Name: Medha Prodduturi

Table of Contents

Project Direction Overview

Purpose

Websites like Zillow and Redfin are great for finding apartments based on the different filters a user enters, but managing multiple listings across different platforms can be overwhelming. The Aptly database is designed for individuals actively searching for their next home, aiming to simplify the tracking and organization of potential rentals. Aptly will stand as a centralized platform which helps users do the following:

1. Organize Listings - Save and track apartments from various websites in one place.
2. Schedule and Manage Tours - Keep tours schedule organized and easily accessible for rescheduling, if needed.
3. Prioritize Your Search - Review and compare saved listings to make informed decisions.

Who is it for?

Aptly is primarily intended for individuals who are actively searching for places to rent/own. It will also be useful for real estate agents and property managers who want to track potential buyers and their interests.

Type of Data

The database will store information about apartments, including property details like address, size, number of bedrooms and bathrooms, rent price, amenities, and the rental agent's information. It will also store user information like name, email, address, preferred location, budget, and desired amenities. Aptly will track saves listings, including any comments and notes the user has, and schedules tours with respective dates and times.

Why I'm Interested?

I've personally experienced the frustration of apartment hunting. It's a time-consuming and often stressful process. I've had to juggle multiple websites, schedule countless tours, and keep track of endless details. I believe Aptly can alleviate these pain points by providing a centralized platform to manage apartment searches, schedule tours, and keep track of important information. By simplifying the process, Aptly can help people find their perfect home more efficiently.

Use Cases and Fields

Use Case #1 - User Creates an Account

1. User visits the app store/website to access Aptly application
2. Aptly asks the user to create an account
3. User enters the required information and submits the details.
4. The system will validate the information and create a new user account, storing the information in the database.

Field	Purpose of Field
UserID	A unique identifier assigned to each user for tracking and referencing them.
AccountID	A unique identifier assigned to each account.
FirstName	The first name of the account holder to be displayed on screens and for addressing them during any communication.
LastName	The last name of the account holder to be displayed on screens and for addressing them during any communication.
Email	The user's email address, used for login, password recovery, and communication.
Password	The user's encrypted password for security.
LocationPreference	The user's preferred location for apartment searches, such as city, state, or neighborhood.
Budget	The user's maximum budget for monthly rent.
Amenities	A list of amenities (in-unit laundry, gym, pool) as well as any other likes/wants of user.

Use Case #2 - User Saves a Listing

1. User finds an apartment on another website
2. User logs into Aptly/installs the browser plug-in for the application
3. User copies the listing URL into Aptly
4. User clicks on "Add Listing" and pastes the listing URL into field
5. Aptly will extract all relevant information
6. Aptly stores the listing information in the database

Field	Purpose of Field
ListingID	A unique identifier assigned for each saved listing.
UserID	The ID of the user who saved the listing.
URL	The URL of the original listing on the website the apartment was found.
Address	The full address of the apartment, including the street, city, state, and zip code.

AptRent	The monthly rent of the apartment.
Description	A list of amenities (in-unit laundry, gym, pool) offered by the apartment.
BedsBaths	The total number of bedrooms and bathrooms in the apartment.
SavedDate	The date the listing was added to the user's saved/liked list.

Use Case #3 - User Tracks Tour Information

1. User schedules a tour on the website where the apartment was found
2. User logs into Aptly
3. User selects a saved listing
4. User enters the scheduled tour date and time
5. System updated the listing information

Field	Purpose of Field
TourID	A unique identifier assigned for each tour.
ListingID	A unique identifier assigned for each saved listing.
TourDate	The date of the scheduled tour.
TourTime	The time of the scheduled tour.

Use Case #4 - User Reviews a Listing

1. User completed a tour or visits the apartment
2. User logs into Aptly
3. User selects the reviewed listing
4. User provides feedback under the "Notes/Comments" section
5. System updates the apartment's rating and reviews.

Field	Purpose of Field
ReviewID	A unique identifier assigned for each review.
ListingID	A unique identifier assigned for each saved listing.
UserID	The ID of the user who wrote the review.
UserRating	The user's rating of the apartment (e.g., 1-5 stars).
UserReview	The user's written review or comments about the apartment.

Use Case #5 - User Exports Saved Listings

1. User logs into Aptly
2. User clicks on “Export Listings”
3. Aptly generates a CSV or PDF file
4. Aptly prompts the user to download the file.
5. User downloads the file

Field	Purpose of Field
ExportID	A unique identifier assigned for each export done.
ListingID	A unique identifier assigned for each saved listing.
UserID	The ID of the user who saved the listing.
URL	The URL of the original listing on the website the apartment was found.
Address	The full address of the apartment, including the street, city, state, and zip code.
AptRent	The monthly rent of the apartment.
BedsBaths	The total number of bedrooms and bathrooms in the apartment.
Notes	A list of amenities (in-unit laundry, gym, pool) offered by the apartment. Any additional comments or notes made by the user about the apartment.
SavedDate	The date the listing was added to the user’s saved list.

Use Case #6 - User Marks Listing As Favorite

1. User logs into Aptly
2. The user views a listing on Aptly.
3. The user clicks the “Favorite” button on the listing.
4. The listing is saved to the user's favorite listings in the database.

Field	Purpose of Field
FavoriteID	A unique identifier assigned for each favorite listing.
UserID	The ID of the user who wrote the review. Foreign key linking to the user who favorited the listing.

ListingID	Foreign key linking to the specific listing marked as favorite.
FavoriteDate	Date when the user marked the listing as favorite.

Use Case #7 - User Filters Listing By Neighborhood

1. User logs into Aptly
2. User clicks on Filter option
3. The user searches for a listing within specific neighborhoods.
4. The system returns only listings within that neighborhood.

Field	Purpose of Field
NeighborhoodID	A unique identifier assigned to each neighborhood.
ListingID	Foreign key linking to the specific listing marked as favorite.
Name	Name of the neighborhood (e.g., “Downtown”, “North End”).
Description	Brief description of the neighborhood’s characteristics (e.g., “Residential area with parks and cafes”).
City	The city the neighborhood is located in, to support searches across cities.

Use Case #8 - Aptly Actively Tracks Changes in Listing

1. User logs into Aptly
2. A listing is updated within a link entered by the user.
3. The system records a new entry in the listing history.
4. The history provides a log of updates for each listing.

Field	Purpose of Field
HistoryID	A unique identifier assigned to each history record.
ListingID	Foreign key linking to the listing that had a change.
ChangeType	Type of change made (e.g., "Price Update", "Availability").
OldValue	Previous value before the change.

NewValue	New value after the change.
ChangedDate	Date the change was made.

Structural Database Rules

Associative Relationship Rules

1. Each user has one account; each account is associated with one user.
   - ➔ Explanation: To further explain this, each user MUST have a unique account, with attributes like UserID, FirstName, LastName, Email, Password, LocationPreference, Budget, and Amenities. This is a mandatory relationship since every user that wants to explore Aptly must create an account first.

2. Each user may save multiple listings; each listing is associated with one user.
   - ➔ Explanation: To further explain Rule #2, this relationship is optional for users (they don't HAVE TO save a listing) but mandatory for listings (each saved listing MUST belong to a user).

3. Each listing may have multiple tours scheduled; each tour is associated with one listing.
   - ➔ Explanation: This rule indicates that tours can be scheduled for saved listings. The relationship is optional for listings (they MAY NOT have any tours scheduled) but mandatory for each tour (a tour CANNOT exist without a listing to visit).

4a. Each user may leave reviews for multiple listings; each review is associated with one user.
4b. Each listing can have reviews and each review must be associated with a listing.
   - ➔ Explanation: To further explain Rule #4a, users MAY review listings they've saved. This relationship is optional for users. However, it is mandatory for reviews, as a review MUST belong to a user.

5a. A User can export multiple Listings in a single action, but each Export Action is associated with exactly one User.
5b. A Listing can be exported multiple times, but each export can only happen once.
   - ➔ Explanation: To further explain Rule #5a, a user can export multiple listings in a single export action, and each export action is uniquely tied to one user. This ensures that export actions are user-specific and can include one or more listings selected from the user's saved data.

6a. Each listing can be favored by multiple users; each favorite entry is associated with one listing.
6b. Each user can favor multiple listings; each favorite must be associated with one user.
   - ➔ Explanation: To further explain Rule #6a, a user CAN mark multiple listings as favorites, resulting in a one-to-many relationship. This relationship is mandatory for each favorite entry (it MUST be linked to a user) but optional for users, as they aren't required to favorite any listings.

7. Each listing is associated with one neighborhood; each neighborhood can contain multiple listings.

➔ Explanation: To further explain Rule #7, this relationship enables grouping of listings by neighborhood. It's mandatory for each listing (it MUST be located in a neighborhood) but optional for neighborhoods if there are no listings yet.

8. Each listing can have multiple history records; each history record is associated with one listing.
   ➔ Explanation: To further explain Rule #8, this relationship enables tracking of changes (price, availability) for each listing over time. This relationship is mandatory for each history record (it MUST refer to a listing) but optional for listings if no changes have been logged yet.

Specialization-Generalization Rules

1. A Tour is a scheduled tour, unscheduled tour, or a completed tour.
   - Generalization: Tour
   - Specialization: scheduled tour, unscheduled tour, or a completed tour.
   - Explanation: If the user wants to schedule a tour, it can be planned for a future date & time, making it scheduled. Once the tour has already taken place, the tour will be a completed tour. If the user doesn't schedule a tour, it is unscheduled. Thus, it cannot be several at the same time, which is why "several of these" or "both of these" is not used. Additionally, the relationship here is not partially complete since the list of subtypes is not exhaustive - the tour can only be either of the 3.

