Crime Incident Analysis System: A Comprehensive Database Framework for Chicago

Prathibha Vuyyala
Engineering Science - Data Science
University at Buffalo
Buffalo, US
pvuyyala@buffalo.edu

Tharun Teja Mogili
Engineering Science - Data Science
University at Buffalo
Buffalo, US
tharunte@buffalo.edu

***Abstract*—Efficient crime analysis is vital for enhancing public safety and supporting data-driven decision-making by law enforcement agencies. This project introduces a robust database framework tailored for analyzing crime incidents in Chicago. The proposed system addresses limitations of traditional record-keeping methods by leveraging a relational database structure to ensure data integrity, scalability, and advanced analytical capabilities. Designed using an Entity-Relationship (E/R) model and implemented with SQL, the framework supports complex queries and efficient data retrieval for identifying crime trends, patterns, and correlations. Real-world crime datasets were acquired, processed, and integrated to validate the schema and demonstrate the database's ability to handle diverse use cases. By incorporating normalization principles and optimizing query performance, the system offers a scalable solution adaptable to the dynamic needs of law enforcement agencies. This framework highlights the potential of database systems to streamline crime analysis and improve resource allocation strategies, ultimately contributing to safer communities.**

I. Introduction

In an increasingly urbanized world, efficient crime data management and analysis are critical for maintaining public safety and enabling data-driven decision-making. This project focuses on developing a robust database framework for analyzing crime incidents in Chicago, addressing the limitations of conventional methods such as spreadsheets and paper-based systems. By utilizing a relational database design, the proposed system ensures enhanced data integrity, scalability, and analytical efficiency. Key features of the framework include a well-defined schema, support for complex queries, and the ability to identify crime patterns, correlations, and trends effectively.

The system incorporates real-world datasets sourced from the Chicago Data Portal, which were processed and normalized to support the advanced querying capabilities. This initiative underscores the importance of database systems in transforming raw crime data into actionable insights, thereby equipping law enforcement agencies with the tools required for informed resource allocation and effective crime prevention strategies.

II. Problem Statement

A. Problem Statement and Why Database Instead of an Excel File

The proposed database system aims to address the challenges associated with tracking and analyzing crime data in an urban environment, specifically focusing on Chicago for this project. Currently, many law enforcement agencies rely on spreadsheets or paper-based systems to record incidents, leading to several significant issues:

- **Data Integrity:** Manual entry methods increase the likelihood of errors, resulting in incorrect or incomplete data. This can lead to significant consequences, such as misidentifying crime patterns or failing to respond effectively to emerging crime trends. Inconsistent data can undermine the reliability of the information used by law enforcement agencies to make critical decisions.
- **Data Retrieval:** Searching through large Excel files or paper records for specific incidents can be time-consuming and inefficient. As the volume of crime reports grows, the inability to quickly locate relevant information can delay responses to ongoing incidents and hinder investigative processes. Officers and analysts often spend excessive time sifting through records instead of focusing on proactive measures to address crime.
- **Analysis Capabilities:** Excel lacks robust tools for complex data analysis, making it challenging to identify trends, correlations, and patterns in crime data. Without sophisticated analytical capabilities, law enforcement agencies may struggle to understand the underlying causes of crime, assess the effectiveness of policing strategies, or predict future criminal activity. The inability to perform in-depth analyses limits agencies' capacity to develop data-driven approaches to crime prevention
- **Collaboration Issues:** Multiple stakeholders, including police officers, detectives, crime analysts, and city officials, may need access to the data simultaneously. Traditional file-sharing methods, such as emailing spreadsheets or physical reports, can be cumbersome and lead to version control issues. This lack of collaboration can hinder effective communication among departments and impede timely decision-making.
- **Scalability and Adaptability:** As cities grow and crime dynamics evolve, existing systems often fail to scale efficiently. Relying on spreadsheets or paper-based methods makes it difficult to adapt to new challenges, such as emerging crime types or shifts in community demograph-

ics. A database system can provide the flexibility needed to incorporate new data sources and adapt to changing requirements.

B. Significance of the Problem and Project Contribution

Efficient crime analysis is pivotal for enhancing public safety and enabling strategic resource allocation by law enforcement agencies. Traditional methods, such as paper-based records or spreadsheet systems, suffer from limitations like data inconsistency, inefficiency in retrieval, and lack of advanced analytical capabilities. These challenges hinder the ability to identify crime trends, analyze patterns, and develop effective prevention strategies.

This project contributes to addressing these issues by designing a structured database framework tailored for crime incident analysis in Chicago. The system ensures data integrity, supports complex queries, and facilitates real-time analysis, making it a scalable and adaptable tool for diverse law enforcement needs. By leveraging a robust relational schema and integrating real-world datasets, the project enhances the ability to uncover meaningful insights, thereby empowering stakeholders to make informed decisions. This contribution underscores the transformative potential of database systems in improving the effectiveness and efficiency of crime analysis processes.

III. Target User

The primary users of the proposed database system will be law enforcement agencies, including police departments, crime analysts, and public safety officials. These users will utilize the system to input, retrieve, and analyze crime data efficiently. The database will serve as a centralized repository for all crime-related incidents, facilitating easier access to critical information.

A. Key User Groups

1) Law Enforcement Agencies:

- **Police Officers:** Utilize the database for reporting and accessing real-time crime data to make informed decisions during patrols and investigations. They can input new incident reports, check for outstanding warrants, and view historical crime trends in specific neighborhoods.
- **Detectives and Investigators:** Rely on the system to track ongoing investigations, analyze crime patterns, and correlate incidents. They will use advanced search capabilities to connect various data points and identify potential suspects or patterns of criminal behavior.

2) Crime Analysts: These professionals will leverage the database to perform in-depth analyses of crime trends and statistics. They will generate reports that inform strategies for crime prevention, resource allocation, and community safety initiatives. Their work will be critical for producing analytical insights that can guide policy-making and operational strategies.

3) Public Safety Officials: City planners and public safety officials will use the data to understand crime distribution across neighborhoods. This information is crucial for developing community programs, enhancing public safety measures, and informing policy decisions related to law enforcement and urban development.

4) Researchers and Academics:: Researchers in criminology and sociology may access anonymized data for academic studies and analyses. This group will help validate the database's utility and contribute to the body of knowledge surrounding crime trends, social behaviors, and law enforcement efficacy.

IV. Real-Life Scenario

Consider a scenario in a major city where the police department faces increasing crime rates. Officers on patrol record incidents using paper reports, which are then manually entered into an Excel spreadsheet by administrative staff. This process is not only time-consuming but also prone to errors, leading to gaps in data and ineffective crime prevention strategies. With the implementation of the Crime Analysis and Reporting System, officers can directly enter incident reports into the database via a user-friendly interface. Crime analysts can quickly access the data to generate real-time reports, identifying crime hotspots and trends over time. Additionally, the database administrator will oversee the system's functionality, ensuring data security, integrity, and accessibility for authorized personnel.

V. Data Acquisition, Design, and Integration

A. Data Source

The primary dataset for this project is the Chicago Crimes Dataset, available through the City of Chicago's Open Data Portal. This dataset, covering crime incidents from 2001 to the present, includes key variables such as crime type, location, date, and law enforcement responses, providing a solid foundation for crime pattern analysis.

The dataset consists of several related tables:

- **District:** Information about police districts, including contact details and social media accounts.
- **FBI Code:** Crime classifications with descriptions and related offenses.
- **Community Area:** Details of Chicago's community areas, including population and geographical side.
- **IUCR:** Crime categories and their descriptions.
- **Ward:** Data on city wards, aldermen, and office contacts.
- **Neighborhood:** Neighborhood names linked to community areas.
- **Crime:** Detailed records of individual crime incidents, including case numbers, locations, and arrest data.

This dataset ensures comprehensive and scalable crime analysis, supporting advanced query execution and trend analysis for real-world applications. It is accessible through the City of Chicago's Open Data Portal: Chicago Crimes Dataset.

B. Schema and E/R Diagram

The database schema, based on an Entity-Relationship (E/R) model, ensures data integrity and supports efficient crime analysis. A detailed view of the schema is shown in the E/R diagram (Fig. 1).

