# SmartMove: Moving Made For You

Dan Gallagher, Kelly Feng, Jiwon Kang, Steven Wu

## Introduction

When moving to a new neighborhood, it is relatively easy to find general statistics (crime, education, etc.). However, one's "compatibility" with a neighborhood is slightly more intangible, despite being a critical factor to one's fulfillment within their environment. We hope to change that, making the moving process more personal and helping match users with locations that fit the complexities of their lifestyle.

Pursuant to this, our core realization was a simple one: you can tell a lot about a neighborhood's lifestyle based on its businesses. On its face, this could be as simple as identifying opportunities (or lack thereof) to enjoy the things you love. Even further, understanding the businesses that comprise a neighborhood helps us gain a better understanding of that neighborhood's values. Are a lot of the businesses wheelchair accessible? Are they child-friendly? Exploring these often overlooked aspects helps us gain a more robust and more individualized understanding of a region.

Our app thus combines business data from Yelp with more "traditional" neighborhood statistics from Zillow and the US census, extracting potential areas of insight and equipping our users with the ability to mix and match a variety of features to find their perfect home.

## Data

- **Yelp businesses** - https://www.yelp.com/dataset
    1. **Description:** A 49.3 MB, 209,393 row JSON file representing properties of various businesses in different locations. The properties include star ratings, zip code, latitude and longitude, category tags, and various business attributes (such as whether a business is wheelchair accessible or good for kids). While most of the data could be extracted into a relational style, business attributes, category tags, and business hours were all JSON dictionaries. We use this data to let our users filter by and explore relevant business-related information
    2. **Statistics**:
       Median star ratings: 4; Standard deviation: 1.05
       Median review count: 11; Standard deviation: 149.5
       States with over 1000 businesses represented: 8
       State with most businesses represented: AZ (60803)
       Number of businesses with no attributes listed: 17923

- **Zillow (home values & rental prices) :** https://www.zillow.com/research/data/
    1. **Home values description:** csv with 30329 rows and 309 columns of average home values for every month of every year from January 1996 to January 2021. We use only the last 6 months of data.
       **Statistics:**
         Minimum average value: $14980
         Maximum average value: $6557351
         Mean value: $244936
         Standard deviation of home values: $248784
    2. **Rental prices description**: csv with 2485 rows and 309 columns of average rental prices for every month of every year from January 1996 to January 2021. We use only the last 6 months of data.
       **Statistics:**
         Minimum average rent: $715
         Maximum average rent: $27141
         Mean rent: $1856
         Standard deviation of rent: $975

- **US Census Bureau**: County Population by Characteristics (2010-2019) -
  Table:
  https://www.census.gov/data/tables/time-series/demo/popest/2010s-counties-detail.html
  Dataset:
  https://www.census.gov/data/datasets/time-series/demo/popest/2010s-counties-detail.htm
    1. **Description**: A 172 MB, 716,376 row x 80 column csv file containing breakdown of US population by age, gender, and ethnicity. We use this to break down demographic information for relevant zip codes.

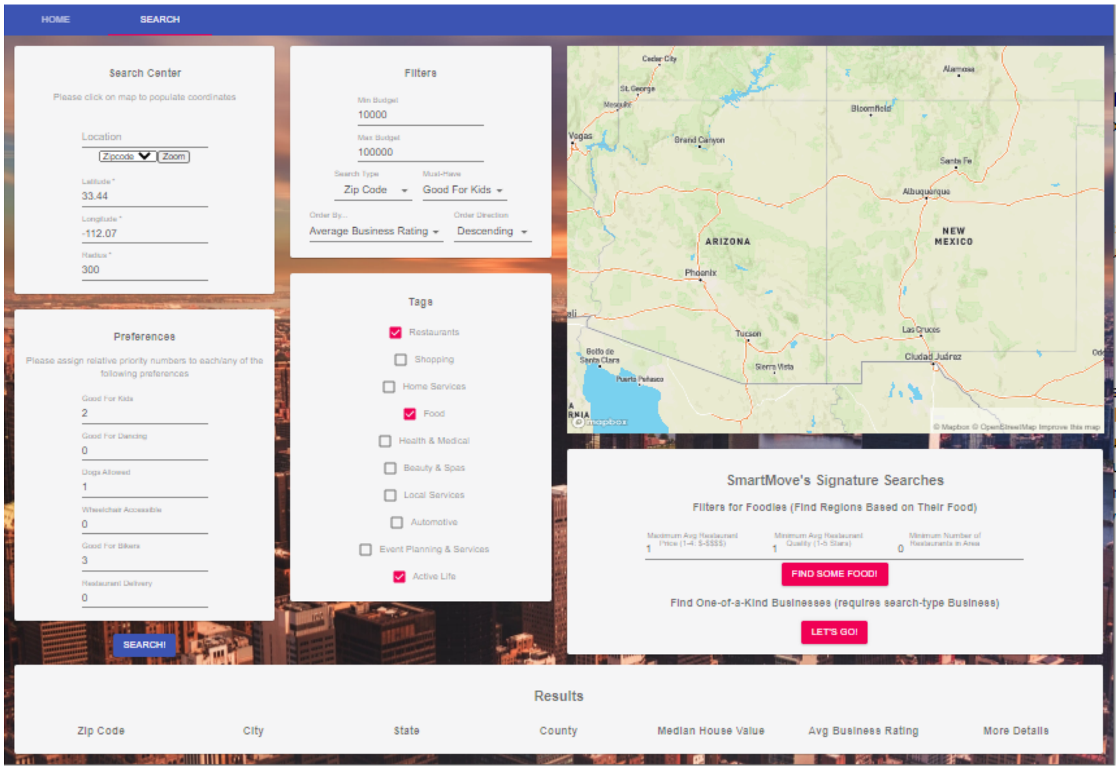
ii. **Statistics**: Total US population in 2019: 328,239,523

