

## Milestone 2 - Report

### Team details:

Team Name: Summer

#### Team members:

Aaron Flierl (arflierl), Aditya sai (aprasann), Harman Singh (harmansi)

Project Name: **Real estate database optimization**

### Project overview:

The database is based on the city of Buffalo's property assessment of 2019, which contains information about properties and their related data, maintained across connected relations. It contains 9 relations: *property\_details*, *property\_specs*, *address*, *property\_class*, *police\_district*, *evaluation*, *basement\_type*, *heat\_type* and *customers* that provide detailed information about the properties and the area where they are located.

The goal of the project is to provide a structured and convenient means to organize, store, retrieve and manipulate this data to help real estate agents provide befitting property consultation to their clients.

### Problem statement:

The property assessment of a city can result in a gigantic spreadsheet containing hundreds of tightly coupled columns that provide important information about the thousands of properties located in the city. This makes it very difficult to retrieve and update the information, as each column may have multiple other columns that directly depend on it and updating it without taking care of the dependents may result in corrupting the entire database.

The problem this database aims to solve is to organize the assessment outcome into well defined relations and provide the end users the ability to easily retrieve, manipulate and update the information without violating the integrity of the database.

In order to achieve this, the data has been organized into 9 tables labeled *property\_details*, *property\_specs*, *address*, *property\_class*, *police\_district*, *evaluation*, *basement\_type*, *heat\_type* and *customers* with each table aiming at a particular aspect of the property assessment.

This allows end users such as real estate brokers to easily find properties and land available for sale or development for a wide variety of clients based on their needs.

It also enables the broker to discuss details with both parties (buyer and seller) without directly divulging the critical information to either party as each property and customer has a unique identifier associated with them.

In addition, maintaining a relational database solves the problem of updating changes to multiple properties, or inserting additional information not included in the assessment roll.

Furthermore, customers may be interested in various details of the property and the broker can provide clients a user interface connected to the database which fetches and displays the information about the various properties suitable for the clients.

### **Target User(s):**

The target users for this database will be real estate brokerage firms. The brokerage firms themselves will be responsible for inserting and updating the data as it changes. Database access would be limited to the agencies who have purchased the database, and security measures would be put in place to allow access to only authorized users. Although the property assessment itself is public, the schema of this database is what makes the assessment roll data useful for the target user.

There would also be a database administrator who is familiar with the various integrity constraints of the database schema and supervises any updates to schemas and relations. This person would work for the company that provides this database service, and nobody would have the ability to edit the database schema without the administrator's approval.

### **Dataset acquisition:**

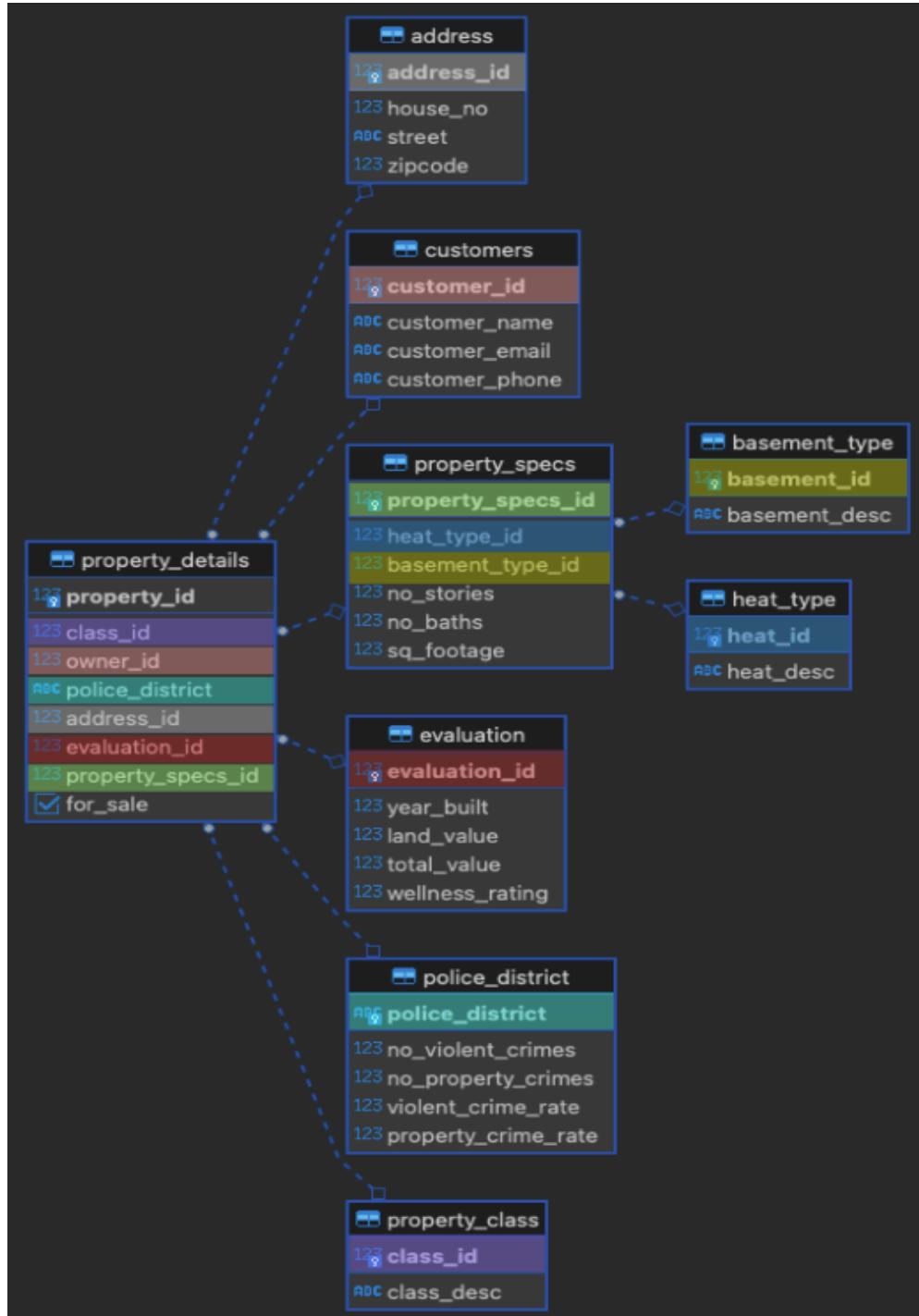
- The dataset used for the project is the 2019-20 property assessment roll for the City of Buffalo, which contains information pertaining to the year's assessed value of properties within the City of Buffalo.
- The data was in .csv format and provided a vast amount of information.
- The dataset was acquired from the website listed in the references section of this report, and team member Aaron Flierl acquired this dataset.
- The dataset was researched by team members Aaron Flierl and Harman Singh. The particular research involved finding out what information was sought after by real estate investors, private property owners, and real estate agents and agencies alike.
- Additional research was performed by all team members to find out what additional data could be acquired to improve the experience of database users. Examples of this data include the 'violent\_crime\_rate' and 'property\_crime\_rate' attributes of the *police\_district* relation, as well as the relations *heat\_type* and *basement\_type*, and the information associated with them.

**Handling large dataset: Problems encountered & their fixes:**

**Problem:** The major problem that we faced while handling the dataset was that some of the rows in the .csv file were misaligned and distorted. As a result, some of the rows had illegal data. For example: in some rows, the “Total\_value” column in the *evaluation* relation had values like “184000 False”, because the data from the boolean “for\_sale” column had wrongly moved into the ‘total\_value’ column due to the misalignment.

**Fix:** We wrote scripts to only insert the legal values (i.e, insert 184000 instead to ‘184000 False’) in such columns while loading the data from the .csv file.

### Schema design:

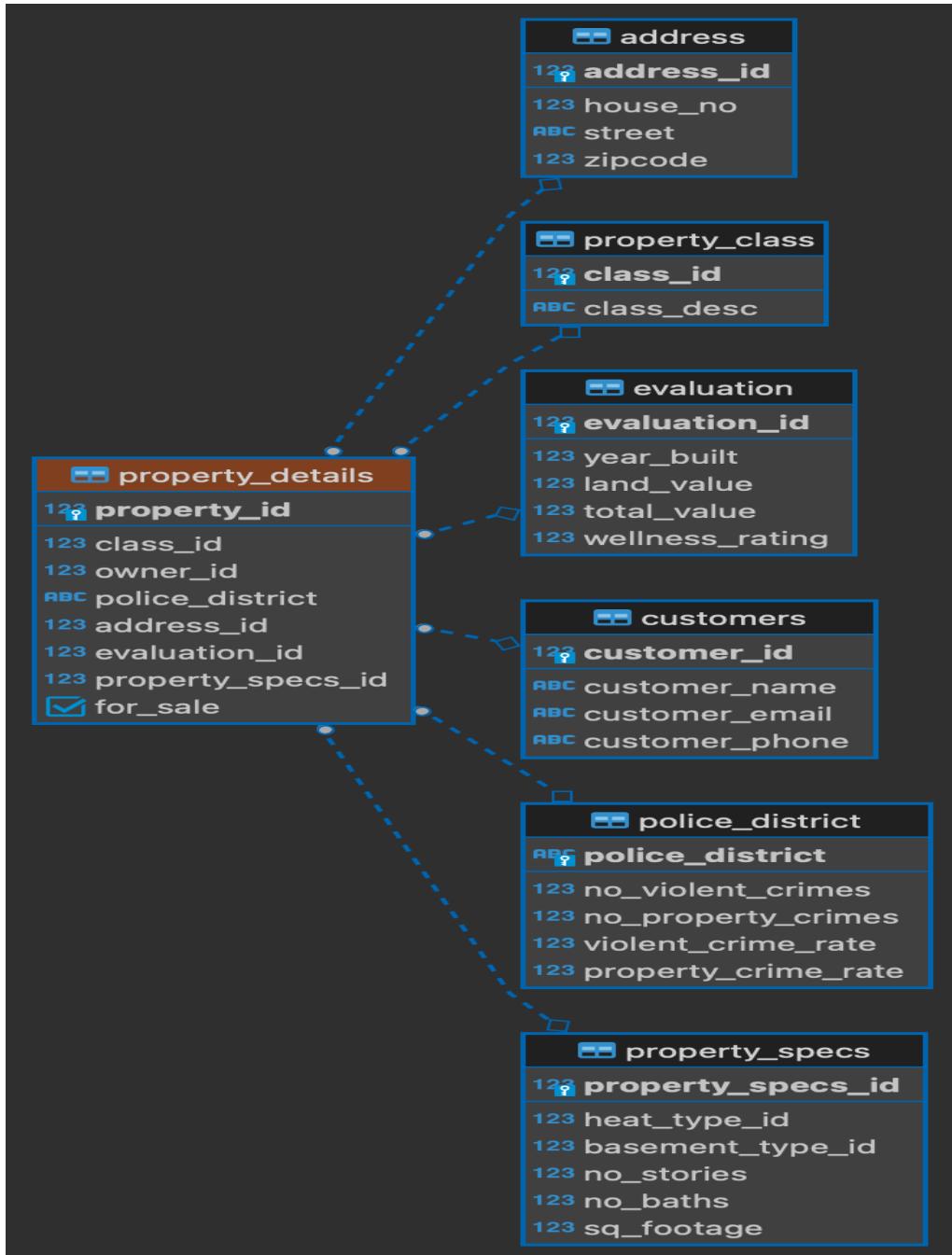


The Primary key is the first attribute in the table diagram and is denoted by the 'key' symbol right next to it.

## Details of each relation:

### Relation `property_details`:

It contains Foreign keys to `address`, `property_specs`, `property_class`, `police_district`, `evaluation`, `customers` relations which can be used to retrieve information such as the property address, owner information, and various other specifications associated with a property. In addition, it also contains a flag labeled 'for\_sale' to denote whether the property is available for sale.



**DDL command:**

```
CREATE TABLE property_details (
    property_id SERIAL PRIMARY KEY,
    class_id int NOT NULL,
    owner_id int NULL,
    police_district varchar(255) NULL,
    address_id int NOT NULL,
    evaluation_id int NOT NULL,
    property_specs_id int NOT NULL,
    for_sale bool NULL,
    FOREIGN KEY (class_id) REFERENCES property_class(class_id) ON DELETE SET NULL
    ON UPDATE CASCADE,
    FOREIGN KEY (owner_id) REFERENCES customers(customer_id) ON DELETE CASCADE
    ON UPDATE CASCADE,
    FOREIGN KEY (police_district) REFERENCES police_district(police_district) ON DELETE
    SET NULL ON UPDATE cascade,
    FOREIGN KEY (address_id) REFERENCES address(address_id) ON DELETE SET NULL
    ON UPDATE cascade,
    FOREIGN KEY (evaluation_id) REFERENCES evaluation(evaluation_id) ON DELETE SET
    NULL ON UPDATE cascade,
    FOREIGN KEY (property_specs_id) REFERENCES property_specs(property_specs_id) ON
    DELETE SET NULL ON UPDATE cascade
);
```

**Attribute details:**

- **property\_id** (**SERIAL**): The primary key of the relation, used to uniquely identify each property.
- **class\_id** (**integer**): The foreign key which references the relation *property\_class*. With the class\_id, one can obtain the class information of the property.
- **owner\_id** (**integer**): Foreign key which references the relation *customers*. The owner information can be obtained from the *customers* relation using the owner\_id.
- **police\_district** (**varchar**): The foreign key which references the relation *police\_district*. This contains the police district name in which the property is located and the crime related data of the district can be obtained from the *police\_district* relation using the *police\_district* attribute.
- **address\_id** (**integer**): Foreign key which references the relation *address*. The address of the property can be obtained from the *address* relation using the address\_id.
- **evaluation\_id** (**integer**): Foreign key which references the relation *evaluation*. The property evaluation and wellness rating can be accessed from the *evaluation* relation

using evaluation\_id.

