

# Crime Investigation Database: Enhancing Efficiency and Accuracy in Criminal Investigations

1<sup>st</sup> Gopichandh Golla  
*ggolla*  
University at Buffalo  
Buffalo, USA  
ggolla@buffalo.edu

1<sup>st</sup> Sai Varnitha Rajanala  
*saivarni*  
University at Buffalo  
Buffalo, USA  
saivarni@buffalo.edu

1<sup>st</sup> Pavani Ayanambakam  
*pavaniay*  
University at Buffalo  
Buffalo, USA  
pavaniay@buffalo.edu

## I. INTRODUCTION

### A. Problem Statement

Law enforcement agencies face challenges in efficiently managing and analyzing large volumes of data related to criminal activities, suspects, witnesses, and evidence. The lack of a centralized and organized system often leads to delays in investigations, hindered collaboration among investigators, and potential errors in data management. This project aims to address these issues by developing a comprehensive Crime Investigation Database to streamline data collection, improve information accessibility, and enhance investigative capabilities.

### B. Background

In the realm of law enforcement, the timely and accurate investigation of crimes is paramount to ensure public safety and uphold justice. However, traditional methods of data management, such as paper-based records or fragmented digital files, often prove inadequate for handling the complexity and volume of information involved in criminal investigations. Investigators frequently encounter challenges in accessing relevant data, identifying patterns across cases, and effectively collaborating with other agencies.

### C. Significance of the Problem

The inefficiencies and shortcomings in existing data management practices can have serious consequences for the outcomes of criminal investigations. Delays in accessing critical information may result in missed opportunities to apprehend suspects or prevent further criminal activities. Moreover, inaccuracies or inconsistencies in data recording and sharing can undermine the integrity of evidence and compromise the justice system's credibility. Therefore, addressing these challenges is crucial for improving the effectiveness and reliability of law enforcement efforts.

### D. Potential Contribution

The development of a Crime Investigation Database offers significant potential to revolutionize the way law enforcement agencies handle and analyze data related to criminal activities.

By centralizing and standardizing information management processes, the database enables investigators to efficiently access and query relevant data, facilitating quicker decision-making and more informed investigative strategies. Additionally, the database's ability to track and analyze patterns across cases enhances investigators' capacity to identify trends, connections, and potential suspects, thereby strengthening the overall effectiveness of criminal investigations. Furthermore, the implementation of such a database fosters collaboration and information-sharing among different law enforcement agencies, promoting synergy in investigative efforts and maximizing resource utilization. Ultimately, the project's contribution lies in its potential to enhance the efficiency, accuracy, and efficacy of criminal investigations, leading to improved public safety and bolstered confidence in the justice system. While Excel files can be useful for storing and organizing data, they have several limitations that make them less suitable for managing complex and interconnected datasets in the context of a Crime Investigation System:

1. **Data Integrity:** Excel files lack robust mechanisms for enforcing data integrity and ensuring consistency across multiple sheets or workbooks. Without proper validation rules and relational constraints, it's easier for errors to occur, such as inconsistent data entry or accidental deletions.
2. **Scalability:** As the volume of data grows, Excel files may become unwieldy and slow to navigate, especially when dealing with large datasets spanning multiple tables or sheets. This can hinder the efficiency of data retrieval and analysis, particularly in complex investigative scenarios involving numerous cases, suspects, witnesses, and evidence.

### E. Target Users

The target users for the Crime Investigation Database are law enforcement agencies, criminal investigators, forensic analysts, and other judicial authorities involved in crime investigation and legal proceedings. This database is designed to enhance their ability to manage large volumes of data efficiently, identify patterns in criminal activities, and facilitate quicker decision-making in investigative processes.

Here is a scenario: The victim, found at a specific location (building), was a member of a gym (get fit now member). We suspect someone with a gym membership might be involved. Goal: Identify potential suspects based on gym membership and location. Data Used: • Person table (including name and address) • Get fit now member table (including membership ID and person ID) • Building table (including building ID and name) Now, let's discuss how the data would be used and what kinds of queries could be asked: Retrieve Crime Scene Report: Query to retrieve the crime scene report based on the date (Jan. 15, 2018) and location (SQL City). Investigative Queries: Retrieve all information about a specific crime, including suspects, witnesses, evidence, and assigned detective. List all crimes of a certain type or within a specified date range. Identify suspects or witnesses who are associated with multiple crimes. Find patterns or connections between different crimes or suspects. Track the status of ongoing investigations and update them as necessary. Update Operations: Add new crimes, suspects, witnesses, or evidence to the database as they are discovered during investigations. Update the status of a crime (e.g., from open to closed) when it has been resolved. Assign detectives to investigate new cases or reassign them as needed. Document statements provided by witnesses during interviews. Overall, the Crime Investigation System database would support both querying the data for investigative purposes and updating the database with new information as the investigation progresses, providing a comprehensive tool for managing and solving crimes.

## II. ER DIAGRAM AND RELATIONAL SCHEMA

### A. Relationships among the entities

person and drivers license: The person table has a one-to-one relationship with the drivers license table through license id. This means each person is associated with a unique driver's license.

person and facebook event checkin: The person table has a one-to-many relationship with the facebook event checkin table through id (in person) and person id (in facebook event checkin). A single person can check into many events, but each event check-in is associated with only one person.

person and interview: The person table has a one-to-one relationship with the interview table through id (in person) and person id (in interview). This implies that each person has a unique interview record.

