State Property management Database

Mahitha Balumuri
UBID: mahithab
Data Models and Query Languages
University at Buffalo, SUNY

Sahithi PSL
UBID: spabbath
Data Models and Query Languages
University at Buffalo, SUNY

Abstract—**There are several obstacles in organizing, gaining access to, and making use of crucial information when there isn't a centralized, effective system for handling property data for building. Stakeholders find it challenging to keep track of property details, evaluate valuations, and efficiently plan building operations in the absence of a specialized database system.**

I. Background

In modern civilization, infrastructure, urban landscapes, and economic progress are significantly shaped by building projects. Nonetheless, managing property data related to building projects effectively continues to be a concern. In the past, spreadsheets, manual procedures, and inconsistent systems were used for property data administration, which resulted in inefficiencies, errors, and restricted accessibility. A unified, effective system to handle property data is becoming more and more necessary as building projects get more intricate and multidimensional. Stakeholders, such as developers, contractors, architects, and regulatory agencies, face difficulties in organizing, gaining access to, and making use of vital information that is necessary for project planning, execution, and monitoring in the absence of a specialized database solution. Timelines, finances, and overall project performance are hampered by difficulties with managing construction operations, evaluating property specifics, and arranging construction.

A major weakness in the capacities of the construction sector in an age of data-driven decision-making and technology improvements is the lack of a strong property data management system. Thus, the development of all-encompassing database systems that facilitate property data administration, improve teamwork, and allow for well-informed decision-making throughout the building ecosystem is imperative.

Queries to Search:

- Find all properties owned by a specific owner and their deed details.
- To find the census tract for a particular property.
- To find the price details for a particular property.
- Find properties with their latest sale price over a certain amount.
- Get a list of properties, including owner and price details, that have changed hands in the last year.
- Find owners who have a property within a specific ZIP code with a land value exceeding a given amount.
- Find all properties of a certain class with their total living area within a specific range.

II. Potential

The potential of addressing the challenges posed by the lack of a centralized and efficient system for managing property data for construction is vast and multifaceted. By implementing a dedicated database solution, several opportunities arise to improve processes, enhance decision-making, and drive efficiency across the construction industry.

1. Streamlined Data Management: A centralized database allows for the consolidation of property data, streamlining the storage, organization, and retrieval of information related to construction projects. This ensures data integrity, reduces redundancy, and minimizes the risk of errors associated with manual processes or disparate systems.

2. Enhanced Collaboration: A dedicated database facilitates seamless collaboration among stakeholders involved in construction projects, including developers, architects, contractors, and regulatory authorities. By providing a unified platform for sharing and accessing property data, communication barriers are reduced, leading to improved coordination, alignment of objectives, and faster decision-making.

3. Improved Decision-Making: Access to accurate and up-to-date property data empowers decision-makers to make informed choices throughout the project lifecycle. By leveraging insights derived from the database, stakeholders can assess project feasibility, identify potential risks, allocate resources efficiently, and optimize project timelines and budgets.

4. Efficient Resource Allocation: A comprehensive database enables stakeholders to better understand property attributes, market trends, and regulatory requirements, allowing for more effective resource allocation. By aligning resources with project needs and market demands, organizations can minimize waste, maximize productivity, and enhance project outcomes.

5. Facilitated Compliance: Regulatory compliance is a critical aspect of construction projects, and a dedicated database can streamline compliance efforts by centralizing

relevant documentation, automating compliance checks, and providing audit trails. This ensures adherence to legal and regulatory requirements, mitigates compliance risks, and enhances overall project governance.

6. Support for Future Growth: As construction projects become increasingly complex and data-intensive, the scalability and flexibility of a dedicated database solution are crucial. By designing the database with scalability in mind, organizations can adapt to evolving project requirements, accommodate growth, and incorporate new technologies and functionalities as needed.

In conclusion, addressing the challenges of managing property data for construction through a dedicated database solution presents significant opportunities to improve efficiency, collaboration, decision-making, and compliance across the construction industry. By harnessing the potential of advanced data management technologies, organizations can unlock value, drive innovation, and achieve success in their construction endeavours.

III. POTENTIAL USERS

The target users for a dedicated database solution for managing property data for construction projects include developers, architects, contractors, regulatory authorities, real estate professionals, financial institutions, facility managers, and other project stakeholders. These users rely on property data to assess feasibility, coordinate construction activities, ensure regulatory compliance, evaluate market trends, facilitate transactions, assess project finances, manage facilities, and collaborate effectively throughout the construction lifecycle.

IV. PREPROCESSING

Upon acquiring the dataset from an external source, preprocessing became imperative due to the presence of missing data. Our efforts primarily revolved around null value removal, data type adjustments, duplicate record elimination, and table partitioning, with each table saved as an individual CSV file. Given the dataset's considerable size, we opted to process only the initial 25,000 rows. These meticulous preprocessing steps were vital to ensuring data quality and readiness for subsequent analytical endeavors and modeling tasks.
Upon acquisition, the datasets were initially copied into PGAdmin using the COPY statement in SQL, facilitating seamless integration into the database management system. This method ensured efficient data transfer and storage, laying the groundwork for subsequent preprocessing steps.
A sample query executed during this process could be:

```
COPY address_info(House_Number, Street, Zipcode)
FROM '/path/to/address_info.csv' DELIMITER ',' CSV
HEADER;
```

V. INITIAL RELATIONS

Table 1: OWNER info

- Owner1: The name of the current property owner. (Data type: VARCHAR).
- Previous Owner: The name of the previous property owner, if applicable. (Data type: VARCHAR).
- Mail3 and Mail4: Additional mailing information, which could include secondary address lines like apartment or suite numbers. (Data type: VARCHAR).

Table 2: House info

- House Number: The number assigned to a building or lot along a street to identify it. (Data type: INTEGER)
- Street: The name of the street where the property is located. (Data type: VARCHAR).
- City: The city where the property is located. (Data type: VARCHAR).
- State: The state where the property is located. (Data type: VARCHAR).
- Zipcode: The postal code associated with the property location. (Data type: VARCHAR)
- SBL: The unique identifier for a property, often used in tax assessment. (Data type: VARCHAR).

Table 3: Deed info

- Deed Book: The ledger book where property deeds are recorded for reference. (Data type: VARCHAR).
- Deed Page: The specific page number within the deed book where the deed can be found. (Data type: INTEGER).
- Deed Date: The date on which the deed was recorded. (Data type: DATE).
- Roll: A unique identifier for a group or page of recorded deeds. (Data type: VARCHAR).

Table 4: Census table

- Census_ID: A surrogate key to uniquely identify each census tract record. (Data type: INTEGER).
- Census_tract: The geographical region in an area defined for the census. (Data type: VARCHAR).

Table 5: Property details

- SBL: The unique identifier for the property(Data type: VARCHAR).
- Front: The width of the front of the property. (Data type: NUMERIC).
- Depth: The depth of the property from front to back. (Data type: NUMERIC).
- Property Class: A code that categorizes the type of property. (Data type: VARCHAR)
- Property Class Description: A text description of the property class code. (Data type: TEXT).
- Total Living Area: The square footage of livable space on the property. (Data type: NUMERIC).

