**HomePros**

Find your dream city


Team Members

- Aditya Garg (adigarg@seas.upenn.edu / @AdiG1123)
- Jinyeong Park (jyp@seas.upenn.edu / @jinyeong-park)
- Tasnia Nowrin (nowrint@seas.upenn.edu /@tasnianowrin)
- Yang Fu (yangfu@seas.upenn.edu / @yangflorafu)


# 1. Intro

Our website concept, HomePros, is your go-to platform for making the best decisions when it comes to finding your ideal place to live. We tackle the challenges of information overload and uninformed choices by harnessing housing data, economic factors, and social indicators.

At HomePros, we use housing data, economic factors, and social indicators to help people find the best places to live. By analyzing trends in housing prices, tax burden, and safety, we provide valuable insights for informed decision-making about where to call home.

## Our Key Features/Functionalities:

- Data-Driven Insights: We analyze housing prices, safety records, tax info and weather to provide you with valuable insights.
- Personalized Recommendations: Tailored to your preferences, our platform offers recommendations that align with your unique needs.
- Simplified Decision-Making: Say goodbye to the complexity of choice. We simplify the process, helping you make decisions with confidence.

## List of features to implement in the application

- Show recommendations of a list of cities to live in based on editorial suggestions (based on pre-set filters)
- Search by a city name, state name, and filter price ranges and crime scores, then get recommended cities to live in matching the search criteria.

- Show city information by each specific city with avg. price to buy and rent a home, and relevant crime, weather and tax information, also visualize historical information with housing price trends.
- Show states and what cities are in the states.

### List of Extra features

- Integrated live hourly weather information for each city though an external API call to https://api.weather.gov/points/${latitude},${longitude} and polls every 20 minutes for updates
- Pulling images for each city from an external API where `imgSource` is a combination of city and state https://source.unsplash.com/random/1280x720/?${imgSource},USA

# 2. Architecture

### List of Technologies Used:

- Data Storage: AWS RDS (MySQL)
- Backend: Node.js
- Frontend: JavaScript and ReactJS

### System Architecture:

The architecture adopts a microservices approach, leveraging components such as AWS RDS, Node.js backend, and ReactJS frontend. Interaction between microservices is orchestrated to ensure seamless user experiences.

- Data Storage and Management: MySQL on AWS RDS is employed for structured data storage. The database schema is meticulously designed to efficiently manage real estate data, ensuring data integrity. Robust data management is ensured through the implementation of data migration and backup strategies.

- Data Flow: User requests are processed by the Node.js backend, which communicates with the AWS RDS MySQL database for data retrieval and storage. The processed data is then sent to the frontend, developed in React.js, to be dynamically displayed to the user using charts, graphs and maps.

- Backend Architecture: The Node.js backend is structured around a RESTful API design, facilitating easy data interchange and integration with the frontend and database.

- Frontend Architecture: The ReactJS is employed to craft a modular and interactive frontend and provide a seamless user experience with minimal reloads and dynamic content updates. The frontend communicates seamlessly with the backend through well-defined APIs. User interface components are designed for a user-friendly experience, providing an intuitive interface for users.

- Database Integration: AWS RDS (MySQL) is integrated to provide a robust and scalable storage solution, ensuring data integrity, security, and quick access.

- Scalability and Performance: Our architecture leverages the scalability features of AWS and the efficient processing capabilities of Node.js, ensuring the system remains responsive and scalable as user demand grows. We have optimized our database queries for faster retrievals and utilized pagination features to optimize performance.

# 3. Data

## 1. US Cities

### a. Description

A complete list of all US Cities including military locations.

### b. Link: [https://simplemaps.com/data/us-cities](https://simplemaps.com/data/us-cities)

### c. Size Statistics

Size: ~5 MB
Number Rows: 30844
Number Attributes: 17

### d. Summary Statistics

| Attributes | city | state_id | county_name | military |
|---|---|---|---|---|
| count | 30844 | 30844 | 30844 | 30844 |
| unique | 20721 | 52 | 1906 | 2 |
| freq | 29 | 1837 | 400 | 30748 |

| Attributes | population |
|---|---|

| | |
|---|---|
| count | 30844 |
| mean | 13017.46 |
| std | 179089.8 |
| min | 0 |
| 25% | 284 |
| 50% | 977 |
| 75% | 3835 |
| max | 18972871 |

### e. Data Usage

The US city data is what we use to identify city and link them to the corresponding tax, crime, house sales, and rent data. The longitude and latitude are used to get the weather from the weather api. Additionally, we also use the population from this data set to display in the city info page.

## 2. Redfin Housing

### a. Description

The Redfin dataset provides information about various housing statistics by city over a period from 2012-2023.

