Intelli Invest

W205 Fall 2015 Project Report

Max Yan

1. Introduction

The goal of Intelli Invest is to make real estate investment in the US easier for investors. There are over 100 major cities, 3,000 counties, and 40,000 different zip codes over the US where one could potentially buy an investment property, yet an investor has no easy way to get data, historical or current, on the price, rent, population, crime, and school about any particular neighborhood. Furthermore, it is impossible to compare two or a handful of zip codes on a high level along various metrics such as rental return, price fluctuation, and school ratings.

Intelli Invest attempts to solve this problem by gathering data from different sources, batch or live, summarizing data in an intuitive way, and allowing flexible data visualization from the front end. This report discusses the architecture of the application, the acquisition and management of data, and some implementation details of the system.

2. Architecture

The system includes three parts: a back-end which downloads, parses, and pushes data into the central database, a mid-end layer which acts on user requests to load data from databases or APIs, and a front-end which can be either an IPython notebook or a web-based dashboard for data visualization.

2.1 Requirements

Intelli Invest has three major requirements from a business point of view – a common database handling layer, a central user requests handling module, and a light weight front-end.

Back-end: Common Database Handling

The common database handling component in the back-end is required for extensibility and maintaining best practices of batch processing. Currently, Intelli Invest stores data from Zillow, US Census, Great Schools, and Google Geo API.

