Vehicle Insurance Database: Streamlining Data Management for Enhanced Analysis

Sneha Pydipati
Engineering Science Data Science
University at Buffalo

Prem Kumar Garikapati
Engineering Science Data Science
University at Buffalo

Sri Nikitha Sarikonda
Engineering Science Data Science
University at Buffalo

***Abstract*—The Vehicle Insurance Management Database is designed to efficiently manage data related to users, customers, vehicles, insurance policies, claims, agents, payments, and police records. This database addresses the challenges of data organization, integrity, and analysis, providing a robust platform for stakeholders including insurance companies, agents, and customers. By leveraging relational database principles, this system enhances data retrieval, reporting, and overall decision-making processes.**

***Index Terms*—Database Management, Data Integrity, Insurance Data, Relational Database, Data Analysis, User Management**

I. Problem Statement

The vehicle insurance industry faces significant challenges in managing extensive data related to users, customers, vehicles, insurance policies, claims, agents, payments, and police records. Existing data management systems often suffer from issues such as data redundancy, inefficiencies in data retrieval, and a lack of data integrity, which hinder stakeholders — including insurance companies, agents, and customers — from making informed decisions and providing quality service.

Multiple data sources and unstructured storage lead to duplicated information, complicating data accuracy and consistency. Additionally, slow query performance due to poorly structured databases results in delays when accessing critical information, negatively impacting customer service and operational efficiency. This lack of efficiency can have far-reaching consequences, including decreased customer satisfaction and lost business opportunities.

To address these challenges, the objective is to develop a comprehensive Vehicle Insurance Management Database that provides a centralized, structured, and efficient system for managing all relevant data. This database will enhance data integrity, streamline data retrieval processes, and enable insightful analysis, ultimately improving decision-making and operational efficiency for stakeholders in the vehicle insurance industry.

II. Data Sources

The insurance claims dataset, sourced from Kaggle, comprises over 32,000 records of insurance claims, including details on policyholders, claims, and payments. This comprehensive dataset provides a unique opportunity to analyze claims patterns, identify trends, and develop predictive models to improve claims management. By leveraging this [1] dataset, stakeholders can gain valuable insights into the claims process, ultimately enhancing operational efficiency and reducing costs. To address missing data in certain columns, we utilized Python's Faker library to generate realistic and synthetic data, ensuring a more robust and complete dataset for analysis here [2]

III. Database Design

Our database design is structured to efficiently manage data related to insurance claims, policyholders, and insurance policies. The design incorporates entity-relationship (E/R) modeling to accurately represent the relationships between various entities, such as linking policyholders to their respective claims and policies. This structured approach facilitates organized storage, retrieval, and analysis of information, enabling comprehensive insights into claims patterns and enhancing overall operational efficiency within the insurance domain.

A. Relational Schema

1) Users: Stores user information for system access. The primary key is `user_id`. The attributes include:

- `user_id (INTEGER), username (VARCHAR), password (VARCHAR), role (VARCHAR), created_at (TIMESTAMP)`

2) Customers: Stores customer details linked to users. The primary key is `customer_id`, with `user_id` as a foreign key referencing the Users table. The attributes include:

- `customer_id (INTEGER), ,user_id (INTEGER), customer_name (VARCHAR),customer_address (VARCHAR),customer_age (INTEGER) ,dl_num (CHAR) ,created_at (TIMESTAMP)`

3) Vehicles: Contains details about vehicles owned by customers. The primary key is `vehicle_id`, with `customer_id` as a foreign key referencing the Customers table. The attributes include:

- `vehicle_id (INTEGER) ,model (VARCHAR) ,vehicle_age (NUMERIC) ,segment (VARCHAR) ,fuel_type (VARCHAR)`

,airbags (INTEGER) ,ncap_rating (INTEGER) ,is_parking_camera (VARCHAR) ,is_speed_alert (VARCHAR) ,is_brake_assist (VARCHAR)

4) Insurance: Stores insurance policy details. The primary key is policy_id, with vehicle_id and customer_id as foreign keys referencing the Vehicles and Customers tables, respectively. The attributes include:

- policy_id (VARCHAR) ,subscription_length (NUMERIC) ,claim_status (VARCHAR)