- **Crime:** The `Crime` table records details of individual crime incidents, including attributes such as `case_number`, `date`, and crime type. It links to other tables via foreign keys like `iucr_no`, `district_no`, and `ward_no`.
- **District:** The `District` table stores information about police districts, with `district_no` as the primary key. It contains details such as district names, addresses, and contact information.
- **FBI Code:** The `FBI Code` table classifies crimes using `fbi_code_no` as the primary key. It provides descriptions of various crime types, enabling efficient categorization and reporting.
- **IUCR:** The `IUCR` table stores Illinois Uniform Crime Reporting codes, with `iucr_no` as the primary key. It categorizes crimes into primary and secondary classifications for detailed analysis.
- **Ward:** The `Ward` table holds information about city wards, identified by `ward_no`. It includes data such as alderman names and ward office details.
- **Neighborhood:** The `Neighborhood` table links neighborhoods to community areas via `neighborhood_name` and `community_area_no`. It supports neighborhood-level crime analysis and urban planning.
- **Community Area:** The `Community Area` table provides demographic and geographic details for each area, identified by `community_area_no`, which helps contextualize crime data geographically.

This schema, as illustrated in the E/R diagram (Fig. 1), facilitates comprehensive crime data analysis by linking crime incidents to key geographic, political, and classification entities.

C. Data Loading and Transformation

1) Transformation Process: : The data transformation process was conducted to ensure the dataset conformed to the schema defined in the E/R diagram and was ready for efficient loading into the database. This involved converting the `ward_no` column to a nullable integer type to handle missing values appropriately. For community area data, commas were removed from population values, and the data was converted to integers for consistency. Missing values in the ward dataset were addressed using domain knowledge, including filling default values for zip codes, addresses, and name suffixes where applicable. Additionally, uneven spaces in zip code entries were trimmed to maintain uniformity. These preprocessing steps ensured the dataset's integrity and compatibility with the designed database structure.

2) Loading Process: The data loading process involved populating the database tables with cleaned datasets using SQL `COPY` commands. Each table, including `district`, `fbi_code`, `community_area`, `iucr`, `ward`, `neighborhood`, and `crime`, was loaded from corresponding preprocessed CSV files. The files were structured to match the schema defined in the database, ensuring seamless integration. The `COPY` commands efficiently imported the data from local directories, with the `CSV HEADER` option used to incorporate column headers. The process ensured that the relationships defined by foreign key constraints were preserved, maintaining data consistency across tables. This method facilitated the efficient initialization of the database, enabling subsequent analysis and query execution.

3) Challenges and Solutions: During the data loading and transformation process, several challenges were encountered, primarily related to data inconsistencies and missing values. One major issue was handling null values in critical columns, such as `ward_no` and `latitude/longitude`, which required converting columns to nullable types and filling in missing information using domain knowledge. Another challenge was cleaning and standardizing the data, particularly in the `population` and `zip_code` fields, where inconsistent formatting (e.g., commas in numeric values) was present. To address these issues, scripts were written to clean and transform the data, ensuring consistency and compatibility with the database schema. By implementing these solutions, the integrity and reliability of the data were ensured, enabling smooth database operation and analysis.

VI. Database Schema Refinement

All tables within the schema are in Boyce-Codd Normal Form (BCNF), adhering to the identified functional dependencies. The schema has been carefully designed to ensure that each table is in its optimal form, and as such, there is no need for further decomposition. The detailed schema, including the functional dependencies and BCNF checks, is presented in Table 1 on the following page.

A. Transformations from Milestone 1

1) **Normalization:** We followed Database Design Rules to ensure that each table adhered to Boyce-Codd Normal Form (BCNF). This process minimizes redundancy and prevents update anomalies. All tables in the schema, including the `Crime`, `District`, `FBI Code`, and others, have been normalized to ensure efficient data management and integrity.

2) **Attribute Type Standardization:** Certain fields were refined for more efficient storage and querying. For instance, the `latitude` and `longitude` columns in the `Crime` table were defined with the `FLOAT` type, while columns such as `ward_no` in the `Crime` table were adjusted to a nullable integer type. Data types like `BOOLEAN` were used for flags, such as `arrest` and `domestic` in the `Crime` table, providing clearer semantics for these binary attributes.

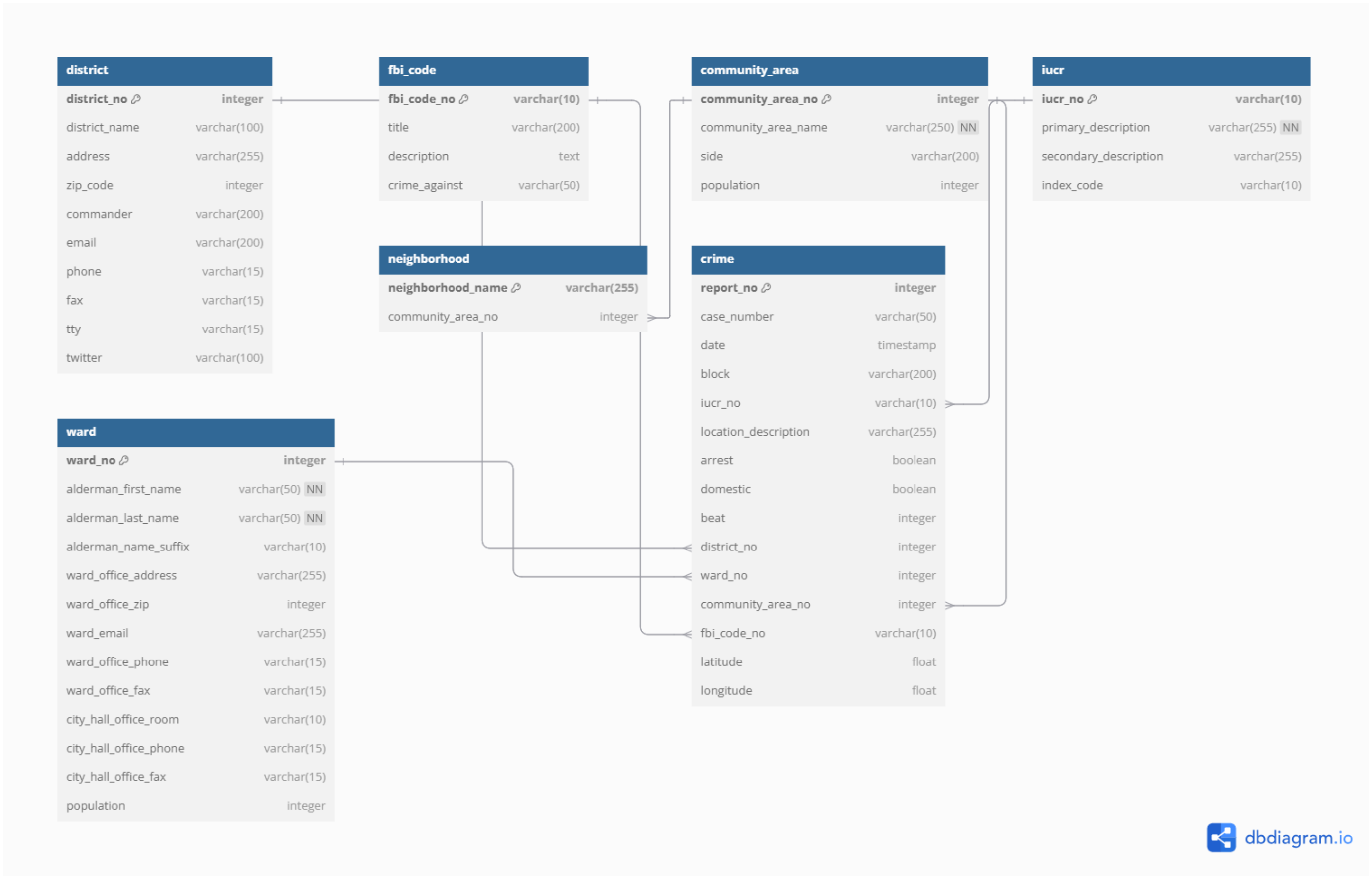


Fig. 1. Entity-Relationship Diagram of the Database Schema

3) **Primary and Foreign Keys:** Establishing relationships between tables is crucial for maintaining data integrity. Primary keys, such as `report_no` in the `Crime` table and `district_no` in the `District` table, uniquely identify each record. Foreign keys, such as `ward_no` in the `Crime` table, link to the `Ward` table, ensuring consistency across related data. These constraints facilitate accurate data management, ensuring that each record is correctly referenced and that no orphaned data exists.

