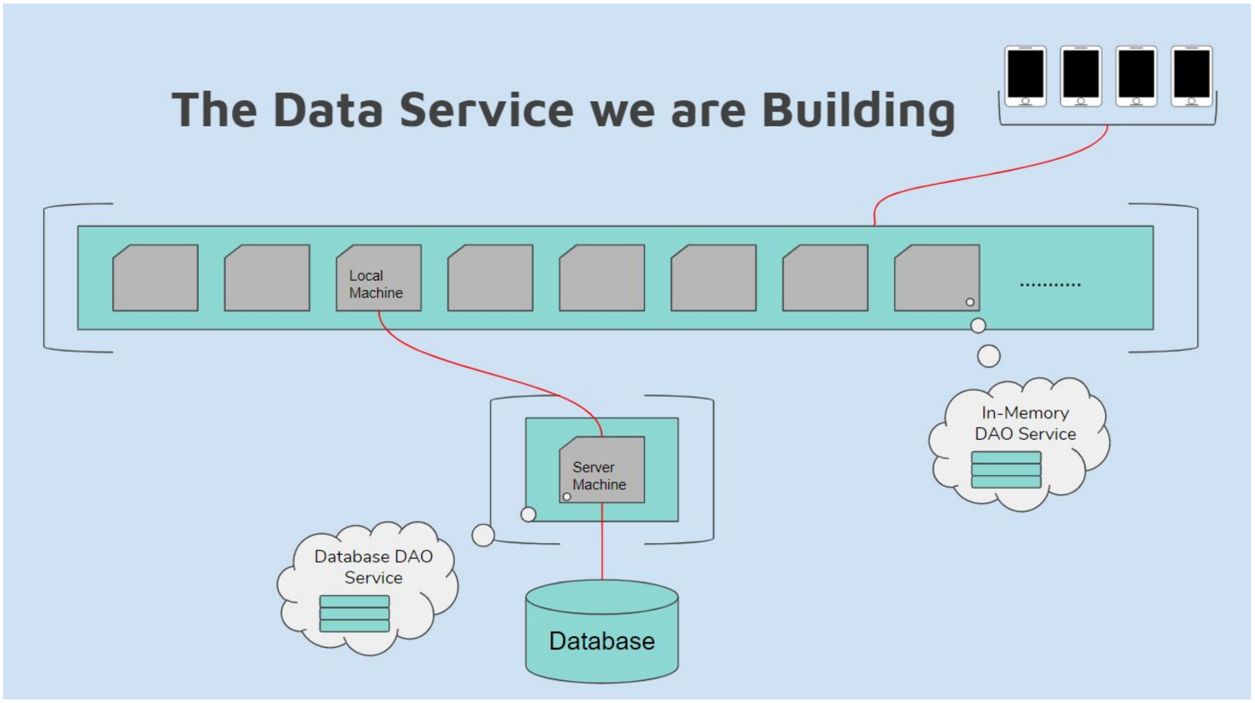
Two-Tiered Mapping Systems

A Look Into Database Design



Colorado School of Mines

CSCI-370 - Advanced Software Engineering

Summer 2019

Team Uber

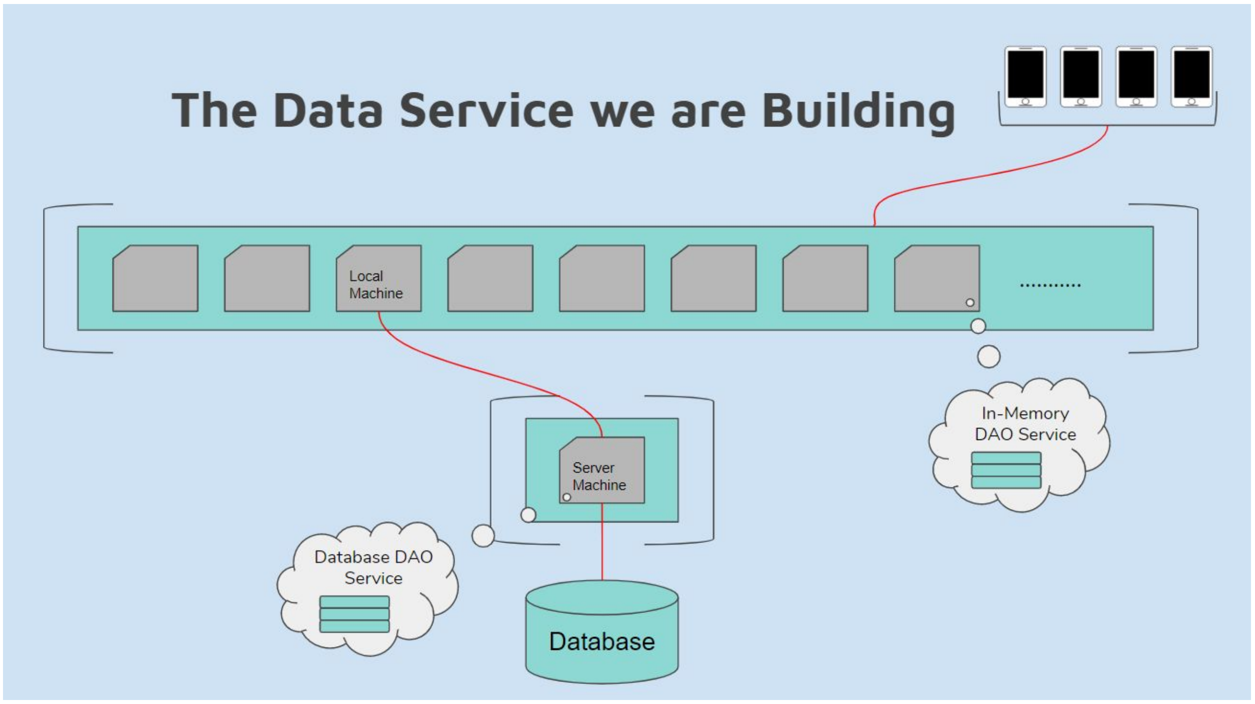
Jerry Bui
Nicholas DeCarlo
Casey Miller
Demonna Wade

Table of Contents

Problem Definition Given by Uber

Uber is an international ride-sharing company. They match drivers with people who need rides at a rate of 15 million rides per day. They also provide mapping data to all of their customers, and due to their size, they have to provide 1.3 gigabytes of mapping data per second. Because of this, the database that provides this data must be designed to handle the heavy flow. Uber has tasked us to deliver a stateless, REpresentational State Transfer (RESTful) Service that is a two-tiered database. A RESTful API Service is a lightweight implementation of HTTP; RESTful services are stateless, which means they do not store information about the client and they treat requests individually. We used HTTP GET as the primary means of communication between the user and Memory DAOs, and between the Memory DAOs and the SQL DAO. This database should be designed in such a way that a larger version of it could handle the massive amounts of data that Uber needs to deliver to its customers. The first tier (a persistence layer) stores data on disk, handles all writes, and act as a source of truth in the system. The second tier (a service layer), should be a horizontally scalable. This means that if we add more local machine's to our service, the faster our system would be able to output data. The local machine is commonly referred to in our report as the in memory DAO.

Depiction Communication Between the Local Machines and Data Server



List of the Functional Requirements

1. RESTful API & GUI & Ingest

a. Application / GUI is fully functional with showcasing names of landmarks for designated ‘rectangle’ area
b. The application / GUI must remain ‘stateless’ (information should not be shared through the server)
c. Application allows user to add more landmarks to the map; afterwards, the landmarks should update and then be visible to all users. (These points are added to the persistence layer)

2. Getting Business layer to work

a. The business layer must interpret data input via the API and use that to generate a mySQL request.
b. The business layer must interpret the results of a mySQL request and sends data forward to the API layer in a data type the API understands.
c. The business layer must categorize the points by ‘bucketing’ them into S2 cells, a grid system used to sort locations on a globe.

3. DAO (Data Access Object) to work with MySQL and In-Memory Data

a. Our DAO consist of two databases, in which the in-memory database requests data from the MySQL database.
b. The data used in our databases must be a manageable sample size of data so that we can incorporate a specific area of target.
c. Request data from In-Memory storage “system” and sends data forward to the Business layer to be processed.

4. Replication and Consistency

a. All machines in the ‘in memory’ layer must output synchronized data even though they may have been updated at different times.
b. All machines must be synchronized to the data of the machine that has gone the longest without being updated.

