Train Ticket Management System

भारतीय प्रौद्योगिकी संस्थान जम्मू
INDIAN INSTITUTE OF TECHNOLOGY JAMMU
विद्याधनं सर्वधन प्रधानम्

Akash Das	2022UCS0078
Aryan Raj	2022UCS0081
Chaitanya Tandon	2022UCS0089
Mohd Sarim Shamim	2022UCS0096
Muthres Gurjar	2022UCS0097

Functional Requirements

User Authentication:

- User should be able to sign up and log in securely to access the system.

Train Search:

- User should be able to search for trains based on journey date, boarding station, destination station, travel class and quota.

Ticket Booking:

- After selecting a train, user should be able to book tickets for multiple passengers and add their details.

Payment System:

- Payments should be secure and PIN-protected.
- Users should be able to use wallet functionality for convenience.
- System must check if the current balance is sufficient to perform transactions.

Functional Requirements

Booking Generation and Management:

- Once the payment is successful, a booking should be generated.
- Users should be able to view their reservations and check their booking status on a personal booking page.
- Users should be able to cancel tickets from the booking page if needed.

Waitlist Management:

- The system should automatically upgrade a waitlisted passenger to confirmed status upon cancellation of some ticket, following a first-come, first-served order.

Admin Features:

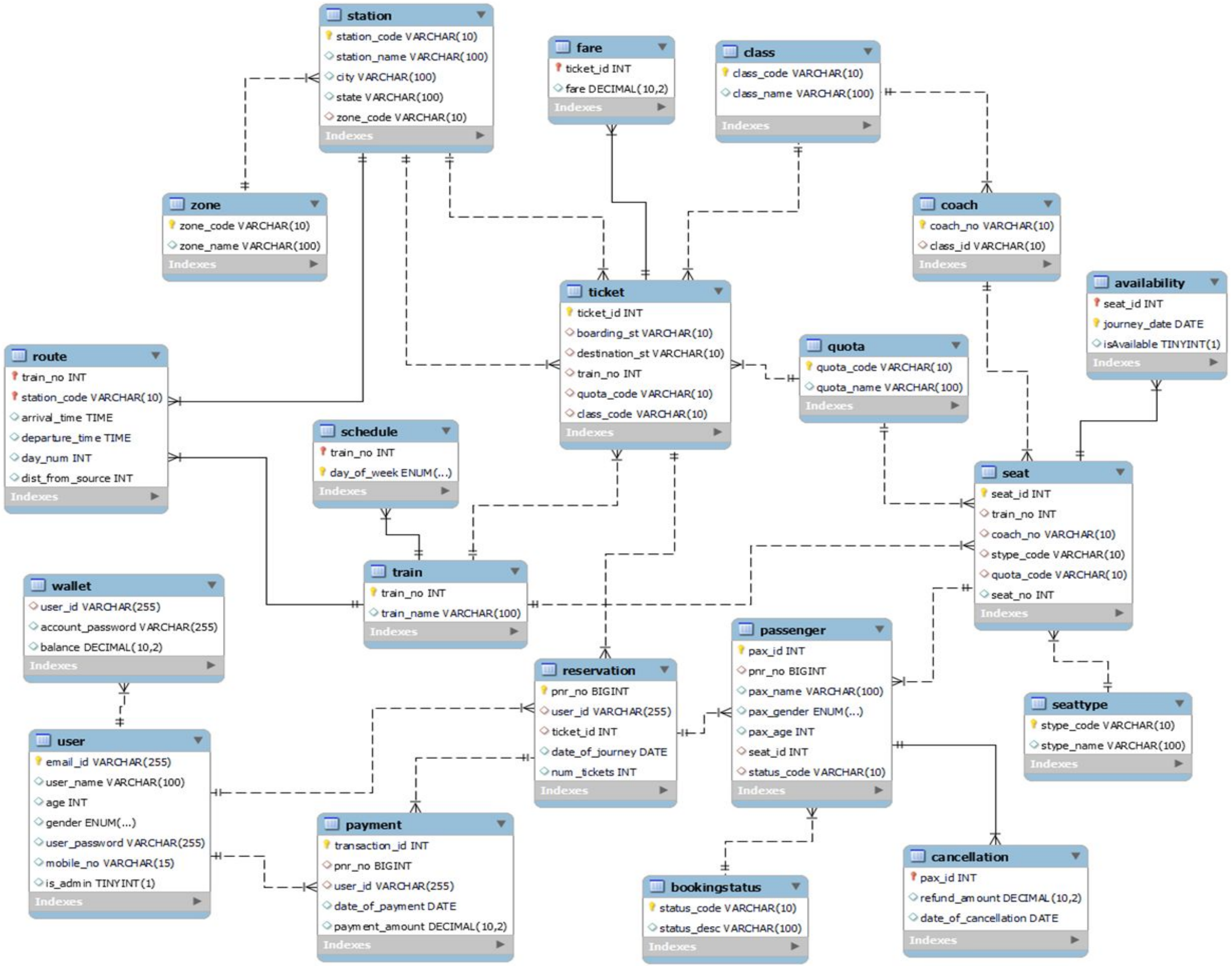
- Administrators should be able to access a dedicated admin page.
- The admin page should allow CRUD operations on the database.