- **Zipcode information dataset:** https://www.unitedstateszipcodes.org/zip-code-database/
  - **i. Description:** A 3 MB 41,701 row x 10 column csv file containing information about zip codes. This includes latitude, longitude, city and state.
  - **ii. Use:** We used this dataset to help map zip codes to latitude and longitude and also get the city, state of where the zip code is located.

## Architecture

SmartMove uses Pandas, MySQL, React, and Node.js to build a highly modular filtering system to find neighborhoods and businesses of interest. It consists mainly of a "Search" page, where users initially filter results and find basic summary statistics, and a "Details" page, where users can drill down on neighborhoods for in-depth, graphically presented information. We used zipcodes as our metric of a "neighborhood" with the hope that this would provide the right amount of granularity, particularly around large cities. After all, two sides of the same city can present vastly different experiences.

### Search Page



Search Center

Please click on map to populate coordinates

Location

Latitude *: 33.44

Longitude *: -112.07

Radius *: 300

Preferences

Please assign relative priority numbers to each/any of the following preferences

Good For Kids: 2

Good For Dancing: 0

Dogs Allowed: 1

Wheelchair Accessible: 0

Good For Bikers: 3

Restaurant Delivery: 0

Filters

Min Budget: 10000

Max Budget: 100000

Search Type: Zip Code

Must-Have: Good For Kids

Order By...: Average Business Rating

Order Direction: Descending

Tags

- ✓ Restaurants
- ☐ Shopping
- ☐ Home Services
- ✓ Food
- ☐ Health & Medical
- ☐ Beauty & Spas
- ☐ Local Services
- ☐ Automotive
- ☐ Event Planning & Services
- ✓ Active Life

SmartMove's Signature Searches

Filters for Foodies (Find Regions Based on Their Food)

Maximum Avg Restaurant Price (1-4: $-$$$$): 1

Minimum Avg Restaurant Quality (1-5 Stars): 1

Minimum Number of Restaurants in Area: 0

Find One-of-a-Kind Businesses (requires search-type Business)

Results

| Zip Code | City | State | County | Median House Value | Avg Business Rating | More Details |
|---|---|---|---|---|---|---|

The focus of the search page was modularity- users should be able to mix-and-match a variety of filters to find exactly what they are looking for. This led our main driver of functionality to be a small number of massive, highly dynamic queries. All results have a “More Details” field, containing buttons that navigate to a details page populated with information about that row’s zipcode.

*Search Center and Map*- Users can select coordinates, either by typing them or by clicking on the map, and select a radius (in miles) to define their geographic search window. Thus, users can narrow down their search to specific regions, if they know they have to move to a general area (say, for a new job).

- The latitude and longitude points are inputted into a Haversine formula, which is used to find the distance across the circumference of a large sphere, similar to Earth. It is thus frequently used as a measure of distance between two coordinates
- The user can also type in a zipcode, state, or city and utilize the “zoom” functionality to automatically zoom in to that location on the map (based on coordinates stored in our database for each location), while simultaneously inputting those coordinates into the latitude and longitude fields.

*Filters*- Users may mix-and-match a variety of different filters, which determine which results are returned and how they are ordered.

