Providing Real Estate Resources for Prospective Home Buyers in Buffalo

Sri Upanya Balu - UBID: 50478734
Roshni Balasubramian - UBID: 50483779
Nicholas Soluri - UBID: 50468066

May 7, 2023

1 INTRODUCTION

Our database contains 93,000 entries detailing real estate in the city of Buffalo. Attributes of the database include property size, address information, structure dimensions, number of rooms, sale price, and many more. Information used in the database was acquired from the city of Buffalo's website (see references).

2 PURPOSE OF THE DATABASE

Rather than create a database, one could simply make use of an excel spreadsheet. However, the number of rows of data is incredibly large and would be too bulky for an excel spreadsheet. We also can use SQL queries on a database. The problem we aim to solve with this database is to prevent potential homebuyers from investing in properties without knowing certain details about them, such as interior and exterior condition grades. Additionally, our database will be designed in a way to allow potential homebuyers to compare statistics about properties from certain streets and zip codes to get a better sense of the surrounding area as well, which would be difficult to do otherwise. Getting a better idea about real estate in Buffalo can help prospective home buyers make informed decisions about where and what type of home they will purchase. The database can also help city officials better understand the conditions of properties in certain neighborhoods and use this info to provide useful services to communities that need them. In general, this database will be used mostly by prospective homebuyers, and the administrator of the database could be city officials, who will oversee every real estate transaction in the city and update the database accordingly.

3 DATABASE SCHEMA DESCRIPTION

The database contains 5 tables in total. The main table is the **properties** table. This table contains the sbl as its key, which is the unique ID given to each property in the city. Other attributes of

this table pertain to property dimensions and monetary value. Additionally, there are 4 attributes in this table - address_id, owner_id, interior_id, and exterior_id - which each serve as foreign keys to connect the main table to each of the other tables. One of the other tables is the **address_info** table, which contains attributes regarding the location of the property, such as latitude, longitude, street_name, and house number. Address_id is the primary key for this table. Another table is the **owner_info** table, which contains attributes regarding the name of the owner(s) and their mailing addresses. This table was deemed necessary because, for many properties, the owner(s) does not actually live at the property, but rather may rent the property out to tenants. In these cases, the owner of the property has a different address, so putting this information in a separate table helps to limit confusion and reduces overcrowding of the main table. Owner_id is the primary key for this table. The other two tables are the **exterior_info** and **interior_info** tables. These tables contain attributes relating to the exterior and interior of the structures respectively. Some attributes in the exterior_info table include acres, wall_description, and construction_grade. Some attributes of the interior_info table include num_bed, num_bath, and heat_type. The primary keys for these tables are exterior_id and interior_id respectively.

4 BOYCE-CODD NORMAL FORM (Task 6)

In order to put the database into Boyce-Codd normal form, the function dependencies (FDs) had to be checked in our initial schema which was turned in for the checkpoint submission. There were multiple functional dependencies which violated BDNF. For example, in the address_info table, the address attribute was sufficient to uniquely define a tuple of the table. However, there was another attribute, address_id, which was also a key, and that address was dependent on. So, one of these attributes would need to be broken down further or removed in order to accommodate BCNF. We decided to remove the address attribute altogether, as it contained information which was already entirely contained within other attributes of the table, such as street name, house number, and zip code. So, because this information was redundant, the attribute was removed, and the table was then compliant with BCNF.

For some of the other tables, there existed attributes which contained additional redundant information and therefore existed in violation of BCNF. For example, the num_rooms attribute of interiror_info existed as a sum of the totals from num_bath, num_kitchen, and num_bedrooms. So, as this attribute was dependent on the 3 aforementioned attributes, and those attributes are not a superkey, num_rooms existed in violation of BCNF. So, we decided to remove this attribute, as it contained redundant and unnecessary information. After repeating this process for a few other attributes spread around the 5 tables, the table was found to satisfy BCNF.

FUNCTIONAL DEPENDENCIES (Task 6 Continued)

Here are the functional dependencies for the schema:

Main Table (properties):

- sbl →width, depth, owner_id, interior_id, exterior_id, land_value, total_value, sale_price, structure_value, value_difference, property_area

Address_info:

- Address_id → street_name, zipcode, latitude, longitude, neighborhood_name, tax_district_name, house_number
- latitude, longitude → street_name, zipcode, neighborhood_name, tax_district_name, house_number, address_id

Exterior_info:

- Exterior_id → acres, wall_description, quality_description, construction_grade, year_built

Owner_info:

- Owner_id → owner_1, owner_2, mail_1, mail_2
- Owner_1, owner_2 → owner_id, mail_1, mail_2

Interior_info:

-Interiod_id → num_units, story_1_area, story_2_area, total_int_area, heat_type, basement_type, num_bed, num_bath, num_kitchen

FUNCTIONAL DEPENDENCY EXPLANATION (Task 6 Continued)

For the main table, sbl is sufficient to define all other attributes, as each sbl represents a unique piece of property. Each of the five other 'id' attributes in the table could be repeated for multiple properties, and therefore the other attributes are not dependent on them. For example, address_id cannot uniquely define qa property, as there are a few unique cases where 2 properties share the same address. Another example is when exterior_id could be the same for multiple properties, as there could be two properties with the same structure and parcel dimensions, such as in apartment complexes. This is similarly true for interior_id, as homes in a neighborhood may be built with the same interior layout. Owner_id could repeat as well, as one owner can own multiple properties. The other attributes in the table describe the dimensions of the property and the property value, none of which act as FDs.

