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

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.

1.2 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

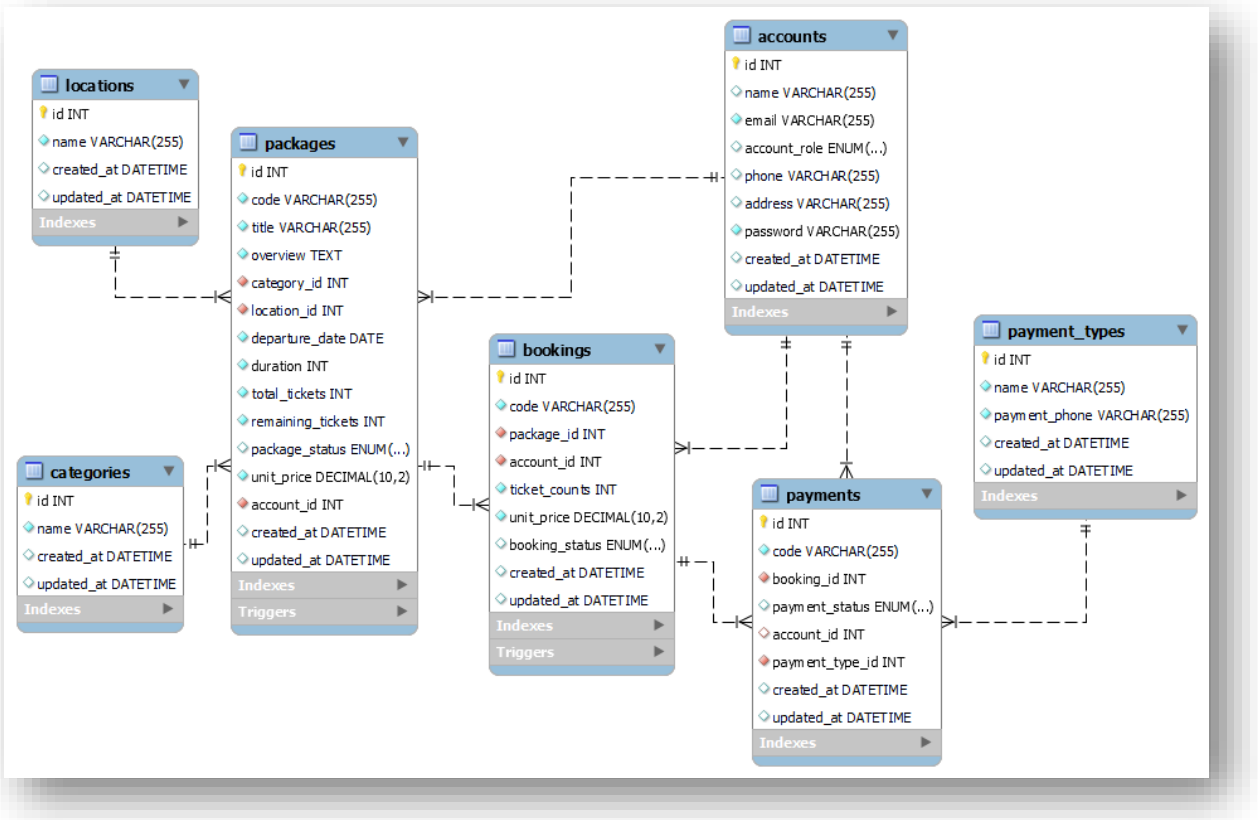


Figure 2.1 ER Diagram

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;

```
mysql> EXPLAIN ANALYZE SELECT * FROM packages WHERE category_id = 1;
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                     |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Index lookup on packages using category_id (category_id=1)  (cost=0.8 rows=3) (actual time=0.0295..0.0409 rows=3 loops=1)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

EXPLAIN ANALYZE SELECT * FROM bookings WHERE package_id = 1;

```
mysql> EXPLAIN ANALYZE SELECT * FROM bookings WHERE package_id = 1;
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                     |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Filter: (bookings.package_id = 1)  (cost=42.4 rows=410) (actual time=0.152..0.417 rows=410 loops=1)
    -> Table scan on bookings  (cost=42.4 rows=414) (actual time=0.15..0.384 rows=414 loops=1)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

(3) Query Cost After Adding Index

EXPLAIN ANALYZE SELECT * FROM packages WHERE category_id = 1;

```
mysql> EXPLAIN ANALYZE SELECT * FROM packages WHERE category_id = 1;
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                     |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Index lookup on packages using idx_packages_category (category_id=1)  (cost=0.8 rows=3) (actual time=0.0223..0.0333 rows=3 loops=1)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

EXPLAIN ANALYZE SELECT * FROM bookings WHERE package_id = 1;

```
mysql> EXPLAIN ANALYZE SELECT * FROM bookings WHERE package_id = 1;
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                     |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Filter: (bookings.package_id = 1)  (cost=42.4 rows=410) (actual time=0.0731..0.331 rows=410 loops=1)
    -> Table scan on bookings  (cost=42.4 rows=414) (actual time=0.0706..0.3 rows=414 loops=1)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