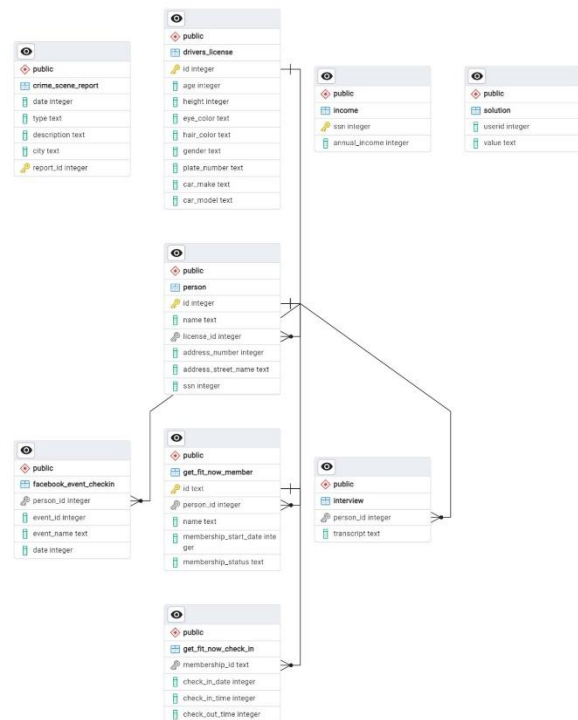
person and get fit now member: The person table has a one-to-one relationship with the get fit now member table through id (in person) and person id (in get fit now member). Each person can be a member of the gym, and each gym membership is linked to a single person.

get fit now member and get fit now check in: The get fit now member table has a one-to-many relationship with the get fit now check in table through id (in get fit now member) and membership id (in get fit now check in). A member can have multiple check-in and check-out times, but each check-in and check-out time is linked to a single gym member.

person and income: The person table has a one-to-one relationship with the income table through the ssn (social security number). This indicates that each person has one record of annual income associated with their SSN.

crime scene report and other tables: The crime scene report table doesn't seem to be directly related to the other tables in the diagram provided. If it is meant to be related, the relationship is not indicated in this diagram.

solution table: The solution table does not appear to have a direct relationship with any other table in the schema provided.



### B. Relational Schema

#### crime\_scene\_report

- date (integer): The date of the crime.
- type (text): Type of crime committed.
- description (text): Description of the crime.
- city (text): City where the crime occurred.

#### drivers\_license

- id (integer, PRIMARY KEY): Unique identifier for a driver's license.
- age (integer): Age of the driver.
- height (integer): Height of the driver in centimeters.
- eye\_color (text): Eye color of the driver.
- hair\_color (text): Hair color of the driver.
- gender (text): Gender of the driver.
- plate\_number (text): Car's license plate number.
- car\_make (text): Make of the car.
- car\_model (text): Model of the car.

#### person

- id (integer, PRIMARY KEY): Unique identifier for a person.
- name (text): Name of the person.

- **license\_id** (integer, FOREIGN KEY): Links to **drivers\_license(id)**.
- **address\_number** (integer): House number of the person's address.
- **address\_street\_name** (text): Street name of the person's address.
- **ssn** (integer): Social Security Number of the person.

#### **facebook\_event\_checkin**

- **person\_id** (integer, FOREIGN KEY): Links to **person(id)**.
- **event\_id** (integer): Identifier for the event.
- **event\_name** (text): Name of the event.
- **date** (integer): Date of the event.

#### **interview**

- **person\_id** (integer, FOREIGN KEY): Links to **person(id)**.
- **transcript** (text): Text of the interview transcript.

#### **get\_fit\_now\_member**

- **id** (text, PRIMARY KEY): Unique membership ID.
- **person\_id** (integer, FOREIGN KEY): Links to **person(id)**.
- **name** (text): Name of the member.
- **membership\_start\_date** (integer): Start date of the membership.
- **membership\_status** (text): Status of the membership (e.g., active, inactive).

#### **get\_fit\_now\_check\_in**

- **membership\_id** (text, FOREIGN KEY): Links to **get\_fit\_now\_member(id)**.
- **check\_in\_date** (integer): Date of check-in.
- **check\_in\_time** (integer): Time of check-in.
- **check\_out\_time** (integer): Time of check-out.

#### **income**

- **ssn** (integer, PRIMARY KEY): Social Security Number, uniquely identifying an income record.
- **annual\_income** (integer): Annual income associated with the SSN.

#### **solution**

- **userid** (integer): User identifier.
- **value** (text): Solution or answer value.

### **III. DATA COLLECTION**

The database comprises several tables, including crime scene report, drivers license, person, Facebook event checkin, interview, get fit now member, get fit now check in, income, and solution. Each table serves a specific function in the collection and analysis of data.

The crime scene report table documents crime incidents by logging the date, type of crime, a description, and the city where it occurred. This table plays a crucial role by potentially linking individuals or locations to criminal activities. Next, the drivers license table captures detailed personal and vehicular information such as age, height, eye and hair color, gender, license plate number, and car details. It connects individuals to their vehicles, which can be critical

in solving crimes based on vehicle sightings or suspect descriptions.

The person table is central to linking various personal identifiers, housing data such as a unique ID, name, license ID which connects to the drivers license table, and the individual's address and SSN. This establishes a connection between people and potential crimes through personal data or witness statements collected in other tables like the interview table, which contains transcripts of interviews that could provide clues or confirm alibis.

Tables such as the Facebook event checkin track which events individuals have attended, offering insights into their locations at specific times which could be crucial in establishing whereabouts during crimes. The get fit now member and check in tables detail fitness center activities, including membership details and exact check-in and check-out times, pinpointing individual locations at precise times.

