Sunday, June 26, 2016

Campspot Scheduling API

Application Instructions and Writeup

WORK STATEMENT

Most destination resorts don't host guests who are willing to stay for only one night. Their guests are not simply passing through the area; they've often travelled far with lots of luggage specifically to stay there, and it's not worth it for them to stay one night. As a result, a reservation system for resorts needs to prevent one night gaps in its reservation grid, or schedule of inventory. If one-night gaps occur on the grid, they will not be sold, and this results in lost income for the resort owner. For example, Campspot uses a one-night gap rule, as this is a common problem for camping resort owners. However, owners' gap problems aren't limited to one-night gaps. Some resorts only host guests who stay a minimum three or four nights. Those resorts may want to prevent one-, two-, and three-night gaps.

To support this use case, Campspot has a configurable business rule called a "gap rule" that allows resort owners to define which types of gaps are not allowed.

BUILDING AND RUNNING THE API

FIRST, CLONE THIS GIT REPOSITORY

```
#clone the repo and change directory to it
$ git clone --depth=1 https://github.com/virtualadrian/campspot-node
campspot-node
$ cd campspot-node
```

RESTORE PACKAGES AND DEPENDENCIES

```
# restore packages and dependencies
# you may need to perform this as root or a sudo user
$ npm install
```

HOW TO START THE APPLICATION

```
# requires gulp to be installed globally
$ npm install -g gulp
# run the api - this should open a browser with an API runner
$ gulp serve
```

At this point you can submit requests to the local API that just started. An API runner along with extra documentation should open in your default browser. It makes requests to the demo endpoint at *http://localhost:3030/api/camp/search-demo*, however the 'original' implementation can be found at:

> http://localhost:3030/api/camp/search
>
> or if you want to try the Interval Tree Search (explained below) you may use:
>
> http://localhost:3030/api/camp/search?interval=1

The api endpoint accepts a POST with a JSON body and returns a response for each gap rule that was sent in. The format for the expected JSON payload is the same as the one that was sent in the original requirement document.

HOW TO EXECUTE UNIT TESTS

```
# mocha was set up as a gulp task to run unit tests
$ gulp mocha

# run tests without linting
$ gulp mocha --nolint
```

OTHER GULP TASK

```
# default: cleans dist & coverage directory. babel compile ES6 to ES5
$ gulp

# lint: use ESLint to verify code
$ gulp lint
```

APPROACH

HIGH LEVEL APPROACH DESCRIPTION

The requirement presented ends up being a need to solve an interval search problem. There are a few approaches that we could take with the type of challenge being presented. I chose to explore two, a linear algorithm and an interval tree search algorithm. (Note: Interval Trees are similar to Binary Trees, I can provide a more in depth explanation if one is desired)