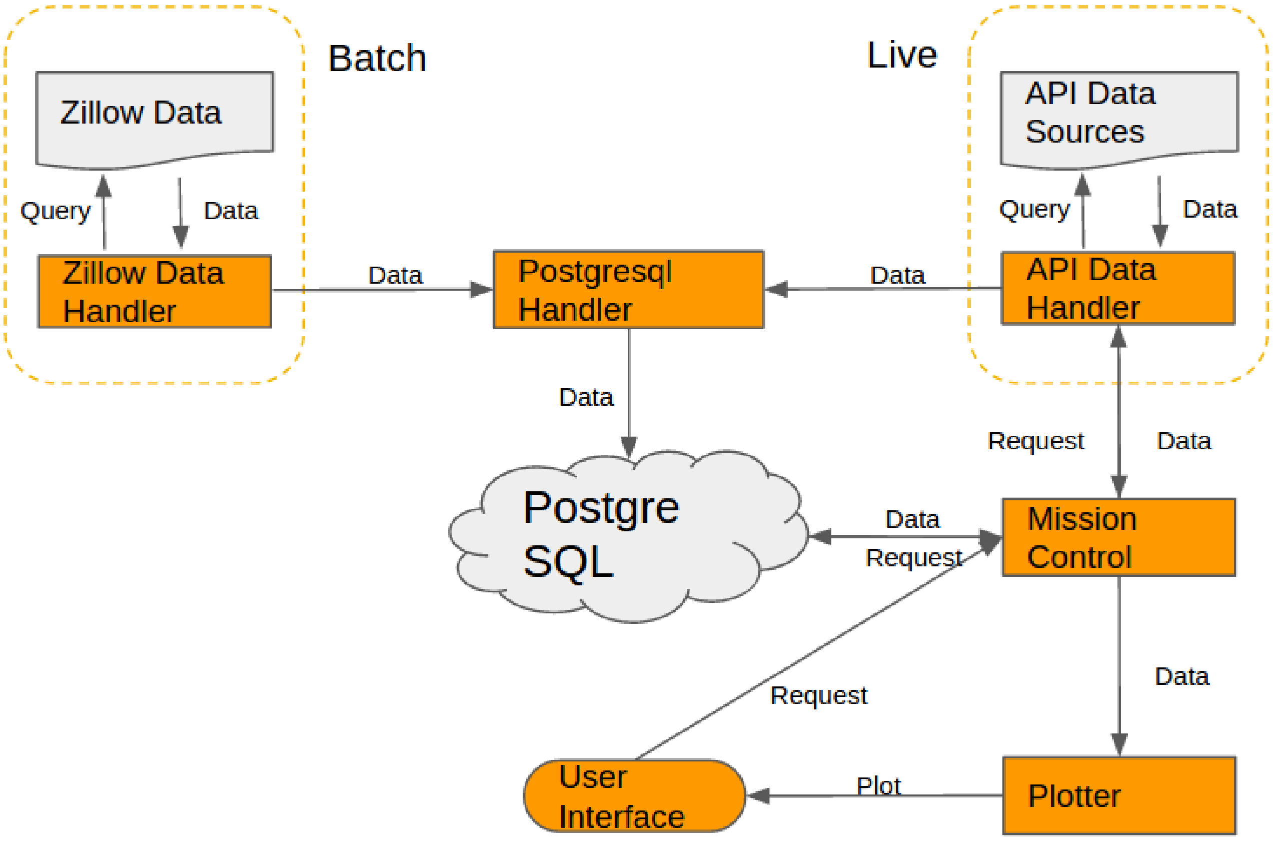
Mid-end: Central User Request Handling

We want the system to be accessible to every investor, and therefore the only thing a user has to do is to specify what he / she wants plotted, leaving the data retrieval and summary to the system. Since the system relies on data from multiple sources, the mid-end component acts like a task scheduler, knowing what data are requested, where to look for these data (i.e. where to look first and where to next), what sanity checks are needed, and what to do with the data in the end.

Front-end: Lightweight

The front-end should be kept as light as possible – responsible only for sending user requests to the central mid-end component, and rendering visualizations. The reason is that end-users could be accessing Intelli Invest from mobile devices in the future and therefore holding lots of data in front-end memory is undesirable.

2.2 General Architecture
The diagram below shows a high level architecture of the system.



Zillow Data Handler, **API Data Handler**, and **Postgresql Handler** together represent the backend of the system. Within a Lambda structure, *Zillow Data Handler* is considered the batch process, querying the Zillow Research Data periodically and updates the database with the new data only, whereas *API Data Handler* is the live / streaming process, querying and pushing data into the database as users requested. A *Postgresql Handler* is constructed to provide a common interaction layer with PostgreSQL, such that a developer may push data in any form to PostgreSQL without worrying too much about details of parsing and syntax.

Mission Control is a mid-end process and acts as a central task scheduler for user's requests. As in the diagram, *Mission Control* is aware of what plot was requested by the user, interprets the plotting needs as data needs, and then interacts with both PostgreSQL and API Data Handler for data, and finally passes the data onto the *Plotter*. The *Mission Control* component also degrades gracefully in case it cannot find any data or on other exceptions.

User Interface is the front-end solution for data visualization. Currently, the front-end consists of an IPython notebook and a web-based dashboard. JavaScript rendering is used and raw data is stored remotely to keep front-end resource consumption to the minimum.

2.3 Component Structure
The table below provides a high-level description of the components and their source code.

Name	**Directory**	**Description**
zillow_data_handler.py	/Project	Back-end process to download and push Zillow

		Data
api_data_handler.py	/Project	Back-end generic interface for loading data using one of the APIs
major_cities_handler.py	/Project	One-off process that runs on system deployment, loading the population and coordinates of all US major cities (i.e. population > 100,000) and store them in PostgreSQL
postgresql_handler.py	/Project	Common PostgreSQL interaction module
data_models.py	/Project	A single place to hold datamodels (i.e. table name and schema) of all PostgreSQL tables, for scalability
mission_control.py	/Project	Mid-end central task scheduler for processing users' plotting requests
plotter.py	/Project	Mid-end / Front-end component responsible for aggregating the data and doing visualizations
population.py	/Project/api	For any zip_code, find the population of the closest (i.e. distance < 50 miles) major city (i.e. population > 100,000) using US Census data and Google Geocode API
google_geo.py	/Project/api	Common interface to extract information (e.g. state, county, coordinates) about a location using Google Geocoding API
great_schools.py	/Project/api	Common interface for the system to retrieve school ratings via GreatSchools.org API

3. Data Acquisition

3.1 Sources of Data

The system relies on data from Zillow and a number of APIs.

Data from Zillow Research portal are primarily the monthly price and listed rent data for different types of properties (e.g. single family home, condos, multi-units etc.), at different levels of granularity (e.g. state, metro, city, county, zip code, and neighborhood), between October 1996 and October 2015. These data are stored as .csv files and hosted in Zillow's remote server. For the purpose of the project, we are primarily concerned with the data for single family home at a zip code level. The size of the data is medium, at approximately 250MB. The data acquisition strategy for Zillow data is the batch process by *Zillow Data Handler*, and is triggered monthly.

In addition, the system also relies on data from external API calls. Currently, Intelli Invest supports acquiring data through US Census API and GreatSchools API. These data come into the system on a per-request basis and responses to API calls come as .json files, which are parsed subsequently by individual parsers.

