Popular Destinations in Russia

CSE 460/560 Data Models and Query Language Semester Project
Phase - 2 Report

Priya Ghanshyambhai Patel
M.S in ES (Data Science)
priyagha@buffalo.edu

Sam Anderson
B.S in Computer Science
sranders@buffalo.edu

Shweta Mukund Hatote
M.S in ES (Data Science)
shwetamu@buffalo.edu

Abstract – The growing popularity of tourism in Russia necessitates a comprehensive exploration of flight data to identify the country's most coveted destinations. This project aims to create a robust database and an advanced query system for aggregating, managing, and analyzing flight, ticketing, and booking data, providing valuable insights for stakeholders in the tourism and airline industries, local businesses, and government entities.

INTRODUCTION

Tourism is pivotal for Russia's economic and cultural vitality, featuring diverse attractions from Moscow and St. Petersburg to the natural wonders of Siberia. However, the absence of a centralized system poses challenges, impeding data-driven decision-making in route planning, pricing, and marketing. This gap hampers resource optimization, leading to empty seats and oversubscribed routes for airlines. Addressing these challenges is crucial for equitable economic development, as identifying and promoting attractive destinations can ensure a more balanced distribution of benefits. Lastly, a comprehensive understanding of popular destinations enhances the overall travel experience, fostering improved customer satisfaction and engagement with Russia's diverse tourism offerings.

IMPLEMENTATION

1. Data Understanding:

Understand the dataset, including the nature of flight and airport data.

Identify key entities, attributes, and relationships within the dataset.

Determine the purpose of the database and the types of queries and updates it needs to support.

2. Data Preprocessing and Cleaning:

The data was already cleaned, preprocessed, and standardized. Hence, there was no need to implement this step.

3. Create tables:

Based on the understanding of the dataset, we created SQL tables for each entity, defining appropriate data types and constraints.

Ensuring the tables capture the relationships between entities using foreign keys.

- ***Bookings*** – This table describes the booking details (book_ref, book_data, total_amount)

- ***Tickets*** – This table describes the details mentioned on the tickets (ticket_no, book_ref, passenger_id, passenger_name, contact_data)

- ***Airports_data*** – This table describes the details of the airport (airport_code, airport_name, city, co-ordinates, timezone)

- ***Flights*** – This table describes the details of all the flights in respective to the airports (flight_id, flight_no, scheduled_departure, scheduled_arrival, departure_airport, arrival_airport, status, aircraft_code, actual_departure, actual_arrival)

- ***Ticket_flights*** – This table describes the details of the flight-ticket details (ticket_no, flight_id, fare_conditions, amount)

- ***Aircrafts_data*** – This table describes the details of the aircraft (aircraft_code, model, range)

- ***Boarding_passes*** – This table describes the details of the boarding passes given to the passengers (ticket_no, flight_id, boarding_no, seat_no)

- ***Seats*** – This table describes the details of the seat allotted (aircraft_code, seat_no, fare_condition)

4. Data Normalization:

The dataset used was already normalized. Below is the BCNF analysis to support the fact that the dataset is normalized.