Table 6: Price details

- Land Value: The value of the land portion of the property. (Data type: NUMERIC).
- Total Value: The total value of the property, combining land and buildings/improvements. (Data type: NUMERIC).
- Sale Price: The price at which the property was last sold. (Data type: NUMERIC).

Table 7: Link table

- SBL: A foreign key linking to Property details. (Data type: VARCHAR)
- Owner_ID: A foreign key potentially linking to an Owner info record. (Data type: INTEGER).
- deed_ID: A foreign key potentially linking to a Deed info record. (Data type: INTEGER).
- Price_ID: A foreign key potentially linking to a Price details record. (Data type: INTEGER).

VI. BCNF RELATIONS

owner_info Table:

- Owner_ID: A unique identifier for each owner (Data type: INT).
- Owner1: The name of the primary owner of the property (Data type: VARCHAR(255)).
- Previous_Owner: The name of the previous owner, if different from the current owner (Data type: VARCHAR(255)).
- Mail3: An additional field for mailing information, could be used for address or special instructions (Data type: VARCHAR(255)).
- Mail4: Another field for mailing information.(Data type: VARCHAR(255)).

Keys:

PK: Owner_ID

No FKs.

Gives each owner entry a unique identity. Even in the event that some information is duplicated, each owner may be uniquely referenced.

CREATE SQL Query:

```
CREATE TABLE owner_info (
   Owner_ID INT PRIMARY KEY,
   Owner1 VARCHAR(255),
   Previous_Owner VARCHAR(255),
   Mail3 VARCHAR(255),
   Mail4 VARCHAR(255)
);
```

property_info Table:

- SBL: The unique identifier for a property (Data type: VARCHAR(255)).
- Front: The measurement of the frontage of the property (Data type: NUMERIC).
- Depth: The depth of the property lot (Data type: NUMERIC).
- Property_Class: A code representing the type of property, such as residential, commercial (Data type: VARCHAR(50)).
- Total_Living_Area: The total living space area of the property (Data type: NUMERIC).

Keys:

PK: SBL

Unique identifier for each property, commonly used in property assessment databases to reference specific parcels of land.

CREATE SQL Query:

```
CREATE TABLE property_info (
   SBL VARCHAR(255) PRIMARY KEY,
   Front NUMERIC,
   Depth NUMERIC,
   Property_Class VARCHAR(50),
   Total_Living_Area NUMERIC
);
```

address_info Table:

- House_Number: The numerical identifier of the property within its street (Data type: INTEGER).
- Street: The name of the street on which the property is located (Data type: VARCHAR(255)).
- Zipcode: The postal code for the property location (Data type: VARCHAR(20)).
- SBL: Foreign Key, references the property_info table (Data type: VARCHAR(255)).

PK: Composite key (House_Number, Street, Zipcode)

This composite key uniquely identifies each address.

FK: SBL

Ensures referential integrity between the address and the specific property it belongs to. ON DELETE CASCADE ON UPDATE CASCADE actions maintain data integrity across related tables.

CREATE SQL Query:

```
CREATE TABLE address_info (
   SBL VARCHAR(255) REFERENCES
property_info(SBL) ON DELETE CASCADE ON
UPDATE CASCADE,
   House_Number INTEGER,
   Street VARCHAR(255),
   Zipcode VARCHAR(20),
          PRIMARY KEY (House_Number, Street,
Zipcode,SBL)
);
```

zipcode_info Table:

- Zipcode: The postal code, unique to a geographic area that may cover multiple streets (Data type: VARCHAR(20)).
- State: The U.S. state where the property is located (Data type: VARCHAR(255)).
- City: The city or locality where the property is located (Data type: VARCHAR(255)).

PK: Zipcode

Uniquely identifies an area's postal code, which is typically associated with a city and state.

CREATE SQL Query:

```
CREATE TABLE zipcode_info (
   Zipcode VARCHAR(20) PRIMARY KEY,
   State VARCHAR(255),
```

```
   City VARCHAR(255)
);
```

deed_info Table:

- Deed_ID: A unique identifier for each deed record (Data type: INT).
- Deed_Book: The book number where the deed is recorded (Data type: NUMERIC).
- Deed_Page: The page number of the deed book where the specific deed is recorded (Data type: NUMERIC).
- Deed_Date: The date the deed was recorded (Data type: DATE).
- Roll: An identifier that may be used for tax roll or other recording purposes (Data type: VARCHAR(255)).

PK: Deed_ID

A surrogate key that uniquely identifies each deed entry.

<u>CREATE SQL Query:</u>

```
CREATE TABLE deed_info (
   Deed_ID INT PRIMARY KEY,
   Deed_Book NUMERIC,
   Deed_Page NUMERIC,
   Deed_Date DATE,
   Roll VARCHAR(255)
);
```

census_info Table:

- Census_ID: A unique identifier for each census tract record (Data type: INT).
- Census_Tract: The census tract number which is a geographic region defined for the purpose of taking a census (Data type: VARCHAR(255)).

PK: Census_ID

A surrogate key assigned to ensure a unique identifier for each census tract entry

<u>CREATE SQL Query:</u>

```
CREATE TABLE census_info (
   Census_ID INT PRIMARY KEY,
   Census_Tract VARCHAR(255)
);
```

property_class_info Table:

- Property_Class: A unique code that categorizes the type of property (Data type: VARCHAR(50)).
- Property_Class_Description: A text description of the property class code (Data type: TEXT).

PK: Property_Class

Uniquely identifies the classification of a property, such as residential, commercial, etc.

<u>CREATE SQL Query:</u>

```
CREATE TABLE property_class_info (
   Property_Class VARCHAR(50) PRIMARY KEY,
   Property_Class_Description TEXT
);
```

price_info Table:

- Price_ID: A unique identifier for each set of price-related details for properties (Data type: INT).
- Land_Value: The assessed value of the land component of the property (Data type: NUMERIC).
- Total_Value: The total assessed value of the property, including land and improvements/buildings (Data type: NUMERIC).
- Sale_Price: The most recent sale price of the property (Data type: NUMERIC).

PK: Price_ID

A surrogate key to uniquely identify each set of pricing details for properties.

<u>CREATE SQL Query:</u>

```
CREATE TABLE price_info (
   Price_ID INT PRIMARY KEY,
   Land_Value NUMERIC,
   Total_Value NUMERIC,
   Sale_Price NUMERIC
);
```

link_table Table:

- SBL: Foreign Key, references the property_info table (Data type: VARCHAR(255)).
- Owner_ID: Foreign Key, references the owner_info table (Data type: INT).
- Deed_ID: Foreign Key, references the deed_info table (Data type: INT).
- Price_ID: Foreign Key, references the price_info table (Data type: INT).
- Census_ID: Foreign Key, references the census_info table; when the primary key it references is deleted, Census_ID is set to NULL (Data type: INT).

PK: Composite key (SBL, Owner_ID, Deed_ID, Price_ID, Census_ID)

This composite key connects entries across the different tables, ensuring a unique tuple for property, owner, deed, price, and census information.

FKs: SBL, Owner_ID, Deed_ID, Price_ID, Census_ID

Each FK links to its corresponding primary key in the relevant table, ensuring referential integrity. For Census_ID, ON DELETE SET NULL allows for the census information to be optional or updated without affecting the rest of the link table entries.

