Railway Reservation and Management System

Neelima Srilakshmi Bollempalli Shilpa Gabriel

Computer Science Engineering Science

[neelimas@buffalo.edu](mailto:neelimas@buffalo.edu) [shilpaga@buffalo.edu](mailto:shilpaga@buffalo.edu)

Venkata Lakshmi Krishna Tejaswi Gudimetla

Engineering Science

vgudimet@buffalo.edu

**Problem Statement -** The railway sector's growth and technological advancements demand an integrated and centralized database system. There's an imperative need to create a platform where third-party applications can tap into a centralized system, thus making railway schedules, reservations, and other crucial data seamlessly available for all.For such a dynamic and vast system with myriad third-party applications accessing it, using Excel is a sub-optimal solution. Databases offer scalability, robustness, multi-user support, and can efficiently handle large volumes of concurrent requests. It can serve real-time data to multiple applications, ensuring data integrity and consistency, which are paramount for railway operations. With the digital transformation of almost all sectors, the railway system still lags in providing a unified platform for third-party integrations. This limitation has hindered the development of innovative applications, which can revolutionize the passenger experience. It is crucial to address this gap as railways serve millions daily, and modern solutions can optimize operations and improve passenger experience significantly. By creating the Railway Reservation and Management System (RRMS), we pave the way for a plethora of third-party applications. These applications can range from ticket booking platforms, real-time train trackers to advanced analytical tools predicting ticket demands. The system's contribution is crucial as it not only directly benefits the railway operations but also propels an ecosystem of innovative solutions enhancing the end-user experience.

# BACKGROUND

With the digital transformation of almost all sectors, the railway system still lags in providing a unified platform for third-party integrations. This limitation has hindered the development of innovative applications, which can revolutionize the passenger experience. It's crucial to address this gap as railways serve millions daily, and modern solutions can optimize operations and improve passenger experience significantly. By creating the Railway Reservation and Management System (RRMS), we pave the way for a plethora of third-party applications. These applications can range from ticket booking platforms, real-time train trackers to advanced analytical tools predicting ticket demands. The system's contribution is crucial as it not only directly benefits the railway operations but also propels an ecosystem of innovative solutions enhancing the end-user experience.

# Potential and contribution

By creating the Railway Reservation and Management System (RRMS), we pave the way for a plethora of third-party applications. These applications can range from ticket booking platforms, real-time train trackers to advanced analytical tools predicting ticket demands. The system's contribution is crucial as it not only directly benefits the railway operations but also propels an ecosystem of innovative solutions enhancing the end-user experience.

# Target users

## Passengers

They can utilize various third-party applications powered by RRMS to check train schedules and make reservations.

## Employees

The railway staff uses the system for internal operations, station management, train scheduling, and reservation management. In addition to above, the employees also use the system to address customers enquires about train schedules and reservations.

## Third-party Application Developers

Developers can tap into the RRMS to build diverse applications catering to different passenger needs.

## Analytics Team

With access to RRMS, they can derive insights into popular routes, peak travel times, and other patterns for decision-making and predicting trends.

# Database Administrators

The database is owned and maintained by the Information Technology (IT) Team of the organization. It is their responsibility to provide database access to the employees of different teams. Also, the IT team makes sure that insert, delete, and update access on the database tables be given only to authorized employees. Senior database administrator is responsible for the strategic oversight and ensuring the system's scalability and robustness. Operational database administrator manages the day-to-day operations, ensuring data integrity and system efficiency.

# Database tables

The data for this project has been collected mainly from Kaggle (<https://www.kaggle.com/code/kerneler/starter-indian-railways-dataset-ced2e5fd-7/input>). Then, we used python’s Faker module to generate employees, passengers, and reservations data.

* Table name: ***stations***

This table consists of details about the railway stations such as the station code, station name, station zone, station state and station address.

Attributes:

* + ***station\_code*** VARCHAR(20) : Unique identifier of the railway station.
  + ***station\_name*** VARCHAR(100) : Name of the railway station.
  + ***station\_zone*** VARCHAR(20) : Railway zone to which the station belongs.
  + ***station\_state*** VARCHAR(20) : State to which the railway station belongs.
  + ***station\_address*** VARCHAR(150) : Address of the railway station

Primary key: station\_code

* Table name: ***trains***

The table “trains” has all the details about trains such as the train number, train name, starting station, ending station, arrival and departure time, travel distance, travel duration, etc.

Attributes:

* ***train\_number*** VARCHAR(20) NOT NULL : uniquely identifies the train. It cannot be null because it is the primary key of the table
* ***train\_name*** VARCHAR(100) NOT NULL: Name of the train.
* ***from\_station\_code*** VARCHAR (20): It is the station code from which the train begins its journey.
* ***departure*** TIME : The time when the train is set to depart.
* ***to\_station\_code*** VARCHAR (20): It is the destination station code.
* ***arrival*** TIME : The time when the train is set to arrive at the destination station.
* ***distance*** SMALLINT : Total distance between the starting and destination station.
* ***duration\_hours*** SMALLINT : The hours component of the time taken to complete the journey.
* ***duration\_minutes*** SMALLINT: The minutes component of the time taken to complete the journey.
* sleeper BOOLEAN : Specifies whether the train has a sleeper coach.
* chair\_car BOOLEAN : Specifies whether the train has a chair car.
* first\_class BOOLEAN: Specifies whether the train has a first class coach.
* first\_ac BOOLEAN: Specifies whether the train has a first class ac coach.
* second\_ac BOOLEAN: Specifies whether the train has a second class ac coach.
* third\_ac BOOLEAN: Specifies whether the train has a third class ac coach.
* return\_train VARCHAR(20) : The train number of the return train.
* train\_type VARCHAR(20) : Specifies if the train is a passenger train, express, mail, etc.
* total\_seats SMALLINT : Specifies the total number of seats in the train

Primary key: train\_number

Foreign key: to\_station\_code and from\_station\_code references station\_code from “stations” table.

ON UPDATE CASCADE: Updates the rows in “trains” table if the station\_code is updated in the “stations” table.

ON DELETE CASCADE: Deletes the rows in “trains” table if the station\_code is deleted in the “stations” table.

* Table name: schedules

The table “schedules” has the details about train schedules and contains attributes such as schedule id, train number, train name, station code, station name, arrival, and departure time. The ‘station\_code’ is the foreign key that references ‘station\_code’ from “stations” table.

Attributes:

* schedule\_id VARCHAR(20) : Unique identifier of the train schedule.
* train\_number VARCHAR(20) : Unique identifier of the train number.
* station\_code VARCHAR(20) : Unique identifier of the railway station.
* arrival TIME : Time at which the train is set to arrive at a station.
* departure TIME : Time at which the train is set to depart from a station.

Primary key: schedule\_id

Foreign key: station\_code references station\_code from “stations” table

ON UPDATE CASCADE: Updates the rows in “schedules” table if the station\_code is updated in the “stations” table.

ON DELETE CASCADE: Deletes the rows in “schedules” table if the station\_code is deleted in the “stations” table.

* Table name: employees

The table “employees” consists of railway staff details such as employee id, employee first name, employee last name, employee contact details, designation, and salary.

Attributes:

* emp\_id integer NOT NULL : Unique identifier of employees
* station\_id VARCHAR(20) : Unique identifier of the railway station.
* emp\_fname VARCHAR(50) : Employee first name.
* emp\_lname VARCHAR(50) : Employee last name.
* emp\_contact\_number VARCHAR(15) NOT NULL: Employee mobile number.
* emp\_email VARCHAR(50) : Employee email id.
* emp\_yoj integer : Employee year of joining.
* emp\_city VARCHAR(30) : Employee’s city.
* emp\_state VARCHAR(30) : Employee’s state.
* emp\_designation VARCHAR(30) : Designation of the employee.
* emp\_salary integer : Employee’s salary

Primary key: emp\_id

Foreign key: station\_id references station\_code in “stations” table.

ON UPDATE CASCADE: Updates the rows in “employees” table if the station\_code is updated in the “stations” table.

ON DELETE CASCADE: Deletes the rows in “employees” table if the station\_code is deleted in the “stations” table.

* Table name: passengers

The table “passengers” stores all the details related to passengers such as passenger id, passenger first name, passenger last name, passenger contact details, passenger city and state.

Attributes:

* + passenger\_id integer NOT NULL: Uniquely identifies the passenger
  + passenger\_fname VARCHAR(50): Passenger’s first name
  + passenger\_lname VARCHAR(50): Passenger’s last name
  + passenger\_phone VARCHAR(20) : Passenger’s contact number
  + passenger\_email VARCHAR(100): Passenger’s email address.
  + passenger\_city VARCHAR(100) : Passenger’s city.
  + passenger\_state VARCHAR(100): Passenger’s state.

Primary key: passenger\_id

* Table name: seat\_availability

The table “seat\_availability” consists of train number, total seats in that train, date, and total seats available that are not yet booked.

Attributes:

* + train\_number VARCHAR(20) NOT NULL : Uniquely identifies the train
  + date DATE : Date of journey
  + total\_available INT: Number of seats available for booking in the train

Primary key : train\_number, date

* Table name: reservations

The table “reservations” consists details about the train tickets that passengers have reserved. It has the passenger details, date of journey, train details and the seat reserved.

Attributes:

* + Reservation\_id integer NOT NULL: Uniquely identifies the reservations.
  + passenger\_id integer NOT NULL: Uniquely identifies passenger
  + passenger\_fname VARCHAR(50) : Passenger’s first name
  + passenger\_lname VARCHAR(50) : Passenger’s last name
  + date date NOT NULL: Date of journey
  + train\_number VARCHAR(50) NOT NULL : Uniquely identifies the train\_number
  + departure time without time zone: Departure time of the train
  + from\_station\_code VARCHAR(20) : Station code of the railway station from where the journey begins.
  + to\_station\_code VARCHAR(20) : Station code of the destination railway station.
  + arrival time without time zone : Time of arrival at the destination.
  + seat\_number VARCHAR(10) : Seat number allotted for the passenger.

