Database for Properties in Buffalo

ZeMing Zhang
zemingzh@buffalo.edu

Deepak Raj Mohan Raj
dmohanra@buffalo.edu

***Abstract*—The city of Buffalo is a growing real estate market with numerous properties available for purchase. However, the vast amount of data regarding property types, sizes, values, owners, locations, and room counts can be overwhelming for individuals looking to make informed decisions. To address this issue, our team proposes a structured database that will provide a user-friendly frontend visual on the dataset as a whole. Our project aims to create a database that displays information about properties available for sale in the city of Buffalo. The database will contain over 90,000 rows of tax parcels in the city of Buffalo, along with information on vacant land available for sale and property tax details. With this database, users can easily search, filter, sort, and compare properties based on their needs and preferences, leading to more informed decisions and a more efficient real estate market in Buffalo.**

***Index Terms*—postgresql, execution, analysis, query, decompose**

I. Introduction

The real estate market in the city of Buffalo is growing rapidly, with a plethora of properties available for purchase. However, potential buyers often find it challenging to navigate the vast amount of data available regarding property types, sizes, values, owners, locations, and room counts. To address this issue, our team proposes the development of a structured database that will provide a user-friendly frontend visual on the dataset as a whole. The database will contain over 90,000 rows of tax parcels in the city of Buffalo, along with information on vacant land available for sale and property tax details. Our project aims to create a database that displays information about properties available for sale in the city of Buffalo. With this database, users can easily search, filter, sort, and compare properties based on their needs and preferences, leading to more informed decisions and a more efficient real estate market in Buffalo. This project will benefit individuals, business people, home owners, investors, government contractors, and anyone interested in the Buffalo real estate market. In this report, we will discuss the project description and problem statement, as well as the methods and technologies used to create the database.

II. Target User

Our target users for this project include not only individuals interested in purchasing a property in the city of Buffalo, but also business people, home owners, investors, government contractors, and anyone interested in the Buffalo real estate market. The database will contain detailed information about tax parcels, property sizes, values, owners, locations, room counts and types, and vacant land available for sale. Additionally, the database will include property tax details, which can be helpful for private companies looking to invest in Buffalo's development, and local business owners who may need to make informed decisions regarding their properties. Our team will be responsible for administering the database, which will include managing both the front end and back end of the system. The front end will be designed to provide a user-friendly interface for users to interact with the data, allowing them to search, filter, sort, and compare properties based on their needs and preferences. The back end of the system will be responsible for maintaining the database, ensuring data accuracy, and performing regular backups to prevent data loss.

III. Loading the data into database

A. Python Script

The dataset for our project was sourced from here. The dataset was then imported and stored in a dataframe. After identifying irrelevant columns and removing those with a high percentage of null values, data preprocessing was performed to remove special characters. Subsequently, a connection to PostgreSQL was established, and a database was created. Finally, we automated a script to create a relational schema for a table named "main" and successfully loaded the preprocessed dataframe into the database.

B. SQL Script

In this work, a SQL script has been utilized to generate decomposed tables from the primary data table. The appropriate foreign key references and constraints have been implemented for each table based on its respective nature. Additionally, certain null values have been retained in the tables as their complete removal would significantly reduce the available data.

IV. Changes Implemented from Milestone 1

The changes made in the final project report from the initial milestone report were due to the enhanced knowledge and understanding of database management gained through the course. During the milestone report, we had divided the dataset into smaller tables based solely on the uniqueness of the columns or combination of columns data. However, we have now decomposed the dataset by implementing functional dependencies and normalization techniques. As a result, we have included the updated ERD diagram and modified database schema in this final report. This allowed us to better structure the data and improve the overall efficiency of the database.

V. Functional Dependency

In a relational database, functional dependency plays a crucial role in determining the relationships between various attributes in a table. The presence of functional dependency ensures data consistency and helps in reducing redundancy. In this paper, we analyze the functional dependencies present in a real estate dataset that contains information on property deeds, house locations, and owner details. The primary key in this dataset is SBL, and we have identified multiple functional dependencies involving various attributes such as building style, construction grade, exterior wall type, heat type, and overall condition. Understanding these functional dependencies is essential for efficient querying and management of the database. In this section, we discuss the functional dependencies present in the dataset and their significance in maintaining data integrity. In this dataset, the following functional dependencies are present:

1) building_style_code → building_style_description
2) building_style_description → building_style_code
3) construction_grade → construction_grade_description
4) construction_grade_description → construction_grade
5) construction_quality_code → construction_quality_description
6) exterior_wall_code → exterior_wall_description
7) exterior_wall_description → exterior_wall_code
8) heat_type → heat_type_description
9) heat_type_description → heat_type
10) overall_condition → overall_condition_description
11) overall_condition_description → overall_condition

VI. Decomposition of tables into BCNF

In order to determine whether a given determinant is a candidate key, each determinant must be checked individually. As an example, consider the functional dependency building_style_code → building_style_description. In this dependency, building_style_code serves as the determinant. To determine whether it is a candidate key, its ability to uniquely identify each row in the table must be evaluated. However, since SBL is already the primary key of the table, building_style_code cannot be a candidate key as it does not provide unique identification for each row.

This process of evaluating each determinant for candidacy as a key can be applied to all of the determinants in the table. Upon evaluation, it is observed that none of the determinants meet the criteria to be a candidate key.

Therefore, the table fails to meet BCNF requirements, which mandate that each determinant should be a candidate key. To bring the table into BCNF, it must be decomposed into multiple tables, each with a unique candidate key.

Based on the functional dependencies provided, the main table can be decomposed into the following tables to satisfy BCNF:

- This is the Main, which is the original table decomposed into separate tables and contains only non-transitive functional dependencies.

TABLE I
Main

Attribute	Description
sbl	This is the primary key for the main table.
front	The length of the front of the property.
depth	The length of the depth of the property.
property_class_description	Description for the designated use of the property for clients to understand.
roll	Unique identifier for a roll.
total_value	Total value of the property.
land_value	Land value of the property.
number_of_units	Number of units in the property.
sale_price	Sale price of the property.
year_built	The year the property was built.
first_story_area	Area of the first story of the property.
second_story_area	Area of the second story of the property.
total_living_area	Total living area of the property.
overall_condition	The overall condition of the property.
heat_type	The type of heating system used in the property.
no_of_stories	Number of stories in the property.
no_of_fireplaces	Number of fire places in the property.
no_of_beds	Number of bedrooms in the property.
no_of_baths	Number of bathrooms in the property
no_of_kitchens	Number of kitchens in the property.
census_tract	Unique identifier for a census tract.
census_block	Unique identifier for a census block
acres	Total acres of the property.
add_area	Additional area of the property.
attic_area	Area of the attic of the property.
building_style_code	The Building Style Code is a standardized code used in Open Data Buffalo to classify buildings according to their architectural style.
central_air	Whether or not the property has central air.
construction_quality_code	The Construction Quality Code is a numeric value used in Open Data Buffalo to rate the quality of construction of a building.
construction_grade	Construction grade is used to classify the quality of construction materials and methods used for buildings.
exterior_wall_code	The type of exterior wall material used.
homestead_code	Homestead code for the property.
story_height	The height of the story of the property.
wall_a	The length of wall A of the property.
wall_b	The length of wall B of the property.
wall_c	The length of wall C of the property.
property_class_code	The Property Class Code is a standard- ized code used in Open Data Buffalo to classify proper- ties according to their designated use.

- This is the Building Style, which includes building style code and its corresponding description, and satisfies the functional dependency building_style_code → building_style_description.

TABLE II
Building Style

Attribute	Description
building_style_code	The Building Style Code is a standardized code used in Open Data Buffalo to classify buildings according to their architectural style.
building_style_description	Description for style of building for clients to understand.

- This is the ConstructionGrade and contains construction grade and its description. The table satisfies the functional dependency construction_grade → construction_grade_description.

TABLE III
CONSTRUCTION GRADE

Attribute	Description
construction_grade	Construction grade is used to classify the quality of construction materials and methods used for buildings.
construction_grade_description	Description for construction grade to understand.

- This is the ConstructionQuality, which contains construction quality code and its corresponding description, and satisfies the functional dependency construction_quality_code → construction_quality_description.

TABLE IV
CONSTRUCTION QUALITY

Attribute	Description
construction_quality_code	The Construction Quality Code is a numeric value used in Open Data Buffalo to rate the quality of construction of a building.
construction_quality_description	Description for quality of construction for clients to understand.

- This is the ExteriorWall, which includes exterior wall code and its corresponding description and satisfies the functional dependency exterior_wall_code → exterior_wall_description.

