Joel Whalen, Connor Laitinen

Michael Kowalczyk

CS 480 Senior Project

April 25h, 2018

UP Trail Networks

For a Senior Project, we consider ourselves fortunate. Most don't get the opportunity to not only expand their knowledge building and researching a new system, but to help a local Professor and the greater community too. The previous solution of haphazardly throwing together text files (with incorrectly formatted date time information) was not only not working well, but could not be scaled to anything larger. That is what we achieved with UP Trail Networks, a collector of trail usage data, streamlined with graph based analytics and database storage. Everything in the full stack was built from almost nothing, from client to server to database and everything maintaining it in between. In this document we will go through, in detail, the original problem, what we planned to do, what actually happened, and what we could have done better.

The Problem

Spanning the area outside the city of Marquette, Dr. Joshua Thompson of the Math and CS Department at NMU has placed 28 Bluettooth-enabled motion sensors whose purpose is to track the usage of the Marquette City trail network. Each sensor has a known latitude and longitude, and builds a text file that lists a time stamp for every time the motion sensor detects something. There is an Android application that users must use to manually travel to each node and collect these files, as well as transfer the files later. A sample of "TRAIL-4.txt" would look like this:

```
1  12:18:09 11/26/17/4
2  12:18:23 11/26/17/4
…
```

Notice the redundancy of both the incremental index on the left (we could simply get the length of the readlines() function), and the number 4 on the right telling us which trail node the time stamp belongs

to, even though the file is already named accordingly. There is also no location data, so we must depend on an outside dictionary defining the latitude and longitude for each trail node. Most important, however, is the lack of any ISO date standard, which is supposed to be in the form YYYY-MM-DDTHH:MM:SS.

That's it. There is no central repository or file history, no analytics, just raw text data in the wrong format that needs to be collected, updated, and interpreted manually. Furthermore, the time stamp data said nothing of the path an individual would take or how long they spent on the trail network. All we have is a series of time stamps and the individual node locations that recorded them. Our projected solution was to fix all of these problems in one fell swoop, but reality had other plans.

The Projected Solution

We were to build a website running NodeJS with MySQL doing the heavy lifting of running date calculations, data storage, and queries. The already existing Android application would need a file transfer protocol so it could automatically send the files to the Nodejs server. Python would do data analysis to determine a path one may take based on the proximity of nodes and timestamps. All of this would be fed into the client side as graphs and maps with a responsive design, working on multiple devices, to make visualization easier. Throughout all of this, we would collaborate through two separate github repositories: one for the Android application, the other for the website and database.

The Actual Solution

Unfortunately, building the infrastructure to piece all of these moving parts together took the bulk of our time and resources, and as such we had to cut features like path and speed calculations. We also struggled with adapting to Android, a technology that neither Connor or I were very familiar with. That all said, we managed to deliver on everything else.

For our first task of gathering the data, we decided that Android was too unfamiliar and we weren't getting anywhere with the existing code. Fortunately enough, NodeJS supports uploading

groups of files. With a simple form and a get/post, the files are placed onto the server, taken into the database by the Python module, and then removed once database operations are completed.

When it came to converting time stamps into ISO date time objects for our database, we realized MySQL was too unwieldy for the task. It was too difficult to administer (even a simple root password change bricked my MySQL installation) and we didn't need joins or multiple tables because our data was so simple. Also MongoDB is supported across more platforms (especially so for NodeJS) and the documentation was much easier to read, so we switched to Mongo. This was our best course correction besides removing the Android application from our development cycle. Python's MongoDB driver, Pymongo, made chopping up the text files, converting the dates, and placing them into the database a piece of cake. Once we had each line of the file isolated to the time stamp itself, it was one line of Python to convert to ISO datetime:

```
#Apparently MongoDB eats this as a string instead of a date,
#so date conversion must be done after it's been added to the database.
return datetime.strptime(rawstr, "%H:%M:%S %m/%d/%y").isoformat(" ")
```

Notice how the format string in the strptime parameter matches the raw text data. The comment above is important too, because we will have to deal with ISO date format in Javascript as for some reason Mongo could not understand Python's ISO format.