<u>CREATE SQL Query:</u>

```
CREATE TABLE link_table (
   SBL VARCHAR(255) REFERENCES
property_info(SBL) ON DELETE CASCADE ON
UPDATE CASCADE,
   Owner_ID INT REFERENCES owner_info(Owner_ID)
ON DELETE CASCADE ON UPDATE CASCADE,
```

```
    Deed_ID INT REFERENCES deed_info(Deed_ID) ON DELETE CASCADE ON UPDATE CASCADE,
    Price_ID INT REFERENCES price_info(Price_ID) ON DELETE CASCADE ON UPDATE CASCADE,
    Census_ID INT REFERENCES census_info(Census_ID) ON DELETE SET NULL ON UPDATE CASCADE,
    PRIMARY KEY (SBL, Owner_ID, Deed_ID, Price_ID, Census_ID)
);
```

VII. Dependencies

Owner_info

Owner_ID -> Owner1, Previous Owner, Mail3, Mail4

This relation is accurately defined because `Owner_ID` uniquely identifies all other attributes, establishing a functional dependency.

In Boyce-Codd Normal Form (BCNF), if the left side of the functional dependency is a key and the relation is non-trivial, it is considered to be in BCNF. In this case, both conditions are met, affirming that the table is indeed in BCNF. Additionally, there are no further nested relations within this table. Therefore, it can be concluded that this relation satisfies BCNF criteria, ensuring data integrity and minimizing redundancy effectively.

Property_info

SBL-> Front, depth, property_class, Total_living_area

Property_class->Property class description

In the given relation, where `SBL` (which presumably stands for "Serial Block Number") serves as the primary key, the functional dependency is not entirely consistent with Boyce-Codd Normal Form (BCNF). While `SBL` uniquely identifies all other attributes (i.e., `Front`, `depth`, `property_class`, and `Total_living_area`), the dependency on `property_class` introduces a potential issue. If `property_class` is non-prime (not part of the primary key), it implies a partial dependency, violating BCNF.

To resolve this, we decompose the relation into two tables: R1 with attributes `SBL`, `Front`, `depth`, `property_class`, and `Total_living_area`, and R2 with attributes `property_class` and `Property class description`. Now, R1 holds the first table correctly because `SBL` remains the primary key, and all other attributes are fully functionally dependent on it. Similarly, R2 holds the second table correctly because `property_class` is now the primary key, uniquely identifying each tuple, while `Property class description` is fully functionally dependent on it. This decomposition ensures adherence to BCNF, maintaining data integrity and reducing redundancy effectively. In our database, we call the R2 table as property_class_info table.

Address_info table

Zipcode -> State, City

In the `address_info` table, the combination of `Zipcode`, `State`, and `City` does not form the primary key; rather, it represents a functional dependency where `Zipcode` uniquely determines `State` and `City`. However, considering additional columns such as `street` and `house_number`, we define the primary key as the combination of `House_Number`, `Street`, and `Zipcode`. This primary key ensures each record in the table is uniquely identified, allowing for effective data retrieval and integrity. As a result, the original relation violates Boyce-Codd Normal Form (BCNF) because the left side of the functional dependency (`Zipcode`) is not a key, yet the relation is non-trivial. However, the combination of `House_Number`, `Street`, and `Zipcode` acts as a key and defines all the attributes, ensuring the functional dependency holds. To address this violation, we decompose the table into two relations: one for `zipcode_info` containing `Zipcode`, `State`, and `City`, and another for the remaining attributes (`House_Number`, `Street`, `Zipcode`). This decomposition aligns with BCNF principles, ensuring efficient data management and minimal redundancy while maintaining data integrity.

Dependency for (`House_Number`, `Street`, `Zipcode) -> (`House_Number`, `Street`, `Zipcode)

Deed_info table

Deed_ID -> Deed_Book, Deed_page, Deed_date, Roll

In the `Deed_info` table, `Deed_ID` serves as the primary key, uniquely identifying each record in the table. The other attributes (`Deed_Book`, `Deed_page`, `Deed_date`, `Roll`) are functionally dependent on `Deed_ID`, forming a relation where each `Deed_ID` corresponds to a specific combination of `Deed_Book`, `Deed_page`, `Deed_date`, and `Roll`. This setup ensures data integrity and efficient retrieval of information. As `Deed_ID` uniquely identifies each record, and all other attributes are fully functionally dependent on it, the relation complies with Boyce-Codd Normal Form (BCNF), enhancing database organization and minimizing redundancy effectively.

Census_info

Census_ID -> census_track

In the `Census_info` table, `Census_ID` functions as the primary key, uniquely identifying each entry within the table. The attribute `census_track` is dependent on `Census_ID`, indicating a relationship where each `Census_ID` corresponds to a specific `census_track`. This design ensures data integrity and facilitates efficient data retrieval. With `Census_ID` serving as the primary key and all other attributes being fully functionally dependent on it, the relation conforms to Boyce-Codd Normal Form (BCNF). This adherence to BCNF principles aids in organizing the database effectively and minimizing redundancy, ensuring optimal database management.

Price_info

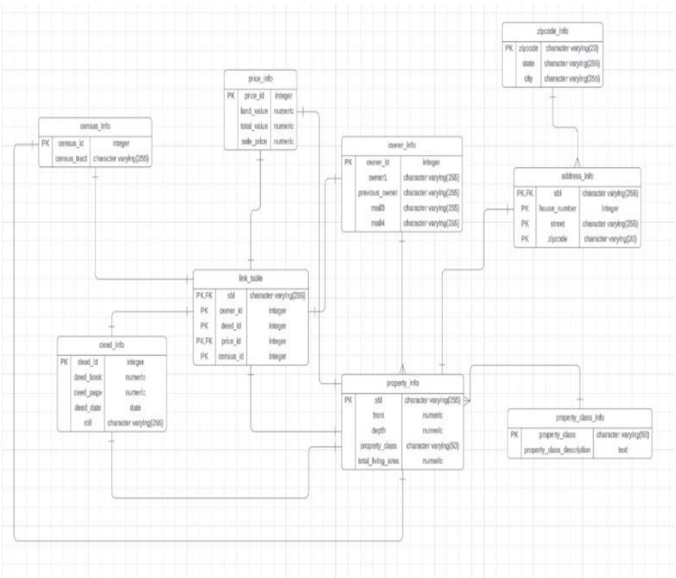
price_id -> land_value, total_value, sale_price

In the `Price_info` table, `price_id` serves as the primary key, uniquely identifying each entry. The attributes `land_value`, `total_value`, and `sale_price` are dependent on `price_id`, forming a relationship where each `price_id` corresponds to specific pricing details. This structure ensures data integrity and facilitates efficient retrieval of pricing information. With `price_id` as the primary key and all other attributes fully functionally dependent on it, the relation conforms to Boyce-Codd Normal Form (BCNF), supporting effective database management and minimizing redundancy.

link_table
In the `link_table` table, the dependencies can be described as follows:
- SBL → Property Details: Each Serial Block Number (SBL) uniquely identifies a property, establishing a functional dependency where SBL determines the associated property details.
- Owner_ID → Owner Information: Each Owner ID (Owner_ID) uniquely identifies an owner, forming a functional dependency where Owner_ID determines the corresponding owner information.
- Deed_ID → Deed Information: Each Deed ID (Deed_ID) uniquely identifies a deed, establishing a functional dependency where Deed_ID determines the related deed information.
- Price_ID → Price Information: Each Price ID (Price_ID) uniquely identifies pricing details, forming a functional dependency where Price_ID determines the associated price information.
- Census_ID → Census Information: Each Census ID (Census_ID) uniquely identifies census data, establishing a functional dependency where Census_ID determines the related census information.

