Lab 1: Travel Booking System

Group Members:

Anuj Rajan Lalla
Roll Number: B22AI061

Abhinav Swami
Roll Number: B22AI003

Course: CSN4030

Lab 1: Travel Booking System

September 1, 2024

0.1 ER Diagram [10 points]

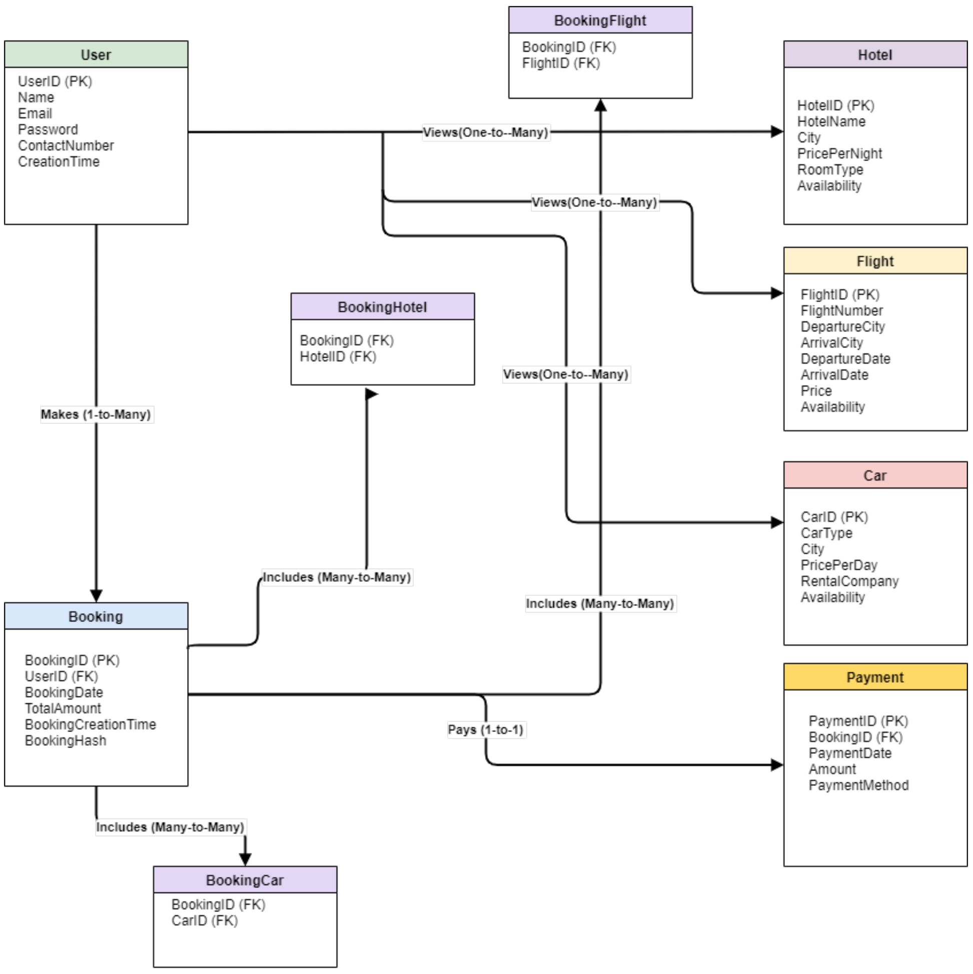


Figure 1: ER Diagram for the Travel Booking System

0.1.1 Explanation

The ER (Entity-Relationship) diagram for the Travel Booking System provides a visual representation of the system's data structure. It outlines the key entities such as **User**, **Booking**, **Flight**, **Hotel**, **Car**, and **Payment**, and their relationships:

- **User**: The **User** entity is linked to **Booking**, representing a one-to-many relationship, indicating that a user can make multiple bookings.
- **Booking**: The **Booking** entity acts as a central hub, connecting to other entities like **Flight**, **Hotel**, and **Car** through associative entities like **BookingFlight**, **BookingHotel**, and **BookingCar**. This setup reflects the many-to-many relationships between bookings and these services.
- **Flight, Hotel, Car**: These entities store specific information about flights, hotels, and cars, respectively, and are linked to bookings to represent the services included in a user's booking.