While not currently needed due to the limited number of APIs supported, an *API Data Handler* module was also created to provide a central interface for doing all the API calls more systematically.

3.2 Transformations
There are three types of transformations for Intelli Invest: that of Zillow Data, that of API responses, and that of physical locations.

Data retrieved from Zillow Research Portal comes in the format below:

Id	**RegionaName**	**RegionID**	**City**	**State**	**2015-10**	**2015-09**	**2015-08**
1	77493	568dfv	Katy	TX	150,000	152,000	148,000
2	77545	582efg	Keywood	TX	210,000	220,000	200,000

There are a couple of issues with the raw format:

- Columns such as 'Id' and 'RegionID' are fields completely internal to Zillow;
- Column 'RegionaName' was; really a zip code
- Having columns '2015-10', '2015-09', '2015-08' is a concise way of displaying data, but inefficient for performing time series analysis later on;
- It is impractical to change the schema of any table every month we get new data.

As a result, we designed the schema for our time series data and implemented parsing logic in *Postgres DB Handler* to transform the above table in the following:

key	**zip_code**	**state**	**year_month**	**median_price**	**median_rent**
77493_201510	77493	TX	201510	150,000	NULL
77493_201509	77493	TX	201509	152,000	NULL
77493_201508	77493	TX	201508	148,000	NULL
77545_201510	77545	TX	201510	210,000	NULL

The data post transformation come with meaningful and indexed columns, making queries faster and more intuitive.

3.3 Aligning Different Sources
We noticed that since the system retrieves data from different sources, these sources tend to have slightly different encoding practices for the same physical locations. As an example, while Zillow refers to the city “St. Louis, MO”, US Census refers to the same place “Saint Louis city, Missouri”, and GreatSchools refers to it “St Louis, MO”. And since zip_code is not available for all data sources of the system, this makes it impossible to perform joining and aggregations on the name.

As a solution, we created another API that relies on Google Geocode API, and converts different encoding of physical locations into a universally unique key (named place_id), and perform join operations, when required, on the new place_id, instead using the any combination of the name of the county, city, metro, or state.

4. Data Storage

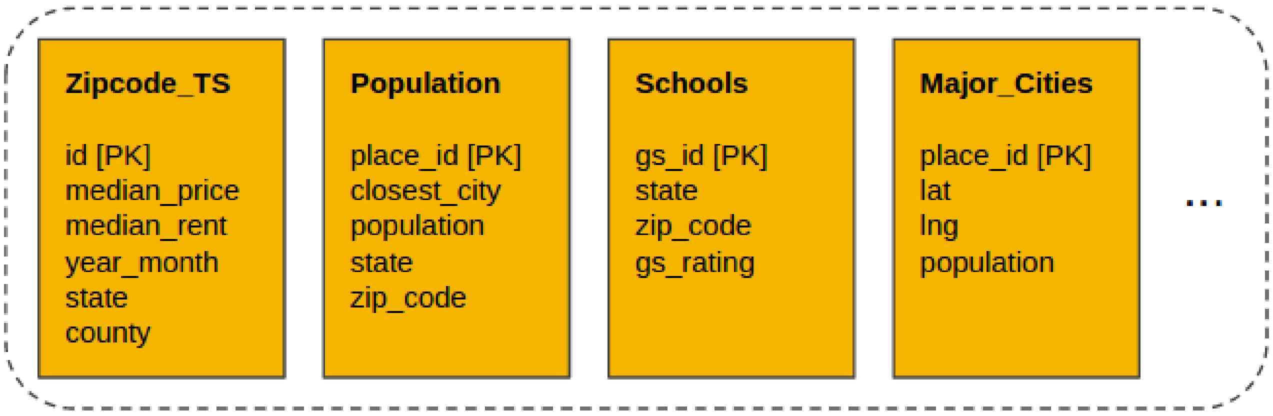
4.1 Choice of Database
We explored storing data in both DynamoDB (i.e. NoSQL document store) and PostgreSQL, and eventually decided to keep the data in PostgreSQL, with the following reasoning.

First, the system has a relatively simple structure and very stable schema, where each data source (and additional future data sources) has its own table. This dictates that the flexibility offered by a document store would have little advantage over a relational database.