VII. Constraints & Database Validation

A. Constraints

1) PrimaryKeyConstraints:

- **District:** district_no serves as the primary key to uniquely identify each district in the dataset.
- **FBI Code:** fbi_code_no is the unique identifier for each FBI crime classification, ensuring accurate classification of crime types.
- **Community Area:** community_area_no serves as the primary key to uniquely identify each community area in Chicago, linking crime data to specific regions.
- **IUCR:** iucr_no uniquely identifies each crime category, enabling detailed classification of criminal activities.
- **Ward:** ward_no uniquely identifies each ward in the city, linking specific crime incidents to local governance areas.
- **Neighborhood:** neighborhood_name serves as the primary key, uniquely identifying each neighborhood and connecting it to the relevant community area.
- **Crime:** report_no serves as the primary key, uniquely identifying each crime report and linking it to other tables like `District`, `FBI Code`, and `Community Area`.

2) Foreign Key Constraints:

- **Crime and District:** – `district_no` references `district(district_no)`, maintaining valid district associations.
- **Crime and FBI Code:** – `fbi_code_no` references `fbi_code(fbi_code_no)`, ensuring valid crime type associations.
- **Crime and IUCR:** – `iucr_no` references `iucr(iucr_no)`, ensuring valid crime classification.
- **Crime and Ward:** – `ward_no` references `ward(ward_no)`, ensuring valid ward associations.
- **Neighborhood and Community Area:** – `community_area_no` references `community_area(community_area_no)`, maintaining valid community area associations.

3) Check Constraint:

- A Check Constraint was added to the zip_code column in the District table to ensure that only valid five-digit

TABLE I
DATABASE TABLES WITH FUNCTIONAL DEPENDENCIES AND BCNF CHECK

Table Name	**Primary Key**	**Functional Dependencies (FD's)**	**BCNF Check**
District	district_no	district_no → district_name, address, zip_code, commander, email, phone, fax, tty, twitter	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
FBI Code	fbi_code_no	fbi_code_no → title, description, crime_against	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Community Area	community_area_no	community_area_no → community_area_name, side, population	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
IUCR	iucr_no	iucr_no → primary_description, secondary_description, index_code	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Ward	ward_no	ward_no → alderman_first_name, alderman_last_name, alderman_name_suffix, ward_office_address, ward_office_zip, ward_email, ward_office_phone, ward_office_fax, city_hall_office_room, city_hall_office_phone, city_hall_office_fax, population	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Neighborhood	neighborhood_name	neighborhood_name → community_area_no	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Crime	report_no	report_no → case_number, date, block, iucr_no, location_description, arrest, domestic, beat, district_no, ward_no, community_area_no, fbi_code_no, latitude, longitude	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.

zip codes (ranging from 10000 to 99999) are inserted.

- This constraint enhances data integrity by preventing invalid or incorrect zip codes from being stored in the database.
- By enforcing this rule at the database level, we maintain consistency and ensure the accuracy of location-based data in the District table.

4) Unique Constraint:

- A Unique Constraint was added to the `district_name` column in the `District` table to ensure that each district name is unique across all records, preventing duplicate entries.
- This constraint enforces data integrity by ensuring that no two districts can have the same name, thus maintaining consistency in the database.

5) On Update Cascade:

- The On Update Cascade constraint ensures that when a referenced value in a parent table is updated, the corresponding values in the child tables are automatically updated to maintain referential integrity.
- In the `neighborhood` and `crime` tables, this constraint is applied to foreign keys such as `community_area_no`, `district_no`, and others, ensuring that updates in the referenced parent tables are propagated to the related child tables.
- This automatic update mechanism helps to maintain data consistency across the database, reducing the risk of orphaned records or mismatched references.

6) On Delete Cascade:

- The On Delete Caascade constraint ensures that when a record in a parent table is deleted, the corresponding records in the child tables are automatically deleted to maintain referential integrity.
- In the `neighborhood` and `crime` tables, this constraint is applied to foreign keys such as `community_area_no`, `district_no`, and others, ensuring that deleting a referenced record automatically removes related records in the child tables.
- This mechanism helps to prevent orphaned records and ensures the database remains clean by removing related data when a parent record is deleted.

B. Validations

- A subset of the real crime dataset was reviewed to ensure that the data types and values align with the schema, confirming consistency between the data and the database design.

- Foreign key relationships were validated, ensuring that all district_no values in the crime table exist in the district table, maintaining referential integrity.
- The schema was checked for compliance with Boyce-Codd Normal Form (BCNF), ensuring that non-prime attributes depend on the entire primary key, and that no violations of normalization rules were found.
- Redundant data was reviewed in the dataset, specifically between the district and neighborhood tables, and no redundancy issues were identified.
- Data completeness was validated by checking for any missing or null values in critical fields like district_no and ward_no in the crime table, and no such issues were found.
- Duplicate records were checked in the crime table, and no unnecessary data duplication was observed.
- The ON UPDATE CASCADE and ON DELETE CASCADE functionality were reviewed to confirm that updates and deletions in parent tables (e.g., district) would propagate correctly to child tables (e.g., crime), ensuring data integrity.

VIII. Handling Large Datasets

Initially, the database was validated using a small dataset to ensure the schema, constraints, and relationships were functioning correctly. All basic operations, including inserts, updates, and deletes, were successfully executed without any issues. However, as the dataset size increased significantly, a noticeable slowness in query execution, particularly with select queries, was observed.

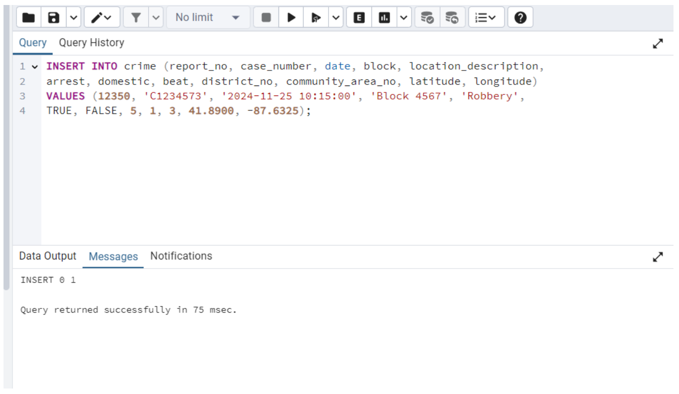
To address this, several insert, delete, update, and select queries were designed and executed to debug the performance bottlenecks. These select queries utilized advanced SQL operations such as joins, grouping, ordering, and subqueries to simulate real-world scenarios. This process helped identify problematic queries and evaluate their performance under larger datasets.

To overcome the slowness, indexing strategies were adopted for critical columns such as primary keys and frequently queried foreign keys. The use of the PostgreSQL EXPLAIN tool provided detailed execution plans, which were analyzed to pinpoint inefficiencies and guide optimization efforts. These adjustments significantly improved query performance, ensuring scalability and efficiency for larger datasets.

A. Testing Database Using SQL Queries

1) Insert Queries Execution: We executed two INSERT queries to add sample data into the crime and district tables. The first query inserted a crime record into the crime table, while the second added a ward record into the ward table. Both queries were executed successfully without any slowness issues. The data was correctly inserted into the tables, and no errors were encountered during the process.

The following snapshots display the successful execution of the INSERT queries. These queries were executed without any issues.