- **property\_specs\_id** (integer): Foreign key which references the relation *property\_specs*. The property specifications such as number of stories, number of bathrooms, total square footage etc can be accessed from the *property\_specs* relation using *property\_specs\_id*.
- **for\_sale** (boolean): Indicates if the property is up for sale or not.

#### Primary key description:

- For the relation *property\_details*, we have selected **property\_id** as the primary key. This value is of type SERIAL, and thus guaranteed to be unique for each property.
- We chose this as the primary key because it is safer to use a system generated unique key rather than data such as property address to uniquely identify the property as it provides abstraction and real estate brokers will be able to freely discuss the property with sellers without giving away any sensitive information.

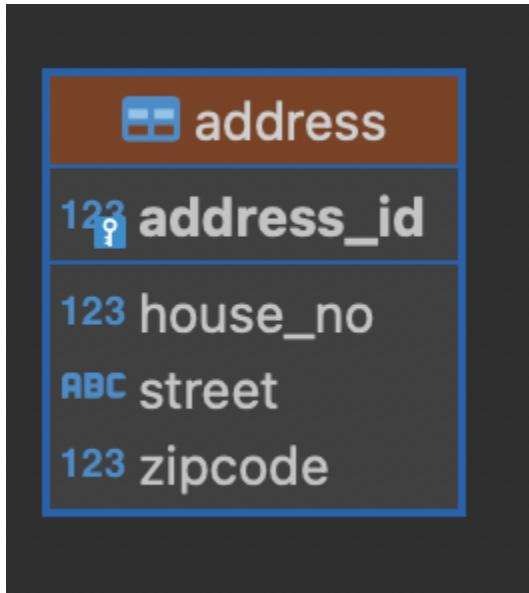
#### Functional dependencies:

$\text{property\_id} \rightarrow (\text{class\_id}, \text{owner\_id}, \text{police\_district}, \text{address\_id}, \text{evaluation\_id}, \text{property\_specs\_id}, \text{for\_sale})$

#### BCNF:

This relation is in BCNF as:

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, and each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key(*property\_id*) set consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key *property\_id*.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation

**Relation address:****DDL command:**

```
CREATE TABLE public.address (
    address_id SERIAL PRIMARY KEY,
    house_no int NULL,
    street varchar(255) NULL,
    zipcode int NOT NULL
);
```

**Attribute details:**

- **address\_id** (serial): The primary key of the relation, used to uniquely identify each record. It is also a foreign key constraint in the *property\_details* relation.
- **house\_no** (integer): The official number of the property as per the registered address.
- **street** (varchar): The name of the street where the property is located.
- **zipcode** (integer): The zipcode of the area where the property is located. This attribute must have a value in every tuple. It cannot be NULL.

**Primary key description:**

The primary key for this relation is **address\_id**. It is of type SERIAL and thus guaranteed to be unique for each tuple in this relation.

**Functional dependencies:**

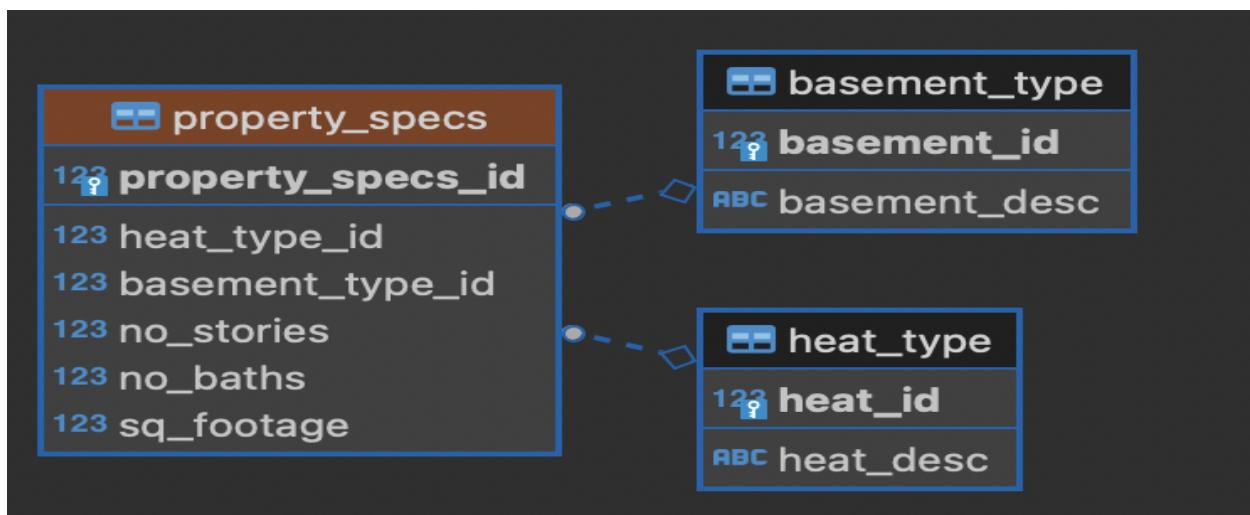
`address_id → house_no, street, zipcode`

**BCNF:**

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set (`address_id`) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key `address_id`.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation

**Relation `property_specs`:**

It contains information about the property specifications such as the basement type, heat type, number of stories, number of bathrooms and total square footage.



**DDL command:**

```
CREATE TABLE property_specs (
    property_specs_id SERIAL primary KEY,
    heat_type_id int,
    basement_type_id int,
    no_stories int NULL,
    no_baths int NULL,
    sq_footage int NULL,
    FOREIGN KEY (heat_type_id) REFERENCES heat_type(heat_id) ON DELETE SET NULL ON UPDATE CASCADE,
    FOREIGN KEY (basement_type_id) REFERENCES basement_type(basement_id) ON DELETE SET NULL ON UPDATE CASCADE
);
```

**Attribute details:**

- **property\_specs\_id** (serial): The primary key of the relation, used to uniquely identify each record. It is a foreign key constraint in the *property\_details* relation.
- **heat\_type\_id** (integer): The foreign key which references the relation *heat\_type*. With heat\_type\_id, one can obtain the information about how the property is heated.
- **basement\_type\_id** (integer): The foreign key which references the relation *basement\_type*. Using basement\_type\_id, one can identify the basement type associated with the property.
- **no\_stories** (integer): The number of stories for the associated property.
- **no\_baths** (integer): The number of bathrooms in the property.
- **sq\_footage** (integer): The square footage of the property.

**Primary key description:**

The primary key for this relation is **property\_specs\_id**. It is of type SERIAL, and thus guaranteed to be unique for each tuple in this relation.

**Functional dependencies:**

$\text{property\_specs\_id} \rightarrow \text{heat\_type\_id}, \text{basement\_type\_id}, \text{no\_stories}, \text{no\_baths}, \text{sq\_footage}$

**BCNF:**

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set (property\_specs\_id) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key property\_specs\_id.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation.

**Relation *basement\_type*:**

This relation provides information about the various types of basements associated with a property such as Masonry Wall, Concrete, Crawl Space etc.

**DDL command:**

```

CREATE TABLE basement_type (
    basement_id SERIAL PRIMARY KEY,
    basement_desc varchar(255) NULL
);
  
```

**Attribute details:**

- **basement\_id** (serial): The primary key of the relation, used to uniquely identify each basement type. It is also a foreign key constraint in the *property\_specs* relation.
- **basement\_desc** (varchar): It contains the description of the various types of basements associated with a property.

**Primary key description:**

The primary key for this relation is **basement\_id**. It is of type SERIAL, and thus guaranteed to be unique for each tuple in this relation.

**Functional dependencies:**

$\text{basement\_type} \rightarrow \text{basement\_desc}$

**BCNF:**

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set (*basement\_id*) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key *basement\_id*.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation

### Relation ***heat\_type***:

This relation provides information about the various types of heating systems present in the properties such as Furnace, Boiler, Ductless, Electric etc.



**DDL command:**

```

CREATE TABLE heat_type (
    heat_id SERIAL PRIMARY KEY,
    heat_desc varchar(255) NULL
);
  
```

**Attribute details:**

- **heat\_id** (serial): The primary key of the relation, used to uniquely identify each heat type. It is also a foreign key constraint in the *property\_specs* relation.
- **heat\_desc** (varchar): It contains the description of the various types of heating systems present in a property.

**Primary key description:**

The primary key for this relation is **heat\_id**. It is of type SERIAL, and thus guaranteed to be unique for each tuple in this relation.

**Functional dependencies:**

$\text{heat\_id} \rightarrow \text{heat\_desc}$

**BCNF:**

This relation is in BCNF as:

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set ( $\text{heat\_id}$ ) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key  $\text{heat\_id}$ .
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation.

**Relation evaluation:**

It contains information about the various factors that contribute to the value of a property such as year built, land value, total value and the wellness rating.

evaluation	
123	evaluation_id
123	year_built
123	land_value
123	total_value
123	wellness_rating

**DDL command:**

```
CREATE TABLE evaluation (
    evaluation_id serial primary KEY,
    year_built int NULL,
    land_value int NULL,
    total_value int NULL,
    wellness_rating int NULL
);
```

**Attribute details:**

- **evaluation\_id** (serial): The primary key of the relation, used to uniquely identify each record. It is also a foreign key constraint in the *property\_details* relation.
- **year\_built** (integer): The year a property was built.
- **land\_value** (integer): The value of the land a property is located on.
- **total\_value** (integer): The combined value of the land plus the construction on it.
- **wellness\_rating** (integer): The overall rating of the property based on its condition, ranging from 1 to 5 with 5 being the best.

**Primary key description:**

The primary key for this relation is **evaluation\_id**. It is of type SERIAL, and thus guaranteed to be unique for each tuple in this relation.

**Functional dependencies:**

$\text{evaluation\_id} \rightarrow \text{year\_built}, \text{land\_value}, \text{total\_value}, \text{wellness\_rating}$

**BCNF:**

This relation is in BCNF as:

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.

- It is in 2NF, as it is in 1NF and the primary key set (evaluation\_id) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key evaluation\_id.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation.

### **Relation property\_class:**

This relation provides the class information of the property such as if the property is zoned for commercial, industrial, residential, religious, and other purposes.

<b>property_class</b>	
<b>class_id</b>	<b>class_desc</b>

### **DDL command:**

```
CREATE TABLE property_class (
    class_id int primary KEY,
    class_desc varchar(255)
);
```

### **Attribute details:**

- **Class\_id** (integer): The primary key of the relation, contains the unique ID for each class.
- **Class\_desc** (varchar): This attribute contains the description for each class.

### **Primary Key Discussion:**

The primary key for this relation is the **class\_id**. We have chosen this as the primary key because it is guaranteed to be unique by its definition and application in the assessment roll itself.

**Functional dependencies:**

Class\_id → class\_desc

**BCNF:**

This relation is in BCNF as:

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set (class\_id) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key class\_id.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation.

**Relation police\_district:**

This relation is maintained to organize crime-related data of each police district including the number and rate of violent/ property crimes.

police_district
police_district
123 no_violent_crimes
123 no_property_crimes
123 violent_crime_rate
123 property_crime_rate

**DDL command:**

```
CREATE TABLE police_district(  
    police_district varchar(255) NOT NULL,  
    no_violent_crimes int NOT NULL,  
    no_property_crimes int NOT NULL,  
    violent_crime_rate numeric(4,2) NOT NULL,  
    property_crime_rate numeric(4,2) NOT NULL,  
    PRIMARY KEY (police_district)  
)
```

**Attribute details:**

- **police\_district** (varchar): name of the police district. It serves as the primary key because each police district has a unique name (from District A to E) and hence cannot be NULL.
- **no\_violent\_crimes** (integer): the number of violent crimes reported in the district. This attribute must have a value in every tuple. It cannot be NULL.
- **no\_property\_crimes** (integer): Number of property related crimes such as break-ins, thefts etc. This attribute must have a value in every tuple. It cannot be NULL.
- **violent\_crime\_rate** (numeric): The number of violent crimes committed per thousand people in the district.
- **property\_crime\_rate** (numeric): The number of property crimes committed per thousand people in the district.

**Primary Key description:**

We have chosen **police\_district** as the primary key because there are only 6 unique police district values in the assessment roll itself, and by definition they must be unique. Also, no property can belong to two separate districts.

**Functional dependencies:**

Police\_district → no\_violent\_crimes,  
no\_property\_crimes, violent\_crime\_rate, property\_crime\_rate

**BCNF:**

This relation is in BCNF as:

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set (police\_district) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key police\_district.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation.

**Relation customers:**

This relation is maintained to organize data related to the customer base of the database user(for example, a real estate agency). It contains the contact information of the customer.

customers	
123	customer_id
ABC	customer_name
ABC	customer_email
ABC	customer_phone

**DDL command:**

```
CREATE TABLE customers(
    customer_id SERIAL primary KEY,
    customer_name varchar(255) NOT NULL,
    customer_email varchar(255),
    customer_phone varchar(255)
);
```

**Attribute details:**

- **customer\_id** (serial): Contains a unique ID for each owner. This attribute is the primary key of the relation.
- **customer\_name** (varchar): The full name of the owner. This attribute must have a value in every tuple. It cannot be NULL.
- **customer\_email** (varchar): The email ID of the owner.
- **customer\_phone** (varchar): The contact number of the owner.

**Primary Key Description:**

The primary key for this relation is **customer\_id**. It is of type SERIAL, and thus guaranteed to be unique for each tuple in this relation.

We chose this as the primary key because it is safer to use a system generated unique key than any customer related information to uniquely identify the customers.

