

CMP7214 – COURSEWORK

Airport database design

Coursework Assignment Report

Daniel Rimaru - 19134702

Zan Zver - 18133498



BIRMINGHAM CITY
University

Faculty of Computing, Engineering and the Built Environment
Birmingham City University

Contents

1	Introduction	4
2	Domain of interest	5
3	Database analysis	5
3.1	List of Entity/Attributes	5
3.2	Simple Relationships	5
3.3	Connectivities, Cardinalities and Participation	5
4	ERD Mapping	5
4.1	Mapping 1:1 Relationships	5
4.2	Mapping 1:M Relationships	9
4.3	Mapping N: M Relationships	10
4.4	Mapping A Three-way Relationship	12
5	ERD	13
6	Database implementation	13
6.1	Physical Design	13
6.2	Create tables	13
6.3	Assign Foreign Keys	17
6.4	Create views	20
6.5	Relationship Schema	22
6.6	Create Procedure	23
6.7	Dummy Data Insertion	24
7	Testing of the database	29
8	Queries	44
8.1	Daniel	45
8.1.1	Query1	45
8.1.2	Query2	45
8.1.3	Query3	45
8.1.4	Optimisation technique	45
8.2	Zan	45
8.2.1	Query1	45
8.2.2	Query2	46
8.2.3	Query3	47
8.2.4	Optimisation technique	48
9	Conclusion	52

10	Appendixes	53
10.1	A0	53
10.2	A1	55
10.3	A2	59
10.4	A3	71
	10.4.1 A3.1	71
	10.4.2 A3.2	71
	10.4.3 A3.3	71
	10.4.4 A3.4	71
	10.4.5 A3.5	72
	10.4.6 A3.6	72
	10.4.7 A3.7	72
	10.4.8 A3.8	72
	10.4.9 A3.9	72
	10.4.10 A3.10	73
	10.4.11 A3.11	73
	10.4.12 A3.12	73
	10.4.13 A3.13	73
10.5	A4	73
10.6	A5	75
10.7	A6	76
10.8	A7	77
10.9	A8	77
10.10	A9	77
10.11	A10	80
10.12	A11	80
10.13	A12	81
11	References	82

List of Figures

1	Flight - Runway ERD	5
2	Flight table	6
3	Runway table	6
4	Parking Token - Customer ERD	6
5	Customer Parking Spot table	6
6	Parking Token table	6
7	Customer Parking Spot - Parking Token ERD	6
8	Customer table	7
9	Parking Token table	7
10	Department Manage Employee ERD	7
11	Department table	7

12	Employee table	7
13	Employee Is Named Name ERD	7
14	Employee table	8
15	Names table representing employees	8
16	Customer Is Named Name ERD	8
17	Customers table	8
18	Names table representing customers	8
19	Flight - Airline ERD	9
20	Flight table	9
21	Airline table	9
22	Department Works Employee ERD	9
23	Department table	9
24	Employee table	9
25	CompanyVehicle - Department ERD	10
26	Company Vehicle table	10
27	Department table	10
28	Customer - Ticket - Flight ERD	10
29	Customer table	10
30	Ticket table	11
31	Flight table	11
32	Employee - Certificate - Qualification ERD	11
33	Employee table	11
34	Certificate table	11
35	Qualification table	11
36	Employee - Company Vehicle – Employee Parking Spot ERD	12
37	Employee table	12
38	Employee Parking Spot table	12
39	Company Vehicle table	12
40	ERD	13
41	Result of CustomerNames view	21
42	Result of EmployeeNames view	22
43	Relationship schema from UI	23
44	Result of GetParkingToBePaid procedure	24
45	Example of all test passes	44
46	Example of all test passes	44
47	Query1a result - Zan	46
48	Query1b result - Zan	46
49	Query2 result - Zan	47
50	Query3 result - Zan	48
51	Physical design	56
52	Docker showcase	59

1 Introduction

This report writes about database system development for airport. Goal of the project is to create highly available relational database that can store airports data.

We would like to give a huge thanks to Ben Sutton who is the Director of Data and Strategic Analytics at MAG airport. Ben has given us some insights regarding how airport operates from behind the scenes and how their IT department works. Based on that input, we have gained a strong base of which we worked of.

Software that is used in this report can be found in appendix A11 (10.12).

2 Domain of interest

We have chosen an airport as our domain of interest. Since airports operate differently with complex structures underneath, we have chosen to contact one of the UK's airports for help. Manchester Airport Group (MAG) decided to help us with our journey of constructing an efficient database. Ben Sutton is the Director of Data and Strategic Analytics, who was our contact in this project.

Contacting MAG airport, we received some critical information, specifically what data is stored. For example, pilot data is not stored since pilots operate under airlines, therefore we do not store their data. Another example is customer (who is flying) data. Airports do get minimal information about customers due to privacy concerns.

Business rules are further explained in appendix A0 (10.1).

3 Database analysis

3.1 List of Entity/Attributes

Discussion on entities and attributes can be seen in appendix A5 (10.6).

3.2 Simple Relationships

Relationship mapping is listed in appendix A6 (10.7).

3.3 Connectivities, Cardinalities and Participation

Appendix A4 (10.5) describes all of the connectivities, cardinalities and participation for the database.

4 ERD Mapping

4.1 Mapping 1:1 Relationships

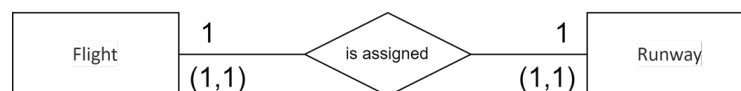


Figure 1: Flight - Runway ERD

Flight_ID	Plane_Model	Departure_Time	Destination	Gate	Runway_FK	Airline_FK
1	Model_1	2022-01-10 01:00:00.000	Destination_1	Gate_1	1	1
2	Model_2	2022-02-20 22:22:00.000	Destination_2	Gate_2	1	2
3	Model_3	2022-03-30 09:30:00.000	Destination_3	Gate_3	2	2

Figure 2: Flight table

Runway_ID	Length	Width
1	100	10
2	200	20
3	300	30

Figure 3: Runway table



Figure 4: Parking Token - Customer ERD

Parking_Spot_ID	Parking_Type	Price_Per_Hour
1	{"1*":"Car Parking"}	6.00
2	{"1*":"Motor Parking"}	6.00
3	{"1*":"Car Parking", "2*":"Handicap parking"}	5.00

Figure 5: Customer Parking Spot table

Token_ID	Date	Parking_Spot_FK
1	2022-01-10 11:11:00.000	1
2	2022-02-20 22:00:00.000	2
3	2022-03-30 15:30:00.000	3

Figure 6: Parking Token table



Figure 7: Customer Parking Spot - Parking Token ERD

Customer_ID	Express_Lane	Token_FK
1	0	1
2	1	2
3	0	

Figure 8: Customer table

Token_ID	Date	Parking_Spot_FK
1	2022-01-10 11:11:00.000	1
2	2022-02-20 22:00:00.000	2
3	2022-03-30 15:30:00.000	3

Figure 9: Parking Token table

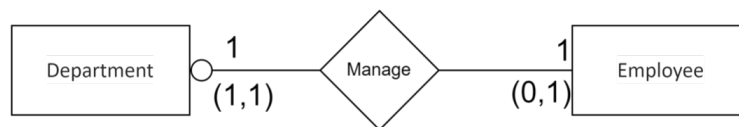


Figure 10: Department Manage Employee ERD

Department_ID	Department_Name	Department_Location
1	Department_1	Location_1
2	Department_2	Location_2
3	Department_3	Location_3

Figure 11: Department table

Employee_ID	Hire_Date	Termination_Date	Title	Employment_Type	Salary	Supervisor	Address_Line_1	Address_Line_2	Postcode	City	Birth_Date	Parking_Spot_FK	Vehicle_FK	Department_FK	Manage_Department
1	2022-01-01		Title_1	Type_1	90000.00		Address_Line_1_1		Postcode_1	City_1	1980-12-01	1	1	1	1
2	2022-02-02	2022-12-31	Title_2	Type_2	60000.00	1	Address_Line_1_2	Address_Line_2_2	Postcode_2	City_2	1980-12-03	2	2	2	2
3	2022-03-03		Title_3	Type_3	30000.00	2	Address_Line_1_3		Postcode_3	City_3	1980-12-03	2		2	