- Search Type: Specifies whether to return zipcodes (with aggregate statistics) or individual businesses. Zipcode-based filters also filter the selection of businesses by excluding and businesses in zipcodes that do not fit the criteria.
- Budget: Specifies a range for a region’s median home values (based on the Zillow dataset), which can provide a relative approximation of how expensive a house might be in the neighborhood.
  *Note:* While we originally planned to also include the opportunity to search by rental index as well, the rental index data actually turned out to be quite sparse in terms of represented locations, and so was not particularly useful as a filter.
- Must-have: Finds zipcodes with particularly high ratios of businesses that were listed as having a given attribute, thus allowing users to filter by “lifestyle” factors most important to them. In order to maintain simplicity for the user, these filters were made booleans via a behind-the-scenes thresholding process. Each zipcode’s proportion of businesses with the selected feature was compared against each other in the form of percentile ranking. Only zipcodes in the 70th percentile or higher were chosen as being true for the given feature.

- Order: Specifies attributes by which to order results. Options dynamically change when Search Type is changed.

  **A Note on Must-have:* Note that not every business in the Yelp dataset had a dictionary of attributes. To form the denominators of our ratios, such businesses were not considered. However, businesses that contained a dictionary but no value for the selected attribute were considered, based on the logic that, if a business took the time to list certain features but not others, the unmentioned features are probably not core to that business. There is of course some error in this assumption, especially since attributes can also come from user reviews, but given the sparsity of the attribute data, we thought it more accurate than excluding all nulls from the equation.

*Preferences*- While the Must-have filter is helpful for finding regions with “make-or-break” conditions, the end goal of our app was to tailor moving to the individual. To this end, we allow users to weight lifestyle factors by priority (the number may be anything, as they represent priority relative to the others). We use this information to create a “Compatibility Score” via a weighted sum of the various attribute percentiles, which we then use to order results.

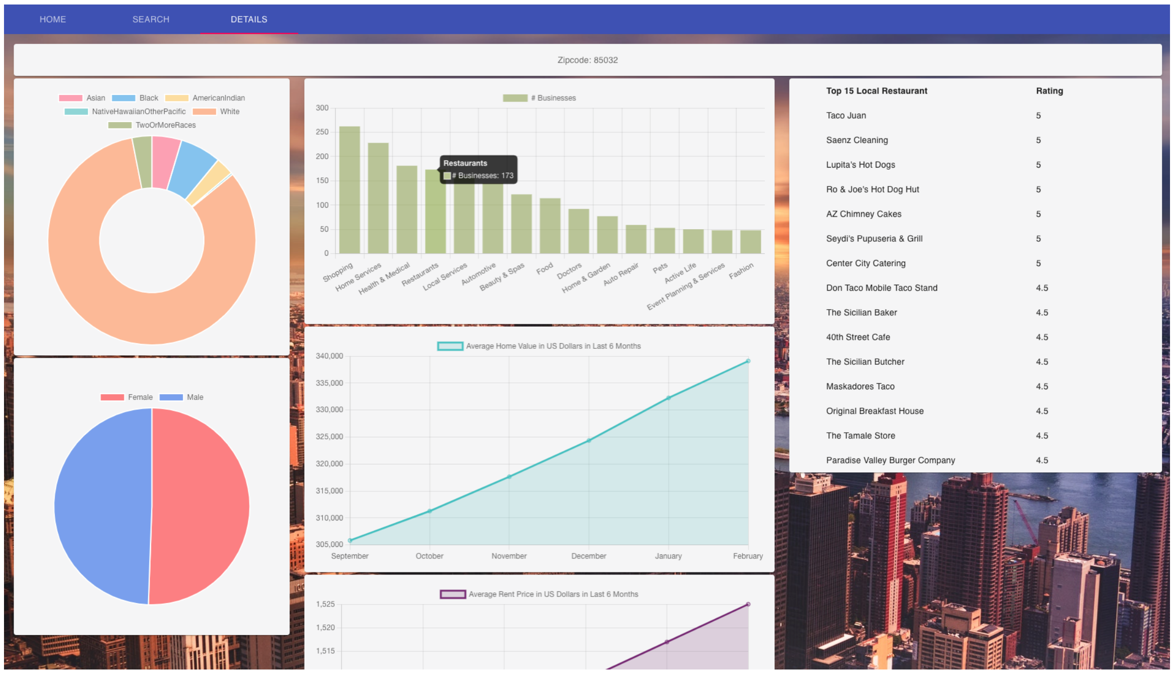
*Tags*- When searching for businesses, users can filter down to only businesses with one of the specified tags.