5) Claims: Records claims made by customers. The primary key is claim_id, with policy_id and customer_id as foreign keys referencing the Insurance and Customers tables, respectively. The attributes include:

- claim_id (INTEGER) ,policy_id (VARCHAR) ,customer_id (INTEGER) ,claim_date (DATE) ,claim_amount (DECIMAL) ,claim_status (VARCHAR) ,resolution_date (DATE)

6) Agents: Stores information about insurance agents. The primary key is agent_id, with user_id as a foreign key referencing the Users table. The attributes include:

- agent_id (INTEGER) ,user_id (INTEGER) ,agent_name (VARCHAR) ,contact_number (VARCHAR) ,email (VARCHAR)

7) Payments: Manages payment records for insurance policies. The primary key is payment_id, with policy_id and customer_id as foreign keys referencing the Insurance and Customers tables, respectively. The attributes include:

- payment_id (INTEGER) ,policy_id (VARCHAR) ,customer_id (INTEGER) ,payment_date (DATE) ,amount (INTEGER) ,payment_method (VARCHAR) ,payment_status (VARCHAR)

8) Police: Stores information related to police records. The primary key is police_id, with user_id as a foreign key referencing the Users table. The attributes include:

- police_id (INTEGER) ,user_id (INTEGER) ,police_name (TEXT) ,police_station (TEXT) ,police_contact (TEXT)

B. Functional Dependencies and BCNF Justification

1) Users Table:

- user_id → username, password, role, created_at
- The primary key is user_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

2) Customers Table:

- customer_id → user_id, customer_name, customer_address, customer_age, dl_num, created_at
- The primary key is customer_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

3) Vehicles Table:

- vehicle_id → model, vehicle_age, segment, fuel_type, airbags, ncap_rating, is_parking_camera, is_speed_alert, is_brake_assist, customer_id
- The primary key is vehicle_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

4) Insurance Table:

- policy_id → subscription_length, claim_status, vehicle_id, customer_id
- The primary key is policy_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

5) Claims Table:

- claim_id → policy_id, customer_id, claim_date, claim_amount, claim_status, resolution_date
- The primary key is claim_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

6) Agents Table:

- agent_id → user_id, agent_name, contact_number, email
- The primary key is agent_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

7) Payments Table:

- payment_id → policy_id, customer_id, payment_date, amount, payment_method, payment_status
- The primary key is payment_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

8) Police Table:

- police_id → user_id, police_name, police_station, police_contact
- The primary key is police_id, which determines all other attributes.
- There are no partial or transitive dependencies.
- The table is in BCNF.

C. ER Diagram

The ER diagram represents an extended version of the database schema with the addition of four new tables that

were not present in Milestone 1. These new tables enhance the functionality and granularity of the database.

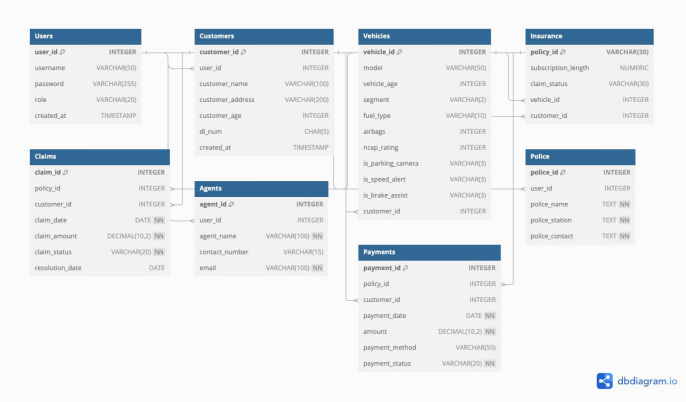

Fig. 1. Updated ER Diagram

IV. CONSTRAINTS AND DATA VALIDATION

In our database, various constraints and data validation mechanisms are implemented to ensure the data integrity, accuracy, and consistency of the data stored across different tables. These constraints are crucial for maintaining high-quality data and preventing errors during data entry and manipulation. Below are the key types of constraints utilized in the database design, along with relevant SQL queries:

A. Primary Key Constraints

Primary key constraints uniquely identify records in a table, ensuring that no two rows can have the same value in the primary key columns.