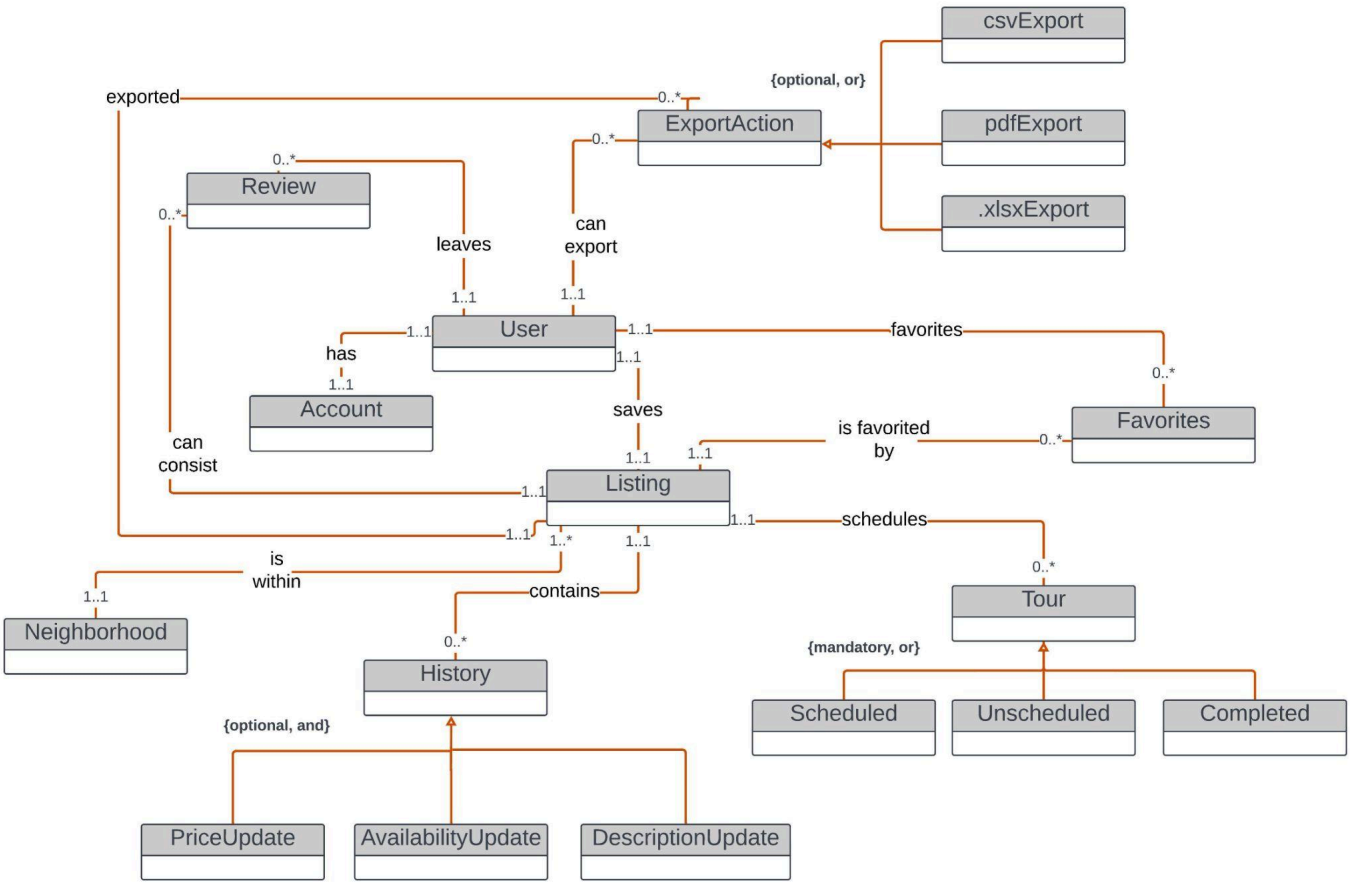
2. A Export can be a CSV Export, PDF Export, .xlsx Export, or none of these.
   - Generalization: Export Data
   - Specialization: CSV Export, PDF Export, .xlsx Export, or none of these
   - Explanation: When a user wants to export their listings data, they can choose to export in different formats (CSV, PDF, .xlsx) or other formats that are not identified (meaning, the list is not exhaustive and the relationship is partially complete). Additionally, the relationship is also disjoint, since one export cannot be multiple formats simultaneously.

3. A Change/Update can be price update, availability update, description update, several of these, or none of these.
   - Generalization: Listing History
   - Specialization: Price update, availability update, description update, several of these, or none of these.
   - Explanation: History updates can be a variety of things like changes in price, availability, or amenities/description of the listing. "Several of these" is used here because the relationship is overlapping, since an update can be a price and availability update at the same time for example. "None of these" is used since there could be other updates made that are not indicated here, so it is partially complete.

Conceptual Entity-Relationship Diagram

The conceptual ERD diagram I designed below provides a very high-level overview of the key entities, relationships, as well as the structural database rules defined above. Here, the primary entities defined are User, Account, Listing, Tour, Review, Neighborhood, Favorites, Export Action, and Listing History. The diagram illustrates the associative and specialization-generalization rules, displaying how each entity is

related as well as the main and subtypes of each rule. Not only does this capture the logical structure of the database to model user interactions but also portrays the application features in Aptly very clearly.



Full DBMS Physical ERD

Aptly Attributes:

User Table

Attribute	Datatype	Reasoning	Example Data
first_name	VARCHAR(255)	This is the first name of the account holder, up to 255 characters of the name.	Nick
last_name	VARCHAR(255)	This is the last name of the account holder, up to 255 characters of the name.	Miller

Account Table

username	VARCHAR(255)	This is the username of the account holder, up to 255 characters of the name.	nick_miller
email	VARCHAR(255)	This is the email of the account holder, up to 255 characters of the name.	nick.miller@gmail.com
password	VARCHAR(255)	Every account has a password stored in encrypted text format in the database. 255 characters is the limit.	@JessDayMiller079
created_date	DATE	The date the account is created on.	2023-08-25

Listing Table

listing_title	VARCHAR(255)	The title of the listing, up to 255 characters, for easy identification.	Luxury Apartment
bed_bath	VARCHAR(255)	The number of bedrooms and bathrooms in the apartment.	2B1B
location	VARCHAR(255)	The address of the apartment in the listing.	321 Monroe St, Chicago IL
listing_description	VARCHAR(255)	A brief description of the listing details including the amenities available.	The unit has a balcony with the apartment including gym, pool, and laundry unit.
listing_price	DECIMAL(12)	The price of the listing, stored as a decimal for accuracy.	1200.00
available_on	DATE	The date the listing becomes available for rent.	2025-01-01

listed_on	DATE	The date the listing was created in the system.	2024-11-20

Review Table

rating	DECIMAL(12)	The rating given to the listing, allowing decimals for half ratings.	4.5
review_comments	VARCHAR(255)	Comments added by the user about the listing.	Great location, close to the subway!
reviewed_date	DATE	The date the review was submitted.	2024-11-22

Export Table

file_format	VARCHAR(10)	Format of the exported file, limited to predefined options (CSV, PDF, XSLX.)	CSV
file_name	VARCHAR(255)	Name of the file exported, for user reference.	saved_listings.csv
exported_date	DATE	The date the export action was performed.	2024-11-23

Tour Table

tour_type	VARCHAR(64)	The type of tour (Scheduled, Completed, Unscheduled).	Scheduled
tour_feedback	VARCHAR(255)	Feedback from the user about the tour.	Landlord was great but the house needs a lot of renovation.

History Table

updated_type	VARCHAR(64)	Type of update made (Price, Availability, Description, etc.).	Price Update

updated_date	DATE	The date the update occurred.	2024-11-24

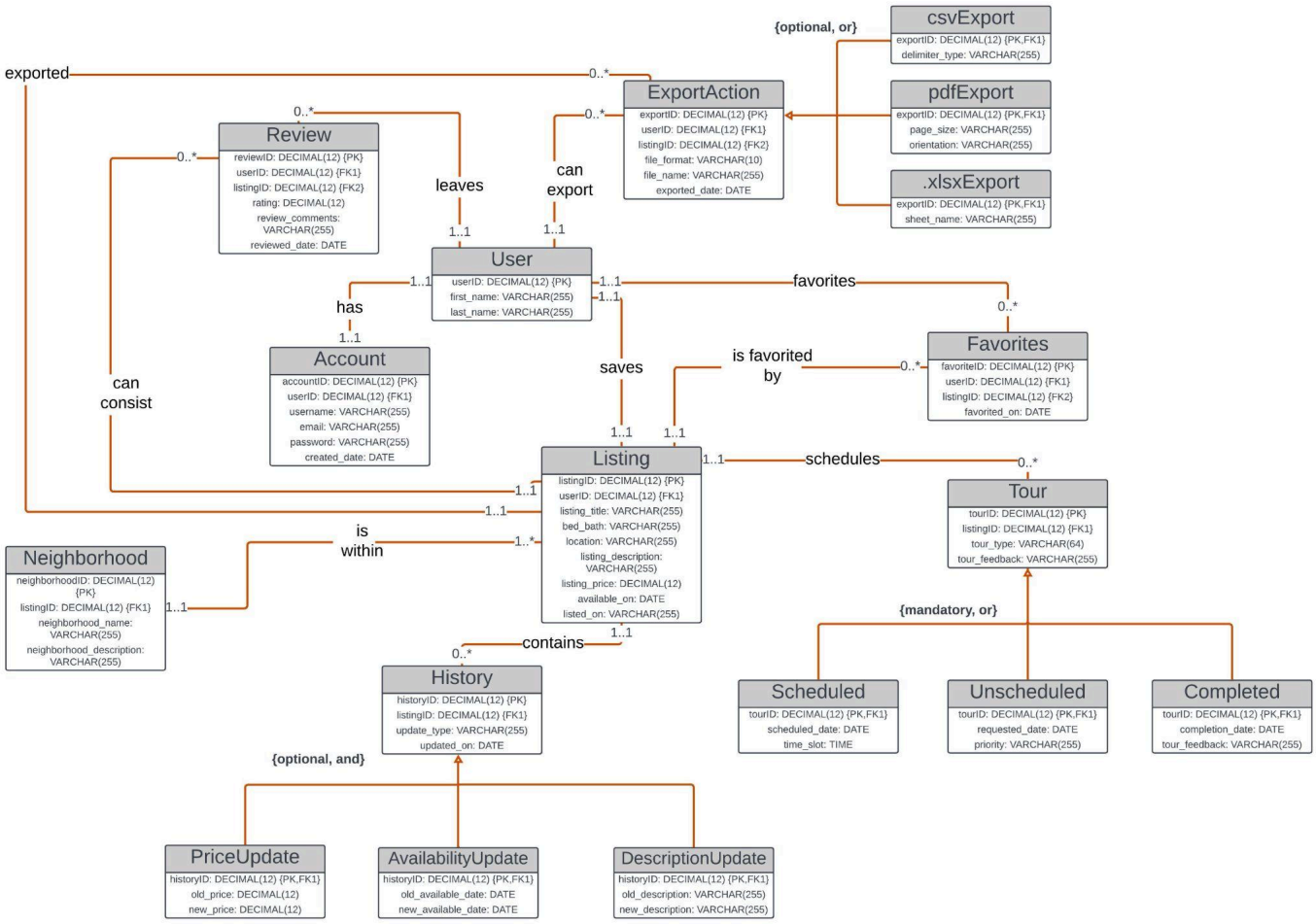
Neighborhood Table

neighborhood_name	VARCHAR(255)	Name of the neighborhood, up to 255 characters.	Manhattan 35th St.
neighborhood_description	VARCHAR(255)	Brief description of the neighborhood.	Closer to the tech buildings.

