Geospatial Database

Advanced Software Engineering
June 21, 2017

Client: Ricky Walker

Khanh Duong, Ryan Hunt, Tim Walker, Huan Wang

Colorado School of Mines
Golden, CO

I. Introduction

Client Description

Uber is a San Francisco based technology company most famous for their mobile app transportation service. Uber's app is a taxi-like system in which a user requests a ride to a location and a Uber-partner driver will drive them to their desired location. Unlike taxi services, Uber drivers use their own cars and the prices are determined before the service is provided, not afterword.

Since beginning the service in 2010, Uber has expanded their market to over 616 cities worldwide. This massive expansion around the world means Uber needs systems that scale incredibly well so they can continue to increase their production without recreating their systems.

Uber's primary market is tied to transportation, so Uber requires reliable databases—services that are always available when someone needs a ride. The market of transportation is heavily reliant on the geospatial information, so Uber's scalable and reliable database must store information about the world.

Product Vision

The primary goal for the project is to build a service that allows for the storing and retrieval of geospatial data. There is a massive amount of data that needs to be recorded—an entire world of information—so the service needs to be built upon a system that can scale to query that much data in a reasonable time.

RDBMSs do not scale well enough, and databases like MongoDB with built in geospatial indices are not reliable enough for a company like Uber which requires access to the database at all times.

To solve these problems, this product is built on top of the NoSQL database Cassandra which has great stability and incredible scaling potential. The product must retain the scalability that Cassandra provides while still providing useful geospatial information.

II. Requirements

The product must be a reliable, scalable database that allows developers to efficiently store and query geospatial data. This product consists of three parts: the Cassandra database, a client side interface, and the Service Layer which interacts with both the client and the Cassandra cluster. The Thrift compiler is used to facilitate communication between the service layer and the client application. This protocol allows data transportation to happen seamlessly between different programming languages.

The client's specification for the project made clear that the goal of the project is scalability of the final system. While the project must meet all of the functional requirements listed below, the project should focus on the non-functional requirements (specifically scalability).

Functional Requirements

The product must have a geospatial database whose data includes the following information:

- Unique identifier (UUID)
- Geometry/location
- Additional geographic information

The product must also have a web app which will make interfacing with the database easier for potential clients. These interfaces must include, but are not limited to:

- Querying the database to receive geospatial data about a region
- Modifying the data
  - Adding features
  - Deleting features

    - Modifying a feature
- Running an historical query on a region which shows a prior database state

To facilitate the interaction between the client-side and the database, a Java service layer will be designed that will query the database. The service layer and the web app, written in different languages, will communicate using Thrift.

Non-Functional Requirements

Most importantly, the geospatial database must be horizontally scalable. In this context, scalability means that changing the size of the database will not require changing in the codebase or upgrading the machine with larger memory and more expensive CPU. The only required change will be adding more machines to the cluster to fulfil the demand in increasing the size of the capacity and the throughput of the database cluster.

III. System Architecture

The system consists of three major components: the Cassandra database cluster, stateless service layer, and client interface. The Cassandra database stores the geospatial data. The service layer handles all interaction with the client, and provisions any queries or updates with the database. The client interface will be a web app created with end users in mind to make querying and modifying features simple.

The client app and the service layer will communicate to each other using the Apache Thrift framework. Thrift is a software generation tool that allows for a single client-server contract that is written in the Thrift language to be distributed to any number of clients running different languages on unique architectures.

A diagram of these three main pieces and their interactions can be seen below in **Figure 3.1**. Each component is described in greater detail as well.

Database

Apache Cassandra is an open-source, distributed NoSQL database designed by the Apache Software Foundation. The database is intentionally "masterless", affording no single points of failure to the system. This feature is achieved by transparent data replication and automatic node failover. The intended system architecture for a production Cassandra cluster is a discrete collection of data centers each running "commodity" servers. The advantage of running these commodity servers is clear: rapid horizontal scalability.

The Cassandra project has five clear objectives, each of which works to solve some of the more common issues experienced with other NoSQL databases:

- **Decentralization** - Every node has exactly the same role in the cluster, and any node can service any client request.
- **Replication Support** - Replication can be fine tuned for failover and disaster recovery under any topology, including multi-datacenter configurations.
- **Scalability** - Read and write throughput increases linearly with the addition of nodes to the cluster.
- **Fault Tolerant** - Since replication occurs automatically across nodes, failed nodes in the cluster may be replaced with zero downtime.
- **Adjustable Consistency** - The consistency of a cluster may be adjusted to match the application, ranging anywhere from single node consistency all the way to full cluster consistency.