Example:

```
CREATE TABLE IF NOT EXISTS Users (
    user_id INTEGER PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(255),
    role VARCHAR(20),
    created_at TIMESTAMP
    DEFAULT CURRENT_TIMESTAMP
);
```

In this example, `user_id` is the primary key for the `Users` table, ensuring each user has a unique identifier.

B. Foreign Key Constraints

Foreign key constraints establish relationships between tables, ensuring referential integrity. They prevent orphaned records by ensuring that a record in one table cannot reference a non-existent record in another table.

Example:

```
CREATE TABLE IF NOT EXISTS Customers (
    customer_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    customer_name VARCHAR(100),
    customer_address VARCHAR(200),
    customer_age INTEGER,
    FOREIGN KEY (user_id)
    REFERENCES Users(user_id)
    ON DELETE SET NULL
);
```

Here, `user_id` in the `Customers` table is a foreign key referencing `user_id` in the `Users` table. If a user is deleted, the `ON DELETE SET NULL` action ensures that the corresponding `user_id` in the `Customers` table is set to NULL.

C. Unique Constraints

Unique constraints ensure that all values in a column are distinct, preventing duplicate entries.

Example:

```
CREATE TABLE IF NOT EXISTS Agents (
    agent_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    agent_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    FOREIGN KEY (user_id)
    REFERENCES Users(user_id)
    ON DELETE SET NULL
);
```

In this case, the `email` column in the `Agents` table has a unique constraint, ensuring that no two agents can have the same email address.

D. Check Constraints

Check constraints enforce specific conditions on the values in a column, ensuring that data entered into the database meets certain criteria.

Example:

```
ALTER TABLE Customers ADD CONSTRAINT
CHK_Customers_Age CHECK (customer_age > 0);
ALTER TABLE Payments ADD CONSTRAINT
CHK_Payments_Amount CHECK (amount > 0);
```

The first query ensures that `customer_age` must be greater than zero, while the second query ensures that the `amount` in the `Payments` table is a positive value.

E. Not Null Constraints

Not null constraints ensure that a column cannot have a NULL value, meaning that a value must be provided for that column when a new record is created.

Example:

```
CREATE TABLE IF NOT EXISTS Claims (
    claim_id INTEGER PRIMARY KEY,
    policy_id VARCHAR(30),
    claim_date DATE NOT NULL,
    claim_amount DECIMAL(10, 2) NOT NULL,
    claim_status VARCHAR(20) NOT NULL,
    FOREIGN KEY (policy_id)
    REFERENCES Insurance(policy_id)
    ON DELETE CASCADE
);
```

In this example, the `claim_date` and `claim_amount` columns are marked as NOT NULL, ensuring that every claim must have a date and an amount associated with it.

F. Importance of Constraints and Data Validation

The implementation of these constraints in our database is vital for: - **Data Integrity**: Ensuring that all data entries are accurate and reliable. - **Consistency**: Maintaining uniformity across the database by adhering to defined rules. - **Error Prevention**: Reducing the likelihood of data entry errors and inconsistencies. - **Simplified Data Management**: Allowing users and developers to focus on data analysis rather than data quality issues.

In summary, the constraints and data validation mechanisms in the database are essential for maintaining high-quality data, ensuring that the system operates effectively and meets the needs of its users.

V. SQL Query Testing

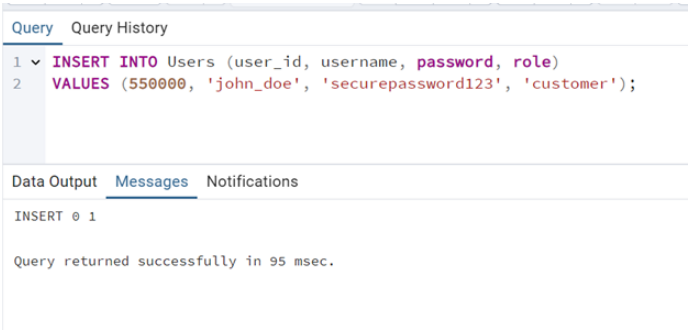

Fig. 2. Insert a New User

This SQL query inserts a new record into the Users table. It adds a user with user_id 550000, username ”john_doe”, password ”securepassword123”, and assigns the role of ”customer.” The query executes successfully, confirming one record was inserted.

