HOTEL MANAGEMENT SYSTEM

A DBMS PROJECT REPORT [DBMS REPORT]

Submitted by

Kabinesh Sakthivel [RA2311003011613]
Janani Somasundaram [RA2311003011641]
Supriya Namasivayam [RA2311003011661]

Under the Guidance of

Dr.RAMAPRABHA
ASSISTANT PROFESSOR
DEPARTMENT OF
COMPUTING TECHNOLOGIES

in partial fulfillment of the requirementsfor the degree

of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING
with specialization in (SPECIALIZATION NAME)

DEPARTMENT OF COMPUTING
TECHNOLOGIES, COLLEGE OF ENGINEERING
AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND
TECHNOLOGY
KATTANKULATHUR- 603 203

MAY 2025

Department of Computational Intelligence

SRM Institute of Science & Technology

Own Work* Declaration Form

This sheet must be filled in (each box ticked to show that the condition has been met). It must be signed and dated along with your student registration number and included with all assignments you submit – work will not be marked unless this is done.

To be completed by the student for all assessments

Degree/ Course :

Student Name :

Registration Number :

Title of Work : Hotel management system.

I / We hereby certify that this assessment compiles with the University's Rules and Regulations relating to Academic misconduct and plagiarism**, as listed in the University Website, Regulations, and the Education Committee guidelines.

I / We confirm that all the work contained in this assessment is my / our own except where indicated, and that I / We have met the following conditions:

- Clearly referenced / listed all sources as appropriate
- Referenced and put in inverted commas all quoted text (from books, web, etc)
- Given the sources of all pictures, data etc. that are not my own
- Not made any use of the report(s) or essay(s) of any other student(s) either past or present
- Acknowledged in appropriate places any help that I have received from others (e.g. fellow students, technicians, statisticians, external sources)
- Compiled with any other plagiarism criteria specified in the Course handbook / University website

I understand that any false claim for this work will be penalized in accordance with the University policies and regulations.

DECLARATION:
I am aware of and understand the University's policy on Academic misconduct and plagiarism and I certify that this assessment is my / our own work, except where indicated by referring, and that I have followed the good academic practices noted above.
If you are working in a group, please write your registration numbers and sign with the date for every student in your group.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that 21CSC205P – Database Management System Project report titled "**HOTEL MANAGEMENT SYSTEM**" is the bonafide work of "**Kabinesh Sakthivel [RA2311 003011613], Janani Somasundaram [RA2311003011641], Supriya Namasivayam [RA2311003011661]"**who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr.RAMAPRABHA J

ASSISTANT PROFESSOR
DEPARTMENT OF
COMPUTING TECHNOLOGIES

SIGNATURE

Dr. G. NIRANJANA

PROFESSOR &HEAD
DEPARTMENT OF
COMPUTING TECHNOLOGIES

ACKNOWLEDGEMENTS

We express our humble gratitude to **Dr. C. Muthamizhchelvan**, Vice-Chancellor, SRM Institute of Science and Technology, for the facilities extended for the project work and his continued support.

We extend our sincere thanks to **Dr. Leenus Jesu Martin M,** Dean-CET, SRM Institute of Science and Technology, for his invaluable support.

We wish to thank **Dr. Revathi Venkataraman**, Professor and Chairperson, School of Computing, SRM Institute of Science and Technology, for her support throughout the project work.

We encompass our sincere thanks to, **Dr. M. Pushpalatha**, Professor and Associate Chairperson, School of Computing and **Dr. C. Lakshmi,** Professor and Associate Chairperson, School of Computing, SRM Institute of Science and Technology, for their invaluable support.

We are incredibly grateful to our Head of the Department, **Dr. G. Niranjana**, Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for her suggestions and encouragement at all the stages of the project work.

We want to convey our thanks to our DBMS Coordinators, **Dr. Muthu Kumaran A**

M J and **Dr. Jayapradha J** Department of Computing Technologies, and our Audit Professor **Dr. Malathy C,** Department of Networking And Communications, SRM Institute of Science and Technology, for their inputs during the project reviews and support.

We register our immeasurable thanks to our Faculty Advisors, Dr.Mathan Kumar, Department of Computing Technologies, SRM Institute of Science and Technology, for leading and helping us to complete our course.

Our inexpressible respect and thanks to our faculty, **Dr Ramaprabha J,** Department of Computing Technologies, S.R.M Institute of Science and Technology, for providing us with an opportunity to pursue our project under her mentorship. She provided us with the freedom and support to explore the research topics of our interest. Her passion for solving problems and making a difference in the world has always been inspiring.

We sincerely thank all the staff and students of Computing Technologies, School of Computing, S.R.M Institute of Science and Technology, for their help during our project. Finally, we would like to thank our parents, family members, and friends for their unconditional love, constant support and encouragement

Kabinesh Sakthivel [RA2311003011613]
Janani Somasundaram [RA2311003011641]
Supriya Namasivayam [RA2311003011661]

ABSTRACT

The Hotel Management System project is a comprehensive database-driven application designed to automate and optimize core operations within a hotel environment. This includes functionalities such as room booking, guest check-in/check-out, billing, and administrative controls. The objective is to reduce manual errors, enhance data accessibility, and improve customer satisfaction through efficient digital processes.At the heart of the system is a structured relational database, incorporating advanced features such as constraints, sets, joins, views, triggers, and cursors. These components ensure data consistency, relational integrity, and facilitate complex queries and operations. Triggers automate actions based on specific conditions, while cursors help in handling row-by-row processing for large datasets.A critical aspect of the system is its Concurrency Control mechanism, which guarantees safe and consistent database updates when multiple users access or modify data concurrently. This is achieved using transaction management commands such as START TRANSACTION, COMMIT, and ROLLBACK. Additionally, row-level locking (SELECT FOR UPDATE) prevents data conflicts,procedures (e.g., ConfirmBookingAndPaymentSafe) ensure atomic and secure operations during booking and payment processing. The use of error handling with EXIT HANDLER FOR SQLEXCEPTION allows for automatic rollback in case of failures.The system also incorporates a robust Recovery Mechanism to safeguard data against system failures, crashes, or accidental deletions. Automated daily backups are scheduled using Linux crontab jobs, which execute the mysqldumpcommand at 2:00 AM each day. This ensures that the most recent data can be restored promptly, minimizing downtime and data loss.Overall, this project demonstrates an efficient, reliable, and secure hotel management solution by integrating key database technologies and best practices in data handling, concurrency, and recovery.

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.

CHAPTER 1: INTRODUCTION

1.1 General

This project is a comprehensive database management system developed to streamline and optimize the various operational processes involved in managing hotel activities. It provides an integrated solution for handling essential functions such as room bookings, customer information, staff records, billing operations, and service management. The system is built using MySQL, a robust and widely adopted relational database management system known for its reliability, scalability, and performance. By following structured database design principles such as normalization, data integrity, and indexing, the project ensures that data is stored logically, retrieved efficiently, and maintained accurately. For room bookings, the system supports real-time availability tracking, automated reservation handling, check-in and check-out management, and allocation of rooms based on customer preferences. The customer module stores detailed profiles, including personal identification, contact details, booking history, and special requests, which facilitates personalized service and efficient customer relationship management. Employee records are maintained within the staff database, which includes information on roles, schedules, attendance, and salary, thereby assisting in workforce planning and payroll operations. The billing system automatically generates invoices based on room charges and additional services such as dining, laundry, or spa usage, while supporting various payment methods and secure transaction logging. Service records are also maintained, ensuring that tasks related to housekeeping, maintenance, and other customer services are properly scheduled, monitored, and executed in a timely manner. The overall aim of the project is to enhance operational efficiency, reduce manual errors, and provide easy access to accurate data for informed decision-making. Its user-friendly interface and modular structure make it adaptable for different hotel sizes and suitable for future expansions like integration with online booking platforms, mobile applications, and advanced data analytics tools.