List of the Non-Functional Requirements

1. Final service should be packaged using Maven
2. All code should be written in Java
3. Performance should scale with number of working machines
4. MySQL is the recommended database - but PostgreSQL to be used: CPW's Server
5. Points of data should be categorized using S2 Cells (S2 Cells are generally used in mapping systems to sort the surface of a sphere into many rectangles)
6. ZooKeeper should be used for replication and communication between machines
7. Dropwizard should be used to build a RESTful API

Potential Project Risks

1. Technical

   a. Developing the database
   b. Maintaining multiple machines for testing
      i. Testing in-sync memory
      ii. Multiple inputs to multiple machines at the same time
   c. Consistent data type communication between the layers
   d. Setting up ZooKeeper correctly to handle the multiple machine communication
   e. Efficiently giving each machine the same starting data set
   f. Measuring the timestamps between sending and receiving data

2. Skill

   a. Limited MySQL knowledge
   b. Limited Zookeeper knowledge
   c. Limited Maven packaging and dependency knowledge
   d. Limited understanding on "stress testing"

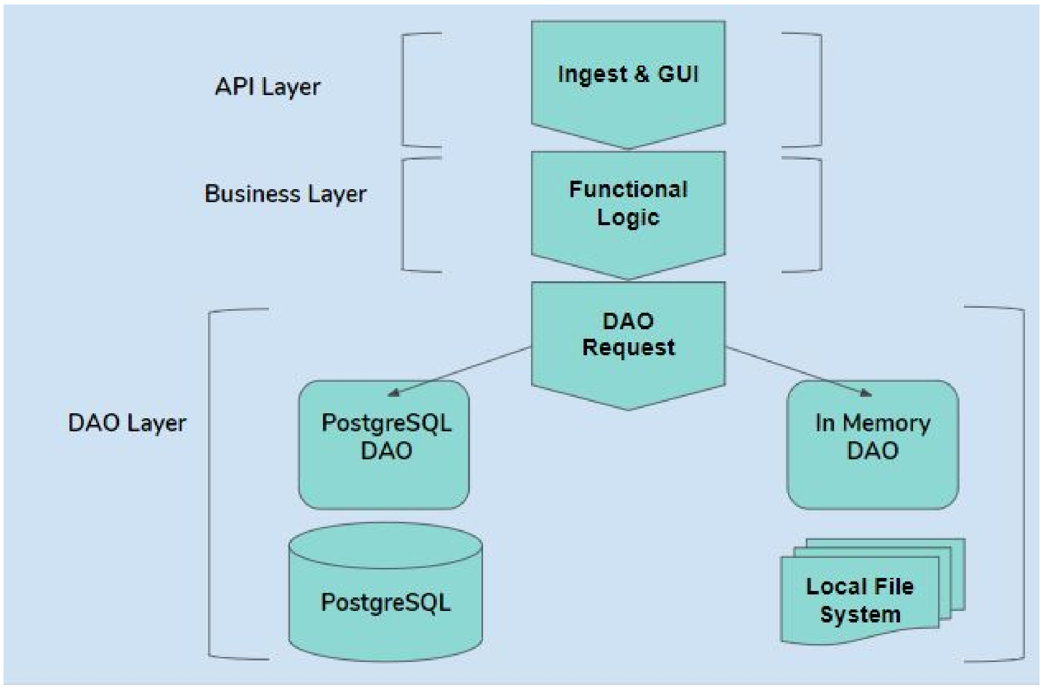
Definition of Done

1. Demonstrable performance increase
   a. Show that the multi-tier service is significantly faster than a service that requests data directly from a database.
   b. Show that our service gets faster as the number of in-memory machines is increased.
2. Package deliverable of the product
   a. Our project should be packed into one jar file for ease of use

System Architecture

Service Summary

Each tier has an available RESTful API Service that helps the users communicate with different systems such as: Java, ZooKeeper, and PostgreSQL. The service is made up of three layers: API layer, Business layer and the Data Access Object (DAO) layer. A DAO is just a Java object that delivers data from a database. If the service exists on an in-memory machine, which are all the machines that don't communicate directly with the database, the DAO layer utilizes the In-Memory DAO. Otherwise, the DAO Layer requests the PostgreSQL database. The business layer consists of a functional logic to decipher which tier the service exists in; the business layer then handles the DAO request properly. The business layer also connects to ZooKeeper, which enables each of the local machines to compare their current data state and retrieve updates if required. The API layer consists of a means to ingest data and a GUI to intake user commands. Dropwizard is being used to help create an easy and quick to use RESTful API.

