**Bus Ticketing Database**

This report provides the design and implementation of a Bus Ticketing database which efficiently manages data related to customers, bus routes, drivers, ticket prices, schedules, and transactions. By using Python to generate data programmatically, ensured it reflects real-world scenarios without relying on actual datasets. The database was designed with data integrity, scalability, and realistic constraints.

**Database Overview**

It is built with several interlinked tables; each has a designated purpose:

* Customer Table: Stores personal information and pass validity details of passengers.
* Driver Table: Stores the contact and professional details of bus drivers.
* BusRoute Table: Captures key details about routes, including starting and destination points.
* TimeTable Table: Contains all the stops and timings for each route.
* Price Table: Has ticket prices for all specific routes and stops.
* Ticket Table: Integrates all relevant information to have a booking history.

**Database Schema**

The tables in the database should be connected to ensure the data is consistent, reliable.

A diagram of a function

Description automatically generated

The Customer Table is connected to the Ticket Table through the Customer\_id (foreign key) which is a one-to-many relationship. Implies that each customer can book multiple tickets. Similarly, the Driver Table is linked to the Ticket Table through Driver\_id, ensuring that every ticket is assigned to a driver while each driver can handle multiple tickets across different routes.

The BusRoute Table is connected to the TimeTable Table using Route\_num, ensuring that each route has its stops and schedules clearly outlined. The Price Table is also linked to the BusRoute Table, with prices corresponding to specific route segments.

The Ticket Table serves as the driving table, integrating customer, driver, route, schedule, and pricing details to provide a complete record for each booking.

**Data Types**

Used various types of data in database to handle information effectively:

* **Nominal Data:** Included categorical data without order like gender (e.g., "Male" or "Female"), pass types (e.g., "Monthly" or "Day"), and route stops.
* **Ordinal Data:** involved attributes which have meaningful order, such as driver experience levels.
* **Interval Data:** Consists of attributes with meaningful difference but does not have true zero, such as bus stop timings or booking timestamps.
* **Ratio Data:** Incorporated columns with true zeros, like ticket prices or passenger ages.

**Data Generation**

The data for this database was generated using Python, leveraging the Faker library to create realistic and randomized values. This approach ensured that the data mimics real-world scenarios without relying on pre-existing datasets.

Please find the below link for the python code used for data generation:

<https://github.com/pratapponnam/SQLAssignment/blob/main/Data_Generation.ipynb>

A screenshot of a computer program

Description automatically generated

The **Customer** table was populated with attributes such as row\_id (a unique identifier), Name (realistic customer names), and phone\_number (UK-specific phone numbers generated using Faker, with 10% of values left intentionally blank to simulate missing data). Email addresses (email\_id) were similarly generated, with 5% of the rows left blank to account for incomplete records. The pass\_type field was randomly assigned one of four values: Monthly, Weekly, Day, or NULL, with validity dates (pass\_valid\_from and pass\_valid\_to) dynamically calculated based on the type of pass. Gender and age were also included, with constraints ensuring realistic values (e.g., age between 18 and 60).

A screenshot of a computer

Description automatically generated

The **Driver** table has attributes such as row\_id (unique identifier), name (realistic names), and phone\_number (unique numbers),Email ID, DOB (ensuring drivers are aged between 25 and 60), Experience (random values between 1 and 30 years), and Age (calculated dynamically based on the date of birth).

A screenshot of a computer

Description automatically generated

The **BusRoute** table was populated with information about bus routes, including Row\_id (unique identifier), Route\_num (a unique integer for each route), and Start\_stop and Destination\_stop (generated as random city names). Both forward and return routes were created by swapping the start and destination stops.

A screenshot of a computer

Description automatically generated

For the **TimeTable** table, data was generated to capture stop details for each route. Attributes included Row\_id (unique identifier), route\_num and Intermediate stops (stop\_name) were generated as random street names, with stop timings (time) dynamically calculated by adding 8–12 minutes between consecutive stops.

A screenshot of a computer

Description automatically generated

The **Price** table has ticket pricing information for specific routes and stop pairs. Attributes included Row\_id (unique identifier), Route\_num, and Start\_stop and End\_stop (derived from the timetable). Ticket prices were calculated dynamically based on travel time, with adult prices ranging from 2 to 15 and child prices set at 50% of adult prices.