IV. BCNF Analysis

Relation: bookings

A: book_ref

B: book_date

C: total_amount

FD: A->BC

A+ = ABC, A is a superkey

Bookings is in BCNF

Relation: tickets

A: ticket_num

B: book_ref

C: passenger_id

D: passenger_name

E: contact_data

FD: A->BCDE, C->ABDE

A+ = ABCDE, A is a superkey

C+ = ABCDE, C is a superkey

Tickets is in BCNF

Relation: ticket_flights

A: ticket_no

B: flight_id

C: fare_conditions

D: amount

FD: A->BCD

A+ = ABCD, A is a superkey

Ticket_flights is in BCNF

Relation: boarding_passes

A: ticket_num

B: flight_id

C: boarding_num

D: seat_num

FD: A->BCD

A+ = ABCD, A is a superkey

Boarding_passes is in BCNF

Relation: flights

A: flight_id

B: flight_num

C: scheduled_departure

D: scheduled_arrival

E: departure_airport

F: arrival_airport

G: status

H: aircraft_code

I: actual_departure

J: actual_arrival

FD: A->BCDEFGHIJ

A+ = ABCDEFGHIJ, A is a superkey

flights is in BCNF

Relation: airports_data

A: airport_code

B: airport_name

C: city

D: coordinates

E: timezone

FD: A->BCDE, B->ACDE

A+ = ABCDE, A is a superkey

B+ = ABCDE, B is a superkey

airports_data is in BCNF

Relation: aircrafts_data

A: aircraft_code

B: model

C: range

FD: A->BC

A+ = ABC, A is a superkey

aircrafts_data is in BCNF

Relation: seats

A: aircraft_code

B: seat_no

C: fare_conditions

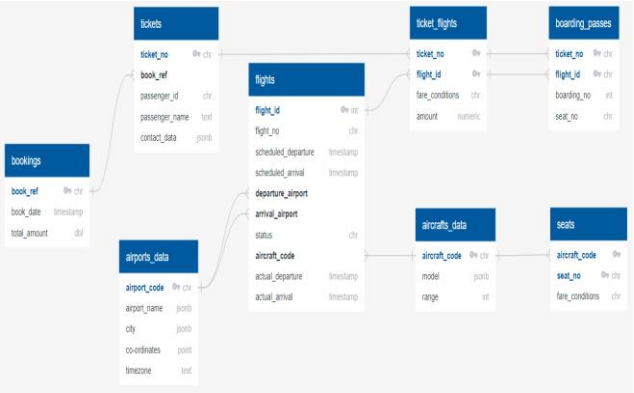
FD: A->BC

A+ = ABC, A is a superkey

seats is in BCNF

5. ER Diagram & Relation:

Identified the connections between tables by recognizing relationships and foreign key dependencies. Built a relational schema based on the identified entities and their relationships. Applied constraints wherever necessary, to maintain data integrity.



6. Queries Implemented:

To better understand the database and the relationship between the tables, we implemented different queries.

Below are the queries we implemented and the output of those queries.

- ***Insert Queries***

In the given query, a new seat entry with seat number '9G' and the fare condition 'Economy' is inserted into the 'seats' table for the aircraft with the code '319'. Following the insertion, a SELECT statement retrieves and displays the last 10 seat entries for the same aircraft, ordered

by seat number in descending order. This allows for a verification of the successful insertion and provides insight into the existing seat data for the specified aircraft.

Inserting in seats relation

```
Active Connection
-- Active: 1702008639736@@127.0.0.1@5432@airport@bookings
Execute
INSERT into seats values('319', '9G', 'Economy');
```

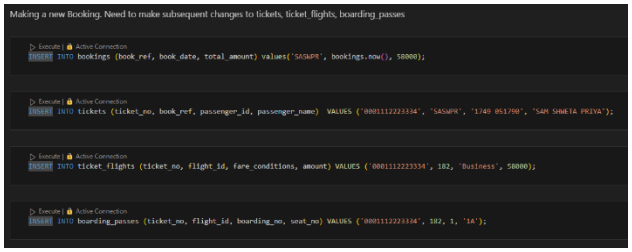
```
Active Connection
# Checking if values are inserted
Execute
SELECT * FROM seats
WHERE aircraft_code = '319'
ORDER BY seat_no DESC
LIMIT 10;
```

Output –

	aircraft_code character(3)	seat_no varchar(4)	* fare_conditions varchar(10)
1	319	9G	Economy
2	319	9F	Business
3	319	9E	Economy
4	319	9D	Economy
5	319	9C	Economy
6	319	9B	Economy
7	319	9A	Economy
8	319	8F	Economy
9	319	8E	Economy
10	319	8D	Economy

Here we can see that a new seat information has been inserted in the “Seats” table.

Similarly, the below provided queries initiate a booking for 'SAM SHWETA PRIYA' on flight 182 in the 'Business' class. A booking reference 'SASWPR' is created with a total amount of 58000, and a corresponding ticket, ticket flight details, and boarding pass are recorded.



- ***Delete Queries***

As we were just trying to understand the overall scenario of the database, we tried out a few insertions into different tables. However, these are unrelated insertions and will affect the result. Thus, we then used the deleted operation to delete these insertions. Below are two such queries written.

The below query deletes the seat entry with seat number '9G' for the aircraft '319' from the 'seats' table. Subsequently, a SELECT statement retrieves and displays the last 10 seat entries for the same aircraft, ordered by seat number in descending order. This action provides an overview of the remaining seat configuration after the specified deletion.

