Storm Haven: A Disaster-Conscious Housing Recommendation System

Group Members: Phillip Gao, Nikhita Pabbaraju, Nandini Swami, Evan Zheng

Code

https://github.com/Phillip-Gao/StormHaven

Demo Video

https://drive.google.com/file/d/1OogIFJV6P4v5vnopVBcUIhqytnPQReu_/view?usp=sharing

Live Demo Slides

https://docs.google.com/presentation/d/13dg7MINU1MnA4_eKXXWgXOnS8Pc5mWHFCmre9vntFzs/

Link to Data Cleaning (Colab)

https://colab.research.google.com/drive/1s74ByNOYSjXLzUAR8eGaJcjF25XVjmIu?usp=sharing

Extra Credit

The frontend (client) is deployed on Vercel. The backend (server) is deployed on Render. Please note that the Render instance spins down with inactivity, so please give the website up to 2 minutes to load the query requests.

- Frontend Link: https://stormhaven.vercel.app
- Backend Link: https://stormhaven.onrender.com/search_properties

1. Introduction

Homebuyers often face the difficult task of balancing affordability and safety when searching for homes in disaster-prone regions. Recent devastating hurricanes impacting states on the East Coast of the United States underscore the need for tools that integrate housing and disaster data. StormHaven addresses this challenge by providing users with a platform to make well-informed decisions based on a combination of housing prices and disaster risk information.

Functionality:

StormHaven is a housing recommendation system that allows users to search for properties. In addition to traditional housing search features, the system provides insights into disaster risks for each property, by displaying recent disasters and properties affected by these events. StormHaven empowers users to evaluate both affordability and safety when considering housing options. Stormhaven loads on the Dashboard page with a welcome screen. Users can easily find an overview of the properties database, like 20 most disaster affected locations and those affected in the past two years, and different analytics, like safest properties per state. The FindHouses page allows users to search for properties using various filters such as location, price, number of bedrooms, and bathrooms. Users can manage the columns in their results by hiding and sorting columns and users can favorite or unfavorite different property entries to save for later. Clicking the property ID allows users to view the full details of the property. The DisasterRisks page allows users to search by disasters for locations or disasters that have specific public/governmental assistance. Users may also search for disasters within a certain time frame, if a city had some type of assistance, and more. The Favorites page allows users to view property entries the user has favorited in a table with all the associated information. The user may click the property ID to view the full details of the property like on the FindHouses page, remove favorites from the list, and add customized notes to each property.

2. Architecture

Technologies Used:
1. Frontend: React.js, Material-UI (MUI)
2. Backend: Node.js, Express.js
3. Database Management: PostgreSQL for structured data storage and querying
4. Data Processing: Python (Pandas, NumPy) for data cleaning and transformation
5. Version Control: GitHub
6. Deployment: Vercel (frontend) and Render (backend)

System Architecture:
1. Frontend: Provides an interactive user interface to explore properties and disaster data.
2. Backend: Serves APIs for querying property and disaster data and integrates processed datasets.
3. Database: Normalized PostgreSQL schema with indexing and foreign key relationships for efficient querying.

Application Pages:

1. Home Page (Dashboard): A welcome page that introduces users to the platform's functionality and provides quick access to key features.
    a. Overview:
        i. View the 20 Most Disaster-Affected Regions.
        ii. View properties affected by disasters in the past 2 years.
        iii. View properties with no disasters in high risk areas. (Complex Query 2)
    b. Analytics:
        i. View the safest properties per state. (Complex Query 3)
        ii. View frequent disasters for high price properties. (Complex Query 1)
        iii. View most affected properties in their location. (Complex Query 4)
2. FindHouses Page: A search tool that allows users to filter properties by property ID, city, state, sale status, price range, number of bathrooms and/or bedrooms, and acres.
    a. Reset filters button.
    b. Clicking on property ID displays a card that has all the associated information for that property.
    c. Filter columns by ascending or descending, hide columns, manage columns, and properties that contain or do not contain specific attribute(s).
    d. Users can favorite or unfavorite property entries.
3. DisasterRisks Page: A search tool that allows users to search disasters by disaster number, the type of assistance, city, and the time frame of the disaster
    a. Reset filters button.
    b. Filter columns by ascending or descending, hide columns, manage columns, and properties that contain or do not contain specific attribute(s).
    c. Filter columns by ascending or descending, hide columns, manage columns, and properties that contain or do not contain specific attribute(s).
    d. Users can add customized notes to each property entry to note personalizations.
    e. Remove button for each favorited property entry.