In the `link_table` relation, each foreign key (SBL, Owner_ID, Deed_ID, Price_ID, Census_ID) forms the left side of a functional dependency. These foreign keys reference the primary keys of other tables, making them part of a superkey. For example, the SBL foreign key references the primary key of the property_info table, ensuring that SBL uniquely identifies each property and thus qualifies as a superkey. Additionally, the right side of each dependency consists of prime attributes from the referenced tables, fulfilling the condition for Boyce-Codd Normal Form (BCNF). Therefore, the `link_table` relation adheres to BCNF, maintaining data integrity and minimizing redundancy effectively.

VIII. ER DIAGRAM



owner_info to link_table:
Likely one-to-many (1:N) as one owner can be associated with multiple property transactions or deeds but each link_table entry is associated with one owner.

deed_info to link_table:
Likely one-to-many (1:N) as a single deed entry might pertain to multiple property transactions over time, but each link_table entry should reference one unique deed.

price_info to link_table:
Likely one-to-many (1:N) as a single set of price details could be associated with multiple properties (in the case of re-assessments, etc.), but each link_table entry references one price_info.

census_info to link_table:
Likely one-to-many (1:N) as one census tract can encompass multiple properties, but each link_table entry is associated with one census tract.

property_info to address_info:
Likely one-to-one (1:1) assuming each SBL is unique to one property and thus has one unique address.

property_info to link_table:
Likely one-to-many (1:N) because each property can have different associated transactions, owners, or price assessments over time, captured in the link_table.

property_info to property_class_info:
Likely many-to-one (N:1) as many properties can share the same property class but each property is assigned one property class.

address_info to zipcode_info:
Likely many-to-one (N:1) since many addresses can share the same zipcode, which corresponds to one city and state.

link_table is a junction table that captures the relationships between properties, their owners, the deeds associated with them, their price assessments, and their census information.

zipcode_info provides a reference for city and state based on the zipcode, which is used by addresses in address_info.

property_info keeps the property details like front, depth, and living area, and it refers to a general class in property_class_info, which describes the type of property.

link Table to All Tables (One-to-One): Each record in the link_table is associated with one and only one specific record in the owner_info, deed_info, price_info, and census_info tables. This suggests a very rigid structure where every aspect of a property (ownership, deed, pricing, and census details) is encapsulated in a single linked record, which is not common in typical real estate or property databases.

Census to Property (One-to-One): Each property is linked to a unique census tract entry. This would be a one-to-one (1:1) relationship, which is unusual since census tracts typically encompass multiple properties. If this is indeed the intended design, it might reflect a highly specialized dataset where each property is the only one in its census tract or where the census_info actually represents a smaller division within a tract.

Price to Property (One-to-One): This relationship indicates that each property has a single, unique set of pricing information. It's one-to-one (1:1), which is standard for representing the assessed value and sale price of a property as these figures are inherently tied to individual properties.

Owner to Property (One-to-Many): The owner_info to property_info relationship is one-to-many (1:N), indicating that while a single owner can own multiple properties, each property in the database is recorded with a single owner. This could reflect the primary ownership and does not account for joint or multiple ownership scenarios.

IX. QUERIES

Query 1 :

```sql
SELECT z.Zipcode, AVG(p.Sale_Price) AS Avg_Sale_Price
FROM address_info a
JOIN zipcode_info z ON a.Zipcode = z.Zipcode
JOIN link_table l ON a.SBL = l.SBL
JOIN price_info p ON l.Price_ID = p.Price_ID
GROUP BY z.Zipcode;
```

	zipcode [PK] character varying (20)	avg_sale_price numeric
1	14213.0	121757.214285714286
2	14201.0	204812.740259740260
3	14208.0	20012.454422990570
4	14222.0	222856.021178027796
5	14203.0	823883.117154811715
6	14207.0	31219.125000000000
7	14220.0	55756.412307692308
8	14210.0	38306.262511803588
9	14215.0	21831.515477792732
10	14212.0	17140.693271295634
11	14209.0	131115.265363128492
12	14211.0	18582.590683716075
13	14214.0	1084400.200000000000
14	14216.0	112293.500000000000
15	14202.0	445106.458407079646
16	14206.0	40023.704945799458
17	14204.0	49431.947029348604
18	14200.0	89500.000000000000
19	14225.0	58868.600000000000

Total rows: 19 of 19   Query complete 00:00:00.108

This SQL query calculates the average sale price of properties grouped by zipcode. It joins four tables: address_info, zipcode_info, link_table, and price_info to aggregate property sale prices by their respective zipcodes. The resulting dataset lists each unique zipcode with the average sale price of properties located within that zipcode.

Query 2:

```sql
select * from zipcode_info

INSERT INTO zipcode_info (zipcode, state, city)
VALUES ('14205', 'NY', 'Buffalo');

select * from zipcode_info
```

	zipcode [PK] character varying (20)	state character varying (255)	city character varying (255)
1	14200.0	NY	BUFFALO
2	14201.0	NY	BUFFALO
3	14202.0	NY	BUFFALO
4	14203.0	NY	BUFFALO
5	14204.0	NY	BUFFALO
6	14206.0	NY	BUFFALO
7	14207.0	NY	BUFFALO
8	14208.0	NY	BUFFALO
9	14209.0	NY	BUFFALO
10	14210.0	NY	BUFFALO
11	14211.0	NY	BUFFALO
12	14212.0	NY	BUFFALO
13	14213.0	NY	BUFFALO
14	14214.0	NY	BUFFALO
15	14215.0	NY	BUFFALO
16	14216.0	NY	BUFFALO
17	14220.0	NY	BUFFALO
18	14222.0	NY	BUFFALO
19	14225.0	NY	BUFFALO

Total rows: 19 of 19   Query complete 00:00:00.062

The INSERT statement adds a new row with the zipcode '14205', state 'NY', and city 'Buffalo' to the zipcode_info table. Following this, the SELECT * FROM zipcode_info command fetches and displays all records from the zipcode_info table to confirm the insertion and show the current state of data.

Query 3:

```sql
select * from zipcode_info

delete from zipcode_info where zipcode = '14205'

select * from zipcode_info
```

```
DELETE 1

Query returned successfully in 37 msec.
```

This SQL sequence deletes the record with the zipcode '14205' from the zipcode_info table and verifies the deletion by displaying the table contents before and after the operation.