Primary key: reservation\_id

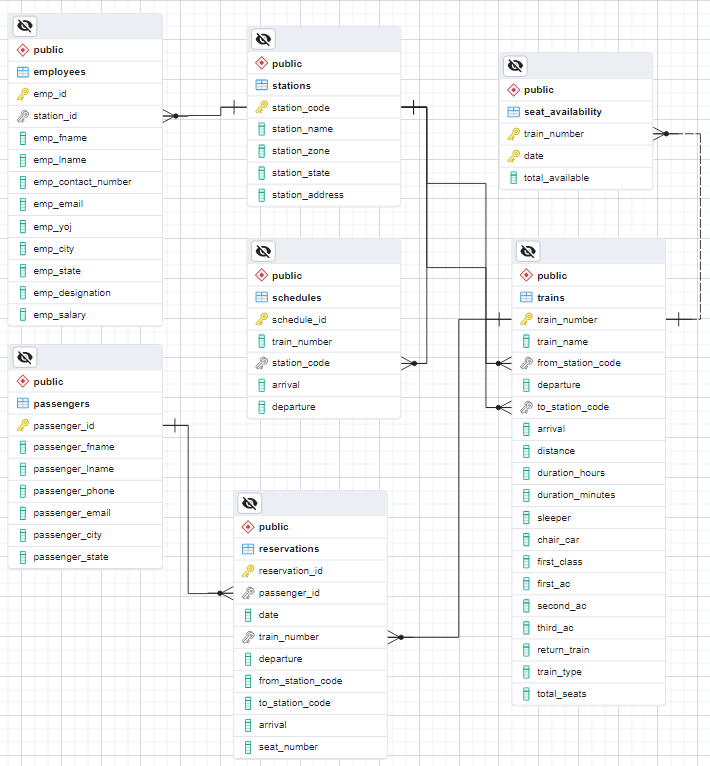
Foreign key: passenger\_id references passenger\_id in “passengers” table, train\_number references train\_number in “trains” table.

ON UPDATE CASCADE: Updates the rows in “reservations” table if the passenger\_id is updated in the “passengers” table. Similarly, updates the rows in “reservations” table if the train\_number is updated in the “trains” table.

ON DELETE CASCADE: Deletes the rows in “reservations” table if the passenger\_id is deleted in the “passengers” table. Similarly, deletes the rows in “reservations” table if the train\_number is deleted in the “trains” table.

## ER Diagram

The ERD is a graphical representation of the entities, relationships, and attributes involved in a database. ER diagrams are commonly used in database design to model and visualize the structure of a database. Below is the ERD of the “Railway Management” database –



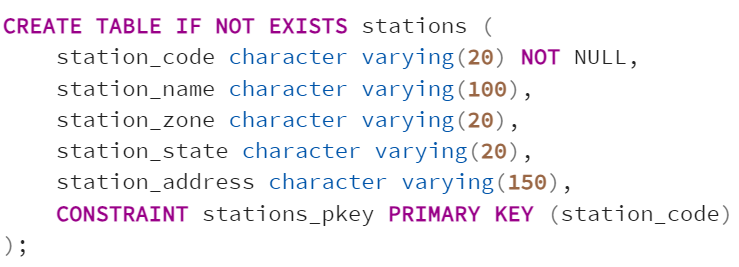
# Table creation AND DATA

## List of Realtion Schemas

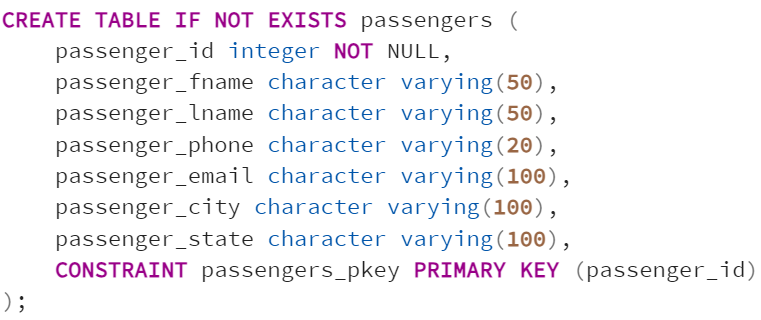
* trains (train\_number, train\_name, from\_station\_code, departure, to\_station\_code, arrival, distance, duration\_hours, duration\_minutes, sleeper, chair\_car, first\_class, first\_ac, second\_ac, third\_ac, return\_train, train\_type, total\_seats)
* stations (station\_code, station\_name,station\_zone,station\_state,station\_address)
* schedules (schedule\_id, train\_number, station\_code, arrival,departure)
* passengers (passenger\_id, passenger\_fname ,passenger\_lname ,passenger\_phone ,passenger\_email ,passenger\_city ,passenger\_state )
* employees (emp\_id, station\_id, emp\_fname, emp\_lname, emp\_contact\_number,emp\_email, emp\_yoj, emp\_city, emp\_state, emp\_designation, emp\_salary)
* seat\_availability (train\_number, date,total\_available)
* reservations (reservation\_id, passenger\_id, date, train\_number, departure, from\_station\_code, to\_station\_code, arrival, seat\_number)

