Data Engineering - CSL4030

Lab Assignment - 01

Irwindeep Singh
B22AI022

Sarthak Malviya
B22AI034

1 Hotel Management System

We have designed a hotel management system incorporating `MySQL` for database management. Further, we use `Python-Flask` app for backend.
The codebase used for the project along-with a `README.md` file can be found at: GitHub

2 Entity-Relation (ER) Diagram

The following Entity-Relation (ER) Diagram represents the schema for the Hotel Management System. It outlines the main entities involved in the system, such as Room, Guest, Booking, Payment, and User, along with their respective attributes. The relationships between these entities, such as "Booked By" and "Made By," are also depicted to illustrate how the data is interconnected within the system.

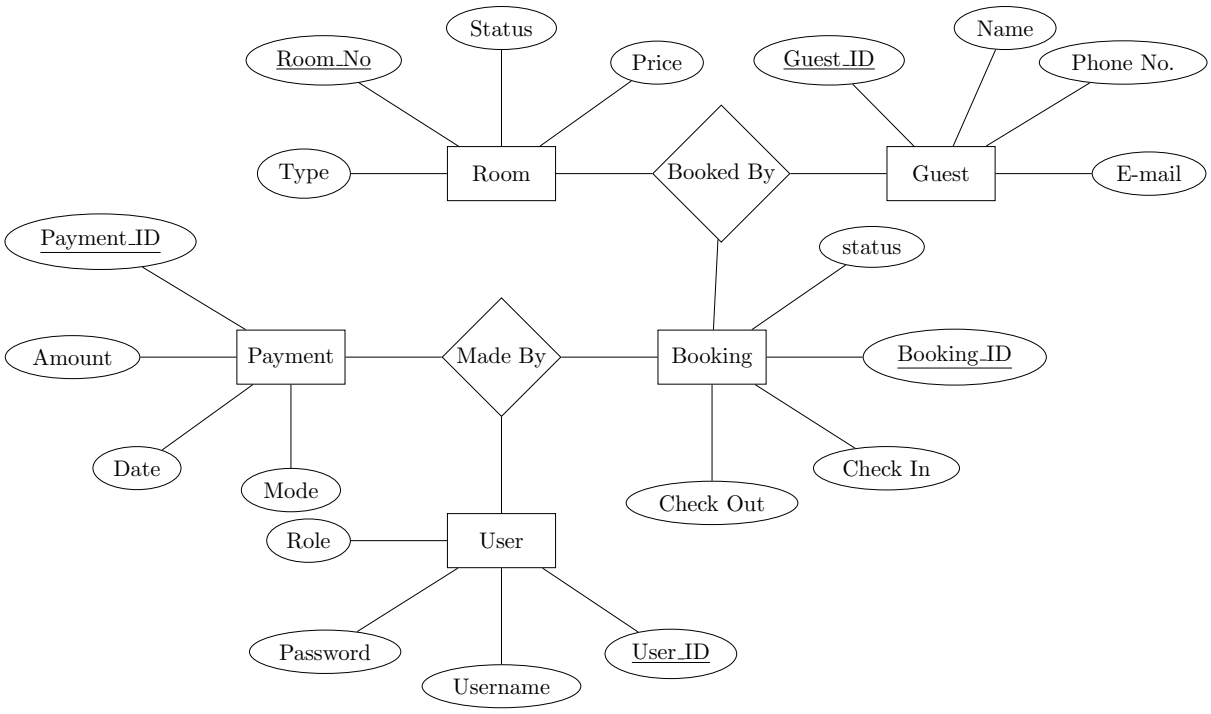

Figure 1: ER Diagram for Hotel Management System

3 Table Creation, Insertion of Dummy Records

We have used a dataset named `hotel_management` for this assignment, the same can be created with the following `MySQL` queries:

```
CREATE DATABASE hotel_management;
USE hotel_management;
```