Figure 12: Employee table

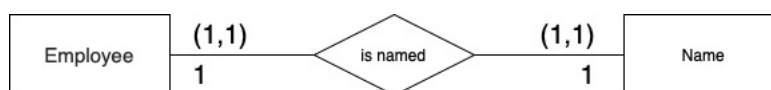


Figure 13: Employee Is Named Name ERD

Employee_ID	Hire_Date	Termination_Date	Title	Employment_Type	Salary	Supervisor	Address_Line_1	Address_Line_2	Postcode	City	Birth_Date	Parking_Spot_FK	Vehicle_FK	Department_FK	Manage_Department
1	2021-12-01		Bonds trader	Piece rate	86896.59		Russell Trail	6643	27325	Brianfort	2020-01-14	181	101	13	9
5	2022-10-16		Electrical engineer	Casual	10384.87	1	Williams Valley	16439	91433	Andersonechester	2000-03-28	57	229	13	9
9	2020-07-28		Database administrator	Fixed term	41861.67	5	Haynes Fields	3359	23809	North James	2020-03-30	5	117	1	13

Figure 14: Employee table

Person_ID	First_Name	Middle_Name	Last_Name
1	Ariana	Jeremy	Barrera
5	Lisa	Heather	Mendez
9	John	Kristen	Gutierrez

Figure 15: Names table representing employees

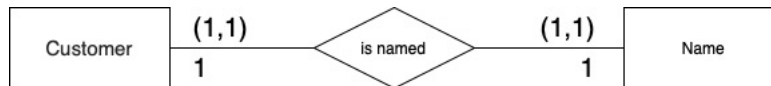


Figure 16: Customer Is Named Name ERD

Customer_ID	Express_Lane	Token_FK
1	0	205
5	0	3449
9	0	2393

Figure 17: Customers table

Person_ID	First_Name	Middle_Name	Last_Name
1	Heather	Jonathan	Young
5	Jessica		Ibarra
9	Tiffany	Amanda	Allen

Figure 18: Names table representing customers

4.2 Mapping 1:M Relationships



Figure 19: Flight - Airline ERD

Flight_ID	Plane_Model	Departure_Time	Destination	Gate	Runway_FK	Airline_FK
1	Model_1	2022-01-10 01:00:00.000	Destination_1	Gate_1	1	1
2	Model_2	2022-02-20 22:22:00.000	Destination_2	Gate_2	1	2
3	Model_3	2022-03-30 09:30:00.000	Destination_3	Gate_3	2	2

Figure 20: Flight table

Company_ID	Company_Name	Revenue
1	Company_1	20000.00
2	Company_2	40000.00
3	Company_3	80000.00

Figure 21: Airline table



Figure 22: Department Works Employee ERD

Department_ID	Department_Name	Department_Location
1	Department_1	Location_1
2	Department_2	Location_2
3	Department_3	Location_3

Figure 23: Department table

Employee_ID	Hire_Date	Termination_Date	Title	Employment_Type	Salary	Supervisor	Address_Line_1	Address_Line_2	Postcode	City	Birth_Date	Parking_Spot_FK	Vehicle_FK	Department_FK	Manage_Department
1	2022-01-01		Title_1	Type_1	90000.00		Address_Line_1_1		Postcode_1	City_1	1980-12-01	1	1	1	1
2	2022-02-02	2022-12-31	Title_2	Type_2	60000.00	1	Address_Line_1_2	Address_Line_2_2	Postcode_2	City_2	1980-12-03	2	2	2	2
3	2022-03-03		Title_3	Type_3	30000.00	2	Address_Line_1_3		Postcode_3	City_3	1980-12-03	2		2	

Figure 24: Employee table



Figure 25: CompanyVehicle - Department ERD

Vehicle_ID	Vehicle_Name	Vehicle_Driving_License_Requirement	Department_FK
1	Vehicle_1	License_1	1
2	Vehicle_2	License_1	1
3	Vehicle_3	License_2	2

Figure 26: Company Vehicle table

Department_ID	Department_Name	Department_Location
1	Department_1	Location_1
2	Department_2	Location_2
3	Department_3	Location_3

Figure 27: Department table

4.3 Mapping N: M Relationships

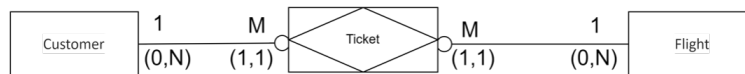


Figure 28: Customer - Ticket - Flight ERD

Customer_ID	Express_Lane	Token_FK
1	0	1
2	1	2
3	0	

Figure 29: Customer table

Ticket_ID	Customer_FK	Flight_FK
1	1	1
2	2	1
3	3	3

Figure 30: Ticket table

Flight_ID	Plane_Model	Departure_Time	Destination	Gate	Runway_FK	Airline_FK
1	Model_1	2022-01-10 01:00:00.000	Destination_1	Gate_1	1	1
2	Model_2	2022-02-20 22:22:00.000	Destination_2	Gate_2	1	2
3	Model_3	2022-03-30 09:30:00.000	Destination_3	Gate_3	2	2

Figure 31: Flight table

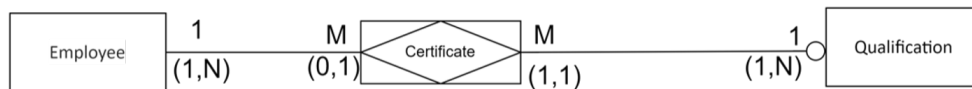


Figure 32: Employee - Certificate - Qualification ERD

Employee_ID	Hire_Date	Termination_Date	Title	Employment_Type	Salary	Supervisor	Address_Line_1	Address_Line_2	Postcode	City	Birth_Date	Parking_Spot_FK	Vehicle_FK	Department_FK	Manage_Department
1	2022-01-01		Title_1	Type_1	90000.00		Address_Line_1_1		Postcode_1	City_1	1980-12-01	1	1	1	1
2	2022-02-02	2022-12-31	Title_2	Type_2	60000.00	1	Address_Line_1_2	Address_Line_2_2	Postcode_2	City_2	1980-12-03	2	2	2	2
3	2022-03-03		Title_3	Type_3	30000.00	2	Address_Line_1_3		Postcode_3	City_3	1980-12-03	2		2	

Figure 33: Employee table

Employee_FK	Certificate_ID	Certificate_Name	Achievement_Level	Qualification_FK
1	1	Certificate_1	Achievement_1	1
1	2	Certificate_2	Achievement_2	2
2	3	Certificate_1	Achievement_3	3

Figure 34: Certificate table

Qualification_ID	Qualification_Type	Qualification_Name
1	Type_1	Name_1
2	Type_2	Name_2
3	Type_3	Name_3

Figure 35: Qualification table

4.4 Mapping A Three-way Relationship

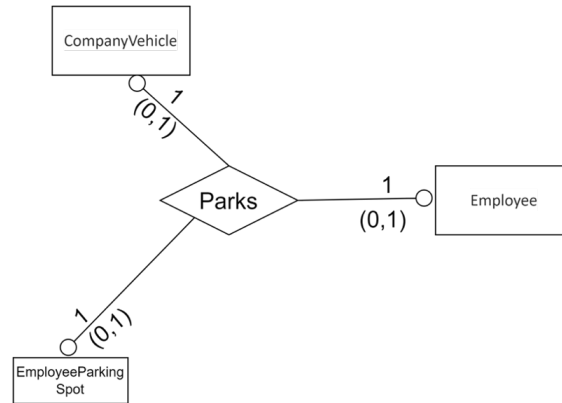


Figure 36: Employee - Company Vehicle – Employee Parking Spot ERD

Employee_ID	Hire_Date	Termination_Date	Title	Employment_Type	Salary	Supervisor	Address_Line_1	Address_Line_2	Postcode	City	Birth_Date	Parking_Spot_FK	Vehicle_FK	Department_FK	Manage_Department
1	2022-01-01		Title_1	Type_1	90000.00		Address_Line_1_1		Postcode_1	City_1	1980-12-01	1	1	1	1
2	2022-02-02	2022-12-31	Title_2	Type_2	60000.00	1	Address_Line_1_2	Address_Line_2_2	Postcode_2	City_2	1980-12-03	2	2	2	2
3	2022-03-03		Title_3	Type_3	30000.00	2	Address_Line_1_3		Postcode_3	City_3	1980-12-03	2		2	

Figure 37: Employee table

Parking_Spot_ID	Parking_Type
1	{"1":"Car Parking", "2":"CEO parking"}
2	{"1":"Motor Parking"}
3	{"1":"Car Parking", "2":"Handicap parking"}

Figure 38: Employee Parking Spot table

Vehicle_ID	Vehicle_Name	Vehicle_Driving_License_Requirement	Department_FK
1	Vehicle_1	License_1	1
2	Vehicle_2	License_1	1
3	Vehicle_3	License_2	2

