PES UNIVERSITY

Department of CSE

B Tech CSE - 6th Sem - Elective 4

UE22CS343BB3 - DATABASE TECHNOLOGIES

Jan – May 2025

ASSIGNMENT #1

QUERY PERFORMANCE ANALYSIS REPORT
of
Taxi Booking Management System
(DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH)

by

SRN: PES1UG22CS650

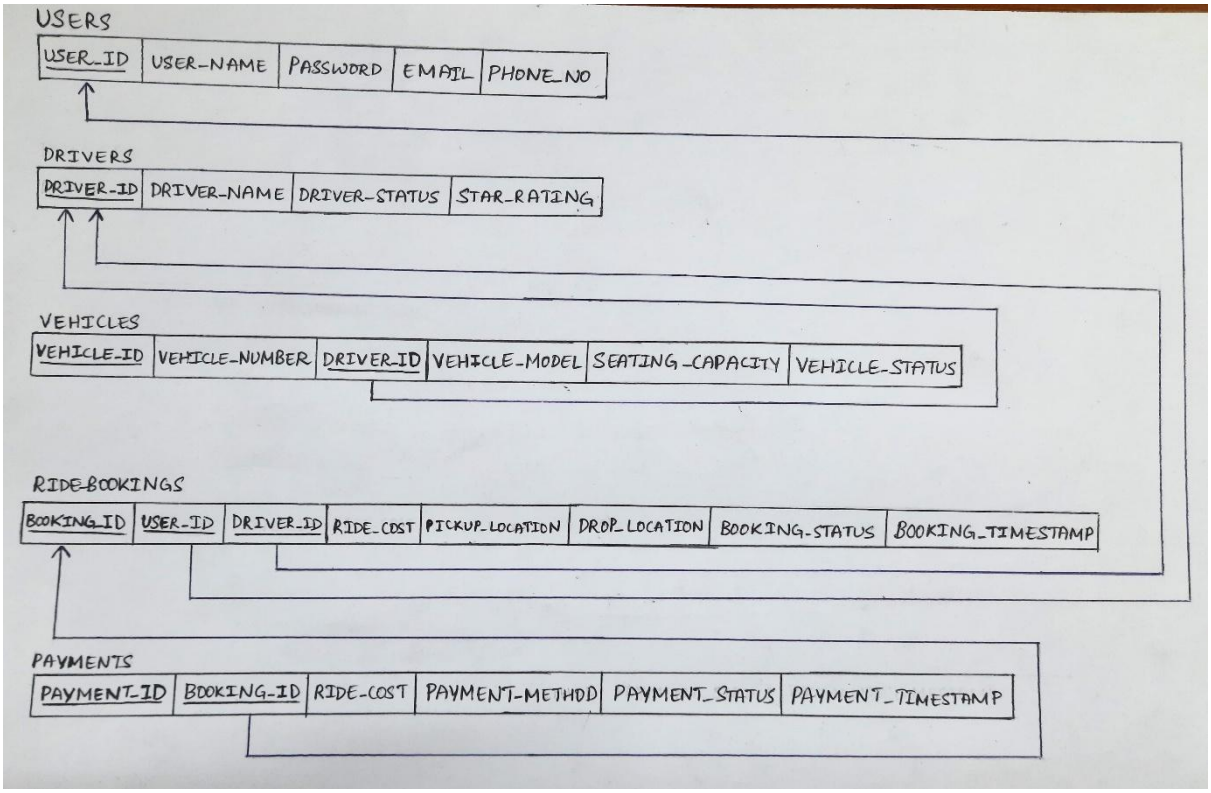
Name: TEJAS GOWRISH

Class of:

Prof. Raghu B. A.

a) Database Preparation:

Relational Schema:



Details of the tables in the database schema:

Table name	Description	Attributes
USERS	Stores user details and login information.	• User_id (Primary key) • User_name • Password • Email • Phone_no
DRIVERS	Stores driver details as well as their average ratings (provided by users from experience)	• Driver_id (Primary key) • Driver_name • Driver_status (‘available’, ‘una-vailable’) • Star_rating
VEHICLES	Stores details of the ve-hicles as well as the	• Vehicle_id (Primary key) • Vehicle_number

	drivers who drive them.	• Driver_id (Foreign key references DRIVERS table) • Vehicle_model • Seating_capacity • Vehicle_status ('active', 'inactive', 'maintenance')
RIDE_BOOKINGS	Stores details of ride bookings made by users	• Booking_id (Primary key) • User_id (Foreign key references the USERS table) • Driver_id (Foreign key references the DRIVERS table) • Ride_cost • Pickup_location • Drop_location • Booking_status ('confirmed', 'pending', 'on-the-way', 'arrived', 'canceled'; default - 'pending') • Booking_timestamp
PAYMENTS	Stores details of payments made by users after rides.	• Payment_id (Primary key) • Booking_id (Foreign key that references the RIDE_BOOKINGS table) • Ride_cost • Payment_method ('cash', 'card', 'upi', 'net-banking', 'coupon') • Payment_status ('processing', 'completed', 'failed') • Payment_timestamp

Creation of tables:

(File name – tables.sql)

Database creation code -

```sql
create database DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH;
-- drop database DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH;
use DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH;
```

Creating the tables –

```sql
create table users
(
    user_id int auto_increment primary key,
    user_name varchar(50) not null,
    password varchar(20) not null,
    email varchar(100) not null,
    phone_no varchar(10) not null
);

create table drivers
(
    driver_id int auto_increment primary key,
    driver_name varchar(50) not null,
    -- vehicle_id int unique not null,
    driver_status enum('available', 'unavailable') default 'available',
    star_rating int,
    check (star_rating >= 0 and star_rating <= 5)
    -- constraint fk_driver_vehicle foreign key(vehicle_id) references vehicles(vehicle_id)
    -- foreign key(vehicle_id) references vehicles(vehicle_id)
);
-- drop table drivers;
```

```sql
create table vehicles
(
    vehicle_id int auto_increment not null primary key,     -- vehicle ID in the system
    vehicle_number varchar(10) unique not null,     -- license number plate
    driver_id int unique,
    vehicle_model varchar(50),
    seating_capacity int not null,
    vehicle_status enum('active', 'inactive', 'maintenance') default 'active',
    constraint fk_vehicle_driver foreign key(driver_id) references drivers(driver_id) on delete cascade
);
create table ride_bookings
(
    booking_id int auto_increment primary key,
    user_id int,
    driver_id int,
    ride_cost decimal(10, 2) not null,
    pickup_location varchar(200) not null,
    drop_location varchar(200) not null,
    booking_status enum('confirmed', 'pending', 'on-the-way', 'arrived', 'canceled') not null default 'pending',
    booking_timestamp datetime,
    constraint fk_booking_user foreign key(user_id) references users(user_id),
    constraint fk_booking_driver foreign key(driver_id) references drivers(driver_id)
);
create table payments
(
    payment_id int auto_increment primary key,
    booking_id int,
    ride_cost decimal(10, 2) not null,
    payment_method enum('cash', 'card', 'upi', 'net-banking', 'coupon') not null,
    payment_status enum('processing', 'completed', 'failed') not null default 'processing',
    payment_timestamp datetime,
    constraint fk_payment_booking foreign key(booking_id) references ride_bookings(booking_id)
);
```

- **Populating the database:**

Python script to populate the USERS, DRIVERS and VEHICLES tables (insertion_users_drivers.py) –

```python
import mysql.connector
import random
from random import randint, randrange
import string
from faker import Faker
from password_generator import PasswordGenerator
```

```
faker = Faker() # faker - generate fake data
pwo = PasswordGenerator()   # generate random password

# Connect to db
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root123",
    database="DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH"
)
cursor = conn.cursor()

#
**********************************************************************************************
*****************************

# Insert users

num_users = 1500
users_data = []

for i in range(num_users):
    user_name = faker.name()
    password = pwo.generate()   # generate random password
    email = faker.email()
    phone_no = ''.join(random.choices(string.digits, k=10))  # 10-digit phone number
    users_data.append((user_name, password, email, phone_no))

cursor.executemany(
    """
    INSERT INTO users (user_name, password, email, phone_no)
    VALUES (%s, %s, %s, %s)
    """, users_data)

print(f"{num_users} users inserted successfully!")



#
**********************************************************************************************
*****************************
```

```
# Insert drivers

num_drivers = 350
drivers_data = []

for i in range(1, num_drivers + 1):
    driver_name = faker.name()
    # vehicle_id = i  # 1:1 mapping b/w drivers and vehicles
    driver_status = random.choice(["available", "unavailable"])
    star_rating = random.randint(1, 5)
    # drivers_data.append((driver_name, vehicle_id, driver_status, star_rating))
    drivers_data.append((driver_name, driver_status, star_rating))

# cursor.executemany(
#     """
#     INSERT INTO drivers (driver_name, vehicle_id, driver_status, star_rating)
#     VALUES (%s, %s, %s, %s)
#     """, drivers_data)
cursor.executemany(
    """
    INSERT INTO drivers (driver_name, driver_status, star_rating)
    VALUES (%s, %s, %s)
    """, drivers_data)

# Fetch driver IDs after insertion
cursor.execute("SELECT driver_id FROM drivers")
driver_ids = [row[0] for row in cursor.fetchall()]  # Extract driver IDs

print(f"{num_drivers} drivers inserted successfully")


#
********************************************************************************************************
*****************************

# insert vehicles

num_vehicles = 350
vehicles_data = []

vehicle_models = {"Tata Indica" : 4, "Tata Indigo" : 4, "Tata Indigo Manza" : 4,
```

```
            "Suzuki Swift" : 4, "Suzuki Swift Dezire" : 4, "Suzuki Baleno" : 4, "Suzuki Ciaz" : 4, "Suzuki Ertiga"
: 7,
            "Hyundai Aura" : 4, "Hyundai Creta" : 7, "Hyundai i10" : 4, "Hyundai i20" : 4}

vehicle_statuses = ["active", "inactive", "maintenance"]

for i in range(1, num_vehicles + 1):
    vehicle_number =
f"KA{random.randint(10,99)}{random.choice(string.ascii_uppercase)}{random.randint(1000,9999)}"
    vehicle_model = random.choice(list(vehicle_models.keys()))
    seating_capacity = vehicle_models[vehicle_model]
    vehicle_status = random.choice(vehicle_statuses)
    driver_id = driver_ids[i - 1]
    vehicles_data.append((vehicle_number, driver_id, vehicle_model, seating_capacity, vehicle_status))

cursor.executemany(
    """
    INSERT INTO vehicles (vehicle_number, driver_id, vehicle_model, seating_capacity, vehicle_status)
    VALUES (%s, %s, %s, %s, %s)
    """, vehicles_data)


print(f"{num_vehicles} vehicles inserted successfully")

#
**********************************************************************************************
**************************

conn.commit()
cursor.close()
conn.close()

print("All data inserted successfully")
```