### b. Link

https://www.redfin.com/news/data-center/

### c. Size Statistics

Size: 2.82 GB
Number Rows: 4,951,858
Number Attributes: 58

### d. Summary Statistics

Years range from 2012 - 2023

| Attribute | median_sale_price | median_list_price | median_ppsf | median_list_ppsf | homes_sold | pending_sales | new_listings | inventory | price_drops | off_market_in_two_weeks |
|---|---|---|---|---|---|---|---|---|---|---|
| count | 4946090 | 4176015 | 4884692 | 4157222 | 4946485 | 3889796 | 4155614 | 4561782 | 2679010 | 3988437 |
| mean | 306076.5 | 343153.1 | 188.284 | 189.1665 | 15.65936 | 15.55387 | 21.4832 | 52.08728322 | 0.266376 | 0.285986131 |
| std | 430661.5 | 1462608 | 6772.561 | 4182.29 | 59.26392 | 57.03739 | 79.62284 | 222.5792603 | 0.182032 | 0.309523357 |
| min | 1 | 290 | 0.00025 | 0.003992 | 1 | 1 | 1 | 1 | 9.92E-05 | 0 |
| 25% | 135000 | 159000 | 86.48987 | 97.65396 | 1 | 2 | 2 | 5 | 0.146341 | 0 |
| 50% | 219000 | 249900 | 128.0648 | 139.7133 | 4 | 4 | 6 | 15 | 0.225 | 0.210526316 |
| 75% | 357500 | 399000 | 196.0885 | 210.3718 | 12 | 12 | 17 | 42 | 0.333333 | 0.5 |
| max | 3.29E+08 | 1E+09 | 12033000 | 5500000 | 4662 | 3794 | 7167 | 25563 | 1 | 2 |

| Attribute | region | city | state |
|---|---|---|---|
| count | 4951858 | 4951858 | 4951858 |
| unique | 22341 | 15816 | 50 |
| freq | 1294 | 6980 | 400618 |

### e. Data Usage

We utilized this data to order cities, calculate index score, and plot information about city home sales information.

## 3. Crime

### a. Description

Crime data from the FBI for select cities including violent and property crimes broken down by specific crimes.

**b.** Link

https://cde.ucr.cjis.gov/LATEST/webapp/#/pages/home

**c.** Size Statistics

Size: 9.9 MB
Number Rows: 74040
Number Attributes: 14

**d.** Summary Statistics

8619 unique cities

| Attribute | State | City |
|---|---|---|
| count | 74040 | 74040 |
| unique | 60 | 8619 |
| freq | 6694 | 133 |

| | Population | Violent Crime | Murder | Rape (revised) | Rape (legacy) | Robbery | Aggravated Assault | Property Crime | Burglary | Larceny-theft | Motor vehicle theft | Arson |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 74023 | 73332 | 74035 | 65144 | 17814 | 74030 | 73984 | 73887 | 73956 | 73985 | 73723 | 62141 |
| mean | 21866.43 | 98.1999 | 1.236051 | 9.117156 | 17.85074 | 32.95033 | 128.9639 | 555.741 | 157.3779 | 399.3772 | 51.62076 | 3.287427 |
| std | 121420.3 | 841.4211 | 11.41814 | 55.75569 | 216.3732 | 338.0542 | 1255.535 | 3077.825 | 1018.943 | 2195.965 | 391.7534 | 24.07737 |
| min | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25% | 2293 | 2 | 0 | 0 | 0 | 0 | 2 | 22 | 6 | 13 | 0 | 0 |
| 50% | 5914 | 10 | 0 | 1 | 1 | 1 | 8 | 81 | 21 | 58 | 3 | 0 |
| 75% | 16557 | 35 | 0 | 5 | 5 | 8 | 36 | 316 | 76 | 242 | 14 | 1 |
| max | 86163 | 5299 | 765 | 2814 | 1399 | 2977 | 124815 | 1427 | 1079 | 11793 | 1859 | 1672 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 33 | 3 | | | 5 | 1 | | 60 | 76 | 1 | 1 | |

### e. Data Usage

We used the crime data to calculate the total crimes for a single city for each year. This was used in sorting, calculating our index score, and displaying information in graph form.

## 4. Tax

### a. Description

Tax rates by state for state local tax burden for the year 2023

### b. Link

https://wisevoter.com/state-rankings/property-taxes-by-state/

### c. Size Statistics

Size: 9.9 MB
Number Rows: 50
Number Attributes: 6

### d. Summary Statistics

50 unique states

| Attribute | State | PropertyTaxRate | IncomeTaxRate | CorporateTaxRate | StateLocalTaxBurden | SalesTaxRate |
|---|---|---|---|---|---|---|
| Count | 50 | 50 | 50 | 50 | 50 | 50 |
| Unique | 50 | NA | NA | NA | NA | NA |
| Mean | NA | 0.009994 | 0.025436 | 0.0595 | 0.10556 | 0.050884 |
| Std | NA | 0.004571 | 0.018867 | 0.028235 | 0.020503 | 0.019803 |
| Min | NA | 0.0031 | 0 | 0 | 0.046 | 0 |
| Max | NA | 0.0213 | 0.07 | 0.115 | 0.159 | 0.0725 |

### e. Data Usage

We use the tax burden for both city and state ordering, calculating the index score for both city and state, and displaying a cities index score as part of city info. State info also contains all cities and the index score for each city.

