NANYANG TECHNOLOGICAL UNIVERSITY

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4125: Capstone Project: Property Search

Name	Matric No.
Kee Kai Teng	U2020909E
Zhong Shaojie	U2020758K
Darryl Chia Shen Yang	U2022120B
Sa Ziheng	U2022717L

Contribution list

Task	Contributors
Conception of Problem Statement and Implementation Plan	Darryl Chia Kee Kai Teng Sa Ziheng Zhong Shaojie
Collection of data and pre-processing	Darryl Chia Kee Kai Teng Sa Ziheng Zhong Shaojie
Implementation of Algorithm	Sa Ziheng Zhong Shaojie
Implementation of backend system (Postgres)	Kee Kai Teng Zhong Shaojie
Frontend design (streamlit)	Kee Kai Teng Sa Ziheng Zhong Shaojie
Implementation of pipelines for data freshness)	Darryl Chia Kee Kai Teng Zhong Shaojie
Report Writing	Darryl Chia Kee Kai Teng Sa Ziheng Zhong Shaojie
Video Preparation	Darryl Chia Sa Ziheng

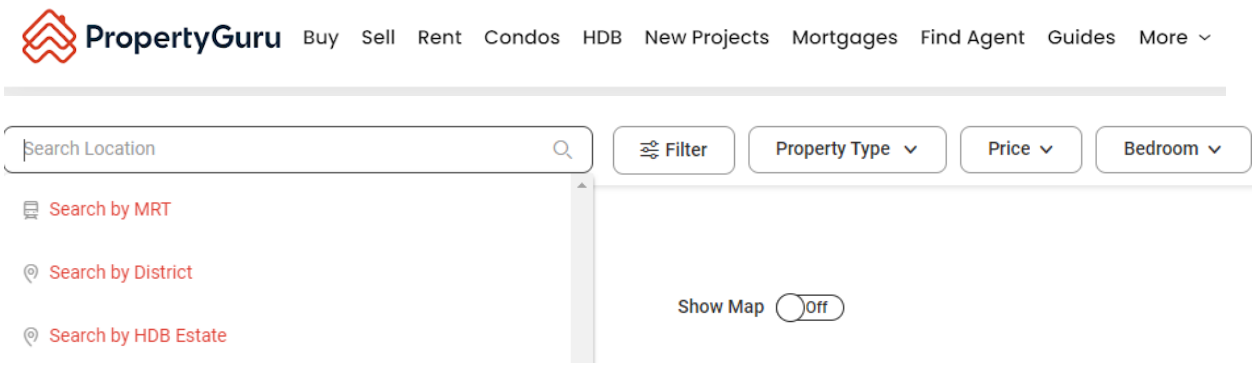
Problem Statement

While many existing online property portals in Singapore such as PropNex and PropertyGuru serve well in providing general property search services, they have several limitations when it comes to catering to individual preferences. Some of the observed flaws include a restrictive filtering system based solely on property traits and location which leads to an inadequate prioritisation of individual accessibility and convenience, a notable shortfall in the customisation of user profiles (such as the ability to input frequently visited places). For example, existing platforms may display properties near specified amenities based on user filters, without incorporating the critical factor of travel time. This oversight means that the showcased properties may be in close proximity to amenities, but far in terms of actual time required to travel to these locations depending on chosen mode of transport thereby resulting in suboptimal selection of properties for the user. The lack of flexibility ultimately hinders the platform's ability to provide tailored and personalised recommendations based on individual user preferences and needs.

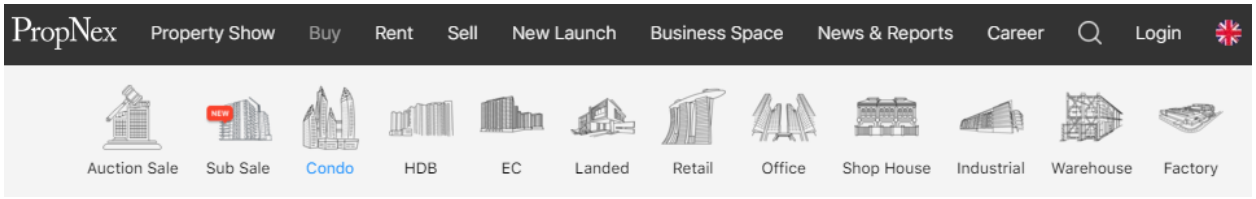
The goal of this project is to address the lack of personalisation in home seekers' journey in the search of their new homes, by providing them with a more holistic and personalised property search experience. This approach aims at leveraging the existing online property portals and connecting them seamlessly with external data sources to enhance the user experience of home search.

