RELATIONAL DATABASES

Assignment 2: Database Implementation

Student: S244679

Word count: 1076

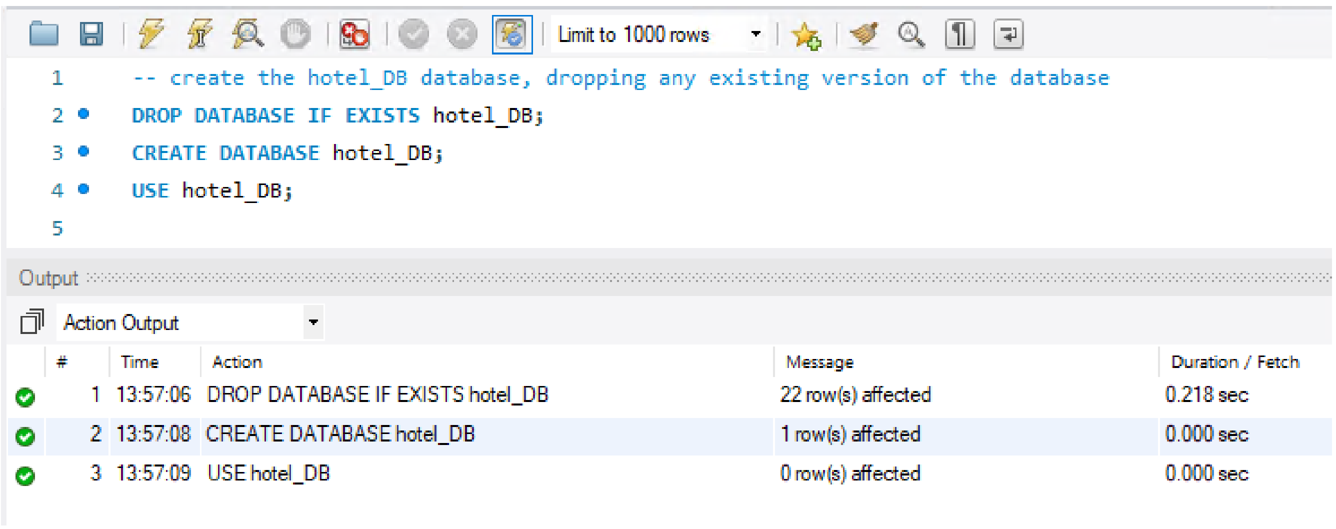
Contents

1 Introduction

This document describes the process of implementing the hotel database, designed in Assignment 1. Implementation and testing were performed using MySQL Workbench 8.0 and all screenshots are taken from that tool.

2 Database creation

The script ***database_creation.sql*** contains the SQL commands that create the hotel_DB database including the tables, views, triggers, stored procedures. It first drops any existing database before using CREATE DATABASE.



```
-- create the hotel_DB database, dropping any existing version of the database
DROP DATABASE IF EXISTS hotel_DB;
CREATE DATABASE hotel_DB;
USE hotel_DB;
```

```
#  Time      Action                               Message              Duration / Fetch
1  13:57:06  DROP DATABASE IF EXISTS hotel_DB     22 row(s) affected   0.218 sec
2  13:57:08  CREATE DATABASE hotel_DB             1 row(s) affected    0.000 sec
3  13:57:09  USE hotel_DB                         0 row(s) affected    0.000 sec
```

***Figure 1** – Shows the creation and usage of the database*

The script continues with CREATE TABLE commands based on the physical design model. They define tables (with column names, data types, and primary key). When the design linked tables together, a FOREIGN KEY constraint is defined to reference the other table. Foreign key ‘actions’ define what happens when referenced data is deleted.

Action	Description
ON DELETE SET NULL	When a row in the table, referenced by the foreign key, is deleted, the reference in this table is set to NULL
ON DELETE RESTRICT	MySQL will prevent deletion of any row in the foreign table that is currently being referenced by this table.
ON DELETE CASCADE	When a row in the table, referenced by the foreign key, is deleted, the row in this table that is referencing it will also be deleted.
ON DELETE SET DEFAULT	When a row in the table, referenced by the foreign key, is deleted, the reference in this table is set to its default value.

***Table 1** – Explanation of ON DELETE actions*

The order of table creation is important. When declaring a foreign key, the table being referenced must already exist – *see Table 2*.

Creation Order	Table	Other tables referenced
1	Staff	staff
2	room_type	-
3	bathroom_type	-
4	room_price	room_type, bathroom_type
5	Room	room_price
6	Address	-
7	company_account	address
8	Guest	address, company_account
9	Marketing	guest
10	Invoice	-
11	Promotion	-
12	Reservation	invoice, promotion, guest
13	check_in	staff, reservation
14	check_out	staff, reservation
15	complaint_category	-
16	Complaint	reservation, complaint_category, staff
17	complaint_resolution	complaint, staff
18	cleaning_session	staff
19	room_clean	room, cleaning_session

Table 2 *–Order of creation of database tables*

CHECK constraints were added to limit CHAR codes to a defined set of values or to enforce data format. DEFAULT value was set when appropriate.

A trigger was created to demonstrate they provide more customised error messages than a regular CHECK.

MySQL automatically indexes the primary key of a table. Query execution time was significantly improved by creating additional indexes. Using indexes is a balance, as they improve query performance but can introduce overhead when modifying data in large datasets (Silberschatz, 2011).

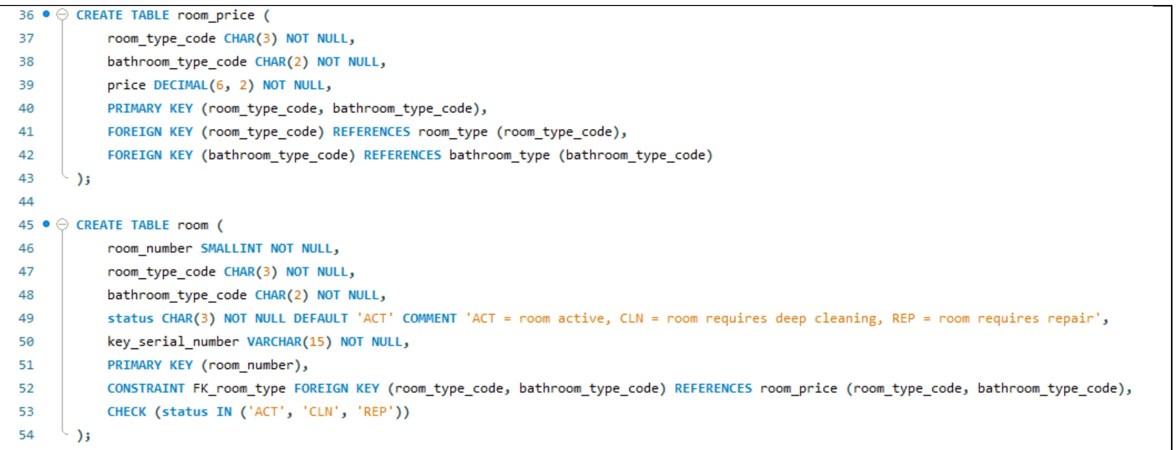
Views were used to combine commonly used data together by using table JOINs. Views allow extra values to be derived and make query design less complex. (Connolly, 2015)

Stored procedures were used to implement two complicated queries.

Finally, the script configures the access control by defining Roles, Users and GRANTs.

Table 3 highlights certain aspects of the database creation.

Section	Screenshot and Comments
room_price & room table creation	The room_price table is created with a Composite Primary Key. Each of the columns have been marked as NOT NULL as providing the data is mandatory. It is linked to two other tables via Foreign Key constraints. status in the room table uses DEFAULT to default the room as ACT (active). The other possible values are declared in a COMMENT. A CHECK command has been used to enforce the possible values of status.

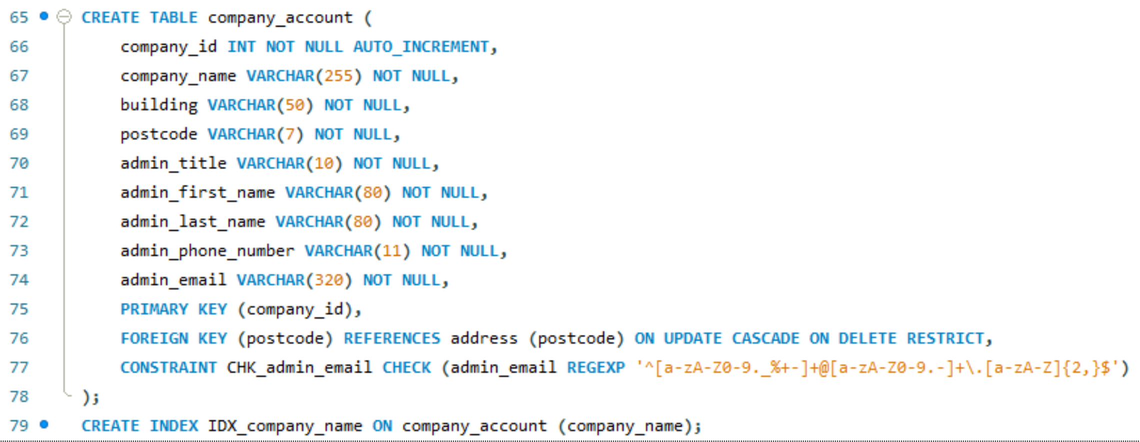


```sql
CREATE TABLE room_price (
    room_type_code CHAR(3) NOT NULL,
    bathroom_type_code CHAR(2) NOT NULL,
    price DECIMAL(6, 2) NOT NULL,
    PRIMARY KEY (room_type_code, bathroom_type_code),
    FOREIGN KEY (room_type_code) REFERENCES room_type (room_type_code),
    FOREIGN KEY (bathroom_type_code) REFERENCES bathroom_type (bathroom_type_code)
);

CREATE TABLE room (
    room_number SMALLINT NOT NULL,
    room_type_code CHAR(3) NOT NULL,
    bathroom_type_code CHAR(2) NOT NULL,
    status CHAR(3) NOT NULL DEFAULT 'ACT' COMMENT 'ACT = room active, CLN = room requires deep cleaning, REP = room requires repair',
    key_serial_number VARCHAR(15) NOT NULL,
    PRIMARY KEY (room_number),
    CONSTRAINT FK_room_type FOREIGN KEY (room_type_code, bathroom_type_code) REFERENCES room_price (room_type_code, bathroom_type_code),
    CHECK (status IN ('ACT', 'CLN', 'REP'))
);
```

company_account

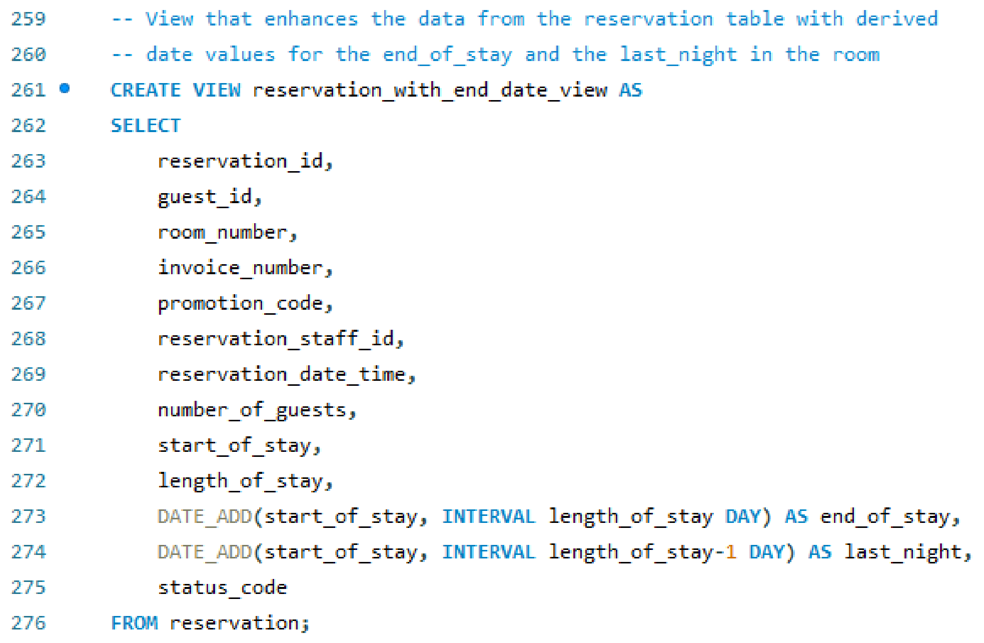
The company id is automatically allocated and incremented by the database to ensure uniqueness. The postcode uses ON UPDATE CASCADE to ensure a change of postcode in the address table updates this table too.
A REGEXP CHECK constraint has been applied to the e-mail address to check it contains valid characters before and after the @ sign and has a domain name with a full stop and at least two characters after that.
An index was added to improve performance of searches by company_name.



```sql
CREATE TABLE company_account (
    company_id INT NOT NULL AUTO_INCREMENT,
    company_name VARCHAR(255) NOT NULL,
    building VARCHAR(50) NOT NULL,
    postcode VARCHAR(7) NOT NULL,
    admin_title VARCHAR(10) NOT NULL,
    admin_first_name VARCHAR(80) NOT NULL,
    admin_last_name VARCHAR(80) NOT NULL,
    admin_phone_number VARCHAR(11) NOT NULL,
    admin_email VARCHAR(320) NOT NULL,
    PRIMARY KEY (company_id),
    FOREIGN KEY (postcode) REFERENCES address (postcode) ON UPDATE CASCADE ON DELETE RESTRICT,
    CONSTRAINT CHK_admin_email CHECK (admin_email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$')
);
CREATE INDEX IDX_company_name ON company_account (company_name);
```

reservation_with_end_date_view

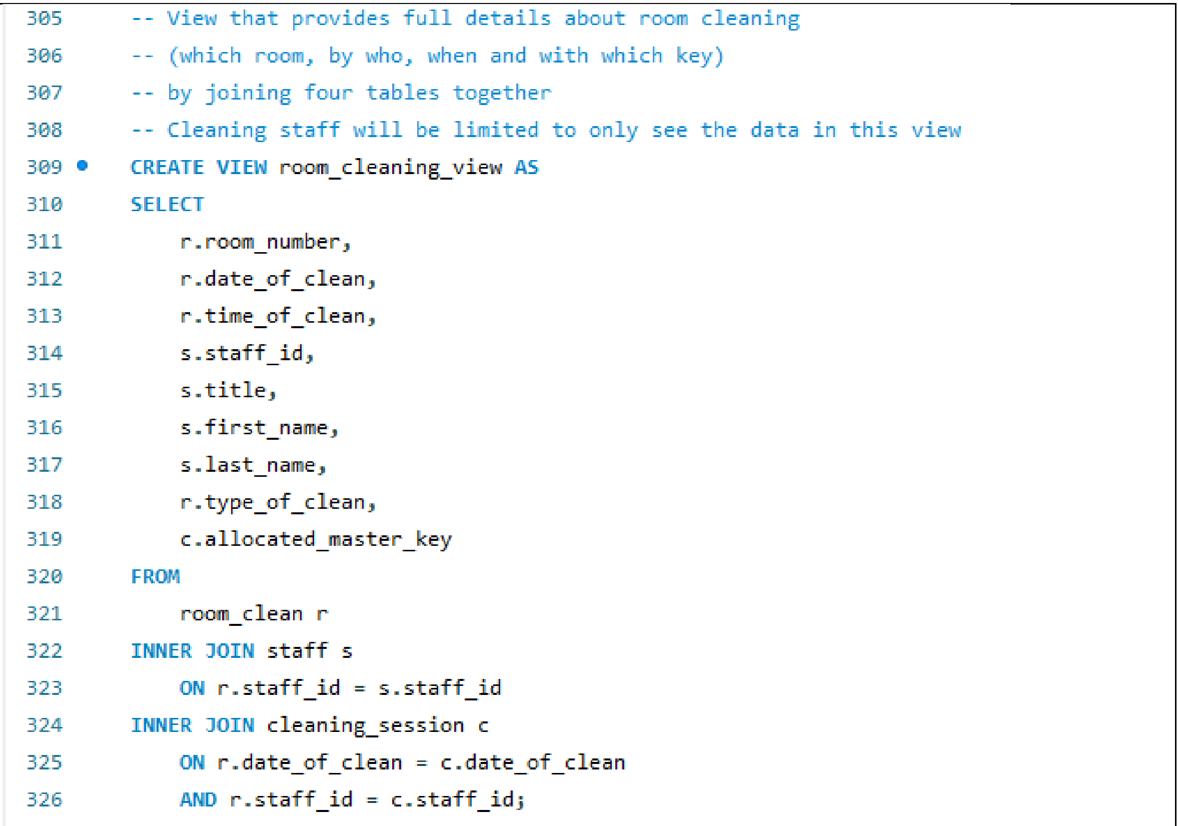
This view enhances the data stored in the reservation table by deriving the date of the end of stay in the hotel and the date of the last night.



```sql
-- View that enhances the data from the reservation table with derived
-- date values for the end_of_stay and the last_night in the room
CREATE VIEW reservation_with_end_date_view AS
SELECT
    reservation_id,
    guest_id,
    room_number,
    invoice_number,
    promotion_code,
    reservation_staff_id,
    reservation_date_time,
    number_of_guests,
    start_of_stay,
    length_of_stay,
    DATE_ADD(start_of_stay, INTERVAL length_of_stay DAY) AS end_of_stay,
    DATE_ADD(start_of_stay, INTERVAL length_of_stay-1 DAY) AS last_night,
    status_code
FROM reservation;
```

room_cleaning_view

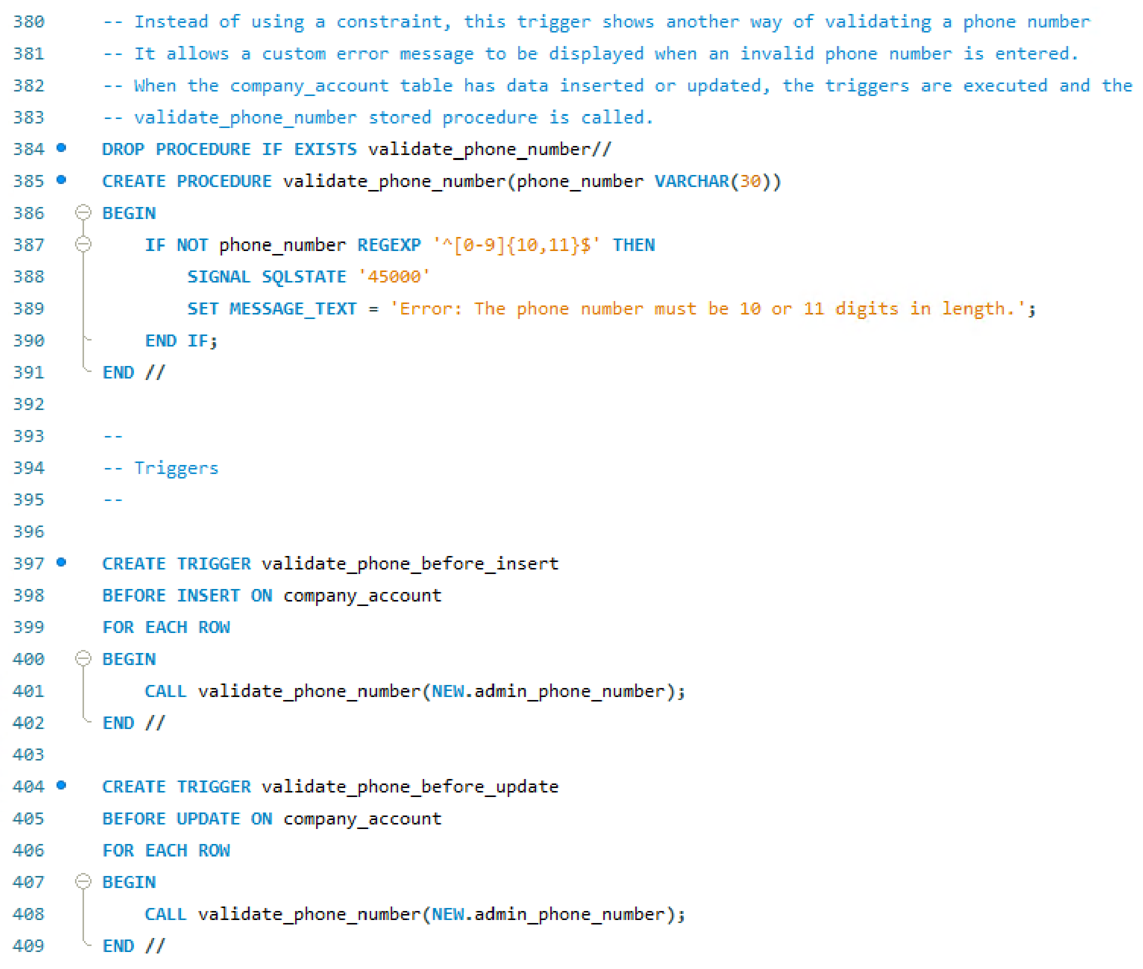
Cleaning staff in the hotel are given very limited access to the database and are only allowed to view the data combined into this one view.



```sql
-- View that provides full details about room cleaning
-- (which room, by who, when and with which key)
-- by joining four tables together
-- Cleaning staff will be limited to only see the data in this view
CREATE VIEW room_cleaning_view AS
SELECT
    r.room_number,
    r.date_of_clean,
    r.time_of_clean,
    s.staff_id,
    s.title,
    s.first_name,
    s.last_name,
    r.type_of_clean,
    c.allocated_master_key
FROM
    room_clean r
INNER JOIN staff s
    ON r.staff_id = s.staff_id
INNER JOIN cleaning_session c
    ON r.date_of_clean = c.date_of_clean
    AND r.staff_id = c.staff_id;
```

Admin phone number validation

This stored procedure and these insert/update triggers show how a custom error message can be displayed if a telephone number of incorrect length is entered into the table. See test results in Section 6.

