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.

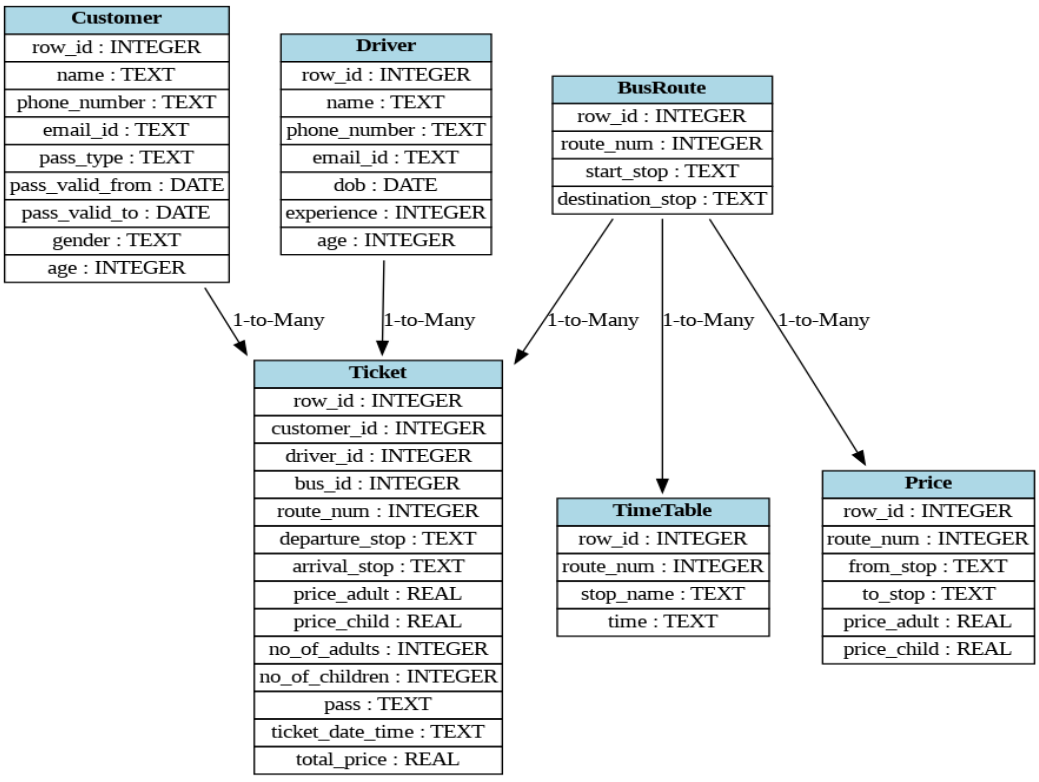
Database Overview

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.



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:

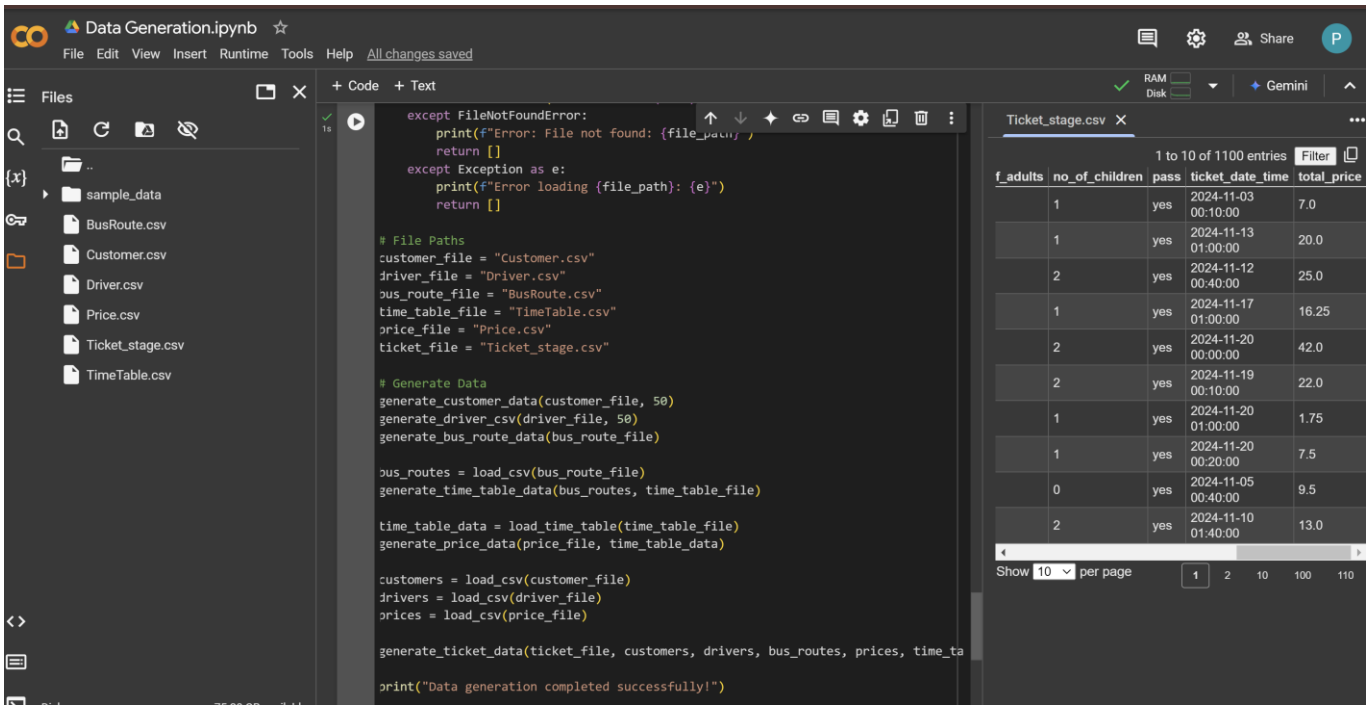
- **Nominal Data:** Included categorical data without order like gender (e.g., "Male" or "Female"), pass types (e.g., "Monthly" or "Day"), and route stops.
- **Ordinal Data:** involved attributes which have meaningful order, such as driver experience levels.
- **Interval Data:** Consists of attributes 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