Figure 39: Company Vehicle table

5 ERD

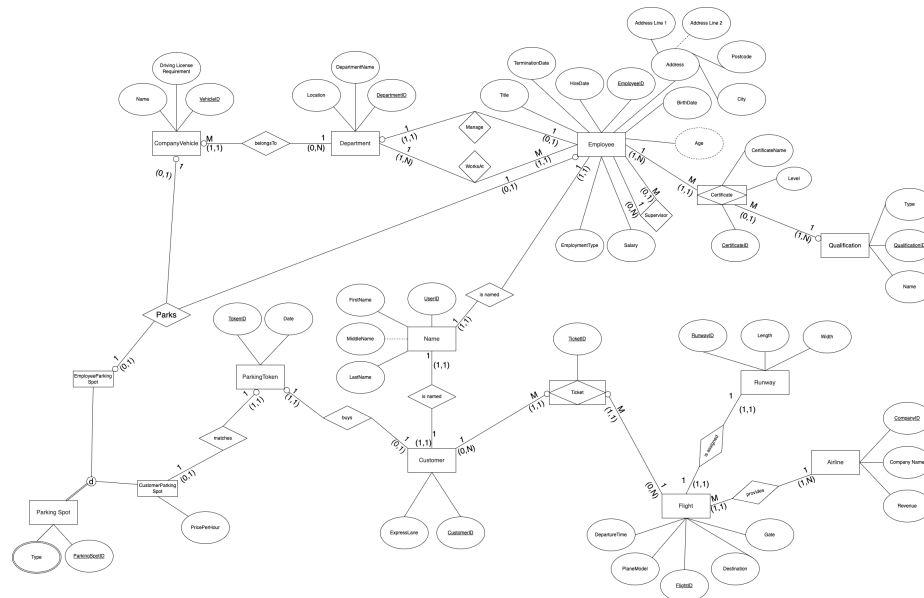


Figure 40: ERD

6 Database implementation

6.1 Physical Design

In this use case, the database would be storing data from airport. Due to the nature of this business, the airport needs to be operational at all times, meaning our database would need to be highly available.

Solving high availability (HA) can be done in numerous ways. The simplest one is to use a cloud provider (example: Amazon Web Services - AWS) and rely on their infrastructure. For example, AWS RDS has a minimum of 99.95% monthly up-time percentage (during 31 days database would be inaccessible for 0.372 hours) (Amazon [n.d.](#)). Since this is an industry example, we have replicated what AWS can do on a smaller scale.

Further discussion can be seen in appendix A1 (10.2).

6.2 Create tables

This section describes how tables are created. SQL DDL statements are included in here, deeper description can be found in appendix 3 (10.4).

Extended discussion can be found at 10.4.1.

```
01 | -- Airline table
02 | CREATE TABLE `Airline` (
```

```
03 | `Company_ID` int(11) NOT NULL AUTO_INCREMENT,  
04 | `Company_Name` varchar(64) NOT NULL,  
05 | `Revenue` decimal(15,2) NOT NULL COMMENT 'Revenue needs to  
    be at least 10000 per month',  
06 | PRIMARY KEY (`Company_ID`),  
07 | CHECK(`Revenue` >= 10000)  
08 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 1: Airline table creation

Extended discussion can be found at 10.4.2.

```
01 | -- Certificate table  
02 | CREATE TABLE `Certificate` (  
03 | `Employee_FK` int(11) NOT NULL,  
04 | `Certificate_ID` int(11) NOT NULL,  
05 | `Certificate_Name` varchar(64) NOT NULL,  
06 | `Achievement_Level` varchar(32) NOT NULL,  
07 | `Qualification_FK` int(11) DEFAULT NULL,  
08 | PRIMARY KEY (`Certificate_ID`)  
09 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 2: Certificate table creation

Extended discussion can be found at 10.4.3.

```
01 | -- Compan Vehicle table  
02 | CREATE TABLE `CompanyVehicle` (  
03 | `Vehicle_ID` int(11) NOT NULL AUTO_INCREMENT,  
04 | `Vehicle_Name` varchar(48) NOT NULL,  
05 | `Vehicle_Driving_License_Requirement` varchar(24) NOT NULL  
    ,  
06 | `Department_FK` int(11) NOT NULL,  
07 | PRIMARY KEY (`Vehicle_ID`)  
08 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 3: Compan Vehicle table creation

Extended discussion can be found at 10.4.4)

```
01 | -- Customer table  
02 | CREATE TABLE `Customer` (  
03 | `Customer_ID` int(11) NOT NULL AUTO_INCREMENT,  
04 | `Express_Lane` tinyint(1) NOT NULL DEFAULT 0 COMMENT 'Data  
    needs to be 0 (default) if there is no express lane or  
    1 if user has express lane',  
05 | `Token_FK` int(11) DEFAULT NULL,  
06 | PRIMARY KEY (`Customer_ID`),  
07 | CHECK(`Express_Lane` <= 1 or `Express_Lane` >= 0)  
08 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 4: Customer table creation

Extended discussion can be found at 10.4.5.

```
01 | -- Customer Parking Spot table
02 | CREATE TABLE `CustomerParkingSpot` (
03 |     `Parking_Spot_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |     `Parking_Type` longtext CHARACTER SET utf8mb4 COLLATE
05 |     utf8mb4_bin NOT NULL CHECK (json_valid(`Parking_Type`)),
06 |     `Price_Per_Hour` decimal(15,2) NOT NULL COMMENT 'Minimum
07 |     pricing per hour is 5 GBP and maximum is 300 GBP',
08 |     PRIMARY KEY (`Parking_Spot_ID`),
09 |     CHECK(`Price_Per_Hour`>=5 and `Price_Per_Hour`<=300)
10 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 5: Customer parking spot table creation

Extended discussion can be found at 10.4.6.

```
01 | -- Department table
02 | CREATE TABLE `Department` (
03 |     `Department_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |     `Department_Name` varchar(32) NOT NULL,
05 |     `Department_Location` varchar(24) NOT NULL,
06 |     PRIMARY KEY (`Department_ID`)
07 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 6: Department table creation

Extended discussion can be found at 10.4.7.

```
01 | -- Employee table
02 | CREATE TABLE `Employee` (
03 |     `Employee_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |     `Hire_Date` date NOT NULL,
05 |     `Termination_Date` date NULL,
06 |     `Title` varchar(200) NOT NULL,
07 |     `Employment_Type` varchar(24) NOT NULL,
08 |     `Salary` decimal(15,2) NOT NULL COMMENT 'Salary is minimum
09 |     of 2160 per year. Person is paid 9/h (minimum) * 5h (
10 |     minimum per week) * 4 (whole month) * 12 (months in year
11 |     ) = 2160',
12 |     `Supervisor` int(11) DEFAULT NULL COMMENT 'ID of the
13 |     Employee that is supervisor',
14 |     `Address_Line_1` varchar(48) NOT NULL,
15 |     `Address_Line_2` varchar(48) DEFAULT NULL,
16 |     `Postcode` varchar(32) NOT NULL,
17 |     `City` varchar(48) NOT NULL,
18 |     `Birth_Date` date NOT NULL,
19 |     `Parking_Spot_FK` int(11) DEFAULT NULL,
20 |     `Vehicle_FK` int(11) DEFAULT NULL,
21 |     `Department_FK` int(11) NOT NULL,
22 |     `Manage_Department` int(11) DEFAULT NULL,
23 |     PRIMARY KEY (`Employee_ID`),
24 |     CHECK(`Salary`>=2160)
```



```
21 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3
22 | COMMENT='Storing employee details';
```

Listing 7: Employee table creation

Extended discussion can be found at 10.4.8.

```
01 | -- Employee Parking Spot table
02 | CREATE TABLE `EmployeeParkingSpot` (
03 |   `Parking_Spot_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |   `Parking_Type` longtext CHARACTER SET utf8mb4 COLLATE
      utf8mb4_bin NOT NULL CHECK (json_valid(`Parking_Type`)),
05 |   PRIMARY KEY (`Parking_Spot_ID`)
06 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 8: Employee parking spot table creation

Extended discussion can be found at 10.4.9.

