Understanding Airline Passenger Satisfaction

1st Pavan Pajjuri
Engineering Science Data Science
University at Buffalo
Buffalo, USA
spajjuri@buffalo.edu

2nd Sai Krishna Valdas
Engineering Science Data Science
University at Buffalo
Buffalo, USA
svaldas@buffalo.edu

***Abstract*—The airline industry faces significant challenges in understanding passenger satisfaction and improving overall service quality. With various service aspects contributing to passenger experiences, it becomes crucial to analyze and track satisfaction levels effectively. This project aims to develop a comprehensive relational database that captures detailed information about passengers, flights, service ratings, and delays. The database system will enable stakeholders to gain insights into the factors influencing passenger satisfaction and make data-driven decisions to enhance service offerings. By leveraging this database, complex querying and reporting can be performed, supporting detailed analysis that surpasses the capabilities of traditional spreadsheet methods. The system's design incorporates key features such as primary and foreign keys, normalization, and constraints to ensure data integrity, scalability, and performance, ultimately helping airlines improve their service quality.**

***Index Terms*—Airline Industry, Passenger Satisfaction, Service Quality, Database Design, Relational Database, Data Integrity, Query Optimization, Data Analysis, Reporting, Data-Driven Decisions, Database Constraints, Normalization, Complex Queries.**

I. Introduction

A relational database is essential for managing the growing volume of passenger and flight data in the airline industry. Unlike Excel, which struggles with performance issues as data size increases, a database offers scalability and efficient data storage, ensuring long-term growth. It can handle large datasets with ease, supporting airlines as they expand and accumulate more passenger information.

Databases maintain data integrity through constraints like primary keys, foreign keys, and unique constraints, which prevent issues like orphan records. Unlike Excel, where users can accidentally introduce errors, databases ensure consistency and reliability. Additionally, they support concurrent access, allowing multiple users to modify data simultaneously without risk of corruption, making them more suitable for real-time operational environments.

With SQL, databases can execute complex queries and generate detailed reports, providing deeper insights into passenger satisfaction and service performance. Unlike Excel, which requires manual manipulation, SQL queries are more accurate, flexible, and efficient, enabling data-driven decision-making. These capabilities enhance analysis and reporting, which is crucial for optimizing service quality and operational efficiency.

Databases also offer robust security features, including access control, which ensures that only authorized users can modify sensitive passenger data. This is critical in industries like aviation where data protection is paramount. Additionally, automated updates, stored procedures, and triggers streamline data management, while advanced backup and recovery solutions ensure data integrity and business continuity, further enhancing operational efficiency.

A. Target Users

The target users for the Airline Passenger Satisfaction database project include:

1) **Airline Management:** Executives and managers who need access to aggregated passenger satisfaction data to make strategic decisions about service improvements, operational efficiency, and marketing strategies.
2) **Customer Service Teams:** Staff responsible for addressing passenger complaints and inquiries. They would utilize the database to analyze customer feedback and identify common issues that require attention.
3) **Marketing Professionals:** Marketers who can analyze passenger demographics and satisfaction ratings to tailor promotional campaigns and enhance customer loyalty programs.
4) **Data Analysts:** Individuals tasked with extracting insights from the database. They will perform queries to assess service ratings and identify trends, helping to shape future airline services.

B. Database

The project uses Airplane passengers data from Kaggle Dataset from here and combines it with customer satisfaction dataset available in kaggle from here Using this data we have divided the data into corresponding tables. The five tables taken as CSV files are:

1) `passengers.csv`
2) `flights.csv`
3) `satisfaction_ratings.csv`
4) `delays.csv`
5) `overall_satisfaction.csv`

C. Initial Schema

We outline the database schema for the Airline Passenger Satisfaction project here. The schema consists of five tables:

Passengers, Flights, Satisfaction Ratings, Delays, and Overall Satisfaction. Below is a description of the primary keys, foreign keys, and columns for each table.

Primary Key and Foreign Keys

Table Name	Primary Key	Foreign Keys	Justification
Passengers	PassengerID	None	Unique identifier for each passenger.
Flights	FlightID	None	Unique identifier for each flight.
Satisfaction Ratings	RatingID	PassengerID, FlightID	Each rating is unique and relates to a specific passenger and flight.
Delays	DelayID	FlightID	Each delay record is unique and associated with a specific flight.
Overall Satisfaction	SatisfactionID	PassengerID	Each record of overall satisfaction is linked to a specific passenger.

Fig. 1. Primary Key and Foreign Key.

The database schema for the Airline Passenger Satisfaction project, along with a brief description of each column in the five tables:

***1. Passengers**: Columns and Description*

PassengerID
Data Type: SERIAL
Description: Unique identifier for each passenger (Primary Key).

Gender
Data Type: VARCHAR(10)
Description: Gender of the passenger (e.g., Male, Female).

CustomerType
Data Type: VARCHAR(50)
Description: Type of customer (e.g., Loyal Customer, Disloyal Customer).

Age
Data Type: INTEGER
Description: Actual age of the passenger (must be non-negative).

Defaults and Nulls:

- PassengerID: No default, auto-incremented, cannot be NULL.
- Gender: Default 'Not Specified', can be NULL.
- CustomerType: Default 'Disloyal Customer', can be NULL.
- Age: No default, cannot be NULL.

***2. Flights**: Columns and Description*

Column Name	Data Type	Description
PassengerID	SERIAL	Unique identifier for each passenger (Primary Key).
Gender	VARCHAR(10)	Gender of the passenger (e.g., Male, Female).
CustomerType	VARCHAR(50)	Type of customer (e.g., Loyal Customer, Disloyal Customer).
Age	INTEGER	Actual age of the passenger (must be non-negative).

Fig. 2. Image related to the Passengers table.

FlightID
Data Type: INTEGER
Description: Unique identifier for each flight (Primary Key).

FlightDistance
Data Type: FLOAT
Description: Distance of the flight in miles or kilometers.

TypeOfTravel
Data Type: VARCHAR(50)
Description: Purpose of the flight (e.g., Personal Travel, Business Travel).

Class
Data Type: VARCHAR(20)
Description: Travel class (e.g., Business, Eco, Eco Plus).

Defaults and Nulls:

- FlightID: No default, cannot be NULL.
- FlightDistance: Default 0.0, cannot be NULL.
- TypeOfTravel: Default 'Personal Travel', can be NULL.
- Class: Default 'Economy', can be NULL.

Column Name	Data Type	Description
FlightID	INTEGER	Unique identifier for each flight (Primary Key).
FlightDistance	FLOAT	Distance of the flight in miles or kilometers.
TypeOfTravel	VARCHAR(50)	Purpose of the flight (e.g., Personal Travel, Business Travel).
Class	VARCHAR(20)	Travel class (e.g., Business, Eco, Eco Plus).