Deleting the inserted value in seats relation

```
Execute | Active Connection
DELETE FROM seats
WHERE aircraft_code = '319' and seat_no ='9G';
```

```
Active Connection
# Checking if values are deleted
Execute
SELECT * FROM seats
WHERE aircraft_code = '319'
ORDER BY seat_no DESC
LIMIT 10;
```

Output –

	aircraft_code character(3)	seat_no varchar(4)	* fare_conditions varchar(10)
1	319	9F	Business
2	319	9E	Economy
3	319	9D	Economy
4	319	9C	Economy
5	319	9B	Economy
6	319	9A	Economy
7	319	8F	Economy
8	319	8E	Economy
9	319	8D	Economy
10	319	8C	Economy

Similarly, the below queries sequentially remove records related to booking reference 'SASWPR': first deleting the boarding pass for ticket '0001112223334' on flight '182', then removing the ticket-flight association, followed by deleting the ticket entry associated with 'SASWPR'. Finally, the booking record with reference 'SASWPR' is deleted from the 'bookings' table, effectively canceling the entire booking.

Deleting the new booking we made, need to go backwards this time beacuse of the foreign keys constraint

```
Execute | Active Connection
DELETE FROM boarding_passes
WHERE ticket_no = '0001112223334' and flight_id = '182';

Execute | Active Connection
DELETE FROM ticket_flights
WHERE ticket_no = '0001112223334' and flight_id = '182';

Execute | Active Connection
DELETE FROM tickets
WHERE book_ref = 'SASWPR';

Execute | Active Connection
DELETE FROM bookings
WHERE book_ref = 'SASWPR';
```

- ***Select Queries*** –

Here we decided to run a few queries to achieve our goal as well as try to figure out a few other details about the relation schema.

Select #1

The below query retrieves distinct passenger names for tickets associated with Boeing 777-300 flights arriving in Moscow. It involves joining tables for tickets, ticket-flights, flights, aircraft data, and airport data, filtering based on the aircraft model ('Boeing 777-300') and the destination city ('Moscow').

Finding out the passenger who travelled to moscow by Boeing 777-300

```
Execute | Active Connection
SELECT DISTINCT t.passenger_name
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
JOIN flights f ON tf.flight_id = f.flight_id
JOIN aircrafts_data ad ON f.aircraft_code = ad.aircraft_code
JOIN airports_data a ON f.arrival_airport = a.airport_code
WHERE ad.model ->> 'en' = 'Boeing 777-300' AND a.city ->> 'en' = 'Moscow';
```

Output –

	passenger_name
1	ADELINA ANDREEVA
2	ADELINA EFIMOVA
3	ADELINA EGOROVA
4	ADELINA FOMINA
5	ADELINA GAVRILOVA
6	ADELINA ILINA
7	ADELINA IVANOVA
8	ADELINA KALININA
9	ADELINA KOROLEVA
10	ADELINA KUZMINA
11	ADELINA KUZNECOVA
12	ADELINA MEDVEDEVA
13	ADELINA NAUMOVA
14	ADELINA NAZAROVA
15	ADELINA NIKIFOROVA

Select #2

The below query counts the number of flights departing from each airport in the month of June 2017. It involves self-joining the 'flights' table and filtering for scheduled departures within the specified date range. The results are grouped by the departure airport and ordered in descending order based on the count of flights ('n_flights').

Finding out the busiest airport for June 2017

```
Execute | Active Connection
SELECT f1.departure_airport, COUNT(*) AS n_flights
FROM flights f1
JOIN flights f2 ON f1.flight_id = f2.flight_id
WHERE CAST(f1.scheduled_departure AS DATE) >= CAST('2017-06-01' AS DATE) AND
    CAST(f1.scheduled_departure AS DATE) < CAST('2017-07-01' AS DATE) AND
    EXISTS ( SELECT 1 FROM flights f3 WHERE f3.departure_airport = f1.departure_airport )
GROUP BY 1
ORDER BY 2 DESC;
```

Output –

	* departure_airport character(3)	n_flights
1	DME	1581
2	SVO	1464
3	LED	932
4	VKO	844
5	OVB	518
6	KJA	347
7	SVX	338
8	PEE	304
9	ROV	304
10	BZK	300
11	AER	285
12	OVS	270
13	SGC	260
14	HMA	252
15	NUX	249