After running the script –

```
1500 users inserted successfully!
350 drivers inserted successfully
350 vehicles inserted successfully
All data inserted successfully
```

Python script to insert 10000+ records into the BOOKINGS table (insertion_bookings_10000.py) –

```
import mysql.connector
import random
from datetime import datetime, timedelta

# Connect to database
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root123",
    database="DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH"
)
cursor = conn.cursor()

# Generate 10,000 ride bookings
locations = ['MG Road', 'Indiranagar', 'Kormangala', 'Whitefield', 'Hebbal', 'Silk Board', 'Marathahalli',
'Jayanagar', 'Yelahanka',
            'Electronic City', 'KR Puram', 'Banashankari', 'HSR Layout', 'BTM Layout', 'Shivajinagar',
'Malleshwaram', 'Domlur',
            'Vijayanagar', 'Peenya', 'Basavangudi','Bannerghatta Road', 'Old Airport Road', 'Bellandur', 'CV Raman
Nagar',
            'Kammanahalli', 'Kalyan Nagar', 'Rajajinagar', 'Sarjapur Road', 'Ulsoor', 'Yeshwantpur', 'Airport Road',
'Banaswadi',
            'Bommanahalli', 'Brookefield', 'Cox Town', 'Devanahalli', 'HBR Layout', 'Hosur Road', 'Jakkur',
'Kaggadasapura',
            'Kengeri', 'Kumaraswamy Layout', 'Langford Town', 'Majestic', 'Nagarbhavi', 'Nagawara', 'New BEL
Road',
            'Rajarajeshwari Nagar', 'Ramamurthy Nagar', 'Sahakar Nagar', 'Sanjay Nagar', 'Thippasandra', 'Tumkur
Road', 'Varthur',
            'Vasanth Nagar', 'Vidyaranyapura', 'Vijaya Bank Layout', 'Wilson Garden', 'Yelahanka New Town',
'Yeshwantpur',
```

```
        'Anjanapura', 'Attibele', 'Begur', 'Begur Road', 'Bilekahalli', 'Bommasandra', 'Chandapura',
'Chikkalasandra',
        'Doddathoguru', 'Gottigere', 'Hosakerehalli', 'Hulimavu', 'Jigani', 'Kanakapura Road', 'Konanakunte',
'Kudlu Gate',
        'Madiwala', 'Mahadevapura', 'Nagarbhavi', 'Padmanabhanagar', 'Hegade Nagar', 'Hennur', 'Hoskote',
'Hulimavu', 'Huskur',
        'Kadugodi', 'Kaggalipura', 'Kodichikkanahalli', 'Kothanur', 'Kudlu', 'Kumbalgodu', 'Marsur',
'Nelamangala', 'Panathur',
        'Rachenahalli', 'Ramanagara', 'Rayasandra', 'Sompura', 'Subramanyapura', 'Thubarahalli', 'Tindlu',
'Tippasandra']

for i in range(20000):
    user_id = random.randint(1, 1500)
    driver_id = random.randint(1, 350)
    pickup_location = random.choice(locations)
    drop_location = random.choice(locations)
    booking_status = random.choice(["confirmed", "pending", "on-the-way", "arrived", "canceled"])
    ride_cost = round(random.uniform(50, 500), 2) if(pickup_location != drop_location and booking_status !=
'canceled') else 0.00
    booking_timestamp = datetime.now() - timedelta(days=random.randint(0, 30))

    query = "INSERT INTO ride_bookings (user_id, driver_id, ride_cost, pickup_location, drop_location,
booking_status, booking_timestamp) VALUES (%s, %s, %s, %s, %s, %s, %s)"
    values = (user_id, driver_id, ride_cost, pickup_location, drop_location, booking_status, booking_timestamp)

    cursor.execute(query, values)

# Commit and close connection
conn.commit()
cursor.close()
conn.close()

print("200000 ride bookings inserted successfully!")
```

After running the python script –

```
200000 ride bookings inserted successfully!
```

Python script to insert 10000+ records into the PAYMENTS table –

```
import mysql.connector
import random
from datetime import datetime, timedelta

conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root123",
    database="DBT25_A1_PES1UG22CS650_TEJAS_GOWRISH"
)
cursor = conn.cursor()

# Fetch all valid booking_ids and corresponding ride_costs from ride_bookings table
cursor.execute("SELECT booking_id, ride_cost, booking_timestamp FROM ride_bookings")
booking_data = cursor.fetchall()  # List of tuples

if not booking_data:
    print("Error: No ride bookings found in the database!")
    conn.close()
    exit()

payment_methods = ["cash", "card", "upi", "net-banking", "coupon"]
payment_statuses = ["processing", "completed", "failed"]


payment_data = []

for booking_id, ride_cost, booking_timestamp in booking_data:
    if ride_cost == 0.00:
        continue  # Skip payments for canceled rides

    payment_method = random.choice(payment_methods)
    payment_status = random.choice(payment_statuses)

    payment_timestamp = booking_timestamp + timedelta(minutes=random.randint(5, 120))  # payment
timestamp after booking timestamp

    payment_data.append((booking_id, ride_cost, payment_method, payment_status, payment_timestamp))

cursor.executemany("""
```

```
    INSERT INTO payments (booking_id, ride_cost, payment_method, payment_status, payment_timestamp)
    VALUES (%s, %s, %s, %s, %s)
""", payment_data)


conn.commit()
cursor.close()
conn.close()

print(f"{len(payment_data)} payment records inserted successfully!")
```

After running the python script –

```
15878 payment records inserted successfully!
```

b) Queries Creation and Performance Measurement AND

c) Indexing for Query Performance Improvement:

Executing 'SELECT *' and 'SELECT COUNT(*)' statements on the tables:

Users table –

```
19 •  select * from users;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

user_id	user_name	password	email	phone_no
1	TEJAS GOWRISH	PES1UG22CS650	tejasgowrish@gmail.com	9876543210
2	Isaac Watts MD	=7X#ac	julie30@example.com	3394543679
3	Stephen Rodriguez	=K!Lse1	shannonjacob@example.net	9683468258
4	Terri Morris	<bE=2MWZlI	richardwilson@example.com	2995874540
5	Brett Hamilton	Q0Q+z9%r8r	colerandall@example.org	8041831890
6	Jacob Hill	0h!7(%e>Wfv_GfE	jeremiahjohnson@example.net	2773902917
7	Ellen Griffith	rcLIT1^7+To^Oj9	michael85@example.org	0037273799
8	John King	I5Dp3y$	lunadaniel@example.net	5259707576
9	Heather Cook	+B4vH7yk	vanessa94@example.com	2609582063
10	Natalie Burton	O2NxD?+Te4ei#	ucook@example.org	0926887362
11	Shannon Garcia	kyl%$FF9	lisa15@example.org	5388160425

users 3 ×

```
11 •  select count(*) from users;
```

Result Grid | Filter Rows:

count(*)
1501

Drivers table –

```
20 •  select * from drivers;
```

Result Grid | Filter Rows: | Edit:

driver_id	driver_name	driver_status	star_rating
1	Mrs. Sierra Torres MD	unavailable	3
2	Jeffrey Thompson	available	2
3	Victor Cooper	unavailable	5
4	Joseph Watson	unavailable	2
5	Jennifer Duncan	available	5
6	Breanna Morgan	available	2
7	April Hayes	available	3

```
12 •  select count(*) from drivers;
```

Result Grid | Filter Rows:

count(*)
350

Vehicles table –

```
21 •  select * from vehicles;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

vehicle_id	vehicle_number	driver_id	vehicle_model	seating_capacity	vehicle_status
1	KA98L8980	1	Suzuki Ertiga	7	inactive
2	KA59O2624	2	Hyundai Creta	7	inactive
3	KA47R4197	3	Hyundai i10	4	active
4	KA80A1286	4	Hyundai i20	4	active
5	KA13F2156	5	Suzuki Baleno	4	maintenance
6	KA62W6911	6	Suzuki Swift	4	maintenance
7	KA28H8941	7	Tata Indigo Manza	4	maintenance
8	KA55N1690	8	Suzuki Baleno	4	active
9	KA71G7711	9	Suzuki Swift	4	active
10	KA20U7254	10	Suzuki Ertiga	7	inactive
11	KA12M8850	11	Hyundai Aura	4	maintenance

vehicles 5 ×

```
10 •  select count(*) from vehicles;
```

Result Grid | Filter Rows:

count(*)
350

Ride_bookings table –

```
13 •  select count(*) from ride_bookings;
```

Result Grid | Filter Rows: | Expo

count(*)
20000

```
22 •  select * from ride_bookings;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: | Fetch rows:

booking_id	user_id	driver_id	ride_cost	pickup_location	drop_location	booking_status	booking_timestamp
1	1084	83	0.00	Rajarajeshwari Nagar	Domlur	canceled	2025-02-26 13:40:50
2	437	68	92.23	Domlur	Jayanagar	pending	2025-02-07 13:40:50
3	252	156	329.82	Rajajinagar	Ramanagara	arrived	2025-02-15 13:40:50
4	679	41	373.43	Doddathoguru	Hebbal	confirmed	2025-02-11 13:40:50
5	1299	44	231.37	Begur	Ramamurthy Nagar	on-the-way	2025-02-26 13:40:50
6	1354	247	0.00	Sahakar Nagar	Shivajinagar	canceled	2025-03-03 13:40:50
7	274	203	484.76	Sahakar Nagar	Kadugodi	pending	2025-02-09 13:40:50
8	35	30	421.96	Hoskote	Tumkur Road	arrived	2025-02-08 13:40:50
9	629	96	386.52	Kaggadasapura	Yelahanka	pending	2025-02-28 13:40:50
10	990	298	0.00	Kaggalipura	Yeshwantpur	canceled	2025-03-08 13:40:50
11	809	150	492.38	Devanahalli	Vijayanagar	arrived	2025-02-19 13:40:50

ride_bookings 6 ×

Payments table –

```
23 •  select * from payments;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap

payment_id	booking_id	ride_cost	payment_method	payment_status	payment_timestamp
1	2	92.23	net-banking	processing	2025-02-07 15:16:50
2	3	329.82	cash	failed	2025-02-15 14:11:50
3	4	373.43	net-banking	completed	2025-02-11 14:47:50
4	5	231.37	net-banking	processing	2025-02-26 15:18:50
5	7	484.76	net-banking	processing	2025-02-09 14:56:50
6	8	421.96	cash	completed	2025-02-08 15:06:50
7	9	386.52	net-banking	failed	2025-02-28 14:12:50
8	11	492.38	coupon	completed	2025-02-19 14:38:50
9	12	224.47	coupon	completed	2025-02-21 13:54:50
10	13	220.65	cash	completed	2025-02-21 15:30:50
11	14	463.71	net-banking	failed	2025-02-18 14:32:50

payments 7 ×

```
14 •  select count(*) from payments;
```

Result Grid | Filter Rows:

count(*)
20000

Creation of indexes:

(File name – indexes.sql)

Users table-

```
-- Users Table
create index idx_users_email on users(email);    -- email
create index idx_users_phone on users(phone_no);     -- phone number
```

Drivers table-

```
-- drivers table
create index idx_drivers_status on drivers(driver_status);
create index idx_drivers_rating on drivers(star_rating);
-- drop index idx_drivers_status on drivers;
```

Ride_bookings table -

```
-- ride_bookings table
create index idx_ride_bookings_user on ride_bookings(user_id);
create index idx_ride_bookings_driver on ride_bookings(driver_id);
create index idx_ride_bookings_status on ride_bookings(booking_status);

-- drop index idx_ride_bookings_user on ride_bookings;

create index idx_ride_bookings_pickup_location on ride_bookings(pickup_location);
create index idx_ride_bookings_drop_location on ride_bookings(drop_location);
drop index idx_ride_bookings_pickup_location on ride_bookings;

-- drop index idx_ride_bookings_pickup_location on ride_bookings;
create index idx_ride_bookings_cost on ride_bookings(ride_cost desc);
```

Vehicles table -

```
-- vehicles table
create index idx_vehicles_number on vehicles(vehicle_number);
create index idx_vehicles_status on vehicles(vehicle_status);
create index idx_vehicles_driver on vehicles(driver_id);
```

Payments table -

```
-- payments table
create index idx_payments_booking on payments(booking_id);
create index idx_payments_status on payments(payment_status);
create index idx_payments_timestamp on payments(payment_timestamp desc);
create index idx_payments_ride_cost on payments(ride_cost desc);
```

Running table scans and index scans:

Full table scans generally occur when un-indexed tables are queried. This process is very slow, especially on larger tables, because each record must be individually examined.

E.g. 1)

Full table scan:

On un-indexed attribute ‘phone_no’ of USERS table-

```
13 •    select * from users where phone_no LIKE '%7290';
```

Result Grid | Filter Rows: | Edit: | Export/Import:

user_id	user_name	password	email	phone_no
550	Margaret Stevens	u7)TDBw4	jacksonlarry@example.com	8278007290
NULL	NULL	NULL	NULL	NULL

Running explain-analyze command-

```
14 •    explain analyze select * from users where phone_no LIKE '%7290';
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.01600000

Timing (as measured by the server):
Execution time: 0:00:0.01011080
Table lock wait time: 0:00:0.00003100

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 1501

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 0

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

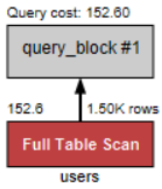
Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 302
Thread Id: 85

Form Editor | Field Types | Query Stats | Execution Plan

Execution plan-



Index scan:

On the indexed attribute 'email' of the USERS table-

```
16 •  SELECT * FROM users WHERE email = 'jacksonlarry@example.com';
```

Result Grid | Filter Rows: | Edit: | Export/Import:

user_id	user_name	password	email	phone_no
550	Margaret Stevens	u7)TDBw4	jacksonlarry@example.com	8278007290
NULL	NULL	NULL	NULL	NULL

Running the explain-analyze command-

```
17 •  explain analyze SELECT * FROM users WHERE email = 'jacksonlarry@example.com';
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00083560
Table lock wait time: 0:00:0.00000600

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 1

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 0

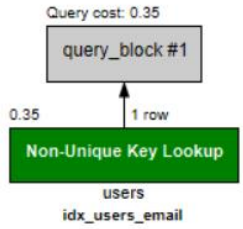
Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
At least one Index was used

Other Info:
Event Id: 326
Thread Id: 85

Execution plan-



As we can see from the above screenshots, both queries return the same tuple from the same record. However, the second query (on the indexed 'email' attribute of the USERS table) executes much faster due to the index on the attribute in the 'where' clause.
Looking at the query statistics of the two queries, the execution times at both the client and server ends are much lower in the second query, indicating its higher efficiency.
We even observe from the execution plans of the two queries that the query cost of the index scan is much lesser than that of the table scan.

The same is observed in the following examples as well –

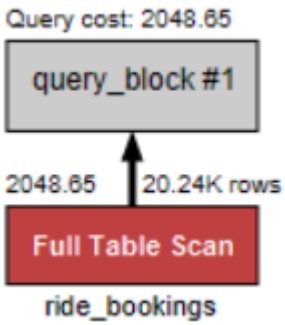
E.g. 2)
Table scan:
Query-

```
19 •  select * from ride_bookings where pickup_location = 'MG Road';
```

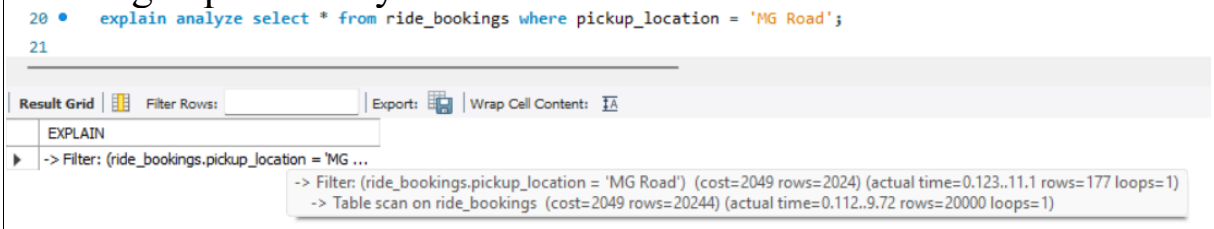
Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

booking_id	user_id	driver_id	ride_cost	pickup_location	drop_location	booking_status	booking_timestamp
29	145	136	100.69	MG Road	Rayasandra	confirmed	2025-03-02 13:40:50
75	795	82	69.20	MG Road	Thubarahalli	confirmed	2025-02-28 13:40:50
259	738	209	197.68	MG Road	Malleshwaram	confirmed	2025-02-25 13:40:50
389	543	108	359.45	MG Road	Wilson Garden	confirmed	2025-02-15 13:40:50

Execution plan-



Running explain-analyze -

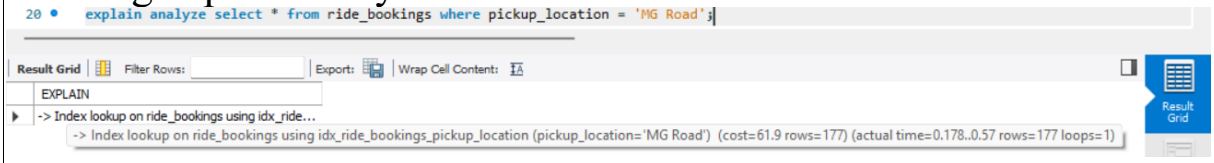


Here, we have a query that performs a full table scan on the RIDE_BOOKINGS table. As we can see, it is not very efficient because it goes through each tuple in the table searching for a match. This would be much more efficient if we use an index on the 'pickup_location' field.

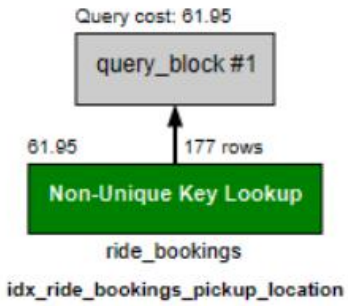
```
create index idx_ride_bookings_pickup_location on ride_bookings(pickup_location);
```

After creating an index for the 'pickup_location' field-

Running explain-analyze -



New execution plan -



Clearly, the performance of the query has improved significantly after the index was created. The query cost as well as the lookup time have both significantly reduced (improved).

E.g. 3)
Table scan:
Query -

```
23 •  select * from drivers where star_rating >= 4;
```

Result Grid | Filter Rows: | Edit:

driver_id	driver_name	driver_status	star_rating
3	Victor Cooper	unavailable	5
5	Jennifer Duncan	available	5
8	David Martin	available	5
10	Shannon Cabrera	unavailable	4

Query statistics-

```
23 •  select * from drivers where star_rating >= 4;
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00068490
Table lock wait time: 0:00:0.00000400

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 151
Rows examined: 350

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 0

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

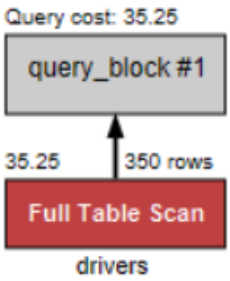
Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 479
Thread Id: 85

Form Editor | Field Types | Query Stats | Execution Plan

Execution plan-



As we can see, the query requires a full table scan of the DRIVERS table despite there being an index on the 'star_rating' field. This is because the '>=' operators forces a scan in search for all values greater than or equal to the RHS (value of 4) on the specified field in the table, which in turn could result in a full table scan if the optimizer decides that scanning the entire table is more efficient.
This is the case with most relational operators (>, <, <=, >=). To make efficient use of the index we have created, we can split the query and combine the results with the "UNION" operator.