Fig. 3. Image related to the Flights table.

***3. Satisfaction Ratings**: Columns and Description*

RatingID
Data Type: SERIAL

Description: Unique identifier for each satisfaction rating (Primary Key).

PassengerID
Data Type: INTEGER
Description: Identifier for the passenger providing the rating (Foreign Key referencing Passengers).

FlightID
Data Type: INTEGER
Description: Identifier for the flight being rated (Foreign Key referencing Flights).

InflightWifiService
Data Type: INTEGER
Description: Satisfaction level for inflight Wi-Fi service (0: Not Applicable, 1-5: Ratings).

SeatComfort
Data Type: INTEGER
Description: Satisfaction level for seat comfort (1-5: Ratings).

EaseOfOnlineBooking
Data Type: INTEGER
Description: Satisfaction level for ease of online booking (1-5: Ratings).

FoodAndDrink
Data Type: INTEGER
Description: Satisfaction level for food and drink (1-5: Ratings).

InflightEntertainment
Data Type: INTEGER
Description: Satisfaction level for inflight entertainment (1-5: Ratings).

OnboardService
Data Type: INTEGER
Description: Satisfaction level for onboard service (1-5: Ratings).

GateLocation
Data Type: INTEGER
Description: Satisfaction level for gate location (1-5: Ratings).

LegRoomService
Data Type: INTEGER
Description: Satisfaction level for legroom service (1-5: Ratings).

Cleanliness
Data Type: INTEGER
Description: Satisfaction level for cleanliness (1-5: Ratings).

Defaults and Nulls:

- RatingID: No default, auto-incremented, cannot be NULL.
- PassengerID: No default, cannot be NULL.
- FlightID: No default, cannot be NULL.
- All satisfaction columns (1-5): No default, cannot be NULL.

Column Name	Data Type	Description
RatingID	SERIAL	Unique identifier for each satisfaction rating (Primary Key).
PassengerID	INTEGER	Identifier for the passenger providing the rating (Foreign Key referencing Passengers).
FlightID	INTEGER	Identifier for the flight being rated (Foreign Key referencing Flights).
InflightWifiService	INTEGER	Satisfaction level for inflight Wi-Fi service (0: Not Applicable, 1-5: Ratings).
SeatComfort	INTEGER	Satisfaction level for seat comfort (1-5: Ratings).
EaseOfOnlineBooking	INTEGER	Satisfaction level for ease of online booking (1-5: Ratings).
FoodAndDrink	INTEGER	Satisfaction level for food and drink (1-5: Ratings).
InflightEntertainment	INTEGER	Satisfaction level for inflight entertainment (1-5: Ratings).
OnboardService	INTEGER	Satisfaction level for onboard service (1-5: Ratings).
GateLocation	INTEGER	Satisfaction level for gate location (1-5: Ratings).
LegRoomService	INTEGER	Satisfaction level for legroom service (1-5: Ratings).
Cleanliness	INTEGER	Satisfaction level for cleanliness (1-5: Ratings).

Fig. 4. Image related to the Satisfaction Ratings table.

***4. Delays**: Columns and Description*

DelayID
Data Type: SERIAL
Description: Unique identifier for each delay record (Primary Key).

FlightID
Data Type: INTEGER
Description: Identifier for the flight with delays (Foreign Key referencing Flights).

DepartureDelayInMinutes
Data Type: FLOAT
Description: Minutes delayed during departure.

ArrivalDelayInMinutes
Data Type: FLOAT
Description: Minutes delayed upon arrival.

Defaults and Nulls:

- DelayID: No default, auto-incremented, cannot be NULL.
- FlightID: No default, cannot be NULL.
- DepartureDelayInMinutes: Default 0, can be NULL.
- ArrivalDelayInMinutes: Default 0, can be NULL.

Column Name	Data Type	Description
DelayID	SERIAL	Unique identifier for each delay record (Primary Key).
FlightID	INTEGER	Identifier for the flight with delays (Foreign Key referencing Flights).
DepartureDelayInMinutes	FLOAT	Minutes delayed during departure.
ArrivalDelayInMinutes	FLOAT	Minutes delayed upon arrival.

Fig. 5. Image related to the Delays table.

***5. Overall Satisfaction**: Columns and Description*

SatisfactionID
Data Type: SERIAL
Description: Unique identifier for each satisfaction record (Primary Key).

PassengerID
Data Type: INTEGER
Description: Identifier for the passenger whose satisfaction is recorded (Foreign Key referencing Passengers).

Satisfaction
Data Type: VARCHAR(40)
Description: Overall satisfaction level (e.g., Satisfaction, Neutral, Dissatisfaction).

Defaults and Nulls:

- SatisfactionID: No default, auto-incremented, cannot be NULL.
- PassengerID: No default, cannot be NULL.
- Satisfaction: Default 'Neutral', can be NULL.

D. Data Loading and Transformation process

***Data Loading**:*

- *Used 'COPY' commands to load data into PostgreSQL tables.*
- *Data loaded from CSV files into tables such as 'Passengers', 'Flights', 'Satisfaction Ratings', etc.*

Column Name	Data Type	Description
SatisfactionID	SERIAL	Unique identifier for each satisfaction record (Primary Key).
PassengerID	INTEGER	Identifier for the passenger whose satisfaction is recorded (Foreign Key referencing Passengers).
Satisfaction	VARCHAR(40)	Overall satisfaction level (e.g., Satisfaction, Neutral, Dissatisfaction).

Fig. 6. Image related to the Overall Satisfaction table.

- *Example of 'COPY' command used for the 'Passengers' table:*

```
COPY passengers (PassengerID, Gender, Custo
FROM '/path/to/passenger_data.csv'
DELIMITER ',' CSV HEADER;
```

- *Each table required a separate 'COPY' command based on its schema and column mapping.*

***Transformations during Data Loading**:*

- *Data Cleaning: Ensured data consistency by cleaning outliers and missing values.*
- *Handling Missing Values:*
  - *Set default values for missing data (e.g., 'Not Specified' for 'Gender', 'Disloyal Customer' for 'CustomerType').*
  - *Used 'NULL' for missing values where applicable (e.g., for 'Age' when not provided).*
- *Data Mapping: Mapped columns from CSV files to the appropriate database table columns (e.g., 'PassengerID', 'Gender', 'Age').*
- *Column Type Conversion: Ensured the data types from the CSV matched the database schema (e.g., 'INTEGER', 'VARCHAR').*
- *Referential Integrity: Ensured foreign keys in tables like 'Satisfaction Ratings' were mapped correctly to the referenced tables ('Passengers', 'Flights').*
- *Normalization: Adjusted the data format to comply with the relational database structure (e.g., ensuring all text is standardized).*