Lastly, the income table lists annual incomes linked by SSN, which might be used to analyze motives or socio-economic profiles. The solution table, currently empty, is designed to record the outcomes of investigations or challenges. It's important to note that the database setup prohibits cascading deletes, ensuring that deleting entries with foreign keys does not remove linked data inadvertently, thus maintaining data integrity.

### **IV. BCNF CONVERSION**

We'll review each table to determine if they are in Boyce-Codd Normal Form (BCNF), ensuring there are no non-trivial functional dependencies (FDs) where the left-hand side is not a superkey.

#### **Crime Scene Report**

Modified Attributes: Added **report\_id** as a primary key.

Functional Dependencies (FDs):

**report\_id** → date, type, description, city

Analysis: With **report\_id** as a primary key, all attributes are functionally dependent on it. There are no non-trivial dependencies where the left side isn't a superkey. Therefore, the table is in BCNF.

#### **Drivers License**

Primary Key: **id**

Functional Dependencies (FDs):

**id** → age, height, eye\_color, hair\_color, gender,

plate\_number, car\_make, car\_model

plate\_number → id, age, height, eye\_color, hair\_color,

gender, car\_make, car\_model

Analysis: **id** and **plate\_number** can each uniquely identify all attributes in the table. There are no non-trivial dependencies on non-superkeys. Thus, the table is in BCNF.

#### **Person**

Primary Key: **id**

Functional Dependencies (FDs):

**id** → name, license\_id, address\_number,

address\_street\_name, ssn

license\_id → id, name, address\_number,  
address\_street\_name, ssn

Analysis: Both id and license\_id are candidate keys and there are no non-trivial dependencies on non-superkeys. The table is in BCNF.

### Facebook Event Checkin

Assumed Keys: person\_id, event\_id as a composite key.

Functional Dependencies (FDs):

event\_id → event\_name, date

person\_id, event\_id → event\_name, date

Analysis: Initially, the table was not in BCNF because event\_id alone determined event\_name and date. By decomposing into facebook\_event\_checkin(person\_id, event\_id) and facebook\_events(event\_id, event\_name, date), we ensure all tables adhere to BCNF.

### Interview

Modified Attributes: Added interview\_id as a primary key.

Functional Dependencies (FDs):

interview\_id → person\_id, transcript

Analysis: With interview\_id as the sole determinant for all attributes and no other attributes functionally determining interview\_id, the table is in BCNF.

### Get Fit Now Member

Primary Key: id

Functional Dependencies (FDs):

id → person\_id, name, membership\_start\_date,

membership\_status

person\_id → id, name, membership\_start\_date,

membership\_status

Analysis: Both id and person\_id function as candidate keys, with no non-trivial dependencies on non-superkeys. The table is in BCNF.

### Get Fit Now Check In

Primary Key: check\_in\_id added.

Functional Dependencies (FDs):

membership\_id, check\_in\_date, → check\_out\_time

Analysis: By adding check\_in\_id as a primary key and assuming no other attributes functionally determine check\_in\_id, the table is restructured into BCNF.

### Income

Primary Key: ssn

Functional Dependencies (FDs):

ssn → annual\_income

Analysis: With ssn uniquely determining annual\_income and no other attributes functionally determining ssn, the table is in BCNF.

### Solution

Primary Key: userid

Functional Dependencies (FDs):

userid → value

Analysis: Given userid uniquely determines value and there are no other dependencies, the table is in BCNF.

Each table has been analyzed and adjusted as necessary to ensure compliance with BCNF, thereby eliminating potential anomalies and maintaining database integrity and efficiency.

## V. IMPLEMENTING SQL QUERIES

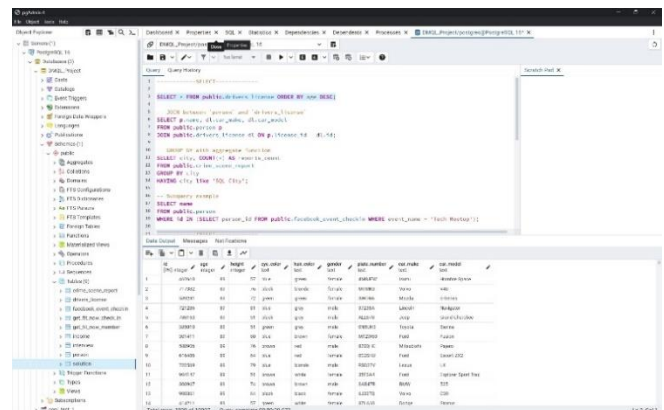
The first task in the project was to install and set up PostgreSQL, a powerful open-source relational database management system. The installation process involved downloading the PostgreSQL installer, running the setup wizard, and configuring the database environment. Additionally, we installed pgAdmin, a graphical administration tool for PostgreSQL, to facilitate database management tasks such as creating and managing databases, tables, and queries. After logging in using the superuser credentials set during installation. Right-clicked on "Databases" and selected "Create > Database". Entered the database name, "DMQL\_Project", and clicked "Save". Then, right-clicked on the newly created database and selected "QueryTool".

We created the tables crime\_scene\_report, drivers\_license, person, facebook\_event\_checkin, interview, get\_fit\_now\_member, get\_fit\_now\_check\_in, income and solution.

After creating the required tables, we imported the respective csv files into the tables. Later, we have run the select, insert and update queries as shown below.