Select #3

The below query first calculates the arrival count for each airport based on flights using the top three aircraft models with non-null ranges. The results are stored in a Common Table Expression (CTE) named 'AirportCounts.' The main query then selects and displays airport names and arrival counts where the count is equal to or exceeds 100, ordering the results in descending order based on the arrival count.

```
▷ Execute | 🔒 Active Connection
WITH AirportCounts AS (
    SELECT b.airport_name->>'en' AS arrival_airport_name, COUNT(*) AS arrival_count
    FROM flights a
    CROSS JOIN airports_data b
    LEFT JOIN aircrafts_data c ON a.aircraft_code = c.aircraft_code
    WHERE a.aircraft_code IN (
        SELECT aircraft_code
        FROM aircrafts_data
        WHERE range IS NOT NULL ORDER BY range DESC LIMIT 3 )
    GROUP BY b.airport_name->>'en' )
SELECT arrival_airport_name, arrival_count
FROM AirportCounts
WHERE arrival_count >= 100
ORDER BY arrival_count DESC;
```

Output –

	arrival_airport_name	arrival_count
1	Domodedovo Internationa	968
2	Sheremetyevo Internationa	744
3	Vnukovo International Airp	467
4	Tolmachevo Airport	363
5	Sochi International Airport	277
6	Khabarovsk-Novy Airport	260
7	Bolshoye Savino Airport	260
8	Pulkovo Airport	259
9	Krasnodar Pashkovsky Int	242
10	Yuzhno-Sakhalinsk Airpor	242
11	Koltsovo Airport	156
12	Yemelyanovo Airport	156
13	Ufa International Airport	156
14	Syktyvkar Airport	138
15	Irkutsk Airport	138
16	Ulyanovsk Baratayevka Air	121
17	Yakutsk Airport	121
18	Bratsk Airport	121
19	Vladivostok International A	121
20	Kazan International Airpor	121
21	Kurumoch International Ai	121

Here we can see that the Domodedovo International airport has the highest number of arrival flights hence, we can conclude that Moscow is the most popular destination. Other popular destinations include Khimki, Novosibirsk, Sochi, Khabarovsk, etc.

Select #4

The below query calculates the percentage of on-time flights by first determining the total number of eligible flights with actual and scheduled arrival times recorded and then counting the subset of flights that are considered on-time (actual arrival within 5 minutes of scheduled arrival). The result is the percentage of on-time flights, derived from the ratio of on-time flights to total flights.

Checking what percent of flights are on time

```
▷ Execute | 🔒 Active Connection
SELECT on_time_flights * 100 / total_flights AS percent_on_time
FROM (
    SELECT COUNT(*) AS total_flights
    FROM flights f1
    WHERE f1.actual_arrival IS NOT NULL
    AND f1.scheduled_arrival IS NOT NULL
    AND (
        SELECT COUNT(*)
        FROM flights f2
        WHERE f1.flight_id = f2.flight_id
        ) > 0 ) total,
        (
        SELECT COUNT(*) AS on_time_flights
        FROM flights f3
        WHERE f3.actual_arrival IS NOT NULL AND
        f3.scheduled_arrival IS NOT NULL AND
        f3.actual_arrival < f3.scheduled_arrival + INTERVAL '5 minutes' AND (
            SELECT COUNT(*)
            FROM flights f4
            WHERE f3.flight_id = f4.flight_id ) > 0 ) on_time
```

Output –

	percent_on_time
1	73

Select #5

The below query retrieves distinct aircraft codes from the 'seats' table, providing a unique list of aircraft codes associated with seat entries.

```
🔒 Active Connection
-- Active: 1702008639736@@127.0.0.1@5432@airport@bookings
▷ Execute
SELECT DISTINCT aircraft_code FROM seats;
```

Output –

	aircraft_code character(3)
1	CN1
2	320
3	CR2
4	773
5	763
6	319
7	733
8	SU9
9	321

- ***Update Queries -***

The below query updates the fare conditions for the seat with seat number '9F' on the aircraft '319' to 'Business' in the 'seats' table. Subsequently, a