```
23 •    select * from drivers where star_rating = 4
24      union
25      select * from drivers where star_rating = 5;
```

Output-

Result Grid | Filter Rows: | Export: | Wrap

driver_id	driver_name	driver_status	star_rating
339	Susan Reese	unavailable	4
341	Michael White	available	4
348	Mrs. Diana Bradley	unavailable	4
3	Victor Cooper	unavailable	5
5	Jennifer Duncan	available	5
8	David Martin	available	5
17	Eric Booth	unavailable	5

Query statistics-

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00341440
Table lock wait time: 0:00:0.00000700

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 151
Rows examined: 302

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 1

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

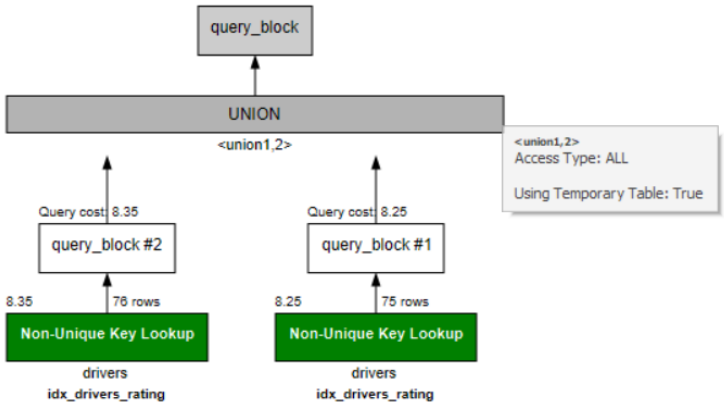
Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

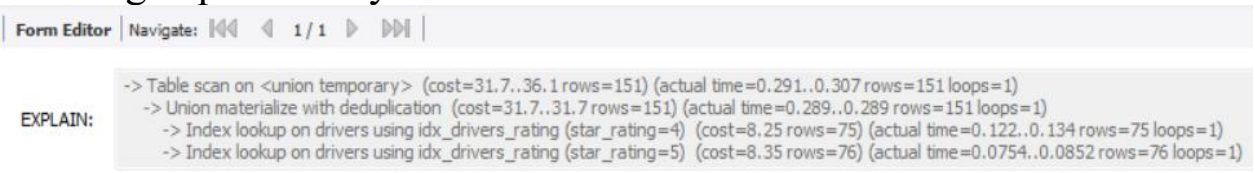
Other Info:
Event Id: 519
Thread Id: 85

Index scan:

New query plan-



Running explain-analyze -

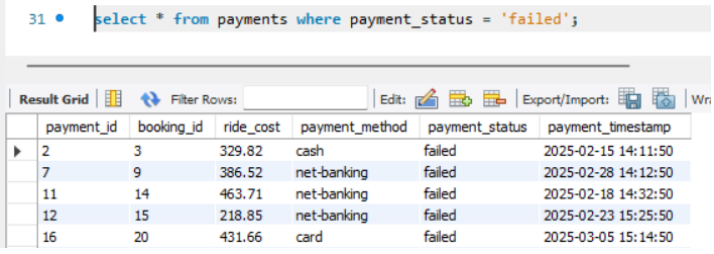


As the above screenshots suggest, the overall query cost has reduced significantly because we made use of the index. However, in this case, the overall execution time is greater than that of the full table scan.
We also see that despite having created an index on the ‘star_ratings’ attribute, a full table scan is still being performed. This is because queries are executed according to the plan that the optimizer deems most optimal, which in this case is a full table scan on the DRIVERS table.

E.g. 4)

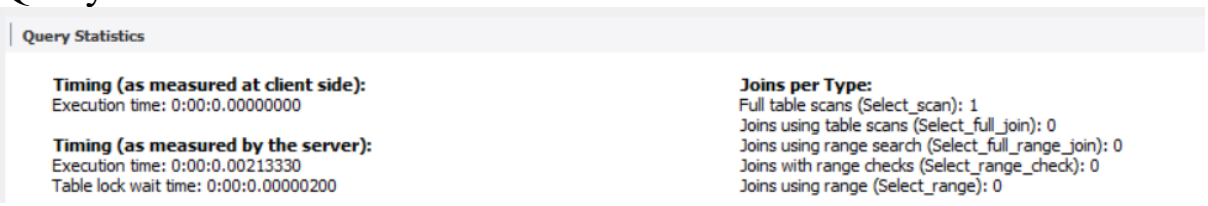
Table scan:

Query-

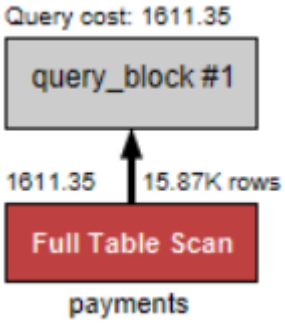


payment_id	booking_id	ride_cost	payment_method	payment_status	payment_timestamp
2	3	329.82	cash	failed	2025-02-15 14:11:50
7	9	386.52	net-banking	failed	2025-02-28 14:12:50
11	14	463.71	net-banking	failed	2025-02-18 14:32:50
12	15	218.85	net-banking	failed	2025-02-23 15:25:50
16	20	431.66	card	failed	2025-03-05 15:14:50

Query statistics-



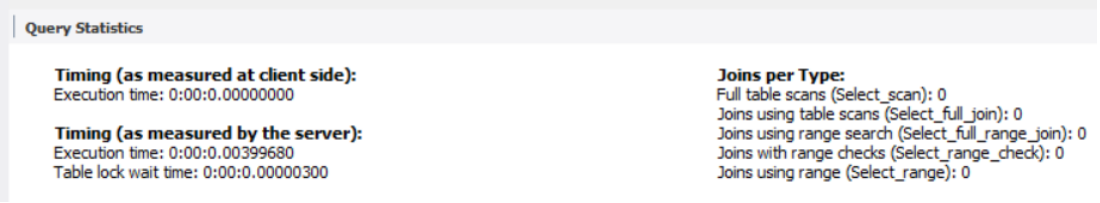
Query execution plan -



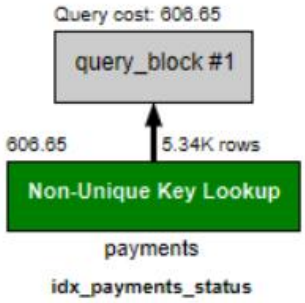
The above screenshots show how the absence of an index cause a full table scan on the table.

Index scan:

Query statistics-



Execution plain-



However, after creating an index on the 'payment_status' attribute of the PAYMENTS table, the full table scan can be completely avoided, resulting in an improvement (reduction) in the query cost.

E.g. 5)
Query -

```
33 •  select * from ride_bookings where pickup_location = 'Nagawara' and drop_location = 'kammanahalli';
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

booking_id	user_id	driver_id	ride_cost	pickup_location	drop_location	booking_status	booking_timestamp
9872	1286	182	214.26	Nagawara	Kammanahalli	on-the-way	2025-02-27 13:40:54
11890	463	275	289.43	Nagawara	Kammanahalli	confirmed	2025-02-14 13:40:55
17893	1219	287	465.78	Nagawara	Kammanahalli	on-the-way	2025-02-10 13:40:58
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Table scan (without index):

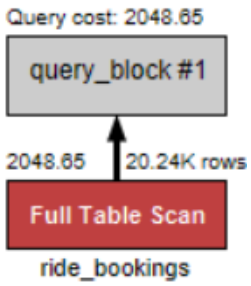
Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.01500000

Timing (as measured by the server):
Execution time: 0:00:0.01791100
Table lock wait time: 0:00:0.00000200

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Execution plan-



Index scan:
Query statistics-

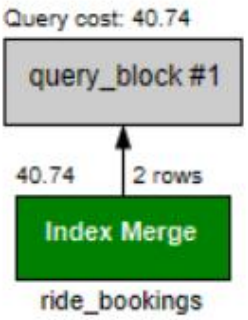
Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00084560
Table lock wait time: 0:00:0.00000400

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 1

Execution plan-



intersect(idx_ride_bookings_pickup_location,idx_ride_bookings_drop_location)

Clearly, from the above screenshots, we see that the indexes created on the 'pickup_location' and 'drop_location' fields of the ride_bookings table has resulted in lower execution time and lower query cost (i.e. better query performance).

Multi-table joins:

E.g. 1)
Query –

```
select * from ride_bookings
join users on ride_bookings.user_id = users.user_id
join drivers on ride_bookings.driver_id = drivers.driver_id;
```

Output –

```
mysql> select * from ride_bookings
    -> join users on ride_bookings.user_id = users.user_id
    -> join drivers on ride_bookings.driver_id = drivers.driver_id limit 10;
```

booking_id	user_id	driver_id	ride_cost	pickup_location	drop_location	booking_status	booking_timestamp	user_id	user_name	password	email	phone_no	driver_id	driver_name	driver_status	star_rating
124	1016	1	84.02	Banaswadi	Thubarahalli	on-the-way	2025-02-28 13:40:50	1016	Jody Bishop	I!eX=1z%>Y3_	dzamora@example.org	5047251983	1	Mrs. Sierra Torres MD	unavailable	3
943	1371	1	75.32	Marathahalli	Kothanur	arrived	2025-03-04 13:40:50	1371	Jeremy West	<=WR0o	mary36@example.net	3183349280	1	Mrs. Sierra Torres MD	unavailable	3
1133	510	1	294.03	Bellandur	Jakkur	pending	2025-02-16 13:40:50	510	Jennifer Garrett	)d(95*$*s5jJ	taylorjanet@example.net	7725554574	1	Mrs. Sierra Torres MD	unavailable	3
1505	793	1	428.76	Huskur	Electronic City	on-the-way	2025-02-07 13:40:50	793	Timothy Harrison	*_7S20IcZ&qBo	emily38@example.net	7536315078	1	Mrs. Sierra Torres MD	unavailable	3
2283	917	1	276.47	Hoskote	Yelahanka	pending	2025-03-05 13:40:51	917	Lindsay Scott	*PxL==4gQOTH<dPe	wleon@example.net	2316157061	1	Mrs. Sierra Torres MD	unavailable	3
2294	1383	1	333.59	HBR Layout	Bannerghatta Road	on-the-way	2025-02-12 13:40:51	1383	Kendra Shepard	fZoZY)1U7>Tj	david78@example.com	4578839464	1	Mrs. Sierra Torres MD	unavailable	3
2540	1452	1	232.41	Chikkalasandra	Jayanagar	pending	2025-02-17 13:40:51	1452	Lori Foley	olmy<!w1X$y_Cb	lopezmaria@example.com	9928263203	1	Mrs. Sierra Torres MD	unavailable	3
3533	999	1	205.54	HSR Layout	Bannerghatta Road	on-the-way	2025-02-16 13:40:51	999	Danielle Bauer	bkoJTV72I$>*	barbara22@example.com	1102439709	1	Mrs. Sierra Torres MD	unavailable	3
3764	1016	1	124.52	Cox Town	Silk Board	on-the-way	2025-02-14 13:40:51	1016	Jody Bishop	I!eX=1z%>Y3_	dzamora@example.org	5047251983	1	Mrs. Sierra Torres MD	unavailable	3
3988	20	1	124.87	Vidyaranyapura	Bommasandra	pending	2025-02-16 13:40:51	20	Jasmine Gonzalez	A-Z)2VLv++&lQ	brownjoshua@example.com	4678976438	1	Mrs. Sierra Torres MD	unavailable	3

```
10 rows in set (0.00 sec)
```

The above query performs an inner join on the RIDE_BOOKINGS, USERS and DRIVERS tables. The natural join operation joins the tables on their common columns.

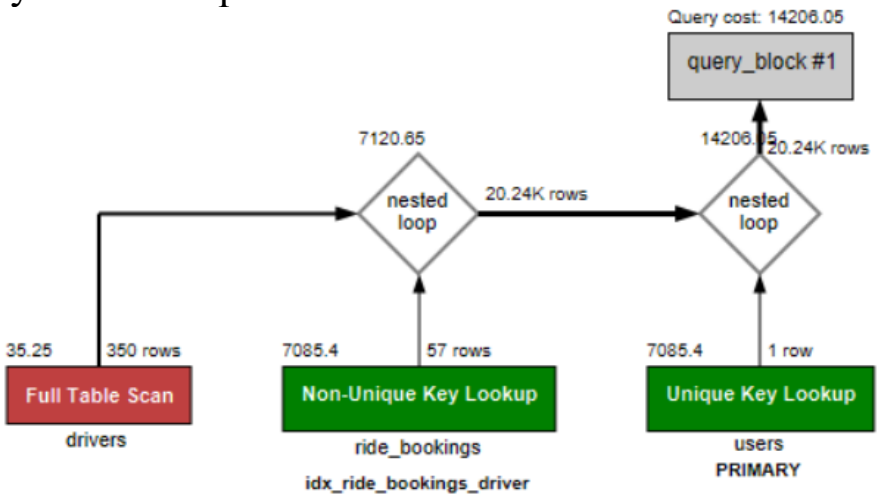
Running explain and analyze –

Form Editor | Navigate: 1 / 1

EXPLAIN:

```
-> Nested loop inner join  (cost=14206 rows=20244) (actual time=0.315..115 rows=20000 loops=1)
    -> Nested loop inner join  (cost=7121 rows=20244) (actual time=0.305..66.1 rows=20000 loops=1)
        -> Table scan on drivers  (cost=35.2 rows=350) (actual time=0.0664..0.435 rows=350 loops=1)
        -> Filter: (ride_bookings.user_id is not null)  (cost=14.5 rows=57.8) (actual time=0.135..0.18 rows=57.1 loops=350)
```

Query execution plan –



The ‘user_id’ field is common to both the RIDE_BOOKINGS table and the USERS table, and the ‘driver_id’ field is common to both the RIDE_BOOKINGS table and the DRIVERS table.