Query 4:

```sql
select * from owner_info where owner1 = 'ROSS TRACEY L'

update owner_info set previous_owner='ROSS TRACEY L',owner1 = 'ROSS K' where owner1 = 'ROSS TRACEY L'

select * from owner_info where owner1 = 'ROSS TRACEY L'
```

```
UPDATE 1

Query returned successfully in 31 msec.
```

This SQL sequence updates the owner_info table by changing the previous_owner field to 'ROSS TRACEY L' and the owner1 field to 'ROSS K' for records where the current owner1 is 'ROSS TRACEY L'. It also includes checks to verify the content of the table before and after the update.

Query 5:

Query Query History

```sql
SELECT pr.Price_ID, pr.Land_Value, pr.Total_Value, pr.Sale_Price
FROM price_info pr
WHERE pr.Land_Value > (SELECT AVG(Land_Value) FROM price_info);
```

Data Output Messages Notifications

	price_id [PK] integer	land_value numeric	total_value numeric	sale_price numeric
1	3	24000	270000	17000
2	4	96200	381000	1
3	7	295900	950000	1
4	21	136000	162000	1
5	24	53000	63000	1
6	34	47900	292000	1
7	39	22000	176000	1
8	40	59000	216000	230000
9	47	62000	307000	28500
10	48	45900	199000	229900
11	51	71100	218000	66000
12	52	75400	324000	186000
13	55	77100	286000	1

Total rows: 1000 of 3798 Query complete 00:00:00.086

This SQL query retrieves the Price_ID, Land_Value, Total_Value, and Sale_Price for properties in the price_info table where the Land_Value exceeds the average land value across all records in the same table. It effectively identifies properties with land values above the average, which could be used for market analysis or premium property segmentation.

Query 6:

Query Query History

```sql
SELECT oi.Owner1, zi.City, COUNT(*) AS Property_Count
FROM owner_info oi
JOIN link_table lt ON oi.Owner_ID = lt.Owner_ID
JOIN property_info pri ON lt.SBL = pri.SBL
JOIN address_info ai ON pri.SBL = ai.SBL
JOIN zipcode_info zi ON ai.Zipcode = zi.Zipcode
GROUP BY oi.Owner1, zi.City
HAVING COUNT(DISTINCT pri.SBL) > 1
ORDER BY Property_Count DESC;
```

Data Output Messages Notifications

	owner1 character varying (255)	city character varying (255)	property_count bigint
1	CITY BUFFALO PERFECTING TITLE	BUFFALO	1676
2	CITY BUFFALO PERFECTING	BUFFALO	621
3	CITY OF BUFFALO	BUFFALO	269
4	CITY OF BUFFALO PERFECTING	BUFFALO	235
5	HYDRAULIC HOUSE LLC	BUFFALO	66
6	WILLIAMSVILLE PROPERTIES	BUFFALO	60
7	BUFFALO NEIGHBORHOOD	BUFFALO	46
8	CITY OF BUFFALO PERFECTING TIT	BUFFALO	35
9	CITY BUFFALO	BUFFALO	34
10	BAILEY GREEN LLC	BUFFALO	21
11	MADONNA OF THE STREETS	BUFFALO	19
12	ST PAUL GROUP LLC	BUFFALO	17
13	RUSINEK RONALD	BUFFALO	15

Total rows: 1000 of 2312 Query complete 00:00:00.387

This SQL query lists property owners and the cities where they own more than one property, sorted by the total number of properties they own in descending order. It joins several tables including owner_info, link_table, property_info, address_info, and zipcode_info to compile this information, grouping by both owner and city to facilitate localized property management insights.

Query 7:

Query Query History

```sql
SELECT di.*, ci.*
FROM link_table lt
JOIN deed_info di ON lt.Deed_ID = di.Deed_ID
JOIN census_info ci ON lt.Census_ID = ci.Census_ID
ORDER BY di.Deed_ID DESC;
```

Data Output Messages Notifications

	deed_id integer	deed_book numeric	deed_page numeric	deed_date date	roll character varying (255)	census_id integer	census_tract character varying (255)
1	25000	11287.0	1473.0	2015-10-26	1	25000	167001.0
2	24999	11345.0	1738.0	2019-04-11	1	24999	23004.0
3	24998	11340.0	638.0	2019-01-18	1	24998	23001.0
4	24997	11190.0	8016.0	2010-10-27	1	24997	164004.0
5	24996	11077.0	7115.0	2004-06-16	1	24996	25022.0
6	24995	11248.0	9163.0	2013-06-28	1	24995	166001.0
7	24994	11094.0	1001.0	2005-04-21	1	24994	10003.0
8	24993	11132.0	7857.0	2007-08-03	1	24993	23004.0
9	24992	11276.0	8714.0	2015-03-13	1	24992	167002.0
10	24991	11239.0	3271.0	2012-05-18	1	24991	28002.0
11	24990	10929.0	2139.0	1998-03-18	1	24990	164003.0
12	24989	11280.0	6826.0	2015-06-10	1	24989	167003.0
13	24988	11153.0	4506.0	2008-12-05	1	24988	167002.0

Total rows: 1000 of 24230 Query complete 00:00:00.118

This SQL query retrieves detailed information on property deeds and corresponding census data, ordered by deed ID in descending order. It joins the deed_info and census_info tables through the link_table, providing a comprehensive view of each property's deed and its census tract information, facilitating analyses that integrate legal and demographic insights.

Query 8:

Query Query History

```sql
SELECT oi.Owner1, SUM(pi.Total_Value) AS Total_Property_Value
FROM owner_info oi
JOIN link_table lt ON oi.Owner_ID = lt.Owner_ID
JOIN price_info pi ON lt.Price_ID = pi.Price_ID
GROUP BY oi.Owner1
ORDER BY Total_Property_Value DESC
LIMIT 5;
```

Data Output Messages Notifications

	owner1 character varying (255)	total_property_value numeric
1	PEOPLE OF THE STATE OF NEWY...	74800000
2	PEOPLE OF THE STATE OF NY	52893300
3	EGP 130 BUFFALO LLC	50058900
4	CITY BUFFALO	34107700
5	TCC BUILDING "R" ASSOCIATES LP	30300000

This SQL query calculates the total property value owned by each property owner and returns the top five owners with the highest total property values. It joins the owner_info, link_table, and price_info tables to aggregate the total values and is ordered in descending order to highlight the owners with the most valuable property portfolios.

Query 9:

Query | Query History

```sql
SELECT ci.Census_Tract, COUNT(lt.SBL) AS Property_Count
FROM census_info ci
LEFT JOIN link_table lt ON ci.Census_ID = lt.Census_ID
GROUP BY ci.Census_Tract
ORDER BY Property_Count DESC;
```

Data Output | Messages | Notifications

	census_tract character varying (255)	property_count bigint
1	36003.0	457
2	37002.0	411
3	35004.0	393
4	28001.0	377
5	168004.0	377
6	25022.0	359
7	168001.0	345
8	30002.0	342
9	28002.0	341
10	168002.0	340
11	15002.0	339
12	38003.0	337
13	37004.0	329

Total rows: 165 of 165 Query complete 00:00:00.069

This SQL query identifies the number of properties in each census tract by counting the SBL identifiers from the link_table for each Census_ID associated with a census tract in the census_info table. The results are grouped by Census_Tract and ordered in descending order to show which tracts have the highest concentration of properties.