Fig. 3. Insert a New Customer

This SQL query inserts a new record into the Customers table. It adds a customer with customer_id 120000, linked to user_id 1, named ”Alice Smith,” residing at ”123 Maple St, Springfield,” aged 30, and holding a driver’s license number ”D1234.” The query executed successfully, confirming one record was added.

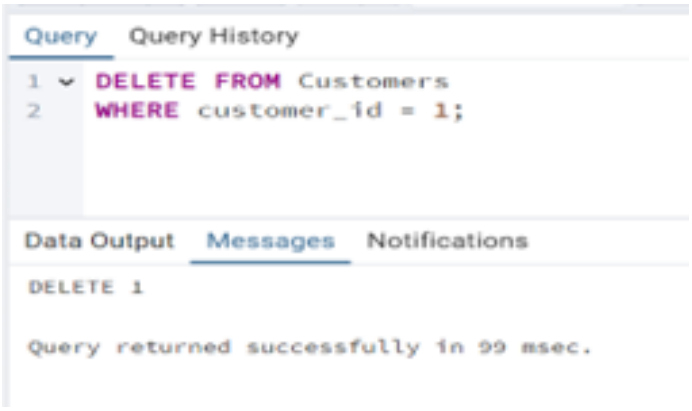

Fig. 4. Delete a Customer by ID

This SQL query deletes a record from the ‘Customers‘ table where the ‘customer_id‘ equals 1. The query executed successfully, confirming that one record was removed.

Fig. 5. Delete a User by Username

This SQL query deletes a record from the ‘Users‘ table where the ‘username‘ is ”john_doe.” The query executed successfully, removing one record.

Query Query History

```
1 UPDATE Vehicles
2 SET model = 'Toyota Corolla'
3 WHERE vehicle_id = 1;
```

Data Output Messages Notifications

```
UPDATE 0

Query returned successfully in 122 msec.
```

Fig. 6. Update a Vehicle's Model

This SQL query attempts to update the 'model' field in the 'Vehicles' table to "Toyota Corolla" for the record with 'vehicle_id' 1. However, no rows were updated, likely because no record with 'vehicle_id' 1 exists.

Query Query History

```
1 UPDATE Claims
2 SET claim_status = 'Approved'
3 WHERE claim_id = 2;
```

Data Output Messages Notifications

```
UPDATE 1

Query returned successfully in 258 msec.
```

Fig. 7. Update Claim Status

This SQL query updates the 'claim_status' field in the 'Claims' table to "Approved" for the record with 'claim_id' 2. The query executed successfully, updating one record.

Query Query History

```
1 SELECT c.customer_id, c.customer_name, u.username
2 FROM Customers c
3 JOIN Users u ON c.user_id = u.user_id;
```

Data Output Messages Notifications

	customer_id integer	customer_name character varying (100)	username character varying (50)
1	120000	Alice Smith	perrymelissa
2	2	Jenna Moreno	laurahall
3	3	Steven Myers	tanderson
4	4	Natasha Adams	johnsonmeredith
5	5	Cameron Long	barnescindy
6	6	Scott Sullivan	susanelliott
7	7	Michelle Zamora	pricerick
8	8	Heather Douglas	ikelley
9	9	Isaac Sandoval	jeffreysmith
10	10	Cheyenne Scott	galvantiffany
11	11	Raymond Lawson	rebeccameyer
12	12	Noah Norton	pamela23

Fig. 8. Select All Customers with Their Associated Users

This SQL query retrieves the 'customer_id', 'customer_name', and 'username' by performing an inner join between the 'Customers' and 'Users' tables. It matches rows where the 'user_id' in the 'Customers' table corresponds to the 'user_id' in the 'Users' table. The resulting data combines customer and user information for all matched records.