**Functional dependencies:**

$\text{Customer\_id} \rightarrow \text{customer\_name}, \text{customer\_email}, \text{customer\_phone}$

**BCNF:**

This relation is in BCNF as:

- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, each attribute name is unique in the relation.
- It is in 2NF, as it is in 1NF and the primary key set (**customer\_id**) consists of only a single attribute.
- It is in 3NF, as it is in 2NF and there are no non-primary key attributes in this relation that is *transitively* dependent on the primary key **customer\_id**.
- Finally, the relation is in BCNF as it is in 3NF and there are also no non-primary attributes functionally defining a primary attribute in the relation.

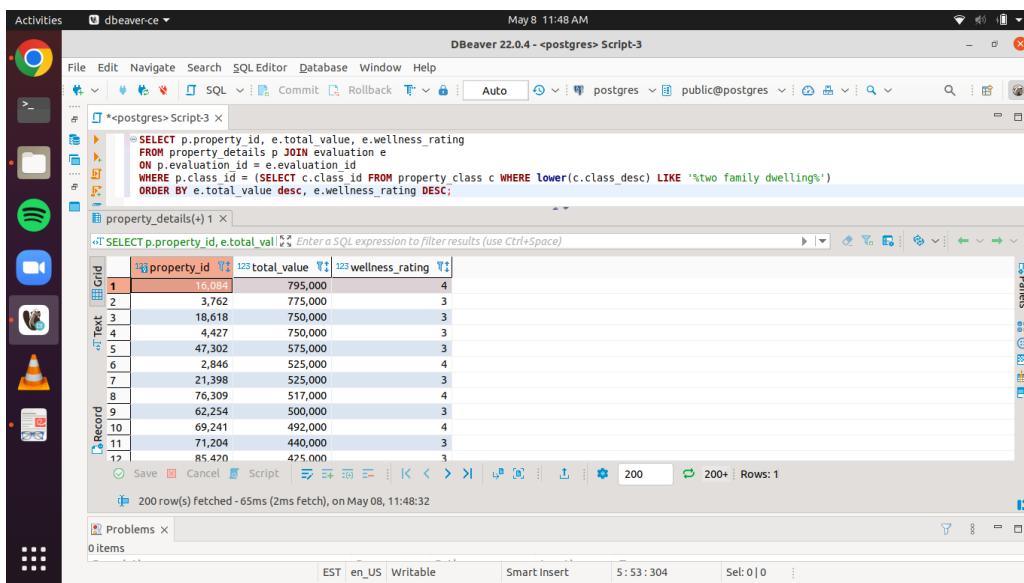
## Queries:

### Select statements:

**order by with subquery:** This query returns property\_id, and its associated year built and wellness rating, for properties with property call description containing the value ‘two family dwelling’. Ordering first by total value, and then by wellness rating

```
SELECT p.property_id, e.total_value, e.wellness_rating
FROM property_details p JOIN evaluation e
ON p.evaluation_id = e.evaluation_id
WHERE p.class_id = (SELECT c.class_id FROM property_class c WHERE
lower(c.class_desc) LIKE '%two family dwelling%')
ORDER BY e.total_value desc, e.wellness_rating DESC;
```

### Result:



The screenshot shows the DBBeaver interface with a SQL Editor window. The query executed is:

```
SELECT p.property_id, e.total_value, e.wellness_rating
FROM property_details p JOIN evaluation e
ON p.evaluation_id = e.evaluation_id
WHERE p.class_id = (SELECT c.class_id FROM property_class c WHERE lower(c.class_desc) LIKE '%two family dwelling%')
ORDER BY e.total_value desc, e.wellness_rating DESC;
```

The results grid displays the following data:

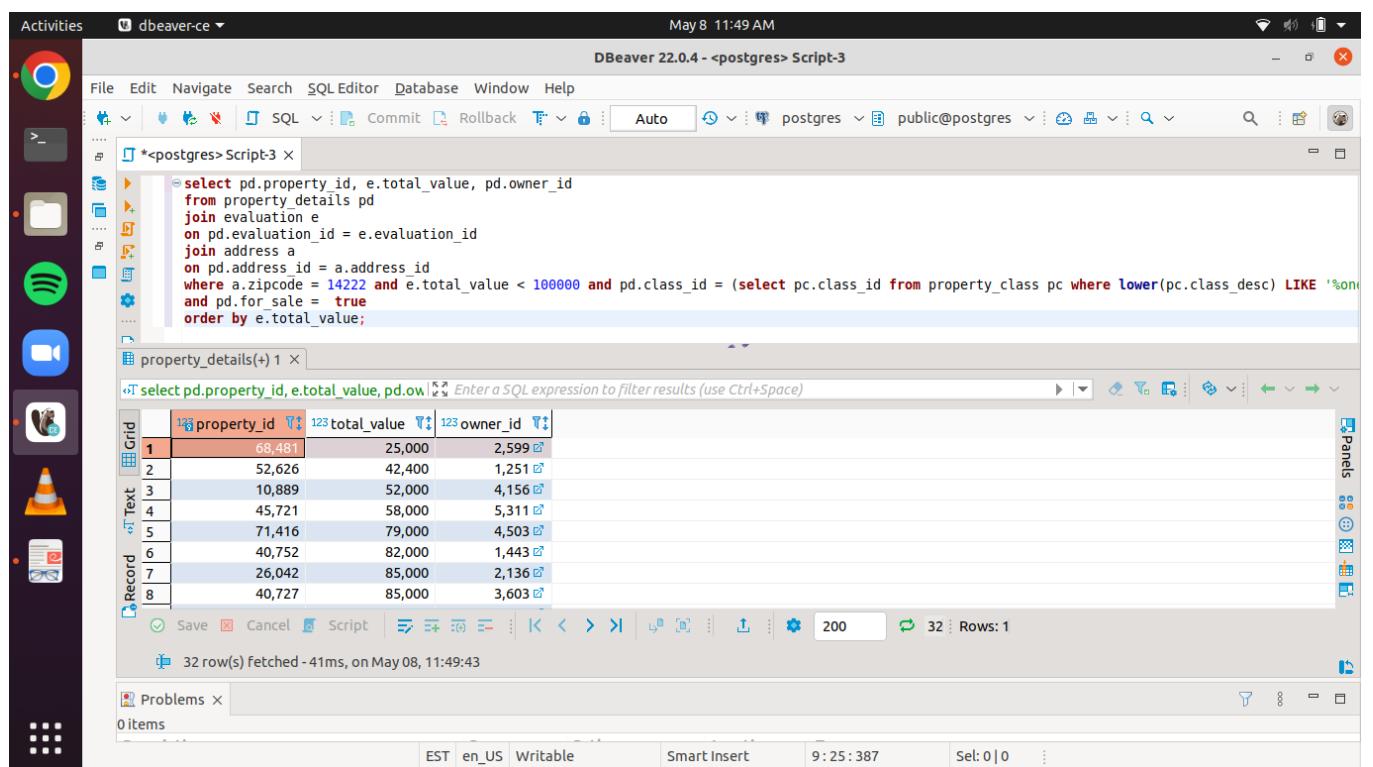
property_id	total_value	wellness_rating
16,084	795,000	4
3,762	775,000	3
18,618	750,000	3
4,427	750,000	3
47,302	575,000	3
2,846	525,000	4
21,398	525,000	3
76,309	517,000	4
62,254	500,000	3
69,241	492,000	4
71,204	440,000	3
85,420	425,000	3

200 row(s) fetched - 65ms (2ms fetch), on May 08, 11:48:32

**order by with multiple joins:** This query returns the properties which are one family homes currently for sale, in a particular zip code, and with a total value under 100,000. Then orders the results by total value.

```
select pd.property_id, e.total_value, pd.owner_id
from property_details pd
join evaluation e
on pd.evaluation_id = e.evaluation_id
join address a
on pd.address_id = a.address_id
where a.zipcode = 14222 and e.total_value < 100000 and pd.class_id = (select
pc.class_id from property_class pc where lower(pc.class_desc) LIKE '%one family%' )
and pd.for_sale = true
order by e.total_value;
```

### Result:



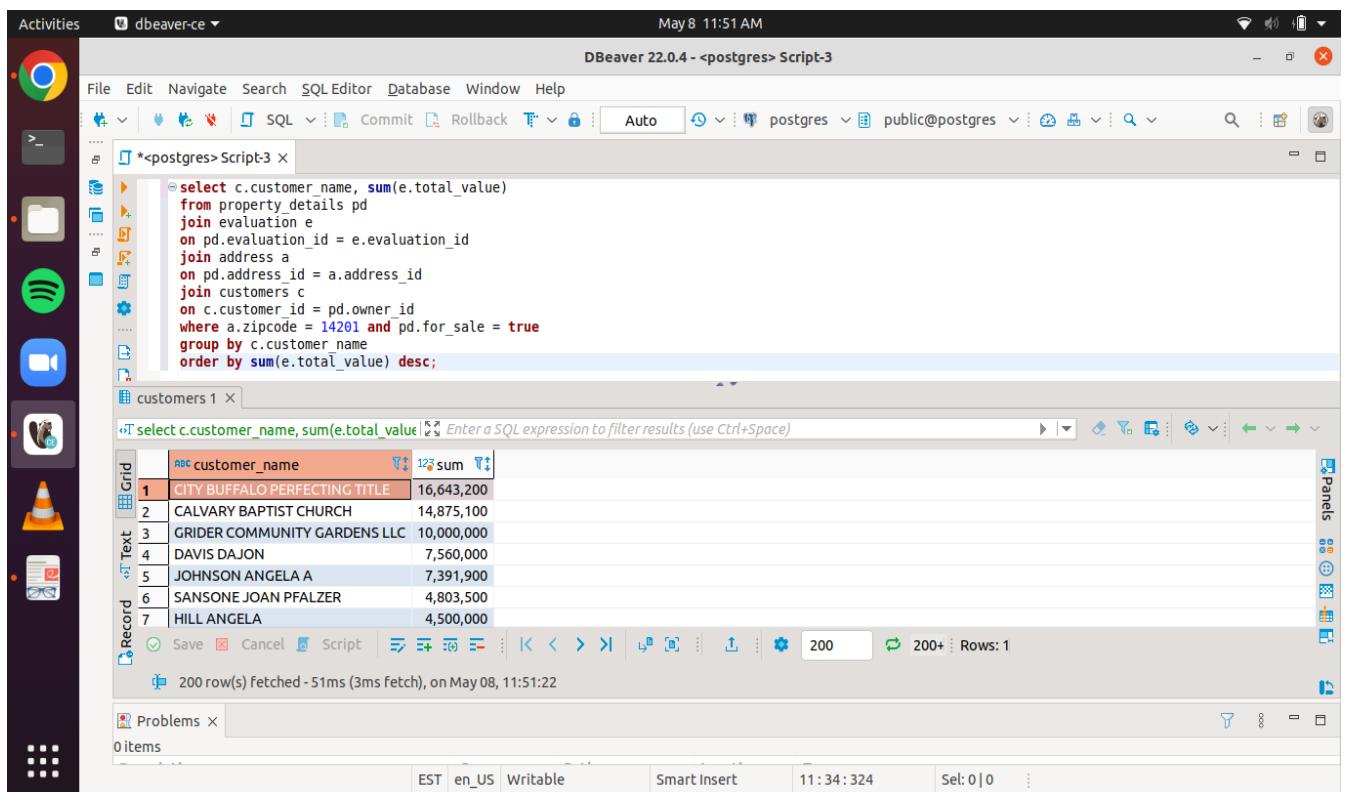
The screenshot shows the DBeaver 22.0.4 interface with a PostgreSQL database connection. The SQL Editor tab is active, displaying the previously written query. The results are shown in a grid below, with 32 rows fetched. The columns are labeled: property\_id, total\_value, and owner\_id. The data includes various property IDs, their total values (e.g., 25,000, 42,400, 52,000), and their respective owners' IDs (e.g., 2,599, 1,251, 4,156).

	property_id	total_value	owner_id
1	68,481	25,000	2,599
2	52,626	42,400	1,251
3	10,889	52,000	4,156
4	45,721	58,000	5,311
5	71,416	79,000	4,503
6	40,752	82,000	1,443
7	26,042	85,000	2,136
8	40,727	85,000	3,603

**group by with order by, including multiple joins:** This query returns the names of customers and the total value of the properties they own in a particular zip code which are currently up for sale.

```
select c.customer_name, sum(e.total_value)
from property_details pd
join evaluation e
on pd.evaluation_id = e.evaluation_id
join address a
on pd.address_id = a.address_id
join customers c
on c.customer_id = pd.owner_id
where a.zipcode = 14201 and pd.for_sale = true
group by c.customer_name
order by sum(e.total_value) desc;
```