Foreign keys are not automatically indexed by default, due to which a full table scan is being performed on the DRIVERS table.

E.g. 2)

Query -

```
40 •  select r.booking_id, u.user_name, d.driver_name, r.pickup_location, r.drop_location
41    from ride_bookings as r
42    join users as u on r.user_id = u.user_id
43    join drivers as d on r.driver_id = d.driver_id;
```

Output -

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

booking_id	user_name	driver_name	pickup_location	drop_location
17966	Rebecca Mills	Mrs. Sierra Torres MD	Madiwala	BTM Layout
18667	Amanda Ruiz	Mrs. Sierra Torres MD	Madiwala	Sahakar Nagar
19127	Tammy Brown	Mrs. Sierra Torres MD	Kormangala	Bilekahalli
19963	James Carter	Mrs. Sierra Torres MD	Hennur	Kudlu Gate
207	Jesse Garcia	Jeffrey Thompson	Begur Road	Jakkur
220	Daniel Rogers	Jeffrey Thompson	Brookefield	Rayasandra
1157	Debra Shepherd	Jeffrey Thompson	Vidyaranyapura	Devanahalli
3331	Robert White	Jeffrey Thompson	Indiranagar	Gottigere

Query statistics-

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00620370
Table lock wait time: 0:00:0.00000600

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

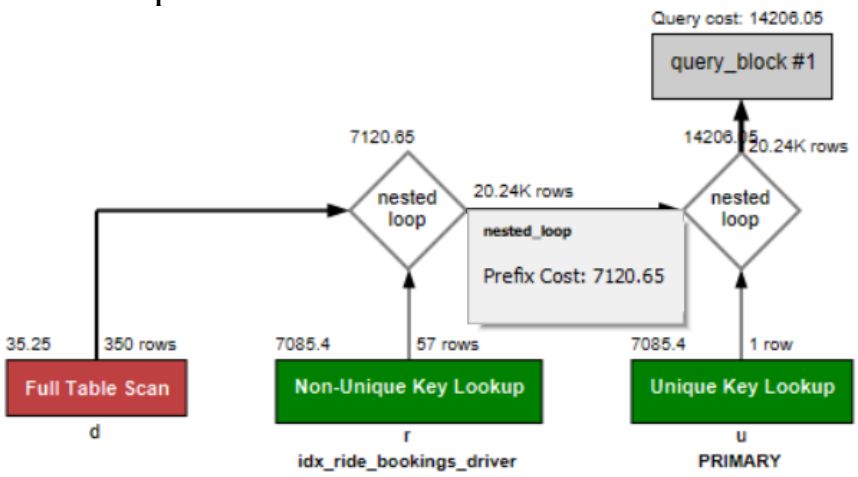
Explain-analyze -

Form Editor | Navigate:

EXPLAIN:

```
-> Nested loop inner join  (cost=14206 rows=20244) (actual time=0.269..92.8 rows=20000 loops=1)
  -> Nested loop inner join  (cost=7121 rows=20244) (actual time=0.262..53.2 rows=20000 loops=1)
    -> Table scan on d  (cost=35.2 rows=350) (actual time=0.0779..0.337 rows=350 loops=1)
    -> Filter: (r.user_id is not null)  (cost=14.5 rows=57.8) (actual time=0.109..0.145 rows=57.1 loops=350)
```

Execution plan -



This query is a modification of the query in E.g. 2, i.e. it performs a select on a subset of columns of the result from the join. Here, we only extract the relevant columns (i.e. booking_id, user_name, driver_name, pickup_location, drop_location) as opposed to all columns being extracted in the previous query.
Foreign keys are not automatically indexed by default, due to which a full table scan is being performed on the DRIVERS table.

E.g. 3)

Query -

```
52 •  select *
53    from drivers d
54    join vehicles v on d.driver_id = v.driver_id
55    where v.vehicle_status = 'active';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

driver_id	driver_name	driver_status	star_rating	vehicle_id	vehicle_number	driver_id	vehicle_model	seating_capacity	vehicle_status
3	Victor Cooper	unavailable	5	3	KA47R4197	3	Hyundai i10	4	active
4	Joseph Watson	unavailable	2	4	KA80A1286	4	Hyundai i20	4	active
8	David Martin	available	5	8	KA55N1690	8	Suzuki Baleno	4	active
9	Gregory Acosta	available	2	9	KA71G7711	9	Suzuki Swift	4	active
12	Pamela Sanders	unavailable	1	12	KA72O6278	12	Suzuki Swift Dezire	4	active
17	Eric Booth	unavailable	5	17	KA78W2505	17	Hyundai i10	4	active
26	Jennifer Grant	unavailable	4	26	KA15M5333	26	Hyundai Creta	7	active
27	Kevin Black	available	1	27	KA88F8958	27	Suzuki Swift	4	active

Query statistics -

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.01600000

Timing (as measured by the server):
Execution time: 0:00:0.01177020
Table lock wait time: 0:00:0.00000800

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

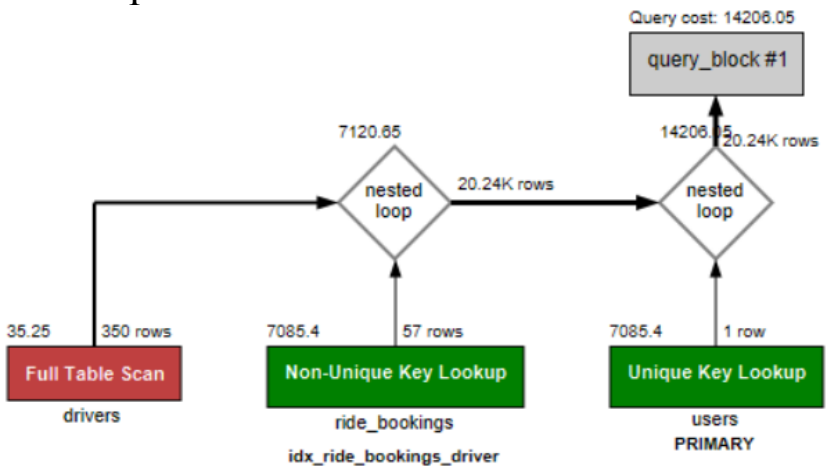
Explain-analyze -

Form Editor | Navigate:

EXPLAIN:

```
-> Nested loop inner join  (cost=46.3 rows=98) (actual time=3.93..4.22 rows=98 loops=1)
    -> Filter: (v.driver_id is not null)  (cost=12.1 rows=98) (actual time=3.91..3.96 rows=98 loops=1)
        -> Index lookup on v using idx_vehicles_status (vehicle_status='active'), with index condition: (v.vehicle_status = 'active')  (cost=12.1 rows=98) (actual time=3.9..3.94 rows=98
loops=1)
```

Execution plan-



This query joins the DRIVERS and VEHICLES tables on the common attribute 'driver_id' and selects only the tuples where the driver_status is 'active'.

Foreign keys are not automatically indexed by default, due to which a full table scan is being performed on the DRIVERS table.

E.g. 4)

Query -

```
58 •  select rb.booking_id, rb.pickup_location, rb.drop_location, rb.ride_cost, rb.booking_status, p.payment_status
59    from ride_bookings rb
60    join users u on rb.user_id = u.user_id
61    join payments p on rb.booking_id = p.booking_id
62    where u.user_name = 'TEJAS GOWRISH';
```

Running explain analyze-

EXPLAIN:

```
-> Nested loop inner join  (cost=1571 rows=2028) (actual time=0.109..0.699 rows=13 loops=1)
  -> Nested loop inner join  (cost=862 rows=2026) (actual time=0.101..0.625 rows=16 loops=1)
    -> Filter: (u.user_name = 'TEJAS GOWRISH')  (cost=153 rows=150) (actual time=0.0479..0.563 rows=1 loops=1)
      -> Table scan on u  (cost=153 rows=1501) (actual time=0.0451..0.463 rows=1501 loops=1)
```

Query statistics-

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.01315820
Table lock wait time: 0:00:0.00000500

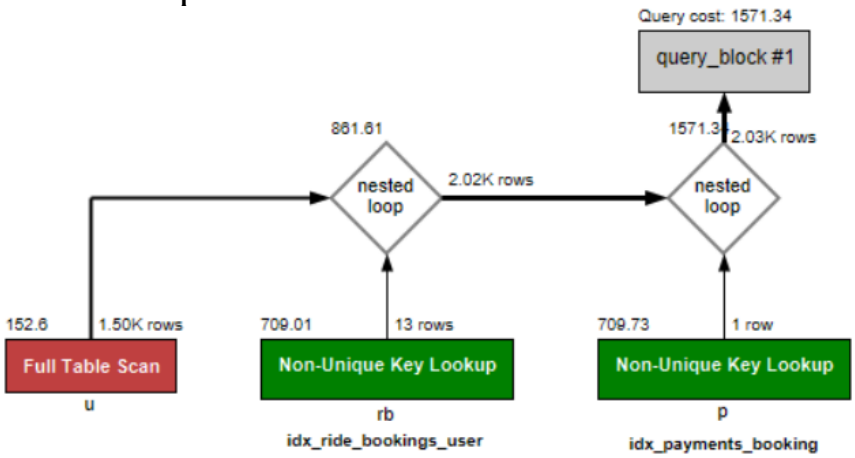
Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Output -

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

booking_id	pickup_location	drop_location	ride_cost	booking_status	payment_status
2575	Domlur	Panathur	498.58	confirmed	failed
2753	Doddathoguru	Vijayanagar	299.89	confirmed	failed
3898	Hennur	CV Raman Nagar	153.22	pending	failed
7938	Bilekahalli	Huskur	395.53	arrived	processing
8394	Konanakunte	Kormangala	447.27	on-the-way	processing
9588	Thippasandra	Nagawara	492.62	confirmed	completed
10046	Airport Road	Kadugodi	169.46	confirmed	completed
10940	Huskur	Bannerghatta Road	476.74	pending	processing
12241	Rachenahalli	Begur	151.53	on-the-way	completed
14144	Anjanapura	Kanakapura Road	192.05	on-the-way	completed
17589	Vijayanagar	Rajajinagar	311.53	pending	processing
18308	Mahadevapura	Yelahanka	298.78	confirmed	processing
19914	Kengeri	Yeshwantpur	486.90	arrived	processing

Execution plan -



This query selects the select booking_id, pickup_location, drop_location, ride_cost, booking_status and payment_status of all bookings made by the user ‘TEJAS GOWRISH’. An inner join is applied to the USERS and RIDE_BOOKINGS tables on the common attribute ‘user_id’. The second join is applied to the DRIVERS and RIDE_BOOKINGS tables on the common attribute ‘booking_id’.

Foreign keys are not automatically indexed by default, due to which a full table scan is being performed on the DRIVERS table.

d) Query Optimization with Varied Join Orders and Types:

Changing the order of the 'JOIN' operations:

Consider a query to retrieve ride details, including user name, driver name, ride cost, and payment status of ride bookings where the payment was complete.

- Order 1-