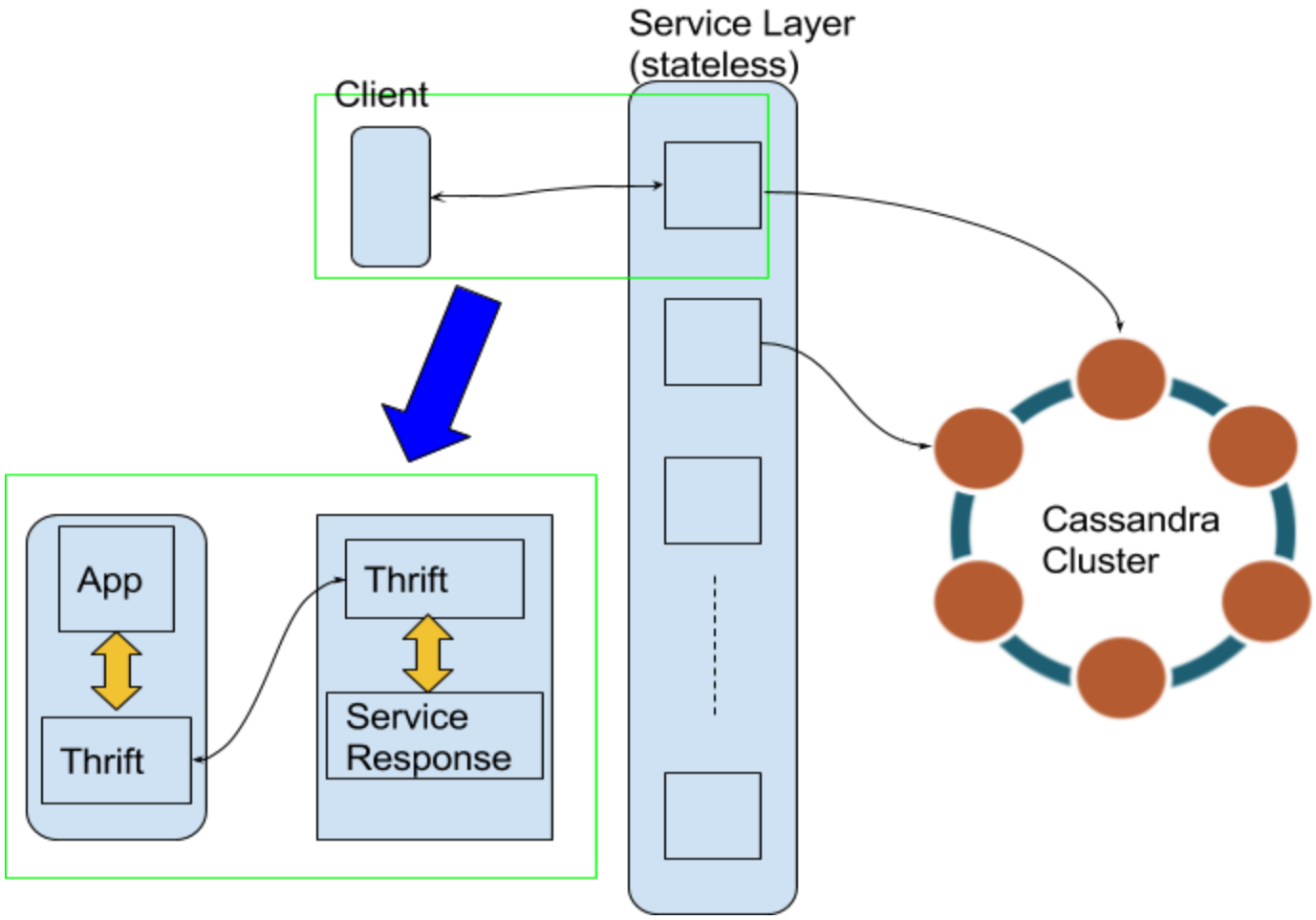

Figure 3.1. System Architecture

Due to the fact that provisioning a full datacenter is out of the scope of the this project, instead a production Cassandra cluster was simulated by internetworking personal computers together. Clusters of varying sizes were built and tested to analyze the feasibility and scalability of our design. This topic is discussed in much further detail in the Scalability section of the report.

