DV1664

Tobias Gustafsson

9 maj 2023

Innehåll

1 Introduction **4**
- 1.1 File-Based Systems . . . 4
- 1.2 Database and DBMS . . . 4
- 1.3 Roles in the database environment . . . 4
- 1.4 Advantages and disadvantages of DMBS . . . 4
- 1.5 Three-Level Architecture . . . 4

2 The Relational Model **4**
- 2.1 Mathematical definition of relation . . . 5
- 2.2 Properties of relations . . . 5
- 2.3 Relational keys . . . 5
- 2.4 Integrity constraints . . . 6
- 2.5 Views . . . 6

3 Relational Algebra **6**

4 MySQL SQL **7**
- 4.1 Range search condition . . . 8
- 4.2 Set membership . . . 8
- 4.3 Pattern matching . . . 8
- 4.4 NULL search condition . . . 9
- 4.5 Ordering . . . 9
- 4.6 GROUP BY . . . 9
- 4.7 Aggregated statements . . . 9
- 4.8 ANY and ALL . . . 10
- 4.9 Join operations . . . 10
- 4.10 EXISTS . . . 10
- 4.11 INSERT, UPDATE and DELETE . . . 10
- 4.12 ISO SQL data types . . . 11

5 Tables **11**
- 5.1 CREATE TABLE . . . 11
- 5.2 ALTER TABLE . . . 11
- 5.3 DROP TABLE . . . 12

6 Views **12**

7 Laborations **12**
- 7.1 Lab 1 - SQL . . . 12
  - 7.1.1 Tables . . . 12
  - 7.1.2 Selection, Projection and Restriction . . . 14
  - 7.1.3 Aggregated Functions . . . 15
  - 7.1.4 Joins . . . 16
  - 7.1.5 Nested Queries . . . 17
  - 7.1.6 IN . . . 17
  - 7.1.7 BETWEEN . . . 18

7.1.8 DELETE, UPDATE, ALTER & INSERT 18
7.1.9 VIEW 19
7.1.10 DROP 20
7.2 Lab 2 and 3 - Triggers, Procedures, and Functions 21
7.2.1 User-Defined Functions 21
7.2.2 Stored Procedures 22
7.2.3 Triggers 23

1 Introduction

1.1 File-Based Systems

A file-based system is a collection of programs that perform services for the en users. Each program defines and manages its own data. There are some limitations to a file-based approach, mainly separation and isolation of data, where each program maintains its own set of data. Also there is a risk of duplication of data, where the same data is held by different programs. Moreover, programs are written in different languages, and cannor easily access each other's files.

1.2 Database and DBMS

A database is a shared organized collection of logically related data. A **database management system** (**DBMS**) enables users to create, maintain and control access to databases.

1.3 Roles in the database environment

Roles in the database environment are Data Administrator (DA), Database Administrator (DBA), Database Designers (Logical and Physical), Application Programmers and End Users.

1.4 Advantages and disadvantages of DMBS

Some advantages of DMBS includes control of data redundancy, data consistency, more information from the same amount of data, sharing of data, improved data integrity, improved security, enforcement of standards and economy of scale. Disadvantages of DMBS are complexity, size, additional hardware costs, cost of conversion, performance and higher impact of failure.

1.5 Three-Level Architecture

The three-level architecture is a framework for database design and management that separates the database system into three levels: external level, conceptual level and internal level. An objective of the three-level architecture is that users should not need to know physical database storage details.

2 The Relational Model

The **relational model** (**RM**) was introduced in 1970 by computer scientist Edgar F. Codd. In a relational model, all data is represented in terms of tuples, grouped into relations. A relational model organizes data into one ore more tables (also **relations**) of columns and rows, with a unique key identifying each row. Rows are also called **records** or **tuples** while columns are also called **attributes**. **Domain** is the set of allowable values for one or more attributes. **Degree** is the number of attributes in a relation. **Cardinality** is the number of tuples in a relation.

A **relational database** is based on the relational model. **Structured Query Language** (**SQL**) is a domain-specific language used for managing data held in a relational database management system (RDBMS). SQL can be divided into sublanguages, such as data definition language (DDL), data control language (DCL) and data manipulation language (DML). Most components of an SQL statement are case insensitive.

Data definition lanugage (**DDL**) is a syntax that is used to define an manipulate the structure of a database. Some common DDL commands include **CREATE**, **ALTER** and **DROP**. A **data manipulation language** (**DML**) is a programming language that is used to retrieve and manipulate data within a database. Some common DML statements include **SELECT**, **UPDATE** and **DELETE**.

