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.

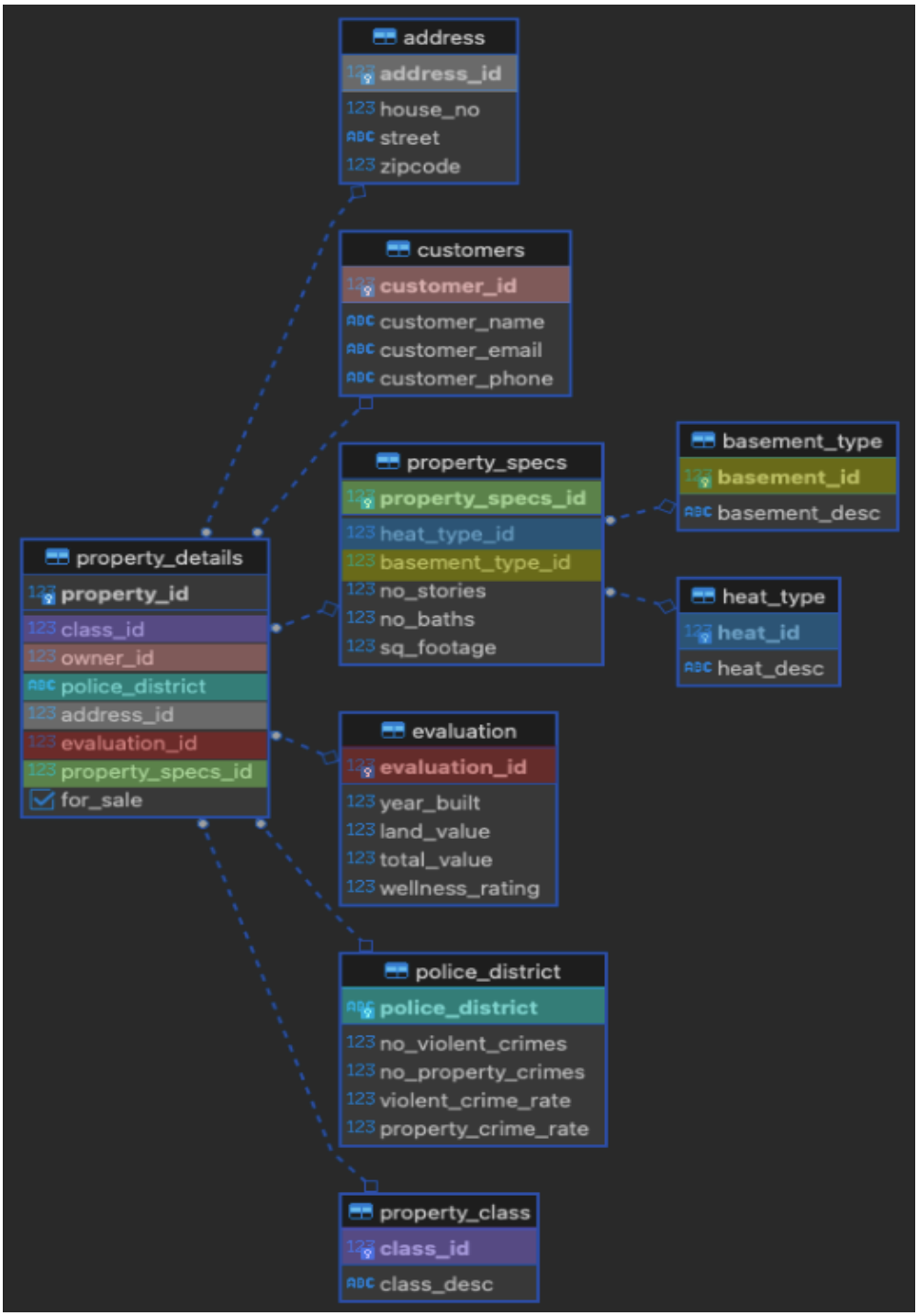
- Additional research was performed by all team members to find out what additional data could be acquired to improve the experience of database users. Examples of this data include the 'violent_crime_rate' and 'property_crime_rate' attributes of the *police_district* relation, as well as the relations *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:

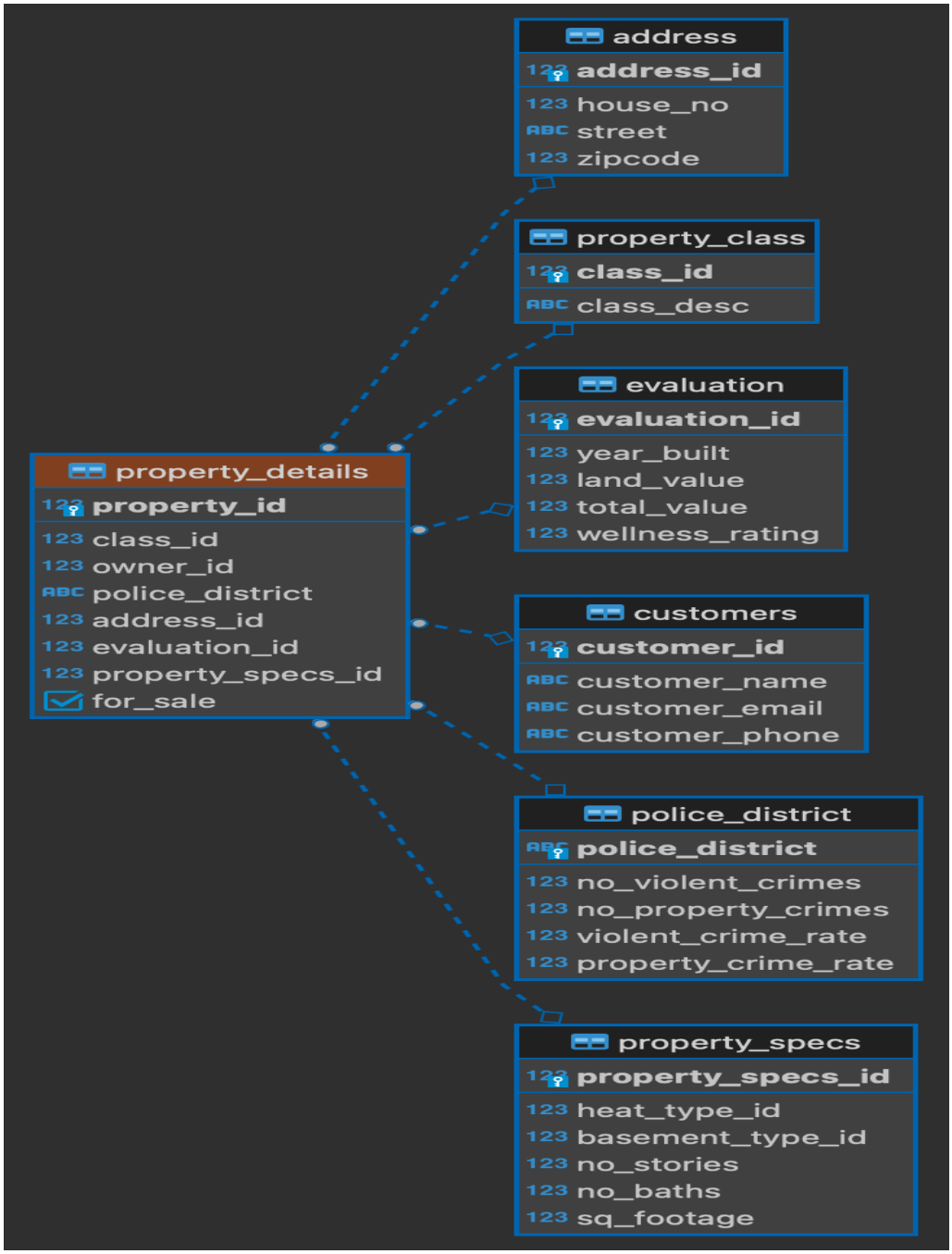


The Primary key is the first attribute in the table diagram and is denoted by the ‘key’ symbol right next to it.

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:

property_id → (class_id, owner_id, police_district, address_id, evaluation_id, property_specs_id, for_sale)

BCNF:

This relation is in BCNF as:

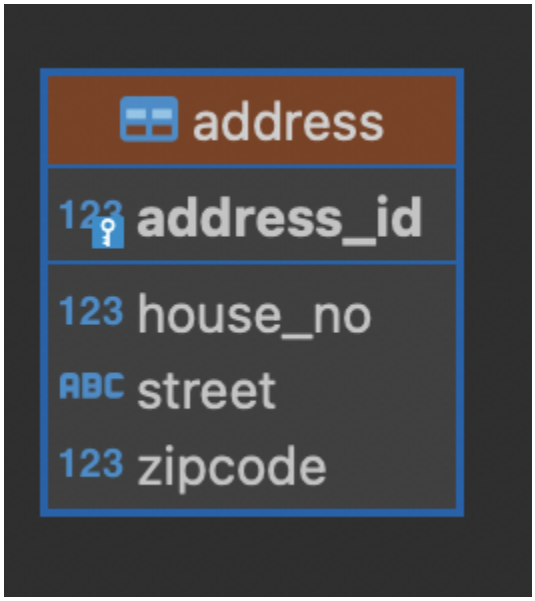
- The relation is clearly in 1NF as there are no multi-valued attributes, no domain changes, the order in which the data is stored does not have significance, and each attribute name is unique in the relation.

- 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:

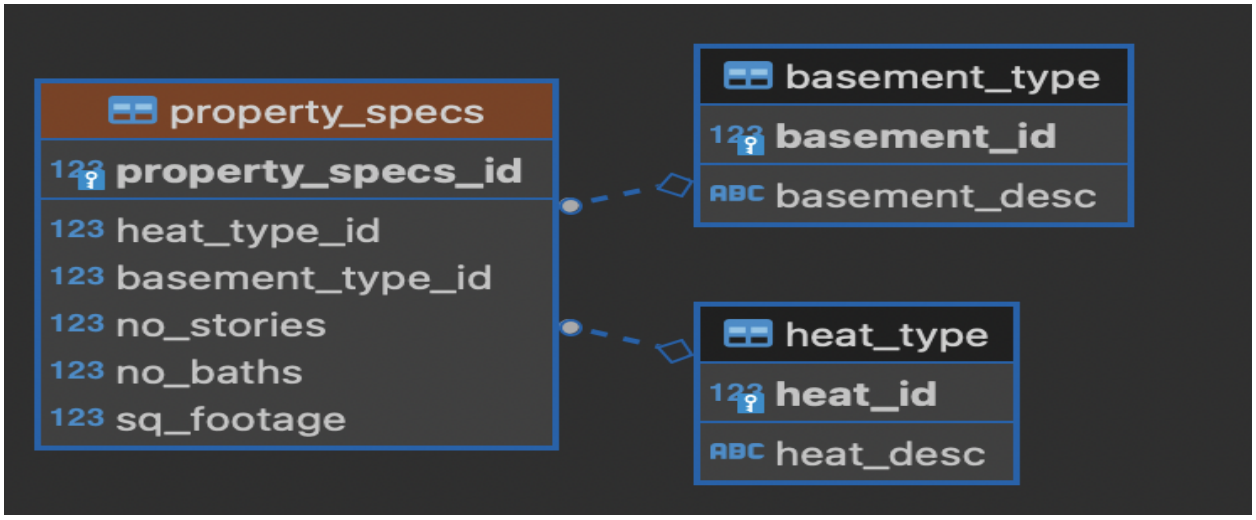
address_id → house_no, street, zipcode

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:

property_specs_id → heat_type_id,basement_type_id,no_stories,no_baths,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:

basement_type → 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:

heat_id → 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.

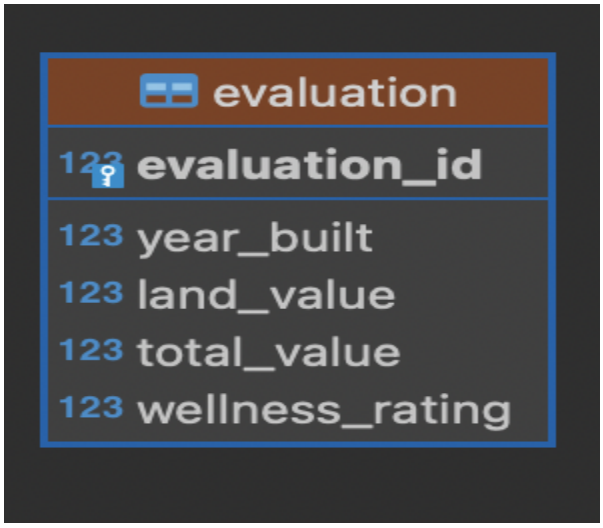
- It is in 2NF, as it is in 1NF and the primary key set (heat_id) consists of only a single attribute.

- 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 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.



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:

evaluation_id → year_built,land_value,total_value,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.

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.