***Challenges Faced**:*

- *Missing Data:*
  - *Some rows in CSV files had missing values which required handling with default values or 'NULL'.*
  - *Columns like 'Gender', 'CustomerType', and 'Age' required extra care during data preprocessing to avoid errors.*
- *Data Integrity:*
  - *Referential integrity issues were encountered when foreign key values in 'Satisfaction Ratings' didn't match values in the 'Passengers' or 'Flights' tables.*

  - *Rows without valid foreign keys would fail to load, requiring sequential loading of referenced tables first.*
- *Large File Size:*
  - *Handling large CSV files required splitting them into smaller chunks or using batch processing to improve performance.*
- *Data Type Mismatches:*
  - *Encountered mismatches where a column was expected to contain 'INTEGER' values but had text or invalid characters, requiring additional preprocessing steps.*
- *Error Handling:*
  - *Errors during the 'COPY' process (e.g., missing columns or invalid data types) required review and adjustment of the 'COPY' commands, especially to handle invalid rows.*

II. Database Schema Refinement

All tables within the schema are in **Boyce-Codd Normal Form (BCNF)**, and follow the functional dependencies identified. Further, there is no need for decomposition of any tables.

A. Transformations from Milestone 1:

- **Normalization:** We followed Database Design Rules to ensure that each table adhered to **Boyce-Codd Normal Form (BCNF)**. This reduces redundancy and prevents update anomalies.
- **Data Type Refinement:** Certain fields required refinement for more efficient storage and querying. For instance, columns like 'Age' in the 'Passengers' table were ensured to have the appropriate data type (e.g., INTEGER) for accurate calculations and storage. Similarly, 'FlightDistance' in the 'Flights' table used a 'FLOAT' type for precise representation of distance.
- **Linking Tables Creation:** To accurately depict many-to-many relationships in the database, such as those between passengers and flights or satisfaction ratings, we created linking tables. The 'Satisfaction Ratings' table links 'PassengerID' and 'FlightID', tracking satisfaction scores for individual flights. Similarly, the 'Delays' table connects 'FlightID' with delay information, ensuring detailed records of each delay per flight.
- **Primary and Foreign Keys:** In database management, establishing relationships between tables is crucial for maintaining data integrity and ensuring consistency across the schema. This is achieved through the use of primary and foreign key constraints. Primary keys, such as 'PassengerID' in the 'Passengers' table and 'FlightID' in the 'Flights' table, uniquely identify each record within their respective tables. Foreign keys create essential linkages between tables; for example, 'PassengerID' in the 'Satisfaction Ratings' table references the 'Passengers' table, and 'FlightID' in the 'Satisfaction Ratings' table references the 'Flights' table. These constraints help ensure that data remains accurate and coherent, facilitating easier management and retrieval of information.

B. Constraints and Checks Implemented:

- **Primary Key Constraints:**
  - *Passengers: PassengerID serves as the primary key to uniquely identify each passenger.*
  - *Flights: FlightID is the unique identifier for each flight.*
  - *Satisfaction Ratings: RatingID uniquely identifies each satisfaction rating.*
  - *Delays: DelayID uniquely identifies each delay record.*
  - *Overall Satisfaction: SatisfactionID uniquely identifies each overall satisfaction record.*
  - *Other tables have their own primary keys ensuring uniqueness within their records.*
- **Foreign Key Constraints:**
  - *Satisfaction Ratings and Passengers:*
    - *PassengerID references Passengers(PassengerID) maintaining valid passenger associations.*
  - *Satisfaction Ratings and Flights:*
    - *FlightID references Flights(FlightID) maintaining valid flight associations.*
  - *Delays and Flights:*
    - *FlightID references Flights(FlightID) maintaining valid flight associations.*
- **ON DELETE SET NULL:**
  - *In Satisfaction Ratings, if a passenger is deleted, their PassengerID turns to NULL in this table.*
  - *In Satisfaction Ratings, if a flight is deleted, its FlightID turns to NULL in this table.*
  - *Prevents broken links by not having references to deleted passengers or flights. This helps manage data integrity gracefully.*
- **ON UPDATE CASCADE:**
  - *If a PassengerID is changed in Passengers, it updates automatically in Satisfaction Ratings.*
  - *If a FlightID is changed in Flights, it updates automatically in Satisfaction Ratings and Delays.*
  - *Ensures all relationships are up-to-date and accurate even after changes.*
- **Data Type Constraints:**
  - *Numerics: Age in Passengers is appropriately defined as an INTEGER for accurate representation.*
  - *Numerics: FlightDistance in Flights is appropriately defined as a FLOAT for precise representation of distance.*
- **Unique Constraints:**
  - *These are implicit through primary keys on each table, ensuring no duplicate records.*
- **CHECK Constraints:**
  - *Satisfaction Ratings: The rating fields (e.g., 'SeatComfort', 'InflightWifiService') in the 'Satisfaction*

TABLE I
DATABASE TABLES WITH FUNCTIONAL DEPENDENCIES AND BCNF CHECK

Table Name	Primary Key	Functional Dependencies (FD's)	BCNF Check
Passengers	PassengerID	PassengerID → Gender, CustomerType, Age	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Flights	FlightID	FlightID → FlightDistance, TypeOfTravel, Class	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Satisfaction Ratings	RatingID	RatingID → PassengerID, FlightID, InflightWifiService, SeatComfort, EaseOfOnlineBooking, FoodAndDrink, InflightEntertainment, OnboardService, GateLocation, LegRoomService, Cleanliness	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Delays	DelayID	DelayID → FlightID, DepartureDelayInMinutes, ArrivalDelayInMinutes	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.
Overall Satisfaction	SatisfactionID	SatisfactionID → PassengerID, Satisfaction	This table is in BCNF, as each determinant is a superkey and functionally determines all other attributes in the relation.