Data Integrity:

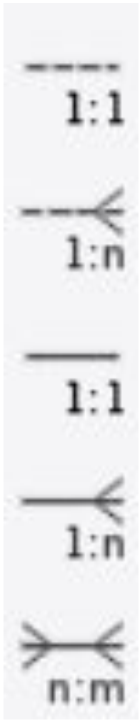
- The system should ensure data consistency by securely handling transactions for both payments and cancellations.

विद्याधनं सर्वधन प्रधानम्

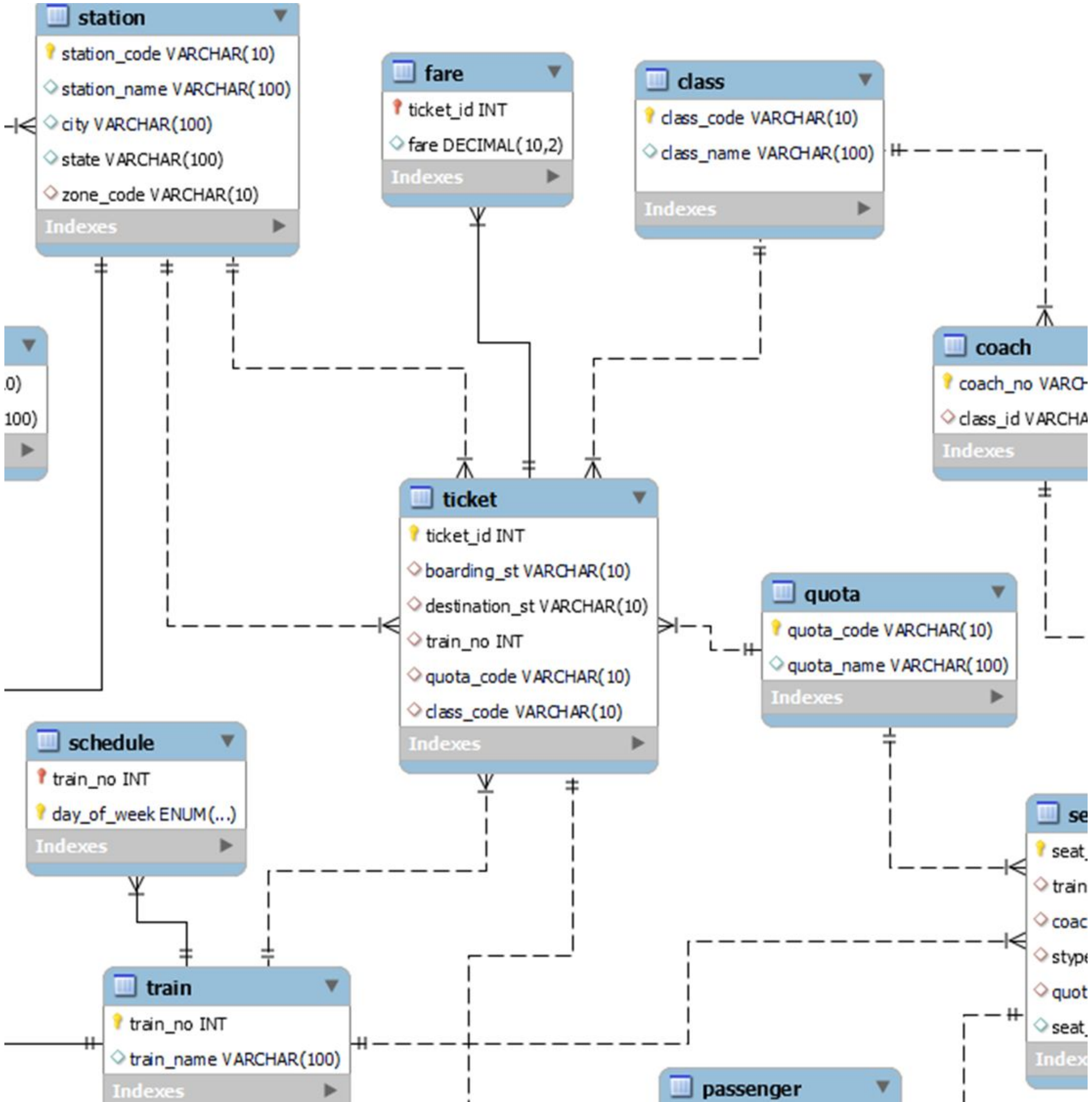
Entity Relationship Diagram



Entity-Relationship Diagram for the Train Ticket Management System (TTMS) database. The design revolve around two central entity ticket and reservation table.