- **Payment**: The **Payment** entity is associated with **Booking** through a one-to-one relationship, ensuring that each booking corresponds to a single payment transaction.

This diagram provides a clear structure for managing various aspects of a travel booking system, emphasizing the interconnectivity and relational data integrity between different components.

0.2 Table Creation and Insertion of Dummy Records [30 points]

This task involves the creation of necessary tables and the insertion of dummy records into these tables using Python interfaced with SQL. The implementation for this task has been carried out in a Python notebook, where the following steps were performed:

- **Table Creation:** SQL commands were executed via Python to create the necessary tables as per the ER diagram. The tables include **User**, **Booking**, **Flight**, **Hotel**, **Car**, and **Payment**, among others.
- **Insertion of Dummy Records:** Sample data was inserted into each table to simulate real-world scenarios and ensure the correct functioning of the database structure.
- **Python-SQL Interface:** The Python notebook utilized appropriate libraries such as `mysql.connector` to interface with the SQL database, enabling the execution of SQL commands directly from Python.

The Python notebook containing the implementation is submitted along with this report. It includes detailed code snippets and explanations for each step, ensuring a clear understanding of the process involved in creating and populating the database tables.

0.3 Normalization Evaluation of Tables [10 points]

This section evaluates the normalization of each table in the Travel Booking System database. All tables have been assessed to ensure they meet at least the requirements for Second Normal Form (2NF). The trade-offs between further normalizing the tables to Third Normal Form (3NF) or Boyce-Codd Normal Form (BCNF) are also discussed.

0.3.1 User Table

Highest Normal Form: 3NF
Reasoning: All attributes (`Name`, `Email`, `Password`, etc.) are fully dependent on the primary key (`UserID`). There are no transitive dependencies.

0.3.2 Flight, Hotel, Car Tables

Highest Normal Form: 3NF
Reasoning: All non-key attributes are fully dependent on the primary key (e.g., `FlightID`, `HotelID`, `CarID`). No transitive dependencies exist.

0.3.3 Booking Table

Highest Normal Form: 3NF
Reasoning: Attributes (`UserID`, `BookingDate`, etc.) are dependent on `BookingID`, with no transitive dependencies.

0.3.4 Payment Table

Highest Normal Form: 3NF
Reasoning: Attributes are fully dependent on `PaymentID`, with no transitive dependencies.

0.3.5 BookingFlight, BookingHotel, BookingCar Tables

Highest Normal Form: BCNF
Reasoning: The composite keys (e.g., `BookingID`, `FlightID`) ensure no partial or transitive dependencies.

0.3.6 Trade-offs Between Normal Forms

- **2NF:** Simple design, but may allow some redundancy.
- **3NF:** Prevents redundancy and anomalies, usually sufficient.
- **BCNF:** Ensures no anomalies but can increase complexity.

Conclusion: The tables are already in 3NF or BCNF, which balances data integrity and performance. Further normalization is unnecessary unless specific performance issues arise.

0.4 Hashing and Indexing Schemes in MySQL [5 points]

MySQL provides various indexing and hashing schemes that significantly improve the performance of database operations by optimizing data retrieval processes. Below is a brief overview of these schemes:

0.4.1 Hashing in MySQL

MySQL uses hash indexes as an efficient means of looking up specific data. Hashing converts a search key into a fixed-size value (hash code) using a hash function, which then points directly to the location of the desired record in memory. This makes hash indexes particularly effective for equality comparisons, such as retrieving rows with a specific key value. However, hash indexes are not suitable for range queries because hash functions do not preserve the order of data.

0.4.2 B-Tree Indexing

The most commonly used indexing scheme in MySQL is B-Tree indexing. B-Trees allow for efficient retrieval of data across a range of values, which is ideal for range queries and sorting. B-Tree indexes are balanced tree structures where all leaf nodes are at the same level, ensuring consistent performance across operations. This indexing scheme is typically used on columns that are frequently involved in WHERE clauses, ORDER BY operations, or JOIN operations.

0.4.3 Full-Text Indexing

MySQL also supports full-text indexing, which is optimized for searching textual data. Full-text indexes enable efficient searches for words or phrases within large text fields, making them suitable for applications that involve complex search queries, such as those in content management systems.

