1. INTRODUCTION & OVERVIEW

Objective
The primary goal of this mini-project is to design, implement, and demonstrate an **Indian Railway Ticket Reservation System** that covers:

1. **Conceptual & Logical Design**: Entities, relationships, constraints, and a normalized relational model.
2. **Implementation**: Creation of tables, foreign keys, triggers, stored procedures, and functions in an RDBMS (MySQL).
3. **Population**: Inserting realistic sample data to test all features (booking, cancellations, seat availability, partial refunds, waitlist/RAC).
4. **Demonstrations & Queries**: Showcasing typical queries (PNR status, seat availability, passenger lists, etc.) and advanced features (partial refunds, eWallet usage, RAC promotion, etc.).

Key Features

- **Booking, modifying, and canceling tickets** for different classes (Sleeper, AC, etc.).
- **Seat availability** tracking; partial seat-level inventory in some designs.
- **Waitlist (WL) & RAC** logic, promoting waitlisted passengers when seats free up.
- **Online and counter** bookings with multiple payment modes (card, cash, UPI, eWallet).
- **Concessions** (senior, student) and partial refunds depending on cancellation time.
- **E-Wallet** integration (with DOB serving as a "password").
- **Detailed table structure** with bridging tables for many-to-many relationships.

2. CONCEPTUAL DESIGN (E-R MODEL)

1. **Entities**
    - **Passenger**: Represents each traveler.
    - **Customer**: The person who creates the booking (may or may not be a passenger).
    - **Train**: Basic train info (ID, name, number, type, operating days).
    - **Station**: Each station in the network.
    - **Train_Routes**: Defines source, destination, distance for each route.
    - **Schedule**: Linking a Train to a Route (and possibly date/time info).

- **Class**: Distinct classes (Sleeper, AC tiers, etc.).
- **Booking**: Overall booking transaction (Online/Counter, date/time, status).
- **Ticket**: Each ticket has a PNR, fare, status (Confirmed, RAC, WL, or Cancelled).
- **Payment**: Payment details (mode, amount, refund info).

2. **Relationships**
   - **Station_Connects**: A route has many stations with a defined sequence.
   - **Stops_At**: A train stops at multiple stations on certain schedules.
   - **Offers**: A train offers certain classes with seat availability.
   - **Ticket_Passenger**: Many passengers can link to one ticket, with seat_number and sub-status.
   - **Booking_Ticket**: Each booking can include multiple tickets.
   - **Pays**: A customer pays for a payment record.
   - **Reserved_On**: Optionally track physical seats (coach_number, seat_number) for each train.
   - **Customer_Ticket**: If multiple customers can be associated with a single ticket, or to keep direct references.

3. RELATIONAL SCHEMA (18 TABLES)

Below is a **summary** of the final tables:

1. **Train** (train_id PK, train_name, train_number, train_type, operating_days, is_active)
2. **Train_Routes** (route_id PK, route_name, source_station, destination_station, distance)
3. **Schedule** (schedule_id PK, train_id FK, route_id FK)
4. **Station** (station_id PK, station_name, station_code, city, state)
5. **Station_Connects** (route_id, station_id, sequence_number, distance_from_source)
6. **Class** (class_id PK, class_code, class_name, base_fare_multiplier)
7. **Passenger** (passenger_id PK, passenger_name, passenger_age, passenger_gender)
8. **Customer** (customer_id PK, customer_name, dob, ewallet_balance, upi_id, card_number, etc.)
9. **Ticket** (ticket_id PK, pnr_number UNIQUE, journey_date, booking_status, fare_amount, class_id FK, [train_id FK], [request_time])

10. **Booking** (booking_id PK, booking_type, booking_date, total_amount, booking_status)
11. **Payment** (payment_id PK, booking_id FK, payment_date, payment_amount, payment_mode, transaction_status, refund_amount, refund_date)
12. **Reserved_On** (train_id, coach_number, seat_number, class_id) – seat-level inventory if used.
13. **Stops_At** (train_id, station_id, day_of_journey, arrival_time, departure_time)
14. **Ticket_Passenger** (ticket_id, passenger_id, seat_number, passenger_status)
15. **Customer_Ticket** (customer_id, ticket_id, booking_date)
16. **Booking_Ticket** (booking_id, ticket_id)
17. **Offers** (train_id, class_id, available_seats)
18. **Pays** (customer_id, payment_id)

This schema covers **all** bridging relationships and supports advanced seat, route, station, passenger, and payment tracking.