1.2 Motivation

Hotels today handle an enormous volume of data related to guests, reservations, services, staff, billing, and daily operations. Managing all this information manually not only becomes time-consuming and labour-intensive but also significantly increases the chances of human error, data redundancy, and inconsistencies. These issues can lead to delays in service, poor customer experience, and inefficiencies in resource management. In such a dynamic environment, a well-structured Database Management System (DBMS) proves to be an essential tool. It replaces error-prone manual processes with automated workflows that ensure accurate, consistent, and up-to-date information across all departments. By organizing data in a structured and relational manner, the DBMS enhances data retrieval speed, improves coordination between various functional areas, and provides a solid foundation for data-driven decision-making. Additionally, it brings reliability by securing sensitive information and enabling controlled access based on user roles. Scalability is another key advantage, as the system can easily accommodate the growing data needs of expanding hotel operations, whether in a single property or across a chain of hotels. In essence, a robust DBMS transforms the hospitality domain by driving operational efficiency, enhancing customer

satisfaction, and laying the groundwork for future technological integrations such as real-time analytics, mobile interfaces, and cloud-based services.

1.3 Objective

Centralized Database for Automation

This project is designed to automate various hotel operations through the implementation of a centralized database. All crucial data, including room bookings, customer details, staff information, billing, and services, are stored and managed in one integrated system. This centralization reduces redundancy, eliminates the need for manual data entry across different departments, and enhances overall coordination within the hotel. Automation improves operational speed and minimizes human error, ensuring a more seamless workflow.

Efficient Room Bookings, Payments, and Employee Management

The system allows for real-time and efficient management of core hotel functions. Room bookings are handled with precision through automated availability tracking, reservation status updates, and customer preference logging. The billing process is streamlined with features like auto-generated invoices, tax calculations, and multi-mode payment support, ensuring transparency and accuracy in financial transactions. For employee management, the system stores and processes data related to staff roles, schedules, attendance, and payroll, improving human resource management and workforce planning.

Use of Normalization, ER Modeling, and Transaction Management

To ensure data accuracy, consistency, and efficiency, the system employs fundamental database design techniques. Normalization is used to eliminate data redundancy and maintain organized, structured data storage. Entity-Relationship (ER) modeling defines the relationships between key entities such as customers, rooms, bookings, services, and staff, creating a clear and logical schema. Additionally, transaction management is implemented to maintain database integrity. This ensures that multiple operations either complete fully or not at all, protecting the system from partial updates or data corruption in case of system failures or interruptions.

1.4 Scope

1. Room Reservation and Availability

The system efficiently manages room reservations by keeping track of all available, reserved, and occupied rooms in real-time. It allows front-desk staff to check room status instantly and allocate rooms based on customer preferences such as room type, view, or amenities. The system also supports future bookings and helps prevent overbooking or reservation conflicts by automatically updating availability.

2. Customer Check-In/Check-Out

Customer check-in and check-out processes are streamlined through automated data entry and validation. At check-in, the system records guest details, assigns rooms, and logs the duration of stay. During check-out, it calculates the final bill based on services used, generates an

invoice, and updates room status to "available." This minimizes manual errors and speeds up the overall guest-handling process.

3. Employee Management

The system maintains a centralized database of all hotel employees, including their personal details, job roles, work schedules, and attendance. It helps management track staff performance, manage shift rotations, and process payroll. With controlled access, only authorized personnel can make changes to employee records, ensuring both efficiency and security.

4. Billing and Reporting

Billing is automated to include charges for room rent, food services, additional facilities, taxes, and discounts. The system generates detailed invoices and stores transaction histories for future reference. Reporting tools provide insights into financial performance, room occupancy trends, and customer preferences. These reports assist management in strategic decision-making and resource planning.

5. Basic Administrative Controls

The system includes basic administrative features such as user account management, role-based access control, and data backup functions. Administrators can define user roles (e.g., receptionist, manager, accountant) to restrict or allow access to specific modules. These controls ensure data security, prevent unauthorized access, and maintain overall system integrity.

1.5 Tools and Technologies

1. Database: MySQL

The backend of the system is powered by MySQL, a widely used open-source relational database management system known for its robustness, performance, and scalability. MySQL is ideal for handling structured hotel data such as customer records, room availability, employee information, and billing details. It supports SQL queries for data manipulation and ensures data integrity through constraints, indexing, and relational design.

2. Design Tool: dbdiagram.io / Draw.io

The database schema and relationships are visually designed using tools like dbdiagram.io or Draw.io. These tools help in planning the structure of the database through Entity-Relationship (ER) diagrams, which depict the connections between different tables such as customers, rooms, bookings, and staff. This visual representation aids in understanding data flow and refining the overall design before implementation.

3. Query Interface: MySQL Workbench or Command Line Interface (CLI)

To interact with the database, tools such as MySQL Workbench or the MySQL Command Line Interface (CLI) are used. MySQL Workbench offers a graphical interface to write, test, and execute SQL queries, manage databases, and visualize schemas. The CLI provides a more

lightweight, text-based alternative for executing queries, managing user permissions, and performing backups or data migrations.

4. Optional Frontend: HTML/PHP (for demo only)

For demonstration purposes, an optional frontend interface can be developed using HTML and PHP. This basic user interface allows users to interact with the database through web forms and displays, enabling functionalities like booking rooms, checking availability, generating bills, and managing employee data. While not mandatory, this frontend can help simulate real-world usage and enhance the project presentation.

1.6 Methodology

1. ER Modeling

The first phase of the system development involves Entity-Relationship (ER) modeling, where the overall structure of the database is visually designed. This includes identifying key entities such as Customers, Rooms, Bookings, Employees, and Services, and defining the relationships among them. The ER diagram helps in visualizing how different data components interact, forming the foundation for an efficient and logically structured database.

2. Schema Creation

Based on the ER model, the next step is to create the actual database schema using SQL. This involves defining tables, their attributes, data types, and constraints such as primary keys, foreign keys, and unique values. The schema acts as a blueprint for how data is stored, linked, and accessed within the database system, ensuring logical organization and structural integrity.

3. SQL Query Writing

Once the schema is in place, SQL queries are written to perform essential operations such as data insertion, updates, deletions, and retrieval. These queries enable the system to manage hotel activities like room bookings, customer check-ins and check-outs, billing, and staff assignments. Complex queries may also be written to generate reports or handle conditional data flows across the system.

4. Data Normalization

To avoid data redundancy and improve data consistency, the system undergoes normalization. This process involves organizing the data across multiple related tables and removing partial, transitive, and duplicate dependencies. By applying various normal forms (typically up to 3NF), the database becomes more efficient, easier to maintain, and less prone to anomalies during updates or deletions.

5. Transaction Management and Recovery

The final phase focuses on ensuring the reliability and integrity of the system through transaction management and recovery mechanisms. Transactions, which are sequences of operations performed as a single unit, are designed to follow ACID properties (Atomicity, Consistency, Isolation, Durability). In case of failures such as power loss or system crashes, recovery protocols are implemented to restore the database to a consistent state, ensuring no data is lost or corrupted.

1.7 Project Organization

1B: ER Modeling

In this phase, a comprehensive Entity-Relationship (ER) model is created to represent the logical structure of the database. It identifies the main entities such as Rooms, Customers, Employees, Bookings, and Services, and maps out the relationships between them, such as one-to-many or many-to-many associations. This ER diagram forms the blueprint of the database and guides the subsequent steps in schema design and development. Proper ER modeling ensures clarity in how data will be connected and stored, which is crucial for maintaining consistency and scalability.

2B: Relational Schema & Table Creation

Following the ER modeling, the next step involves converting the diagram into a relational schema. Tables are defined with appropriate columns, data types, and constraints such as primary keys and foreign keys. Relationships are implemented using joins and referential integrity rules to maintain data consistency across related tables. This step brings the theoretical design into a working structure within the MySQL database, allowing data to be stored in an organized and relational format.

3B: Complex Queries (Joins, Views, Triggers, Cursors)

To interact with the data effectively, complex SQL queries are developed. This includes using joins to retrieve related data from multiple tables, views to simplify complex queries or enhance security, triggers to automate certain actions in response to database events (e.g., updating room status after check-out), and cursors to process query results row-by-row when needed. These advanced features enhance the functionality of the system and support dynamic, real-time hotel operations.