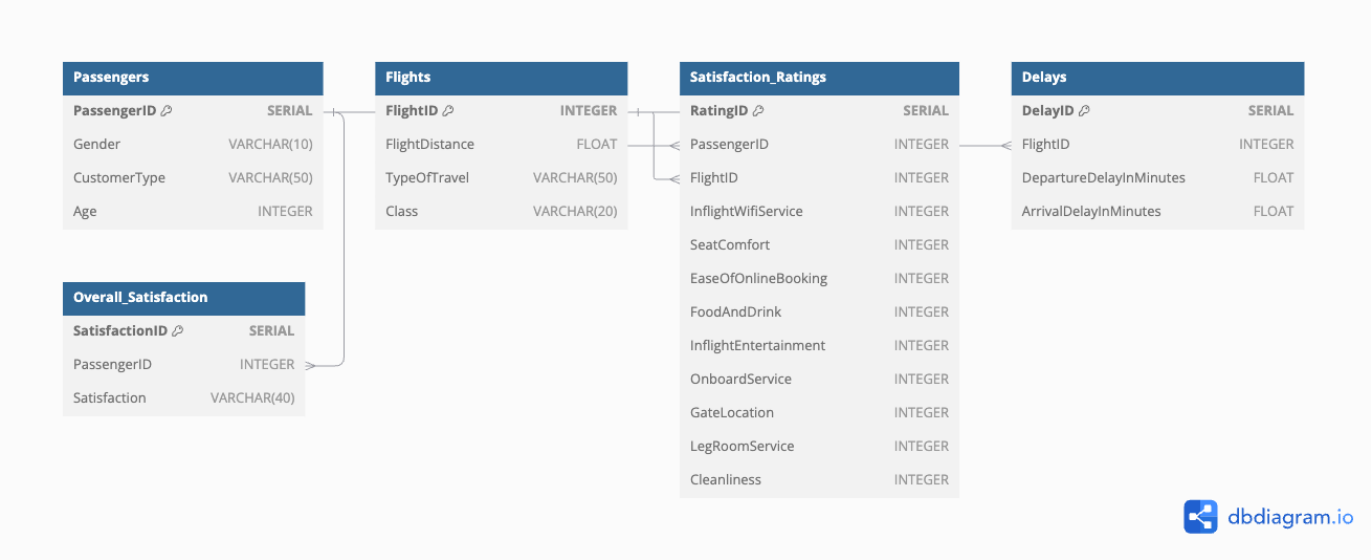

Fig. 7. This is a figure spanning both columns in the document.

Ratings' table are constrained to values between 1 and 5. This ensures that only valid ratings are entered into the database. Example:

```
CHECK (SeatComfort >= 1 AND \
       SeatComfort <= 5)
```

– *Age in Passengers: The 'Age' field is constrained to be greater than or equal to 0, as negative ages are invalid. Example:*

```
CHECK (Age >= 0)
```

– *FlightDistance in Flights: The 'FlightDistance' field in the 'Flights' table is constrained to be a positive number, as a flight distance cannot be negative. Example:*

```
CHECK (FlightDistance >= 0)
```

HANDLING LARGE DATASETS

When dealing with large datasets, particularly in the context of the 'Passengers', 'Flights', 'Satisfaction Ratings', and 'Delays' tables, we encountered query slowdowns due to the large number of records and the complexity of the joins between tables. A typical query to retrieve passengers along with their flight and satisfaction ratings might experience performance

degradation due to the large size of the dataset and the need to perform multiple joins, which can be computationally expensive. For example, retrieving passenger satisfaction ratings for flights that meet specific criteria, such as flights departing from a particular airport or flights with high delay durations, can lead to long query execution times.

To address these performance issues, we implemented indexing strategies to optimize query execution. We added indexes on frequently queried columns, such as ‘PassengerID‘ and ‘FlightID‘, to speed up join operations between the ‘Passengers‘, ‘Satisfaction Ratings‘, and ‘Flights‘ tables. By indexing these columns, we reduced the time required for the database to find and retrieve the necessary data for complex queries. In addition to individual indexes, we also considered using composite indexes to cover multiple columns involved in the joins. This further enhanced performance, especially for queries with complex filter conditions. Overall, the use of indexing, coupled with multi-core processing capabilities, significantly improved query speeds, ensuring that data retrieval from the large dataset was more efficient and responsive.

PROBLEMATIC QUERIES

A. *Select with Join*

Query | Query History

```
SELECT
    p.PassengerID,
    p.CustomerType,
    sr.SeatComfort,
    sr.InflightWifiService,
    f.FlightID,
    f.Class
FROM
    Passengers p
JOIN
    Satisfaction_Ratings sr ON p.PassengerID = sr.PassengerID
JOIN
    Flights f ON sr.FlightID = f.FlightID
WHERE
    f.Class = 'Business' AND p.CustomerType = 'Loyal Customer';
```

Data Output | Messages | Explain | Notifications

	passengerid integer	customertype character varying (50)	seatcomfort integer	inflightwifiservice integer	flightid integer	class character varying (20)
1	2	Loyal Customer	5	1	5619	Business
2	4	Loyal Customer	4	0	6066	Business
3	7	Loyal Customer	5	5	6595	Business
4	8	Loyal Customer	5	2	5279	Business
5	10	Loyal Customer	4	2	4273	Business
6	12	Loyal Customer	4	2	839	Business
7	13	Loyal Customer	5	5	3021	Business
8	14	Loyal Customer	4	1	1310	Business
9	15	Loyal Customer	4	2	2773	Business
10	20	Loyal Customer	4	5	6428	Business
11	21	Loyal Customer	5	4	1848	Business
12	24	Loyal Customer	4	4	3794	Business
13	27	Loyal Customer	2	4	6320	Business

Fig. 8. Query Result: Retrieving Business Class Loyal Customer Satisfaction Ratings

The above query is designed to retrieve satisfaction ratings for passengers who are classified as ”Loyal Customer” and who have traveled in ”Business” class. The query involves joining the ‘Passengers‘, ‘Satisfaction Ratings‘, and ‘Flights‘ tables on multiple columns (‘PassengerID‘ and ‘FlightID‘), while filtering by specific conditions (‘Class‘ and ‘CustomerType‘). This query can be problematic due to the large dataset, as it involves a sequence of joins across tables with potentially millions of rows, which leads to high query costs and slowdowns.

The subsequent figures provide a graphical explanation of how the query is processed, and we can observe that the system performs multiple hash inner joins followed by sequential scans of large tables. These factors contribute to the excessive cost.

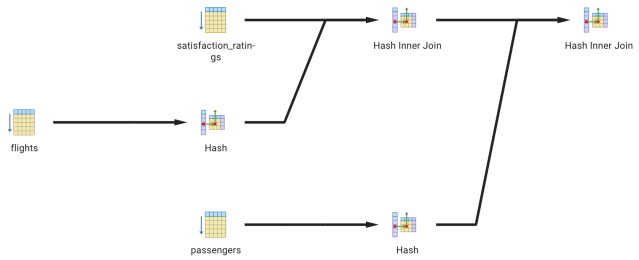

Fig. 9. Graphical Explanation of Query Execution Plan

This execution plan depicts the steps involved in processing the query. The system first performs a hash inner join between the ‘Satisfaction Ratings‘ and ‘Passengers‘ tables, followed by another hash inner join with the ‘Flights‘ table. The system uses a hash-based approach for these joins, which can be inefficient when dealing with large datasets. Although hash joins are often faster than nested loops for large datasets, they still require substantial memory and computational resources when the involved tables are large.

