Bicycle Sharing Data Exploration

Jeannine Endreß, Corey Ford, Michael Hotan, Yeliz Kurt

School of Computer Science and Communication (CSC)
Royal Institute of Technology KTH, Stockholm, Sweden

Abstract. Bicycle sharing systems generate interesting data about inner city commuting behavior. Using a data set from the Citi Bike system in New York City, we created a framework to help users analyze and visualize usage patterns. We incorporated several different database models: a spatial database and a graph database.

1 Introduction

The idea of "open data" is not new but has gained popularity in recent years thanks to the Web. While data visualization has been effectively commercialized with business to business products like Tableau[6] and Qlikview[3], publicly exposing data is a newer, more controversial concept. The open data principle allows people to freely access, use, or republish particular data without licensing restrictions. Open data is often focused on non-textual material such as maps, genomes, chemical compounds and mathematical and scientific formulae.

Bike sharing systems, increasingly popular in recent years, aim to promote the use of active transport modes, multiply travel choices, reduce the demand on automobiles and help to decrease greenhouse gas emission. They provide an affordable, secure, and enviromentally sustainable alternative for inner city residents. The data stored about such systems includes trip times, station statistics, and customer information. Recently, many bike sharing systems have opened up their data to the public for analysis.

2 Data set

Citi Bike, a bicycle sharing system in New York City, publishes data on trips taken within the system, with one CSV file per month from July 2013–February 2014 [5]. For each trip, the files contain the start and stop times as well as the start and stop stations with name, ID and exact coordinates. Furthermore, every trip entry provides the ID of the bike used and the gender and birthyear of the user that made the trip. Additonally, the entry gives information about the user type, distinguishing between "Customer" (short-term pass holder) and "Subscriber" (annual member).

Unfortunately, Citi Bike does not publish user IDs so that it is impossible to associate trips with a specific user. While this is understandable from a privacy perspective, it limits us to more general analyses about gender and age.

3 Architecture

3.1 Database systems

We modeled the Citi Bike data in two different database management systems, hoping to draw on the strengths of each for performing different types of queries.

PostGIS One database is a PostGIS[2] spatial database with two tables.

The first table, "**station**", comprises as primary key the *station ID* and the columns *station name* as well as *point* that stores the station location as a PostGIS point geography. **station** has two indexes: one over the *station ID* since it is used as foreign key for joining the **station** with the **trip** table, and another over the *point* column because the location is used for many calculations, e.g. for showing a station on the map or for calculating the distance of a trip.

The second table, "**trip**", has the columns *bike ID*, *start time*, *end time*, *start station*, *end station*, *user type*, *birthyear* and *gender*. The primary key of trip is the *bike ID* composed with *start time* since this uniquely identifies a trip. This table has an index over the primary key as well as over the *bike ID* for future analyses of single bikes.

Neo4j The other database is a Neo4j[1] graph database, accessed in embedded mode (not through a standalone server).

The database stores nodes of three types: *Trip*, *Station*, and *Bike*. On *Trip* nodes, attributes store the start/end times and user information; *Station* nodes store the Citi Bike station ID, station name, and geographical coordinates; *Bike* nodes store just the bike ID.

Three types of relationships connect each *Trip* node with the *Station* and *Bike* nodes relevant to that trip: *STARTS_AT* and *ENDS_AT* relationships point to the start/end stations, while a *USES* relationship points to the bike used.

3.2 Application

We implemented a multi-layer Java application on top of the databases. The general architecture of the resulting system is depicted in figure 1.

Importing We designed our system to be modular and scalable. Therefore the process of importing, filtering, and providing data to our system was designed to be standardized and usable across different database structures.

Our parser is configurable in both buffer size and data source location. Each database module receives the parse data and loads it into the respective database. This modularity allows us to add and replace database infrastructures as needed. The two current databases independently load data into their databases.

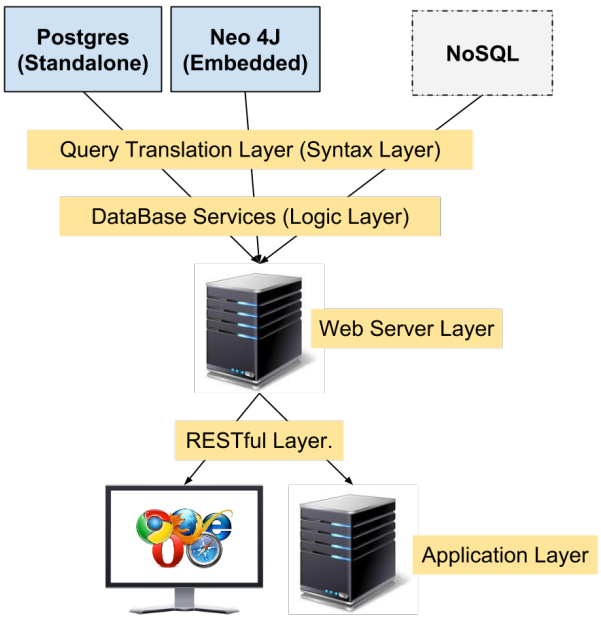

Fig. 1. System Architecture.