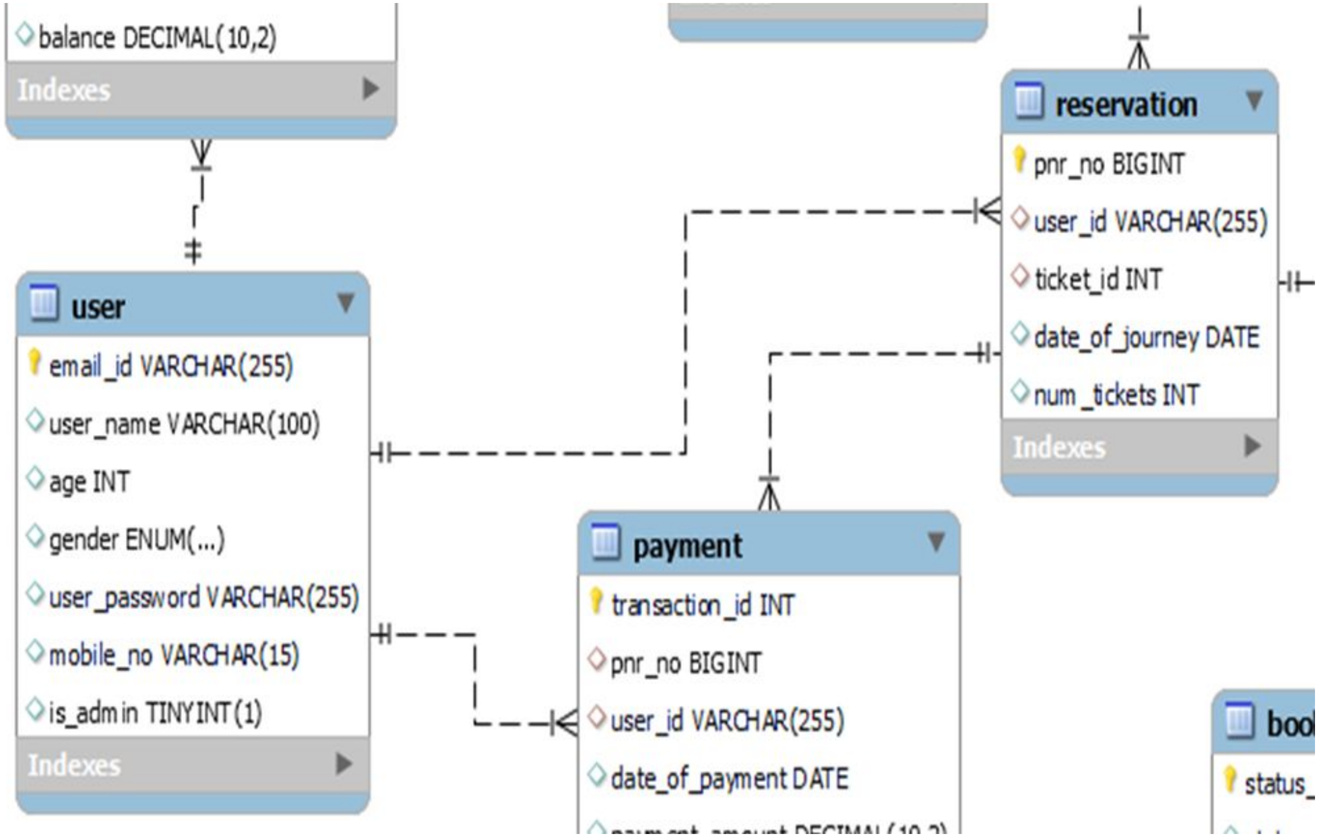
Entity Relationship Diagram



Ticket Entity

The Ticket table stores details of booked journeys, including boarding and destination stations, train number, quota, and coach class. It is connected to the Station, Train, Quota, and Class tables, which provide information about the train's route, travel conditions, and reservation types.