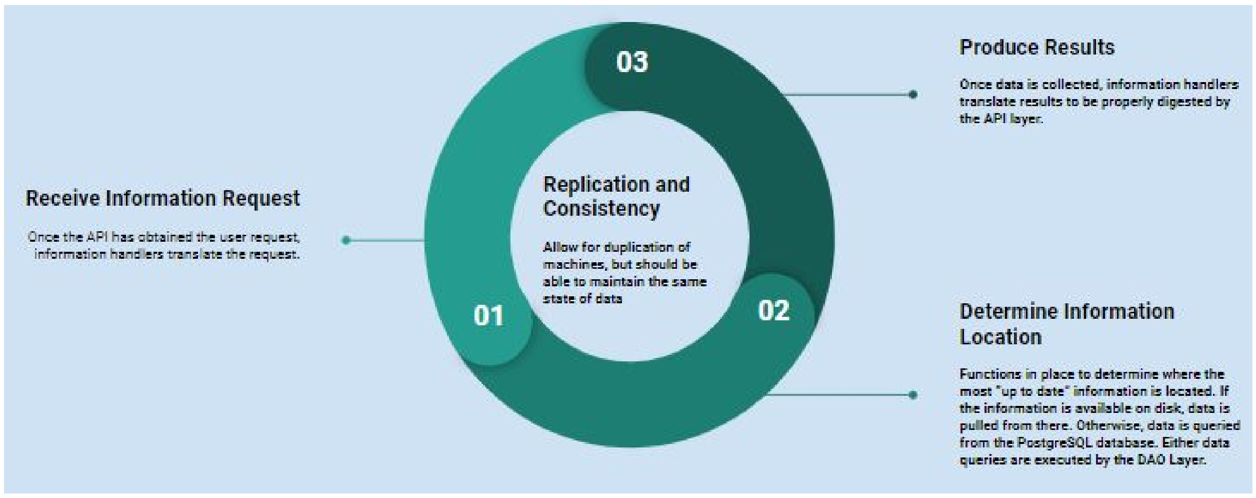


Design Challenges

For the business layer to determine which type of machine it is running on, we had it read from configuration files. Since all Memory DAO machines cannot update instantaneously, they often have varying amounts of data until all are synchronized. They still must produce the same results when queried at any given point in time. This required ZooKeeper to restrict newly updated machines from outputting their newest data points.

Business Layer Logic Cycle

The business layer is designed to execute upon an event request. As the user requests information using the API layer, the business layer determines the course of action. Depending on which machine the data is requested from, the business layer decides whether or not to make a “In-Memory” DAO Request or “PostgreSQL” DAO request. Once the DAO layer retrieves that data, the business layer properly handles the results and sends it back to the API layer to be rendered to the user. While this cycle continues, ZooKeeper constantly receives updates to maintain data consistency. So far, the team was able to design a simple client watcher that looks to ZooKeeper for data updates.



Receive Information Request
Once the API has obtained the user request, information handlers translate the request.
01
02
03
Replication and Consistency
Allow for duplication of machines, but should be able to maintain the same state of data
Produce Results
Once data is collected, information handlers translate results to be properly digested by the API layer.
Determine Information Location
Functions in place to determine where the most "up to date" information is located. If the information is available on disk, data is pulled from there. Otherwise, data is queried from the PostgreSQL database. Either data queries are executed by the DAO Layer.

Design Challenges

The communication with ZooKeeper required an intricate design. We must understand when, and how often, the service should be “pinging” ZooKeeper. Maintaining a consistent data “log” amongst several local machines is also another design trouble point. The local machines need to output identical data even when one machine has more updated versions; in this instance, we would only be able to output the most recently collective version.

One important final deliverable was to prove that when more ‘in memory data’ devices are added to the system, the system’s speed improves drastically. To demonstrate this, we performed stress tests on our system when it has different numbers of in memory data devices and compared how each of them performed.