Favorites Table

favorited_on	DATE	The date the user marked the listing as a favorite.	2024-11-24

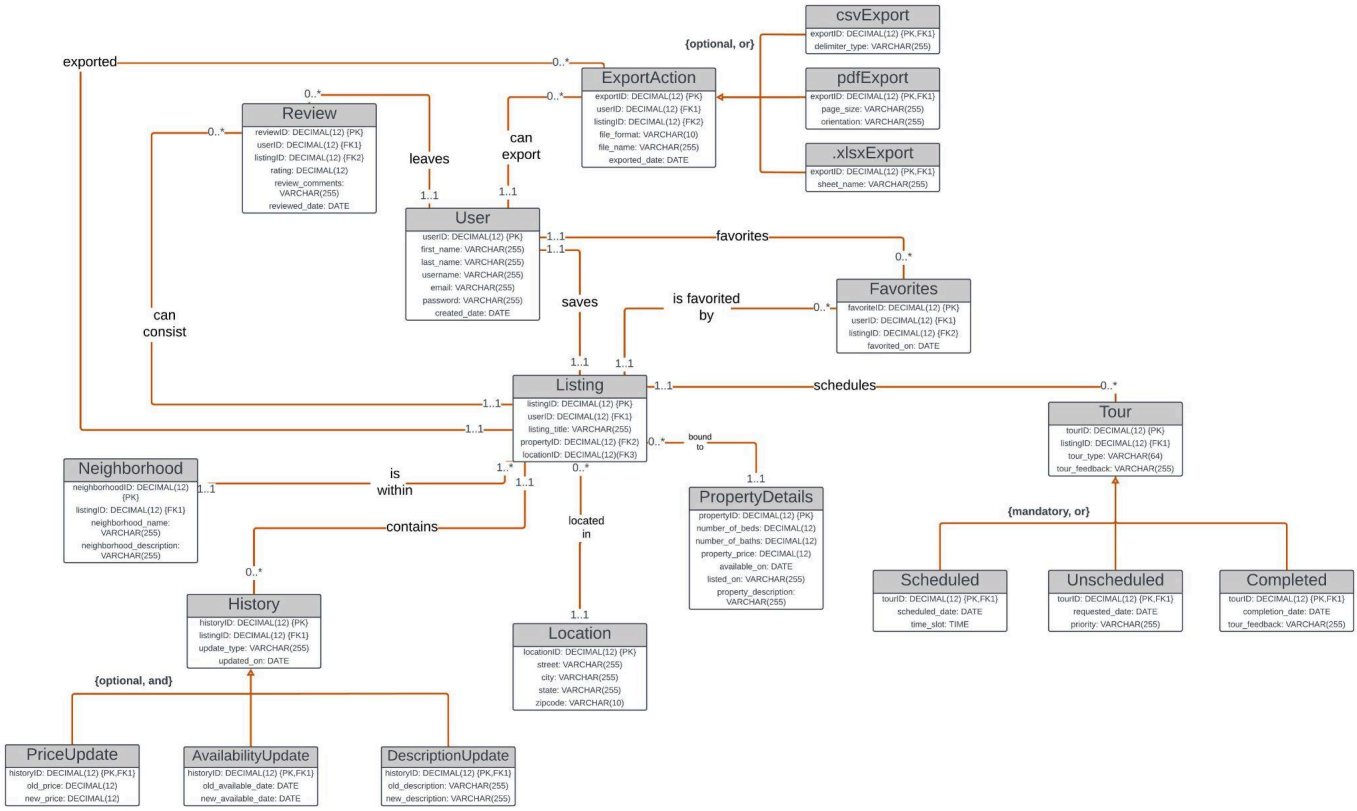
Based on the above tables, here's my ERD with Attributes included:



After building the Physical ERD with all attributes, I noticed two places where Normalization is still required.

1. Currently, the user and account table are separate. I can normalize this into one table since all the account information is directly related to the user and it is a one-to-one relationship.
2. The Listing table can also be normalized by separating the location and the different attributes describing the property into their own respective entities. This can avoid redundancy if these attributes are reused across multiple listings.

   By implementing 1 and 2, I get the below normalized Physical ERD:



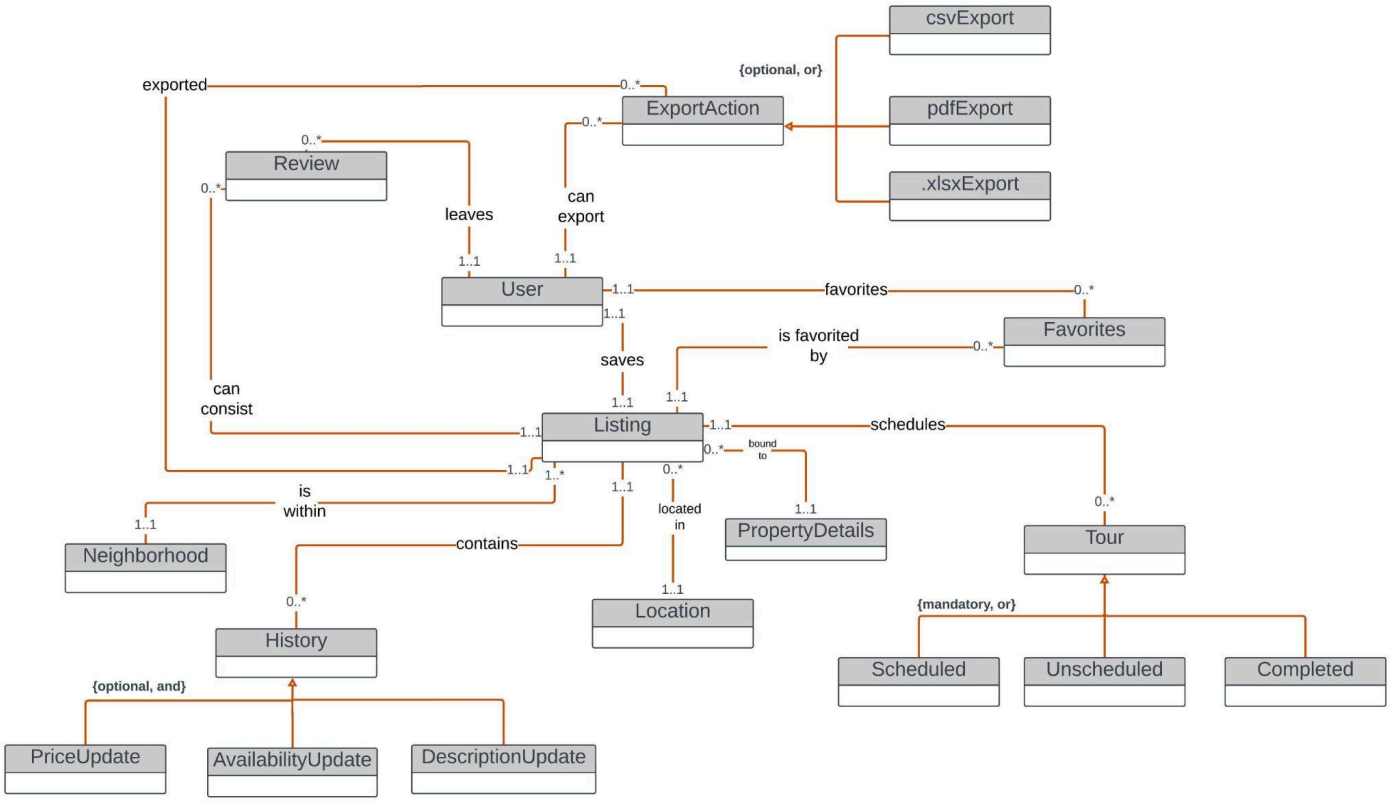
Specifically, for 2, I ended up with 2 extra entities that then give us 2 new associative relationships.

- Location Table - location_id (PK), address, city, state, zip code.
  - New Associative Relationship Rule: Each listing must have exactly one location and each location can be associated with multiple listings.
- Property Details Table - property_id (PK), number_of_beds, number_of_baths, property_price, available_on, listed_on, property_description.
  - New Associative Relationship Rule: Each listing must have exactly one property type and each property type can be associated with multiple listings.

Although the Location Table is not normalized to BCNF and I could create more tables to further breakdown to zipcode being national/international, street to its number, etc., I think this would

overcomplicate the database and is not necessary for Aptly to still perform all of its functionalities. However, the Property Details Table is normalized to BCNF.