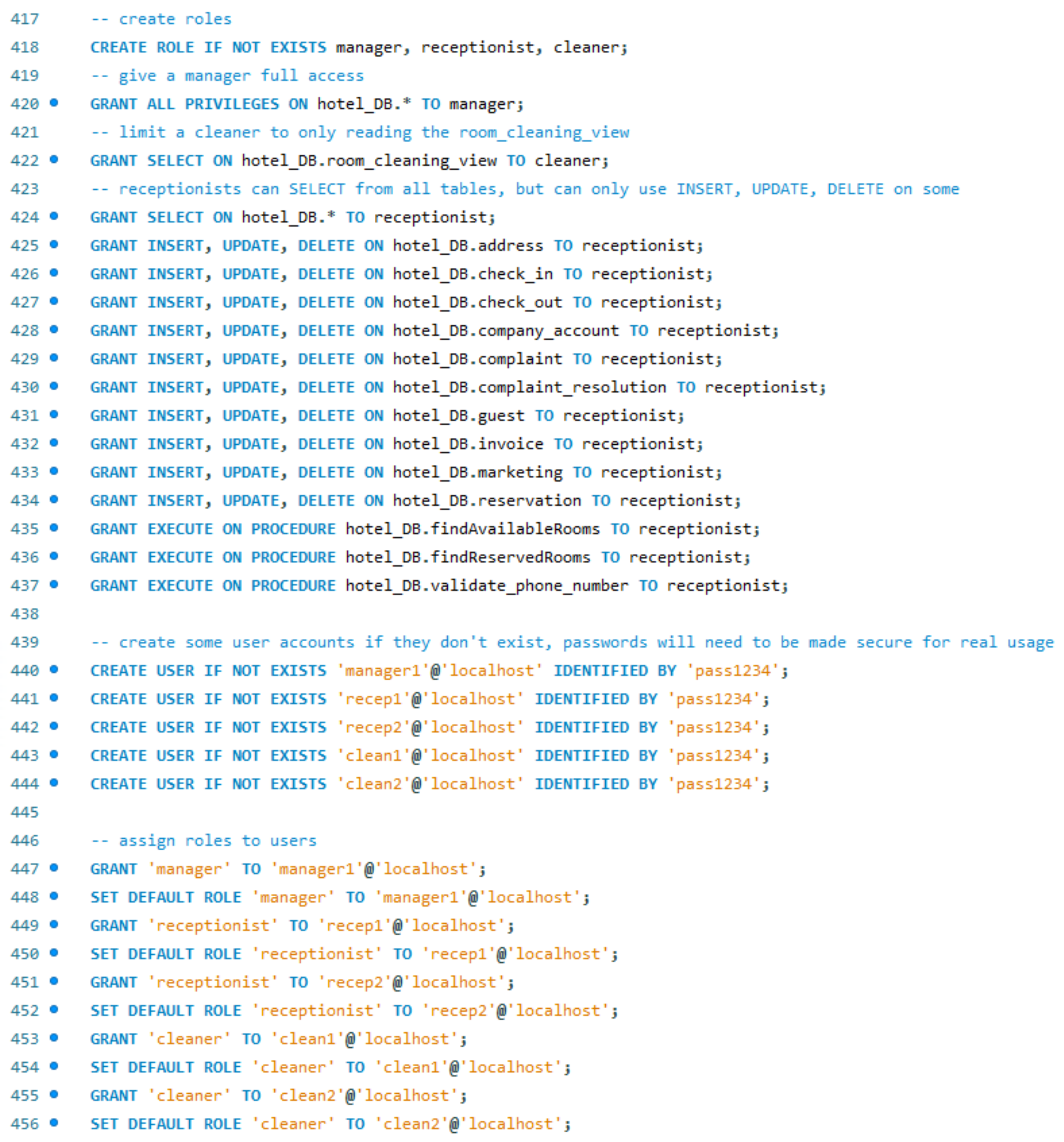


```sql
-- Instead of using a constraint, this trigger shows another way of validating a phone number
-- It allows a custom error message to be displayed when an invalid phone number is entered.
-- When the company_account table has data inserted or updated, the triggers are executed and the
-- validate_phone_number stored procedure is called.
DROP PROCEDURE IF EXISTS validate_phone_number//
CREATE PROCEDURE validate_phone_number(phone_number VARCHAR(30))
BEGIN
    IF NOT phone_number REGEXP '^[0-9]{10,11}$' THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: The phone number must be 10 or 11 digits in length.';
    END IF;
END //

--
-- Triggers
--

CREATE TRIGGER validate_phone_before_insert
BEFORE INSERT ON company_account
FOR EACH ROW
BEGIN
    CALL validate_phone_number(NEW.admin_phone_number);
END //

CREATE TRIGGER validate_phone_before_update
BEFORE UPDATE ON company_account
FOR EACH ROW
BEGIN
    CALL validate_phone_number(NEW.admin_phone_number);
END //
```

Access control	This section shows the creation of different Roles and user accounts. The manager is given access to all tables. The receptionists can view everything and modify tables necessary for daily operation. The cleaning staff are limited to the single view. 

```sql
-- create roles
CREATE ROLE IF NOT EXISTS manager, receptionist, cleaner;
-- give a manager full access
GRANT ALL PRIVILEGES ON hotel_DB.* TO manager;
-- limit a cleaner to only reading the room_cleaning_view
GRANT SELECT ON hotel_DB.room_cleaning_view TO cleaner;
-- receptionists can SELECT from all tables, but can only use INSERT, UPDATE, DELETE on some
GRANT SELECT ON hotel_DB.* TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.address TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.check_in TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.check_out TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.company_account TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.complaint TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.complaint_resolution TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.guest TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.invoice TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.marketing TO receptionist;
GRANT INSERT, UPDATE, DELETE ON hotel_DB.reservation TO receptionist;
GRANT EXECUTE ON PROCEDURE hotel_DB.findAvailableRooms TO receptionist;
GRANT EXECUTE ON PROCEDURE hotel_DB.findReservedRooms TO receptionist;
GRANT EXECUTE ON PROCEDURE hotel_DB.validate_phone_number TO receptionist;

-- create some user accounts if they don't exist, passwords will need to be made secure for real usage
CREATE USER IF NOT EXISTS 'manager1'@'localhost' IDENTIFIED BY 'pass1234';
CREATE USER IF NOT EXISTS 'recep1'@'localhost' IDENTIFIED BY 'pass1234';
CREATE USER IF NOT EXISTS 'recep2'@'localhost' IDENTIFIED BY 'pass1234';
CREATE USER IF NOT EXISTS 'clean1'@'localhost' IDENTIFIED BY 'pass1234';
CREATE USER IF NOT EXISTS 'clean2'@'localhost' IDENTIFIED BY 'pass1234';

-- assign roles to users
GRANT 'manager' TO 'manager1'@'localhost';
SET DEFAULT ROLE 'manager' TO 'manager1'@'localhost';
GRANT 'receptionist' TO 'recep1'@'localhost';
SET DEFAULT ROLE 'receptionist' TO 'recep1'@'localhost';
GRANT 'receptionist' TO 'recep2'@'localhost';
SET DEFAULT ROLE 'receptionist' TO 'recep2'@'localhost';
GRANT 'cleaner' TO 'clean1'@'localhost';
SET DEFAULT ROLE 'cleaner' TO 'clean1'@'localhost';
GRANT 'cleaner' TO 'clean2'@'localhost';
SET DEFAULT ROLE 'cleaner' TO 'clean2'@'localhost';
```

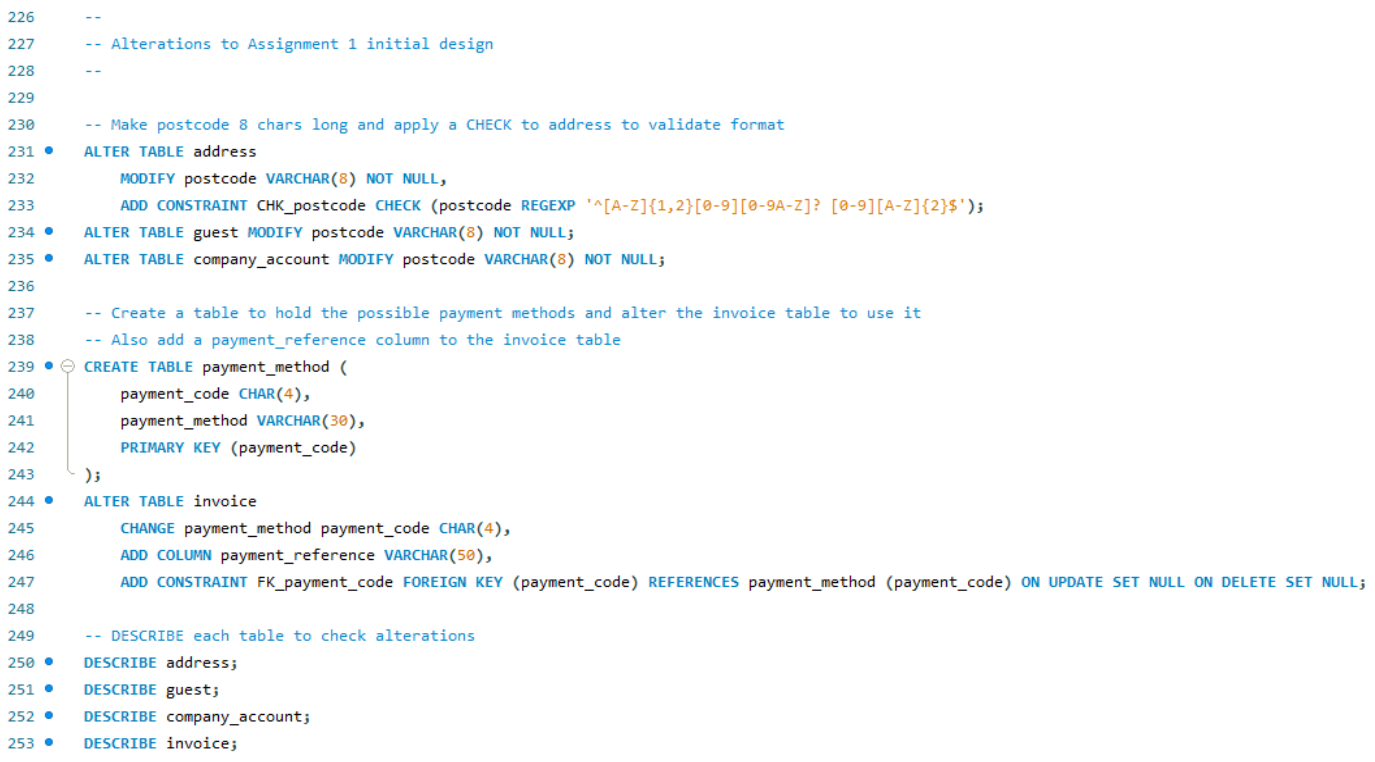
***Table 3** – Key features of the database creation script*

3 Changes from Assignment 1 design

A UK postcode format requires a space separator, so *postcode* columns were changed from 7 to 8 characters and a REGEXP format CHECK was added (MySQL, 8.0)

The invoice table contained a free-form text column for method of payment which risked data entry inconsistencies, potentially harming payment reporting accuracy. It was replaced with a *payment_code* column and a new payment_method lookup table. A *payment_reference* column was also added so that payments can be linked with a payment processor (e.g Stripe).

See Appendix 1 for updated physical design.



```
--
-- Alterations to Assignment 1 initial design
--

-- Make postcode 8 chars long and apply a CHECK to address to validate format
ALTER TABLE address
    MODIFY postcode VARCHAR(8) NOT NULL,
    ADD CONSTRAINT CHK_postcode CHECK (postcode REGEXP '^[A-Z]{1,2}[0-9][0-9A-Z]? [0-9][A-Z]{2}$');
ALTER TABLE guest MODIFY postcode VARCHAR(8) NOT NULL;
ALTER TABLE company_account MODIFY postcode VARCHAR(8) NOT NULL;

-- Create a table to hold the possible payment methods and alter the invoice table to use it
-- Also add a payment_reference column to the invoice table
CREATE TABLE payment_method (
    payment_code CHAR(4),
    payment_method VARCHAR(30),
    PRIMARY KEY (payment_code)
);
ALTER TABLE invoice
    CHANGE payment_method payment_code CHAR(4),
    ADD COLUMN payment_reference VARCHAR(50),
    ADD CONSTRAINT FK_payment_code FOREIGN KEY (payment_code) REFERENCES payment_method (payment_code) ON UPDATE SET NULL ON DELETE SET NULL;

-- DESCRIBE each table to check alterations
DESCRIBE address;
DESCRIBE guest;
DESCRIBE company_account;
DESCRIBE invoice;
```

***Figure 2** – Table Alterations section of the database creation script*

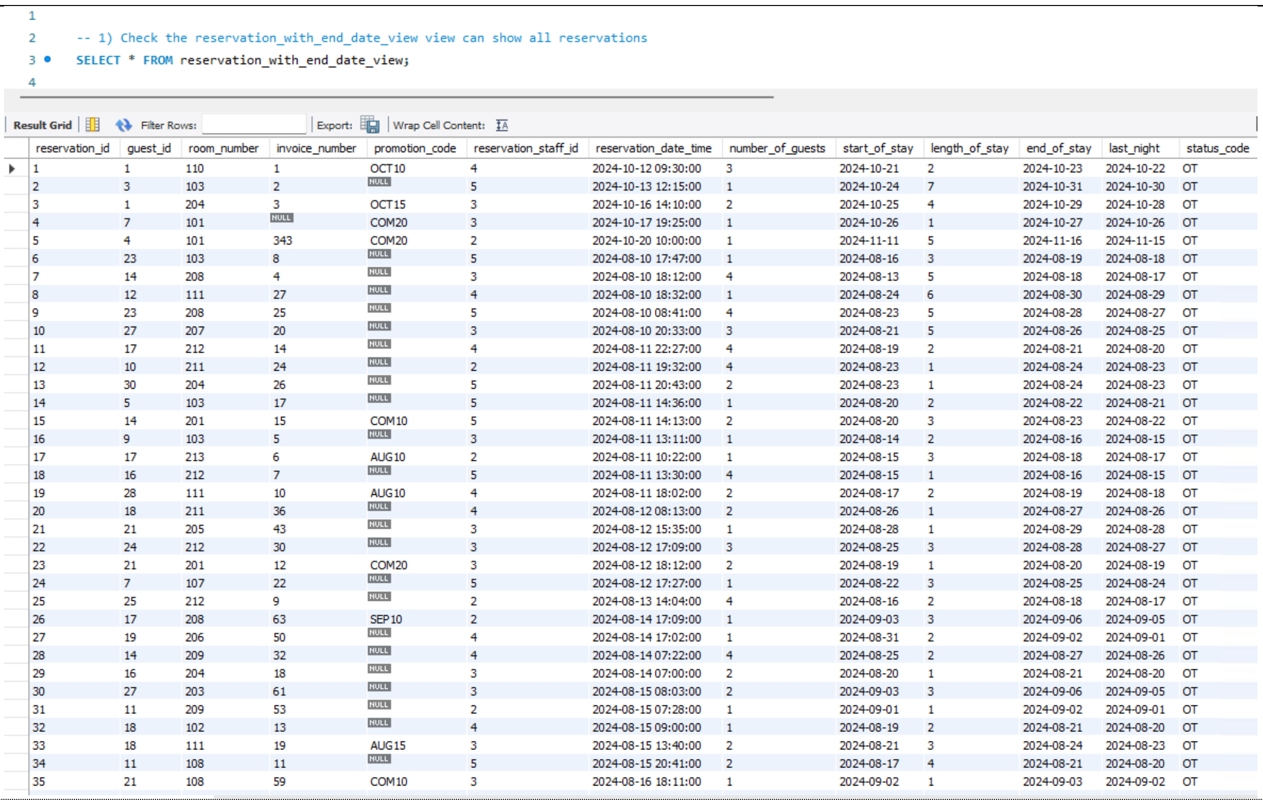
4 Test data

The script ***test_data_population.sql*** INSERTs example data from assignment along with additional randomly generated test data which can be used to fully exercise the example queries. Emphasis was given to realistic room reservations and associated data aiming to show active hotel usage during Autumn 2024.

5 Using the data

The file ***select_script_of_example_queries.sql*** implements Assignment 1's example queries and others required to test database functionality. The method of table joining was carefully chosen considering the possibility of NULL table rows. Table 4 explains the queries, highlights usage of different SQL keywords and shows test results.

Query Id	**Purpose / Evidence / Comments**
1	Select all rows from reservation_with_end_date_view to check it shows all reservations. SELECT * was used to choose all columns from the table. SUCCESSFUL



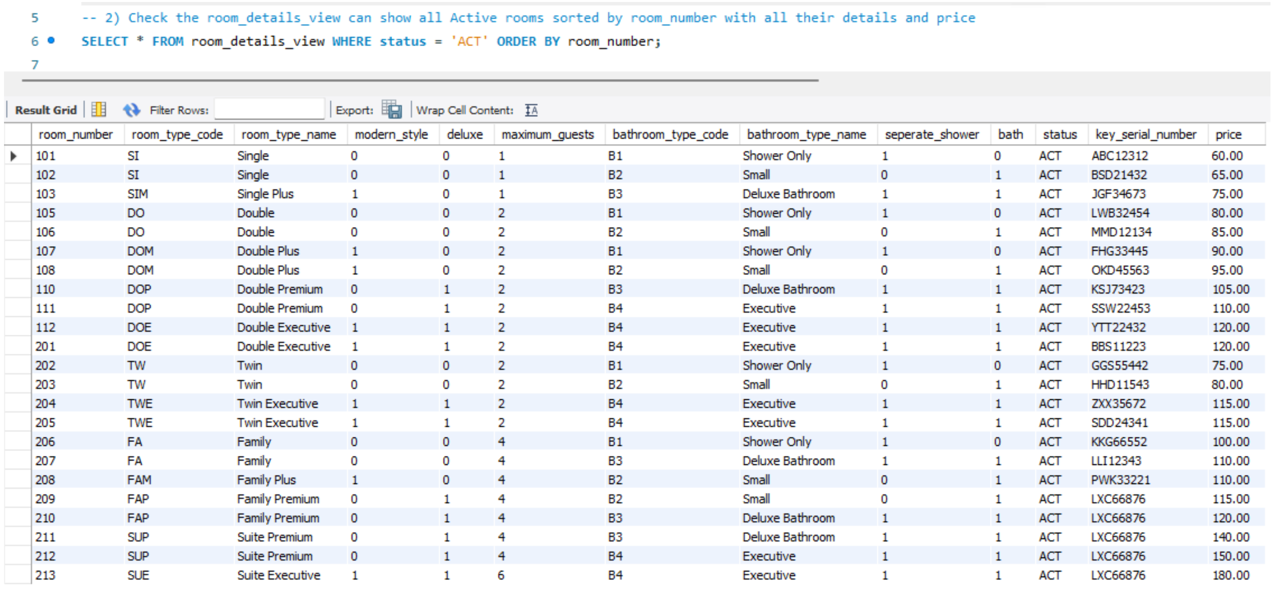
```sql
-- 1) Check the reservation_with_end_date_view view can show all reservations
SELECT * FROM reservation_with_end_date_view;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

reservation_id	guest_id	room_number	invoice_number	promotion_code	reservation_staff_id	reservation_date_time	number_of_guests	start_of_stay	length_of_stay	end_of_stay	last_night	status_code
1	1	110	1	OCT10	4	2024-10-12 09:30:00	3	2024-10-21	2	2024-10-23	2024-10-22	OT
2	3	103	2	NULL	5	2024-10-13 12:15:00	1	2024-10-24	7	2024-10-31	2024-10-30	OT
3	1	204	3	OCT15	3	2024-10-16 14:10:00	2	2024-10-25	4	2024-10-29	2024-10-28	OT
4	7	101	NULL	COM20	3	2024-10-17 19:25:00	1	2024-10-26	1	2024-10-27	2024-10-26	OT
5	4	101	343	COM20	2	2024-10-20 10:00:00	1	2024-11-11	5	2024-11-16	2024-11-15	OT
6	23	103	8	NULL	5	2024-08-10 17:47:00	1	2024-08-16	3	2024-08-19	2024-08-18	OT
7	14	208	4	NULL	3	2024-08-10 18:12:00	4	2024-08-13	5	2024-08-18	2024-08-17	OT
8	12	111	27	NULL	4	2024-08-10 18:32:00	1	2024-08-24	6	2024-08-30	2024-08-29	OT
9	23	208	25	NULL	5	2024-08-10 08:41:00	4	2024-08-23	5	2024-08-28	2024-08-27	OT
10	27	207	20	NULL	3	2024-08-10 20:33:00	3	2024-08-21	5	2024-08-26	2024-08-25	OT
11	17	212	14	NULL	4	2024-08-11 22:27:00	4	2024-08-19	2	2024-08-21	2024-08-20	OT
12	10	211	24	NULL	2	2024-08-11 19:32:00	4	2024-08-23	1	2024-08-24	2024-08-23	OT
13	30	204	26	NULL	5	2024-08-11 20:43:00	2	2024-08-23	1	2024-08-24	2024-08-23	OT
14	5	103	17	NULL	5	2024-08-11 14:36:00	1	2024-08-20	2	2024-08-22	2024-08-21	OT
15	14	201	15	COM10	5	2024-08-11 14:13:00	2	2024-08-20	3	2024-08-23	2024-08-22	OT
16	9	103	5	NULL	3	2024-08-11 13:11:00	1	2024-08-14	2	2024-08-16	2024-08-15	OT
17	17	213	6	AUG10	2	2024-08-11 10:22:00	1	2024-08-15	3	2024-08-18	2024-08-17	OT
18	16	212	7	NULL	5	2024-08-11 13:30:00	4	2024-08-15	1	2024-08-16	2024-08-15	OT
19	28	111	10	AUG10	4	2024-08-11 18:02:00	2	2024-08-17	2	2024-08-19	2024-08-18	OT
20	18	211	36	NULL	4	2024-08-12 08:13:00	2	2024-08-26	1	2024-08-27	2024-08-26	OT
21	21	205	43	NULL	3	2024-08-12 15:35:00	1	2024-08-28	1	2024-08-29	2024-08-28	OT
22	24	212	30	NULL	3	2024-08-12 17:09:00	3	2024-08-25	3	2024-08-28	2024-08-27	OT
23	21	201	12	COM20	3	2024-08-12 18:12:00	2	2024-08-19	1	2024-08-20	2024-08-19	OT
24	7	107	22	NULL	5	2024-08-12 17:27:00	1	2024-08-22	3	2024-08-25	2024-08-24	OT
25	25	212	9	NULL	2	2024-08-13 14:04:00	4	2024-08-16	2	2024-08-18	2024-08-17	OT
26	17	208	63	SEP10	2	2024-08-14 17:09:00	1	2024-09-03	3	2024-09-06	2024-09-05	OT
27	19	206	50	NULL	4	2024-08-14 17:02:00	1	2024-08-31	2	2024-09-02	2024-09-01	OT
28	14	209	32	NULL	4	2024-08-14 07:22:00	4	2024-08-25	2	2024-08-27	2024-08-26	OT
29	16	204	18	NULL	3	2024-08-14 07:00:00	2	2024-08-20	1	2024-08-21	2024-08-20	OT
30	27	203	61	NULL	3	2024-08-15 08:03:00	2	2024-09-03	3	2024-09-06	2024-09-05	OT
31	11	209	53	NULL	2	2024-08-15 07:28:00	1	2024-09-01	1	2024-09-02	2024-09-01	OT
32	18	102	13	NULL	4	2024-08-15 09:00:00	1	2024-08-19	2	2024-08-21	2024-08-20	OT
33	18	111	19	AUG15	3	2024-08-15 13:40:00	2	2024-08-21	3	2024-08-24	2024-08-23	OT
34	11	108	11	NULL	5	2024-08-15 20:41:00	2	2024-08-17	4	2024-08-21	2024-08-20	OT
35	21	108	59	COM10	3	2024-08-16 18:11:00	1	2024-09-02	1	2024-09-03	2024-09-02	OT

2

Select all rows from room_details_view where the status is Active to test the view shows the correct rooms with all their details and is sorted by room_number.
Uses a WHERE statement to filter by status and ORDER BY to sort ascending.
SUCCESSFUL



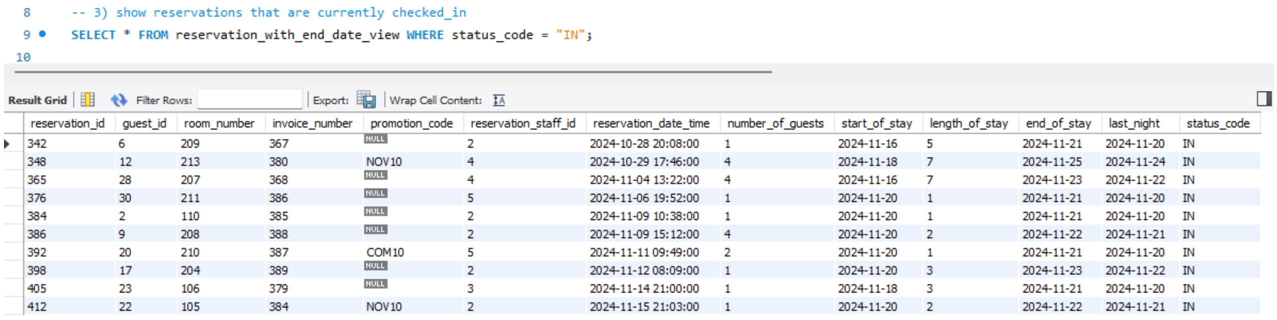
```sql
-- 2) Check the room_details_view can show all Active rooms sorted by room_number with all their details and price
SELECT * FROM room_details_view WHERE status = 'ACT' ORDER BY room_number;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