# 4. Database

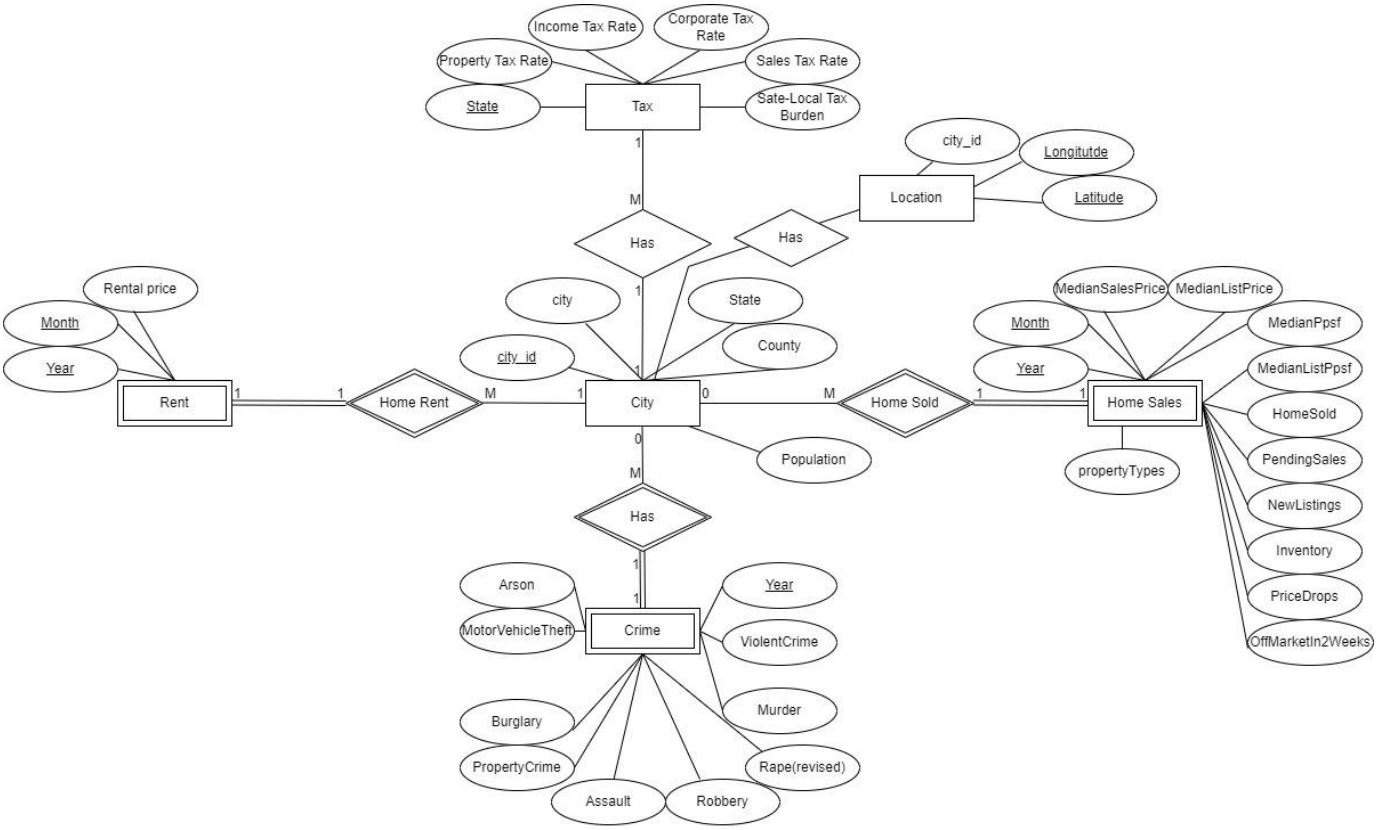
To clean the data we used Python with libraries Pandas and Numpy. First we loaded the csv data into a dataframe and merge any dataframes in which there are multiple csv files (i.e. Crime dataset).

For all data we will drop any null values where our expected primary key value is missing because we are only focused on presenting available data we do not need to do any imputation. Any date columns will be converted to the datetime type then split into the year and month columns. All states will be represented using the full state name and city names will remain the same and cleaned of any non alpha characters. We will also perform entity resolution since the state and city are critical keys for our application. Any numerical columns will be first cleaned by removing non digit characters except '.' and converted to floats.

The rental data is differently formatted since the year's are the column names so we melted the dataframe so that the years become a single column with the respective rent data becoming rows. The remaining data types will be formatted as previously mentioned.

All tax data will be projected into a dataframe where the year equals 2023 since we do not need historical data.

