

***Database Project***

|  |  |
| --- | --- |
| ***Submitted By:*** | *Shamama Aslam (sp23-bcs-143)*  *Noor Fatima (Sp23-bcs-109)* |
| ***Submitted To:*** | *Dr. Abid Sohail Bhutta* |
| ***Course :*** | *Database Management System* |
| ***Date:*** | *21-12-2024* |
| ***Section:*** | *SP23-BCS-C* |

# ***Introduction:***

In our current era of automated systems with it being either software or hardware, it’s not advisable to be using manual system. Hostels without a management system are usually done manually. Registration forms verification to other data saving processes are done manually and most at times, they are written on paper. Thus a lot of repetitions can be avoided with an automated system. The drawbacks of existing systems lead to the design of a computerized system that will help reduce a lot of manual inputs. With this system in place, we can improve the efficiency of the system, thus overcome the drawbacks of the existing manual system. This system is designed in favor of the hostel management which helps them to save the records of the students about their rooms and other things. It helps them from the manual work from which it is very difficult to find the record of the students and the mess bills of the students, and the information of about the those ones who had left the hostel years before. This system gives an idea about how a student and fee details, room allocation, mess expenditure are maintained in a better way. The hostel management system will also contain special features like how many students are in a room, student’s id and free rooms or space available. The administration has a unique identity for each member as well as students details.

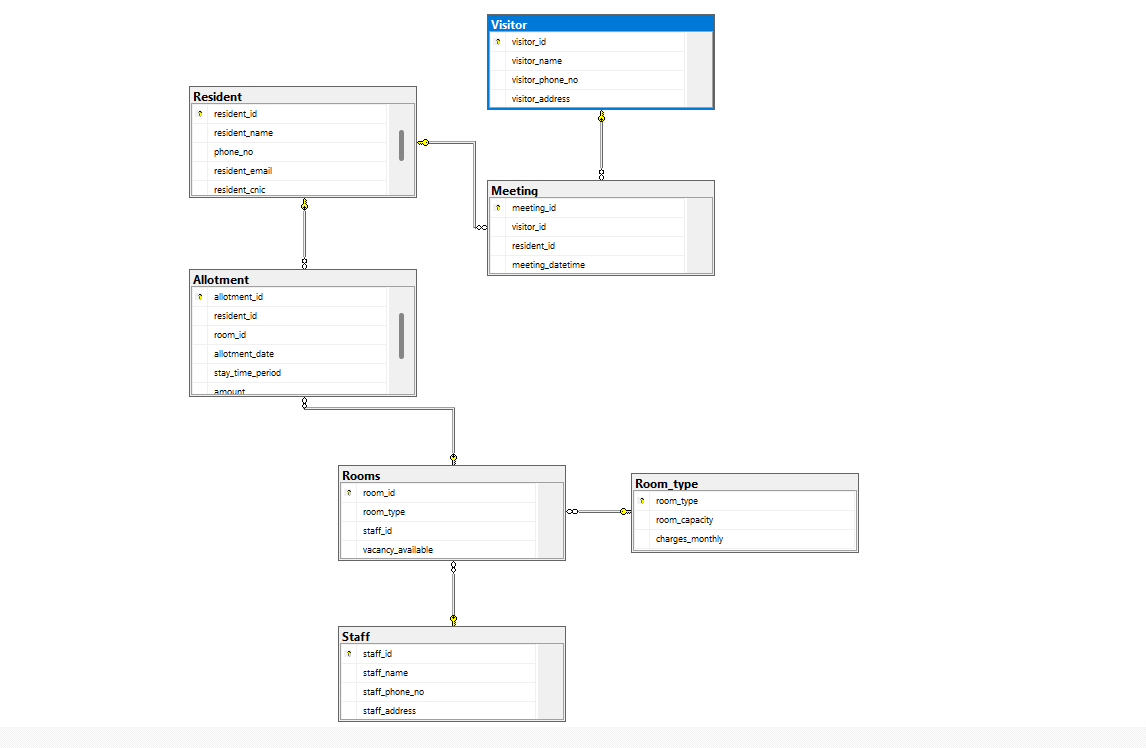
***Topic Description:***

In a hostel, students or residents are provided with accommodation, and the hostel management system is designed to handle various tasks related to room allocation, payments, and overall hostel operations. The system keeps track of available rooms, room types (single, double,) and their capacity. Students are assigned rooms based on their preferences and availability. Each student's private information, as well as the room reservation details, are noted down, and the system manages check-in and check-out processes efficiently.

