

UNIVERSITY OF COMPUTER STUDIES, YANGON

Tour Booking System

Covered By

**DATABASE TRANSACTION MANAGEMENT**

Faculty of Information Science

University of Computer Studies, Yangon

Submitted By Semester – VI

September, 2025

Group Member

|  |  |  |
| --- | --- | --- |
| Roll No. | Name | Role |
| YKPT – 21871 | Htet Nadi | Leader |
| YKPT – 21924 | Ye Wont Aung | Co-Leader |
| YKPT – 22461 | Than Thar Linn Latt | Member |
| YKPT – 21809 | Phu Phu May Maung | Member |
| YKPT – 21442 | Shwe Sin Phoo Lwin | Member |
| YKPT – 21727 | Pan Kay Thwel Oo | Member |
| YKPT – 0000 | May Thin Nwe | Member |

**Contents**

**Chapter 1: Introduction**

1.1 Project Description 4

1.2 Objectives 4

**Chapter 2: Database Management System**

2.1 Entity Relationship (ER) Diagram 5

2.2 Data Dictionary 6

2.3 Data Insertion in Relational Tables 9

2.3.1 Table Creation 9

2.3.2 Data Insertion 12

**Chapter 3: Functionality**

3.1 Functionalities that Admin can perform 15

3.1.1 Procedure for Admin 15

3.1.2 Function for Admin 16

3.1.3 Trigger for Admin 17

3.2 Functionalities that Customer can perform 18

3.2.1 Procedure for Customer 18

3.2.2 Function for Customer 20

3.2.3 Trigger for Customer 21

3.3 Query Execution with MySQL 22

3.3.1 Index Query 22

3.3.2 Query Evaluation Plans 24

3.3.3 Implementing Transformations Based Optimizations 26

**Chapter 4: Conclusion**

4.1 Conclusion 29

4.2 References 29

**Chapter 1**

**Introduction**

* 1. **Project Description**

The Tour Booking System is a comprehensive database-driven system designed to manage tour packages, customer bookings, and payment processing for a travel agency. The system provides an organized and scalable way to handle administrative tasks, customer interactions, and financial transactions while ensuring data integrity, efficiency, and security.

This database system enables administrators to manage tour packages, monitor bookings, and track payments, while customers can view available tours, make reservations, and check booking details. The system enforces role-based access control, ensuring that only authorized administrators can modify packages or confirm payments, while customers have restricted access for making bookings and viewing their own records.

* 1. **Objectives**

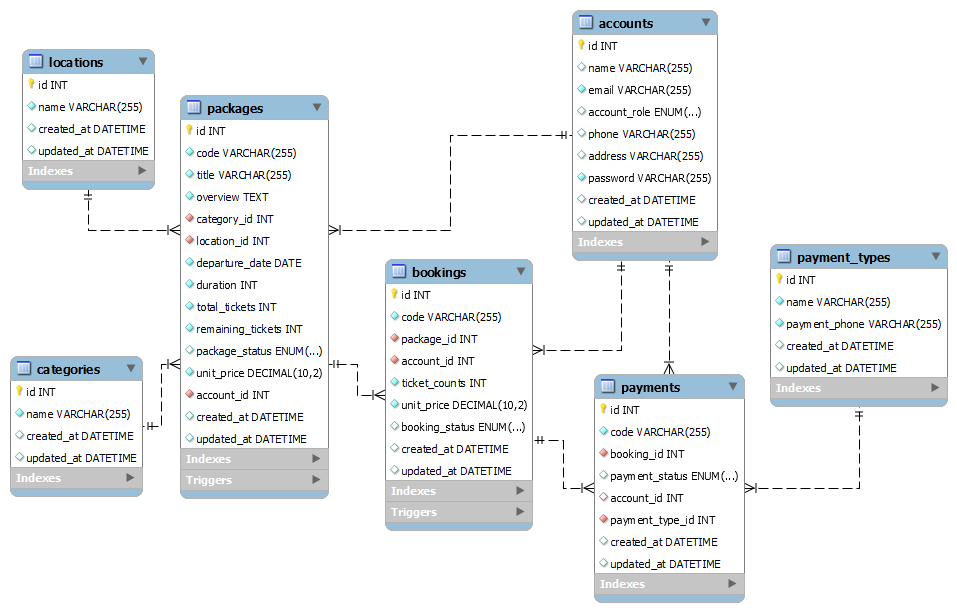
The objectives of the Tour Booking System are:

1. Efficient Tour Package Management
2. Streamlined Booking Process
3. Secure Payment Management
4. Role-Based Access Control
5. Data Integrity and Validation

**Chapter 2**

**Database Management System**

**2.1 Entity Relationship (ER) Diagram**



**Figure 2.1 ER Diagram**

**2.2 Data Dictionary**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tour Booking System | | | | | | |
| Table | **Field** | **Type** | **Null** | **Default** | **Key** | **Foreign Key Table** |
| accounts | id | int | No |  | PK |  |
| name | varchar(255) | Yes |  |  |  |
| email | varchar(255) | No |  | UNI |  |
| account\_role | enum | Yes | ‘CUSTOMER’ |  |  |
| phone | varchar(255) | Yes |  |  |  |
| address | varchar(255) | Yes |  |  |  |
| password | varchar(255) | No |  |  |  |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |
|  |  |  |  |  |  |  |
| categories | id | int | No |  | PK |  |
| name | varchar(255) | No |  | UNI |  |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |
|  |  |  |  |  |  |  |
| locations | id | int | No |  | PK |  |
| name | varchar(255) | No |  | UNI |  |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |
|  |  |  |  |  |  |  |
| packages | id | int | No |  | PK |  |
| code | varchar(255) | No |  | UNI |  |
| title | varchar(255) | No |  |  |  |
| overview | text | Yes |  |  |  |
| category\_id | int | No |  | FK | categories |
| location\_id | int | No |  | FK | locations |
| departure\_date | date | No |  |  |  |
| duration | int | No |  |  |  |
| total\_tickets | int | No |  |  |  |
| remaining\_tickets | int | No |  |  |  |
| package\_status | enum | No | ‘AVAILABLE' |  |  |
| unit\_price | decimal(10.2) | No |  |  |  |
| account\_id | int | No |  | FK | accounts |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |
|  |  |  |  |  |  |  |
| bookings | id | int | No |  | PK |  |
| code | varchar(255) | No |  | UNI |  |
| package\_id | varchar(255) | No |  | FK | packages |
| account\_id | int | No |  | FK | accounts |
| ticket\_counts | int | No |  |  |  |
| unit\_price | decimal(10.2) | No |  |  |  |
| booking\_status | enum | Yes | ‘PENDING' |  |  |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |
|  |  |  |  |  |  |  |
| payment\_types | id | int | No |  | PK |  |
| name | varchar(255) | No |  | UNI |  |
| payment\_phone | varchar(255) | No |  |  |  |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |
|  |  |  |  |  |  |  |
| payments | id | int | No |  | PK |  |
| code | varchar(255) | No |  | UNI |  |
| booking\_id | int | No |  | FK | bookings |
| payment\_status | enum | Yes | ‘PENDING' |  |  |
| account\_id | Int | Yes | NULL | FK | accounts |
| payment\_type\_id | int | No |  | FK | payment\_types |
| created\_at | datetime | Yes | current\_timestamp |  |  |
| updated\_at | datetime | Yes | current\_timestamp |  |  |

**2.3 Data Insertion in Relational Tables**

The following figures are data inserted into relational tables.