## Create Tables

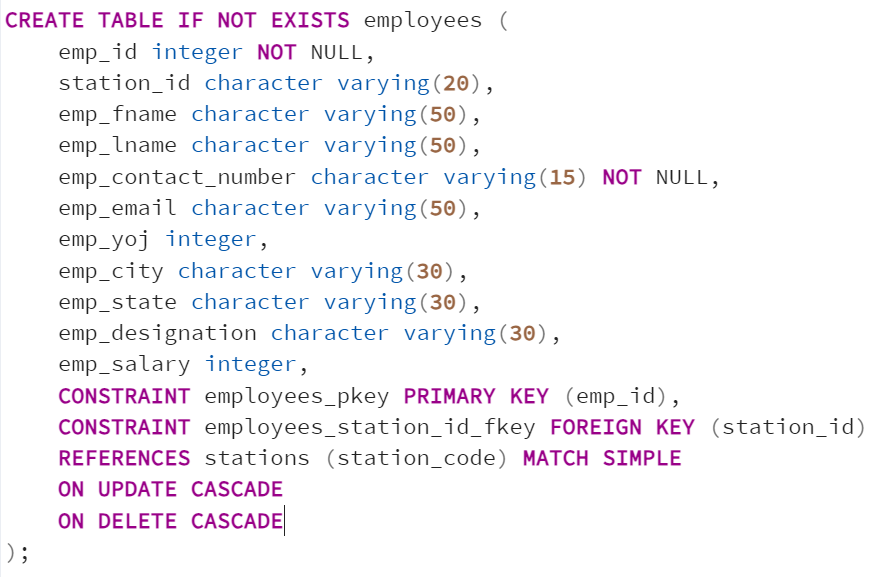
1. Stations



1. Passengers



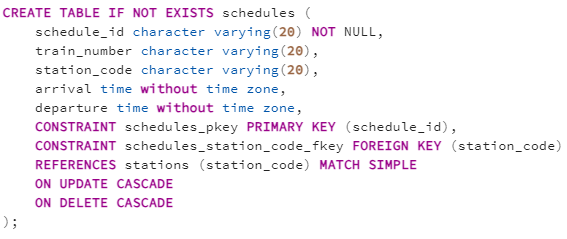
1. Employees



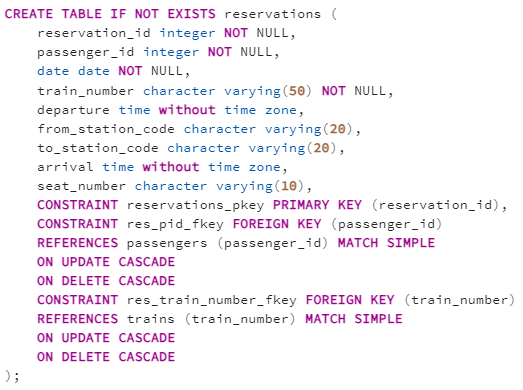
1. Trains



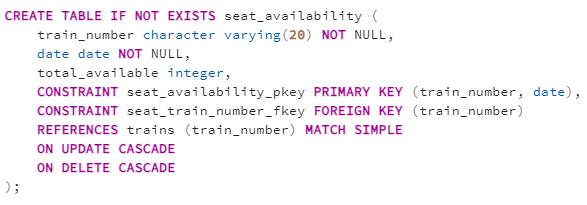
1. Schedules



1. Reservations

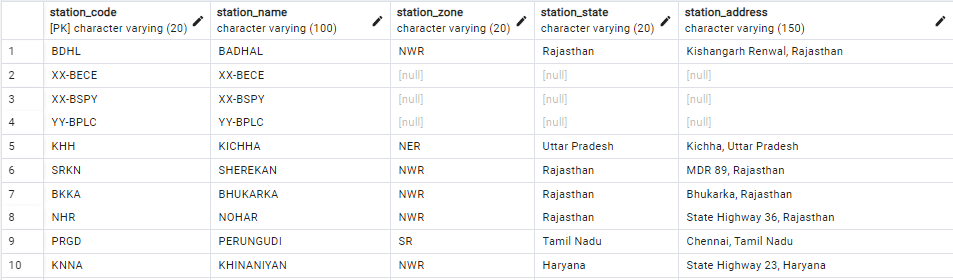


1. Seat\_availability

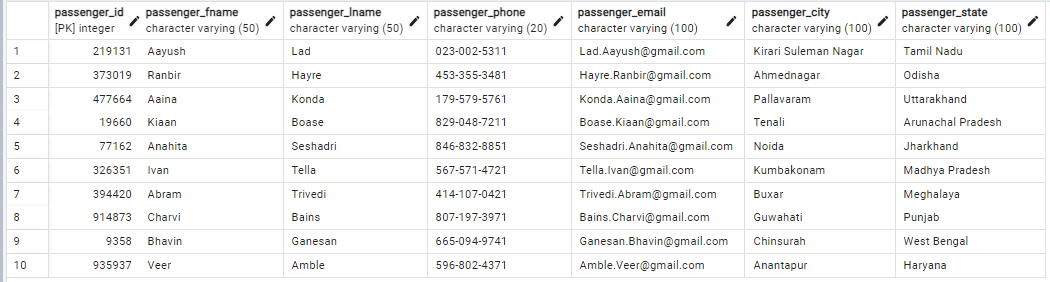


## Inserting Data

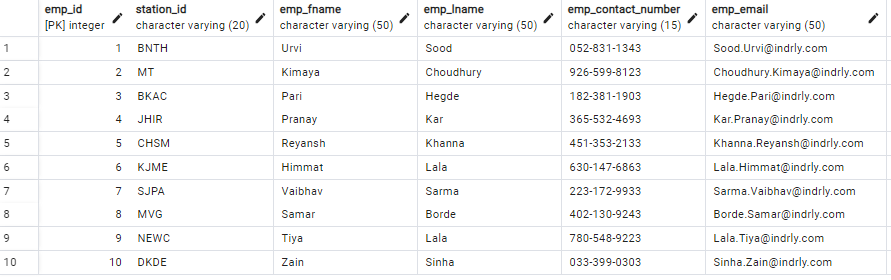
* Stations – 8990 records



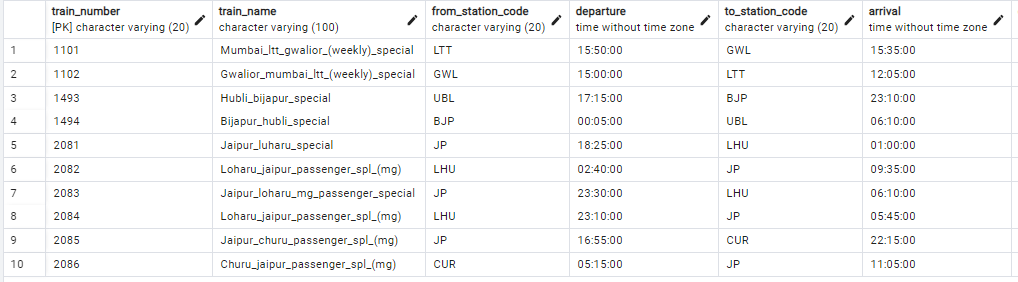
* Passengers – 500 records



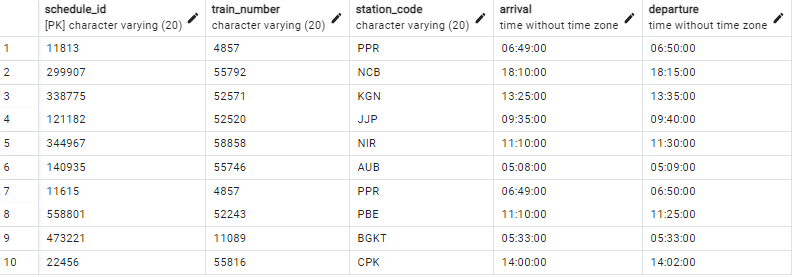
* Employees – 2999 records



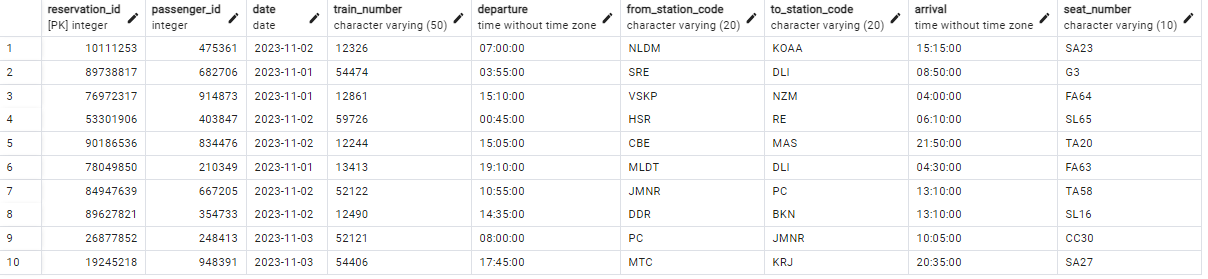
* Trains – 5206 records



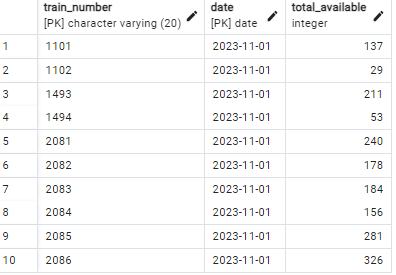
* Schedules – 417080 records



* Reservations – 100 records



* Seat\_Availability – 36442 records



# Normalization

Normalization is a process of organizing data in a database to reduce redundancy and dependency. The main goal of normalization is to ensure that each table in a database only represents one specific type of data, with each data point represented in only one place. There are several levels of normalization, including First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF). BCNF is a more advanced level of normalization that addresses certain types of functional dependencies that cannot be handled by 3NF.

BCNF is important because it eliminates all anomalies that can occur due to functional dependencies between attributes that are not part of any candidate key. This ensures that the database is free from any update, insertion, or deletion anomalies. A database in BCNF is well-structured and avoids data redundancies and inconsistencies. We say a relation R is in BCNF if whenever X ->A is a nontrivial FD that holds in R and X is a super key.

Let us check each of the tables to see if they satisfy BCNF –

* Stations

Table name: stations

Columns: station\_code, station\_name, station\_zone, station\_state, station\_address

Primary key: station\_code

Functional Dependencies:

Station\_code -> station\_name, station\_zone, station\_state, station\_address

In the above scenario, the FD is a non-trivial FD and station\_code is the primary key (hence a part of super key). So, the table "stations" is in BCNF.

* Passengers

Table Name: passengers

Columns: passenger\_id, passenger\_fname , passenger\_lname , passenger\_phone , passenger\_email , passenger\_city , passenger\_state

Primary key: passenger\_id

Functional Dependencies:

Passenger\_id -> passenger\_fname , passenger\_lname , passenger\_phone , passenger\_email , passenger\_city , passenger\_state

In the above scenario, the FD is a non-trivial FD and passenger\_id is the primary key (hence a part of super key). So, the table "passengers" is in BCNF.

* Employees

Table name: employees

Columns: emp\_id, station\_id, emp\_fname, emp\_lname, emp\_contact\_number, emp\_email, emp\_yoj, emp\_city, emp\_state, emp\_designation, emp\_salary

Primary key: emp\_id

Functional Dependencies:

emp\_id -> station\_id, emp\_fname, emp\_lname, emp\_contact\_number, emp\_email, emp\_yoj, emp\_city, emp\_state, emp\_designation, emp\_salary

In the above scenario, the FD is a non-trivial FD and emp\_id is the primary key (hence a part of super key). So, the table "employees" is in BCNF.

* Trains

Table name: trains

Columns: train\_number, train\_name, from\_station\_code, departure, to\_station\_code, arrival, distance, duration\_hours, duration\_minutes, sleeper, chair\_car, first\_class, first\_ac, second\_ac, third\_ac, return\_train, train\_type, total\_seats

Primary key: train\_number

Functional Dependencies:

Train\_number -> train\_name, from\_station\_code, departure, to\_station\_code, , arrival, distance, duration\_hours, duration\_minutes, sleeper, chair\_car, first\_class, first\_ac, second\_ac, third\_ac, return\_train, train\_type, total\_seats

In the above scenario, the FD is a non-trivial FD and train\_number is the primary key (hence a part of super key). So, the table "trains" is in BCNF.

* Schedules

Table name: schedules

Columns: schedule\_id, train\_number, station\_code, arrival, departure

Primary key: schedule\_id

Funtional Dependencies:

Schedule\_id -> train\_number, station\_code, arrival, departure

In the above scenario, the FD is a non-trivial FD and schedule\_id is the primary key (hence a part of super key). So, the table "schedules" is in BCNF.

* Reservations

Table name: reservations

Columns: reservation\_id, passenger\_id, date, train\_number, departure, from\_station\_code, to\_station\_code, arrival, seat\_number

Primary key: reservation\_id

Functional Dependencies:

reservation\_id -> passenger\_id, date

Train\_number -> train\_number, departure, from\_station\_code, arrival, to\_station\_code, seat\_number

In the above scenario, the FDs is a non-trivial FDs and reservation\_id is the primary key (hence a part of super keys). So, the table "reservations" is in BCNF.

* Seat\_availability

Table name: seat\_availability

Columns: train\_number, date, total\_available

Primay key: train\_number, date

Functional Dependencies:

Train\_number -> total\_seats, total\_available

Date is already a primary key

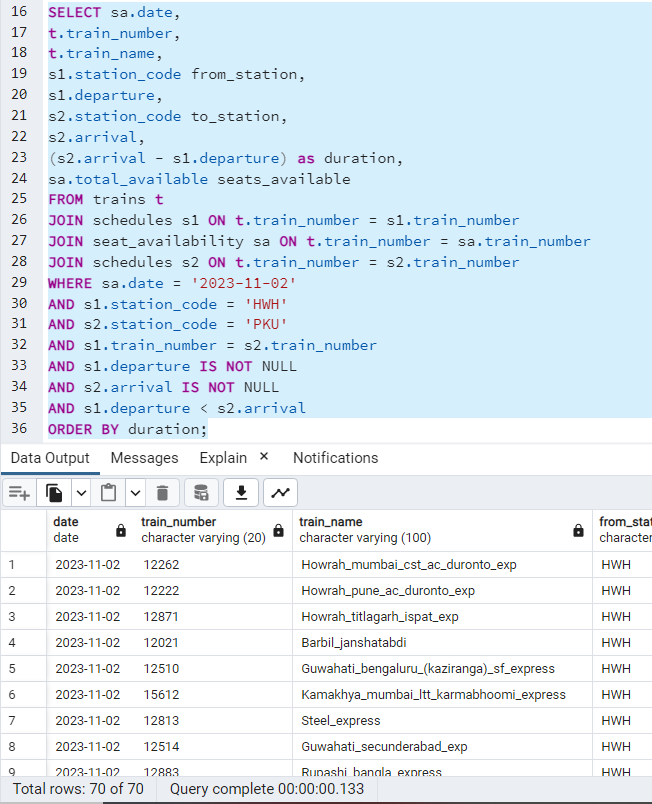
In the above scenario, the FD is a non-trivial FD and train\_number and date are the primary keys (hence a part of super key). So, the table "seat\_availability" is in BCNF.