## Result:

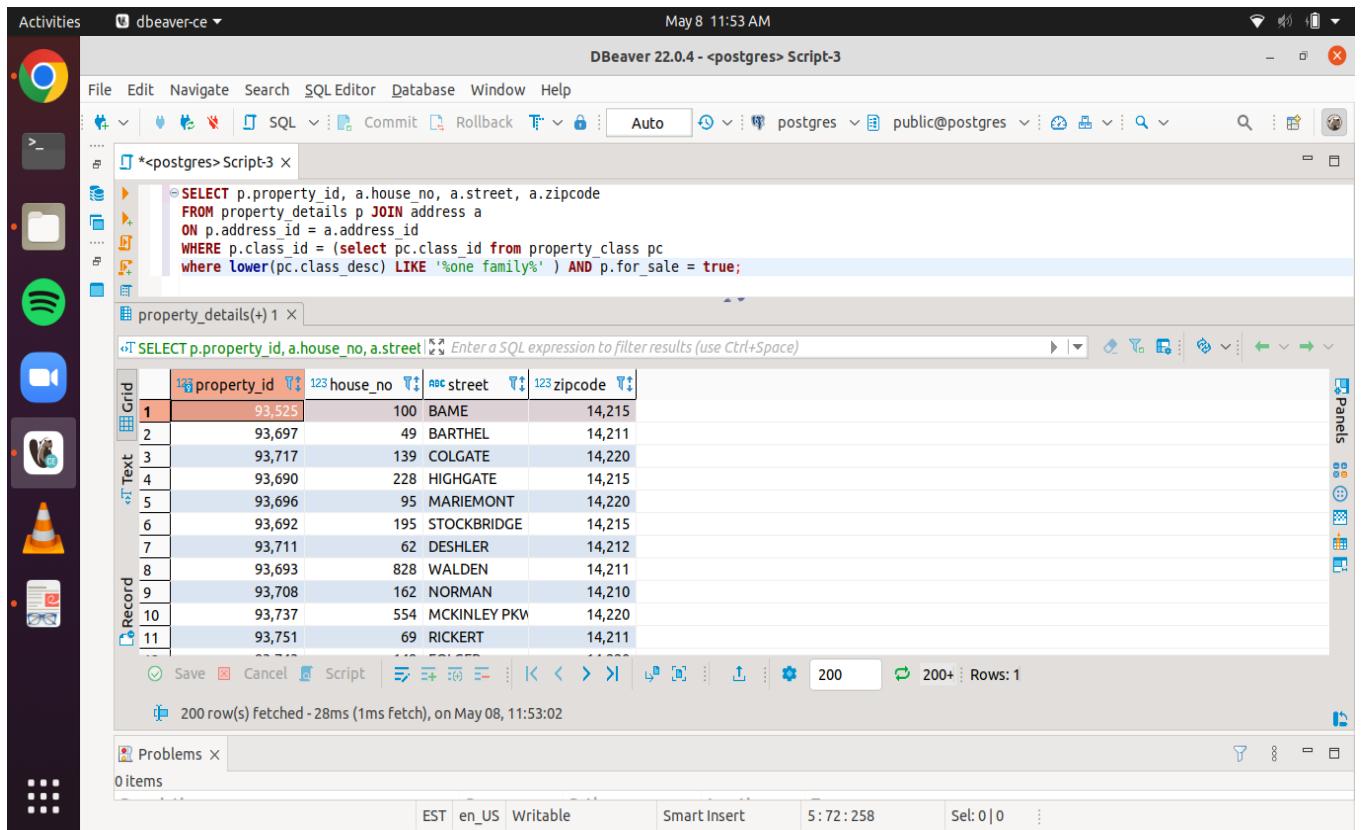


customer_name	sum
CITY BUFFALO PERFECTING TITLE	16,643,200
CALvary BAPTIST CHURCH	14,875,100
GRIDER COMMUNITY GARDENS LLC	10,000,000
DAVIS DAJON	7,560,000
JOHNSON ANGELA A	7,391,900
SANSONE JOAN PFALZER	4,803,500
HILL ANGELA	4,500,000

**join:** This query returns the property id, house number, street, and zip code of all the properties with the property class which corresponds to one family dwellings, which are currently up for sale.

```
SELECT p.property_id, a.house_no, a.street, a.zipcode
FROM property_details p JOIN address a
ON p.address_id = a.address_id
WHERE p.class_id = (select pc.class_id from property_class pc
where lower(pc.class_desc) LIKE '%one family%' ) AND p.for_sale = true;
```

## Result:



```
May 8 11:53 AM
DBeaver 22.0.4 - <postgres> Script-3
File Edit Navigate Search SQL Editor Database Window Help
* <postgres> Script-3
SELECT p.property_id, a.house_no, a.street, a.zipcode
FROM property_details p JOIN address a
ON p.address_id = a.address_id
WHERE p.class_id = (select pc.class_id from property_class pc
where lower(pc.class_desc) LIKE '%one family%' ) AND p.for_sale = true;
property_details(+)
SELECT p.property_id, a.house_no, a.street| Enter a SQL expression to filter results (use Ctrl+Space)
Grid
1 93,525 100 BAME 14,215
2 93,697 49 BARTHEL 14,211
3 93,717 139 COLGATE 14,220
4 93,690 228 HIGHGATE 14,215
5 93,696 95 MARIEMONT 14,220
6 93,692 195 STOCKBRIDGE 14,215
7 93,711 62 DESHLER 14,212
8 93,693 828 WALDEN 14,211
9 93,708 162 NORMAN 14,210
10 93,737 554 MCKINLEY PKW 14,220
11 93,751 69 RICKERT 14,211
Record
200 row(s) fetched - 28ms (1ms fetch), on May 08, 11:53:02
Problems
Items
EST en_US Writable Smart Insert 5:72:258 Sel: 0 | 0
```

## **Insert/Update/Delete statements:**

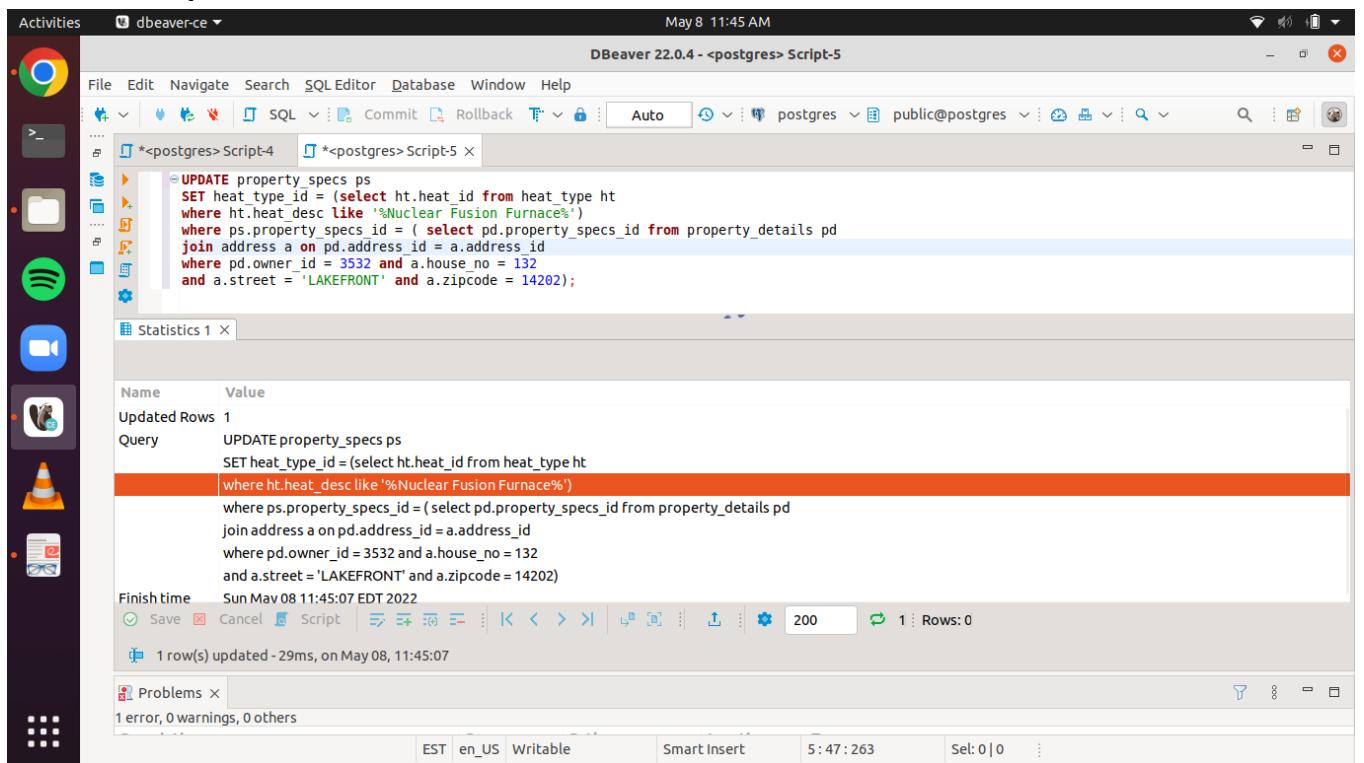
**Update with multiple joins and subquery:** Here, we are updating the heat\_type\_id of a specific property to have a new heat type, Nuclear Fusion Furnace, which has a heat\_type\_id of 6, which was assigned automatically, as it is a value with type SERIAL, and also guaranteed and required to be unique. This reflects a real-life scenario where a customer ( who would know their customer\_id ( which is = owner\_id ) calls and gives us the address of their property, and customer ID, and lets us know that they have a new heating system and want us to update their property details and also perform a new evaluation of their property value and wellness rating ( shown in the subsequent query ).

```
UPDATE property_specs ps  
SET heat_type_id = (select ht.heat_id from heat_type ht  
where ht.heat_desc like '%Nuclear Fusion Furnace%')  
where ps.property_specs_id = (select pd.property_specs_id from property_details pd  
    join address a on pd.address_id = a.address_id  
    where pd.owner_id = 3532 and a.house_no = 132  
    and a.street = 'LAKEFRONT' and a.zipcode = 14202);
```

## **Before Update:**

The screenshot shows the DBeaver interface with the following details:

- Toolbar:** File, Edit, Navigate, Search, SQL Editor, Database, Window, Help.
- Top Bar:** Activities (dbeaver-ce), Date (May 8 11:44 AM), Version (DBeaver 22.0.4 - <postgres> Script-4).
- Script Editor:** Two tabs: \*<postgres> Script-4 and \*<postgres> Script-5. The current tab contains a complex PostgreSQL query involving joins between property\_specs, heat\_type, and address tables.
- Result Grid:** Shows the results of the executed query. The grid has three columns: property\_specs\_id, heat\_type\_id, and heat\_desc. One row is displayed with values 1, 265, and Boiler respectively.
- Status Bar:** Shows "1 row(s) fetched - 23ms, on May 08, 11:34:04".
- Bottom Panel:** Problems panel showing 1 error, 0 warnings, 0 others.

**Successful Update:**


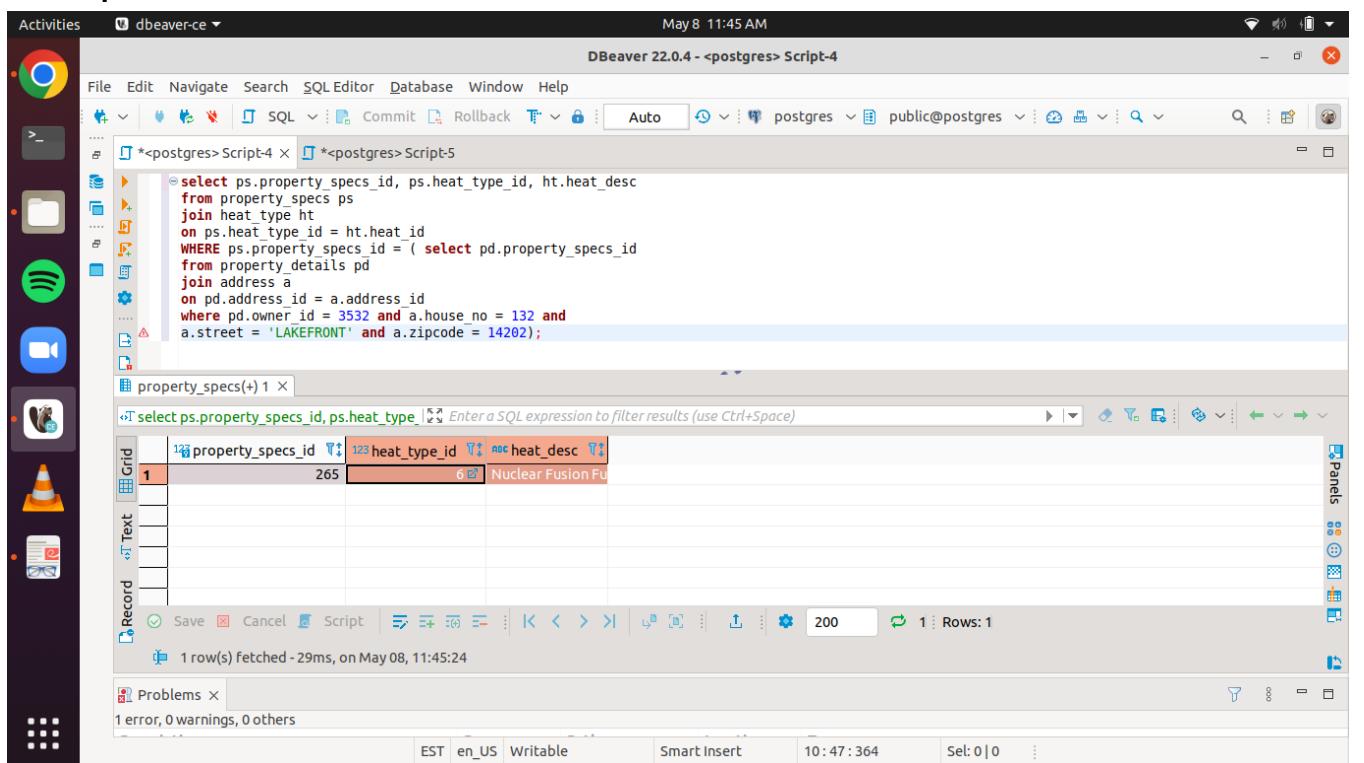
The screenshot shows the DBeaver interface with a successful update operation. The SQL Editor tab contains the following query:

```
UPDATE property_specs ps
SET heat_type_id = (select ht.heat_id from heat_type ht
where ht.heat_desc like '%Nuclear Fusion Furnace%')
where ps.property_specs_id = ( select pd.property_specs_id from property_details pd
join address a on pd.address_id = a.address_id
where pd.owner_id = 3532 and a.house_no = 132
and a.street = 'LAKEFRONT' and a.zipcode = 14202);
```

The results pane shows:

- Name: Updated Rows
- Value: 1
- Name: Query
- Value: UPDATE property\_specs ps
`SET heat_type_id = (select ht.heat_id from heat_type ht
where ht.heat_desc like '%Nuclear Fusion Furnace%')`
- Name: Finish time
- Value: Sun May 08 11:45:07 EDT 2022

At the bottom, it says "1 row(s) updated - 29ms, on May 08, 11:45:07".