Backend (Server):

- routes.js: Establishes the database connection and defines a series of query-based API route handlers for disaster and property-related data. The key functions, organized as route handlers, include complex analytics queries: identifying frequent disaster types in high-priced areas, properties unaffected by disasters in the last five years, and the safest cities based on disaster history. Other queries retrieve disaster trends, recent disasters, disaster-prone areas, and details about properties and their relationship with disasters. The search_properties and search_disasters functions dynamically filter and retrieve data based on user-provided query parameters. The module exports all defined functions, enabling their use in routing within the server.
- index.js: Serves as the main entry point for the application. Initializes the server and maps API endpoints to their respective route handlers. Key functionalities include allowing requests from any origin and registering various routes to serve data for different parts of the application, such as the dashboard, FindHouses, and DisasterRisks pages. Uses routes.js to link specific API endpoints to functions that handle database queries and logic and includes. Routes specific to searching properties and disasters are also included to handle API requests and provide dynamic data to the frontend.

Frontend (Client):

- App.js: Defines the main React application component, setting up the structure and navigation for the app. It uses React Router for client-side routing, enabling navigation between different pages. Each page is linked to a corresponding route, such as /FindHouses or /Favorites. Applies a consistent design theme and defines a primary color palette for the interface. Provides utility functions, addFavorite and removeFavorite, for managing a global list of favorite items.
- PageNavbar.js: A reusable navigation bar that provides users with easy access to different pages in Storm Haven. Uses React's useState and useEffect hooks to dynamically generate navigation links, highlight the currently active page, provide a theme and background color, and logo.
- Dashboard.js: Serves as a central hub for displaying disaster-related insights and analytics for properties. It is structured into two main sections: Overview and Analytics, both of which fetch data dynamically from backend APIs. Using React's useState, the component manages various states to control data visibility (showOverview and showAnalytics), loading indicators, and error handling for API responses. It also maintains specific states for fetched data.
- FindHouses.js: A search interface that allows users to explore property listings based on customizable filters. Users can filter properties by parameters and fetch property data from a backend API for initial loading and dynamically update the results based on user input through the search function. Results are displayed in a table and updates the favorites state to reflect changes in the table. Also includes the PropertyCard component.
- DisasterRisks.js: Provides a comprehensive interface for users to search and analyze disaster data. Includes filters for narrowing down disaster records based on attributes. Fetches disaster records and trends from specified backend endpoints upon mounting and displays the results. The search function dynamically constructs a query string based on the user's filter inputs and updates the disaster data accordingly.
- Favorites.js: Provides a detailed interface for managing a list of favorite properties stored in the browser's localStorage. Asynchronously fetches property details from a backend API, constructing a dataset (data) that combines the fetched details with any notes previously added.
- PropertyCard.js: Provides a detailed modal view of a specific property's information and associated disaster data. Fetches property details and disaster data from two different endpoints when a propertyId is provided and disaster-related metrics are dynamically calculated.

3. Data

Data Set 1: FEMA Disaster Data

Source: FEMA Open Data

Description: This dataset provides detailed information on declared disasters in the United States, including the type of disaster, affected locations, and key dates such as incident start and end dates.

