only some specific pointer based data structures.

References

1. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry. Springer, Berlin, Heidelberg (1997). https://doi.org/10.1007/978-3-662-03427-9_1
2. Brodal, G.S., Gfeller, B., Jørgensen, A.G., Sanders, P.: Towards Optimal Range Medians. Theoretical Computer Science **412**(24), 2588–2601 (2011). https://doi.org/10.1016/j.tcs.2010.05.003
3. Chazelle, B., Guibas, L.J.: Fractional cascading i: A data structuring technique. Algorithmica 1 (1986)
4. Dietz, P.F.: Fully persistent arrays. In: Workshop on Algorithms and Data Structures. vol. 382, pp. 67–74 (1989)
5. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. Journal of Computer and System Sciences (1989)
6. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and Related Techniques for Geometry Problems. In: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing. pp. 135–143. STOC '84, ACM, New York, NY, USA (1984). https://doi.org/10.1145/800057.808675
7. Gfeller, B., Sanders, P.: Towards Optimal Range Medians. In: Automata, Languages and Programming. pp. 475–486. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02927-1_40
8. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA (2003)
9. Skala, M.: Array range queries. Springer-Verlag Berlin Heidelberg (2013)