room_number	room_type_code	room_type_name	modern_style	deluxe	maximum_guests	bathroom_type_code	bathroom_type_name	seperate_shower	bath	status	key_serial_number	price
101	SI	Single	0	0	1	B1	Shower Only	1	0	ACT	ABC12312	60.00
102	SI	Single	0	0	1	B2	Small	0	1	ACT	BSD21432	65.00
103	SIM	Single Plus	1	0	1	B3	Deluxe Bathroom	1	1	ACT	JGF34673	75.00
105	DO	Double	0	0	2	B1	Shower Only	1	0	ACT	LWB32454	80.00
106	DO	Double	0	0	2	B2	Small	0	1	ACT	MMD12134	85.00
107	DOM	Double Plus	1	0	2	B1	Shower Only	1	0	ACT	FHG33445	90.00
108	DOM	Double Plus	1	0	2	B2	Small	0	1	ACT	OKD45563	95.00
110	DOP	Double Premium	0	1	2	B3	Deluxe Bathroom	1	1	ACT	KSJ73423	105.00
111	DOP	Double Premium	0	1	2	B4	Executive	1	1	ACT	SSW22453	110.00
112	DOE	Double Executive	1	1	2	B4	Executive	1	1	ACT	YTT22432	120.00
201	DOE	Double Executive	1	1	2	B4	Executive	1	1	ACT	BBS11223	120.00
202	TW	Twin	0	0	2	B1	Shower Only	1	0	ACT	GGS55442	75.00
203	TW	Twin	0	0	2	B2	Small	0	1	ACT	HHD11543	80.00
204	TWE	Twin Executive	1	1	2	B4	Executive	1	1	ACT	ZXX35672	115.00
205	TWE	Twin Executive	1	1	2	B4	Executive	1	1	ACT	SDD24341	115.00
206	FA	Family	0	0	4	B1	Shower Only	1	0	ACT	KKG66552	100.00
207	FA	Family	0	0	4	B3	Deluxe Bathroom	1	1	ACT	LLI12343	110.00
208	FAM	Family Plus	1	0	4	B2	Small	0	1	ACT	PWK33221	110.00
209	FAP	Family Premium	0	1	4	B2	Small	0	1	ACT	LXC66876	115.00
210	FAP	Family Premium	0	1	4	B3	Deluxe Bathroom	1	1	ACT	LXC66876	120.00
211	SUP	Suite Premium	0	1	4	B3	Deluxe Bathroom	1	1	ACT	LXC66876	140.00
212	SUP	Suite Premium	0	1	4	B4	Executive	1	1	ACT	LXC66876	150.00
213	SUE	Suite Executive	1	1	6	B4	Executive	1	1	ACT	LXC66876	180.00

3

Select all rows from reservation_with_end_date_view that are marked as currently checked-in to check it shows the correct information.
SUCCESSFUL



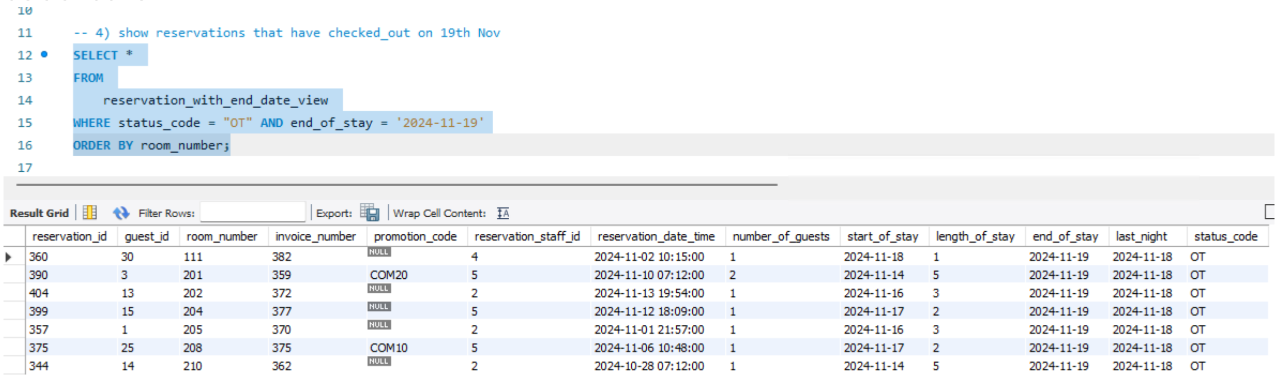
```sql
-- 3) show reservations that are currently checked_in
SELECT * FROM reservation_with_end_date_view WHERE status_code = "IN";
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

reservation_id	guest_id	room_number	invoice_number	promotion_code	reservation_staff_id	reservation_date_time	number_of_guests	start_of_stay	length_of_stay	end_of_stay	last_night	status_code
342	6	209	367	NULL	2	2024-10-28 20:08:00	1	2024-11-16	5	2024-11-21	2024-11-20	IN
348	12	213	380	NOV10	4	2024-10-29 17:46:00	4	2024-11-18	7	2024-11-25	2024-11-24	IN
365	28	207	368	NULL	4	2024-11-04 13:22:00	4	2024-11-16	7	2024-11-23	2024-11-22	IN
376	30	211	386	NULL	5	2024-11-06 19:52:00	1	2024-11-20	1	2024-11-21	2024-11-20	IN
384	2	110	385	NULL	2	2024-11-09 10:38:00	1	2024-11-20	1	2024-11-21	2024-11-20	IN
386	9	208	388	NULL	2	2024-11-09 15:12:00	4	2024-11-20	2	2024-11-22	2024-11-21	IN
392	20	210	387	COM10	5	2024-11-11 09:49:00	2	2024-11-20	1	2024-11-21	2024-11-20	IN
398	17	204	389	NULL	2	2024-11-12 08:09:00	1	2024-11-20	3	2024-11-23	2024-11-22	IN
405	23	106	379	NULL	3	2024-11-14 21:00:00	1	2024-11-18	3	2024-11-21	2024-11-20	IN
412	22	105	384	NOV10	2	2024-11-15 21:03:00	1	2024-11-20	2	2024-11-22	2024-11-21	IN

4

Select the reservations that checked out on 19th November to check the correct information is returned.
Uses two clauses AND'd together in the WHERE statement to filter both by status code and date.
SUCCESSFUL



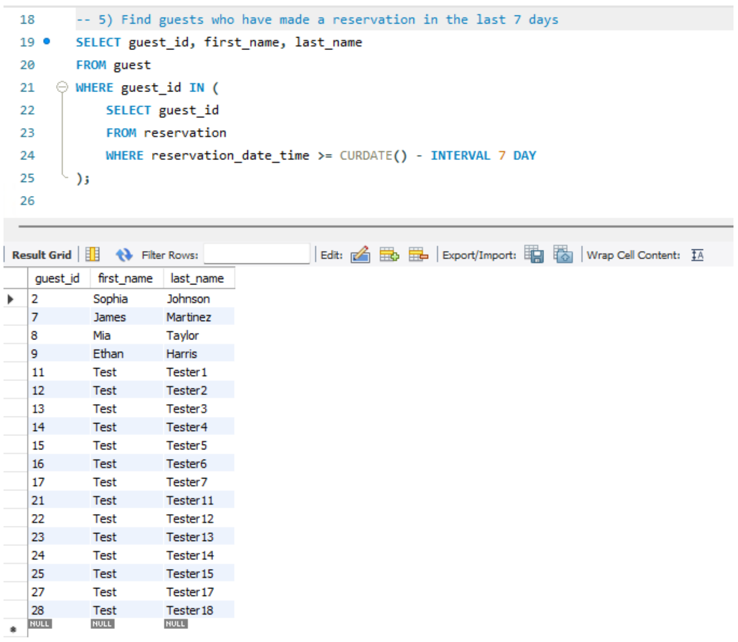
```
-- 4) show reservations that have checked_out on 19th Nov
SELECT *
FROM
    reservation_with_end_date_view
WHERE status_code = "OT" AND end_of_stay = '2024-11-19'
ORDER BY room_number;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

reservation_id	guest_id	room_number	invoice_number	promotion_code	reservation_staff_id	reservation_date_time	number_of_guests	start_of_stay	length_of_stay	end_of_stay	last_night	status_code
360	30	111	382	NULL	4	2024-11-02 10:15:00	1	2024-11-18	1	2024-11-19	2024-11-18	OT
390	3	201	359	COM20	5	2024-11-10 07:12:00	2	2024-11-14	5	2024-11-19	2024-11-18	OT
404	13	202	372	NULL	2	2024-11-13 19:54:00	1	2024-11-16	3	2024-11-19	2024-11-18	OT
399	15	204	377	NULL	5	2024-11-12 18:09:00	1	2024-11-17	2	2024-11-19	2024-11-18	OT
357	1	205	370	NULL	2	2024-11-01 21:57:00	1	2024-11-16	3	2024-11-19	2024-11-18	OT
375	25	208	375	COM10	5	2024-11-06 10:48:00	1	2024-11-17	2	2024-11-19	2024-11-18	OT
344	14	210	362	NULL	2	2024-10-28 07:12:00	1	2024-11-14	5	2024-11-19	2024-11-18	OT

5

Retrieve the names (with ids) of guests that have made a reservation in the last week.
Specific columns were named in the SELECT statement to return just the data required.
Makes use of a subquery. The nested query gets the guest ids matching the date criteria (using current date and some date arithmetic) and the top query uses those ids to extract the names from the guest table.
SUCCESSFUL



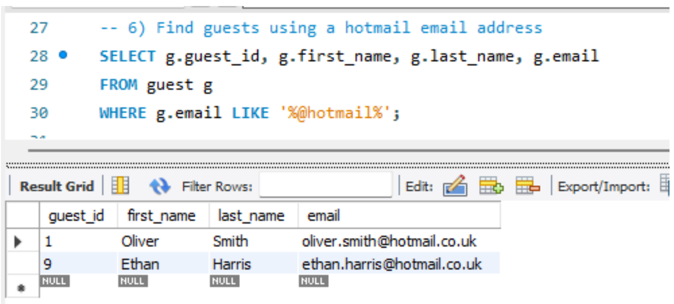
```
-- 5) Find guests who have made a reservation in the last 7 days
SELECT guest_id, first_name, last_name
FROM guest
WHERE guest_id IN (
    SELECT guest_id
    FROM reservation
    WHERE reservation_date_time >= CURDATE() - INTERVAL 7 DAY
);
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

guest_id	first_name	last_name
2	Sophia	Johnson
7	James	Martinez
8	Mia	Taylor
9	Ethan	Harris
11	Test	Tester1
12	Test	Tester2
13	Test	Tester3
14	Test	Tester4
15	Test	Tester5
16	Test	Tester6
17	Test	Tester7
21	Test	Tester11
22	Test	Tester12
23	Test	Tester13
24	Test	Tester14
25	Test	Tester15
27	Test	Tester17
28	Test	Tester18
NULL	NULL	NULL

6

Retrieve the names of guests that use a Hotmail email address.
Uses the LIKE string-matching feature of MySQL.
SUCCESSFUL



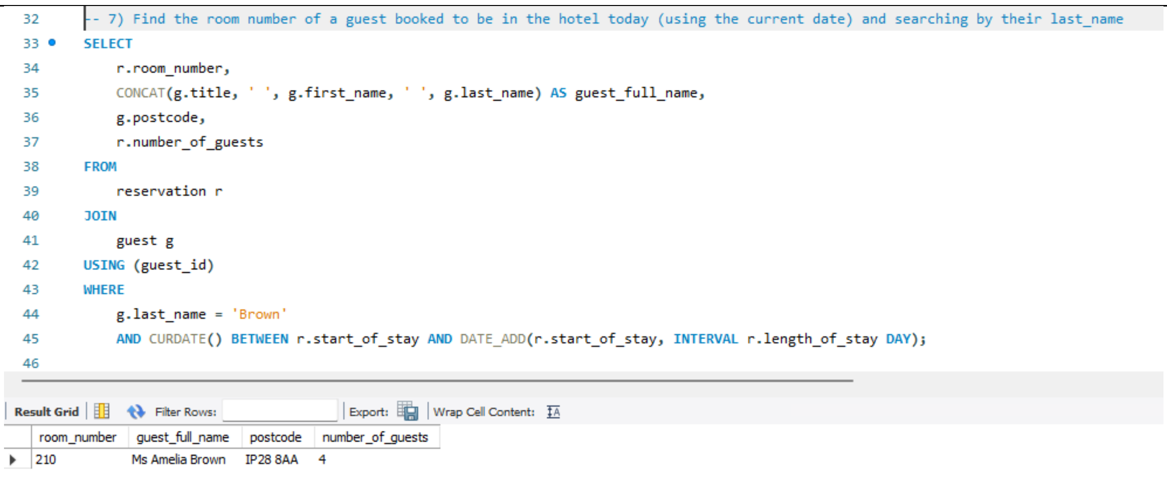
```
-- 6) Find guests using a hotmail email address
SELECT g.guest_id, g.first_name, g.last_name, g.email
FROM guest g
WHERE g.email LIKE '%@hotmail%';
```

Result Grid | Filter Rows: | Edit: | Export/Import:

guest_id	first_name	last_name	email
1	Oliver	Smith	oliver.smith@hotmail.co.uk
9	Ethan	Harris	ethan.harris@hotmail.co.uk
NULL	NULL	NULL	NULL

7

Find the room number of a guest currently booked into the hotel searching by their last name.
USING is used to join two tables as they both have guest_id columns.
CONCAT is used to build the guest's full name from component attributes.
DATE_ADD is used to derive the last day the guest will stay in the hotel and BETWEEN is used to check that today's date is in that booking time window.
SUCCESSFUL



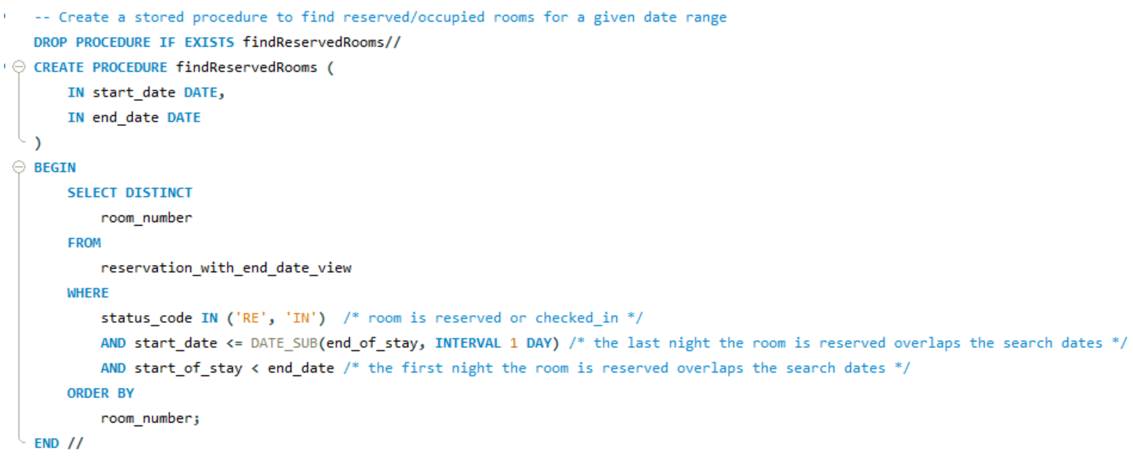
```
-- 7) Find the room number of a guest booked to be in the hotel today (using the current date) and searching by their last_name
SELECT
    r.room_number,
    CONCAT(g.title, ' ', g.first_name, ' ', g.last_name) AS guest_full_name,
    g.postcode,
    r.number_of_guests
FROM
    reservation r
JOIN
    guest g
USING (guest_id)
WHERE
    g.last_name = 'Brown'
    AND CURDATE() BETWEEN r.start_of_stay AND DATE_ADD(r.start_of_stay, INTERVAL r.length_of_stay DAY);
```

```
room_number  guest_full_name   postcode  number_of_guests
210          Ms Amelia Brown   IP28 8AA  4
```

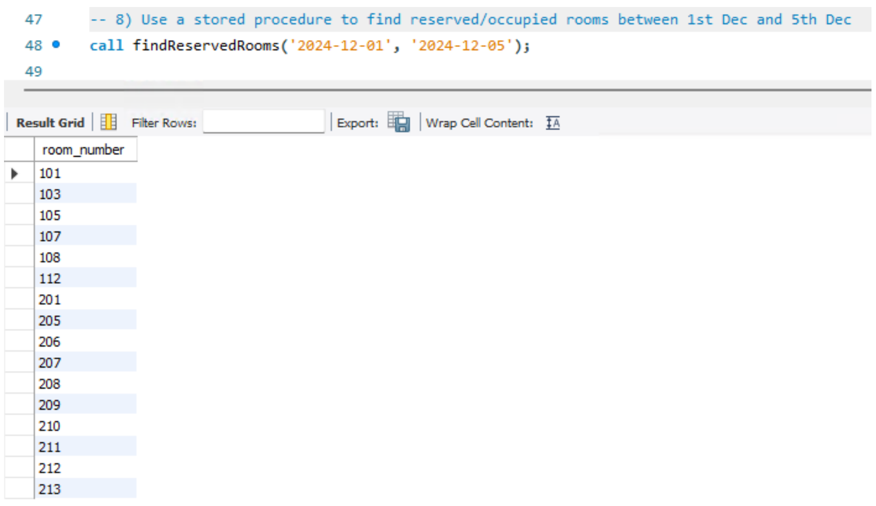
8

Use a stored procedure to find **reserved/occupied** rooms between 1st Dec and 5th Dec.
The stored procedure call is simple, requiring just two dates, but the stored procedure itself may get multiple matches for the same room in the time window, so uses DISTINCT to report each room only once.
SUCCESSFUL

Note: Stored procedure is defined in the database_creation.sql script