**After Update:**


The screenshot shows the DBeaver interface after the update. The SQL Editor tab contains the same query as before, but now it includes a WHERE clause to filter the results.

```
select ps.property_specs_id, ps.heat_type_id, ht.heat_desc
from property_specs ps
join heat_type ht
on ps.heat_type_id = ht.heat_id
WHERE ps.property_specs_id = ( select pd.property_specs_id
from property_details pd
join address a
on pd.address_id = a.address_id
where pd.owner_id = 3532 and a.house_no = 132 and
a.street = 'LAKEFRONT' and a.zipcode = 14202);
```

The results pane shows a single row in a grid:

property_specs_id	heat_type_id	heat_desc
1	265	Nuclear Fusion Fu

At the bottom, it says "1 row(s) fetched - 29ms, on May 08, 11:45:24".

**update:** Here, we are updating the same property shown in the query above, which has evaluation\_id = 265. Since it has a new furnace, the wellness rating has increased from 4 to 5. Also, the total value was updated to 2,000,000.

```
UPDATE evaluation e
SET total_value=2000000, wellness_rating=5
WHERE evaluation_id= ( select pd.evaluation_id
    from property_details pd
    join address a
    on pd.address_id = a.address_id
    where pd.owner_id = 3532 and a.house_no = 132 and
    a.street = 'LAKEFRONT' and a.zipcode = 14202);
```

### Before Update:

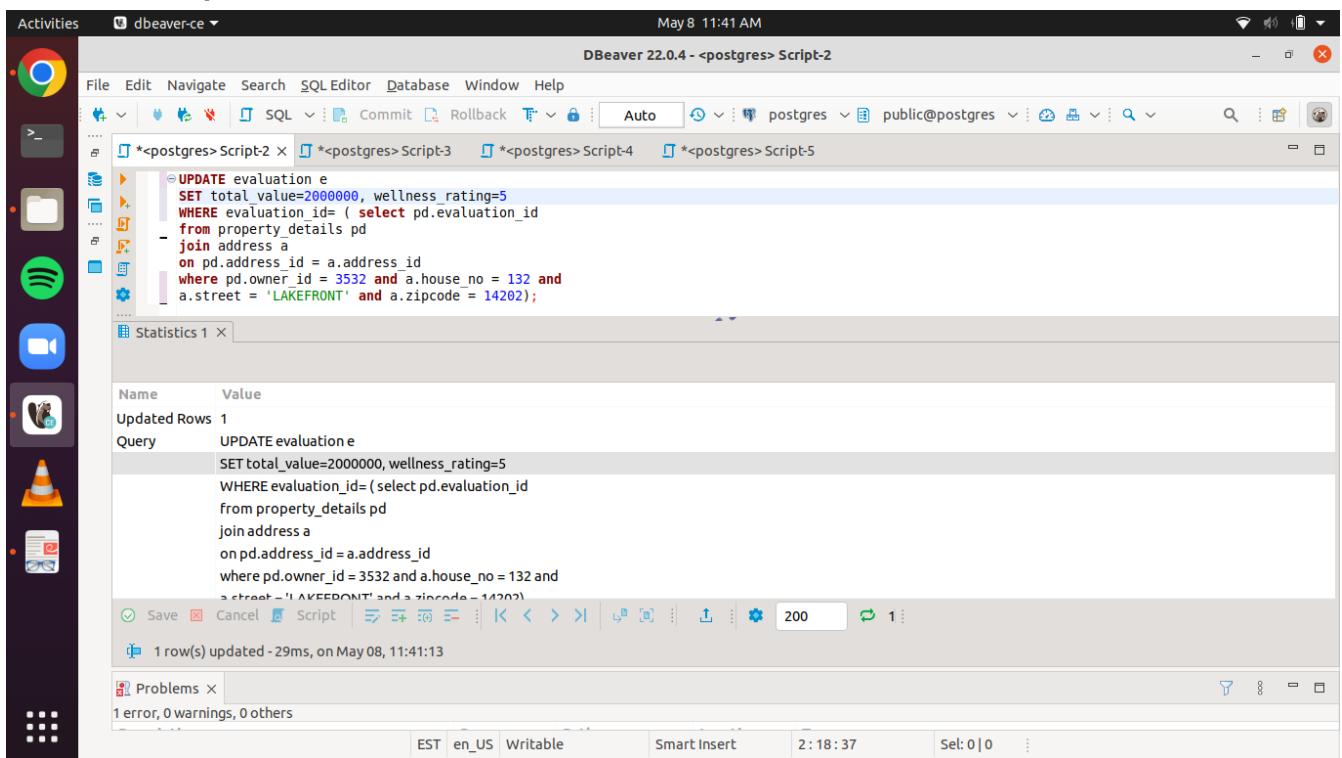
The screenshot shows the DBeaver interface with the following details:

- Toolbar:** Activities, dbeaver-ce, File, Edit, Navigate, Search, SQL Editor, Database, Window, Help.
- Menubar:** File, Edit, Navigate, Search, SQL Editor, Database, Window, Help.
- Script Editor:** \*<postgres> Script-2, \*<postgres> Script-3, \*<postgres> Script-4, \*<postgres> Script-5. The fourth script contains the initial SELECT query.
- Table View:** Shows the 'evaluation' table with the following data:

	house_no	street	zipcode	total_value	wellness_rating
1	132	LAKEFRONT	14,202	200,000	4

- Status Bar:** May 8 11:34 AM, 1 row(s) fetched - 38ms, on May 08, 11:34:24, EST, en\_US, Writable, Smart Insert, 12:47:419, Sel: 0 | 0.

## Successful Update:

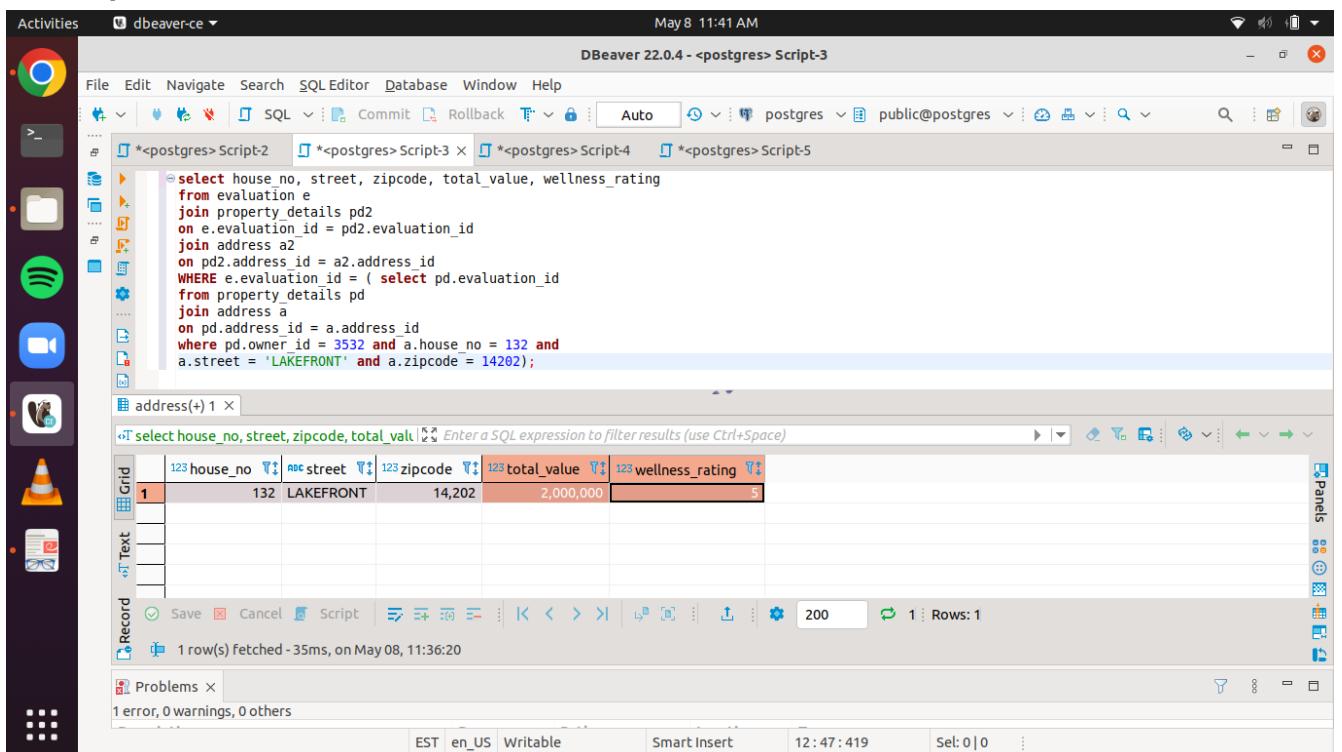


The screenshot shows the DBBeaver interface with a successful update operation. The SQL Editor tab contains the following query:

```
UPDATE evaluation e
SET total_value=2000000, wellness_rating=5
WHERE evaluation_id= ( select pd.evaluation_id
from property_details pd
join address a
on pd.address_id = a.address_id
where pd.owner_id = 3532 and a.house_no = 132 and
a.street = 'LAKEFRONT' and a.zipcode = 14202);
```

The results pane shows "Updated Rows 1" and the query again. A message at the bottom indicates "1 row(s) updated - 29ms, on May 08, 11:41:13". The Problems pane shows "1 error, 0 warnings, 0 others".

## After Update:



The screenshot shows the DBBeaver interface after the update. The SQL Editor tab contains a query to select data from the evaluation and property\_details tables:

```
select house_no, street, zipcode, total_value, wellness_rating
from evaluation e
join property_details pd2
on e.evaluation_id = pd2.evaluation_id
join address a2
on pd2.address_id = a2.address_id
WHERE e.evaluation_id = ( select pd.evaluation_id
from property_details pd
join address a
on pd.address_id = a.address_id
where pd.owner_id = 3532 and a.house_no = 132 and
a.street = 'LAKEFRONT' and a.zipcode = 14202);
```

The results pane displays a single row of data in a grid format:

house_no	street	zipcode	total_value	wellness_rating
132	LAKEFRONT	14,202	2,000,000	5

A message at the bottom indicates "1 row(s) fetched - 35ms, on May 08, 11:36:20". The Problems pane shows "1 error, 0 warnings, 0 others".

**insert:** here, we are inserting a new heat type, Wood Burning Fireplace.

```
INSERT INTO heat_type
(heat_desc)
VALUES('Wood Burning Fireplace');
```

### Before Insert:

The screenshot shows the DBBeaver interface with the 'heat\_type' table selected in the left sidebar. The table has two columns: 'heat\_id' and 'heat\_desc'. The data is as follows:

heat_id	heat_desc
1	Furnace
2	Boiler
3	Hybrid
4	Ductless
5	Electric
6	Nuclear Fusion Furnace

### Successful insert:

The screenshot shows the DBBeaver interface after the insertion. The 'heat\_type' table now includes a new row:

heat_id	heat_desc
1	Furnace
2	Boiler
3	Hybrid
4	Ductless
5	Electric
6	Nuclear Fusion Furnace
7	Wood Burning Fireplace

The SQL Editor tab shows the executed query:

```
INSERT INTO heat_type
(heat_desc)
VALUES('Wood Burning Fireplace');
```

The Statistics panel shows the following results:

Name	Value
Updated Rows	1
Query	INSERT INTO heat_type (heat_desc) VALUES('Wood Burning Fireplace')
Finish time	Sun May 08 11:58:54 EDT 2022

**After Insert:**

The screenshot shows the DBeaver 22.0.4 interface with the following details:

- Toolbar:** Activities, dbeaver-ce, File, Edit, Navigate, Search, SQL Editor, Database, Window, Help.
- Database Navigator:** Shows the `postgres` database, specifically the `public` schema, containing tables like `address`, `basement_type`, `customers`, `evaluation`, `heat_type`, `police_district`, `property_class`, `property_detail`, and `property_specs`.
- SQL Editor:** Displays the query `select * from heat_type ht`. The results are shown in a table titled "heat\_type 1".
- Table Results:** The table has two columns: `heat_id` and `heat_desc`. The data is as follows:

heat_id	heat_desc
1	Furnace
2	Boiler
3	Hybrid
4	Ductless
5	Electric
6	Nuclear Fusion Furnace
14	Wood Burning Fireplace