#	Node
1.	→ Hash Inner Join **Hash Cond**: (sr.passengerid = p.passengerid)
2.	→ Hash Inner Join **Hash Cond**: (sr.flightid = f.flightid)
3.	→ Seq Scan on satisfaction_ratings as sr
4.	→ Hash
5.	→ Seq Scan on flights as f **Filter**: ((class)::text = 'Business'::text)
6.	→ Hash
7.	→ Seq Scan on passengers as p **Filter**: ((customertype)::text = 'Loyal Customer'::text)

Fig. 10. Hash Level Analysis of Query Execution

In the hash level analysis (Figure 10), we can observe that the query execution relies on multiple hash joins, which contribute to performance issues. Each hash operation requires loading the entire dataset into memory, which increases the computational load and can lead to slow execution times,

especially when the ‘Flights‘, ‘Satisfaction Ratings‘, and ‘Passengers‘ tables are large. Additionally, the sequential scans (as seen in the plan) further contribute to the query’s inefficiency, as it has to scan through entire tables multiple times. As a result, the overall execution time increases dramatically.

By indexing the frequently queried columns (such as ‘PassengerID‘ and ‘FlightID‘), we can reduce the reliance on full-table scans and hash joins, thereby improving query performance significantly. These improvements would help reduce the processing time, optimize resource usage, and speed up data retrieval.

B. Select with GroupBy and OrderBy

Query | Query History

```
1  SELECT
2      f.FlightID,
3      AVG(sr.SeatComfort) AS AvgSeatComfort
4  FROM
5      Flights f
6  JOIN
7      Satisfaction_Ratings sr ON f.FlightID = sr.FlightID
8  GROUP BY
9      f.FlightID
10 ORDER BY
11     AvgSeatComfort DESC;
12
```

Data Output | Messages | Explain | Notifications

	flightid [PK] integer	avgseatcomfort numeric
1	986	5.0000000000000000
2	5549	5.0000000000000000
3	4489	5.0000000000000000
4	4866	5.0000000000000000
5	1924	5.0000000000000000
6	6502	5.0000000000000000
7	3437	5.0000000000000000
8	6542	5.0000000000000000
9	676	5.0000000000000000
10	2814	5.0000000000000000
11	2577	5.0000000000000000
12	2277	5.0000000000000000

Fig. 11. The query executed to fetch the average seat comfort for each flight, ordered by the average seat comfort.

The SQL query presented here calculates the average seat comfort for each flight using the ‘AVG()‘ function and joins the ‘Flights‘ and ‘Satisfaction Ratings‘ tables. It groups the results by ‘FlightID‘ and orders them by the average seat comfort in descending order. In this query, the filtering and grouping operations, along with sorting, cause performance issues when dealing with large datasets.

Graphical Explanation: As shown in the execution plan, the query performs multiple sequential scans of the ‘Flights‘ and ‘Satisfaction Ratings‘ tables, and a hash join is used to combine the data. A sort operation is then applied after the aggregate (AVG) operation. The graphical explanation helps

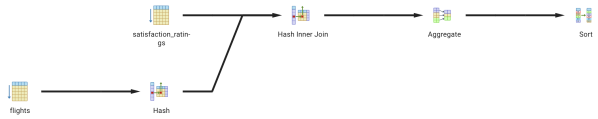

Fig. 12. Graphical execution plan for the problematic query.

visualize the complex operations involved in this query, which can lead to performance bottlenecks, especially with large datasets.

	#	Node
	1.	→ Sort
	2.	→ Aggregate
	3.	→ Hash Inner Join **Hash Cond**: (sr.flightid = f.flightid)
	4.	→ Seq Scan on satisfaction_ratings as sr
	5.	→ Hash
	6.	→ Seq Scan on flights as f

Fig. 13. Hash level analysis of the query execution.

Hash Level Analysis: The hash-level analysis of the query (as seen in the third screenshot) reveals the performance bottleneck in this query. The ‘Hash‘ operations and sequential scans on the ‘Satisfaction Ratings‘ and ‘Flights‘ tables take significant time, especially when dealing with large volumes of data. The database engine performs hash joins and aggregate calculations in memory, which can be resource-intensive. Since the query involves grouping and sorting operations after performing the joins, it leads to increased execution costs. Indexing the ‘FlightID‘ column in both the ‘Satisfaction Ratings‘ and ‘Flights‘ tables would significantly optimize the performance by reducing the number of sequential scans and improving the efficiency of the join and sorting operations.

C. Select with Join and SubQuery

The above query fetches passengers whose seat comfort rating is above the average for a particular flight. While this query executes successfully, it involves a subquery in the WHERE clause. This can be problematic as the database engine must first execute the subquery, compute the average, and then filter the main query based on the result. This results in slower execution times, especially when dealing with large datasets, as it requires scanning the satisfaction ratings table multiple times.

The graphical execution plan for this query shows that multiple sequential scans occur on the ‘Satisfaction Ratings‘ and ‘Passengers‘ tables. This is inefficient when handling large amounts of data. In particular, the subquery inside the WHERE clause adds to the overall execution time by requiring the engine to process the filter condition for every row in the main query.

The hash join between the ‘Passengers‘ table and the ‘Satisfaction Ratings‘ table is performed after the subquery

```sql
SELECT
    p.PassengerID,
    p.CustomerType,
    sr.SeatComfort
FROM
    Passengers p
JOIN
    Satisfaction_Ratings sr ON p.PassengerID = sr.PassengerID
WHERE
    sr.SeatComfort > (
        SELECT AVG(SeatComfort)
        FROM Satisfaction_Ratings
        WHERE FlightID = 3448
    );
```

Data Output | Messages | Explain | Notifications

	passengerid integer	customertype character varying (50)	seatcomfort integer
1	1	Loyal Customer	3
2	2	Loyal Customer	5
3	4	Loyal Customer	4
4	6	Loyal Customer	3
5	7	Loyal Customer	5
6	8	Loyal Customer	5
7	9	Loyal Customer	5
8	10	Loyal Customer	4
9	11	Loyal Customer	5
10	12	Loyal Customer	4
11	13	Loyal Customer	5
12	14	Loyal Customer	4
13	15	Loyal Customer	4
14	18	Loyal Customer	4
15	19	Loyal Customer	5
16	20	Loyal Customer	4
17	21	Loyal Customer	5
18	23	Loyal Customer	4
19	24	Loyal Customer	4
20	28	Loyal Customer	4

Fig. 14. Query result for the passengers with seat comfort ratings above average for FlightID 3448.