Query Query History

```
1 SELECT *
2 FROM Vehicles
3 ORDER BY vehicle_age DESC;
```

Data Output Messages Notifications

	vehicle_id [PK] integer	model character varying (50)	vehicle_age numeric	segment character varying (20)	fuel_type character varying (10)	airbags integer
1	12739	M4	20.0	C2	Diesel	6
2	15111	M8	16.4	B1	CNG	2
3	4764	M4	9.2	C2	Diesel	6
4	16030	M4	9.2	C2	Diesel	6
5	10206	M4	9.0	C2	Diesel	6
6	15136	M6	8.8	B2	Petrol	2
7	7784	M6	7.8	B2	Petrol	2
8	2506	M4	7.8	C2	Diesel	6
9	12476	M4	7.8	C2	Diesel	6
10	13553	M2	7.6	C1	Petrol	2
11	13868	M7	7.6	B2	Petrol	6
12	5682	M4	7.2	C2	Diesel	6
13	12514	M6	7.0	B2	Petrol	2
14	13489	M3	6.8	A	Petrol	2

Fig. 9. Select Vehicles Ordered by Vehicle Age

This SQL query retrieves all columns ('*') from the 'Vehicles' table and orders the records by 'vehicle_age' in descending order ('DESC'). The result displays vehicles sorted from the oldest to the newest based on their age.

Query Query History

```
SELECT claim_status, COUNT(*) AS total_claims
FROM Claims
GROUP BY claim_status;
```

Data Output Messages Notifications

	claim_status character varying (20)	total_claims bigint
1	0	18349
2	Approved	1
3	1	1254

Fig. 10. Select Total Claims Grouped by Claim Status

This SQL query counts the total number of claims for each 'claim_status' in the 'Claims' table. It groups the data by 'claim_status' and provides the count of claims for each status in the 'total_claims' column. The result shows the number of claims for each unique status.

Query Query History

```
SELECT c.customer_name, cl.claim_amount
FROM Customers c
JOIN Claims cl ON c.customer_id = cl.customer_id
WHERE cl.claim_amount > 1000;
```

Data Output Messages Notifications

	customer_name character varying (100)	claim_amount numeric (10,2)
1	Steven Myers	50755.00
2	Natasha Adams	65814.00
3	Scott Sullivan	2497.00
4	Michelle Zamora	25528.00
5	Heather Douglas	96743.00
6	Isaac Sandoval	37332.00
7	Cheyenne Scott	67133.00
8	Raymond Lawson	34407.00
9	Noah Norton	70519.00
10	David Nichols	66130.00
11	Raven Mayer	39005.00

Fig. 11. Select Customers with Claims Greater than a Specific Amount

This SQL query retrieves the 'customer_name' and 'claim_amount' for customers whose claim amounts exceed 1000. It performs an inner join between the 'Customers' and 'Claims' tables, matching rows where 'customer_id' in the 'Customers' table corresponds to 'customer_id' in the 'Claims' table. The result shows customers with claims greater than 1000.

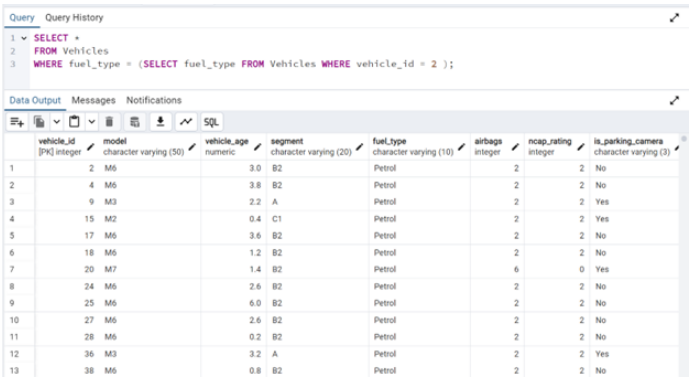

Fig. 12. Select Vehicles with a Specific Fuel Type Using a Subquery

This SQL query selects all columns ('*') from the 'Vehicles' table where the 'fuel_type' matches the 'fuel_type' of the vehicle with 'vehicle_id = 2'. It uses a subquery to retrieve the 'fuel_type' of the specific vehicle ('vehicle_id = 2') and filters the results based on this value. The output displays vehicles with the same 'fuel_type'.