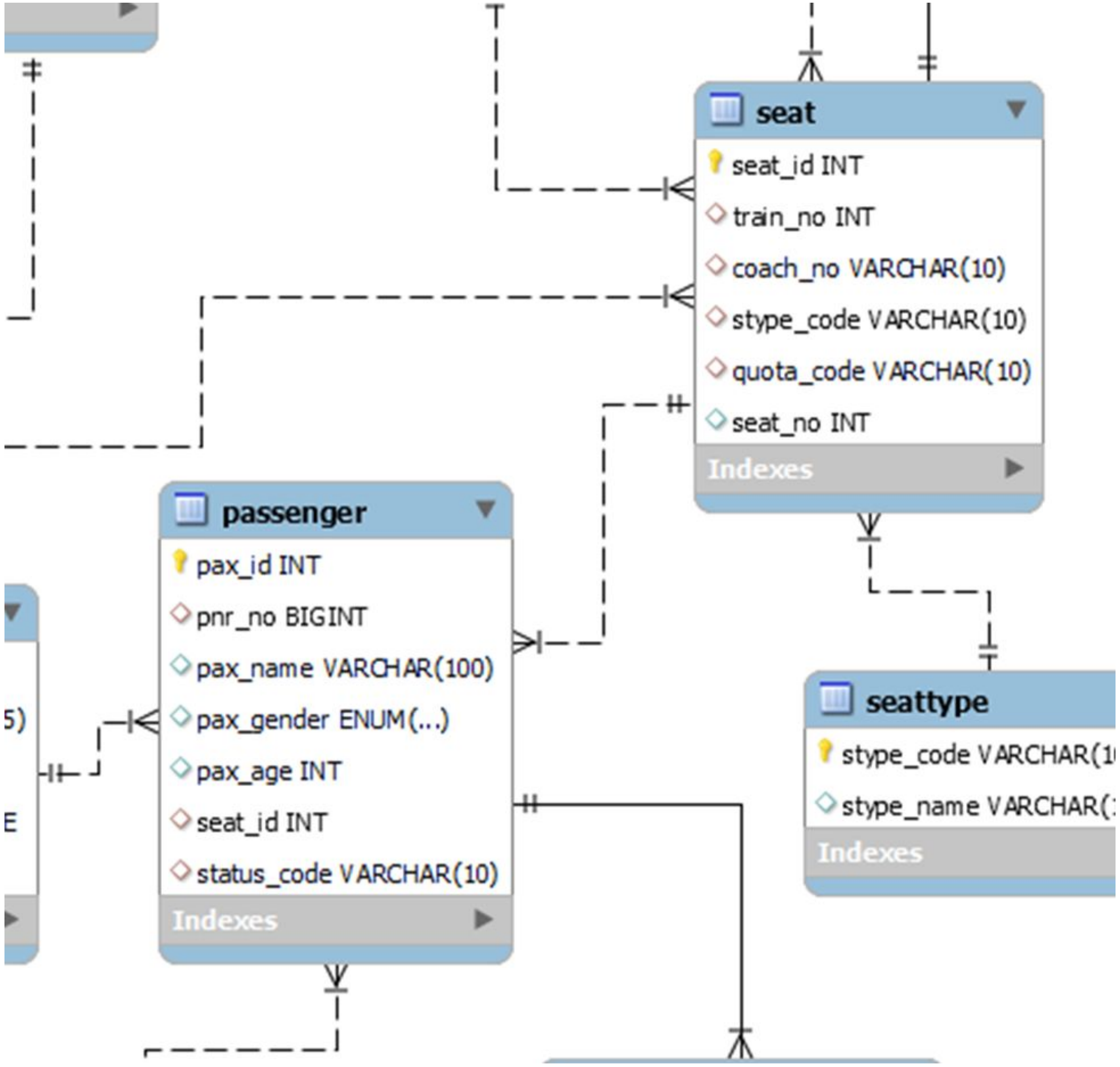
Entity Relationship Diagram



Reservation Entity

The Reservation table creates a unique PNR for each booking and links it to a user from the User table. The User table stores user details like email, name, contact information, and wallet balance. The Reservation table also includes the number of tickets booked and the journey date.