# PROBLEMS WHILE HANDLING LARGE DATA

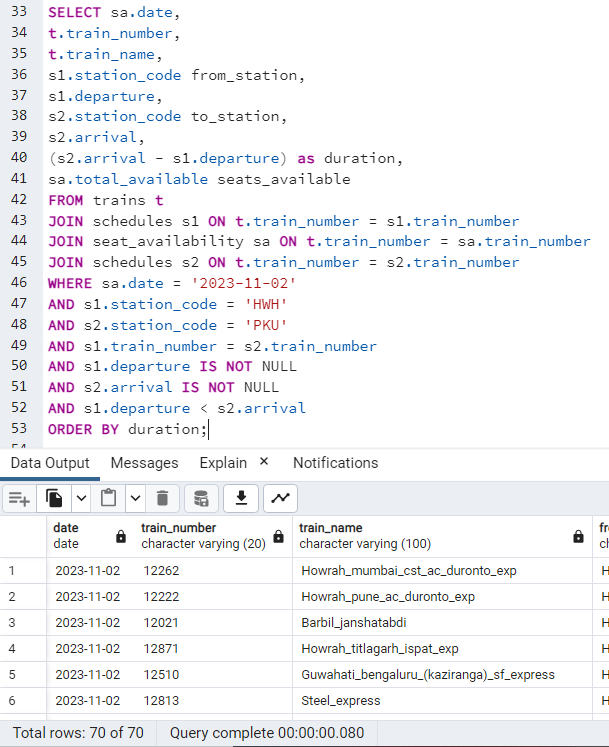
The database consists of huge tables with thousands of rows and often times it is difficult to query large tables due to memory and storage limitations. One such problem we faced while handling the large data is while running a query to view all the trains between two stations on a specific date. This query requires us to join three tables – trains, schedules, and seat\_availability. In addition to the above mentioned inner join, we also need to use a self join between two instances of schedules table. To put the enormity into perspective, the trains table consists of 5000 rows, seat\_availability has circa 36000 rows, and the schedules table consists of 417K rows. To make sure that the query returns an output faster, we have

1. Tried to use only primary keys to join the tables
2. Selected only required columns
3. Avoided using subqueries

Though we wrote the query using the above techniques, it took around 133 msec to retrieve 70 rows .



Hence, we have chosen to create an index on the station\_code column of the schedules table. This technique has reduced the time taken for the same query to run by 40%. Below is the screenshot of the same query after the index has been created. More on query optimization will be discussed in the following sections of this document.



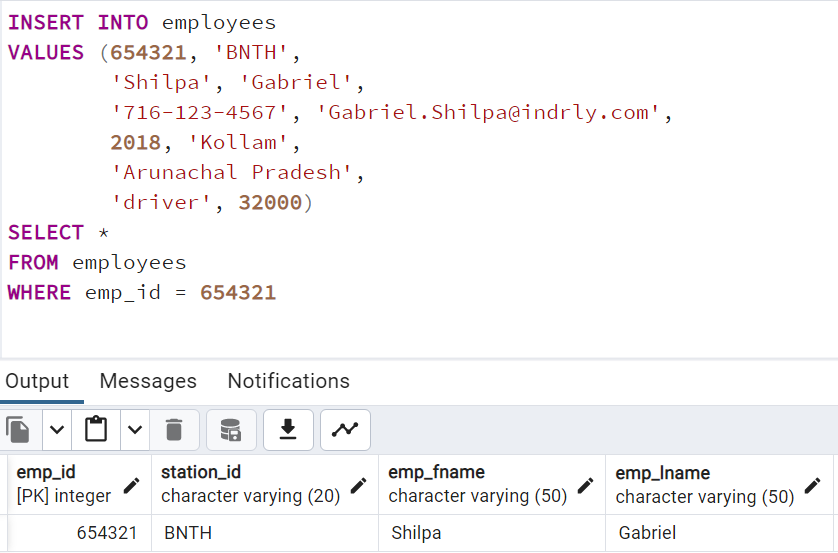
# Queries

## INSERT Queries

1. Insert into passengers table



1. Insert into employees table



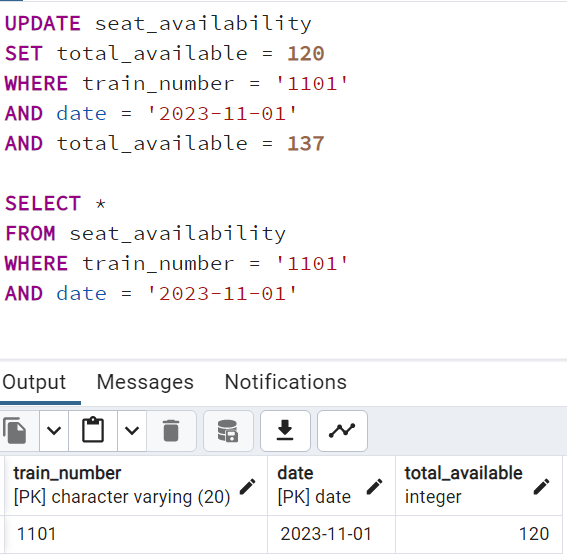
## UPDATE Queries

1. Update the station\_code in the stations table



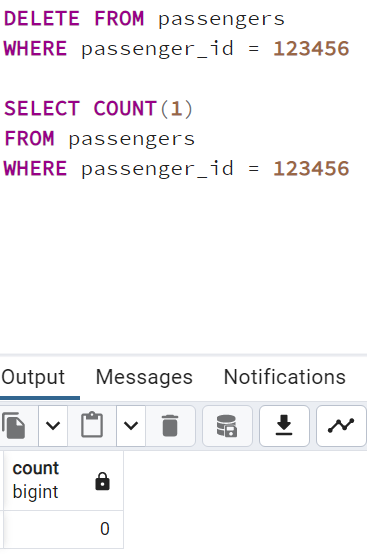
Also, station\_code is considered a foreign key in employees, trains, and schedules. And, as we have seen above, they all have the update cascade constraints. It means that, if any value is updated in the station\_code column in the stations table, it must update the value in the corresponding column in the three tables.

1. Update seat\_availability table

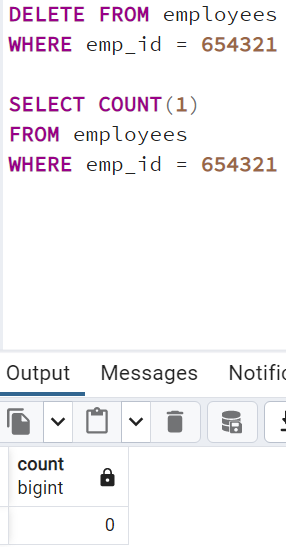


## DELETE Queries

1. Delete from passengers table



1. Delete from employees table



## SELECT Queries

1. Retrieve the station name, station code along with count of trains starting from that station. Select only those stations that have more than 100 trains starting from there.

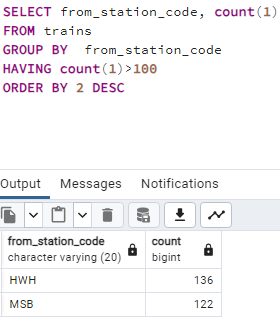
*SELECT from\_station\_code, count(1)*

*FROM trains*

*GROUP BY  from\_station\_code*

*HAVING count(1)>100*

*ORDER BY 2 DESC*



1. For a given day, select the reservation, passengers, and train details

*SELECT*

*r.reservation\_id,*

*r.passenger\_id,*

*p.passenger\_fname,*

*p.passenger\_lname,*

*r.train\_number,*

*t.train\_name,*

*r.date,*

*r.departure,*

*r.from\_station\_code,*

*r.seat\_number*

*FROM*

*reservations r*

*JOIN*

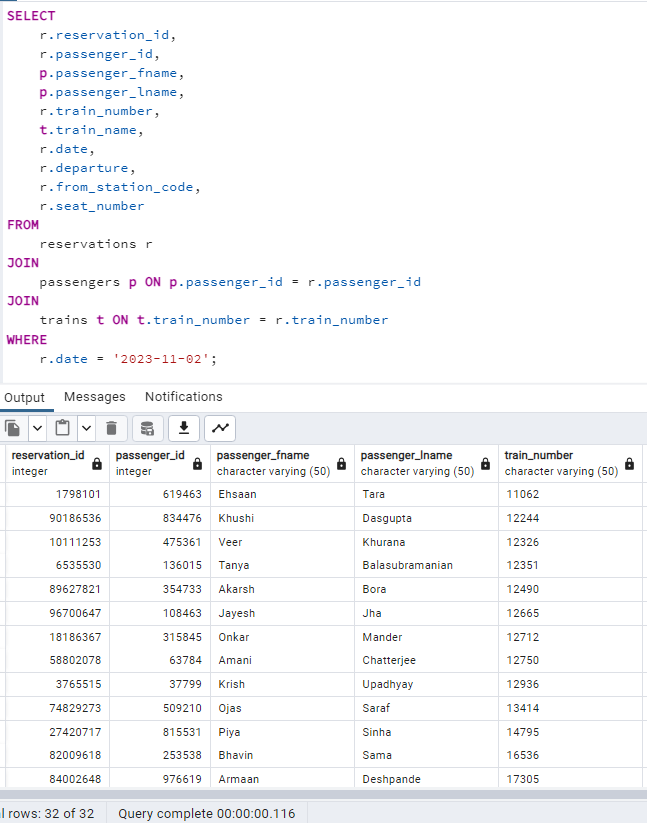
*passengers p ON p.passenger\_id = r.passenger\_id*

*JOIN*

*trains t ON t.train\_number = r.train\_number*

*WHERE*

*r.date = '2023-11-02';*



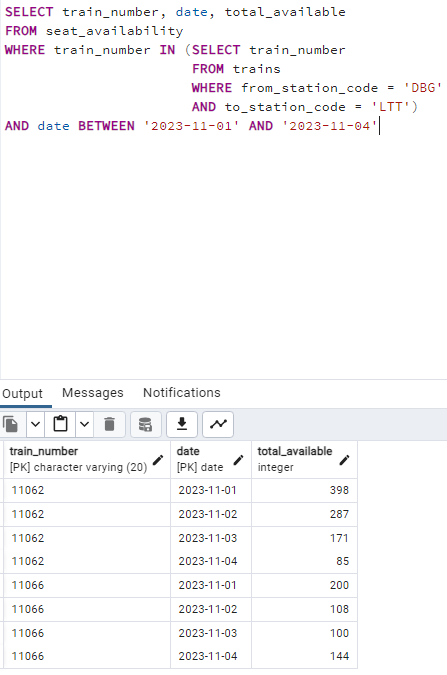
1. Given a range of dates, departure and arrival stations, retrieve the train numbers and available seats

