Hotel Management System

Siri Duggineni
UBIT: siridugg
Person Number: 50547810

Abstract—**Traditional hotel management often relies on spreadsheets for storing guest data, room inventory, and reservations. However, spreadsheet-based systems pose challenges in data integrity, scalability, information retrieval, and security. This project aims to address these limitations by developing a relational database for efficient hotel management.**

I. Introduction

Effective hotel management is essential for delivering exceptional guest experiences and maximizing profitability. However, many hotels, particularly smaller businesses, struggle with the limitations and inefficiencies of spreadsheet-based data management. This outdated approach leads to errors in bookings, difficulties in tracking guest preferences, and a lack of actionable insights, hindering a hotel's ability to optimize its operations. This project introduces a relational database system designed to streamline hotel management, enhance data integrity, and facilitate informed decision-making.

A. *Problem Statement*

The hospitality industry relies heavily on the efficiency of data management processes. In smaller hotels, manual systems based on spreadsheets are often the default. These systems become increasingly inadequate as the scale of operations grows. Spreadsheets raise concerns regarding data integrity due to manual data entry, limited scalability in accommodating large datasets, inefficient extraction of actionable insights, and potential security risks for sensitive guest information.

Excel spreadsheets, while offering basic data organization, possess inherent limitations that render them unsuitable for managing the growing complexity, volume, and security demands of hotel operations. These shortcomings include vulnerability to data inconsistencies due to manual entry, performance degradation with large datasets, limited analytical capabilities, and inadequate security features. In sharp contrast, a relational database system addresses these challenges by enforcing data integrity through defined data types and relationships, providing scalability to accommodate growth, enabling powerful querying for extracting insights, and offering robust security measures to protect sensitive guest information and ensure regulatory compliance. For hotels prioritizing efficiency, data-driven decision-making, and the safeguarding of guest data, a purpose-built database solution is essential.

B. *Background*

Hotel management systems often rely on spreadsheets, which are inadequate for handling the growing complexity, volume, and security requirements of hotel data. Manual data entry increases error risk, while limited analysis and security features in spreadsheets hinder data-driven optimization and raise compliance concerns.

C. *Problem Significance*

Operational Inefficiencies: Data inconsistencies and manual processes cause delays and bottlenecks.
Missed Revenue: Lack of centralized data obscures trends, hindering optimal pricing and resource allocation decisions.
Compromised Guest Experience: Inaccurate records and slow processes reduce service quality and personalization, hindering guest satisfaction.
Security Risks: Spreadsheets offer minimal data protection, increasing the risk of breaches and non-compliance with regulations.

D. *Project Significance*

This project addresses these challenges by developing a relational database system tailored for hotel management. Key contributions include:
Centralized Data: A single, reliable source of hotel data ensures consistency and accessibility.
Analytics for Decision-Making: Tools for reporting and analyzing key metrics will inform revenue management, marketing, and operations strategies.
Enhanced Guest Experience: Detailed guest profiles enable personalized service and foster improved customer satisfaction.
Robust Security: Database security features safeguard sensitive guest information and ensure regulatory compliance.

II. Potential Contribution

A. *Enhanced Efficiency and Operations:*

Streamlining Front Desk Operations: The database will automate tasks like reservations, check-in/check-out, and room assignments, freeing staff for guest interaction and value-added services.
Real-time Data Availability: Staff across departments (housekeeping, sales) will have instant access to accurate data, leading to faster response times and improved decision-making.
Improved Reporting and Analytics: Powerful reporting tools

will enable hotel management to analyze trends, track performance metrics, and identify opportunities for cost optimization and revenue growth.

B. Elevated Guest Experience:

Personalized Service: Guest preferences and history can be stored, allowing staff to anticipate needs and tailor services accordingly.
Faster Check-in/Check-out: Online check-in and self-service kiosks integrated with the database can streamline arrival and departure processes.
Improved Communication: Automated guest communication and feedback mechanisms can enhance engagement and satisfaction.

C. Data-Driven Decision Making:

Dynamic Pricing Strategies: Real-time occupancy data allows dynamic pricing adjustments to maximize revenue based on demand.
Targeted Marketing Campaigns: Guest segmentation based on preferences and demographics from the database enables personalized marketing campaigns for higher conversion rates.
Resource Allocation: Data analysis will provide insights into resource optimization, such as staffing schedules based on historical occupancy patterns.

D. Crucial Nature of This Contribution:

Competition in the hospitality industry is fierce. A robust and well-maintained hotel management database offers a crucial advantage by:
Enhancing Efficiency: Boosting productivity and reducing operational costs.
Elevating Guest Experience: Increasing guest satisfaction and loyalty, leading to repeat business and positive online reviews.
Enabling Data-Driven Decisions: Providing valuable insights for strategic planning, marketing strategies, and revenue management.