Summary Statistics: 468,709 rows, 14 attributes

Pre-Processing:

- Dropped Unnecessary Columns: Removed the hash and lastRefresh columns as they were irrelevant for analysis.
- Date Formatting: Converted date columns (designatedDate, entryDate, updateDate, closeoutDate) to the correct datetime format using pd.to_datetime.
- Dropped rows with missing or unparseable dates to ensure data consistency.
- Filtering by County/Parish: Retained rows where the placeName column contained the terms "(County)" or "(Parish)" to focus on county-level data.
- Cleaning placeName: Removed the strings "(County)" or "(Parish)" from the placeName column, leaving only the name of the county.

Data Set 2: Real Estate Listings

Source: Kaggle Real Estate Dataset

Description: This dataset provides comprehensive information on real estate listings in the United States, including attributes such as price, zip code, number of bedrooms, and lot size. It is used to analyze housing affordability, calculate property-specific metrics, and link housing data to disaster information.

Summary Statistics: 2,264,973 rows, 12 attributes

Pre-Processing:

- Removed irrelevant columns: Dropped the 'brokered_by' column as the specific agency was not relevant to the analysis.
- Handling missing values: Checked for and dropped rows with missing values (NaN) in 'zip_code' and 'city' columns, as these were crucial for analysis.
- Feature Engineering: Calculated new features:
    - 'price_per_sqft': Price divided by house size.
    - 'price_per_acre': Price divided by acre lot size.
    - 'bed_bath_ratio': Number of bedrooms divided by the number of bathrooms.
    - 'price_per_bedroom': Price divided by the number of bedrooms.
    - 'has_acreage': A boolean column indicating if the property has any lawn space (acre_lot > 0).
- Merging external data:
    - Merged with zip code data (uszips.csv) to add 'county_name' information to the dataset.

Data Set 3: U.S. Zip Codes

Source: SimpleMaps

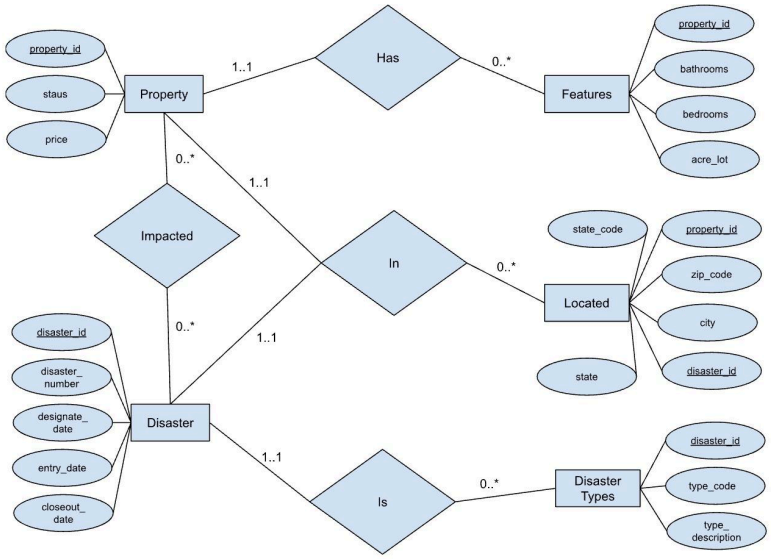
Description: This dataset contains essential information on zip codes across the United States, including state and county associations. It is crucial for linking housing data to disaster information and normalizing location data across datasets.

Summary Statistics: 33,784 rows, 17 attributes

Pre-Processing:

- Retained only the necessary columns: zip, state_id, state_name, county_name

4. Database



Data Ingestion Procedure

1. Data Source Download: Download disaster data from FEMA Open Data, property data from Kaggle Real Estate, and U.S. zip codes from SimpleMaps
2. Data Import into Python: Import CSV files into a Python project on Google Colab (using read_csv()). Ensure that Pandas and NumPy are already installed and imported into our project.
3. Data Reading: For disasters, properties, and zip codes, read data from a CSV file into a Pandas DataFrame. Make sure that we handle any parse errors.
4. Data Cleaning and Pre-processing: performed cleaning and pre-processing (as described above in part 3) using Pandas and NumPy
5. Export Processed Data to CSVs: export Pandas DataFrames into CSV files (using to_csv())
6. Database Setup: create a PostgreSQL RDS instance using AWS, and save credentials/database information
7. Connection: connect to PostgreSQL RDS instance using DataGrip
8. Schema Creation: in DataGrip, run SQL CREATE TABLE statements for property, features, disaster, disaster_types, and located
9. Data Import into Tables: import cleaned and pre-processed CSV files into tables
10. Verification: run a few sample queries to ensure that ensure data is loaded as expected and that relationships are correctly established.