ER diagram




Number of Instances in Each Table

| Table Name | Number of Instances |
|---|---|
| City | 39160 |
| Rent | 123639 |
| Crime | 68537 |
| Home Sales | 2040412 |
| Location | 39160 |
| Tax | 50 |

## Normal Form

**City:**

FD: {city_id} → {city, state, county, population}, {city, state, county} → {city_id, population}

CK: city_id, {city, state, county}

Closures: $city\_id^+$ = {city_id, city, state, county, population}, $\{city, state, county\}^+$ = {city, state, county, city_id, population}

Normal Form: Since all functional dependencies have a determinant that is a super key the relation is in BCNF.

**Location:**

FD: {Longitude, latitude} → {city_id}, {city_id} → {Latitude, Longitude}

CK: city_id, {latitude, longitude}

Closures: $city\_id^+$ = {city_id, latitude, longitude}, $\{latitude, longitude\}^+$ = {city_id, latitude, longitude}

Normal Form: Since all functional dependencies have a determinant that is a super key the relation is in BCNF.

**Home Sales:**

FD: {City_id, Year, Month} → {Propertytype, mediansalesprice, medianlistprice, medianppsf, medianlistppsf, homesold, pendingsales, newlistings, inventory, pricedrops, offmarketin2weeks, propertyType}

CK: {City_id, Year, Month}

Normal Form: Since there is only a single functional dependency which has a determinant that is a super key the relation is in BCNF.

**Crime:**

FD: {City_id, year} -> {violentCrime, Murder, Rape(revised), Robbery, Assault, PropertyCrime, Burglary, LarcentyTheft, MotorVehicleTheft, Arson}

CK: {City_id, year}

Normal Form: Since there is only a single functional dependency which has a determinant that is a super key the relation is in BCNF.

**Rent:**

FD: {City_id, Year, Month} -> {rentalPrice}

CK: {City_id, Year, Month}

Normal Form: Since there is only a single functional dependency which has a determinant that is a super key the relation is in BCNF.

**Tax:**

FD: {State} -> {PropertyTaxRate, IncomeTaxRate, CorporateTaxRate, SalesTaxRate, StateLocalTaxBurden}

CK: {State}

Normal Form: Since there is only a single functional dependency which has a determinant that is a super key the relation is in BCNF.

**Conclusion:** All relations are in BCNF.

# 5. Web App description

### List of Pages Description

1. Page 1 (Homepage): This is the main page where users can search for city and/or state. This page has two search boxes. The user can enter a city and the state or just a state and hit search. Upon searching the user will be redirected to the state or cities information page.
2. Page 2 (StateInfoPage): This page shows a detailed overview of a state including some key statistics in a table on the left of the page and an interactive map showing the index scores of the cities in that state.
3. Page 3 (CityPage): This page shows a detailed overview of a city including some key statistics in a table on the left of the page and an interactive map showing the location of the city.
4. Page 4 (Cities): The page can be accessed through the navigation bar and shows all the cities with an option for advanced search criteria. The advanced search includes the fields tax burden, average home price, average rent price, total crime, and population. You can also sort results by each of the factors in ascending order. Each card also has the city's current weather and can be clicked to lead to the city specific information page.
5. Page 5 (States): The page can be accessed through the navigation bar and shows all the states linking each state to the StateInfoPage. The state cards can be sorted by tax burden, alphabetically, index score, or total crime in ascending order.

# 6. API Specification

*Optional parameters are marked with an asterisk (*)
**Route 1**
Description: Returns top cities to live in from editorial calculated index score.
Request Path: /top_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string,“index_score”: float},....]**

---

**Route 2**

Description: Returns cities ranked by 2022 average home sales price, from cheapest to most expensive.
Request Path: /salesrank_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string, “avg_sales_price”: float},....]**

---

**Route 3**

Description: Returns safest cities to live in from lowest to highest crime rates.
Request Path: /safest_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string, “crime_rate”: float},....]**

---

**Route 4**

Description: Returns cities ranked by 2022 average rental price, from cheapest to most expensive.
Request Path: /rentrank_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string, “avg_rental_price”: float},....]**

---

**Route 5**

Description: Returns cities ranked by state local tax burden, from lowest tax to highest.
Request Path: /taxrank_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string, “tax_burden”: float},....]**

---

**Route 6**

Description: Returns cities ranked by name alphabetically, from A to Z.
Request Path: /namerank_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string},....]**

---

**Route 7**

Description: Returns the most popular housing city market by how many houses were sold in the last year, from highest to lowest.
Request Path: /homesold_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*, city (string)*
Response Type: JSON array
**Return Parameters: [{“city” : string, “county” : string, “state”: string, “homes_sold”: int},....]**

---

**Route 8**

Description: Returns top states ranking by our calculated index score.
Request Path: /top_states
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10)
Response Type: JSON array
**Return Parameters: [{“state”: string, “tax_burden”: float, “total_crimes”: int, “index_score”: float} ,....]**

---

**Route 9**

Description: Returns top states ranking by state local state burden, from lowest to highest.
Request Path: /taxrank_states
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*
Response Type: JSON array
**Return Parameters: [{“state”: string, “tax_burden”: float},....]**