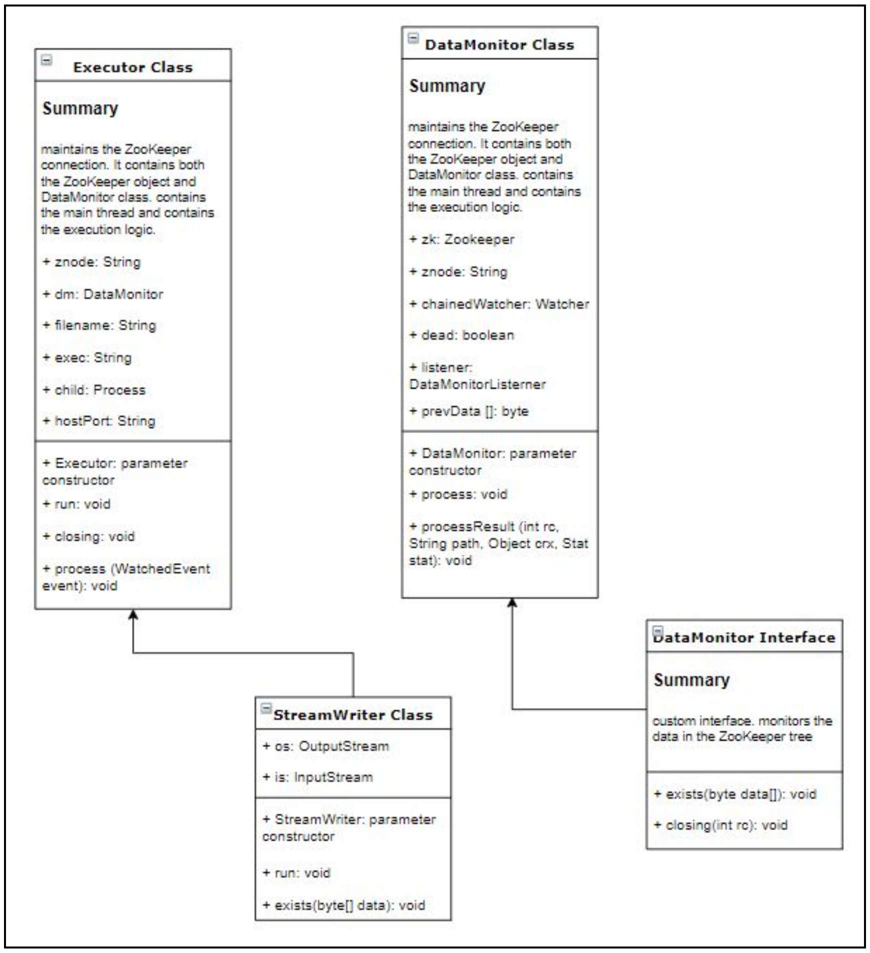
In addition to testing the speed with different amounts of ‘in memory’ devices, we also demonstrated a difference in speed when the business layer gets data from the PostgreSQL vs from the ‘in memory data’ devices.

Technical Design

Simple Client Watcher Program Structure

This is a UML diagram that describes the class structure for communicating with ZooKeeper. Each packaged service has these classes available to accomplish “in-sync” data retention.

UML Diagram of ZooKeeper


Executor Class
Summary
maintains the ZooKeeper connection. It contains both the ZooKeeper object and DataMonitor class. contains the main thread and contains the execution logic.
+ znode: String
+ dm: DataMonitor
+ filename: String
+ exec: String
+ child: Process
+ hostPort: String
+ Executor: parameter constructor
+ run: void
+ closing: void
+ process (WatchedEvent event): void
DataMonitor Class
Summary
maintains the ZooKeeper connection. It contains both the ZooKeeper object and DataMonitor class. contains the main thread and contains the execution logic.
+ zk: Zookeeper
+ znode: String
+ chainedWatcher: Watcher
+ dead: boolean
+ listener: DataMonitorListerner
+ prevData []: byte
+ DataMonitor: parameter constructor
+ process: void
+ processResult (int rc, String path, Object crx, Stat stat): void
StreamWriter Class
+ os: OutputStream
+ is: InputStream
+ StreamWriter: parameter constructor
+ run: void
+ exists(byte[] data): void
DataMonitor Interface
Summary
custom interface. monitors the data in the ZooKeeper tree
+ exists(byte data[]): void
+ closing(int rc): void