```
-- Create a stored procedure to find reserved/occupied rooms for a given date range
DROP PROCEDURE IF EXISTS findReservedRooms//
CREATE PROCEDURE findReservedRooms (
    IN start_date DATE,
    IN end_date DATE
)
BEGIN
    SELECT DISTINCT
        room_number
    FROM
        reservation_with_end_date_view
    WHERE
        status_code IN ('RE', 'IN')  /* room is reserved or checked_in */
        AND start_date <= DATE_SUB(end_of_stay, INTERVAL 1 DAY) /* the last night the room is reserved overlaps the search dates */
        AND start_of_stay < end_date /* the first night the room is reserved overlaps the search dates */
    ORDER BY
        room_number;
END //
```



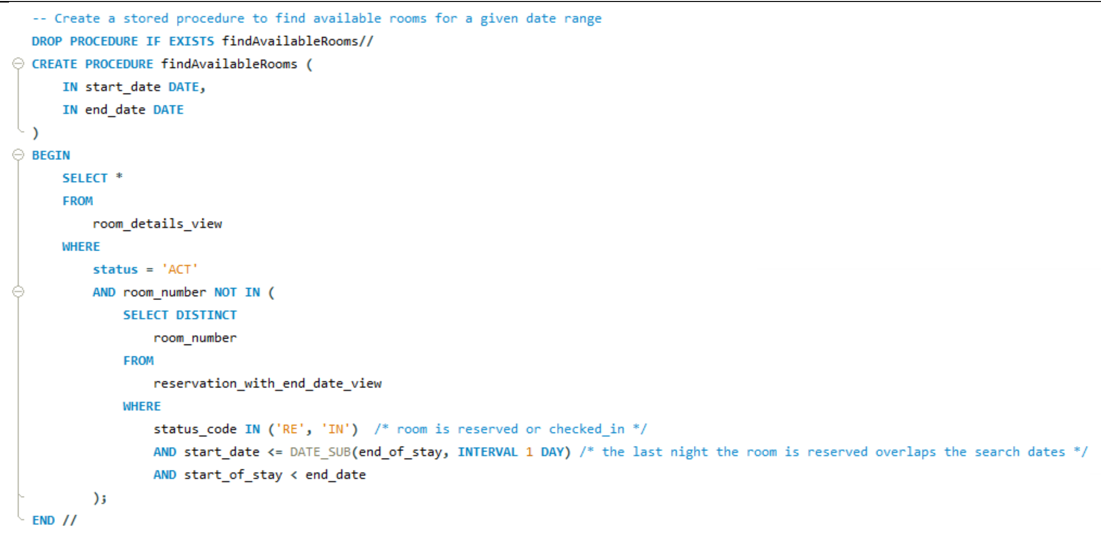
```
-- 8) Use a stored procedure to find reserved/occupied rooms between 1st Dec and 5th Dec
call findReservedRooms('2024-12-01', '2024-12-05');
```

```
room_number
101
103
105
107
108
112
201
205
206
207
208
209
210
211
212
213
```

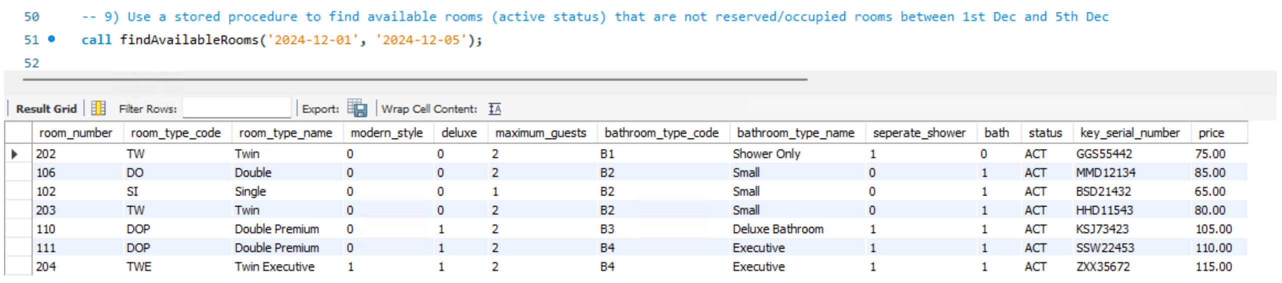
9

Use a stored procedure to find **available** rooms between 1st Dec and 5th Dec.
The stored procedure call is simple, requiring just two dates, but the stored procedure itself uses a nested query to find Active rooms that are NOT already reserved in the time window. Full details about the room are returned to help the receptionist discuss a possible booking with a customer.
SUCCESSFUL

Note: Stored procedure is defined in the database_creation.sql script



```sql
-- Create a stored procedure to find available rooms for a given date range
DROP PROCEDURE IF EXISTS findAvailableRooms//
CREATE PROCEDURE findAvailableRooms (
    IN start_date DATE,
    IN end_date DATE
)
BEGIN
    SELECT *
    FROM
        room_details_view
    WHERE
        status = 'ACT'
        AND room_number NOT IN (
            SELECT DISTINCT
                room_number
            FROM
                reservation_with_end_date_view
            WHERE
                status_code IN ('RE', 'IN')  /* room is reserved or checked_in */
                AND start_date <= DATE_SUB(end_of_stay, INTERVAL 1 DAY) /* the last night the room is reserved overlaps the search dates */
                AND start_of_stay < end_date
        );
END //
```



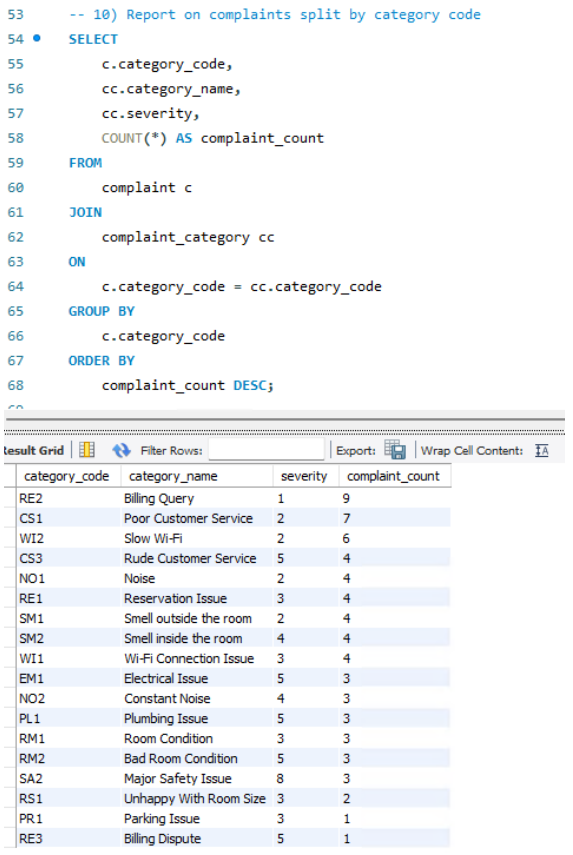
```sql
-- 9) Use a stored procedure to find available rooms (active status) that are not reserved/occupied rooms between 1st Dec and 5th Dec
call findAvailableRooms('2024-12-01', '2024-12-05');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

room_number	room_type_code	room_type_name	modern_style	deluxe	maximum_guests	bathroom_type_code	bathroom_type_name	seperate_shower	bath	status	key_serial_number	price
202	TW	Twin	0	0	2	B1	Shower Only	1	0	ACT	GGS55442	75.00
106	DO	Double	0	0	2	B2	Small	0	1	ACT	MMD12134	85.00
102	SI	Single	0	0	1	B2	Small	0	1	ACT	BSD21432	65.00
203	TW	Twin	0	0	2	B2	Small	0	1	ACT	HHD11543	80.00
110	DOP	Double Premium	0	1	2	B3	Deluxe Bathroom	1	1	ACT	KSJ73423	105.00
111	DOP	Double Premium	0	1	2	B4	Executive	1	1	ACT	SSW22453	110.00
204	TWE	Twin Executive	1	1	2	B4	Executive	1	1	ACT	ZXX35672	115.00

10

Report on the complaints raised by guests. Joins to the complaint_category table to be able to report with a meaningful description and the severity rating. The results are ordered descending from the greatest number of complaints to least. The results are first GROUPed BY category_code and then COUNT is used total the number of each complaint type.
SUCCESSFUL



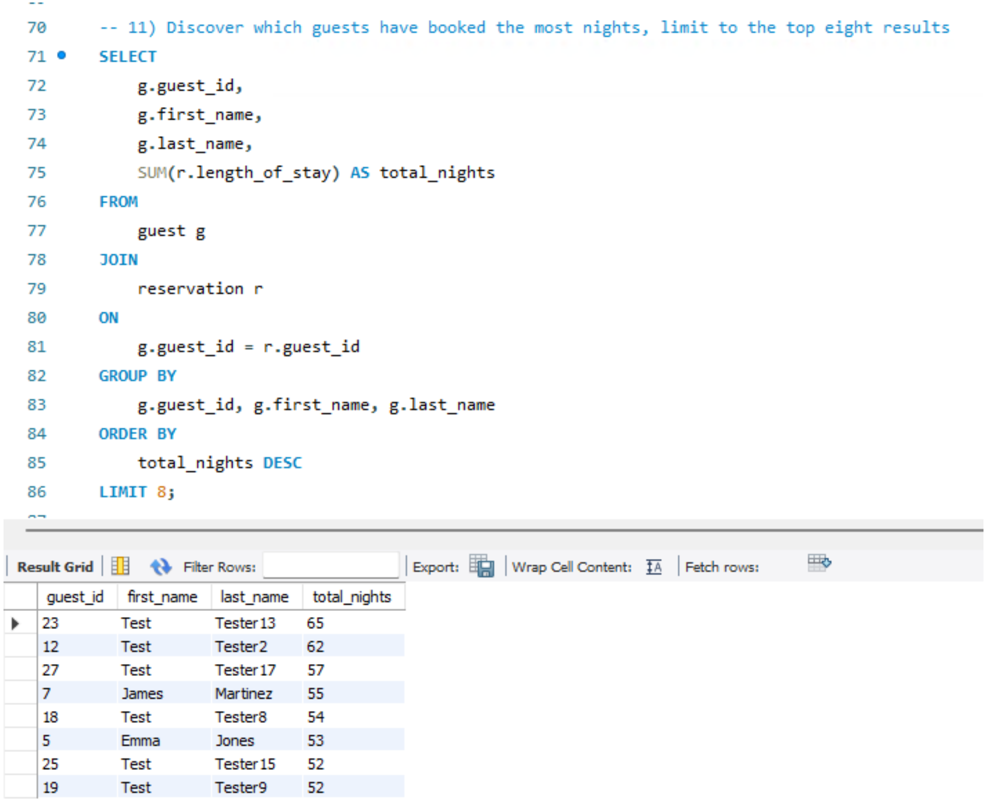
```sql
-- 10) Report on complaints split by category code
SELECT
    c.category_code,
    cc.category_name,
    cc.severity,
    COUNT(*) AS complaint_count
FROM
    complaint c
JOIN
    complaint_category cc
ON
    c.category_code = cc.category_code
GROUP BY
    c.category_code
ORDER BY
    complaint_count DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

category_code	category_name	severity	complaint_count
RE2	Billing Query	1	9
CS1	Poor Customer Service	2	7
WI2	Slow Wi-Fi	2	6
CS3	Rude Customer Service	5	4
NO1	Noise	2	4
RE1	Reservation Issue	3	4
SM1	Smell outside the room	2	4
SM2	Smell inside the room	4	4
WI1	Wi-Fi Connection Issue	3	4
EM1	Electrical Issue	5	3
NO2	Constant Noise	4	3
PL1	Plumbing Issue	5	3
RM1	Room Condition	3	3
RM2	Bad Room Condition	5	3
SA2	Major Safety Issue	8	3
RS1	Unhappy With Room Size	3	2
PR1	Parking Issue	3	1
RE3	Billing Dispute	5	1

11

Report on the guests that have booked the most nights.
GROUP_BY is used to group the data so that the results of each guest are together, then SUM is used to calculate the total_nights. Finally LIMIT is used to report on just the top eight guests.
SUCCESSFUL



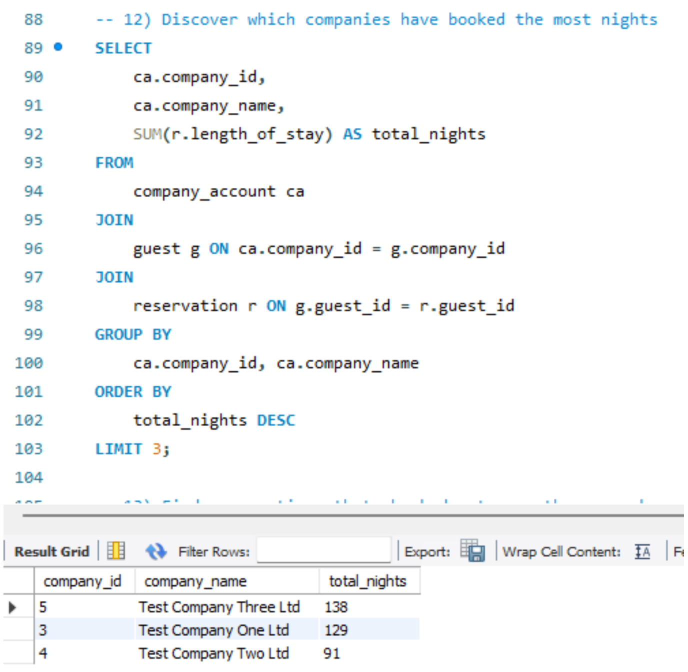
```sql
-- 11) Discover which guests have booked the most nights, limit to the top eight results
SELECT
    g.guest_id,
    g.first_name,
    g.last_name,
    SUM(r.length_of_stay) AS total_nights
FROM
    guest g
JOIN
    reservation r
ON
    g.guest_id = r.guest_id
GROUP BY
    g.guest_id, g.first_name, g.last_name
ORDER BY
    total_nights DESC
LIMIT 8;
```

```
Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

  guest_id  first_name  last_name  total_nights
▶ 23        Test        Tester13   65
  12        Test        Tester2    62
  27        Test        Tester17   57
  7         James       Martinez   55
  18        Test        Tester8    54
  5         Emma        Jones      53
  25        Test        Tester15   52
  19        Test        Tester9    52
```

12

Like above, this time reporting on the top 3 companies whose guests have booked the most nights.
Inner JOINs are used to insist that there are matching rows in all three tables.
SUCCESSFUL



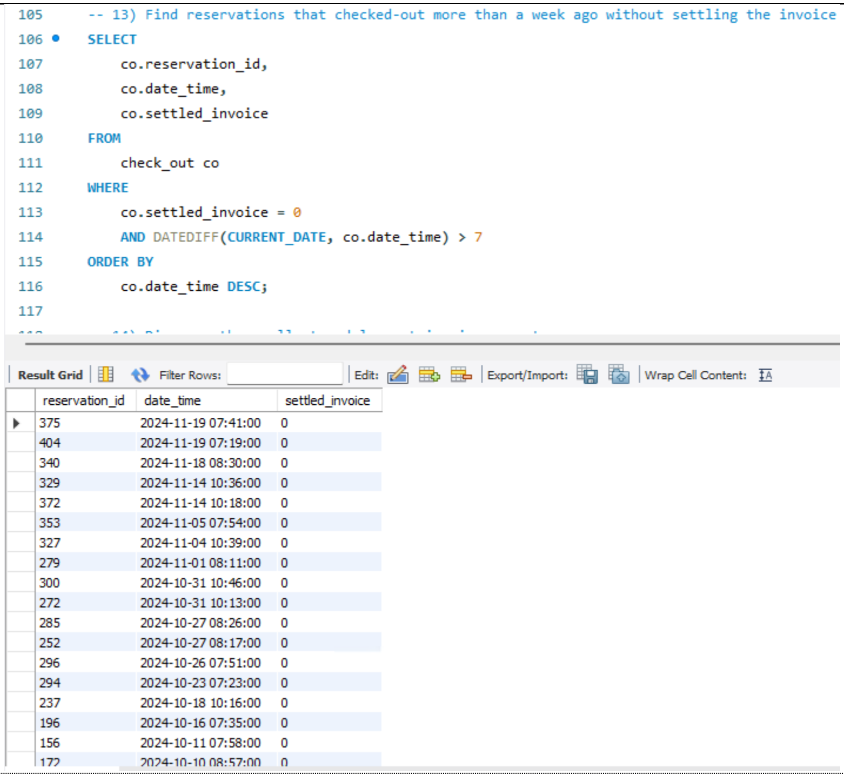
```sql
-- 12) Discover which companies have booked the most nights
SELECT
    ca.company_id,
    ca.company_name,
    SUM(r.length_of_stay) AS total_nights
FROM
    company_account ca
JOIN
    guest g ON ca.company_id = g.company_id
JOIN
    reservation r ON g.guest_id = r.guest_id
GROUP BY
    ca.company_id, ca.company_name
ORDER BY
    total_nights DESC
LIMIT 3;
```

```
Result Grid | Filter Rows: | Export: | Wrap Cell Content: | F

  company_id  company_name           total_nights
▶ 5           Test Company Three Ltd  138
  3           Test Company One Ltd    129
  4           Test Company Two Ltd    91
```

13

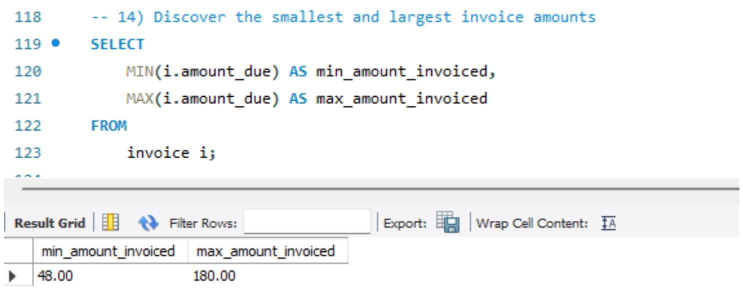
Report on reservations that checked out more than a week ago and still haven't settled their invoice.
DATEDIFF is used to calculate the number of days since check-out.
SUCCESSFUL



```sql
-- 13) Find reservations that checked-out more than a week ago without settling the invoice
SELECT
    co.reservation_id,
    co.date_time,
    co.settled_invoice
FROM
    check_out co
WHERE
    co.settled_invoice = 0
    AND DATEDIFF(CURRENT_DATE, co.date_time) > 7
ORDER BY
    co.date_time DESC;
```

reservation_id	date_time	settled_invoice
375	2024-11-19 07:41:00	0
404	2024-11-19 07:19:00	0
340	2024-11-18 08:30:00	0
329	2024-11-14 10:36:00	0
372	2024-11-14 10:18:00	0
353	2024-11-05 07:54:00	0
327	2024-11-04 10:39:00	0
279	2024-11-01 08:11:00	0
300	2024-10-31 10:46:00	0
272	2024-10-31 10:13:00	0
285	2024-10-27 08:26:00	0
252	2024-10-27 08:17:00	0
296	2024-10-26 07:51:00	0
294	2024-10-23 07:23:00	0
237	2024-10-18 10:16:00	0
196	2024-10-16 07:35:00	0
156	2024-10-11 07:58:00	0
172	2024-10-10 08:57:00	0

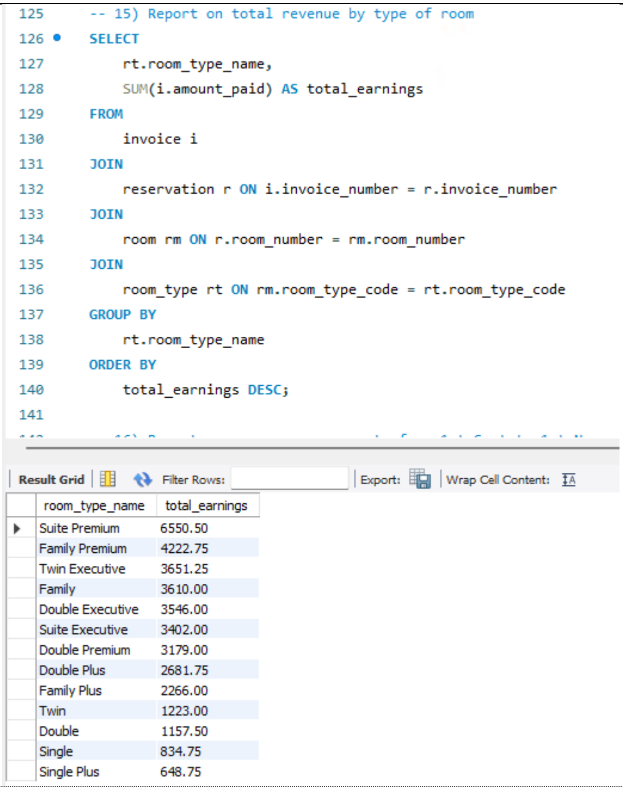
14 Calculate the smallest and largest invoice amounts.
Uses MIN and MAX to find those values.
SUCCESSFUL



```sql
-- 14) Discover the smallest and largest invoice amounts
SELECT
    MIN(i.amount_due) AS min_amount_invoiced,
    MAX(i.amount_due) AS max_amount_invoiced
FROM
    invoice i;
```

min_amount_invoiced	max_amount_invoiced
48.00	180.00

15 Report the total revenue by type of room sorted from highest to lowest.
SUCCESSFUL



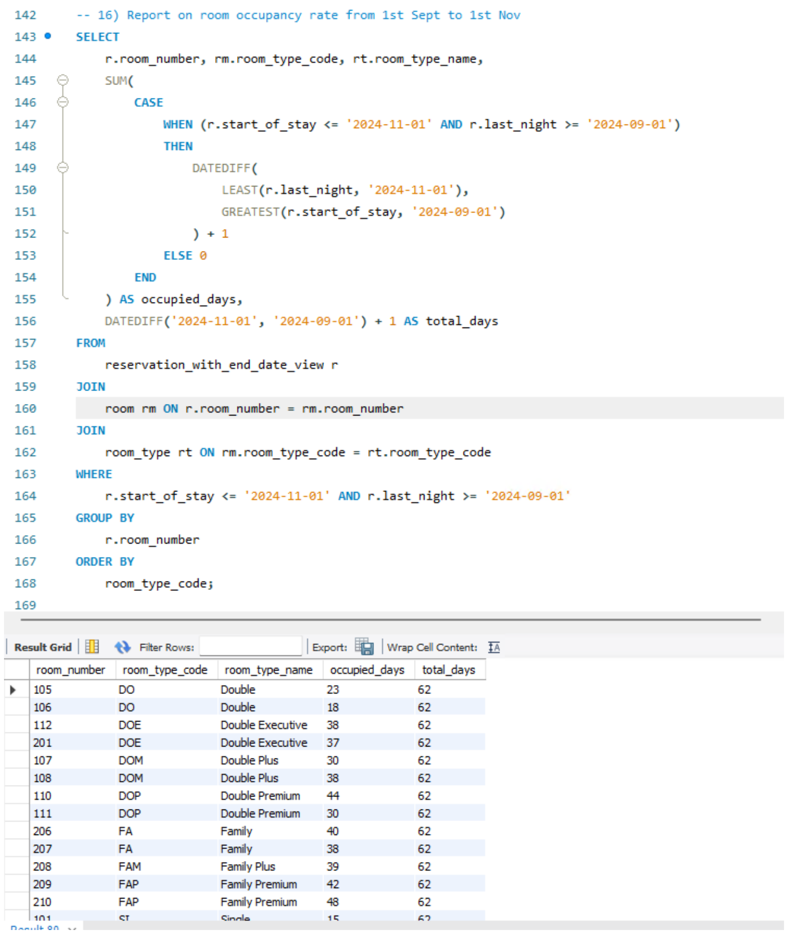
```sql
-- 15) Report on total revenue by type of room
SELECT
    rt.room_type_name,
    SUM(i.amount_paid) AS total_earnings
FROM
    invoice i
JOIN
    reservation r ON i.invoice_number = r.invoice_number
JOIN
    room rm ON r.room_number = rm.room_number
JOIN
    room_type rt ON rm.room_type_code = rt.room_type_code
GROUP BY
    rt.room_type_name
ORDER BY
    total_earnings DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

room_type_name	total_earnings
Suite Premium	6550.50
Family Premium	4222.75
Twin Executive	3651.25
Family	3610.00
Double Executive	3546.00
Suite Executive	3402.00
Double Premium	3179.00
Double Plus	2681.75
Family Plus	2266.00
Twin	1223.00
Double	1157.50
Single	834.75
Single Plus	648.75