Second, from the user's perspective, the system needs to have low latency. In other words, a user should submit a plot request and get results plotted as quickly as possible, depending on the availability of data. We ran multiple database read and write experiments in a controlled environment with DynamoDB and PostgreSQL, and on average PostgreSQL is at least 60 times faster on writing and 5 to 10 times faster on reading (See Appendix for more details).

Lastly, from a continuous development point of view, configuring a new table is much more straightforward and therefore greatly reduces the complexity of the database interaction layer.

4.2 Arrangement of Tables
Below is a general depiction of the tables in the PostgreSQL database.



There are two features to this arrangement:
- Each table represents one and only one data source, interacts with one method in the common PostgreSQL Database Handler, and has one defined schema in the datamodels.py module;

- No foreign keys available in any tables, and each table exists independently from others

The reason for this implementation is three-fold:
- Orthogonality and Integrity: since each table interacts with one and only one component in the DB Handler, a developer only needs to test the functionality of that module thoroughly, reducing the potential for technical errors.

- Easier Manipulation: since each table exists independently from others and represents a single data source, a developer can simply create a new table for any new data sources added. In addition, in case of errors or bugs in the handling of one data source, one only needs to truncate the data in that specific table, without compromising data produced by other processes.

- Reduced Complexity for DB Handler: we would like to keep only minimal complexity for the common PostgreSQL DB Handler module. This is partly achieved by exposing only .put and .get methods to the user and developer, without the trouble of complex joining and aggregating operations on the database layer. Furthermore, we believe performing these operations on the database end introduces database latency and likelihood of locking tables, which negatively impacts user experience.

5. Implementation Details

5.1 Population Data

The population for any zip code that a user sees in the final visualization is the population of the major city closest to that zip code based on 2013 Census Bureau Forecast.

We decided to implement it this way because at a zip code level, the population of that zip code does not have a lot of economic bearing for the economic diversity and stability of that area. Population data is ultimately a measure of employment prospect and infrastructure, on a city or metro level, and therefore, we believe that people can and do live in one place while commuting to a nearby city for work.

The implementation follow through the below steps:

1. On deployment, a static process (i.e. major_cities_handler.py) will combine the information from Google Geocode API (name, state, latitude, longitude) and US Census (name, state, population) of all major cities (with a population greater than 100,000) in the US;
2. On user request, the system will fetch the coordinates of the requested zip codes;
3. The system will first calculate the Euclidean distance between a zip code to all major cities in that state to determine the closest city;
4. The system will then calculate the actual haversine distance between the zip code and its closest major city;
   - If the actual distance is less than or equal to 80 kilometers, which is approximately 40 minutes driving on US highway, we will use the population data of that major city;
   - Otherwise, we will not use the population data and the value will remain NULL;

5.2 GreatSchools Data

GreatSchools.org provides an API call that works on (zip code, state) pair, which makes querying a lot easier. The actual implementation is as follows:

1. A user request school information for a (zip code, state) pair;
2. The GreatSchools API to return 20 schools within a radius of 5 miles from the center of the requested zip code, all the actual parameters is definable by user;
3. GreatSchools API then parses responses to get gs_id, name, state, zip_code, and ratings;
4. GreatSchools API directly pushes this data into the db.

A conscious decision was made here to store the raw data as opposed to processed data (e.g. taking the average rating) because the later leads to loss of information such as the name and id of the school, and because one would and could do more with raw data in the future iterations.

5.3 Mission Control

Mission Control is the center piece in the mid-end processes of Intelli Invest. It is designed with three key principles:

- Always check if data already exists in the database before going through APIs;
- Always only do the minimum required work and nothing more;
- Always update the database after getting results back from API calls

All three principles are engineered towards lower latency on user requests and less reliance on external data, which we have no control of.

The first principle ensures the system will not conduct often expensive but completely unnecessary API calls for existing data. It is also a measure in place to maintain integrity of the system, as the structure of API responses may change due to its external nature, which directly imposes downtime to the system.

The second principle governs that if the user requested 50 entries and if the database already has 30 entries, then unless the 30 entries were outdated, *Mission Control* will only request an additional 20 entries from the API. This is to keep traffic to external data sources to the minimum and keep everything within the system's control.