### Query 1:

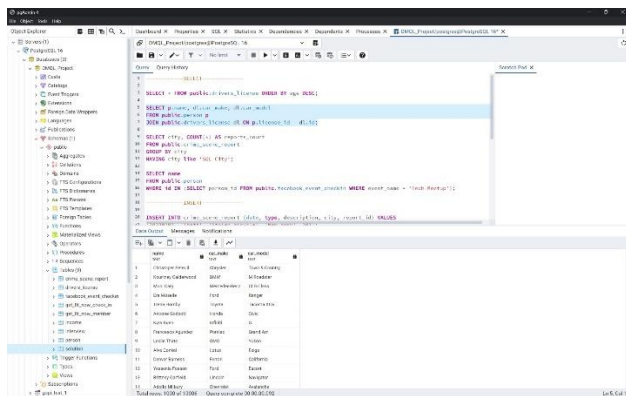
This query selects all columns (\*) from the table drivers\_license in the public schema. It orders the results by the age column in descending order (DESC). The execution of the query 1 is shown below



id	person_id	car_make	car_model	year	license_plate	sex	height	weight	hair_color	eye_color	skin_color	address_street_name	address_number	city	state	zip_code	last_updated
1	1000001	BMW	3 Series	2018	ABC123	M	175	70	Brown	Blue	Fair	123 Main St	456	New York	NY	10001	2023-10-27
2	1000002	Ford	F-150	2017	DEF456	F	165	60	Black	Brown	Tan	789 Oak St	321	Los Angeles	CA	90001	2023-10-27
3	1000003	Toyota	Camry	2019	GHI789	M	180	75	Black	Green	Dark	101 Pine St	210	Chicago	IL	60601	2023-10-27
4	1000004	Honda	Civic	2018	JKL012	F	160	55	Blonde	Blue	Fair	432 Elm St	567	San Francisco	CA	94101	2023-10-27
5	1000005	Mercedes	C-Class	2020	MNO345	M	185	80	Black	Brown	Dark	654 Maple St	890	Miami	FL	33101	2023-10-27
6	1000006	Vauxhall	Innova	2016	PQR678	F	155	50	Black	Green	Dark	987 Cedar St	109	Seattle	WA	98101	2023-10-27
7	1000007	Subaru	Outback	2017	STU901	M	170	65	Black	Blue	Fair	210 Birch St	321	Portland	OR	97201	2023-10-27
8	1000008	Jeep	Wrangler	2018	VWX234	F	168	62	Black	Brown	Tan	543 Spruce St	678	Denver	CO	80201	2023-10-27
9	1000009	Volvo	S60	2019	YZA567	M	182	78	Black	Blue	Dark	876 Ash St	901	Phoenix	AZ	85001	2023-10-27
10	1000010	Nissan	Altima	2017	BCD890	F	162	58	Black	Green	Dark	109 Hickory St	210	San Antonio	TX	78201	2023-10-27

### Query 2:

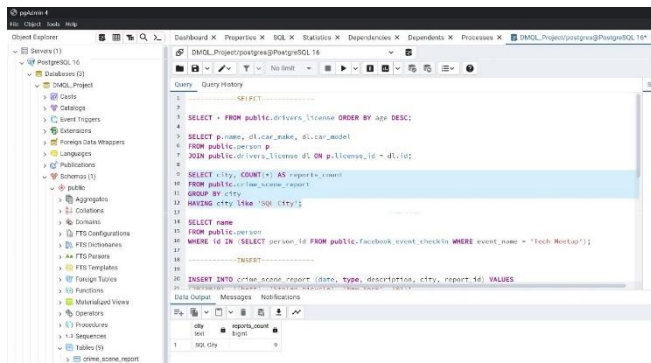
This query selects the name column from the person table and the car\_make and car\_model columns from the drivers\_license table. It performs an inner join between the person and drivers\_license tables based on the condition p.license\_id = dl.id. The execution of the query 2 is shown below



### Query 3:

This query selects the city column from the crime\_scene\_report table. It counts the number of reports for each city (COUNT(\*)) and aliases the count as reports\_count. It groups the results by city. It filters the results to only include cities that match the pattern 'SQL City' using the HAVING clause.

The execution of the query 3 is shown below

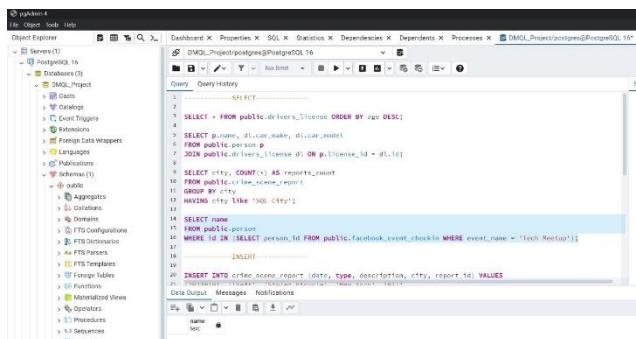


### Query 4:

This query selects the name column from the person table.

It filters the results to only include rows where the id column matches any person\_id values retrieved from the subquery.

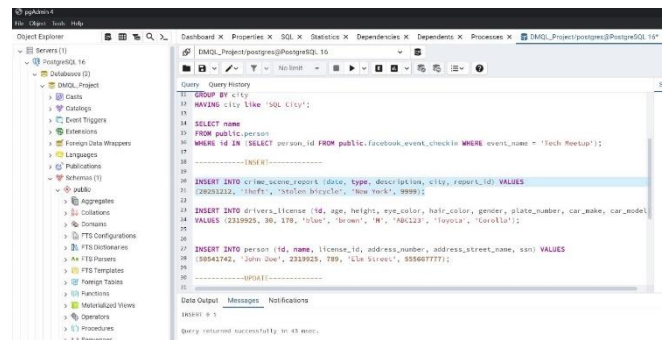
The subquery selects person\_id from the facebook\_event\_checkin table where the event\_name is 'Tech Meetup'. The execution of the query 4 is shown below



### Query 5:

This query inserts a new row into the crime\_scene\_report table.