III. Target User

The hotel management database system is designed for use by various hotel personnel with differing roles and needs:

Front Desk Staff:
Tasks: Check-in/check-out, room assignments, booking reservations, updating guest profiles, accessing guest preferences, handling inquiries and requests, generating room status reports.
Importance: Front desk staff are the primary users, interacting with the database constantly to manage the core hotel operations.
Hotel Management:
Tasks: Analyzing occupancy trends, generating revenue and performance reports, pricing adjustments, marketing and sales analysis, setting strategic directions, staff scheduling.
Importance: Management relies heavily on the database for accurate data to make informed decisions regarding pricing, promotions, staffing needs, and resource allocation.
Housekeeping:
Tasks: Updating room cleaning status (vacant, occupied, under cleaning), scheduling maintenance tasks, tracking inventory (linens, supplies), reporting maintenance needs.
Importance: Housekeeping requires real-time data on room status to optimize cleaning schedules and ensure a smooth guest experience.
Sales and Marketing:
Tasks: Accessing guest history and preferences, targeting personalized offers and promotions, tracking campaign effectiveness, identifying market trends.
Importance: The database serves as a resource for targeted marketing campaigns and developing loyalty programs to enhance customer relationships.
Guests (Indirectly):
Tasks: Guests might indirectly interact with the database through features like online booking portals, room service ordering systems, or self-service kiosks integrated with the database.
Importance: The guest experience is greatly influenced by the accuracy of data stored – available rooms, correct billing, and the ability to seamlessly personalize their stay.
Database Administrator
A dedicated IT professional handles database administration, ensuring optimal performance, security, data integrity, and implementing software updates.
Tasks: Ensuring database security (access controls, backups, disaster recovery) Performance optimization (tuning queries, indexes) Resolving database issues and errors Implementing software updates and schema changes as needed.

A. Real-Life Scenario: A Busy Holiday Weekend

Front Desk Overwhelmed: During peak season, the front desk handles a surge in new bookings and check-ins. They rely on the database to:

- Quickly find available rooms matching guest preferences.
- Process payments securely.
- Register new guests, capturing their details accurately.

Housekeeping Efficiency: Housekeeping needs real-time updates on room status from the database to prioritize cleaning tasks, ensuring rooms are ready for incoming guests.

Management's Strategic Decision: Hotel management analyzes historical occupancy data from the database to compare this holiday season with previous years. This allows them to:

- Adjust pricing dynamically for remaining rooms.
- Plan staffing for upcoming high-traffic periods.

Marketing's Post-Holiday Campaign: The marketing team accesses guest data from the database to segment guests based on demographics and stay preferences. They can target personalized offers to encourage return visits.

Guest Convenience: A web portal connected to the database offers:

- Guests can make online bookings with real-time room availability.

IV. E/R DIAGRAM

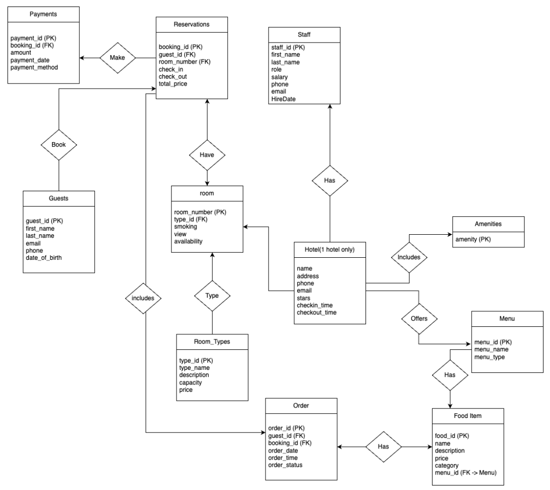


Fig. 1. Entity Relationship Diagram.

V. DATASET GENERATION

A realistic dataset was programmatically generated using a Python script and the Faker library to populate the hotel management database. The dataset includes entities such as Hotel, Rooms, Guests, Reservations, Staff, Amenities, and Reviews.

Hotel

- **hotel_id** (Primary Key): Unique identifier for the hotel. (INT)
- **name**: Name of the hotel. (VARCHAR)
- **address**: Hotel's physical address. (VARCHAR)
- **phone**: Hotel's phone number. (VARCHAR)
- **email**: Hotel's email address. (VARCHAR)
- **stars**: Hotel's star rating (e.g., 3, 4, 5). (INT)
- **checkin_time**: Check-in time for guests. (TIME)
- **checkout_time**: Check-out time for guests. (TIME)

Guest

- **guest_id** (Primary Key): Unique identifier for the guest. (INT)
- **first_name**: Guest's first name. (VARCHAR)
- **last_name**: Guest's last name. (VARCHAR)
- **email**: Guest's email address. (VARCHAR)
- **phone**: Guest's phone number. (VARCHAR)
- **date_of_birth**: Guest's date of birth. (DATE)

Reservation (Booking)

- **booking_id** (Primary Key): Unique identifier for the reservation. (INT)
- **guest_id** (Foreign Key): References the Guest table. (INT) - ON DELETE CASCADE
- **room_number** (Foreign Key): References the Room table. (INT) - ON DELETE CASCADE
- **check_in**: Date of check-in for the reservation. (DATE)
- **check_out**: Date of check-out for the reservation. (DATE)
- **total_price**: Total price calculated for the reservation. (DECIMAL)