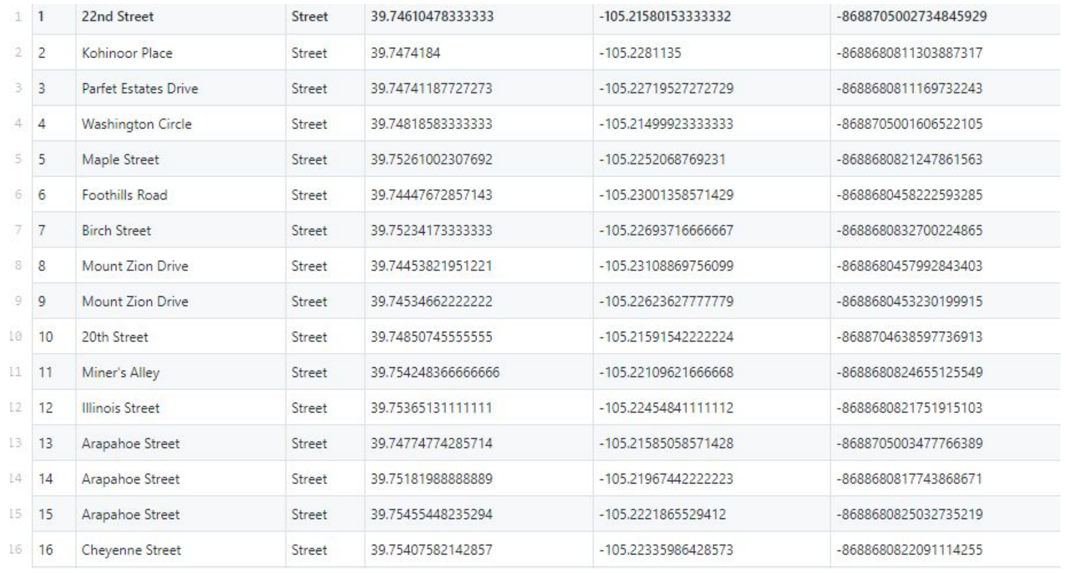
Design Challenges

Determining the exact data to be managed by ZooKeeper was a challenge. We followed the clients recommendation and used a “watermark” for each machine instance. The “watermark” would represent the current data status. Allowing ZooKeeper to maintain this information permitted each instance to know that amount of data the other instances have to ensure data consistency..

Database Format

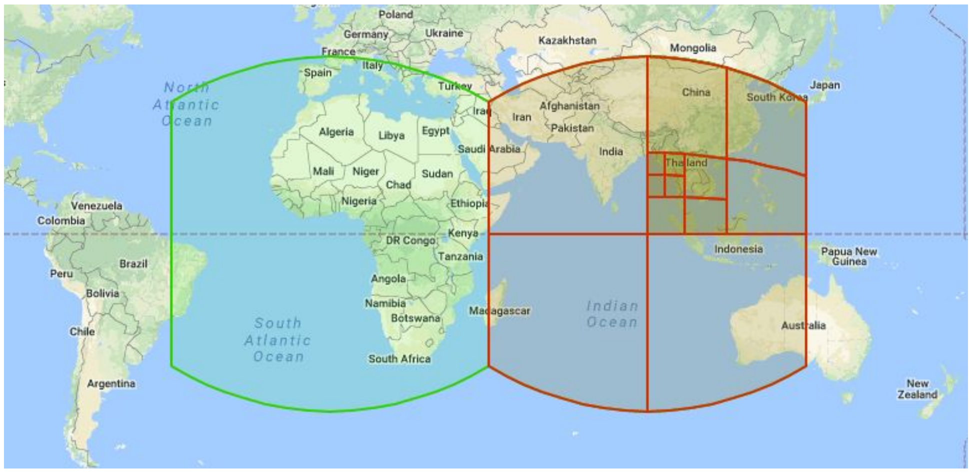
The information and data being used for this portion of the project is given from OSM (OpenStreetMaps). OSM is a free open source technology that provides a large amount of data pertaining to the world's road networks. We developed a simple program that can parse through an .osm file and output a .csv file with only the information we needed. The six columns correspond to the following features of each landmark: ID, landmark name, type of landmark, the latitude, longitude, and S2 Cell. These features comprise the data our database stores about each landmark.