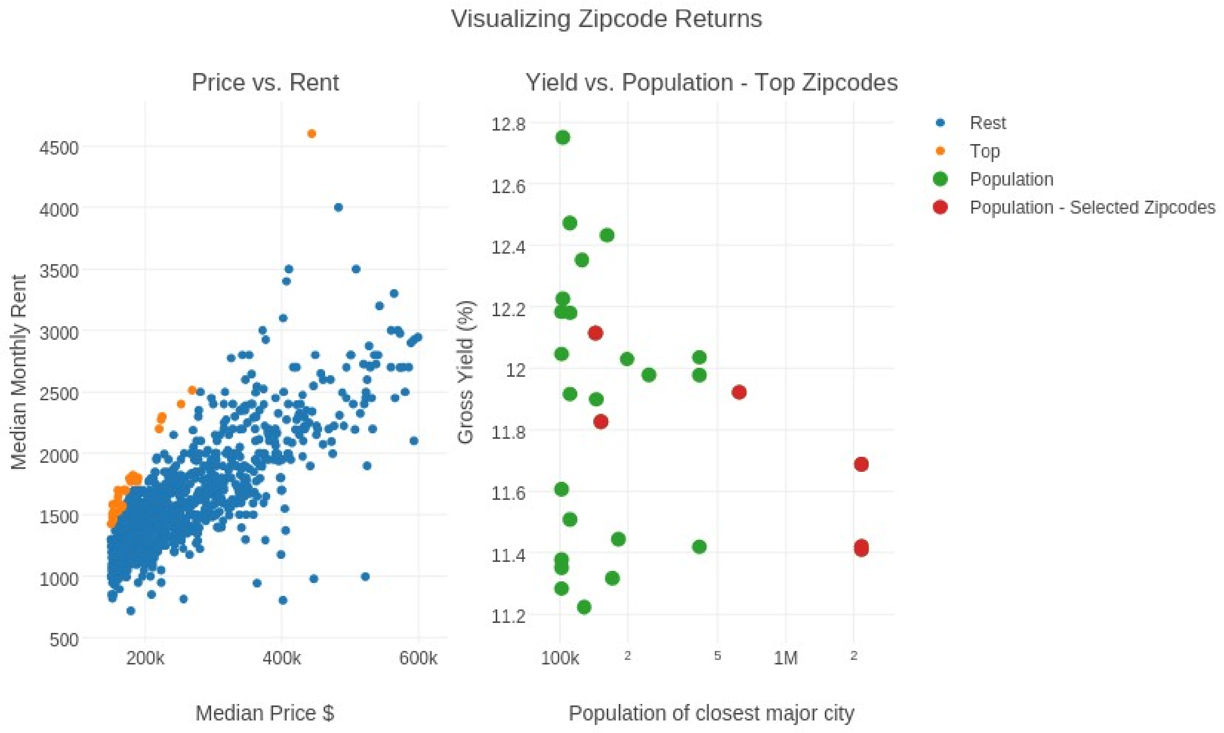
The third principle means that once the data is returned from the APIs, we will use the data directly for the current request, and meanwhile we will also store these data to the central database, to ensure that if shortly after some user requests the same data or data that partly overlaps with the current data, *Mission Control* is able to locate it in the database without going through APIs again.