*SmartMove’s Signature Searches*- An area for users to utilize more advanced filters designed by the SmartMove team.

- “Filters for Foodies” filters based on aggregate restaurant information.
- “Find One-of-a-Kind Businesses” finds all businesses in the given area that have a unique name, allowing users to find “mom-and-pop” stores.

**Details Page**

The details page contains information about the zip code indicated at the top. The doughnut chart (top left) shows the population in the zip code grouped by ethnicity. The pie chart (bottom left) shows the population in the zipcode grouped by gender. The bar chart in light green (top center) shows the number of businesses grouped by business category within the zip code. The line chart in light blue (middle center) shows the home values in the last 6 months. The line chart in purple (bottom center) shows the rental prices in the last 6 months. All values of the components of the charts can be displayed by just hovering over the component. Finally, the list (top right) shows the top rated 15 restaurants that are found only in that zip code. The query used to return the top 15 restaurants queries is fairly complex as it uses NOT EXISTS, multiple CTEs and multiple JOINs.



| Top 15 Local Restaurant | Rating |
|---|---|
| Taco Juan | 5 |
| Saenz Cleaning | 5 |
| Lupita's Hot Dogs | 5 |
| Ro & Joe's Hot Dog Hut | 5 |
| AZ Chimney Cakes | 5 |
| Seydi's Pupuseria & Grill | 5 |
| Center City Catering | 5 |
| Don Taco Mobile Taco Stand | 4.5 |
| The Sicilian Baker | 4.5 |
| 40th Street Cafe | 4.5 |
| The Sicilian Butcher | 4.5 |
| Maskadores Taco | 4.5 |
| Original Breakfast House | 4.5 |
| The Tamale Store | 4.5 |
| Paradise Valley Burger Company | 4.5 |

## Database

### Data Ingestion

Originally, the tables were populated following the tutorial on Canvas (MySQLWorkbench). However, we found that it took a tremendous amount of time and crashed frequently, so we populated our tables using python with boto3 to connect to our aws RDS instance(scripts to populate tables are included in the .zip).

### Pre-processing

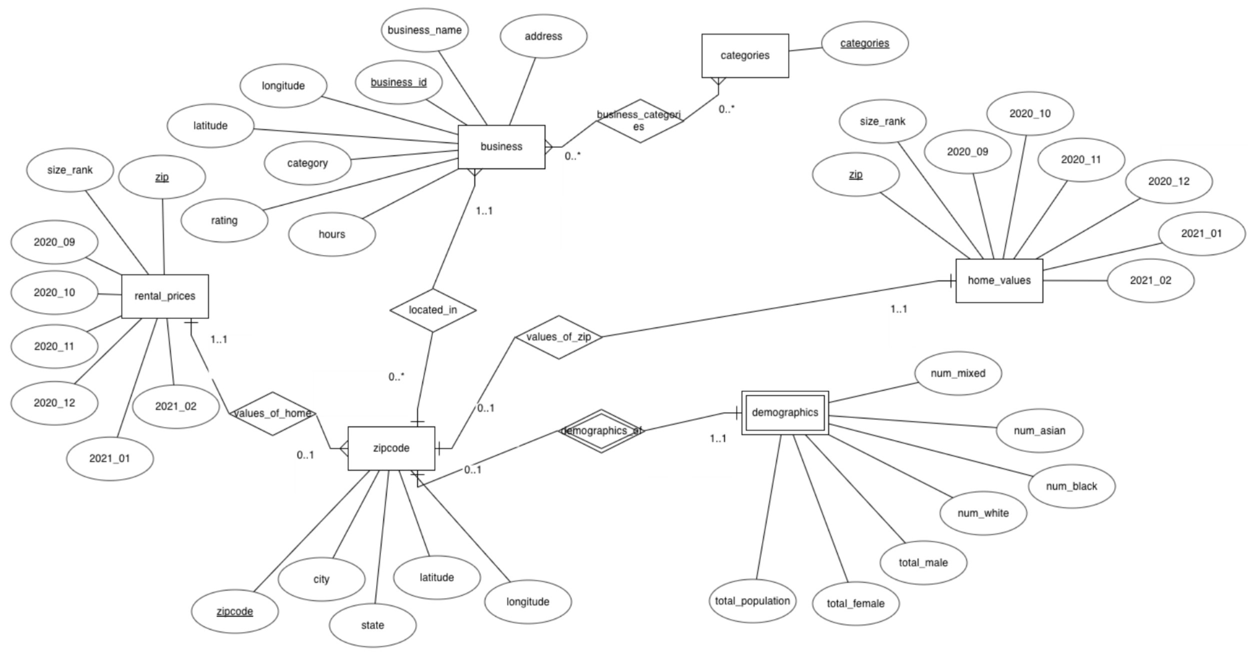
1. Removing Rows: We removed attributes from the datasets that we didn't need. Then, we removed rows that had missing values in any of the columns that we were interested in.
2. Entity Resolution: We changed the names of columns to match the datasets (ie. city and city_name are the same). We also changed values to match the corresponding values in other datasets (ie. value of sex 'F' and 'Female' are the same).
3. Parsing JSON: Some of our most relevant data was in nested JSON format (particularly, business attributes and tags). In order to work with this data relationally, we had to either expand each key-value pair into a new column (as we did for attributes, which were relatively independent from each other and mostly boolean) or explode the values