Now that we have our time stamps, we need to match them with the latitude and longitude data and insert each document into MongoDB. We keep a list of latitudes, longitudes, and their associated trail node name in a master csv file, which can be easily updated by Dr. Thompson.

```
71      def write_to_db(self):
72          client = MongoClient(DB_URL + self.db_name)
73          db = client[self.db_name]
74
75          #Insert Latitudes, Longitudes, and Timestamp data
76          for node in self.trail_nodes:
77              node_name = node["node_name"]
78              times = node["timestamps"]
79              db_input = []
80              for time in times:
81                  db_input.append({
82                      "timestamp": time,
83                      "latitude": self.latlongs_data[node_name][0],
84                      "longitude": self.latlongs_data[node_name][1],
85                      "node": node_name
86                  })
87              timestamp_result = db["nodes"].insert_many(db_input)
88
89          pprint("Timestamps insertion acknowledged: " + str(timestamp_result.acknowledged))
```

The downside to inserting the latitude, longitude, and trail node name for each time stamp is that we get a lot of repeated data. We tried to mitigate this by placing a separate collection in the database for each trail node, but that turned out to be a gigantic mess when it came time to execute efficient queries, as MongoDB doesn't support iterating through multiple collections, and manually doing it in Javascript is a nightmare.

In addition to the inefficiency of our data, Mongo isn't creating date time objects out of the ISO dates that Pymongo gives it. Instead they are saved as strings, which makes date time calculations impossible. For a Mongo function like "$gte" to work with date data, it must be a BSON ISODate object. According to the documentation, BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB. After much stackoverflow research, we came across a post to do the conversion in Javascript. It works, but it must be called every time new data is added, which can only get worse over time.

```
var sanitize_timestamps = function(db, callback) {
    // Get the Trail nodes
    var collection = db.collection("nodes");
    // Convert from strings to isodates
    collection.find().forEach(function(doc) {
        doc.timestamp = new Date(doc.timestamp);
        collection.save(doc);
    });
}
```

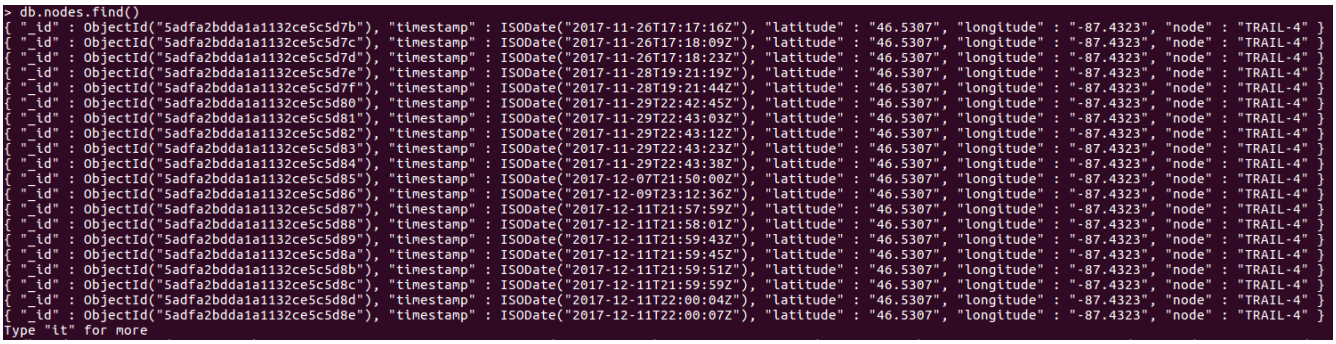
Struggling with MongoDB and Pymongo was worth it. The following MongoDB query is executed on the NodeJS server and quickly retrieves specific date related data for the client's graphs and maps.

```
var getAllTrailData = function(db, startDate, endDate, callback) {
    var collection = db.collection("nodes");
    collection.aggregate([
        {
            "$match": {
                "timestamp": {
                "$gte": new Date(startDate),
                "$lte": new Date(endDate)
                }
            }
        },
        {
            "$group": {
                "_id": {"node_name": "$node", "lat": "$latitude", "long": "$longitude"},
                "count": {"$sum": 1 }
            }
        }
    ]).toArray(function(err, docs) {
        assert.equal(null, err);
        console.log(docs);
        callback(docs);
    });
}
```