2.1 Mathematical definition of relation

The **cartesian product** of two sets A and B, denoted $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$; see (1).

$$A \times B = \{(a, b) \mid a \in A \; and \; b \in B\} \tag{1}$$

Any subset of a cartesian product is a relation. The cartesian product of n sets $(D_1, D_2, ..., D_n)$ is defined as:

$$D_1 \times D_2 \times ... \times D_n = \{(d_1, d_2, ..., d_n) \mid d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n\} \tag{2}$$

2.2 Properties of relations

A relation name is distinct from all other relation names in a relational schema. Each cell of a relation contains exacly one single value. Each attribute has a distinct name. Values of an attribute are all from the same domain. Each tuple is distinct. The order of the attributes and the order of the tuples has no significance.

2.3 Relational keys

A **superkey** is an attribute or a set of attributes, that uniquely identifies a tuple within a relation. The set of all attributes is always a superkey.

A **candidate key** (or simply a **key**) is a minimal superkey. It is any set of columns that have a unique combination of values in each row, with the constraint that removing any column could produce duplicate combination of values. A relation can have multiple candidate keys, each with a different number of attributes.

A **primary key** is a candidate key selected to identify tuples uniquely within a relation (no null values).

Alternate keys are candidate keys that are not selected to be primary key.

A **foreign key** is an attribute, or a set of attributes, within one relation, that matches a candidate key of some (possibly the same) relation.

2.4 Integrity constraints

Null represents the absence of a value or unknown data. Null can be used to deal with incomplete or exceptional data.

Entity integrity is a concept that ensures that each row of a relation has a unique and non-null primary key value.

Referential integrity is a concept that ensures that if a value of one attribute of a relation references a value of another attribute (in the same or another relation), then the referenced value must exist.

Referential integrity helps to make a row unique and it increases the performance of search operations. Disadvantages include: indexes take additional disk space, indexes slow down INSERT, UPDATE and DELETE, because in each operation the indexes must also be updated.

2.5 Views

A **base relation** is a table whose tuples are stored in the database. A **view** is a virtual relation that does not necessarily exist in the database, but is produced upon request. Views are dynaminc, meaning that changes made to base relations that affect view attributes are immediately reflected in the view.

Views provides a mechanism that can be used to hide parts of a database from certain users. They can permit users to access data in a customized way. A view can also simplify complex operations on base relations.

3 Relational Algebra

In database theory, **relational algebra** is a theory that uses algebraic structures for modeling data and defining queries. It was introduced by Edgar F. Codd. The main application of relational algebras is to provide a theoretical foundation for relational databases. There are five primitive operators in relational algebra: selection, projection, cartesian product, union and set difference.

Selection (σ) works on a single relation R and defines a relation that contains only those tuples of R that satisfy the specified condition (**predicate**).

Projection ($\prod$) works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.

Union ($\cup$) of two relations R and S defines a relation that contains all the tuples in R and S, where duplicates are eliminated. R and S must be union-compatible, i.e. R and S must have the same number of attributes and the columns must have similar data types. In MySQL, the keyword for union is UNION.

Set difference ($\setminus$) of two relations R and S defines a relation that contains all the tuples in R, that are not in S. R and S must be union-compatible. In MySQL, the keyword for

set difference is EXCEPT.

Intersection ($\cap$) of two relations defines a relation that contains all the tuples that are in both R and S. R and S must be union-compatible. In MySQL, the keyword for intersection is INTERSECT.

4 MySQL SQL

The `SELECT` statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

```
SELECT [DISTINCT | ALL]
[SQL_NO_CACHE]
[into_option]
[FROM table_references]
[WHERE condition]
[GROUP BY {col_name | expr | position }]
[HAVING where_condition]
[ORDER BY {col_name | expr | position } [ASC | DESC]]
[LIMIT row_count]
```

SELECT specifies which columns are to appear in output.

SELECT DISTINCT is used to return only distinct values.

SQL_NO_CACHE is used to disable query cache for the server. Meaning it never checks the query cache to check whether the result is already cached.

FROM specifies which table(s) to be used.

WHERE filters rows.

GROUP BY forms groups of rows with same column value.

HAVING groups subject to some condition. It was added to SQL because WHERE cannot be used with aggregate functions. It is similar to WHERE, but instead of filtering individual rows, HAVING filters groups

ORDER BY specifies the order of the output.

LIMIT is used to specify the number of records to return.

The order of the clauses cannot be changed. Only SELECT and FROM are mandatory. For selecting all columns the * can be used.

```
SELECT * FROM table_name;
SELECT table_name.* FROM table_name; -- Useful for displaying all
columns from table_name if joined with another table.
```

For naming a column, the AS clause can be used. For renaming *column1* to *value1* one would use:

```
SELECT column1, column2 AS value1 FROM table_name;
```

4.1 Range search condition

BETWEEN is an operator that selects values within a given range. The negated version of BETWEEN is NOT BETWEEN. BETWEEN includes the endpoints of the range.

```
SELECT * FROM table_name
    WHERE column1 BETWEEN value1 AND value2;
```