Service Layer

The service layer is the interface that sits between the client and the Cassandra database. This layer services requests for geospatial data within some client-specified bounding box, and returns the appropriate data.

Java was the language requested by the client for the implementation of this layer. The service layer connects to the database using the official Datastax Cassandra Java driver, which ensures minimal latency and optimal throughput. The service layer interfaces with the client using a Thrift-defined client-server contract, which dramatically simplifies updates and modifications to the API.

The service layer contains practically all of the business logic for the system. The service layer is responsible for the process of loading the geospatial data into the database, which involves the

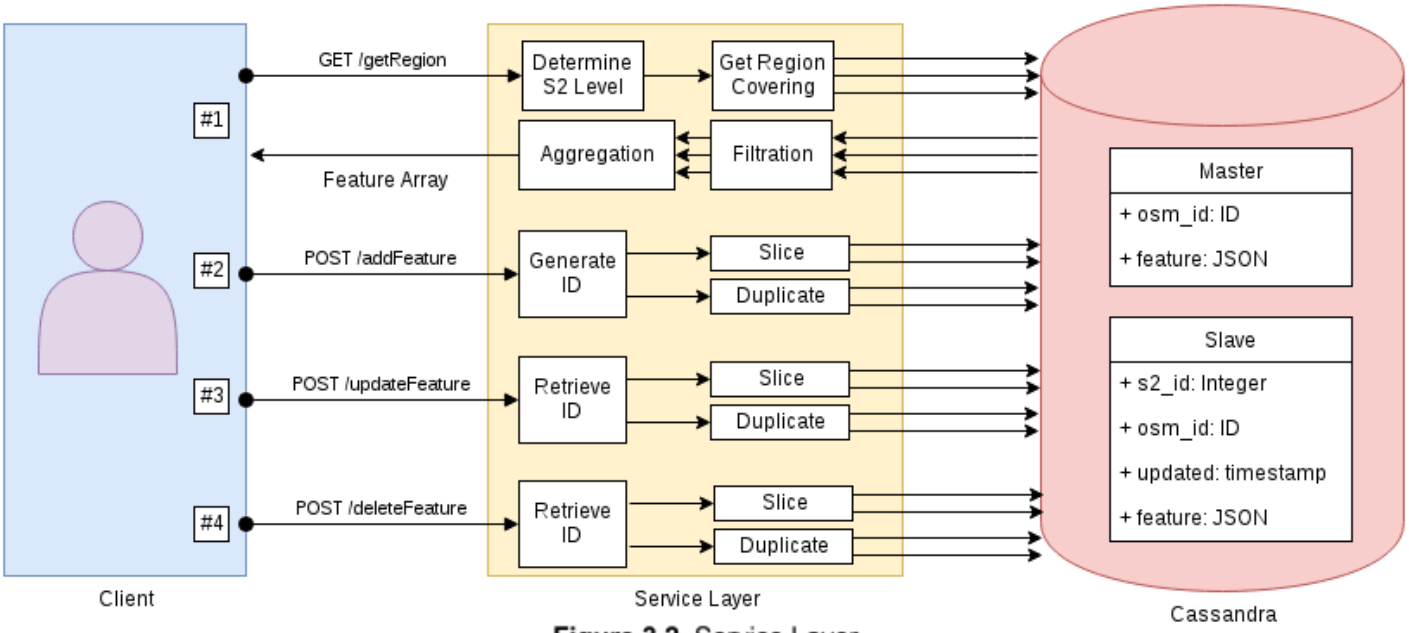

Figure 3.2. Service Layer

denormalization of the geospatial data into the database. It is also responsible for coordinating the reconstruction of data from multiple sources in the midst of a client request. The design of the service layer is in many ways the keystone of the system design. A block diagram of our service layer is shown in **Figure 3.2**.

The service layer is able to handle four different types of client requests. These types are (1) Bounding-Box Queries, (2) Feature-Add Requests, (3) Feature-Update Requests, and (4) Feature-Delete Requests.