TABLE V
EXTERIOR WALL

Attribute	Description
exterior_wall_code	This is the primary key to link it to the main table that we created.
exterior_wall_description	Description of the exterior wall material used in the building.

- This is the HeatType, which contains heat type and its description and satisfies the functional dependency heat_type → heat_type_description.

TABLE VI
HEAT TYPE

Attribute	Description
heat_type	This is the primary key to link it to the main table that we created.
heat_type_description	Description of the type of heating system in the building.

- This is the OverallCondition, which includes overall condition and its description and satisfies the functional dependency overall_condition → overall_condition_description.

TABLE VII
OVERALL CONDITION

Attribute	Description
overall_condition	This is the primary key to link it to the main table that we created.
overall_condition_description	Description of the overall condition of the building.

- The table ”deed” contains information about the deeds, including the SBL (property identifier), the book and page numbers, the date, and the type of deed. It also has a foreign key reference to the ”main” table.

TABLE VIII
DEED

Attribute	Description
sbl	Unique identifier for a property.
deed_book	Unique identifier for a deed book.
deed_page	Unique identifier for a deed page.
deed_date	The date of the deed.
deed_type_code	Code used to identify the type of deed.

- The table ”house_location” contains information about the location of the house, including the SBL, house number, street name, zip code, latitude, longitude, tax district, council district, police district, and neighborhood. It also has a foreign key reference to the ”main” table.

TABLE IX
HOUSE LOCATION

Attribute	Description
sbl	Unique identifier for a property.
house_number	The house number of the property.
street	The name of the street where the property is located.
zipcode	The postal code of the property.
zipcode extension	The extension of the postal code of the property.
latitude	The latitude of the property.
longitude	The longitude of the property.
tax_district	Unique identifier for a tax district.
council_district	Unique identifier for a council district.
council_district 1	Unique identifier for a council district (alternative).
police_district	Unique identifier for a police district.
neighborhood	Unique identifier for a neighborhood.

- The table ”owner_details” contains information about the owners of the property, including the SBL, owner names, previous owner name, and owner mailing address. It also has a foreign key reference to the ”main” table.

TABLE X
OWNER DETAILS

Attribute	Description
sbl	Unique identifier for a property.
owner	The name of the current owner of the property.
owner2	The name of the secondary owner of the property.
previous_owner	The name of the previous owner of the property.
owner_mail	The mailing address of the property owner.

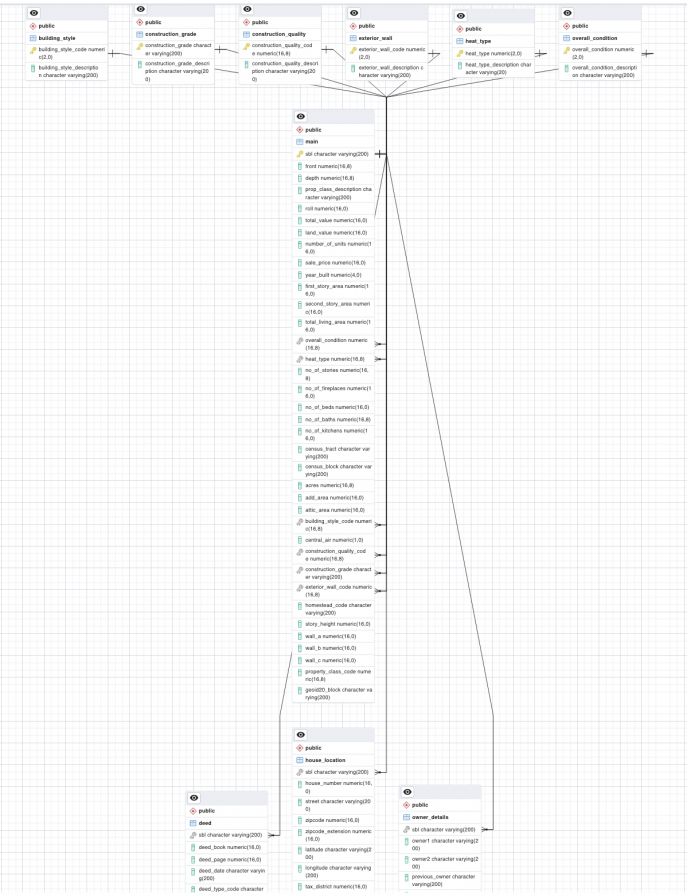

Fig. 1. ERD Diagram

Note that the tables ”deed, ”house_location”, ”owner_details” were decomposed for convenience, creating separate tables for each of the above components, while still maintaining the relationship with the ”main” table using foreign keys.