4B: Normalization

The database is then normalized to reduce data redundancy and enhance integrity. Normalization is performed through several stages or normal forms (usually up to Third Normal Form – 3NF). This process involves restructuring tables to ensure that each piece of data is stored only once and that dependencies are logical and efficient. Normalization makes the database easier to maintain and improves query performance by organizing data more effectively.

5B: Concurrency Control and Recovery

The final phase addresses the reliability of the system by implementing concurrency control and recovery mechanisms. Concurrency control ensures that multiple users can access and manipulate data simultaneously without causing inconsistencies, using techniques such as locking and isolation levels. Recovery procedures are also defined to handle unexpected failures, such as power outages or system crashes, by ensuring that all transactions either complete successfully or are rolled back to maintain database integrity. This phase guarantees the robustness and trustworthiness of the system under real-world usage conditions.

CHAPTER 2: 1B – IDENTIFICATION OF ENTITY RELATIONSHIPS AND ER MODEL

2.1 Major Entities

The **Customer** table stores essential details about individuals who book rooms or use the services of the hotel.

- **ID**: A unique identifier assigned to each customer (Primary Key).
- **Name**: The full name of the customer.
- **Address**: The residential or contact address of the customer.
- **Phone**: The customer's contact number for communication and verification purposes.

The **Room** table contains information about each room available in the hotel.

- **Room_No**: A unique room number used to identify each room (Primary Key).
- **Type**: The category of the room (e.g., Single, Double, Suite).
- **Status**: Indicates the current availability status (e.g., Available, Occupied, Maintenance).
- **Price**: The cost per night or per day associated with the room.

The **Booking** table records data related to room reservations made by customers.

- **Booking_ID**: A unique identifier for each booking transaction (Primary Key).
- **Customer_ID**: A foreign key referencing the Customer table to link the booking with a customer.
- **Room_No**: A foreign key referencing the Room table to specify which room is booked.
- **CheckIn**: The date and time when the customer is scheduled to check in.
- **CheckOut**: The expected or actual check-out date and time.

The **Payment** table keeps track of financial transactions related to bookings.

- **Payment_ID**: A unique identifier for each payment entry (Primary Key).
- **Booking_ID**: A foreign key referencing the Booking table to connect payment with a specific booking.
- **Amount**: The total amount paid by the customer for the booking.
- **Date**: The date on which the payment was made.

The **Staff** table manages employee-related data for the hotel.

- **Staff_ID**: A unique identifier for each staff member (Primary Key).
- **Name**: Full name of the staff member.
- **Role**: The job position or duty (e.g., Receptionist, Housekeeping, Manager).
- **Salary**: The monthly or hourly wage/salary paid to the staff member.

2.2 Relationships

One-to-Many:
In this relationship, a single customer can make multiple bookings for different rooms. This means that a customer can reserve several rooms during various stays at the hotel. For example, a customer might book a single room for a business trip, and later book multiple rooms for a family vacation. This relationship is one-to-many because one customer can have many associated bookings, but each booking is linked to only one customer.

Many-to-One:
A single room can be booked by multiple customers over time, but each individual booking is related to one specific room. This relationship is many-to-one because many bookings can be made for one room, but each booking refers to a unique instance of the room (i.e., the same room can be booked by different customers at different times). This ensures that each booking record corresponds to a specific room, even though the same room can be used by different guests at various times.

One-to-Many:
In this case, each booking made by a customer is linked to a single payment transaction. However, a payment record can only be associated with one booking. This is a one-to-many relationship because one booking can generate only one payment record, but each payment record corresponds to only one booking. For example, if a customer books a room and pays for it, that payment is tied to a specific booking, ensuring that payments are accurately recorded for each reservation.

One-to-Many:
In this relationship, a hotel can employ multiple staff members, each having their own roles and responsibilities. The hotel is considered the "one" side of the relationship, and each staff member is linked to the hotel they work for. This is a one-to-many relationship because one hotel can employ many staff members, but each staff member works for one hotel. For example, a hotel may have several employees working in different departments like housekeeping, reception, and management.

2.3 ER Diagram

CHAPTER 3: 2B – RELATIONAL SCHEMA & TABLE CREATION

3.1 Schema Design

Customer

The **Customer** entity stores personal details of the individuals making bookings at the hotel. It contains the following attributes:

- **Customer_ID**: This is the primary key for the customer entity, uniquely identifying each customer.
- **Name**: The full name of the customer.
- **Phone**: The contact number of the customer for communication purposes such as booking confirmations or any other inquiries.
- **Email**: The email address of the customer, used for sending booking confirmations, invoices, and promotional offers.

These attributes help in identifying the customer and facilitating communication regarding their bookings, preferences, and hotel-related services.

Room

The **Room** entity defines the hotel's available rooms. It contains the following attributes:

- **Room_No**: This is the primary key for the room entity, uniquely identifying each room in the hotel.
- **Type**: Specifies the category of the room, such as Single, Double, Suite, or Deluxe, indicating its size, amenities, and features.
- **Price**: Represents the cost of booking the room per night. This may vary depending on the room type, view, or season.
- **Status**: Indicates whether the room is available, booked, under maintenance, or any other relevant status. This helps in tracking room availability in real-time.

These attributes allow the hotel to manage its rooms and pricing, and monitor the availability status, which is crucial for the booking process.

Booking

The **Booking** entity captures information about customer reservations. It contains the following attributes:

- **Booking_ID**: The primary key for the booking entity, uniquely identifying each reservation.
- **Customer_ID**: A foreign key that links the booking to a specific customer, referencing the **Customer** entity.

- **Room_No**: A foreign key that links the booking to a specific room, referencing the **Room** entity.
- **CheckIn**: The date on which the customer is scheduled to check into the room.
- **CheckOut**: The date on which the customer is scheduled to check out of the room.

These attributes allow the hotel to track the customer's stay and the rooms they have reserved. The relationship between customer, room, and booking helps in managing the hotel's occupancy and availability.

Payment

The **Payment** entity records the payment details for each booking. It contains the following attributes:

- **Payment_ID**: The primary key for the payment entity, uniquely identifying each payment record.
- **Booking_ID**: A foreign key that links the payment to a specific booking, referencing the **Booking** entity.
- **Amount**: The total amount paid by the customer for the booking.
- **Payment_Date**: The date on which the payment was made, helping in tracking the payment transaction.

These attributes ensure that all payments made by customers are properly linked to their bookings and stored for record-keeping, billing, and reporting purposes.

Staff

The **Staff** entity stores details about the hotel employees. It contains the following attributes:

- **Staff_ID**: The primary key for the staff entity, uniquely identifying each staff member.
- **Name**: The full name of the staff member.
- **Role**: The job position or role of the staff member, such as Front Desk, Housekeeping, Manager, etc.
- **Salary**: The compensation paid to the staff member for their work.

These attributes help manage hotel personnel, track employee roles and responsibilities, and facilitate payroll management.

3.2 SQL Table Creation Examples

CHAPTER 4: 3B – COMPLEX QUERIES, CONSTRAINTS, JOINS, VIEWS, TRIGGERS, CURSORS

4.1 Constraints

PrimaryKey
A Primary Key is a field (or combination of fields) in a table that uniquely identifies each record in that table. It ensures that each row in the table is distinct and cannot have duplicate values. A primary key constraint automatically creates a unique index on the column(s), which helps improve query performance.

- Example: In the Customer table, the Customer_ID is the primary key. It ensures that each customer is uniquely identified and that no two customers can have the same ID.

ForeignKey
A Foreign Key is a field (or combination of fields) in one table that uniquely identifies a row in another table. The foreign key establishes a link between two tables, ensuring referential integrity. It ensures that the values in the foreign key column match values in the primary key column of another table, or are NULL if allowed. This helps maintain consistency and relationships between tables.