**2.3.1 Table Creation**

create table accounts (

id int auto\_increment primary key,

name varchar(255),

email varchar(255) not null unique,

account\_role enum('ADMIN', 'CUSTOMER') default 'CUSTOMER',

phone varchar(255) null,

address varchar(255) null,

password varchar(255) not null,

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP

);

create table categories (

id int auto\_increment primary key,

name varchar(255) not null unique,

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP

);

create table locations (

id int auto\_increment primary key,

name varchar(255) not null unique,

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP

);

create table packages (

id int auto\_increment primary key,

code varchar(255) not null unique,

title varchar(255) not null,

overview text not null,

category\_id int not null,

location\_id int not null,

departure\_date date not null,

duration int not null,

total\_tickets int not null,

remaining\_tickets int not null,

package\_status enum('AVAILABLE', 'UNAVAILABLE', 'FINISHED') default 'AVAILABLE',

unit\_price decimal(10, 2) not null,

account\_id int not null,

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP,

foreign key (category\_id) references categories (id),

foreign key (location\_id) references locations (id),

foreign key (account\_id) references accounts (id),

check (remaining\_tickets <= total\_tickets)

);

create table payment\_types (

id int auto\_increment primary key,

name varchar(255) not null unique,

payment\_phone varchar(255) not null,

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP

);

create table bookings (

id int auto\_increment primary key,

code varchar(255) not null unique,

package\_id int not null,

account\_id int not null,

ticket\_counts int not null,

unit\_price decimal(10, 2) not null,

booking\_status enum('PENDING', 'REQUESTING', 'RESERVED', 'CANCELLED') default 'PENDING',

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP,

foreign key (package\_id) references packages (id),

foreign key (account\_id) references accounts (id)

);

create table payments (

id int auto\_increment primary key,

code varchar(255) not null unique,

booking\_id int not null,

payment\_status enum('PENDING', 'SUCCESS', 'FAIL') default 'PENDING',

account\_id int null,

payment\_type\_id int not null,

created\_at datetime default CURRENT\_TIMESTAMP,

updated\_at datetime default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP,

foreign key (booking\_id) references bookings (id),

foreign key (account\_id) references accounts (id),

foreign key (payment\_type\_id) references payment\_types (id)

);

**2.3.2 Data Insertion**

**Create Admin Accounts**

INSERT INTO accounts (name, email, password, phone, account\_role) VALUES

('admin1', 'admin1@gmail.com', 'password', '09999888777', 'ADMIN'),

('admin2', 'admin2@gmail.com', 'password', '09999888666', 'ADMIN');

**Create Customer Accounts**

INSERT INTO accounts (name, email, password, phone) VALUES

('Aung Aung', 'aung@gmail.com', 'password', '09777888777'),

('Su Su', 'su@gmail.com', 'password', '09777888666'),

('Moe Moe', 'moe@gmail.com', 'password', '09777888555'),

('Hla Hla', 'hla@gmail.com', 'password', '09777888444');

**Create Categories**

INSERT INTO categories (name) VALUES

('Relaxation'), ('Pagoda'), ('Beach'), ('History');

**Create Locations**

INSERT INTO locations (name) VALUES

('Yangon'),

('Bagan'),

('Chaung Tha'),

('Mandalay'),

('Naypyitaw'),

('Inle Lake');

**Create Payment Types**

INSERT INTO payment\_types (name, payment\_phone) VALUES

('KBZ Pay', '09990001111'),

('Wave Pay', '09990002222'),

('AYA Pay', '09990003333');

**Create Packages**

INSERT INTO packages

(code, title, overview, category\_id, location\_id, departure\_date, duration, total\_tickets, remaining\_tickets, unit\_price, account\_id, package\_status)

VALUES

-- Category 1 (Relaxation)

('PKG-001', 'Adventure in Mountains', 'Explore the majestic mountains with guided tours.', 1, 1, '2025-10-01', 5, 20, 18, 500.00, 1, 'AVAILABLE'),

('PKG-002', 'Mountain Hiking Challenge', 'A thrilling hiking experience for adventure seekers.', 1, 2, '2025-11-05', 7, 15, 12, 650.00, 1, 'AVAILABLE'),

-- Category 2 (Pagoda)

('PKG-003', 'Beach Relaxation Getaway', 'Relax on pristine beaches with all-inclusive amenities.', 2, 3, '2025-09-15', 4, 30, 29, 400.00, 1, 'AVAILABLE'),

('PKG-004', 'Sunset Cruise Escape', 'Enjoy sunset cruises and beach parties.', 2, 4, '2025-10-20', 3, 25, 24, 350.00, 1, 'AVAILABLE'),

-- Category 3 (Beach)

('PKG-005', 'City Cultural Tour', 'Discover historical landmarks and local culture.', 3, 5, '2025-12-01', 6, 20, 18, 300.00, 1, 'AVAILABLE'),

('PKG-006', 'Nightlife Exploration', 'Experience the vibrant city nightlife and local cuisine.', 3, 6, '2025-12-10', 5, 15, 15, 320.00, 1, 'AVAILABLE'),

-- Category 4 (History)

('PKG-007', 'Ancient Pagoda Tour', 'Visit historic pagodas and learn about Burmese history.', 4, 2, '2025-11-12', 4, 25, 25, 450.00, 1, 'AVAILABLE'),

('PKG-008', 'Historical Yangon Walk', 'A walking tour through Yangon''s historical sites.', 4, 1, '2025-12-05', 3, 20, 20, 280.00, 1, 'AVAILABLE'),

-- Extra Package: UNAVAILABLE (sold out)

('PKG-009', 'Hidden Lakes Adventure', 'A secret journey to hidden lakes with full bookings.', 1, 6, '2025-11-20', 3, 10, 0, 600.00, 1, 'UNAVAILABLE'),

-- Extra Package: FINISHED (departure already past)

('PKG-010', 'Old Kingdom Exploration', 'Historic kingdom tour, already departed.', 4, 2, '2025-08-01', 4, 12, 5, 700.00, 1, 'FINISHED');

**Create Bookings**

INSERT INTO bookings

(code, package\_id, account\_id, ticket\_counts, unit\_price, booking\_status)