---

**Route 10**

Description: Returns top states ranking by total crimes, from lowest to highest.
Request Path: /safest_states
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*
Response Type: JSON array
**Return Parameters:[{“state”: string, “total_crimes”: int},....]**

---

**Route 11**

Description: Returns states ranked by name alphabetically, from A to Z.
Request Path: /namerank_states
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*

Response Type: JSON array
**Return Parameters: [{“state”: string},....]**

---

**Route 12**
Description: Returns information on the most popular housing state market by how many houses were sold in the last year, from highest to lowest.
Request Path: /homesold_states
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), state (string)*
Response Type: JSON array
**Return Parameters: [{“state”: string, “state_homes_sold”: int},....]**

---

**Route 13**
Description: Returns basic city information for a specific city, state inputted by a user.
Request Path: /city
Method: GET
Request Parameter(s): None
Query Parameter(s): city(string), state(string)
Response Type: JSON array
**Return Parameters: [{“city_id”: int, “city”: string, “county”: string, “state”: string, “population”: int, “latitude”: float, “longitude”: float},....]**

---

**Route 14**
Description: Returns housing sales prices trends by month from a specific city from all available data.
Request Path: /monthly_house_prices
Method: GET
Request Parameter(s): None
Query Parameter(s): city(string), state(string)
Response Type: JSON array
**Return Parameters: [{“city”: string, “county”: string, “state”: string, “month”: int, “year”: int, “median_sales_price”: float, “median_list_price”: float, “property_type”: string},....]**

---

**Route 15**
Description: Returns rent prices trends by month from a specific city from all available data.
Request Path: /monthly_rent_prices
Method: GET
Request Parameter(s): None
Query Parameter(s): city(string), state(string)
Response Type: JSON array
**Return Parameters: [{“city”: string, “county”: string, “state”: string, “month”: int, “year”: int, “rental_price”: float},....]**

---

**Route 16**
Description: Returns total crime numbers by month from a specific city from all available data.

Request Path: /yearly_crime
Method: GET
Request Parameter(s): None
Query Parameter(s): city(string), state(string)
Response Type: JSON array
**Return Parameters: [{"city": string, "county": string, "state": string, "year": int,"total_crimes": int },....]**

---

**Route 17**
Description: Returns basic tax information for a specific state inputted by a user.
Request Path: /state
Method: GET
Request Parameter(s): None
Query Parameter(s): state(string)
Response Type: JSON array
**Return Parameters: [{"state": string, "property_tax_rate": float, "income_tax_rate": float, "corporate_tax_rate": float, "state_local_tax_burden": float, "sales_tax_rate": float,},....]**

---

**Route 18**
Description: Returns top 10 recommended cities to live in based on user selected criteria in search query.
Request Path: /search_cities
Method: GET
Request Parameter(s): None
Query Parameter(s): page (int)* (default:1), pageSize (int)* (default:10), city(string)*, state(string)*, population_low(int)* (default: 0), population_high(int)* (default: 10,000,000) , total_crimes_low(int)* (default: 0), total_crimes_high(int)* (default: 100,000), avg_sales_price_low(float)* (default: 0.0), avg_sales_price_high(float)* (default: 10,000,000.0), avg_rental_price_low(float)* (default: 0.0), avg_rental_price_high(float)* (default: 100,000.0), tax_burden_low(float)* (default: 0), tax_burden_high(float)* (default: 1), order(string)*
Response Type: JSON array
**Return Parameters: [{"city" : string, "county" : string, "state": string, "population": int, "total_crimes": int, "avg_sales_price": float, "avg_rental_price": float, "tax_burden": float, "latitude": float, "longitude": float},....]**

# 7. Queries

We have 18 queries in total. To keep the report more compact, we listed 5 queries here and please check the full 18 queries in the code.

Query 18 is the most complex query and we use it in the web app "Advanced Search" page to process user search criterias. For example, users can input housing sales price, rental price, crime rate, tax between a low and high range to get customized recommendations for which city to live in.

Query 3 and Query8 are also complex queries and we use them in the web app "City" and "State" pages to calculate the crime rate for each city, and index score for each state, and give user recommendations based on crime rate and index score.

**Query 1**: /top_cities

Return top cities to live in from editorial calculated index score. Editorial selection criteria is 20% weight on total crime, 30% on average home sales price, 30% on average rental price, 15% on tax burden and 5% on population. Default to show city rankings across all states, and can input state/city to limit only ranking in selected state/city.

The complex calculation formula in the query used each relative attribute average, standard deviation, and the assigned weight together to calculate standardized z-score, and using the z-score with weight to get a combined index score.