For the address_info table, address_id can uniquely define tuples, and therefore each of the other attributes depend on it. Also, latitude and longitude together act as an FD with respect to each of the other attributes. This is because there can only be one property per each combination of latitude and longitude coordinates. Also, tax districts and zip codes are separate entities which overlap, so neither can define the other. These FDs agree with BCNF because the left hand sides - address_info, latitude, and longitude - are all members of candidate keys for the relation.

For the exterior_info table, the only attribute which can uniquely define a tuple is exterior_id. Also, none of the other attributes are reliant on each other. For example, year_built cannot necessarily determine construction_grade, as some buildings may stay in good condition longer than others.

The owner_info table has keys (owner_id) and (owner_1, owner_2) which can uniquely define tuples. No other functional dependencies exist outside of those which include these keys. This is due to the fact that multiple property owners could live at a single address, which means that (mail_1, mail_2) cannot uniquely define a tuple.

For the interiror_info table, interior_id is the only attribute which can uniquely define a tuple. Also, none of the other attributes depend on any attribute except for interior_id. For example, the number of bedrooms does not necessarily determine the number of bathrooms.

5 FINALIZED E/R DIAGRAM (Task 7):

Compared with the E/R diagram from Milestone 1, this one has had some items removed which were found to be redundant and/or in violation of BCNF. A few examples of these attributes include the num_rooms attribute in interior_info and the age attribute in the exterior_info table. These attributes were redundant and existed in violation of BCNF, so we removed them. After this, we were able to finalize the E/R diagram.

6 INDEXING (Task 8)

We did not encounter any particular issues when handling the larger dataset which would warrant using indexing to solve. The problems which we encountered were related to the larger dataset not satisfying BCNF, which we solved by removing redundant information and re-organizing our tables. However, we were able to take advantage of indexing to solve one of our problematic queries (see section 8 below). For this query, creating an index helped to make the original query run much quicker.

7 QUERIES (Task 9)

1. SELECT example 1:

Query Query History

```
1  select distinct * from owner_info order by owner_id asc
```

Data Output Messages Notifications

	owner_id bigint	owner_1 text	owner_2 text
1	1	BILLUPS AMON	ESSIE BILLUPS
2	2	BROWN AND JONES MEDICAL GRO…	[null]
3	3	HANLEY WILLIAM P JR.	CYNTHIA C. HANLEY
4	4	COSTISICK DOROTHY	[null]
5	5	DELAWARE PLACE ASSOCIATES, LLC	[null]
6	6	ULRICH MARIE E	[null]
7	7	HOLLWAY ROBERT C	[null]
8	8	WRIGHT JOAN	[null]
9	9	CAMPBELL WILLIAM M &	CAMPBELL PATRICIA
10	10	KEEFER WILLIAM P	[null]

Figure 1: The above query shows the first 10 rows of unique owner ids

2. SELECT example 2:

Figure 2: This query shows that the highest value for a property in the dataset is $89 million.

3. SELECT example 3:

Query | Query History

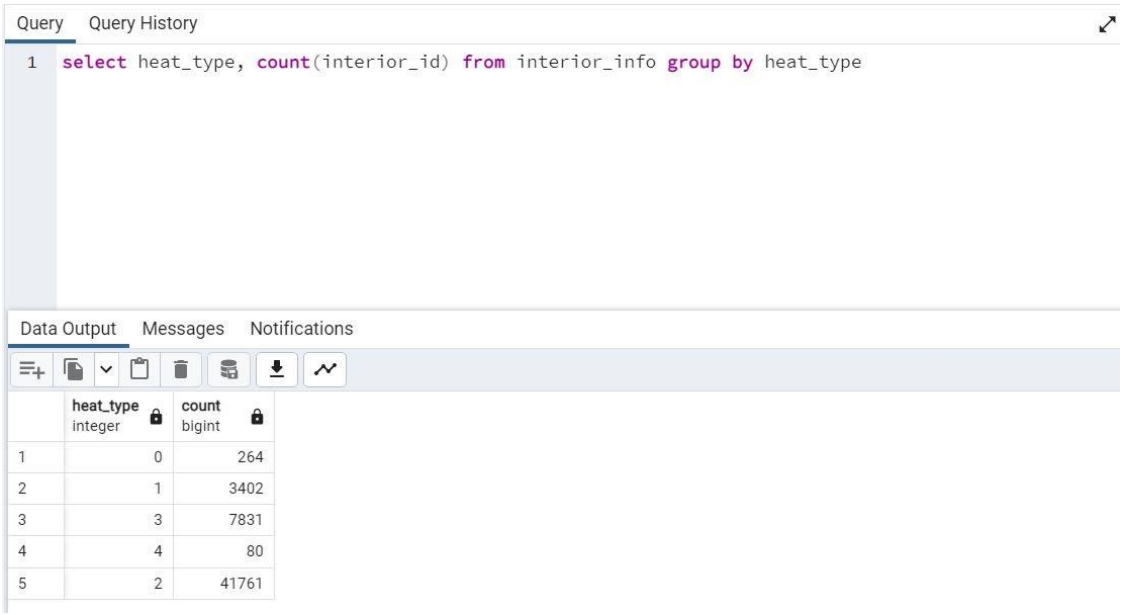
```
1  select distinct zipcode from address_info
```

Data Output | Messages | Notifications

	zipcode integer
1	14200
2	14201
3	0
4	14218
5	14215
6	14213
7	14220
8	14212
9	14208
10	14222

Figure 3: This query shows the first 10 rows of unique zipcodes in Buffalo. Properties with a 0 zipcode were not listed with one in the dataset.