To ensure that our database decomposition is lossless, we performed a natural join of all tables and compared it to the original dataset. This process returned all columns equivalent to the original dataset, confirming that our decomposition did not result in any loss of information. Specifically, we created a stored procedure named ”get_main_data” to perform the natural join on all tables present in the database and retrieve the entire dataset. Our approach validates the correctness of our database design and ensures that all relevant data is retained in the decomposed tables.

VII. PROBLEMS WITH LARGE DATASET

- Handling large datasets can pose challenges for data analysis, including slow query times and increased memory usage.
- During our analysis, we encountered similar issues while working with a larger dataset.
- To address these challenges, we adopted indexing concepts to optimize query performance and improve efficiency.
- One of the indexing techniques we used was creating an index on the ”zipcode” column in the ”house_location” table.
- By running the ”EXPLAIN ANALYZE” command on a query that filtered results by zipcode, we were able to assess the effectiveness of our indexing approach and optimize our queries for faster execution.

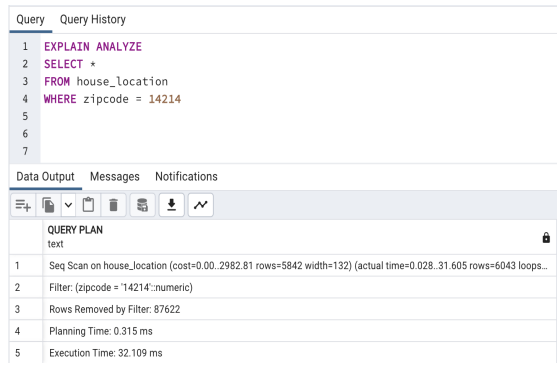

Fig. 2. Without Index

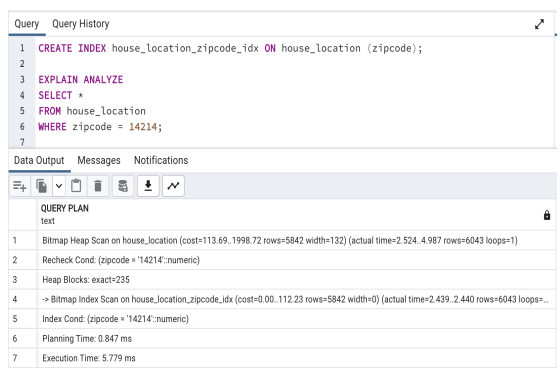

Fig. 3. With Index

VIII. SAMPLE QUERIES

A. Select queries

- The query retrieves aggregated data about houses, including their building style, construction grade, total value, and year built, from two joined tables. It applies filters to include only groups with more than 10 houses and an average total value greater than 50000, and sorts the results in descending order based on the total number of houses in each group.

Query Query History

```
SELECT bs.building_style_description, cg.construction_grade_description, COUNT(*) AS total_houses,
  AVG(m.total_value) AS avg_total_value, MIN(m.year_built) AS min_year_built, MAX(m.year_built) AS max_year_built
FROM main m
JOIN building_style bs ON m.building_style_code = bs.building_style_code
JOIN construction_grade cg ON m.construction_grade = cg.construction_grade
GROUP BY bs.building_style_description, cg.construction_grade_description
HAVING COUNT(*) > 10 AND AVG(m.total_value) > 50000
ORDER BY total_houses DESC;
```

Data Output Messages Notifications

	building_style_description character varying (200)	construction_grade_description character varying (200)	total_houses bigint	avg_total_value numeric	min_year_built numeric	max_year_built numeric
1	Old style	Average	51049	106725.890712844522	1780	2021
2	UNKNOWN	UNKNOWN	25996	359856.890598553624	[null]	[null]
3	Cape cod	Average	3819	101647.812563760147	1892	2021
4	Colonial	Average	2151	145655.127847512785	1838	2021
5	Ranch	Average	1636	104710.330073349633	1915	2021

Fig. 4. Using group by

- The query retrieves the property values and attributes from two related tables, filtering by a threshold for total living area and sorting by descending order of that attribute. The result provides insights into high-value properties and their characteristics.