Before optimization complex query (we simplified this query and listed it in the query optimizations)

```
WITH Crime2019 AS
(SELECT city_id, (violent_crime + property_crime) AS total_crimes
FROM Crime
WHERE year = 2019
AND violent_crime >= 0 AND property_crime >= 0)
, HomeSales2022 AS
(SELECT city_id, AVG(median_sale_price) AS avg_sales_price
FROM HomeSales
WHERE year = 2022
GROUP BY city_id
HAVING avg_sales_price >= 0)
, Rent2022 AS
(SELECT city_id, AVG(rental_price) AS avg_rental_price
FROM Rent
WHERE year = 2022
GROUP BY city_id
HAVING avg_rental_price >= 0)
, TaxBurden AS
(SELECT state, state_local_tax_burden AS tax_burden
FROM Tax
WHERE state_local_tax_burden >= 0)
, CityPop AS
(SELECT city_id, city, county, state, population
FROM City
WHERE population >= 0
AND state LIKE '%'
AND city LIKE '%')
SELECT c.city, c.county, c.state, c.population, cri.total_crimes,
h.avg_sales_price, r.avg_rental_price, t.tax_burden,
((0.2 * ((cri.total_crimes - AVG(cri.total_crimes) OVER()) /
STDDEV(cri.total_crimes) OVER()))+
(0.3 * ((h.avg_sales_price - AVG(h.avg_sales_price) OVER()) /
```

```
STDDEV(h.avg_sales_price) OVER())) +
(0.3 * ((r.avg_rental_price - AVG(r.avg_rental_price) OVER()) / STDDEV(
r.avg_rental_price) OVER())) +
(0.15 * ((t.tax_burden - AVG(t.tax_burden) OVER()) / STDDEV(t.tax_burden)
OVER())) +
(0.05 * ((c.population - AVG(c.population) OVER()) / STDDEV(c.population)
OVER())))
AS index_score
FROM TaxBurden t
JOIN CityPop c
ON t.state = c.state
JOIN Crime2019 cri
ON c.city_id = cri.city_id
JOIN HomeSales2022 h
ON c.city_id = h.city_id
JOIN Rent2022 r
ON c.city_id = r.city_id
```

After optimization query

```
WITH CityState AS
(SELECT city_id, city, county, state
FROM City
WHERE state LIKE '%'
AND city LIKE '%')
SELECT c.city, c.county, c.state, i.index_score
FROM CityState c
JOIN CityIndexScore i
ON c.city_id = i.city_id
ORDER BY index_score
LIMIT 10
```

**Query 2**: /salesrank_cities

Returns cities ranked by 2022 average home sales price, from cheapest to most expensive. Default to show city rankings across all states, and can input state/city to limit only ranking in one state/city.

```
WITH HomeSales2022 AS
(SELECT city_id, avg_sales_price
FROM HomeSummary2022
WHERE avg_sales_price >= 0)
, CityState AS
(SELECT city_id, city, county, state
FROM City
WHERE state LIKE '%'
```

```
AND city LIKE '%')
SELECT c.city, c.county, c.state, h.avg_sales_price
FROM CityState c
JOIN HomeSales2022 h
ON c.city_id = h.city_id
ORDER BY h.avg_sales_price
LIMIT 10
```

**Query 3**: /safest_cities
Returns safest cities to live in from lowest crime to highest crime rates. Default to show city rankings across all states, and can input state/city to limit only ranking in one state/city.

```
WITH Crime2019 AS
(SELECT city_id, (violent_crime + property_crime) AS total_crimes
FROM Crime
WHERE year = 2019 AND violent_crime >= 0 AND property_crime >= 0)
, CityState AS
(SELECT city_id, city, county, state, population
FROM City
WHERE state LIKE '%'
AND city LIKE '%'
AND population >= 1)
SELECT c.city, c.county, c.state, cri.total_crimes / c.population *
100000 AS crime_rate
FROM CityState c
JOIN Crime2019 cri
ON c.city_id = cri.city_id
ORDER BY crime_rate
LIMIT 10;
```

**Query 8**: /top_states
Returns top states ranking by our calculated index score, based on crime and tax burden combined.

```
WITH Crime2019 AS
(SELECT city_id, (violent_crime + property_crime) AS total_crimes
FROM Crime
WHERE year = 2019
AND violent_crime >= 0 AND property_crime >= 0)
, TaxBurden AS
(SELECT state, state_local_tax_burden AS tax_burden
FROM Tax
WHERE state_local_tax_burden >= 0)
, CityState AS
```

```
(SELECT city_id, state
FROM City
WHERE state LIKE '%')
SELECT c.state, t.tax_burden, SUM(cri.total_crimes) AS total_crimes,
       (0.5 * ((SUM(cri.total_crimes) - AVG(SUM(cri.total_crimes)) OVER
()) / STDDEV(SUM(cri.total_crimes)) OVER ())) +
       (0.5 * ((SUM(t.tax_burden) - AVG(SUM(t.tax_burden)) OVER ()) /
               STDDEV(SUM(t.tax_burden)) OVER ())) AS index_score
FROM TaxBurden t
JOIN CityState c
ON t.state = c.state
JOIN Crime2019 cri
ON c.city_id = cri.city_id
GROUP BY t.state, t.tax_burden
ORDER BY index_score
LIMIT 10
```