Room

- **room_number** (Primary Key): Unique identifier for the room. (INT)
- **type_id** (Foreign Key): References the Room Type table. (INT)
- **smoking**: Indicates if the room is smoking or non-smoking. (BOOLEAN)
- **view**: Description of the room view (e.g., city, ocean). (VARCHAR)
- **availability**: Indicates room availability (e.g., Available, Occupied). (VARCHAR)

Room Type

- **type_id** (Primary Key): Unique identifier for the room type. (INT)
- **type_name**: Name of the room type (e.g., Standard, Suite). (VARCHAR)
- **description**: Description of the room type amenities. (VARCHAR)
- **capacity**: Maximum number of guests allowed in the room type. (INT)
- **price**: Base price per night for the room type. (DECIMAL)

Staff

- **staff_id** (Primary Key): Unique identifier for the staff member. (INT)
- **first_name**: Staff member's first name. (VARCHAR)
- **last_name**: Staff member's last name. (VARCHAR)
- **role**: Staff member's role (e.g., Front Desk, Housekeeping). (VARCHAR)
- **hotel_id** (Foreign Key): References the Hotel table. (INT) - ON DELETE RESTRICT

Amenity

- **amenity** (Primary Key): Name of the amenity (e.g., Pool, Spa, Gym, WiFi). (VARCHAR)

Menu

- **menu_id** (Primary Key): Unique identifier for the menu. (INT)
- **menu_name**: Name of the menu (e.g., Breakfast Menu, Dinner Menu). (VARCHAR)
- **menu_type**: Type of menu (e.g., Breakfast, Lunch, Dinner). (VARCHAR)

Food Item

- **food_id** (Primary Key): Unique identifier for the food item. (INT)
- **name**: Name of the food item. (VARCHAR)
- **description**: Description of the food item. (VARCHAR)
- **price**: Price of the food item. (DECIMAL)
- **category**: Category of the food item (e.g., Appetizer, Main Course). (VARCHAR)
- **menu_id** (Foreign Key): References the Menu table. (INT) – ON DELETE CASCADE

Order

- **order_id** (Primary Key): Unique identifier for the order. (INT)
- **guest_id** (Foreign Key): References the Guest table. (INT) – ON DELETE CASCADE
- **reservation_id** (Foreign Key, Optional): References the Reservation table. (INT) – ON DELETE RESTRICT
- **order_date**: Date the order was placed. (DATE)
- **order_time**: Time the order was placed. (TIME)
- **order_status**: Status of the order (e.g., Placed, Preparing, Delivered). (VARCHAR)

Order Items (Junction Table)

- **order_id** (Foreign Key): References the Order table. (INT)
- **food_id** (Foreign Key): References the Food Item table. (INT)
- **quantity**: Quantity of the food item ordered. (INT)
- **price**: Price of the food item (snapshot at the time of order). (DECIMAL)

Composite Primary Key: The combination of (*order_id*, *food_id*) serves as the primary key for this table. Both BCNF enforces stricter data integrity by eliminating partial dependencies. This reduces data redundancy and the potential for anomalies during data manipulation.

VII. Task 5: Handling Larger Datasets

In our project, we encountered challenges related to handling larger datasets, which impacted query performance and overall system responsiveness. Challenges as the size of our dataset increased, several challenges emerged:
Slow Query Performance: Executing queries on larger datasets resulted in prolonged query execution times, leading to degraded application performance.
Memory Overhead: Loading large datasets into memory caused increased memory consumption, which could potentially result in out-of-memory errors and hinder system scalability.
Indexing Limitations: Insufficient or improper indexing negatively impacted query performance, especially for complex queries or those involving multiple joins.
Strategies Adopted: To overcome these challenges, we implemented various indexing concepts: Primary Keys: Defined primary keys on tables to ensure efficient access to individual records, thereby enhancing query performance for single-record operations

TABLE I
Analysis of Relations for BCNF

Relation	Functional Dependencies (FDs)	BCNF Violation	Notes
Hotel	$hotel\_id \rightarrow \{name, address, phone, email, stars, checkin\_time, checkout\_time\}$	No	This is a simple key dependency.
Guest	$guest\_id \rightarrow \{first\_name, last\_name, email, phone, date\_of\_birth\}$	No	Similar to Hotel, a simple key dependency.
Room	$room\_number \rightarrow \{type\_id, smoking, view, availability\}$	No	Key dependency on the primary key.
Room Type	$type\_id \rightarrow \{type\_name, description, capacity, price\}$	No	Similar to Room, a simple key dependency.
Reservation	$booking\_id \rightarrow \{guest\_id, room\_number, check\_in, check\_out, total\_price\}$	Potential	FDs: $guest\_id \rightarrow \{room\_number\}$ (partial dependency)
Staff	$staff\_id \rightarrow \{first\_name, last\_name, role\}$	No	Key dependency on the primary key.
Amenity	$amenity \rightarrow$ (unique name)	No	Amenity name likely serves as a unique identifier.
Menu	$menu\_id \rightarrow \{menu\_name, menu\_type\}$	No	Key dependency on the primary key.
Food Item	$food\_id \rightarrow \{name, description, price, category\}$	No	Key dependency on the primary key.
Order	$order\_id \rightarrow \{guest\_id, reservation\_id \text{ (optional)}, order\_date, order\_time, order\_status\}$	Potential	FDs: $guest\_id \rightarrow \{reservation\_id\}$ (partial dependency)
Order Items	$\{order\_id, food\_id\} \rightarrow quantity, price$	No	Composite key dependency $(order\_id, food\_id)$