Similar products

PropertyGuru



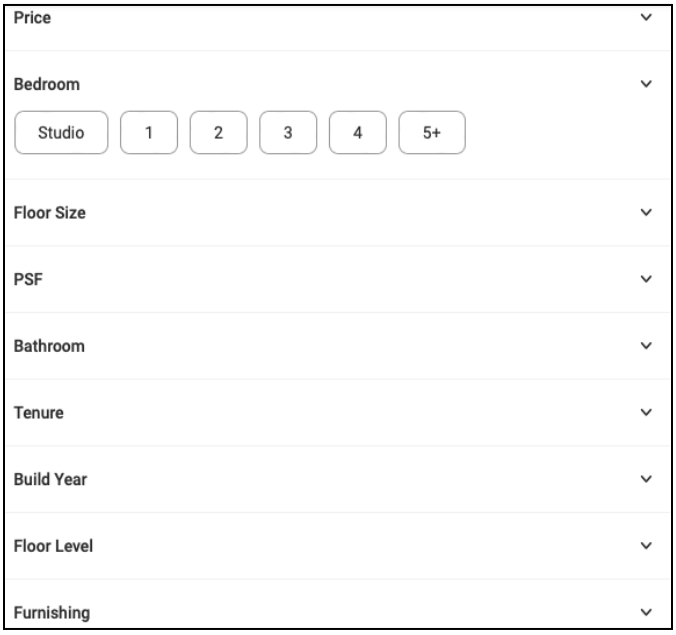
Propnex



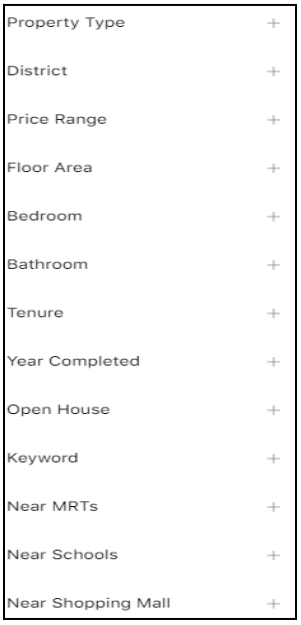
PropertyGuru and Propnex are Singapore-based websites helping people with property related services. These services include buying, selling or renting a property, legal guides on housing related matters and providing information of property agents. Their range of property covers private and public housing, ranging from HDB (Housing Development Board) flats to private landed houses. Its search feature is shown above. Users are able to search for a particular location and these websites will identify all available listings in the area, taking into account any filters that the user has input. Some of the different filters for both companies are shown below. For example, searching using an MRT station will show listings near that station.

Available filters on these platforms include:

1) Traits of property
    a) Properties Type
    b) Price, PSF (Per Square Feet)
    c) Bedroom/ Bathroom numbers
    d) Floor Level
    e) Building Age
2) Location
    a) Proximity to MRT and Major Streets
    b) Proximity to Amenities



Search filters of PropertyGuru



Search filters of Propnex

Data

A modular strategy is applied to distinctly separate Data Processing from Data Extraction. This deliberate decoupling enhances system flexibility and streamlines the integration of new data types and ensures that any modifications or enhancements to either part can be executed with greater ease through the principles of modular programming.

Sources

The data sources used in this project are extracted from and provided by OneMap and PropertyGuru. OneMap is Singapore's authoritative national map developed by the Singapore Land Authority, which offers precise routing data for various transportation modes (bus, MRT, car, cycling, walking) and grants access to over 100 thematic layers depicting the location of amenities and facilities. Additionally, it provides the corresponding geographical coordinates for a specific address. PropertyGuru is a platform facilitating connections between property sellers and buyers, which enables information from available property listings to be extracted.

Extraction

To obtain information on PropertyGuru, BeautifulSoup and Selenium packages are used to scrape on all open listing pages. One of the key challenges faced during this process was that PropertyGuru has implemented reCAPTCHA, a tool that protects websites from spam and abuse, starting from page 10 of all listings. To overcome this problem, the reCAPTCHA is resolved manually when scraping the website for the first time. Subsequently, it is recommended to scrape the first 9 pages on a frequent basis so as to update the existing list of properties without running into reCAPTCHA. A unique property identifier obtained from the listing is used to ensure that there are no duplicate properties in the database.

Processing

PropertyGuru Listings

After extracting all the open listings from PropertyGuru, data processing is needed to transform the text data into a suitable format before it can be displayed on the dashboard. A significant hurdle in the data processing phase was the largely unstructured nature of the text data. This necessitated the use of custom string matching patterns to identify and extract the relevant information before it could be transformed into the appropriate data type. The figure below is an example of how the cost per square feet for a property from the listing page is extracted.

```python
def clean_cost_psf(cost_psf):
    if cost_psf is None:
        return None
    cost_psf = cost_psf.strip()
    pattern = r"\d{1,3}(?:,\d{3})*(?:\.\d{2})?"
    cost_psf = re.search(pattern, cost_psf)
    cost_psf = float(cost_psf.group(0).replace(',', ''))
    return cost_psf
```

Extracting text from html using regex

Coordinates (Property)

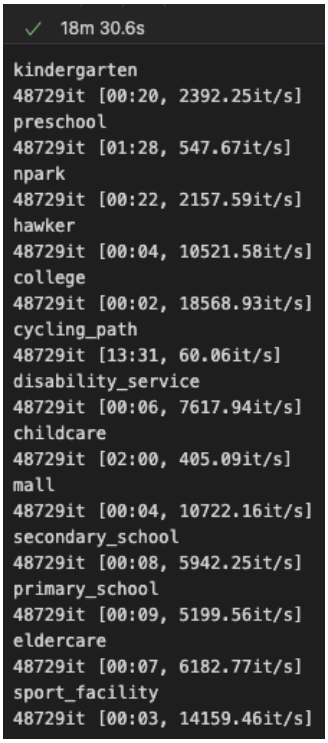
As PropertyGuru does not have an explicit field that provides the coordinates to a specific address, OneMap Search API is utilised to obtain the corresponding geographical coordinates to the address of each property listing. In order to optimise the processing performance and minimise API call frequency, a local database is created for storing OneMap address search results. Instead of calling the API multiple times, the local database, which is saved as a Python dictionary, serves as a cache allowing quick look up of $O(1)$ constant time. This is significantly faster than calling the OneMap API multiple times.