Consistent to the normalized Physical ERD, my conceptual ERD is updated to the below:



Now that I have a complete physical ERD, I started creating my tables and sequences. I created sequences for all tables with primary keys (and left out those with solely foreign keys). Below is the query screenshot of creating sequences.

```
24  -- Creating Sequences
25  CREATE SEQUENCE user_seq START WITH 1;
26  CREATE SEQUENCE property_seq START WITH 1;
27  CREATE SEQUENCE location_seq START WITH 1;
28  CREATE SEQUENCE listing_seq START WITH 1;
29  CREATE SEQUENCE review_seq START WITH 1;
30  CREATE SEQUENCE favorites_seq START WITH 1;
31  CREATE SEQUENCE neighborhood_seq START WITH 1;
32  CREATE SEQUENCE export_seq START WITH 1;
33  CREATE SEQUENCE tour_seq START WITH 1;
34  CREATE SEQUENCE history_seq START WITH 1;
```

I continued to create the tables that are included in my ERD. Below is the screenshot of the UserInfo (User entity in diagram), Property_Details, LocationInfo (Location in ERD), and Listing tables.

```sql
--TABLES
CREATE TABLE UserInfo(
    userID DECIMAL(12) NOT NULL PRIMARY KEY,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_date DATE NOT NULL,
);

CREATE TABLE Property_Details(
    propertyID DECIMAL(12) NOT NULL PRIMARY KEY,
    number_of_beds DECIMAL(12) NOT NULL,
    number_of_baths DECIMAL(12) NOT NULL,
    property_description VARCHAR(255) NOT NULL,
    property_price DECIMAL(12) NOT NULL,
    available_on DATE NOT NULL,
    listed_on DATE NOT NULL,
);

CREATE TABLE LocationInfo(
    locationID DECIMAL(12) NOT NULL PRIMARY KEY,
    street VARCHAR(255) NOT NULL,
    city VARCHAR(255) NOT NULL,
    state VARCHAR(255) NOT NULL,
    zipcode VARCHAR(10) NOT NULL,
);

CREATE TABLE Listing(
    listingID DECIMAL(12) NOT NULL PRIMARY KEY,
    userID DECIMAL(12) NOT NULL,
    propertyID DECIMAL(12) NOT NULL,
    locationID DECIMAL(12) NOT NULL,
    listing_title VARCHAR(255) NOT NULL,
    FOREIGN KEY(userID) REFERENCES UserInfo(userID),
    FOREIGN KEY(propertyID) REFERENCES Property_Details(propertyID),
    FOREIGN KEY(locationID) REFERENCES LocationInfo(locationID)
);
```

Below is the screenshot of the Review, Favorites, and Neighborhood tables:

```sql
CREATE TABLE Review(
    reviewID DECIMAL(12) NOT NULL PRIMARY KEY,
    userID DECIMAL(12) NOT NULL,
    listingID DECIMAL(12) NOT NULL,
    rating DECIMAL(12) NOT NULL,
    review_comments VARCHAR(255),
    reviewed_on DATE NOT NULL,
    FOREIGN KEY(userID) REFERENCES UserInfo(userID),
    FOREIGN KEY(listingID) REFERENCES Listing(listingID)
);

CREATE TABLE Favorites(
    favoriteID DECIMAL(12) NOT NULL PRIMARY KEY,
    userID DECIMAL(12) NOT NULL,
    listingID DECIMAL(12) NOT NULL,
    favorited_on DATE NOT NULL,
    FOREIGN KEY(userID) REFERENCES UserInfo(userID),
    FOREIGN KEY(listingID) REFERENCES Listing(listingID)
);

CREATE TABLE Neighborhood(
    neighborhoodID DECIMAL(12) NOT NULL PRIMARY KEY,
    listingID DECIMAL(12) NOT NULL,
    neighborhood_name VARCHAR(255) NOT NULL,
    neighborhood_description VARCHAR(255) NOT NULL,
    FOREIGN KEY(listingID) REFERENCES Listing(listingID)
);
```

Below is the screenshot of the Export_Action entity along with all of its subtypes:

```sql
CREATE TABLE Export_Action(
    exportID DECIMAL(12) NOT NULL PRIMARY KEY,
    userID DECIMAL(12) NOT NULL,
    listingID DECIMAL(12) NOT NULL,
    files_format VARCHAR(10) NOT NULL CHECK (files_format IN ('CSV', 'PDF', 'Xlsx')),
    file_name VARCHAR(255) NOT NULL,
    exported_on DATE NOT NULL,
    FOREIGN KEY(userID) REFERENCES UserInfo(userID),
    FOREIGN KEY(listingID) REFERENCES Listing(listingID)
);

CREATE TABLE csvExport(
    exportID DECIMAL(12) NOT NULL PRIMARY KEY,
    delimited_type VARCHAR(255) NOT NULL,
    FOREIGN KEY(exportID) REFERENCES Export_Action(exportID)
);

CREATE TABLE pdfExport(
    exportID DECIMAL(12) NOT NULL PRIMARY KEY,
    page_size VARCHAR(255) NOT NULL,
    orientation VARCHAR(255),
    FOREIGN KEY(exportID) REFERENCES Export_Action(exportID)
);

CREATE TABLE xlsxExport(
    exportID DECIMAL(12) NOT NULL PRIMARY KEY,
    sheet_name VARCHAR(255),
    FOREIGN KEY(exportID) REFERENCES Export_Action(exportID)
);
```

Below is the screenshot of the Tour entity along with all of its subtypes:

```sql
CREATE TABLE Tour(
    tourID DECIMAL(12) NOT NULL PRIMARY KEY,
    listingID DECIMAL(12) NOT NULL,
    tour_type VARCHAR(64) NOT NULL CHECK (tour_type IN ('Scheduled', 'Unscheduled', 'Completed')),
    tour_feedback VARCHAR(255),
    FOREIGN KEY(listingID) REFERENCES Listing(listingID)
);

CREATE TABLE Scheduled(
    tourID DECIMAL(12) NOT NULL PRIMARY KEY,
    scheduled_on DATE NOT NULL,
    time_slot TIME NOT NULL,
    FOREIGN KEY(tourID) REFERENCES Tour(tourID)
);

CREATE TABLE Unscheduled(
    tourID DECIMAL(12) NOT NULL PRIMARY KEY,
    requested_date DATE NOT NULL,
    priority VARCHAR(255),
    FOREIGN KEY(tourID) REFERENCES Tour(tourID)
);

CREATE TABLE Completed(
    tourID DECIMAL(12) NOT NULL PRIMARY KEY,
    completion_date DATE NOT NULL,
    tour_feedback VARCHAR(255),
    FOREIGN KEY(tourID) REFERENCES Tour(tourID)
);
```

Below is the screenshot of the History entity along with all of its subtypes:

```sql
CREATE TABLE History(
    historyID DECIMAL(12) NOT NULL PRIMARY KEY,
    listingID DECIMAL(12) NOT NULL,
    update_type VARCHAR(255) NOT NULL CHECK (update_type IN ('Price Update', 'Availability Update', 'Description Update')),
    updated_on DATE NOT NULL,
    FOREIGN KEY(listingID) REFERENCES Listing(listingID)
);

CREATE TABLE PriceUpdate(
    historyID DECIMAL(12) NOT NULL PRIMARY KEY,
    old_price DECIMAL(12) NOT NULL,
    new_price DECIMAL(12) NOT NULL,
    FOREIGN KEY (historyID) REFERENCES History(historyID)
);

CREATE TABLE AvailabilityUpdate(
    historyID DECIMAL(12) NOT NULL PRIMARY KEY,
    old_availability_date DATE NOT NULL,
    new_availability_date DATE NOT NULL,
    FOREIGN KEY (historyID) REFERENCES History(historyID)
);

CREATE TABLE DescriptionUpdate(
    historyID DECIMAL(12) NOT NULL PRIMARY KEY,
    old_description VARCHAR(255) NOT NULL,
    new_description VARCHAR(255) NOT NULL,
    FOREIGN KEY (historyID) REFERENCES History(historyID)
);
```

Stored Procedure Execution and Explanations

For my first use case which describes the process of a user creating an account, I implemented a transaction using SQL Server. My stored procedure definition is as below:

```sql
CREATE PROCEDURE AddAccount @userID DECIMAL(12), @first_name VARCHAR(255), @last_name VARCHAR(255),
@username VARCHAR(255), @email VARCHAR(255), @password VARCHAR(255)
AS
BEGIN
    INSERT INTO UserInfo(userID, first_name, last_name, username, email, password, created_date)
    VALUES(@userID, @first_name,  @last_name, @username,@email, @password, GETDATE());
END;
GO
```

I named this procedure AddAccount since the user can only create one type of account. It's parameters correspond to the UserInfo table that I created previously. Since the created_date column is just the date the account was created (the current date basically), I do not need a parameter for that; I simply passed the GETDATE() function to automatically retrieve that information.

I added a couple rows of data in my stored procedure execution. Rather than hard coding values for userID column, I used the previously created sequence user_seq to attain these values.

```sql
DECLARE @current_user_seq INT = NEXT VALUE FOR user_seq;
BEGIN TRANSACTION AddAccount;
EXECUTE AddAccount @current_user_seq, 'Ted', 'Mosby', 'ted_mosby', 'ted.mosby@gmail.com', 'tmosby';
COMMIT TRANSACTION AddAccount;

DECLARE @new_user_seq DECIMAL(12);
SET @new_user_seq = NEXT VALUE FOR user_seq;
EXEC AddAccount @new_user_seq, 'Robin', 'Scherbatsky', 'robin_s', 'robin.scherbatsky@gmail.com', 'rscherbs';
```

I added fictional names Ted Mosby and Robin Scherbatsky that mirror realistic information. When I select and output everything from the UserInfo table following the creation of stored procedure and execution of it, the table looks like this:

```
242  SELECT * FROM UserInfo;
243
```

Results Messages

	userID	first_name	last_name	username	email	password	created_date
1	1	Ted	Mosby	ted_mosby	ted.mosby@gmail.com	tmosby	2024-12-02
2	2	Robin	Scherbatsky	robin_s	robin.scherbatsky@gmail.com	rscherbs	2024-12-02

My second stored procedure is created in correspondence to my second use case. When I performed normalization earlier, I separated the listing table into 2 different tables; one of which is the PropertyDetails table. When Aptly extracts relevant information from a listing, it will store details like the number of beds and baths, the price of the property, along with the description, etc in the PropertyDetails table.