4.2 Set membership

IN is an operator that allows you to specify multiple values in a WHERE clause.

```
SELECT * FROM table_name WHERE column1 IN (value1, value2,...);

SELECT * FROM table_name WHERE column1 IN (SELECT STATEMENT);
```

4.3 Pattern matching

LIKE is an operator that is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator: The percent sign (%) represents zero, one, or multiple characters. The underscore sign (_) represents one, single character.

```
SELECT * FROM table_name WHERE column1 LIKE pattern;
```

```
Find any values that start with an "a"
WHERE value1 LIKE 'a%'

Find any values that end with an "a"
WHERE value1 LIKE '%a'

Find any values that have "or" in any position
WHERE value1 LIKE '%or%'

Find any values that have "r" in the second position
WHERE value1 LIKE '_r%'

Find any values that starts with "a" and ends with "o"
WHERE value1 LIKE 'a%o'
```

4.4 NULL search condition

To test if something is NULL in MySQL, the IS NULL and IS NOT NULL operators can be used.

4.5 Ordering

ORDER BY is a keyword used to sort the result-set.

```
SELECT * FROM table_name
    ORDER BY column1,column2,... ASC|DESC;
```

4.6 GROUP BY

The GROUP BY statement groups rows that have the same values into summary rows. GROUP BY is often used with aggregate functions.

```
SELECT column_name(s) FROM table_name WHERE condition
    GROUP BY column_name(s) ORDER BY column_name(s);
```

4.7 Aggregated statements

According to the ISO standard, there are five aggregate functions:

COUNT returns number of values in a specified column. COUNT(*) counts all rows of a table, regardless of whether nulls or duplicate values occur.

```
SELECT COUNT(*) AS value1 FROM table_name;
```

SUM returns the sum of the values in a specified column.

AVG returns the average of the values in a specified column.

MIN returns the smallest value in a specified column.

MAX returns the largest value in a specified column.

Each operates on a single column of a table and returns a single value. COUNT, MIN and MAX apply to numeric and non-numeric fields, while AVG and SUM is used on numeric fields only. Apart from COUNT(*), each function eliminated nulls first, and operates on remaining non-null values.

Aggregate functions can be used only in SELECT list and in HAVING clause. DISTINCT can be used before a column name to eliminate duplicates. Example with HAVING:

```
SELECT Brand, COUNT(Color) AS color,
    SUM(PricePerDay) AS totalCost
    FROM Cars GROUP BY Brand
    HAVING COUNT(Brand) > 1 ORDER BY Brand DESC;
```

4.8 ANY and ALL

ANY and ALL may be used with subqueries that produce a single column of numbers. With ALL, condition will only be true if it is satisfied by all values produced by the subquery. With ANY, condition will be true if it is satisfied by any values produced by the subquery. Example:

```
SELECT * FROM Cars WHERE PricePerDay > ANY (
    SELECT PricePerDay FROM Cars WHERE Brand = 'Volvo');
```

4.9 Join operations

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

INNER JOIN returns all records that have matching values in both tables. Functionally, the JOIN and INNER JOIN keyword are the same.

```
-- Show all cars that have been booked at least once
SELECT DISTINCT c.* FROM Cars c JOIN Bookings b
    ON c.CarNumber = b.CarNumber ORDER BY c.CarNumber;
```

LEFT JOIN returns all records from the left table, and the matched records from the right table.

RIGHT JOIN returns all records from the right table, and the matched records from the left table.

CROSS JOIN returns all records from both tables.

4.10 EXISTS

EXISTS and NOT EXISTS are for use with subqueries.

```
-- Show all cars that have been booked at least once
SELECT * FROM Cars c WHERE EXISTS (
    SELECT * FROM Bookings b WHERE c.CarNumber = b.CarNumber);
```

4.11 INSERT, UPDATE and DELETE

INSERT syntax:

```
INSERT INTO table_name [(columnlist)] VALUES (dataValueList);
```

UPDATE syntax:

```
UPDATE table_name SET column1 = value1 [, column2 = value2,..]
    [WHERE condition]
```

DELETE syntax:

```
DELETE FROM table_name [WHERE condition]
```

Sometimes it is necessary to set `SQL_SAFE_UPDATES` to 0:

```
SET SQL_SAFE_UPDATES = 0;
```

4.12 ISO SQL data types

Data type	Declarations
boolean	BOOLEAN
character	CHAR, VARCHAR
exact numeric	NUMERIC, DECIMAL, INTEGER, SMALLINT, BIGINT
approximate numeric	FLOAT, REAL, DOUBLE PRECISION

5 Tables

5.1 CREATE TABLE

CREATE TABLE syntax:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype
    ...
);
```

Example:

```
CREATE TABLE Houses (
    HouseID INT,
    Owner VARCHAR(255),
    City VARCHAR(255),
    PRIMARY KEY (PersonID),
    FOREIGN KEY (Owner) REFERENCES Persons(Owner)
);
```

5.2 ALTER TABLE

The ALTER TABLE statement is used to add, delete or modify columns in an existing table. It is also used to add and drop various constraints on an existing table.

Syntax for adding a column:

```
ALTER TABLE table_name ADD column_name datatype;
```

Syntax for deleting a column:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Syntax for changing datatype of a column:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

Syntax for changing default value of a column:

```
ALTER TABLE table_name
    ALTER column_name { SET DEFAULT value_1 | DROP DEFAULT };
```

5.3 DROP TABLE

DROP TABLE syntax that removes a table and all rows within it:

```
DROP TABLE table_name [ RESTRICT | CASCADE];
```

With RESTRICT, if any other objects depend for their existence on continued existence of this table, SQL does not allow the request.

The TRUNCATE TABLE statement is used to delete data inside a table, but not the table itself.

6 Views

Create view syntax:

```
CREATE VIEW view_name AS
    SELECT column1, column2,...
    FROM table_name
    WHERE condition;
```

7 Laborations

7.1 Lab 1 - SQL

7.1.1 Tables

```
USE dv1664;

DROP TABLE IF EXISTS Cars;
CREATE TABLE Cars (
  CarNumber INT PRIMARY KEY,
```

```
    Brand VARCHAR(255),
    Model VARCHAR(255),
    Color VARCHAR(255),
    PricePerDay INT
);
SELECT * FROM Cars;

INSERT INTO Cars
VALUES
  (1, 'Peugeot', '208', 'Blue', 800),
  (2, 'Peugeot', '3008', 'Green', 700),
  (3, 'Volkswagen', 'Polo', 'Red', 600),
  (4, 'Volvo', 'V70', 'Silver', 1200),
    (5, 'Tesla', 'X', 'Black', 2000),
  (6, 'SAAB', '9-5', 'Green', 850),
  (7, 'Volvo', 'V40', 'Red', 900),
  (8, 'Fiat', '500', 'Black', 1050),
  (9, 'Volvo', 'V40', 'Green', 850),
  (10, 'Fiat', '500', 'Red', 950),
  (11, 'Volkswagen', 'Polo', 'Blue', 700),
  (12, 'BMW', 'M3', 'Black', 1599),
  (13, 'Volkswagen', 'Golf', 'Red', 1500);


DROP TABLE IF EXISTS Customers;
CREATE TABLE Customers (
  CustomerNumber INT PRIMARY KEY,
    CustomerName VARCHAR(255),
    BirthDate DATE
);
SELECT * FROM Customers;

INSERT INTO Customers
VALUES
  (1, 'Alice Andersson', '1990-05-05'),
  (2, 'Oscar Johansson', '1975-08-10'),
  (3, 'Nora Hansen', '1981-10-27'),
  (4, 'William Johansen', '2000-01-17'),
  (5, 'Lucía García', '1987-12-13'),
  (6, 'Hugo Fernández', '1950-03-16'),
  (7, 'Sofia Rossi', '1995-08-04'),
  (8, 'Francesco Russo', '2000-02-26'),
  (9, 'Olivia Smith', '1972-05-23'),
  (10, 'Oliver Jones', '1964-05-08'),
  (11, 'Shaimaa Elhawary', '1999-12-23'),
  (12, 'Mohamed Elshabrawy', '1997-11-07'),
  (13, 'Jing Wong', '1947-07-15'),
  (14, 'Wei Lee', '1962-09-29'),
  (15, 'Aadya Singh', '1973-01-01'),
  (16, 'Aarav Kumar', '1986-06-28'),
  (17, 'Louise Martin', '1994-04-22'),
```

```
  (18, 'Gabriel Bernard', '1969-12-01'),
  (19, 'Emma Smith', '1971-03-18'),
  (20, 'Noah Johnson', '1800-12-16'),
  (21, 'Alice Silva', '1988-12-04'),
  (22, 'Miguel Santos', '1939-12-29');

DROP TABLE IF EXISTS Bookings;
CREATE TABLE Bookings (
  CustomerNumber INT,
    CarNumber INT,
    StartDate DATE,
    EndDate DATE
    -- PRIMARY KEY (CustomerNumber, CarNumber, StartDate)
    -- CONSTRAINT PK_Bookings PRIMARY KEY (CustomerNumber, CarNumber,
    ↪ StartDate)
);
SELECT * FROM Bookings;