16

Report on the occupancy of each room for a two month window.
LEAST and GREATEST were used to limit the number of days counted from a booking that partially overlaps the reporting dates.
SUCCESSFUL



```sql
-- 16) Report on room occupancy rate from 1st Sept to 1st Nov
SELECT
    r.room_number, rm.room_type_code, rt.room_type_name,
    SUM(
        CASE
            WHEN (r.start_of_stay <= '2024-11-01' AND r.last_night >= '2024-09-01')
            THEN
                DATEDIFF(
                    LEAST(r.last_night, '2024-11-01'),
                    GREATEST(r.start_of_stay, '2024-09-01')
                ) + 1
            ELSE 0
        END
    ) AS occupied_days,
    DATEDIFF('2024-11-01', '2024-09-01') + 1 AS total_days
FROM
    reservation_with_end_date_view r
JOIN
    room rm ON r.room_number = rm.room_number
JOIN
    room_type rt ON rm.room_type_code = rt.room_type_code
WHERE
    r.start_of_stay <= '2024-11-01' AND r.last_night >= '2024-09-01'
GROUP BY
    r.room_number
ORDER BY
    room_type_code;
```

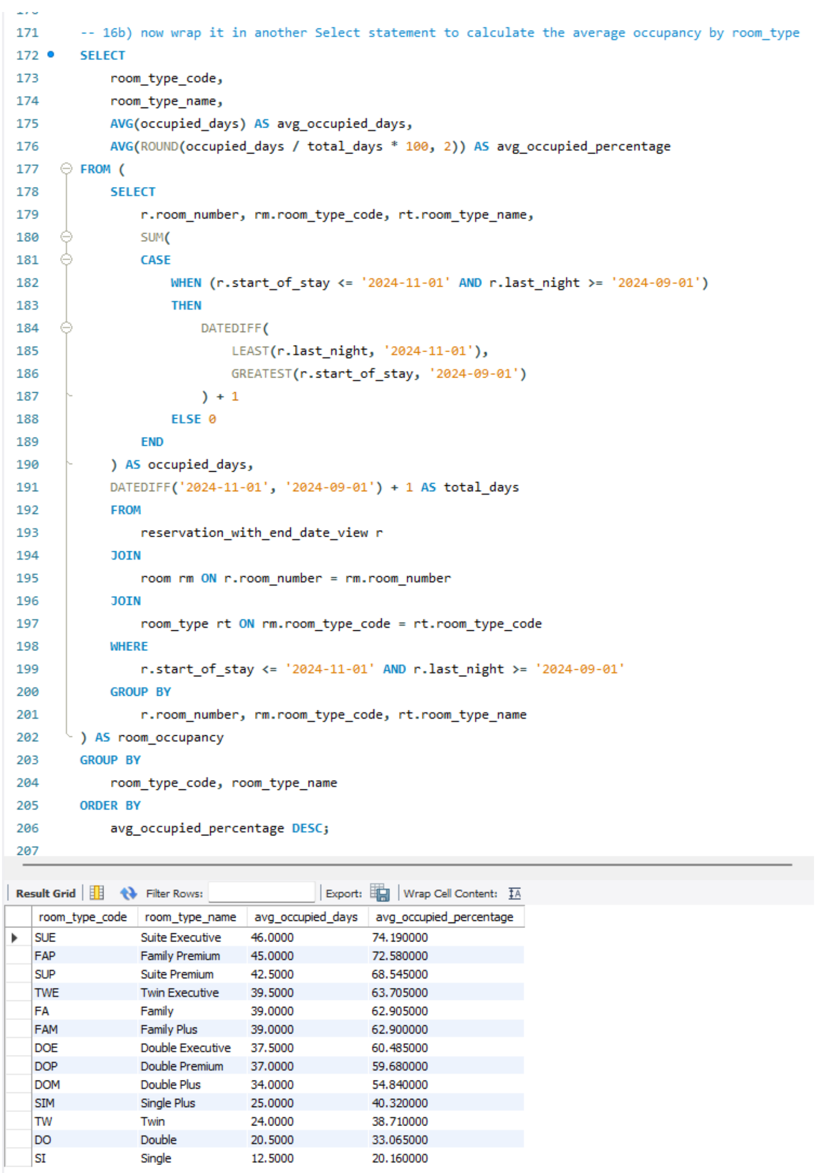
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

room_number	room_type_code	room_type_name	occupied_days	total_days
105	DO	Double	23	62
106	DO	Double	18	62
112	DOE	Double Executive	38	62
201	DOE	Double Executive	37	62
107	DOM	Double Plus	30	62
108	DOM	Double Plus	38	62
110	DOP	Double Premium	44	62
111	DOP	Double Premium	30	62
206	FA	Family	40	62
207	FA	Family	38	62
208	FAM	Family Plus	39	62
209	FAP	Family Premium	42	62
210	FAP	Family Premium	48	62
101	SI	Single	15	62

16b

Report room occupancy by room type and provide an average of the occupied days and the percentage of the total possible days that the rooms of those types were occupied.

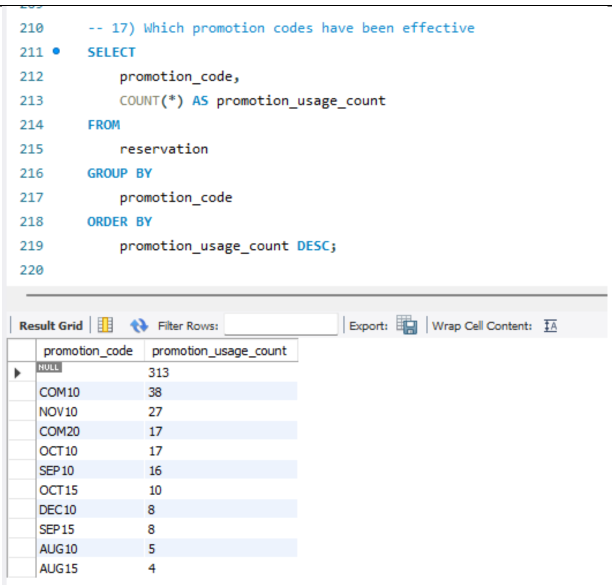
This is achieved by wrapping the (16) query in another query. AVG and ROUND are used to calculate the required values.
SUCCESSFUL



```sql
-- 16b) now wrap it in another Select statement to calculate the average occupancy by room_type
SELECT
    room_type_code,
    room_type_name,
    AVG(occupied_days) AS avg_occupied_days,
    AVG(ROUND(occupied_days / total_days * 100, 2)) AS avg_occupied_percentage
FROM (
    SELECT
        r.room_number, rm.room_type_code, rt.room_type_name,
        SUM(
        CASE
            WHEN (r.start_of_stay <= '2024-11-01' AND r.last_night >= '2024-09-01')
            THEN
                DATEDIFF(
                    LEAST(r.last_night, '2024-11-01'),
                    GREATEST(r.start_of_stay, '2024-09-01')
                ) + 1
            ELSE 0
        END
    ) AS occupied_days,
    DATEDIFF('2024-11-01', '2024-09-01') + 1 AS total_days
    FROM
        reservation_with_end_date_view r
    JOIN
        room rm ON r.room_number = rm.room_number
    JOIN
        room_type rt ON rm.room_type_code = rt.room_type_code
    WHERE
        r.start_of_stay <= '2024-11-01' AND r.last_night >= '2024-09-01'
    GROUP BY
        r.room_number, rm.room_type_code, rt.room_type_name
) AS room_occupancy
GROUP BY
    room_type_code, room_type_name
ORDER BY
    avg_occupied_percentage DESC;
```

room_type_code	room_type_name	avg_occupied_days	avg_occupied_percentage
SUE	Suite Executive	46.0000	74.190000
FAP	Family Premium	45.0000	72.580000
SUP	Suite Premium	42.5000	68.545000
TWE	Twin Executive	39.5000	63.705000
FA	Family	39.0000	62.905000
FAM	Family Plus	39.0000	62.900000
DOE	Double Executive	37.5000	60.485000
DOP	Double Premium	37.0000	59.680000
DOM	Double Plus	34.0000	54.840000
SIM	Single Plus	25.0000	40.320000
TW	Twin	24.0000	38.710000
DO	Double	20.5000	33.065000
SI	Single	12.5000	20.160000

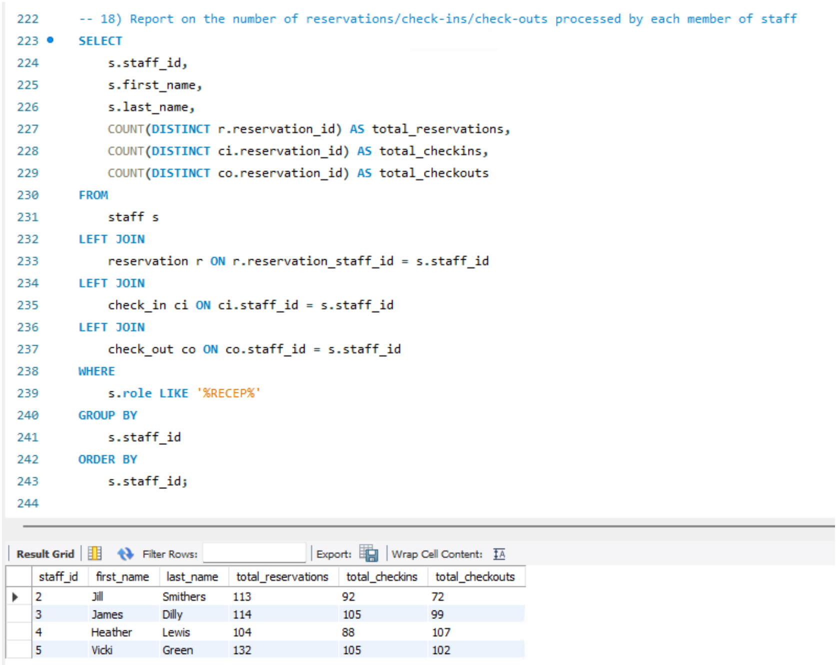
17 Report on which promotion codes have been most effective.
Results show that, as expected, no promotion code was used for many bookings, but that the 10% company discount was the most used.
SUCCESSFUL



```sql
-- 17) Which promotion codes have been effective
SELECT
    promotion_code,
    COUNT(*) AS promotion_usage_count
FROM
    reservation
GROUP BY
    promotion_code
ORDER BY
    promotion_usage_count DESC;
```

promotion_code	promotion_usage_count
NULL	313
COM10	38
NOV10	27
COM20	17
OCT10	17
SEP10	16
OCT15	10
DEC10	8
SEP15	8
AUG10	5
AUG15	4

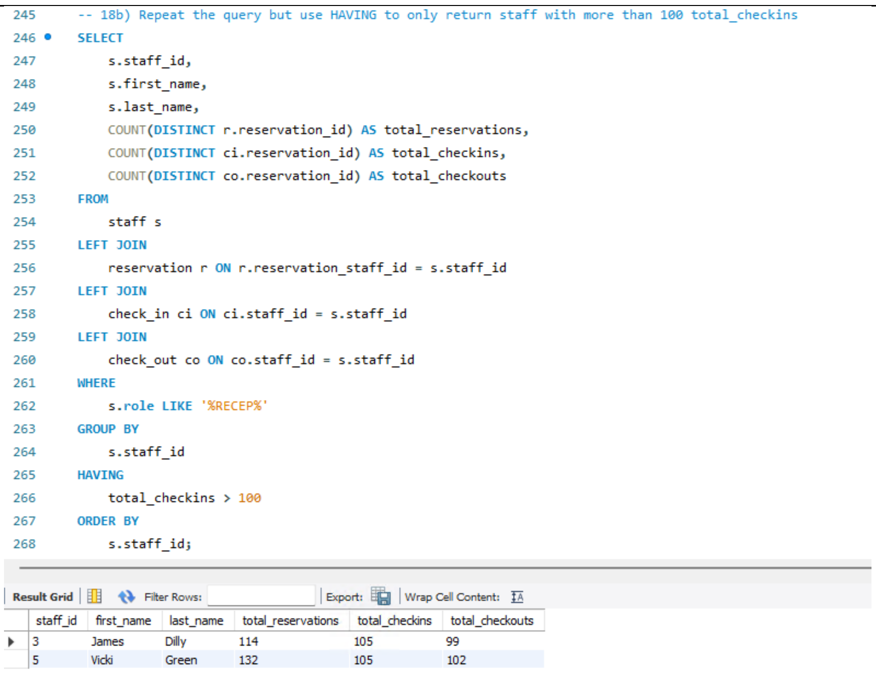
18 Report on the number of reservations, check-ins, check-outs have been processed by each member of staff. COUNT(DISTINCT) was used to achieve this. AS was used to name the counted values.
LEFT_JOIN was used to be sure that staff wouldn't be ignored from the results even if they hadn't processed any reservations. LIKE was used to make sure the member of staff was a type of receptionist. (Coronel, Morris, Rob, 2020)
SUCCESSFUL



```sql
-- 18) Report on the number of reservations/check-ins/check-outs processed by each member of staff
SELECT
    s.staff_id,
    s.first_name,
    s.last_name,
    COUNT(DISTINCT r.reservation_id) AS total_reservations,
    COUNT(DISTINCT ci.reservation_id) AS total_checkins,
    COUNT(DISTINCT co.reservation_id) AS total_checkouts
FROM
    staff s
LEFT JOIN
    reservation r ON r.reservation_staff_id = s.staff_id
LEFT JOIN
    check_in ci ON ci.staff_id = s.staff_id
LEFT JOIN
    check_out co ON co.staff_id = s.staff_id
WHERE
    s.role LIKE '%RECEP%'
GROUP BY
    s.staff_id
ORDER BY
    s.staff_id;
```

staff_id	first_name	last_name	total_reservations	total_checkins	total_checkouts
2	Jill	Smithers	113	92	72
3	James	Dilly	114	105	99
4	Heather	Lewis	104	88	107
5	Vicki	Green	132	105	102

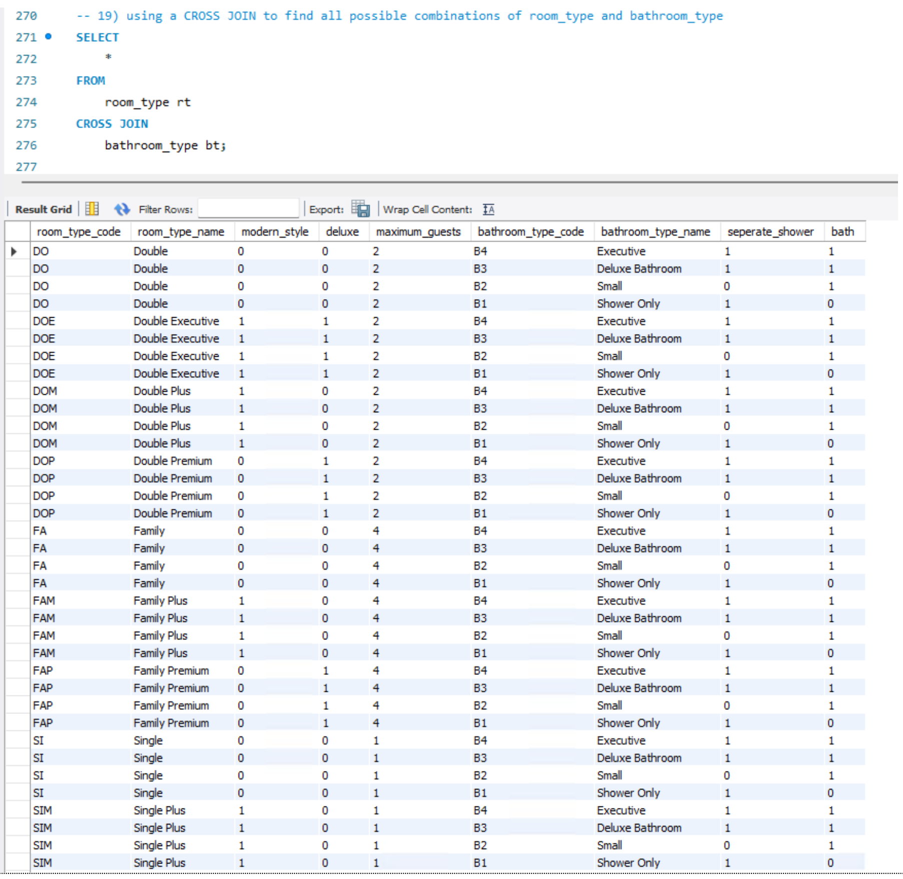
18b Similar to Query 18, but the results were filtered so that only staff processing more than 100 check-ins were shown. The HAVING feature was used to implement the greater than 100 filter.
SUCCESSFUL



```sql
-- 18b) Repeat the query but use HAVING to only return staff with more than 100 total_checkins
SELECT
    s.staff_id,
    s.first_name,
    s.last_name,
    COUNT(DISTINCT r.reservation_id) AS total_reservations,
    COUNT(DISTINCT ci.reservation_id) AS total_checkins,
    COUNT(DISTINCT co.reservation_id) AS total_checkouts
FROM
    staff s
LEFT JOIN
    reservation r ON r.reservation_staff_id = s.staff_id
LEFT JOIN
    check_in ci ON ci.staff_id = s.staff_id
LEFT JOIN
    check_out co ON co.staff_id = s.staff_id
WHERE
    s.role LIKE '%RECEP%'
GROUP BY
    s.staff_id
HAVING
    total_checkins > 100
ORDER BY
    s.staff_id;
```

staff_id	first_name	last_name	total_reservations	total_checkins	total_checkouts
3	James	Dilly	114	105	99
5	Vicki	Green	132	105	102

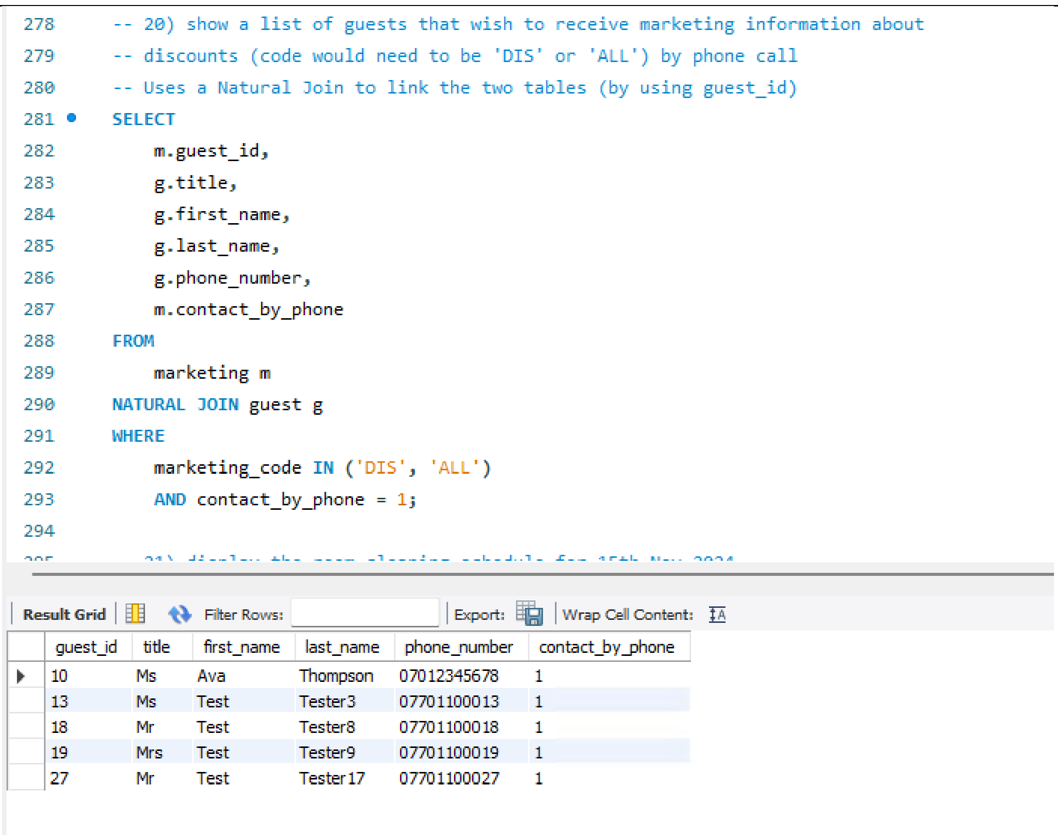
19 Used a CROSS JOIN to report on all possible combinations between room_type and bathroom_type.
SUCCESSFUL



```sql
-- 19) using a CROSS JOIN to find all possible combinations of room_type and bathroom_type
SELECT
    *
FROM
    room_type rt
CROSS JOIN
    bathroom_type bt;
```

room_type_code	room_type_name	modern_style	deluxe	maximum_guests	bathroom_type_code	bathroom_type_name	seperate_shower	bath
DO	Double	0	0	2	B4	Executive	1	1
DO	Double	0	0	2	B3	Deluxe Bathroom	1	1
DO	Double	0	0	2	B2	Small	0	1
DO	Double	0	0	2	B1	Shower Only	1	0
DOE	Double Executive	1	1	2	B4	Executive	1	1
DOE	Double Executive	1	1	2	B3	Deluxe Bathroom	1	1
DOE	Double Executive	1	1	2	B2	Small	0	1
DOE	Double Executive	1	1	2	B1	Shower Only	1	0
DOM	Double Plus	1	0	2	B4	Executive	1	1
DOM	Double Plus	1	0	2	B3	Deluxe Bathroom	1	1
DOM	Double Plus	1	0	2	B2	Small	0	1
DOM	Double Plus	1	0	2	B1	Shower Only	1	0
DOP	Double Premium	0	1	2	B4	Executive	1	1
DOP	Double Premium	0	1	2	B3	Deluxe Bathroom	1	1
DOP	Double Premium	0	1	2	B2	Small	0	1
DOP	Double Premium	0	1	2	B1	Shower Only	1	0
FA	Family	0	0	4	B4	Executive	1	1
FA	Family	0	0	4	B3	Deluxe Bathroom	1	1
FA	Family	0	0	4	B2	Small	0	1
FA	Family	0	0	4	B1	Shower Only	1	0
FAM	Family Plus	1	0	4	B4	Executive	1	1
FAM	Family Plus	1	0	4	B3	Deluxe Bathroom	1	1
FAM	Family Plus	1	0	4	B2	Small	0	1
FAM	Family Plus	1	0	4	B1	Shower Only	1	0
FAP	Family Premium	0	1	4	B4	Executive	1	1
FAP	Family Premium	0	1	4	B3	Deluxe Bathroom	1	1
FAP	Family Premium	0	1	4	B2	Small	0	1
FAP	Family Premium	0	1	4	B1	Shower Only	1	0
SI	Single	0	0	1	B4	Executive	1	1
SI	Single	0	0	1	B3	Deluxe Bathroom	1	1
SI	Single	0	0	1	B2	Small	0	1
SI	Single	0	0	1	B1	Shower Only	1	0
SIM	Single Plus	1	0	1	B4	Executive	1	1
SIM	Single Plus	1	0	1	B3	Deluxe Bathroom	1	1
SIM	Single Plus	1	0	1	B2	Small	0	1
SIM	Single Plus	1	0	1	B1	Shower Only	1	0