Query 10:

Query | Query History

```sql
SELECT pi.SBL, pi.Property_Class, pr.Sale_Price
FROM link_table lt
JOIN property_info pi ON lt.SBL = pi.SBL
JOIN price_info pr ON lt.Price_ID = pr.Price_ID
WHERE pr.Sale_Price = (SELECT MAX(Sale_Price) FROM price_info);
```

Data Output | Messages | Notifications

	sbl character varying (255)	property_class character varying (50)	sale_price numeric
1	1114500001025000	464	89000000

Total rows: 1 of 1 Query complete 00:00:00.063

This SQL query retrieves the Serial Block Number (SBL), Property Class, and Sale Price for the property with the highest sale price in the database. It joins the link_table, property_info, and price_info tables to correlate property identifiers with their pricing details, specifically filtering for the property that matches the maximum sale price found in the price_info table.

Query 11:

Query | Query History

```sql
CREATE OR REPLACE FUNCTION update_property_class()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.Property_Class NOT IN (SELECT Property_Class FROM property_class_info) THEN
        INSERT INTO property_class_info (Property_Class, Property_Class_Description)
        VALUES (NEW.Property_Class, NULL);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_property_class_trigger
AFTER INSERT ON property_info
FOR EACH ROW
EXECUTE FUNCTION update_property_class();

select * from property_class_info
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 47 msec.
```

Total rows: 99 of 99 Query complete 00:00:00.047

```sql
select * from property_class_info

INSERT INTO property_info (SBL, Front, Depth, Property_Class, Total_Living_Area)
VALUES ('123456789', 50, 100, '211', 2000);
```

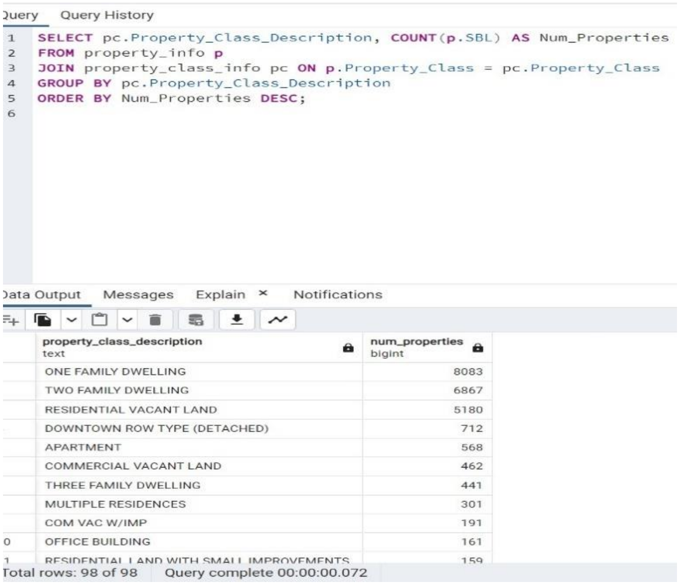
Data Output | Messages | Notifications

property_class [PK] character varying (50)	property_class_description text
714	LITE IND MANFTR
833	RADIO
834	TELEVISION OTHER THAN COMMUNITY ANTENNA
836	COMMUNICATIONS
837	CELL TOWER
851	SOLID WASTES
872	ELEC - SUBSTATION
873	GAS MEAS STATION
960	PUBLIC PARK
963	CITY/TOWN/VILLAGE PUBLIC PARKS
211	[null]

Total rows: 99 of 99 Query complete 00:00:00.058

This SQL sequence includes a PL/pgSQL function and a trigger to maintain the property_class_info table's integrity and consistency. The function update_property_class() checks if the Property_Class of a new row inserted into the property_info table exists in the property_class_info table. If it doesn't, the function inserts the new Property_Class into property_class_info with a NULL description. The trigger update_property_class_trigger is set to activate after each insert operation on property_info, ensuring that any new property classes are recorded appropriately. The INSERT INTO property_info statement tests the trigger by adding a new property entry, which should invoke the trigger to check and possibly update the property_class_info table.

Query 12:



```sql
SELECT pc.Property_Class_Description, COUNT(p.SBL) AS Num_Properties
FROM property_info p
JOIN property_class_info pc ON p.Property_Class = pc.Property_Class
GROUP BY pc.Property_Class_Description
ORDER BY Num_Properties DESC;
```

property_class_description text	num_properties bigint
ONE FAMILY DWELLING	8083
TWO FAMILY DWELLING	6867
RESIDENTIAL VACANT LAND	5180
DOWNTOWN ROW TYPE (DETACHED)	712
APARTMENT	568
COMMERCIAL VACANT LAND	462
THREE FAMILY DWELLING	441
MULTIPLE RESIDENCES	301
COM VAC W/IMP	191
OFFICE BUILDING	161
RESIDENTIAL LAND WITH SMALL IMPROVEMENTS	159

Total rows: 98 of 98 Query complete 00:00:00.072

This SQL query determines the number of properties in each property class by counting the SBL values from the property_info table, grouped by their corresponding descriptions from the property_class_info table. The results are ordered in descending order to display the property classes with the highest number of properties at the top, providing a clear overview of property distribution by type.

X. SIGNIFICANT CHALLENGES

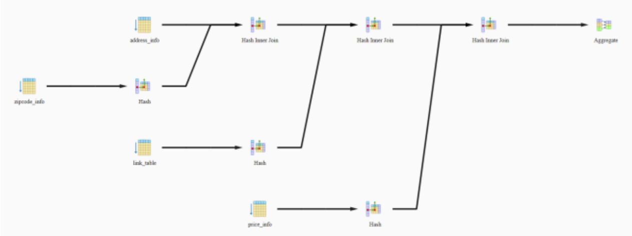
When dealing with the extensive dataset in our database, we had substantial difficulties with the performance of searches, especially with queries that took an inordinate amount of time to run. The main cause of this problem was the ineffective retrieval of data during join operations and the aggregation procedures, which were further amplified by the large amount of data being processed.

In order to tackle these limitations in performance, we used a strategic methodology by including indexing on pivotal columns that were often utilized in join conditions and where clauses. As an example, we have established indexes on the Owner_ID column in the owner_info database, and on the SBL and Price_ID columns in the link_table and price_info tables, respectively. The implementation of these indexes greatly enhanced the query execution performance by maximizing the database's capacity to swiftly discover and obtain the required data.

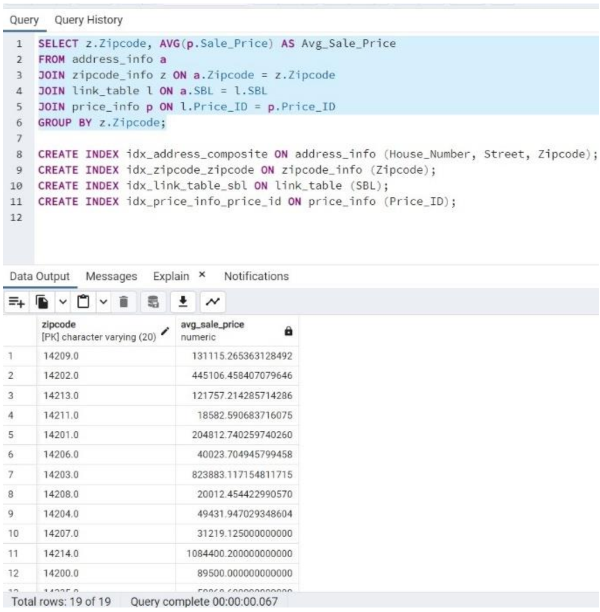
The use of these indexes decreased the time complexity of looking through extensive amounts of data, consequently improving the overall efficiency of our database operations. This method not only alleviated the immediate problems of sluggish query times, but also enhanced the system's performance, making it more resilient to the difficulties presented by large-scale data processing.