INSERT INTO Bookings
VALUES
  (1, 6, '2018-01-02', '2018-01-15'),
  (2, 1, '2018-01-03', '2018-01-05'),
  (4, 3, '2018-01-03', '2018-01-04'),
  (5, 8, '2018-01-04', '2018-01-30'),
  (6, 10, '2018-01-10', '2018-01-13'),
  (1, 1, '2018-01-20', '2018-01-25'),
  (2, 13, '2018-01-21', '2018-01-30'),
  (3, 6, '2018-01-22', '2018-01-30'),
  (1, 2, '2018-01-29', '2018-02-01'),
  (2, 5, '2018-02-02', '2018-02-06'),
  (6, 1, '2018-02-20', '2018-02-25'),
  (7, 6, '2018-02-21', '2018-02-24'),
  (8, 3, '2018-02-21', '2018-02-28'),
  (10, 3, '2018-02-22', '2018-02-26'),
  (9, 12, '2018-02-22', '2018-02-28'),
  (10, 13, '2018-03-01', '2018-03-10'),
  (11, 1, '2018-03-04', '2018-03-09'),
  (10, 3, '2018-03-11', '2018-03-14'),
  (8, 6, '2018-03-14', '2018-03-17'),
  (9, 5, '2018-03-14', '2018-03-30'),
  (7, 12, '2018-03-18', '2018-03-20'),
  (6, 8, '2018-03-18', '2018-04-02');
```

7.1.2 Selection, Projection and Restriction

```
-- SELECTION, PROJECTION and RESTRICTION

-- Show all customers with all their information.
SELECT * FROM Customers;

-- Show all customers, but only with their name and birthdate.
```

```
SELECT CustomerName, Birthdate FROM Customers;

-- Show all cars that cost more than 1000:- per day.
SELECT * FROM Cars WHERE PricePerDay > 1000;

-- Show all Volvo cars, only with their brand name and their model.
SELECT Brand, Model FROM Cars WHERE Brand = 'Volvo';

-- Show all customers, only with their names, in a sorted fashion based on their name.
-- Both in ascending and descending order.
SELECT CustomerName FROM Customers ORDER BY CustomerName ASC;
SELECT CustomerName FROM Customers ORDER BY CustomerName DESC;

-- Show all customers, only with their names, that were born in 1990 or later
-- in a sorted fashion based on their birthdate.
SELECT CustomerName FROM Customers WHERE YEAR(BirthDate) >= 1990 ORDER
    ↪ BY BirthDate;

-- Show all cars that are red and cost less than 1500.
SELECT * FROM Cars WHERE Color = 'Red' AND PricePerDay < 1500;

-- Show all customers, only with their names, that were born between 1970-1990.
SELECT CustomerName FROM Customers WHERE YEAR(BirthDate) BETWEEN 1970
    ↪ AND 1990;

-- Show all bookings that are longer than 6 days.
SELECT * FROM Bookings WHERE DATEDIFF(EndDate,StartDate) > 5;

-- Show all bookings that overlap with the interval 2018-02-01 - 2018-02-25.
SELECT * FROM Bookings WHERE StartDate < '2018-02-25' AND EndDate > '
    ↪ 2018-02-01';

-- Show all customers whose first name starts with an O.
SELECT * FROM Customers WHERE CustomerName LIKE 'O%';
```

7.1.3 Aggregated Functions

```
-- AGGREGATED FUNCTIONS
-- Show the average price per day for the cars.
SELECT AVG(PricePerDay) AS 'Average Price' FROM Cars;

-- Show the total price per day for the cars.
SELECT SUM(PricePerDay) AS 'Total Price' FROM Cars;

-- Show the average price for red cars.
SELECT AVG(PricePerDay) AS 'Average Red Car Price' FROM Cars WHERE Color = 'Red';
    ↪

-- Show the total price for all cars grouped by the different colors.
SELECT Color, AVG(PricePerDay) FROM Cars GROUP BY Color;
```

```
-- Show how many cars are of red color.
SELECT COUNT(*) AS 'Red Cars Count' FROM Cars WHERE Color = 'Red';

-- Show how many cars exists of each color.
SELECT Color, COUNT(Color) AS 'Number of cars' FROM Cars GROUP BY Color;

-- Show the car that is the most expensive to rent. (Do not hard code this
-- with the most expensive price, instead use ORDER and LIMIT.)
SELECT * FROM Cars ORDER BY PricePerDay DESC LIMIT 1;
```

7.1.4 Joins

```
-- JOINS
-- Show the Cartesian product between Cars and Bookings.
SELECT * FROM Cars CROSS JOIN Bookings;


-- Show the Cartesian product between Customers and Bookings.
SELECT * FROM Customers CROSS JOIN Bookings;


-- Show the results of converting the previous two joins to inner joins.
SELECT * FROM Cars INNER JOIN Bookings ON Cars.CarNumber=Bookings.CarNumber;


-- Show the names of all the customers that has made a booking.
SELECT CustomerName FROM Customers INNER JOIN Bookings
ON Customers.CustomerNumber=Bookings.CustomerNumber;


-- Same as the previous but without all the duplicates.
SELECT DISTINCT CustomerName FROM Customers INNER JOIN Bookings
ON Customers.CustomerNumber=Bookings.CustomerNumber;