Entity Resolution Efforts:

- Common Identifiers:
    - To link data from FEMA disasters to the Kaggle Real Estate listings, common identifiers such as location data (state, county, and zip codes) needed to be universal. We noticed that the real estate listings provided city and state names for properties, while the FEMA data provided only US zip codes for the disasters. To address this, we merged the disasters table with zip code data to add city information to the dataset. This was critical in ensuring that the same locations were consistently recognized across different datasets, and allowed us to join the tables later on.
- Data Standardization:
    - All datasets underwent a standardization process to align data formats, especially for location identifiers and dates. For example, we removed the strings "(County)" or

"(Parish)" leaving only the name of the city in disasters. For the designated date, entry date, and closeout date, all date formats were standardized to YYYY-MM-DD to facilitate date searching and comparisons.

- Data Cleaning:
    - Each dataset was cleaned to remove incomplete or irrelevant entries. For example, we checked for missing values (NaN) in city columns and removed those rows from the properties table. We also removed columns that were not important in querying or were not used in results. In disasters, we removed the hash and lastRefresh columns as this information would not be important to the user of StormHaven.

Entity Descriptions:

1. Property: Represents individual properties listed in the system.
    - Attributes: property_id (unique identifier), status (sale/build readiness), and price (property cost).
2. Disaster: Represents disaster events recorded by FEMA.
    - Attributes: disaster_id (unique identifier), disaster_number, designate_date, entry_date, and closeout_date.
3. Features: Stores additional details about properties
    - Attributes: property_id (unique identifier), bathrooms, bedrooms, and acre_lot.
4. Disaster Types: Categorizes disasters into distinct types and assistance types.
    - Attributes: disaster_id (unique identifier), type_code (type of disaster), and type_description (type of assistance).
5. Located: Links properties and disasters to geographic information.
    - Attributes: property_id (unique identifier), disaster_id (unique identifier), state_code, zip_code, city, and state.

Relationship Descriptions:

1. Property and Features:
    - One-to-one: Every property has a set of features.
    - One-to-many: A set of features can apply to multiple properties.
2. Property and Located:
    - One-to-one: Every property is located in one location.
    - One-to-many: A location can have multiple properties.
3. Property and Disaster:
    - One-to-many: A property can be affected by multiple disasters.
    - One-to-many: A disaster can affect multiple properties.
4. Disaster and Located:
    - One-to-one: Every disaster is located in one location.
    - One-to-many: A location can have multiple disasters
5. Disaster and Disaster Type:
    - One-to-one: Every disaster can only be categorized as one type of disaster.
    - One-to-many: Multiple disasters can be of the same disaster type.

3CF:

1. Property:
    a. Property_id → Price, Status
    b. Candidate Key = Property_id
        i. {Property_id}+ = {Property_id, Price, Status}
    c. LHS = Super key + No transitive dependencies → 3NF

2. Features:
    a. Property_id → Property_id, Bathrooms, Bedrooms, Acre_lot
    b. Candidate Key = Property_id
        i. {Property_id}+ = {Property_id, Bathrooms, Bedrooms, Acre_lot}
    c. LHS = Super key + No transitive dependencies → 3NF
3. Disaster_types:
    a. Disaster_id → Type_code, Type_description
    b. Candidate Key = Disaster_id
        i. {Disaster_id}+ = {Disaster_id, Type_code, Type_description}
    c. LHS = Super key + No transitive dependencies → 3NF
4. Disaster:
    a. Disaster_id → Disaster_number, Designated_date, Entry_date, Closeout_date
    b. Candidate Key = Disaster_id
        i. {Disaster_id}+ = {Disaster_id, Disaster_number, Designated_date, Entry_date, Closeout_date}
    c. LHS = Super key + No transitive dependencies → 3NF
5. Located:
    a. Property_id, Disaster_id → State_code, Zip_code, City, State
    b. Candidate Key = Property_id, Disaster_id
        i. {Property_id, Disaster_id }+ = {Property_id, Disaster_id , State_code, Zip_code, City, State}
    c. LHS = Super key + No transitive dependencies → 3NF