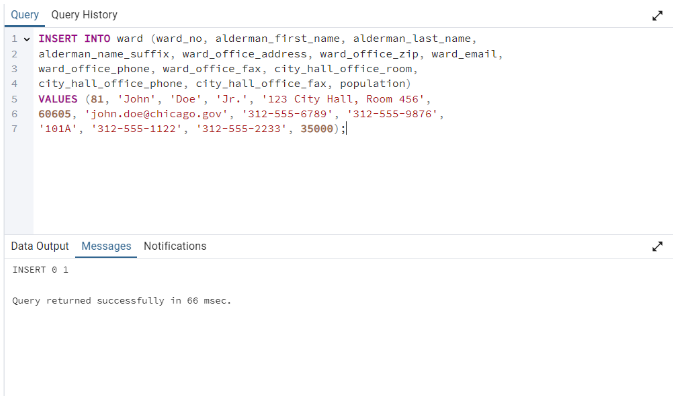


```
INSERT INTO crime (report_no, case_number, date, block, location_description,
arrest, domestic, beat, district_no, community_area_no, latitude, longitude)
VALUES (12350, 'C1234573', '2024-11-25 10:15:00', 'Block 4567', 'Robbery',
TRUE, FALSE, 5, 1, 3, 41.8900, -87.6325);
```

```
INSERT 0 1

Query returned successfully in 75 msec.
```

Fig. 2. Insert Query for Crime Table Execution



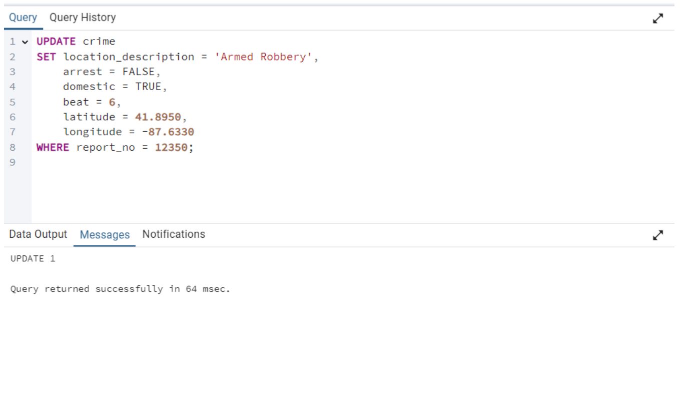
```
INSERT INTO ward (ward_no, alderman_first_name, alderman_last_name,
alderman_name_suffix, ward_office_address, ward_office_zip, ward_email,
ward_office_phone, ward_office_fax, city_hall_office_room,
city_hall_office_phone, city_hall_office_fax, population)
VALUES (81, 'John', 'Doe', 'Jr.', '123 City Hall, Room 456',
60605, 'john.doe@chicago.gov', '312-555-6789', '312-555-9876',
'101A', '312-555-1122', '312-555-2233', 35000);
```

```
INSERT 0 1

Query returned successfully in 66 msec.
```

Fig. 3. Insert Query for Ward Table Execution

2) Update Queries Execution: We executed two UPDATE queries to modify records in the crime and ward tables. The first query updated the location_description, arrest, and beat fields for the crime record with report_no 12350 in the crime table. The second query updated the population and ward_office_phone fields for the ward with ward_no 81 in the ward table. Both queries were executed successfully without encountering any issues, including slowness, during the process. The updates were applied correctly and the data in the respective tables was modified as expected.



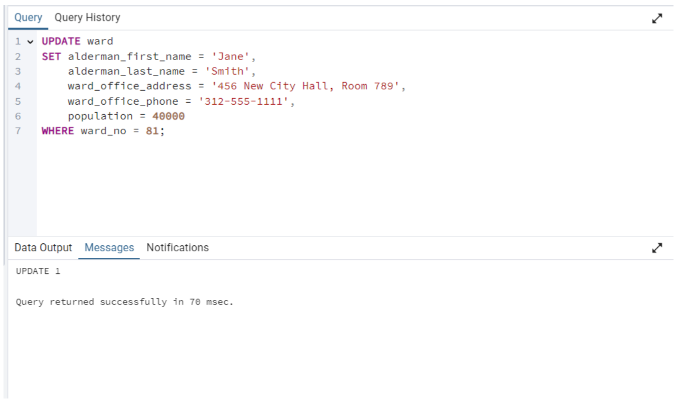
```
UPDATE crime
SET location_description = 'Armed Robbery',
    arrest = FALSE,
    domestic = TRUE,
    beat = 6,
    latitude = 41.8950,
    longitude = -87.6330
WHERE report_no = 12350;

```

```
UPDATE 1

Query returned successfully in 64 msec.
```

Fig. 4. Update Query for Crime Table Execution



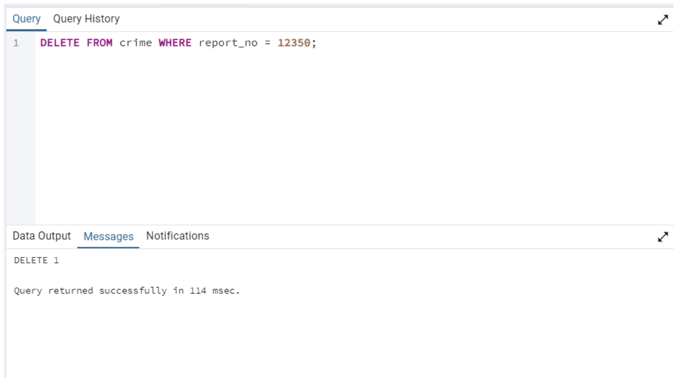
```sql
UPDATE ward
SET alderman_first_name = 'Jane',
    alderman_last_name = 'Smith',
    ward_office_address = '456 New City Hall, Room 789',
    ward_office_phone = '312-555-1111',
    population = 40000
WHERE ward_no = 81;
```

```
UPDATE 1

Query returned successfully in 70 msec.
```

Fig. 5. Update Query for Ward Table Execution

3) Delete Queries Execution: We executed two DELETE queries to remove specific records from the crime and ward tables. The first query deleted a crime record with report_no 12350 from the crime table, and the second query deleted a ward record with ward_no 81 from the ward table. Both DELETE operations were executed successfully without encountering any performance issues or slowness. The records were efficiently removed from the respective tables, ensuring the integrity of the data.

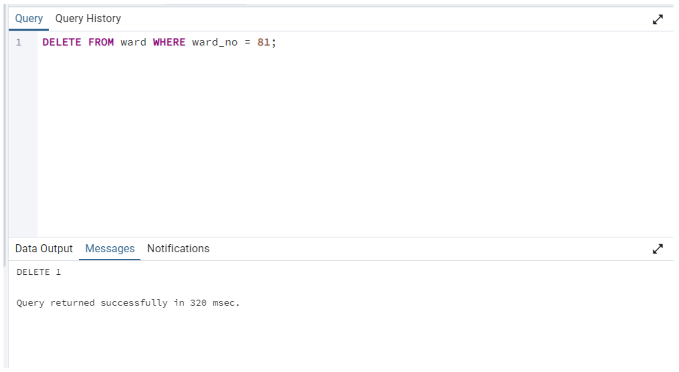


```sql
DELETE FROM crime WHERE report_no = 12350;
```

```
DELETE 1

Query returned successfully in 114 msec.
```

Fig. 6. Delete Query for Crime Table Execution



```sql
DELETE FROM ward WHERE ward_no = 81;
```

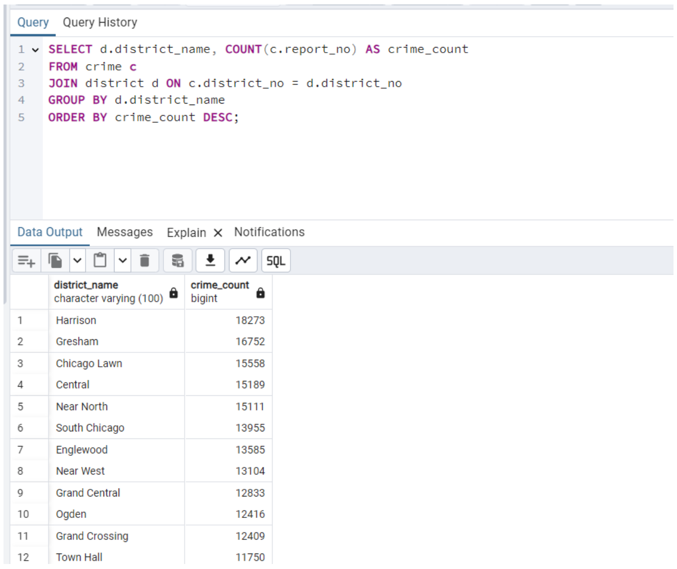
```
DELETE 1

Query returned successfully in 320 msec.
```