For the bounding-box queries, the client supplies a geographical bounding box, using latitude and longitude coordinates, to match the visible portion of the map on the client's screen. The service layer takes this box, estimates a practical level of detail (number of features) for the response, and performs queries against the database at the determined detail level and in the containing geographic boxes. The service then collects the results of these queries, filters duplicates from this collection, and returns a list of features to the client.

For the feature-add requests, the client simply supplies a proposed feature to the service layer, and the service layer calculates an ID for the feature. The service layer then parses the feature and determines how to effectively denormalize the feature and add it to the database.

For the feature-update requests, the procedure is roughly the same as the feature-add requests, save that the service layer is no longer required to calculate a new ID for the feature. It may instead use the existing ID that is assigned to the feature. Again, the service layer will parse the feature and denormalize it into the database.

Finally, for the feature-delete requests, the client need only supply the ID of the feature. The service layer will then work to remove all instances of that feature from the database.

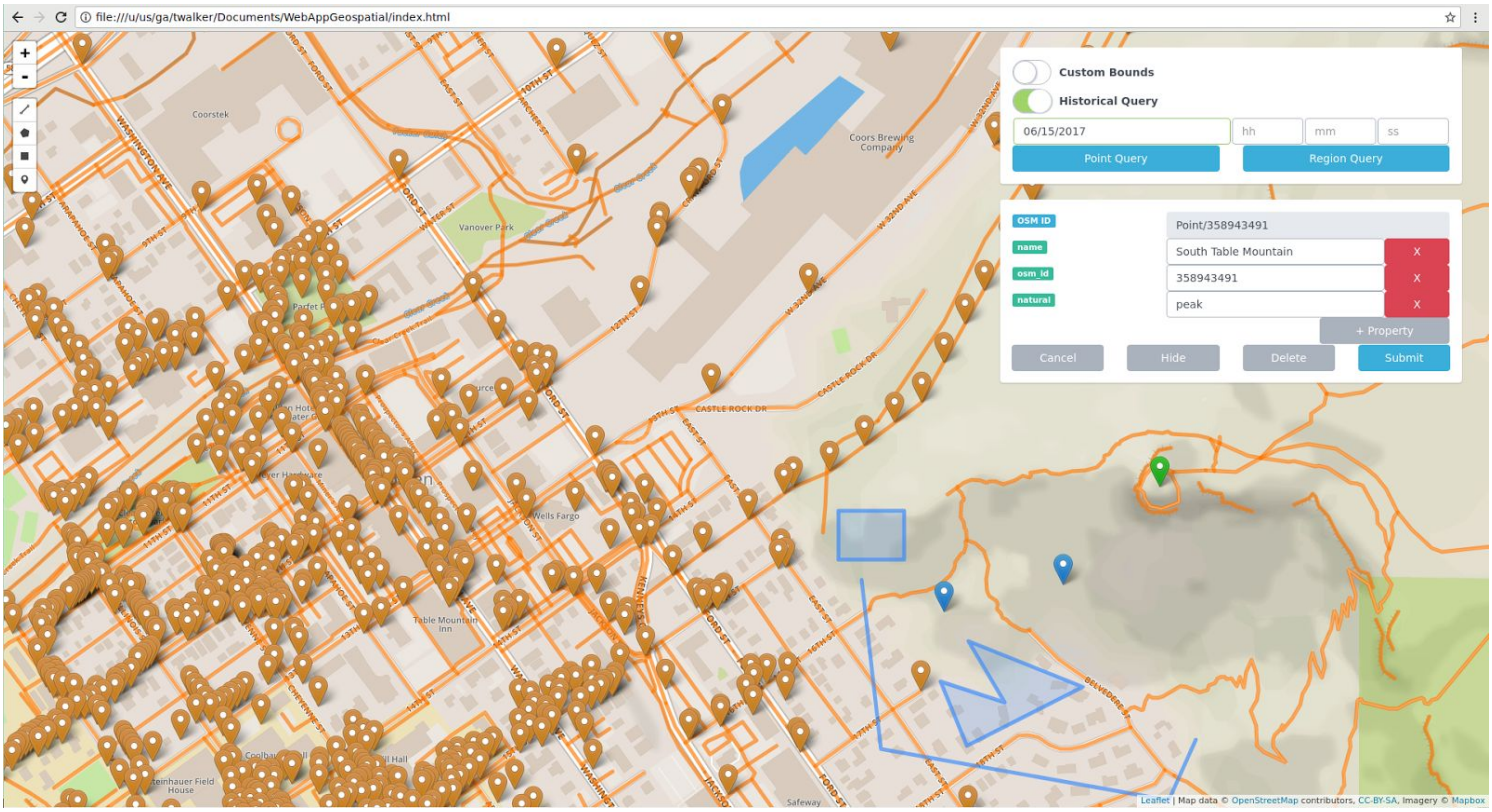

Figure 3.3. Client Web Application

Client Application

In order to make the system user friendly, a client application had to be developed. This application was developed as a web app that communicates with the service layer with the Thrift-defined client-server contract. A screenshot of the application is shown in **Figure 3.3**.

The web app consists primarily of a map built on the Leaflet library for interactive maps. The Leaflet interface allows for significant customizability and has strong library support, allowing for more features more quickly.

The application also has a simple interface that allows the user to interact with the database. The panel in the top right of **Figure 3.3** is used to initiate a query of the database. Both Region queries (which finds all features in the user's window) and Point queries (which find the features in a single s2 cell located at the center of the window) are supported. Both queries support historical queries, showing a prior state of the database.

Below the Query Panel is the Feature Modification Panel. Using this interface, a user can modify a feature (by changing, adding, or removing properties) or delete a feature. The selected feature is highlighted green.

A second library, Leaflet.Draw, allows the user to create new GeoJSON geometries. Combined with the Feature Modification Panel, users can create new Points, Polygons, and Lines and fully customize the properties of each new feature. These new features are added to the editing layer, colored blue, to inform the user that they are not part of the database until they are submitted.