BCNF:

1. All functional dependencies in every table have a superkey on the LHS.
2. There are no partial, transitive, or non-superkey dependencies.
    a. Therefore, the schema is in BCNF.

5. Queries (Optimized Versions)

See appendix for screenshots of the code:

1. Complex Query 1 - Finds the most frequent disaster types in locations where the average property price exceeds $500,000, grouped by type_code.

   Implementation details: This query uses a CTE named Disaster_Frequency to aggregate the number of disasters associated with properties where the price exceeds $500,000. It joins the Located, Property, Disaster, and Disaster_Types tables to filter and group the data by disaster type, counting the occurrences of each type. The final output, selected from the CTE, is sorted to show the most common disaster types at the top, limited to the top 10 entries.

2. Complex Query 2 - Lists properties that have no disasters recorded in the past 5 years but are located in areas historically affected by high-risk disasters (e.g., type_code = 'HM')

   Implementation Details: This query uses two CTEs to segregate recent disaster occurrences and areas categorized under the 'HM' disaster type. It then joins these filtered sets with property data, employing a LEFT JOIN to ensure properties not affected by recent disasters are selected, if they lie within high-risk areas.

3. Complex Query 3 - Retrieve cities that have been impacted by fewer disasters than the average number of disasters per city in their state, showing property_id, city, and state.

Implementation Details: This query calculates disaster counts for each county within a state and then determines the average number of disasters per county for each state. It then selects counties where the disaster count is less than or equal to the state average, ordering the results by the disaster count in ascending order, using a row number to uniquely identify each row in the result set.

4. Complex Query 4 - Retrieves statistical information about properties located in cities and states affected by at least two different types of disasters.

   Implementation Details: This query utilizes a CTE to filter cities and states affected by at least two types of disasters. It then calculates and ranks these locations based on their average property prices, using the ROUND and AVG functions to ensure precise and understandable financial data presentation.

5. Query 5 - Lists the top 20 cities, states, and counties with the highest number of unique properties affected by disasters, sorted in descending order based on the count of affected properties.

   Implementation Details: The query utilizes the Located and Disaster tables, joining them on the disaster_id to access properties linked to specific disasters. It aggregates data by city, state, and county, counting distinct property IDs to ensure each property is only counted once per location. The results are sorted by the count of affected properties in descending order to identify the most impacted locations.

6. Performance Evaluation

Caching – Caching was used for all of our queries since they appeared on our dashboard. By caching the results of these complex queries in materialized views, we enhance the dashboard's performance, providing instant access to the data. Since the underlying data related to properties and disasters does not change often, using materialized views allows us to maintain an efficient and responsive system by periodically refreshing the views rather than recalculating them with every user query. This approach not only reduces the load on our servers but also ensures that the dashboard remains swift and efficient

Query	Original runtime	Optimized runtime	Inputs used from tables	Optimization techniques used and why they were used.
#1	30 s	8 s	Located - property_id - disaster_id Disaster - disaster_id Disaster_Types - type_code - disaster_id Property - property_id - price	**(Restructuring Query)** Moved the price filter (p.price > 500000) directly into the main query's WHERE clause which made it immediately reduce the dataset to only those properties that meet the price criterion. This was previously in a 'HAVING' clause. Removed an extra CTE called High_Priced_Locations from the initial query which reduced the overhead of managing an additional temporary data structure. Changed the grouping to only group by type_code, simplifying the grouping mechanism. It counts disaster IDs directly associated with high-priced properties.
#2	76 s	3 s	Located	**(Indexing)** Improved by using indexing on type_code in Disaster