20 Report on guests that wish to receive a phone call about discounts available.
Uses a NATURAL JOIN to link the two tables as MySQL will realise that both have guest_id in common.
SUCCESSFUL

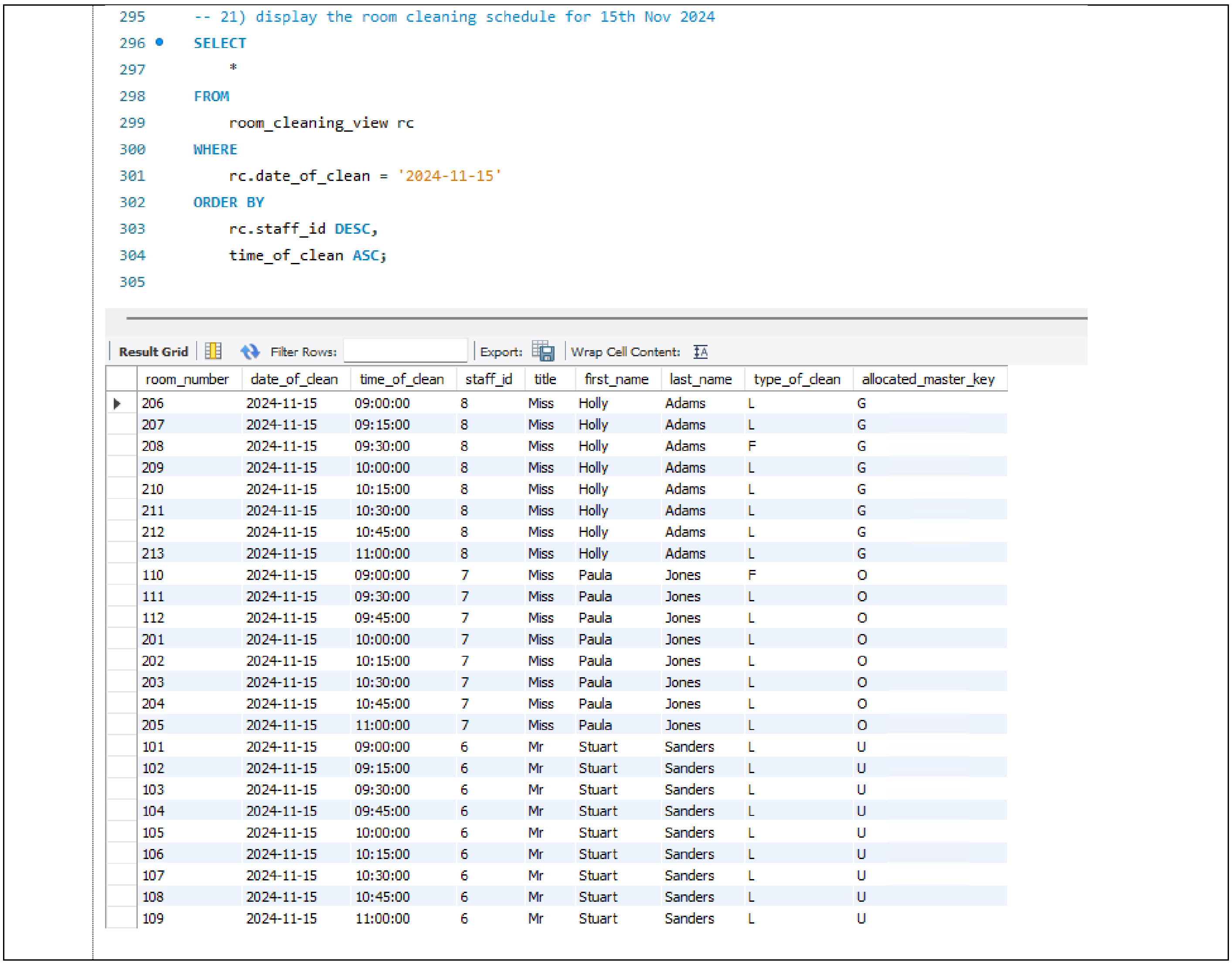


```sql
-- 20) show a list of guests that wish to receive marketing information about
-- discounts (code would need to be 'DIS' or 'ALL') by phone call
-- Uses a Natural Join to link the two tables (by using guest_id)
SELECT
    m.guest_id,
    g.title,
    g.first_name,
    g.last_name,
    g.phone_number,
    m.contact_by_phone
FROM
    marketing m
NATURAL JOIN guest g
WHERE
    marketing_code IN ('DIS', 'ALL')
    AND contact_by_phone = 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

guest_id	title	first_name	last_name	phone_number	contact_by_phone
10	Ms	Ava	Thompson	07012345678	1
13	Ms	Test	Tester3	07701100013	1
18	Mr	Test	Tester8	07701100018	1
19	Mrs	Test	Tester9	07701100019	1
27	Mr	Test	Tester17	07701100027	1

21	Show a room cleaning rota for a specific day. This is a query that the users of the database with a 'cleaner' role will be able to run as they are GRANTed to SELECT from the room_cleaning_view. SUCCESSFUL



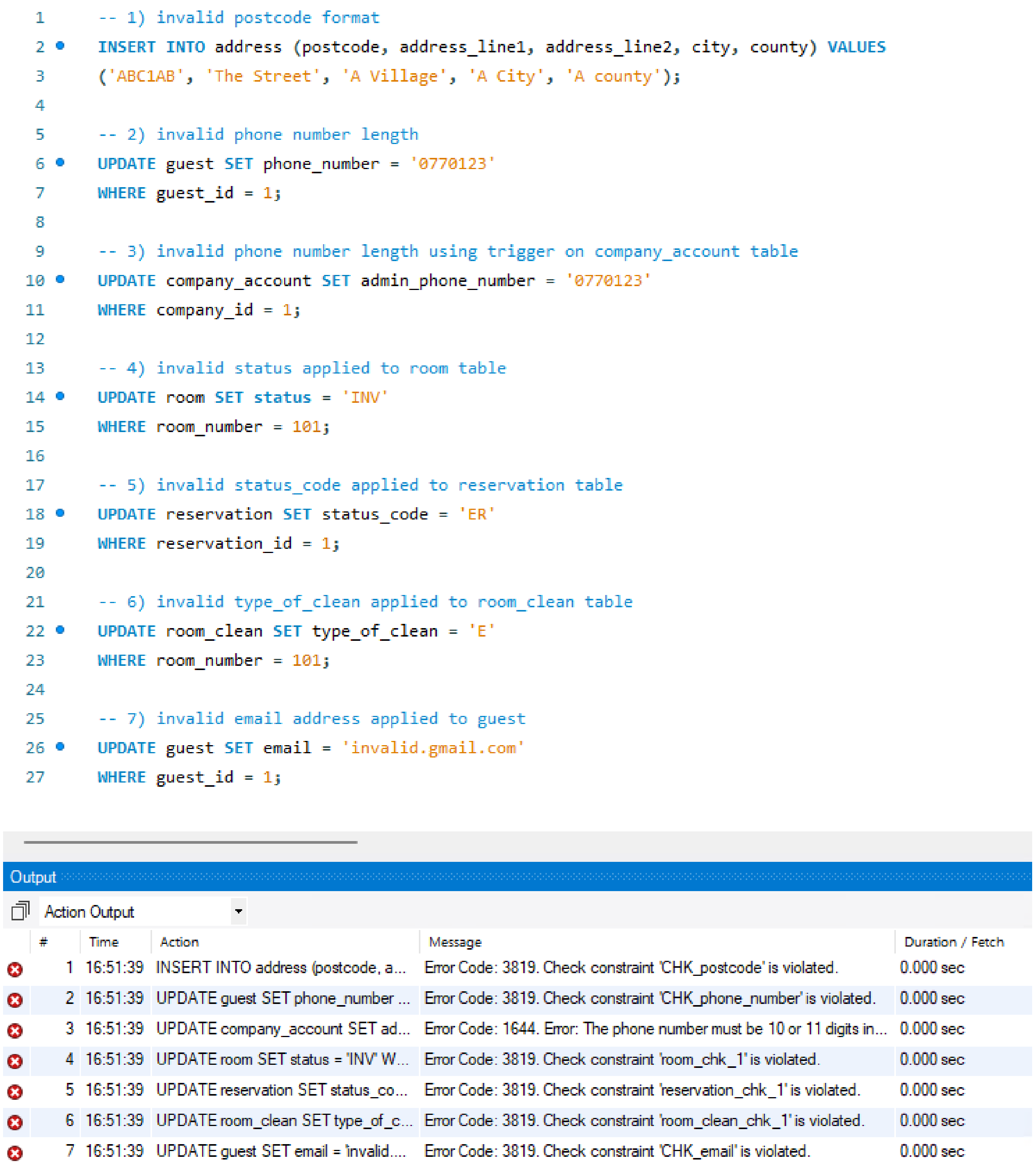
```sql
-- 21) display the room cleaning schedule for 15th Nov 2024
SELECT
    *
FROM
    room_cleaning_view rc
WHERE
    rc.date_of_clean = '2024-11-15'
ORDER BY
    rc.staff_id DESC,
    time_of_clean ASC;
```

room_number	date_of_clean	time_of_clean	staff_id	title	first_name	last_name	type_of_clean	allocated_master_key
206	2024-11-15	09:00:00	8	Miss	Holly	Adams	L	G
207	2024-11-15	09:15:00	8	Miss	Holly	Adams	L	G
208	2024-11-15	09:30:00	8	Miss	Holly	Adams	F	G
209	2024-11-15	10:00:00	8	Miss	Holly	Adams	L	G
210	2024-11-15	10:15:00	8	Miss	Holly	Adams	L	G
211	2024-11-15	10:30:00	8	Miss	Holly	Adams	L	G
212	2024-11-15	10:45:00	8	Miss	Holly	Adams	L	G
213	2024-11-15	11:00:00	8	Miss	Holly	Adams	L	G
110	2024-11-15	09:00:00	7	Miss	Paula	Jones	F	O
111	2024-11-15	09:30:00	7	Miss	Paula	Jones	L	O
112	2024-11-15	09:45:00	7	Miss	Paula	Jones	L	O
201	2024-11-15	10:00:00	7	Miss	Paula	Jones	L	O
202	2024-11-15	10:15:00	7	Miss	Paula	Jones	L	O
203	2024-11-15	10:30:00	7	Miss	Paula	Jones	L	O
204	2024-11-15	10:45:00	7	Miss	Paula	Jones	L	O
205	2024-11-15	11:00:00	7	Miss	Paula	Jones	L	O
101	2024-11-15	09:00:00	6	Mr	Stuart	Sanders	L	U
102	2024-11-15	09:15:00	6	Mr	Stuart	Sanders	L	U
103	2024-11-15	09:30:00	6	Mr	Stuart	Sanders	L	U
104	2024-11-15	09:45:00	6	Mr	Stuart	Sanders	L	U
105	2024-11-15	10:00:00	6	Mr	Stuart	Sanders	L	U
106	2024-11-15	10:15:00	6	Mr	Stuart	Sanders	L	U
107	2024-11-15	10:30:00	6	Mr	Stuart	Sanders	L	U
108	2024-11-15	10:45:00	6	Mr	Stuart	Sanders	L	U
109	2024-11-15	11:00:00	6	Mr	Stuart	Sanders	L	U

***Table 4** – Select script explanation*

6 Testing database constraints

The file ***invalid_data_to_test_constraints.sql*** is designed to fail and is used to prove the CHECK constraints are functional. Figure 3 shows the invalid data being used and the MySQL errors. Note #3 which uses the trigger to display a custom error message.



```sql
-- 1) invalid postcode format
INSERT INTO address (postcode, address_line1, address_line2, city, county) VALUES
('ABC1AB', 'The Street', 'A Village', 'A City', 'A county');

-- 2) invalid phone number length
UPDATE guest SET phone_number = '0770123'
WHERE guest_id = 1;

-- 3) invalid phone number length using trigger on company_account table
UPDATE company_account SET admin_phone_number = '0770123'
WHERE company_id = 1;

-- 4) invalid status applied to room table
UPDATE room SET status = 'INV'
WHERE room_number = 101;

-- 5) invalid status_code applied to reservation table
UPDATE reservation SET status_code = 'ER'
WHERE reservation_id = 1;

-- 6) invalid type_of_clean applied to room_clean table
UPDATE room_clean SET type_of_clean = 'E'
WHERE room_number = 101;

-- 7) invalid email address applied to guest
UPDATE guest SET email = 'invalid.gmail.com'
WHERE guest_id = 1;
```

Output

Action Output

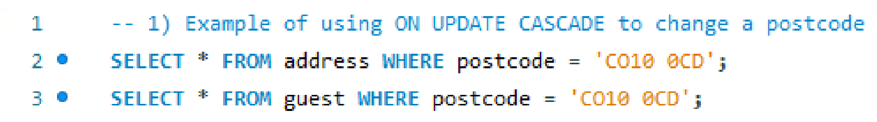
#	Time	Action	Message	Duration / Fetch
1	16:51:39	INSERT INTO address (postcode, a...	Error Code: 3819. Check constraint 'CHK_postcode' is violated.	0.000 sec
2	16:51:39	UPDATE guest SET phone_number ...	Error Code: 3819. Check constraint 'CHK_phone_number' is violated.	0.000 sec
3	16:51:39	UPDATE company_account SET ad...	Error Code: 1644. Error: The phone number must be 10 or 11 digits in...	0.000 sec
4	16:51:39	UPDATE room SET status = 'INV' W...	Error Code: 3819. Check constraint 'room_chk_1' is violated.	0.000 sec
5	16:51:39	UPDATE reservation SET status_co...	Error Code: 3819. Check constraint 'reservation_chk_1' is violated.	0.000 sec
6	16:51:39	UPDATE room_clean SET type_of_c...	Error Code: 3819. Check constraint 'room_clean_chk_1' is violated.	0.000 sec
7	16:51:39	UPDATE guest SET email = 'invalid....	Error Code: 3819. Check constraint 'CHK_email' is violated.	0.000 sec

***Figure 3** – Errors caught by CHECK constraints and validation triggers*

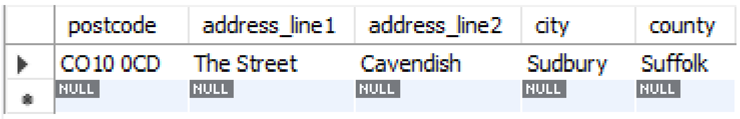
7 Advanced SQL features

The file ***advanced_script.sql*** tested various advanced features of MySQL beyond simple SELECT statements, including data insertion/deletion and table modification. Each stage of the tests is documented in Table 5.

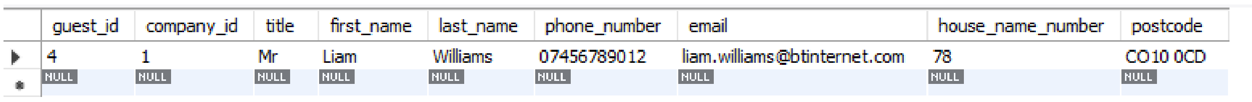
Test Id	**Purpose / Evidence / Comments**
1	Demonstrate the behaviour of an ON UPDATE CASCADE foreign key constraint. In this example the guest and address tables are used. SELECT statements are used to show the same postcode in both tables.



```sql
-- 1) Example of using ON UPDATE CASCADE to change a postcode
SELECT * FROM address WHERE postcode = 'CO10 0CD';
SELECT * FROM guest WHERE postcode = 'CO10 0CD';
```

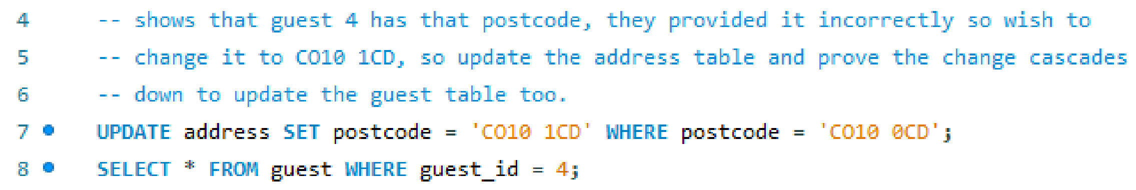


postcode	address_line1	address_line2	city	county
CO10 0CD	The Street	Cavendish	Sudbury	Suffolk
NULL	NULL	NULL	NULL	NULL

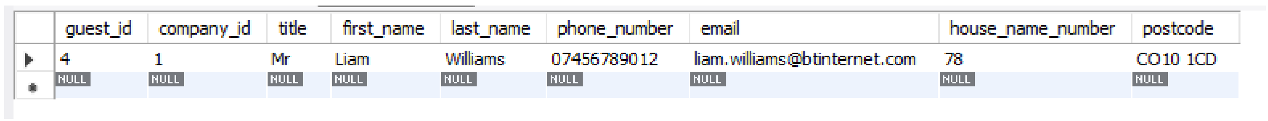


guest_id	company_id	title	first_name	last_name	phone_number	email	house_name_number	postcode
4	1	Mr	Liam	Williams	07456789012	liam.williams@btinternet.com	78	CO10 0CD
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

UPDATE with a WHERE clause is used to update the postcode in the address table and show how the UPDATE cascades to the guest table



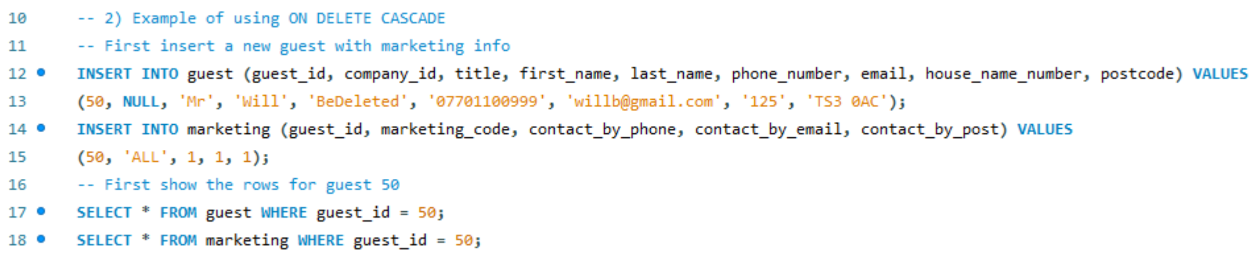
```sql
-- shows that guest 4 has that postcode, they provided it incorrectly so wish to
-- change it to CO10 1CD, so update the address table and prove the change cascades
-- down to update the guest table too.
UPDATE address SET postcode = 'CO10 1CD' WHERE postcode = 'CO10 0CD';
SELECT * FROM guest WHERE guest_id = 4;
```



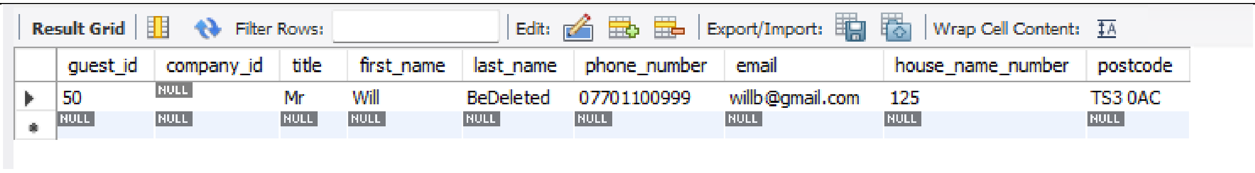
guest_id	company_id	title	first_name	last_name	phone_number	email	house_name_number	postcode
4	1	Mr	Liam	Williams	07456789012	liam.williams@btinternet.com	78	CO10 1CD
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

SUCCESSFUL

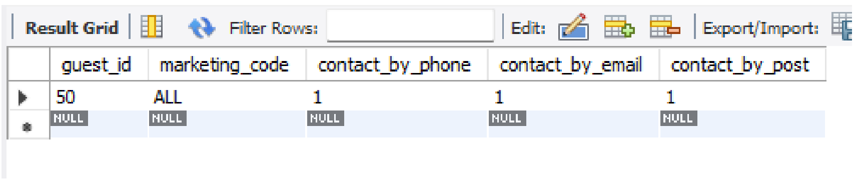
Test Id	**Purpose / Evidence / Comments**
2	Demonstrate the behaviour of an ON DELETE CASCADE foreign key constraint. In this example the guest and marketing tables are used. Guest 50 is INSERTed into both tables.