-- Show all the Volkswagen cars that has been booked at least once.
SELECT DISTINCT Cars.* FROM Cars INNER JOIN Bookings
ON Cars.CarNumber=Bookings.CarNumber WHERE Brand='Volkswagen';


-- Show all the customers that has rented a Volkswagen.
SELECT DISTINCT Customers.* FROM Customers
INNER JOIN Bookings ON Customers.CustomerNumber=Bookings.CustomerNumber
INNER JOIN Cars ON Bookings.CarNumber=Cars.CarNumber WHERE Brand='Volkswagen'
    ↪ ;

-- Alternatively
SELECT * FROM Customers WHERE CustomerNumber IN (
  SELECT CustomerNumber FROM Bookings WHERE CarNumber IN (
    SELECT CarNumber FROM Cars WHERE Brand='Volkswagen'
    )
```

```
);


-- Show all cars that has been booked at least once.
SELECT DISTINCT Cars.* FROM Cars INNER JOIN Bookings
ON Cars.CarNumber = Bookings.CarNumber ORDER BY CarNumber;

-- Alternatively
SELECT * FROM Cars WHERE CarNumber IN (SELECT CarNumber FROM Bookings);


-- Show all cars that has never been booked.
SELECT DISTINCT Cars.* FROM Cars LEFT JOIN Bookings
ON Cars.CarNumber = Bookings.CarNumber WHERE Bookings.CarNumber IS NULL;

-- Alternatively
SELECT * FROM Cars WHERE CarNumber NOT IN (SELECT CarNumber FROM
    ↪ Bookings);


-- Show all the black cars that has been booked at least once.
SELECT DISTINCT Cars.* FROM Cars INNER JOIN Bookings
ON Cars.CarNumber = Bookings.CarNumber WHERE Color = 'Black';

-- Alternatively
SELECT * FROM Cars WHERE Color='Black' AND CarNumber IN (SELECT CarNumber
    ↪ FROM Bookings);
```

7.1.5 Nested Queries

```
-- NESTED QUERIES
-- Show all the cars that cost more than the average.
SELECT * FROM Cars WHERE PricePerDay > (SELECT AVG(PricePerDay) FROM Cars);

-- Show the car with the lowest cost with black color.
SELECT * FROM Cars WHERE Color='Black' ORDER BY PricePerDay LIMIT 1;

-- Show the car which has the lowest cost.
SELECT * FROM Cars ORDER BY PricePerDay LIMIT 1;

-- Show all the black cars that has been booked at least once by using a sub query.
SELECT * FROM Cars WHERE Color='Black' AND CarNumber IN (SELECT CarNumber
    ↪ FROM Bookings);
```

7.1.6 IN

```
-- IN
-- Show all cars that has the cost 700, 800, and 850.
SELECT * FROM Cars WHERE PricePerDay IN (700, 800, 850);
```

```
-- Show all the customers that born in 1990, 1995, and 2000. (Hint: YEAR function).
SELECT * FROM Customers WHERE YEAR(BirthDate) IN (1990, 1995, 2000);

-- Show all the bookings that start on 2018-01-03, 2018-02-22, or 2018-03-18.
SELECT * FROM Bookings WHERE StartDate IN ('2018-01-03','2018-03-18','
    ↪ 2018-02-22');
```

7.1.7 BETWEEN

```
-- BETWEEN
-- Show all cars whose price is in the range 600 - 1000.
SELECT * FROM Cars WHERE PricePerDay BETWEEN 600 AND 1000;

-- Show all the customers who are born between 1960 - 1980.
SELECT * FROM Customers WHERE YEAR(BirthDate) BETWEEN 1960 AND 1980;

-- Show all bookings that last between 2 - 4 days.
SELECT * FROM Bookings WHERE DATEDIFF(EndDate,StartDate) < 4
AND DATEDIFF(EndDate,StartDate) > 0;


-- A mix of everything
-- Show all the cars that are eligible for booking between 2018-01-10 - 2018-01-20.
SELECT * FROM Cars WHERE CarNumber NOT IN (
  SELECT CarNumber FROM Bookings WHERE StartDate < '2018-01-20' AND EndDate
    ↪ > '2018-01-10'
);

-- Show the car that has been booked the most.
SELECT * FROM Cars WHERE CarNumber IN (
  SELECT CarNumber FROM Bookings GROUP BY CarNumber ORDER BY COUNT(
    ↪ CarNumber) DESC
) LIMIT 1;


-- Show all the customers who are born in January or February and has booked at least one
    ↪ car.
SELECT * FROM Customers WHERE CustomerNumber IN (SELECT CustomerNumber
    ↪ FROM Bookings)
AND MONTH(BirthDate) BETWEEN 1 AND 2;
```

7.1.8 DELETE, UPDATE, ALTER & INSERT