```
01 | -- Flight table
02 | CREATE TABLE `Flight` (
03 |   `Flight_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |   `Plane_Model` varchar(64) NOT NULL,
05 |   `Departure_Time` datetime NOT NULL,
06 |   `Destination` varchar(48) NOT NULL,
07 |   `Gate` varchar(24) NOT NULL,
08 |   `Runway_FK` int(11) NOT NULL,
09 |   `Airline_FK` int(11) NOT NULL,
10 |   PRIMARY KEY (`Flight_ID`)
11 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 9: Flight table creation

Extended discussion can be found at 10.4.10.

```
01 | -- Person Name table
02 | CREATE TABLE `Name` (
03 |   `Person_ID` int(11) NOT NULL,
04 |   `First_Name` varchar(48) NOT NULL,
05 |   `Middle_Name` varchar(48) DEFAULT NULL,
06 |   `Last_Name` varchar(48) NOT NULL,
07 |   `Is_Employee` tinyint(1) NOT NULL DEFAULT 0 COMMENT 'Data
      needs to be 0 (default) if person is not employee or 1
      if they are employee',
08 |   PRIMARY KEY (`Person_ID`,`Is_Employee`),
09 |   CHECK(`Is_Employee`<=1 or `Is_Employee`>=0)
10 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 10: Name creation

Extended discussion can be found at 10.4.11.

```
01 | -- Parking Token table
02 | CREATE TABLE `ParkingToken` (
03 |   `Token_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |   `Date` datetime NOT NULL,
05 |   `Parking_Spot_FK` int(11) NOT NULL,
06 |   PRIMARY KEY (`Token_ID`)
07 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 11: Parking Token table creation

Extended discussion can be found at 10.4.12.

```
01 | -- Runway table
02 | CREATE TABLE `Runway` (
03 |   `Runway_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |   `Length` decimal(10,0) NOT NULL COMMENT
05 |   'Length needs to be at least 10, combined with Width it
    can be helipad. Longest it can be is 3km.',
06 |   `Width` decimal(10,0) NOT NULL COMMENT 'Width needs to be
    at least 10, combined with Length it can be helipad. Max
    width acceptable is 500m.',
07 |   PRIMARY KEY (`Runway_ID`),
08 |   CHECK(`Length`>=10 and `Length`<=3000),
09 |   CHECK(`Width`>=10 and `Width`<=500)
10 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 12: Runway table creation

Extended discussion can be found at 10.4.13.

```
01 | -- Qualification table
02 | CREATE TABLE `Qualification` (
03 |   `Qualification_ID` int(11) NOT NULL AUTO_INCREMENT,
04 |   `Qualification_Type` varchar(48) NOT NULL,
05 |   `Qualification_Name` varchar(48) NOT NULL,
06 |   PRIMARY KEY (`Qualification_ID`)
07 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

Listing 13: Qualification table creation

6.3 Assign Foreign Keys

To reference other tables from one table, foreign keys have been assigned. Foreign keys are primary keys in other tables, therefore if anything would happen to that record (lets say it gets removed), this would effect our table. In the examples bellow, cascade was used (for delete and update) in order to apply same changes as it is in primary table. Depends on use case, there are other techniques (no action, remove, etc...) that could be used. Limitations can take place, meaning

there can be different techniques for delete and update (one can be remove while the other is cascade).

Code blocks bellow are showing DDL that was used in order to assign foreign keys to different tables.

```
01 | -- Certificate table
02 | ALTER TABLE Airport_DB.Certificate
03 | ADD CONSTRAINT Certificate_FK_Employee
04 | FOREIGN KEY (Employee_FK)
05 | REFERENCES Airport_DB.Employee(Employee_ID)
06 | ON DELETE CASCADE ON UPDATE CASCADE;
07 |
08 | ALTER TABLE Airport_DB.Certificate
09 | ADD CONSTRAINT Certificate_FK_Qualification
10 | FOREIGN KEY (Qualification_FK)
11 | REFERENCES Airport_DB.Qualification(Qualification_ID)
12 | ON DELETE CASCADE ON UPDATE CASCADE;
```

Listing 14: Certificate table foreign keys creation

```
01 | -- Company vehicle table
02 | ALTER TABLE Airport_DB.CompanyVehicle
03 | ADD CONSTRAINT CompanyVehicle_FK_Department
04 | FOREIGN KEY (Department_FK)
05 | REFERENCES Airport_DB.Department(Department_ID)
06 | ON DELETE CASCADE ON UPDATE CASCADE;
```

Listing 15: Company vehicle table foreign keys creation

```
01 | -- Customer table
02 | ALTER TABLE Airport_DB.Customer
03 | ADD CONSTRAINT Customer_FK_ParkingToken
04 | FOREIGN KEY (Token_FK)
05 | REFERENCES Airport_DB.ParkingToken(Token_ID);
```

Listing 16: Customer table foreign keys creation

```
01 | -- Employee table
02 | ALTER TABLE Airport_DB.Employee
03 | ADD CONSTRAINT Employee_FK_EmployeeParkingSpot
04 | FOREIGN KEY (Parking_Spot_FK)
05 | REFERENCES Airport_DB.EmployeeParkingSpot(Parking_Spot_ID)
06 | ON DELETE CASCADE ON UPDATE CASCADE;
07 |
08 | ALTER TABLE Airport_DB.Employee
09 | ADD CONSTRAINT Employee_FK_CompanyVehicle
10 | FOREIGN KEY (Vehicle_FK)
11 | REFERENCES Airport_DB.CompanyVehicle(Vehicle_ID)
12 | ON DELETE CASCADE ON UPDATE CASCADE;
```

```
13 |  
14 | ALTER TABLE Airport_DB.Employee  
15 | ADD CONSTRAINT Employee_FK_DepartmentID  
16 | FOREIGN KEY (Department_FK)  
17 | REFERENCES Airport_DB.Department(Department_ID)  
18 | ON DELETE CASCADE ON UPDATE CASCADE;  
19 |  
20 | ALTER TABLE Airport_DB.Employee  
21 | ADD CONSTRAINT Employee_FK_DepartmentManage  
22 | FOREIGN KEY (Manage_Department)  
23 | REFERENCES Airport_DB.Department(Department_ID)  
24 | ON DELETE CASCADE ON UPDATE CASCADE;  
25 |  
26 | ALTER TABLE Airport_DB.Employee  
27 | ADD CONSTRAINT Employee_FK_Employee  
28 | FOREIGN KEY (Supervisor)  
29 | REFERENCES Airport_DB.Employee(Employee_ID)  
30 | ON DELETE CASCADE ON UPDATE CASCADE;
```

Listing 17: Employee table foreign keys creation

```
01 | -- Flight table  
02 | ALTER TABLE Airport_DB.Flight  
03 | ADD CONSTRAINT Flight_FK_Runway  
04 | FOREIGN KEY (Runway_FK)  
05 | REFERENCES Airport_DB.Runway(Runway_ID)  
06 | ON DELETE CASCADE ON UPDATE CASCADE;  
07 |  
08 | ALTER TABLE Airport_DB.Flight  
09 | ADD CONSTRAINT Flight_FK_Airline  
10 | FOREIGN KEY (Airline_FK)  
11 | REFERENCES Airport_DB.Airline(Company_ID)  
12 | ON DELETE CASCADE ON UPDATE CASCADE;
```

Listing 18: Flight table foreign keys creation

```
01 | -- Name table  
02 | ALTER TABLE Airport_DB.Name  
03 | ADD CONSTRAINT Name_FK_Customer  
04 | FOREIGN KEY (Person_ID)  
05 | REFERENCES Airport_DB.Customer(Customer_ID)  
06 | ON DELETE CASCADE ON UPDATE CASCADE;  
07 |  
08 | ALTER TABLE Airport_DB.Name  
09 | ADD CONSTRAINT Name_FK_Employee  
10 | FOREIGN KEY (Person_ID)  
11 | REFERENCES Airport_DB.Employee(Employee_ID)  
12 | ON DELETE CASCADE ON UPDATE CASCADE;
```

Listing 19: Name table foreign keys creation

```
01 | -- Ticket table
02 | ALTER TABLE Airport_DB.Ticket
03 | ADD CONSTRAINT Ticket_FK_Flight
04 | FOREIGN KEY (Flight_FK)
05 | REFERENCES Airport_DB.Flight(Flight_ID)
06 | ON DELETE CASCADE ON UPDATE CASCADE;
07 |
08 | ALTER TABLE Airport_DB.Ticket
09 | ADD CONSTRAINT Ticket_FK_Customer
10 | FOREIGN KEY (Customer_FK)
11 | REFERENCES Airport_DB.Customer(Customer_ID)
12 | ON DELETE CASCADE ON UPDATE CASCADE;
```

Listing 20: Ticket table foreign keys creation

6.4 Create views

Views in SQL are essentially virtual tables (Sequeda, Depena, and Miranker 2009). They help us to create tables that are non-existing on the disk. With the view, we can also limit the information that is accessible from the table. As an example, we have generated two views that are returning either just customer names or just employee names.