*SELECT train\_number, date, total\_available*

*FROM seat\_availability*

*WHERE train\_number IN (SELECT train\_number FROM trains WHERE from\_station\_code = 'DBG' AND to\_station\_code = 'LTT')*

*AND date BETWEEN '2023-11-01' AND '2023-11-04'*



1. How many stations do the top 10 trains that travel the most distance stop at?

*WITH TB1 AS(*

*SELECT train\_number,*

*train\_name,*

*from\_station\_code,*

*to\_station\_code,*

*distance*

*FROM trains*

*WHERE distance IS NOT NULL*

*ORDER BY distance DESC*

*LIMIT 10)*

*SELECT TB1.train\_number,*

*TB1.train\_name,*

*TB1.from\_station\_code,*

*TB1.to\_station\_code,*

*TB1.distance,*

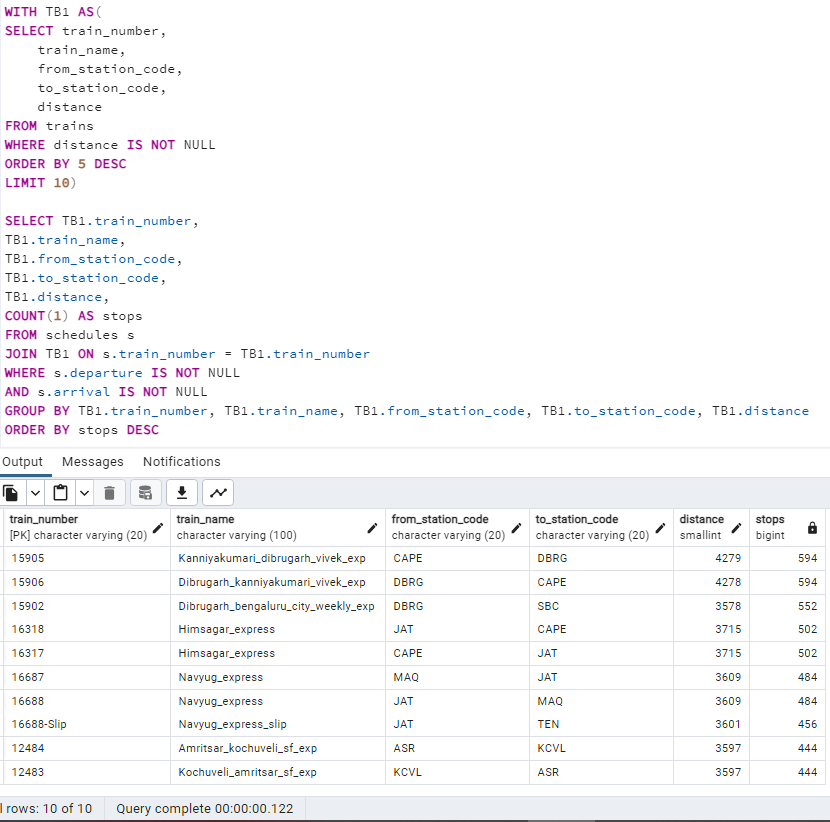
*COUNT(1) AS stops*

*FROM schedules s*

*JOIN TB1 ON s.train\_number = TB1.train\_number*

*GROUP BY TB1.train\_number, TB1.train\_name, TB1.from\_station\_code, TB1.to\_station\_code, TB1.distance*

*ORDER BY stops DESC*

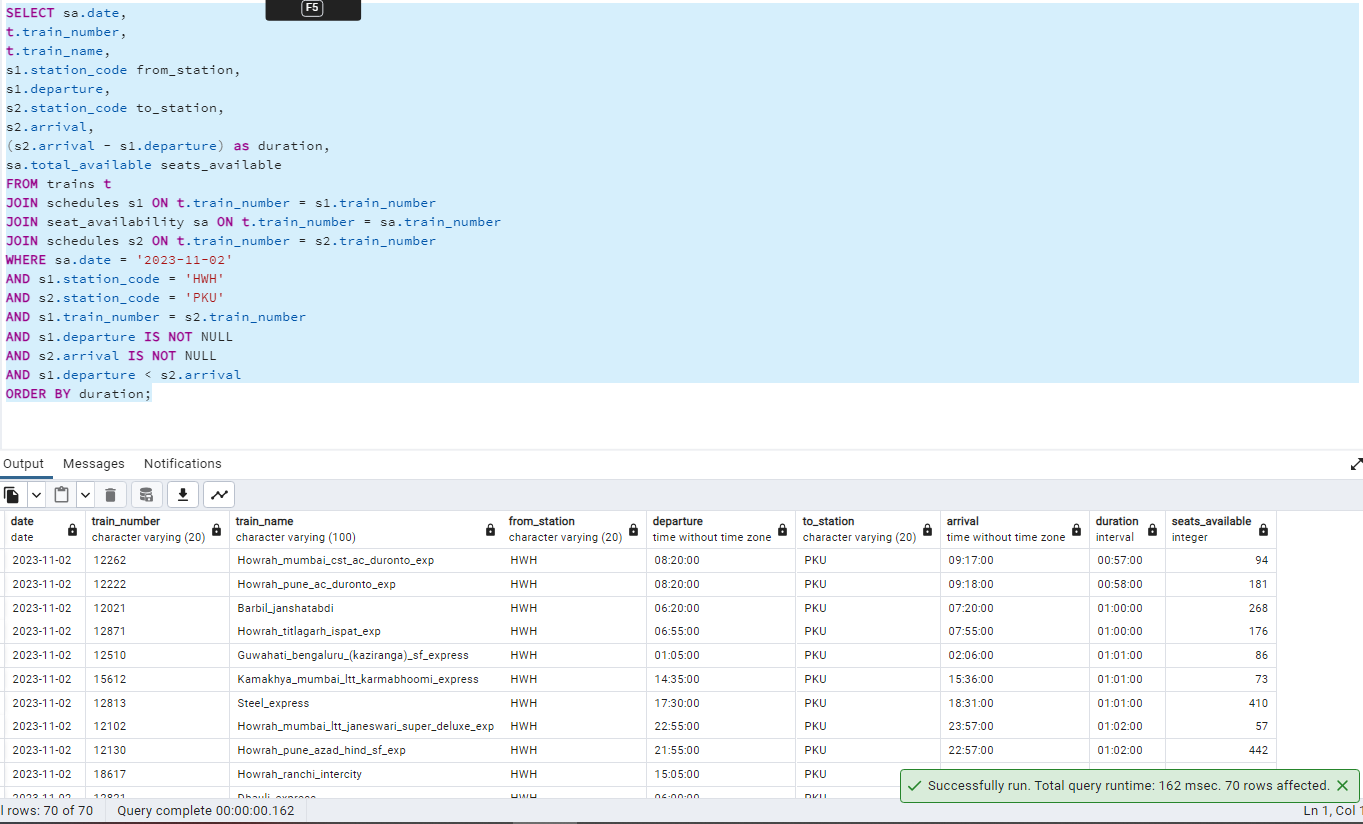


# QUERY OPTIMIZATION USING INDEXING

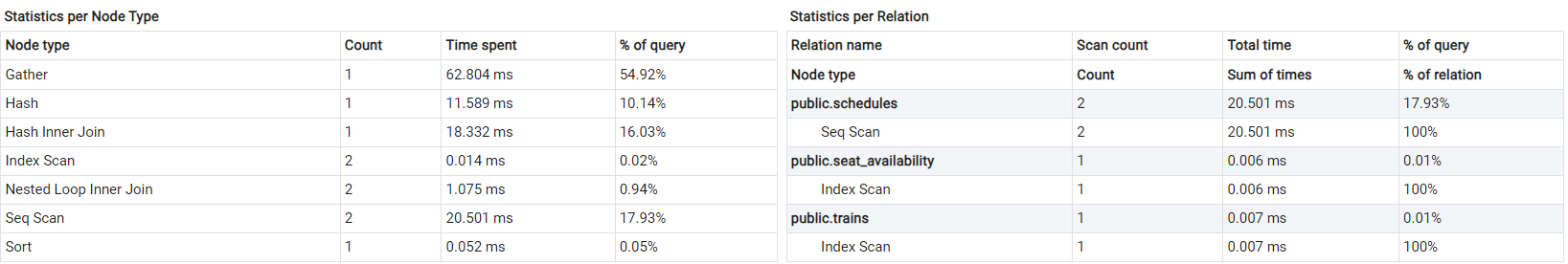
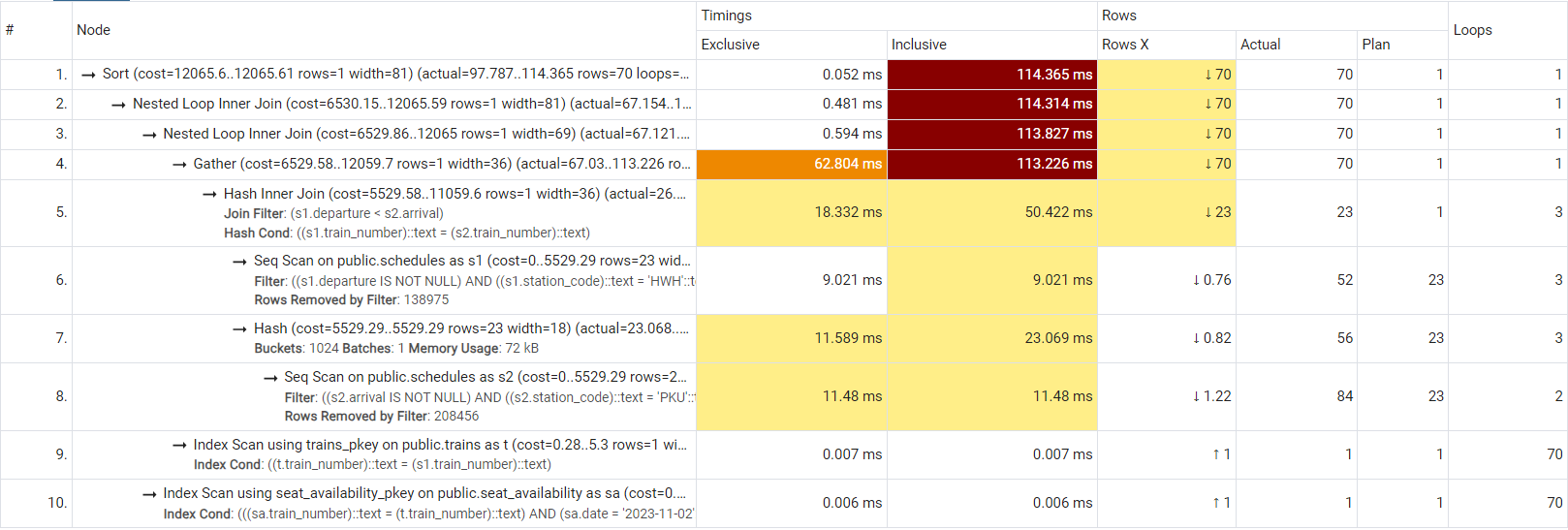
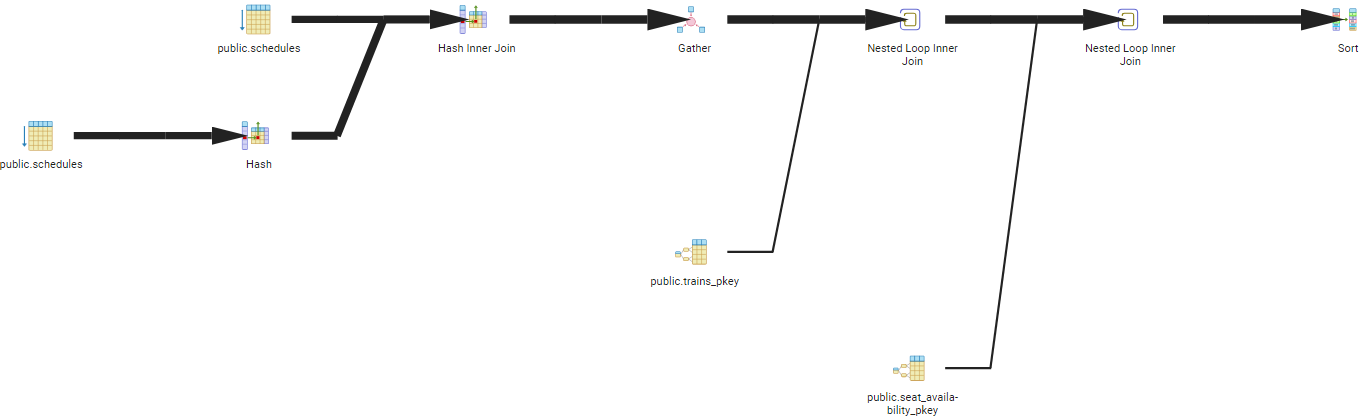
Indexing is a critical component of query optimization in relational databases, as it accelerates data retrieval by enabling quick lookups, reducing disk I/O, and improving query performance, especially when used in WHERE clauses and join conditions. Indexes also support unique constraints and enhance sorting and range queries. However, they come with storage and maintenance overhead, requiring careful selection of indexed columns and periodic maintenance to prevent fragmentation. Striking a balance between indexing benefits and associated costs is essential for efficient database performance.