A. PROBLEMATIC QUERIES

Query Query History

```
SELECT *
FROM Claims
WHERE customer_id = 123;
```

Data Output Messages Notifications

```
Successfully run. Total query runtime: 203 msec.
1 rows affected.
```

Fig. 13. Retrieve all claims for a specific customer

This SQL query retrieves all columns ('*') from the 'Claims' table where the 'customer_id' is 123. The query successfully ran, affecting one row in the result.

Query Query History

```
SELECT *
FROM Vehicles
WHERE customer_id = 123;
```

Data Output Messages Notifications

```
Successfully run. Total query runtime: 146 msec.
1 rows affected.
```

Fig. 14. Get all vehicles for a specific customer

This SQL query retrieves all columns (‘*‘) from the ‘Vehicles‘ table where the ‘customer_id‘ is 123. The query successfully ran, affecting one row in the result.

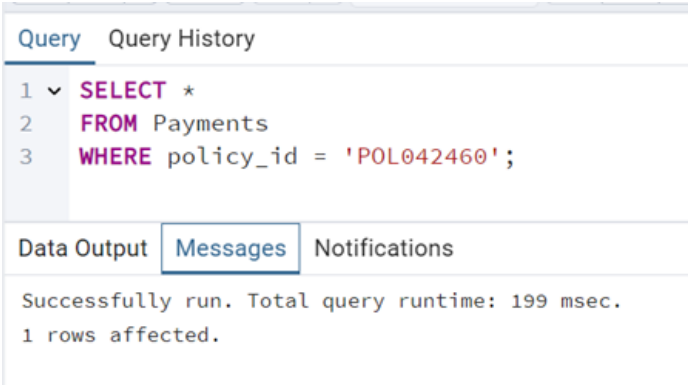

Fig. 15. Find all payments made for a specific policy

This SQL query retrieves all columns (‘*‘) from the ‘Payments‘ table where the ‘policy_id‘ is ’POL042460’. The query was successfully executed, returning one row in the result.

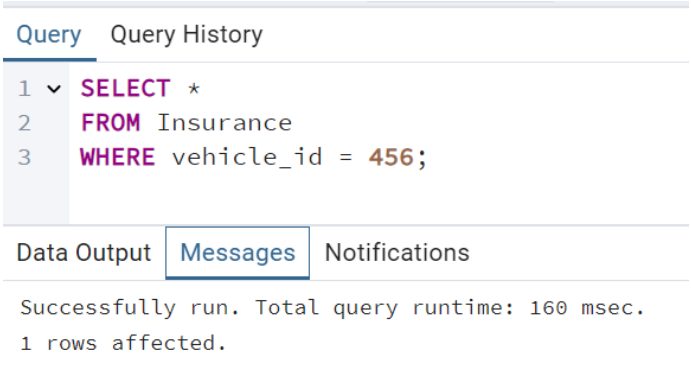

Fig. 16. Retrieve all insurance policies for a specific vehicle

This SQL query retrieves all columns (‘*‘) from the ‘Insurance‘ table where the ‘vehicle_id‘ is 456. The query executed successfully, returning one row in the result.

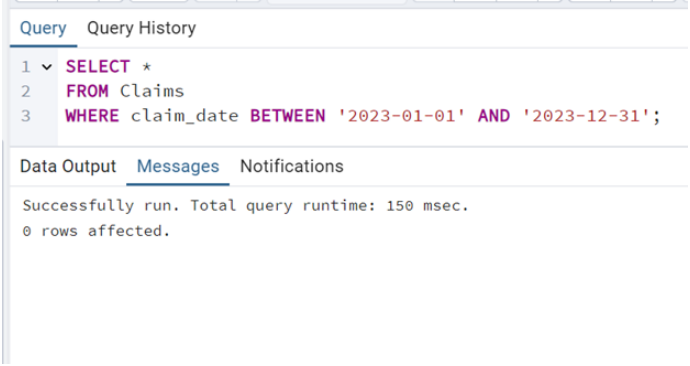

Fig. 17. Retrieve all claims filed within a specific date range

This SQL query retrieves all columns (‘*‘) from the ‘Claims‘ table where the ‘claim_date‘ falls between January 1, 2023, and December 31, 2023. The query executed successfully but returned no rows, indicating no records matched the specified date range.