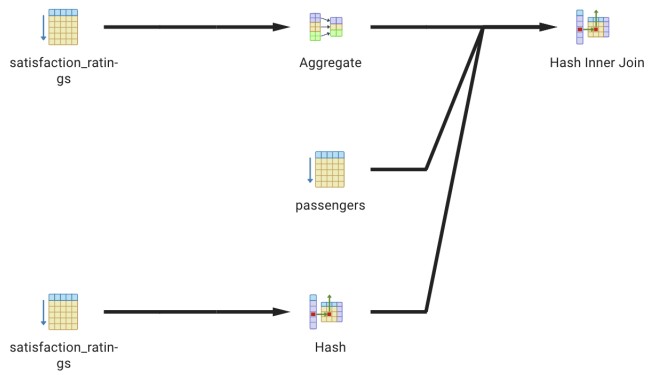

Fig. 15. Graphical execution plan showing inefficient subquery execution.

execution. This leads to higher computational cost and slower response time, especially when these tables contain millions of records. Additionally, the absence of indexes on critical columns like ‘FlightID‘, ‘PassengerID‘, and ‘SeatComfort‘ results in sequential scans, which are less efficient for large datasets.

#	Node
1.	→ Hash Inner Join **Hash Cond**: (p.passengerid = sr.passengerid)
2.	→ Aggregate
3.	→ Seq Scan on satisfaction_ratings as satisfaction_ratings **Filter**: (flightid = 3448)
4.	→ Seq Scan on passengers as p
5.	→ Hash
6.	→ Seq Scan on satisfaction_ratings as sr **Filter**: ((seatcomfort)::numeric > $0)

Fig. 16. Hash level analysis showing sequential scans and join operations.

The hash level analysis further illustrates that multiple hash operations are required due to the sequential scans on both the ‘Satisfaction Ratings‘ and ‘Passengers‘ tables. Indexing the ‘FlightID‘, ‘PassengerID‘, and ‘SeatComfort‘ columns would greatly improve the performance of this query by enabling the database engine to use indexed lookups instead of scanning the entire tables, thus reducing execution time and improving efficiency.

Indexing for Performance Optimization

In the Image, we see that several indexes were created to optimize query performance, particularly when joining tables or filtering rows based on certain conditions. The following indexes were created:

- **PassengerID Index** on the Satisfaction_Ratings table: This index is used to speed up the join between the ‘Passengers‘ and ‘Satisfaction Ratings‘ tables by PassengerID.
- **FlightID Index** on the Satisfaction_Ratings table: This index speeds up filtering for a specific flight in the subquery.
- **SeatComfort Index** on the Satisfaction_Ratings table: This index helps filter rows with specific ‘SeatComfort‘ ratings greater than the average.
- **FlightID Group Index** on the Satisfaction_Ratings table: This index optimizes the GROUP BY operation for ‘FlightID‘ in aggregate queries.
- **SeatComfort Order Index** on the Satisfaction_Ratings table: This index optimizes the ORDER BY operation on the ‘SeatComfort‘ column.

Now, let’s compare the query execution before and after indexing by looking at the execution plans.

In the older analysis, before indexing, we can see that the database is using sequential scans (‘Seq Scan‘) for the ‘Satisfaction Ratings‘ table, which is slower for large datasets. Specifically, the query had to filter rows based on ‘FlightID‘

#	Node
1.	→ Hash Inner Join **Hash Cond**: (p.passengerid = sr.passengerid)
2.	→ Aggregate
3.	→ Seq Scan on satisfaction_ratings as satisfaction_ratings **Filter**: (flightid = 3448)
4.	→ Seq Scan on passengers as p
5.	→ Hash
6.	→ Seq Scan on satisfaction_ratings as sr **Filter**: ((seatcomfort)::numeric > $0)

Fig. 17. Query Execution Plan Before Indexing

and ‘SeatComfort‘. This resulted in more time-consuming operations as the entire table had to be scanned.

After the indexes were created, as shown in the next Image:

1.	→ Hash Inner Join **Hash Cond**: (p.passengerid = sr.passengerid)
2.	→ Aggregate
3.	→ Bitmap Heap Scan on satisfaction_ratings as satisfaction_ratings **Recheck Cond**: (flightid = 3448)
4.	→ Bitmap Index Scan using idx_flightid_group **Index Cond**: (flightid = 3448)
5.	→ Seq Scan on passengers as p
6.	→ Hash
7.	→ Seq Scan on satisfaction_ratings as sr **Filter**: ((seatcomfort)::numeric > $0)

Fig. 18. Query Execution Plan After Indexing

We notice a significant improvement. The database uses a ‘Bitmap Index Scan‘ for ‘FlightID‘, which is much faster than the sequential scan. The index ‘idx flightid group‘ is now utilized for filtering the rows, reducing the need to scan the entire table. Additionally, the use of indexes on ‘SeatComfort‘ helped optimize the ‘WHERE‘ clause filtering, improving query performance.

This demonstrates that indexing not only speeds up join operations by providing quick access to relevant rows but also reduces the amount of data that needs to be scanned, especially for large datasets with conditions like filtering and ordering. By creating indexes, we significantly improved the query execution time, as reflected in the reduction of sequential scans and the adoption of more efficient index-based scans.

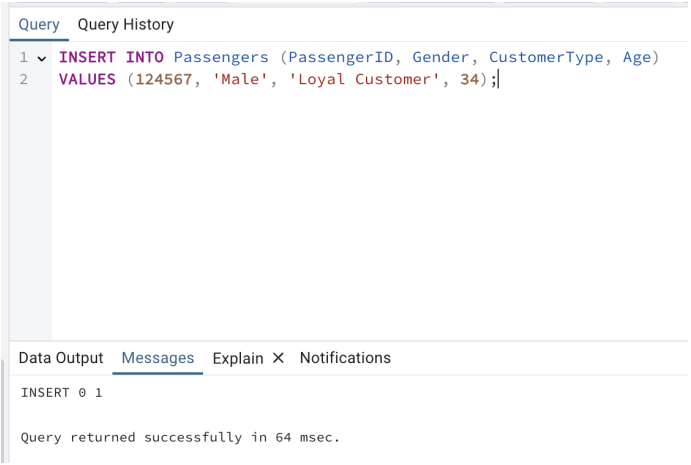

Fig. 19. Insert Query for Passengers Table executed successfully.