4. DATA POPULATION

After creating the tables, we inserted **sample data** to test every feature, focusing on realistic examples:

1. **Customer**: eWallet balances, DOB as password, card and UPI details.
2. **Passenger**: Basic passenger info (name, age, gender).
3. **Train**, **Train_Routes**, **Station**, **Station_Connects**, **Schedule**: Provide the route/station relationships.
4. **Class** and **Offers**: Setting base fare multipliers per class and seat availability.
5. **Booking**, **Ticket**, **Booking_Ticket**: Linking each booking to multiple tickets, with final statuses (Confirmed, RAC, WL, Cancelled).
6. **Payment**, **Pays**: Payment records with partial or full refunds.
7. **Reserved_On** (optional inserts) if you want seat-level details.

Example:

```
sql
CopyEdit
INSERT INTO Ticket (
  ticket_id, pnr_number, journey_date, booking_status,
  fare_amount, ticket_type, concession_category, class_id
```

```
)
VALUES
(601, 'PNR1234561', '2025-05-01', 'Confirmed', 1500.00,
'Adult', 'None', 302),
...
```

Likewise, we added a **`refund_amount`** and **`refund_date`** to **Payment** records to handle partial refunds:

```
sql
CopyEdit
INSERT INTO Payment (
  payment_id, booking_id, payment_date, payment_amount,
  payment_mode, transaction_status, refund_amount,
refund_date
)
VALUES
(901, 801, '2025-04-15 10:31:00', 1500.00, 'Credit Card',
'Success', 0.00, NULL),
(903, 803, '2025-04-17 09:20:00', 800.00, 'UPI', 'Success',
200.00, '2025-04-17 10:00:00'),
...
```

This sample data ensures we can test seat availability, partial refunds, bookings in all statuses, etc.

5. TYPICAL QUERIES (WITH SUBQUERIES)

We demonstrated many **SELECT** queries:

1. **PNR Status Tracking**
   ```
   sql
   CopyEdit
   ```

   ```
   SELECT t.ticket_id, t.pnr_number, t.booking_status, ...
   ```

2. ```
   FROM Ticket t
   ```
3. ```
   WHERE t.pnr_number = :pnrNumber;
   ```
4.
   - Possibly includes subqueries to gather passenger details from `Ticket_Passenger` or references to `Class` and `Train`.
5. **Train Schedule Lookup** (using **Stops_At**).

6. **Available Seats Query** (using **Offers** and optional references to **Reserved_On**).
7. **List Passengers Traveling** (using **Ticket**, **Ticket_Passenger**, etc.).
8. **Retrieve WL/RAC Passengers**.
9. **Total Refund** for a cancelled train.
10. **Total Revenue** over a period, net of refunds.
11. **Cancellation Records** with their `refund_amount` from **Payment**.
12. **Busiest Route** by passenger count.
13. **Itemized Bill** for a ticket (fare breakdown, passenger listing, final net payment).

We also provided **extended** queries for advanced logic (top 3 trains by revenue, station-based lookups, frequent travelers by `Customer_Ticket`, etc.).

6. TRIGGERS

To automate seat availability changes, partial refunds, or other business rules, we created triggers such as:

1. **`trg_decrement_seats_on_ticket_confirm`**:
    - **AFTER INSERT** on **Ticket**.
    - If `booking_status = 'Confirmed'`, decrement `Offers.available_seats`.
    - Could also update `Reserved_On` to mark a seat as "Booked."
2. **`trg_increment_seats_on_cancel`**:
    - **AFTER UPDATE** on **Ticket**.
    - If status changes from `'Confirmed'` to `'Cancelled'`, increment seats in `Offers`, then call a chain promotion procedure to handle waitlist → RAC → Confirmed.
3. **`trg_partial_refund_on_cancel`** (optional)
    - If the time difference is < 24 hours from journey_date, only partial refund (like 50%). Otherwise, a full refund.
4. **PNR Generation, Payment Timestamp**
    - **`trg_auto_generate_pnr`**: If `pnr_number` is NULL, set it to a random or time-based code.

- **`trg_insert_payment_timestamp`**: If `payment_date` is NULL, fill with current timestamp.

These triggers **enforce** consistency and **automate** business logic so the developer or application doesn't have to handle every seat or refund detail manually.

7. STORED PROCEDURES

7.1 Core Procedures

1. **`sp_create_booking_with_customer_ticket`**
   - Creates a **Booking**, a **Ticket**, links them via **Booking_Ticket**, then also inserts a record into **Customer_Ticket**.
   - Inserts a **Payment** record (and references **Pays** to connect the right customer).
2. **`sp_cancel_booking`**
   - Cancels a booking, updates all related tickets, sets them to `'Cancelled'`.
   - The seat increments or refund logic is triggered via **AFTER UPDATE** triggers or direct updates in the procedure.
3. **`sp_chain_promotions`**
   - A loop that repeatedly checks `Offers.available_seats`, finds the earliest `'RAC'` ticket, sets it to `'Confirmed'`, decrements seats, then optionally moves `'WL'` → `'RAC'`.
4. **`sp_generate_invoice`** / **`sp_print_ticket_details`**
   - Produces an itemized breakdown of the booking/ticket, listing fare, seats, passenger names, partial refunds, etc.