These features allow the web app to accomplish the goal of making interfacing with the database easier for the client.

IV. Technical Design: S2

Google's s2 library maps the world into unique cells. This functionality is beneficial to this project because the service will query the database with longitude/latitude information, which s2 can easily map into a cell. S2 cells have varying sizes, called levels, ranging from level 30 (0.48 cm^2) to level 1 (85,011,012 km^2). A cell at level n is made up of four cells of level n+1.

The s2 library determines these cells by mapping the globe onto a cube and then creating a Hilbert curve on each face. The Hilbert curve is a fractal space-filling curve which is notable because it somewhat retains spatial locality—i.e. if you stretch out the hilbert curve into a straight line, two points close together on that line will map to points close together on the face of the cube. The first six iterations of the Hilbert curve are shown in **Figure 4.1**.

The Cassandra database works best when the primary key has clear partitions, and the s2 library allows for those clear partitions. The preservation of locality by the Hilbert curve means that often, nearby cells will be stored on the same cluster, leading to more optimal queries on the database. In the final version of the service, features are placed into their corresponding s2 cell based on their coordinates and their size.

Unfortunately, the cells of different levels overlap in physical space. This means that design of the database either needed to be limited to certain s2 levels, or a system needed to be designed that would determine which levels to query on and which levels to place features into. Section V discusses the final design for how features were stored in the database and on which levels they were determined to be stored.

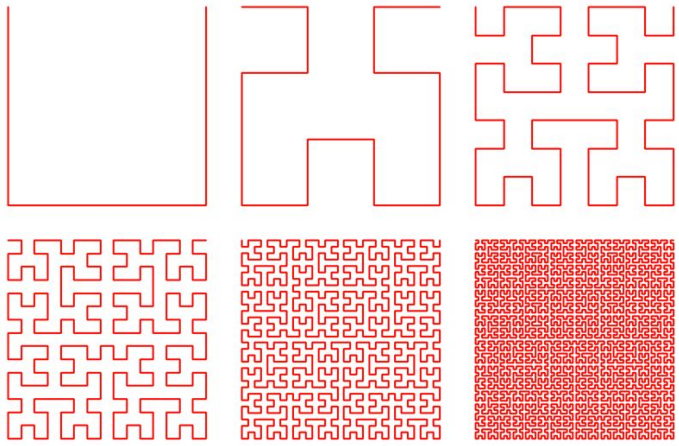

Figure 4.1. The Hilbert Curve (6 iterations)
Image credit to http://datagenetics.com/blog/march22013/index.html

V. Technical Design: Storing

The proposed database stores data in increments of S2 Cell. The planet is divided into cells that ranges from level 1 to level 30 with 30 being the smallest cell. The database only makes use of levels 13 to 4, which is an effective range of 1.9 km^2 to 498108.4 km^2. The features are first extracted from the GeoJSON file and its size is approximated to its S2 Cell level. Afterwards, that feature is mapped to the corresponding S2 Cell and level and stored into the database. For features that extend over multiple cells, there are three possible approaches to storing them: duplication into multiple cells, slicing a feature into the corresponding cells, and a combination of the two.