* This part, (0.5 * ((SUM(cri.total_crimes) - AVG(SUM(cri.total_crimes)) OVER()) / STDDEV( SUM(cri.total_crimes)) OVER())), calculates the standardized score (z-score) for the total_crime by taking the total crime of the current state subtracting the average total crime over all states and then dividing by the standard deviation over all states. We weighed the result to be 0.5 of the index score

**Query 18**: /search_cities

Returns top recommended cities to live in based on user selected criteria. (if users select housing price between $500k-$1M, crime numbers below 10, tax burden and etc.), then show the cities within the user input range. The below ${} and city state names are all from user input in the web page.

```
WITH Crime2019 AS
    (SELECT city_id, (violent_crime + property_crime) AS total_crimes
    FROM Crime
    WHERE year = 2019
    AND violent_crime >= 0 AND property_crime >= 0
    AND (violent_crime + property_crime) BETWEEN ${total_crimes_low} AND
${total_crimes_high})
    , HomeSales2022 AS
    (SELECT city_id, avg_sales_price
    FROM HomeSummary2022
    WHERE avg_sales_price BETWEEN ${avg_sales_price_low} AND
${avg_sales_price_high})
    , Rent2022 AS
    (SELECT city_id, avg_rental_price
    FROM RentSummary2022
    WHERE avg_rental_price BETWEEN ${avg_rental_price_low} AND
```

```
${avg_rental_price_high})
    , TaxBurden AS
    (SELECT state, state_local_tax_burden AS tax_burden
    FROM Tax
    WHERE state_local_tax_burden BETWEEN ${tax_burden_low} AND
${tax_burden_high})
    , CityPop AS
    (SELECT city_id, city, county, state, population
    FROM City
    WHERE population BETWEEN ${population_low} AND ${population_high}
    AND state LIKE '%'
    AND city LIKE '%')
    SELECT c.city, c.county, c.state, c.population, cri.total_crimes,
h.avg_sales_price, r.avg_rental_price, t.tax_burden
    FROM TaxBurden t
    JOIN CityPop c
    ON t.state = c.state
    JOIN Crime2019 cri
    ON c.city_id = cri.city_id
    JOIN HomeSales2022 h
    ON c.city_id = h.city_id
    JOIN Rent2022 r
    ON c.city_id = r.city_id
    ORDER BY avg_sales_price
    LIMIT 10
```

# 8. Performance evaluation

**Query optimizations summary** (details after the before/after optimization time table):

- 1) Use summary tables (alternative for materialized view in MySQL) to improve query speed.
- 2) Optimize selection and projection as early as possible to include only relevant tuples and attributes in joins.
- 3) Standardize city rankings with index score and store the calculated results in a summary table.
- 4) Add index on (state, city) to improve query speed whenever we search for city/state information.

Running time recorded from Datagrip Explain Plan -> Explain Analysis Actual total time.
* Other queries not listed were added after optimization, and used the new optimization methods from creation.

| | Before optimization | After optimization | Optimization Actions |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Query1 /top_cities | 6,070ms | 0.63ms | All 1) 2) 3) 4) optimizations above used. |
| Query2 /salesrank_cities | 5,861ms | 19ms | 1) 2) 4) optimizations above used. |
| Query3 /safest_cities | 111ms | 94ms | 2) 4) optimizations above used. |
| Query5 /taxrank_cities | 0.7ms | 0.2ms | 4) optimizations above used. |
| Query7 /homesold_cities | 5,532ms | 22ms | 1) 2) 4) optimizations above used. |
| Query8 /top_states | 101ms | 67ms | 2) optimizations above used. |
| Query13 /city | 14ms | 0.03ms | 2) 4) optimizations above used. |
| Query14 /monthly_house_prices | 4ms | 0.54ms | 2) 4) optimizations above used. |
| Query15 /monthly_rent_prices | 5ms | 0.14ms | 2) 4) optimizations above used. |
| Query16 /yearly_crime | 3ms | 0.06ms | 2) 4) optimizations above used. |
| Query18 /search_cities | 6,070ms | 35ms | 1) 2) 4) optimizations above used. |

**Query optimizations details**:

1) Use summary tables (alternative for materialized view in MySQL) to improve query speed
- Our original queries like Q1, Q2, Q7 and Q18 with housing market information were very slow. HomeSales is a big table with 2,040,412 rows and we need to group by city_id and calculate average and sum to show the 2022 yearly price. It cost more than 5,500ms to run the query each time. We repeatedly reaggregate the same data each time, so we decided to use materialized views caching to store the data.
- Because MySQL doesn't support materialized views, we checked common practices online, and added 2 summary tables, one for home sales and one for rent, for the latest 2022 aggregated data. So, when users check the housing prices, it doesn't need to go back to the original data table with the slow group by aggregations from large tables. New queries can use the already aggregated data from summary tables.