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;
```

id	code	title	status	booking_count	tickets_booked	total_tickets	remaining_tickets
1	PKG-001	Adventure in Mountains	AVAILABLE	1	2	20	18
2	PKG-002	Mountain Hiking Challenge	AVAILABLE	1	3	15	12
3	PKG-003	Beach Relaxation Getaway	AVAILABLE	1	1	30	29
4	PKG-004	Sunset Cruise Escape	AVAILABLE	1	1	25	24
5	PKG-005	City Cultural Tour	AVAILABLE	1	2	20	18
6	PKG-006	Nightlife Exploration	AVAILABLE	0	NULL	15	15
7	PKG-007	Ancient Pagoda Tour	AVAILABLE	0	NULL	25	25
8	PKG-008	Historical Yangon Walk	AVAILABLE	0	NULL	20	20
9	PKG-009	Hidden Lakes Adventure	UNAVAILABLE	0	NULL	10	0
10	PKG-010	Old Kingdom Exploration	FINISHED	0	NULL	12	5

10 rows in set (0.00 sec)

(2) 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';
```

payment_code	payment_status	booking_code	tickets	unit_price	customer	package
PAY-001	PENDING	BOOK-001	2	500.00	Aung Aung	Adventure in Mountains
PAY-004	PENDING	BOOK-004	2	300.00	Su Su	City Cultural Tour

2 rows in set (0.00 sec)

Queries for Customer

(1) View All My Bookings with Package Info

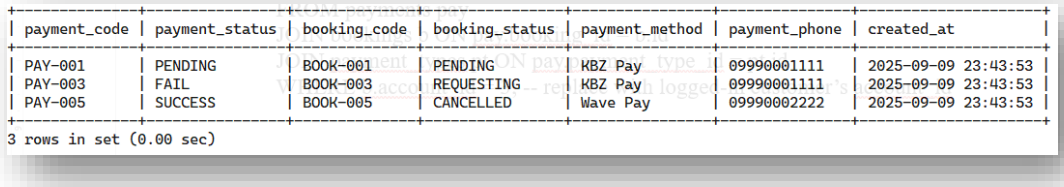
```
SELECT
   b.code AS booking_code,
   p.title AS package_title,
   b.ticket_counts,
   b.unit_price,
   (b.ticket_counts * b.unit_price) AS total_price,
   b.booking_status,
   p.departure_date
FROM bookings b
JOIN packages p ON b.package_id = p.id
WHERE b.account_id = 3; -- replace with logged-in customer's account_id
```

booking_code	package_title	ticket_counts	unit_price	total_price	booking_status	departure_date
BOOK-001	Adventure in Mountains	2	500.00	1000.00	PENDING	2025-10-01
BOOK-003	Mountain Hiking Challenge	3	650.00	1950.00	REQUESTING	2025-11-05
BOOK-005	Sunset Cruise Escape	1	350.00	350.00	CANCELLED	2025-10-20

3 rows in set (0.00 sec)

(2) 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
```



payment_code	payment_status	booking_code	booking_status	payment_method	payment_phone	created_at
PAY-001	PENDING	BOOK-001	PENDING	KBZ Pay	09990001111	2025-09-09 23:43:53
PAY-003	FAIL	BOOK-003	REQUESTING	KBZ Pay	09990001111	2025-09-09 23:43:53
PAY-005	SUCCESS	BOOK-005	CANCELLED	Wave Pay	09990002222	2025-09-09 23:43:53

3 rows in set (0.00 sec)

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

$$\pi_{\text{b.id, b.code, a.name, p.title, b.ticket\_counts, b.unit\_price, total\_price, b.booking\_status}} \Big(\sigma_{\text{b.booking\_status='RESERVED'} \wedge \text{p.category\_id=2}} \big((\text{bookings} \bowtie_{\text{b.account\_id=a.id}} \text{accounts}) \bowtie_{\text{b.package\_id=p.id}} \text{packages} \big) \Big)$$

After Optimizing

$$\pi_{\text{b.id, b.code, a.name, p.title, b.ticket\_counts, b.unit\_price, total\_price, b.booking\_status}} \Big(\big((\sigma_{\text{b.booking\_status='RESERVED'}} (\text{bookings}) \bowtie_{\text{b.account\_id=a.id}} \text{accounts}) \bowtie_{\text{b.package\_id=p.id}} (\sigma_{\text{p.category\_id=2}} (\text{packages})) \big) \Big)$$

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

$$\pi_{\text{b.id, b.code, b.booking\_status, } (\pi_{\text{title}} (\sigma_{\text{p.id = b.package\_id}} (\text{packages})))} \; (\sigma_{\text{b.account\_id = 3}} (\text{bookings b}))$$

After Optimizing

$$\pi_{\text{b.id, b.code, b.booking\_status, p.title}} \; (\sigma_{\text{b.account\_id = 3}} (\text{bookings b}) \; \bowtie_{\text{b.package\_id = p.id}} \; (\text{packages p}))$$

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