VALUES

('BOOK-001', 1, 3, 2, 500.00, 'PENDING'),

('BOOK-002', 3, 4, 1, 400.00, 'RESERVED'),

('BOOK-003', 2, 3, 3, 650.00, 'REQUESTING'),

('BOOK-004', 5, 4, 2, 300.00, 'PENDING'),

('BOOK-005', 4, 3, 1, 350.00, 'CANCELLED');

**Create Payments**

INSERT INTO payments

(code, booking\_id, payment\_status, account\_id, payment\_type\_id)

VALUES

('PAY-001', 1, 'PENDING', NULL, 1),

('PAY-002', 2, 'SUCCESS', 4, 2),

('PAY-003', 3, 'FAIL', 3, 1),

('PAY-004', 4, 'PENDING', NULL, 3),

('PAY-005', 5, 'SUCCESS', 3, 2);

**Chapter 3**

**Functionality**

**3.1 Functionalities that Admin can perform**

**3.1.1 Procedure for Admin**

**Confirming the payment and updating the booking status**

DELIMITER //

CREATE PROCEDURE ConfirmPayment(

IN paymentId INT,

IN adminId INT

)

BEGIN

DECLARE bookingId INT;

DECLARE currentStatus ENUM('PENDING','SUCCESS','FAIL');

-- 1. Get current payment info

SELECT booking\_id, payment\_status

INTO bookingId, currentStatus

FROM payments

WHERE id = paymentId;

-- 2. Only proceed if payment is PENDING

IF currentStatus = 'PENDING' THEN

-- 3. Update payment status to SUCCESS and record admin who confirmed

UPDATE payments

SET payment\_status = 'SUCCESS',

account\_id = adminId,

updated\_at = NOW()

WHERE id = paymentId;

-- 4. Update related booking status to RESERVED

UPDATE bookings

SET booking\_status = 'RESERVED',

updated\_at = NOW()

WHERE id = bookingId;

ELSE

-- Optional: raise an error if payment already confirmed or failed

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Payment is not in PENDING status';

END IF;

END;

//

DELIMITER ;

-- Example: Get the total bookings for package with ID 1

SELECT GetBookingCount(1) AS total\_bookings;

**3.1.2 Function for Admin**

**Function for getting booking count of a package**

DELIMITER //

CREATE FUNCTION GetBookingCount(p\_id INT)

RETURNS INT

DETERMINISTIC

BEGIN

DECLARE total INT;

SELECT COUNT(\*)

INTO total

FROM bookings

WHERE package\_id = p\_id;

RETURN total;

END;

//

DELIMITER ;

-- Example: Admin with ID 1 confirms payment with ID 2

CALL ConfirmPayment(2, 1);

**3.1.3 Trigger for Admin**

**Check if account is admin before inserting package**

DELIMITER //

CREATE TRIGGER CheckAdminRoleBeforePackage

BEFORE INSERT ON packages

FOR EACH ROW

BEGIN

DECLARE user\_role ENUM('ADMIN','CUSTOMER');

-- Get the role of the account creating the package

SELECT account\_role INTO user\_role

FROM accounts

WHERE id = NEW.account\_id;

-- If not an ADMIN, throw an error

IF user\_role <> 'ADMIN' THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Only admins can create packages!';

END IF;

END;

//

DELIMITER ;

-- Example 1: Admin creating a package (succeeds)

INSERT INTO packages

(code, title, overview, category\_id, location\_id, departure\_date, duration, total\_tickets, remaining\_tickets, unit\_price, account\_id, package\_status)

VALUES

('PKG-011', 'New Adventure', 'Exciting new tour.', 1, 1, '2025-11-25', 4, 20, 20, 500.00, 1, 'AVAILABLE');

-- Example 2: Non-admin trying to create a package (fails)

INSERT INTO packages

(code, title, overview, category\_id, location\_id, departure\_date, duration, total\_tickets, remaining\_tickets, unit\_price, account\_id, package\_status)

VALUES

('PKG-012', 'Unauthorized Tour', 'Should fail.', 1, 1, '2025-12-01', 5, 15, 15, 400.00, 3, 'AVAILABLE');

**3.2 Functionalities that Customer can perform**

**3.2.1 Procedure for Customer**

**Update remaining tickets and packages status from packages when a booking is cancelled**

DELIMITER //

CREATE PROCEDURE CancelBooking(IN bookingId INT)

BEGIN

DECLARE pkgId INT;

DECLARE tickets INT;

DECLARE current\_status ENUM('PENDING','REQUESTING','RESERVED','CANCELLED');

DECLARE pkg\_remaining INT;

-- 1. Get booking info

SELECT package\_id, ticket\_counts, booking\_status

INTO pkgId, tickets, current\_status

FROM bookings

WHERE id = bookingId;

-- 2. Only proceed if booking is not already cancelled

IF current\_status <> 'CANCELLED' THEN

-- 3. Update booking status to CANCELLED

UPDATE bookings

SET booking\_status = 'CANCELLED', updated\_at = NOW()

WHERE id = bookingId;

-- 4. Add tickets back to the package

UPDATE packages

SET remaining\_tickets = remaining\_tickets + tickets,

updated\_at = NOW()

WHERE id = pkgId;

-- 5. Check and update package status

SELECT remaining\_tickets INTO pkg\_remaining

FROM packages

WHERE id = pkgId;

-- If remaining tickets > 0 and departure date in future, set to AVAILABLE

IF pkg\_remaining > 0 THEN

UPDATE packages

SET package\_status = 'AVAILABLE',

updated\_at = NOW()

WHERE id = pkgId;

END IF;

ELSE

-- Optional: raise an error if already cancelled

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Booking is already cancelled';

END IF;

END;

//

DELIMITER ;

CALL CancelBooking(3);

**3.2.2 Function for Customer**

**Calculate the total price of a booking**

DELIMITER //

CREATE FUNCTION TotalBookingPriceByBooking(b\_id INT)

RETURNS DECIMAL(10,2)

DETERMINISTIC

BEGIN

DECLARE unit DECIMAL(10,2);

DECLARE tickets INT;

DECLARE total DECIMAL(10,2);

-- Get unit price and ticket count from bookings table

SELECT unit\_price, ticket\_counts

INTO unit, tickets

FROM bookings

WHERE id = b\_id;

-- Calculate total price

SET total = unit \* tickets;

RETURN total;

END;

//

DELIMITER ;

-- example

SELECT

b.id AS booking\_id,

b.code AS booking\_code,

a.name AS customer\_name,

p.title AS package\_title,

b.ticket\_counts,

b.unit\_price,

TotalBookingPriceByBooking(b.id) AS total\_price,

b.booking\_status

FROM bookings b

JOIN accounts a ON b.account\_id = a.id

JOIN packages p ON b.package\_id = p.id

ORDER BY b.id;

**3.2.3 Trigger for Customer**

**Check if account is customer before booking**

DELIMITER //

CREATE TRIGGER CheckCustomerRoleBeforeBooking

BEFORE INSERT ON bookings

FOR EACH ROW

BEGIN

DECLARE user\_role ENUM('ADMIN','CUSTOMER');