```sql
-- 2) Example of using ON DELETE CASCADE
-- First insert a new guest with marketing info
INSERT INTO guest (guest_id, company_id, title, first_name, last_name, phone_number, email, house_name_number, postcode) VALUES
(50, NULL, 'Mr', 'Will', 'BeDeleted', '07701100999', 'willb@gmail.com', '125', 'TS3 0AC');
INSERT INTO marketing (guest_id, marketing_code, contact_by_phone, contact_by_email, contact_by_post) VALUES
(50, 'ALL', 1, 1, 1);
-- First show the rows for guest 50
SELECT * FROM guest WHERE guest_id = 50;
SELECT * FROM marketing WHERE guest_id = 50;
```

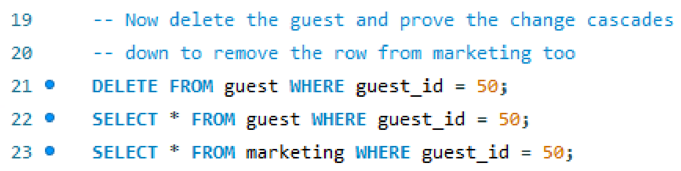


guest_id	company_id	title	first_name	last_name	phone_number	email	house_name_number	postcode
50	NULL	Mr	Will	BeDeleted	07701100999	willb@gmail.com	125	TS3 0AC
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

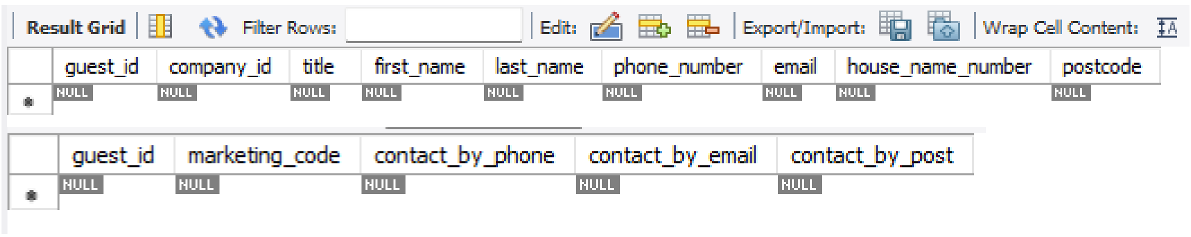


guest_id	marketing_code	contact_by_phone	contact_by_email	contact_by_post
50	ALL	1	1	1
NULL	NULL	NULL	NULL	NULL

Then DELETE FROM is used to delete from the guest table and prove the marketing row is also removed.



```
-- Now delete the guest and prove the change cascades
-- down to remove the row from marketing too
DELETE FROM guest WHERE guest_id = 50;
SELECT * FROM guest WHERE guest_id = 50;
SELECT * FROM marketing WHERE guest_id = 50;
```

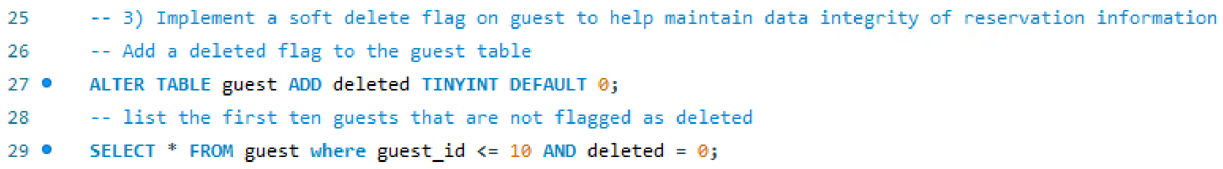


guest_id	company_id	title	first_name	last_name	phone_number	email	house_name_number	postcode
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

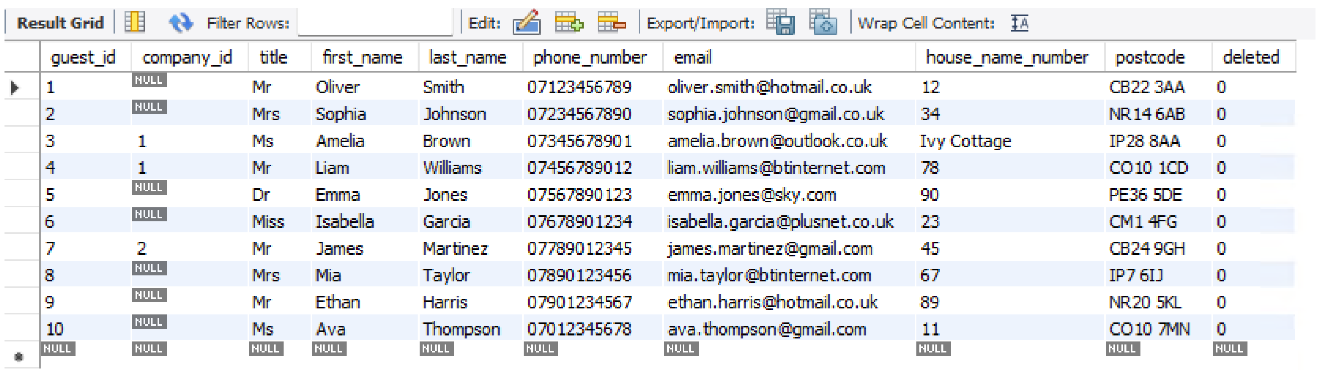
guest_id	marketing_code	contact_by_phone	contact_by_email	contact_by_post
NULL	NULL	NULL	NULL	NULL

SUCCESSFUL

3 Use ALTER TABLE to implement a Soft Delete flag on the guest table to help maintain data integrity of reservation information when a guest needs to be deleted.

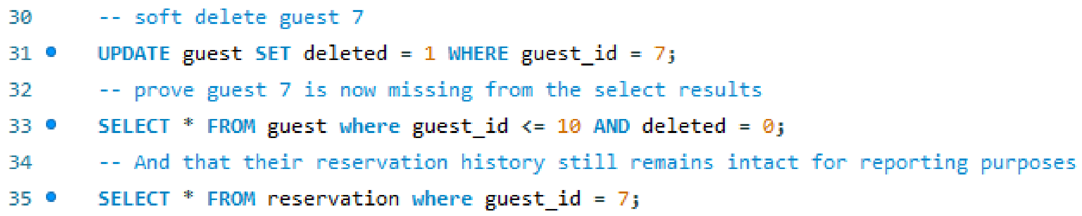


```
-- 3) Implement a soft delete flag on guest to help maintain data integrity of reservation information
-- Add a deleted flag to the guest table
ALTER TABLE guest ADD deleted TINYINT DEFAULT 0;
-- list the first ten guests that are not flagged as deleted
SELECT * FROM guest where guest_id <= 10 AND deleted = 0;
```

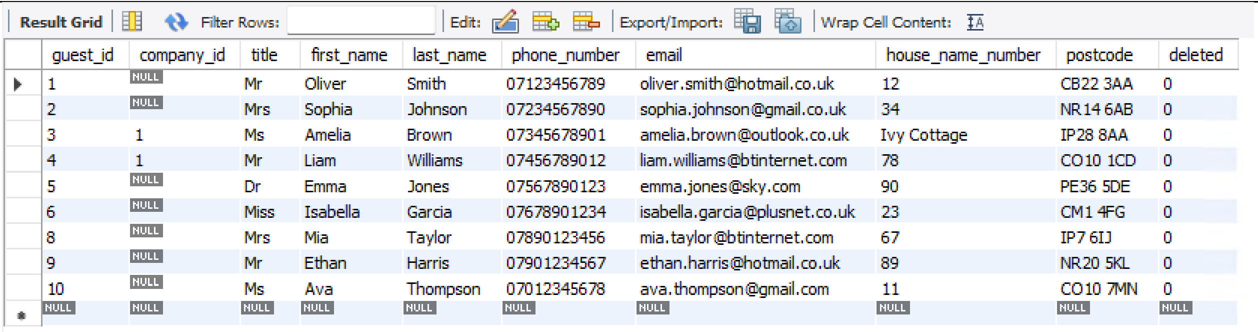


guest_id	company_id	title	first_name	last_name	phone_number	email	house_name_number	postcode	deleted
1	NULL	Mr	Oliver	Smith	07123456789	oliver.smith@hotmail.co.uk	12	CB22 3AA	0
2	NULL	Mrs	Sophia	Johnson	07234567890	sophia.johnson@gmail.co.uk	34	NR14 6AB	0
3	1	Ms	Amelia	Brown	07345678901	amelia.brown@outlook.co.uk	Ivy Cottage	IP28 8AA	0
4	1	Mr	Liam	Williams	07456789012	liam.williams@btinternet.com	78	CO10 1CD	0
5	NULL	Dr	Emma	Jones	07567890123	emma.jones@sky.com	90	PE36 5DE	0
6	NULL	Miss	Isabella	Garcia	07678901234	isabella.garcia@plusnet.co.uk	23	CM1 4FG	0
7	2	Mr	James	Martinez	07789012345	james.martinez@gmail.com	45	CB24 9GH	0
8	NULL	Mrs	Mia	Taylor	07890123456	mia.taylor@btinternet.com	67	IP7 6IJ	0
9	NULL	Mr	Ethan	Harris	07901234567	ethan.harris@hotmail.co.uk	89	NR20 5KL	0
10	NULL	Ms	Ava	Thompson	07012345678	ava.thompson@gmail.com	11	CO10 7MN	0
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

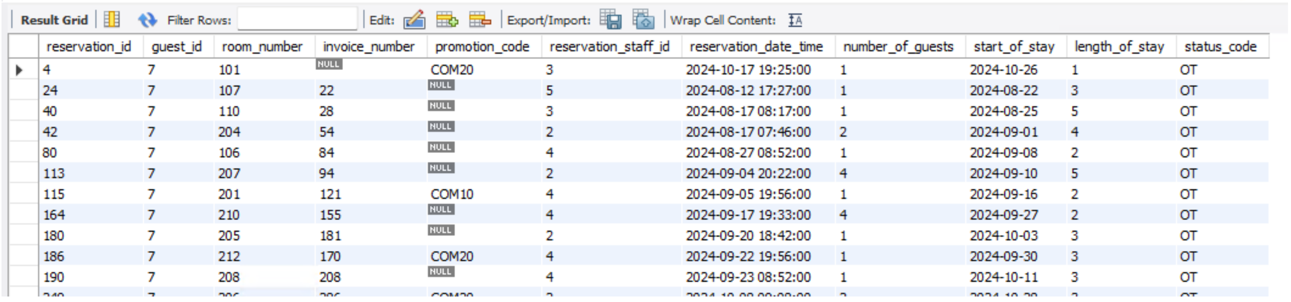
Soft delete guest 7 by using UPDATE and setting the deleted column to 1.
Use SELECT to prove guest 7 can be removed from the results and yet the reservations remain for reporting.



```
-- soft delete guest 7
UPDATE guest SET deleted = 1 WHERE guest_id = 7;
-- prove guest 7 is now missing from the select results
SELECT * FROM guest where guest_id <= 10 AND deleted = 0;
-- And that their reservation history still remains intact for reporting purposes
SELECT * FROM reservation where guest_id = 7;
```



	guest_id	company_id	title	first_name	last_name	phone_number	email	house_name_number	postcode	deleted
▶	1	NULL	Mr	Oliver	Smith	07123456789	oliver.smith@hotmail.co.uk	12	CB22 3AA	0
	2	NULL	Mrs	Sophia	Johnson	07234567890	sophia.johnson@gmail.co.uk	34	NR14 6AB	0
	3	1	Ms	Amelia	Brown	07345678901	amelia.brown@outlook.co.uk	Ivy Cottage	IP28 8AA	0
	4	1	Mr	Liam	Williams	07456789012	liam.williams@btinternet.com	78	CO10 1CD	0
	5	NULL	Dr	Emma	Jones	07567890123	emma.jones@sky.com	90	PE36 5DE	0
	6	NULL	Miss	Isabella	Garcia	07678901234	isabella.garcia@plusnet.co.uk	23	CM1 4FG	0
	8	NULL	Mrs	Mia	Taylor	07890123456	mia.taylor@btinternet.com	67	IP7 6IJ	0
	9	NULL	Mr	Ethan	Harris	07901234567	ethan.harris@hotmail.co.uk	89	NR20 5KL	0
	10	NULL	Ms	Ava	Thompson	07012345678	ava.thompson@gmail.com	11	CO10 7MN	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

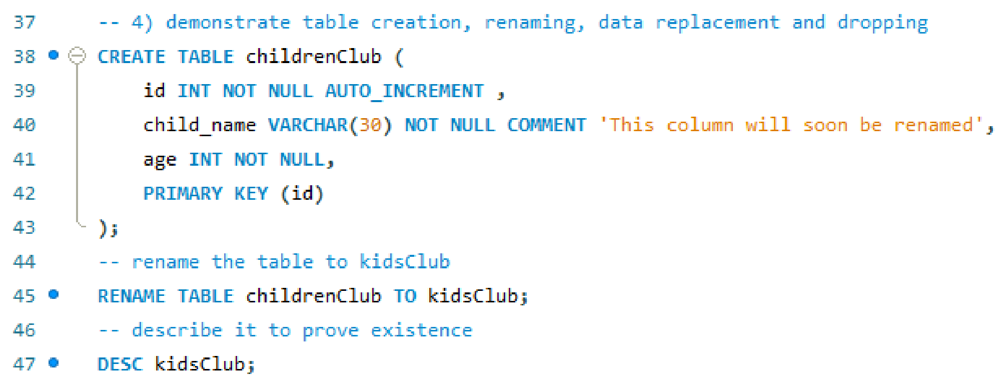


	reservation_id	guest_id	room_number	invoice_number	promotion_code	reservation_staff_id	reservation_date_time	number_of_guests	start_of_stay	length_of_stay	status_code
▶	4	7	101	NULL	COM20	3	2024-10-17 19:25:00	1	2024-10-26	1	OT
	24	7	107	22	NULL	5	2024-08-12 17:27:00	1	2024-08-22	3	OT
	40	7	110	28	NULL	3	2024-08-17 08:17:00	1	2024-08-25	5	OT
	42	7	204	54	NULL	2	2024-08-17 07:46:00	2	2024-09-01	4	OT
	80	7	106	84	NULL	4	2024-08-27 08:52:00	1	2024-09-08	2	OT
	113	7	207	94	NULL	2	2024-09-04 20:22:00	4	2024-09-10	5	OT
	115	7	201	121	COM10	4	2024-09-05 19:56:00	1	2024-09-16	2	OT
	164	7	210	155	NULL	4	2024-09-17 19:33:00	4	2024-09-27	2	OT
	180	7	205	181	NULL	2	2024-09-20 18:42:00	1	2024-10-03	3	OT
	186	7	212	170	COM20	4	2024-09-22 19:56:00	1	2024-09-30	3	OT
	190	7	208	208	NULL	4	2024-09-23 08:52:00	1	2024-10-11	3	OT
	[illegible]	7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

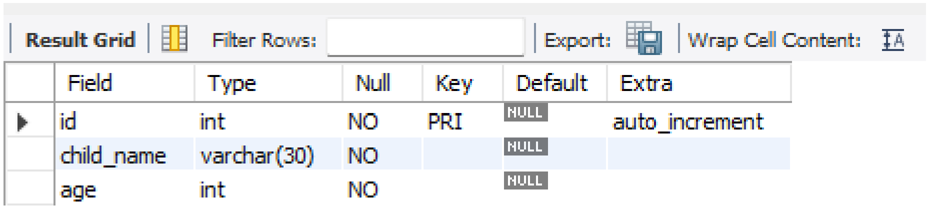
SUCCESSFUL

4 Demonstrate table creation, renaming, data replacement and dropping

A table is CREATEd and RENAMEd.

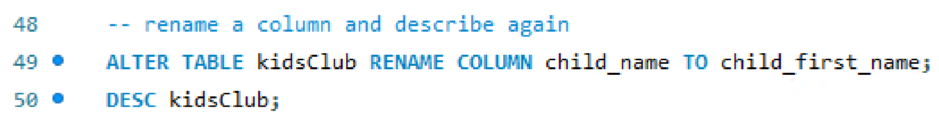


```
-- 4) demonstrate table creation, renaming, data replacement and dropping
CREATE TABLE childrenClub (
    id INT NOT NULL AUTO_INCREMENT ,
    child_name VARCHAR(30) NOT NULL COMMENT 'This column will soon be renamed',
    age INT NOT NULL,
    PRIMARY KEY (id)
);
-- rename the table to kidsClub
RENAME TABLE childrenClub TO kidsClub;
-- describe it to prove existence
DESC kidsClub;
```

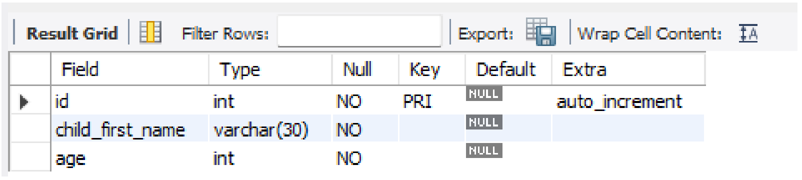


	Field	Type	Null	Key	Default	Extra
▶	id	int	NO	PRI	NULL	auto_increment
	child_name	varchar(30)	NO		NULL	
	age	int	NO		NULL	

The table is ALTERed to rename a column. DESC is used to describe the table.

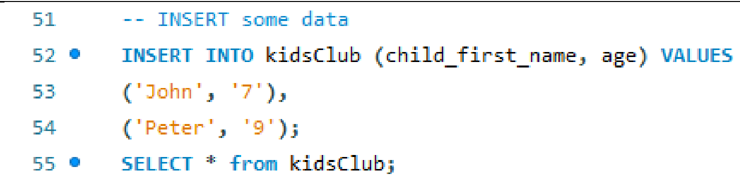


```
-- rename a column and describe again
ALTER TABLE kidsClub RENAME COLUMN child_name TO child_first_name;
DESC kidsClub;
```

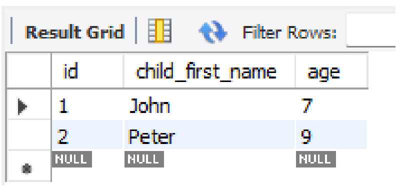


	Field	Type	Null	Key	Default	Extra
▶	id	int	NO	PRI	NULL	auto_increment
	child_first_name	varchar(30)	NO		NULL	
	age	int	NO		NULL	

Test data is INSERTed.



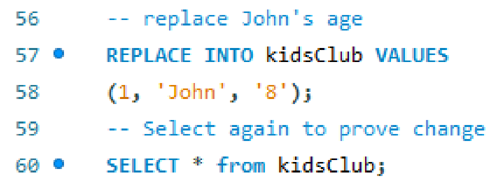
```sql
-- INSERT some data
INSERT INTO kidsClub (child_first_name, age) VALUES
('John', '7'),
('Peter', '9');
SELECT * from kidsClub;
```



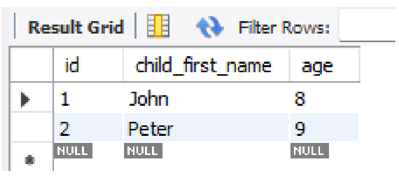
Result Grid | Filter Rows:

id	child_first_name	age
1	John	7
2	Peter	9
NULL	NULL	NULL

REPLACE is used to change an age value.



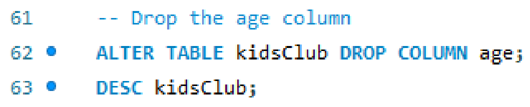
```sql
-- replace John's age
REPLACE INTO kidsClub VALUES
(1, 'John', '8');
-- Select again to prove change
SELECT * from kidsClub;
```



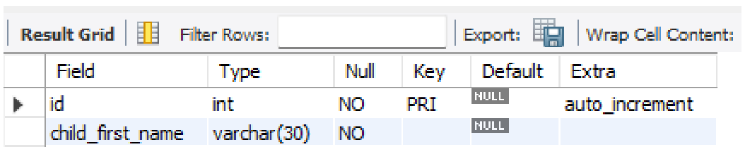
Result Grid | Filter Rows:

id	child_first_name	age
1	John	8
2	Peter	9
NULL	NULL	NULL

The age column is removed by using DROP.



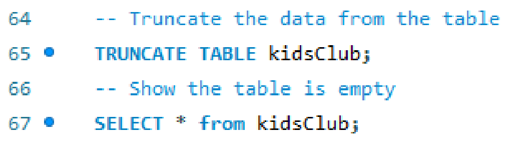
```sql
-- Drop the age column
ALTER TABLE kidsClub DROP COLUMN age;
DESC kidsClub;
```



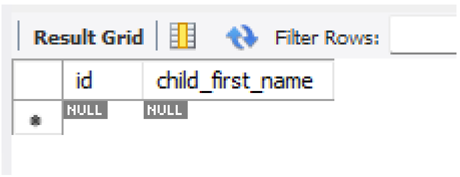
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
child_first_name	varchar(30)	NO		NULL	

TRUNCATE is used to empty the table and reset the auto increment



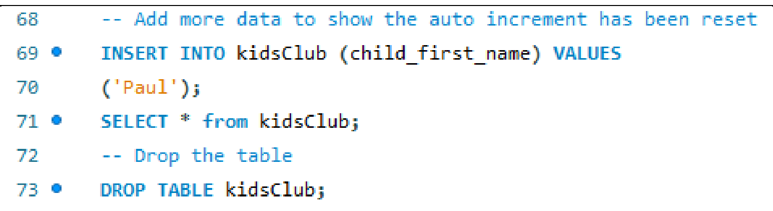
```sql
-- Truncate the data from the table
TRUNCATE TABLE kidsClub;
-- Show the table is empty
SELECT * from kidsClub;
```



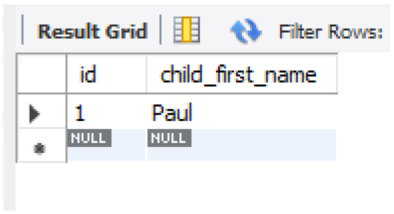
Result Grid | Filter Rows:

id	child_first_name
NULL	NULL

More data is added to show the id starts at 1 again and finally DROP is used to delete this demo table from the database.