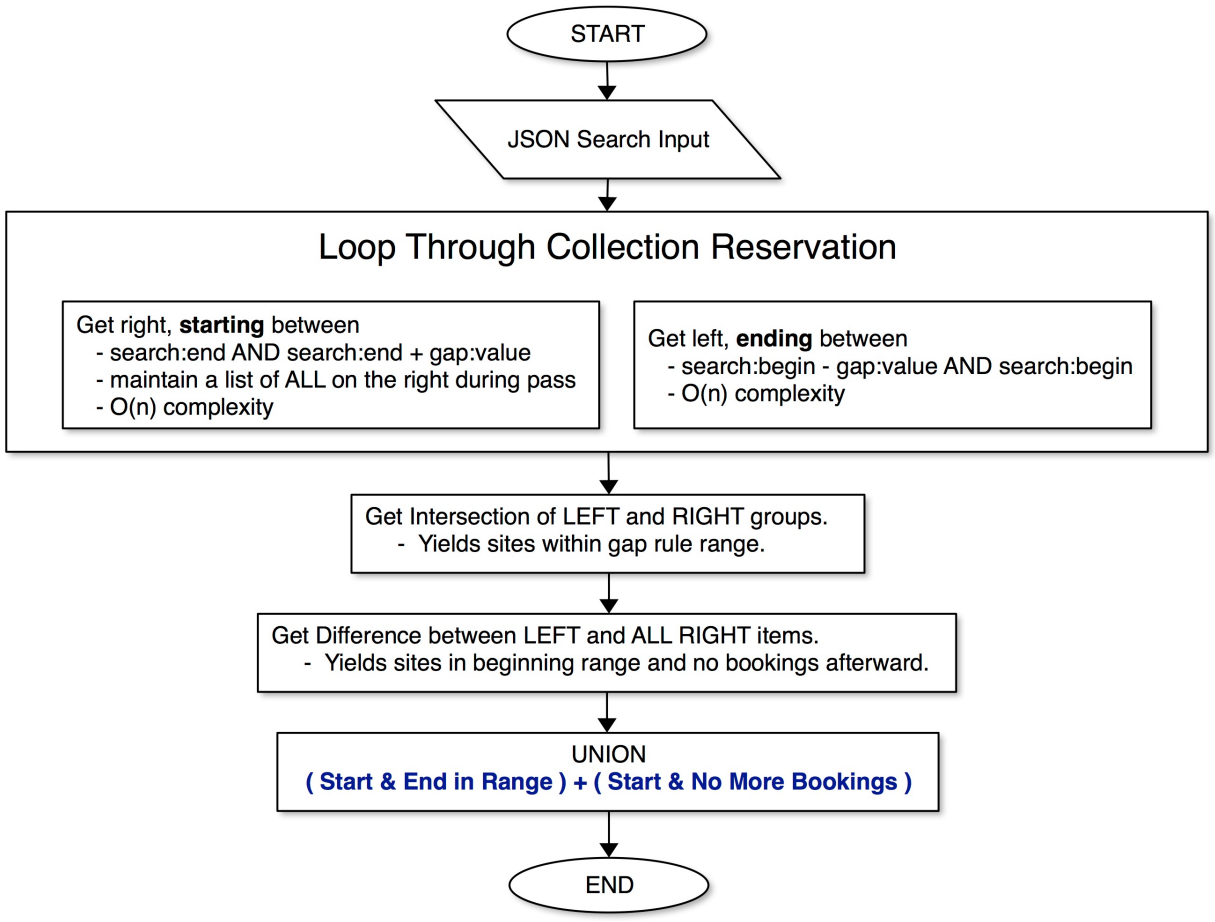
LINEAR SEARCH

The first choice was a linear algorithm where we iterate over the list of reservations and split the items into two groups. A left group, ending before the start point of the search parameters, but

within range of the start minus the gap rule value. A right group, which starts after the ending point of the search parameters, but within range of the end plus the gap rule value.

Additionally we have to track any campsites that do not have a reservation in the right group, meaning that it is completely open. Once we have these sets we can obtain the INTERSECTION of the LEFT and RIGHT groups, which will yield reservations that match our gap rule. Additionally we need to perform get DIFFERENCE between the LEFT set and ALL RIGHT items. This yields any camp sites that are completely open afterward. Finally we UNION the two sets to get our answer, and obviously exclude any overlaps. This resulting performance is an overall of O(n), a linear cost. I am not counting the added performance hit for the INTERSECT, DIFFERENCE, etc. because those are trivial sets numbering in the teens of records.

To get a better visual idea of this approach see the diagram below:



INTERVAL TREE SEARCH

The second choice is to construct an Interval Tree and then search the tree for the same information. This would provide us with a faster search, but we have the added cost to load the tree. That cost is O(n logN), we then have to search the tree which is a cost of O(logN + m), with m being the number of overlapping results.

The added cost would be inconsequential if we searched the data more than one time. One example of this would be if there was an initial load and then we searched over and over. I wanted to try this algorithm because it could have performed faster. However for our use case, loading the interval tree in order to search faster was too great of a cost.

I have both implementations live in the solution and the unit tests run them both. We would have to discuss further where this is being used and if the assumptions, outlined below, that I made are correct.

CHOSEN APPROACH

In the end loading the tree added too much overhead and actually ended up being about 10 times slower than a simple linear search given the size of our data sets and manner of searching. Since the benefit of an Interval Tree would only arise if we had to query the data structure multiple times once it was loaded, that make the cost of the load operation worth it. So, I chose to stick with the Linear Search given the current parameters.

ASSUMPTIONS

The entire approach was dictated by the following assumptions, ergo they are important and could change the solution completely:

- The API would always receive the same type of payload and simply apply the business rules to the data provided.
- There was no possibility of storing this data in a Database and letting the DBMS Optimizer take care of all the logic behind searching, allowing us to simply do a BETWEEN, or 2, in the WHERE clause.
- The reservation information will not be stored for searching later, this validating the choice of a Linear search.
- The client calling our API is capable of receiving a JSON response.
- The chosen technology was NodeJS, ES6, BableJS, etc. because that is the largest part of the current tech stack at Campspot. A similar solution would have been slightly easier to implement using DropWizard/Java or WebAPI/C# since those solutions are a bit more mature in terms of age and enterprise support.

- Final assumption? I was going to enjoy myself if I was going to do this, and I have wanted to throw together my own ES6 Node seed project for a while. I actually started off with a different one I found on GitHub and have been customizing it for the last few days. There's still a lot I'd do to the solution before I'd give it my blessing to move it to higher environments or to other developers, but I believe that what I have thus far fulfills the spirit of the exercise.

WRAP UP

There are a few other things I took into consideration along with what was presented, and I followed only two performance testing approaches. If this is an integral part of the business where we could see some big wins we definitely want to do a little more research.

Also, there is likely an optimal gap rule that should be set and followed, and likely shares a lot of similarity to Price Elasticity. Meaning it likely will vary by market and segment of the population, along with geographical area and so on. This means that we need a way to have those metrics. Long story short, this seems like a small problem at first, but there are very many facets to the business issue. The IT side of it may seem more difficult, but ultimately the business side is a more interesting challenge.