```
-- Delete Update, Alter & Insert

-- Necessary to do certain operations, eg. update table withour a where that uses a KEY
    ↪ column
SET SQL_SAFE_UPDATES = 0;
```

```
-- There is a customer born in 1800 according to the records, this is obviously not possible so
    ↪ delete that customer.
DELETE FROM Customers WHERE YEAR(BirthDate) = 1800;

SELECT * FROM Customers ORDER BY YEAR(BirthDate);
DELETE FROM Customers WHERE CustomerNumber=20 AND YEAR(BirthDate)=1800;


-- The Tesla X car that is available for renting needs to have its price increased by 200:-.
SELECT * FROM Cars WHERE Brand = 'Tesla';
UPDATE Cars SET PricePerDay = PricePerDay + 200 WHERE CarNumber = 5;


-- All the Peugeot cars also needs to be increased in price, in this case by 20%.
UPDATE Cars SET PricePerDay = PricePerDay*1.20 WHERE Brand = 'Peugeot';


-- Now we fast forward into the future and Sweden has changed its currency to Euros (€).
-- Fix both the data itself (assume the conversion rate is 10SEK == 1 EUR) and the table
-- so it can handle the new prices.
UPDATE Cars SET PricePerDay = PricePerDay*0.1;


-- Can we construct a PK in the Bookings table without adding a new column?
-- If yes, do that. If not, add another column that allows you to uniquely identify each
    ↪ booking.
ALTER TABLE Bookings ADD PRIMARY KEY (CustomerNumber, CarNumber, StartDate);
```

7.1.9 VIEW

```
-- View
-- Create a view, that shows all the information about black cars.
DROP VIEW IF EXISTS view_black_cars;
CREATE VIEW view_black_cars AS SELECT * FROM Cars WHERE Color = 'Black';
SELECT * FROM view_black_cars;


-- Create a view that shows all information about black cars and the addition of the weekly
    ↪ price as a column.
DROP VIEW IF EXISTS view_black_cars_info;
CREATE VIEW view_black_cars_info AS SELECT Cars.*, PricePerDay*7 AS '
    ↪ PricePerWeek' FROM Cars;
SELECT * FROM view_black_cars_info;


-- Try and insert a car into both views created. What happens? Why? What's the difference
    ↪ between the views?
INSERT INTO view_black_cars VALUES (99, 'Ferrari', 'LaFerrari', 'Red', 700); -- Works
```

```
INSERT INTO view_black_cars_info VALUES (98, 'Peugeot', '308', 'Green', 50, 350); --
    ↪ Does not work

-- A view is a virtual table. An added column doesn't exist in any table in the database.
-- When you insert into view_black_cars MySQL knows which table that has to get
    ↪ updated
-- But when you try insert into view_black_cars_info, there is no table that can store
-- the information about weekly price.


-- Create a view that shows all the cars available for booking at this current time.
DROP VIEW IF EXISTS currently_available;
CREATE VIEW currently_available AS SELECT * FROM Cars WHERE CarNumber NOT
    ↪ IN
  (SELECT CarNumber FROM Bookings WHERE EndDate > NOW() AND StartDate <
    ↪ NOW());
SELECT * FROM currently_available;

-- Alter the previous view, with the condition that the cars have to be available for at least 3
    ↪  days of renting.
DROP VIEW IF EXISTS currently_available_3;
CREATE VIEW currently_available_3 AS SELECT * FROM Cars WHERE CarNumber
    ↪ NOT IN
  (SELECT CarNumber FROM Bookings WHERE EndDate > NOW()
    AND StartDate < DATE_ADD(NOW(), INTERVAL 2 DAY));
SELECT * FROM currently_available_3;
```

7.1.10 DROP

```
-- DROP
SET SQL_SAFE_UPDATES = 0;

-- Drop the table Cars.
DROP TABLE Cars;

-- Why didn't it work? Fix so that you can drop the table.
-- It did work

-- Delete all the rows of table Customers.
DELETE FROM Customers; -- SQL_SAFE_UPDATES has to be 0.

-- Alternatively
TRUNCATE TABLE Customers;


-- What's the difference between DROP TABLE and DELETE?
-- DROP TABLE removes the entire TABLE from a database while
-- DELETE only removes rows from a table.
```

7.2 Lab 2 and 3 - Triggers, Procedures, and Functions

7.2.1 User-Defined Functions

```
use dv1664;

-- 1. USER DEFINED FUNCTIONS
-- 1. Create a function that checks if a car is available for renting between two dates.
-- The input to the function should be the starting and ending dates of the rental,
-- the cars number, and it should return 0 if it is not available and 1 if it is available between
    ↪ the two dates.
drop function available_for_renting;
DELIMITER //
CREATE FUNCTION available_for_renting (car_num INT, date_start DATE, date_end
    ↪ DATE) RETURNS INT
READS SQL DATA
BEGIN
  DECLARE count_bookings INT;
    SET count_bookings = (
    SELECT COUNT(*) FROM Bookings
        WHERE car_num = CarNumber
        AND StartDate <= date_end AND EndDate >= date_start
    );