6. Sample Results

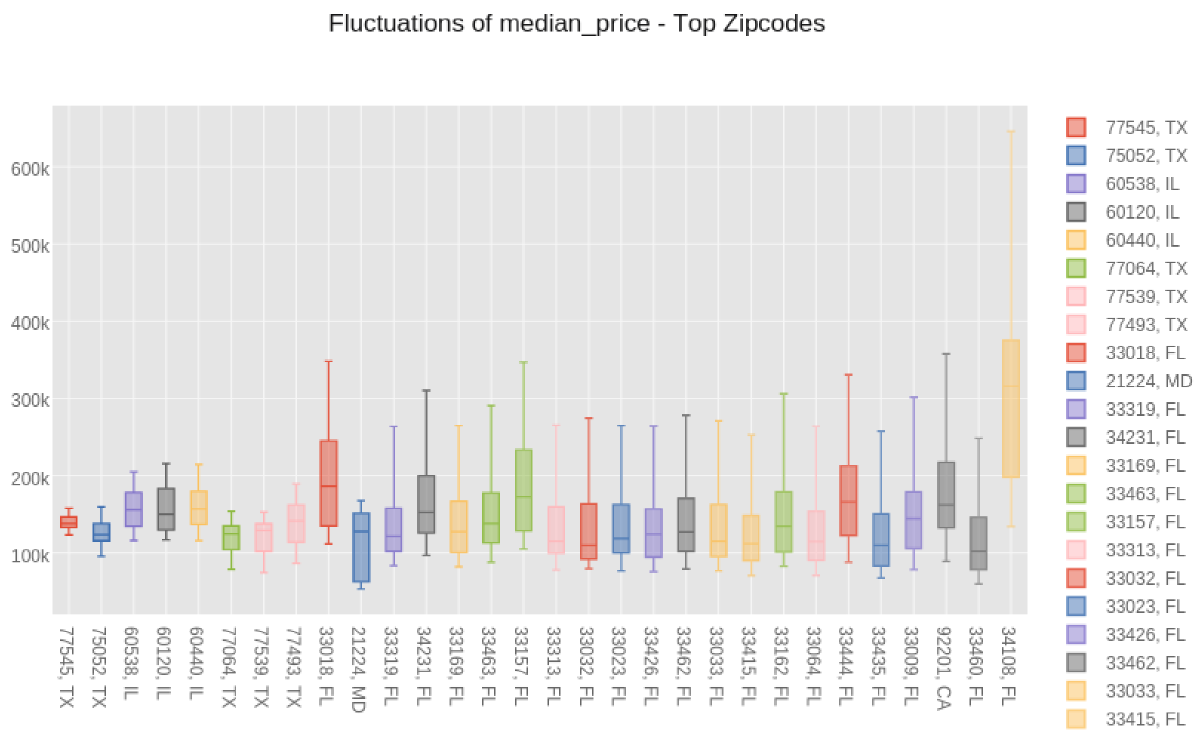
This section showcases some sample results of the system. In reality, these results are interactive charts rendered with JavaScript. A sample front end is also built with Shiny and available at https://maxyan.shinyapps.io/Intelli_Invest_Final.

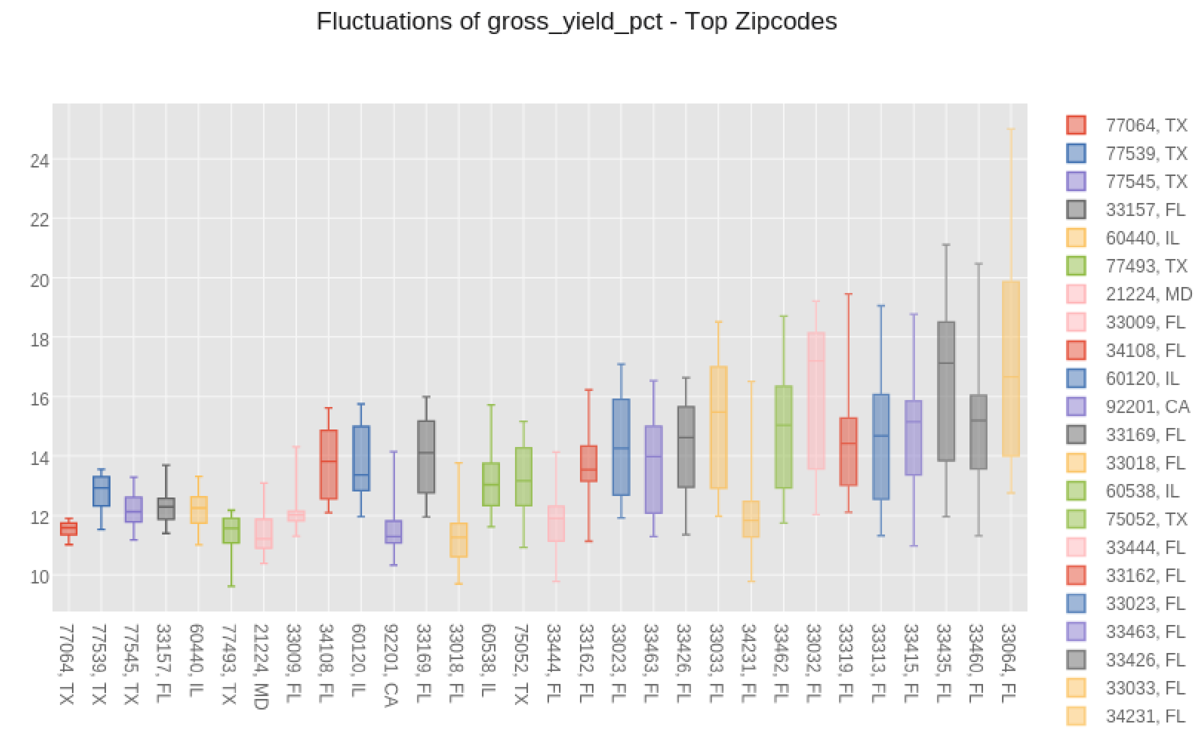
The first plot shows the median price versus median rent in October 2015 for all the zip codes between $150,000 and $600,000 (left plot). The system also separates zip codes that tend to have a higher rental return from the lower ones (i.e. orange dots versus blue dots).