```sql
-- Add more data to show the auto increment has been reset
INSERT INTO kidsClub (child_first_name) VALUES
('Paul');
SELECT * from kidsClub;
-- Drop the table
DROP TABLE kidsClub;
```

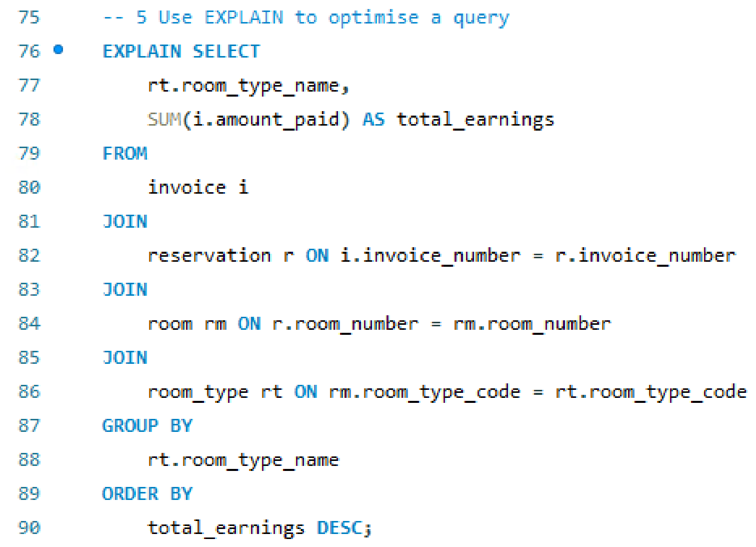


Result Grid | Filter Rows:

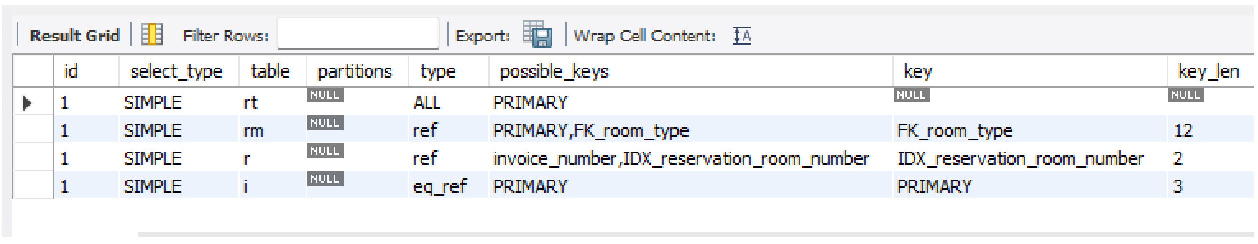
id	child_first_name
1	Paul
NULL	NULL

SUCCESSFUL

5 To optimise a query the EXPLAIN command is used.



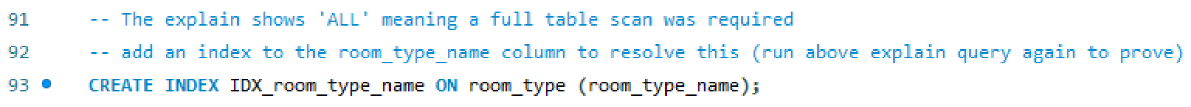
```sql
-- 5 Use EXPLAIN to optimise a query
EXPLAIN SELECT
    rt.room_type_name,
    SUM(i.amount_paid) AS total_earnings
FROM
    invoice i
JOIN
    reservation r ON i.invoice_number = r.invoice_number
JOIN
    room rm ON r.room_number = rm.room_number
JOIN
    room_type rt ON rm.room_type_code = rt.room_type_code
GROUP BY
    rt.room_type_name
ORDER BY
    total_earnings DESC;
```



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

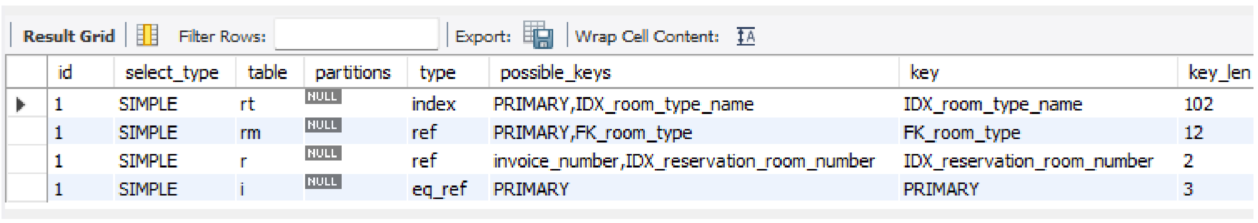
id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	rt	NULL	ALL	PRIMARY	NULL	NULL
1	SIMPLE	rm	NULL	ref	PRIMARY,FK_room_type	FK_room_type	12
1	SIMPLE	r	NULL	ref	invoice_number,IDX_reservation_room_number	IDX_reservation_room_number	2
1	SIMPLE	i	NULL	eq_ref	PRIMARY	PRIMARY	3

The 'ALL' in the first row of the results shows that a full table scan was required, so add an index to the room_type_name column to improve the performance.



```sql
-- The explain shows 'ALL' meaning a full table scan was required
-- add an index to the room_type_name column to resolve this (run above explain query again to prove)
CREATE INDEX IDX_room_type_name ON room_type (room_type_name);
```

Running EXPLAIN again shows that the index is being used.



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

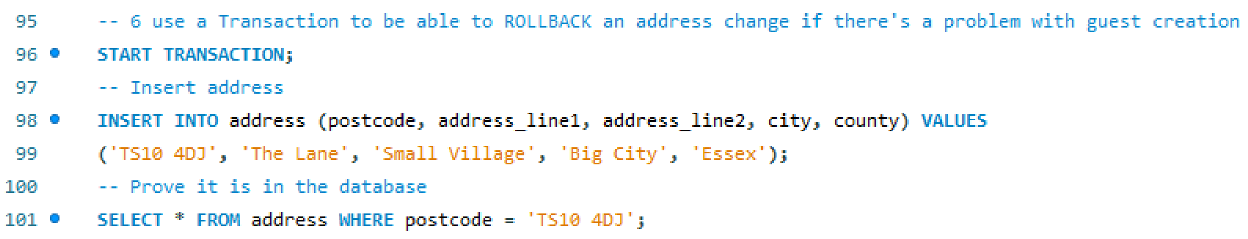
id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	rt	NULL	index	PRIMARY,IDX_room_type_name	IDX_room_type_name	102
1	SIMPLE	rm	NULL	ref	PRIMARY,FK_room_type	FK_room_type	12
1	SIMPLE	r	NULL	ref	invoice_number,IDX_reservation_room_number	IDX_reservation_room_number	2
1	SIMPLE	i	NULL	eq_ref	PRIMARY	PRIMARY	3

SUCCESSFUL

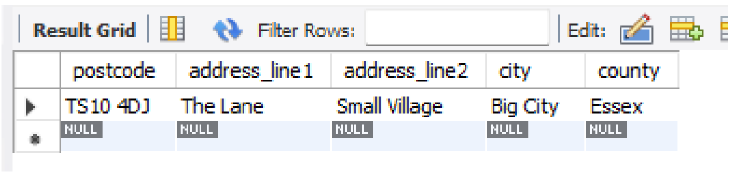
6

Demonstrate the use of Transaction to allow an address change to be rolled back if there is a problem with the creation of the guest record.

First the transaction is started and the new address added.

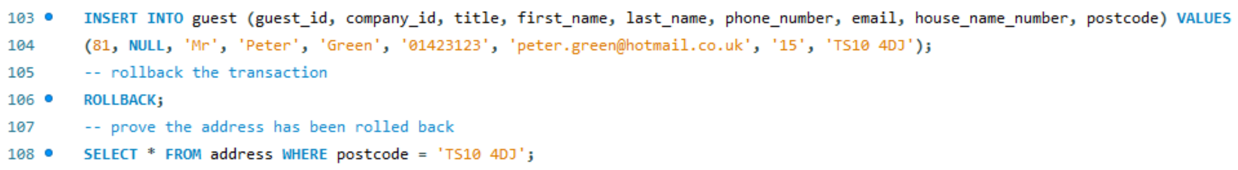


```sql
-- 6 use a Transaction to be able to ROLLBACK an address change if there's a problem with guest creation
START TRANSACTION;
-- Insert address
INSERT INTO address (postcode, address_line1, address_line2, city, county) VALUES
('TS10 4DJ', 'The Lane', 'Small Village', 'Big City', 'Essex');
-- Prove it is in the database
SELECT * FROM address WHERE postcode = 'TS10 4DJ';
```

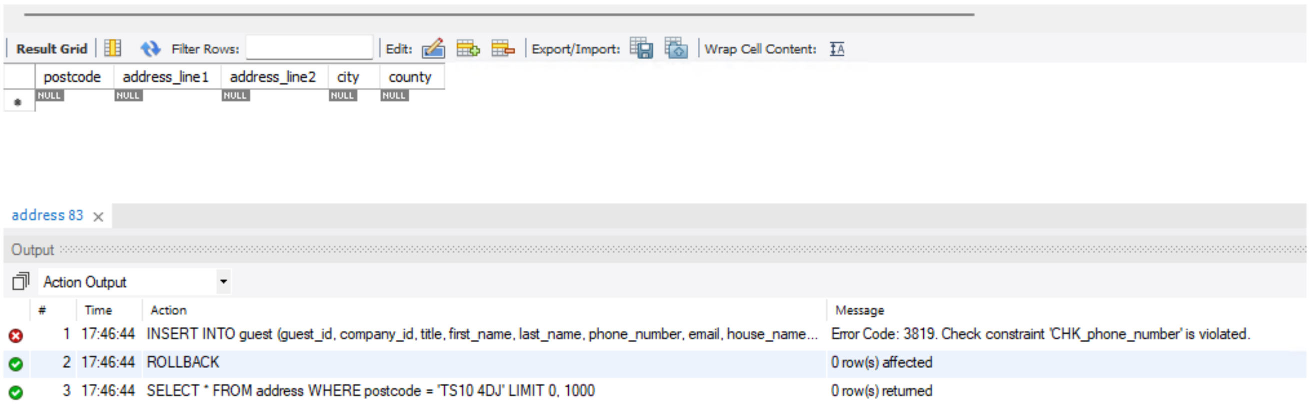


```
Result Grid | Filter Rows: | Edit:
   postcode   address_line1   address_line2   city       county
▶  TS10 4DJ   The Lane        Small Village   Big City   Essex
*  NULL       NULL            NULL            NULL       NULL
```

Then a guest with an invalid phone number is inserted (MySQL shows an error) so the ROLLBACK command is used. Finally, the address table is searched to prove that the address that was added has been rolled back.



```sql
INSERT INTO guest (guest_id, company_id, title, first_name, last_name, phone_number, email, house_name_number, postcode) VALUES
(81, NULL, 'Mr', 'Peter', 'Green', '01423123', 'peter.green@hotmail.co.uk', '15', 'TS10 4DJ');
-- rollback the transaction
ROLLBACK;
-- prove the address has been rolled back
SELECT * FROM address WHERE postcode = 'TS10 4DJ';
```



```
Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:
   postcode   address_line1   address_line2   city   county
*  NULL       NULL            NULL            NULL   NULL

address 83 ×
Output
Action Output
   #  Time      Action                                                                                                Message
✗  1  17:46:44  INSERT INTO guest (guest_id, company_id, title, first_name, last_name, phone_number, email, house_name...  Error Code: 3819. Check constraint 'CHK_phone_number' is violated.
✓  2  17:46:44  ROLLBACK                                                                                              0 row(s) affected
✓  3  17:46:44  SELECT * FROM address WHERE postcode = 'TS10 4DJ' LIMIT 0, 1000                                       0 row(s) returned
```

SUCCESSFUL

***Table 5** – Advanced Feature tests*

8 Deployment considerations

If the database was deployed in a real hotel further consideration and enhancement would be required to address the following points:

Software - MySQL Enterprise Edition is recommended for additional performance monitoring and security features.

Concurrency Control - ensures that multiple users can access and modify the hotel database simultaneously without data conflicts or integrity issues. E.g. when two receptionists attempt to assign the last available room, mechanisms like locking can be used to ensure only one transaction succeeds preventing double-booking.

Encryption – The database contains information about its guests. Data in specific columns could be encrypted using AES_ENCRYPT & AES_DECRYPT (Elmasri & Navathe, 2016). MySQL Enterprise Edition allows database files to be fully encrypted using MySQL Enterprise Transparent Data Encryption (MySQL-TDE, 2024)

Backups - contents of the database is vital for the hotel business, so regular encrypted backups must be taken and kept off-site. (Bradford, 2012). Full backups could be made each week using mysqldump. Incremental backups, that record changes since full backup could be captured daily or hourly. This can be achieved by capturing the mySQL binary logs; restored using mysqlbinlog.

Database maintenance – It will be important to follow best practice advice and regularly check logs and apply database patches. Performance monitoring tools can be used to check for CPU or Storage bottlenecks (GeeksForGeeks, 2024). Individual queries could be optimised using EXPLAIN and applying indexes.

Partitioning – can be used to improve performance as the database becomes large over time. It splits-up the data in the tables. Horizontal partitioning divides the table rows based on conditions (so reservations could be divided by year). Vertical partitioning splits a table into two smaller tables by grouping columns, the primary key being shared across partitions.

Security - This implementation already makes use of roles & granted access control, but more refinement may be required. Audit logging can be used to detect potential security breaches. Tools like MySQL's AUDIT plugin can be configured to log activity by user/timestamp.

GDPR (Data Protection Act, 2018) - If a guest requests deletion from the database, a soft delete will not suffice, and a hard delete would harm reporting and data integrity. A solution is that their data be overwritten with dummy values to removes their details but maintain data integrity.

PCI DSS (PCI Security Standards Council, 2024) – care needs to be taken when handling payment information. PCI defines the safeguards companies must use. This database does not hold payment information. A third-party payment processor (such as Stripe) will need to be used who themselves will be compliant with the required security standards.

9 Conclusion

Reflecting back on this project, I feel it has given me valuable experience in database implementation and the use of MySQL Workbench. I significantly underestimated the amount of time the design would take to implement and test, however the end result is good and meets all the objectives of the initial design.

If this database was to be used in a production environment I would want to monitor the performance of each of the queries and optimise as required. I would need to gain experience of the Enterprise Edition of MySQL Workbench to be confident using it in production.

References

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Database System Concepts (6th ed.). McGraw-Hill.

Connolly, T., & Begg, C. (2015). Database Systems: A Practical Approach to Design, Implementation, and Management (6th ed.). Pearson Education.

MySQL (8.0) Documentation. (n.d.). *MySQL 8.0 Reference Manual*. Retrieved from https://dev.mysql.com/doc/refman/8.0/en/

Elmasri, R., & Navathe, S. (2016). Fundamentals of Database Systems (7th ed.). Pearson.

Coronel, C., Morris, S., & Rob, P. (2020). *Database Systems: Design, Implementation, & Management*. 13th ed. Cengage Learning.

MySQL Enterprise Transparent Data Encryption (MySQL-TDE, 2024) https://www.mysql.com/products/enterprise/tde.html

Bradford, R. (2012). Effective *MySQL Backup and Recovery (Oracle Press)*. McGraw Hill.

GeeksForGeeks, 2024). MySQL Database Maintenance Best Practices. https://www.geeksforgeeks.org/mysql-database-maintenance-best-practices/

Data Protection Act (2018). *UK General Data Protection Regulation (GDPR)*. Available at https://www.gov.uk/data-protection

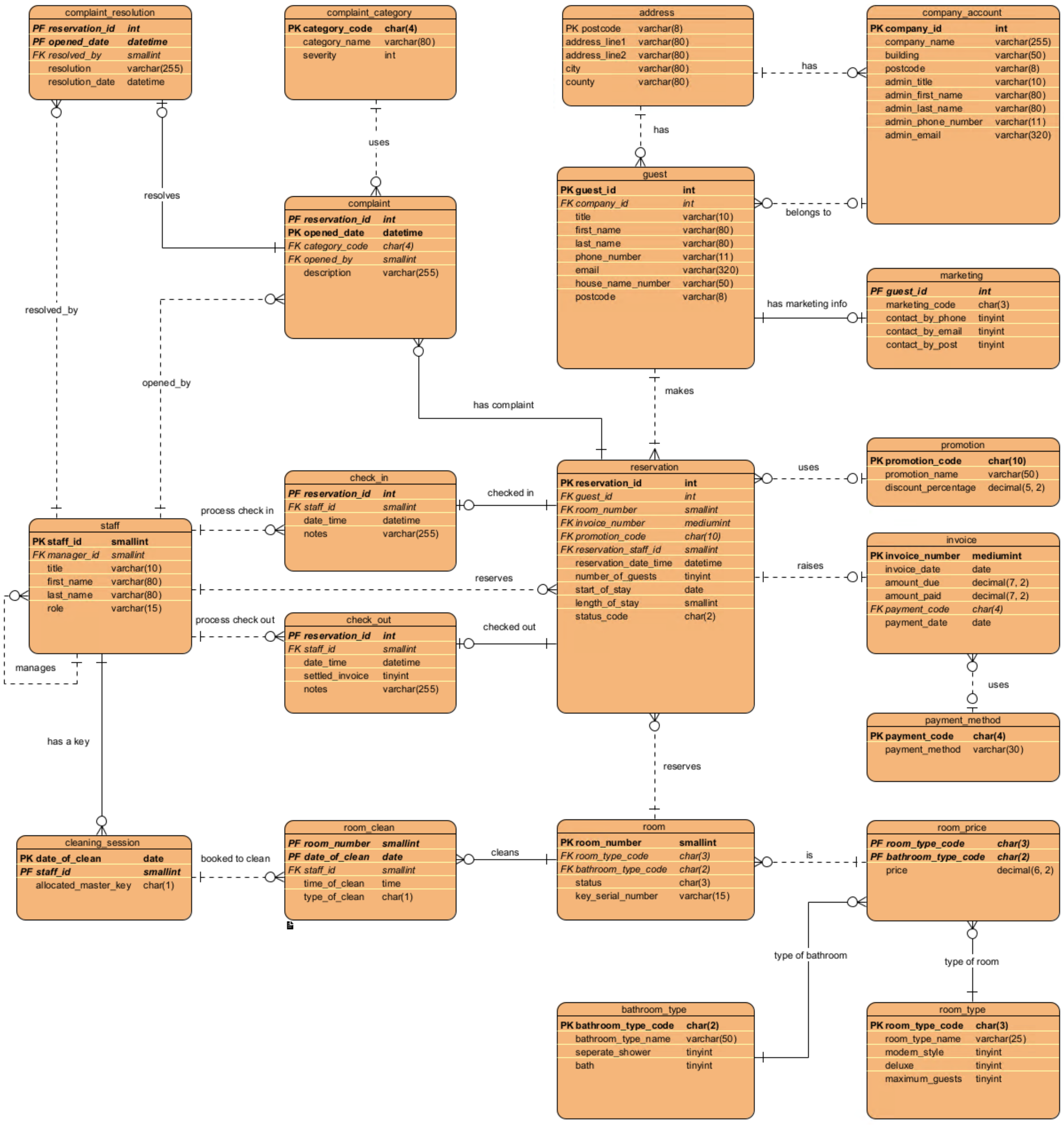
PCI Security Standards Council (2024). *PCI DSS Requirements and Standards*. Retrieved from https://pcisecuritystandards.org/

Appendices

Appendix A – Updated Physical Design Model

This updated model shows:

- the invoice table using the new payment_method table
- postcode columns having a length of 8 characters.



complaint_resolution
PF reservation_id int
PF opened_date datetime
FK resolved_by smallint
resolution varchar(255)
resolution_date datetime
complaint_category
PK category_code char(4)
category_name varchar(80)
severity int
address
PK postcode varchar(8)
address_line1 varchar(80)
address_line2 varchar(80)
city varchar(80)
county varchar(80)
company_account
PK company_id int
company_name varchar(255)
building varchar(50)
postcode varchar(8)
admin_title varchar(10)
admin_first_name varchar(80)
admin_last_name varchar(80)
admin_phone_number varchar(11)
admin_email varchar(320)
has
uses
has
resolves
complaint
PF reservation_id int
PK opened_date datetime
FK category_code char(4)
FK opened_by smallint
description varchar(255)
guest
PK guest_id int
FK company_id int
title varchar(10)
first_name varchar(80)
last_name varchar(80)
phone_number varchar(11)
email varchar(320)
house_name_number varchar(50)
postcode varchar(8)
belongs to
marketing
PF guest_id int
marketing_code char(3)
contact_by_phone tinyint
contact_by_email tinyint
contact_by_post tinyint
has marketing info
resolved_by
opened_by
has complaint
makes
promotion
PK promotion_code char(10)
promotion_name varchar(50)
discount_percentage decimal(5, 2)
uses
check_in
PF reservation_id int
FK staff_id smallint
date_time datetime
notes varchar(255)
checked in
process check in
reservation
PK reservation_id int
FK guest_id int
FK room_number smallint
FK invoice_number mediumint
FK promotion_code char(10)
FK reservation_staff_id smallint
reservation_date_time datetime
number_of_guests tinyint
start_of_stay date
length_of_stay smallint
status_code char(2)
staff
PK staff_id smallint
FK manager_id smallint
title varchar(10)
first_name varchar(80)
last_name varchar(80)
role varchar(15)
reserves
invoice
PK invoice_number mediumint
invoice_date date
amount_due decimal(7, 2)
amount_paid decimal(7, 2)
FK payment_code char(4)
payment_date date
raises
process check out
check_out
PF reservation_id int
FK staff_id smallint
date_time datetime
settled_invoice tinyint
notes varchar(255)
checked out
manages
uses
payment_method
PK payment_code char(4)
payment_method varchar(30)
has a key
reserves
cleaning_session
PK date_of_clean date
PF staff_id smallint
allocated_master_key char(1)
booked to clean
room_clean
PF room_number smallint
PF date_of_clean date
FK staff_id smallint
time_of_clean time
type_of_clean char(1)
cleans
room
PK room_number smallint
FK room_type_code char(3)
FK bathroom_type_code char(2)
status char(3)
key_serial_number varchar(15)
is
room_price
PF room_type_code char(3)
PF bathroom_type_code char(2)
price decimal(6, 2)
type of bathroom
type of room
bathroom_type
PK bathroom_type_code char(2)
bathroom_type_name varchar(50)
seperate_shower tinyint
bath tinyint
room_type
PK room_type_code char(3)
room_type_name varchar(25)
modern_style tinyint
deluxe tinyint
maximum_guests tinyint