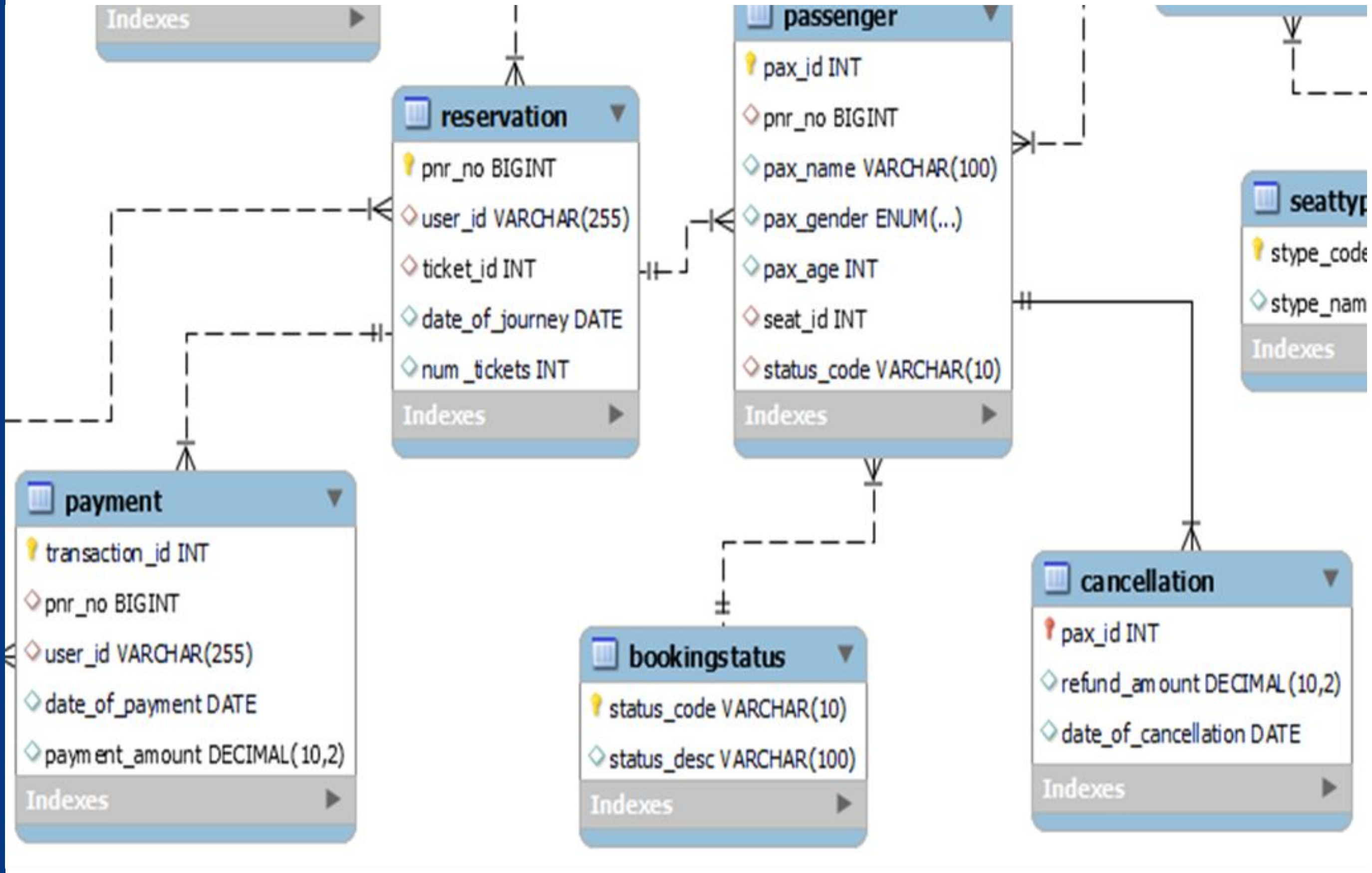
Entity Relationship Diagram



Passengers and Seats Entity

The Passenger table stores details of individual travelers for each PNR, including their name, gender, age, and assigned seat. The Seat table handles seat assignments within coaches, connected to trains, classes, and quotas. The Availability table keeps track of seat availability for specific journey dates.