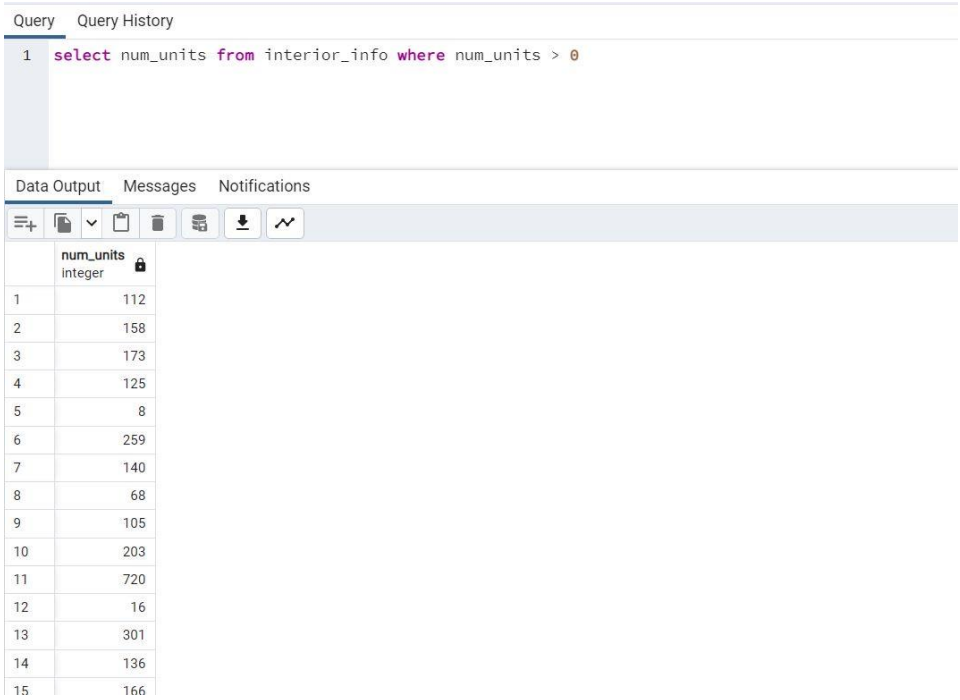
4. SELECT example 4:



	heat_type integer	count bigint
1	0	264
2	1	3402
3	3	7831
4	4	80
5	2	41761

Figure 4: The above query shows the amount of properties which use each of the unique heat types

5. SELECT example 5:



	num_units integer
1	112
2	158
3	173
4	125
5	8
6	259
7	140
8	68
9	105
10	203
11	720
12	16
13	301
14	136
15	166

Figure 5: For many properties, such as one-family houses, the num_units is equal to 0. The above query shows the number of units for each property which can have more than 1 unit, such as apartment buildings, dormitory rooms, or offices.

6. Update Query:

Here is an initial SELECT query before performing an update query:

Figure 6: The initial SELECT query before the update

Here is the UPDATE query:

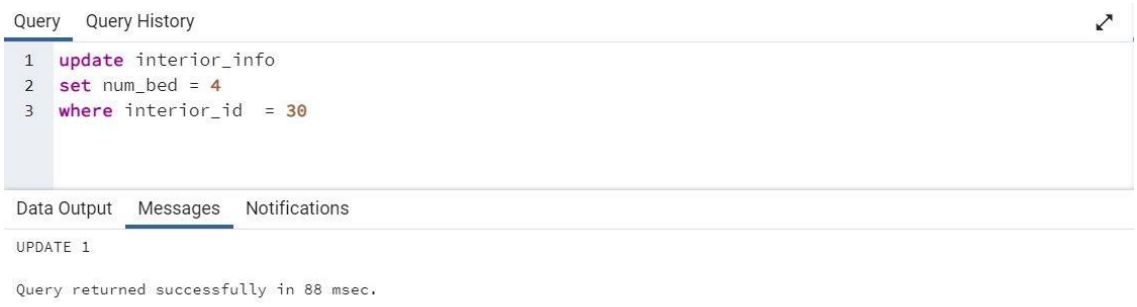


Figure 7: The UPDATE Query

And now we will run the same SELECT query as before, after the update:

Figure 8: We can see that the SELECT query now returns num_bed as 4 rather than 3 after the update