Further discussion regarding views can be seen in appendix A7 (10.8).

```
01 | CREATE VIEW CustomerNames AS
02 | SELECT Person_ID, First_Name, Middle_Name, Last_Name
03 | FROM Name n
04 | WHERE n.Is_Employee = 0;
```


Listing 21: Creating customer view

Since view represents a virtual table, calling a view (Listing 22) is the same as calling the table.

```
01 | SELECT *
02 | FROM CustomerNames;
```

Listing 22: Calling customer view

Figure 41 shows output of the CustomerNames view (Listing 21).

SELECT * FROM CustomerNames |  Enter a SQL expression to filter results (use Ctrl+Space)

	123 Person_ID ▼	ABC First_Name ▼	ABC Middle_Name ▼	ABC Last_Name ▼
1	1	Heather	Jonathan	Young
2	5	Jessica	[NULL]	Ibarra
3	9	Tiffany	Amanda	Allen
4	13	Robert	[NULL]	Rasmussen
5	17	Jessica	Jeffrey	Maxwell
6	21	Marilyn	Sara	Barrera
7	25	Sarah	Jose	White
8	29	Jorge	Zachary	Martin
9	33	Kristen	Randy	Parks
10	37	Brittany	Luke	Young
11	41	Jacqueline	[NULL]	Johnson
12	45	David	[NULL]	Waller
13	49	Margaret	Megan	Hoffman
14	53	Jennifer	Jason	Hall
15	57	Andrea	Christopher	Flowers
16	61	Tracy	Brian	Lee
17	65	Jessica	Mark	Martinez
18	69	Travis	Caroline	Johnston
19	73	Ryan	Jasmine	Silva
20	77	Sarah	Manuel	Smith
21	81	Amber	Kim	Powers
22	85	Melanie	[NULL]	Harper
23	89	Paul	Andrew	Larsen
24	93	Christopher	Martin	Stone
25	97	Thomas	Christina	Summers
26	101	Shannon	Kevin	Moyer
27	105	Antonio	[NULL]	Benton
28	109	Kimberly	[NULL]	Martinez
29	113	Paul	[NULL]	Benitez
30	117	Christine	Laura	Young
31	121	Jenna	Norman	Ayala
32	125	Brian	[NULL]	Wilkins
33	129	John	Angela	Hamilton
34	133	Susan	[NULL]	Miller

Figure 41: Result of CustomerNames view

```

01 | CREATE VIEW EmployeeNames AS
02 | SELECT Person_ID, First_Name, Middle_Name, Last_Name
03 | FROM Name n
04 | WHERE n.Is_Employee = 1;

```

Listing 23: Creating Employee view

Employee names are returned (Listing 24) in identical format as customer names (Listing ??).


```

01 | SELECT *
02 | FROM EmployeeNames;

```

Listing 24: Calling employee view

With the Figure 42, we can see that employee names are returned (image is cropped).

SELECT * FROM EmployeeNames |  Enter a SQL expression to filter results (use Ctrl+Space)

	PERSON_ID	first_Name	Middle_Name	Last_Name
1	1	Ariana	Jeremy	Barrera
2	5	Lisa	Heather	Mendez
3	9	John	Kristen	Gutierrez
4	13	Nicole	Ryan	Walters
5	17	Casey	Nicole	Crosby
6	21	Timothy	Kathleen	Walls
7	25	William	Bryan	Davis
8	29	Lori	[NULL]	Rivera
9	33	Michael	Frank	Jordan
10	37	Ashley	Keith	Herrera
11	41	Kristina	Sandra	Moyer
12	45	Dennis	[NULL]	Stevens
13	49	Robert	Rachael	Logan
14	53	Caitlin	Michael	Jones
15	57	Jennifer	Brian	Reilly
16	61	David	Amanda	Williams
17	65	Jerry	[NULL]	Kelly
18	69	Stephanie	Samantha	Stewart
19	73	Michael	Frank	Cooper
20	77	Meghan	Michael	Fox
21	81	Anthony	Matthew	Myers
22	85	Matthew	Dean	Watkins
23	89	Sarah	Danny	Miller
24	93	Samantha	Linda	Watkins
25	97	Christopher	Connor	Robinson
26	101	Shawn	Brady	Young
27	105	Michael	Larry	Simmons
28	109	Kelsey	Gwendolyn	Sanchez
29	113	Amy	Cindy	Hamilton
30	117	Michelle	[NULL]	Wallace
31	121	Scott	Eric	Dawson
32	125	Julie	Megan	Dillon
33	129	Jesse	Susan	Adams
34	133	Jessica	Randy	Shah

Figure 42: Result of EmployeeNames view

6.5 Relationship Schema

After deploying DDL, we can inspect how the database looks in the UI. Comparing the relationship schema (figure 43) to our ER diagram (figure 40), we can see that

it looks similar to what was planned. There are some differences (relationship not showing, views being displayed) that are not in the ER diagram, other than that, it looks the same.

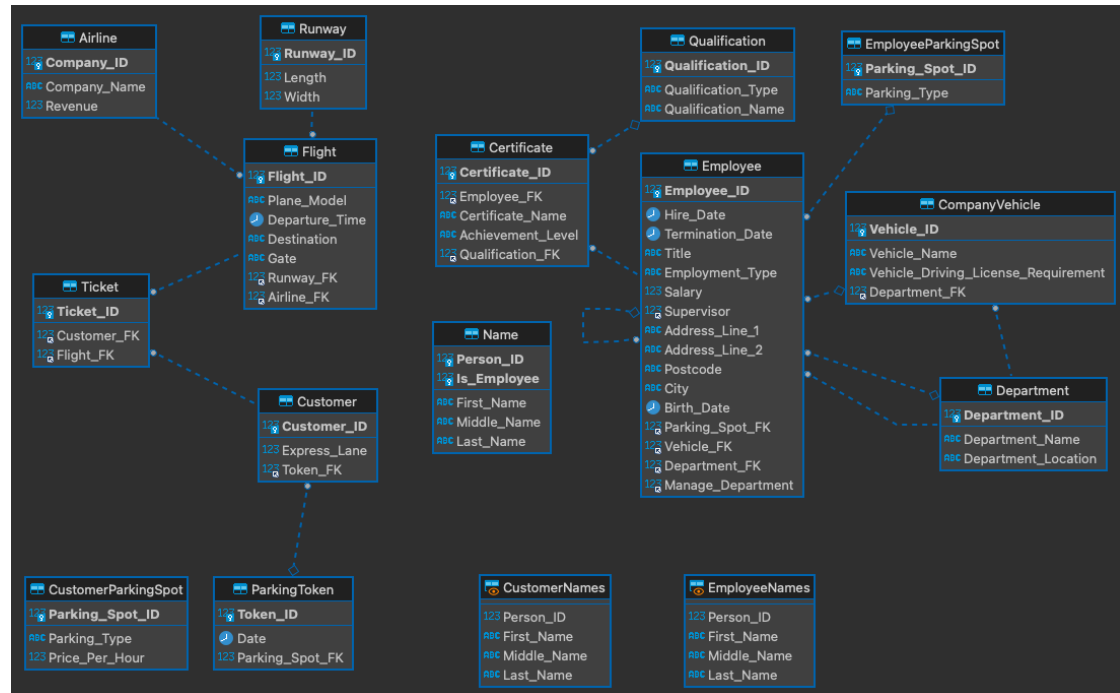


Figure 43: Relationship schema from UI

6.6 Create Procedure

Frequently used queries can be saved in SQL as stored procedures (Harrison and Feuerstein 2006). At the airport, parking is calculated by the hour. Instead of repeating SQL code, we can create the procedure and call it every time we need it.