-- Get the role of the account making the booking

SELECT account\_role INTO user\_role

FROM accounts

WHERE id = NEW.account\_id;

-- If not a CUSTOMER, throw an error

IF user\_role <> 'CUSTOMER' THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Only customers can create bookings!';

END IF;

END;

//

DELIMITER ;

--example

-- Assuming account\_id = 1 is an ADMIN

INSERT INTO bookings (code, package\_id, account\_id, ticket\_counts, unit\_price)

VALUES ('BOOK-007', 1, 1, 1, 500.00);

**3.3 Query Execution with MySQL**

**3.3.1 Index Query**

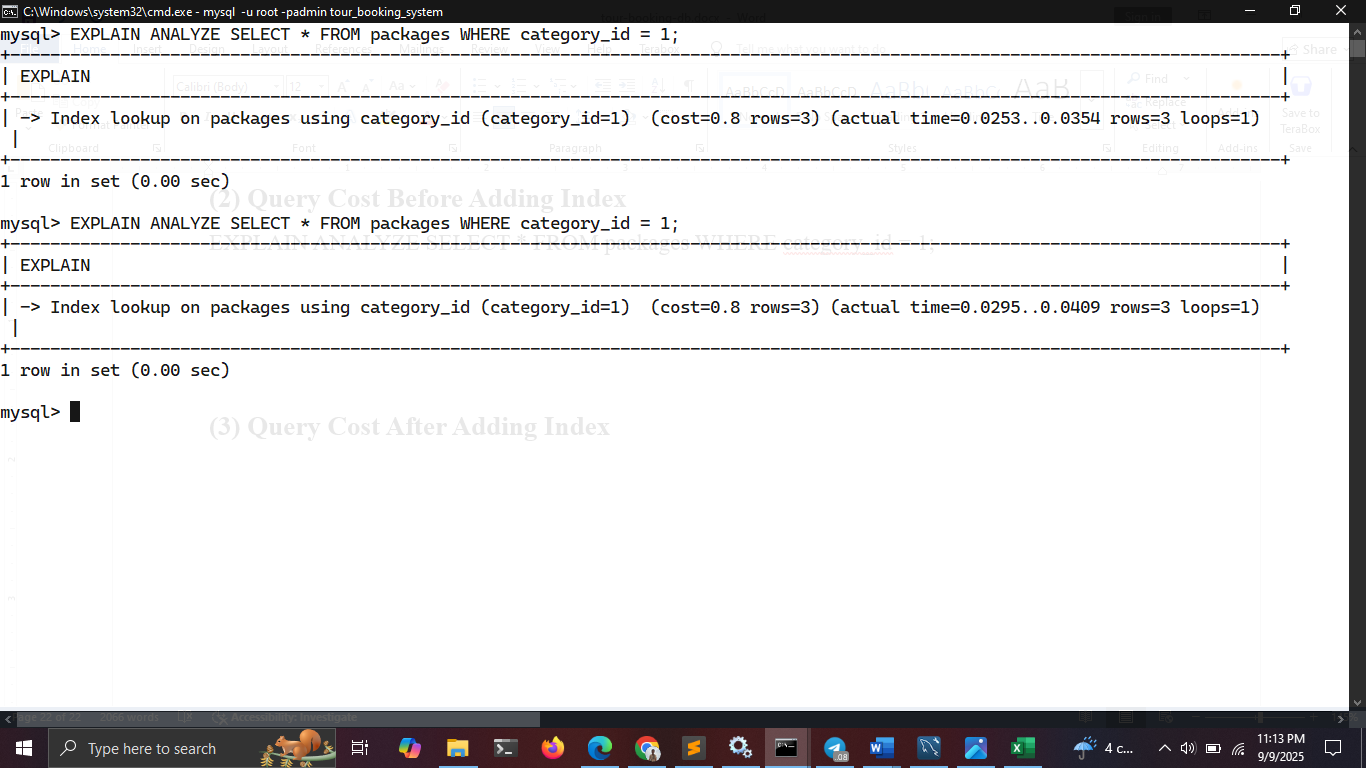
**(1) Create Index**

CREATE INDEX idx\_packages\_category ON packages(category\_id);

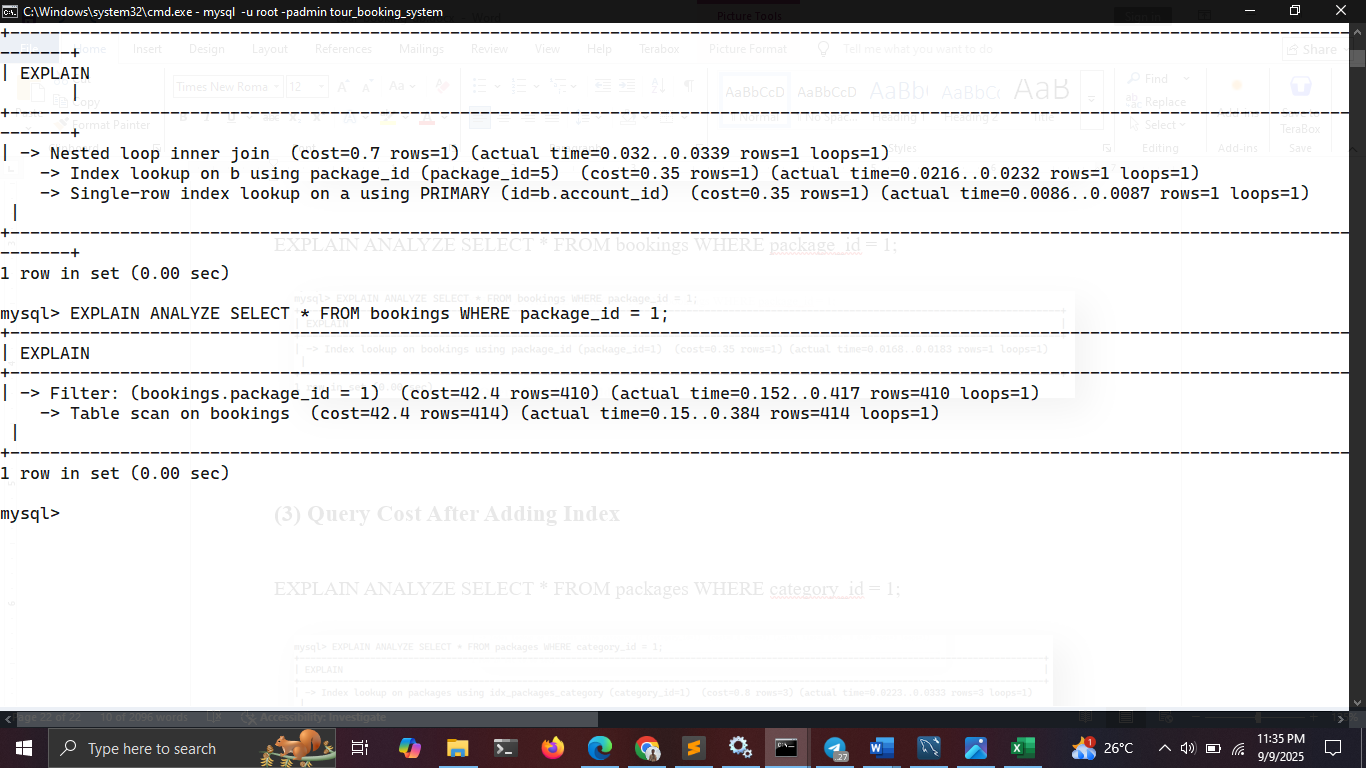
CREATE INDEX idx\_bookings\_package ON bookings(package\_id);