Amenities

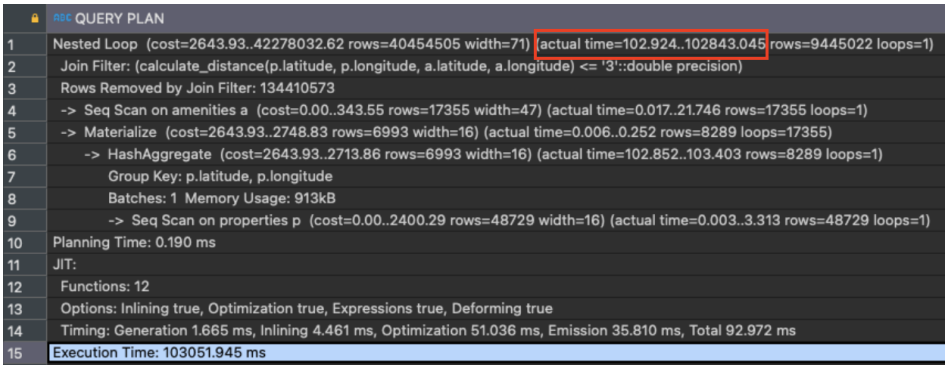
Another key design consideration in the application was to precompute the distance to all amenities for each listed property and shortlist all the nearby amenities based on a specified threshold e.g all amenities within 3km radius. By pre-computing these distances and shortlisting the nearby amenities beforehand, it ensures that the information is readily available to be displayed on the map when it is recommended as the top few ideal properties based on the user's filters. This approach eliminates the need for on-the-fly computation of the distances to all

amenities, which is computationally expensive and time-consuming. This design choice not only improves the application's performance but also provides users with quick and accurate property recommendations.

This computation in Python utilising a for loop, iterating every single property against every single amenity, results in about 9 million iterations and took a total of 18.5 mins to complete. In comparison, when this computation is done in a Postgres database, the time taken to perform this calculation is 1.7 mins (102 secs).



Time Taken in Python



Time Taken in Postgres

PostgreSQL Database

PostgresSQL database is designed to handle substantial volume of data and can scale horizontally to accommodate the growing demands of an application. This scalability is particularly beneficial when handling expensive CSV files that may surpass available memory, providing a more efficient and scalable alternative to loading the entire CSV file into memory. This shift reduces the data in memory, resulting in a more responsive application.

Another rationale behind opting for a relational database stems from the structured nature of the data. Given that the dataset exhibits a well-defined and organised structure with consistent formats and relationships between various elements, utilising a relational database aligns seamlessly with these characteristics.This choice not only facilitates effective data management but also streamlines the retrieval and manipulation of structured information within the application.

Furthermore, the inclusion of parallel query execution capabilities in PostgreSQL significantly contributes to enhanced application performance. This feature enables the distribution of query workloads across multiple CPUs or cores, resulting in expedited query execution—an invaluable asset, especially when dealing with extensive data. As shown above, offloading computations from Python onto the PostgreSQL database results in a significant reduction in processing time.

Implementation

Data Pipeline

Given that the application provides optimal location recommendations based on travel time, it is imperative that the data is kept as current as possible. This is because any changes, such as the introduction of new park connectors or new properties could significantly impact the travel time and, consequently, the location recommendations provided by the application. Hence, it's important to ensure the data is up-to-date in order to provide users with the most accurate and relevant recommendations.

To ensure the continual relevance and accuracy of the data featured on the application, a systematic approach was implemented to retrieve new listings found on PropertyGuru and update the list of nearby amenities for each property based on the latest location of all amenities. The respective scripts are orchestrated through the *cron* scheduling tool, enabling them to run at predefined intervals automatically, allowing us to proactively update the database with any new open listing found on PropertyGuru as well as the latest locations of all amenities.

```
*/30 0-2,3-23 * * * ./pipeline_update_properties.sh
15 2 * * * ./pipeline_update_amenities.sh
```

Scheduling of pipeline using crontab

```
# Step 1: Scrape recent listings from property guru
os.makedirs(unprocessed_dir, exist_ok=True)
scrape_latest_results(…

# Step 2: Extract information from raw html file
os.makedirs(raw_processed_dir, exist_ok=True)
extract_listings(…

# Step 3: Clean and process the data
os.makedirs(processed_dir, exist_ok=True)
process_search_results(…

# Step 4: Upsert property listings to postgres
upsert_to_postgres(…

# Step 5: Upsert to postgres      You, 1 second ago • U
calculate_property_amenities()
create_agg_property_table()
```

Pipeline to get latest property listings

```
# Get amenities      You, 1 second
get_childcare(…

get_college(…

get_disability_service(…

get_eldercare(…

get_hawker(…

get_kindergarten(…

get_malls(…

get_nparks(…

get_preschool(…

get_primary_school(…

get_secondary_school(…

get_sports_facility(…

get_cycling_path(…

# Upload amenities to postgres
upload_amenities(…

# Calculate nearby amenities
calculate_property_amenities()
create_agg_property_table()
```

Pipeline to get latest amenities

Algorithm

Overview

The algorithm accepts the coordinates of the users frequently visited locations, the mode of transport as well as the frequency per week.

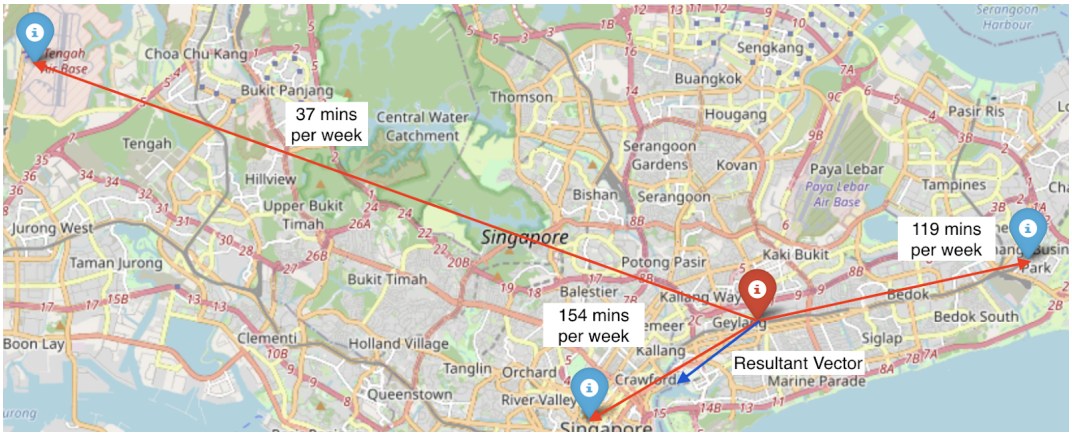
To get the initial search location, a weighted initialization is performed using the coordinates as well as the frequency to get a search location.