7. INSERT query:

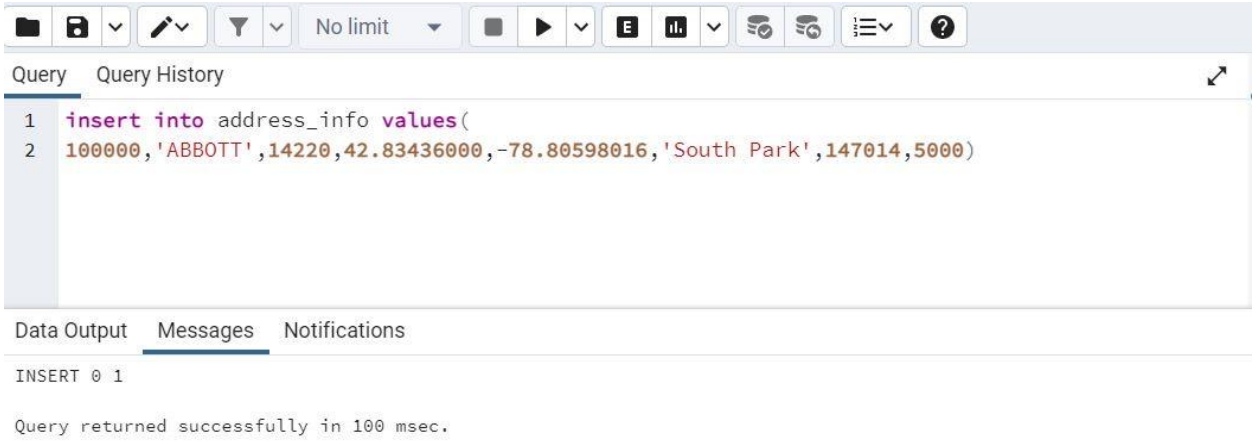
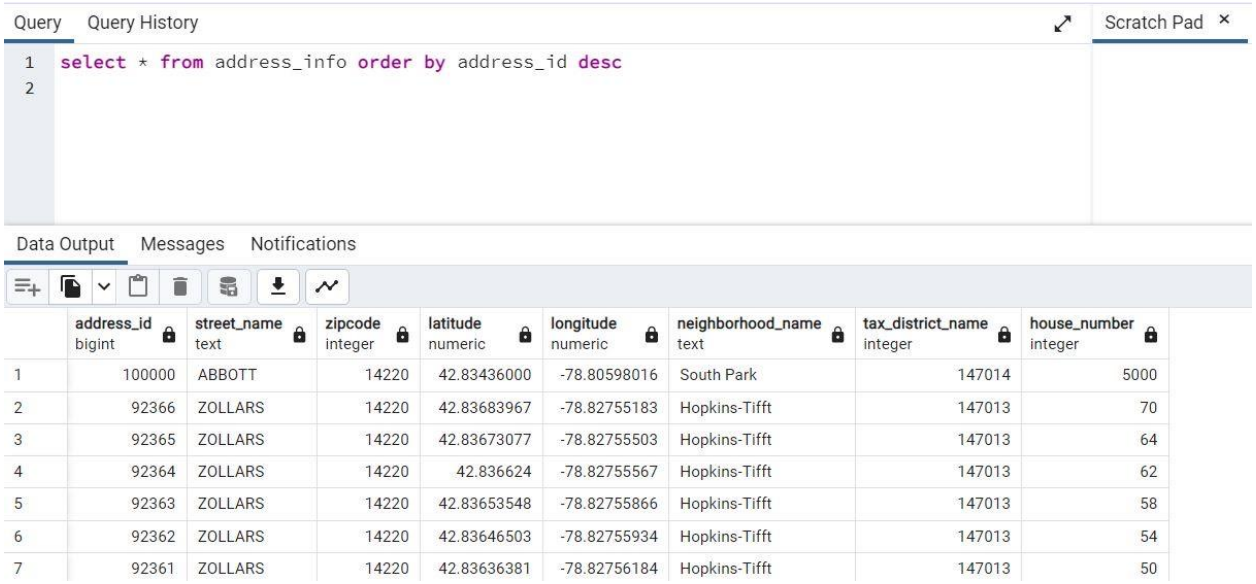


Figure 9: We use the above query to insert a new row into address_info

Proof of insertion:



	address_id bigint	street_name text	zipcode integer	latitude numeric	longitude numeric	neighborhood_name text	tax_district_name integer	house_number integer
1	100000	ABBOTT	14220	42.83436000	-78.80598016	South Park	147014	5000
2	92366	ZOLLARS	14220	42.83683967	-78.82755183	Hopkins-Tifft	147013	70
3	92365	ZOLLARS	14220	42.83673077	-78.82755503	Hopkins-Tifft	147013	64
4	92364	ZOLLARS	14220	42.836624	-78.82755567	Hopkins-Tifft	147013	62
5	92363	ZOLLARS	14220	42.83653548	-78.82755866	Hopkins-Tifft	147013	58
6	92362	ZOLLARS	14220	42.83646503	-78.82755934	Hopkins-Tifft	147013	54
7	92361	ZOLLARS	14220	42.83636381	-78.82756184	Hopkins-Tifft	147013	50

Figure 10: We can see our new row was successfully inserted above

8. DELETE query:

Query | Query History

```
1  delete from address_info where address_id = 100000
```

Data Output | Messages | Notifications

```
DELETE 1

Query returned successfully in 52 msec.
```

Figure 11: We run a query to delete the row which was added in the above INSERT query

Proof of deletion:

Query | Query History | Scratch Pad

```
1  select * from address_info order by address_id desc
2
```

Data Output | Messages | Notifications

	address_id bigint	street_name text	zipcode integer	latitude numeric	longitude numeric	neighborhood_name text	tax_district_name integer	house_number integer
1	92366	ZOLLARS	14220	42.83683967	-78.82755183	Hopkins-Tifft	147013	70
2	92365	ZOLLARS	14220	42.83673077	-78.82755503	Hopkins-Tifft	147013	64
3	92364	ZOLLARS	14220	42.836624	-78.82755567	Hopkins-Tifft	147013	62
4	92363	ZOLLARS	14220	42.83653548	-78.82755866	Hopkins-Tifft	147013	58
5	92362	ZOLLARS	14220	42.83646503	-78.82755934	Hopkins-Tifft	147013	54
6	92361	ZOLLARS	14220	42.83636381	-78.82756184	Hopkins-Tifft	147013	50
7	92360	ZOLLARS	14220	42.83621676	-78.8275636	Hopkins-Tifft	147013	48

Figure 12: We see that the value has been successfully deleted

8 QUERY EXECUTION ANALYSIS (Task 10)

Problematic Query 1:

Query Query History

```
1 (select * from exterior_info where wall_description =  'Alum/vinyl')
2 intersect
3 (select * from exterior_info where year_built = 1900)
```

Data Output Messages Explain Notifications

	exterior_id bigint	acres numeric	wall_description text	quality_description text	construction_grade text	year_built integer
1	3415	0.2	Alum/vinyl	[null]	C	1900
2	1124	0.09	Alum/vinyl	[null]	C	1900
3	183	0.135009	Alum/vinyl	[null]	C	1900
4	7811	0.078053	Alum/vinyl	[null]	C	1900
5	4799	0.149816	Alum/vinyl	[null]	C	1900
6	7611	0.123049	Alum/vinyl	[null]	C	1900
7	7131	0.14	Alum/vinyl	[null]	B	1900
8	947	0.1	Alum/vinyl	[null]	D	1900
9	2114	0.241047	Alum/vinyl	[null]	D	1900
10	6761	0.173485	Alum/vinyl	[null]	C	1900
11	7799	0.287397	Alum/vinyl	[null]	C	1900
12	5727	0.06685	Alum/vinyl	[null]	C	1900
13	65	0.085514	Alum/vinyl	[null]	C	1900
14	3308	0.096074	Alum/vinyl	[null]	B	1900
15	4428	0.114669	Alum/vinyl	[null]	C	1900

Total rows: 357 of 357 Query complete 00:00:00.166

Figure 13: This query uses an INTERSECT statement to select items which satisfy 2 conditions and takes 166 milliseconds.. Here is the EXPLAIN diagram for it:

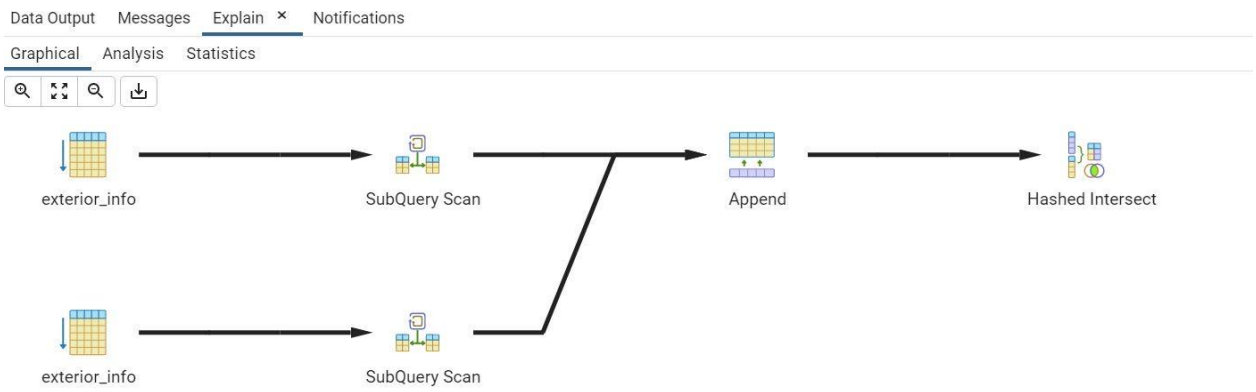


Figure 14: Above shows the EXPLAIN output for the query

To improve the runtime of this query, we can use AND:

Query Query History

```
1  select * from exterior_info where wall_description =  'Alum/vinyl' and year_built = 1900
2
```

Data Output Messages Explain Notifications

	exterior_id bigint	acres numeric	wall_description text	quality_description text	construction_grade text	year_built integer
1	3	0.047521	Alum/vinyl	[null]	C	1900
2	31	0.105211	Alum/vinyl	[null]	C	1900
3	35	0.09	Alum/vinyl	[null]	D	1900
4	44	0.121442	Alum/vinyl	[null]	C	1900
5	56	0.060744	Alum/vinyl	[null]	C	1900
6	65	0.085514	Alum/vinyl	[null]	C	1900
7	68	0.039945	Alum/vinyl	[null]	C	1900
8	73	0.140129	Alum/vinyl	[null]	C	1900
9	98	0.065381	Alum/vinyl	[null]	C	1900
10	161	0.049242	Alum/vinyl	[null]	C	1900
11	183	0.135009	Alum/vinyl	[null]	C	1900
12	196	0.074839	Alum/vinyl	[null]	C	1900
13	209	0.088636	Alum/vinyl	[null]	C	1900
14	246	0.09433	Alum/vinyl	[null]	C	1900
15	298	0.111639	Alum/vinyl	[null]	D	1900

Total rows: 357 of 357 Query complete 00:00:00.063

Figure 15: Note above that the runtime improved by 103 milliseconds. Here is the EXPLAIN output:

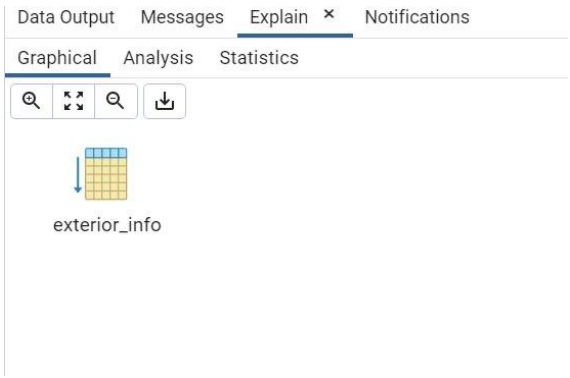
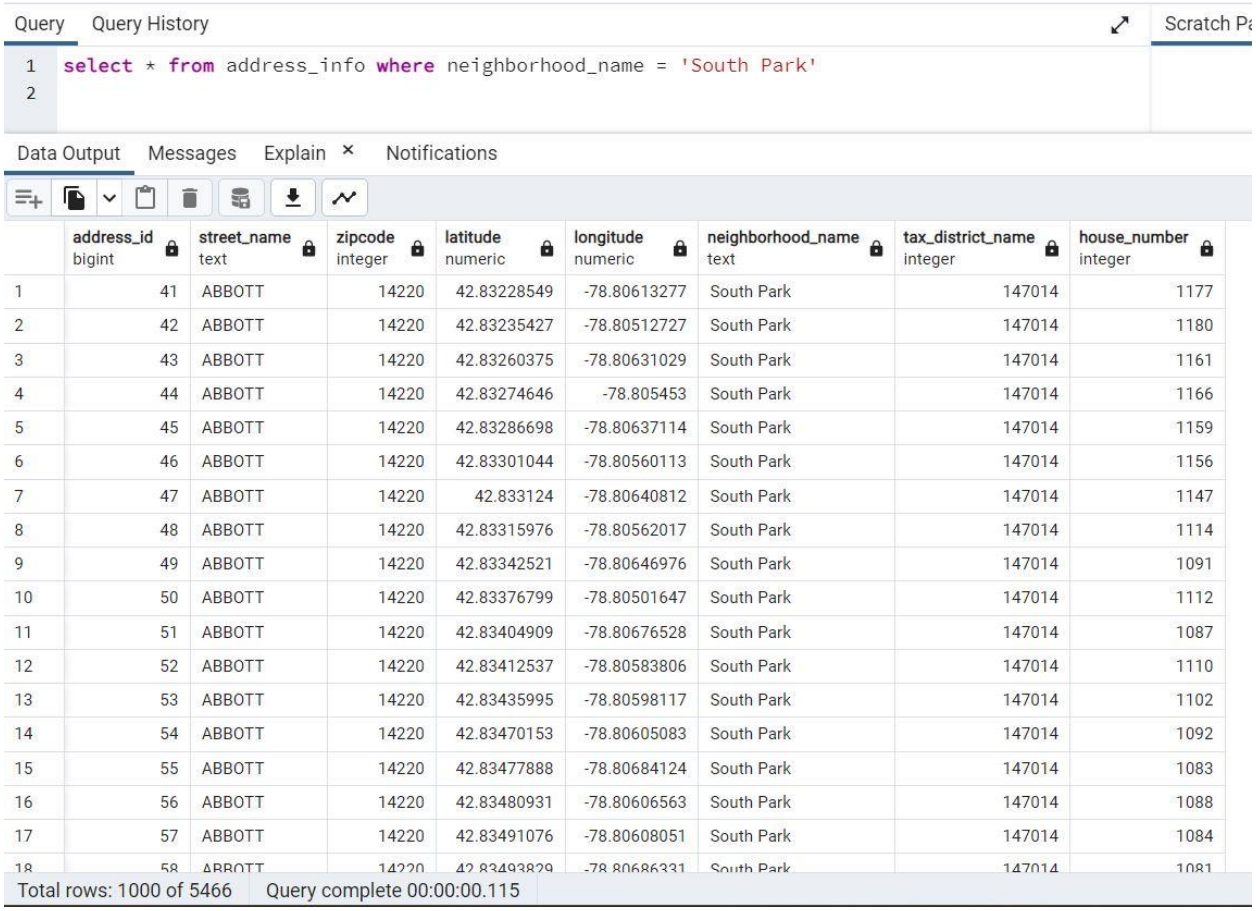


Figure 16: The EXPLAIN output for our improved query. We can see that the search process is much less complex thanks to our improved query.

Problematic query 2:



	address_id bigint	street_name text	zipcode integer	latitude numeric	longitude numeric	neighborhood_name text	tax_district_name integer	house_number integer
1	41	ABBOTT	14220	42.83228549	-78.80613277	South Park	147014	1177
2	42	ABBOTT	14220	42.83235427	-78.80512727	South Park	147014	1180
3	43	ABBOTT	14220	42.83260375	-78.80631029	South Park	147014	1161
4	44	ABBOTT	14220	42.83274646	-78.805453	South Park	147014	1166
5	45	ABBOTT	14220	42.83286698	-78.80637114	South Park	147014	1159
6	46	ABBOTT	14220	42.83301044	-78.80560113	South Park	147014	1156
7	47	ABBOTT	14220	42.833124	-78.80640812	South Park	147014	1147
8	48	ABBOTT	14220	42.83315976	-78.80562017	South Park	147014	1114
9	49	ABBOTT	14220	42.83342521	-78.80646976	South Park	147014	1091
10	50	ABBOTT	14220	42.83376799	-78.80501647	South Park	147014	1112
11	51	ABBOTT	14220	42.83404909	-78.80676528	South Park	147014	1087
12	52	ABBOTT	14220	42.83412537	-78.80583806	South Park	147014	1110
13	53	ABBOTT	14220	42.83435995	-78.80598117	South Park	147014	1102
14	54	ABBOTT	14220	42.83470153	-78.80605083	South Park	147014	1092
15	55	ABBOTT	14220	42.83477888	-78.80684124	South Park	147014	1083
16	56	ABBOTT	14220	42.83480931	-78.80606563	South Park	147014	1088
17	57	ABBOTT	14220	42.83491076	-78.80608051	South Park	147014	1084
18	58	ABBOTT	14220	42.83493829	-78.80686331	South Park	147014	1081

Figure 17: This query retrieves all addresses in the South Park neighborhood. It takes 115 milliseconds to complete. We can use indexing to improve this runtime by reducing the amount of items which the query needs to look through.