Reservation and Order have FDs where a non-key attribute (gues_id) determines another non-key attribute (room_number or reservation_id, respectively). These are partial dependencies that violate BCNF.

VI. Decomposition Strategies:

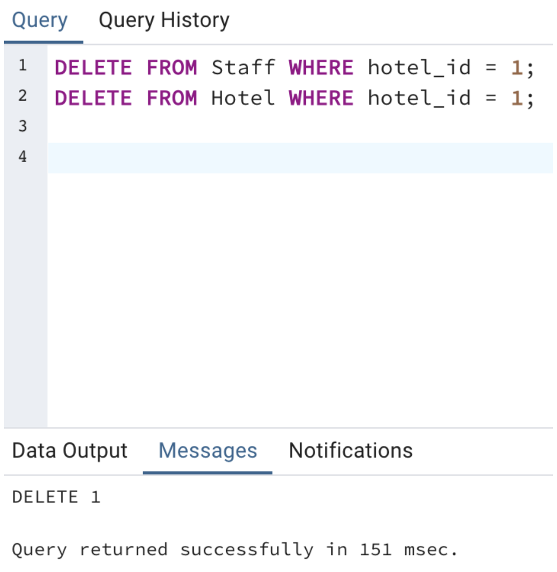
Reservation: The partial dependency in Reservation is guest_id $\rightarrow$ room_number. This indicates that the room_number can be determined solely by the guest_id, even though room_number is not part of the primary key of Reservation.
Decompose Reservation into two relations: Reservation (booking_id, guest_id, check_in, check_out, total_price). Guest Room (guest_id, room_number). This introduces a new relation to capture the many-to-many relationship between guests and rooms for reservations.
Order: The partial dependency in Order is guest_id $\rightarrow$ reservation_id. Similar to Reservation, reservation_id can be determined by guest_id, although it's not part of the primary key of Order.
Decompose Order into two relations: Order (order_id, guest_id, order_date, order_time, order_status). Guest Reservation (guest_id, reservation_id). Similar to Reservation decomposition, this captures the optional relationship with reservations.

To overcome these challenges, we implemented various indexing concepts: Primary Keys: Defined primary keys on tables to ensure efficient access to individual records, thereby enhancing query performance for single-record operations.
Foreign Keys: Established foreign key relationships between tables to facilitate faster query execution when joining related tables, improving overall query performance.
Composite Indexes: Created composite indexes on columns frequently used together in queries, optimizing query execution by providing faster access paths to data subsets.
Partial Indexes: Implemented partial indexes for commonly used query conditions, reducing index size and improving query performance by excluding unnecessary data from indexing.
Query Optimization: Optimized queries by rewriting them to use appropriate join strategies, avoiding unnecessary subqueries, and incorporating query hints to guide the query optimizer.
By implementing these indexing strategies, we observed significant improvements in query performance and system responsiveness. Queries that previously exhibited slow performance now execute more efficiently, resulting in faster response times for end users.

VIII. SQL Queries

A. DELETE Operations

DELETE FROM Staff WHERE hotel_id = 1;
DELETE FROM Hotel WHERE hotel_id = 1;
Here, ”DELETE FROM Staff WHERE hotel_id = 1;”, removes all staff records associated with a specific hotel identified by the ID 1. The second query, ”DELETE FROM Hotel WHERE hotel_id = 1;”, deletes the record of a particular hotel identified by the ID 1 from the database. DELETE FROM

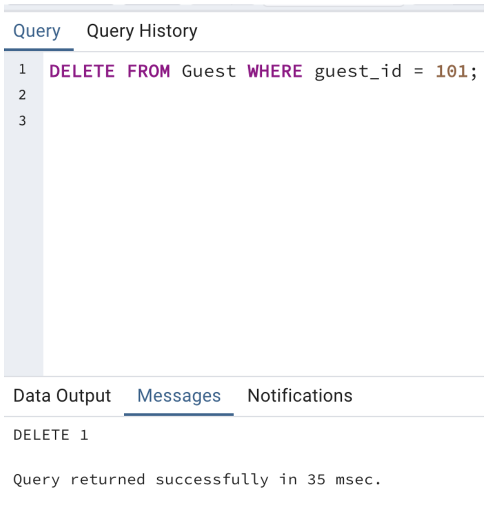


```
DELETE FROM Staff WHERE hotel_id = 1;
DELETE FROM Hotel WHERE hotel_id = 1;
```

```
DELETE 1

Query returned successfully in 151 msec.
```

Guest WHERE gues_id = 101;
The query ”DELETE FROM Guest WHERE guest_id = 101;” removes the record of a specific guest identified by the ID 101 from the ”Guest” table.