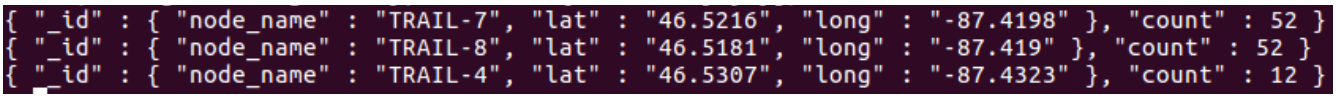
Understanding aggregate functions in MongoDB was an incredibly rewarding experience. It took hours of researching documentation, fixing bugs, and trial and error, but the final product here is worth it. The above query acts as a pipeline; whatever documents are returned from the "$match" statement (documents that are greater than startDate and less than endDate) are filtered out and passed down to the next segment of the pipeline. Then the "$group" statement assembles distinct documents defined by the "_id" accumulator, and sums up the individual count of each unique "_id". Dollar signs imply variables which are interpreted by Mongo as either functions like "$gte" (greater than or equal) or

database fields like “$node” (which would resolve to something like “TRAIL-4”). The query is followed by a toArray() function because Mongo queries return a cursor object which must be interpreted by Javascript. In this case we are interpreting it as an array of dictionaries, where each entry is a single trail node and the latitude, longitude, and number of timestamps that fit the end and start dates provided. To help visualize this, the first image is what’s in the database, and the second image is what the query returns on this data.

Before query:



After query:



As you can see, aggregate functions in MongoDB make analysis of this trail data much easier. From here we can take these values and put them into a heat map or a bar graph, or any number of analytical tools.

For the heat map, we used Google’s Javascript API. Initializing it is very simple, and updating the map data is even easier. To be clear, a heat map is “a representation of data in the form of a map or diagram in which data values are represented as colors.” Determining the intensity and radius of the colors is done by the number of LatLng objects at each location. Google’s API makes this easy by allowing a weight parameter so we can simply pass in the number of points at each trail node instead of looping through and creating each individual LatLng object.

```
19      for(var i in data){
20        var lat = parseFloat(data[i]["_id"]["lat"]);
21        var long = parseFloat(data[i]["_id"]["long"]);
22        finalArray.push({location: new google.maps.LatLng(lat, long), weight: data[i]["count"]});
23      }
24      return finalArray;
```

Every day progress on the project was managed by our github repository, which was an incredibly useful tool. Since I was working on the server back end, and Connor was on the client, we didn't conflict often. When we did conflict, however, github saved our work flow with an easy way to merge and manage our changes. This allowed us to simultaneously work on different features (and sometimes the same feature) without interrupting each other too much. It was a great lesson on teamwork, and how to code with others' solutions.

What Could Have Been Done Better

This is the type of project that could be iterated on many, many times. There are a multitude of features that we never even anticipated in our projected solution, let alone got around to implementing. An example would be building a MongoDB API for the user to run their own custom analytics on instead of relying strictly on the forms we present to them on the client. Our data analysis only goes so far as to filter by date, which is still a huge improvement, but we wanted to do more things like by hour or day of the week. Neither of these would be too difficult to add on top of the infrastructure already built, and would make a nice addition.

While the server and website work, there's an issue with closing the database connection that shuts MongoDB down and requires a restart. It doesn't do anything to the data, but is a bug that I couldn't fix in time and I figured it's notable enough to write about here.

Overall we are proud to have combined so many technologies together successfully. From the big ones like NodeJS, Python, and MongoDB, to all the libraries needed in each like the NodeJS-Mongo Driver, Pymongo, Pytest, Chart.js, Google Heatmaps API, and Socket.IO. Our server successfully manages file operations, database operations, and client-server communications. We have

built a central location for researchers like Dr. Thompson to launch more complex analysis, and we hope to hear more feedback that could turn into even better features in the future.