3.1 Table Creation

We have Used 6 tables named - `rooms, payments, cancellations, guests, bookings, staff` which were created with the following queries:

```
CREATE TABLE rooms(room_no INT PRIMARY KEY, type VARCHAR(20), status VARCHAR(20), price FLOAT);
CREATE TABLE payments(payment_id INT PRIMARY KEY, mode VARCHAR(20), amount FLOAT, payment_time DATETIME);
CREATE TABLE cancellations(cancel_id INT AUTO_INCREMENT PRIMARY KEY, room_no INT, guest_id INT,
                    cancel_time DATETIME, note TEXT);
CREATE TABLE guests(guest_id INT PRIMARY KEY, fname VARCHAR(20), lname VARCHAR(20), phone_no VARCHAR(20),
                    email VARCHAR(255));
```

```
CREATE TABLE bookings(
    booking_id INT PRIMARY KEY,
    room_no INT,
    guest_id INT,
    booking_time DATETIME,
    total_occupants INT,
    travel_mode VARCHAR(20),
    check_in DATE,
    check_out DATE,
    payment_id INT
    );

CREATE TABLE staff(
    id INT PRIMARY KEY,
    email_id VARCHAR(255),
    password VARCHAR(255),
    fname VARCHAR(20),
    lname VARCHAR(20),
    role VARCHAR(20)
    );
```

3.2 Insertion of Dummy Records

To insert dummy records into the dataset, we must first connect to the dataset (via `localhost`). We have utilised `MySQLdb` (a python library to view and manipulate `MySQL` databases. The following program assumes that we have the password for `MySQL` user stored in the operating system environment. We are first populating the rooms, then we select 5 random rooms and populate bookings with those 5 rooms. With bookings known, we are updating room status and populating payments.

```
import MySQLdb
from MySQLdb import Error
import os

try:
    connection = MySQLdb.connect(
        host="localhost",
        user="root",
        password=os.getenv('MySQL'),
        database="hotel_management"
    )
    print("Connected to Database Successfully!")
except Error as e:
    print(f"Error Occurred: {e}")
```

Now, we can insert dummy records in all tables by obtaining a `cursor` object from the established connection, and with that `cursor`, we can execute queries and populate tables as follows:

```
cursor = connection.cursor()

# Creating empty rooms with different types and prices
for room_no in range(101, 201):
    if room_no <= 125: type, price = "Standard", 2500.0
    elif room_no <= 150: type, price = "Deluxe", 3200.0
    elif room_no <= 175: type, price = "Executive", 4000.0
    else: type, price = "Premium", 6500.0

    cursor.execute(f"INSERT INTO rooms VALUES({room_no}, '{type}', 'Empty', {price})")

connection.commit()
```

```
import numpy as np

# Assuming we have lists containing all the variables used
for id in range(1, 6):
    room_no = np.random.randint(low=101, high=200)
    query = f"{id}, {room_no}, '{id}', '{book_times[id-1]}', {occupants[id-1]}, '{travel_modes[id-1]}', "
    query += f"'{in_dates[id-1]}', '{out_dates[id-1]}', {id}"
    cursor.execute(f"INSERT INTO bookings VALUES({query})")
    cursor.execute(f"UPDATE rooms SET status='Booked' WHERE room_no={room_no}")

    query = f"{id}, '{fnames[id-1]}', '{lnames[id-1]}', '{phone_nos[id-1]}', '{emails[id-1]}'"
    cursor.execute(f"INSERT INTO guests VALUES({query})")

    cursor.execute(f"SELECT price FROM rooms WHERE room_no={room_no}")
    amount = cursor.fetchone()[0]
    cursor.execute(f"SELECT DATEDIFF(booking_till, booking_date) AS number_of_days WHERE room_no={room_no}")
    amount *= cursor.fetchone()[0]
    cursor.execute(f"INSERT INTO payments VALUES({id}, 'Cash', {amount}, {book_times[id-1]})")

connection.commit()
```