My stored procedure for this looks like the below:

```
22  CREATE PROCEDURE AddPropertyDetails @propertyID DECIMAL(12), @number_of_beds DECIMAL(12), @number_of_baths DECIMAL(12),
23  @property_description VARCHAR(255), @property_price DECIMAL(12), @available_on DATE
24  AS
25  BEGIN
26      INSERT INTO Property_Details(propertyID, number_of_beds, number_of_baths, property_description, property_price, available_on, listed_on)
27      VALUES(@propertyID, @number_of_beds,  @number_of_baths, @property_description, @property_price, @available_on, GETDATE());
28  END;
29  GO
```

I added a couple rows of data for this stored procedure too. Rather than hard coding values for the propertyID column, I used the previously created sequence property_seq to attain these values.

```
31  DECLARE @current_property_seq INT = NEXT VALUE FOR property_seq;
32  BEGIN TRANSACTION AddPropertyDetails;
33  EXECUTE AddPropertyDetails @current_property_seq, 2, 2, 'Spacious 2b2b with attached balcony and amazing view!', 1299, '1/1/2025';
34  COMMIT TRANSACTION AddPropertyDetails;
35
36  DECLARE @new_property_seq DECIMAL(12);
37  SET @new_property_seq = NEXT VALUE FOR property_seq;
38  EXEC AddPropertyDetails @new_property_seq, 3, 2, 'Modern 3b2b with smart appliances', 2500, '1/15/2025';
```

When I select and output everything from the PropertyDetails table after my stored procedure creation and execution, the result is as below:

```
266  SELECT * FROM Property_Details;
267
```

Results Messages

	propertyID	number_of_beds	number_of_baths	property_description	property_price	available_on	listed_on
1	1	2	2	Spacious 2b2b with attached balcony and amazing view!	1299	2025-01-01	2024-12-02
2	2	3	2	Modern 3b2b with smart appliances	2500	2025-01-15	2024-12-02

Question Identification and Explanations

Question 1: What are the details of each apartment listing, including the associated user, location, and any reviews?

Explanation: This query helps the organization analyze how different apartments are performing based on user feedback (ratings and comments) and their locations. By linking apartments with their users and reviews, we can assess which listings are most popular, identify trends in user satisfaction, and make data-driven decisions to improve the rental experience.

Question 2: What updates have been made to a specific listing, including changes in price, availability, or description?

Explanation: This query is crucial for tracking the history of modifications made to each apartment listing. By keeping track of updates (such as price changes, availability adjustments, or new descriptions), we can monitor how listings evolve over time. This can help in understanding trends in rental prices, the impact of availability changes, and whether the descriptions accurately reflect the apartment's features. This also supports transparency and accountability in the system.

Question 3: Which neighborhoods have the most apartment listings and what are the average prices for those neighborhoods?

Explanation: This query provides a high-level view of how apartment listings are distributed across neighborhoods. By focusing on the total number of listings and the average price, the company can identify which neighborhoods are more popular or have higher rental activity. Additionally, we can assess if there are any recent feedback trends that might impact decisions related to marketing, pricing strategies, or improving apartment features. This summary view is useful for targeted decision-making based on location-specific data.

Query Executions and Explanations

Query 1

For the first query, the requirement is to retrieve information from at least 4 tables joined by my associative relationships. Here, I felt it would be most useful to retrieve a list of apartments with their associated user, location details, and any rating/notes the user left for that listing. It would allow the user to view detailed information about each apartment, including who listed it, its location, price and address, and feedback.

I combined UserInfo, Listing, Porperty_Details, LocationInfo, and Review tables for this query:

```
--1st query
SELECT CONCAT(u.first_name, ' ', u.last_name) AS 'Full Name',
li.listing_title AS 'Listing Title',
CONCAT(p.number_of_beds, 'B', p.number_of_baths, 'B') AS 'Number of Beds & Baths',
CONCAT('$', p.property_price) AS 'Apartment Rent',
p.available_on AS 'Move-In Date',
CONCAT(lo.street, ', ', lo.city, ' ' , lo.state, ' ', lo.zipcode) AS 'Property Address',
r.rating AS 'Apartment Rating', r.review_comments AS 'User Notes'
FROM UserInfo u
JOIN Listing li ON u.userID = li.userID
JOIN Property_Details p ON li.propertyID = p.propertyID
JOIN LocationInfo lo ON li.locationID = lo.locationID
JOIN Review r ON li.listingID = r.listingID;
```

With the 2 rows of data I have thus far included, my output looks something like this:

Results Messages

	Full Name	Listing Title	Number of Beds & Baths	Apartment Rent	Move-In Date	Property Address	Apartment Rating	User Notes
1	Ted Mosby	Beautiful 2B2B in Boston	2B2B	$1299	2025-01-01	123 Elm St, Boston MA 02115	5	Amazing apartment with great views!
2	Robin Scherbatsky	Spacious 3B2B in Cambridge	3B2B	$2500	2025-01-15	456 Maple St, Cambridge MA 02139	4	Nice place but a bit pricey.

I combined the address (the street, city, state and zipcode) columns into 1 column called Property Address for a more concise view. The same applies for the Number of Beds & Baths column.

Query 2

For this query, since the requirement is to retrieve information from the subtypes and supertypes, I chose my History entity supertype and the AvailabilityUpdate and DescriptionUpdate subtypes. I felt this is the most useful query since it helps identify all updates made to each specific listing.

This query tracks the history of changes made to apartment lists, allowing users to understand how properties evolved over time. Price adjustments, availability modifications or description updates are crucial for users to track trends (like price hikes) and understand the timeline of each property.

It of course provides a sense of transparency between the tenant and landlord and can allow the apartment seeker to predict long-term trends or future changes that could occur.

For this query, I was able to join the History, AvailabilityUpdate, DescriptionUpdate, Listing, and LocationInfo Tables:

```sql
SELECT CONCAT(lo.street, ', ', lo.city, ' ' , lo.state, ' ', lo.zipcode) AS 'Property Address',
       li.listing_title AS 'Listing Title',
       CASE
           WHEN a.historyID IS NOT NULL THEN CONCAT(a.old_availability_date, ' changed to ', a.new_availability_date)
           ELSE 'No availability update'
       END AS 'Updated Availability Date',
       CASE
           WHEN d.historyID IS NOT NULL THEN CONCAT(d.old_description, ' changed to ', d.new_description)
           ELSE 'No description update'
       END AS 'Updated Description',
       h.updated_on AS 'Date Updated'
FROM History h
LEFT JOIN AvailabilityUpdate a ON h.historyID = a.historyID
LEFT JOIN DescriptionUpdate d ON h.historyID = d.historyID
JOIN Listing li ON h.listingID = li.listingID
JOIN LocationInfo lo ON li.locationID = lo.locationID;
```

With the 2 rows of data I have thus far included, my output looks something like this:

Results Messages

	Property Address	Listing Title	Updated Availability Date	Updated Description	Date Updated
1	123 Elm St, Boston MA 02115	Beautiful 2B2B in Boston	No availability update	No description update	2024-12-02
2	456 Maple St, Cambridge MA 02139	Spacious 3B2B in Cambridge	2025-01-01 changed to 2025-01-15	No description update	2024-12-02
3	456 Maple St, Cambridge MA 02139	Spacious 3B2B in Cambridge	No availability update	Modern 3b2b with smart appliances changed to Newly …	2024-12-02

Query 3

For the third query the requirement is to create a view and choose at least one from each group provided. From Group 1, I chose to use the ORDER BY statement. From Group 2, I chose to implement at least one aggregate function and join 4 or more tables.

My view summarizes the listings per neighborhood, finding the total listings, the average rent within that neighborhood. This query is useful because it can help users identify popular/high-demand areas based on the number of listings and average price in that area. Based on the budget of the user, they can decide whether it falls within that average rent and filter neighborhoods to target more budget-conscious areas. This can also be used by real-estate professionals to identify which areas are performing well.

For this query, however, I felt the data I had so far was not enough. Therefore, I added a few more rows of data into the UserInfo, LocationInfo, Property_Details, Listing, and Neighborhood tables. After I inserted this data using sequences and INSERT statements, the output of all the tables looked something like this:

```sql
SELECT * FROM Neighborhood;
SELECT * FROM Listing;
SELECT * FROM UserInfo;
SELECT * FROM Property_Details;
SELECT * FROM LocationInfo;
```

Results Messages

	neighborhoodID	listingID	neighborhood_name	neighborhood_description
1	1	1	Back Bay	Upscale area with historic charm and trendy shops.
2	2	2	Harvard Square	Bustling neighborhood with lots of cafes and bookst…
3	3	3	Back Bay	Upscale area with historic charm and trendy shops.
4	4	4	Harvard Square	Bustling neighborhood with lots of cafes and bookst…
5	5	5	Manhattan – Midtown	Bustling neighborhood with iconic attractions and u…

	listingID	userID	propertyID	locationID	listing_title
1	1	1	1	1	Beautiful 2B2B in Boston
2	2	2	2	2	Spacious 3B2B in Cambridge
3	3	3	3	3	Parkside 3B2B in Boston
4	4	4	4	4	Studio in Cambridge near Downtown
5	5	5	5	5	Legendary Penthouse in NYC

	userID	first_name	last_name	username	email	password	created_date
1	1	Ted	Mosby	ted_mosby	ted.mosby@gmail.com	tmosby	2024-12-02
2	2	Robin	Scherbatsky	robin_s	robin.scherbatsky@gmail.com	rscherbs	2024-12-02
3	3	Marshall	Eriksen	marshall_e	marshall.eriksen@gmail.com	me123	2024-12-02
4	4	Lily	Aldrin	lily_a	lily.aldrin@gmail.com	la123	2024-12-02
5	5	Barney	Stinson	barney_s	barney.awesome@gmail.com	suits123	2024-12-02

	propertyID	number_of_beds	number_of_baths	property_description	property_price	available_on	listed_on
1	1	2	2	Spacious 2b2b with attached balcony and amazing vie…	1299	2025-01-01	2024-12-02
2	2	3	2	Modern 3b2b with smart appliances	2500	2025-01-15	2024-12-02
3	3	3	2	Spacious 3-bedroom apartment with park view	2100	2025-02-01	2024-12-02
4	4	1	1	Modern studio apartment near downtown	1500	2025-01-10	2024-12-02
5	5	2	2	Luxurious penthouse with city skyline views	5000	2025-03-01	2024-12-02

	locationID	street	city	state	zipcode
1	1	123 Elm St	Boston	MA	02115
2	2	456 Maple St	Cambridge	MA	02139
3	3	789 Pine St	Boston	MA	02130
4	4	101 Walnut Ave	Cambridge	MA	02139
5	5	1 Legendary Ave	New York	NY	10001