Query Query History

```
SELECT
    sbl,
    total_living_area,
    building_style_description,
    construction_grade_description
FROM
    main
    JOIN building_style ON main.building_style_code = building_style.building_style_code
    JOIN construction_grade ON main.construction_grade = construction_grade.construction_grade
WHERE
    total_living_area > (SELECT AVG(total_living_area) FROM main)
ORDER BY
    total_living_area DESC;
```

Data Output Messages Notifications

	sbl character varying (200)	total_living_area numeric (16)	building_style_description character varying (200)	construction_grade_description character varying (200)
1	0890600007001111	13004	Mansion	Excellent
2	1005400002027000	10508	Mansion	Excellent
3	1005400002003000	10220	Mansion	Good
4	0896200002008000	10186	Mansion	Excellent
5	0893800001007000	9717	Mansion	Excellent

Fig. 5. Using subquery

- This SQL query retrieves information from multiple tables by joining them, applies aggregations to group data by neighborhood, and then orders neighborhoods by the average total value of the properties and returns the top 10 neighborhoods.

Query Query History

```
WITH location_info AS (
    SELECT
        h.sbl, h.house_number, h.street, h.zipcode, h.latitude, h.longitude, h.neighborhood, m.year_built, m.total_value, m.land_value,
        m.number_of_units, m.no_of_stories, m.total_living_area, bs.building_style_description, cg.construction_grade_description,
        cq.construction_quality_description, ew.exterior_wall_description
    FROM
        house_location h JOIN main m ON h.sbl = m.sbl JOIN building_style bs ON m.building_style_code = bs.building_style_code
        JOIN construction_grade cg ON m.construction_grade = cg.construction_grade
        JOIN construction_quality cq ON m.construction_quality_code = cq.construction_quality_code
        JOIN exterior_wall ew ON m.exterior_wall_code = ew.exterior_wall_code WHERE h.zipcode = 14214
),
grouped_info AS (
    SELECT
        li.neighborhood, AVG(li.year_built) AS avg_year_built, AVG(li.total_value) AS avg_total_value, AVG(li.land_value) AS avg_land_value,
        SUM(li.number_of_units) AS total_units, AVG(li.no_of_stories) AS avg_no_of_stories, AVG(li.total_living_area) AS avg_total_living_area,
        COUNT(*) AS num_buildings FROM location_info li GROUP BY li.neighborhood
),
ordered_info AS (
    SELECT
        gi.neighborhood, gi.avg_year_built, gi.avg_total_value, gi.avg_land_value, gi.total_units, gi.avg_no_of_stories, gi.avg_total_living_area,
        gi.num_buildings, ROW_NUMBER() OVER (ORDER BY gi.avg_total_value DESC) AS rank FROM grouped_info gi
)
SELECT * FROM ordered_info WHERE rank <= 10;
```

Data Output Messages Notifications

	neighborhood character varying (200)	avg_year_built numeric	avg_total_value numeric	avg_land_value numeric	total_units numeric	avg_no_of_stories numeric	avg_total_living_area numeric	num_buildings bigint	rank bigint
1	Kensington-Bailey	[null]	376500.0000000000000	29500.0000000000000	41	0.0000000000000000000	0.0000000000000000000	2	1
2	Central	[null]	335000.0000000000000	239600.0000000000000	108	0.0000000000000000000	0.0000000000000000000	1	2
3	Parkside	1911.7002854424357755	316800.9982905598291	60058.8034188034019	514	1.8136752136752137	2191.0649657264957265 0	1170	3
4	North Park	1918.4740587443599744	316179.5881962721311	70754.5881962721311	36	1.9581962721311475 4	2330.0942622950819672	244	4
5	Central Park	1926.8406115537117904	296591.9774011129644	69326.1233521657535	856	1.6772128692263653	2122.3765015734462077	1062	5

Fig. 6. using CTE

- The SQL query uses a recursive CTE (common table expression) to generate a list of all houses in the "Riverside" neighborhood and their corresponding property class descriptions and living area sizes.