SELECT statement retrieves and displays the last 10 seat entries for the same aircraft, ordered by seat number in descending order. This allows verification of the successful update and provides insight into the current seat configuration for the specified aircraft.

```
▷ Execute | 🔒 Active Connection
UPDATE seats
SET fare_conditions = 'Business'
WHERE aircraft_code = '319' and seat_no = '9F';
```

```
🔒 Active Connection
# Checking if the values got updated
▷ Execute
SELECT * FROM seats
WHERE aircraft_code = '319'
ORDER BY seat_no DESC
LIMIT 10;
```

Output –

	aircraft_code character(3)	seat_nc varchar(4)	fare_conditions varchar(10)
1	319	9F	Business
2	319	9E	Economy
3	319	9D	Economy
4	319	9C	Economy
5	319	9B	Economy
6	319	9A	Economy
7	319	8F	Economy
8	319	8E	Economy
9	319	8D	Economy
10	319	8C	Economy

Similarly, the below query updates the total amount for the booking with reference '000012' in the 'bookings' table to 25000. Following the update, a SELECT statement retrieves and displays the booking entry with the updated information for reference '000012'.

```
▷ Execute | 🔒 Active Connection
UPDATE bookings
SET total_amount = 25000
WHERE book_ref = '000012';
```

```
🔒 Active Connection
# Checking if the update is done
▷ Execute
SELECT * FROM bookings
WHERE book_ref = '000012';
```

Output –

	book_re character(	book_date timestamp with time zon	total_amount numeric
1	000012	2017-07-14 02:02:00-04	25000.00

7. Query Execution Analysis:

Now out of all the queries that we executed so far, three of them had an execution issue of running for a longer time or being too expensive, thus making them less optimized. We worked on it and cut down their execution time by making them more optimized.

- ***Select #2***

Unoptimized query –

```
▷ Execute | 🔒 Active Connection
EXPLAIN
SELECT f1.departure_airport, COUNT(*) AS n_flights
FROM flights f1
JOIN flights f2 ON f1.flight_id = f2.flight_id
WHERE CAST(f1.scheduled_departure AS DATE) >= CAST('2017-06-01' AS DATE) AND
    CAST(f1.scheduled_departure AS DATE) < CAST('2017-07-01' AS DATE) AND
    EXISTS ( SELECT 1 FROM flights f3 WHERE f3.departure_airport = f1.departure_airport )
GROUP BY 1
ORDER BY 2 DESC;
```

Query plan –

```
QUERY PLAN
Sort  (cost=4684.81..4686.06 rows=100 width=12)
  Sort Key: (count(*)) DESC
  ->  GroupAggregate  (cost=3816.60..4681.49 rows=100 width=12)
        Group Key: f1.departure_airport
        ->  Nested Loop  (cost=3816.60..4678.85 rows=328 width=4)
              ->  Merge Join  (cost=3816.31..3829.17 rows=328 width=8)
                    Merge Cond: (f1.departure_airport = f3.departure_airport)
                    ->  Sort  (cost=2158.99..2159.81 rows=328 width=8)
                          Sort Key: f1.departure_airport
                          ->  Seq Scan on flights f1  (cost=0.00..2145.28 rows=328 width=8)
                                Filter: ((scheduled_departure)::date >= '2017-06-01'::date) AND ((scheduled_departure)::date < '201
                    ->  Sort  (cost=1657.32..1657.58 rows=104 width=4)
                          Sort Key: f3.departure_airport
                          ->  HashAggregate  (cost=1652.80..1653.84 rows=104 width=4)
                                Group Key: f3.departure_airport
                                ->  Seq Scan on flights f3  (cost=0.00..1488.64 rows=65664 width=4)
              ->  Index Only Scan using flights_pkey on flights f2  (cost=0.29..2.59 rows=1 width=4)
                    Index Cond: (flight_id = f1.flight_id)
```

Here, the revised query simplifies the original by directly selecting and aggregating from the 'flights' table without the need for self-joins. It retains the logic of counting the number of flights departing from each airport within the specified date range and presents a more concise and efficient version.