4 Normalisation

- **1NF (First Normal Form):**
  - Each table has a primary key.
  - All columns contain atomic values (e.g., VARCHAR for strings, INT for integers, FLOAT for prices).
  - **Conclusion:** The database is in 1NF.
- **2NF (Second Normal Form):**
  - All non-key attributes are fully functionally dependent on the entire primary key in all the tables.
  - **Conclusion:** The database is in 2NF.
- **3NF (Third Normal Form):**
  - There are no transitive dependencies in any of the table.
  - The database is in 2NF.
  - **Conclusion:** The database is in 3NF.
- **BCNF (Boyce-Codd Normal Form):**
  - For every functional dependency $X \to Y$, X is the superkey in all the tables.
  - The database is in 3NF.
  - **Conclusion:** The database is in BCNF

5 Hashing and Indexing Schemes

Hashing and indexing are critical mechanisms for optimizing query performance in MySQL databases. Both techniques are designed to enhance data retrieval efficiency, but they operate in fundamentally different ways.

- **Hashing:** Hashing is a technique used to distribute data across a hash table to speed up access. In `MySQL`, hashing is employed primarily in the context of hash indexes, which are a type of in-memory index used by the `MEMORY` storage engine. When a hash index is used, the hash function converts the key into a hash value, which determines the index location. This method is effective for equality searches but inefficient for range queries since hash functions do not maintain any order of data.
- **Indexing:** Indexing, on the other hand, involves creating data structures that improve the speed of data retrieval operations. `MySQL` supports various types of indexes, with B-tree and R-tree indexes being the most common:
  - **B-Trees:** B-tree (Balanced Tree) indexes are organized as a balanced tree structure, allowing efficient searching, insertion, and deletion operations. *They are the default index type for most `MySQL` storage engines.*
  - **R-Trees:** R-trees are used for indexing multi-dimensional information, such as spatial data.

6 Hash Function Design

To design a hash function (for rooms table) incorporating common characters of `"B22AI022"` and `"B22AI034"`, we have utilized the following python program:

```python
def hash_function(key: int, table_size: int) -> int:
    key_str, PRIME_1, PRIME_2 = str(key), 31, 7

    hash_value = 0 # Initialize hash value
    for char in key_str:
        hash_value = (hash_value * PRIME_1 + int(char)) % table_size

    common_chars = 'B22AI0' # Common characters in "B22AI022" and "B22AI034"
    for char in common_chars:
        hash_value = (hash_value * PRIME_2 + ord(char)) % table_size

    return hash_value % table_size

# Example usage:
key1, key2 = 22022022, 22023034
table_size = 97

print(f"Hash value for key '{key1}': {hash_function(key1, table_size)}")
print(f"Hash value for key '{key2}': {hash_function(key2, table_size)}")
```

```
Output:
    Hash value for key '22022022': 65
    Hash value for key '22023034': 10
```

7 Clustering Indexing

Since, clustering indexing is already applied `rooms` due to the presence of the primary key (`room_no`). This means that the rows of the table are physically ordered on disk according to the `room_no` key.

8 Secondary Indexing

With the connection to `MySQL` client made above, we can apply secondary indexing (i.e., indexing on attribute other than primary attribute) on the `rooms` table with the following python program:

```python
cursor.execute("CREATE INDEX idx_room_type ON rooms (type)")
connection.commit()

# Verifying the indexing
cursor.execute("SHOW INDEX FROM rooms")
indices = cursor.fetchall()
for index in indices:
    print(index)
```

```
Output:
    ('rooms', 0, 'PRIMARY', 1, 'room_no', 'A', 100, None, None, '', 'BTREE', '', '', 'NO')
    ('rooms', 1, 'idx_room_type', 1, 'type', 'A', 4, None, None, 'YES', 'BTREE', '', '', 'NO')
```