Now that I added the data, I created my view:

```sql
CREATE VIEW NeighborhoodSummary AS
SELECT
    lo.city AS City,
    n.neighborhood_name AS NeighborhoodName,
    COUNT(DISTINCT li.listingID) AS TotalListings,
    AVG(p.property_price) AS AveragePrice
FROM Listing li
JOIN LocationInfo lo ON li.locationID = lo.locationID
JOIN Property_Details p ON li.propertyID = p.propertyID
JOIN Neighborhood n ON li.listingID = n.listingID
GROUP BY lo.city, n.neighborhood_name;
```

With a SELECT statement, I was able to retrieve the information that was required:

```sql
SELECT
    City,
    NeighborhoodName AS 'Neighborhood Name',
    TotalListings AS 'Total Number of Listings',
    CONCAT('$', FORMAT(AveragePrice, 'N2')) AS 'Average Rent'
FROM NeighborhoodSummary
ORDER BY TotalListings DESC;
```

The output of the above view and SELECT statement was as below:

Results Messages

	City	Neighborhood Name	Total Number of Listings	Average Rent
1	Boston	Back Bay	2	$1,699.50
2	Cambridge	Harvard Square	2	$2,000.00
3	New York	Manhattan – Midtown	1	$5,000.00

Index Identification and Creations

Primary Key Indexes

Since modern databases automatically index primary key columns, here is my list of primary keys:

- UserInfo.userID
- Property_Details.propertyID
- LocationInfo.locationID
- Listing.listingID
- Review.reviewID
- Favorites.favorieID
- Neighborhood.neighborhoodID
- Export_Action.exportID
- Tour.tourID
- History.historyID

Foreign Key Indexes

As for the foreign keys, all of them need an index. Below is my table identifying each foreign key column, whether or not the index is unique, and why so.

Column	Unique?	Description
Listing.userID	Not unique	The foreign key in Listing referencing user is not unique because there can be many listings from the same user.
Listing.propertyID	Not unique	The foreign key in Listing referencing details of the property is not unique because there can be many listings with the same property details (since the details are basic - like price,

		availability, etc).
Listing.locationID	Not unique	The foreign key in Listing referencing the location is not unique because there can be many listings belonging to the same address (multiple flats within the same apartment).
Review.userID	Not unique	The foreign key in Review referencing the user is not unique because a user can leave multiple reviews for different listings.
Review.listingID	Not unique	The foreign key in Review referencing Listing is not unique because a listing can have multiple reviews.
Export_Action.userID	Not unique	The foreign key in Export_Action referencing userInfo is not unique because a single user can export multiple listings.
Export_Action.listingID	Not unique	The foreign key in Export_Action referencing Listing is not unique because a single listing can be exported many times.
Favorites.userID	Not unique	The foreign key in Favorites referencing UserInfo is not unique because a single user can have multiple favorite listings.
Favorites.listingID	Not unique	The foreign key in Favorites referencing Listing is not unique because a single listing can be favorited by multiple users.
Tour.listingID	Not unique	The foreign key in Tour referencing Listing is not unique because a single listing can be toured multiple times.
History.listingID	Not unique	The foreign key in History referencing Listing is not unique because a single listing can be

		updated multiple times.
Neighborhood.listingID	Not unique	The foreign key in Neighborhood referencing listing is not unique because a single listing can belong to multiple neighborhoods.

Although in the above table, all foreign keys being non-unique is straight-forward, I did want to further explain why Neighborhood.listingID is categorized as not unique here. In a typical apartment listing scenario, a listing would generally belong to only one neighborhood. However, I want Aptly to cover as many possible situations as possible. Meaning, there could be a situation where in some cities, neighborhoods may overlap in their borders (cross-boundary neighborhoods), so a listing could be considered part of multiple neighborhoods. Additionally, sometimes, a listing may be categorized under more than one neighborhood for marketing or classification reasons. For example, an apartment in a popular district could be listed in multiple neighborhood categories like "Downtown" and "Urban."

3 Query Driven Indexes

In terms of the 3 query driven indexes, I was able to identify 3 non-primary and non-foreign query-driven indexes.

The first one is **Listing.listing_title**. My reasoning for this is that it's common for queries to filter apartments based on their listing title, especially when searching or filtering by specific apartment names or keywords within the title. As a result, indexing the listing_title column will improve query performance when searching for listings by the name or part of the title. This would be a non-unique index because multiple listings could have similar or identical titles.

The next one is **Property_Details.property_price**. Many queries will likely filter or aggregate listings based on their price (ex. finding apartments within a certain price range). Since price is a common filtering criterion, indexing property_price can speed up queries that involve filtering or sorting by price. This would also be a non-unique index because multiple listings can share the same price.

My last one would be **Neighborhood.neighborhood_name**. Since neighborhoods are often used as a key criterion for filtering or grouping listings in queries (ex. showing all apartments in a particular neighborhood or avoiding a certain neighborhood for crime/safety purposes), indexing neighborhood_name could help improve the performance of such queries. This would be a non-unique index because multiple lists can belong to the same neighborhood, meaning the neighborhood name may repeat.

Index Creation Queries

Post identification of all columns that need indexes, I created them accordingly. Below are the queries for the same:

```sql
--Creating Indexes
CREATE INDEX ListingUserIdx
ON Listing(userID);

CREATE INDEX ListingPropertyIdx
ON Listing(propertyID);

CREATE INDEX ListingLocationIdx
ON Listing(locationID);

CREATE INDEX ReviewUserIdx
ON Review(userID);

CREATE INDEX ReviewListingIdx
ON Review(listingID);


CREATE INDEX Export_ActionUserIdx
ON Export_Action(userID);

CREATE INDEX Export_ActionListingIdx
ON Export_Action(listingID);

CREATE INDEX FavoritesUserIdx
ON Favorites(userID);

CREATE INDEX FavoritesListingIdx
ON Favorites(listingID);

CREATE INDEX TourListingIdx
ON Tour(listingID);

CREATE INDEX HistoryListingIdx
ON History(listingID);

CREATE INDEX NeighborhoodListingIdx
ON Neighborhood(listingID);
```

The below queries are for query-driven indexes:

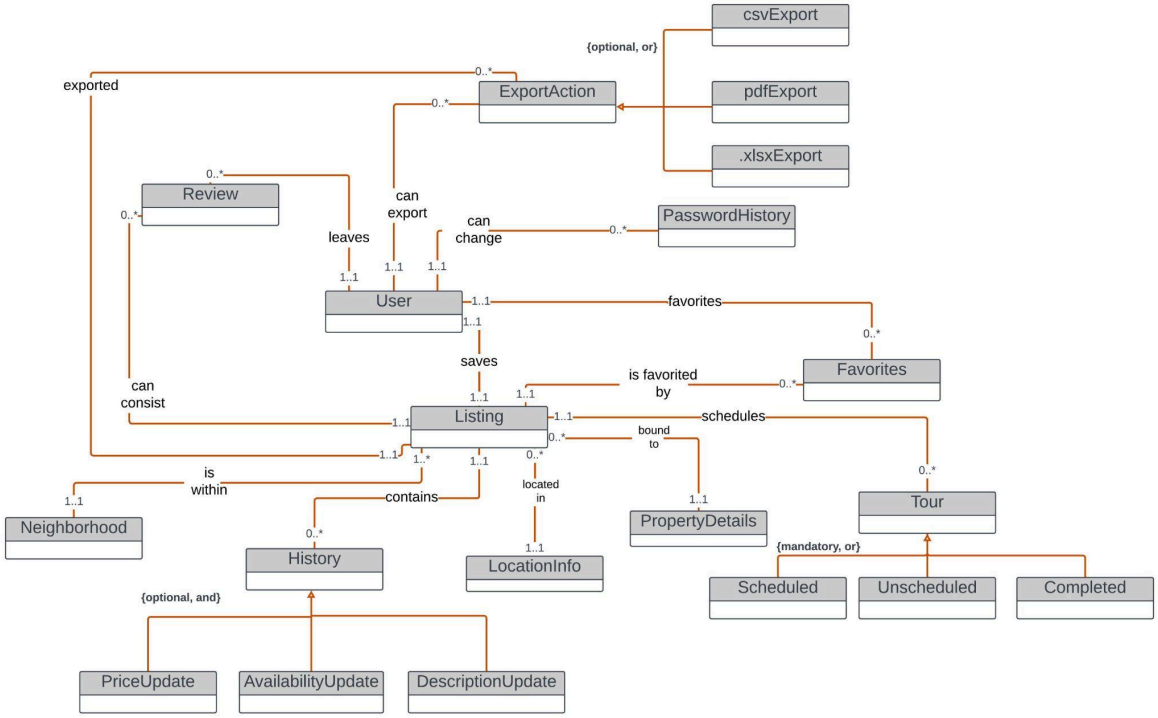
```sql
CREATE INDEX ListingTitleIdx
ON Listing(listing_title);

CREATE INDEX PropertyPriceIdx
ON Property_Details(property_price);

CREATE INDEX NeighborhoodNameIdx
ON Neighborhood(neighborhood_name);
```