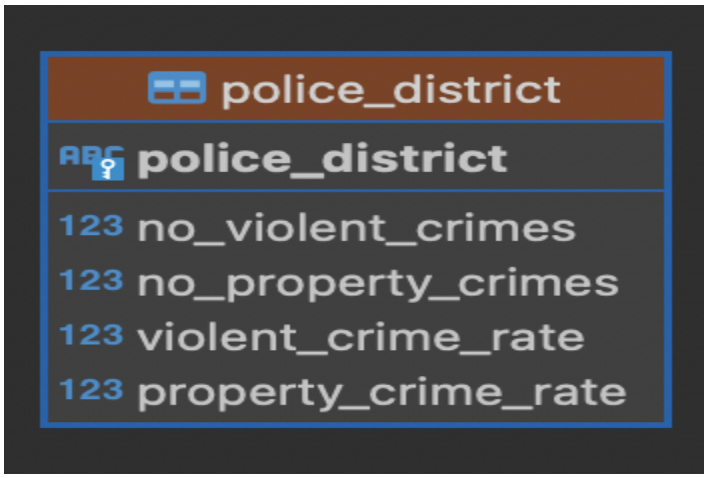
- It is in 2NF, as it is in 1NF and the primary key set (class_id) consists of only a single attribute.

- 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.



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

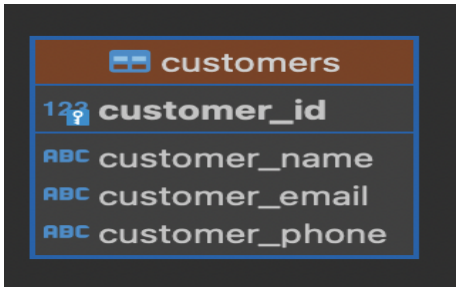
BCNF:

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.



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:

Customer_id → customer_name, customer_email,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.

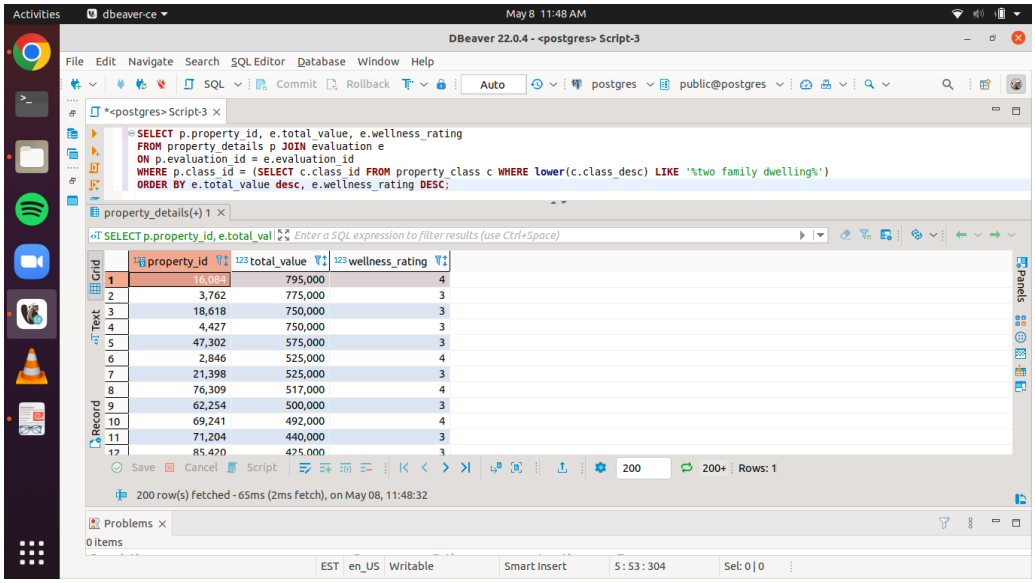
Queries:

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;
```

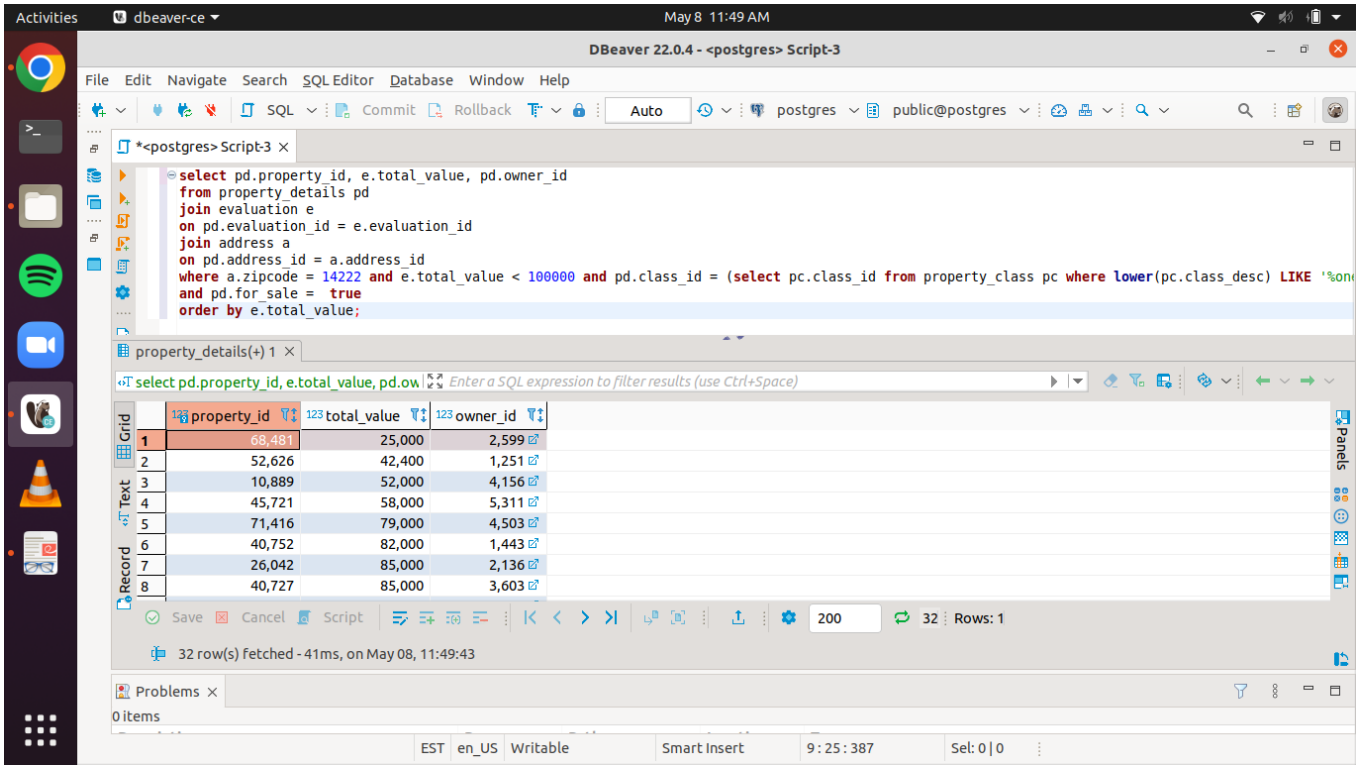
Result:



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;
```

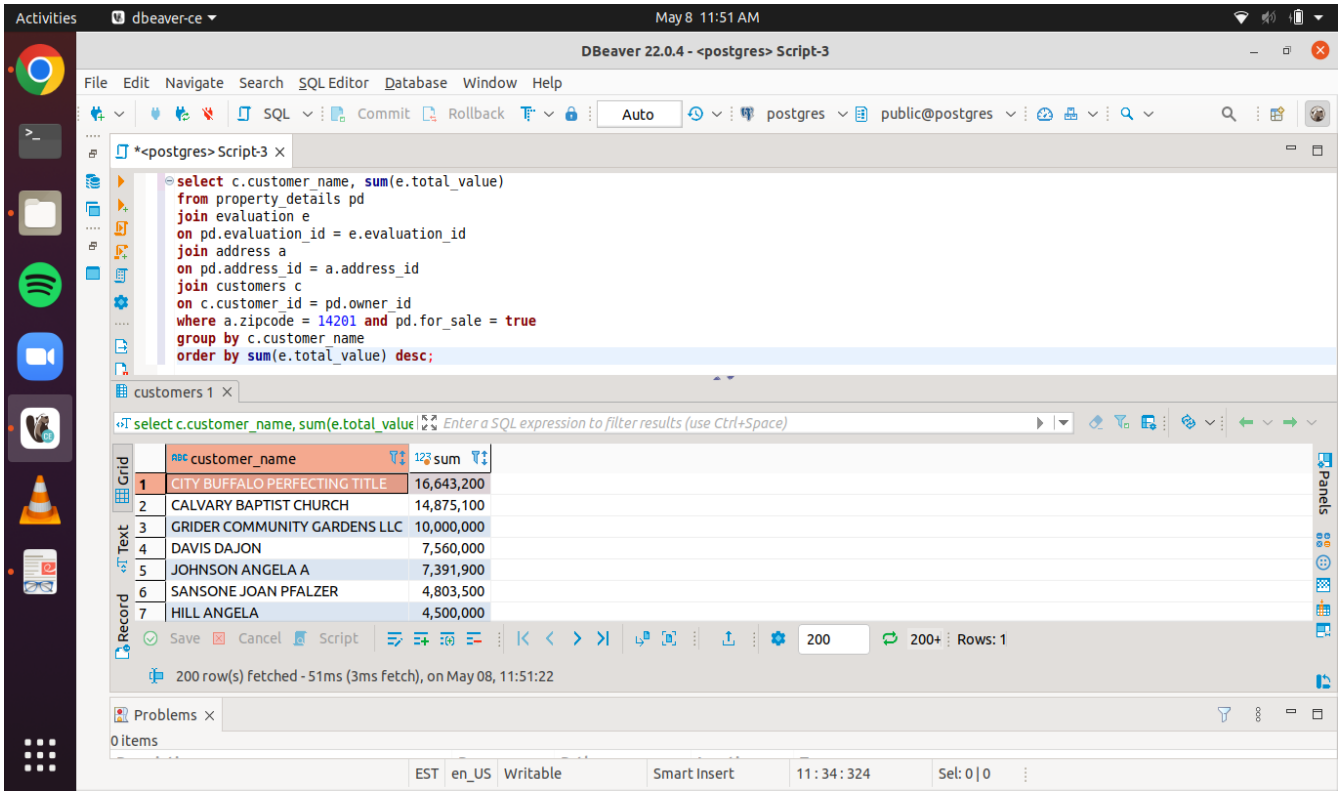
Result:



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;
```

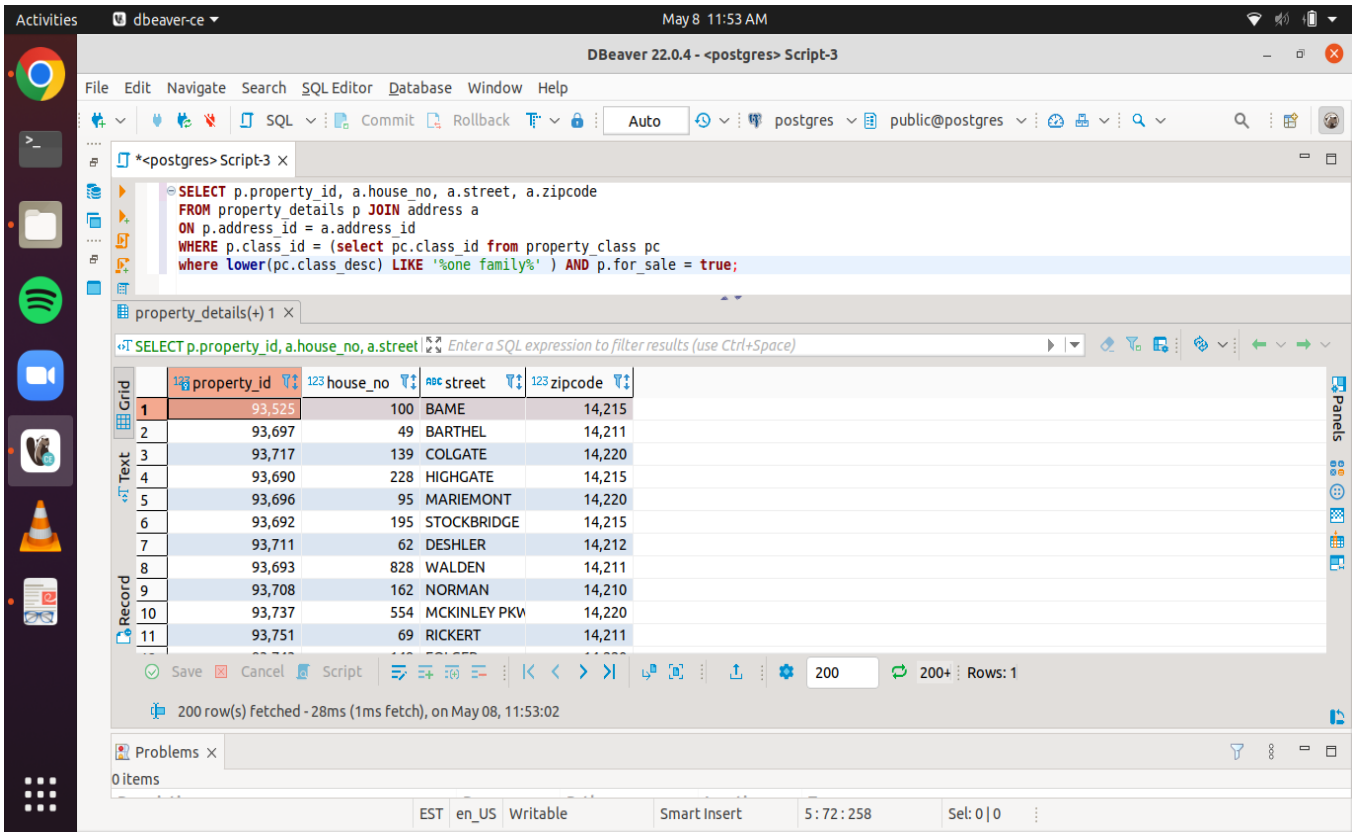
Result:



***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:

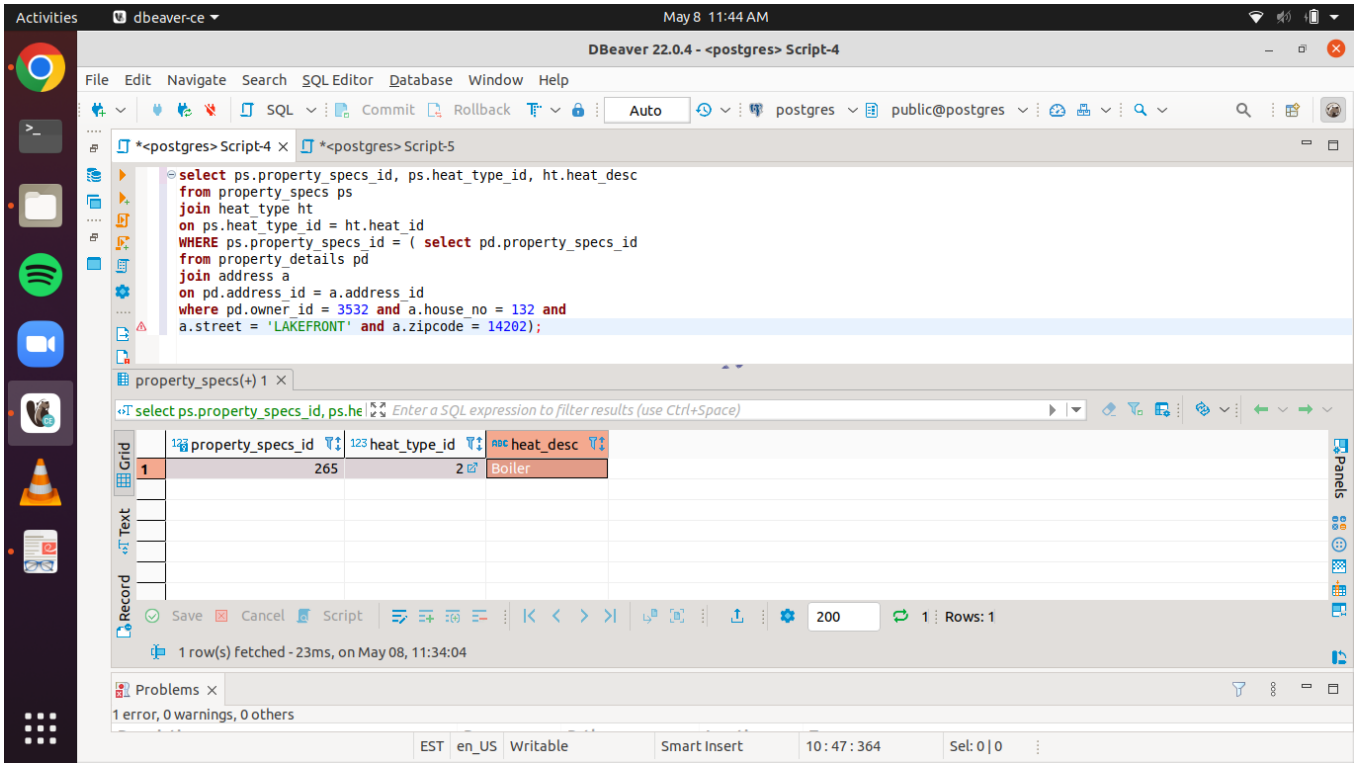


Insert/Update/Delete statements:

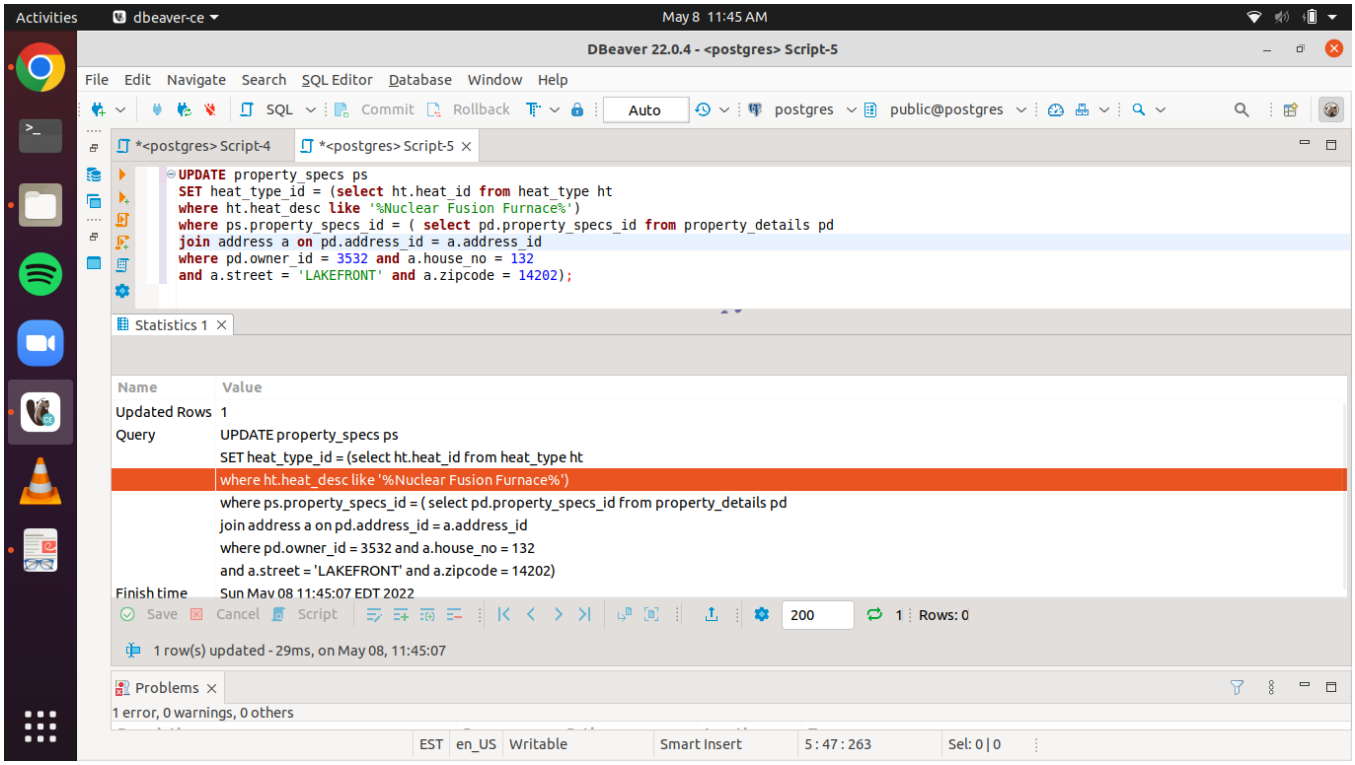
***Update with multiple joins and subquery*:** Here, we are updating the heat_type_id of a specific property to have a new heat type, Nuclear Fusion Furnace, which has a heat_type_id of 6, which was assigned automatically, as it is a value with type SERIAL, and also guaranteed and required to be unique. 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);
```

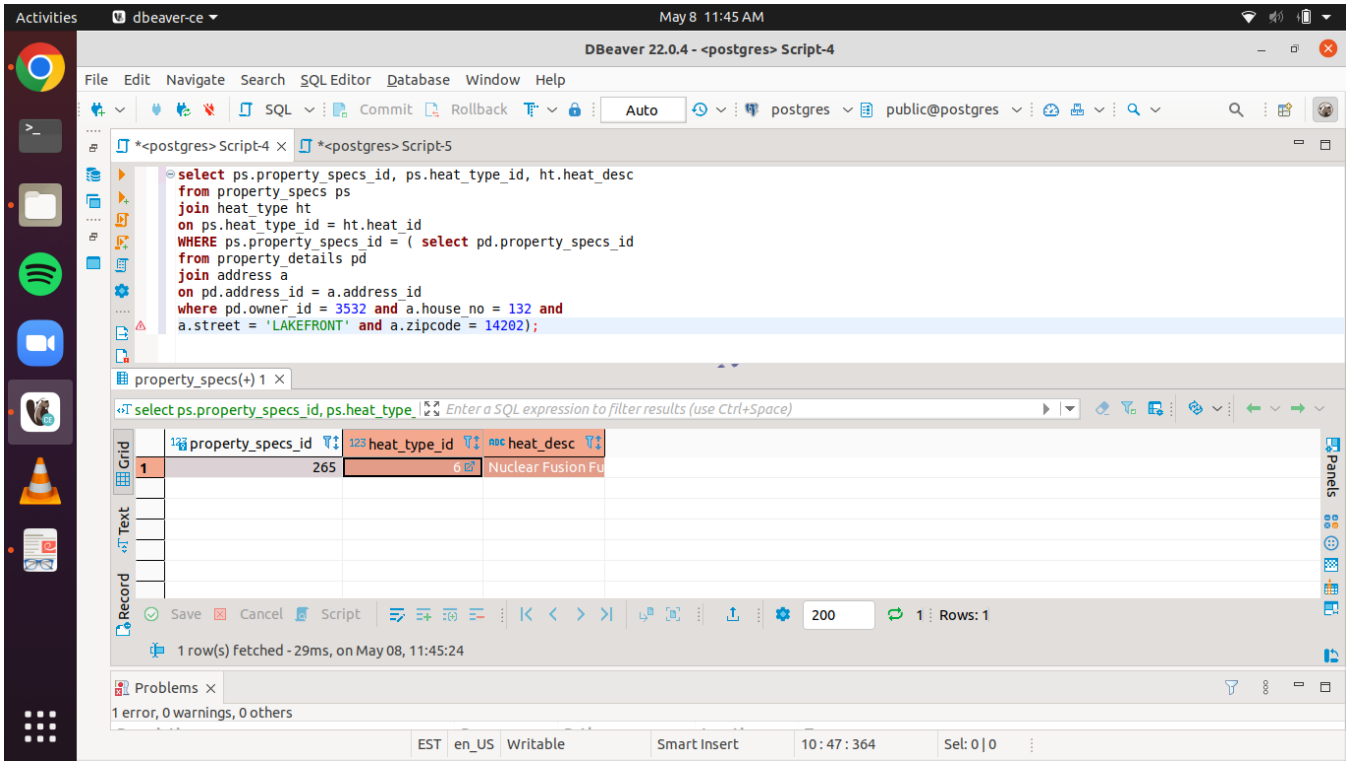
Before Update:



Successful Update:



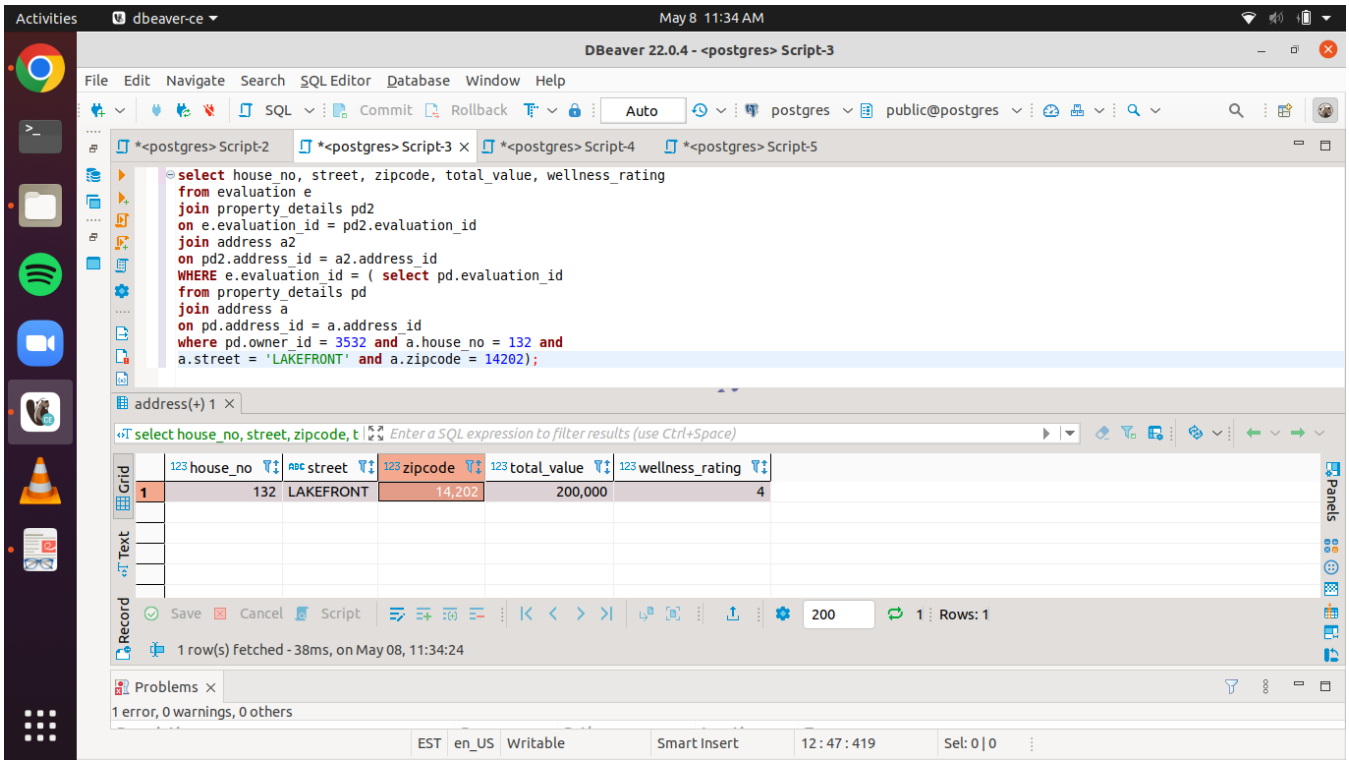
After Update:



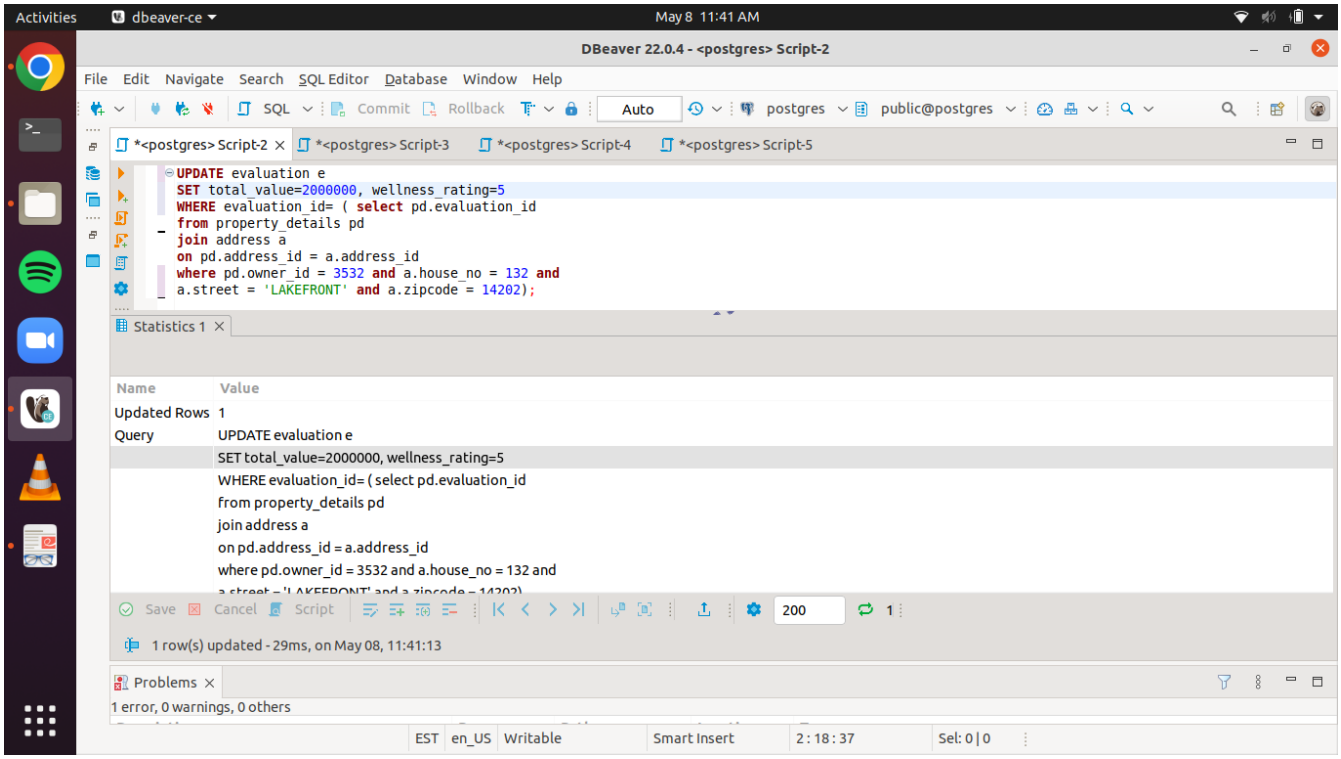
***update*:** Here, we are updating the same property shown in the query above, which has evaluation_id = 265. Since it has a new furnace, the wellness rating has increased from 4 to 5. Also, the total value was updated to 2,000,000.

```
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);
```

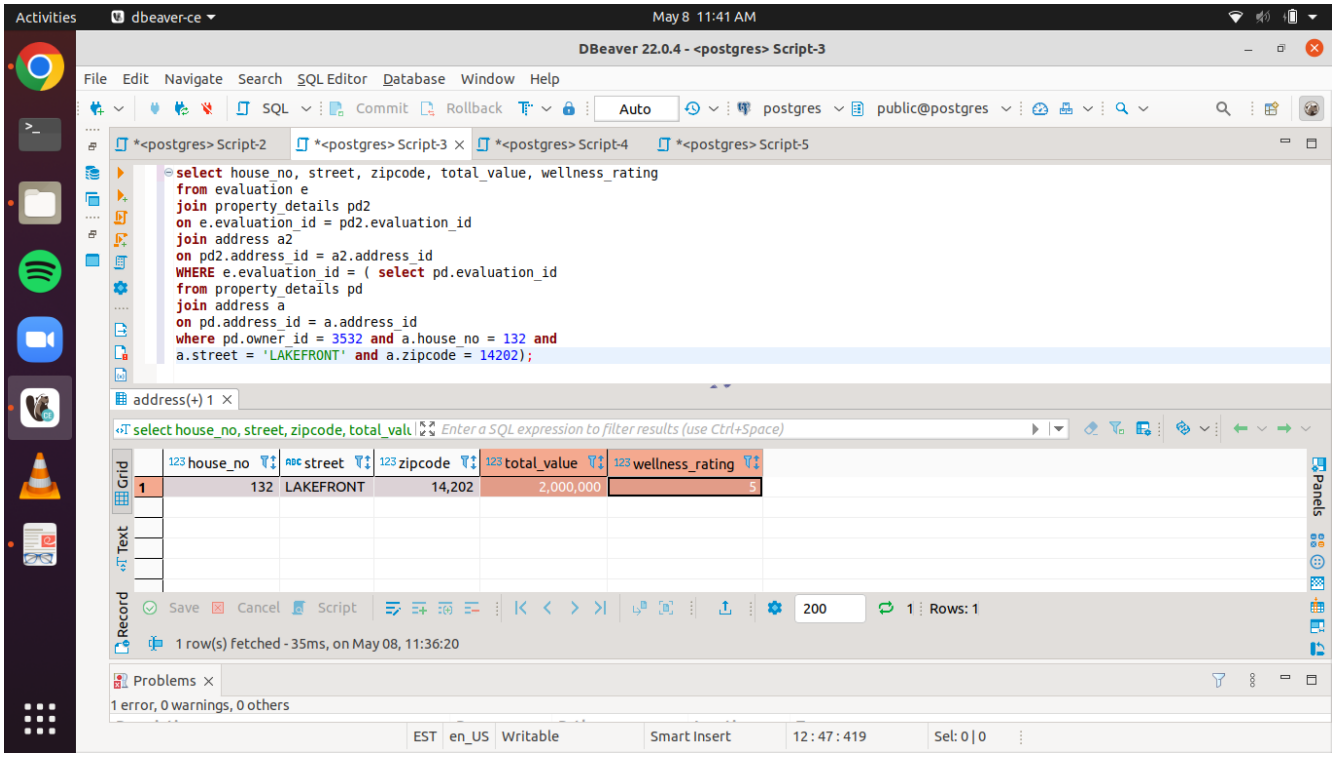
Before Update:



Successful Update:



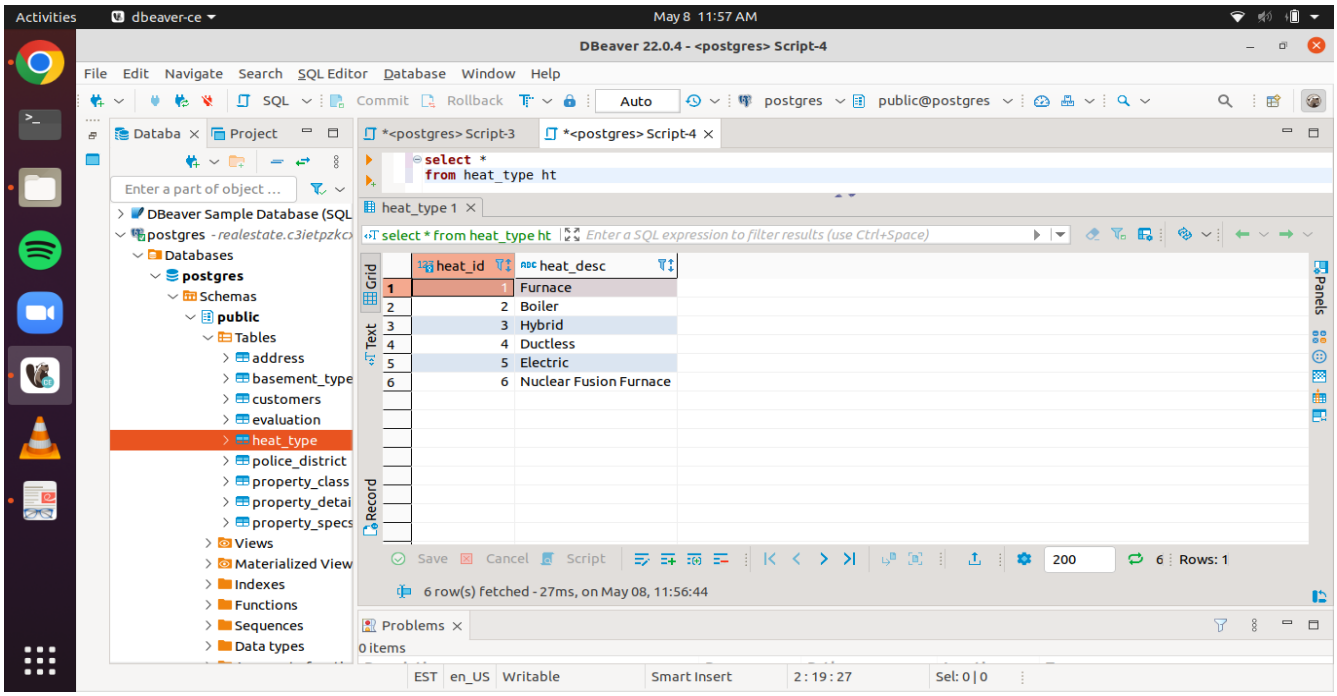
After Update:



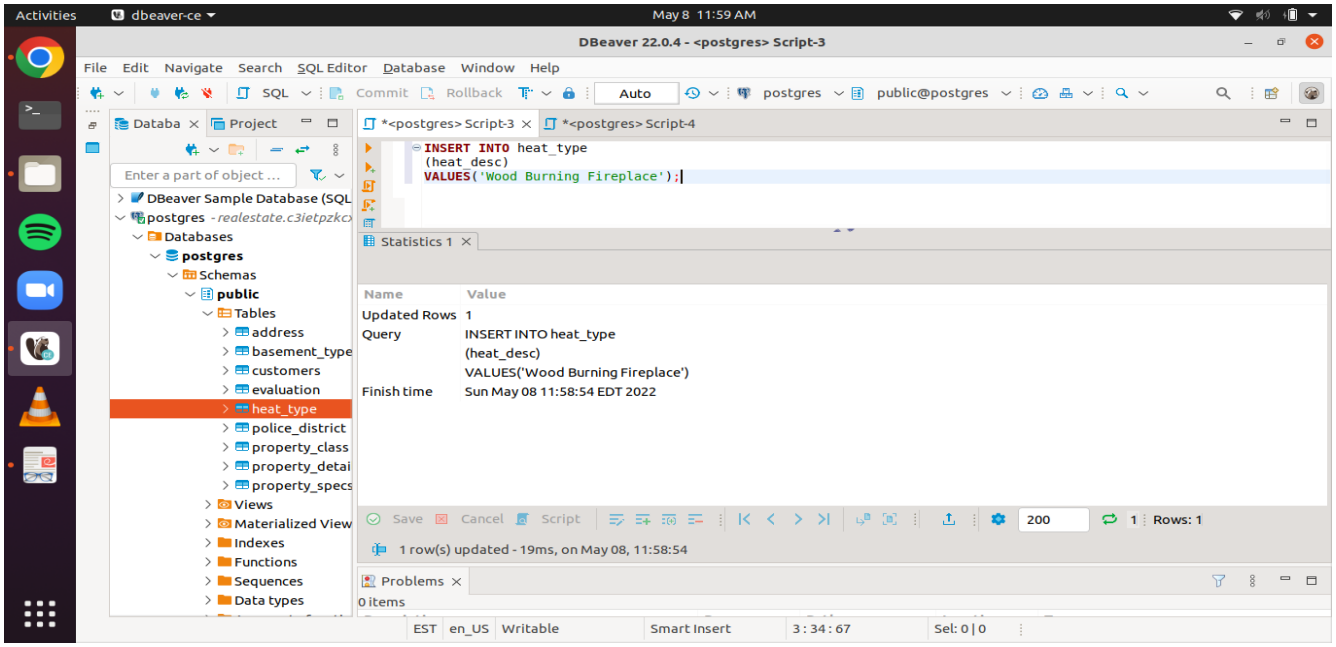
insert: here, we are inserting a new heat type, Wood Burning Fireplace.