XI. QUERY OPTIMIZATIONS

Query 1 optimization:



The query's execution plan is built to take use of indexes and hash joins for effective data retrieval. The newly constructed composite and single-column indexes are used by the plan to hash the address_info and zipcode_info tables first. By enabling speedier access to the required data, these indexes accelerate joins and hash operations. The data are combined to determine the average selling price categorized by zipcode after the hash joins between the price_info, link_table, and zipcode_info have been completed. The output is supplied on time and in the intended order thanks to the final sorting of results, which is made possible by the effective use of indexes. The efficiency of indexing in streamlining SQL queries with several joins and aggregations is shown by the noticeably quicker execution time that this organized method yields.



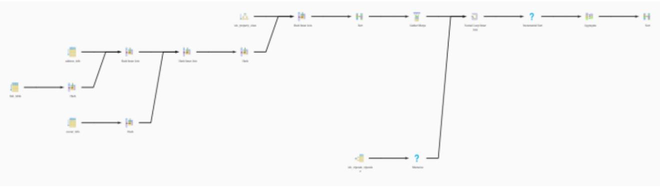
```sql
SELECT z.Zipcode, AVG(p.Sale_Price) AS Avg_Sale_Price
FROM address_info a
JOIN zipcode_info z ON a.Zipcode = z.Zipcode
JOIN link_table l ON a.SBL = l.SBL
JOIN price_info p ON l.Price_ID = p.Price_ID
GROUP BY z.Zipcode;

CREATE INDEX idx_address_composite ON address_info (House_Number, Street, Zipcode);
CREATE INDEX idx_zipcode_zipcode ON zipcode_info (Zipcode);
CREATE INDEX idx_link_table_sbl ON link_table (SBL);
CREATE INDEX idx_price_info_price_id ON price_info (Price_ID);
```

	zipcode [PK] character varying (20)	avg_sale_price numeric
1	14209.0	131115.265363128492
2	14202.0	445106.458407079646
3	14213.0	121757.214285714286
4	14211.0	18582.590683716075
5	14201.0	204812.740259740260
6	14206.0	40023.704945799458
7	14203.0	823883.117154811715
8	14208.0	20012.454422990570
9	14204.0	49431.947029348604
10	14207.0	31219.125000000000
11	14214.0	1084400.200000000000
12	14200.0	89500.000000000000

Total rows: 19 of 19 Query complete 00:00:00.067

The optimization of the query focused on enhancing join efficiency by creating several targeted indexes on key columns used in the joins, which greatly improved the query's execution time. Indexes such as idx_address_composite on the address_info table and idx_zipcode_zipcode on the zipcode_info table facilitated faster access to data by reducing the lookup time during join operations. Additional indexes on link_table and price_info for columns SBL and Price_ID respectively ensured that the database engine could quickly filter and aggregate data based on the sales price. These improvements led to a more streamlined execution plan, allowing the database to utilize the indexes for rapid retrieval and efficient computation of the average sale price grouped by zipcode. As a result, the query's execution time was significantly reduced from 0.108 seconds to 0.067 seconds, demonstrating a substantial increase in performance due to these optimizations. This not only made the query faster but also reduced the load on the database, making it more efficient for handling similar queries in the future.

Query 6 Optimization:



The execution plan illustrates that the query performance was initially constrained by multiple join operations and the absence of efficient pathways for data retrieval. To address this, the optimization strategy focused on introducing appropriate indexing, specifically targeting the SBL column in the property_info table, which is crucial for the join conditions. The introduction of the idx_property_class index aimed to streamline the process by reducing the cost associated with searching and retrieving rows across the linked tables. By enhancing the access path with this index, the database system could leverage more efficient hash join operations, thus reducing the time required to execute the joins and subsequently improving the overall query execution time. This strategic use of indexing effectively minimized the load and processing time, optimizing the query's performance as evidenced by the shorter execution duration in the optimized plan.

Query | Query History

```sql
SELECT oi.Owner1, zi.City, COUNT(*) AS Property_Count
FROM owner_info oi
JOIN link_table lt ON oi.Owner_ID = lt.Owner_ID
JOIN property_info pri ON lt.SBL = pri.SBL
JOIN address_info ai ON pri.SBL = ai.SBL
JOIN zipcode_info zi ON ai.Zipcode = zi.Zipcode
GROUP BY oi.Owner1, zi.City
HAVING COUNT(DISTINCT pri.SBL) > 1
ORDER BY Property_Count DESC;

CREATE INDEX idx_property_class ON property_info(SBL);
```

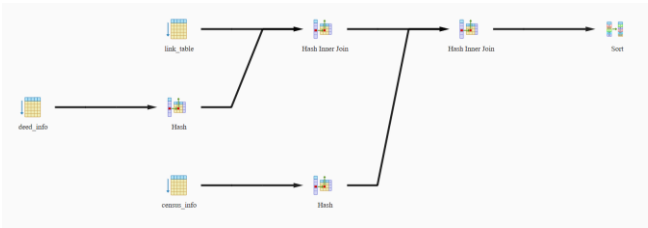
Data Output | Messages | Explain | Notifications

	owner1 character varying (255)	city character varying (255)	property_count bigint
1	CITY BUFFALO PERFECTING TITLE	BUFFALO	1676
2	CITY BUFFALO PERFECTING	BUFFALO	621
3	CITY OF BUFFALO	BUFFALO	269
4	CITY OF BUFFALO PERFECTING	BUFFALO	235
5	HYDRAULIC HOUSE LLC	BUFFALO	66
6	WILLIAMSVILLE PROPERTIES	BUFFALO	60
7	BUFFALO NEIGHBORHOOD	BUFFALO	46
8	CITY OF BUFFALO PERFECTING TIT	BUFFALO	35
9	CITY BUFFALO	BUFFALO	34
10	BAILEY GREEN LLC	BUFFALO	21
11	MADONNA OF THE STREETS	BUFFALO	19

Total rows: 1000 of 2312 Query complete 00:00:00.311

The optimization of Query 6 involved creating an index on the SBL field of the property_info table to enhance the performance of join operations used in the query. This field is crucial as it links property_info with link_table and address_info, thereby impacting the overall efficiency of the data retrieval process. The creation of the index idx_property_class on property_info(SBL) significantly improved the speed of the join operations by reducing the time required to locate the relevant rows in the property_info table. As a result, the execution time of the query decreased from 0.387 seconds to 0.311 seconds. This reduction underscores the importance of proper indexing in optimizing SQL queries, particularly those that involve multiple joins on large datasets, leading to more efficient and faster data processing.