Duplication

The simplest approach to solving this issue is to store the feature on all the cells that it overlaps. Because the feature is stored on the level that is closest to its size, the worst case scenario is that a

feature overlaps the corners of four S2 Cells. This means any given feature within level 13 and 4 can be duplicated 4 times. Features under level 4 would be duplicated more while features beyond level 13 would not be duplicated at all.

A more serious issue occurs when trying to query from this approach on the database. When the region query lands on a part of a feature, that entire feature has to be returned. This issue is very prominent near large borders, where the entire border has to be returned regardless of how small the overlap is. One potential solution is to apply a caching layer between the service layer and the database cluster to reduce the stress to the database directly. But applying the caching layer requires additional hardware resources which cannot be implemented with the current scope of the project. As a result, cutting large features to only return a portion of them became the solution.

Cutting

The second approach involves cutting features that cross S2 Cell boundaries into smaller features. This reduces the redundancy of the duplication method at the cost of increased overhead in the handling of features.

When the features are added to the database they need to be divided along the boundaries for each S2 Cell at each S2 level. When the service layer queries the database, the features need to be reconstructed into the master feature so the client has the entire feature and not only a segment. This additional processing needs to happen not only when a feature is loaded in, but when the service layer performs any interaction with the database, including modifying features.

Despite the difficulty of the cutting method, it provides large benefits to the service. By reducing the amount of data that has to be returned on a particular query, large speed ups can be obtained especially on hotspot areas that would be queried by many people (like the U.S. border). With the cutting method, the service can run faster without additional elements outside the scope of the project, like a caching system.

Combined

The final database uses the duplicating method as the core of the database because it ensures correctness, while implementing the cutting algorithm to trim unwanted information. Trimming the data is important to send it back to the user at a reasonable speed and size. Implementing the cuts increases the amount of data that has to be duplicated. A feature stored on one level 4 cell is copied and cut in every level down to 13. The worst case scenario is a large feature being replicated 13 times.

The duplications might seem excessive, but they are very effective. Data is transmitted back at a reasonable speed and all features a user would be interested to know about within a region are included.

Contrary to this method, Uber chooses to use the pure duplication method in tandem with a massive caching layer to prevent querying the same large features multiple times. This is not possible with the equipment available during the field session.

VI. Technical Design: History

One of the system requirements is to include a way to query the database and receive its state at a particular time in the past. The difficulties in this implementation occurred because any prior state needed to be reached (i.e. regular backups would not be sufficient) and because of difficulties in Cassandra.

To be able to query for any prior state of the database, when features are modified or “deleted” from the database the old version of the feature must remain in the database. In order to keep track of

which version should be visible at a particular time, each feature was assigned a timestamp at entry into the database.

The timestamp for a feature is defined as the time that that feature was last correct. This means that new features are added to the database with the maximum timestamp, and when a feature is deleted from the database its timestamp is updated to the moment it was deleted. In order to avoid the new feature showing up earlier than it ought to in an historical query, a second null feature must be added with the same osm_id and the current timestamp, but with a null json value. Modified features simply require changing the current database feature's timestamp to the present and adding a new feature with the updated json.

Cassandra only allows SELECT queries on the primary key, where every preceding level of the primary key is also constrained. This meant the timestamp had to be part of the primary key, as early as possible so as to make the historical queries efficient. As can be seen in **Figure 6.1**, time is the second entry of the primary key after the partition key of (level, s2_id).