$$\text{search location} = \left[\frac{1}{N}\sum_{i=1}^{N} lat_i \cdot f_i \quad \frac{1}{N}\sum_{i=1}^{N} long_i \cdot f_i\right]$$
$$lat_i \text{ is the latitude of the } i_{th} \text{ frequently visited place}$$
$$long_i \text{ is the longitude of the } i_{th} \text{ frequently visited place}$$
$$f_i \text{ is the weekly travel frequency of the } i_{th} \text{ frequently visited place}$$

Based on the search location, the travel time to each of the frequently visited locations using the OneMap Route API is obtained. Subsequently, vectors were generated for each of the frequently visited locations to the search location. The magnitude of each vector can be considered as the cost of travelling to this location which is denoted by the equation below. The direction of the vector is the forward azimuth from the search location to the frequently visited location.

$$\text{cost}_i = \exp\left(t_i \cdot f_i\right)$$
$$t_i \text{ is the travel time in hours to the } i_{th} \text{ frequently visited place}$$
$$\text{cost}_i \text{ is the cost of travelling to the } i_{th} \text{ frequently visited place}$$

Upon resolving the cost vectors, a single resultant vector with the associated azimuth/bearing can be computed. The search location will be updated in the direction of the resultant vector by a distance of 2km (learning rate). Below is an example of the different cost vectors as well as the resultant cost vector.



Visual Representation of Algorithm

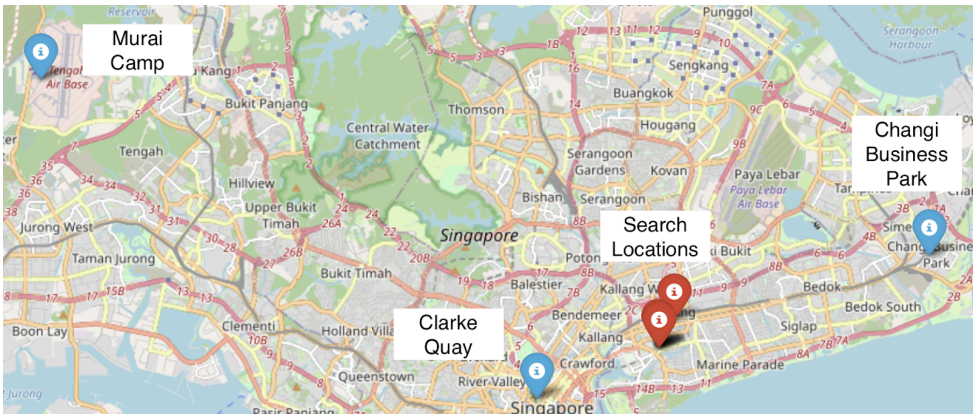
The above procedure is repeated 10 times with the updated search location. The search location with the shortest total travelling time per week will be selected as the ideal location. Properties closest to this ideal location will be displayed to the user. It is worth noting that while occasionally adjusting the search location in the direction of the resultant vector may lead to longer travel times, such instances are rare. Consequently, the overarching assumption that moving the search location in the direction of the resultant vector generally leads to lower travel times remains valid.

Unweighted vs Weighted Initialization

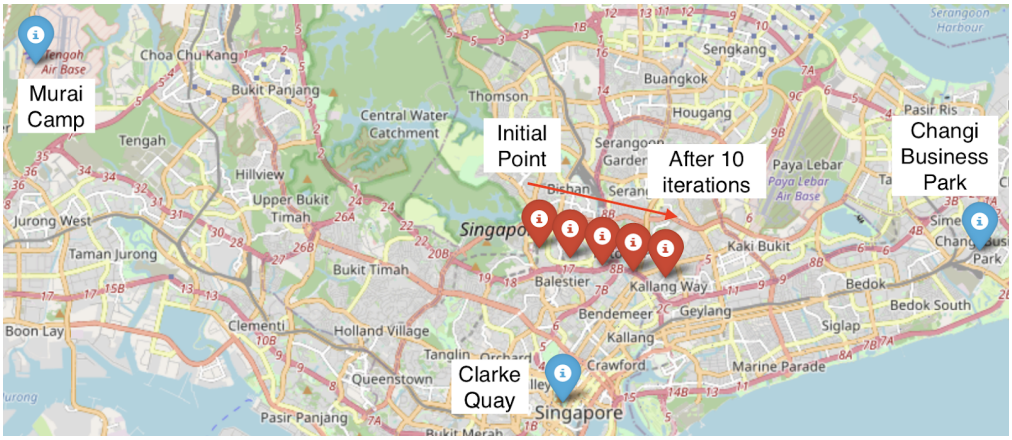
To illustrate that the weighted initialisation helps the algorithm converge faster, an example is showcased below, where the updated search location at every iteration is plotted.