**(2) Query Cost Before Adding Index**

EXPLAIN ANALYZE SELECT \* FROM packages WHERE category\_id = 1;

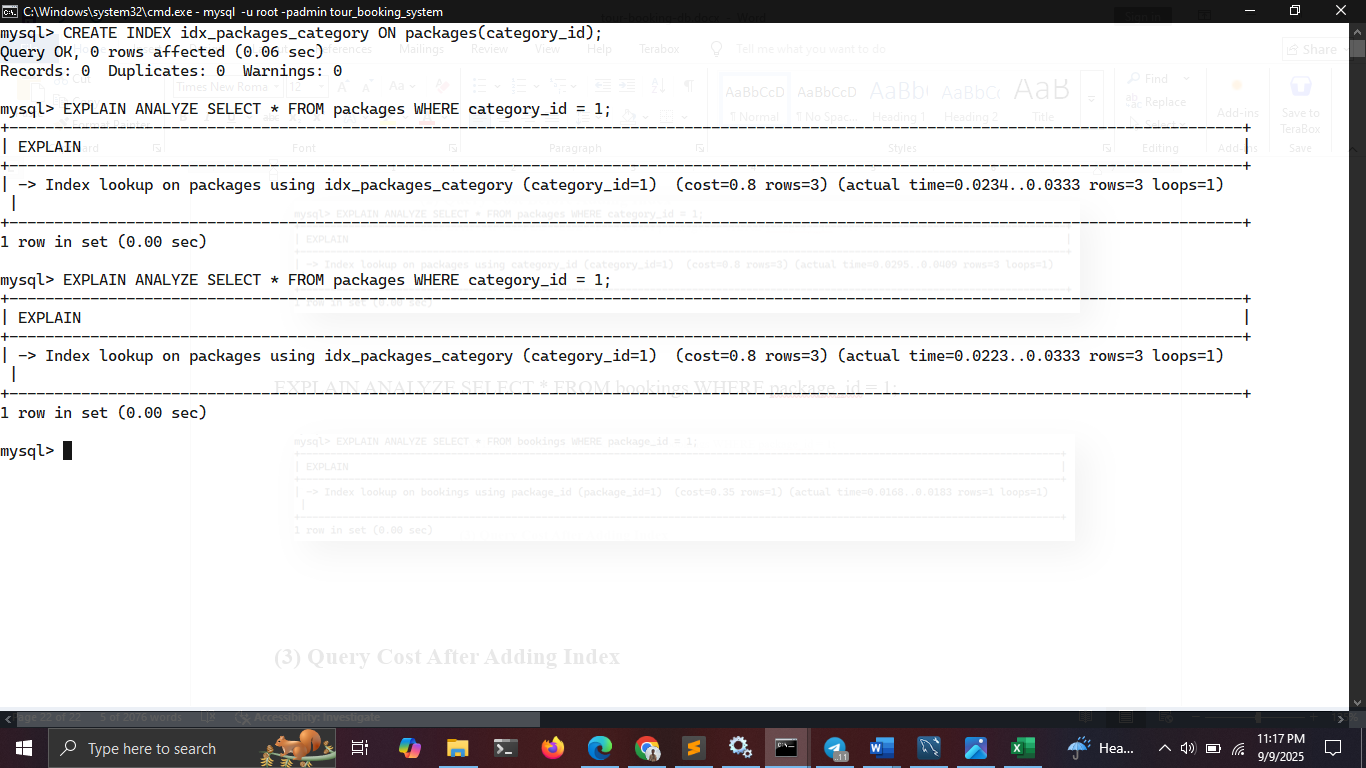


EXPLAIN ANALYZE SELECT \* FROM bookings WHERE package\_id = 1;