.csv File Example



1	22nd Street	Street	39.74610478333333	-105.21580153333332	-8688705002734845929
2	Kohinoor Place	Street	39.7474184	-105.2281135	-8688680811303887317
3	Parfet Estates Drive	Street	39.74741187727273	-105.22719527272729	-8688680811169732243
4	Washington Circle	Street	39.74818583333333	-105.21499923333333	-8688705001606522105
5	Maple Street	Street	39.75261002307692	-105.2252068769231	-8688680821247861563
6	Foothills Road	Street	39.74447672857143	-105.23001358571429	-8688680458222593285
7	Birch Street	Street	39.75234173333333	-105.22693716666667	-8688680832700224865
8	Mount Zion Drive	Street	39.74453821951221	-105.23108869756099	-8688680457992843403
9	Mount Zion Drive	Street	39.74534662222222	-105.22623627777779	-8688680453230199915
10	20th Street	Street	39.74850745555555	-105.21591542222224	-8688704638597736913
11	Miner's Alley	Street	39.754248366666666	-105.22109621666668	-8688680824655125549
12	Illinois Street	Street	39.75365131111111	-105.22454841111112	-8688680821751915103
13	Arapahoe Street	Street	39.74774774285714	-105.21585058571428	-8688705003477766389
14	Arapahoe Street	Street	39.75181988888889	-105.21967442222223	-8688680817743868671
15	Arapahoe Street	Street	39.75455448235294	-105.2221865529412	-8688680825032735219
16	Cheyenne Street	Street	39.75407582142857	-105.22335986428573	-8688680822091114255

Representation of Recursive Definition of S2 Cells



Design Challenges

To reduce the time it takes for our database to process a query, we indexed our database by S2 Cell. This makes it so that only the landmarks around the requested region are evaluated to see if they are within the region. Since S2 cells are defined recursively, there are multiple different levels of S2 cells that we could use. We chose a level by moving up/down to different levels until our database requests were gaining efficiency from being indexed by S2 Cell.

Project Risks

The tasked projected does not have any direct risk because the client stated the prototype will not be used directly. The code written we wrote will not be implemented in a "true" functional system. The following is an excerpt from Uber's legal statement given to us by our client: "This project will be used as a training, classroom exercise for the team to learn from. There can be no direct benefit for the client because of the work being done cannot be replacing any full-time work that can be done at the client's company". Since our project will never be used, it has no real risks.

Elements of Software Quality Plan

Element	Code Quality Implementation
Unit Testing	White box testing is being used to test the internal structure/design/implementation of the following code units. Test cases are derived using inputs and determining appropriate outputs. Each code unit is being tested for proper functionality and intended use. Tests do not cover value but primarily focuses on tests that impact the behavior of the service. 1. API 2. Business Layer 3. ZooKeeper 4. Data Parsing 5. SQL Database
Interface Testing	Used to verify that all code units work in union when integrated. Test cases are derived at the level of scope of the service, and are only passed if all units of code are individually correct. Interface testing is to be done only once all code has been unit tested. The focus is towards verifying that all parts of the service fit together. 1. API and Business Layer a. Testing to make sure the data the API outputs can be read by the business layer, and vice-versa. 2. Business Layer with ZooKeeper a. Testing that In-Memory Java data types are being used to communicate configurations and data points to ZooKeeper. 3. Business Layer with Database

	a. In-Memory Java data types are being used to properly query the database for information to be returned to the API. 4. Database with Parsed Data a. Testing that our method of parsing data outputs the data in a format that can be read into the database.
Security Testing	Data available in the database should not be directly accessible by an end-user without proper credentials through the web UI. This is to ensure the service is free from any vulnerabilities, threats, and risks that may cause a big loss. Through the web UI, the end-user should only be able to add data points and unable to change or alter data points. Using security auditing, team members review lines of code for security flaws in querying data and manage this by testing the PostgreSQL Server in the command line.
Code Reviews	Before a large functionality can be added to the code baseline, a code review must be conducted with the team. The person committing changes is required to share the knowledge about the improvement and detail where the addition is to be added. The reviewers must be able to understand the programming technique and contribute to code improvement and consolidation of the change. Overall, there should be consistency in the code base and comments to make sure the code is easy to read and understand. The goal should be to help reduce bugs and unused code.

Ethical Consideration of Improper Quality Assurance

Since our client does not actually plan on using our service, there is no real ethical consequence to poor quality assurance. But if the client were to depend on the development of the team's prototype, the poor quality assurance would result in:

1. Inefficient transportation and travel
2. Unreliable information
3. Misuse of end-user security information

Initial Testing and Hypothesis

When our group was first given our project, we studied and learned about multi-tiered systems in order to have a better understanding of what it is capable of. Our group came up with the hypothesis that with more systems implemented, the faster the queries would be returned. However, after some initial testing with our code, we realized that the single tiered system was doing better than our multi-tiered. We found that replicated ZooKeeper connections were restricting performance. With that in mind we revisited our code, and reiterated our design and tested thoroughly for any hiccups that could be caused that would showcase the performance issue of multi-tiered vs. single tiered.