- Example: In the Booking table, the Customer_ID is a foreign key that links each booking to a specific customer in the Customer table. It ensures that a booking can only be made by an existing customer and prevents orphaned records in the Booking table.

NOTNULL
The NOT NULL constraint ensures that a field cannot have a NULL value. This constraint is applied to fields that are mandatory for each record, ensuring that the field always contains a valid value. By enforcing the NOT NULL constraint, you can ensure that critical data is never missing or incomplete.

- Example: In the Room table, the Room_No and Price fields should be set to NOT NULL, as a room number and price are essential for a valid room record. Without these values, a room record would be incomplete and invalid.

CHECK
The CHECK constraint is used to ensure that a field's value meets certain conditions or requirements before the data is entered or updated in the table. It allows you to enforce data integrity by restricting the values allowed in a column based on a specified condition.

- Example: In the Room table, the Price field might have a CHECK constraint like `CHECK (Price > 0)`. This ensures that the price of a room must always be greater than 0, preventing invalid or erroneous data (e.g., negative prices) from being entered into the database.

4.2 Joins

In SQL, a **JOIN** is a clause used to combine rows from two or more tables based on a related column between them. The purpose of a join is to combine data that is spread across multiple tables, often based on a common attribute, such as a `Customer_ID`, `Room_No`, or `Booking_ID`.

Types of Joins

1. **INNER JOIN**:
   - **Description**: The `INNER JOIN` returns only the rows where there is a match in both tables. If there is no match, the row is excluded from the result.
   - **Use Case**: This is the most common join used when you only need the records that have matching data in both tables.
   - **Example**:

     ```
     sql
     CopyEdit
     SELECT Customer.Name, Booking.Booking_ID
     FROM Customer
     INNER JOIN Booking
     ON Customer.Customer_ID = Booking.Customer_ID;
     ```

     - This query will return only the customers who have made bookings.
2. **LEFT JOIN (or LEFT OUTER JOIN)**:
   - **Description**: The `LEFT JOIN` returns all the rows from the left table (the first one) and the matching rows from the right table (the second one). If there is no match in the right table, the result will contain `NULL` for columns from the right table.
   - **Use Case**: This is useful when you want to include all records from the left table, even if there is no matching record in the right table.
   - **Example**:

     ```
     sql
     CopyEdit
     SELECT Customer.Name, Booking.Booking_ID
     FROM Customer
     LEFT JOIN Booking
     ON Customer.Customer_ID = Booking.Customer_ID;
     ```

     - This query will return all customers, including those who haven't made any bookings (with `NULL`values for `Booking_ID`).
3. **RIGHT JOIN (or RIGHT OUTER JOIN)**:
   - **Description**: The `RIGHT JOIN` is the opposite of the `LEFT JOIN`. It returns all the rows from the right table and the matching rows from the left table. If there is no match in the left table, the result will contain `NULL`for columns from the left table.

- **Use Case**: This is useful when you want to include all records from the right table, even if there is no matching record in the left table.
- **Example**:

```
sql
CopyEdit
SELECT Staff.Name, Booking.Booking_ID
FROM Staff
RIGHT JOIN Booking
ON Staff.Staff_ID = Booking.Staff_ID;
```

  - This query will return all bookings, including those without an assigned staff member (with `NULL`values for `Staff.Name`).

4. **FULL JOIN (or FULL OUTER JOIN)**:
   - **Description**: The `FULL JOIN` returns all rows when there is a match in either the left table or the right table. It returns `NULL` for columns where there is no match in one of the tables.
   - **Use Case**: This is useful when you want to include all records from both tables, even if there is no match between them.
   - **Example**:

```
sql
CopyEdit
SELECT Customer.Name, Payment.Amount
FROM Customer
FULL JOIN Payment
ON Customer.Customer_ID = Payment.Customer_ID;
```

     - This query will return all customers and payments, including those customers who haven't made any payments and those payments that are not linked to any customer.

5. **CROSS JOIN**:
   - **Description**: The `CROSS JOIN` returns the Cartesian product of the two tables, i.e., it returns all possible combinations of rows from both tables. It doesn't require a condition to join.
   - **Use Case**: This is used when you want to generate all possible combinations of records from two tables.
   - **Example**:

```
sql
CopyEdit
SELECT Customer.Name, Room.Room_No
FROM Customer
CROSS JOIN Room;
```

     - This query will return all combinations of customers and rooms, even if they are not related to each other.

Example Use Case in a Hotel Management System:

Let's consider a few practical examples using your Hotel Management System:

1. **INNERJOIN:**
   Get a list of all customers and the rooms they have booked:

   ```
   sql
   CopyEdit
   SELECT Customer.Name, Room.Room_No
   FROM Customer
   INNER JOIN Booking
   ON Customer.Customer_ID = Booking.Customer_ID
   INNER JOIN Room
   ON Booking.Room_No = Room.Room_No;
   ```

   - This will return customers who have made bookings along with the room number they booked.
2. **LEFTJOIN:**
   Get all customers and their bookings (if any):

   ```
   sql
   CopyEdit
   SELECT Customer.Name, Booking.Booking_ID
   FROM Customer
   LEFT JOIN Booking
   ON Customer.Customer_ID = Booking.Customer_ID;
   ```

   - This will return all customers, including those who have not made any bookings (with `NULL` values for `Booking_ID`).
3. **RIGHTJOIN:**
   Get all rooms and the bookings assigned to them:

   ```
   sql
   CopyEdit
   SELECT Room.Room_No, Booking.Booking_ID
   FROM Room
   RIGHT JOIN Booking
   ON Room.Room_No = Booking.Room_No;
   ```

   - This will return all bookings, including those that have no associated room (with `NULL` values for `Room_No`).
4. **FULLJOIN:**
   Get all customers and all payments they've made:

   ```
   sql
   CopyEdit
   SELECT Customer.Name, Payment.Amount
   FROM Customer
   FULL JOIN Payment
   ```

```
ON Customer.Customer_ID = Payment.Customer_ID;
```

- This will return all customers and their payment records, even if they haven't made a payment or if a payment is not linked to a customer.

4.3 Views

A **view** in SQL is a virtual table that provides a way to store a SQL query for later use. It doesn't store data itself, but instead, it displays data stored in one or more tables. Views allow you to simplify complex queries, enhance security by restricting access to certain columns, and present data in a more user-friendly way.

How Views Work:

- **Virtual Table**: A view acts like a table, but it does not actually store any data. Instead, it stores a SQL query that is executed each time you query the view.
- **Simplification**: Views simplify complex SQL queries by encapsulating them into a single object. This helps to reuse common queries and makes it easier to manage and maintain the code.

Advantages of Using Views:

1. **Data Abstraction**: Views provide an abstraction layer between the user and the underlying tables. You can present data in a simpler or more organized form.
2. **Security**: You can restrict access to specific columns or rows of a table by creating a view. For example, you can create a view that shows only a subset of columns from a table and grant users access to the view without exposing sensitive information.
3. **Simplified Queries**: Views encapsulate complex joins, aggregations, or calculations, making it easier to reuse and maintain queries.
4. **Consistency**: Using views ensures consistency, as the same query logic is used whenever the view is queried.

Types of Views

1. **Simple Views**:
   - These views are created based on a single table and do not involve complex joins or aggregations.
   - **Example**: A view showing customer names and their corresponding bookings.

     ```
     sql
     CopyEdit
     CREATE VIEW CustomerBookings AS
     SELECT Customer.Name, Booking.Booking_ID
     FROM Customer
     INNER JOIN Booking ON Customer.Customer_ID =
     Booking.Customer_ID;
     ```

2. **Complex Views**:
   - These involve multiple tables and may include joins, aggregations, or nested queries.