The hostel management system also tracks the financial side by monitoring the payments of residents. It tracks the due dates of fee payments and generates receipts for payment. The system also tracks requests for maintaining rooms and common areas. The hostel staff, wardens, and housekeeping are also tracked through the system. The system tracks information like the schedule and attendance of staff.

Besides, the system records visitor's data to maintain security and proper documentation of guest entries. The management system for hostels helps the administrator to generate reports on occupancy, payments, and maintenance to streamline hostel operations. All these features make it easier to manage daily operations, improve efficiency, and give better experience for both residents and staff.

***Database Diagram:***

******

***Schema (SQL SERVER ):***

Create database Hostel

use Hostel

CREATE TABLE Resident (

resident\_id INT PRIMARY KEY, resident\_name VARCHAR(100), resident\_phone\_no VARCHAR(30),

resident\_email VARCHAR(100),

resident\_cnic VARCHAR(15)

);

CREATE TABLE Room\_type (

room\_type VARCHAR(20) PRIMARY KEY, -- Added PRIMARY KEY

room\_capacity INT, -- Corrected column name capitalization

charges\_monthly INT

);

CREATE TABLE Rooms (

room\_id INT PRIMARY KEY,

room\_type VARCHAR(20),

staff\_id INT,

vacancy\_available VARCHAR(20),

FOREIGN KEY (room\_type) REFERENCES Room\_type(room\_type),

FOREIGN KEY (staff\_id) REFERENCES Staff(staff\_id)

);

CREATE TABLE Allotment (

allotment\_id INT PRIMARY KEY,

resident\_id INT,

room\_id INT,

allotment\_date DATE,

stay\_time\_period INT,

amount int,

FOREIGN KEY (resident\_id) REFERENCES Resident(resident\_id),

FOREIGN KEY (room\_id) REFERENCES Rooms(room\_id)

);

CREATE TABLE Visitor (

visitor\_id INT PRIMARY KEY,

visitor\_name VARCHAR(100),

visitor\_phone\_no VARCHAR(20),

visitor\_address VARCHAR(100)

);

CREATE TABLE Meeting (

meeting\_id INT PRIMARY KEY,

visitor\_id INT,

resident\_id INT,

meeting\_datetime DATETIME,

FOREIGN KEY (resident\_id) REFERENCES Resident(resident\_id),

FOREIGN KEY (visitor\_id) REFERENCES Visitor(visitor\_id)

);

CREATE TABLE Staff (

staff\_id INT PRIMARY KEY,

staff\_name VARCHAR(100),

staff\_phone\_no VARCHAR(40),

staff\_address VARCHAR(100)

);

***Insertion Code By using Faker Python:***

import pyodbc

import random

from faker import Faker

faker = Faker()

# Define your connection details

server = 'NOORAY\\SQLEXPRESS' # or use your actual server name

database = 'Hostel'

# Create the connection string with Windows Authentication

connection\_string = f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};Trusted\_Connection=yes'

try:

# Connect to the database

connection = pyodbc.connect(connection\_string)

cursor = connection.cursor()

print("Connected to the database successfully!")

# Get the current maximum resident\_id from the Resident table

cursor.execute("SELECT MAX(resident\_id) FROM Resident")

max\_resident\_id = cursor.fetchone()[0]

if max\_resident\_id is None:

max\_resident\_id = 0 # If the table is empty, start from 1

print(f"Max Resident ID: {max\_resident\_id}")

# Insert Resident Data (with incremented resident\_id)

num\_residents = 100

print("-- Inserting Resident Data")

for i in range(1, num\_residents + 1):

max\_resident\_id += 1 # Increment resident\_id

resident\_query = """

INSERT INTO Resident (resident\_id, resident\_name, resident\_phone\_no, resident\_email, resident\_cnic)

VALUES (?, ?, ?, ?, ?);

"""

cursor.execute(resident\_query, (max\_resident\_id, faker.name(), faker.phone\_number(), faker.email(), faker.ssn()))

print(f"Inserted Resident {max\_resident\_id}")

# Get the current maximum staff\_id from the Staff table

cursor.execute("SELECT MAX(staff\_id) FROM Staff")

max\_staff\_id = cursor.fetchone()[0]

if max\_staff\_id is None:

max\_staff\_id = 0 # If the table is empty, start from 1

print(f"Max Staff ID: {max\_staff\_id}")

# Insert Staff Data (with incremented staff\_id)

num\_staff = 20

print("-- Inserting Staff Data")

for i in range(1, num\_staff + 1):

max\_staff\_id += 1 # Increment staff\_id

staff\_query = """

INSERT INTO Staff (staff\_id, staff\_name, staff\_phone\_no, staff\_address)

VALUES (?, ?, ?, ?);