VI. QUERY OPTIMIZATION

The queries create indexes on specific columns to optimize the performance of SELECT operations. Indexes improve the query runtime by enabling faster lookups for specific conditions. Here’s what each query achieves

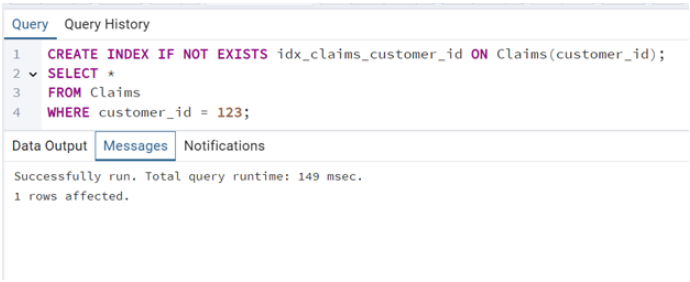

Fig. 18. Retrieve all claims for a specific customer after indexing

An index was created on customer_id, significantly enhancing the search efficiency for claims linked to a specific customer.

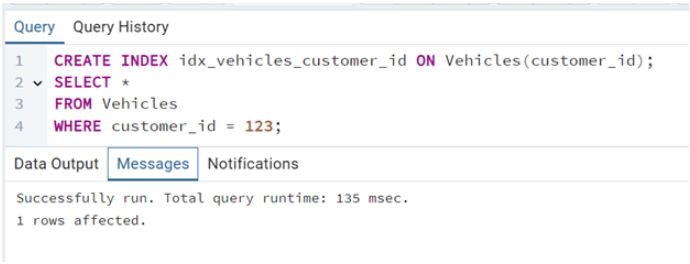

Fig. 19. Get all vehicles for a specific customer after indexing

An index on customer_id allows for quick retrieval of vehicles associated with a specific customer.

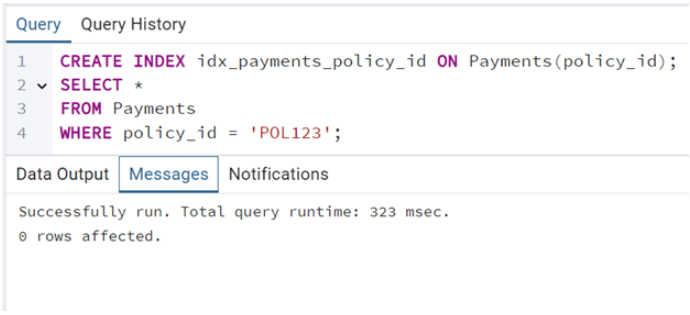

Fig. 20. Find all payments made for a specific policy after indexing

An index on policy_id improves the search for payment records by policy ID, although in this case, no matching rows were found.

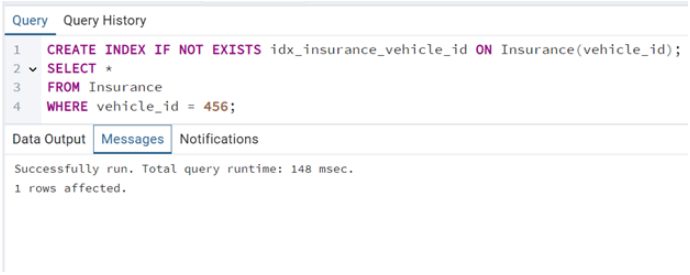

Fig. 21. Retrieve all insurance policies for a specific vehicle after indexing

An index on vehicle_id speeds up the query for insurance details associated with a particular vehicle.

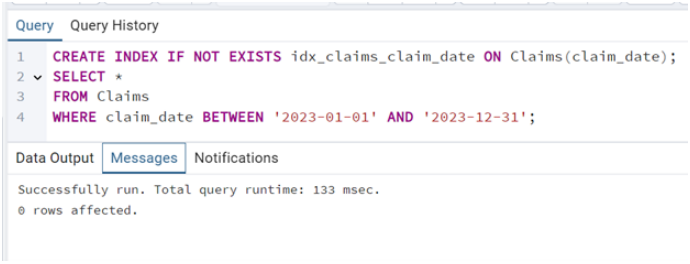

Fig. 22. Retrieve all claims filed within a specific date range after indexing

An index on claim_date accelerates filtering operations for claims within a specific date range, even when no rows are matched.