7.2 Other Example Procedures

- **`sp_update_waitlist`** or **`sp_promote_RAC_to_confirmed`**: Alternative naming for chain logic.
- **`sp_bulk_add_stations`**: Insert multiple stations from a CSV-like input.
- **`sp_merge_duplicate_passengers`**: Merges two passenger records if discovered to be duplicates.

8. USER-DEFINED FUNCTIONS

We introduced **functions** that encapsulate small computations or lookups:

1. **`fn_calculate_fare(distance, class_id, concession_category)`**: Returns the final fare.
2. **`fn_seat_availability(train_id, class_id, journey_date)`**: Returns how many seats remain (combining `Offers` or date-level logic).
3. **`fn_penalty_for_late_cancellation(p_journey_date)`**: Determines the penalty ratio for partial refunds depending on how close to departure.
4. **`fn_find_free_seat_in_reserved_on(p_train_id, p_class_id)`**: Finds a free seat from `Reserved_On`.
5. **`fn_get_customer_booking_count(p_customer_id)`**: Returns how many bookings a given customer has (linking **Customer_Ticket**, **Booking_Ticket**).

Each function can be called within triggers or procedures to unify the logic.

9. KEY LOGIC FLOWS

1. **Seat Availability**
   - **Offers** holds an overall seat count by `(train_id, class_id)`.
   - **Reserved_On** can list physical seat numbers for each coach.
   - A **trigger** or procedure updates seats if a ticket is `'Confirmed'` or `'Cancelled'`.
2. **Waitlist & RAC**
   - If **Offers.available_seats** = 0, new tickets get `'WL'` or `'RAC'`.
   - Upon cancellation, the system increments seats, then **promotes** earliest `'RAC'` to `'Confirmed'`, earliest `'WL'` to `'RAC'`.
   - This can be done via `sp_chain_promotions` or in a **trigger** calling that procedure.
3. **Cancellation & Partial Refunds**
   - If a ticket is cancelled too close to the journey date, a trigger or procedure sets `refund_amount` to partial. Otherwise it may be a full refund.
   - `Payment` includes columns for `refund_amount` and `refund_date`.
4. **eWallet & DOB as Password**

- The system can store each customer's `ewallet_balance` and `dob`.
- A booking procedure verifies `p_dob = Customer.dob`; if using eWallet, checks `ewallet_balance >= payment_amount`, then deducts.

5. **Data Integrity**
   - Foreign keys across bridging tables (e.g., **Booking_Ticket**, **Customer_Ticket**, **Pays**) maintain references.
   - Triggers ensure seat logic or refunds are applied automatically and consistently.

10. CONCLUSION & FUTURE EXTENSIONS

Achievements:

- We've **designed** a complete schema of 18 tables, addressing every detail from seat availability and route schedules to bridging relationships for multi-passenger and multi-customer scenarios.
- We've **populated** them with sample data that covers various statuses (Confirmed, WL, RAC, Cancelled), partial refunds, eWallet usage, etc.
- We've demonstrated a wide range of **queries**, **stored procedures**, **functions**, and **triggers** for typical railway operations—seat checks, ticket creation, promotions, cancellations, partial refunds, PNR generation, etc.
- The system is **modular**: you can easily add or refine triggers (e.g., advanced seat-level tracking, better waitlist priorities) or procedures (like sp_print_ticket_invoice).

Potential Extensions:

- Real-time seat partitioning for **RAC** (two RAC passengers sharing a berth).
- **Security** enhancements (storing card data hashed or externalizing payment info).
- **Performance** improvements for large data sets (indexes on frequently joined columns, partitioning older data).
- **Front-end application** integration with a user-friendly interface.

Through this project, we've **simulated** the processes of:

1. **Conceptualizing** a complex domain (Indian Railway Reservations).
2. **Building** a robust **relational schema** with bridging tables, keys, and constraints.
3. **Populating** meaningful data.
4. **Automating** business rules with triggers, stored procedures, and user-defined functions.

5. **Demonstrating** queries that answer real operational needs (PNR checks, seat availability, cancellations, promotions, refunds).

The result is a **fully functional** database system that can be showcased with **live demos** and **SQL queries** illustrating every aspect of a typical railway reservation workflow.