9 Clustering vs Secondary Indexing

9.1 Clustering Index (Primary Key Index)

- **Storage:**
  - **Physical Order:** The table data is physically ordered on disk according to the clustering index. This means that the rows are stored in the same order as the clustering index.
  - **Space Efficiency:** Because the data is stored in a sorted order, it can be more space-efficient when accessing data sequentially. However, if there are frequent updates, inserts, or deletes, maintaining this order can result in fragmentation and potential space overhead.
- **Execution Time:**
  - **Data Retrieval:** Accessing data using the clustering index is very efficient, especially for range queries. For example, retrieving all records within a specific range is fast because the data is stored in a sorted order.
  - **Search Performance:** Searching for a specific row using the clustering index is quick because it directly accesses the row using the indexed key.
  - **Insert/Update/Delete Performance:** These operations can be slower due to the need to maintain the sorted order of the table, which might involve reordering and shifting of rows.

9.2 Secondary Index (Non-Clustered Index)

- **Storage:**
  - **Separate Storage:** The secondary index is stored separately from the table data. It contains a copy of the indexed column(s) and pointers to the corresponding rows in the table.
  - **Space Overhead:** There is additional space overhead because the secondary index stores duplicate information. The more indexes you have, the more storage space is required.
- **Execution Time:**
  - **Data Retrieval:** Retrieving data using a secondary index can be efficient for specific queries where the indexed column is used. However, it involves an additional lookup step to fetch the actual row from the table.
  - **Search Performance:** Searching for a specific value in a secondary index is typically fast because secondary indexes use data structures like B-trees or hash tables, which allow for efficient lookups.
  - **Insert/Update/Delete Performance:** These operations can be slower with secondary indexes compared to tables with only a clustering index. Every time a row is inserted, updated, or deleted, all secondary indexes need to be updated, which can add overhead.

9.3 Summary

- **Clustering Index:**
  - **Pros:** Efficient for range queries, good for data retrieval when the data is frequently accessed in a sorted order.
  - **Cons:** Can cause performance issues with frequent inserts/updates due to the need to maintain data order; less flexible as there is only one clustering index per table.
- **Secondary Index:**
  - **Pros:** Flexible and allows for multiple secondary indexes; improves query performance for specific column searches; good for queries where the indexed columns are frequently used.
  - **Cons:** Additional storage space is required for indexes; can result in slower insert/update/delete operations due to the need to maintain multiple indexes.

10 Additional Queries

1. Add information about the inclusion of a new wing comprising 5 deluxe suites:

```
INSERT INTO rooms
VALUES
    (201, 'Deluxe', 'Empty', 3200.0),
    (202, 'Deluxe', 'Empty', 3200.0),
    (203, 'Deluxe', 'Empty', 3200.0),
    (204, 'Deluxe', 'Empty', 3200.0),
    (205, 'Deluxe', 'Empty', 3200.0);
```

2. Prepare a report on all guest bookings and cancellations in the month of August, 2024:

```
SELECT
    'Booking' AS type,
    booking_id AS id,
    room_no,
    booking_time AS time,
FROM bookings
WHERE booking_date >= '2024-08-01 00:00:00' AND booking_date < '2024-09-01 00:00:00'

UNION ALL

SELECT
    'Cancellation' AS type,
    cancellation_id AS id,
    room_no,
    cancel_time AS time,
FROM cancellations
WHERE cancel_time >= '2024-08-01 00:00:00' AND cancellation_date < '2024-09-01 00:00:00'
ORDER BY date;
```

3. Cancel all guest bookings made after 7PM for August 15, 2024:

```
INSERT INTO cancellations (room_no, guest_id, cancel_time, note)
SELECT room_no, guest_id, '2024-08-31 00:00:00', 'Additional Query #3'
FROM bookings
WHERE booking_time >= '2024-08-15 19:00:00';

DELETE FROM bookings
WHERE booking_time >= '2024-08-15 19:00:00';
```