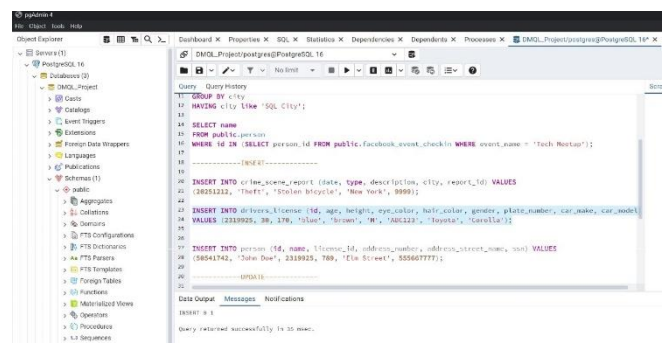
It specifies values for the date, type, description, city, and report\_id columns. The execution of the query 5 is shown below



### Query 6:

This query inserts a new row into the drivers\_license table.

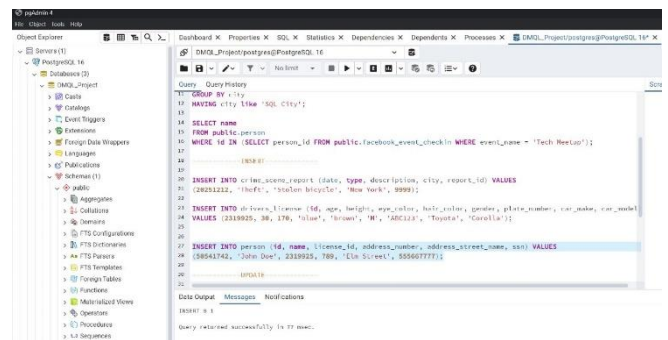
It specifies values for the id, age, height, eye\_color, hair\_color, gender, plate\_number, car\_make, and car\_model columns. The execution of the query 6 is shown below



### Query 7:

This query inserts a new row into the person table.

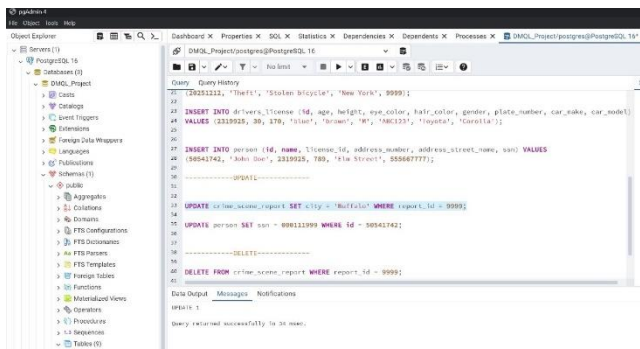
It specifies values for the id, name, license\_id, address\_number, address\_street\_name, and ssn columns.



### Query 8:

This query updates the city column in the crime\_scene\_report table. It sets the value of city to 'Buffalo' where the report\_id is 9999.

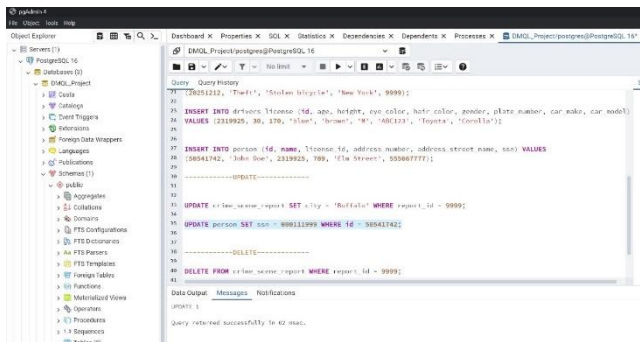




### Query 9:

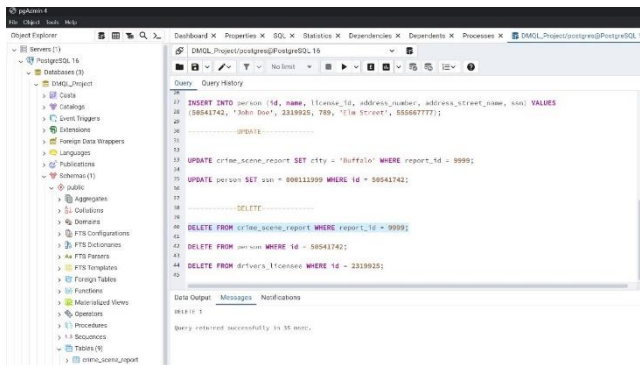
This query updates the ssn column in the person table.