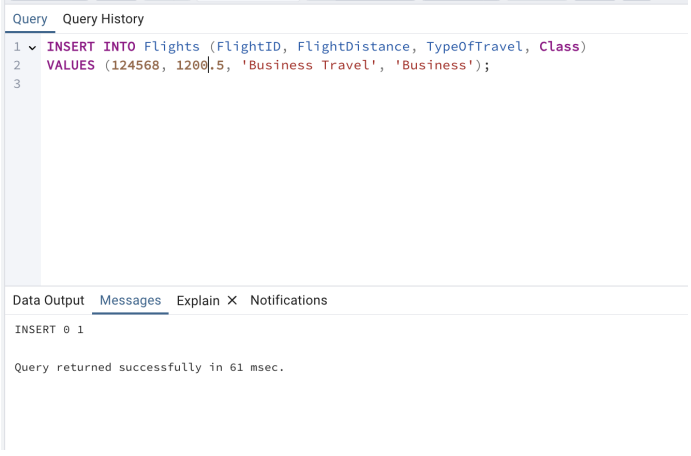

Fig. 20. Insert Query for Flights Table executed successfully.

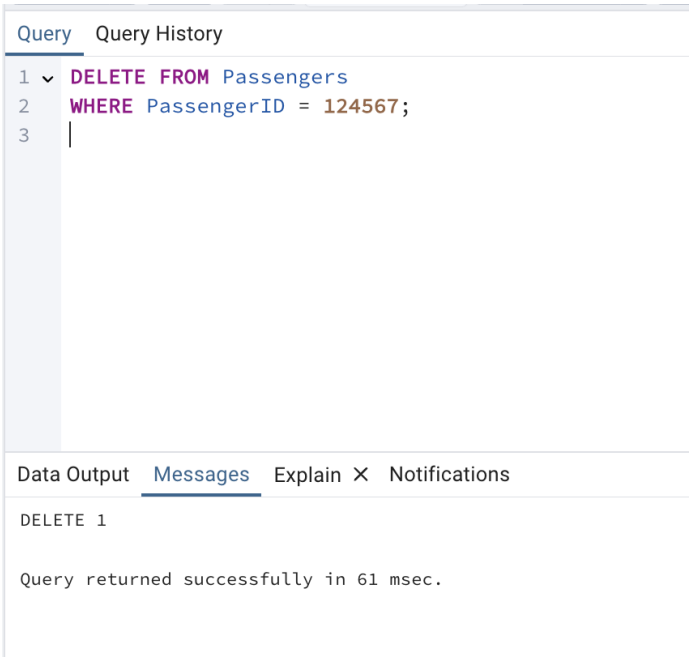

Fig. 21. Delete Query for Passengers Table executed successfully.

Fig. 22. Delete Query for Passengers Table executed successfully.

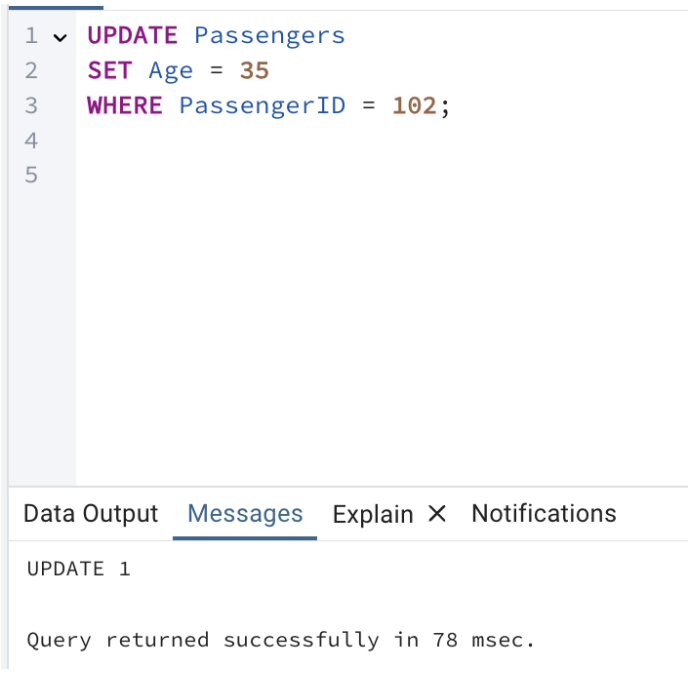

Fig. 23. Update Query for Passengers Table executed successfully.

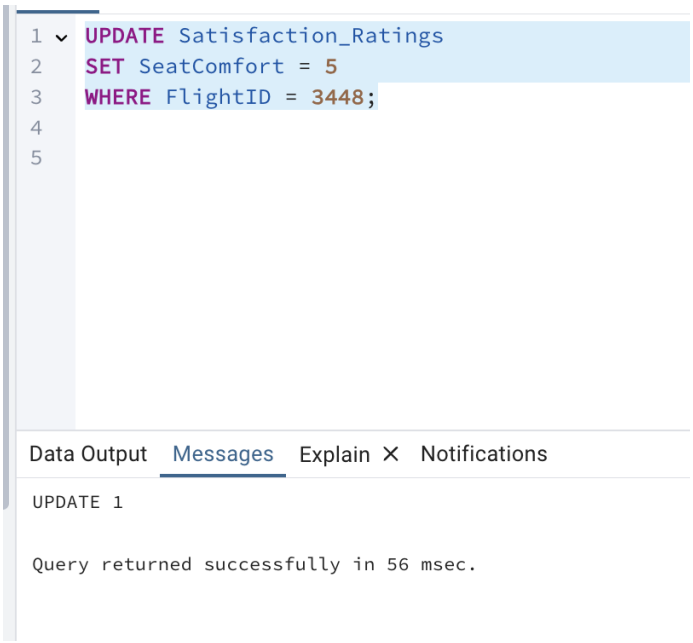

Fig. 24. Update Query for Satisfaction Ratings Table executed successfully.

```sql
SELECT p.PassengerID, sr.SeatComfort, sr.InflightWifiService
FROM Passengers p
JOIN Satisfaction_Ratings sr ON p.PassengerID = sr.PassengerID
WHERE p.PassengerID = 104;
```

Data Output | Messages | Explain | Notifications

	passengerid integer	seatcomfort integer	inflightwifiservice integer
1	104	5	3

```sql
SELECT f.FlightID, AVG(sr.SeatComfort) AS AvgSeatComfort
FROM Flights f
JOIN Satisfaction_Ratings sr ON f.FlightID = sr.FlightID
GROUP BY f.FlightID;
```

Data Output | Messages | Explain | Notifications

	flightid [PK] integer	avgseatcomfort numeric
1	1489	5.0000000000000000
2	6114	4.5000000000000000
3	4790	5.0000000000000000
4	273	5.0000000000000000
5	3936	3.0000000000000000
6	2574	3.5000000000000000
7	5761	5.0000000000000000
8	5843	4.3333333333333333
9	5729	3.6666666666666667
10	5468	5.0000000000000000

Fig. 26. Select query with GROUP BY to calculate average SeatComfort for each FlightID.