## Query 1: For a given date, to and from station codes, retrieve all the available trains and the total seats available on each train.

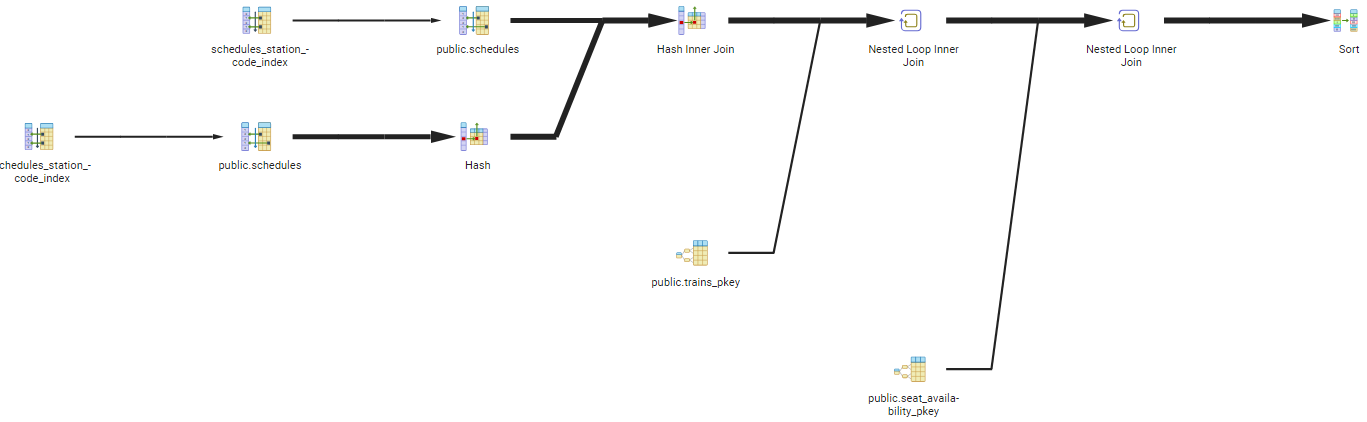
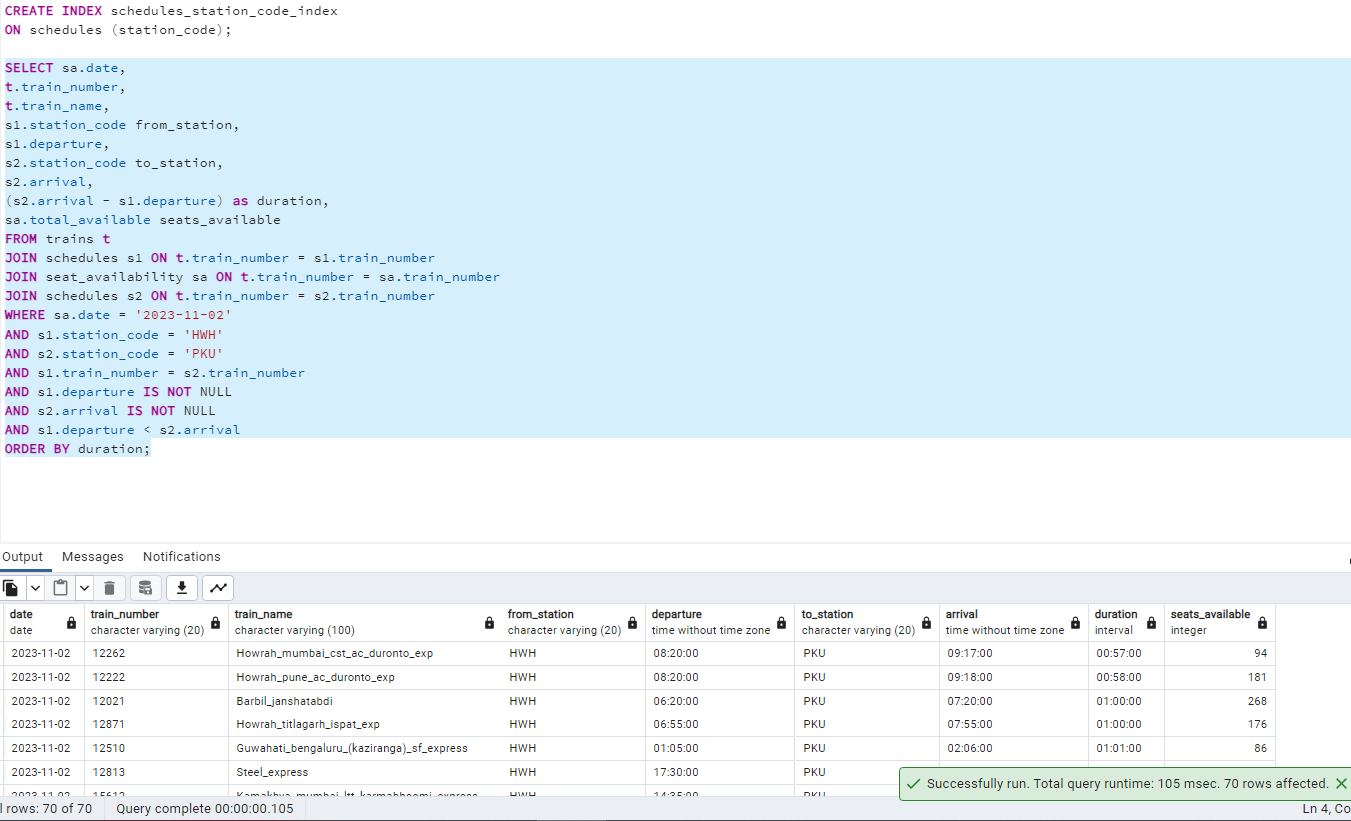
As we can see below, the query consists of an inner join between trains table and schedules table to retrieve the train numbers, another inner join between trains table and seat\_availability table to retrieve the seats available, and a self join on the schedules table. The query returns 70 rows in 162 msec. We can see that the query uses train\_number column to join the trains and schedules table. However, schedule\_id, which is the primary key of schedules table isn’t being used here.



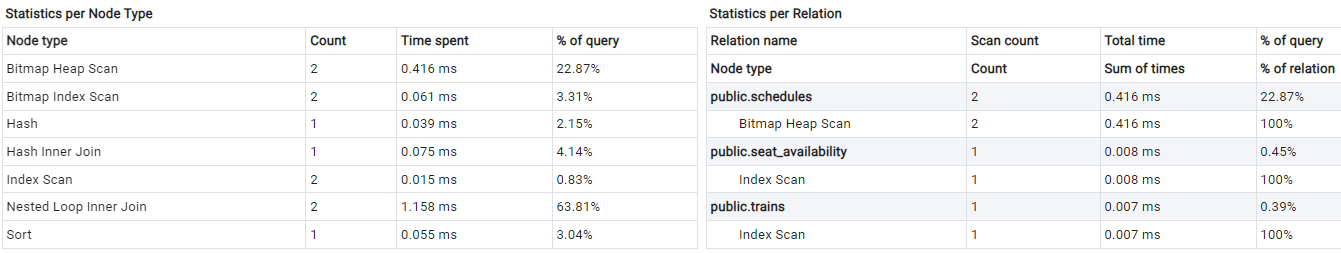
We can observe the below query plan – which shows us that there is a self join between two instances of schedules table, then an inner join with trains, and then an inner join with seats\_availability.