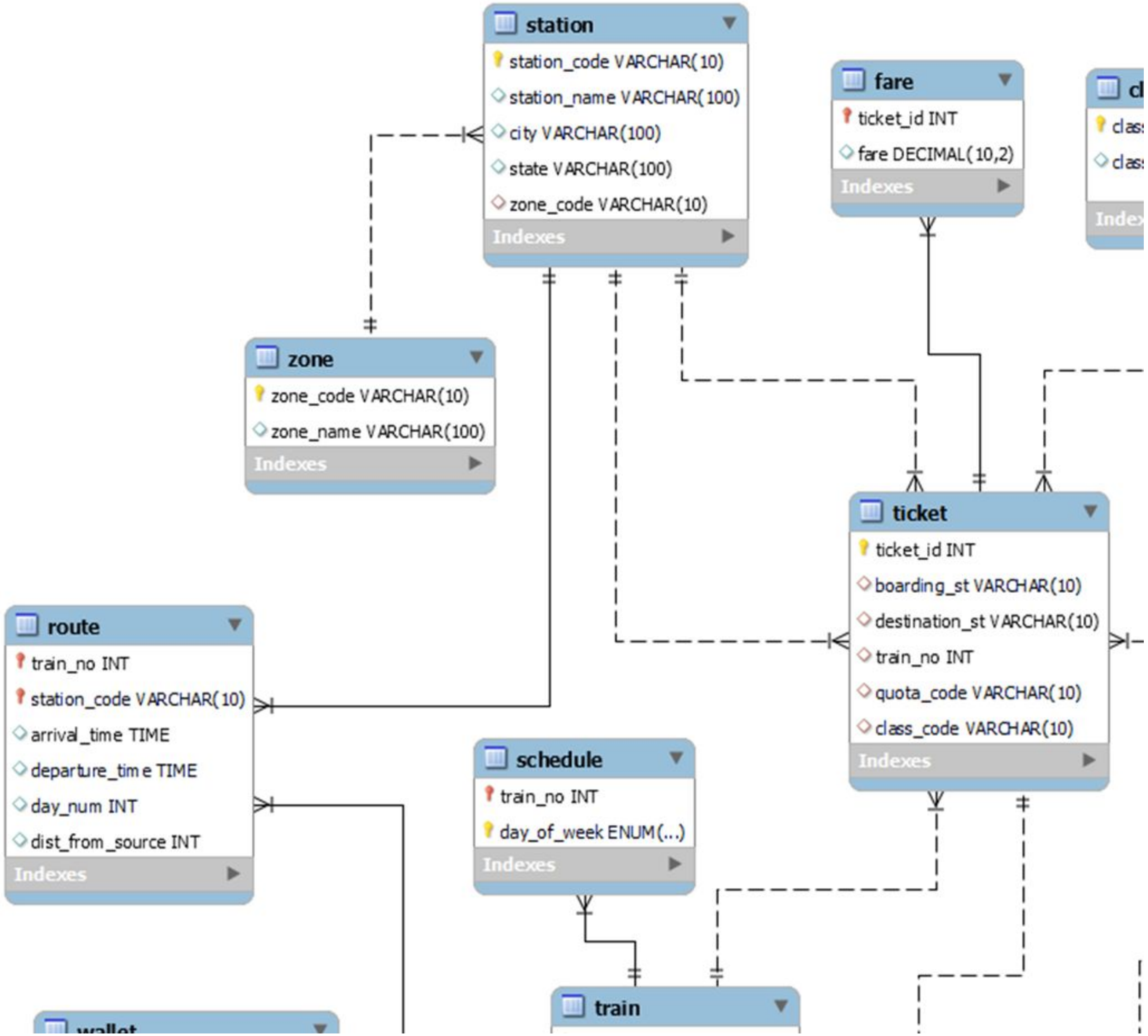
Entity Relationship Diagram



Payment and Cancellation Entity

The Payment table records transactions for reservations, while the Cancellation table tracks refund amounts for canceled bookings. These tables link financial activities to passenger and their reservations.

Entity Relationship Diagram



Train Scheduling and Routes

The Train table is linked to the Route table, which specifies stations, timings, and distances for each train. The Schedule table defines the operating days of trains, while the Coach and SeatType tables describe the train's setup.

Functional Dependencies

1. User Table

Schema:

- email_id (Primary Key)
- user_name
- age
- gender
- user_password
- mobile_no
- is_admin

FDs:

1. `email_id → user_name, age, gender, user_password, mobile_no, is_admin`

2. Station Table

Schema:

- station_code (Primary Key)
- station_name
- city
- state
- zone_code (Foreign Key)

FDs:

1. `station_code → station_name, city, state, zone_code`
   (Holds because `station_code` is the primary key and uniquely identifies each station.)
2. `zone_code → zone_name`
   (Holds because `zone_code` is a foreign key and references `Zone` table, where `zone_code → zone_name`.)

3. Availability Table

Schema:

- seat_id (Foreign Key)
- journey_date
- isAvailable

FDs:

1. `(seat_id, journey_date) → isAvailable`
   (Holds because `(seat_id, journey_date)` is the primary key and uniquely identifies availability for each seat on a given date.)

Functional Dependencies

4. Seat Table

Schema:

- seat_id (Primary Key)
- train_no (Foreign Key)
- coach_no (Foreign Key)
- stype_code (Foreign Key)
- quota_code (Foreign Key)
- seat_no

FDs:

1. `seat_id → train_no, coach_no, stype_code, quota_code, seat_no`
   (Holds because `seat_id` is the primary key and uniquely identifies each seat.)

5. Ticket Table

Schema:

- ticket_id (Primary Key)
- boarding_st (Foreign Key)
- destination_st (Foreign Key)
- train_no (Foreign Key)
- quota_code (Foreign Key)
- class_code (Foreign Key)

FDs:

1. `ticket_id → boarding_st, destination_st, train_no, quota_code, class_code`
   (Holds because `ticket_id` is the primary key and uniquely identifies each ticket.)