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

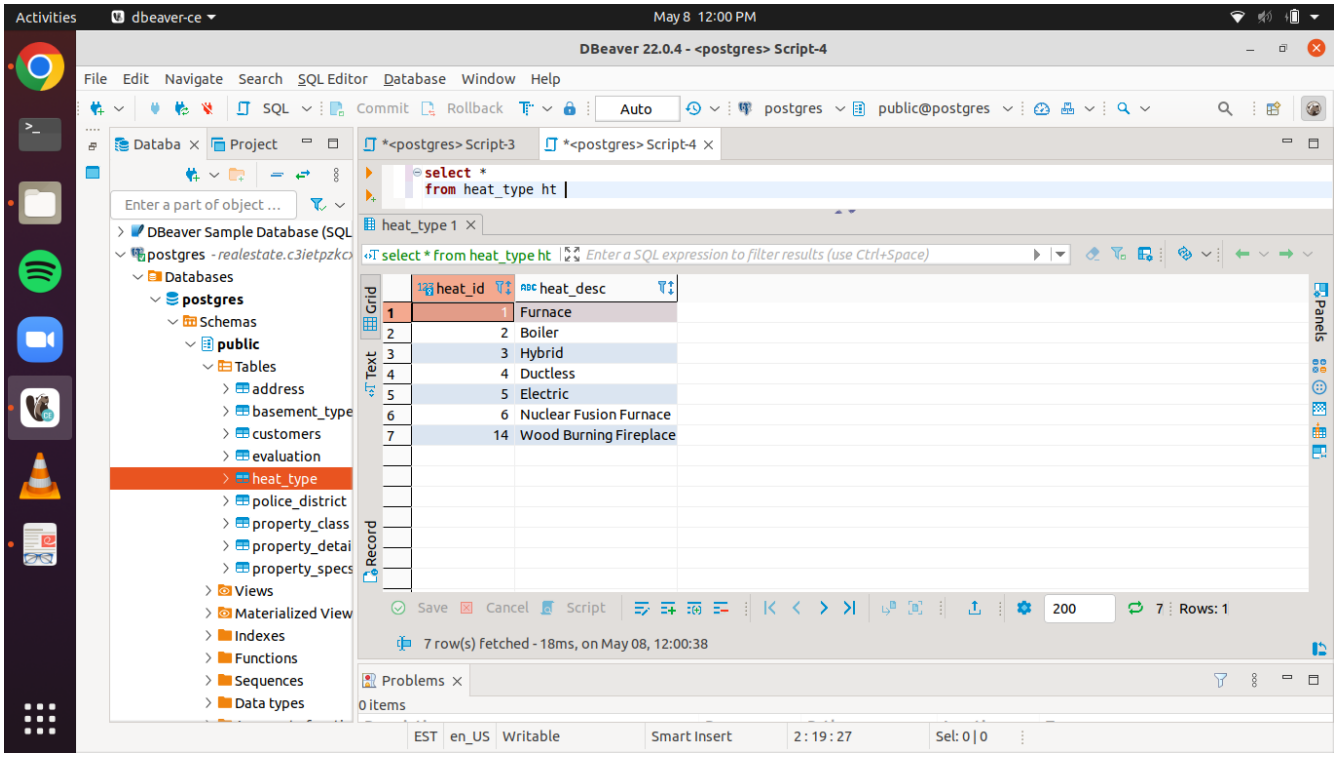
Before Insert:



Successful insert:



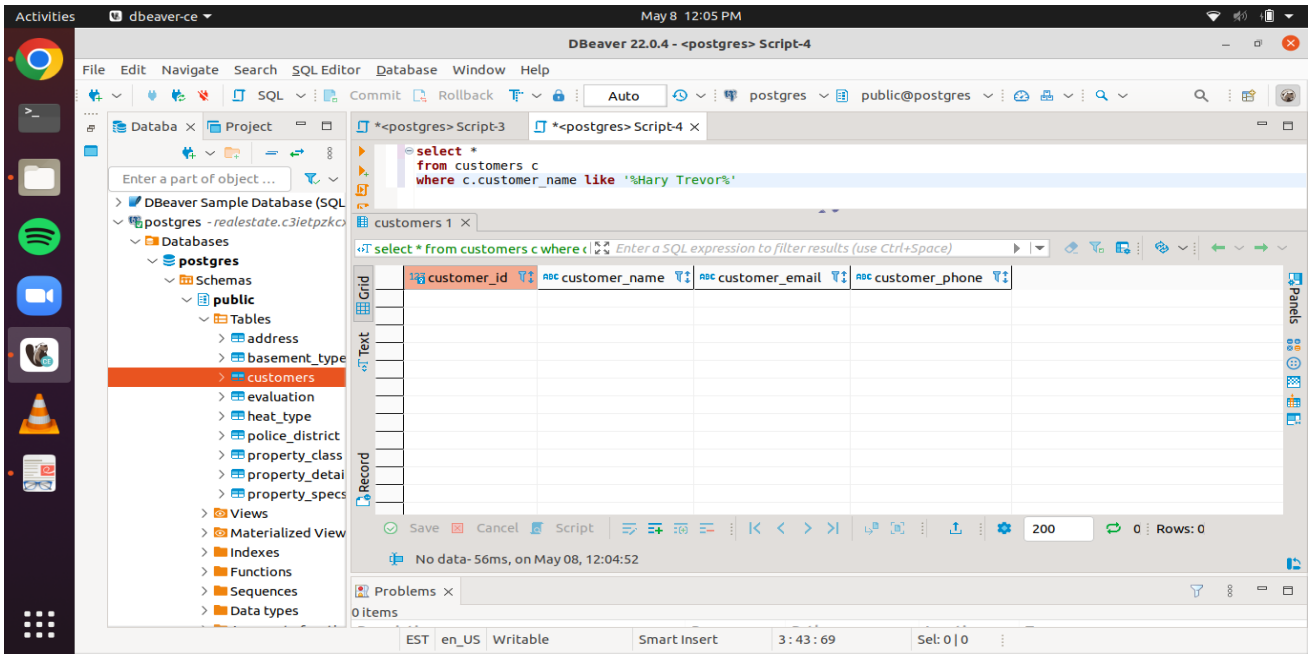
After Insert:



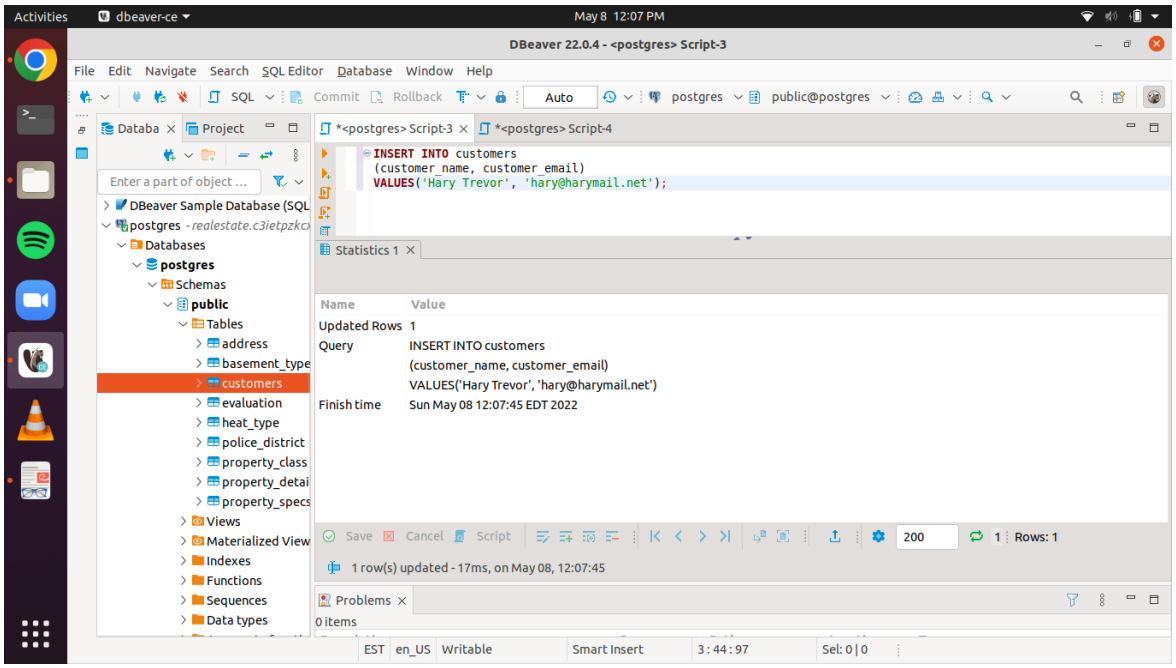
insert: here, we are inserting a new customer, whose name is Hary Trevor and whose email is hary@harymail.net.

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

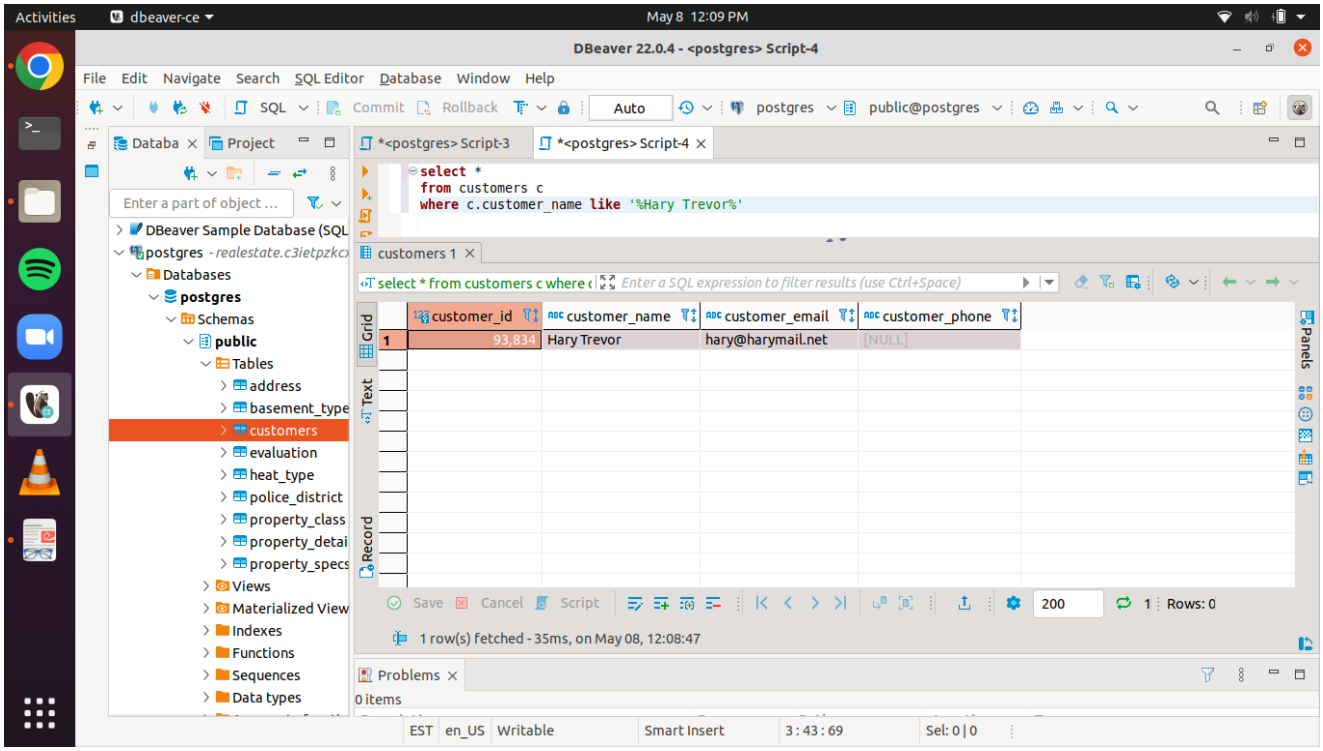
Before Insert:



Successful Insert:



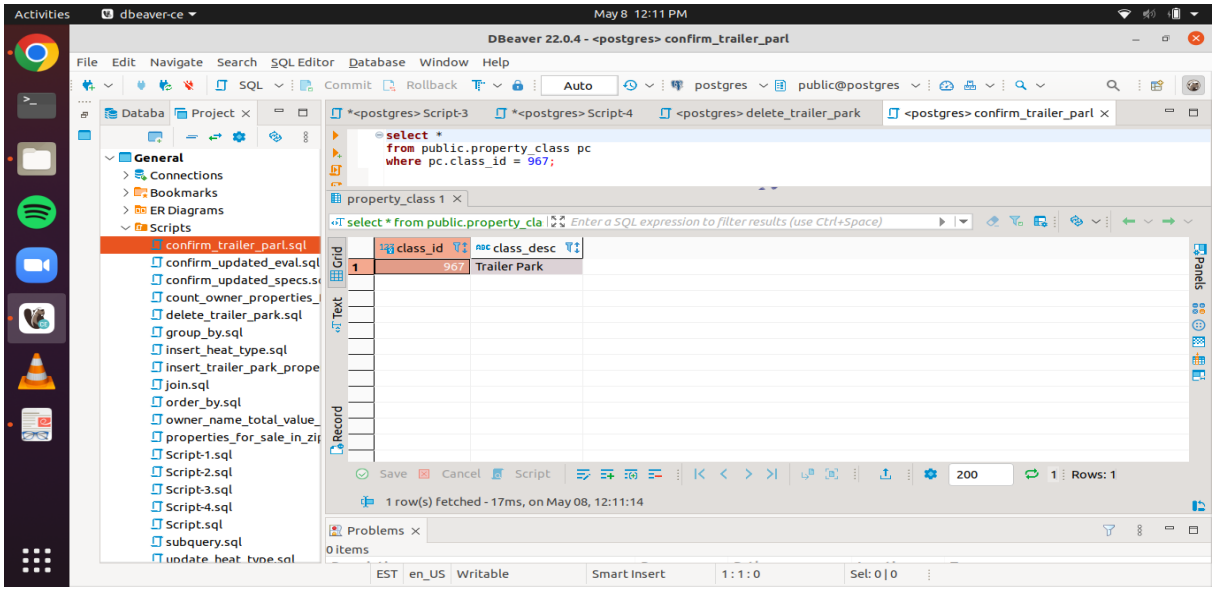
After Insert:



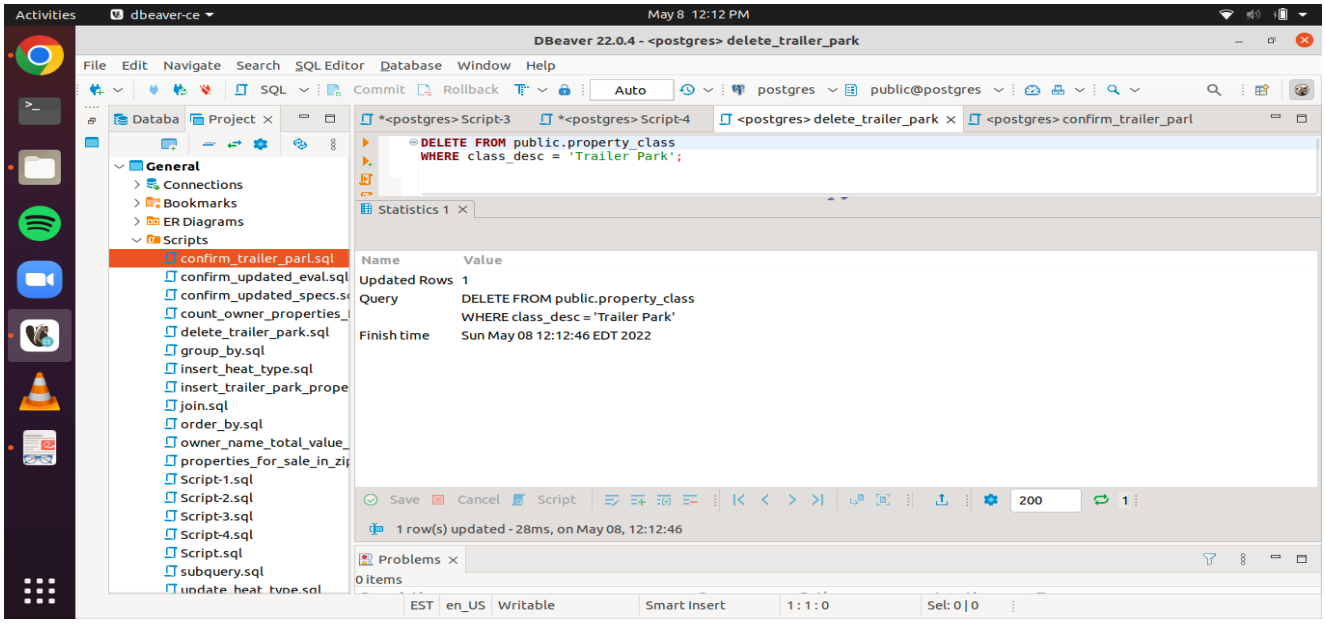
***delete*:** here, we delete the property class, 'Trailer Park'.

```
DELETE FROM property_class
WHERE class_desc = 'Trailer Park';
```

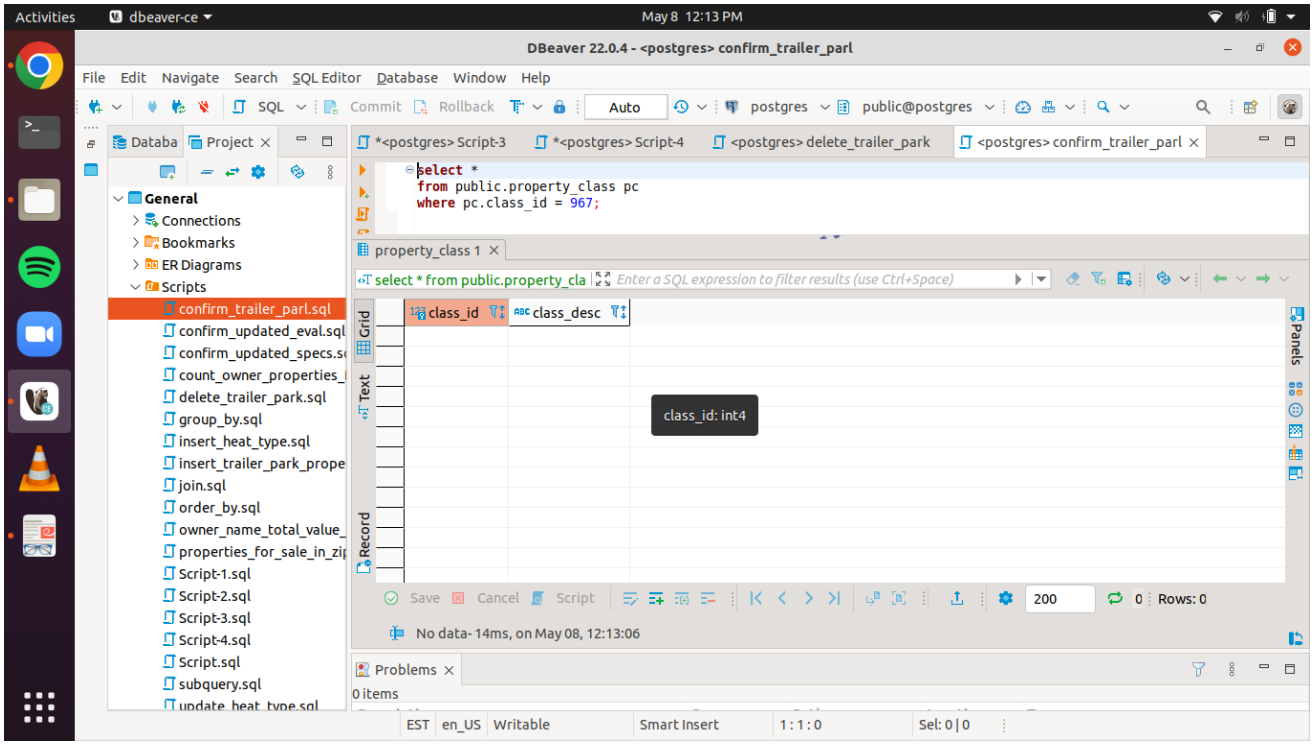
Before Delete:



Successful Delete:



After Delete:

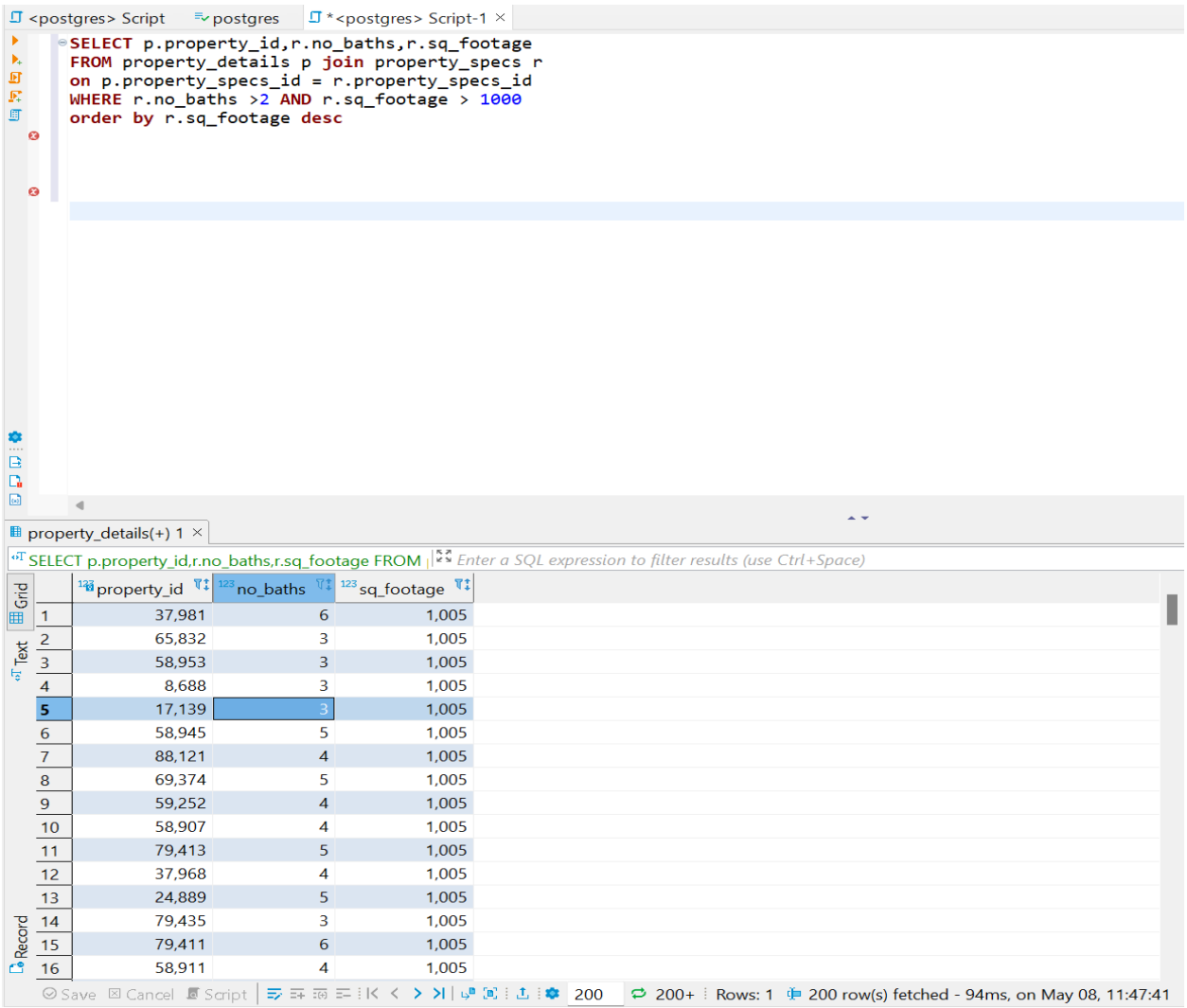


Views:

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:

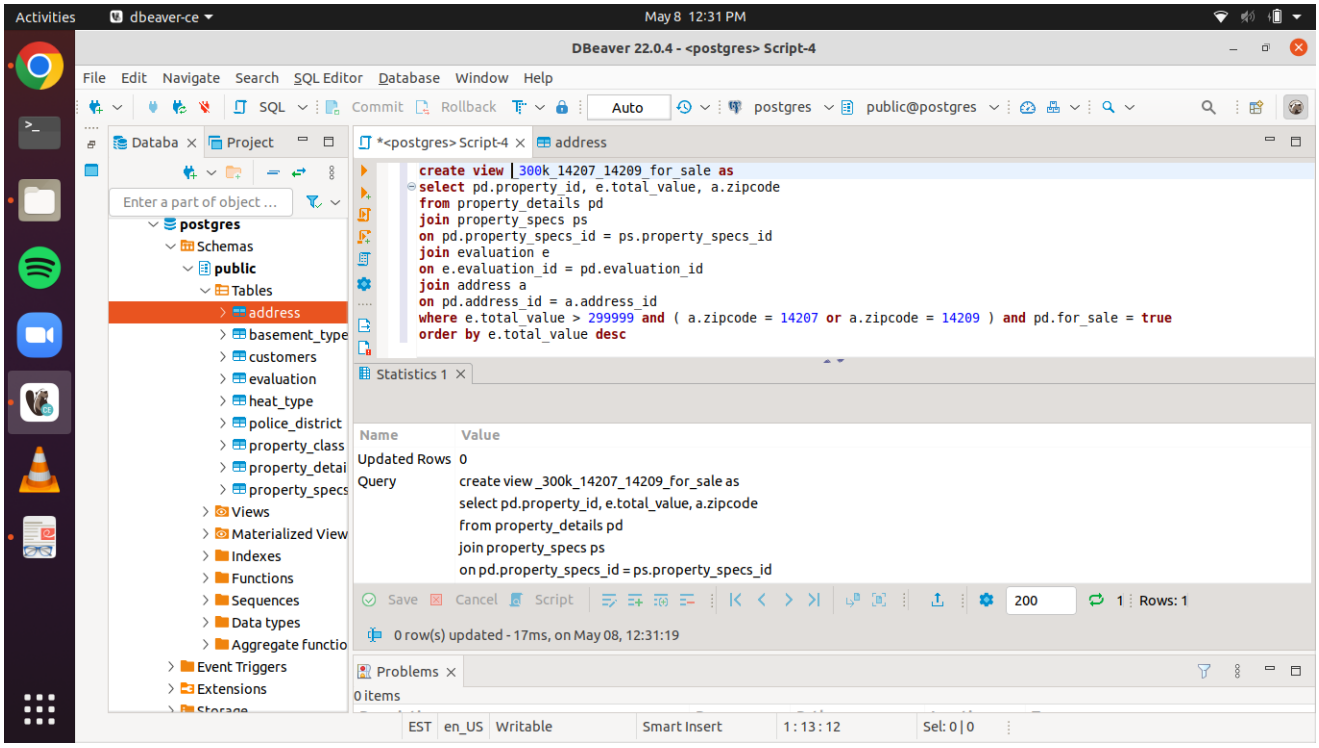


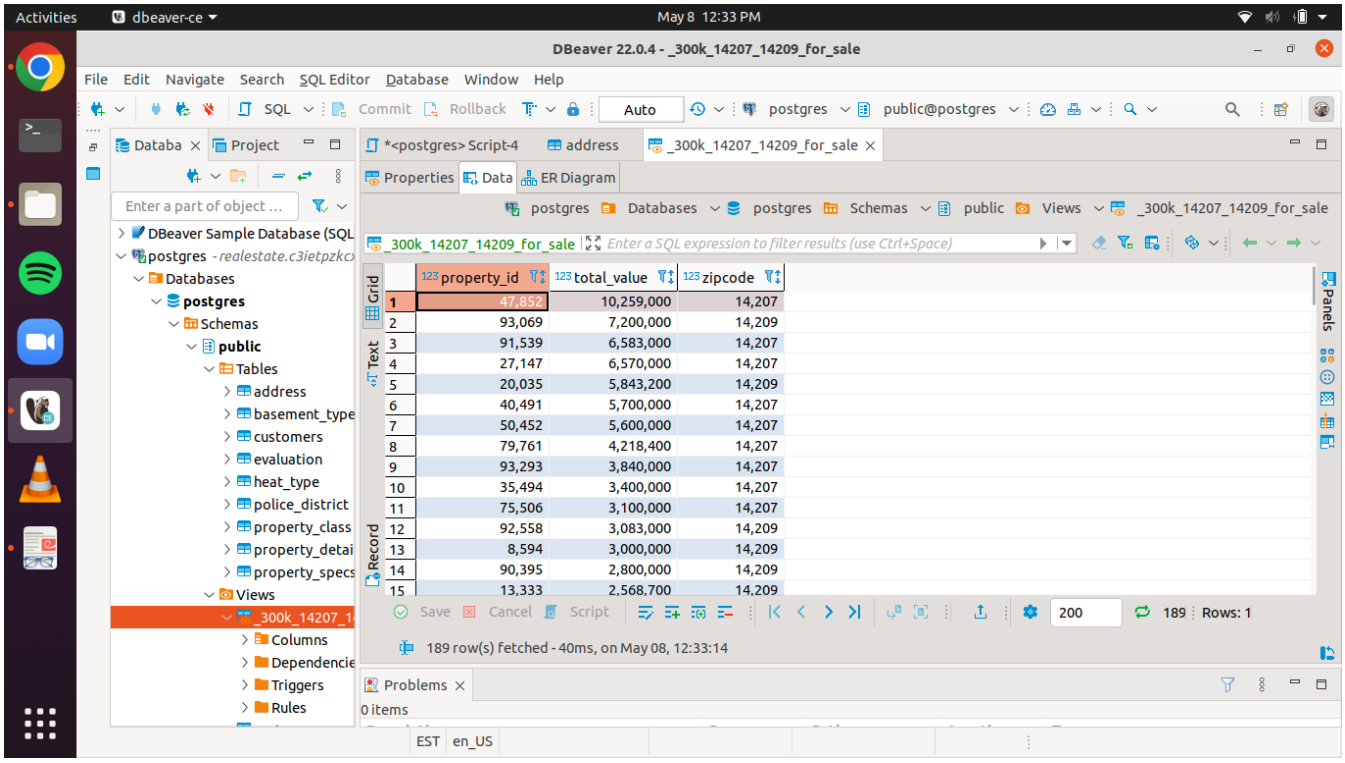
B.)

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:



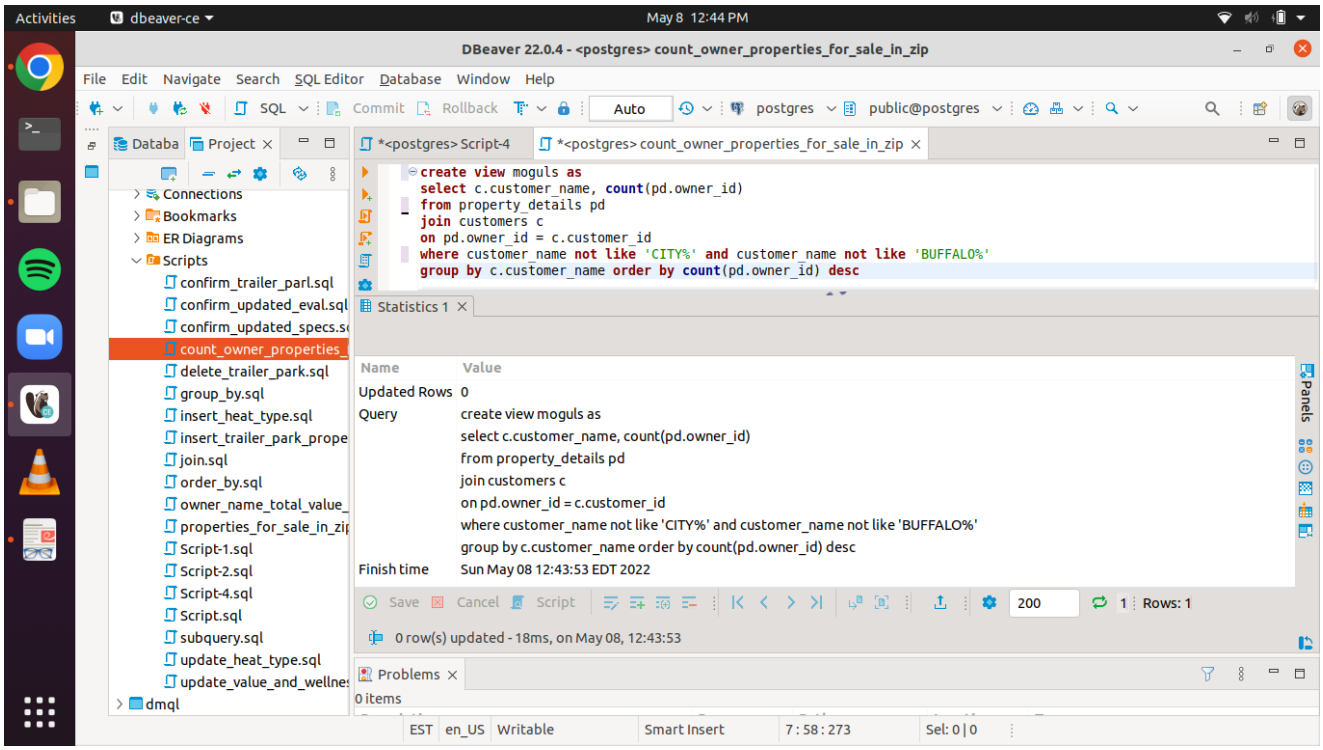

Activities
dbeaver-ce
May 8 12:33 PM
DBeaver 22.0.4 - _300k_14207_14209_for_sale
File Edit Navigate Search SQL Editor Database Window Help
SQL Commit Rollback Auto postgres public@postgres
Databa Project
Enter a part of object ...
DBeaver Sample Database (SQL
postgres - realestate.c3ietpzkc
Databases
postgres
Schemas
public
Tables
address
basement_type
customers
evaluation
heat_type
police_district
property_class
property_detai
property_specs
Views
_300k_14207_1
Columns
Dependencie
Triggers
Rules
*<postgres> Script-4
address
_300k_14207_14209_for_sale
Properties Data ER Diagram
postgres Databases postgres Schemas public Views _300k_14207_14209_for_sale
_300k_14207_14209_for_sale Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
Record
Panels
property_id total_value zipcode
1 47,852 10,259,000 14,207
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
Save Cancel Script
200 189 Rows: 1
189 row(s) fetched - 40ms, on May 08, 12:33:14
Problems
0 items
EST en_US

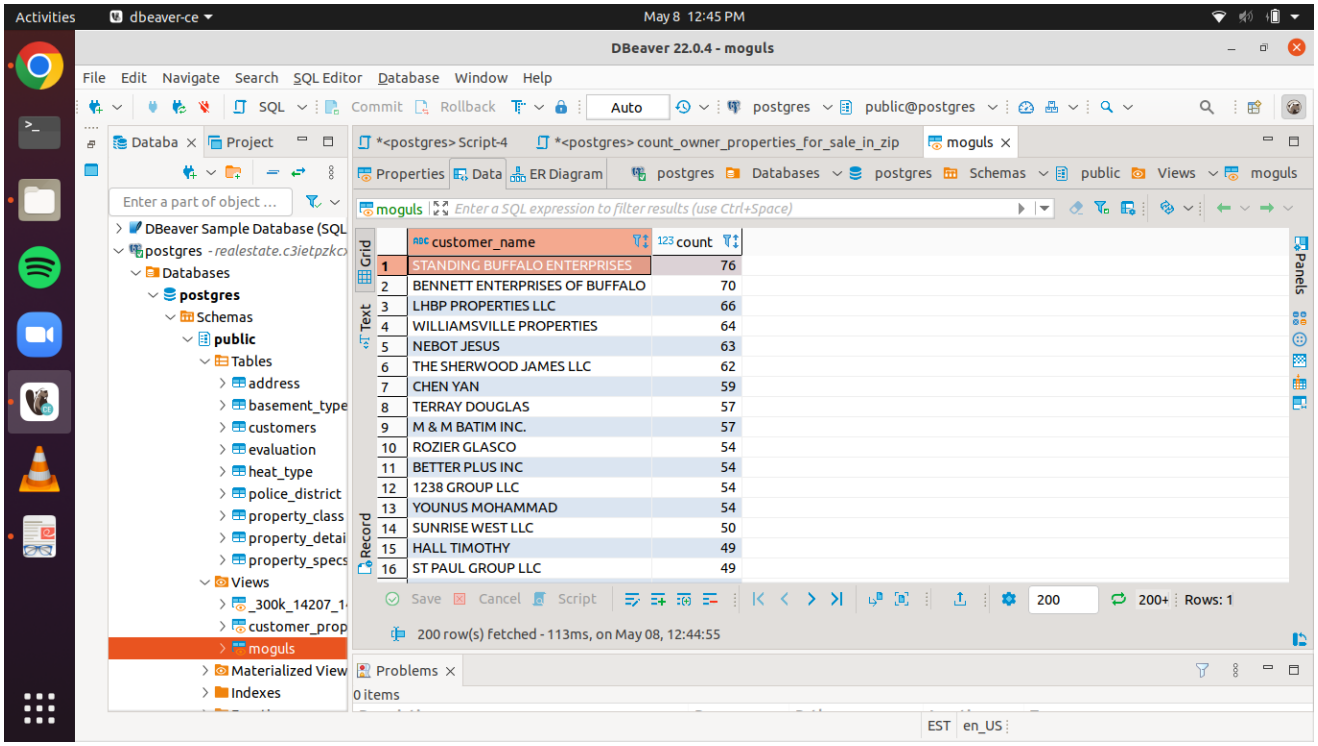
C.)

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%.