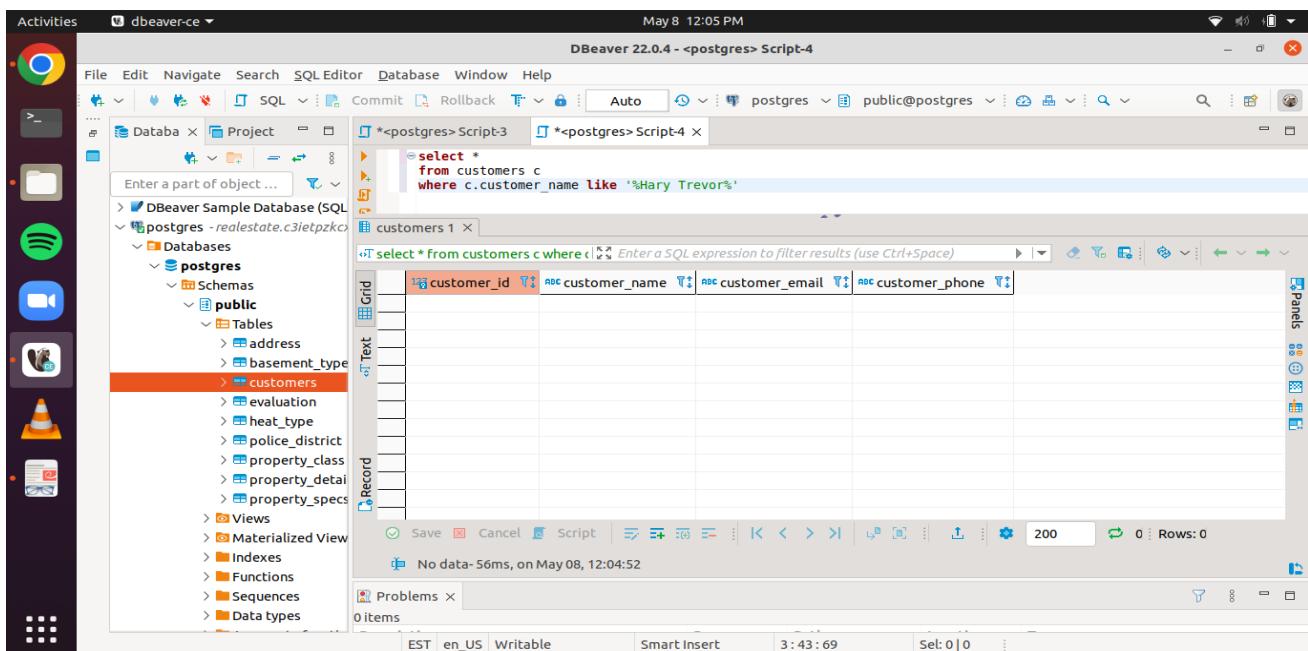
**Bottom Status Bar:** EST en\_US Writable, Smart Insert, 2:19:27, Sel: 0 | 0

**insert:** here, we are inserting a new customer, whose name is Harry Trevor and whose email is hary@harymail.net.

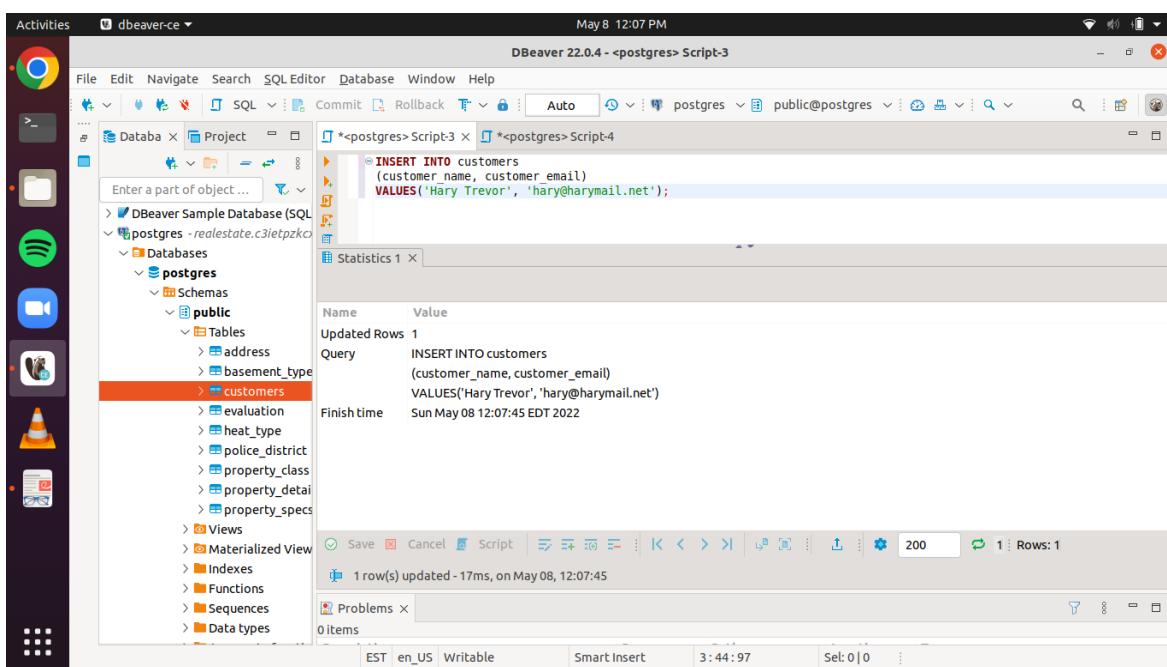
## INSERT INTO customers

```
(customer_name, customer_email)  
VALUES('Hary Trevor', 'hary@harymail.net');
```

## **Before Insert:**



### **Successful Insert:**



**After Insert:**

The screenshot shows the DBeaver 22.0.4 interface with the following details:

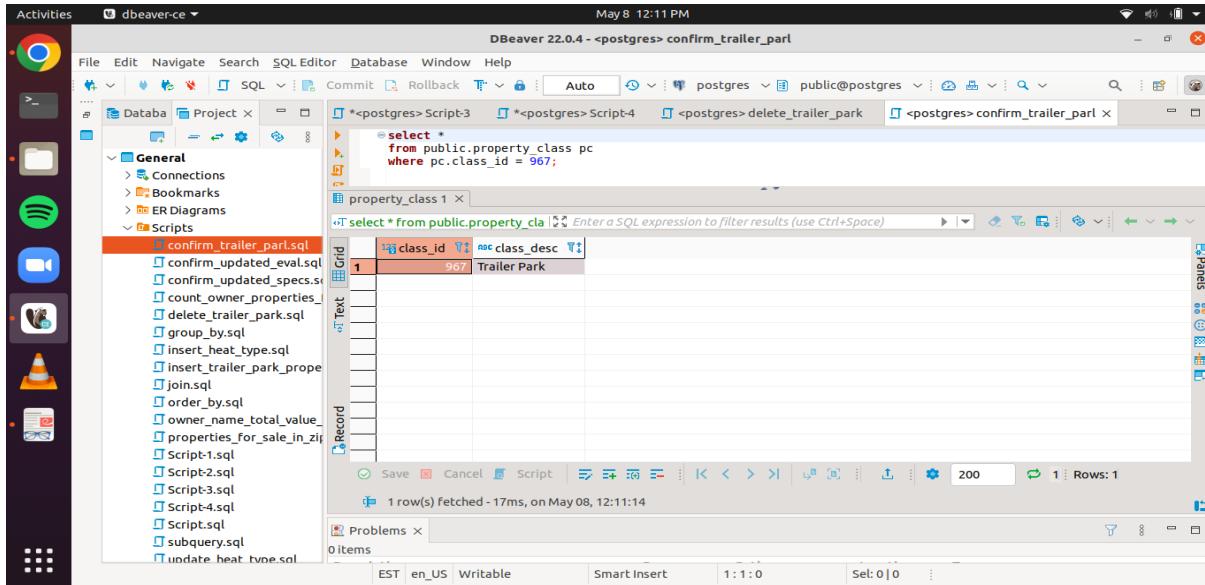
- Toolbar:** Activities, dbeaver-ce, File, Edit, Navigate, Search, SQL Editor, Database, Window, Help.
- Header:** May 8 12:09 PM, DBeaver 22.0.4 - <postgres> Script-4.
- Left Sidebar:** Shows a file browser with various icons like Chrome, Spotify, and a video camera, and a database tree for the 'postgres' database under the 'public' schema, with 'customers' selected.
- SQL Editor:** Contains the following SQL query:

```
select *  
from customers c  
where c.customer_name like '%Mary Trevor%'
```
- Result Grid:** Displays the results of the query in a table format. The table has columns: customer\_id, customer\_name, customer\_email, and customer\_phone. One row is shown, with customer\_id 93834, customer\_name 'Mary Trevor', customer\_email 'hary@harymail.net', and customer\_phone as [NULL].
- Status Bar:** Shows '1 row(s) fetched - 35ms, on May 08, 12:08:47'.
- Bottom Navigation:** EST, en\_US, Writable, Smart Insert, 3:43:69, Sel: 0 | 0.

**delete:** here, we delete the property class, 'Trailer Park'.

**DELETE FROM** property\_class  
**WHERE** class\_desc = 'Trailer Park';

### Before Delete:



The screenshot shows the DBeaver interface with a database connection to 'postgres' on 'public@postgres'. In the SQL Editor tab, a query is run:

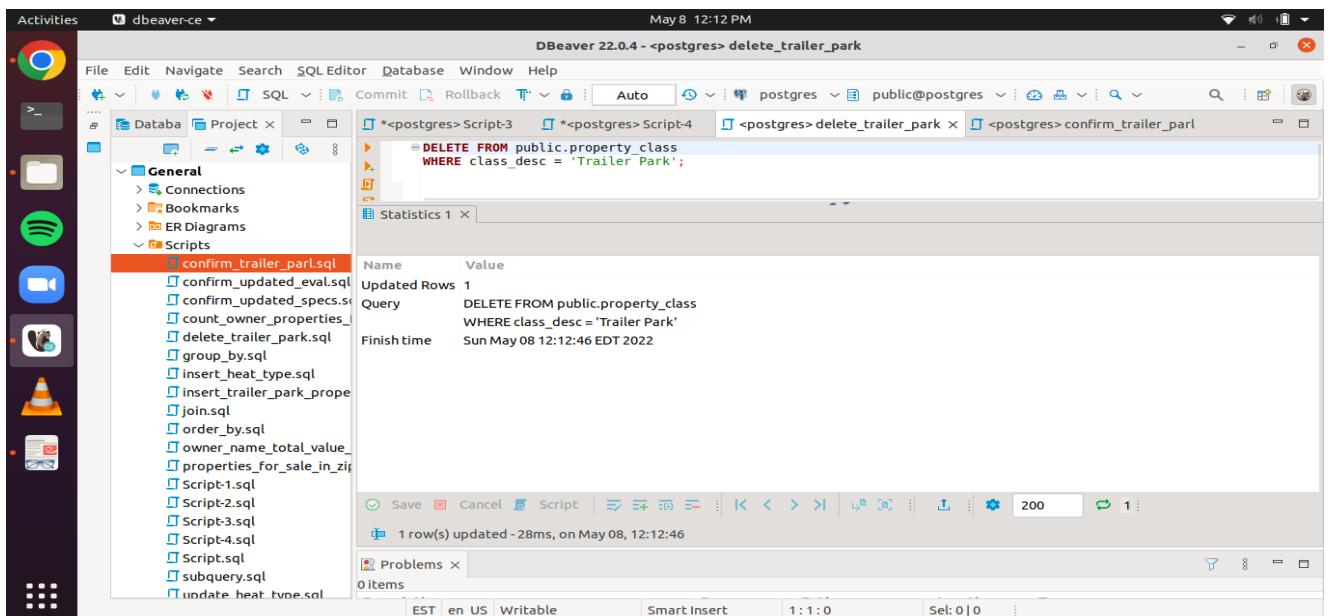
```
*<postgres> Script-3 *<postgres> Script-4 <postgres> delete_trailer_park <postgres> confirm_trailer_park
select *
from public.property_class pc
where pc.class_id = 967;
```

The results are displayed in a table titled 'property\_class 1' with two columns: 'class\_id' and 'class\_desc'. One row is shown:

class_id	class_desc
967	Trailer Park

Below the table, a message indicates: '1 row(s) fetched - 17ms, on May 08, 12:11:14'.

### Successful Delete:



The screenshot shows the DBeaver interface with a database connection to 'postgres' on 'public@postgres'. In the SQL Editor tab, a DELETE query is run:

```
*<postgres> Script-3 *<postgres> Script-4 <postgres> delete_trailer_park <postgres> confirm_trailer_park
DELETE FROM public.property_class
WHERE class_desc = 'Trailer Park';
```

The results are displayed in a table titled 'Statistics 1' with columns 'Name' and 'Value'. The statistics show:

Name	Value
Updated Rows	1
Query	DELETE FROM public.property_class WHERE class_desc = 'Trailer Park'
Finish time	Sun May 08 12:12:46 EDT 2022

Below the table, a message indicates: '1 row(s) updated - 28ms, on May 08, 12:12:46'.