Optimized query –

```
▷ Execute | 🔒 Active Connection
EXPLAIN
SELECT departure_airport, count(*) as n_flights
FROM flights WHERE CAST(scheduled_departure AS DATE) >= CAST('2017-06-01' AS DATE)
AND CAST(scheduled_departure AS DATE) < CAST('2017-07-01' AS DATE)
GROUP BY 1
ORDER BY 2 DESC;
```

Query plan –

```
QUERY PLAN
Sort  (cost=2165.77..2166.02 rows=100 width=12)
 Sort Key: (count(*)) DESC
 ->  GroupAggregate  (cost=2158.99..2162.45 rows=100 width=12)
       Group Key: departure_airport
       ->  Sort  (cost=2158.99..2159.81 rows=328 width=4)
             Sort Key: departure_airport
             ->  Seq Scan on flights  (cost=0.00..2145.28 rows=328 width=4)
                   Filter: (((scheduled_departure)::date >= '2017-06-01'::date) AND ((scheduled_departure)::date < '2017-07-0
```

- ***Select #3***

Unoptimized query –

```
Execute | Active Connection
EXPLAIN
WITH AirportCounts AS (
    SELECT b.airport_name->>'en' AS arrival_airport_name, COUNT(*) AS arrival_count
    FROM flights a
    CROSS JOIN airports_data b
    LEFT JOIN aircrafts_data c ON a.aircraft_code = c.aircraft_code
    WHERE a.aircraft_code IN (
        SELECT aircraft_code
        FROM aircrafts_data
        WHERE range IS NOT NULL ORDER BY range DESC LIMIT 3 )
    GROUP BY b.airport_name->>'en' )
SELECT arrival_airport_name, arrival_count
FROM AirportCounts
WHERE arrival_count >= 100
ORDER BY arrival_count DESC;
```

Query plan –

```
QUERY PLAN
Sort  (cost=32281.71..32281.79 rows=35 width=40)
 Sort Key: (count(*)) DESC
 ->  HashAggregate  (cost=32279.42..32280.81 rows=35 width=40)
       Group Key: (b.airport_name ->> 'en'::text)
       Filter: (count(*) >= 100)
       ->  Hash Join  (cost=10.78..19474.94 rows=2560896 width=32)
             Hash Cond: (a.aircraft_code = aircrafts_data.aircraft_code)
             ->  Seq Scan on flights a  (cost=0.00..1488.64 rows=65664 width=4)
             ->  Hash  (cost=9.48..9.48 rows=104 width=77)
                   ->  Nested Loop  (cost=1.25..9.48 rows=104 width=77)
                         ->  HashAggregate  (cost=1.25..1.28 rows=3 width=16)
                               Group Key: aircrafts_data.aircraft_code
                               ->  Limit  (cost=1.21..1.21 rows=3 width=20)
                                     ->  Sort  (cost=1.21..1.23 rows=9 width=20)
                                           Sort Key: aircrafts_data.range DESC
                                           ->  Seq Scan on aircrafts_data  (cost=0.00..1.09 rows=9 width=20)
                                                 Filter: (range IS NOT NULL)
                         ->  Materialize  (cost=0.00..4.56 rows=104 width=61)
                               ->  Seq Scan on airports_data b  (cost=0.00..4.04 rows=104 width=61)
```

The refined query eliminates the CROSS JOIN and LEFT JOIN operations, offering a more straightforward approach. It directly joins the 'flights' table with 'airports_data' based on arrival airports and aircraft codes. This results in improved efficiency while still calculating and filtering arrival counts for airports with counts greater than or equal to 100, ordered in descending order of count.

Optimized query –

```
Execute | Active Connection
EXPLAIN
WITH AirportCounts AS
    (SELECT b.airport_name->>'en' AS arrival_airport_name, COUNT(*) AS arrival_count
    FROM flights a
    JOIN airports_data b ON a.arrival_airport = b.airport_code
    WHERE a.aircraft_code IN
        (SELECT aircraft_code
        FROM aircrafts_data
        ORDER BY range DESC LIMIT 3)
    GROUP BY b.airport_name->>'en')
    SELECT arrival_airport_name, arrival_count
    FROM AirportCounts
    WHERE arrival_count >= 100
    ORDER BY arrival_count DESC;
```