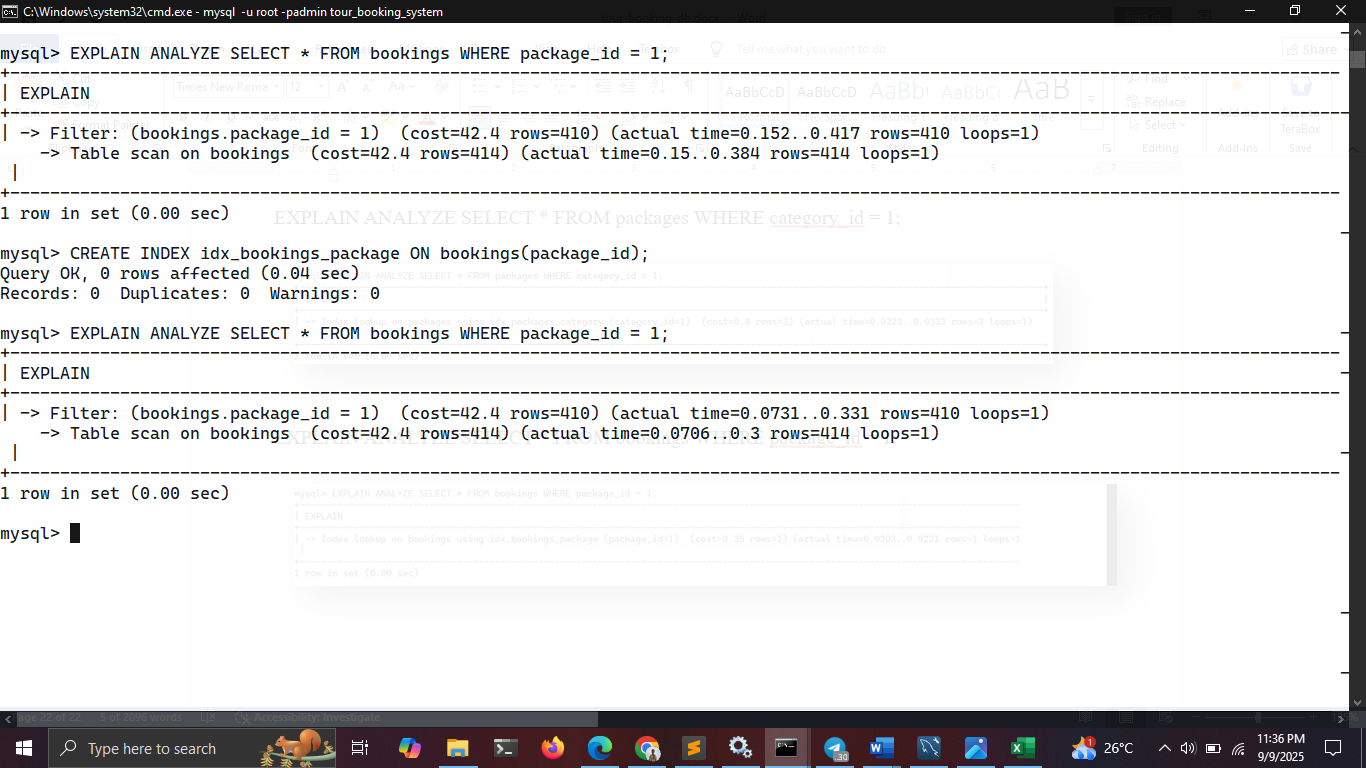


**(3) Query Cost After Adding Index**

EXPLAIN ANALYZE SELECT \* FROM packages WHERE category\_id = 1;



EXPLAIN ANALYZE SELECT \* FROM bookings WHERE package\_id = 1;



**3.3.2 Query Evaluation Plans**

**Queries for Admin**

1. **View All Packages With Booking Counts**

SELECT

p.id AS id,

p.code AS code,

p.title AS title,

p.package\_status AS status,

COUNT(b.id) AS booking\_count,

SUM(b.ticket\_counts) AS tickets\_booked,

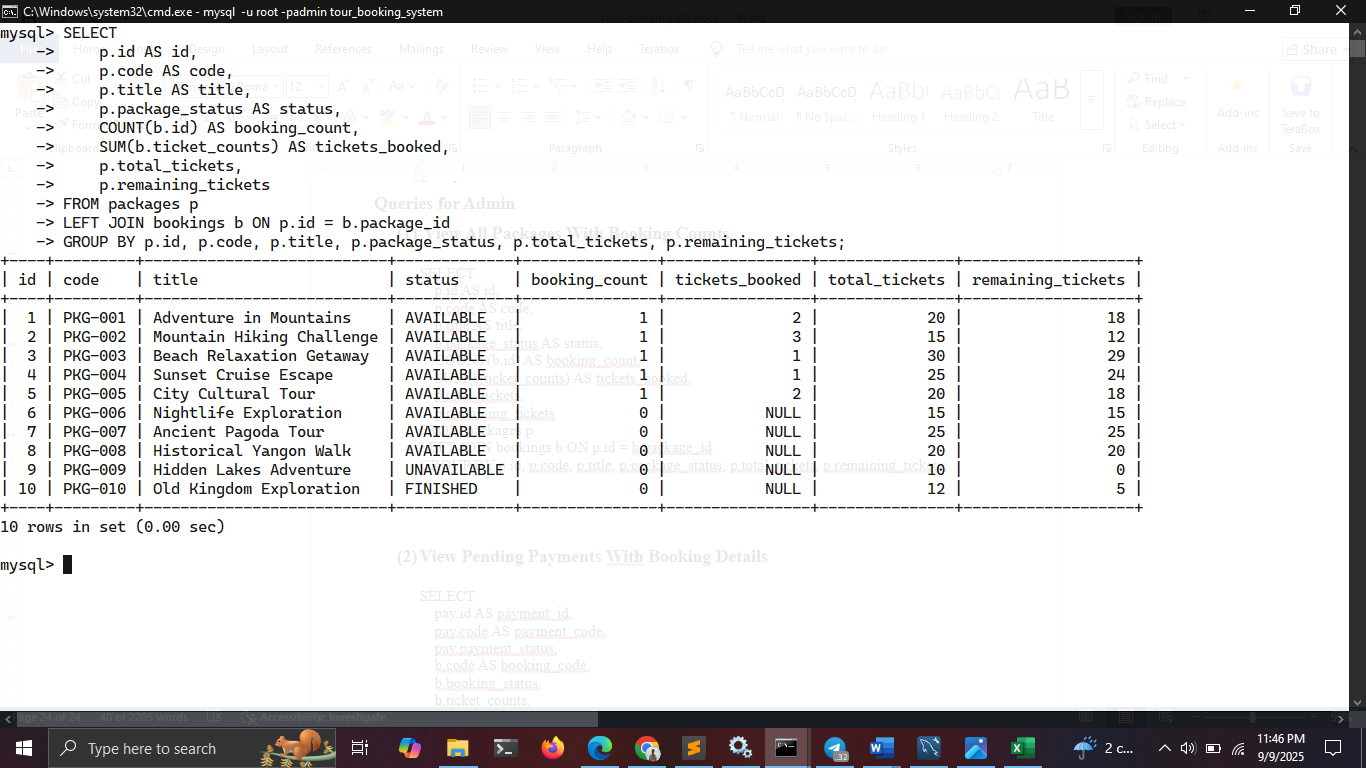
p.total\_tickets,

p.remaining\_tickets

FROM packages p

LEFT JOIN bookings b ON p.id = b.package\_id

GROUP BY p.id, p.code, p.title, p.package\_status, p.total\_tickets, p.remaining\_tickets;