**After Delete:**

The screenshot shows the DBBeaver 22.0.4 interface with a PostgreSQL database connection named 'confirm\_trailer\_park'. The current tab is 'Script-3' which contains the following SQL query:

```
select *  
from public.property_class pc  
where pc.class_id = 967;
```

The results pane displays a single row from the 'property\_class' table:

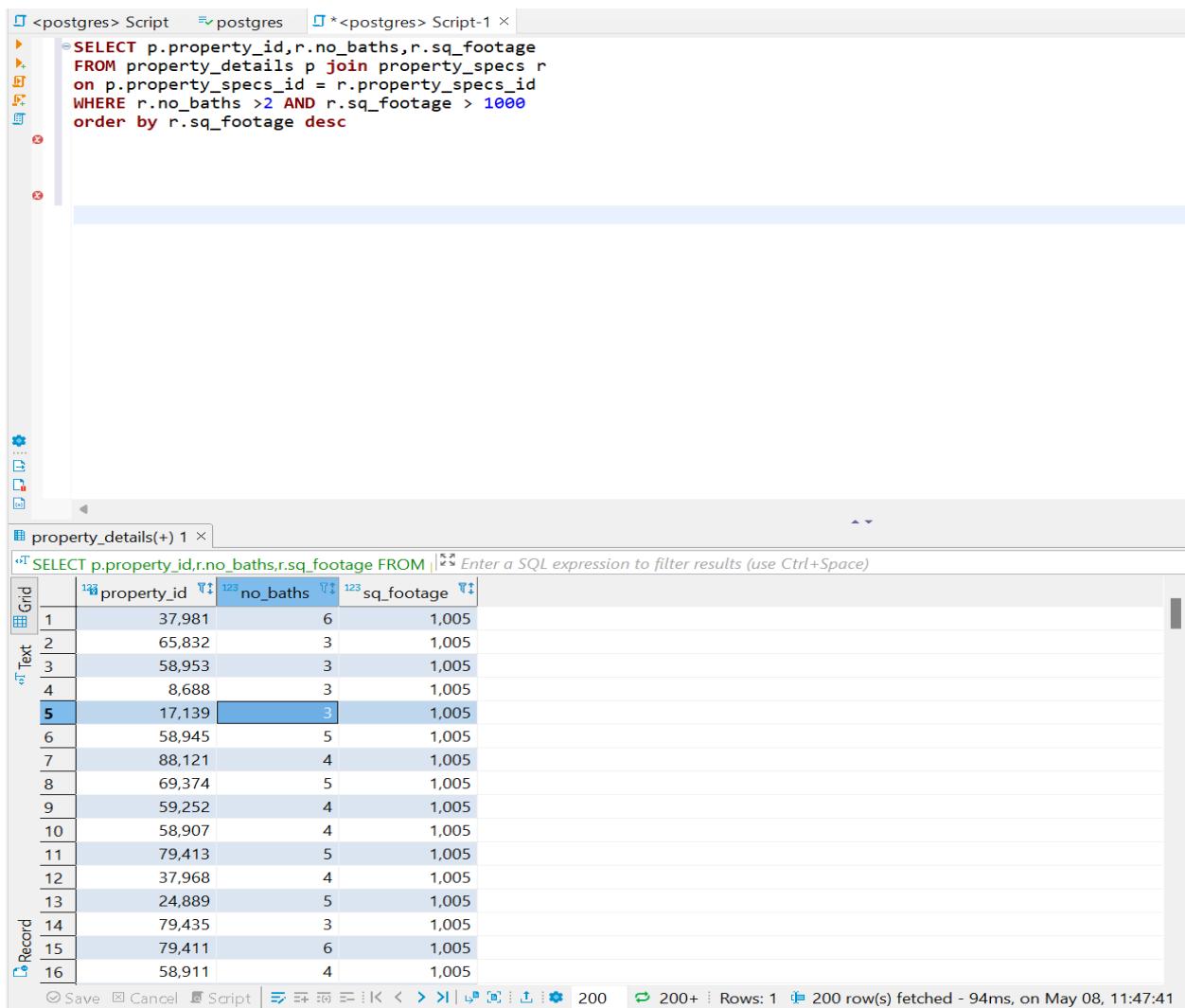
class_id	class_desc
967	Trailer Park

A tooltip 'class\_id: int4' is visible over the 'class\_id' column header. The status bar at the bottom indicates 'No data- 14ms, on May 08, 12:13:06'.

**Views:**

A) Here a view is created for viewing all large properties in the database which have more than two baths and is larger in size (greater than 1000 sq ft in size)

```
CREATE VIEW prop_details AS
SELECT p.property_id, r.no_baths, r.sq_footage
FROM property_details p join property_specs r
on p.property_specs_id = r.property_specs_id
WHERE r.no_baths > 2 AND r.sq_footage > 1000
order by r.sq_footage desc;
```

**View:**


The screenshot shows a PostgreSQL database interface with the following details:

- Query Editor:** Displays the SQL query:
 

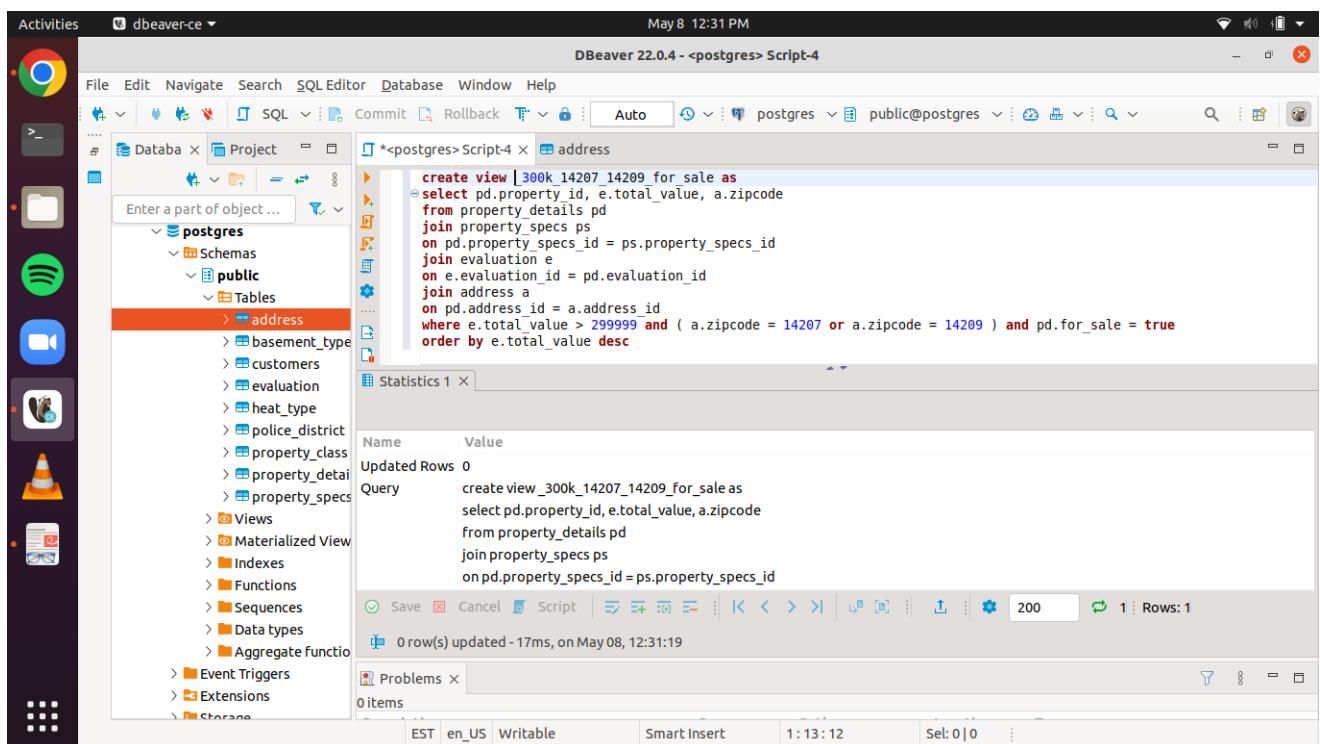
```
SELECT p.property_id, r.no_baths, r.sq_footage
      FROM property_details p join property_specs r
      on p.property_specs_id = r.property_specs_id
      WHERE r.no_baths > 2 AND r.sq_footage > 1000
      order by r.sq_footage desc
```
- Result Grid:** Shows the output of the query in a tabular format. The columns are labeled "property\_id", "no\_baths", and "sq\_footage". The data consists of 16 rows, with the 5th row highlighted in blue.
- Grid Headers:** The first row of the grid contains the column names and their data types.
- Record Number:** The number 16 is displayed at the bottom left of the grid, indicating the total number of rows.
- Status Bar:** At the bottom, it shows "Rows: 1" and "200 row(s) fetched - 94ms, on May 08, 11:47:41".

B.)

Here, a view is created which returns all properties in the database which are for sale and valued at \$300,000 or more, and are in zip code 14207 or 14209

```
create view _300k_14207_14209_for_sale as
select pd.property_id, e.total_value, a.zipcode
from property_details pd
join property_specs ps
on pd.property_specs_id = ps.property_specs_id
join evaluation e
on e.evaluation_id = pd.evaluation_id
join address a
on pd.address_id = a.address_id
where e.total_value > 299999 and (a.zipcode = 14207 or a.zipcode = 14209) and pd.for_sale
= true
order by e.total_value desc;
```

View:



The screenshot shows the DBeaver 22.0.4 interface with the title "DBeaver 22.0.4 - \_300k\_14207\_14209\_for\_sale". The left sidebar displays the database structure for the "postgres" schema, including tables like address, customers, evaluation, heat\_type, police\_district, property\_class, property\_detail, property\_specs, and views such as \_300k\_14207\_14209\_for\_sale. The main panel shows a results grid for the view \_300k\_14207\_14209\_for\_sale, with columns: property\_id, total\_value, and zipcode. The first row is highlighted in orange.

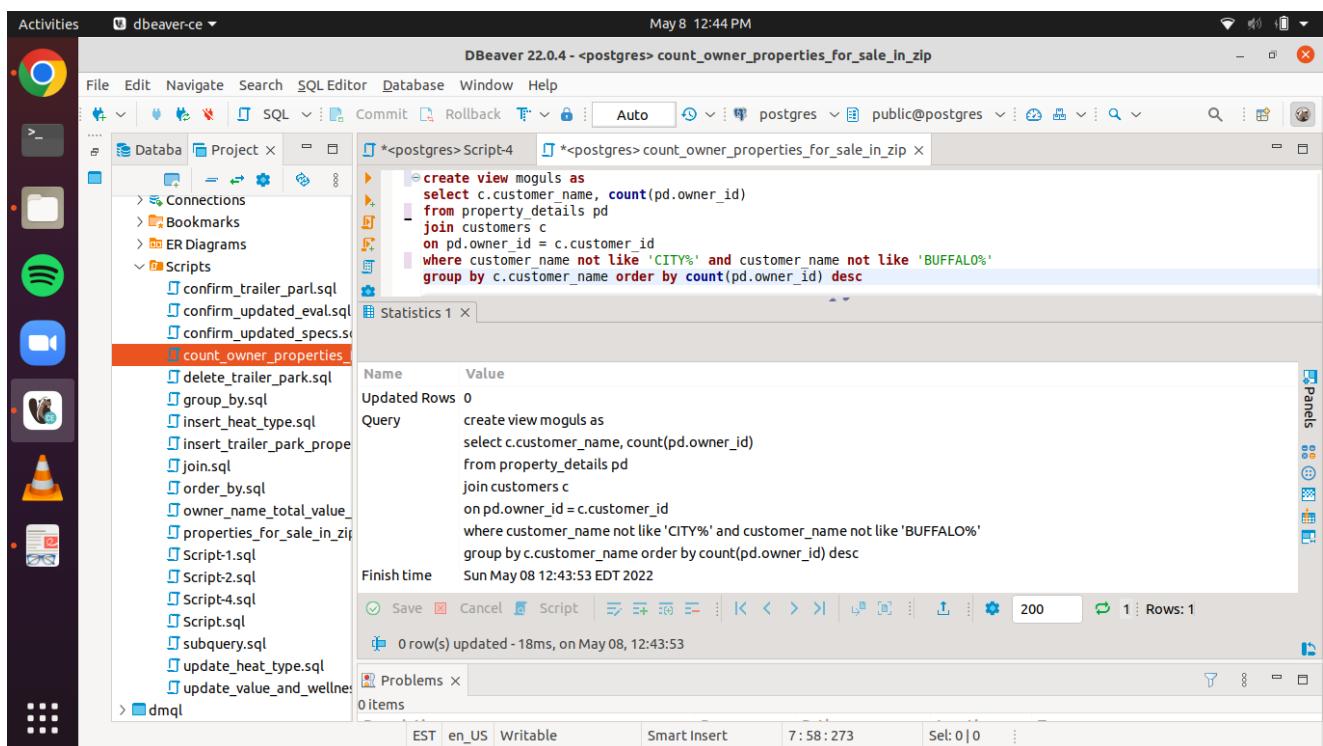
	property_id	total_value	zipcode
1	47,852	10,259,000	14,209
2	93,069	7,200,000	14,209
3	91,539	6,583,000	14,207
4	27,147	6,570,000	14,207
5	20,035	5,843,200	14,209
6	40,491	5,700,000	14,207
7	50,452	5,600,000	14,207
8	79,761	4,218,400	14,207
9	93,293	3,840,000	14,207
10	35,494	3,400,000	14,207
11	75,506	3,100,000	14,207
12	92,558	3,083,000	14,209
13	8,594	3,000,000	14,209
14	90,395	2,800,000	14,209
15	13,333	2,568,700	14,209

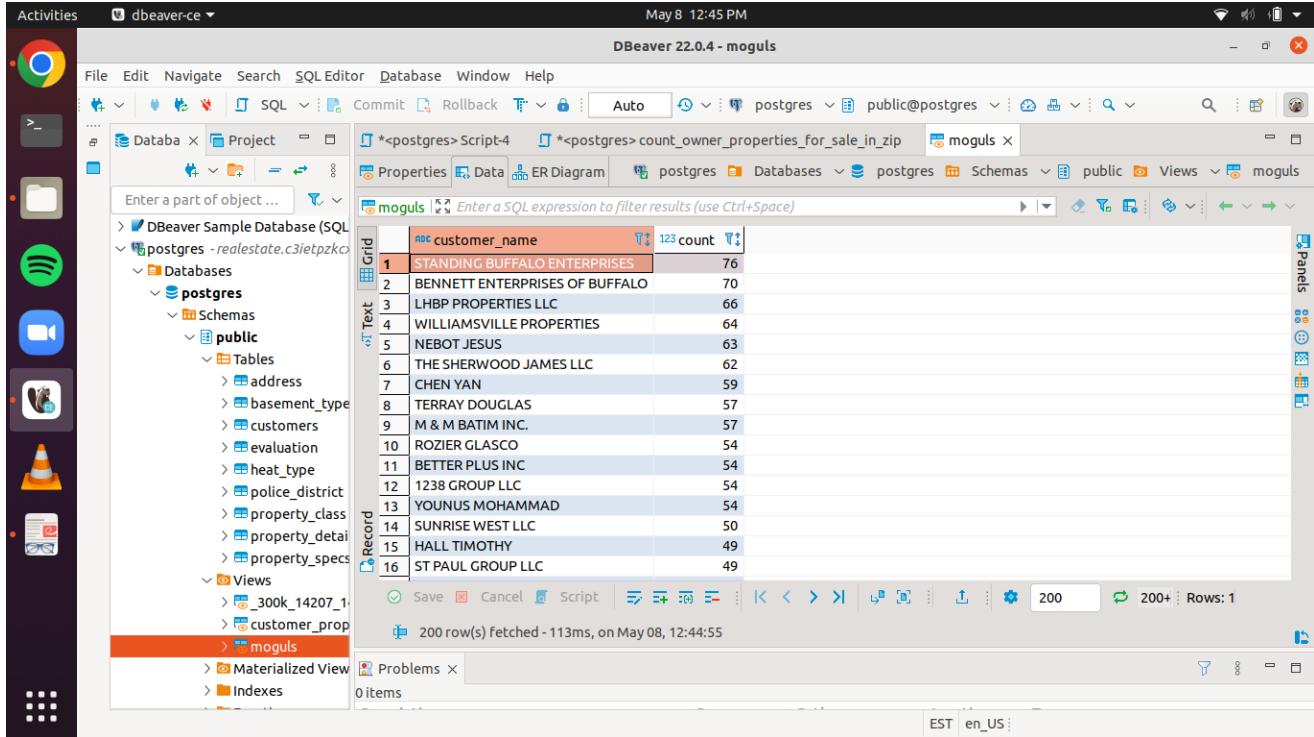
C.)