			- property_id - disaster_id - city - state Disaster - disaster_id - designated_date Disaster_Types - type_code - disaster_id Property - property_id - city - state	because we were filtering by looking for type code 'HM' which is one of the most common type codes amongst disasters. **(Restructuring Query)** The original query used GROUP BY along with MAX(cj.price) to handle potential duplicates and to aggregate data. In the streamlined query, we use DISTINCT to remove duplicates directly.
#3	29 s	4 s	Located - county_name - state Disaster - disaster_id	**(Indexing)** We used two indexes to help optimize this query. We used an index on county_name, state (idx_Located_county_state) which is used to quickly group records by county_name and state in the CityDisasters subquery. We also used an index on disaster_id (idx_Located_disaster_id) to enhance the performance of the join between the Located and Disaster tables. When filtering and joining on disaster_id, the database can quickly locate the relevant rows in the Located table, reducing the time needed for lookup. **(Restructuring Query)** The original query had previously performed a nested subquery: a query to find the number of disasters per city nested within another query to find the average number of disasters per city for every state. We restructured this to include them as two separate temporary tables: CityDisasters and StateAverages. This allowed us to reuse the query results from CityDisasters in two cases—calculating StateAverages and returning the satisfying cities—without having to perform the calculations twice.
#4	21.5 s	13s	Located - city - state - disaster_id Disaster - disaster_id Disaster_Types - type_code - disaster_id Property - property_id - price	**(Restructuring Query)** Our previous query performs a detailed join on every record from the Located, Disaster, and Disaster_Types tables and then filters after processing the joins. Our optimized query limits the dataset before joining with the Property table by using the HAVING clause to ensure that only cities and states with the required number of disaster types are included.

7. Technical Challenges

Standardizing Data: Merging FEMA disaster records with real estate listings was a significant challenge due to inconsistent formats and details. FEMA data focused on disaster locations and timelines, while real estate data provided property-specific attributes. These differences required extensive cleaning and transformation to align the datasets, ensuring compatibility for accurate analysis. For example, a large challenge we encountered was that the real estate listings provided city and state names for properties, while the FEMA data provided only US zip codes for the disasters. Since our app relied on joining the real estate data to FEMA data, we had to add a field to either table that they could join.

Frontend Dashboard: On the dashboard, where users interact with buttons to load queried data, we encountered several issues. The first challenge was that one button's state affected another, leading to incorrect data being displayed. Resolving this required isolating state management in React using useState and useEffect, ensuring each button triggered only its intended action. Another challenge was toggling between the "loading…" screen and the results screen. At times, the website would be stuck on the "loading…" screen despite the data already being loaded. To address this, we introduced new variables to control the loading and displaying of data. For example, isLoadingAnalytics controls the display of the "loading…" for analytics.

Backend Searching: The most challenging part of developing FindHouses and DisasterRisks was developing the search queries for those pages. Specifically, we could not search for properties or disasters if any of the features like city, price, designated date, etc. were not provided. To address this, we made sure to provide default values if a certain feature was not provided. For example, if the designated date range for a disaster was not provided, we set the minimum date to be year 1000 and the maximum date to be year 3000. These bounds ensure that all of the FEMA disasters would be included in the search if no min or max were provided.

Complex Queries
Queries on tables with millions of rows, like filtering properties or identifying high-risk areas, were slow and computationally heavy. We optimized performance using indexing, query caching, and schema restructuring, which reduced response times while maintaining accuracy for real-time analytics. At times, when we just needed to view portions of the returned results, we made use of the LIMIT condition or added an additional WHERE condition.

Deployment
When exploring the extra credit option of deployment, we faced several issues. Firstly, we discovered that our project had a mono repo format: both our frontend and backend were stored in the same project. This meant that we could not host the entire project on one platform. Thus, to address this issue, we had to explore deploying the client side and the server side of the code on different platforms. We were able to connect our GitHub repository to Vercel and have it run only the client code. Next, we connected a duplicate version of the GitHub repository to Render and had it run the server side. Connecting the query results from the Render website to the Vercel frontend, we were able to deploy our full-stack project!