Query Query History

```
WITH RECURSIVE house_list AS (
  SELECT main.sbl, prop_class_description, total_living_area
  FROM main
  JOIN house_location ON main.sbl = house_location.sbl
  WHERE neighborhood = 'Riverside'

  UNION


  SELECT main.sbl, main.prop_class_description, main.total_living_ar
  FROM main
  JOIN house_list ON main.sbl = house_list.sbl
)

SELECT *
FROM house_list;
```

Data Output Messages Notifications

	sbl character varying (200)	prop_class_description character varying (200)	total_living_area numeric (16)
1	0658200002007000	RESIDENTIAL VACANT LAND	0
2	0773300004028000	ONE FAMILY DWELLING	1327
3	0773400001021000	TWO FAMILY DWELLING	2342
4	0773400003006000	TWO FAMILY DWELLING	2288
5	0774200002013000	COM VAC W/IMP	0

Fig. 7. Using Recursive

B. Data Manipulation

- For the INSERT INTO queries, the first one is inserting a new building style into the 'building_style' table with a unique ID and description, while the second one is inserting a new record into the 'main' table with various property details such as land value, sale price, and number of bedrooms.
- In the DELETE queries, the first one is deleting a record from the 'construction_grade' table where the construction grade is marked as unknown, while the second one is deleting any records from the 'house_location' table where the zip code value is null.
- For the UPDATE queries, the first one is updating a record in the 'construction_quality' table where the construction quality code is 2.8 and changing its description to 'Above average'. The second query is updating a record in the 'main' table where the SBL code is '0658200002007000' and changing its total value to 5500.

Query Query History

```
--Inserting
INSERT INTO building_style (building_style_code, building_style_description) VALUES (16, 'Cottage');


INSERT INTO main (sbl, front, depth, prop_class_description, roll, total_value, land_value, number_of_units, sale_price, year_built,
                  first_story_area, second_story_area, total_living_area, overall_condition, heat_type, no_of_stories, no_of_fireplaces,
                  no_of_beds, no_of_baths, no_of_kitchens, census_tract, census_block, acres, add_area, attic_area, building_style_code,
                  central_air, construction_quality_code, construction_grade, exterior_wall_code, homestead_code,
                  story_height, wall_a, wall_b, wall_c, property_class_code, geo1d20_block)
VALUES ('1114100005012001', 30, 80, 'COM VAC W/IMP', 8, 5000, 2500, 11, 480000, 2000,
        1500, 0, 1500, 3, 2, 1, 3, 2, 1, 2,
        '25.02','2012', 0.5, 0, 0, 10, 1, 1, 'UNKNOWN', 0,
        'N', 8, 100, 200, 300, 331, '360298025022012');

--Delete
DELETE FROM construction_grade WHERE construction_grade = 'UNKNOWN';

DELETE FROM house_location WHERE zipcode IS NULL;

--Updating
UPDATE construction_quality SET construction_quality_description = 'Above average' WHERE construction_quality_code = 2.8;

UPDATE main SET total_value = 5500 WHERE sbl = '0658200002007000';
```

Data Output Messages Notifications

```
UPDATE 1

Query returned successfully in 898 msec.
```

Fig. 8. Data Manipulation Queries

IX. Query Execution Analysis

In any database system, query performance is a critical aspect that directly impacts the overall system's efficiency. Identifying problematic queries and optimizing their execution can

significantly enhance the system's performance. This report aims to analyze three queries that are currently experiencing performance issues, by examining their respective costs. To address these issues, we will provide a detailed execution plan using the EXPLAIN tool in PostgreSQL, outlining our strategies for improving the queries' performance. The report's findings and recommendations will assist in optimizing the database system's performance and ensuring efficient query execution.

- The query was tested without an appropriate index on the attribute "sbl". The query execution time was observed to be longer. To improve the execution performance, appropriate indexes should be applied. In this experiment, the index on "sbl" was temporarily removed for testing purposes, and the results showed that the query execution time was significantly reduced after the index was reapplied.

Query Query History

```
ALTER TABLE deed DROP CONSTRAINT fk_deed_main;
ALTER TABLE house_location DROP CONSTRAINT fk_house_location_main;
ALTER TABLE owner_details DROP CONSTRAINT fk_owner_details_main;
ALTER TABLE main DROP CONSTRAINT main_pkey;

EXPLAIN ANALYZE
SELECT *
FROM main
JOIN house_location
ON main.sbl = house_location.sbl;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Hash Join (cost=5568.56..17567.29 rows=93714 width=301) (actual time=46.872..347.579 rows=93665 loops=1)
2	Hash Cond: ((main.sbl)::text = (house_location.sbl)::text)
3	-> Seq Scan on main (cost=0.00..4302.15 rows=93715 width=182) (actual time=0.067..54.030 rows=93715 loops=1)
4	-> Hash (cost=2749.14..2749.14 rows=93714 width=119) (actual time=46.231..46.237 rows=93665 loops=1)
5	Buckets: 32768 Batches: 4 Memory Usage: 3841kB
6	-> Seq Scan on house_location (cost=0.00..2749.14 rows=93714 width=119) (actual time=0.028..15.850 rows=93665 loops=1)
7	Planning Time: 8.712 ms
8	Execution Time: 350.421 ms

Fig. 9. Without Index

Query Query History