Here, a view is created which returns the count of all properties owned by customers in the database, whose customer name does not begin with 'city' or 'buffalo', which reflects the private real estate moguls or large real estate agencies in our customer database.

```
create view moguls as
select c.customer_name, count(pd.owner_id)
from property_details pd
join customers c
on pd.owner_id = c.customer_id
where customer_name not like 'CITY%' and customer_name not like 'BUFFALO%'
group by c.customer_name order by count(pd.owner_id) desc;
```

View:





### Execution analysis of problematic queries and fixes:

#### A) Problematic query:

```

select pd.property_id, c.customer_name, a.house_no , a.street , a.zipcode,
e.total_value
from property_details pd join customers c on pd.owner_id =c.customer_id
join address a on pd.address_id =a.address_id
join evaluation e on pd.evaluation_id = e.evaluation_id
order by e.total_value desc;
    
```

Problem: A lot of the customer inquiries would be dependent on their budget and getting the result in some ordering of the total\_value of the properties would be a common occurrence. The query above is one such example.

Without any indexing, the query takes around 3.3 seconds to execute.

Fix: to optimize the execution of all such queries, we implemented indexing on the ordering of total\_value. With indexing in place, the same query can be executed in 1.5 seconds. Hence, the execution time is reduced by 54%.

**Fix Query:**

```
CREATE INDEX evaluation_desc_total_value ON evaluation (total_value DESC NULLS LAST);
```

**RESULT:****Query cost before fix:**

**Runtime:** 3 secs 299msecs

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the database structure for 'postgres' under 'DMQL Real Estate'. The main area shows the results of a query on the 'evaluation' table. The table has columns: property\_id, customer\_name, house\_no, street, zipcode, and total\_value. The results show 27 rows of data. A message at the bottom right of the results pane says: "Successfully run. Total query runtime: 3 secs 299 msec. 91395 rows affected."

property_id	customer_name	house_no	street	zipcode	total_value
1	PAWLICK EDWARD J	1	SEYMOUR H KNOX III	14203	145851800
2	JONES YOLANDA	3425	MAIN	14214	85557000
3	CITY OF BUFFALO	338	GRIDER	14215	83767400
4	WILLIAMS DOUGLAS	257	GENESEE WEST	14202	58000000
5	BAEZ ARTURO R	3449	BAILEY	14215	57680900
6	SCHIFANO SALVATORE	100	HIGH	14203	56026000
7	MBINGA GERVAIS	951	MAIN	14203	55800000
8	ANGEL QUINTON & W	75	MAIN	14203	52000000
9	YOUNG RAYMOND I	929	MAIN	14203	49100000
10	VARA CHARLES A	400	FOREST	14213	46822000
11	LARSON SCOTT	875	ELLICOTT	14203	44109000
12	KHOKHAR CONTRACTING	65	NIAGARA SQ	14202	40910200
13	PUTMAN LAURIE J	1001	MAIN	14203	39150000
14	CITY BUFFALO PERFECTING TITLE	565	ABBOTT	14220	35000000
15	PARSHALL DANIEL T	2157	MAIN	14214	34685300
16	RIVERA ASHLEY M	263	WASHINGTON	14203	32793900
17	CITY BUFFALO PERFECTING TITLE	60	CARLTON	14203	31958200
18	MARRANZINI JOHN	140	GENESEE	14203	30300000
19	PAPAJ KEVIN	372	MICHIGAN	14203	30000000
20	BEGUM RASHEDA	92	FERRY WEST	14213	29315800
21	FEELEY WILLIAM A	130	ELMWOOD SOUTH	14201	29000000
22	CITY BUFFALO PERFECTING TITLE	95	FRANKLIN	14202	28477200
23	SNAJCZUK JAMES R	875	ELLICOTT	14203	27468000
24	THOMAS MARIE L	250	DELAWARE	14202	27117330
25	PARK RITZ DEVELOPERS LLC	726	EXCHANGE	14210	26000000
26	COLLINS SCOTT R	960	WASHINGTON	14203	25955000
27	SUAREZ RAMON JUAN JR	153	FRAN	14203	25955000

**Query cost after fix:****Runtime:** 1second 592 msec

The screenshot shows the PgAdmin 4 interface. On the left, the 'Servers' tree view is expanded to show 'DMQL Real Estate' and its 'Databases' (including 'postgres'). The 'Tables' node under 'postgres' is also expanded, showing nine tables: address, basement\_type, customers, evaluation, heat\_type, police\_district, property\_class, property\_details, and property\_id. The main window displays a table titled 'postgres/postgres@DMQL Real Estate'. The table has columns: property\_id, customer\_name, house\_no, street, zipcode, and total\_value. The data consists of 91395 rows. A message at the bottom right of the table area says 'Successfully run. Total query runtime: 1 secs 592 msec. 91395 rows affected.'

property_id	customer_name	house_no	street	zipcode	total_value
1	PAWLIK EDWARD J		SEYMORE H KNOX III	14203	145851800
2	JONES YOLANDA	3425	MAIN	14214	85557000
3	CITY OF BUFFALO	338	GRIDER	14215	83767400
4	WILLIAMS DOUGLAS	257	GENESEE WEST	14202	58000000
5	BAEZ ARTURO R	3449	BAILEY	14215	57680900
6	SCHIFANO SALVATORE	100	HIGH	14203	56026000
7	MBINGA GERVAIS	951	MAIN	14203	55800000
8	ANGEL QUINTON & W	75	MAIN	14203	52000000
9	YOUNG RAYMOND I	929	MAIN	14203	49100000
10	VARA CHARLES A	400	FOREST	14213	46822000
11	LARSON SCOTT	875	ELLIOTT	14203	44109000
12	KHOKHAR CONTRACTING	65	NIAGARA SQ	14202	40910200
13	PUTMAN LAURIE J	1001	MAIN	14203	39150000
14	CITY BUFFALO PERFECTING TITLE	565	ABBOTT	14220	35000000
15	PARSHALL DANIEL T	2157	MAIN	14214	34685300
16	RIVERA ASHLEY M	263	WASHINGTON	14203	32789300
17	CITY BUFFALO PERFECTING TITLE	60	CARLTON	14203	31958200
18	MARRANZINI JOHN	140	GENESEE	14203	30300000
19	PAPAJ KEVIN	372	MICHIGAN	14203	30000000
20	BEGUM RASHEDA	92	FERRY WEST	14213	29315800
21	FEELEY WILLIAM A	130	ELMWOOD SOUTH	14201	29000000
22	CITY BUFFALO PERFECTING TITLE	95	FRANKLIN	14202	28477200
23	SNAJCZUK JAMES R	875	ELLIOTT	14203	27468000
24	THOMAS MARIE L	250	DELAWARE	14202	27117330
25	PARK RITZ DEVELOPERS LLC	726	EXCHANGE	14210	26000000
26	COLLINS SCOTT R	960	WASHINGTON	14203	25955000
27	SUAREZ RAMON JUAN JR	153	FRAN		✓ Successfully run. Total query runtime: 1 secs 592 msec. 91395 rows affected.

**B) Problematic query:**

```
SELECT p.property_id, e.total_value, e.wellness_rating  
FROM property_details p JOIN evaluation e  
ON p.evaluation_id = e.evaluation_id  
WHERE p.class_id = (SELECT c.class_id FROM property_class c WHERE  
lower(c.class_desc) LIKE '%two family dwelling%')  
ORDER BY e.total_value DESC, e.wellness_rating DESC;
```

Problem: Some of the tables are of type varchar and contain string data. In order to perform case-insensitive comparison, we need to call the function lower() for each record in the table. This increases the overhead and the query takes a long time to execute. The query above is one such example.

Without any indexing, the query takes around 1.5 seconds to execute.

Fix: To fix this, we implemented indexing on the lower() function of such attributes. With indexing in place, the same query can be executed in 0.4 seconds. Hence, the execution time is reduced by 70%.

**Fix Query:**

```
CREATE INDEX property_class_lower_class_desc_idx ON property_class (lower(class_desc));
```

**RESULT:****Query cost before fix:****Runtime: 1 second 54 msec**

Scratch Pad    Query Editor    Query History    Explain    Messages    Notifications    Data Output

	property_id	total_value	wellness_rating
1	16084	795000	4
2	3762	775000	3
3	18618	750000	3
4	4427	750000	3
5	47302	575000	3
6	2846	525000	4
7	21398	525000	3
8	76309	517000	4
9	62254	500000	3
10	69241	492000	4
11	71204	440000	3
12	85420	425000	3
13	93282	425000	3
14	64461	425000	3
15	21090	415000	3
16	87746	400000	4
17	29472	395000	3
18	61048	380000	4
19	84366	380000	4
20	18222	375000	3
21	37988	365000	3
22	49781	360000	3
23	13277	350000	4
24	72112	350000	4
25	86162	350000	3
26	74429	350000	3
27	7860	345000	3
...	...	...	...

✓ Successfully run. Total query runtime: 1 secs 54 msec. 26041 rows affected.

**Query cost after fix:****Runtime: 0 sec 448 msec**

Scratch Pad    Query Editor    Query History    Explain    Messages    Notifications    Data Output

	property_id	total_value	wellness_rating
1	16084	795000	4
2	3762	775000	3
3	18618	750000	3
4	4427	750000	3
5	47302	575000	3
6	2846	525000	4
7	21398	525000	3
8	76309	517000	4
9	62254	500000	3
10	69241	492000	4
11	71204	440000	3
12	85420	425000	3
13	93282	425000	3
14	64461	425000	3
15	21090	415000	3
16	87746	400000	4
17	29472	395000	3
18	61048	380000	4
19	84366	380000	4
20	18222	375000	3
21	37988	365000	3
22	49781	360000	3
23	13277	350000	4
24	72112	350000	4
25	86162	350000	3
26	74429	350000	3
27	7860	345000	3
...	...	...	...

✓ Successfully run. Total query runtime: 448 msec. 26041 rows affected.

**C) Query:**

```
INSERT INTO heat_type  
(heat_desc)  
VALUES('Fusion');
```

```
INSERT INTO heat_type  
(heat_desc)  
VALUES('FUSION');
```

Problem: Some of the tables are of type varchar and contain string data. In such columns, we can add multiple varchar values which only differ in the upper/ lowercase but essentially have the same value and meaning. This issue can not be fixed by just adding the UNIQUE constraint to the column since “abc” and “ABC” are treated as two different varchar values.

Fix: To fix this, we implemented UNIQUE indexing on the lower() function of such attributes.

With unique indexing in place, the case- insensitive uniqueness is applied on such columns

**Fix Query:**

```
CREATE UNIQUE INDEX heat_type_lower_heat_desc_idx ON heat_type (lower(heat_desc));
```

**RESULT:**

Now duplicate heat type values with different cases will not be allowed. This prevents duplicate heat type values from being created.

---

```
ERROR: duplicate key value violates unique constraint "heat_type_lower_heat_desc_idx"  
DETAIL: Key (lower(heat_desc::text))=(fusion) already exists.  
SQL state: 23505
```

**Contribution:****Aaron Flierl:**

- Dataset acquisition and research
- Queries and Views
- Contributions to milestone 1 and 2 report
- Database Schema Design
- BCNF discussion

**Harman Singh:**

- Database Schema Design.
- DDL scripts.
- Dataset analysis and correction of invalid data.
- Scripts to load data into the tables.
- Implemented indexing in database.
- Database optimization.
- Identified problematic queries & their fixes
- Contributions to milestone 1 and 2 report

**Aditya sai:**

- Contributions to milestone 1 and 2 report
- Contribution to database schema research and design
- Implemented views
- Functional dependencies and BCNF of each relation

**References:**

<https://data.buffalony.gov/Government/2019-2020-Assessment-Roll/kckn-jafw>

**Ullman, J. and Widom, J., 2002.** *A first course in database systems*. Upper Saddle River, NJ: Prentice Hall.