Appendix

Complex Query 1: Find the most frequent disaster types in locations where the average property price exceeds $500,000, grouped by type_code.

- Part of Analytics in Dashboard

```
WITH Disaster_Frequency AS (
    SELECT dt.type_code, COUNT(d.disaster_id) AS disaster_count
    FROM Located l
    JOIN Property p ON l.property_id = p.property_id
    JOIN Disaster d ON l.disaster_id = d.disaster_id
    JOIN Disaster_Types dt 1<->1: ON d.disaster_id = dt.disaster_id
    WHERE p.price > 500000
    GROUP BY dt.type_code
)
SELECT type_code, disaster_count
FROM Disaster_Frequency
ORDER BY disaster_count DESC
LIMIT 10;
```

Complex Query 2: List properties that have no disasters recorded in the past 5 years but are located in areas historically affected by high-risk disasters (e.g., type_code = 'HM')

- Part of Overview in Dashboard

```
CREATE INDEX idx_disaster_types_type_code ON Disaster_Types(type_code);
```

```
WITH Recent_Disasters AS (
  SELECT DISTINCT l.property_id
  FROM Located l
  JOIN Disaster d ON l.disaster_id = d.disaster_id
  WHERE d.designateddate >= CURRENT_DATE - INTERVAL '5 YEARS'
),
High_Risk_Areas AS (
    SELECT DISTINCT l.city, l.state
    FROM Located l
    JOIN Disaster_Types dt ON l.disaster_id = dt.disaster_id
    WHERE dt.type_code = 'HM'
)
SELECT p.property_id, l.city, l.state
FROM Property p
JOIN Located l ON p.property_id = l.property_id
WHERE (l.city, l.state) IN (SELECT city, state FROM High_Risk_Areas)
AND p.property_id NOT IN (SELECT property_id FROM Recent_Disasters)
LIMIT 100;
```

Complex Query 3: Retrieve cities that have been impacted by fewer disasters than the average number of disasters per city in their state, showing property_id, city, and state.

- Part of Analytics in Dashboard

```
CREATE INDEX idx_located_county_state ON located(county_name, state);
```

```
CREATE INDEX idx_located_disaster_id ON located(disaster_id);
```

```
WITH CityDisasters AS (
  SELECT l.county_name, l.state, COUNT(*) AS disaster_count
  FROM public.located l
  JOIN public.disaster d ON l.disaster_id = d.disaster_id
  GROUP BY l.county_name, l.state
),
StateAverages AS (
  SELECT state, SUM(disaster_count) / COUNT(county_name) AS avg_disasters_per_city
  FROM CityDisasters
  GROUP BY state
)
  SELECT ROW_NUMBER() OVER (ORDER BY cd.disaster_count ASC) AS row_index, cd.county_name, cd.state, cd.disaster_count
  FROM CityDisasters cd
  JOIN StateAverages sa ON cd.state = sa.state
  WHERE cd.disaster_count <= sa.avg_disasters_per_city
  ORDER BY cd.disaster_count ASC;
```

Complex Query 4: Retrieves statistical information about properties located in cities and states affected by at least two different types of disasters

- Part of Analytics in Dashboard

```
WITH CityDisasterCounts AS (
  SELECT
      l.city,
      l.state,
      COUNT(DISTINCT dt.type_code) AS num_disaster_types
  FROM Located l
  JOIN Disaster d ON l.disaster_id = d.disaster_id
  JOIN Disaster_Types dt 1<->1: ON d.disaster_id = dt.disaster_id
  GROUP BY l.city, l.state
  HAVING COUNT(DISTINCT dt.type_code) >= 2
)
  SELECT
      ROW_NUMBER() OVER (ORDER BY AVG(p.price)::numeric DESC) AS row_index,
      cdc.city,
      cdc.state,
      ROUND(AVG(p.price)::numeric, 2) AS average_price
  FROM Property p
  JOIN Located l ON p.property_id = l.property_id
  JOIN CityDisasterCounts cdc ON l.city = cdc.city AND l.state = cdc.state
  GROUP BY cdc.city, cdc.state
  ORDER BY average_price DESC;
```