6. Reservation Table

Schema:

- pnr_no (Primary Key)
- user_id (Foreign Key)
- ticket_id (Foreign Key)
- date_of_journey
- num_tickets

FDs:

1. `pnr_no → user_id, ticket_id, date_of_journey, num_tickets`
   (Holds because `pnr_no` is the primary key and uniquely identifies each reservation.)

Functional Dependencies

7. Quota Table

Schema:

- quota_code (Primary Key)
- quota_name

FDs:

1. `quota_code → quota_name` (Holds because `quota_code` is the primary key and uniquely identifies each quota.)

8. Wallet Table

Schema:

- user_id (Foreign Key)
- account_password
- balance

FDs:

1. `user_id → account_password, balance` (Holds because `user_id` uniquely identifies each wallet entry and determines password and balance.)

9. Reservation Table

Schema:

- pnr_no (Primary Key)
- user_id (Foreign Key)
- ticket_id (Foreign Key)
- date_of_journey
- num_tickets

FDs:

1. `pnr_no → user_id, ticket_id, date_of_journey, num_tickets` (Holds because `pnr_no` is the primary key and uniquely identifies each reservation.)

Normalization

3NF and Dependency Preserving Algorithm
Input: A Relation R_i and a set F of FDs over R_i

---

i=0
for each FD $\alpha \rightarrow \beta$ in F
do
  if none of the schemes R_j $(1 \leq j \leq i)$ contain β
    i=i+1;
    R_i = $\alpha\beta$;
  if none of the schemes R_j $(1 \leq j \leq i)$ contain a candidate key for R
    i=i+1;
    R_i = any candidate key for R;
return $(R_1, R_2, R_3, \ldots R_n)$

Normalization

Relation R

= (train_no, station_code, arrival_time, departure_time, day_num, dist_from_source, station_name, city, state, zone_code, zone_name)

Functional Dependencies

1. (train_no, station_code) → (arrival_time, departure_time, day_num, dist_from_source)
2. (station_code) → (station_name, city, state, zone_code)
3. (zone_code) → (zone_name)

Normalization

Set $i = 0$. We process each FD

→ Create R_1

$= (\alpha, \beta)$

= (train_no, station_code, arrival_time, departure_time, day_num, dist_from_source)

→ Create R_2

= (station_code, station_name, city, state, zone_code)

→ Create R_3

= (zone_code, zone_name)

Normalization

Since Candidate key = (train_no, station_code) is present in R_1, no additional relation is needed

Thus, the algorithm terminates

R_1 = (train_no, station_code, arrival_time, departure_time, day_num, dist_from_source)

R_2 = (station_code, station_name, city, state, zone_code)

R_3 = (zone_code, zone_name)

Query Optimisation

What is Query Optimization?

Query optimization is the process of enhancing the efficiency of SQL queries to minimize their execution time and resource usage while ensuring the same results. The database management system (DBMS) achieves this by selecting the best possible execution plan from multiple available strategies.

Why is Query Optimization Important?

- Efficient use of resources (CPU, memory, and disk).
- Faster response times, critical for real-time applications.
- Lower operational costs, especially for large-scale systems.

Theory Behind Query Optimization

1. **Heuristic-Based Optimization (Query Rewriting)**:
   - **Predicate Pushdown**: Reduces data processing by moving filters closer to data retrieval.
   - **Join Reordering**: Changes join sequence to minimize intermediate rows.
   - **Projection Pruning**: Limits columns processed, reducing resource consumption.
2. **Join Optimization Techniques**:
   - **Nested Loop Join**: Efficient for small datasets or indexed tables.
   - **Sort-Merge Join**: Effective for pre-sorted datasets.
   - **Hash Join**: Ideal for large datasets, hashes rows of one table for fast lookups.
   - **Semi-Joins/Anti-Joins**: Optimizes queries involving existence or non-existence checks.
3. **Other Methods**:
   - **Parallel Query Execution**: Splits workload across multiple processors.
   - **Query Hints**: Suggestions for specific optimizations.
   - **Caching**: Stores frequently accessed data in memory.
   - **Subquery Flattening**: Transforms subqueries into joins for faster execution.