into new rows (as we did for tags, which often came multiple to a business). In order to keep our resulting dataset as compact as possible, we also sifted through the dozens of attributes to extract the 10 we thought might be most valuable for prospective movers.

   a. Null handling for parsed data: after parsing data, we needed to make a distinction between data for which there was no dictionary, and data for which there was a dictionary, but no key:value pair.

4. Identifying Indices: We checked if every row in the datasets were unique, if they were not unique then we inputted a new column as a key or identify two columns to form a key.

## ER Diagram



## # Instances in Table

| business | zipcode | demographics | rental_prices | home_values | business_categories |
|---|---|---|---|---|---|
| 124891 | 41701 | 12567 | 2485 | 30329 | 542931 |

## Normal Form

FDs for each table:

Zipcode

<u>Zipcode</u> -> {all other attributes}

Business

<u>Business_id</u> -> {all other attributes}

Demographics

<u>Zipcode</u> -> {all other attributes}

Rental_prices

Zip -> {all other attributes}

Home_values

Zip -> {all other attributes}

Business_categories (

{business_id, categories} -> {}

For all the tables, the left side is a superkey thus this is in BCNF and also 3NF.

## Queries

*Percentile-Rank Attributes*- for each zipcode, find the ratio of businesses that listed an attribute as true, out of all businesses that had some set of attributes listed. Rank those zipcodes by percentile, compared to all other non-zero zipcodes in the dataset. Zipcodes with a ratio of 0 were not used because the massive sparsity of our attribute data would lead all nonzero attributes to be over the 70th percentile. However, they were still included in the table to equalize the size of each attribute table to facilitate the creation of our Compatibility Score. This query was run for 6 different tables.

```
CREATE TABLE GoodForKids
WITH att_count AS(
SELECT postal_code AS zipcode,
 (CASE WHEN attributesGoodForKids = 'True' THEN COUNT(if(
attributesGoodForKids = 'True', business_id, NULL)) ELSE 0
  END) AS att, COUNT(business_id) AS total
  FROM business
     WHERE attributesGoodForKids IN ('Unlisted', 'True', 'False')
  GROUP BY postal_code),
att_0 AS
     (SELECT zipcode, att, total, 0 AS att_rat, 0 AS percentile
  FROM att_count
  WHERE att = 0),
att_ratio AS(
     SELECT zipcode, att, total, att/total AS att_rat,
ROUND(PERCENT_RANK() OVER (ORDER BY att/total), 2) AS percentile
     FROM att_count a
WHERE att > 0)
SELECT *
FROM att_0
UNION
```

```
SELECT *
FROM att_ratio;
```

*Zipcode Search*- Combine the specified filters to output zipcodes.

Get the distances for each zipcode from the chosen coordinates within the chosen radius.

Filter by zipcodes where the chosen “must-have” passes our 70th percentile threshold.

Filter by zipcodes which fall within the specified budget (based off Zillow home values data)

Join relevant CTEs together, select relevant fields, grouping by zipcode to find aggregate data like average business rating, order by the specified field, and set a limit of 50 rows.