Query 7 Optimization:



The execution plan demonstrates that PostgreSQL employs a series of hash operations followed by hash inner joins to merge data from the link_table, deed_info, and census_info tables. After hashing the data from each table, hash joins efficiently combine rows based on the join condition. The process concludes with a sort operation, which is likely necessary to order the results as specified in the query, ensuring that the output is correctly sequenced according to the specified ORDER BY clause. This method is particularly efficient for handling large datasets where hash-based methods can significantly speed up join operations compared to other methods like nested loops or merge joins.

Query | Query History

```sql
SELECT di.*, ci.*
FROM link_table lt
JOIN deed_info di ON lt.Deed_ID = di.Deed_ID
JOIN census_info ci ON lt.Census_ID = ci.Census_ID
ORDER BY di.Deed_ID DESC;

-- Index creation for link_table
CREATE INDEX idx_link_table_owner_id ON link_table(Owner_ID);
CREATE INDEX idx_link_table_deed_id ON link_table(Deed_ID);
CREATE INDEX idx_link_table_price_id ON link_table(Price_ID);
CREATE INDEX idx_link_table_census_id ON link_table(Census_ID);

-- Index creation for census_info
CREATE INDEX idx_census_info_census_id ON census_info(Census_ID);

-- Create indexes for the columns used in ORDER BY clause
CREATE INDEX idx_deed_id_desc ON deed_info(Deed_ID DESC);
```

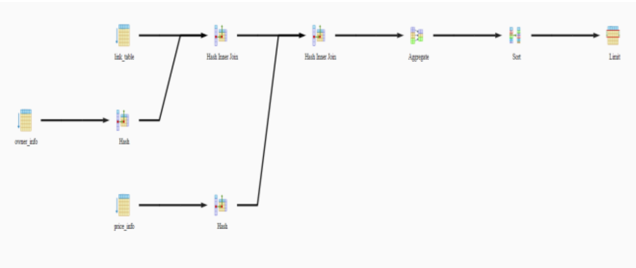
Data Output | Messages | Explain | Notifications

	deed_id integer	deed_book numeric	deed_page numeric	deed_date date	roll character varying (255)	census_id integer	census_tract character varying (255)
1	25000	11287.0	1473.0	2015-10-26	1	25000	167001.0
2	24999	11345.0	1738.0	2019-04-11	1	24999	23004.0
3	24998	11340.0	638.0	2019-01-18	1	24998	23001.0
4	24997	11190.0	8016.0	2010-10-27	1	24997	164004.0
5	24996	11077.0	7115.0	2004-06-16	1	24996	25022.0
6	24995	11248.0	9163.0	2013-06-28	1	24995	166001.0
7	24994	11094.0	1001.0	2005-04-21	1	24994	10003.0
8	24993	11132.0	7857.0	2007-08-03	1	24993	23004.0
9	24992	11276.0	8714.0	2015-03-13	1	24992	167002.0
10	24991	11239.0	3271.0	2012-05-18	1	24991	28002.0
11	24990	10929.0	2139.0	1998-03-18	1	24990	164003.0

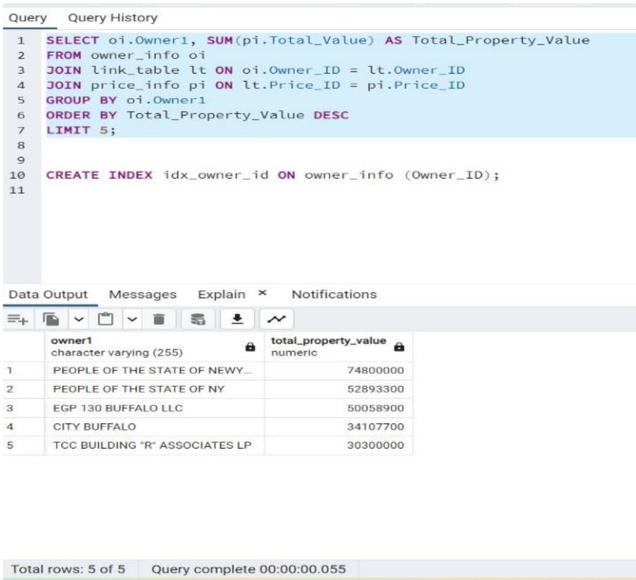
Total rows: 1000 of 24230 Query complete 00:00:00.081

The optimization of our SQL query involved creating several indexes to enhance the performance of join operations and ordering. Specifically, indexes were added to the link_table on columns like Owner_ID, Deed_ID, Price_ID, and Census_ID, and to the census_info and deed_info tables on their respective ID columns. An additional index was also created specifically for the Deed_ID in descending order to optimize the sorting operation dictated by the ORDER BY clause in query. These indexes aim to reduce the time required for the database to locate and retrieve data during join and sort operations, effectively reducing the overall execution time from 0.118 seconds to 0.081 seconds. This improvement demonstrates the significant impact that well-considered indexing can have on the performance of database queries, particularly those involving large datasets and multiple joins.

Query -8 Optimization:



The execution plan for Query 8 before optimization shows a series of hash operations followed by hash inner joins involving the tables owner_info, link_table, and price_info. The process starts with hashing the owner_info and link_table and joining them based on Owner_ID. Next, it joins price_info using Price_ID, resulting in a combined dataset. This dataset is then aggregated to compute the total property value per owner and finally sorted and limited to the top 5 results.



Query Query History

```
SELECT oi.Owner1, SUM(pi.Total_Value) AS Total_Property_Value
FROM owner_info oi
JOIN link_table lt ON oi.Owner_ID = lt.Owner_ID
JOIN price_info pi ON lt.Price_ID = pi.Price_ID
GROUP BY oi.Owner1
ORDER BY Total_Property_Value DESC
LIMIT 5;


CREATE INDEX idx_owner_id ON owner_info (Owner_ID);
```

Data Output Messages Explain Notifications

	owner1 character varying (255)	total_property_value numeric
1	PEOPLE OF THE STATE OF NEWY...	74800000
2	PEOPLE OF THE STATE OF NY	52893300
3	EGP 130 BUFFALO LLC	50058900
4	CITY BUFFALO	34107700
5	TCC BUILDING "R" ASSOCIATES LP	30300000

Total rows: 5 of 5 Query complete 00:00:00.055

The optimization of this query involved creating an index on the Owner_ID column in the owner_info table. This index likely helped in speeding up the join operation between owner_info and link_table by providing quicker access to the rows needed for the join, hence improving the efficiency of the hash operation. By optimizing the join condition, the database engine was able to reduce the time spent scanning and hashing the data, which directly contributed to the decrease in execution time from 0.079 seconds before optimization to 0.055 seconds after optimization. The indexed approach allows the database to more rapidly match rows based on Owner_ID, reducing overall processing time and improving query performance, especially in scenarios where the data set size is large or the query is run frequently.

XII. CONTRIBUTION

Name	Contribution %
P S L Sahithi	50
Mahitha Balumuri	50

XIII. REFERENCES

1. https://www.w3schools.com/sql/sql_join.asp
2. https://docs.oracle.com/cd/E12151_01/doc.150/e12155/triggers_proc_mysql.htm#g1049668
3. https://data.buffalony.gov/Government/2020-2021-Assessment-Roll/8h79-5n5h/about_data