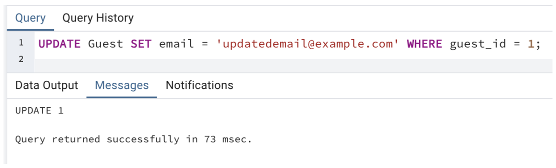
```
DELETE FROM Guest WHERE guest_id = 101;
```

```
DELETE 1

Query returned successfully in 35 msec.
```

B. UPDATE Operations

UPDATE Guest SET email = ’updatedemail@example.com’ WHERE guest_id = 1;

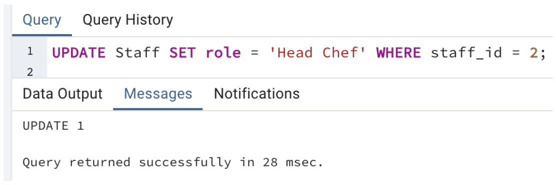


```
UPDATE Guest SET email = 'updatedemail@example.com' WHERE guest_id = 1;
```

```
UPDATE 1

Query returned successfully in 73 msec.
```

UPDATE Staff SET role = ’Head Chef’ WHERE staff_id = 2;



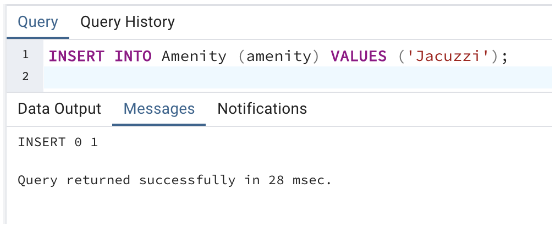
```
UPDATE Staff SET role = 'Head Chef' WHERE staff_id = 2;
```

```
UPDATE 1

Query returned successfully in 28 msec.
```

C. INSERT Operations

INSERT INTO Amenity (amenity) VALUES (’Jacuzzi’);

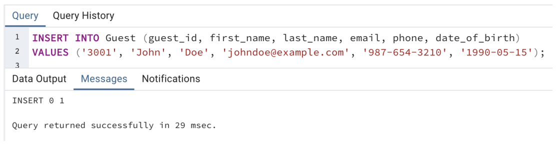


```
INSERT INTO Amenity (amenity) VALUES ('Jacuzzi');
```

```
INSERT 0 1

Query returned successfully in 28 msec.
```

INSERT INTO Guest (guest_id, first_name, last_name, email, phone, date_of_birth) VALUES (’3001’, ’John’, ’Doe’, ’johndoe@example.com’, ’987-654-3210’, ’1990-05-15’);



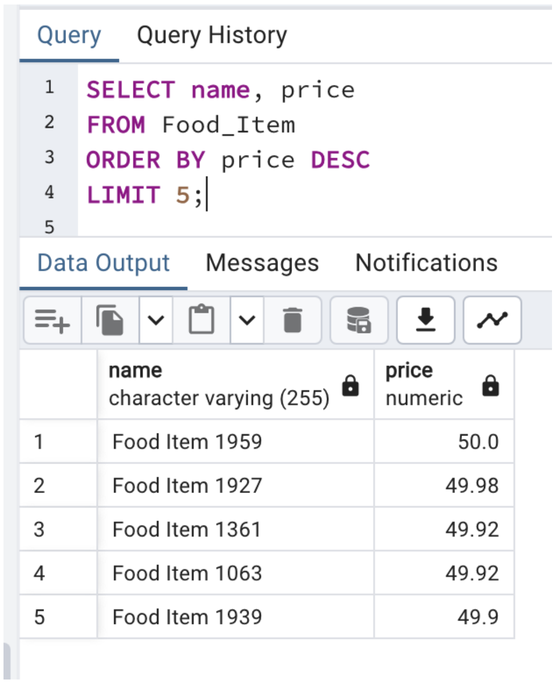
```
INSERT INTO Guest (guest_id, first_name, last_name, email, phone, date_of_birth)
VALUES ('3001', 'John', 'Doe', 'johndoe@example.com', '987-654-3210', '1990-05-15');
```

```
INSERT 0 1

Query returned successfully in 29 msec.
```

D. SELECT Operations

SELECT category, SUM(price) AS total_revenue
FROM Food_Item
GROUP BY category;