```
select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost, p.payment_status
from ride_bookings rb
join users u on rb.user_id = u.user_id
join drivers d on rb.driver_id = d.driver_id
join payments p on rb.booking_id = p.booking_id
where p.payment_status = 'completed';
```

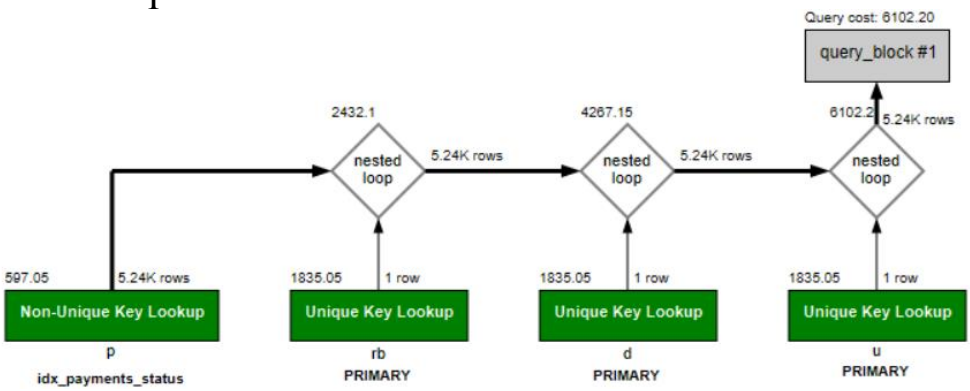
Query statistics -

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.01284400
Table lock wait time: 0:00:0.00000600

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Execution plan -



Explain-analyze -

EXPLAIN:

```
-> Nested loop inner join  (cost=6102 rows=5243) (actual time=0.151..25.7 rows=5243 loops=1)
  -> Nested loop inner join  (cost=4267 rows=5243) (actual time=0.148..20.4 rows=5243 loops=1)
    -> Nested loop inner join  (cost=2432 rows=5243) (actual time=0.144..15.6 rows=5243 loops=1)
      -> Filter: (p.booking_id is not null)  (cost=597 rows=5243) (actual time=0.136..8.85 rows=5243 loops=1)
```

Potential issues-
If the ride_bookings table is very large, joining all the tables may slow down execution.

- Order 2-

```
select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost, p.payment_status
from payments p
join ride_bookings rb on p.booking_id = rb.booking_id
join users u on rb.user_id = u.user_id
join drivers d on rb.driver_id = d.driver_id
where p.payment_status = 'completed';
```

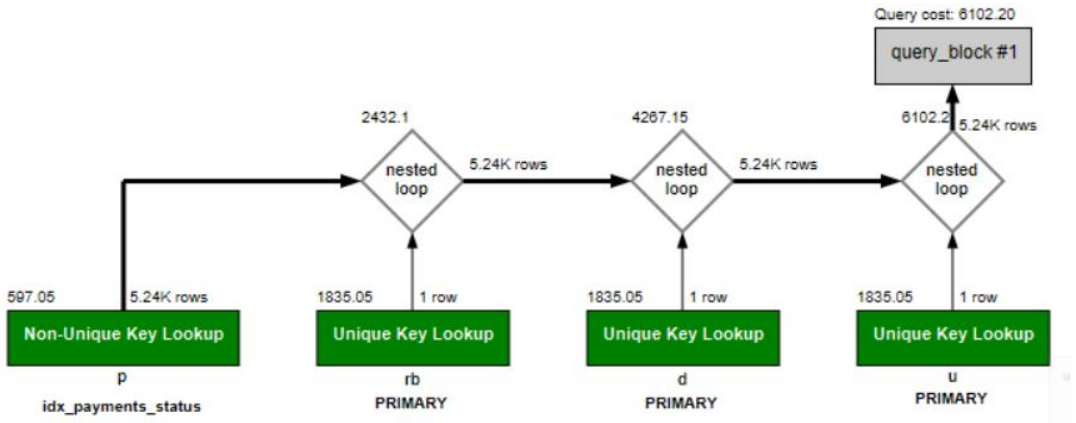
Query statistics-

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00744340
Table lock wait time: 0:00:0.00000400

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Execution plan-



Explain anaylze-

```
EXPLAIN:
-> Nested loop inner join  (cost=6102 rows=5243) (actual time=0.17..26 rows=5243 loops=1)
    -> Nested loop inner join  (cost=4267 rows=5243) (actual time=0.165..20.2 rows=5243 loops=1)
        -> Nested loop inner join  (cost=2432 rows=5243) (actual time=0.161..14.9 rows=5243 loops=1)
            -> Filter: (p.booking_id is not null)  (cost=597 rows=5243) (actual time=0.152..7.68 rows=5243 loops=1)
```

As we can see, there is a slight reduction in the execution time. However, the query cost remains the same. This shows how the query optimizer always chooses the plan it deems most optimal in order to minimize the query cost.

Output (in both cases)-

booking_id	user_name	driver_name	ride_cost	payment_status
4	Monica Pennington	Justin Wallace	373.43	completed
8	Tammy Lloyd	Deborah Gomez	421.96	completed
11	Jenna Le	Stephanie Jones	492.38	completed
12	Jenny Schneider	Valerie Jones	224.47	completed
13	Jeremy West	Deborah Myers	220.65	completed
17	Monica Collins	Michelle Nguyen	64.98	completed
23	Mark Olsen	Jeffrey Kirk	185.00	completed

Using different types of JOIN:

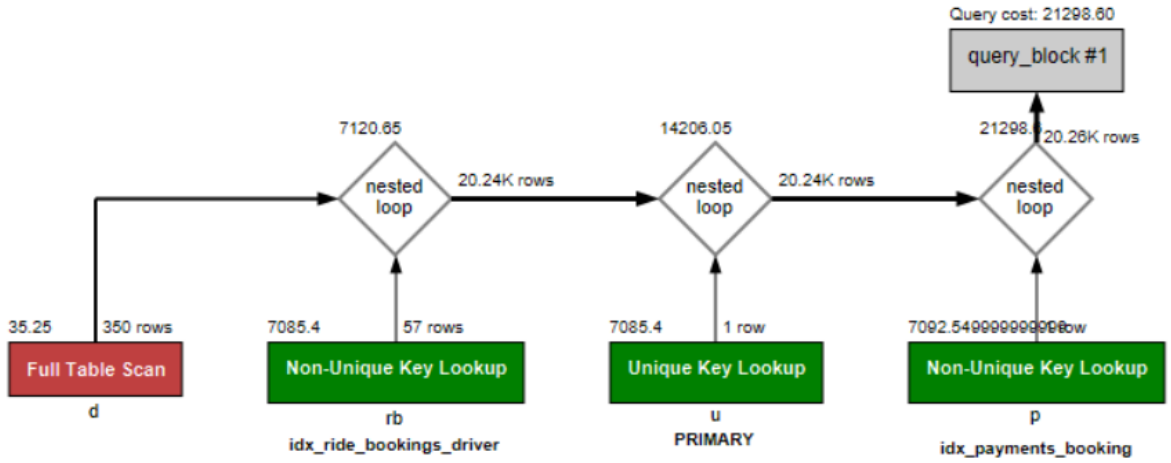
- **Left Join:**

Query-

```
select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost, p.payment_status
from ride_bookings rb
join users u on rb.user_id = u.user_id
join drivers d on rb.driver_id = d.driver_id
left join payments p on rb.booking_id = p.booking_id;
```

(To extract booking information – booking_id, user_name, driver_name, ride_cost and payment_status irrespective of the payment_status)

Execution plan-



Output -

	booking_id	user_name	driver_name	ride_cost	payment_status
▶	124	Jody Bishop	Mrs. Sierra Torres MD	84.02	processing
	943	Jeremy West	Mrs. Sierra Torres MD	75.32	failed
	1133	Jennifer Garrett	Mrs. Sierra Torres MD	294.03	processing
	1505	Timothy Harrison	Mrs. Sierra Torres MD	428.76	completed
	2283	Lindsay Scott	Mrs. Sierra Torres MD	276.47	failed
	2294	Kendra Shepard	Mrs. Sierra Torres MD	333.59	processing
	2540	Lori Foley	Mrs. Sierra Torres MD	232.41	completed
	3533	Danielle Bauer	Mrs. Sierra Torres MD	205.54	completed

Explain analyze-

EXPLAIN:

```
-> Nested loop inner join  (cost=6102 rows=5243) (actual time=0.192..29 rows=5243 loops=1)
    -> Nested loop inner join  (cost=4267 rows=5243) (actual time=0.188..22.7 rows=5243 loops=1)
        -> Nested loop inner join  (cost=2432 rows=5243) (actual time=0.184..16.8 rows=5243 loops=1)
            -> Filter: (p.booking_id is not null)  (cost=597 rows=5243) (actual time=0.175..8.58 rows=5243 loops=1)
```

- **Inner join with the ‘exists’ clause:**

Query-