Here is an examples of optimized queries:-

Seat Availability Query

Original Query:

```
SELECT s.seat_id,
       s.stype_code,
       c.coach_no,
       s.seat_no
FROM Seat s
JOIN Coach c ON s.coach_no = c.coach_no
JOIN Availability a ON s.seat_id = a.seat_id
WHERE s.train_no = %s
  AND c.class_id = %s
  AND s.quota_code = %s
  AND a.journey_date = %s
  AND a.isAvailable = TRUE
LIMIT %s;
```

Problems in the Query:

Late Predicate Application: Filters like `s.train_no = %s`, `c.class_id = %s`, and `a.journey_date = %s` are applied after the joins, processing unnecessary rows.

Optimized Query:

```
WITH FilteredSeats AS (
    SELECT seat_id, stype_code, coach_no, seat_no
    FROM Seat
    WHERE train_no = %s AND quota_code = %s
),
FilteredAvailability AS (
    SELECT seat_id
    FROM Availability
    WHERE journey_date = %s AND isAvailable = TRUE
)
SELECT fs.seat_id,
       fs.stype_code,
       c.coach_no,
       fs.seat_no
FROM FilteredSeats fs
JOIN Coach c ON fs.coach_no = c.coach_no AND c.class_id = %s
JOIN FilteredAvailability fa ON fs.seat_id = fa.seat_id
LIMIT %s;
```

Optimization Techniques Used:

1. **Predicate Pushdown:**
    - Filters (`train_no = %s` and `quota_code = %s`) are applied in the `FilteredSeats` CTE before the join.
    - Filters (`journey_date = %s` and `isAvailable = TRUE`) are applied in the `FilteredAvailability` CTE before the join.

Benefits of Optimization:

1. **Improved Query Execution Time:**
   - Reduced row processing and table scans minimize computational costs.
2. **Reduced Memory Usage:**
   - Smaller intermediate datasets avoid unnecessary memory overhead.
3. **Scalability:**
   - Optimized queries handle larger datasets more effectively without significant performance degradation.

Introduction to Transactions

- A **transaction** is a sequence of operations performed as a single unit.
- Ensures data consistency, integrity, and security.
- Key in maintaining **ACID** properties: Atomicity, Consistency, Isolation, Durability.

Where have we used Transactions?:

- **Payment Handling** (Booking and payment processing)
- **Cancellation Request** (Refund processing and seat availability management)

Payment Handling Process

Step 0: Check wallet balance

- Check if wallet has sufficient balance to continue the transaction.

Step 1: Start Transaction

- Start a new transaction: `mydb.start_transaction()`.

Step 2: Update Wallet Balances

- Deduct total fare from the user's wallet.
- Transfer the deducted amount to the IRCTC wallet.

Step 3: Insert Reservation

- Insert details of the reservation into the `Reservation` table.
- Retrieve the generated PNR number (`pnr_no`).

Step 4: Record Payment

- Insert payment details into the `Payment` table with the `pnr_no`.

Payment Handling Process

Step 5: Fetch Available Seats

- Query for available seats based on user preferences and journey details.

Step 6: Assign Seats to Passengers

- For each available seat:
  - Insert passenger details into the `Passenger` table with status 'CNF'.
  - Mark the seat as unavailable in the `Availability` table.

Step 7: Handle Waitlisted Passengers

- If there are more passengers than seats:
  - Insert passenger details with status 'WL' for the remaining passengers.

Step 8: Commit Transaction

- Finalize the transaction: `mydb.commit()`.

Cancellation Process

Start Transaction

- Begin a new transaction: `mydb.start_transaction()`.

Step 2: Verify Journey Date

- Query to check the journey date for the given `pax_id`.
- If the journey date has passed, return an error and terminate the process.

Step 3: Check Passenger Status

- Fetch the current status of the passenger.
- If the status is `CAN` (already cancelled), return an error.

Step 4: Retrieve Payment Details

- Query the `Payment`, `Passenger`, and `Reservation` tables to calculate:
    - Total payment.
    - Refund amount (80% of the ticket price per passenger).
- Validate that the number of tickets is greater than 0.

Cancellation Process

Step 5: Update Passenger Status

- Mark the passenger's status as `CAN` in the `Passenger` table.

Step 6: Adjust Wallet Balances

- Refund the calculated amount to the user's wallet.
- Deduct the refund amount from the IRCTC wallet.

Step 7: Record Cancellation

- Insert the cancellation details (e.g., refund amount, date) into the `Cancellation` table.

Step 8: Handle Seat and Waitlist

- If the seat is freed:
  - Assign the seat to the first passenger on the waitlist (`WL`).
  - If no waitlisted passengers exist, mark the seat as available.

Step 9: Commit Transaction

- Finalize the transaction: `mydb.commit()`.

भारतीय प्रौद्योगिकी संस्थान जम्मू
INDIAN INSTITUTE OF TECHNOLOGY JAMMU
विद्याधनं सर्वधन प्रधानम्

THANK YOU