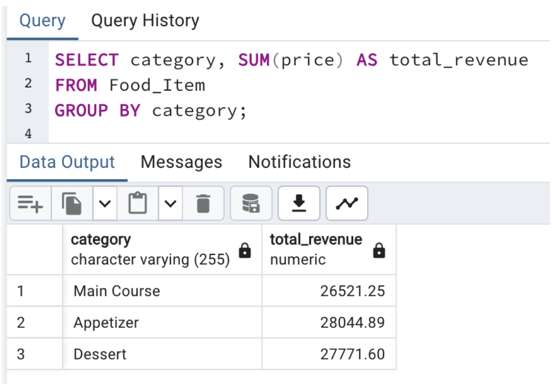

Query Query History

```
SELECT name, price
FROM Food_Item
ORDER BY price DESC
LIMIT 5;
```

Data Output Messages Notifications

	name character varying (255)	price numeric
1	Food Item 1959	50.0
2	Food Item 1927	49.98
3	Food Item 1361	49.92
4	Food Item 1063	49.92
5	Food Item 1939	49.9

SELECT name, price
FROM Food_Item
ORDER BY price DESC
LIMIT 5;

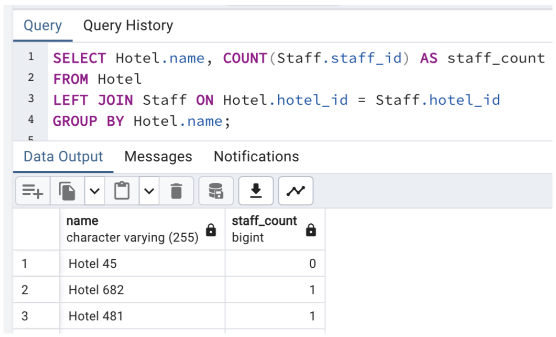

Query Query History

```
SELECT category, SUM(price) AS total_revenue
FROM Food_Item
GROUP BY category;
```

Data Output Messages Notifications

	category character varying (255)	total_revenue numeric
1	Main Course	26521.25
2	Appetizer	28044.89
3	Dessert	27771.60

SELECT Hotel.name, COUNT(Staff.staff_id) AS staff_count
FROM Hotel
LEFT JOIN Staff ON Hotel.hotel_id = Staff.hotel_id
GROUP BY Hotel.name;


Query Query History

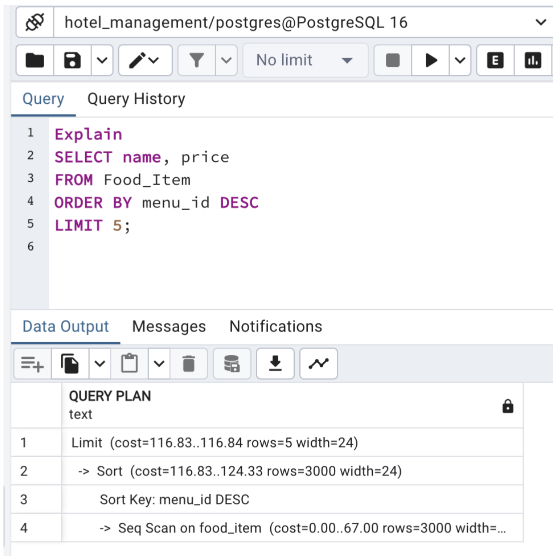
```
SELECT Hotel.name, COUNT(Staff.staff_id) AS staff_count
FROM Hotel
LEFT JOIN Staff ON Hotel.hotel_id = Staff.hotel_id
GROUP BY Hotel.name;
```

Data Output Messages Notifications

	name character varying (255)	staff_count bigint
1	Hotel 45	0
2	Hotel 682	1
3	Hotel 481	1

IX. Query Execution Analysis

Example 1 Explain
SELECT name, price
FROM Food_Item
ORDER BY menu_id DESC
LIMIT 5;
We identified this query as problematic due to its high cost of 116.83. Without appropriate indexing, the ORDER BY operation on the menu_id column could lead to sub optimal performance, especially as the dataset grows larger. To address this, we implemented indexing by creating an index on the menu_id column in the Food_Item table. This indexing strategy significantly reduced the cost to 0.28, leading to improved query performance.