Location	Coordinates	Frequency Per Week	Mode of Transport
Changi Business Park	[1.3349, 103.9629]	7	Public Transport
Clarke Quay	[1.3869, 103.7008]	7	Public Transport
Murai Camp	[1.2929, 103.8472]	1	Drive

As shown by the two images below, it can be observed that under the unweighted initialization, the search locations tend to move towards the search location under weighted initialization. Hence by having a better starting search location, better results can be achieved with the same number of iterations. This also allows us to reduce the update distance per iteration to allow for a more fine grain search.



Weighted Initialisation



Unweighted Initialisation

The second benefit of the weighted initialisation is that it allows us to put more weights on locations that are more frequently visited per week and hence allow the travel time per trip for these places to be much lower. The table below shows the travel time per trip to each of the locations on initialisation.

	Weighted		Unweighted	
	Initial Location	Best Location	Initial Location	Best Location
Changi Business Park	17 mins	17 mins	37 mins	29 mins
Clarke Quay	22 mins	22 mins	30 mins	18 mins
Murai Camp	37 mins	37 mins	30 mins	33 mins

Despite only needing to travel to Murai Camp once a week, the travelling time is approximately equal to the other 2 locations in the unweighted initialisation.

Cost Function

$$\text{cost}_i = \exp(t_i \cdot f_i)$$
$$\text{cost}_i = \text{cost of travelling to the } i_{th} \text{ frequently visited place}$$
$$\text{Total Cost} = \sum_{i=1}^{N} \text{cost}_i$$

The travel cost to each location is expressed as an exponential function of travel time in hours, rather than minutes multiplied by the weekly travel frequency. This decision safeguards against an unrealistic escalation of costs as seen in the following animation where the blue line represents the equation $cost_i = f_i * \exp(60 * t_i)$ and the red line represents the cost function $cost_i = \exp(f_i * t_i)$. The fundamental rationale behind adopting the exponential function lies in the belief that travel costs rise at an accelerating pace with each passing minute. Integrating the weekly travel frequency into the equation guides the algorithm toward locations with lower travel times for more frequented places. Simultaneously, the exponential function acts as a balancing mechanism, preventing travel times for less frequented locations from becoming excessively large. This ensures a nuanced and balanced consideration of travel times across various locations based on both travel time and the frequency of visitation.

Robust Search

In addition to initialising with the weighted point, the approach involves the initialisation of neighbouring points around the weighted point. This strategic step is implemented to facilitate a more comprehensive and exhaustive search process. By extending the initialisation to include adjacent points, it helps to capture a broader range of potential solutions within the proximity of the weighted point. This initialisation strategy enhances the algorithm's ability to explore and evaluate a diverse set of possibilities, promoting a more robust and effective search process.

Parallelising Search

In the computations above, the major bottleneck occurs when retrieving the travel time through the OneMap API. N API calls are made at every iteration. This takes up a bulk of the time in running the algorithm. In order to reduce this bottleneck, asynchronous http requests using the *async* and *aiohttp* packages in Python are implemented. This allows simultaneous API calls to be made and wait for the responses thereafter, instead of waiting for a response before making the next call. However, the number of asynchronous calls is limited by the OneMap Route API rate limit which is about 12 simultaneous API calls.

The table below shows the difference in time to complete 10 iterations of the algorithm for a different number of API calls per iteration. As shown, the synchronous API calls has a time complexity of $O(\#\ of\ API\ calls)$ while the asynchronous API calls scales much better.

	Synchronous	Asynchronous
3 API calls per iteration	31.2 secs	11.7 secs
6 API calls per iteration	59.0 secs	11.5 secs
9 API calls per iteration	89.0 secs	18.1 secs

Synchronous vs Asynchronous API Calls

Besides using asynchronous API calls, the bulk of the calculations were performed using *Numpy* arrays instead of performing them on Python Lists. As *Numpy* utilises vectorization, parallel processing and integration with C, it allows for faster matrix computation. However, it is noted that given the small number of operations per iteration, the improvement in speed is likely to be negligible. This will be useful when using an API with a higher rate limit such as GoogleRoutes API, where more search locations can be generated at a single time to perform a more complete and robust search.

```
def resolve_cost_vectors(azimuth, costs):
    v_cost_vectors =  costs * np.cos(azimuth*np.pi/180)
    h_cost_vectors = costs * np.sin(azimuth*np.pi/180)
    cost_vectors = np.hstack([v_cost_vectors, h_cost_vectors])
    resultant_cost_vectors = np.sum(cost_vectors,axis=0)
    resultant_azimuth = np.degrees(np.arctan2(resultant_cost_vectors[:,1], resultant_cost_vectors[:,0])).reshape(-1,1)
    return resultant_azimuth
```

Resolving Vectors using Numpy

Data Product

Integrating the aforementioned components, a data product is developed using Streamlit, with a primary focus on determining the most suitable dwelling location for users based on their commuting patterns.

In tandem with this development, a concise User Guide is prepared (Refer to the Appendix for a snapshot of the User Guide), outlining the identified pain points addressed by the product and providing comprehensive instructions for users.

Upon user configuration input, the algorithm is run and the output is used to query the PostgreSQL database to retrieve relevant listings in proximity. Simultaneously, these user-configured parameters are logged into a dedicated PostgreSQL database. This functionality lays the foundation for valuable data analyses. These analyses have potential applications in public transport development and urban planning, which underscores the broader significance of this product.

Conclusion

In summary, this product serves as a supplementary tool for prospective home seekers, enabling them to locate optimal living locations based on their daily routines. This product leverages well-established property platforms such as PropertyGuru or other prominent real estate websites to provide a unique value proposition of streamlining the search process, empowering users to conduct more targeted searches for their ideal homes.