- **Example**: A view that shows room availability, along with the customer names and booking details.

```
sql
CopyEdit
CREATE VIEW RoomAvailability AS
SELECT Room.Room_No, Room.Type, Booking.Booking_ID,
Customer.Name
FROM Room
LEFT JOIN Booking ON Room.Room_No = Booking.Room_No
LEFT JOIN Customer ON Booking.Customer_ID =
Customer.Customer_ID
WHERE Room.Status = 'Available';
```

3. **Materialized Views** (Not supported in all databases):
   - Unlike regular views, materialized views store the query result physically and can be periodically refreshed. This improves performance for complex queries but uses more storage.
   - **Example**: A materialized view to store aggregated payment information.

```
sql
CopyEdit
CREATE MATERIALIZED VIEW PaymentSummary AS
SELECT Customer.Customer_ID, SUM(Payment.Amount) AS
TotalPayment
FROM Customer
JOIN Payment ON Customer.Customer_ID =
Payment.Customer_ID
GROUP BY Customer.Customer_ID;
```

4.4 Triggers

Triggers in SQL

A **trigger** is a special kind of stored procedure that automatically executes or fires when certain events occur on a specific table or view in a database. Triggers are commonly used to enforce business rules, maintain data integrity, and automate tasks like logging changes, updating data, or preventing incorrect data from being inserted or updated.

How Triggers Work:

- **Automatic Execution**: A trigger runs automatically in response to events such as insertions, updates, or deletions on a specified table or view.
- **Event-Driven**: Triggers are defined to respond to specific database events. For example, a trigger might execute when a new record is inserted, a record is updated, or a record is deleted.

- **Associated with Tables**: Triggers are always associated with a specific table and operate on data in that table.

Types of Triggers:

1. **BEFORE Trigger**:
   - Executes **before** an insert, update, or delete operation is performed on the table.
   - **Use Case**: You can use `BEFORE` triggers to validate or modify data before it gets inserted or updated in the table.
   - **Example**: Check if a customer's age is over 18 before allowing them to make a booking.

```
sql
CopyEdit
CREATE TRIGGER CheckAgeBeforeBooking
BEFORE INSERT ON Booking
FOR EACH ROW
BEGIN
  IF NEW.Age < 18 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Customer must be 18 or older
to make a booking.';
  END IF;
END;
```

2. **AFTER Trigger**:
   - Executes **after** an insert, update, or delete operation is performed on the table.
   - **Use Case**: `AFTER` triggers are often used for logging changes, updating related tables, or performing additional actions after the main operation has completed.
   - **Example**: After a payment is made, update the room status to "Booked."

```
sql
CopyEdit
CREATE TRIGGER UpdateRoomStatusAfterPayment
AFTER INSERT ON Payment
FOR EACH ROW
BEGIN
  UPDATE Room
  SET Status = 'Booked'
  WHERE Room_No = NEW.Room_No;
END;
```

3. **INSTEAD OF Trigger**:
   - Executes in place of an insert, update, or delete operation. Instead of performing the usual operation, it performs the action defined within the trigger.
   - **Use Case**: `INSTEAD OF` triggers are commonly used with views to handle insert, update, or delete actions on views where direct modification isn't allowed.
   - **Example**: Update a view instead of a direct table update.

```
sql
CopyEdit
CREATE TRIGGER InsteadOfUpdateBooking
INSTEAD OF UPDATE ON BookingView
FOR EACH ROW
BEGIN
  UPDATE Booking
  SET CheckIn = NEW.CheckIn, CheckOut = NEW.CheckOut
  WHERE Booking_ID = OLD.Booking_ID;
END;
```

Trigger Components:

- **Trigger Event**: The action that activates the trigger (e.g., INSERT, UPDATE, DELETE).
- **Trigger Timing**: Specifies when the trigger should fire relative to the operation (BEFORE, AFTER, INSTEAD OF).
- **Trigger Body**: The SQL statements that will be executed when the trigger is fired.

Trigger Syntax:

The general syntax for creating a trigger is:

```
sql
CopyEdit
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
   -- SQL statements to execute
END;
```

Use Cases for Triggers in a Hotel Management System:

1. **Preventing Invalid Data**:
   - **Use Case**: Ensure that no booking is made for a room that is already marked as "Booked."

```
sql
CopyEdit
CREATE TRIGGER PreventDoubleBooking
BEFORE INSERT ON Booking
FOR EACH ROW
BEGIN
  DECLARE room_status VARCHAR(20);
  SELECT Status INTO room_status
  FROM Room
  WHERE Room_No = NEW.Room_No;
```

```
  IF room_status = 'Booked' THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Room is already booked.';
  END IF;
END;
```

2. **Logging Changes**:
   - **Use Case**: Track when a room's status changes (e.g., from available to under maintenance) and log these changes in a `RoomStatusHistory` table.

```
sql
CopyEdit
CREATE TRIGGER LogRoomStatusChange
AFTER UPDATE ON Room
FOR EACH ROW
BEGIN
  IF OLD.Status != NEW.Status THEN
    INSERT INTO RoomStatusHistory (Room_No, Old_Status,
New_Status, Change_Date)
    VALUES (NEW.Room_No, OLD.Status, NEW.Status, NOW());
  END IF;
END;
```

3. **Updating Dependent Data**:
   - **Use Case**: Automatically update the total payment amount when a new payment is added.

```
sql
CopyEdit
CREATE TRIGGER UpdateTotalPaymentAfterPayment
AFTER INSERT ON Payment
FOR EACH ROW
BEGIN
  UPDATE Booking
  SET Total_Payment = Total_Payment + NEW.Amount
  WHERE Booking_ID = NEW.Booking_ID;
END;
```

4. **Automatic Calculations**:
   - **Use Case**: Automatically calculate and update the `TotalAmount` field in a `Booking` table when a new payment is made.

```
sql
CopyEdit
CREATE TRIGGER CalculateTotalAmount
AFTER INSERT ON Payment
FOR EACH ROW
BEGIN
  DECLARE total_amount DECIMAL(10, 2);
  SELECT SUM(Amount) INTO total_amount
```

```
  FROM Payment
  WHERE Booking_ID = NEW.Booking_ID;

  UPDATE Booking
  SET TotalAmount = total_amount
  WHERE Booking_ID = NEW.Booking_ID;
END;
```

5. **Data Validation**:
   - **Use Case**: Ensure that the `CheckOut` date is later than the `CheckIn` date before inserting a booking.

```
sql
CopyEdit
CREATE TRIGGER ValidateBookingDates
BEFORE INSERT ON Booking
FOR EACH ROW
BEGIN
  IF NEW.CheckOut <= NEW.CheckIn THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'CheckOut date must be later than
CheckIn date.';
  END IF;
END;
```

Advantages of Using Triggers:

- **Automated Actions**: Triggers automate tasks that need to occur when certain events happen in the database, such as updating related records or enforcing business rules.
- **Data Integrity**: Triggers can help ensure the integrity of data by performing checks or updates automatically whenever the underlying data changes.
- **Auditing**: You can use triggers to automatically track and log changes in the database, which is useful for auditing and maintaining historical records.
- **Consistency**: Triggers ensure that business rules are consistently applied every time a relevant action occurs in the database.

Disadvantages of Using Triggers:

- **Performance Overhead**: Triggers introduce additional processing, which may impact database performance, especially for complex operations or large datasets.
- **Complexity**: If not well-managed, triggers can make the system harder to understand and maintain, especially when multiple triggers are applied to the same table.
- **Debugging Challenges**: Debugging triggers can be difficult because they execute automatically and may not be visible to the user.

4.5 Cursors

A **cursor** is a database object used to retrieve, manipulate, and navigate through a result set row by row. Cursors allow for processing of SQL query results one row at a time, which is

particularly useful when you need to perform complex operations that cannot be handled with a single query.

How Cursors Work:

- **Result Set**: A cursor is associated with a result set from a query.
- **Row-by-Row Processing**: Instead of retrieving the entire result set at once, a cursor lets you process the data one row at a time.
- **Iteration**: Cursors allow iteration over the result set, making it possible to work with individual rows as needed.

Types of Cursors:

1. **Implicit Cursors**:
   - Automatically created by SQL when a SELECT statement is executed.
   - No explicit declaration or programming required for their creation.
   - Primarily used for simple queries that return a single result set.
2. **Explicit Cursors**:
   - Explicitly declared by the user in stored procedures or functions to handle more complex queries and control the row-by-row processing.
   - Allows more flexibility, such as looping through the result set, updating rows, or performing calculations.
   - They are explicitly defined using SQL commands like `DECLARE`, `OPEN`, `FETCH`, and `CLOSE`.