Reiterated Performance Testing Results

After carefully running both systems at the same time, our group was able to measure the capabilities of both systems when running 10 request within 10 seconds. The outcome of the test depends on the computer hardware. Each test returns a map of data points within a randomly selected region of our data. The “Single” DAO test uses one database access point to retrieve data. The “Multi-Tiered” DAO used test uses some number (1-4) of in-memory data access points to retrieve data.

Machine Metrics:

Kernel Name	Linux
Amount of CPUs	8
CPU MHz	3309.187
Amount of Sockets	1
Amount of Cores per Socket	4
Amount of Threads per Core	2

Results:

DAO Used	Test 1	Test 2	Test 3	Test 4	Average
Single	1821	2041	1742	1635	1809.75
Multi-Tiered (1 machine)	3549	3601	3647	3622	3604.75
Multi-Tiered (2 machine)	3850	3794	3726	3880	3812.5
Multi-Tiered (3 machine)	3831	3438	3828	3744	3710.25
Multi-Tiered (4 machine)	2328	2196	2200	2272	2324

Summary of Testing

The final testing of our program clearly shows that, multi-tiered machines work better than single tiered machines. We also partially show that the project is horizontally scalable: with more added ‘machines’ to the system, the more productivity overall the system will have until a breaking point based on the computer specifications is reached. In the above tests, there is an improvement from one machine to two machines, but after that, the performance decreases. This is because the computer the testing was done on was unable to run more than two machines without the test results being impacted. The computer was simply bogged down because it had to run so many machines that it could not quickly retrieve data from all of them.

Results of Usability Tests

The client made it very clear to us that our software would never be used by anyone except ourselves. Our only deliverable is a report of our testing results comparing our two data systems. Because of this, the software is only used by the creators, making usability tests unnecessary; it is sufficient as long as we can understand it.

Conclusion and Future Work

Within our scope and given timeline for our project, our group sought out to deliver a data-processing service and the results of product testing of a two tiered system that stores map information. The results for the product testing can lead towards the transition of multi-tiered systems where data is trinkled from an original source into local machines that can then output their systems for general use.

Our user interface does what is expected: generate areas of interest and allow users to create points on a given map all communicating with a local or server database.

Since a lot of this management of data required us to sign in into our PostgreSQL server provided by the school, we were able to create a code that would securely and keep usernames and passwords. However, within product testing and implementation of ZooKeeper, we found ourselves manually typing in our passwords multiple times; our group ended up hardcoding a superuser and pass to access the database. We did not have time to implement an automatic testing environment for our systems. We still ran unit tests by hand, but since there were many independent systems and they don’t update often, automatic tests were not worth the effort.

Lessons Learned

- Maven is an incredibly powerful, albeit sometimes frustrating tool, for compiling projects. Getting Maven initially set up was challenging, but it made importing external libraries much simpler. It also got rid of a lot of potential headaches when we needed our jar files.
- When stress testing our data service we learned that one must be very cautious when writing bash scripts. Instead of pinging our database for data and disregarding it 4000 times, we accidentally pinged our database and saved the data into 4000 different files. This caused immense lag and memory issues. Writing a bash script wrong can cost the programmer a lot.
- PostgreSQL is a limited manageable database server that could have been easily replaced with any other type of SQL service such as MySQL, however we used PostgreSQL as it was a free option that was given by our academic advisors. PostgreSQL was said to be able to copy .csv files and put them into a data table for use. However, once we parsed data from our java file, we had a hard time using the \COPY

command in our Java program (Java is able to implement and connect to the PostgreSQL server and run commands) since Java had an issue reading the ‘\’ within a string order. Our group ended up opening the datatable in the command line and running the \COPY command there. With other resources (MySQL, Microsoft SQL, or Amazon Web Services SQL) we would’ve been able to connect through Java and run a simpler command in order to copy .csv data into a datatable and have it run where users could implement their own .csv files if they were correctly formatted. Later iterations of our program are able to parse OpenStreetMap files and insert the data directly with one INSERT command, using no CSV intermediate.