It sets the value of ssn to 000111999 where the id is 50541742. The execution of the query is shown below



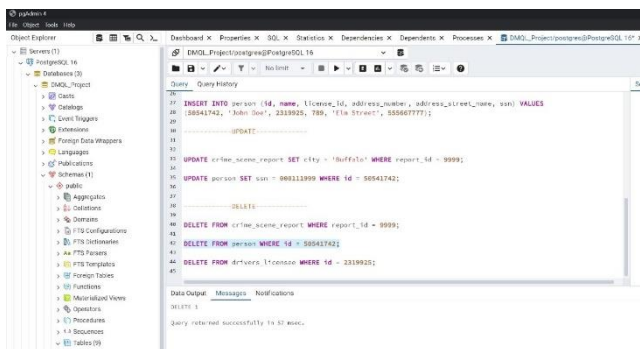
### Query 10:

This query deletes rows from the crime\_scene\_report table where the report\_id is 9999. The execution of the query is shown below



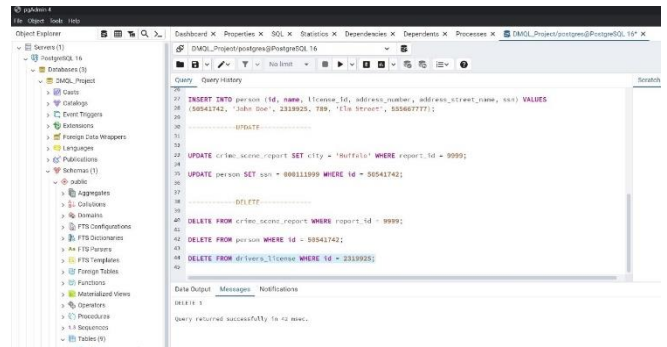
### Query 11:

This query deletes rows from the person table where the id is 50541742. The execution of the query is shown below



### Query 12:

This query deletes rows from the drivers\_license table where the id is 2319925.



## VI. PROBLEMS WHILE HANDLING THE LARGER DATASET

Containing tables such as crime\_scene\_report, drivers\_license, person, and others, handling large datasets can indeed present specific challenges, especially concerning performance

### Problems with Large Datasets

**Slow Query Performance:** As the size of the tables grows, queries that need to scan entire tables or join multiple large tables can become significantly slower. This is particularly problematic for operations like filtering (WHERE clauses), sorting (ORDER BY clauses), and joining tables on non-indexed columns.

**High Memory Usage:** Large datasets can consume a substantial amount of memory, especially during complex queries involving multiple joins and aggregations, which may lead to increased I/O operations if the data doesn't fit in memory.

**Inefficient Indexing:** Over-indexing can lead to excessive maintenance overhead during insertions, updates, and deletions, while under-indexing can lead to slow retrievals. Finding the right balance is key.

### Adopted Solutions:

#### Strategic Indexing:

**Primary and Foreign Keys:** Ensuring all primary keys are indexed (usually automatic in most DBMSs) and indexing foreign keys to speed up JOIN operations. For instance, indexing person\_id in the facebook\_event\_checkin table or license\_id in the person table can significantly expedite joins with the person table.

**Frequently Queried Columns:** Columns used frequently in WHERE clauses, such as date in crime\_scene\_report and city, were indexed to speed up searches and filter operations.

#### Query Optimization:

**Query Refactoring:** Rewriting queries to exploit the strengths of the database schema and its indexes. This includes using EXISTS instead of IN for subqueries, or adjusting the JOIN order to ensure that the database engine can utilize indexes effectively.

**Use of EXPLAIN:** Regular use of the EXPLAIN statement to analyze query performance and understand how indexes are being used, helping to fine-tune them.

Implementing these indexing strategies and optimizations led to markedly improved query performance, especially in

reducing the execution time for complex queries involving large datasets. Memory usage during peak times was also optimized by reducing the need for full table scans.

## VII. QUERY EXECUTION ANALYSIS

By understanding query optimization techniques and their impact on PostgreSQL's execution plans, we can effectively improve query performance and database efficiency. Here, we have given 3 problematic queries and detailed execution plan to improve these queries.

### Query1:

#### *Problem with IN Clause Using Subqueries*

When you use a subquery with an IN clause, the database engine executes the subquery first, collecting all the results into a temporary list. The outer query then scans through its target table and checks for each row if its value exists in the temporary list produced by the subquery. Large datasets can produce large temporary lists, which consume considerable memory and processing power.

*Before applying the execution plan:*

The screenshot shows the query execution plan for the following query: `EXPLAIN ANALYSE SELECT name FROM public.person WHERE id IN (SELECT person_id FROM public.facebook_event_checkin WHERE event_name = 'Tech Meetup')`. The plan is as follows:

Step	Operation	Cost	Rows	Width	Actual Time	Actual Rows	Loops
1	Sort	209.96	210.01	14	0.489	0.491	0
2	Sort Key: p.name						
3	Sort Method: quicksort Memory: 25kB						
4	Nested Loop	4.72	209.55	19	0.468	0.469	0
5	Seq Scan on facebook_event_checkin fec	4.43	63.81	19	0.467	0.467	0
6	Recheck Cond: (event_name = 'Tech Meetup':text)						
7	Bitmap Index Scan on idx_facebook_event_checkin_event_name	0.00	4.43	19	0.463	0.464	0
8	Index Cond: (event_name = 'Tech Meetup':text)						
9	Index Scan using person_pkey on person p	0.29	7.67	18	(never executed)		
10	Index Cond: (id = fec.person_id)						
11	Planning Time				10.830 ms		
12	Execution Time				2.515 ms		

#### *Improvement with JOIN:*

Switching to a JOIN operation, particularly in the form of an INNER JOIN, allows the database to more efficiently use indexes and join algorithms.

#### *Simpler Execution Plan:*

The execution plan for a JOIN is often simpler and more direct than for a subquery with an IN clause, making it easier for the database to optimize.