Query plan –

```
QUERY PLAN
Sort  (cost=2195.81..2195.90 rows=35 width=40)
 Sort Key: (count(*)) DESC
 ->  HashAggregate  (cost=2193.53..2194.91 rows=35 width=40)
       Group Key: (b.airport_name ->> 'en'::text)
       Filter: (count(*) >= 100)
       ->  Hash Join  (cost=6.62..2070.41 rows=24624 width=32)
             Hash Cond: (a.arrival_airport = b.airport_code)
             ->  Hash Semi Join  (cost=1.28..1936.23 rows=24624 width=4)
                   Hash Cond: (a.aircraft_code = "ANY_subquery".aircraft_code)
                   ->  Seq Scan on flights a  (cost=0.00..1488.64 rows=65664 width=8)
                   ->  Hash  (cost=1.24..1.24 rows=3 width=16)
                         ->  Subquery Scan on "ANY_subquery"  (cost=1.21..1.24 rows=3 width=16)
                               ->  Limit  (cost=1.21..1.21 rows=3 width=20)
                                     ->  Sort  (cost=1.21..1.23 rows=9 width=20)
                                           Sort Key: aircrafts_data.range DESC
                                           ->  Seq Scan on aircrafts_data  (cost=0.00..1.09 rows=9 width=20)
             ->  Hash  (cost=4.04..4.04 rows=104 width=65)
                   ->  Seq Scan on airports_data b  (cost=0.00..4.04 rows=104 width=65)
```

- ***Select #4***

Unoptimized query –

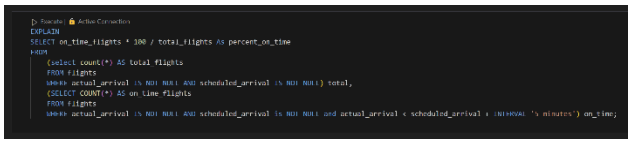
```
Execute | Active Connection
EXPLAIN
SELECT on_time_flights * 100 / total_flights AS percent_on_time
FROM (
    SELECT COUNT(*) AS total_flights
    FROM flights f1
    WHERE f1.actual_arrival IS NOT NULL
    AND f1.scheduled_arrival IS NOT NULL
    AND (
        SELECT COUNT(*)
        FROM flights f2
        WHERE f1.flight_id = f2.flight_id
        ) > 0 ) total,
        (
        SELECT COUNT(*) AS on_time_flights
        FROM flights f3
        WHERE f3.actual_arrival IS NOT NULL AND
        f3.scheduled_arrival IS NOT NULL AND
        f3.actual_arrival < f3.scheduled_arrival + INTERVAL '5 minutes' AND (
            SELECT COUNT(*)
            FROM flights f4
            WHERE f3.flight_id = f4.flight_id ) > 0 ) on_time
```

Query plan –

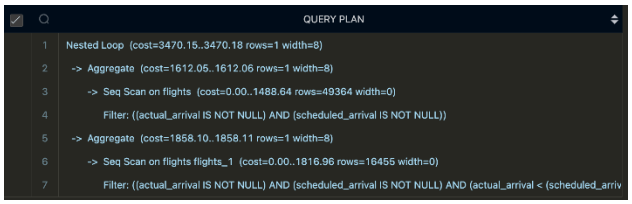
```
QUERY PLAN
Nested Loop  (cost=1096666.05..1096666.08 rows=1 width=8)
 ->  Aggregate  (cost=548182.58..548182.59 rows=1 width=8)
       ->  Seq Scan on flights f1  (cost=0.00..548141.44 rows=16455 width=0)
             Filter: ((actual_arrival IS NOT NULL) AND (scheduled_arrival IS NOT NULL) AND ((SubPlan 1) > 0))
             SubPlan 1
               ->  Aggregate  (cost=8.31..8.32 rows=1 width=8)
                     ->  Index Only Scan using flights_pkey on flights f2  (cost=0.29..8.31 rows=1 width=0)
                           Index Cond: (flight_id = f1.flight_id)
 ->  Aggregate  (cost=548483.47..548483.48 rows=1 width=8)
       ->  Seq Scan on flights f3  (cost=0.00..548469.76 rows=5485 width=0)
             Filter: ((actual_arrival IS NOT NULL) AND (scheduled_arrival IS NOT NULL) AND (actual_arrival < (scheduled_arriv
             SubPlan 2
               ->  Aggregate  (cost=8.31..8.32 rows=1 width=8)
                     ->  Index Only Scan using flights_pkey on flights f4  (cost=0.29..8.31 rows=1 width=0)
                           Index Cond: (flight_id = f3.flight_id)
```

The optimized query simplifies the original by directly calculating the percentage of on-time flights without the need for subqueries and self-joins. It computes the total and on-time flight counts efficiently from the 'flights' table, resulting in a more concise and streamlined representation.