Fig. 7. Delete Query for Ward Table Execution

4) Select Queries Execution:

a) Crime Count by District: The following SQL query was executed to analyze crime distribution across districts:



```sql
SELECT d.district_name, COUNT(c.report_no) AS crime_count
FROM crime c
JOIN district d ON c.district_no = d.district_no
GROUP BY d.district_name
ORDER BY crime_count DESC;
```

	district_name character varying (100)	crime_count bigint
1	Harrison	18273
2	Gresham	16752
3	Chicago Lawn	15558
4	Central	15189
5	Near North	15111
6	South Chicago	13955
7	Englewood	13585
8	Near West	13104
9	Grand Central	12833
10	Ogden	12416
11	Grand Crossing	12409
12	Town Hall	11750

Fig. 8. SQL Query to Count Crimes by District

The query involves a join between the `crime` and `district` tables, aggregation to count crime incidents, and sorting by `crime_count`. The execution plan highlighted the following issues:

1) **Sequential Scans**: The query performs sequential scans on both the `crime` and `district` tables, increasing execution time for large datasets.
2) **Hash Join Overhead**: While efficient for moderate datasets, the hash join consumes significant memory for large tables.
3) **Costly Aggregation and Sorting**: Grouping and sorting operations add computational complexity.
4) **Lack of Indexing**: The absence of indexes on `district_no` exacerbates the performance bottleneck.

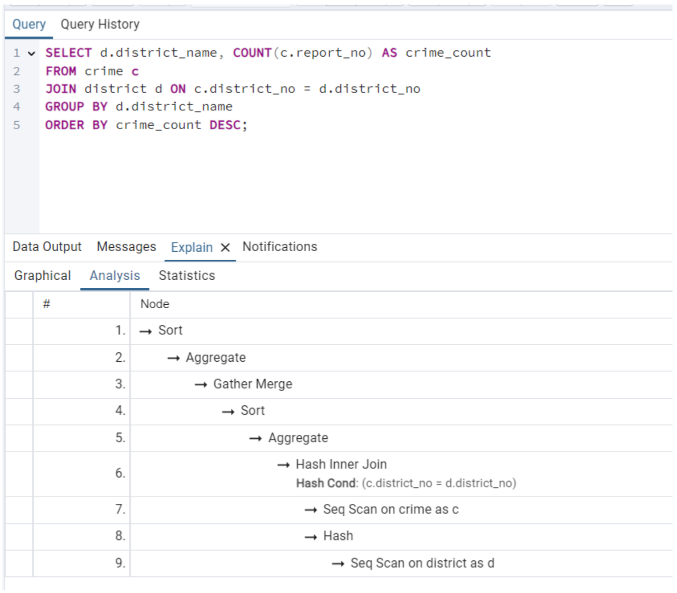
Optimization Recommendations: To address these challenges, the following optimizations are recommended:

1) **Indexing**: Create indexes on the `district_no` column in both tables to speed up join operations.

b) Crime Details by Date and Type: The following SQL query was executed to retrieve crime reports along with their corresponding FBI crime classification for a specified date range:

This query joins the `crime` and `fbi_code` tables using the `fbi_code_no` key, filters records based on a date range, and orders the results in descending order by `date`. The execution plan indicated the following performance challenges:

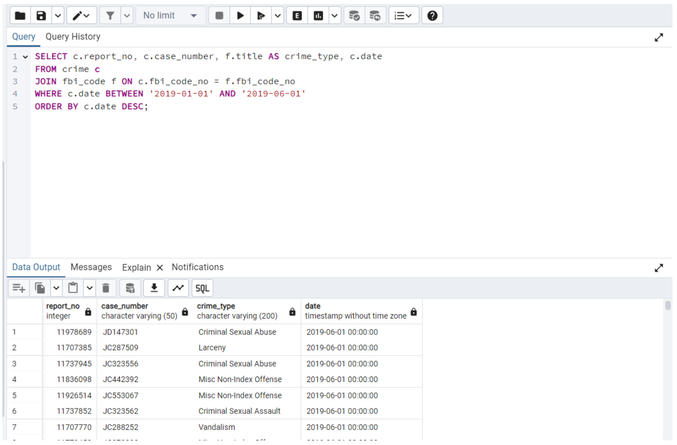
1) **Sequential Scans**: Both the `crime` and `fbi_code` tables are accessed using sequential scans, leading to high I/O costs for large datasets.
2) **Hash Join Overhead**: The hash join used to match `fbi_code_no` across tables adds computational overhead for large datasets.



```sql
SELECT d.district_name, COUNT(c.report_no) AS crime_count
FROM crime c
JOIN district d ON c.district_no = d.district_no
GROUP BY d.district_name
ORDER BY crime_count DESC;
```

```
#    Node
1.   → Sort
2.       → Aggregate
3.           → Gather Merge
4.               → Sort
5.                   → Aggregate
6.                       → Hash Inner Join
                           Hash Cond: (c.district_no = d.district_no)
7.                           → Seq Scan on crime as c
8.                           → Hash
9.                               → Seq Scan on district as d
```

Fig. 9. Execution Plan for Crime Count by District Query



```sql
SELECT c.report_no, c.case_number, f.title AS crime_type, c.date
FROM crime c
JOIN fbi_code f ON c.fbi_code_no = f.fbi_code_no
WHERE c.date BETWEEN '2019-01-01' AND '2019-06-01'
ORDER BY c.date DESC;
```

	report_no integer	case_number character varying (50)	crime_type character varying (200)	date timestamp without time zone
1	11978689	JD147301	Criminal Sexual Abuse	2019-06-01 00:00:00
2	11707385	JC287509	Larceny	2019-06-01 00:00:00
3	11737945	JC323556	Criminal Sexual Abuse	2019-06-01 00:00:00
4	11836098	JC442392	Misc Non-Index Offense	2019-06-01 00:00:00
5	11926514	JC553067	Misc Non-Index Offense	2019-06-01 00:00:00
6	11737852	JC323562	Criminal Sexual Assault	2019-06-01 00:00:00
7	11707770	JC288252	Vandalism	2019-06-01 00:00:00

Fig. 10. SQL Query to Retrieve Crime Details by Date and Type

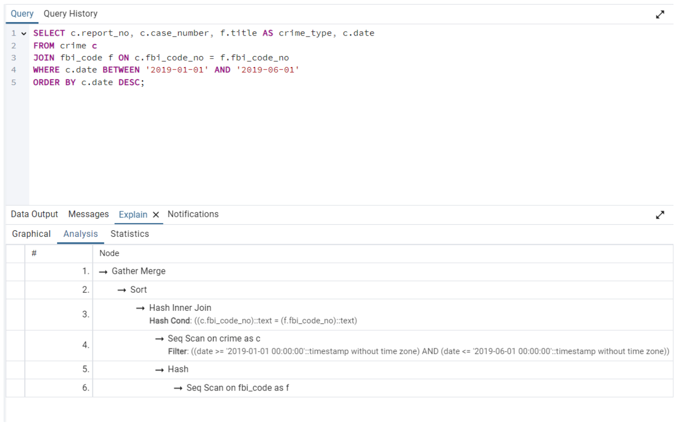
3) **Filtering and Sorting Costs**: The `WHERE` clause and `ORDER BY` operation add significant computational overhead.