*After applying the execution plan:*

The screenshot shows the query execution plan for the following query: `EXPLAIN ANALYSE SELECT DISTINCT p.name FROM public.person p JOIN public.facebook_event_checkin fec ON p.id = fec.person_id WHERE fec.event_name = 'Tech Meetup'`. The plan is as follows:

Step	Operation	Cost	Rows	Width	Actual Time	Actual Rows	Loops
1	Sort	209.96	210.01	14	0.489	0.491	0
2	Sort Key: p.name						
3	Sort Method: quicksort Memory: 25kB						
4	Nested Loop	4.72	209.55	19	0.468	0.469	0
5	Bitmap Heap Scan on facebook_event_checkin fec	4.43	63.81	19	0.467	0.467	0
6	Recheck Cond: (event_name = 'Tech Meetup':text)						
7	Bitmap Index Scan on idx_facebook_event_checkin_event_name	0.00	4.43	19	0.463	0.464	0
8	Index Cond: (event_name = 'Tech Meetup':text)						
9	Index Scan using person_pkey on person p	0.29	7.67	18	(never executed)		
10	Index Cond: (id = fec.person_id)						
11	Planning Time				10.830 ms		
12	Execution Time				2.515 ms		

### Query2:

**Group By:** This operation groups the entire crime\_scene\_report table by city, computing a count for each city.

**Having Clause:** After grouping, the HAVING clause filters these results to include only groups where the city is like 'SQL City'. Since 'like' is used, even though it is used with a full string match pattern, it's generally prepared to handle patterns, which might involve additional overhead.

*Optimizations Made:*

**Early Filtering:** The WHERE clause filters records before any grouping operation occurs. This means the database engine handles a much smaller dataset — only the rows with city equal to 'SQL City' — during the costly grouping operation.

**Reduced Workload:** By filtering first, the database avoids unnecessary calculations of counts for all other cities that are ultimately not required. This reduction in computational overhead can significantly decrease query execution time, especially for large datasets.

*Before applying query execution 2*

The screenshot shows the query execution plan for the following query: `EXPLAIN ANALYSE SELECT city, COUNT(*) AS reports_count FROM public.crime_scene_report GROUP BY city HAVING city like 'SQL City'`. The plan is as follows:

Step	Operation	Cost	Rows	Width	Actual Time	Actual Rows	Loops
1	GroupAggregate	0.28	4.59	9	1.706	1.707	1
2	Group Key: city						
3	Index Only Scan using idx_crime_scene_report_city on crime_scene_report	0.28	4.46	9	1.321	1.332	0
4	Index Cond: (city = 'SQL City':text)						
5	Filter: (city ~ 'SQL City':text)						
6	Heap Fetches: 0						
7	Planning Time				4.754 ms		
8	Execution Time				1.757 ms		

**Simpler Comparison:** Using = for the comparison rather than LIKE is more straightforward and efficient when you know the exact string match is needed. The LIKE operator generally involves pattern matching, which is slower than direct string comparison.

*After applying the execution plan:*

Query History	Scratc
<pre>EXPLAIN ANALYZE SELECT city, COUNT(*) AS reports_count FROM public.crime_scene_report WHERE city = 'SQL City' GROUP BY city;</pre>	
Query Output	Messages Notifications
<p>QUERY PLAN</p> <p>text</p> <p>GroupAggregate (cost=0.28..4.47 rows=1 width=17) (actual time=0.069..0.069 rows=1 loops=1)</p> <p>→ Index Only Scan using idx_crime_scene_report_city on crime_scene_report (cost=0.28..4.43 rows=9 width=9) (actual time=0.045..0.047 rows=9 loops=1)</p> <p>Index Cond: (city = 'SQL City'::text)</p> <p>Heap Fetches: 0</p> <p>Planning Time: 0.220 ms</p> <p>Execution Time: 0.111 ms</p>	

### Query3:

If license\_id is not indexed, this join could be slow.

*Optimization:* license\_id in person and id in drivers\_license are indexed.

Query History	
<pre>1 EXPLAIN ANALYZE SELECT p.name, dl.car_make, dl.car_model 2 FROM public.person p 3 JOIN public.drivers_license dl ON p.license_id = dl.id;</pre>	
Query Output	Messages Notifications
<p>QUERY PLAN</p> <p>text</p> <p>Hash Join (cost=321.25..661.91 rows=10007 width=28) (actual time=2.019..4.273 rows=10006 loops=1)</p> <p>Hash Cond: (dl.id = p.license_id)</p> <p>→ Seq Scan on drivers_license dl (cost=0.00..203.07 rows=10007 width=18) (actual time=0.024..0.500 rows=10007 loops=1)</p> <p>→ Hash (cost=196.11..196.11 rows=10011 width=18) (actual time=1.919..1.920 rows=10011 loops=1)</p> <p>Buckets: 16384 Batches: 1 Memory Usage: 639kB</p> <p>→ Seq Scan on person p (cost=0.00..196.11 rows=10011 width=18) (actual time=0.020..0.908 rows=10011 loops=1)</p> <p>Planning Time: 4.429 ms</p> <p>Execution Time: 4.557 ms</p>	

*After applying the execution plan:*

Query History	
<pre>1 EXPLAIN ANALYZE SELECT p.name, dl.car_make, dl.car_model 2 FROM public.person p 3 JOIN public.drivers_license dl ON p.license_id = dl.id; 4 5 CREATE INDEX ON public.person (license_id); 6 CREATE INDEX ON public.drivers_license (id);</pre>	
Query Output	Messages Notifications
<p>QUERY PLAN</p> <p>text</p> <p>Hash Join (cost=321.25..661.91 rows=10007 width=28) (actual time=1.706..3.726 rows=10006 loops=1)</p> <p>Hash Cond: (dl.id = p.license_id)</p> <p>→ Seq Scan on drivers_license dl (cost=0.00..203.07 rows=10007 width=18) (actual time=0.012..0.407 rows=10007 loops=1)</p> <p>→ Hash (cost=196.11..196.11 rows=10011 width=18) (actual time=1.626..1.627 rows=10011 loops=1)</p> <p>Buckets: 16384 Batches: 1 Memory Usage: 639kB</p> <p>→ Seq Scan on person p (cost=0.00..196.11 rows=10011 width=18) (actual time=0.017..0.733 rows=10011 loops=1)</p> <p>Planning Time: 3.823 ms</p> <p>Execution Time: 4.079 ms</p>	

Indexes transform the way the database engine performs the join operation. Instead of scanning every row in one table and looking for matching rows in the other (a process that can be very slow for large tables), the database can use the indexes to quickly locate the matching rows.