VII. INDEXING

A. Problems Faced Before Indexing

- Searching for users by username in the Users table was slow, especially with a large number of records, requiring full table scans.
- Queries that filtered customers by name in the Customers table took a long time to execute due to the lack of an index, leading to performance bottlenecks.
- Retrieving vehicles based on their model in the Vehicles table was inefficient, resulting in slow response times for queries.
- Accessing insurance policies by policy_id in the Insurance table was cumbersome, requiring full table scans that slowed down the application.
- Filtering claims based on status in the Claims table was slow, especially when there were many claims, leading to long wait times for users.
- Searching for agents by email in the Agents table was slow, causing delays in retrieving agent details.
- Queries that filtered payments by status in the Payments table were inefficient, resulting in longer processing times for reports.

B. Solutions Provided by Indexing

- Indexing the username column in the Users table allows for faster lookups, enabling direct access to records without scanning the entire table.
- An index on the customer_name column in the Customers table significantly speeds up searches, reducing query execution time.
- Indexing the model column in the Vehicles table allows for efficient retrieval of vehicles by model, improving response times.
- An index on the policy_id column in the Insurance table enhances the speed of policy lookups, allowing for quick access to relevant records.
- Indexing the claim_status column in the Claims table accelerates filtering operations, making it easier to retrieve claims based on their status.
- An index on the email column in the Agents table enables quick searches for agents, improving the efficiency of agent-related queries.
- Indexing the payment_status column in the Payments table enhances the performance of queries filtering payments by status.

C. Additional Benefits of Indexing Over Normalized Tables

- Indexes minimize the need for full table scans, which reduces the computational load on the database server, especially during complex queries.
- Indexes allow for the optimization of common access patterns, such as frequently executed queries that involve filtering by specific columns, leading to improved performance.
- As the database grows with more records, indexes help maintain performance without requiring significant changes to the underlying database architecture.
- Indexes improve the efficiency of join operations between tables, as the database can quickly locate matching records based on indexed columns.
- Overall, indexing significantly enhances the performance of queries by reducing the time complexity of data retrieval operations, leading to a better user experience.

By implementing these indexing strategies, the overall efficiency and performance of the vehicle insurance database can be greatly improved, addressing the initial problems faced before indexing.

VIII. BONUS TASK

The website, implemented using `Streamlit`, is designed to interact with a PostgreSQL database, enabling users to execute SQL queries and dynamically visualize results in real-time. The key features include an intuitive interface for custom query input, dynamic data visualization (e.g., tables, bar charts, line graphs), error handling for invalid queries, and predefined query options. Development involves securely connecting to the database, creating a user-friendly frontend, integrating backend logic for query execution, and rendering visual outputs. The app will be tested with various queries for stability and accuracy before deployment to a cloud platform like Streamlit Cloud or Heroku. The deliverables include a fully functional web application and accompanying documentation.

CONCLUSION

The project successfully implements a comprehensive and optimized database system tailored to address the challenges faced by the vehicle insurance industry in managing extensive data related to users, customers, vehicles, insurance policies, claims, agents, payments, and police records. The system ensures data integrity, reduces redundancy, and enhances the efficiency of data retrieval and management, enabling stakeholders—insurance companies, agents, and customers—to make informed decisions and provide quality service. The addition of new tables, extends the database's functionality to cover a broader range of operations, ensuring scalability and adaptability to real-world needs.

Additionally, the bonus task of developing a user-friendly web application using Streamlit, currently in progress, will further benefit users by providing real-time access to critical data and dynamic visualization of query results, improving usability and operational workflows. By centralizing and structuring data management, this project directly addresses the inefficiencies in the vehicle insurance industry, enabling faster data access, enhanced accuracy, and better customer satisfaction, ultimately contributing to more informed decision-making and streamlined operations for all stakeholders.

REFERENCES

[1] Kaggle dataset: https://www.kaggle.com/datasets/litvinenko630/insurance-claims
[2] https://pypi.org/project/Faker/0.7.4/
[3] ER Diagram: https://dbdiagram.io/home