To elevate the overall user experience, the focus of this project centred heavily on optimising computational efficiency. Various computational enhancements were deployed, including the integration of asynchronous API calls, the application of matrix operations in Numpy, enhanced algorithm initialization, and the implementation of offline computations for nearby amenities stored in PostgreSQL. These measures collectively contribute to expediting and refining the user interface, thereby fostering an enhanced and more user-friendly experience for individuals navigating the platform.

Appendix

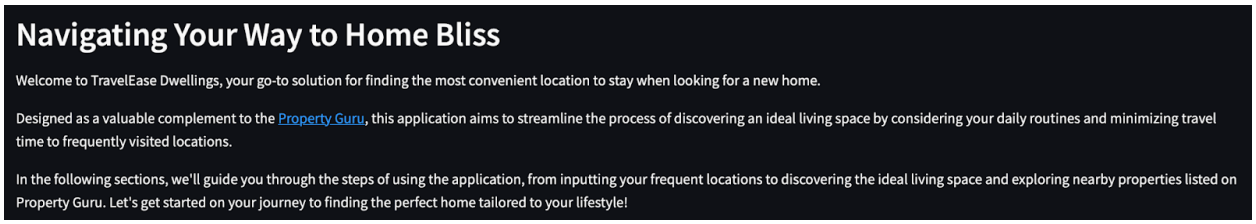
LLM Service Used

1. www.phind.com
    a. Debugging Purposes

Github Link

https://github.com/Shaojieee/Property-Search

User Guide



Exploring Ideal Location and Nearby Properties

Step 9: Explore Your Ideal Location

After clicking, a new map will appear below. Your frequently visited places will be marked in blue, the ideal location in red, and nearby properties in grey.

Step 10: Check Property Listings

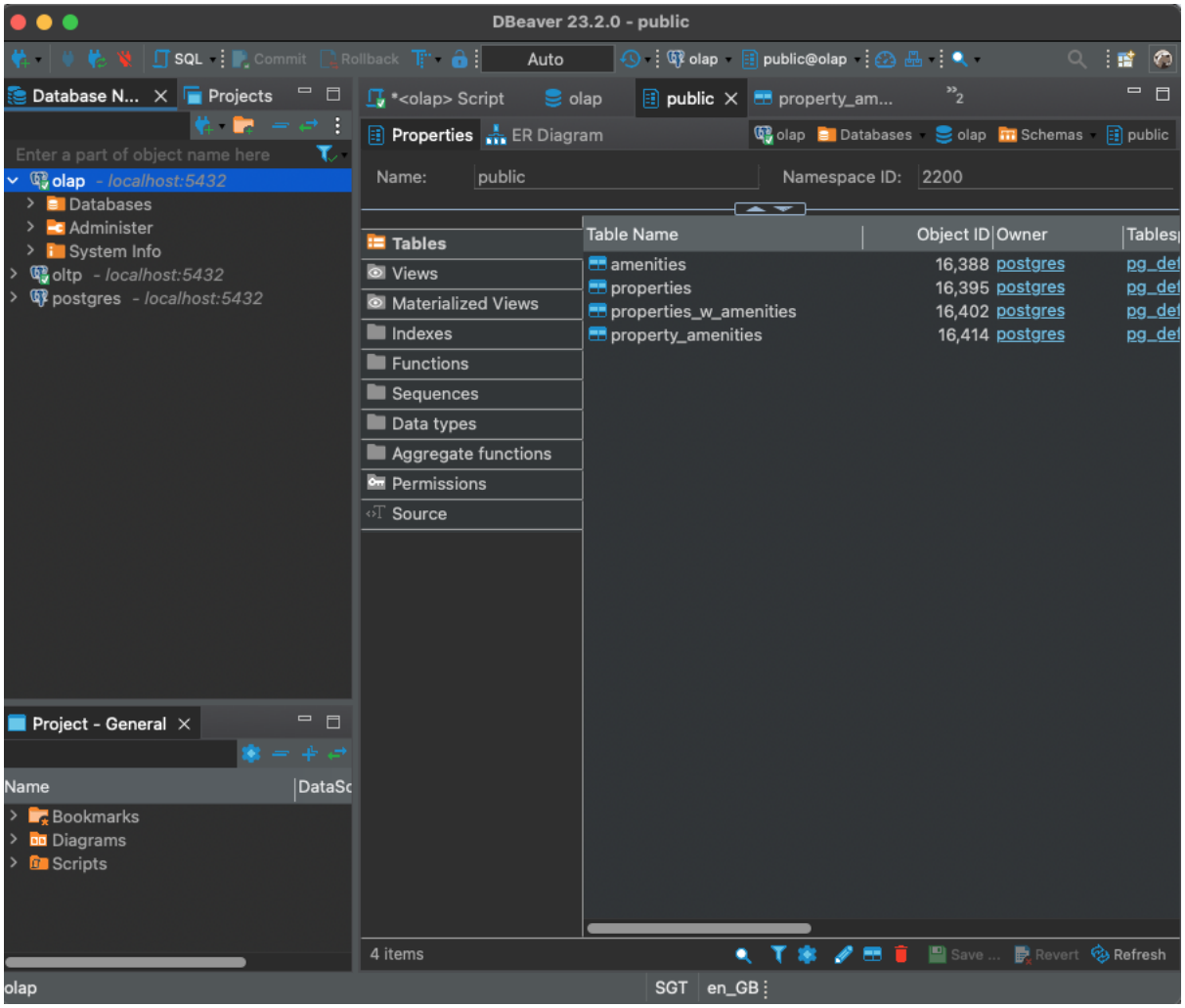
Click on a grey marker to see property details. A blue route will outline the travel route to your frequently visited place and property listing information below.