    IF count_bookings > 0 THEN RETURN 0;
    ELSE RETURN 1;
    END IF;

RETURN var;
END; //
DELIMITER ;

SELECT available_for_renting(13,'2018-01-01', '2018-01-19');


-- 2. Create a function that sums the total amount of days cars have been booked and
    ↪ returns the sum.
DROP FUNCTION sum_booked_days;
DELIMITER //
CREATE FUNCTION sum_booked_days()
RETURNS INT
READS SQL DATA
BEGIN
  DECLARE var_sum INT;
  SELECT SUM(DATEDIFF(EndDate,StartDate)+1) INTO var_sum FROM Bookings;
  RETURN var_sum;
END; //
DELIMITER ;

SELECT sum_booked_days();

-- 3. Extend the previous function so that if there is an input parameter that matches a
```

```
-- cars unique number, then it should only return the sum of the car. If the number
-- doesn't match or it is -1, it returns the total sum as before
DROP FUNCTION sum_booked_days_num;
DELIMITER //
CREATE FUNCTION sum_booked_days_num(car_num INT)
RETURNS INT
READS SQL DATA
BEGIN
  DECLARE var_sum INT DEFAULT sum_booked_days();
    IF (car_num IN (SELECT CarNumber FROM Cars)) THEN
    SELECT SUM(DATEDIFF(EndDate,StartDate)+1) INTO var_sum FROM Bookings
     ↪ WHERE CarNumber = car_num;
  END IF;
  IF (var_sum IS NULL) THEN
    SELECT 0 INTO var_sum;
  END IF;
RETURN var_sum;
END; //
DELIMITER ;

SELECT sum_booked_days_num(1);
```

7.2.2 Stored Procedures

```
use dv1664;

-- STORED PROCEDURES
-- 1. Create a stored procedure that collects all the cars that are available between two dates.
-- The inputs to the procedure is starting and ending dates, and prints all the car numbers
-- that are available to be booked between the two dates.

DROP PROCEDURE GetCarsAvailable;
DELIMITER //
CREATE PROCEDURE GetCarsAvailable(IN start_date DATE, IN end_date DATE)
BEGIN
  SELECT * FROM Cars WHERE CarNumber NOT IN (
    SELECT CarNumber FROM Bookings WHERE StartDate <= end_date AND EndDate
     ↪ >= start_date
       );
END; //
DELIMITER ;

CALL GetCarsAvailable('2018-02-22','2018-02-26');


-- 2. Create a stored procedure that handles the booking of renting a car. The input
-- to the procedure is the starting and ending dates, the cars number,  and the customer
-- number. If it is successful it should return 0, if it is unsuccessful in booking it should
    ↪ return 1.

DROP PROCEDURE BookCar;
```

```
DELIMITER //
CREATE PROCEDURE BookCar(IN start_date DATE, IN end_date DATE, IN
    ↪ car_number INT, IN customer_number INT, OUT result INT)
BEGIN
  DECLARE count_bookings INT;
    SET count_bookings = (
    SELECT COUNT(*) FROM Bookings WHERE car_number = CarNumber
        AND StartDate <= end_date AND EndDate >= start_date
    );

    IF count_bookings != 0 OR customer_number NOT IN (SELECT CustomerNumber
     ↪ FROM Customers)
    OR start_date > end_date THEN SET result := 1;
    ELSE
    INSERT INTO Bookings VALUES (customer_number, car_number, start_date, end_date
     ↪ );
        SET RESULT := 0;
    END IF;

END; //
DELIMITER ;

CALL BookCar('2020-02-22','2020-02-26', 10, 6, @result);
SELECT @result;
SELECT * FROM Bookings;
SELECT * FROM Customers;
```

7.2.3 Triggers

```
USE dv1664;

-- TRIGGERS
-- 1. Add an additional column to Customers that contains the amount of times a customer
-- has booked a car. Then create an after insert trigger on the Bookings table that
-- increments the newly created column in Customers whenever they do a successful booking
    ↪  of a car.

ALTER TABLE Customers ADD NumberOfBookings INT DEFAULT 0;
SELECT * FROM Customers;

DROP TRIGGER IF EXISTS trigger_customers;
DELIMITER //
CREATE TRIGGER trigger_customers
AFTER INSERT ON Bookings
FOR EACH ROW
  BEGIN
      UPDATE Customers SET NumberOfBookings = NumberOfBookings + 1 WHERE
    ↪ CustomerNumber = NEW.CustomerNumber;
   END; //
DELIMITER ;
```

```
INSERT INTO Bookings VALUES (2,5,'2022-05-01','2022-05-05');

-- Would it be possible to do this trigger with a BEFORE trigger? Why/Why not? yes
```