1. **View Pending Payments With Booking Details**

SELECT

pay.code AS payment\_code,

pay.payment\_status,

b.code AS booking\_code,

b.ticket\_counts as tickets,

b.unit\_price,

a.name AS customer,

p.title AS package

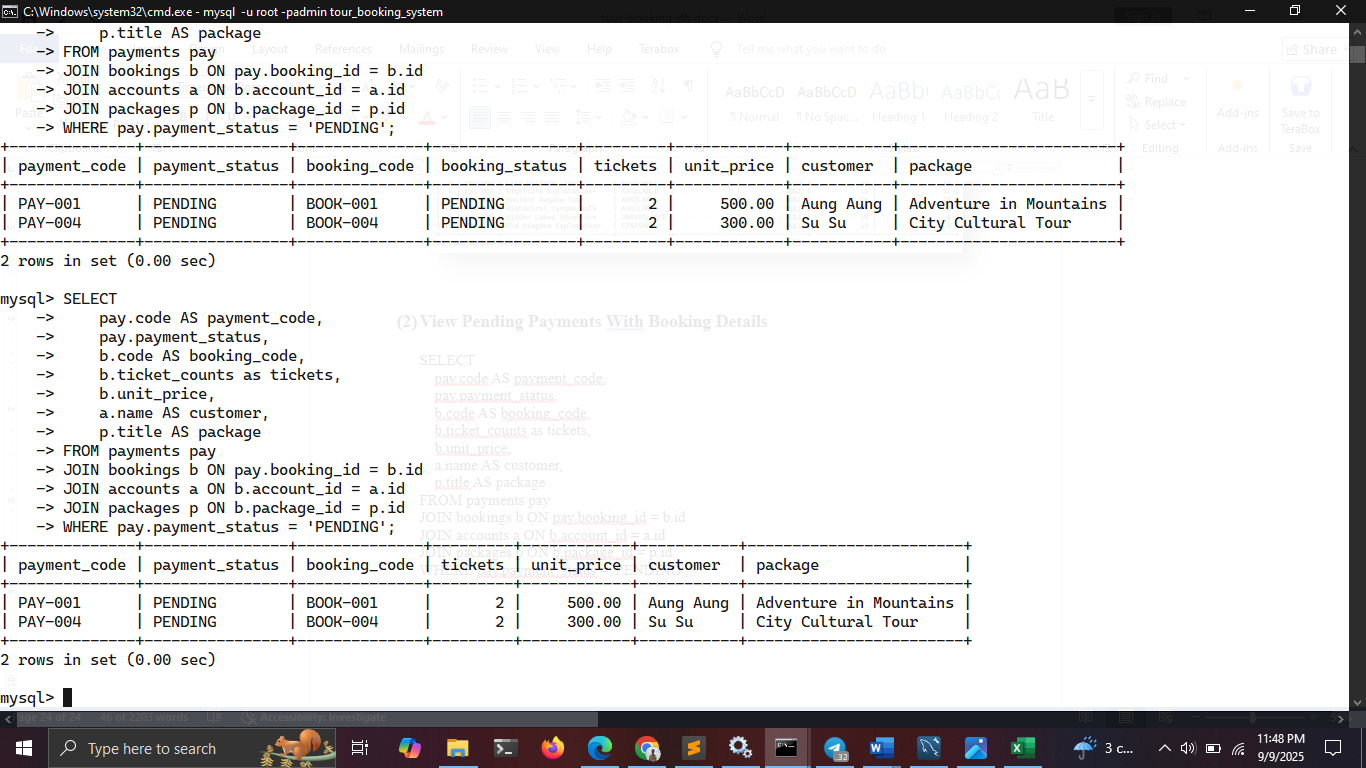
FROM payments pay

JOIN bookings b ON pay.booking\_id = b.id

JOIN accounts a ON b.account\_id = a.id

JOIN packages p ON b.package\_id = p.id

WHERE pay.payment\_status = 'PENDING';