**Note**: *You may notice that we used LEFT OUTER JOINS for some of our joins here, and also that our attribute table defaults to ‘GoodForKids’ if undefined. This was an intentional decision on our part to reduce the size and complexity of queries as much as possible (given that each was already incredibly large and complex as is), albeit at a negligible performance cost. Since not every zipcode in the zipcode table was represented in every other table, using inner joins would essentially act as a partial filter even if the particular item was left blank by the user. Technically, we could handle this by creating a ternary operator for each join and table statement that does not run any join or create any CTE if a filter is left blank. However, using left joins allowed us to gate our filters with 1-2 ternary operators instead of a great deal more, which we think makes the code more easily readable, expandable, and stable. We only did this since these queries were running so fast in the first place, and since the zipcode table was relatively small (~40,000 rows), so the left joins added very little overhead as compared to inner joins.*

```
WITH coordinates AS
        (SELECT zip,
        69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians( latitude ) ) *
        sin( radians(${latitude}) ) ) )  as distance
        FROM zipcode
        WHERE 69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians(latitude) ) *
        sin( radians(${latitude}) ) ) ) <= ${radius}),
attribute AS
        (SELECT zipcode, percentile
        FROM ` + (req.params.attribute === "undefined" ? `GoodForKids
` : `${attribute} `) +
        `WHERE percentile >= .7),
budget AS
        (SELECT zip, 2021_02
        FROM home_values
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget})
SELECT z.zip, z.city, z.state, z.county, 2021_02 AS MedianHomeValue,
AVG(stars) AS    avgBusinessRating
        FROM zipcode z
        JOIN coordinates c ON c.zip = z.zip
        LEFT OUTER JOIN attribute a ON z.zip= a.zipcode
        LEFT OUTER JOIN budget h ON z.zip = h.zip
        LEFT OUTER JOIN business b ON z.zip=b.postal_code
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget}`
                + (req.params.attribute === "undefined" ? `` : ` AND
percentile >= .7`)
    + ` GROUP BY z.zip
        ORDER BY ${order_key} ${order_direction}
        LIMIT 50;`
```

*Find Businesses*- Combine filters to find businesses

```
WITH coordinates AS
        (SELECT business_id AS id,
        69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians( latitude ) ) *
        sin( radians(${latitude}) ) ) )  as distance
        FROM business
        WHERE 69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians(latitude) ) *
        sin( radians(${latitude}) ) ) ) <= ${radius}),
    budget AS
        (SELECT zip, 2021_02
        FROM home_values
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget})
    SELECT b.name, b.city, b.state, b.postal_code AS zip, z.county,
b.stars AS 'rating'
    FROM zipcode z
    LEFT OUTER JOIN budget h ON z.zip = h.zip
    JOIN business b ON b.postal_code = z.zip
    JOIN coordinates c ON b.business_id = c.id
    WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget} `
    + (req.params.attribute === "undefined" ? `` : `AND
${attribute_db} = 'True' `)
    + (tag_string === '(' ? `` : `AND EXISTS (
SELECT *
FROM business_categories
WHERE business_id = b.business_id
AND categories IN ${tag_string}) `) +
`ORDER BY ${order_key} ${order_direction} LIMIT 50;`
    }
```

*Compatibility Score*- For given filters, find the weighted sum of the 6 attributes to create a compatibility score, then sort by that score.

```
WITH coordinates AS
        (SELECT zip,
        69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians( latitude ) ) *
        sin( radians(${latitude}) ) ) )  as distance
        FROM zipcode
        WHERE 69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians(latitude) ) *
        sin( radians(${latitude}) ) ) ) <= ${radius}),
        attribute AS
        (SELECT zipcode, percentile
        FROM ` + (req.params.attribute === "undefined" ? `GoodForKids
` : `${attribute} `) +
        `WHERE percentile >= .7),
        budget AS
        (SELECT zip, 2021_02
        FROM home_values
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget}),
        c_scores AS
            (SELECT a1.zipcode as zip, (${gfk} * a1.percentile +
${gfd} * a2.percentile + ${wa} * a3.percentile + ${da} *
a4.percentile + ${gfb} * a5.percentile + ${rd} * a6.percentile)*100
AS comScore
            FROM GoodForKids a1
            JOIN GoodForDancing a2 ON a1.zipcode = a2.zipcode
            JOIN WheelchairAccessible a3 ON a1.zipcode = a3.zipcode
            JOIN DogsAllowed a4 ON a1.zipcode = a4.zipcode
            JOIN GoodForBikers a5 ON a1.zipcode = a5.zipcode
            JOIN RestaurantDelivery a6 ON a1.zipcode = a6.zipcode)
        SELECT z.zip, z.city, z.state, z.county, 2021_02 AS
MedianHomeValue, AVG(stars) AS avgBusinessRating, cs.comScore AS
'Compatibility Score'
        FROM zipcode z
        JOIN c_scores cs ON z.zip = cs.zip
        JOIN coordinates c ON c.zip = z.zip
        LEFT OUTER JOIN attribute a ON z.zip= a.zipcode
        LEFT OUTER JOIN budget h ON z.zip = h.zip
        LEFT OUTER JOIN business b ON z.zip=b.postal_code
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget} `
```

```
            + (req.params.attribute === "undefined" ? `` : ` AND
percentile >= .7`)
            + `GROUP BY z.zip
        ORDER BY comScore DESC
        LIMIT 50;
```