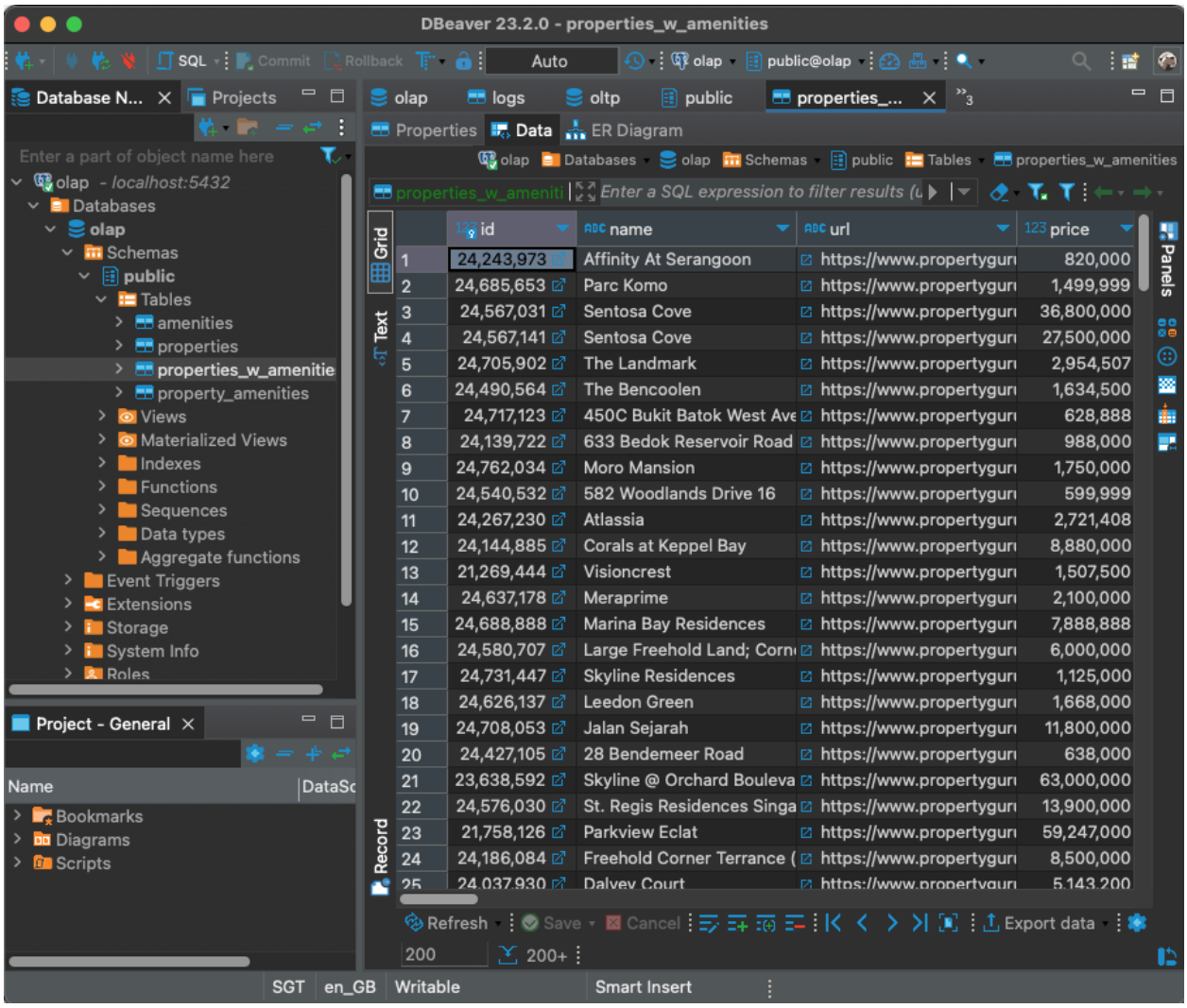
Step 11: Refine Your Search with Filters