Cursor Operations:

1. **DECLARE**: Declares a cursor to define the result set.
2. **OPEN**: Opens the cursor and executes the query associated with it, making the result set available for processing.
3. **FETCH**: Retrieves the next row of data from the cursor. This operation moves the cursor to the next row in the result set.
4. **CLOSE**: Closes the cursor and frees the resources associated with it.

Cursor Example in Hotel Management System:

Suppose we want to calculate and update the total amount paid for each room booked in the `Booking` table after every payment. Here's how we can use a cursor:

```
sql
CopyEdit
DECLARE room_cursor CURSOR FOR
SELECT Room_No, Amount
FROM Payment;

OPEN room_cursor;

FETCH NEXT FROM room_cursor INTO @Room_No, @Amount;
```

```
WHILE @@FETCH_STATUS = 0
BEGIN
    -- Update the total amount for the booked room
    UPDATE Room
    SET Total_Payment = Total_Payment + @Amount
    WHERE Room_No = @Room_No;

    FETCH NEXT FROM room_cursor INTO @Room_No, @Amount;
END;

CLOSE room_cursor;
DEALLOCATE room_cursor;
```

Cursor Components:

1. **Cursor Declaration**: Declaring the cursor with the query that retrieves the necessary data.
2. **Cursor Opening**: Opening the cursor to execute the query and make the data available for processing.
3. **Fetching Rows**: Iterating over the result set by fetching each row and processing it as needed.
4. **Cursor Closing**: Closing the cursor to release resources once the data has been processed.
5. **Deallocating the Cursor**: Deallocating the cursor to free up the memory and resources used.

Advantages of Cursors:

1. **Row-by-Row Processing**: Cursors allow you to process data one row at a time, which is useful for complex operations that cannot be performed in a single SQL statement.
2. **Flexibility**: Cursors provide more flexibility for performing operations like conditional updates, calculations, or row-specific actions.
3. **Complex Logic**: They allow you to implement more complex logic that involves multiple operations on each row of data.

Disadvantages of Cursors:

1. **Performance Overhead**: Cursors can significantly impact performance because they require row-by-row processing, which is typically slower than set-based operations (i.e., operations on entire sets of rows).
2. **Resource Intensive**: Cursors consume memory and resources while they are open, which could lead to potential performance issues if not managed properly.
3. **Complexity**: Using cursors can make SQL code more complex and harder to maintain compared to set-based operations.

Use Cases for Cursors in a Hotel Management System:

1. **Processing Payments**:
    - Use cursors to iterate over payment records and update customer or booking payment totals.

2. **Tracking Room Availability**:
    - Use cursors to iterate through room bookings, checking if a room is booked for a given date, and updating the room's availability status.
3. **Automated Notifications**:
    - Use cursors to iterate through customer records and send out notifications for promotions, booking confirmations, or reminders.

CHAPTER 5: 4B – NORMALIZATION

5.1 First Normal Form (1NF

1NF is the first step in the normalization process, which ensures that the database structure is free from certain types of redundancy and inconsistency. A table is in 1NF if it meets the following criteria:

1. **Atomicity**: All attributes (columns) must contain atomic values, meaning that each column should hold only one value per row. There should be no multiple or repeating values in a single cell. This avoids situations where a single attribute contains a list of values or multiple values for a single record.

   Example**:** If a table has a column called "Phone Numbers" that stores multiple phone numbers for a customer in one cell (e.g., "1234567890, 0987654321"), it violates 1NF. To meet 1NF, each phone number should be stored in its own row or separate column.

2. **Eliminate Repeating Groups**: A repeating group occurs when a table contains multiple columns that store similar data, leading to redundancy. Repeating groups should be removed by creating separate tables. The idea is to avoid storing multiple values for similar attributes in the same row.

   Example**:** A table with columns like "Room_1", "Room_2", "Room_3" for storing a customer's room preferences would violate 1NF. Each room preference should be stored in a new row or separate table, rather than repeating the columns.

3. **Uniqueness of Rows**: Each row in the table should be unique. This is usually achieved by defining a primary key, which ensures that no two rows are identical in the table.

By ensuring atomic values and eliminating repeating groups, 1NF enhances the structure of the database and prepares it for further normalization steps.

5.2 Second Normal Form (2NF)

A table is in Second Normal Form (2NF) if it is in **First Normal Form (1NF)** and if all **partial dependencies** are eliminated. A partial dependency occurs when a non-prime attribute (an attribute that is not part of the primary key) depends only on a part of the composite primary key, rather than the entire key.

To achieve 2NF, the following steps are followed:

1. **Remove Partial Dependencies**: If a table has a **composite primary key** (a primary key that consists of more than one column), 2NF requires that every non-prime attribute must depend on all parts of the composite key. If any non-prime attribute depends only on part of the key, this dependency is considered partial, and the table needs to be split.

2. **Eliminate Redundancy**: In the context of the Room and Booking example, suppose a table contains the following fields:
    - Booking_ID**,** Customer_ID**,** Room_No**,** Room_Type**,** Room_Price
      The primary key is a composite key formed by Booking_ID and Room_No**.** However, Room_Type and Room_Price depend only on Room_No, not on the entire composite key (Booking_ID, Room_No). These are partial dependencies and violate 2NF.
    - Solution**:** To achieve 2NF, we split the table into two:
        1. A Room table: Contains Room_No**,** Room_Type**,** Room_Price
        2. A Booking table: Contains Booking_ID**,** Customer_ID**,** Room_No, and CheckIn**,** CheckOutdetails.
3. **Data Integrity**: By eliminating partial dependencies, 2NF ensures that non-prime attributes are fully dependent on the primary key, leading to a more efficient and logically structured database. This reduces redundancy and ensures that the data is stored in a more meaningful and accurate manner.

5.3 Third Normal Form (3NF)

In Third Normal Form (3NF), a table is considered normalized if it satisfies the conditions of Second Normal Form (2NF) and has no transitive dependencies. Transitive dependencies occur when a non-prime attribute (an attribute that is not part of the primary key) depends on another non-prime attribute, rather than directly on the primary key. This type of dependency can lead to redundancy and inefficiency, so 3NF resolves this issue by ensuring that all non-key attributes depend only on the primary key.

The main goal of 3NF is to eliminate these indirect relationships by restructuring the database so that each non-key attribute is only dependent on the primary key. This process not only reduces redundancy but also enhances data integrity and makes it easier to maintain the database over time.

Steps to Achieve 3NF:

1. **Eliminate Transitive Dependencies**:
   In 2NF, partial dependencies were removed. Now, in 3NF, any transitive dependency must also be removed. A transitive dependency occurs when a non-prime attribute depends on another non-prime attribute rather than directly on the primary key. For example, in a table where the Customer_Name and Customer_Address depend on the Customer_ID, and the Room_Price depends on Room_No, there is a transitive dependency because the Room_Price is indirectly dependent on the Booking_ID through the Room_No.

   To eliminate transitive dependencies, we split the data into separate tables, ensuring that each non-prime attribute is only directly dependent on the primary key.

2. **Create Separate Tables for Non-Key Attributes**:
   If a non-prime attribute is dependent on another non-prime attribute, it should be moved to a separate table. This process helps in reducing redundancy and improving data integrity. For example:
    - In a table where Room_Price is dependent on Room_No and Room_Type depends on Room_No, it's better to separate the Room details (such as

Room_Type and Room_Price) into a Room table, leaving the Booking table to store only the Booking_ID, Customer_ID, Room_No, and booking-specific details.

3. **Ensure Direct Dependency on Primary Key**:
   After eliminating transitive dependencies, every non-prime attribute should depend only on the primary key. For instance, in the case of Customer_Name and Customer_Address, these attributes should depend only on the Customer_ID and not indirectly on other attributes in the table. By creating separate tables for each entity (such as Customer and Room), we ensure that non-key attributes are only dependent on their respective primary keys.