Optimization Recommendations: To address these challenges, the following optimizations are recommended:

1) **Indexing**: Create indexes on the `fbi_code_no` and `date` columns in the `crime` table to improve filtering and join performance.

c) Crime Count and Geographic Analysis by District: The following SQL query was executed to analyze the distribution of crimes across districts, including the total crime count and the average latitude of incidents within a specified date range:

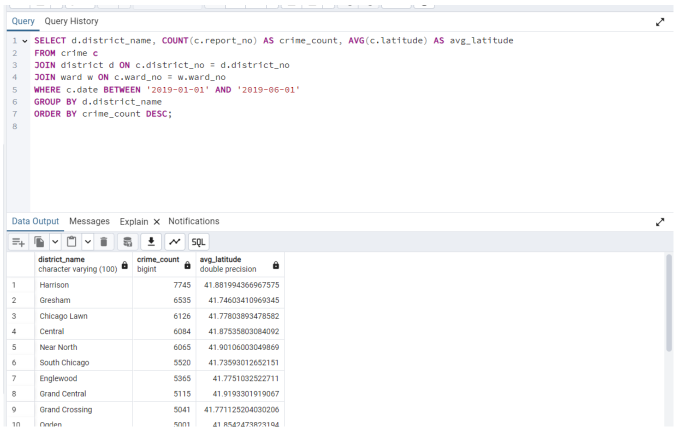
This query performs a multi-table join involving the `crime`, `district`, and `ward` tables. It aggregates the data to compute the crime count and average latitude for each district and orders the results in descending order of `crime_count`.



```sql
SELECT c.report_no, c.case_number, f.title AS crime_type, c.date
FROM crime c
JOIN fbi_code f ON c.fbi_code_no = f.fbi_code_no
WHERE c.date BETWEEN '2019-01-01' AND '2019-06-01'
ORDER BY c.date DESC;
```

```
#    Node
1.   → Gather Merge
2.       → Sort
3.           → Hash Inner Join
             Hash Cond: ((c.fbi_code_no)::text = (f.fbi_code_no)::text)
4.               → Seq Scan on crime as c
                 Filter: ((date >= '2019-01-01 00:00:00'::timestamp without time zone) AND (date <= '2019-06-01 00:00:00'::timestamp without time zone))
5.               → Hash
6.                   → Seq Scan on fbi_code as f
```

Fig. 11. Execution Plan for Crime Details by Date and Type Query



```sql
SELECT d.district_name, COUNT(c.report_no) AS crime_count, AVG(c.latitude) AS avg_latitude
FROM crime c
JOIN district d ON c.district_no = d.district_no
JOIN ward w ON c.ward_no = w.ward_no
WHERE c.date BETWEEN '2019-01-01' AND '2019-06-01'
GROUP BY d.district_name
ORDER BY crime_count DESC;
```

	district_name character varying (100)	crime_count bigint	avg_latitude double precision
1	Harrison	7745	41.881994366967575
2	Gresham	6535	41.74603410969345
3	Chicago Lawn	6126	41.77803893478582
4	Central	6084	41.87535803084092
5	Near North	6065	41.90106003049869
6	South Chicago	5520	41.73593012652151
7	Englewood	5365	41.7751032522711
8	Grand Central	5115	41.9193301919067
9	Grand Crossing	5041	41.771125204030206
10	Ogden	5001	41.8542473823194

Fig. 12. SQL Query for Crime Count and Geographic Analysis by District

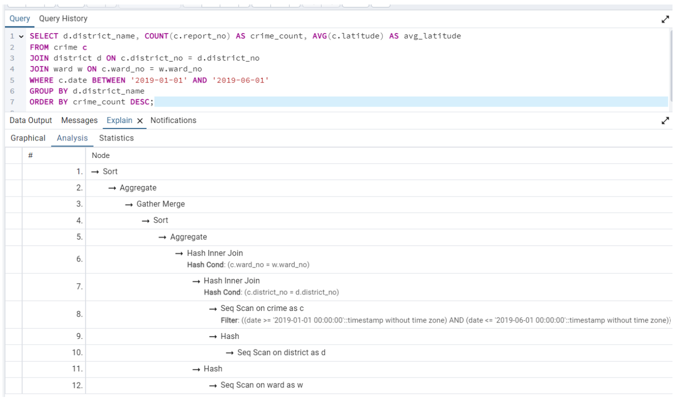
The execution plan revealed the following performance issues:

1) **Sequential Scans**: Sequential scans are performed on the `crime`, `district`, and `ward` tables, leading to high I/O costs.
2) **Multiple Joins**: The inclusion of multiple joins increases computational overhead, particularly for large datasets.
3) **Aggregation Costs**: Calculating the average latitude and grouping by district adds significant computational complexity.
4) **Lack of Indexing**: The absence of indexes on key columns like `ward_no`, `district_no`, and `date` exacerbates the performance bottleneck.

Optimization Recommendations: To address these challenges, the following optimizations are recommended:

1) **Indexing**: Create indexes on the `district_no`, `ward_no`, and `date` columns to speed up join operations and filtering.

d) Top Districts with the Most Crimes: The following SQL query was executed to retrieve crime details for the top three districts with the highest number of crimes. The query uses a subquery to identify the districts with the highest crime counts and then retrieves detailed records for those districts


Query Query History

```sql
SELECT d.district_name, COUNT(c.report_no) AS crime_count, AVG(c.latitude) AS avg_latitude
FROM crime c
JOIN district d ON c.district_no = d.district_no
JOIN ward w ON c.ward_no = w.ward_no
WHERE c.date BETWEEN '2019-01-01' AND '2019-06-01'
GROUP BY d.district_name
ORDER BY crime_count DESC;
```

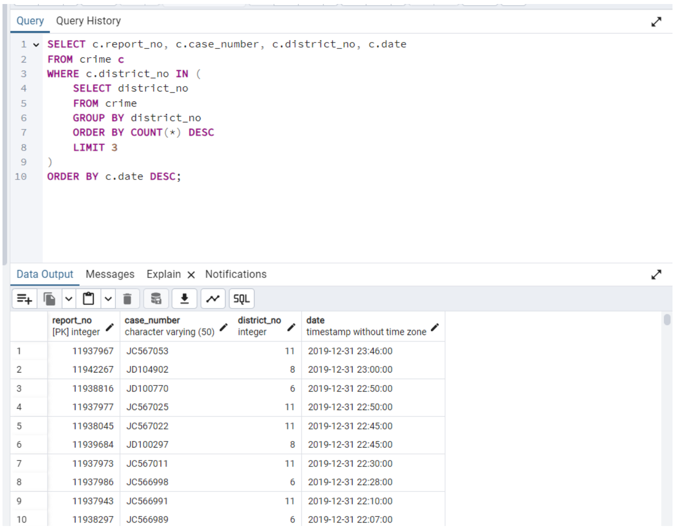
Data Output Messages Explain Notifications

Graphical Analysis Statistics

```
#    Node
1.   → Sort
2.       → Aggregate
3.           → Gather Merge
4.               → Sort
5.                   → Aggregate
6.                       → Hash Inner Join
                         Hash Cond: (c.ward_no = w.ward_no)
7.                           → Hash Inner Join
                             Hash Cond: (c.district_no = d.district_no)
8.                               → Seq Scan on crime as c
                                 Filter: ((date >= '2019-01-01 00:00:00'::timestamp without time zone) AND (date <= '2019-06-01 00:00:00'::timestamp without time zone))
9.                               → Hash
10.                                  → Seq Scan on district as d
11.                          → Hash
12.                              → Seq Scan on ward as w
```

Fig. 13. Execution Plan for Crime Count and Geographic Analysis Query

This query combines an outer query for retrieving details and a subquery to compute the top three districts based on crime count. The final results are ordered by the crime date in descending order.