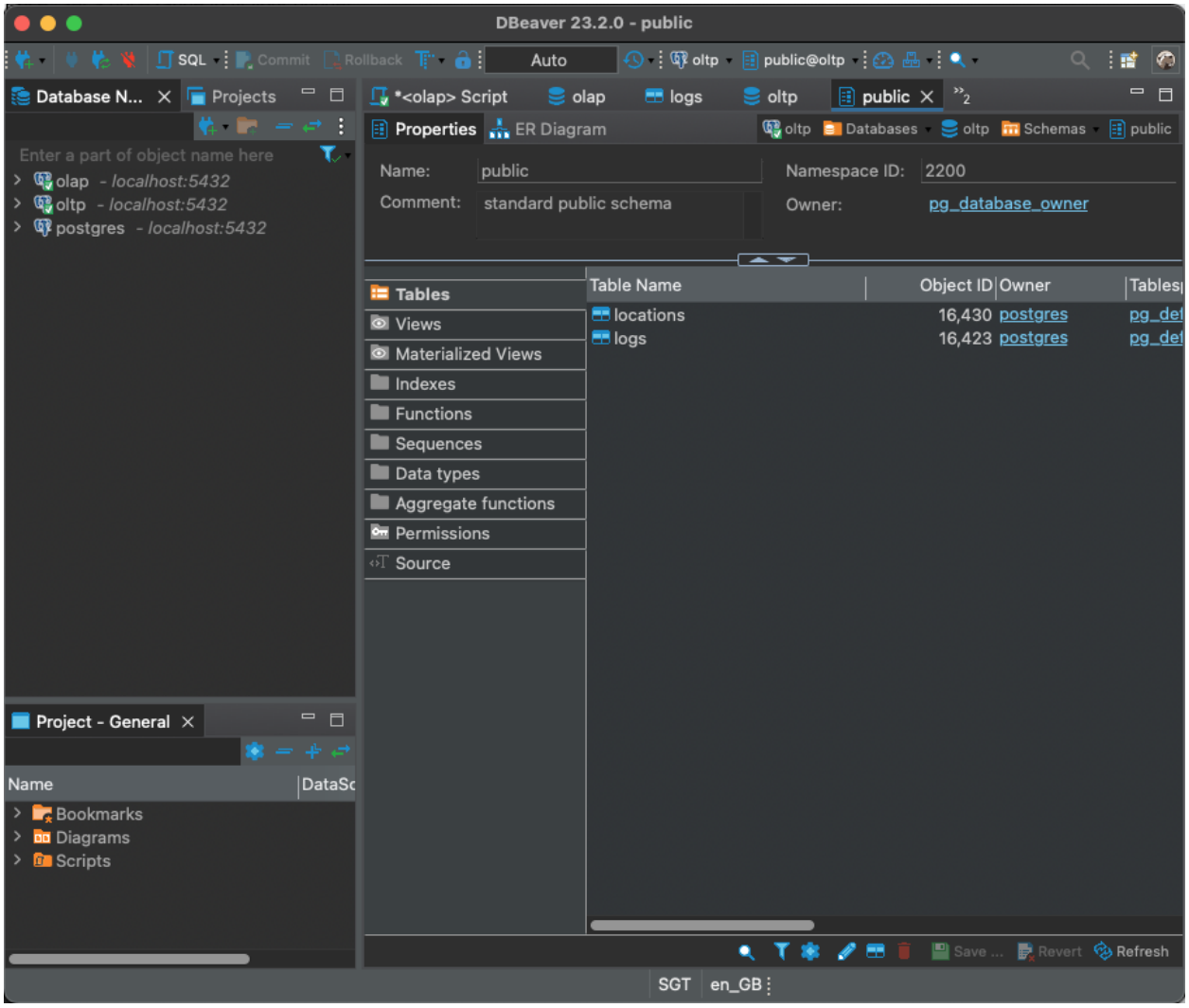
Utilize filters to tailor your property search:

- Amenities: Find properties that match your desired amenities.
- Property Type: Narrow down your choices based on property types.
- Price: Set a price range to fit your budget.

Snapshot of Database




DBeaver 23.2.0 - properties_w_amenities
Database N...
Projects
olap - localhost:5432
Databases
olap
Schemas
public
Tables
amenities
properties
properties_w_amenitie
property_amenities
Views
Materialized Views
Indexes
Functions
Sequences
Data types
Aggregate functions
Event Triggers
Extensions
Storage
System Info
Roles
Project - General
Bookmarks
Diagrams
Scripts
olap
logs
oltp
public
properties_...
Properties
Data
ER Diagram
Databases
Schemas
Tables
Grid
Text
Record
Panels
id
name
url
price
1 24,243,973 Affinity At Serangoon https://www.propertyguri 820,000
2 24,685,653 Parc Komo https://www.propertyguri 1,499,999
3 24,567,031 Sentosa Cove https://www.propertyguri 36,800,000
4 24,567,141 Sentosa Cove https://www.propertyguri 27,500,000
5 24,705,902 The Landmark https://www.propertyguri 2,954,507
6 24,490,564 The Bencoolen https://www.propertyguri 1,634,500
7 24,717,123 450C Bukit Batok West Ave https://www.propertyguri 628,888
8 24,139,722 633 Bedok Reservoir Road https://www.propertyguri 988,000
9 24,762,034 Moro Mansion https://www.propertyguri 1,750,000
10 24,540,532 582 Woodlands Drive 16 https://www.propertyguri 599,999
11 24,267,230 Atlassia https://www.propertyguri 2,721,408
12 24,144,885 Corals at Keppel Bay https://www.propertyguri 8,880,000
13 21,269,444 Visioncrest https://www.propertyguri 1,507,500
14 24,637,178 Meraprime https://www.propertyguri 2,100,000
15 24,688,888 Marina Bay Residences https://www.propertyguri 7,888,888
16 24,580,707 Large Freehold Land; Corn https://www.propertyguri 6,000,000
17 24,731,447 Skyline Residences https://www.propertyguri 1,125,000
18 24,626,137 Leedon Green https://www.propertyguri 1,668,000
19 24,708,053 Jalan Sejarah https://www.propertyguri 11,800,000
20 24,427,105 28 Bendemeer Road https://www.propertyguri 638,000
21 23,638,592 Skyline @ Orchard Bouleva https://www.propertyguri 63,000,000
22 24,576,030 St. Regis Residences Singa https://www.propertyguri 13,900,000
23 21,758,126 Parkview Eclat https://www.propertyguri 59,247,000
24 24,186,084 Freehold Corner Terrance (https://www.propertyguri 8,500,000
25 24,037,930 Dalvey Court https://www.propertyguri 5,143,200
Refresh
Save
Cancel
Export data
200
200+
SGT
en_GB
Writable
Smart Insert


DBeaver 23.2.0 - public
SQL
Commit
Rollback
Auto
oltp
public@oltp
Database N...
Projects
Enter a part of object name here
olap - localhost:5432
oltp - localhost:5432
postgres - localhost:5432
*<olap> Script
olap
logs
oltp
public
Properties
ER Diagram
oltp
Databases
oltp
Schemas
public
Name:
public
Namespace ID:
2200
Comment:
standard public schema
Owner:
pg_database_owner
Tables
Views
Materialized Views
Indexes
Functions
Sequences
Data types
Aggregate functions
Permissions
Source
Table Name
Object ID
Owner
Tables
locations
16,430
postgres
logs
16,423
postgres
Project - General
Name
DataSc
Bookmarks
Diagrams
Scripts
Save ...
Revert
Refresh
SGT
en_GB