*Before applying execution plan:*

With indexes available, PostgreSQL can opt for more efficient join strategies such as hash joins or merge joins, which are generally faster than nested loop joins when dealing with large datasets. Here's a brief on how each works:

**Hash Join:** PostgreSQL can use the index to build a memory-efficient hash table for the smaller of the two tables, which speeds up the search for matching join keys in the larger table.

**Merge Join:** If both indexed columns (license\_id and id) are also sorted, PostgreSQL might employ a merge join, which efficiently merges two sorted lists.

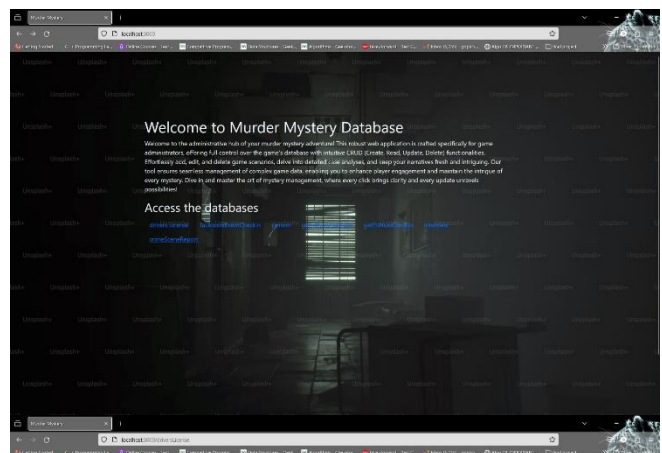
**Index Utilization:** JOINS can efficiently use indexes on the joining columns. If person\_id in facebook\_event\_checkin and id in person are indexed, the database can quickly match rows between the two tables without scanning all rows.

**Single Pass Efficiency:** A JOIN typically consolidates the operations into a single pass over the data, rather than handling a nested operation which may require multiple passes.

## VIII.APPLICATION DEVELOPMENT

We developed a web application for the SQL Murder Mystery This web application is designed to enhance user interaction with the database by allowing administrative users to perform complete CRUD operations. CRUD stands for Create, Read, Update, and Delete, which are the four basic functions required to manage database entries effectively. This functionality allows the administrator to add new data to the database, view existing entries, modify the data, and remove any unnecessary or outdated information. The addition of these capabilities makes it easier for users to manage the database directly through the web interface, improving the accessibility and utility of the SQL Murder Mystery database.

Below are the relevant screenshots of the web application



driverslicense	
Driver's ID: 100280	plate number: 72
<a href="#">View Driver details</a>	
Driver's ID: 100460	plate number: 61
<a href="#">View Driver details</a>	
Driver's ID: 101029	plate number: 62
<a href="#">View Driver details</a>	
Driver's ID: 101198	plate number: 63
<a href="#">View Driver details</a>	
Driver's ID: 101255	plate number: 19
<a href="#">View Driver details</a>	
Driver's ID: 101494	plate number: 48
<a href="#">View Driver details</a>	



A screenshot of a web browser showing a form titled 'Add New Record'. The form contains the following fields: ID, age, height, eye color, hair color, gender, plate number, car make, and car model. Each field has a corresponding input box. At the bottom of the form is a green button labeled 'Add Data'.

## IX. CONTRIBUTIONS

- All team members contributed equally to the project's success, with shared responsibilities ensuring consistent progress.
- Varnitha has done Data collection, database creation and developed ER Diagram and Relational Schema
- Gopichandh managed Front end development, deployment, technical aspects, including SQL queries and database schema setup.
- Pavani worked on BCNF conversions, Backend Development, application API, reporting, and optimization

## X. REFERENCES

- [1] <https://www.kaggle.com/datasets/johnp47/sql-murder-mystery-database>
- [2] <https://github.com/NUKnightLab/sql-mysteries/blob/master/schema.png>
- [3] <https://aws.amazon.com/rds/postgresql/what-is-postgresql/#:~:text=PostgreSQL%20is%20used%20as%20the,%2C%20geospatial%2C%20and%20analytics%20applications.>
- [4] [https://www.w3schools.com/sql/sql\\_syntax.asp](https://www.w3schools.com/sql/sql_syntax.asp)
- [5] <https://medium.com/@malexmad/how-to-use-pgadmin-a9addc7ff46c>
- [6] <https://www.geeksforgeeks.org/best-practices-for-sql-query-optimizations/>
- [7] <https://www.postgresql.org/developer/related-projects/>
- [8] [https://www.pgadmin.org/docs/pgadmin4/8.4/query\\_tool.html](https://www.pgadmin.org/docs/pgadmin4/8.4/query_tool.html)

**IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper before submission to the conference. Failure to remove template text from your paper may result in your paper not being public**