Query | Query History

```sql
SELECT PassengerID, CustomerType, Age
FROM Passengers
ORDER BY Age DESC;
```

Data Output | Messages | Explain | Notifications

	passengerid [PK] integer	customertype character varying (50)	age integer
1	18141	Loyal Customer	85
2	23454	Loyal Customer	85
3	4003	disloyal Customer	85
4	15468	disloyal Customer	85
5	7548	Loyal Customer	85
6	11984	Loyal Customer	85
7	15785	disloyal Customer	85
8	7629	Loyal Customer	85
9	20083	Loyal Customer	80
10	3615	Loyal Customer	80
11	15937	Loyal Customer	80
12	1690	Loyal Customer	80
13	11966	Loyal Customer	80
14	10703	Loyal Customer	80

Fig. 27. Select query to list Passengers sorted by Age in descending order.

```sql
SELECT p.PassengerID, p.CustomerType, sr.SeatComfort
FROM Passengers p
JOIN Satisfaction_Ratings sr ON p.PassengerID = sr.PassengerID
WHERE sr.SeatComfort > (
  SELECT AVG(SeatComfort)
  FROM Satisfaction_Ratings
  WHERE FlightID = 3447
);
```

Data Output | Messages | Explain | Notifications

	passengerid integer	customertype character varying (50)	seatcomfort integer
1	2	Loyal Customer	5
2	7	Loyal Customer	5
3	8	Loyal Customer	5
4	9	Loyal Customer	5
5	11	Loyal Customer	5
6	13	Loyal Customer	5
7	19	Loyal Customer	5
8	21	Loyal Customer	5
9	29	Loyal Customer	5
10	32	Loyal Customer	5

Fig. 28. Select query with a subquery to get passengers whose SeatComfort rating is higher than the average for a specific FlightID.

Conclusion

In this project, we successfully designed and implemented a database for managing and analyzing passenger satisfaction data. The database structure was carefully designed to ensure efficient data storage and retrieval, incorporating multiple tables such as Passengers, Flights, and Satisfaction Ratings. We used various SQL queries to perform complex data manipulation and retrieval operations, including INSERT, DELETE, UPDATE, and SELECT operations, ensuring the flexibility and robustness of the system.

Additionally, we applied indexing to optimize query performance, particularly for frequently queried columns, which greatly improved the execution speed of complex operations. By using indexes, we observed a noticeable reduction in query execution time, especially for join and group by operations. This optimization was crucial for handling large datasets and ensuring scalability as the database grows.

The project also demonstrated the application of relational database principles, including normalization, efficient data retrieval, and the use of constraints for data integrity. Through these efforts, the system now offers fast and reliable data operations, making it well-suited for real-time applications in airline passenger satisfaction analysis.

Overall, this project showcases the importance of proper database design and optimization in handling large-scale datasets, ensuring high performance, and supporting data-driven decision-making processes.

Contributions

Tasks for Pavan Pajjuri

1) Design the overall database schema, including creating tables for Passengers, Flights, and Satisfaction Ratings based on the project requirements.
2) Develop and implement SQL queries for data manipulation, including INSERT, UPDATE, and DELETE operations for maintaining and updating the database records.
3) Write the methodology section of the project report, explaining the database design process, query creation, and data handling strategies.
4) Finalize the conclusion of the report, summarizing the key findings from the database setup, query performance, and any improvements made during the project.

Tasks for Sai Krishna Valdas

1) Populate the database with relevant sample data for Passengers, Flights, and Satisfaction Ratings, ensuring that the data follows the defined schema.
2) Optimize the database schema and SQL queries to improve performance, including indexing and query tuning to handle large datasets effectively.
3) Write the data analysis section of the report, providing explanations and visualizations of the query results, highlighting key patterns and insights.
4) Prepare the final presentation for the project, summarizing the database design, tasks completed, and the results obtained from the analysis.

Bonus Task : Streamlit Application for Airline Passenger Satisfaction Database

In this section, we present the interface of the Streamlit application used to interact with the Airline Passenger Satisfaction Database.

Custom Query Page

The Custom Query page allows users to enter and run custom SQL queries on the database. The user can input SQL queries such as ”select * from passengers limit 5” and retrieve results directly from the database.

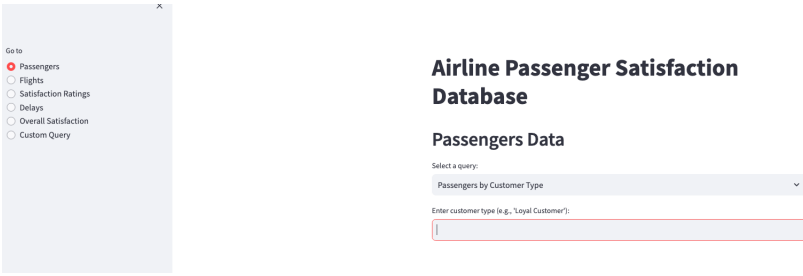

Fig. 29. Custom Query Page in Streamlit Application.

The demonstration of this feature will be provided later for better understanding.

Passengers Data Query Page

The Passengers Data page enables the user to query passengers based on various filters, such as Customer Type. Here, the user can input a specific customer type (e.g., 'Loyal Customer') and retrieve matching records from the database.

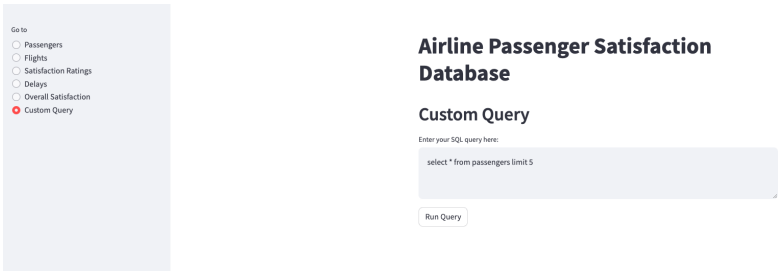

Fig. 30. Passengers Data Query Page in Streamlit Application.

Like the previous feature, a demo will be provided later to showcase the full functionality of this query page.

REFERENCES

[1] Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume 1*, Computer Science Press, 1982.

[2] Garcia-Molina, H., Ullman, J. D., Widom, J., *Database Systems: The Complete Book*, 2nd ed., Pearson Prentice Hall, 2009.

[3] Codd, E. F., "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, 1970.

[4] Date, C. J., *An Introduction to Database Systems*, 8th ed., Addison-Wesley, 2004.

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