New HomeSummary2022 table

```
CREATE TABLE HomeSummary2022 AS
SELECT city_id, AVG(median_sale_price) AS avg_sales_price, SUM(homes_sold) AS
```

```
homes_sold
FROM HomeSales
WHERE year = 2022
GROUP BY city_id
```

New RentSummary2022 table

```
CREATE TABLE RentSummary2022 AS
SELECT city_id, AVG(rental_price) AS avg_rental_price
FROM Rent
WHERE year = 2022
GROUP BY city_id
```

2) Optimize selection and projection as early as possible to include only relevant tuples and attributes in joins. Moved all where filter conditions to the CTEs and only select relevant attributes before join, so the total rows and columns in join reduced.

- Originally in all queries with where conditions, we have the where filter conditions in the main query after join. So all data from each CTEs are joined in the main query, and then later get filtered out.
- Selection (where clause) and projection (select clause) optimization: From the query optimization technique, we learned to push “where” filters and attributes in “select” in appropriate CTEs as early as possible before joining tables. For example, we filter for state and city names in the City table first, and select only relevant columns which will be helpful in join and query result, then only use the matching rows and columns to join with Crime and Housing data. This reduces total rows and columns in join and improves query speed.

3) Standardize city rankings with index score and store the calculated results in summary table

- We want to use an index score to give editorial recommendations. Because the attributes data were with different scales, some in dollars, some in %, etc.., we calculated an index score using 20% weight on total crime, 30% on average home sales price, 30% on average rental price, 15% on tax burden and 5% on population.
- Because the calculation involved 5 tables, to better improve the speed, we put the city_id and its index score in a summary table (alternative for materialized view) so we can query at a faster speed. City index score calculation details in query 1 in section 7 Queries.
- After optimization we only join 2 tables, and show fewer attributes with the pre-calculated score, which makes the query much faster.

4) Add index on (state, city) in the City table to improve query speed whenever we search for city and state information.

- Many of our queries are relevant to searching for specific cities. Before we add the index, the query needs to look through the full City table to find city names and state names that match the search in user input. For example, if we search for San Francisco, California, before adding an index, the query needs to go to every tuple in the relation, go from A-Z in all city names, and from A-Z in all state names to find possible matches.
- With the (state, city) index, the query can use binary search to find states starting with ‘C’, and cities starting with ‘S’ in the state at a faster speed without searching for the full table. So, it only goes through a portion of the tuples in the table, and improves query speed.

# 9. Technical challenges

1) Database Design and Modeling:

- Challenge: Creating an efficient and normalized database schema that meets the project requirements.
- Overcoming the Challenge: Thoroughly analyze the project requirements, normalize the data model, and consider future scalability. Regularly communicate with team members to ensure consensus on the data model.

2) Data Integrity and Validation:

- Challenge: Ensuring data integrity and implementing robust validation mechanisms to prevent incorrect or inconsistent data entry.

- Overcoming the Challenge: Implement constraints, foreign key relationships, and input validation at both the application and database levels. Perform thorough testing to identify and fix potential data integrity issues.

3) Performance Optimization:

- Challenge: Optimizing database performance to handle large datasets and concurrent user access.

- Overcoming the Challenge:
  In addressing this challenge, we applied a multifaceted approach, carefully considering the trade-offs involved:

(1) Database Tuning: Fine-tuning database configurations to align with specific performance requirements. Adjustments were made judiciously, considering the potential implications on overall system stability and resource utilization.

(2) Indexing: Adopting new indexes to enhance query performance. While this can significantly improve retrieval speed, it requires a thoughtful evaluation of potential downsides, such as increased storage requirements and potential impacts on write operations.

(3) Query Optimization: Streamlining and optimizing database queries for efficiency. Each optimization decision was weighed against potential consequences to strike a balance between performance gains and maintaining overall system stability.

(4) Monitoring and Profiling: Regularly monitoring the database and profiling its performance to identify and address bottlenecks promptly. This proactive approach allowed us to detect issues early and implement targeted optimizations.

4) Scalability:

- Challenge: While our team discussed scalability for future growth within the current project scope and time constraints, we didn't implement specific measures.

- Overcoming the Challenge: Consider horizontal and vertical scaling strategies, use load balancing, and optimize resource-intensive queries. Regularly review and adjust the architecture based on usage patterns.

Interesting Findings:

- SQL Optimization Techniques: When applying methods to enhance specific queries, such as adding indexes and employing other techniques, there is typically a tradeoff involved. Therefore, it's crucial to always consider the tradeoffs when aiming to improve particular queries. Additionally, one must assess the potential impact on the current implementation when modifying specific queries and adding indexes.

- While exploring ways to enhance speed, we discovered methods to improve efficiency not only in the database but also on the front end. We considered simple optimizations like image compression for faster loading if we had more time. Although we couldn't address everything due to project scope and time constraints—especially given the course's database-centric nature—we emphasized the database optimization part. Despite this focus, contemplating front-end optimizations provided valuable insights for future development.