We have created an index on the station\_code column of schedules table as shown below. We see that creation of this index has changed the query plan and greatly reduced the time taken to execute the query. The query has been executed in 105msec which is 35% improvement.

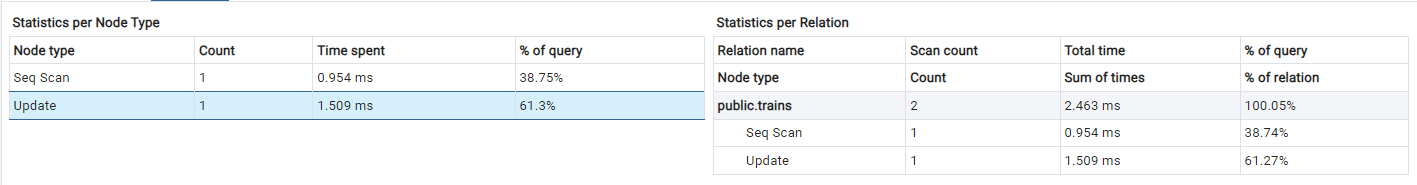
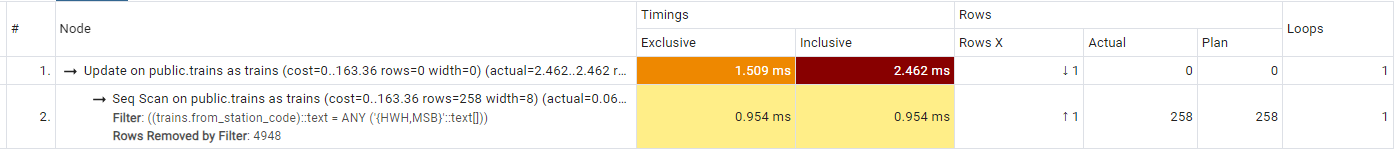
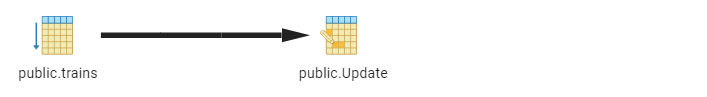
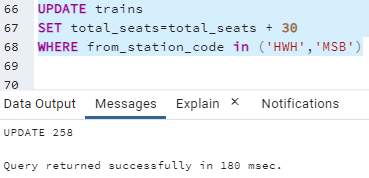


We notice that time taken to sort the data before indexing was 114 msec and it has been reduced to a 1msec now.

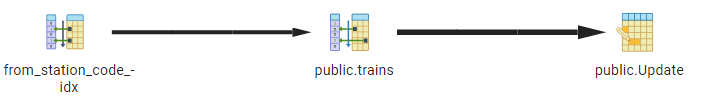
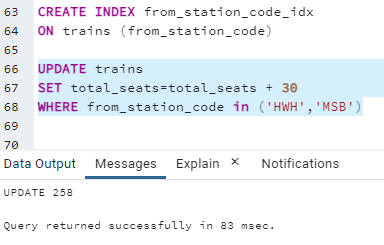


## Query 2: For all trains starting from staion ‘HWH’ or ‘MSB’, increase the number of seats in the train by 30

The query seems to be straightforward as it involves only one table – trains. However, we need to notice that it uses the from\_station\_code column to update the total\_seats. Since, the query is not hitting the primary key (train\_number) of the table, this update takes a lot of time as shown in below screenshot. The query below updates 258 rows in 180msec.



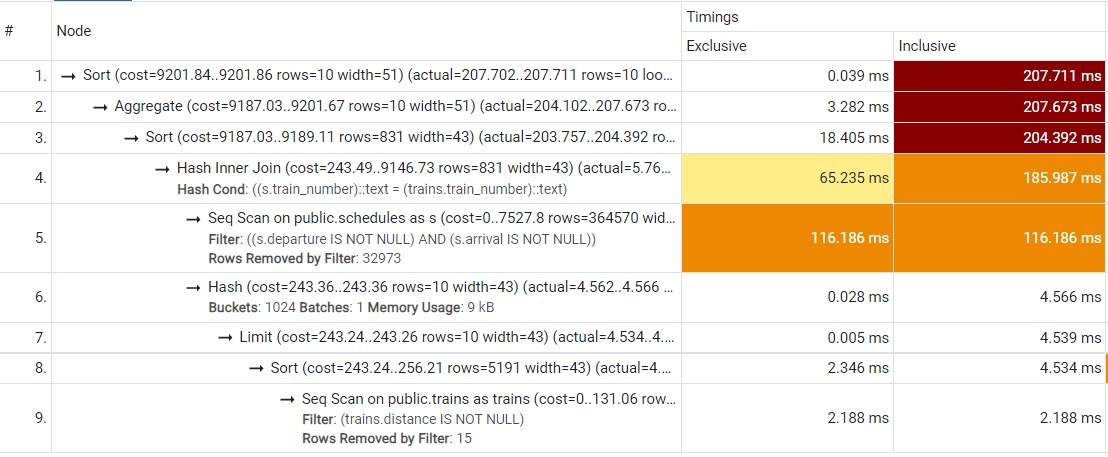
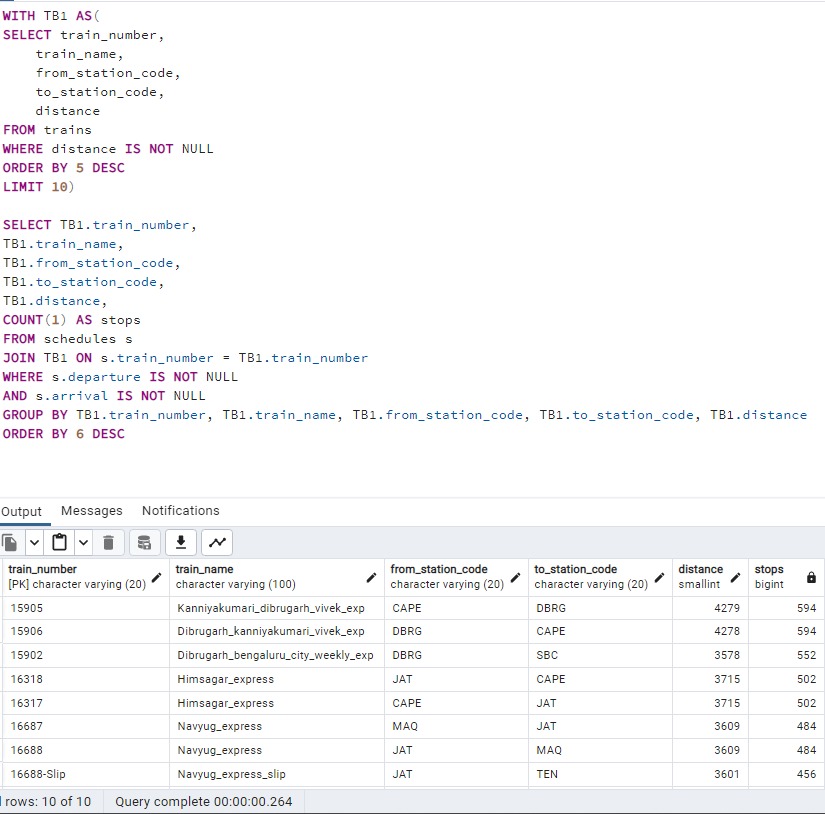
We have created an index on the from\_station\_code column of trains table as shown below. We see that creation of this index has changed the query plan and greatly reduced the time taken to execute the query. The query has been executed in 83msec which is 54% improvement.



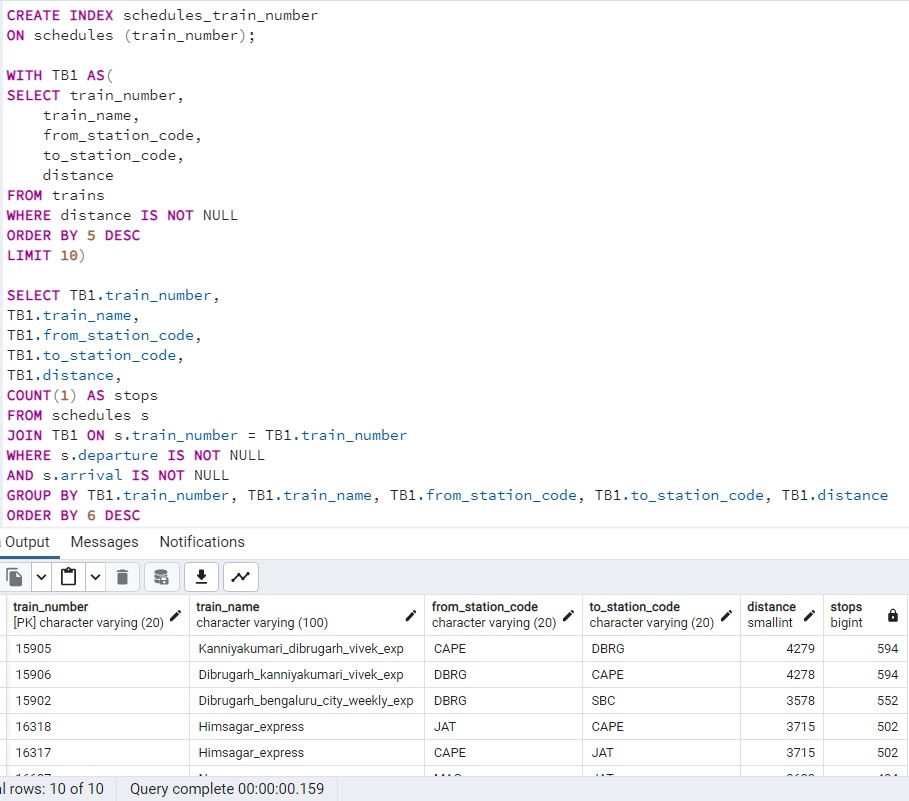
We have also noticed that time taken to scan the table has reduced.