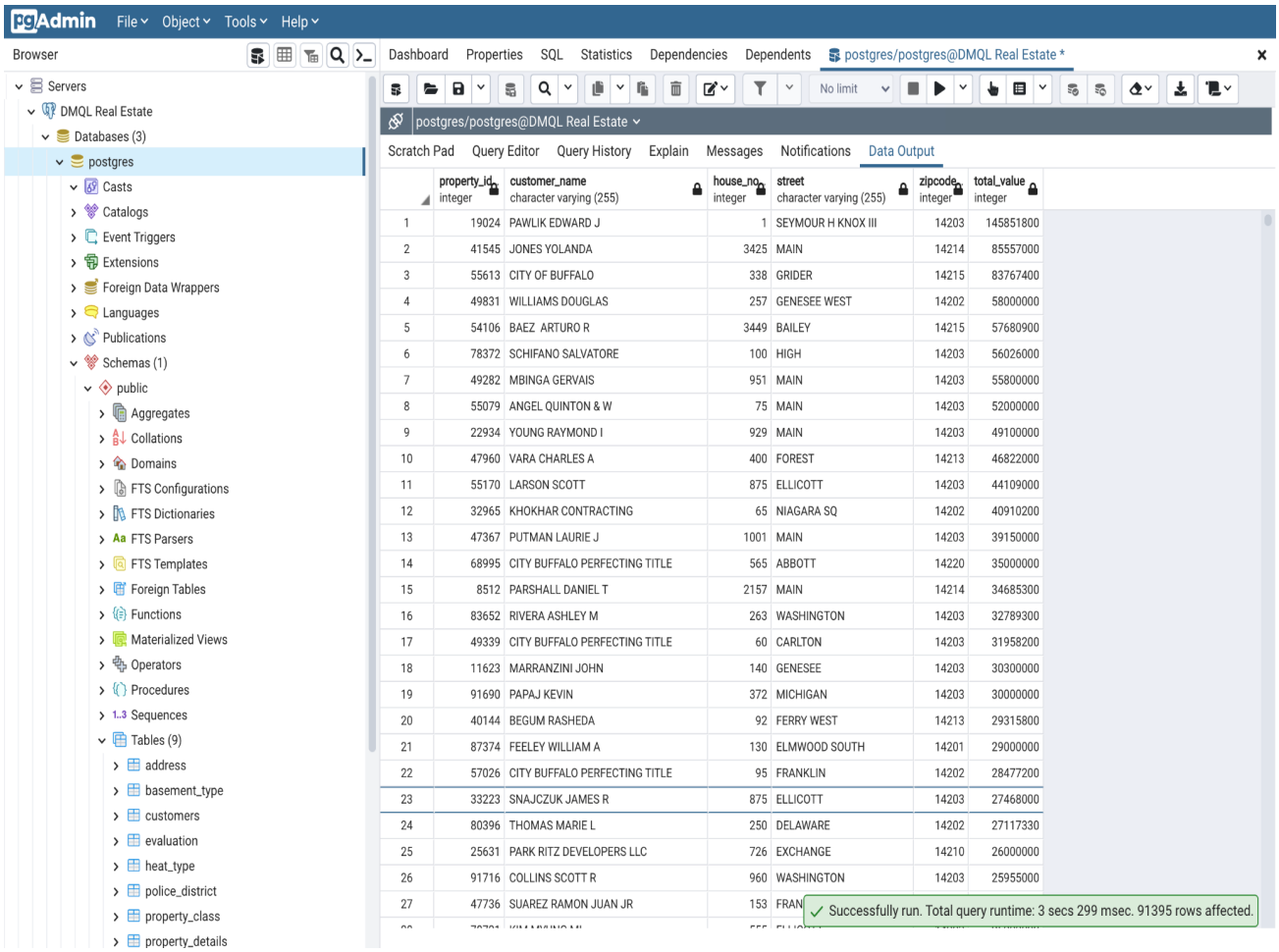
Fix Query:

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

RESULT:

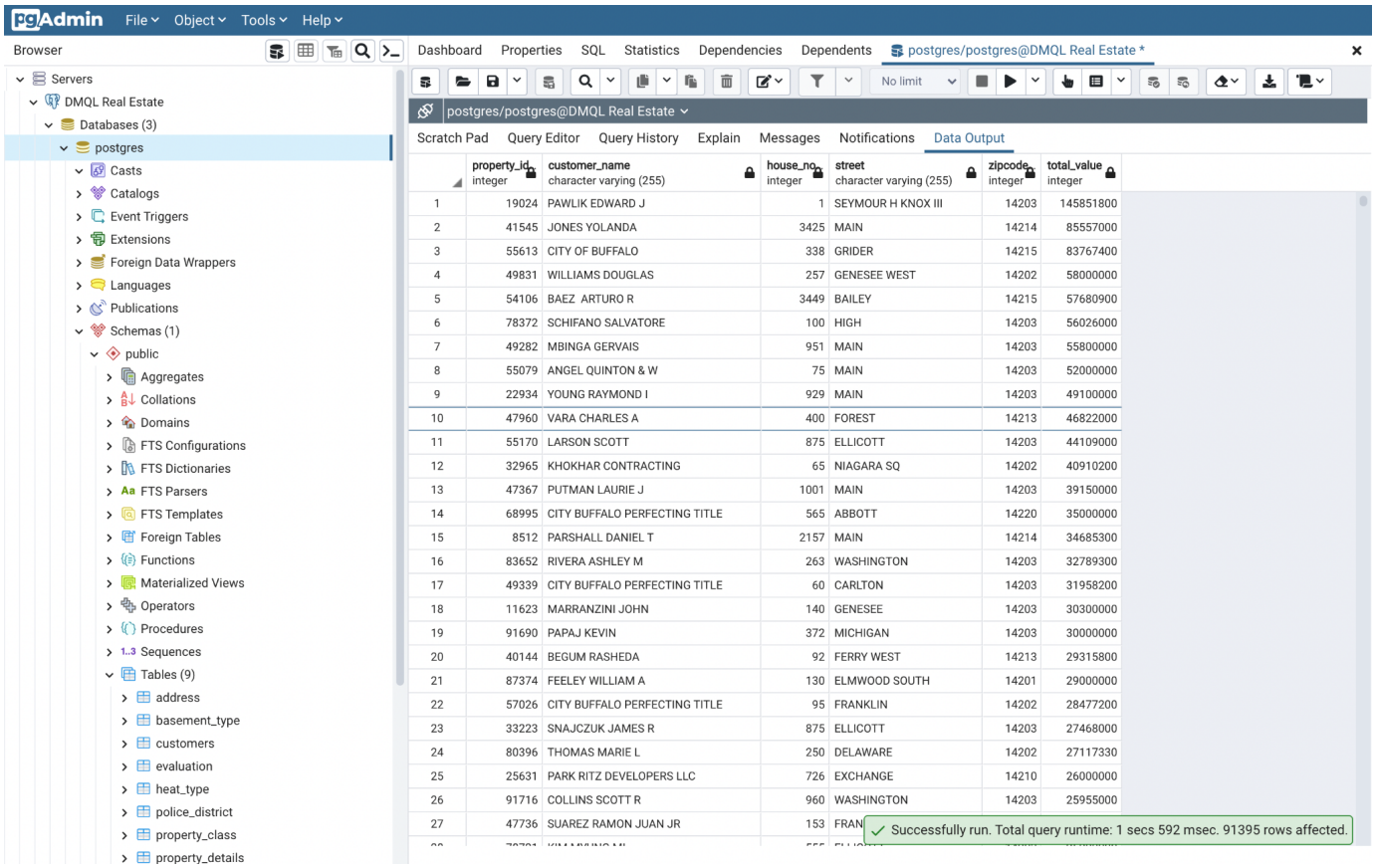
Query cost before fix:

Runtime: 3 secs 299msecs



Query cost after fix:

Runtime: 1second 592 msec



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%.

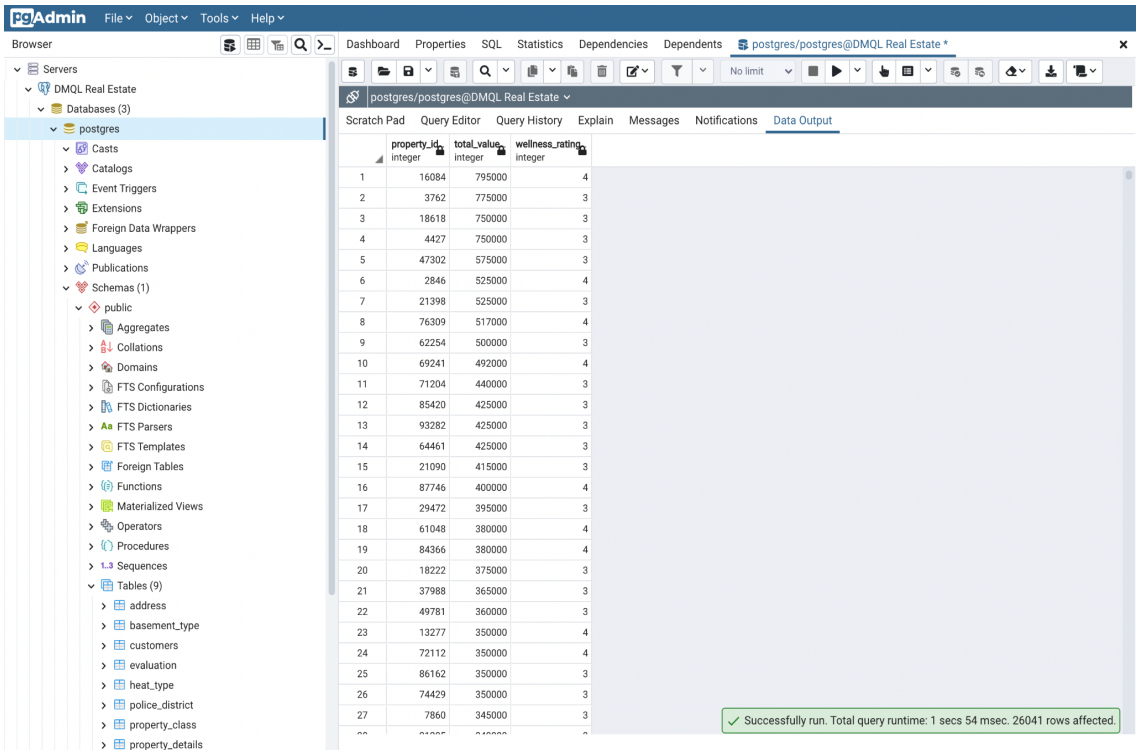
Fix Query:

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

RESULT:

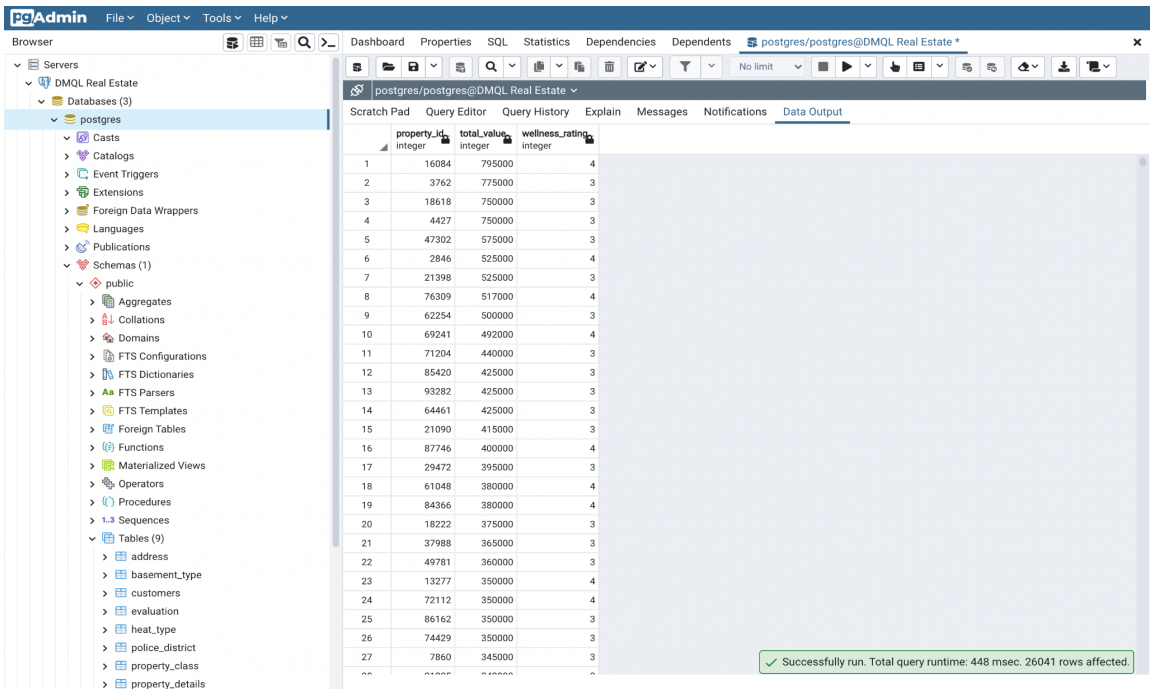
Query cost before fix:

Runtime: 1 second 54 msec



Query cost after fix:

Runtime: 0 sec 448 msec



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.