With this system, regional queries were able to be efficiently run at any point in time, simply by querying the database with the WHERE clause:

```
time<=maxtimestamp(timestamp)
```

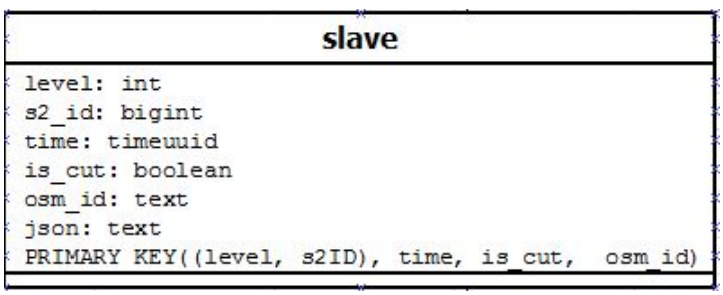

Figure 6.1. Schema

VII. Decisions

Geospatial Divisions: GeoHash vs. S2

The method we are using to store the geospatial data in this project is known as bucketing. The two primary methods to perform bucketing are GeoHash and Google S2.

After examining the high-level specification of both methods, it was found that GeoHash has more restricted levels than Google S2—for certain zoom-in levels, Google S2 has a better resolution. Additionally, Google S2 uses the Hilbert space-filling curve, which encodes 2D coordinates to 1D. This makes it easy to obtain the adjacent buckets. All that is required is to slightly offset the 1D distance because two points close to each other in 1D is mapped close to each other in 2D as well.

Due to all the advantages associated with Google S2, the final decision was to proceed with the Google S2 library. The library's more flexible mapping and more extensive API make it the logical choice for this project.

Formats: XML vs. GeoJSON

By default, OpenStreetMap exports an XML file. The file on its own is sufficient for loading basic nodes, but quickly becomes unusable when dealing with anything that has more than two points. An alternative is GeoJSON, which has greater readability and the support of the JSON format. GeoJSON also takes up less space than XML—requiring only half the space.

The biggest advantage of GeoJSON over XML is that it encodes the geo data in a logically independent way (i.e. a road is independent of another node on the map), while XML represents the data in a relational manner (a way consists of multiple node, etc.). GeoJSON naturally fits better with our NOSQL database design. GeoJSON also has significant

support in web map APIs, while the XML data would be more difficult to work with.

Proceeding with GeoJSON means that all data exported from OpenStreetMap must be converted before loading onto the database. As a bulk load onto the database should only happen very rarely, the conversion issue is of lesser importance. The end user of the database is affected very little by the initial conversions once the database is up and running. As the loading difficulties only affect the developers, and GeoJSON makes other aspects of development easier, GeoJSON became the format of choice to represent the data rather than XML.

Libraries: Leaflet vs. GoogleMaps

We decided to switch to using the Leaflet map API instead of the Google Maps API. Both have similar base features, but Leaflet has a more flexible implementation especially in the tilesets to use and further libraries built on the Leaflet framework. Leaflet also has a more helpful GeoJSON implementation.

Additional libraries for Leaflet make parts of the web app implementation much simpler. In particular, Leaflet.draw makes creating new features with complex geometry significantly easier compared to a Google Maps implementation. Because it has already implemented features that the project requires on the client-side, the logical choice is to convert our existing code to use the Leaflet and Leaflet.draw APIs.

VIII. Results

The final database, service layer, and front-end design each met and exceeded the basic requirements of the project. The system stores and queries Geospatial data from both the present and the past.

The final database design successfully makes use of both the cutting algorithm and the duplication algorithm. Combining the two approaches allow the system to selectively return data to the user, eliminating the need for a caching system and avoiding the challenge of reconstructing features from cut pieces.

As a demonstration of the system's scalability, a Cassandra cluster was built on the Alamode Lab machines at the Colorado School of Mines. This system will be discussed further in the following section.

Geospatial data of the entire world was loaded into the cluster to demonstrate its effectiveness. The cluster, with the help of the service layer and client, is able to:

- Store given data in a GeoJSON format
- Query for features within a region
- Provide information about a feature of interest
- Append or remove information about a feature using the client
- Add/Remove a feature using the client
- Perform a historic query on the data, revealing a previous state of the database

IX. Proving Scalability

The non-functional requirement of scalability is essential to the success of the project. The designed geospatial database was tested for this scalability to prove that it retains the benefits of NoSQL databases that Cassandra provides.

To facilitate the analysis of these systems, a second Web App was developed that would rapidly query a database at specified intervals, recording the amount of data that was received and the time it took to execute the query.

The tool analyzes the throughput of a specific query as a dependent variable based on the load

placed on the server. Using this data, the tool generates a plot of load (client queries per second) against throughput (kB per second).