A screenshot of a computer

Description automatically generated

The **Ticket** table stores transactional data for ticket bookings, with attributes such as row\_wid (unique identifier), Customer\_wid (foreign key referencing the Customer table), Driver\_wid (foreign key referencing the Driver table), and Bus\_id (foreign key referencing the BusRoute table). It also includes Start\_stop, End\_stop, Price\_adult, Price\_child, No\_of\_adults, No\_of\_children, pass (indicating whether the customer has a pass), and Ticket\_date\_time (randomly generated within the current month). The total price for each ticket was calculated dynamically, with discounts applied for pass holders. Constraints such as a composite key (Customer\_wid, Ticket\_date\_time, Route\_num) ensured that a customer could book only one ticket per route on a specific date.

In the data generation process, duplicates were identified and they were removed to ensure data consistency and integrity. This was achieved by creating a temporary table, Ticket\_stage\_unique, which retained only unique records based on the composite key (customer\_id, ticket\_date\_time, route\_num). Using a ROW\_NUMBER() function, duplicate entries were identified and the first occurrence within each duplicate group was retained. These deduplicated records were then inserted into the main Ticket table, ensuring no conflicts with existing data by excluding entries already present in the main table.

Below scripts are used to handle the duplicates

<https://github.com/pratapponnam/SQLAssignment/blob/main/Insert%20Ticket.sql>

A screenshot of a computer

Description automatically generated

**Justification for Separate Tables**

Separating the data into distinct tables was essential for maintaining organization, reducing redundancy, and adhering to best practices in database design.

A single-table approach will cause duplication and thereby increasing database size, as customer details would repeat for every ticket, and driver information for every route. By splitting the data into logical tables, the updates are simpler. For example, updating a customer’s phone number in the **Customer Table** automatically reflects in all related bookings without the need for multiple edits.

This structure also improves scalability. Adding a new route or driver requires updates only to the relevant table, leaving the rest of the database unaffected. Moreover, queries are more efficient, as the system accesses only the required tables instead of scanning through complete database.

**Ensuring Data Integrity through Constraints**

To maintain consistency and accuracy, the database has constraints such as primary keys, foreign keys, and composite keys.

Primary keys, like row\_id in the **Customer Table**, ensure unique entries, while foreign keys link related tables (e.g., customer\_id connects the **Customer Table** to the **Ticket Table**).

Composite keys, such as the combination of Customer\_id, Ticket\_date\_time, and Route\_num in the **Ticket Table**, prevent duplicate bookings for the same route and date.

These constraints not only uphold the integrity of the data. Also, enforce essential business rules such as misuse of the pass.

**Ethical Considerations**

All data was generated using Python's Faker library, ensuring that no real-world or personal information was used. This approach minimizes the risk of privacy violations while allowing the database to simulate realistic scenarios.

Sensitive information, such as phone numbers and email addresses, is stored only in the Customer Table and referenced indirectly via foreign keys. Nullable fields were incorporated to reflect real-world scenarios where users may choose not to share personal details. Additionally, no bias was introduced during data generation; for instance, names, genders, and ages were assigned randomly, ensuring inclusivity and fairness.

**Reporting**

This database is not only used store ticket bookings. However, it is also designed to generate valuable insights and support reporting needs.

A simple query can calculate the total revenue generated today by summing up ticket prices, which gives daily earnings.

A screenshot of a computer program

Description automatically generated

Similarly, the database can track driver performance by counting how many unique trips a driver, like John, completed in the past week.

A screen shot of a computer code

Description automatically generated

These capabilities make the database a powerful resource for both day-to-day operations and strategic decision-making, helping businesses monitor performance, optimize operations, and stay informed.

**Conclusion**

The Bus Ticketing Database is a carefully designed solution that combines efficiency, scalability, and ethical responsibility. With its structured tables, realistic constraints, and focus on privacy, the database provides a robust foundation for managing a dynamic and complex ticketing system.

Github link for Database, queries and CSV files:

<https://github.com/pratapponnam/SQLAssignment/tree/main>

Author: Pratap Bhargav Ponnam

Id: 23027654