```
ALTER TABLE main
ADD CONSTRAINT main_pkey PRIMARY KEY (sbl);
ALTER TABLE deed
ADD FOREIGN KEY (sbl) REFERENCES main(sbl);
ALTER TABLE owner_details
ADD FOREIGN KEY (sbl) REFERENCES main(sbl);
ALTER TABLE house_location
ADD FOREIGN KEY (sbl) REFERENCES main(sbl);

EXPLAIN ANALYZE
SELECT *
FROM main
JOIN house_location
ON main.sbl = house_location.sbl;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Hash Join (cost=7853.59..16524.74 rows=93714 width=301) (actual time=132.766..245.849 rows=93665 loops=1)
2	Hash Cond: ((house_location.sbl)::text = (main.sbl)::text)
3	-> Seq Scan on house_location (cost=0.00..2749.14 rows=93714 width=119) (actual time=0.016..5.448 rows=93665 loops=1)
4	-> Hash (cost=4302.15..4302.15 rows=93715 width=182) (actual time=132.568..132.572 rows=93715 loops=1)
5	Buckets: 32768 Batches: 8 Memory Usage: 2825kB
6	-> Seq Scan on main (cost=0.00..4302.15 rows=93715 width=182) (actual time=0.018..46.831 rows=93715 loops=1)
7	Planning Time: 2.185 ms
8	Execution Time: 248.353 ms

Fig. 10. With Index

- This query selects all records from the main table where the year the property was built is after 1900, there are at least 3 bedrooms, and at least 2 bathrooms. Depending on the size of the main table, this query could be very slow. A possible optimization would be to create an index on the year_built, no_of_beds, and no_of_baths columns.

```
EXPLAIN ANALYZE
SELECT *
FROM public.main
WHERE year_built > 1900
AND no_of_beds >= 3
AND no_of_baths >= 2;
```

Data Output Messages Explain ✕ Notifications

	QUERY PLAN text
1	Seq Scan on main (cost=0.00..5005.01 rows=12579 width=182) (actual time=0.018..49.490 rows=232...
2	Filter: ((year_built > '1900'::numeric) AND (no_of_beds >= '3'::numeric) AND (no_of_baths >= '2'::numeri...
3	Rows Removed by Filter: 70431
4	Planning Time: 0.187 ms
5	Execution Time: 50.678 ms

Fig. 11. Without compound index

```
EXPLAIN ANALYZE
SELECT *
FROM public.main
WHERE year_built > 1900
AND no_of_beds >= 3
AND no_of_baths >= 2;
```

Data Output Messages Explain ✕ Notifications

	QUERY PLAN text
1	Seq Scan on main (cost=0.00..5005.01 rows=12579 width=182) (actual time=0.026..46.943 rows=232...
2	Filter: ((year_built > '1900'::numeric) AND (no_of_beds >= '3'::numeric) AND (no_of_baths >= '2'::numeri...
3	Rows Removed by Filter: 70431
4	Planning Time: 1.018 ms
5	Execution Time: 48.056 ms

Fig. 12. With compound index

- This query updates the total_value column in the main table for all records where the year the property was built is after 2000. Depending on the size of the main table, this query could be very slow. A possible optimization would be to create an index on the year_built column.

```
EXPLAIN ANALYZE
UPDATE public.main
SET total_value = total_value * 1.1
WHERE year_built > 2000;
```

Data Output Messages Explain ✕ Notifications

	QUERY PLAN text
1	Update on main (cost=0.00..4540.18 rows=749 width=328) (actual time=80.630..80.706 rows=0 loops=1)
2	-> Seq Scan on main (cost=0.00..4540.18 rows=749 width=328) (actual time=0.188..68.210 rows=662 loops=1)
3	Filter: (year_built > '2000'::numeric)
4	Rows Removed by Filter: 93053
5	Planning Time: 2.662 ms
6	Execution Time: 81.786 ms