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).

```
select * from Customer;
```

row_id	name	phone_number	email_id	pass_type	pass_valid_from	pass_valid_to	gender	age
1	Rosemary Allen	+44(0)1214960980	dianewalker@example.net	Weekly	2024-11-07	2024-11-14	Male	42
2	Mr Cameron Roberts	(0131) 496 0349	rlee@example.com	Day	2024-11-04	2024-11-04	Female	41
3	Shane Hyde	020 7946 0919	joannagarner@example.net	Day	2024-11-02	2024-11-02	Male	21
4	Dr Anthony Collins	0306 999 0867	georgina40@example.net	Day	2024-11-11	2024-11-11	Female	19
5	Francesca Davies	0909 879 0822	ebryant@example.com	Weekly	2024-11-07	2024-11-14	Male	47
6	Jasmine Griffiths	0191 4960501	christopherlane@example.com	Day	2024-11-01	2024-11-01	Male	24
7	Matthew Fletcher	+44909 879 0825	abdul09@example.com	Weekly	2024-11-15	2024-11-22	Male	57

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).

```
select * from Driver;
```

row_id	name	phone_number	email_id	dob	experience	age
1	Gerard Bradley	7081699511	roymarshall@example.org	1997-02-25	1	27
2	Francesca Jones	8169778778	albert30@example.org	1989-11-19	11	35
3	Dr Howard Turner	1381572760	wyattstewart@example.org	1989-04-05	15	35
4	Pauline Griffiths	5490273520	harpersuzanne@example.org	1993-05-30	15	31
5	Mr Adam Robinson	7644660403	hutchinsondenise@example.net	1987-11-23	11	37
6	Dr David Russell	8079245408	rachaelchapman@example.com	1983-05-25	7	41
7	Debra Walton	3307384823	deborahgibson@example.net	1991-12-30	9	33

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.

```
select * from BusRoute;
```

row_id	route_num	start_stop	destination_stop
1	1	South Melanie	Lake Maureen
2	6	Lake Maureen	South Melanie
3	2	Port Russell	West Terryside
4	7	West Terryside	Port Russell
5	3	East Martin	Marcusfurt
6	8	Marcusfurt	East Martin

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.

```
select * from TimeTable;
```

row_id	route_num	stop_name	time
1	1	South Melanie	07:47:01
2	1	Mitchell walk	07:52:01
3	1	Lauren fall	07:58:01
4	1	Thomas glens	08:04:01
5	1	Karen turnpike	08:10:01
6	1	Laura shores	08:18:01
7	1	Moore shoal	08:25:01

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.

```
select * from Price;
```

row_id	route_num	from_stop	to_stop	price_adult	price_child
1	1	South Melanie	Mitchell walk	3.5	1.75
2	1	South Melanie	Lauren fall	5.0	2.5
3	1	South Melanie	Thomas glens	6.5	3.25
4	1	South Melanie	Karen turnpike	8.0	4.0
5	1	South Melanie	Laura shores	9.5	4.75
6	1	South Melanie	Moore shoal	11.0	5.5

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

```
select * from ticket;
```

row_id	customer_id	driver_id	bus_id	route_num	departure_stop	arrival_stop	price_adult	price_child	no_of_adults	no_of_children	pass	ticket_date_time	total_price
111	4	23	6	8	Jeffrey squares	Peter road	15.5	7.75	2	1	yes	2024-11-17 00:10:00	23.25
112	4	11	8	9	Laura canyon	O'Brien fork	8.0	4.0	2	0	yes	2024-11-19 00:40:00	8.0
113	4	1	10	10	Hopkins corner	Hooper fort	8.0	4.0	3	0	yes	2024-11-19 01:30:00	16.0
114	4	3	7	4	Shane harbors	Collins vista	6.5	3.25	2	2	yes	2024-11-20 00:30:00	13.0
115	4	22	1	1	Moore shoal	Lake Maureen	11.0	5.5	1	0	yes	2024-11-20 01:00:00	0.0
116	4	38	9	5	Palmer fords	Hopkins corner	5.0	2.5	2	1	yes	2024-11-21 00:50:00	7.5

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.

```
SELECT
    SUM(total_price) AS today_revenue
FROM
    Ticket
WHERE
    DATE(ticket_date_time) = DATE('now');
```

today_revenue
220.5

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

```
SELECT
    d.name AS driver_name,
    COUNT(DISTINCT DATE(t.ticket_date_time) || '-' || t.bus_id) AS trips_completed
FROM
    Ticket t
JOIN
    Driver d
ON
    t.driver_id = d.row_id
WHERE
    d.name = 'David Brown'
    AND t.ticket_date_time >= DATE('now', '-7 days')
    AND t.ticket_date_time < DATE('now')
GROUP BY
    d.name;
```

driver_name	trips_completed
David Brown	5

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