**Queries for Customer**

1. **View All My Bookings with Package Info**

SELECT

b.code AS booking\_code,

p.title AS package\_title,

b.ticket\_counts,

b.unit\_price,

(b.ticket\_counts \* b.unit\_price) AS total\_price,

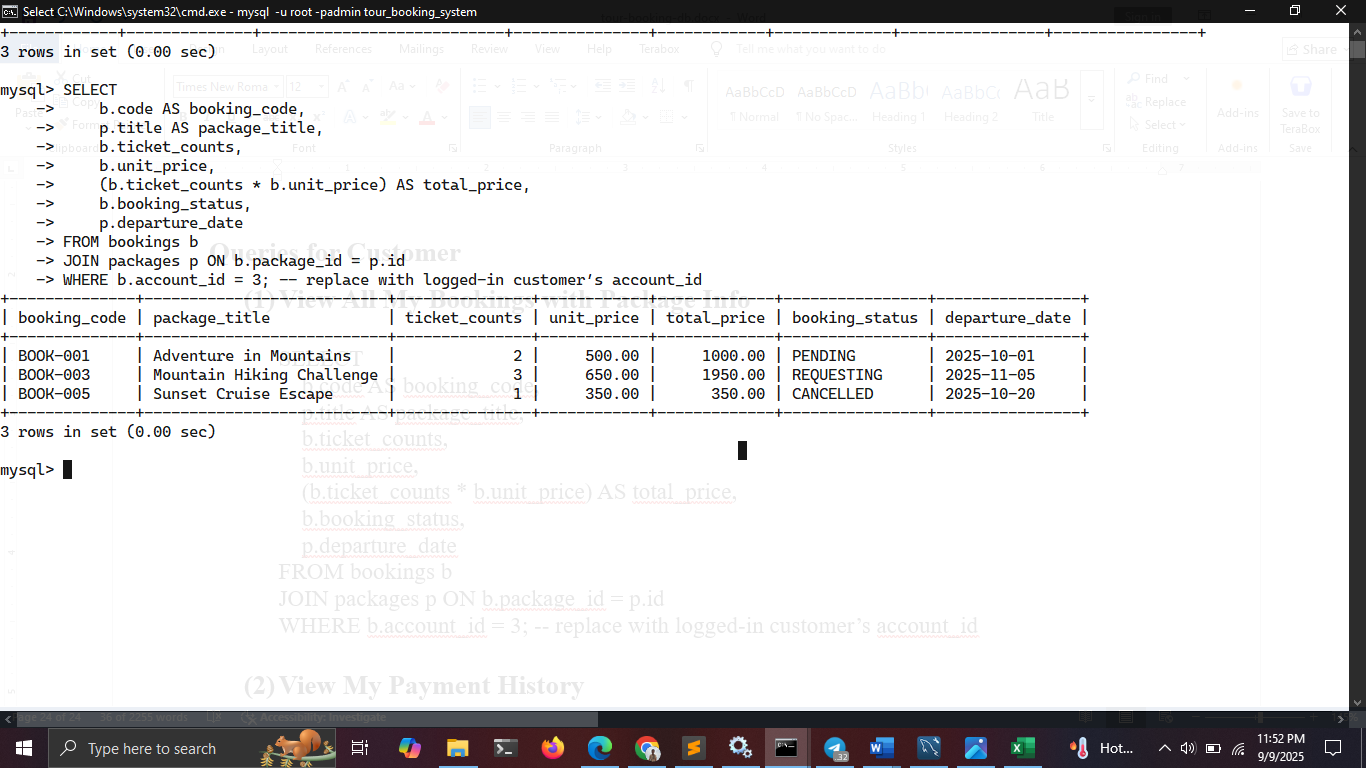
b.booking\_status,

p.departure\_date

FROM bookings b

JOIN packages p ON b.package\_id = p.id

WHERE b.account\_id = 3; -- replace with logged-in customer’s account\_id



1. **View My Payment History**

SELECT

pay.code AS payment\_code,

pay.payment\_status,

b.code AS booking\_code,

b.booking\_status,

pt.name AS payment\_method,

pt.payment\_phone,

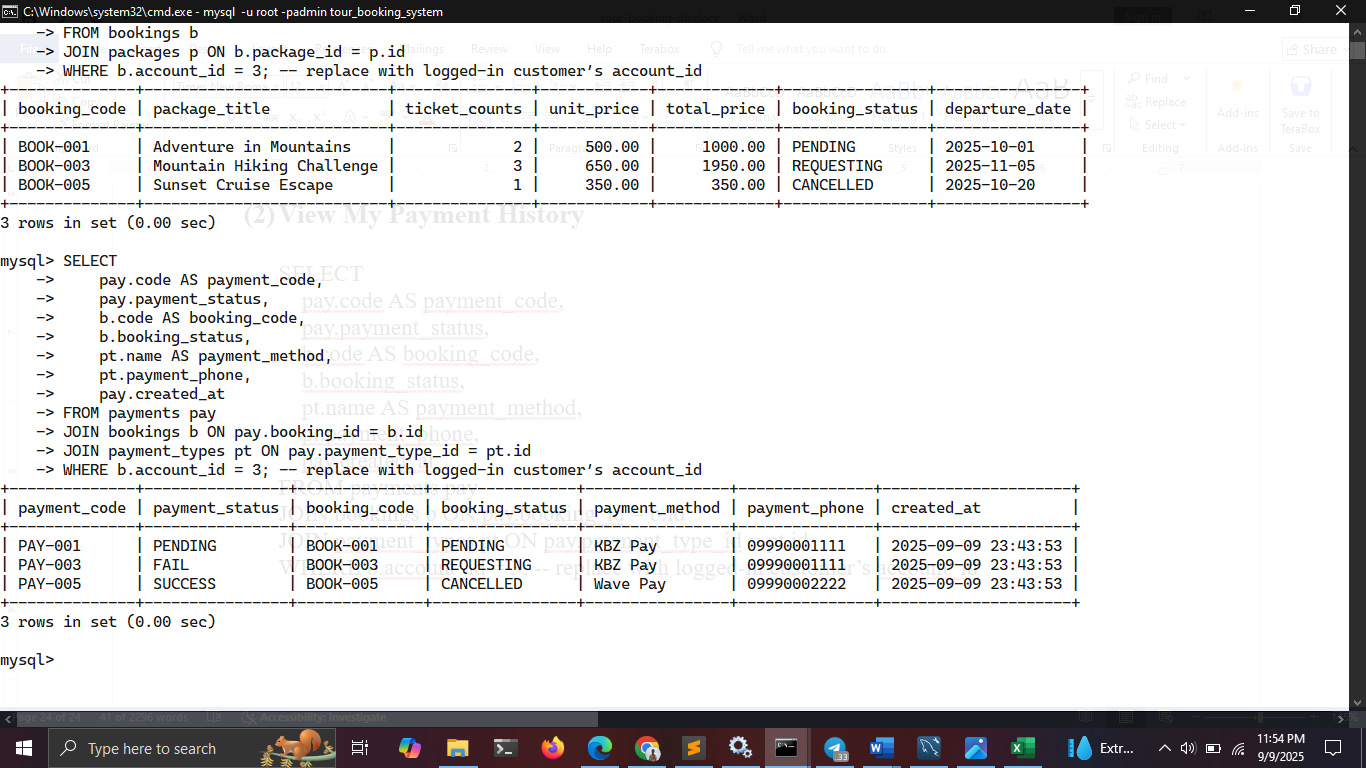
pay.created\_at

FROM payments pay

JOIN bookings b ON pay.booking\_id = b.id

JOIN payment\_types pt ON pay.payment\_type\_id = pt.id

WHERE b.account\_id = 3; -- replace with logged-in customer’s account\_id



**3.3.3 Implementing Transformations Based Optimizations**

**Query Optimizing for Admin**

**View All Reserved Bookings with Package & Customer Info**

**Before Optimizing**

SELECT

b.id AS booking\_id,

b.code AS booking\_code,

a.name AS customer\_name,

p.title AS package\_title,

b.ticket\_counts,

b.unit\_price,

(b.ticket\_counts \* b.unit\_price) AS total\_price,

b.booking\_status

FROM bookings b

JOIN accounts a ON b.account\_id = a.id

JOIN packages p ON b.package\_id = p.id

WHERE b.booking\_status = 'RESERVED'

AND p.category\_id = 2;

**After Optimizing**

SELECT

b.id AS booking\_id,

b.code AS booking\_code,

a.name AS customer\_name,

p.title AS package\_title,

b.ticket\_counts,

b.unit\_price,

(b.ticket\_counts \* b.unit\_price) AS total\_price,

b.booking\_status

FROM (

SELECT id, package\_id, account\_id, code, ticket\_counts, unit\_price, booking\_status

FROM bookings

WHERE booking\_status = 'RESERVED'

) b

JOIN accounts a ON b.account\_id = a.id

JOIN (

SELECT id, title, category\_id

FROM packages

WHERE category\_id = 2

) p ON b.package\_id = p.id;

**Relational Algebra Representation**

**Before Optimizing**

π b.id, b.code, a.name, p.title, b.ticket\_counts, b.unit\_price, total\_price, b.booking\_status

( σ b.booking\_status='RESERVED' ∧ p.category\_id=2

( (bookings ⋈ b.account\_id=a.id accounts) ⋈ b.package\_id=p.id packages )

)

**After Optimizing**

π b.id, b.code, a.name, p.title, b.ticket\_counts, b.unit\_price, total\_price, b.booking\_status

( (σ b.booking\_status='RESERVED' (bookings)

⋈ b.account\_id=a.id accounts)

⋈ b.package\_id=p.id (σ p.category\_id=2 (packages)) )

**Query Optimizing for Customer**

**View Own Bookings with Package Titles**

**Before Optimizing**

SELECT b.id, b.code, b.booking\_status,

(SELECT p.title FROM packages p WHERE p.id = b.package\_id) AS package\_title

FROM bookings b

WHERE b.account\_id = 3;

**After Optimizing**

SELECT b.id, b.code, b.booking\_status, p.title AS package\_title

FROM bookings b

JOIN packages p ON b.package\_id = p.id

WHERE b.account\_id = 3;

**Relational Algebra Representation**

**Before Optimizing**

π b.id, b.code, b.booking\_status, (π title (σ p.id = b.package\_id (packages)))

(σ b.account\_id = 3 (bookings b))

**After Optimizing**

π b.id, b.code, b.booking\_status, p.title

( σ b.account\_id = 3 (bookings b)

⋈ b.package\_id = p.id

(packages p) )

**Chatper 4**

**Conclusion**

**4.1 Conclusion**

The Tour Booking System Database efficiently manages tours, packages, bookings, payments, and user accounts while ensuring data integrity through constraints, foreign keys, and enumerated types.

With functions, procedures, and triggers, key business rules are automated, such as restricting bookings to customers, limiting package creation to admins, and updating booking/package statuses on cancellations or payments.

Query optimizations using joins, predicate pushdown, and indexing improve performance, while relational algebra illustrates the theoretical foundation for query evaluation.

Overall, the system provides a robust, scalable, and maintainable database ready to support a full-featured tour booking application.

**4.2 References**

* <https://www.geeksforgeeks.org/dbms/how-to-design-a-database-for-booking-and-reservation-systems/?utm_source=chatgpt.com>
* <https://www.geeksforgeeks.org/how-to-design-er-diagrams-for-travel-and-tourism-booking-systems/?utm_source=chatgpt.com>
* <https://www.slideshare.net/slideshow/database-system-for-online-travel-booking-system/85379695?utm_source=chatgpt.com>