0.4.4 Spatial Indexing

For applications involving geographic data, MySQL provides spatial indexing. This indexing scheme is used for spatial data types, such as points, lines, and polygons, and allows for efficient querying of spatial relationships and proximity calculations.

0.4.5 Trade-offs and Considerations

- **Hash Indexes:** Offer fast lookups for exact matches but are limited for range queries.
- **B-Tree Indexes:** Provide flexibility for both equality and range queries but may require more storage and maintenance overhead.
- **Full-Text Indexes:** Specialized for text searches, with additional storage and computational requirements.
- **Spatial Indexes:** Essential for geographic data but require careful management to ensure performance.

Conclusion: The choice of indexing and hashing schemes in MySQL depends on the specific query patterns and data characteristics. By selecting the appropriate index type, MySQL can significantly optimize the performance of database operations.

0.5 Custom Hash Function for Booking Table [10 points]

To optimize the storage and retrieval of data in the `Booking` table, a custom hash function was designed using Python. This function takes into consideration the alphabets common in all the roll numbers of the group members, specifically the prefix `"B22AI"`.

0.5.1 Motivation

The motivation behind this approach is to create a unique yet predictable hash based on both the roll number prefix and the `BookingID`. By using the roll number prefix, which is common among all group members, combined with the `BookingID`, the function generates a hash that is consistent and distributed across a fixed number of buckets. This is particularly useful in managing and retrieving records efficiently from a large dataset.

0.5.2 Implementation

The custom hash function was implemented in the Python notebook. The function uses the common prefix `"B22AI"` and combines it with the `BookingID` to generate a hash value. This value is then modded by 1000 to ensure it falls within a specific range, allowing the data to be distributed across 1000 buckets. This approach effectively balances the load across the storage space and reduces potential collisions.

0.5.3 Conclusion

This custom hash function efficiently uses the roll number prefix and `BookingID` to generate a well-distributed hash value, ensuring effective storage and retrieval of data in the `Booking` table.

0.6 Creation of Indexes on booking table [20 points]

This section covers the application of clustering and secondary indexing on the `Booking` table using Python. The implementation was carried out in a Python notebook interfacing with MySQL. Below is a description of the approach taken and the limitations encountered.

0.6.1 Clustering Index

In MySQL, a clustering index is automatically created on the primary key of the table, and MySQL does not allow the creation of multiple clustering indexes. However, following the logic of clustering indexes, we have defined an index on the `BookingDate` column. This ensures that the data is stored in the order of `BookingDate`, which can optimize range queries on booking dates.

0.6.2 Secondary Index

To enhance the performance of specific queries, a secondary index was created on the `BookingHash` column. Unlike clustering indexes, MySQL allows the creation of multiple secondary indexes, which do not affect the physical order of data in the table but significantly improve query performance.

0.6.3 Conclusion

By applying a logical clustering index on `BookingDate` and a secondary index on `BookingHash`, the system achieves a balance between query performance and the limitations imposed by MySQL's indexing capabilities. The Python notebook submitted alongside this report contains the detailed implementation of these indexing strategies.

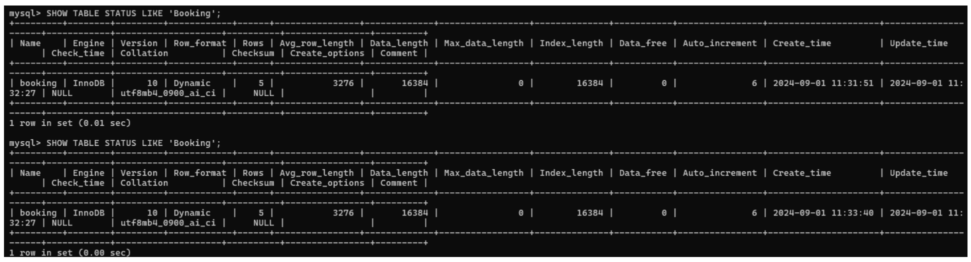
0.7 Comparison of Clustering vs Secondary Indexing [20 points]

This section compares and contrasts the storage and execution times of the clustering and secondary indexing schemes applied to the `Booking` table. The analysis was performed using SQL commands executed in the MySQL environment, with the results measured and documented for each indexing scheme.