hotel_management/postgres@PostgreSQL 16

No limit

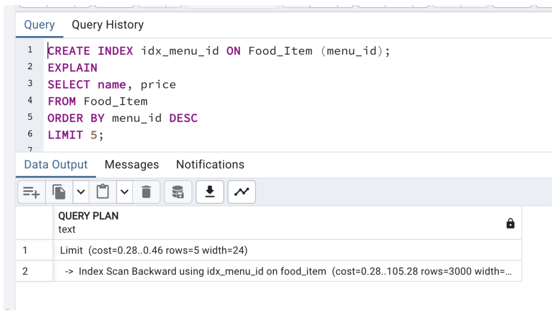
Query Query History

```
Explain
SELECT name, price
FROM Food_Item
ORDER BY menu_id DESC
LIMIT 5;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Limit (cost=116.83..116.84 rows=5 width=24)
2	-> Sort (cost=116.83..124.33 rows=3000 width=24)
3	Sort Key: menu_id DESC
4	-> Seq Scan on food_item (cost=0.00..67.00 rows=3000 width=...

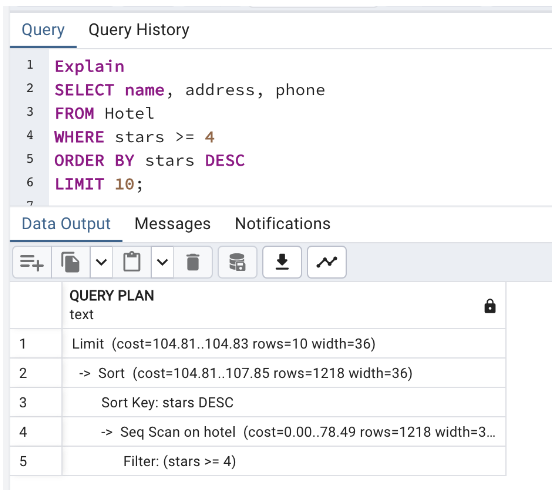
To improve performance, we created an index on the menu_id column in the Food_Item table. This index will optimize the sorting operation by allowing PostgreSQL to efficiently order the results based on the menu_id column. With the proposed index in place, the query execution plan is expected to be more efficient, leading to reduced execution time and improved overall performance. **Example 2** Explain
SELECT name, address, phone
FROM Hotel



```sql
CREATE INDEX idx_menu_id ON Food_Item (menu_id);
EXPLAIN
SELECT name, price
FROM Food_Item
ORDER BY menu_id DESC
LIMIT 5;
```

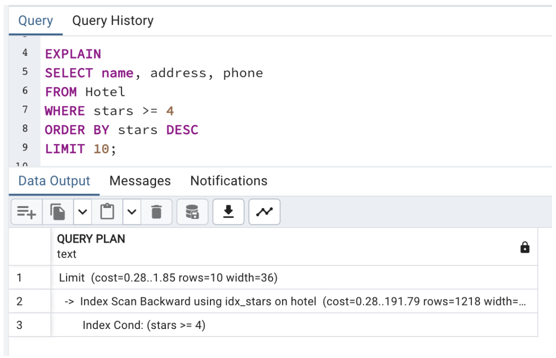
```
QUERY PLAN
text
1  Limit  (cost=0.28..0.46 rows=5 width=24)
2    -> Index Scan Backward using idx_menu_id on food_item  (cost=0.28..105.28 rows=3000 width=...
```

WHERE stars ¿= 4
ORDER BY stars DESC
LIMIT 10;
This query exhibited a high cost of 104.8 due to inefficient sorting operations on the stars column. To optimize performance, we implemented indexing by creating an index on the stars column in the Hotel table. This indexing strategy reduced the cost to 0.28, significantly improving sorting efficiency for queries filtering hotels based on their star ratings.
To improve performance, we created an index on the stars



```sql
Explain
SELECT name, address, phone
FROM Hotel
WHERE stars >= 4
ORDER BY stars DESC
LIMIT 10;
```

```
QUERY PLAN
text
1  Limit  (cost=104.81..104.83 rows=10 width=36)
2    -> Sort  (cost=104.81..107.85 rows=1218 width=36)
3         Sort Key: stars DESC
4         -> Seq Scan on hotel  (cost=0.00..78.49 rows=1218 width=3...
5              Filter: (stars >= 4)
```

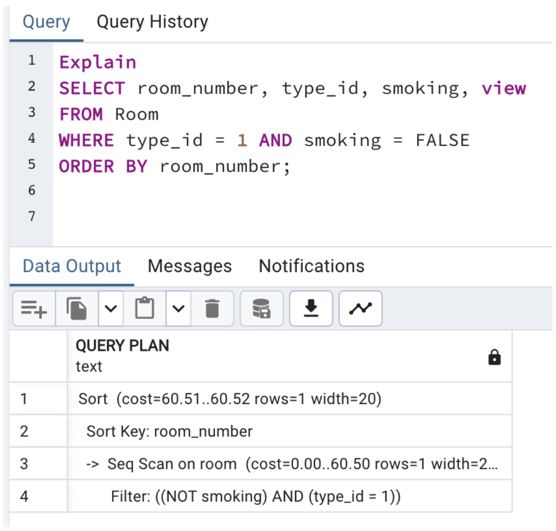


```sql
EXPLAIN
SELECT name, address, phone
FROM Hotel
WHERE stars >= 4
ORDER BY stars DESC
LIMIT 10;
```