Now, we will create an index:

Query Query History

```
1  create index n_name on address_info(neighborhood_name)
2
```

Data Output Messages Explain × Notifications

```
CREATE INDEX

Query returned successfully in 432 msec.
```

Figure 18: Create the index on neighborhood_name

And re-run the query to see if it executes quicker:

Query Query History Scratch Pad ×

```
1  select * from address_info where neighborhood_name = 'South Park'
2
```

Data Output Messages Explain × Notifications

	address_id bigint	street_name text	zipcode integer	latitude numeric	longitude numeric	neighborhood_name text	tax_district_name integer	house_number integer
1	41	ABBOTT	14220	42.83228549	-78.80613277	South Park	147014	1177
2	42	ABBOTT	14220	42.83235427	-78.80512727	South Park	147014	1180
3	43	ABBOTT	14220	42.83260375	-78.80631029	South Park	147014	1161
4	44	ABBOTT	14220	42.83274646	-78.805453	South Park	147014	1166
5	45	ABBOTT	14220	42.83286698	-78.80637114	South Park	147014	1159
6	46	ABBOTT	14220	42.83301044	-78.80560113	South Park	147014	1156
7	47	ABBOTT	14220	42.833124	-78.80640812	South Park	147014	1147
8	48	ABBOTT	14220	42.83315976	-78.80562017	South Park	147014	1114
9	49	ABBOTT	14220	42.83342521	-78.80646976	South Park	147014	1091
10	50	ABBOTT	14220	42.83376799	-78.80501647	South Park	147014	1112
11	51	ABBOTT	14220	42.83404909	-78.80676528	South Park	147014	1087
12	52	ABBOTT	14220	42.83412537	-78.80583806	South Park	147014	1110
13	53	ABBOTT	14220	42.83435995	-78.80598117	South Park	147014	1102
14	54	ABBOTT	14220	42.83470153	-78.80605083	South Park	147014	1092
15	55	ABBOTT	14220	42.83477888	-78.80684124	South Park	147014	1083
16	56	ABBOTT	14220	42.83480931	-78.80606563	South Park	147014	1088
17	57	ABBOTT	14220	42.83491076	-78.80608051	South Park	147014	1084
18	58	ABBOTT	14220	42.83493829	-78.80686331	South Park	147014	1081

Total rows: 1000 of 5466 Query complete 00:00:00.070

Figure 19: It can be seen that the creation of the index helped to save 45 milliseconds compared to beforehand.

Problematic query 3:

This query selects interiors with basements of type 1, 2, or 3:

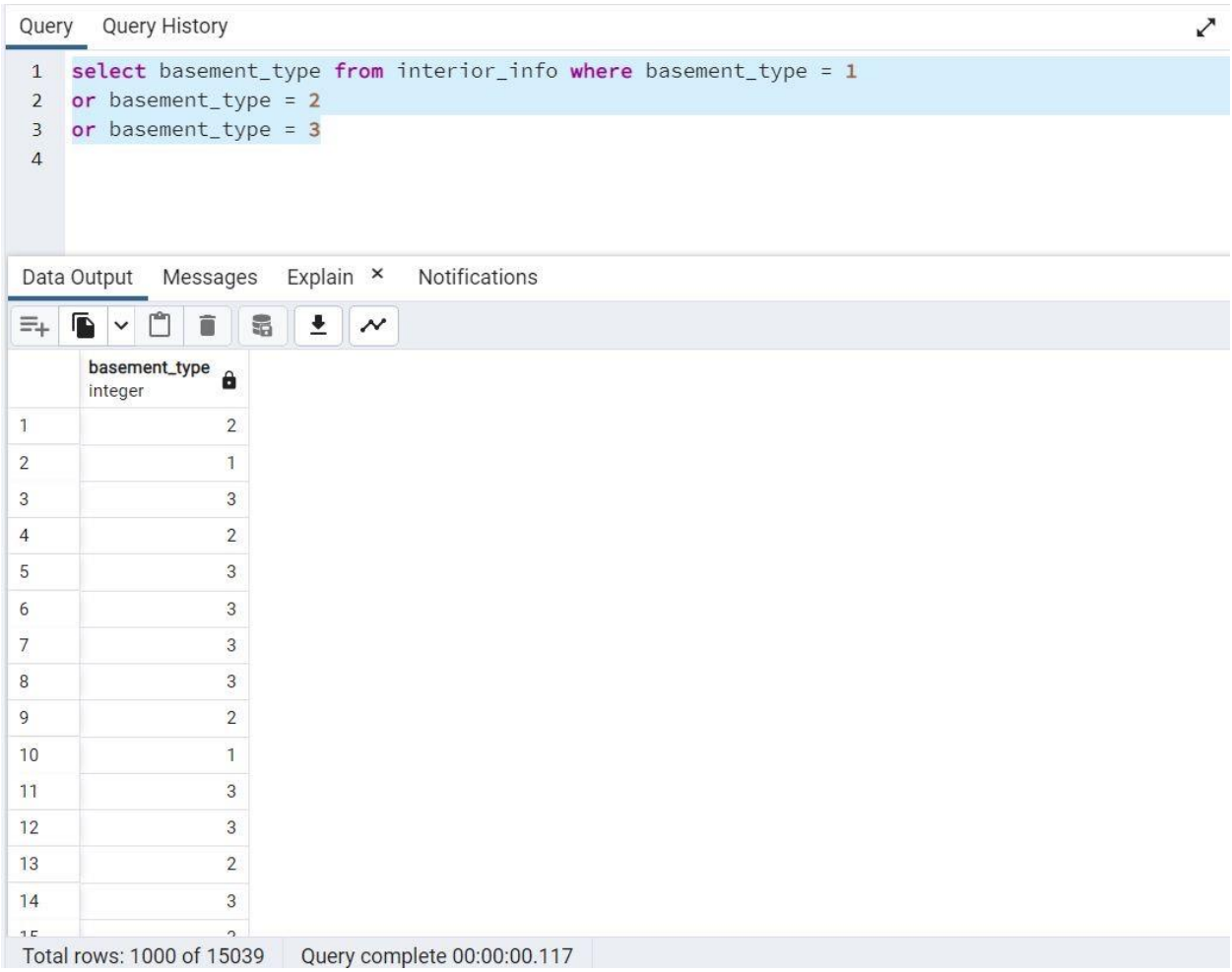


Figure 20: We execute our SELECT query

It was determined that this query could be improved by using the IN operator:

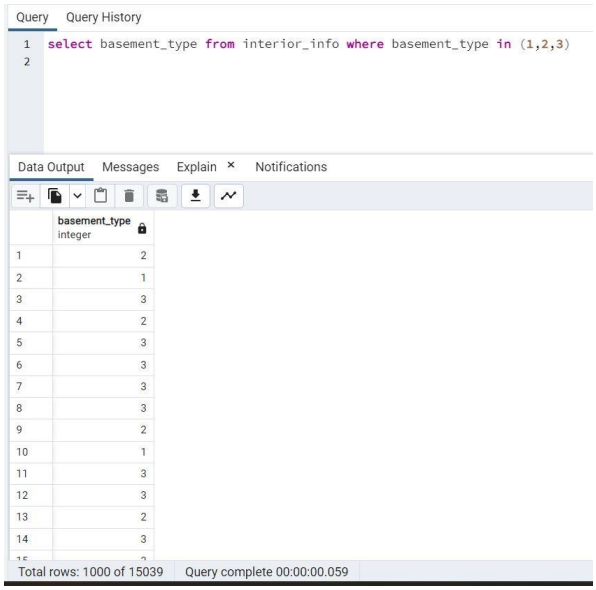


Figure 21: The use of the IN operator improved the execution time by 58 milliseconds.

BONUS TASK (Task 11)

We were able to create a running website to view visualizations of data from our database. We were able to accomplish this using Flask, a python package used for building web applications. We also use psycopg2 in order to retrieve data from our database for use in the site. The site can be accessed using the code provided with our submission and will be demoed during our presentation. From our local machine, the web address is http://127.0.0.1:5000. To plot the graphs, the matplotlib package was used. Here is an example screenshot from the site:

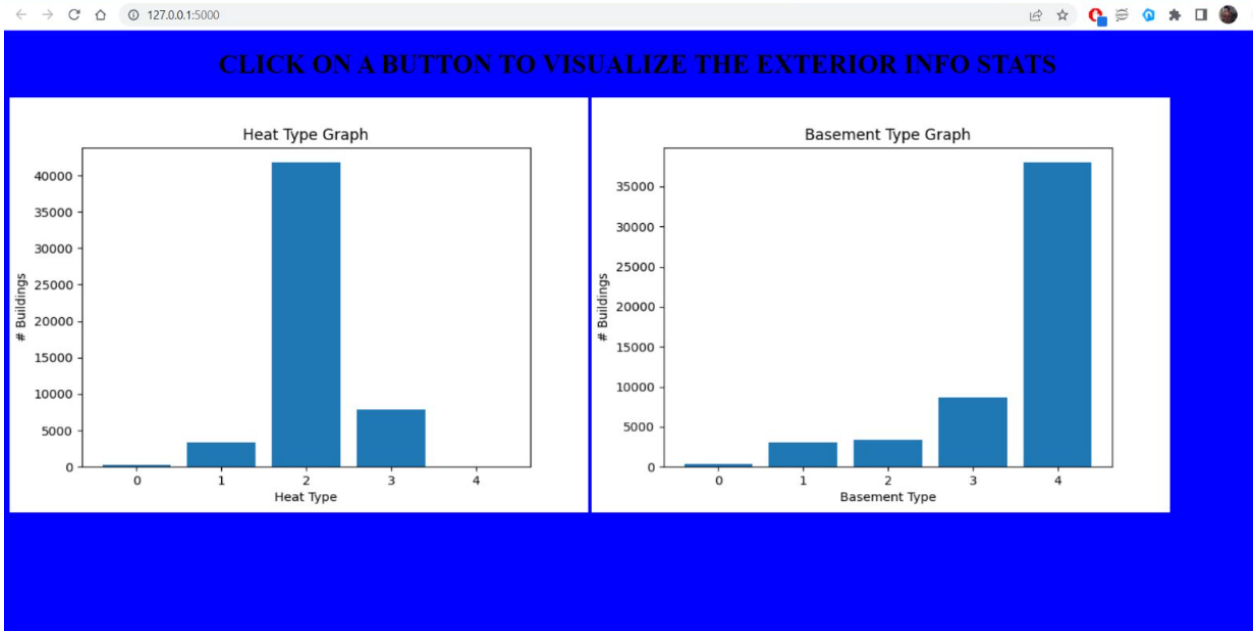


Figure 22: This screenshot from the web app plots the total properties by heat type (left) and the total properties by basement type (right).