0.7.1 Storage Requirements

The storage requirements for the `Booking` table under different indexing schemes were evaluated using the following SQL command before and after the indexing operations:

```
SHOW TABLE STATUS LIKE 'Booking';
```



```
mysql> SHOW TABLE STATUS LIKE 'Booking';
+---------+--------+---------+------------+------+----------------+-------------+-----------------+--------------+-----------+----------------+---------------------+-------------
------+------------+--------------------+----------+----------------+---------+
| Name    | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length | Index_length | Data_free | Auto_increment | Create_time         | Update_time
      | Check_time | Collation          | Checksum | Create_options | Comment |
+---------+--------+---------+------------+------+----------------+-------------+-----------------+--------------+-----------+----------------+---------------------+-------------
------+------------+--------------------+----------+----------------+---------+
| booking | InnoDB |      10 | Dynamic    |    5 |           3276 |       16384 |               0 |        16384 |         0 |              6 | 2024-09-01 11:31:51 | 2024-09-01 11:
32:27 | NULL       | utf8mb4_0900_ai_ci |     NULL |                |         |
+---------+--------+---------+------------+------+----------------+-------------+-----------------+--------------+-----------+----------------+---------------------+-------------
------+------------+--------------------+----------+----------------+---------+
1 row in set (0.01 sec)

mysql> SHOW TABLE STATUS LIKE 'Booking';
+---------+--------+---------+------------+------+----------------+-------------+-----------------+--------------+-----------+----------------+---------------------+-------------
------+------------+--------------------+----------+----------------+---------+
| Name    | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length | Index_length | Data_free | Auto_increment | Create_time         | Update_time
      | Check_time | Collation          | Checksum | Create_options | Comment |
+---------+--------+---------+------------+------+----------------+-------------+-----------------+--------------+-----------+----------------+---------------------+-------------
------+------------+--------------------+----------+----------------+---------+
| booking | InnoDB |      10 | Dynamic    |    5 |           3276 |       16384 |               0 |        16384 |         0 |              6 | 2024-09-01 11:33:40 | 2024-09-01 11:
32:27 | NULL       | utf8mb4_0900_ai_ci |     NULL |                |         |
+---------+--------+---------+------------+------+----------------+-------------+-----------------+--------------+-----------+----------------+---------------------+-------------
------+------------+--------------------+----------+----------------+---------+
1 row in set (0.00 sec)
```

Figure 2: Storage Results Before and After Indexing

0.7.2 Execution Time Performance

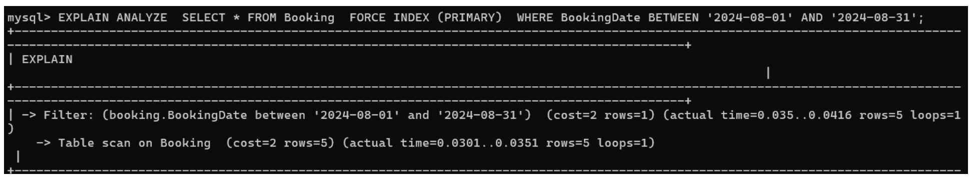
The execution time for queries utilizing the different indexes was measured using the following SQL command:

```
EXPLAIN ANALYZE
SELECT * FROM Booking
FORCE INDEX (PRIMARY)
WHERE BookingDate BETWEEN '2024-08-01' AND '2024-08-31';
```

For each index, the `FORCE INDEX` clause was modified to test performance against the primary key, clustering index on `BookingDate`, and the secondary index on `BookingHash`:

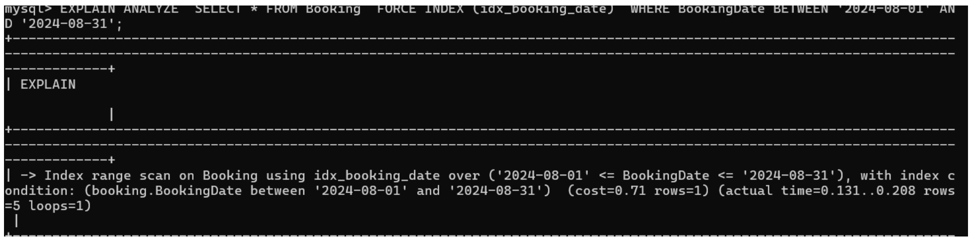
0.7.3 Analysis and Conclusion

- **Storage Efficiency:** The primary key clustering index typically consumes the least amount of additional storage since it is the natural ordering of the table. The clustering index on `BookingDate` and the secondary index on `BookingHash` may increase the storage requirements due to the maintenance of additional data structures.



```
mysql> EXPLAIN ANALYZE  SELECT * FROM Booking  FORCE INDEX (PRIMARY)  WHERE BookingDate BETWEEN '2024-08-01' AND '2024-08-31';
+------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                              |
+------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
| -> Filter: (booking.BookingDate between '2024-08-01' and '2024-08-31')  (cost=2 rows=1) (actual time=0.035..0.0416 rows=5 loops=1
)
    -> Table scan on Booking  (cost=2 rows=5) (actual time=0.0301..0.0351 rows=5 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------
```

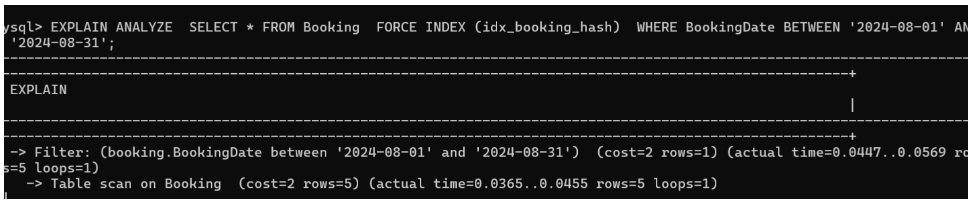
Figure 3: Execution Time with Primary Key Clustering Index



```
mysql> EXPLAIN ANALYZE  SELECT * FROM Booking  FORCE INDEX (idx_booking_date)  WHERE BookingDate BETWEEN '2024-08-01' AN
D '2024-08-31';
+------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
-------------+
| EXPLAIN

             |
+------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
-------------+
| -> Index range scan on Booking using idx_booking_date over ('2024-08-01' <= BookingDate <= '2024-08-31'), with index c
ondition: (booking.BookingDate between '2024-08-01' and '2024-08-31')  (cost=0.71 rows=1) (actual time=0.131..0.208 rows
=5 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------
```

Figure 4: Execution Time with Clustering Index on BookingDate



```
ysql> EXPLAIN ANALYZE  SELECT * FROM Booking  FORCE INDEX (idx_booking_hash)  WHERE BookingDate BETWEEN '2024-08-01' AN
 '2024-08-31';
------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------+
 EXPLAIN
                                                                                                               |
------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------+
 -> Filter: (booking.BookingDate between '2024-08-01' and '2024-08-31')  (cost=2 rows=1) (actual time=0.0447..0.0569 ro
s=5 loops=1)
   -> Table scan on Booking  (cost=2 rows=5) (actual time=0.0365..0.0455 rows=5 loops=1)
```

Figure 5: Execution Time with Secondary Index on BookingHash

- **Execution Time:** The primary key clustering index generally offers the best performance for queries that leverage the primary key. The clustering index on `BookingDate` is expected to perform well for range queries on booking dates, while the secondary index on `BookingHash` is optimized for hash-based lookups.

Conclusion: The choice between clustering and secondary indexing should be driven by the specific query patterns expected in the application. Clustering indexes are highly efficient for range queries, while secondary indexes provide targeted optimization for specific lookup keys. The detailed results and further analysis are documented in the accompanying Python notebook.

0.8 SQL Operations Using Python [15 points]

This section outlines the SQL operations performed using a Python notebook to manage the data in the `User` and `Booking` tables.

0.8.1 Adding Information on 5 New Users

The task of adding information about 5 new users to the `User` table was implemented in the Python notebook. The necessary SQL commands were executed via the Python interface to ensure the users were added correctly.

0.8.2 Preparing a Report on All Bookings Made in August 2024

A report on all bookings made during the month of August 2024 was generated using SQL queries within the Python notebook. The results were fetched and processed to provide insights into the booking activity for that period.

0.8.3 Removing User Profiles Created After 7PM on August 15, 2024

User profiles that were created after 7 PM on August 15, 2024, were removed from the `User` table. This operation was also executed in the Python notebook, ensuring that the specified profiles were deleted as required.

Conclusion: All SQL operations were successfully implemented in the Python notebook, providing a seamless way to interact with the MySQL database and manage the data effectively.