Query Query History

```sql
SELECT c.report_no, c.case_number, c.district_no, c.date
FROM crime c
WHERE c.district_no IN (
    SELECT district_no
    FROM crime
    GROUP BY district_no
    ORDER BY COUNT(*) DESC
    LIMIT 3
)
ORDER BY c.date DESC;
```

Data Output Messages Explain Notifications

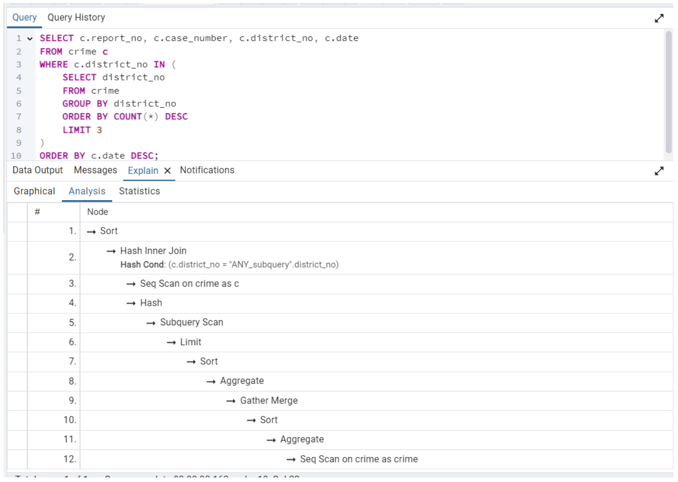
	report_no [PK] integer	case_number character varying (50)	district_no integer	date timestamp without time zone
1	11937967	JC567053	11	2019-12-31 23:46:00
2	11942267	JD104902	8	2019-12-31 23:00:00
3	11938816	JD100770	6	2019-12-31 22:50:00
4	11937977	JC567025	11	2019-12-31 22:50:00
5	11938045	JC567022	11	2019-12-31 22:45:00
6	11939684	JD100297	8	2019-12-31 22:45:00
7	11937973	JC567011	11	2019-12-31 22:30:00
8	11937986	JC566998	6	2019-12-31 22:28:00
9	11937943	JC566991	11	2019-12-31 22:10:00
10	11938297	JC566989	6	2019-12-31 22:07:00

Fig. 14. SQL Query to Retrieve Crime Details for Top Districts

The execution plan revealed the following performance challenges:

1) **Subquery Overhead**: The subquery performs aggregation and sorting operations, which can be computationally expensive for large datasets.
2) **Sequential Scans**: The query involves sequential scans on the `crime` table for both the subquery and the outer query.
3) **Hash Join Costs**: The outer query joins results from the subquery using a hash join, which adds additional computational overhead.
4) **Sorting Overhead**: Sorting the subquery results by `COUNT(*)` and the outer query results by `date` further increases execution time.

Optimization Recommendations To address these performance challenges, the following optimizations are recommended:


Query Query History

```sql
SELECT c.report_no, c.case_number, c.district_no, c.date
FROM crime c
WHERE c.district_no IN (
    SELECT district_no
    FROM crime
    GROUP BY district_no
    ORDER BY COUNT(*) DESC
    LIMIT 3
)
ORDER BY c.date DESC;
```

Data Output Messages Explain Notifications

Graphical Analysis Statistics

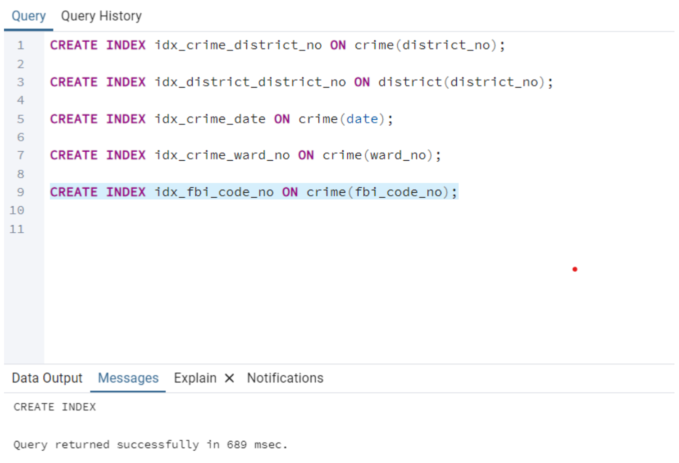
```
#    Node
1.   → Sort
2.       → Hash Inner Join
         Hash Cond: (c.district_no = "ANY_subquery".district_no)
3.           → Seq Scan on crime as c
4.           → Hash
5.               → Subquery Scan
6.                   → Limit
7.                       → Sort
8.                           → Aggregate
9.                               → Gather Merge
10.                                  → Sort
11.                                      → Aggregate
12.                                          → Seq Scan on crime as crime
```

Fig. 15. Execution Plan for Query on Top Districts with the Most Crimes

1) **Indexing**: Create an index on the `district_no` column to improve the efficiency of the subquery and hash join operations.

B. Impact of Indexing on Query Performance

To address the performance bottlenecks identified in the earlier execution plans, indexing was implemented on critical columns across the crime and district tables. The following indexes were created:


Query Query History

```sql
CREATE INDEX idx_crime_district_no ON crime(district_no);

CREATE INDEX idx_district_district_no ON district(district_no);

CREATE INDEX idx_crime_date ON crime(date);

CREATE INDEX idx_crime_ward_no ON crime(ward_no);

CREATE INDEX idx_fbi_code_no ON crime(fbi_code_no);
```

Data Output Messages Explain Notifications

```
CREATE INDEX

Query returned successfully in 689 msec.
```

Fig. 16. Created Indexes

- `idx_crime_district_no`: Index on the `district_no` column in the `crime` table.
- `idx_district_district_no`: Index on the `district_no` column in the `district` table.
- `idx_crime_date`: Index on the `date` column in the `crime` table.

- `idx_crime_ward_no`: Index on the `ward_no` column in the `crime` table.
- `idx_fbi_code_no`: Index on the `fbi_code_no` column in the `crime` table.

These indexes were created to improve the efficiency of join operations, filtering conditions, and sorting operations in the queries.

1) Improved Query Performance: After implementing the indexes, the previously problematic queries were re-executed, and the results showed significant improvements:

- Faster Retrieval: The retrieval of records, especially for queries involving joins and filtering conditions on the indexed columns, was considerably faster.
- Reduced Execution Time: The overall execution time for queries decreased substantially. For instance, queries involving joins on `district_no` and filtering on `date` now completed within milliseconds.
- Optimized Query Plans: The PostgreSQL query planner utilized the newly created indexes effectively, reducing the reliance on sequential scans and hash joins. This is evident in the updated execution plans, which now include index scans for relevant operations.

2) Observation and Conclusion: The implementation of indexes successfully addressed the performance issues observed in earlier executions. Queries that previously relied on computationally expensive operations now leverage efficient index-based retrieval, making the database more responsive and capable of handling larger datasets. The performance gains demonstrate the critical role of indexing in optimizing database queries for real-world applications.

IX. CONCLUSION

A. Project Achievements

This project successfully transformed raw crime datasets from Chicago into a robust relational database, adhering to Boyce-Codd Normal Form (BCNF). The normalization process minimized data redundancy and ensured efficient query performance while maintaining data integrity. Key database tables include Crime, District, Ward, FBI Code, IUCR, Neighborhood, and Community Area, each linked by well-defined primary and foreign key relationships.

The relationships were reinforced with cascading updates and deletions, ensuring referential integrity across the database. Advanced SQL queries were designed and executed to extract meaningful insights, such as identifying crime patterns, analyzing geographical distributions, and studying trends over time. Performance improvements were achieved by indexing key columns like district_no, fbi_code_no, and date, enabling the system to handle large datasets efficiently. This database provides a solid foundation for crime data analysis and decision-making support.

B. Key Insights Gained

This project offered critical insights into database design and management, highlighting the importance of the following:

- **Normalization**: The normalization of tables to BCNF eliminated redundancy and improved query reliability while avoiding anomalies during updates and deletions.
- **Indexing and Query Optimization**: Indexing frequently accessed columns and analyzing execution plans reduced query execution time, enhancing database responsiveness.
- **Relationship Modeling**: Establishing many-to-many relationships through linking tables provided an accurate and scalable representation of complex associations.
- **Constraint Implementation**: Enforcing primary and foreign key constraints ensured valid and consistent references, while cascading updates and deletions maintained data integrity across related tables.
- **Performance Monitoring**: The use of PostgreSQL tools like `EXPLAIN` to analyze query execution plans demonstrated the importance of query tuning in handling large datasets efficiently.