History Table Demonstration

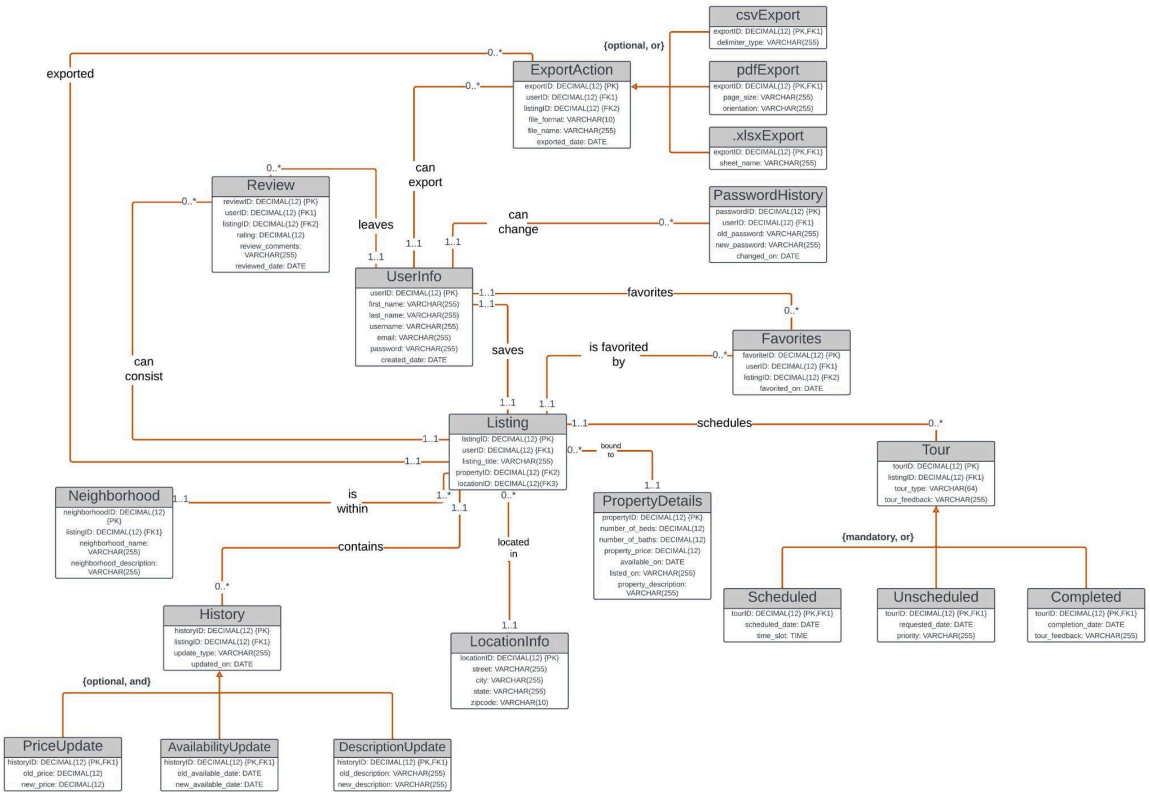
After reviewing all my existing tables, I believe my database so far does a good job of tracking historical values. Main entities like price, description, and availability updates are historically recorded. However, I want to track historical updates for user passwords within the UserInfo table. This information could be useful for security purposes, auditing, or user support. Password changes are frequent events that may require historical tracking to analyze account access patterns or revert unwanted changes.

So, my **new structural database rule** would be: Each user account (UserInfo) may have many password changes, but each password change in PasswordHistory is associated with exactly one user account.

My updated **conceptual ERD** is below:



My updated **physical ERD** is below:



My passwordHistory entity is present and linked to the UserInfo entity.Below are the attributes I added and why:

Attribute	Description	Example Data
passwordID	This is the primary key of the password history table. It is a DECIMAL(12) to allow for many values.	1
userID	This is a foreign key to the UserInfo table, a reference to the user that had the change in password.	1
old_password	This is the password of the user before the change. The datatype mirrors the password datatype in the UserInfo table.	tmosby
new_password	This is the password of the user after the change. The datatype mirrors the password datatype in the UserInfo table.	tmosby_nyc
changed_on	This is the date the password change occurred, with a DATE datatype.	11/15/2024

Here is a screenshot of my table and sequence creation, which has all of the same attributes and datatypes as indicated in the DBMS physical ERD:

```
5   CREATE TABLE passwordHistory(
6       passwordID DECIMAL(12) NOT NULL PRIMARY KEY,
7       userID DECIMAL(12) NOT NULL,
8       old_password VARCHAR(255) NOT NULL,
9       new_password VARCHAR(255) NOT NULL,
10      changed_on DATE NOT NULL,
11      FOREIGN KEY(userID) REFERENCES UserInfo(userID),
12  );
13
14  CREATE SEQUENCE password_seq START WITH 1;
```

Here is a screenshot of my trigger creation which will maintain the passwordHistory table:

```
5   CREATE OR ALTER Trigger passwordChangeTrigger
6   ON UserInfo
7   AFTER UPDATE
8   AS
9   BEGIN
10      DECLARE @old_password VARCHAR(255) = (SELECT password FROM DELETED)
11      DECLARE @new_password VARCHAR(255) = (SELECT password FROM INSERTED)
12
13      IF (@old_password <> @new_password)
14      INSERT INTO passwordHistory(passwordID, userID, old_password, new_password, changed_on)
15      VALUES(NEXT VALUE FOR password_seq, (SELECT userID FROM INSERTED), @old_password, @new_password, GETDATE());
16  END;
```

To further explain the trigger I implemented, it is meant to check if the old password and new password are the same. If a user is trying to change their password, it is required that their new password is not the same as the old password. To explain the trigger line by line:

`CREATE OR ALTER Trigger passwordChangeTrigger ON UserInfo AFTER UPDATE`	This starts the definition of the trigger and names it “passwordChangeTrigger”. The trigger is linked to the UserInfo table, and is executed after any update to that table.
`AS BEGIN`	This syntax starts the trigger block.
`DECLARE @old_password VARCHAR(255) = (SELECT password FROM DELETED)` `DECLARE @new_password VARCHAR(255) = (SELECT password FROM INSERTED)`	This saves the old and new passwords by referencing the DELETED and INSERTED pseudo tables, respectively.
`IF (@old_password <> @new_password)`	This check ensures action is only taken if the password has been updated.
`INSERT INTO passwordHistory(passwordID, userID, old_password, new_password, changed_on)` `VALUES(NEXT VALUE FOR password_seq, (SELECT userID FROM INSERTED), @old_password, @new_password, GETDATE());`	This inserts the record into the passwordHistroy table. The primary key is set by using the password_seq sequence. The old and new passwords are used from the variables. The userID is obtained from the INSERTED pseudo table. The date of the change is obtained by using the built-in GETDATE function.
`END;`	This ends the trigger definition.

To confirm my trigger maintains the password history table, I first took a look at the data in my UserInfo table:

Results Messages

	userID	first_name	last_name	username	email	password	created_date
1	1	Ted	Mosby	ted_mosby	ted.mosby@gmail.com	tmosby	2024-12-09
2	2	Robin	Scherbatsky	robin_s	robin.scherbatsky@gmail.com	rscherbs	2024-12-09
3	3	Marshall	Eriksen	marshall_e	marshall.eriksen@gmail.com	me123	2024-12-09
4	4	Lily	Aldrin	lily_a	lily.aldrin@gmail.com	la123	2024-12-09
5	5	Barney	Stinson	barney_s	barney.awesome@gmail.com	suits123	2024-12-09

If Ted Mosby's password is updated, we can run the following UPDATE SET statements:

```
2   UPDATE UserInfo
3   SET password = 'tmosby_nyc'
4   WHERE userID = 1;
5
6   UPDATE UserInfo
7   SET password = 'tmosby_nyc_123'
8   WHERE userID = 1;
```

Last, I verify that the passwordHistory table has a record of these password changes in the screenshot below:

Results Messages

	passwordID	userID	old_password	new_password	changed_on
1	1	1	tmosby	tmosby_nyc	2024-12-09
2	2	1	tmosby_nyc	tmosby_nyc_123	2024-12-09

When the update is made successfully, both the UserInfo table and the passwordHistory table have changed:

```
2   SELECT * FROM UserInfo;
3   SELECT * FROM passwordHistory;
```

This query gives us this appropriate result, changing Ted Mosby's password in the UserInfo table as well.

Results Messages

	userID	first_name	last_name	username	email	password	created_date
1	1	Ted	Mosby	ted_mosby	ted.mosby@gmail.com	tmosby_nyc_123	2024-12-09
2	2	Robin	Scherbatsky	robin_s	robin.scherbatsky@gmail.com	rscherbs	2024-12-09
3	3	Marshall	Eriksen	marshall_e	marshall.eriksen@gmail.com	me123	2024-12-09
4	4	Lily	Aldrin	lily_a	lily.aldrin@gmail.com	la123	2024-12-09
5	5	Barney	Stinson	barney_s	barney.awesome@gmail.com	suits123	2024-12-09

	passwordID	userID	old_password	new_password	changed_on
1	1	1	tmosby	tmosby_nyc	2024-12-09
2	2	1	tmosby_nyc	tmosby_nyc_123	2024-12-09

Data Visualizations

Create two visualizations of the data in your database using charts, graphs, or other visualizations. Clearly explain the data story conveyed by each visualization. Ensure that the visualizations and data stories are useful and appropriate given the intended use of your database.

1. Bar Graph: Neighborhood vs. Average Rent

For my first visualization, I thought Query #3 (after slight modification) would convey important information. For the sake of the visualization, I added more data into my LocationInfo, Property_Details, Listing, and Neighborhood tables. After doing so, I created a view called NeighborhoodSummary2 to retrieve the name of the neighborhood and calculate the average rent.