*Find zipcodes based on Restaurant Information*- Find zipcodes using both standard filters as well as the average restaurant price, average restaurant rating, and number of restaurants listed.

```
WITH coordinates AS
        (SELECT zip,
        69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians( latitude ) ) *
        sin( radians(${latitude}) ) ) )  as distance
        FROM zipcode
        WHERE 69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians(latitude) ) *
        sin( radians(${latitude}) ) ) ) <= ${radius}),
        attribute AS
        (SELECT zipcode, percentile
        FROM ` + (req.params.attribute === "undefined" ? `GoodForKids
` : `${attribute} `) +
        `WHERE percentile >= .7),
        budget AS
        (SELECT zip, 2021_02
        FROM home_values
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget}),
        business_count AS
        (SELECT postal_code, COUNT(business_id) AS bcount
        FROM business
        GROUP BY postal_code)
        SELECT z.zip, z.city, z.state, z.county, AVG(stars) AS
avgBusinessRating, AVG(attributesRestaurantsPriceRange) AS 'Average
Price ($ - $$$$)', COUNT(attributesRestaurantsPriceRange) AS '#
Restaurants Listed', 2021_02 AS MedianHomeValue
        FROM zipcode z
        JOIN coordinates c ON c.zip = z.zip
        LEFT OUTER JOIN attribute a ON z.zip= a.zipcode
        LEFT OUTER JOIN budget h ON z.zip = h.zip
        JOIN business_count bc ON z.zip=bc.postal_code
        JOIN business b ON z.zip=b.postal_code
        WHERE 2021_02 BETWEEN ${minBudget} AND ${maxBudget}`
                + (req.params.attribute_tbl === "undefined" ? `` : `
AND percentile >= .7 `)
                + `GROUP BY z.zip
        HAVING AVG(stars) >= ${rating} AND
AVG(attributesRestaurantsPriceRange) <= ${r_price} AND
COUNT(attributesRestaurantsPriceRange) >= ${r_count}
        ORDER BY ${order_key} ${order_direction}
        LIMIT 50;
```

*Get One-of-a-Kind Businesses in area*- Find businesses in the specified geographic area that don't exist anywhere else in our dataset

```
WITH coordinates AS
        (SELECT business_id AS id,
        69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians( latitude ) ) *
        sin( radians(${latitude}) ) ) )  as distance
        FROM business
        WHERE 69 * DEGREES(acos(
        cos( radians(latitude) ) *
        cos( radians(${latitude}) ) *
        cos( radians(${longitude}) - radians(longitude ) ) +
        sin( radians(latitude) ) *
        sin( radians(${latitude}) ) ) ) <= ${radius})
    SELECT name, city, state, postal_code AS zip, stars AS 'rating'
    FROM business b
    JOIN coordinates c ON b.business_id = c.id
    WHERE NOT EXISTS (SELECT * FROM business b2 WHERE b.name=b2.name
AND b.business_id <> b2.business_id)
    ORDER BY distance ASC, b.name ASC
    limit 50;
```

*Get top 15 local one-of-a-kind restaurants*- returns the top 15 food related businesses by rating if the business is only located in the inputZip and doesn't exist anywhere else in our dataset.

```
   WITH restaurants AS
            (SELECT b.business_id, postal_code, name, stars
            FROM business b JOIN business_categories bc ON
b.business_id = bc.business_id
            WHERE categories = 'Restaurants'),
        topTenLocalRestaurants AS
            (SELECT DISTINCT r1.business_id, name, stars
            FROM restaurants r1
            WHERE postal_code = ${inputZip} AND NOT EXISTS
                (SELECT *
                FROM restaurants r2
                WHERE r2.name = r1.name AND r2.postal_code !=
${inputZip})
            ORDER BY r1.stars DESC
            LIMIT 10)
    SELECT DISTINCT name, stars
    FROM topTenLocalRestaurants t JOIN business_categories bc ON
t.business_id = bc.business_id
    WHERE categories IN ('Restaurants', 'Food')
    ORDER BY stars DESC
    LIMIT 15;
```

## Performance Evaluation

We optimized performance using three main methods: indexing, frontloading intermediate results, and reordering queries/pushing selections and projections.