C. Recommendations for Future Enhancements

To further improve the system's capabilities and expand its applications, the following enhancements are recommended:

- **Interactive Analytics Platform**: Develop an interactive dashboard with visualizations such as heatmaps, charts, and temporal trend analyses to provide end-users with intuitive access to crime data.
- **Predictive Analysis Capabilities**: Implement machine learning models for predictive analytics, such as forecasting crime hotspots and identifying emerging patterns for proactive decision-making.
- **Role-Based Access Control (RBAC)**: Introduce secure, role-based access controls to protect sensitive crime data while allowing authorized users to access relevant information.
- **API Development**: Provide API support to enable seamless integration with external systems and applications, fostering collaboration and expanding the database's utility.
- **Geospatial Analysis**: Extend the system to include advanced geospatial analysis features, enabling detailed mapping and analysis of crime locations and clusters.

X. BONUS TASK: WEB APPLICATION FOR DATABASE VISUALIZATION

As an additional enhancement to the project, a web-based application was developed to interface with the Chicago Crime Database. This application provides an interactive platform for querying and visualizing data directly from the database, enabling users to explore crime data dynamically. The key features and capabilities of the web application are outlined below.

A. Key Features of the Web Application

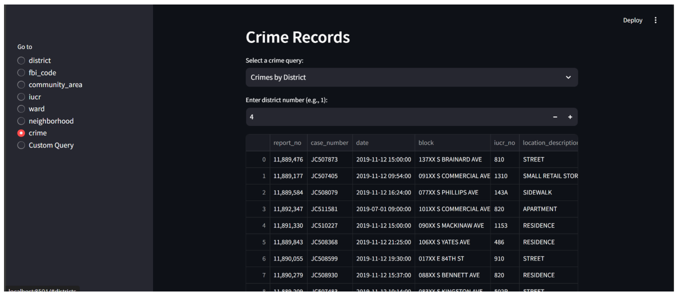
- **Interactive Query Interface:**
  - Users can navigate through different database tables such as `District`, `FBI Code`, `Community Area`, `IUCR`, `Ward`, `Neighborhood`, and `Crime`.

    - Predefined queries allow users to retrieve specific insights, such as crimes by district, FBI codes by crime type, or community areas by population.
- **Custom Query Support:**
    - The application provides a ”Custom Query” feature, enabling users to execute ad-hoc SQL queries and visualize the results dynamically.
- **Advanced Querying:**
    - The app supports advanced SQL operations, including joins, grouping, and filtering, to retrieve detailed insights from the database.
- **Streamlit Framework:**
    - The application was built using Streamlit, a Python-based framework for creating interactive web applications.
    - Integration with PostgreSQL was achieved through the SQLAlchemy library, ensuring seamless connectivity with the database.
- **User-Friendly Design:**
    - A simple and intuitive interface allows users to interact with the database effortlessly.
    - A visually appealing background enhances the user experience, making the application more engaging.
- **Scalable and Responsive:**
    - The application is designed to handle large datasets efficiently, leveraging the optimized database schema and indexing strategies implemented during the project.

B. Conclusion

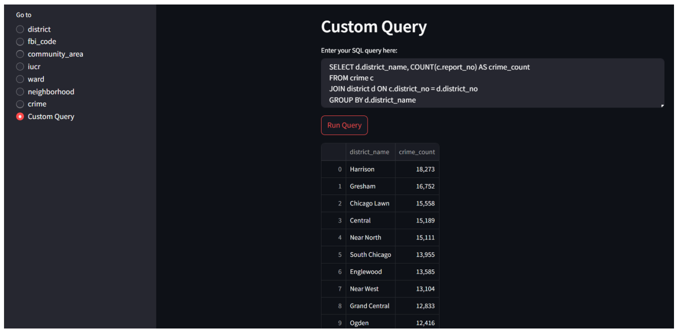
The development of this web application highlights the versatility and practicality of the Chicago Crime Database. By bridging the gap between data storage and user interaction, it enables stakeholders to analyze, visualize, and derive meaningful insights from the data in a user-friendly environment. This enhancement significantly extends the project's applicability and usability, showcasing the potential for real-world deployment.

C. Snapshots of the Web Application



	report_no	case_number	date	block	iucr_no	location_descriptio
0	11,889,476	JC507873	2019-11-12 15:00:00	137XX S BRAINARD AVE	810	STREET
1	11,889,177	JC507405	2019-11-12 09:54:00	091XX S COMMERCIAL AVE	1310	SMALL RETAIL STOR
2	11,889,584	JC508079	2019-11-12 16:24:00	077XX S PHILLIPS AVE	143A	SIDEWALK
3	11,892,347	JC511581	2019-07-01 09:00:00	101XX S COMMERCIAL AVE	820	APARTMENT
4	11,891,330	JC510227	2019-11-12 15:00:00	090XX S MACKINAW AVE	1153	RESIDENCE
5	11,889,843	JC508368	2019-11-12 21:25:00	106XX S YATES AVE	486	RESIDENCE
6	11,890,055	JC508599	2019-11-12 19:30:00	017XX E 84TH ST	910	STREET
7	11,890,279	JC508930	2019-11-12 15:37:00	088XX S BENNETT AVE	820	RESIDENCE
8	11,889,209	JC507483	2019-11-12 10:14:00	083XX S KINGSTON AVE	502R	STREET

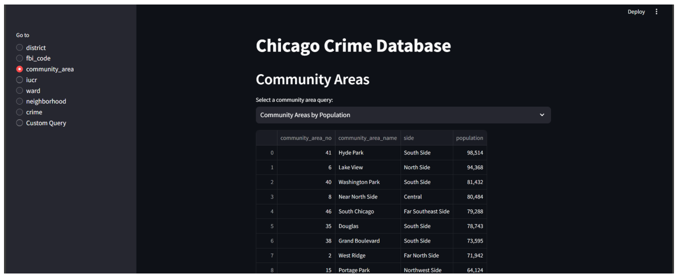
Fig. 17. Homepage of the Web Application



```
SELECT d.district_name, COUNT(c.report_no) AS crime_count
FROM crime c
JOIN district d ON c.district_no = d.district_no
GROUP BY d.district_name
```

	district_name	crime_count
0	Harrison	18,273
1	Gresham	16,752
2	Chicago Lawn	15,558
3	Central	15,189
4	Near North	15,111
5	South Chicago	13,955
6	Englewood	13,585
7	Near West	13,104
8	Grand Central	12,833
9	Ogden	12,416

Fig. 18. Custom Query Execution Interface



	community_area_no	community_area_name	side	population
0	41	Hyde Park	South Side	98,514
1	6	Lake View	North Side	94,368
2	40	Washington Park	South Side	81,432
3	8	Near North Side	Central	80,484
4	46	South Chicago	Far Southeast Side	79,288
5	35	Douglas	South Side	78,743
6	38	Grand Boulevard	South Side	73,595
7	2	West Ridge	Far North Side	71,942
8	15	Portage Park	Northwest Side	64,124

Fig. 19. Query Results Display

XI. Team Contributions

A. Prathibha Vuyyala (pvuyyala)

- Designed and implemented the database schema for the Chicago Crime Database, including tables for crime, district, community_area, fbi_code, iucr, and ward.
- Worked on defining the foreign key relationships and constraints to ensure data integrity and normalization (BCNF).
- Developed SQL queries for data retrieval, including complex joins and subqueries for analysis.
- Implemented the Streamlit app for visualizing the crime data and provided an interactive interface for querying crime records.

B. Tharun Teja Mogili (tharunte)

- Assisted in designing the ERD and defining the key entities in the database, ensuring all relationships between entities were correctly captured.
- Contributed to creating and populating the database with sample data and writing the SQL scripts for loading data into the tables.
- Focused on query optimization by identifying performance bottlenecks and adding indexes for improved query execution time.
- Worked on implementing additional features in the Streamlit app, including user input validation and error handling for custom SQL queries.

References

[1] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 7th ed., Pearson, 2015.

[2] PostgreSQL Global Development Group, *PostgreSQL Documentation*, https://www.postgresql.org/docs/.

[3] C. J. Date, *Database Design and Relational Theory: Normal Forms and All That Jazz*, O'Reilly Media, 2012.

[4] J. D. Ullman, *Principles of Database Systems*, 2nd ed., Computer Science Press, 1982.