Deeper discussion regarding stored procedures can be found in appendix A8 (10.9).

```
01 | CREATE DEFINER=`my_user`@`%`%
02 | PROCEDURE `Airport_DB`.`GetParkingToBePaid` (
    Filter_Customer_ID int(11))
03 | BEGIN
04 |     SELECT c.Customer_ID
05 |         , pt.Token_ID
06 |         , pt.`Date`
07 |         , cps.*
08 |         , cps.Price_Per_Hour * (SELECT TIMESTAMPDIFF
09 |             (HOUR, pt.`Date` , CURRENT_TIMESTAMP)
10 |             * (30 / 60) ) as ToPay
11 | FROM Customer c
```



```

12 |         INNER JOIN ParkingToken pt
13 |         ON c.Token_FK = pt.Token_ID
14 |         INNER JOIN CustomerParkingSpot cps
15 |         ON cps.Parking_Spot_ID = pt.Token_ID
16 |         WHERE c.Customer_ID = Filter_Customer_ID;
17 |     END;

```

Listing 25: Creating stored procedure

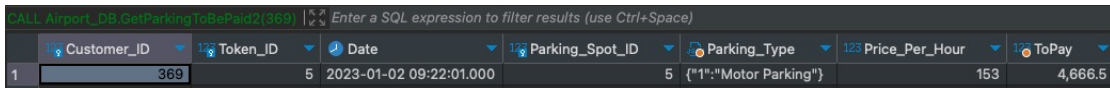
```

01 | CALL Airport_DB.GetParkingToBePaid(369);

```

Listing 26: Calling stored procedure

Result seen in figure 44 showcases that procedure calculated the price to be paid for the user with the ID of 369.



	Customer_ID	Token_ID	Date	Parking_Spot_ID	Parking_Type	Price_Per_Hour	ToPay
1	369	5	2023-01-02 09:22:01.000	5	("Motor Parking")	153	4,666.5

Figure 44: Result of GetParkingToBePaid procedure

6.7 Dummy Data Insertion

To generate fake data at first, simple "INSERT INTO" statements were used. This was done for testing purposes. 3 entries have been added for each table.

```

01 | -- Airline table
02 | INSERT INTO Airport_DB.Airline
03 | (Company_ID, Company_Name, Revenue)
04 | VALUES (1, 'Company_1', 20000.00);
05 |
06 | INSERT INTO Airport_DB.Airline
07 | (Company_ID, Company_Name, Revenue)
08 | VALUES (2, 'Company_2', 40000.00);
09 |
10 | INSERT INTO Airport_DB.Airline
11 | (Company_ID, Company_Name, Revenue)
12 | VALUES (3, 'Company_3', 80000.00);

```

Listing 27: Airline table data insertion

```

01 | -- Certificate table
02 | INSERT INTO Airport_DB.Certificate
03 | (Employee_FK, Certificate_ID, Certificate_Name,
04 | Achievement_Level, Qualification_FK)
05 | VALUES (1, 1, 'Certificate_1', 'Achievement_1', 1);
06 |
07 | INSERT INTO Airport_DB.Certificate
08 | (Employee_FK, Certificate_ID, Certificate_Name,

```

```
09 | Achievement_Level, Qualification_FK)
10 | VALUES (1, 2, 'Certificate_2', 'Achievement_2', 2);
11 |
12 | INSERT INTO Airport_DB.Certificate
13 | (Employee_FK, Certificate_ID, Certificate_Name,
14 | Achievement_Level, Qualification_FK)
15 | VALUES (2, 3, 'Certificate_1', 'Achievement_3', 3);
```

Listing 28: Certificate table data insertion

```
01 | -- Company Vehicle table
02 | INSERT INTO Airport_DB.CompanyVehicle
03 | (Vehicle_ID, Vehicle_Name,
04 | Vehicle_Driving_License_Requirement, Department_FK)
05 | VALUES (1, 'Vehicle_1', 'License_1', 1);
06 |
07 | INSERT INTO Airport_DB.CompanyVehicle
08 | (Vehicle_ID, Vehicle_Name,
09 | Vehicle_Driving_License_Requirement, Department_FK)
10 | VALUES (2, 'Vehicle_2', 'License_1', 1);
11 |
12 | INSERT INTO Airport_DB.CompanyVehicle
13 | (Vehicle_ID, Vehicle_Name,
14 | Vehicle_Driving_License_Requirement, Department_FK)
15 | VALUES (3, 'Vehicle_3', 'License_2', 2);
```

Listing 29: Company table data insertion

```
01 | -- Customer table
02 | INSERT INTO Airport_DB.Customer
03 | (Customer_ID, Express_Lane, Token_FK)
04 | VALUES (1, 0, 1);
05 |
06 | INSERT INTO Airport_DB.Customer
07 | (Customer_ID, Express_Lane, Token_FK)
08 | VALUES (2, 1, 2);
09 |
10 | INSERT INTO Airport_DB.Customer
11 | (Customer_ID, Express_Lane)
12 | VALUES (3, 0);
```

Listing 30: Customer table data insertion

```
01 | -- Customer Parking Spot table
02 | INSERT INTO Airport_DB.CustomerParkingSpot
03 | (Parking_Spot_ID, Parking_Type, Price_Per_Hour)
04 | VALUES (1, '{"1":"Car Parking"}', 6);
05 |
06 | INSERT INTO Airport_DB.CustomerParkingSpot
07 | (Parking_Spot_ID, Parking_Type, Price_Per_Hour)
```

```
08 | VALUES (2, '{"1":"Motor Parking"}', 6);
09 |
10 | INSERT INTO Airport_DB.CustomerParkingSpot
11 | (Parking_Spot_ID, Parking_Type, Price_Per_Hour)
12 | VALUES (3, '{"1":"Car Parking", "2":"Handicap parking"}', 5)
    ;
```

Listing 31: Customer Parking Spot table data insertion

```
01 | -- Department table
02 | INSERT INTO Airport_DB.Department
03 | (Department_ID, Department_Name, Department_Location)
04 | VALUES (1, 'Department_1', 'Location_1');
05 |
06 | INSERT INTO Airport_DB.Department
07 | (Department_ID, Department_Name, Department_Location)
08 | VALUES (2, 'Department_2', 'Location_2');
09 |
10 | INSERT INTO Airport_DB.Department
11 | (Department_ID, Department_Name, Department_Location)
12 | VALUES (3, 'Department_3', 'Location_3');
```

Listing 32: Department table data insertion

```
01 | -- Employee table
02 | INSERT INTO Airport_DB.Employee
03 | (Employee_ID, Hire_Date, Termination_Date, Title,
04 | Employment_Type, Salary, Supervisor, Address_Line_1,
05 | Address_Line_2, Postcode, City, Birth_Date, Parking_Spot_FK,
06 | Vehicle_FK, Department_FK, Manage_Department)
07 | VALUES (1, '2022-1-1', null, 'Title_1', 'Type_1',
08 | 90000, null, 'Address_Line_1_1', null, 'Postcode_1',
09 | 'City_1', '1980-12-1', 1, 1, 1, 1);
10 |
11 | INSERT INTO Airport_DB.Employee
12 | (Employee_ID, Hire_Date, Termination_Date, Title,
13 | Employment_Type, Salary, Supervisor, Address_Line_1,
14 | Address_Line_2, Postcode, City, Birth_Date, Parking_Spot_FK,
15 | Vehicle_FK, Department_FK, Manage_Department) Vehicle_FK,
16 | Department_FK, Manage_Department)
17 | VALUES (2, '2022-2-2', '2022-12-31', 'Title_2',
18 | 'Type_2', 60000, 1, 'Address_Line_1_2', 'Address_Line_2_2',
19 | 'Postcode_2', 'City_2', '1980-12-3', 2, 2, 2, 2);
20 |
21 | INSERT INTO Airport_DB.Employee
22 | (Employee_ID, Hire_Date, Termination_Date, Title,
23 | Employment_Type, Salary, Supervisor, Address_Line_1,
24 | Address_Line_2, Postcode, City, Birth_Date, Parking_Spot_FK,
25 | Vehicle_FK, Department_FK, Manage_Department)
26 | VALUES (3, '2022-3-3', null, 'Title_3', 'Type_3',
27 | 30000, 2, 'Address_Line_1_3', null, 'Postcode_3',
```

```
28 | 'City_3', '1980-12-3', 2, null, 2, null);
```

Listing 33: Employee table data insertion

```
01 | -- Employee Parking Spot table
02 | INSERT INTO Airport_DB.EmployeeParkingSpot
03 | (Parking_Spot_ID, Parking_Type)
04 | VALUES (1, '{"1":"Car Parking", "2":"CEO parking"}');
05 |
06 | INSERT INTO Airport_DB.EmployeeParkingSpot
07 | (Parking_Spot_ID, Parking_Type)
08 | VALUES (2, '{"1":"Motor Parking"}');
09 |
10 | INSERT INTO Airport_DB.EmployeeParkingSpot
11 | (Parking_Spot_ID, Parking_Type)
12 | VALUES (3, '{"1":"Car Parking", "2":"Handicap parking"}');
```