Querying Each database module is responsible for exposing Java abstractions for queries. We were able to offload the transaction management, query translation, index defining onto the Spring Framework [4]. Spring uses the convention of repository representation for mapping java method calls to native query methods. By convention each repository represents a specific data type. We were able to set up repositories in each database module. We were also able to customize our repositories to achieve more advanced queries.

We then added another layer of abstraction that we called the service layer. Services are essentially collections of repositories. We created a service for each of the database modules. Repositories have a common tendency to have overlapping queries. Therefore it is up to the service layer to optimize the most efficient query when there is such a overlap. The next layer on the stack is the controller layer, which presents the mapping from REST HTTP calls to service method calls.

REST API and Web Service To make client code simple and efficient, we exposed a REST API over the databases using HTTP and JSON.

One challenge was designing REST URIs to map to respective queries. Some queries were simple; for example, `/api/stations` requests all stations. However, queries with complex parameters appeared more challenging; since users cannot post new data, we considered disguising more complex queries as POST requests with the query parameters encoded as JSON in the body of the request.

Client Applications The current system allows users to define any type of visualization and/or analytics platform given that they can consume a REST API. Most simply, web clients can use AJAX. One feature not supported by our system is notifications when the state of the data has changed. This was a design trade off: we assumed data would rarely be inserted relative to the amount of time it is queried. Most analysis and visualization specialists use data snapshots.

4 Features

4.1 Implemented queries

We sought to incorporate the strengths of each database model in several different aspects. First, we focused on using PostGIS for spatial queries, and Neo4j for queries of a more structural nature. Then, in attempting to use both databases together to provide the functionality needed for the client application, we used PostGIS more for segmenting and aggregating the data, and Neo4j for then investigating a specific segment of the dataset.

In both databases, we implemented queries to get all records of a specific type (trip, station, or bike) and to get an individual record by ID or name. We also implemented some more specialized queries, including:

- Find trips according to distance, duration, time period, user characteristics, bikes or stations used (PostGIS)
- Count total trips to or from a particular station (PostGIS)
- Find distance between a particular pair of stations (PostGIS)
- Get all trips using a particular bike within a certain time period (Neo4j)
- Count total trips to or from a particular station to each individual other station (Neo4j)

4.2 Client interface

Our demo client application uses Google Maps to visualize data. It creates a basic JQuery API layer to pull data from a running web service. We were not able to implement full exploratory functionality, but we were able to show one specialized feature that incorporates both databases: the outflux of trips from a single station. The user can interactively select different source stations. Animations show a more informative representation of where bike traffic was going, with animation speed relative to the total number of trips between a particular pair of stations. The client application still needs more work to take advantage of the server-side functionality.

5 Conclusions and Future Work

Overall, we were successful in modeling and exploring the Citi Bike data, and creating a framework for analysis and visualization.

Our current implementation sets the basis for a full end to end framework for abstracting away data management for analytics and visualization. We found replicating data on multiple databases requires a rather indepth benchmark and assessment for which databases were more useful for different situations. We did not have time to set up benchmarks or apply sophisticated optimizations, which would be necessary as the project progresses.

Currently the REST API is not mature enough for deployment. There is still a lot of work in standardizing the URI naming conventions with regard to consistency and future query expansion.

We would also like to improve our client web application. The current feature shows how to incorporate REST calls that utilizes two databases to provide base data to build a visualization. We would like to show trips based off temporal, user, and spatial parameters as well. The REST API calls are currently available but the client side application does not utilize them. Therefore, the next step for the client application is to complete the existing features utilizing the rest of the REST API calls.

References

1. Neo4j, http://www.neo4j.org
2. PostGIS, http://postgis.net
3. QlikView, http://www.qlik.com
4. Spring, http://spring.io
5. System data, http://www.citibikenyc.com/system-data
6. Tableau, http://www.tableausoftware.com