This web app was run in two cases: on a database cluster with a single machine, one with four machines, and one with twelve machines. This creates a good estimate of the scalability of the system, because an twelve machine cluster should be faster than a four machine cluster, which will be significantly faster than a one machine cluster (all with the same code).

Running the experiment on both clusters produced the graph shown below in **Figure 9.1**. The throughput of the four and twelve node clusters greatly exceeded the throughput of the single node cluster, and the twelve node cluster had a sizable speedup over the four node cluster. The speed increase from the four node to the twelve node cluster is less substantial than the one-to-four node increase because while the system scales well, the returns are still limited.

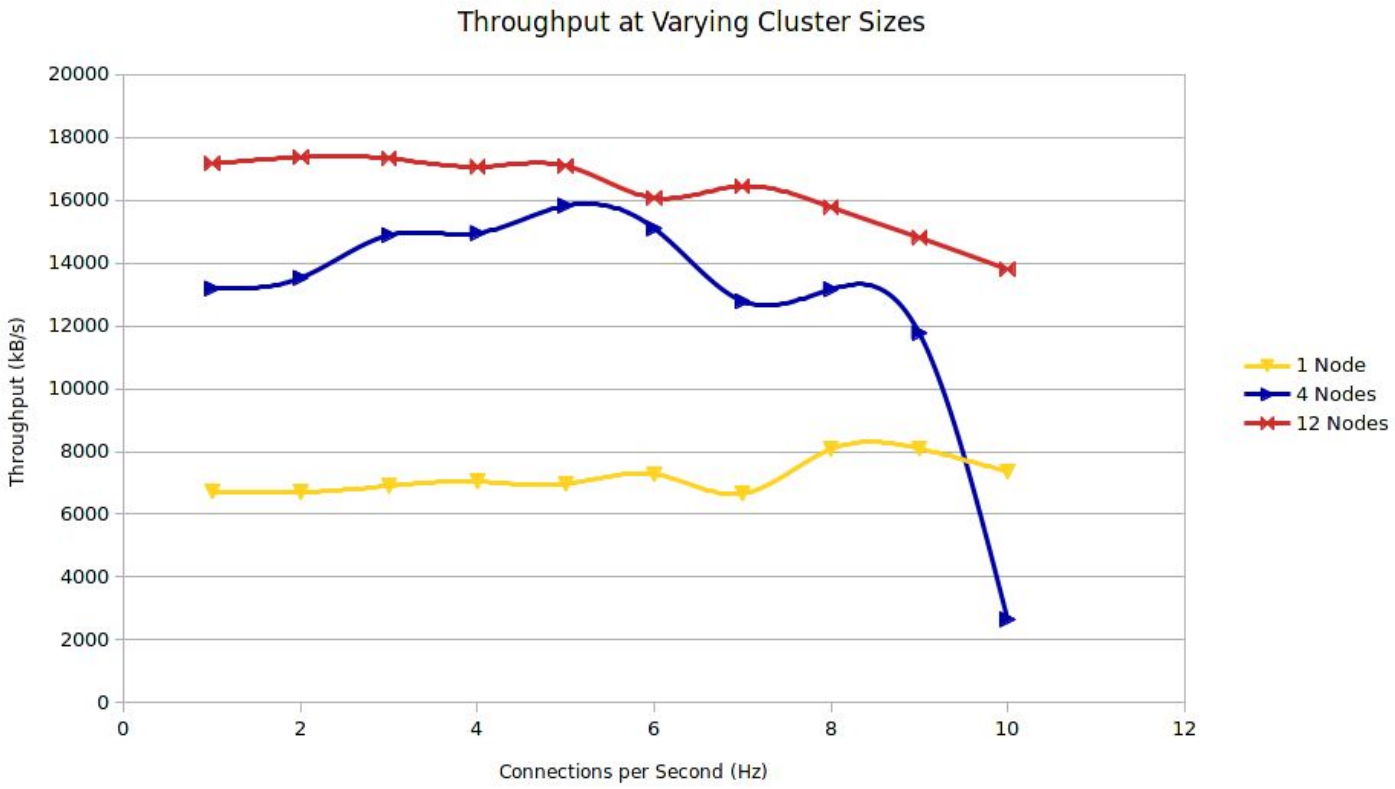

Figure 9.1. Scalability Plot