"""

cursor.execute(staff\_query, (max\_staff\_id, faker.name(), faker.phone\_number(), faker.address()))

print(f"Inserted Staff {max\_staff\_id}")

# Get the current maximum room\_id from the Rooms table

cursor.execute("SELECT MAX(room\_id) FROM Rooms")

max\_room\_id = cursor.fetchone()[0]

if max\_room\_id is None:

max\_room\_id = 0 # If the table is empty, start from 1

print(f"Max Room ID: {max\_room\_id}")

# Insert Room Data (with incremented room\_id)

num\_rooms = 50

print("-- Inserting Room Data")

room\_types = [("Single", 1, 5000), ("Double", 2, 8000), ("Triple", 3, 12000)]

for i in range(1, num\_rooms + 1):

max\_room\_id += 1 # Increment room\_id

room\_type = random.choice(room\_types)[0]

staff\_id = random.randint(1, num\_staff)

vacancy = random.choice(["Available", "Not Available"])

room\_query = """

INSERT INTO Rooms (room\_id, room\_type, staff\_id, vacancy\_available)

VALUES (?, ?, ?, ?);

"""

cursor.execute(room\_query, (max\_room\_id, room\_type, staff\_id, vacancy))

print(f"Inserted Room {max\_room\_id}")

# Get the current maximum allotment\_id from the Allotment table

cursor.execute("SELECT MAX(allotment\_id) FROM Allotment")

max\_allotment\_id = cursor.fetchone()[0]

if max\_allotment\_id is None:

max\_allotment\_id = 0 # If the table is empty, start from 1

print(f"Max Allotment ID: {max\_allotment\_id}")

# Insert Allotment Data (with incremented allotment\_id)

num\_allotments = 80

print("-- Inserting Allotment Data")

for i in range(1, num\_allotments + 1):

max\_allotment\_id += 1 # Increment allotment\_id

resident\_id = random.randint(1, num\_residents)

room\_id = random.randint(1, num\_rooms)

stay\_time\_period = random.randint(1, 12)

allotment\_date = faker.date\_this\_year()

charges\_monthly = random.choice(room\_types)[2]

amount = charges\_monthly \* stay\_time\_period

allotment\_query = """

INSERT INTO Allotment (allotment\_id, resident\_id, room\_id, allotment\_date, stay\_time\_period, amount)

VALUES (?, ?, ?, ?, ?, ?);

"""

cursor.execute(allotment\_query, (max\_allotment\_id, resident\_id, room\_id, allotment\_date, stay\_time\_period, amount))

print(f"Inserted Allotment {max\_allotment\_id}")

# Commit the transaction

connection.commit()

print("Data inserted successfully!")

except Exception as e:

print(f"An error occurred: {e}")

finally:

if 'connection' in locals() and connection:

connection.close()

print("Database connection closed.")

RESULTS :

**REPORTS :**

***Procedures*:**

1. **MonthlyOccupancyReport:**

The stored procedure calculates the monthly occupancy report for rooms in a hotel or accommodation setup. It takes two input parameters: @month and @year, which define the specific month and year for which the occupancy report is generated.

* **Query :**

CREATE PROCEDURE sp\_MonthlyOccupancyReport

@month INT,

@year INT

AS

BEGIN

SELECT

r.room\_type,

COUNT(a.room\_id) as occupied\_rooms,

rt.room\_capacity \* COUNT(DISTINCT r.room\_id) as total\_capacity,

CAST(COUNT(a.room\_id) \* 100.0 / COUNT(DISTINCT r.room\_id) AS DECIMAL(5,2)) as occupancy\_rate

FROM Rooms r

LEFT JOIN Allotment a ON r.room\_id = a.room\_id

JOIN Room\_type rt ON r.room\_type = rt.room\_type

WHERE MONTH(a.allotment\_date) = @month

AND YEAR(a.allotment\_date) = @year

GROUP BY r.room\_type, rt.room\_capacity;

END;

Execution:

EXEC sp\_MonthlyOccupancyReport @month=12, @year=2024;

**2- RevenueByRoomType:**

The stored procedure is designed to calculate the total revenue, total allotments, and average revenue per allotment for each room type within a specified date range. The procedure takes two input parameters: @start\_date and @end\_date, which define the date range for which the report will be generated.