Query 5: Identifies properties affected by the highest number of disaster events in past year

- Part of Overview on Dashboard

```
SELECT
l.city,
l.state,
l.county_name,
COUNT(DISTINCT l.property_id) AS affected_properties
FROM Located l
JOIN Disaster d ON l.disaster_id = d.disaster_id
GROUP BY l.city, l.state, l.county_name
ORDER BY affected_properties DESC
LIMIT 20;
```

Query 6: Retrieves properties with that have been affected by a disaster in the past 2 years

- Part of Overview on Dashboard

```
SELECT DISTINCT p.property_id, p.price, p.status, l.city, l.state, d.designateddate
FROM Property p
JOIN Located l ON p.property_id = l.property_id
JOIN Disaster d ON l.disaster_id = d.disaster_id
WHERE d.designateddate >= NOW() - INTERVAL '2 year'
ORDER BY d.designateddate DESC
LIMIT 100;
```

Query 7: Retrieves properties that match the provided property id, state, city, and/or status (if not provided, default TRUE) and is within the price, bathroom, bedroom, and acre range (if not provided, we set default ranges)

- Part of FindHouses and Favorites

```
SELECT *
FROM public.Property p
JOIN public.Features f ON p.property_id = f.property_id
WHERE ${propertyIdFilter}
  AND ${stateFilter}
  AND ${countyFilter}
  AND ${statusFilter}
  AND price BETWEEN ${priceLow} AND ${priceHigh}
  AND bathrooms BETWEEN ${bathroomsLow} AND ${bathroomsHigh}
  AND bedrooms BETWEEN ${bedroomsLow} AND ${bedroomsHigh}
  AND acre_lot BETWEEN ${acreLotLow} AND ${acreLotHigh}
ORDER BY p.property_id ASC
LIMIT 1000;
```

Query 8: Retrieves the disasters that have happened at a given property id

- Part of PropertyCard, which appears on FindHouses and Favorites

```
SELECT DISTINCT(d.disaster_id), d.disasternumber, d.designateddate, d.closeoutdate, dt.type_code, dt.type_description
FROM public.property p
JOIN public.disaster d ON p.county_name = d.county_name
JOIN public.disaster_types dt ON d.disaster_id = dt.disaster_id
WHERE ${propertyIdFilter}
ORDER BY d.designateddate DESC
LIMIT 1000;
```

Query 9: Retrieves disasters that match the provided disaster id, number, type code, and/or city (if not provided, default TRUE) and is within the designated date range (if not provided, we set a default range)

- Part of DisasterRisks page

```
SELECT *
FROM public.Disaster d
JOIN public.Disaster_Types dt ON d.disaster_id = dt.disaster_id
WHERE ${disasterIdFilter}
  AND ${disasterNumberFilter}
  AND ${typeCodeFilter}
  AND ${countyFilter}
  AND d.designateddate >= '${designatedDateLow}'
  AND d.designateddate <= '${designatedDateHigh}'
ORDER BY d.disasternumber ASC
LIMIT 1000;
```

Query 10: Summarizes disaster counts per year by type code

- Part of DisasterRisks page

```
SELECT
  row_number() OVER (ORDER BY EXTRACT(YEAR FROM designateddate) DESC, COUNT(d.disaster_id) DESC) AS index,
  EXTRACT(YEAR FROM designateddate) AS year,
  dt.type_description,
  COUNT(d.disaster_id) AS disaster_count
FROM disaster d
JOIN disaster_types dt ON d.disaster_id = dt.disaster_id
WHERE EXTRACT(YEAR FROM designateddate) <= 2024
GROUP BY year, dt.type_description
ORDER BY year DESC, disaster_count DESC;
```