Optimized query –



Query plan –



ISSUES FACED

While working on the project focused on popular destinations in Russia using a Flight-Airport dataset, we encountered several challenges associated with handling a large dataset efficiently. Despite the dataset being preprocessed and not requiring additional cleaning, preprocessing, standardization, or normalization, the sheer volume of the data posed some notable issues.

Performance Concerns

Processing a large dataset can lead to performance issues, particularly in terms of query response times and overall system efficiency. We addressed this challenge by adopting indexing concepts, such as creating appropriate database indexes. This significantly improved the speed of data retrieval operations, allowing for faster querying and analysis.

Memory Constraints:

Large datasets can strain memory resources, potentially causing the system to slow down or crash. We implemented memory-efficient data structures and optimized data access patterns to mitigate memory constraints. Additionally, we used server-side pagination for retrieving data in chunks rather than loading the entire dataset at once, contributing to improved memory management.

Optimizing Joins and Aggregations:

Performing joins and aggregations on a large dataset can be computationally intensive and time-consuming. We employed appropriate indexing on columns involved in joins and aggregations and utilized optimized algorithms for these operations. This significantly improved the efficiency of these processes, ensuring that analytical queries related to popular destinations could be executed in a reasonable amount of time.

Query Optimization:

Formulating complex queries without optimization can lead to increased execution times. We carefully considered the queries used in the analysis, ensuring that they were well-optimized and leveraged the available indexes. This involved analyzing the query execution plans and adjusting enhance efficiency.

In conclusion, the successful resolution of these challenges was essential to ensuring that we could conduct the project on popular destinations in Russia effectively using the Flight-Airport dataset. The adoption of indexing concepts and optimization strategies played a pivotal role in overcoming performance bottlenecks and ensuring the efficiency of the system, ultimately facilitating a smooth and effective analysis of the large dataset.

FUTURE SCOPE

Analysis and Aggregation:

We use such SQL queries to analyze flight, ticketing, and booking data to identify popular destinations in Russia. We aggregate data to determine trends, such as top routes, preferred fare conditions, and peak travel times.

Optimization and Resource Allocation:

Based on this we can further optimize resource allocation for airlines by analyzing route planning, pricing strategies, and marketing efforts. We can address inefficiencies to avoid empty seats and oversubscribed routes.

Regional Development and Economic Balance:

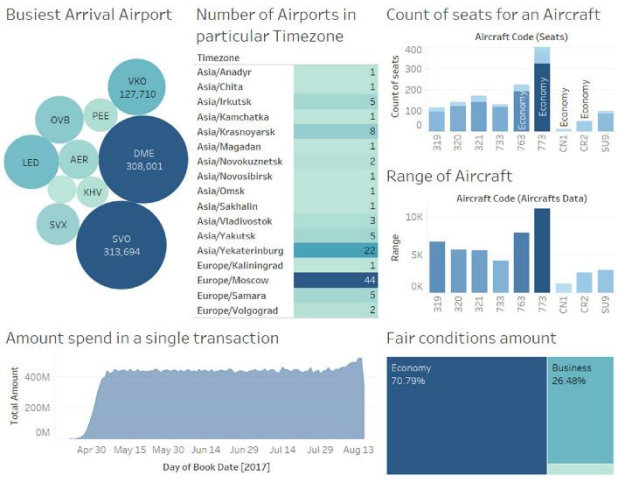
We can also identify and promote attractive destinations to achieve a more balanced

distribution of economic benefits across various regions in Russia.

Enhancing Passenger Experience:

We can use insights from the analysis to enhance the overall travel experience for passengers. We can provide personalized recommendations, a broader range of options, and streamlined booking processes to improve customer satisfaction.

DASHBOARD



https://public.tableau.com/app/profile/priya.patel6174/viz/AirportDatabase/Dashboard1?publish=yes

REFERENCES

Dataset resource – https://postgrespro.com/docs/postgrespro/10/apjs03.html

Real-world Scenario - https://en.wikipedia.org/wiki/Buffalo_Niagara_International_Airport