Practical Data Structures for Frontend Applications: When to use Segment Trees

Joseph Crick Follow
May 1, 2018 · 4 min read

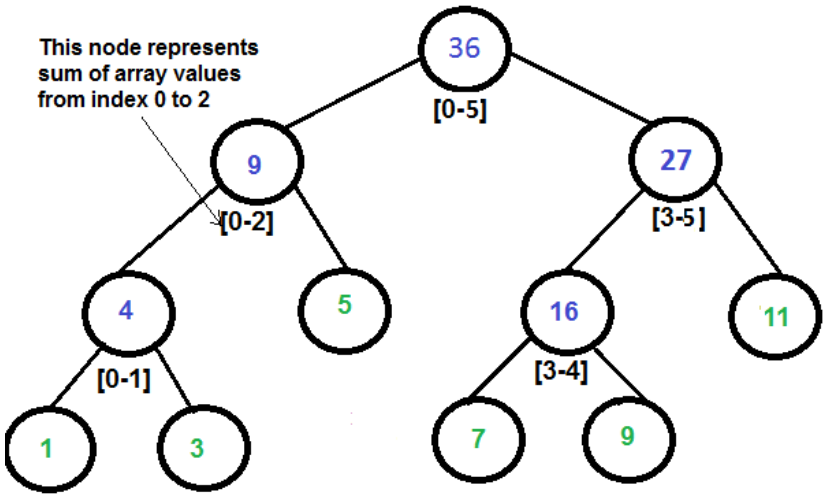
There are a lot of tutorials online showing developers how to write various data structures. There are not a lot of tutorials showing how, when, or whether to use them. In this series, I cover practical uses and implications of data structures in frontend applications. In this edition, we'll review the Segment Tree.

What is a Segment Tree

A Segment Tree is a data structure that can be used to perform range queries and range updates. It is a height-balanced binary tree, usually built on top of an Array. Segment Trees can be used to solve Range Min/Max & Sum Queries and Range Update Queries in O(log n) time.

The Segment Tree works like other tree data structures. It creates query paths that limit the amount of processing required to return data. Each intermediate node of the tree represents a *segment* of the data set. The root node contains the sum of all numbers in the tree. Its children

contain the sums of all the numbers in their respective ranges, and so on down the tree to leaf nodes.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

When to use Segment Trees

Segment Trees are useful whenever you're *frequently* working with ranges of numerical data. The most common use cases for Segment Trees are:

1. Sum all elements in a range.
2. Find the min or max value of elements in a range.
3. Update all elements in a range.

This doesn't mean that using Segment Trees is limited to working with numbers. You can work with Segment Trees, for example, to find all the intervals (or ranges) that match a specific criteria. The classic example of this is the Brackets Problem.

Using Segment Trees in a Frontend App

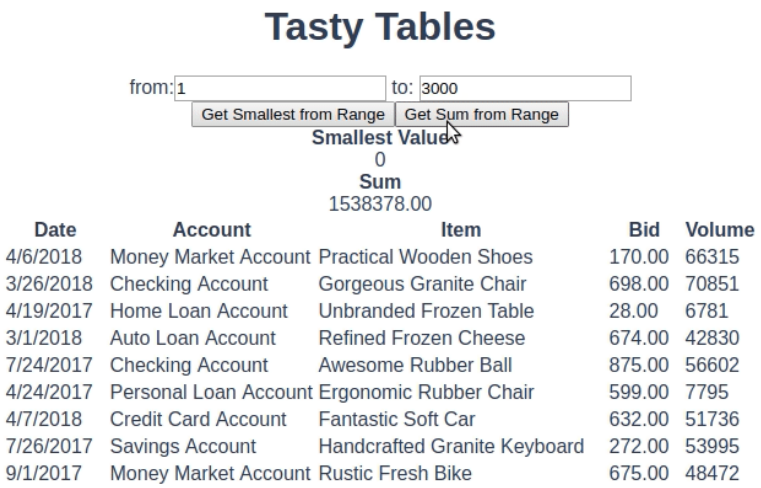
NOTE: *Various JavaScript engines will implement the JavaScript spec. differently. Therefore, from environment to environment performance results may vary.*

The most common way to represent Collections in JavaScript is with Arrays. To find out if it could make sense to use a Segment Tree in a Frontend app, let's contrast using a Segment Tree and an Array for the same task. Here's the criteria we'll use to evaluate them:

- Performance (run time and load time)
- Ease of use, and readability

Setup

- I wrote a quick Bid Grid in Vue, using `vue cli`. Here's what it looks like:



Date	Account	Item	Bid	Volume
4/6/2018	Money Market Account	Practical Wooden Shoes	170.00	66315
3/26/2018	Checking Account	Gorgeous Granite Chair	698.00	70851
4/19/2017	Home Loan Account	Unbranded Frozen Table	28.00	6781
3/1/2018	Auto Loan Account	Refined Frozen Cheese	674.00	42830
7/24/2017	Checking Account	Awesome Rubber Ball	875.00	56602
4/24/2017	Personal Loan Account	Ergonomic Rubber Chair	599.00	7795
4/7/2018	Credit Card Account	Fantastic Soft Car	632.00	51736
7/26/2017	Savings Account	Handcrafted Granite Keyboard	272.00	53995
9/1/2017	Money Market Account	Rustic Fresh Bike	675.00	48472

- I couldn't find a Segment Tree implementation in JavaScript that I liked. So, with a little help from the Algorithms book website, I rolled my own.
- I used `faker` to generate a set of 10,000 bids.

Code

Here's the base code for the grid. Note, it doesn't use any specific collection data structure. Implementation details for the Segment Tree and the Array, follow.

```
<template>
    <div class="hello">
        <h1>{{ msg }}</h1>
        <div>
            <div>
                <label> from:<input type="text" v-mode
                <label> to: <input type="text" v-model
            </div>
            <button v-on:click="getSmallestBidFromRang
            <button v-on:click="getSumFromRange">Get S
        </div>
        <div>
            <span><strong>Smallest Value</strong></spa
            <div>{{smallestValue}}</div>
        </div>
        <div>
            <span><strong>Sum</strong></span>
            <div>{{sum}}</div>
        </div>
        <div class="table">
            <table id="example-1">
                <thead>
                <tr>
                    <th>Date</th>
                    <th>Account</th>
                    <th>Item</th>
                    <th>Bid</th>
                    <th>Volume</th>
                </tr>
                </thead>
                <tbody>
                <tr v-for="item in items">
                    <td>{{item.date.toLocaleDateString
                    <td>{{item.account}}</td>
                    <td>{{item.item}}</td>
                    <td>{{item.bid.toFixed(2).toString
                    <td>{{item.amount}}</td>
                </tr>
                </tbody>
            </table>
        </div>
```

```
    </div>
</template>

<script>
  import faker from "faker";
  import getItems from "./get-items";
```

Here's the Array-based code:

```
methods: {
    getSmallestBidFromRange() {
      const start = parseInt(this.start);
      const end = parseInt(this.end);
      this.smallestValue = this.items
        .filter(bidsInRange(start, end))
        .reduce((cur, prev) => {
          if (prev) {
            return cur;
          } else {
            return cur.bid < prev.bid ? cur : prev;
          }
        }).bid;
    },
    getSumFromRange() {
      const start = parseInt(this.start);
      const end = parseInt(this.end);
      this.sum = this.items
        .filter(bidsInRange(start, end))
        .reduce(
          (prev, cur) => {
            prev.bid += cur.bid;
            return prev;
```

Here's the Segment Tree-based code:

```
    mounted() {
      this.segmentTree = new SegmentTree(this.items, t
    },
    methods: {
      getSmallestBidFromRange() {
        const start = parseInt(this.start);
        const end = parseInt(this.end);
        this.smallestValue = this.segmentTree.rangeMin
      },
      getSumFromRange() {
```

Performance

I tested three things for performance:

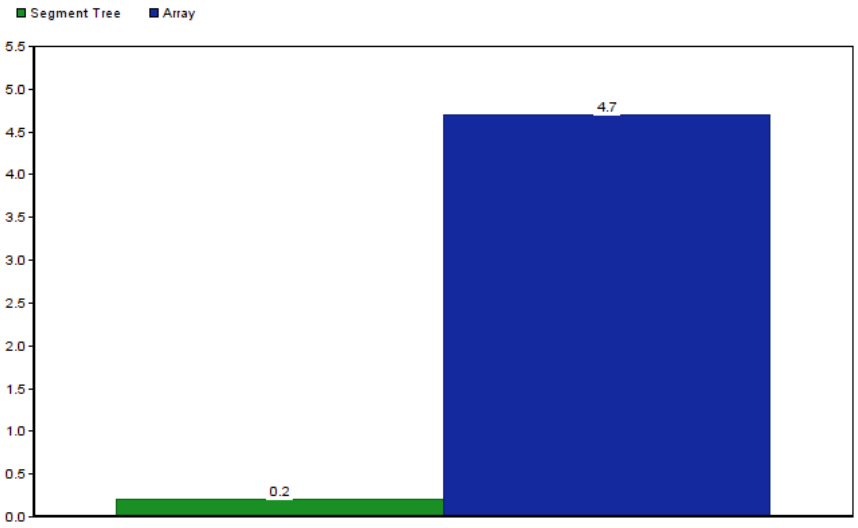
1. Loading the data items into the data structure.
2. Searching the data structure for the min value in a range.
3. Summing the values in a range.

All tests were conducted using Chrome 65.x. The range of data used for each query was 1–3000.

Loading data items
Segment Trees initialize in O(n*log(n)) time. To give you a practical sense of this, it took an average of 2.6 seconds to add 10,000 items to the Segment Tree.

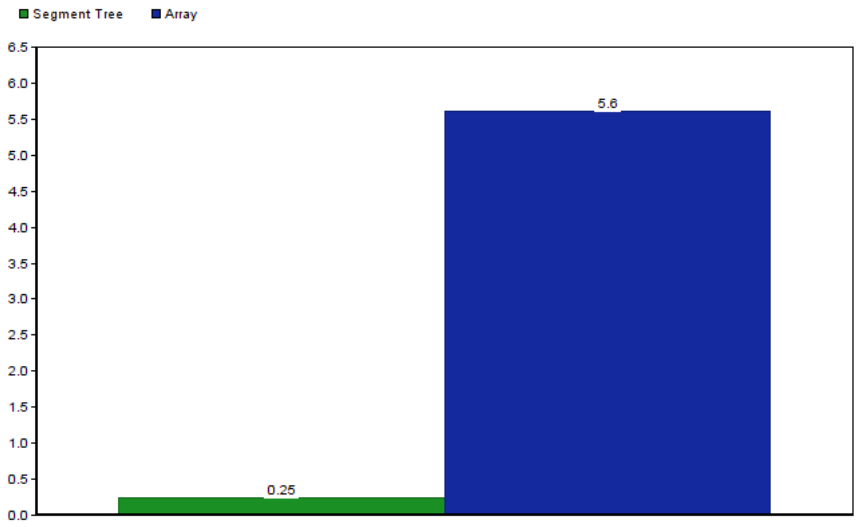
In most cases, on the frontend, data like that in the BidGrid will be provided to an application from a backend API in an Array. In this case, we already have the data in our data structure; there's no need to discuss load time.

Range Minimum Query:
This query finds the smallest value in the range.



The Segment Tree-based query was blisteringly faster than the Array-based query. It was 2,250% faster.

Range Sum:
This query sums all the values in the range.



Again, the Segment Tree was amazing. It was 2,140% faster than the Array method.

Note: In the test above, the initial sum query took around 1 second. All subsequent sum queries were approximately 0.25 seconds—even when the query range was changed.

Ease of Use

Using Segment Tree for this task was easier than using an Array. There was no need to create `filter` or `reduce` methods to get the desired result. The Segment Tree had all the query methods built in.

The code snippet below contrasts the code required for Segment Tree and Array:

```
// Segment Tree
getSmallestBidFromRange() {
  const start = parseInt(this.start);
  const end = parseInt(this.end);
  this.smallestValue = this.segmentTree.rangeMinQuery(
}

// Array
getSmallestBidFromRange() {
  const start = parseInt(this.start);
  const end = parseInt(this.end);
  this.smallestValue = this.items
    .filter(bidsInRange(start, end))
    .reduce((cur, prev) => {
```

Conclusion

The Segment Tree is an *amazing* data structure when you have a search-heavy application that performs a lot of specific range queries on a data set (e.g., sum, min, and max queries). It can definitely make sense to use a Segment Tree in a frontend application, if the needs of the application call for it.

Using a Segment Tree instead of an Array may have some performance costs, such as:

- Segment Tree Initialization. This is a one-time cost for each Segment Tree. Because of this, it is recommended that you defer the initialization of a Segment Tree until after the page has loaded.