Here's my query:

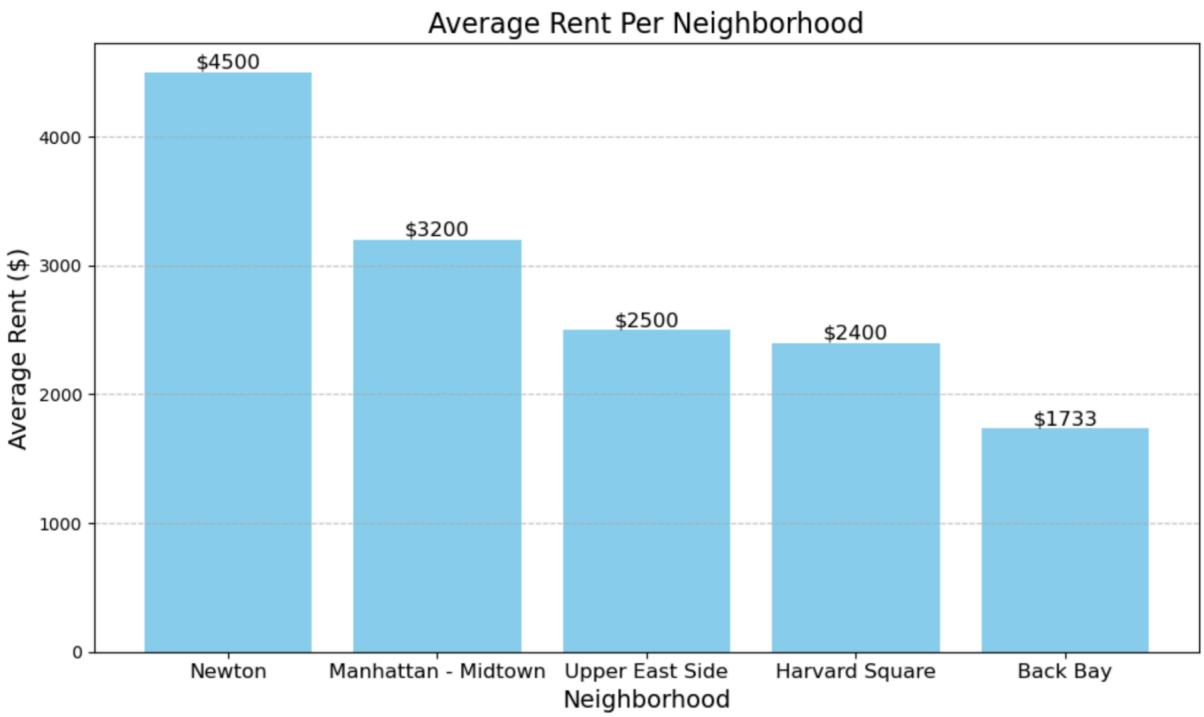
```
77  CREATE VIEW NeighborhoodSummary2 AS
78  SELECT
79      n.neighborhood_name AS NeighborhoodName,
80      AVG(p.property_price) AS AverageRent
81  FROM Listing li
82  JOIN Property_Details p ON li.propertyID = p.propertyID
83  JOIN Neighborhood n ON li.listingID = n.listingID
84  GROUP BY n.neighborhood_name;
85
86  SELECT
87      NeighborhoodName AS 'Neighborhood Name',
88      CONCAT('$', FORMAT(AverageRent, 'N2')) AS 'Average Rent'
89  FROM NeighborhoodSummary2
90  ORDER BY AverageRent DESC;
```

This query outputted the below result:

Results Messages

	Neighborhood Name	Average Rent
1	Newton	$4,500.00
2	Manhattan – Midtown	$3,200.00
3	Upper East Side	$2,500.00
4	Harvard Square	$2,400.00
5	Back Bay	$1,733.00

Based on my result, I created the below Bar Graph:



This bar graph conveys the story of how rent prices vary across different neighborhoods, providing users with a clear overview of the rental market in their city. It shows the average rent in each neighborhood, allowing users to compare the cost of living in each area. By presenting this data visually, the graph helps users make informed decisions about where they might want to search for apartments based on their budget. If a user is looking for more affordable options, they can easily identify the neighborhoods with lower average rents, while those seeking more premium areas can target neighborhoods with higher rent averages.

2. Pie Chart: Number of Ratings vs Apartment

For my second visualization, I added more data into the Review table. After doing so, I was able to simply query the data using a SELECT statement:

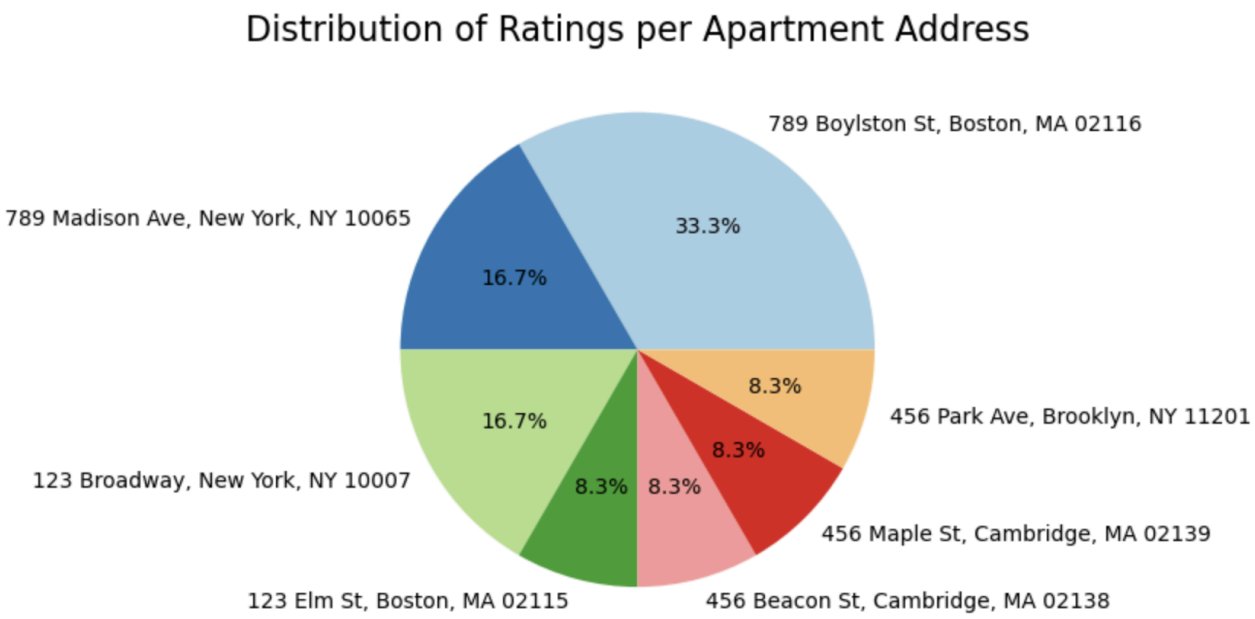
```
60  SELECT
61      CONCAT(lo.street, ', ', lo.city, ', ', lo.state, ' ', lo.zipcode) AS 'Full Address',
62      COUNT(r.reviewID) AS NumberOfRatings
63  FROM Review r
64  JOIN Listing l ON r.listingID = l.listingID
65  JOIN LocationInfo lo ON l.locationID = lo.locationID
66  GROUP BY lo.street, lo.city, lo.state, lo.zipcode
67  ORDER BY NumberOfRatings DESC;
```

This query gave the below result:

Results Messages

	Full Address	NumberOfRatings
1	789 Boylston St, Boston, MA 02116	4
2	789 Madison Ave, New York, NY 10065	2
3	123 Broadway, New York, NY 10007	2
4	123 Elm St, Boston, MA 02115	1
5	456 Beacon St, Cambridge, MA 02138	1
6	456 Maple St, Cambridge, MA 02139	1
7	456 Park Ave, Brooklyn, NY 11201	1

Based on my result, I created the below Pie Chart:



This pie chart tells the story of how user feedback is distributed across various apartment addresses, highlighting the popularity or trustworthiness of different properties. A larger portion of the pie chart for a specific address indicates that the apartment has received more reviews, which might suggest it is more popular or that people have had more experiences with that listing. Conversely, smaller slices indicate properties with fewer reviews, which could signal a newer listing or less engagement from renters.

Both visualizations are tailored to Aptly's objective of helping users effectively track, compare, and evaluate apartments, giving them the tools to make smarter, data-driven decisions while apartment hunting. The bar graph provides insights into rental affordability, while the pie chart helps users assess the quality and reliability of listings based on user feedback.

Summary and Reflection

Aptly is an apartment hunting tracker designed to help users efficiently manage and organize their apartment search process. The application gathers listings from various rental websites and allows users to track apartments they are interested in, those they want to schedule a tour for, or those they wish to save for future reference. The project includes a comprehensive database that captures essential information about apartments, user preferences, listings, and scheduled tours.

The goal of Aptly is to streamline the apartment-hunting experience, reducing the time and effort spent searching and comparing different apartments across multiple platforms. By integrating features like user preferences and real-time updates, the application aims to improve the decision-making process for renters.

The key features of Aptly include:

1. **Apartment Listings:** Provides detailed information on apartments, including rent, size, amenities, and neighborhood, helping users compare and choose based on their needs.
2. **User Reviews:** Allows users to rate and review apartments based on their personal experiences, providing valuable insights to others during their decision-making process.
3. **Tour Scheduling:** Users can schedule and track appointments for apartment tours, ensuring they stay organized and on top of their search.
4. **Apartment Comparison:** Enables users to compare multiple apartments based on factors like rent and amenities, assisting them in making informed decisions.

The project's database consists of multiple entities, including users, reviews, listings, and tours, all interconnected to provide a comprehensive view of each apartment's status and appeal.

In developing Aptly, I learned not only about technical database design and management but also about how to create a user-centric tool for a common real-world problem. The process required a balance between understanding the complexity of real estate data and designing an intuitive system for managing that data efficiently.

The development of Aptly reinforced the importance of database normalization to ensure data consistency and to minimize redundancy. By creating a well-organized database schema, I was able to connect key entities, such as listings, reviews, and user preferences, to build a relational structure that facilitates easy querying and data retrieval. This is particularly important for a project like Aptly, where quick access to up-to-date information about apartments is critical.

In addition to the technical design, working on **Aptly** also emphasized the importance of data visualization. These visualizations not only provide a clear overview of trends and insights but also help users make better decisions by comparing key metrics such as rental prices across neighborhoods or the popularity of apartments based on user reviews.

Overall, working on **Aptly** has been an enriching experience. The project has given me the opportunity to apply my skills in database design, SQL querying, and data visualization to create a tool that addresses a real-world challenge.