* **Query**:

CREATE PROCEDURE sp\_RevenueByRoomType

@start\_date DATE,

@end\_date DATE

AS

BEGIN

SELECT

r.room\_type,

COUNT(DISTINCT a.allotment\_id) as total\_allotments,

SUM(a.amount) as total\_revenue,

AVG(a.amount) as avg\_revenue\_per\_allotment

FROM Allotment a

JOIN Rooms r ON a.room\_id = r.room\_id

WHERE a.allotment\_date BETWEEN @start\_date AND @end\_date

GROUP BY r.room\_type

ORDER BY total\_revenue DESC;

END;

EXECUTION

EXEC sp\_RevenueByRoomType @start\_date = '2024-01-01', @end\_date = '2024-12-31';

**3- StaffPerformanceReport:**

The stored procedure is designed to generate a report that evaluates the performance of hotel staff based on the number of rooms managed, the total allotments (bookings) handled, and the revenue generated by those allotments.

* **QUERY**:

cREATE PROCEDURE sp\_StaffPerformanceReport

AS

BEGIN

SELECT

s.staff\_id,

s.staff\_name,

COUNT(DISTINCT r.room\_id) as rooms\_managed,

COUNT(DISTINCT a.allotment\_id) as total\_allotments,

SUM(a.amount) as revenue\_generated

FROM Staff s

LEFT JOIN Rooms r ON s.staff\_id = r.staff\_id

LEFT JOIN Allotment a ON r.room\_id = a.room\_id

GROUP BY s.staff\_id, s.staff\_name;

END;

EXECUTION:

EXEC sp\_StaffPerformanceReport

***VIEWS***

**1-ResidentStayAnalysis**:

The view is designed to analyze the stay details of residents in a hotel or accommodation setup. It provides a comprehensive report about each resident’s stay, total stay days, total money spent, and the date of their most recent stay. This view combines data from the Resident table (which holds resident details) and the Allotment table (which tracks individual stays and bookings).

* **QUERY**:

CREATE VIEW vw\_ResidentStayAnalysis AS

SELECT

res.resident\_id,

res.resident\_name,

COUNT(DISTINCT a.allotment\_id) as total\_stays,

SUM(a.stay\_time\_period) as total\_stay\_days,

SUM(a.amount) as total\_spent,

MAX(a.allotment\_date) as last\_stay\_date

FROM Resident res

LEFT JOIN Allotment a ON res.resident\_id = a.resident\_id

GROUP BY res.resident\_id, res.resident\_name;

Execution:

**2- MonthlyRevenue**

The view is designed to provide a monthly revenue report for each room type. It aggregates the total revenue, the number of allotments (bookings), and the average revenue per allotment for each room type within each month. This view uses the Allotment table to track the bookings and the Rooms table to identify the type of room booked

* **QUERY**:

CREATE VIEW vw\_MonthlyRevenue AS

SELECT

FORMAT(a.allotment\_date, 'yyyy-MM') AS month,

r.room\_type,

COUNT(DISTINCT a.allotment\_id) AS total\_allotments,

SUM(a.amount) AS total\_revenue,

AVG(a.amount) AS avg\_revenue\_per\_allotment

FROM Allotment a

JOIN Rooms r ON a.room\_id = r.room\_id

GROUP BY FORMAT(a.allotment\_date, 'yyyy-MM'), r.room\_type;

EXECUTION:

**3-VisitorActivity:**

The view is designed to provide a summary of the activity of visitors who meet residents in a hotel or accommodation setting. It aggregates information such as the total number of meetings a visitor has attended, the first and last meeting dates, and the resident they visited.

* **QUERY**:

CREATE VIEW vw\_VisitorActivity AS

SELECT

v.visitor\_id,

v.visitor\_name,

res.resident\_name AS visited\_resident,

COUNT(m.meeting\_id) AS meeting\_count,

MAX(m.meeting\_datetime) AS last\_visit,

MIN(m.meeting\_datetime) AS first\_visit

FROM Visitor v

JOIN Meeting m ON v.visitor\_id = m.visitor\_id

JOIN Resident res ON m.resident\_id = res.resident\_id

GROUP BY v.visitor\_id, v.visitor\_name, res.resident\_name;

EXECUTION:

Select \* from vw\_VisitorActivity

***Materialized View***

1- MonthlyRevenue

The view is designed to provide a detailed report on the monthly revenue generated from residents, broken down by room type. It calculates the total revenue, the number of residents, and the average revenue per resident for each month and room type. The ROLLUP clause adds summary rows, which include aggregates for each month and overall totals across all months.