```
QUERY PLAN
text
1  Limit  (cost=0.28..1.85 rows=10 width=36)
2    -> Index Scan Backward using idx_stars on hotel  (cost=0.28..191.79 rows=1218 width=...
3         Index Cond: (stars >= 4)
```

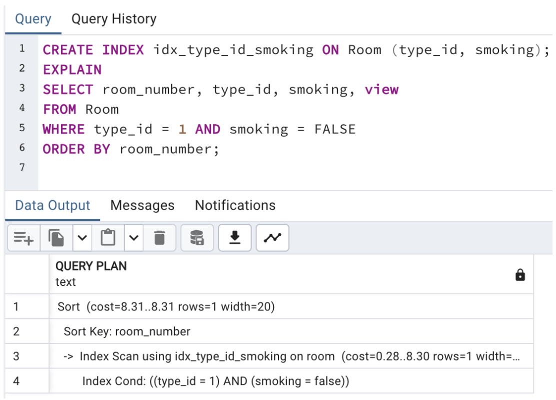
column in the Hotel table. This index will optimize the filtering operation by allowing PostgreSQL to efficiently locate and retrieve records with star ratings greater than or equal to 4. This indexing approach significantly improves sorting efficiency for the ORDER BY operation, thereby reducing execution times for queries filtering hotels based on their star ratings.

Example 3 Explain
SELECT room_number, type_id, smoking, view
FROM Room
This query had a high cost of 60.51 due to inefficient filtering operations on the type_id and smoking columns. To optimize performance, we implemented indexing by creating a composite index on the type_id and smoking columns in the Room table. This indexing strategy reduced the cost to 8.31, enhancing filtering efficiency for queries selecting rooms based on their type and smoking preference.



```sql
Explain
SELECT room_number, type_id, smoking, view
FROM Room
WHERE type_id = 1 AND smoking = FALSE
ORDER BY room_number;
```

```
QUERY PLAN
text
1  Sort  (cost=60.51..60.52 rows=1 width=20)
2    Sort Key: room_number
3    -> Seq Scan on room  (cost=0.00..60.50 rows=1 width=2...
4         Filter: ((NOT smoking) AND (type_id = 1))
```



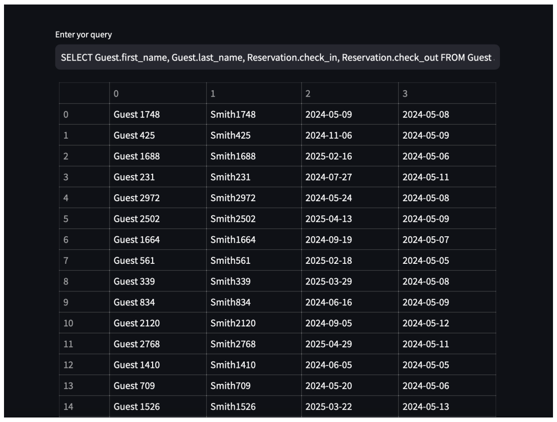
```sql
CREATE INDEX idx_type_id_smoking ON Room (type_id, smoking);
EXPLAIN
SELECT room_number, type_id, smoking, view
FROM Room
WHERE type_id = 1 AND smoking = FALSE
ORDER BY room_number;
```

```
QUERY PLAN
text
1  Sort  (cost=8.31..8.31 rows=1 width=20)
2    Sort Key: room_number
3    -> Index Scan using idx_type_id_smoking on room  (cost=0.28..8.30 rows=1 width=...
4         Index Cond: ((type_id = 1) AND (smoking = false))
```

To improve performance, we created a composite index on the type_id and smoking columns in the Room table. This index will optimize the filtering operation by allowing PostgreSQL to efficiently locate and retrieve records that satisfy the specified conditions (type_id = 1 AND smoking = FALSE). With the proposed index in place, the query execution plan is expected to be more efficient, leading to reduced execution time and improved overall performance.

X. Bonus Task

We have deployed our hotel management system using streamlit and psycopg2.



```
SELECT Guest.first_name, Guest.last_name, Reservation.check_in, Reservation.check_out FROM Guest
```

	0	1	2	3
0	Guest 1748	Smith1748	2024-05-09	2024-05-08
1	Guest 425	Smith425	2024-11-06	2024-05-09
2	Guest 1688	Smith1688	2025-02-16	2024-05-06
3	Guest 231	Smith231	2024-07-27	2024-05-11
4	Guest 2972	Smith2972	2024-05-24	2024-05-08
5	Guest 2502	Smith2502	2025-04-13	2024-05-09
6	Guest 1664	Smith1664	2024-09-19	2024-05-07
7	Guest 561	Smith561	2025-02-18	2024-05-05
8	Guest 339	Smith339	2025-03-29	2024-05-08
9	Guest 834	Smith834	2024-06-16	2024-05-09
10	Guest 2120	Smith2120	2024-09-05	2024-05-12
11	Guest 2768	Smith2768	2025-04-29	2024-05-11
12	Guest 1410	Smith1410	2024-06-05	2024-05-05
13	Guest 709	Smith709	2024-05-20	2024-05-06
14	Guest 1526	Smith1526	2025-03-22	2024-05-13

XI. References

https://en.wikipedia.org/wiki/Boyce-Codd_normal_form
https://www.atlantis-press.com/proceedings/icetis-13/8109
http://www.cmnt.lv/upload-files/ns_8art50_CMNT1806-124_BHS_Guo.pdf