On the right subplot, the system shows the log-scale closest major city population and the all the top zip codes (i.e. orange dots on the left). The red dots are a list of selected zip codes for comparison.

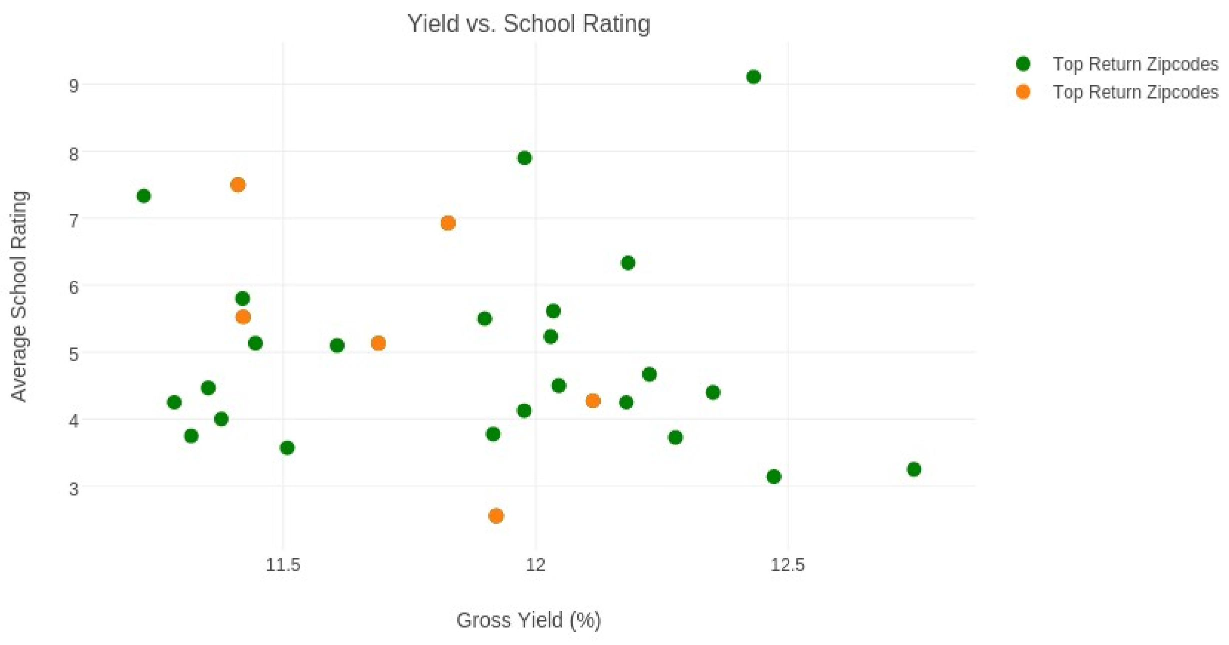


The next two plots indicate the variability of both rental return and median price of the top zip codes (i.e. orange dots in the first plot). Zip codes in both plots are sorted in ascending order by their rental return variability and median price variability. As can be seen, some places are clearly delivering more stable results than others.





The last plot is for visualizing rental returns versus the school rating for the top zip codes. Again, we can easily observe that even though all zip codes look similar in terms of their rental returns in the first plot, they differ vastly in terms of the quality of schools in the local area, an indication of where people prefer to live.



7. Appendix

7.1 Performance Comparison

We conducted multiple tests with PostgreSQL and DynamoDB using Zillow data (20,000 rows in the original .csv file). The time is average time of three runs, rounded down to the nearest second if greater than one second.

	PostgreSQL	**DynamoDB**
Read entire dataset	1	7
Filter on primary key	0.3	1
Filter on columns	0.6	3
Filter on multiple columns	0.7	4
Write to Db	1	1,000
Batch Write to Db	1	60