```
select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost
from ride_bookings rb
join users u on rb.user_id = u.user_id
join drivers d on rb.driver_id = d.driver_id
where exists (
    select 1 from payments p where p.booking_id = rb.booking_id and p.payment_status = 'completed'
);
```

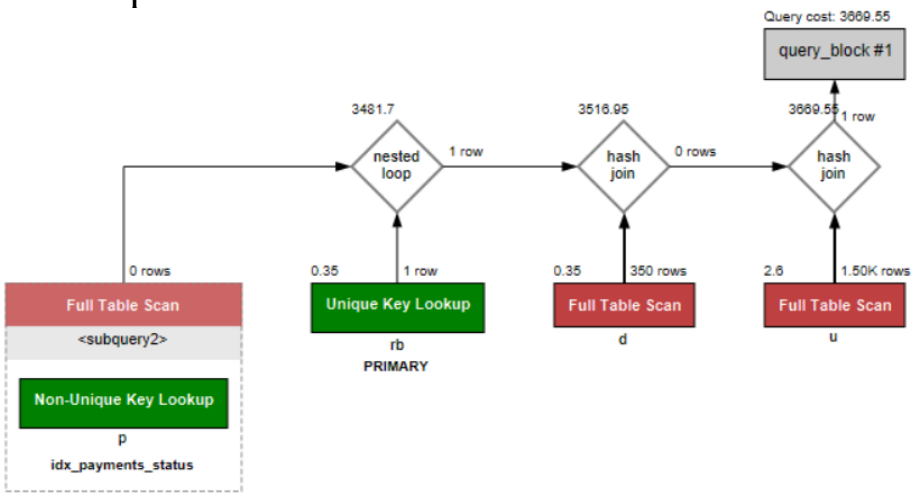
(To extract ride, user and driver info by checking if a payment exists. This eliminates the need to include the PAYMENTS table)

Output-

Result Grid | Filter Rows: | Export:

booking_id	user_name	driver_name	ride_cost
10046	TEJAS GOWRISH	Katherine Lucas	169.46
14144	TEJAS GOWRISH	Gerald Davis	192.05
9588	TEJAS GOWRISH	Tonya Griffin	492.62
12241	TEJAS GOWRISH	Tonya Griffin	151.53
13782	Isaac Watts MD	Todd Lucas	323.44
4912	Isaac Watts MD	Jason Tapia	434.97
17186	Isaac Watts MD	Joseph Johnson	382.68

Execution plan-



Explain analyze-

```
EXPLAIN: -> Inner hash join (u.user_id = rb.user_id)  (cost=972055 rows=5243) (actual time=25.1..30.1 rows=5243 loops=1)
    -> Table scan on u  (cost=2.6 rows=1501) (actual time=0.0366..0.381 rows=1501 loops=1)
    -> Hash
        -> Inner hash join (d.driver_id = rb.driver_id)  (cost=185078 rows=5243) (actual time=18.8..19.5 rows=5243 loops=1)
```

As we can see, this query in particular is very inefficient due to the multiple table scans occurring. The full table scans are forced due to the absence of default indexes on foreign keys.

INNER JOIN returns only matching rows from all tables and is efficient when all records must be related.

On the other hand, LEFT JOIN retains all rows from one table even if there's no match in the joined table. It is useful for identifying missing records (e.g., unpaid rides), but can be slower when many NULL values exist.
EXISTS is optimized for boolean checks, avoiding unnecessary data retrieval by verifying if a related record exists.

Although changing the order of the join operations may improve query performance, it is often simpler and more efficient to use implicit joins (with comma-separated tables in the FROM clause and the conditions in the WHERE clause).
Please see section-e.

e) Query Analysis and Optimization:

Part 1: Query Analysis

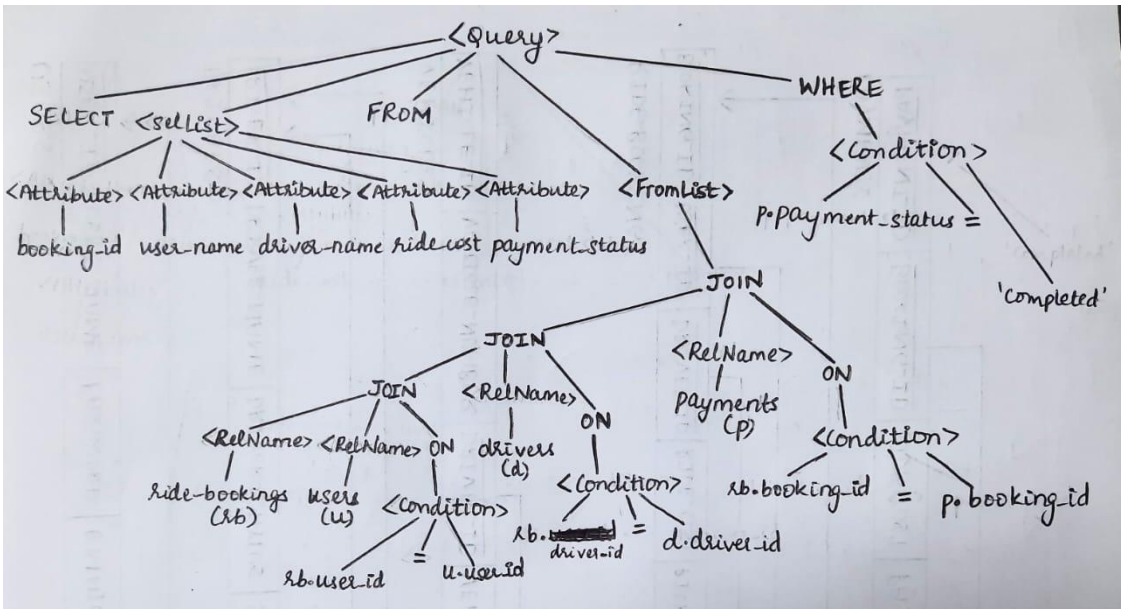
- Query-

```
select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost, p.payment_status
from ride_bookings rb
join users u on rb.user_id = u.user_id
join drivers d on rb.driver_id = d.driver_id
join payments p on rb.booking_id = p.booking_id
where p.payment_status = 'completed';
```

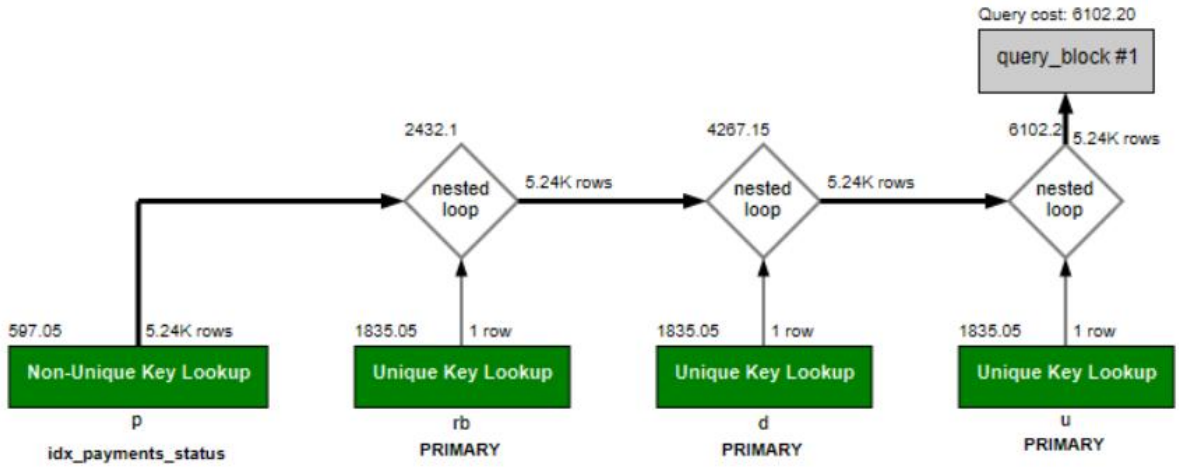
- Output-

booking_id	user_name	driver_name	ride_cost	payment_status
4	Monica Pennington	Justin Wallace	373.43	completed
8	Tammy Lloyd	Deborah Gomez	421.96	completed
11	Jenna Le	Stephanie Jones	492.38	completed
12	Jenny Schneider	Valerie Jones	224.47	completed
13	Jeremy West	Deborah Myers	220.65	completed
17	Monica Collins	Michelle Nguyen	64.98	completed
23	Mark Olsen	Jeffrey Kirk	185.00	completed
31	Christine Guzman	Kathleen Diaz	349.96	completed

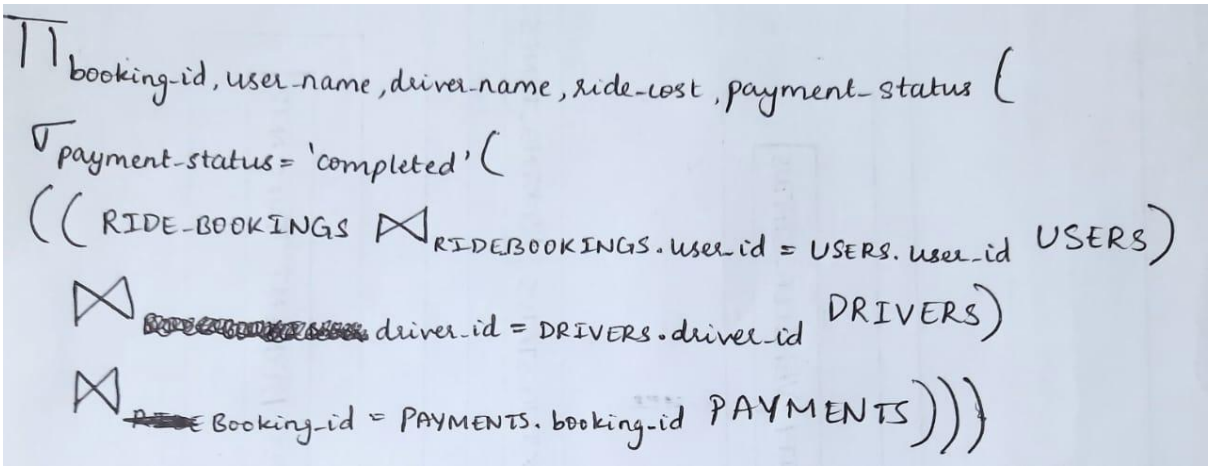
- Parse tree -



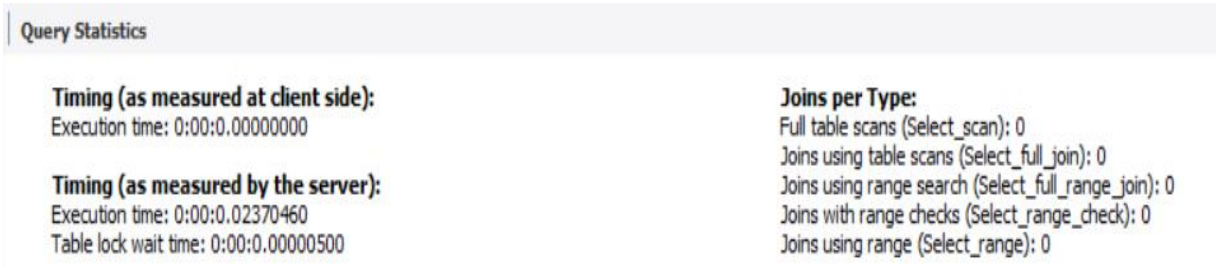
- Execution plan -



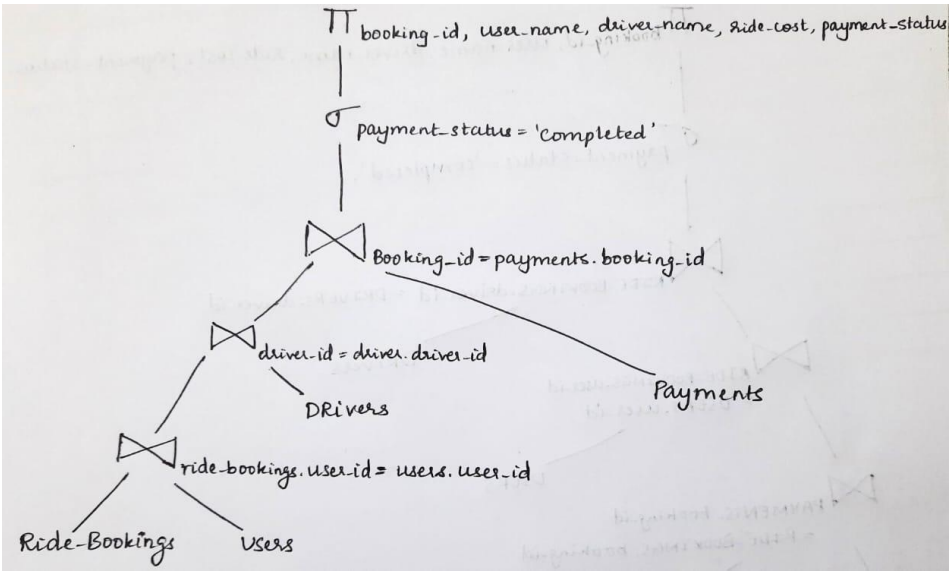
- Relational algebra expression -



- Query statistics -



- Initial query tree (expression tree) -



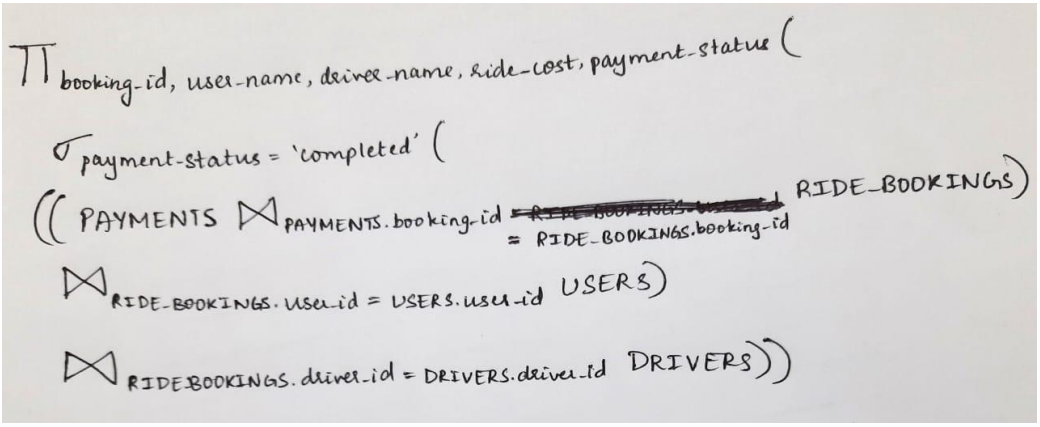
Part 2: Query Optimization

- Optimized query-

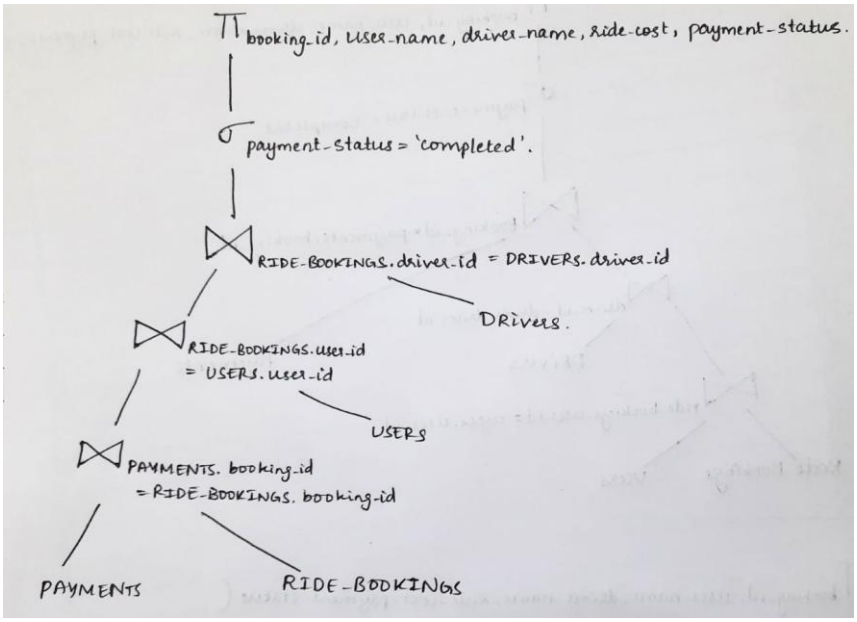
```
select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost, p.payment_status
from payments p
join ride_bookings rb on p.booking_id = rb.booking_id
join users u on rb.user_id = u.user_id
join drivers d on rb.driver_id = d.driver_id
where p.payment_status = 'completed';
```

- How this query is better -
  - It reduces the number of rows to be searched early on (It filters out rows where payment_status = ‘completed’).
  - Joins performed on relevant records
  - Joins are faster due to smaller intermediate table sizes

- Optimized relational algebra expression-



- Optimized expression tree -



Optimizing using implicit joins:

Changing the order of the join operations may improve query performance. However, it is often simpler and more efficient to use implicit joins (with comma-separated tables in the FROM clause and the conditions in the WHERE clause).

E.g –

- Query -