Listing 34: Employee Parking Spot table data insertion

```
01 | -- Flight table
02 | INSERT INTO Airport_DB.Flight
03 | (Flight_ID, Plane_Model, Departure_Time,
04 | Destination, Gate, Runway_FK, Airline_FK)
05 | VALUES (1, 'Model_1', '2022-01-10 01:00:00',
06 | 'Destination_1', 'Gate_1', 1, 1);
07 |
08 | INSERT INTO Airport_DB.Flight
09 | (Flight_ID, Plane_Model, Departure_Time,
10 | Destination, Gate, Runway_FK, Airline_FK)
11 | VALUES (2, 'Model_2', '2022-02-20 22:22:00',
12 | 'Destination_2', 'Gate_2', 1, 2);
13 |
14 | INSERT INTO Airport_DB.Flight
15 | (Flight_ID, Plane_Model, Departure_Time,
16 | Destination, Gate, Runway_FK, Airline_FK)
17 | VALUES (3, 'Model_3', '2022-03-30 09:30:00',
18 | 'Destination_3', 'Gate_3', 2, 2);
```

Listing 35: Flight table data insertion

```
01 | -- Name table
02 | INSERT INTO Airport_DB.Name
03 | (Person_ID, First_Name, Middle_Name, Last_Name, Is_Employee)
04 | VALUES (1, 'FN_1', null, 'LN_1', 1);
05 |
06 | INSERT INTO Airport_DB.Name
07 | (Person_ID, First_Name, Middle_Name, Last_Name, Is_Employee)
08 | VALUES (2, 'FN_2', 'MN_2', 'LN_2', 1);
09 |
10 | INSERT INTO Airport_DB.Name
```

```
11 | (Person_ID, First_Name, Middle_Name, Last_Name, Is_Employee)
12 | VALUES (3, 'FN_3', null, 'LN_3', 1);
13 |
14 | INSERT INTO Airport_DB.Name
15 | (Person_ID, First_Name, Middle_Name, Last_Name, Is_Employee)
16 | VALUES (1, 'FN_4', 'MN_4', 'LN_4', 0);
17 |
18 | INSERT INTO Airport_DB.Name
19 | (Person_ID, First_Name, Middle_Name, Last_Name, Is_Employee)
20 | VALUES (2, 'FN_5', 'MN_5', 'LN_5', 0);
21 |
22 | INSERT INTO Airport_DB.Name
23 | (Person_ID, First_Name, Middle_Name, Last_Name, Is_Employee)
24 | VALUES (3, 'FN_6', null, 'LN_6', 0);
```

Listing 36: Name table data insertion

```
01 | -- Parking Token table
02 | INSERT INTO Airport_DB.ParkingToken
03 | (Token_ID, `Date`, Parking_Spot_FK)
04 | VALUES (1, '2022-01-10 11:11:00',1);
05 |
06 | INSERT INTO Airport_DB.ParkingToken
07 | (Token_ID, `Date`, Parking_Spot_FK)
08 | VALUES (2, '2022-02-20 22:00:00',2);
09 |
10 | INSERT INTO Airport_DB.ParkingToken
11 | (Token_ID, `Date`, Parking_Spot_FK)
12 | VALUES (3, '2022-03-30 15:30:00',3);
```

Listing 37: Parking Token table data insertion

```
01 | -- Runway table
02 | INSERT INTO Airport_DB.Runway
03 | (Runway_ID, `Length`, Width)
04 | VALUES (1, 100, 10);
05 |
06 | INSERT INTO Airport_DB.Runway
07 | (Runway_ID, `Length`, Width)
08 | VALUES (2, 200, 20);
09 |
10 | INSERT INTO Airport_DB.Runway
11 | (Runway_ID, `Length`, Width)
12 | VALUES (3, 300, 30);
```

Listing 38: Runway table data insertion

```
01 | -- Ticket table
02 | INSERT INTO Airport_DB.Ticket
03 | (Ticket_ID, Customer_FK, Flight_FK)
```

```
04 | VALUES (1, 1, 1);
05 |
06 | INSERT INTO Airport_DB.Ticket
07 | (Ticket_ID, Customer_FK, Flight_FK)
08 | VALUES (2, 2, 1);
09 |
10 | INSERT INTO Airport_DB.Ticket
11 | (Ticket_ID, Customer_FK, Flight_FK)
12 | VALUES (3, 3, 3);
```

Listing 39: Ticket table data insertion

```
01 | -- Qualification table
02 | INSERT INTO Airport_DB.Qualification
03 | (Qualification_ID, Qualification_Type, Qualification_Name)
04 | VALUES (1, "Type_1", "Name_1");
05 |
06 | INSERT INTO Airport_DB.Qualification
07 | (Qualification_ID, Qualification_Type, Qualification_Name)
08 | VALUES (2, "Type_2", "Name_2");
09 |
10 | INSERT INTO Airport_DB.Qualification
11 | (Qualification_ID, Qualification_Type, Qualification_Name)
12 | VALUES (3, "Type_3", "Name_3");
```

Listing 40: Qualification table data insertion

To automate data insertion, we have also created Python script. Further discussion can be seen in appendix A2 (10.3).

7 Testing of the database

Testing of the database was done with Python script. Deeper dive can be found in appendix A10 (10.11).

At the start of the script, packages need to be imported. As before, "mysql.connector" is used to connect to our DB.

```
print("=====")
print("Unit testing database")
print("=====")

import mysql.connector
from mysql.connector import Error
```

First function (create_server_connection) connects Python to the database. Function accepts arguments that are required for the connection (host name, port, user name, user password). Connection is then tried to be established, if successful function returns connection with error code as none and error message as none.

Otherwise we are expecting error 2003 (host cannot be reached), 1045 (invalid credentials) or any other error. In the case of 2003 and 1045, no connection or error message is returned just error number (1 or 2). If there is any other error, no connection is returned, error number is 0 and error message is returned.

```
#=====
# SQL Functions
#=====

def create_server_connection(host_name, port, user_name, user_password):
    """
    Return connection (if made), error code (if made) and error (if made)
    """
    try:
        connection = mysql.connector.connect(
            host=host_name,
            user=user_name,
            passwd=user_password,
            port= port
        )
        return connection, None, None
    except mysql.connector.Error as err:
        if (err.errno == 2003):
            return None, 1, None
        elif(err.errno == 1045):
            return None, 2, None
        else:
            print(f"Error: '{err}'")
            return None, 0, err
```

Execute_query executes given query based on the connection on which cursor is generated from. Function then tries to execute the query and if successful it returns 0 with no error messages. Otherwise, 4 error codes are expected:

- 062 - error, duplicate ID
- 1292 - error, incorrect data due to data type
- 4025 - error, incorrect data due to constraints
- 1265 - warning, incorrect data due to data type

Each error returns its own number (1-4) with error message. If an error occurs that is not in the list, 0 is returned with error message.

```
def execute_query(connection, query):
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        connection.commit()
        return 0, None
```

```
except mysql.connector.Error as err:
    if (err.errno == 1062):
        #print("Cannot insert, duplicate entry")
        return 1, err.msg
    elif(err.errno == 1292):
        #print("Incorrect value")
        return 2, err.msg
    elif(err.errno == 4025):
        #print("Incorrect value")
        return 3, err.msg
    elif(err.errno == 1265):
        #print("Incorrect value")
        return 4, err.msg
    else:
        print(f"Error: '{err}'")
        print(f"Query: '{query}'")
```

This function gets IDs from the table. Connection and query (IDquery) are passed as parameters in at first, then "Airport_DB" database is used, afterwards cursor is created based on the connection. Function tries to execute the query, if successful it gets data from the cursor and returns IDs (that are in the position 0) from the table. If unsuccessful, error message is printed with the query that didn't execute.

```
def getQueryResults(connection, IDquery):
    execute_query(connection, "USE Airport_DB;")
    cursor = connection.cursor()
    try:
        cursor.execute(IDquery)
        table = cursor.fetchall()
        return([x[0] for x in table])
    except Error as err:
        print(f"Error: '{err}'")
        print(f"Query: '{IDquery}'")
```

This function gets gets all the data from the table.

```
def get_data(entry):
    output = (str(tuple(entry.keys())).replace("'", ""),
    ↪ str(tuple(entry.values())))
    return output if output[0][-2] != "," else
    ↪ (output[0][: -2] + output[0][-1], output[1][: -2] + output[1][-1])
```

Populate function inserts data into the table (target) based on what is in the list. Function iterates the list and for every element it calls execute_query function with connection (that is passed to the function) and query that needs to be executed. Query itself is generated in populate function, it gets the data based on what is in the list and what table we want to insert into.


```
def populate(connection, lst, target):
    for entry in lst:
        columns, values= get_data(entry)
        query = ("INSERT INTO "+ target+ " "+ columns+ " VALUES "+
        ↪ values).replace("'null'", "null")
        return execute_query(connection, query)
```