Benefits of 3NF:

- **Reduced Redundancy**: By eliminating transitive dependencies, the database avoids the repetition of data. For example, customer information is now stored only in the Customer table, and room details are stored only in the Room table, rather than being repeated across multiple bookings.
- **Improved Data Integrity**: With the elimination of transitive dependencies, updates to non-key attributes (like customer name or room price) are easier to manage, as the information is stored in one place.
- **Better Maintainability**: A database in 3NF is more flexible and easier to modify or extend, as changes to one table do not impact others due to the reduced redundancy and better-defined relationships.

By achieving 3NF, the database is more efficient, easier to maintain, and less prone to anomalies during data insertion, updating, or deletion. This is the final step in the normalization process for ensuring that the database structure is optimized for both performance and integrity.

CHAPTER 6: 5B – CONCURRENCY CONTROL & RECOVERY

6.1 Transactions

Ensuring Atomicity Using START TRANSACTION, COMMIT, and ROLLBACK

Atomicity is a fundamental property of database transactions that ensures that a sequence of operations within a transaction is completed entirely or not executed at all. In other words, either all changes are made to the database, or none are, ensuring the database remains in a consistent state.

In MySQL and other relational database management systems, atomicity is achieved through the use of **START TRANSACTION**, **COMMIT**, and **ROLLBACK** statements.

START TRANSACTION

- The **START TRANSACTION** statement marks the beginning of a transaction.
- This indicates to the database system that all subsequent operations are part of a single transaction and should be treated as such.
- It allows the user to execute multiple SQL commands, and the changes will only be applied to the database once the transaction is committed.
- If anything goes wrong during the transaction, the changes made so far can be rolled back, ensuring no partial updates are made.

COMMIT

- The **COMMIT** statement is used to save all changes made during the current transaction to the database.
- Once **COMMIT** is executed, all the changes made during the transaction are permanently applied to the database.
- The transaction is considered successfully completed and is ended after the **COMMIT** statement.
- For example, when a customer makes a booking and a payment, if both actions are successfully completed, **COMMIT** ensures that these changes are stored in the database.

ROLLBACK

- The **ROLLBACK** statement is used to undo all changes made during the current transaction.
- If something goes wrong during the transaction (e.g., a database error or constraint violation), **ROLLBACK**ensures that all changes made up until that point are discarded, and the database is returned to its previous consistent state.
- **ROLLBACK** ensures that no partial updates are applied to the database in the case of failure, thus maintaining atomicity.

Example

1. A hotel booking system initiates a transaction by calling **START TRANSACTION**.

```
sql
CopyEdit
START TRANSACTION;
```

2. The system performs the first operation, such as creating a new booking record.

```
sql
CopyEdit
INSERT INTO Booking (Booking_ID, Customer_ID, Room_No, CheckIn,
CheckOut)
VALUES (101, 1, 202, '2025-05-01', '2025-05-05');
```

3. The system performs the second operation, such as recording the payment for the booking.

```
sql
CopyEdit
INSERT INTO Payment (Payment_ID, Booking_ID, Amount, Payment_Date)
VALUES (1001, 101, 500, '2025-05-01');
```

4. If both operations are successful, the system executes **COMMIT** to save the changes to the database permanently.

```
sql
CopyEdit
COMMIT;
```

5. However, if an error occurs during one of the operations (for example, if the payment record insertion fails), the system can execute **ROLLBACK** to undo all previous changes made during the transaction.

```
sql
CopyEdit
ROLLBACK;
```

By using **START TRANSACTION**, **COMMIT**, and **ROLLBACK**, the system ensures that either both the booking and payment operations are completed together (atomic), or neither are, preserving the integrity of the data. This ensures the hotel management system's database remains consistent even in the event of unexpected issues or failures during the transaction process.

Advantages of Using Atomic Transactions:

1. **Consistency**: Ensures that the database remains in a consistent state by preventing partial changes that could lead to data corruption.
2. **Error Handling**: Makes it easy to handle errors by rolling back all changes made during a failed transaction.

3. **Data Integrity**: Protects against data inconsistency that might arise from system crashes, hardware failures, or network issues during a multi-step operation.
4. **Predictable State**: The database either reflects the changes of an entire transaction or not at all, providing predictability for future operations.

In a hotel management system, for example, atomicity ensures that a room booking and the corresponding payment are treated as one unit. If the payment process fails after the booking, the entire transaction is rolled back, so the customer is not charged without a room reservation.

6.2 Locking

SELECT ... FOR UPDATE to Lock Rows During Critical Updates

In relational database management systems like MySQL, **SELECT ... FOR UPDATE** is used to lock rows that are being selected for modification, ensuring that other transactions cannot modify or access the same rows concurrently until the transaction is completed. This mechanism is particularly useful when dealing with critical updates, ensuring that data integrity is maintained even in multi-user or multi-threaded environments.

What is SELECT ... FOR UPDATE?

The **SELECT ... FOR UPDATE** statement is a type of SQL query that locks the selected rows for update. When a row is locked using **FOR UPDATE**, other transactions are unable to make changes to the locked rows until the current transaction is completed (i.e., either committed or rolled back). This prevents race conditions, where multiple transactions might attempt to update the same row simultaneously, causing data inconsistencies or conflicts.

How Does SELECT ... FOR UPDATE Work?

1. **Locking Mechanism**: When **SELECT ... FOR UPDATE** is executed, the rows retrieved by the query are locked for the duration of the current transaction. Other transactions that attempt to update these rows will be blocked until the transaction holding the lock completes.
2. **Transaction Context**: **SELECT ... FOR UPDATE** is typically used within a transaction that is started with **START TRANSACTION**. Once the rows are locked, they remain locked until the transaction is either **COMMIT**ted or **ROLLBACK**ped. During this time, other operations that attempt to modify the locked rows will be blocked.
3. **Preventing Concurrent Modifications**: This lock ensures that only one transaction can update the data at a time, preventing issues such as overwriting changes made by other transactions.

Use Case Scenario:

In a hotel management system, imagine you want to update the availability status of a room after a customer checks out. Without proper locking, two different users might try to update the same room status simultaneously, leading to inconsistencies in the database (for example, both users might mark the same room as available when it's not).

Example Workflow Using SELECT ... FOR UPDATE

1. **Start a Transaction**: The process starts by initiating a transaction to ensure that all changes made are treated atomically.

   ```
   sql
   CopyEdit
   START TRANSACTION;
   ```

2. **Lock the Row for Update**: The **SELECT ... FOR UPDATE** statement is used to lock the room row that is going to be updated. This ensures that no other transaction can modify the same row while the current transaction is in progress.

   ```
   sql
   CopyEdit
   SELECT * FROM Room WHERE Room_No = 202 FOR UPDATE;
   ```

   In this case, the row corresponding to room number 202 is locked, preventing other transactions from modifying it.

3. **Perform the Critical Update**: After locking the row, you can safely perform the critical update, such as changing the room status to "Available" after a guest checks out.

   ```
   sql
   CopyEdit
   UPDATE Room SET Status = 'Available' WHERE Room_No = 202;
   ```

4. **Commit the Transaction**: Once the update is complete, **COMMIT** is executed to save the changes and release the lock on the row.

   ```
   sql
   CopyEdit
   COMMIT;
   ```

5. **Rollback in Case of Failure**: If something goes wrong during the transaction (for example, if the payment processing fails), the transaction can be rolled back to ensure no partial changes are made, maintaining the database's consistency.

   ```
   sql
   CopyEdit
   ROLLBACK;
   ```

Advantages of Using SELECT ... FOR UPDATE

- **Prevents Race Conditions**: By locking the rows for update, **SELECT ... FOR UPDATE** ensures that only one transaction can modify the selected rows at a time, preventing multiple transactions from concurrently making conflicting changes to the same data.
- **Data Integrity**: This helps maintain data integrity during critical updates, especially in scenarios where the system is handling financial transactions, customer bookings, or any other sensitive data that needs to be updated accurately.