Though our business dataset was relatively large (124,891 rows x 33 columns after preprocessing), most of our joins involved much smaller zipcode-based tables (which only spanned around 30,000-40,000 rows). As such, joins were actually a fairly negligible source of lag for most of our queries. So, while we pushed selections and projections where possible and reordered our joins to reduce the size of intermediate tables (in an attempt to ease the burden on the query optimizer), these methods were typically not the source of significant performance gains.

Instead, our largest performance bottlenecks came from the queries themselves- particularly queries that required a great number of equality/range searches (for instance, those that used EXIST or PERCENT_RANK). For these, indexing proved far more fruitful. Indexing worked well for these queries because, by providing a more efficient structure to search the data, it allowed us to make a large quantity of searches with fewer page I/Os. Of course, indexes cost both space and make updating more costly (as now the index needs to be updated as well, and in certain cases the data needs to be sorted). While we were not actually updating any data for our app, we tried to act as if we were in a "production" setting with new business data constantly streaming in, and so worked to only index when particularly useful.

Due to the modular nature of our main queries, many needed to pull from the same data. While some common predicates, like distance from coordinates, were both too inexpensive to compute and too variable in the inputs to be solid candidates for caching, the query used to fetch percentile rankings of business attribute ratios was a perfect fit. Even with indexing, creating the percentiles still took near a half a second, so calculating it for every query would add a great deal of overall lag to our application. Originally, we looked to cache the results of part of the query, so only the first query would take the extra time. However, we realized that by expanding the inputted zipcodes to span the entire dataset, we could eliminate any variability in the query and thus make it a perfect fit for materialization. Since MySQL does not support materialized views, we instead used javascript to front-load the intermediate results as a table at the start of each session, and drop the results at the end of the session (though we were not updating our data and so could have just made permanent tables, we decided to act as if we would have to expect updates). In doing so, we shaved .5-1 seconds off almost every single one of our search queries.

| **Query** | **Biggest Bottlenecks** | **Pre-** | **Post-** | **Most Useful** |
|---|---|---|---|---|

|  |  | **optimization Time** | **optimization Time** | **Optimization Methods** |
|---|---|---|---|---|
| Percentile-Rank Attributes | Attribute has to undergo multiple comparisons with rest of data | ~1.062 x 6 | ~.406s x 6 | Indexing on the attribute (makes comparisons more efficient). |
| Find zipcodes | Percentile-ranking the attributes data | .578s | .015s | Front-load Intermediate Results (the core joins take little time due to small zipcode table size) |
| Find businesses | Percentile-ranking the attributes data; business id and business categories undergo many equality searches due to EXISTS clause | 1.485s | .250s | Front-load Intermediate Results; indexing on business categories (making equality searches more efficient) |
| Compatibility Score | Percentile-ranking the attributes data x 6 | 3.45s | .032s | Front-load Intermediate Results (the core joins take little time due to small zipcode table sizes); |
| Food-based Search | Percentile-ranking the attributes data; | 1.219s | .046s | Front-load Intermediate Results (the core joins take little time due to small zipcode table sizes); |
| One-of-a-kind businesses | Business name undergoes many equality searches due to NOT EXISTS clause | 2.172s | .015s | Indexing on business name (makes equality search more efficient) |

## Technical Challenges

- MySQL limitations- As we progressed through our project, we realized that MySQL lacks several useful functions, the most important of which being the ability to materialize views. To compensate for this in a way that could theoretically still handle updates in the data, we dynamically create and drop tables through javascript at the start and end of each session.
- Nested JSON data- The Yelp business dataset was in json format which made preprocessing significantly harder. However we were still able to parse the values from each item and also extract all the attributes in the nested arrays.
- Increased query modularity → Increased query complexity (and flexibility-optimization tradeoffs)- The desire to allow a single query to modularly pull from a varying number of different filters made our queries quite unwieldy. At points, we strategically sacrificed negligible optimization gains to help make our code more manageable and thus easier to scale and retool (such as by using left joins when the left table was small).
- UI (how to fit massive amounts of filters into a non-overwhelming UI?)- We wanted many filters that the user could select, but it made the UI a bit overwhelming. We used drop down menus to reduce the space utilized for filters. In terms of the appearance, we used grids in the styling of the page which definitely helped the UI look much more organized.
- Version Control- Multiple people were working on the same page (the search page) which made it extremely hard to merge our changes using Git. We ended up manually solving it by going through our code and changing the lines individually. However, we definitely learned a lot about how to use Git through these merge conflicts.