## Query 3: How many stations do the top 10 trains that travel the most distance stop at?

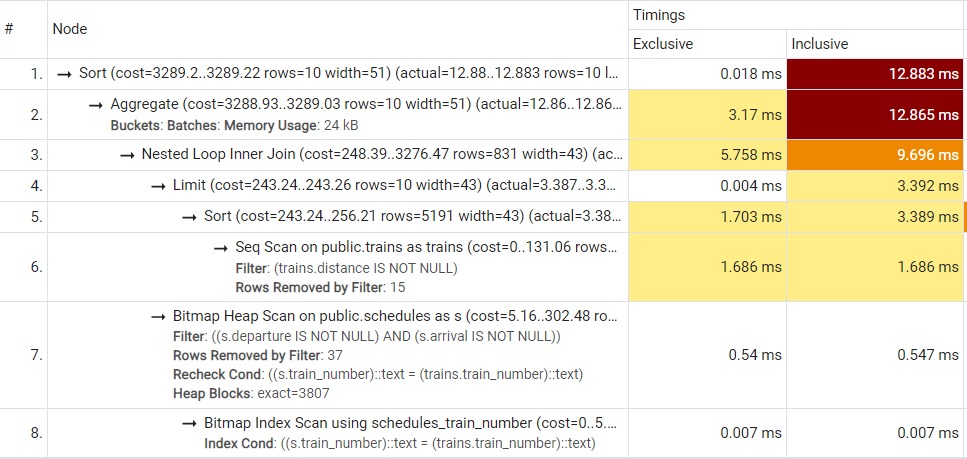
To execute this query, we have used a CTE named TB1 that retrieves train details and sorts them in descending order based on the distance travelled. Then the second query adds a new “Stops” column to count the number of stops each of these 10 trains take throughout their journey. This query has taken 264msec to retrieve 10 rows. Below is the query:



We have created an index on train\_number column of the schedules table to reduce the time taken by the query. We see that it has improved the query performance and reduced the running time to 159msec, which is 40% improvement.

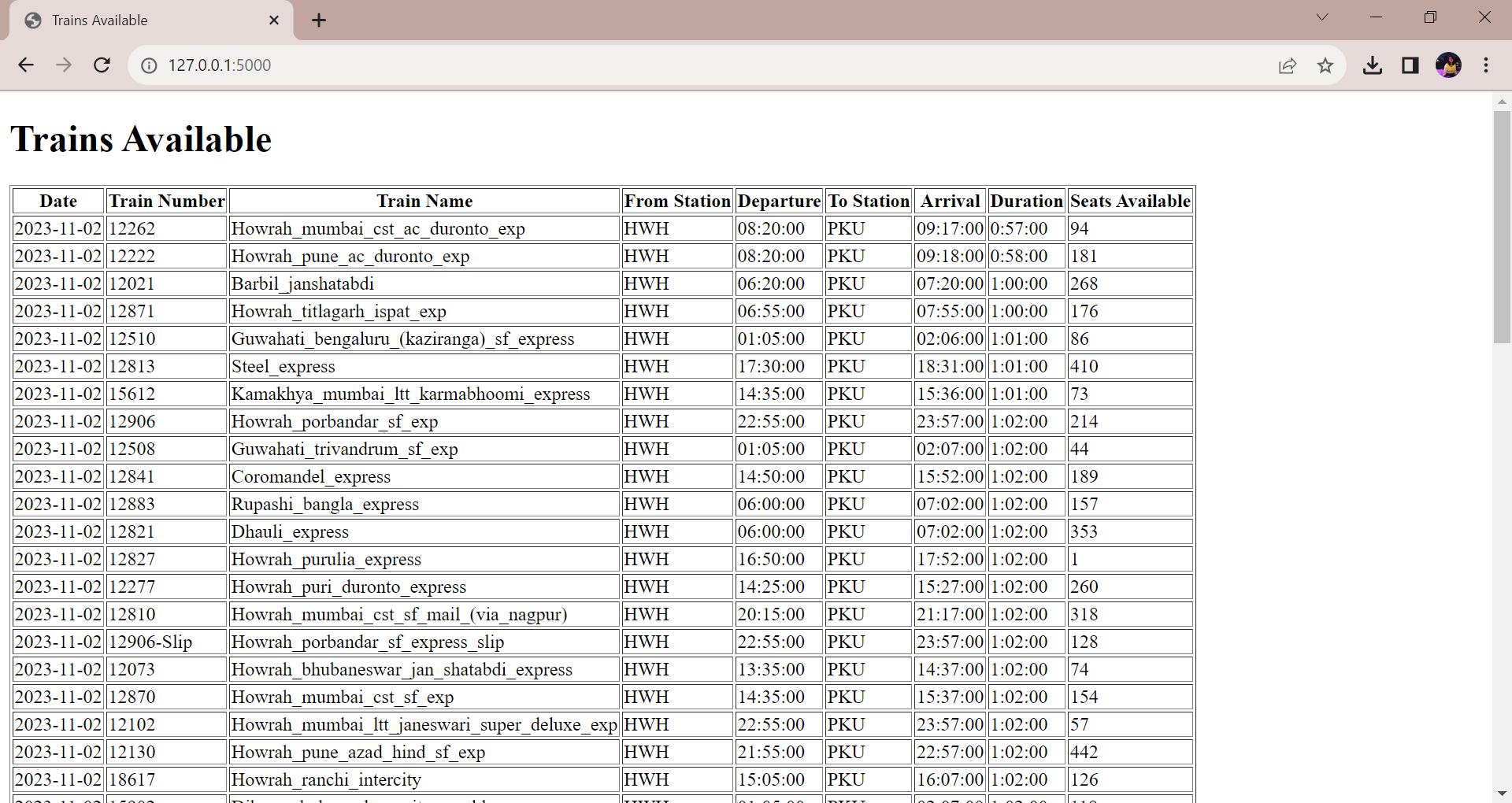
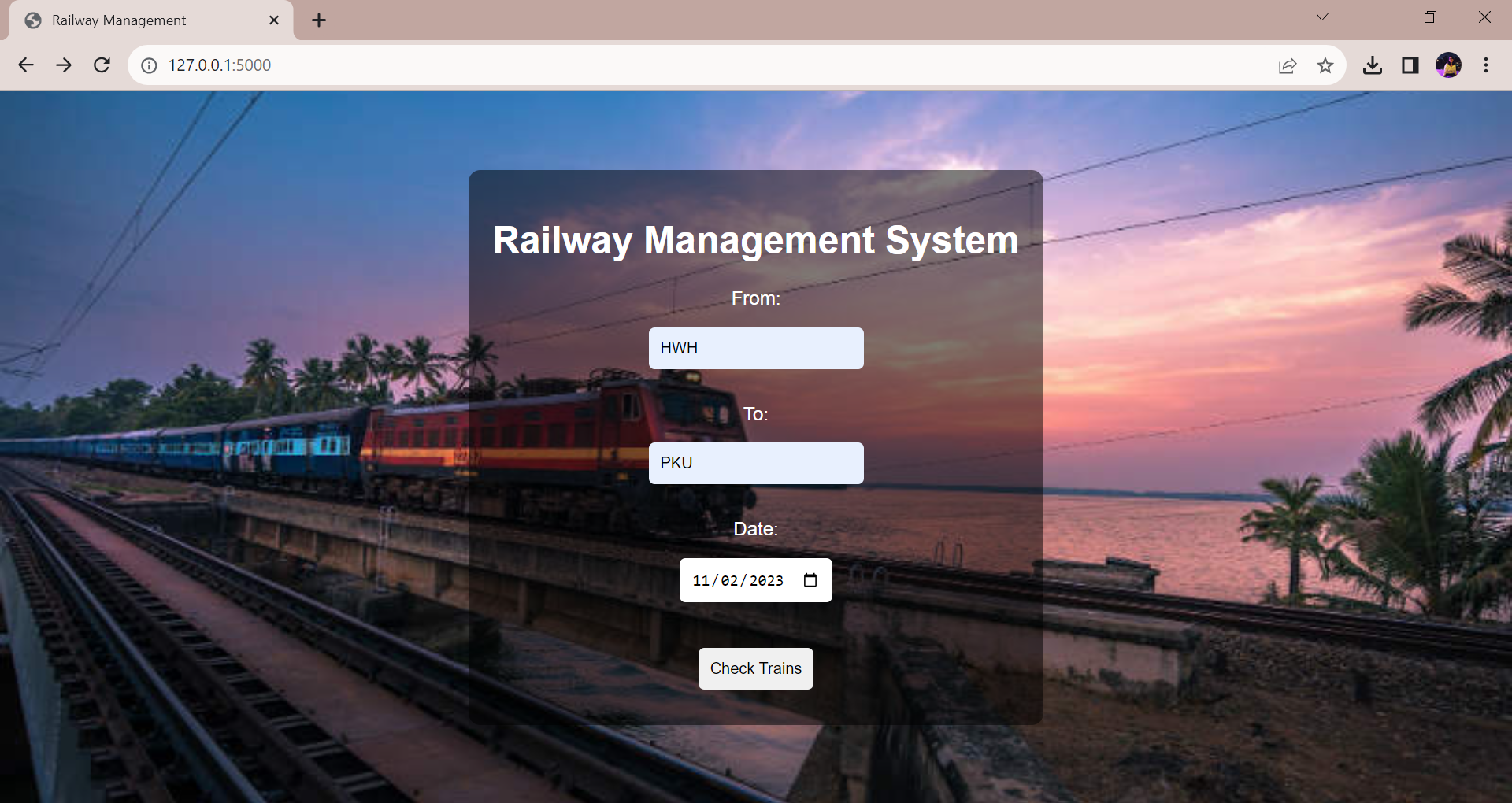


Also, we notice that the time taken to sort the results has been reduced greatly from 207msec to just 12msec which is 94% improvement.



# User Interface

We have created a UI for passengers to check train availability between two stations on a particular date. Along with the trains it also displays the seats available on the trains for that date. The website can be accessed through the URL http://127.0.0.1:5000/ as the page is local hosted.



# Conclusion

Based on the observations above, we can conclude that the Railway Management System is very helpful to the end users such as a passenger trying to check trains and reserve a seat, or a Railway Management employee who is trying to check details about any employees or trains. This database also helps to gain valuable insights that help in achieving operational efficiency of the Railways System.

# contribution

|  |  |
| --- | --- |
| Name | Contribution |
| Shilpa Gabriel |  |
| Neelima Srilakshmi Bollempalli |  |
| Venkata Lakshmi Krishna Tejaswi Gudimetla |  |