```
129 •   select rb.booking_id, u.user_name, d.driver_name, rb.ride_cost, p.payment_status
130     from payments p, ride_bookings rb, users u, drivers d
131     where p.payment_status = 'completed'
132     and p.booking_id = rb.booking_id
133     and rb.user_id = u.user_id
134     and rb.driver_id = d.driver_id;
```

- Output -

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

booking_id	user_name	driver_name	ride_cost	payment_status
4	Monica Pennington	Justin Wallace	373.43	completed
8	Tammy Lloyd	Deborah Gomez	421.96	completed
11	Jenna Le	Stephanie Jones	492.38	completed
12	Jenny Schneider	Valerie Jones	224.47	completed
13	Jeremy West	Deborah Myers	220.65	completed
17	Monica Collins	Michelle Nguyen	64.98	completed
23	Mark Olsen	Jeffrey Kirk	185.00	completed

- Query statistics -

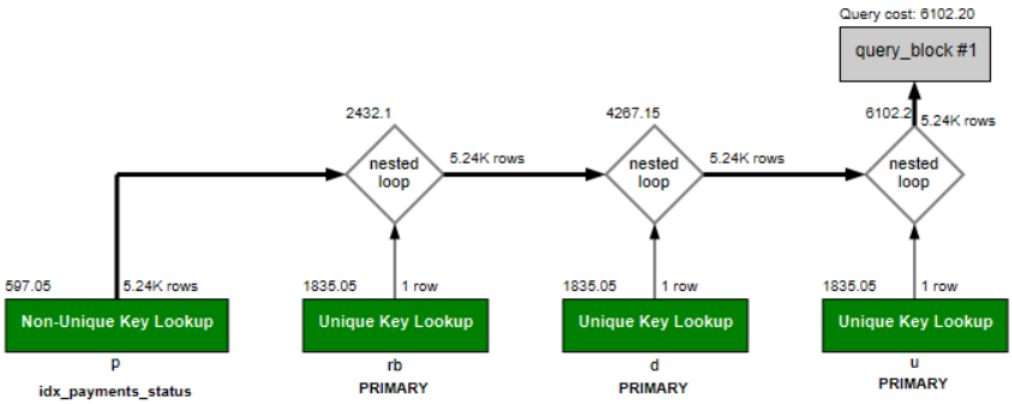
Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

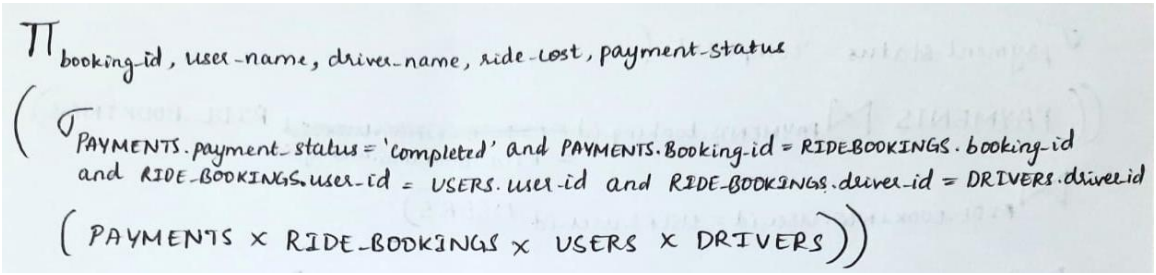
Timing (as measured by the server):
Execution time: 0:00:0.00958150
Table lock wait time: 0:00:0.00000500

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

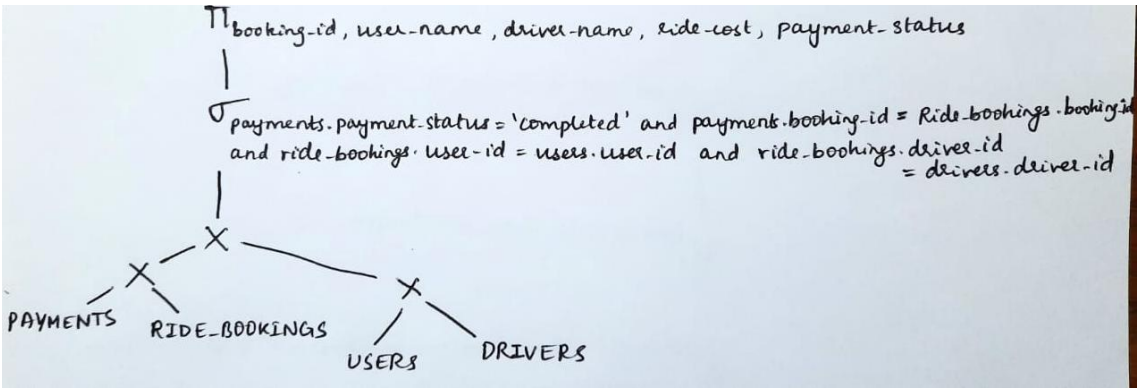
- Execution plan -



- Relational algebra expression -



- Expression tree-



The above query performs the joins implicitly using the conditions in the WHERE clause. These conditions are resolved into JOINS by the executor.
As we can see in the above screenshots, there is a reduction in the execution time. However, the execution plan remains the same, resulting in the same query cost. This is because the query optimizer chooses the most efficient exeuction plan regardless of whether the joins are written explicitly or implicitly.