- **Improved Concurrency Control**: **SELECT ... FOR UPDATE** allows multiple transactions to be processed concurrently, but ensures that conflicting updates are avoided by preventing access to the locked rows until the transaction is completed.
- **Saves Processing Time**: By locking the rows early in the transaction, the system ensures that subsequent operations do not need to perform additional checks for conflicts, saving processing time in high-concurrency scenarios.

Potential Issues to Consider

- **Deadlocks**: One potential drawback of using row-level locks is the possibility of deadlocks, where two transactions lock each other's rows, waiting for each other to release the locks. To avoid deadlocks, it's important to design transactions carefully, ensuring they acquire locks in a consistent order.
- **Performance Overhead**: Locking rows for updates can introduce a performance overhead, especially when dealing with large volumes of data or when transactions are held open for long periods. It's essential to ensure that locks are held for as short a time as possible to minimize contention.

Best Practices

1. **Keep Transactions Short**: To reduce lock contention and the risk of deadlocks, ensure that transactions are as short as possible. Perform the necessary work quickly and commit the changes as soon as possible.
2. **Acquire Locks in a Consistent Order**: If your system requires multiple locks (e.g., locking both a booking and payment record), always acquire locks in a consistent order to avoid deadlocks.
3. **Handle Errors Gracefully**: Always use **ROLLBACK** in case of errors to ensure that any changes made are undone, leaving the database in a consistent state.
4. **Use Optimistic Locking for Non-Critical Updates**: For less critical updates, consider using optimistic locking (e.g., versioning) to avoid the performance overhead of row-level locks, especially when updates are less frequent.

6.3 Recovery

Implementing Manual Recovery Steps Using Transaction Logs and Backups

In any database management system, data recovery is a critical aspect of maintaining the integrity and availability of the data in the event of failures, crashes, or corruption. **Transaction logs** and **backups** are essential tools for implementing a recovery strategy, ensuring that the database can be restored to a consistent state after a failure.

What Are Transaction Logs?

Transaction logs are records of all transactions that have been executed on the database, including changes to data, schema modifications, and other operations. They are crucial for database recovery, as they provide a detailed history of all changes made to the database, allowing for precise restoration of the database to a specific point in time.

In MySQL, the transaction log is often called the **binary log**, which stores all changes to the database, including INSERT, UPDATE, DELETE, and other DDL (Data Definition Language) commands.

What Are Backups?

Backups are copies of the database at a specific point in time, often stored separately from the primary database. They serve as a safety net, allowing the system to be restored to a known, consistent state in case of data loss, corruption, or other failures. Backups can be full, incremental, or differential:

- **Full backups**: Complete copies of the entire database.
- **Incremental backups**: Backups that capture only the changes made since the last backup.
- **Differential backups**: Backups that capture the changes made since the last full backup.

Implementing Manual Recovery Steps

To manually recover a database using transaction logs and backups, follow these general steps:

Step 1: Identifying the Cause of the Failure

The first step in the recovery process is to identify the cause of the database failure. This could be a power outage, hardware failure, corruption in the data files, or an accidental deletion of critical data.

- **Check Database Logs**: Review MySQL's error logs and other system logs to identify the issue and determine if it was a system crash, data corruption, or human error.

Step 2: Restoring the Database from Backup

If the database is completely corrupted or lost, the first step in recovery is to restore the most recent backup. This allows the system to return to a known, consistent state before the failure.

1. **Locate the Backup**: Identify the latest full backup (or incremental/differential backup, depending on the recovery strategy).
2. **Restore the Backup**: Use MySQL commands to restore the backup file to the database.

   For example:

   ```
   bash
   CopyEdit
   mysql -u username -p database_name < /path/to/backup.sql
   ```

   This command will restore the backup into the database, replacing the current state with the backup's contents.

Step 3: Applying Transaction Logs (Point-in-Time Recovery)

Once the backup has been restored, the database is brought to a specific point in time (e.g., the time the backup was taken). To bring the database to the state it was in just before the failure, you need to apply the transaction logs that contain changes made after the backup was created.

1. **Identify the Last Successful Transaction**: Look at the timestamps in the transaction logs to identify the last transaction that was successfully committed before the failure.
2. **Apply the Transaction Logs**: Using the transaction logs (or binary logs in MySQL), replay the changes made after the backup was taken, up to the point of failure. In MySQL, this is typically done with the `mysqlbinlog`utility, which allows you to apply the logs.

   For example:

   ```
   bash
   CopyEdit
   mysqlbinlog /path/to/mysql-bin.000001 | mysql -u username
   -p database_name
   ```

   This command applies all transactions from the binary log to the database. You can specify a range of log files and timestamps to restore the database to the desired point in time.

3. **Commit the Changes**: Once all the transaction logs have been applied, commit the changes to the database. This ensures that all the data modifications from the logs are persisted.

Step 4: Verifying Database Integrity

After the recovery process, it is essential to verify the integrity of the database and ensure that all data is consistent. This can be done through various checks, such as:

1. **Running Integrity Checks**: Use MySQL's built-in `CHECK TABLE` command to check for table corruption or errors.

   Example:

   ```
   sql
   CopyEdit
   CHECK TABLE table_name;
   ```

2. **Query the Data**: Run queries to check if the data appears correct. For example, verify that customer records, booking statuses, and payment transactions are accurate.
3. **Testing Application Functionality**: Run the application that uses the database and ensure that all transactions (such as bookings, payments, and staff management) work correctly.

Step 5: Preventing Future Failures

After successfully recovering the database, it is important to implement strategies to prevent future failures and ensure a more robust recovery plan.

1. **Regular Backups**: Implement a regular backup strategy that includes full, incremental, and differential backups. Schedule backups to run automatically to ensure you always have up-to-date backups available for recovery.
2. **Transaction Log Archiving**: Ensure that transaction logs are archived regularly to support point-in-time recovery. Set up automated processes to manage log files, so they don't fill up storage or become unmanageable.
3. **Testing Recovery Procedures**: Regularly test recovery procedures to ensure that backups are valid and recovery steps can be performed quickly and correctly when needed.
4. **High Availability Setup**: Consider implementing high availability or replication solutions (e.g., master-slave replication in MySQL) to minimize downtime and provide immediate failover in case of failure.
5. **Monitor the Database**: Implement database monitoring to proactively detect issues such as performance degradation, deadlocks, or potential corruption, allowing for faster resolution and minimizing downtime.

Advantages of Transaction Logs and Backups in Recovery

- **Granular Recovery**: Transaction logs allow for granular, point-in-time recovery, enabling the database to be restored to any specific moment, down to the last committed transaction.
- **Minimized Data Loss**: Regular backups and transaction logs reduce data loss, as the database can be restored to the point just before the failure occurred, rather than losing all changes made after the last backup.
- **Flexibility**: You can restore from full backups for a simple recovery or use transaction logs for more complex recovery needs, such as point-in-time recovery or disaster recovery scenarios.
- **Data Integrity**: By applying transaction logs, you ensure that only committed transactions are replayed, preserving the integrity of the database and preventing partial updates.

CHAPTER 7: RESULTS & DISCUSSION

7.1 Sample Outputs

- Query results for available rooms
- Trigger execution logs
- Booking and payment history

7.2 Observations

- Improved query performance after normalization
- Accurate, consistent data flow
- Triggers automate room status management

CHAPTER 8: CONCLUSION & FUTURE ENHANCEMENT

The project successfully demonstrates a working DBMS for hotel operations. Future improvements may include:

- Integration with a web frontend
- SMS/Email notification support
- Admin dashboard with analytics
- Cloud-based storage

CHAPTER 9: REFERENCES

- “Database System Concepts” – Abraham Silberschatz
- MySQL Official Docs – https://dev.mysql.com
- W3Schools, GeeksforGeeks (for syntax examples)
- Lecture notes and project guidelines

CHAPTER NO **TITLE** **PAGE NO.**

LIST OF TABLES

ABBREVIATIONS