Fig. 13. without index

Query Query History

```
CREATE INDEX main_year_built_idx ON main(year_built);

EXPLAIN ANALYZE
UPDATE main
SET total_value = total_value * 1.1
WHERE year_built > 2000;
```

Data Output Messages Explain × Notifications

	QUERY PLAN text
1	Update on main (cost=18.22..1822.39 rows=749 width=328) (actual time=18.105..18.107 rows=0 loops=1)
2	-> Bitmap Heap Scan on main (cost=18.22..1822.39 rows=749 width=328) (actual time=0.463..2.993 rows=662 loops=1)
3	Recheck Cond: (year_built > '2000'::numeric)
4	Heap Blocks: exact=220
5	-> Bitmap Index Scan on main_year_built_idx (cost=0.00..18.04 rows=749 width=0) (actual time=0.367..0.368 rows=662 loops=1)
6	Index Cond: (year_built > '2000'::numeric)
7	Planning Time: 0.651 ms
8	Execution Time: 18.250 ms

Fig. 14. with index

X. Database Objects

In this section, we present the various database components developed to enhance the functionality of our system. Firstly, we describe the creation of a trigger function that automatically updates a column in the ”main” table whenever specific conditions are met. Next, we detail the creation of a function named ”get_zipcode_stats” that generates statistical insights for a given zipcode. We also present the development of a stored procedure called ”get_main_data,” which retrieves the complete dataset by performing a natural join operation on all the tables stored in the database. These components contribute significantly to the automation and optimization of our system, making it more efficient and user-friendly. In the following subsections, we provide further details about each of these database components, including their functionality and implementation details, along with their respective diagrams.

- We have created trigger function that automatically updates the ”total_living_area” column of a ”main” table whenever the ”first_story_area” or ”second_story_area” columns are updated or a new row is inserted.

```
CREATE OR REPLACE FUNCTION update_total_living_area() RETURNS trigger AS $$
BEGIN
    NEW.total_living_area = NEW.first_story_area + NEW.second_story_area;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_total_living_area_trigger
    BEFORE INSERT OR UPDATE OF first_story_area, second_story_area
    ON public.main
    FOR EACH ROW
    EXECUTE FUNCTION update_total_living_area();

update main
set second_story_area = 1000
where sbl = '1114100005012001'
```

Fig. 15. Trigger

- We have also created a function named ”get_zipcode_stats” that takes ”zipcode” as input and returns a table with three columns, namely ”total_houses”, ”avg_total_value”, and ”avg_beds”. The function calculates these values by performing a join operation between two tables and filtering the results based on the input zipcode value.

```
CREATE OR REPLACE FUNCTION get_zipcode_stats(zipcode_val numeric(16,0))
RETURNS TABLE (
    total_houses bigint,
    avg_total_value numeric(16,0),
    avg_beds numeric(16,0)
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        count(*) as total_houses,
        avg(total_value) as avg_total_value,
        avg(no_of_beds) as avg_beds
    FROM
        house_location hl
        JOIN main m ON m.sbl = hl.sbl
    WHERE
        hl.zipcode = zipcode_val;
END;
$$ LANGUAGE plpgsql;


SELECT * FROM get_zipcode_stats(14214);
```

Fig. 16. Function

- In addition, a stored procedure named ”get_main_data” has been developed to retrieve the complete dataset by performing a natural join operation on all the tables stored in the database.

```
CREATE OR REPLACE PROCEDURE get_main_data()
LANGUAGE plpgsql
AS $$
BEGIN
  DROP TABLE IF EXISTS temp_main_data;
  CREATE TEMPORARY TABLE temp_main_data AS
SELECT *
FROM main
natural join building_style
natural join construction_grade
natural join construction_quality
natural join deed
natural join exterior_wall
natural join heat_type
natural join house_location
natural join overall_condition
natural join owner_details;
END;
$$;
```

Fig. 17. Stored Procedure

References

[1] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom’s ”DATABASE SYSTEMS The Complete Book,” Second Edition, published by Prentice Hall in 2008, provides an in-depth and comprehensive understanding of database systems, and served as a valuable resource for our research.

[2] Silberschatz, Korth, and Sudarshan’s book ”Database System Concepts” is a valuable reference for understanding the fundamental concepts of database systems and can be cited in this paper.