QUERY :

CREATE VIEW vw\_MonthlyRevenueDenorm AS

SELECT

YEAR(allotment\_date) as year,

MONTH(allotment\_date) as month,

room\_type,

COUNT(DISTINCT resident\_id) as total\_residents,

SUM(total\_amount) as total\_revenue,

AVG(total\_amount) as avg\_revenue\_per\_resident

FROM resident\_denormalized

GROUP BY YEAR(allotment\_date), MONTH(allotment\_date), room\_type

WITH ROLLUP;

SELECT \* FROM vw\_MonthlyRevenueDenorm;

***History table***

CREATE TABLE Resident\_History (

history\_id INT IDENTITY(1,1) PRIMARY KEY,

resident\_id INT,

resident\_name VARCHAR(100),

resident\_phone\_no VARCHAR(30),

resident\_email VARCHAR(100),

resident\_cnic VARCHAR(15),

action\_type VARCHAR(10),

action\_date DATETIME,

action\_by VARCHAR(100)

);

CREATE TABLE Allotment\_History (

history\_id INT IDENTITY(1,1) PRIMARY KEY,

allotment\_id INT,

resident\_id INT,

room\_id INT,

allotment\_date DATE,

stay\_time\_period INT,

amount INT,

action\_type VARCHAR(10),

action\_date DATETIME,

action\_by VARCHAR(100))

Procedures for insertion/updating/deletion

* **AddNewResident (INSERTION)**

he sp\_AddNewResident stored procedure is designed to add a new resident to the system and assign them to a room. It performs several actions in a single transaction:

1-Adds a new resident to the Resident table.

2-Creates an allotment for the resident by assigning them to a room in the Allotment table.

3-Updates the room status in the Rooms table to mark it as "No" for available vacancy, indicating that the room is now occupied.

The procedure ensures that all actions are executed as a single transaction. If an error occurs at any point during the execution, all changes are rolled back, maintaining the integrity of the data.

**QUERY**:

CREATE PROCEDURE sp\_AddNewResident

@name VARCHAR(100),

@phone VARCHAR(30),

@email VARCHAR(100),

@room\_id INT

AS

BEGIN TRY

BEGIN TRANSACTION;

INSERT INTO Resident (resident\_name, resident\_phone\_no, resident\_email)

VALUES (@name, @phone, @email)

INSERT INTO Allotment (resident\_id, room\_id, allotment\_date)

VALUES (SCOPE\_IDENTITY(), @room\_id, GETDATE());

UPDATE Rooms SET vacancy\_available = 'No'

WHERE room\_id = @room\_id;

COMMIT;

END TRY

BEGIN CATCH

ROLLBACK;

PRINT 'Error: ' + ERROR\_MESSAGE();

END CATCH;

**2-Change room (Updation):**

The sp\_ChangeRoom stored procedure allows for the relocation of a resident from their current room to a new room. It performs the following actions:

1-Updates the availability of the current room: It marks the resident's current room as available (vacancy set to "Yes").

2-Updates the allotment record: It updates the Allotment table to reflect the new room assignment for the resident.

3-Updates the availability of the new room: It marks the new room as occupied (vacancy set to "No").

**QUERY**:

CREATE PROCEDURE sp\_ChangeRoom

@resident\_id INT,

@new\_room\_id INT

AS

BEGIN TRY

BEGIN TRANSACTION;

UPDATE Rooms

SET vacancy\_available = 'Yes'

WHERE room\_id = (

SELECT room\_id

FROM Allotment

WHERE resident\_id = @resident\_id

);

UPDATE Allotment

SET room\_id = @new\_room\_id

WHERE resident\_id = @resident\_id;

UPDATE Rooms

SET vacancy\_available = 'No'

WHERE room\_id = @new\_room\_id;

COMMIT;

END TRY

BEGIN CATCH

ROLLBACK;

PRINT 'Error: ' + ERROR\_MESSAGE();

END CATCH;

3- Resident checkout :

The sp\_CheckOutResident stored procedure is designed to handle the checkout process for a resident, which involves the following actions:

1-Freeing up the room: It updates the room's availability status to mark the room as vacant (set to 'Yes').

2-Removing the allotment record: It deletes the resident's allotment entry in the Allotment table, indicating that the resident is no longer assigned to the room.

3-Deleting the resident record: It deletes the resident's record from the Resident table, effectively removing them from the system.

**QUERY**:

CREATE PROCEDURE sp\_CheckOutResident

@resident\_id INT

AS

BEGIN TRY

BEGIN TRANSACTION;

UPDATE Rooms

SET vacancy\_available = 'Yes'

WHERE room\_id = (

SELECT room\_id

FROM Allotment

WHERE resident\_id = @resident\_id

);

DELETE FROM Allotment

WHERE resident\_id = @resident\_id;

DELETE FROM Resident

WHERE resident\_id = @resident\_id;

COMMIT;

END TRY

BEGIN CATCH

ROLLBACK;

PRINT 'Error: ' + ERROR\_MESSAGE();

END CATCH;