To test removal of the employee, function is created. It accepts an argument with connection and employee ID. In the function, we create two queries (one for removing employee and one for getting number of employees with passed ID). Function firstly makes sure it uses "Airport_DB" database, then it gets number of employees that have requested ID. In next phase, employee with requested ID has been removed and function was called again to check how many employees are there with requested ID. At the end script checks if number effected IDs has changed, if it has it returns 0, otherwise it returns 1.

```
=====
# Test functions
=====

def testRemoveEmployee(dbConnection, employeeID):
    removeQuery = f"DELETE FROM Employee WHERE Employee_ID = {employeeID};"
    countQuery = f"SELECT count(*) FROM (SELECT * From Employee Where
    ↪ Employee_ID = {employeeID}) AS count_of_query;"

    execute_query(dbConnection, "USE Airport_DB;")
    numOfResultsAtTheStart = getQueryResults(dbConnection, countQuery)[0]
    execute_query(dbConnection, removeQuery)
    numOfResultsAtTheEnd = getQueryResults(dbConnection, countQuery)[0]
    if(numOfResultsAtTheStart != numOfResultsAtTheEnd):
        return 0
    elif(numOfResultsAtTheStart == numOfResultsAtTheEnd):
        return 1
```

This function accepts connection and ID of employee who we want to find. Function then creates a query that selects matching employee and executes it. If returned query has length of 1 (it has one item), that item is returned. Otherwise none is returned.

```
def testGetEmployeeID(dbConnection, employeeID):
    employeeQuery = f"SELECT * FROM Employee Where Employee_ID = {employeeID};"
    employeeIDs = getQueryResults(dbConnection, employeeQuery)
    if(len(employeeIDs) == 1):
        return(employeeIDs[0])
    else:
        return(None)
```

The testGetParkingSpotID function works exactly the same as the testGetEmployeeID function. The only difference is in naming, otherwise functionality is the same.

```
def testGetParkingSpotID(dbConnection, parkingSpotID):
    parkingSpotQuery = f"SELECT * FROM CustomerParkingSpot Where
    ↪ Parking_Spot_ID = {parkingSpotID};"
    parkingSpotIDs = getQueryResults(dbConnection, parkingSpotQuery)
    if(len(parkingSpotIDs) == 1):
        return(parkingSpotIDs[0])
    else:
        return(None)
```

Function testEmployeeInsertion accepts 17 arguments. After they are accepted into the function, they are used in the dictionary to build the employee. In next step, function makes sure it uses "Airport_DB", after which it tries to get the employee with the same ID that we have tried to insert. Next it tries to insert the data into the employee table based on the generated dictionary. Based on the results that were generated, we are expecting to return one of the error codes:

- 1 - there isn't an employee with this ID and query had no problems - success
- 2 - there is already employee with the same ID - fail
- 3 - cannot insert employee due to wrong attribute - fail
- 4 - there is already employee with the same ID and cannot insert employee due to wrong attribute - fail
- 0 - other error, print error message - fail

```
def testEmployeeInsertion(dbConnection, employeeID, hireDate, terminationDate,
    ↪ title, employmentType, salary,
    supervisor, addressLine1, addressLine2, postcode, city, birthDate,
    ↪ parkingSpotFK, vehicleFK, departmentFK, manageDepartment):
    employees =[{
        "Employee_ID": employeeID,
        "Hire_Date": hireDate,
        "Termination_Date": terminationDate,
        "Title": title,
        "Employment_Type": employmentType,
        "Salary": salary,
        "Supervisor": supervisor,
        "Address_Line_1": addressLine1,
        "Address_Line_2": addressLine2,
        "Postcode": postcode,
        "City": city,
        "Birth_Date": birthDate,
        "Parking_Spot_FK": parkingSpotFK,
        "Vehicle_FK": vehicleFK,
        "Department_FK": departmentFK,
        "Manage_Department": manageDepartment,
    }]

    execute_query(dbConnection, "USE Airport_DB;")
```

```
getEmployee = testGetEmployeeID(dbConnection, employeeID)
tryInsertingEmployee = populate(dbConnection,employees,"Employee")

if(getEmployee == None and tryInsertingEmployee[0] == 0): # There isn't an
    ↪ employee with that ID in the DB and execute_query returns 0 - no
    ↪ problem, this should be success
    #print("Employee successfully inserted")
    return 1, "Employee successfully inserted"
elif(getEmployee == employeeID and tryInsertingEmployee[0] == 1): # If
    ↪ returned employeeID matches with inserted ID and execute_query returns
    ↪ 1 - employee already exists
    #print("Failed to insert employee, there is already an employee with
    ↪ this ID")
    return 2, "Failed to insert employee, there is already an employee with
    ↪ this ID"
elif(getEmployee == None and tryInsertingEmployee[0] == 2):
    #print("Failed to insert employee, one of the attributes is in wrong
    ↪ type")
    #print(f"Incorrect value: {tryInsertingEmployee[1]}")
    return 3, "Failed to insert employee, one of the attributes is in wrong
    ↪ type", f"Incorrect value: {tryInsertingEmployee[1]}"
elif(getEmployee == employeeID and tryInsertingEmployee[0] == 2):
    #print("Failed to insert employee, one of the attributes is in wrong
    ↪ type and there is already an employee with this ID")
    #print(f"Incorrect value: {tryInsertingEmployee[1]}")
    return 4, "Failed to insert employee, one of the attributes is in wrong
    ↪ type and there is already an employee with this ID", f"Incorrect
    ↪ value: {tryInsertingEmployee[1]}"
else:
    print("Unknown error, read the error message:")
    print(tryInsertingEmployee[1])
    return 0
```

Function testCustomerParkingInsertion works similar to testEmployeeInsertion function. The only difference is in error codes. They return as follows:

- 1 - there isn't a customer parking spot with this ID and query had no problems - success
- 2 - there is already a customer parking spot with the same ID - fail
- 3 - cannot insert customer parking spot due to wrong constraint limitations - fail
- 4 - there is already customer parking spot with the same ID and cannot insert customer parking spot due to wrong constraint limitations - fail
- 5 - cannot insert customer parking spot due to wrong attribute - fail
- 6 - there is already customer parking spot with the same ID cannot insert customer parking spot due to wrong attribute - fail

- 0 - other error, print error message - fail

```
def testCustomerParkingInsertion(dbConnection, parkingSpotID, parkingType,
    pricePerHour):
    customer_parking_spots = [{
        "Parking_Spot_ID": parkingSpotID,
        "Parking_Type": parkingType,
        "Price_Per_Hour": pricePerHour
    }]

    execute_query(dbConnection, "USE Airport_DB;")
    getCustomerParking = testGetParkingSpotID(dbConnection, parkingSpotID)
    tryInsertingCustomer = populate(dbConnection, customer_parking_spots,
        "CustomerParkingSpot")

    if(getCustomerParking == None and tryInsertingCustomer[0] == 0):
        return 1, "Customer parking spot successfully inserted"
    elif(getCustomerParking == parkingSpotID and tryInsertingCustomer[0] == 1):
        return 2, "Failed to insert customer parking spot, there is already an
        customer parking spot with this ID"
    elif(getCustomerParking == None and tryInsertingCustomer[0] == 3):
        return 3, "Failed to insert customer parking spot, one of the
        attributes is in wrong type, check constraint", f"Incorrect value:
        {tryInsertingCustomer[1]}"
    elif(getCustomerParking == parkingSpotID and tryInsertingCustomer[0] == 3):
        return 4, "Failed to insert customer parking spot, one of the
        attributes is in wrong type and there is already an customer
        parking spot with this ID, check constraint", f"Incorrect value:
        {tryInsertingCustomer[1]}"
    elif(getCustomerParking == None and tryInsertingCustomer[0] == 4):
        return 5, "Failed to insert customer parking spot, one of the
        attributes is in wrong type", f"Incorrect value:
        {tryInsertingCustomer[1]}"
    elif(getCustomerParking == parkingSpotID and tryInsertingCustomer[0] == 4):
        return 6, "Failed to insert customer parking spot, one of the
        attributes is in wrong type and there is already an customer
        parking spot with this ID", f"Incorrect value:
        {tryInsertingCustomer[1]}"
    else:
        print("Unknown error, read the error message:")
        print(tryInsertingCustomer[1])
        return 0
```

To call all of the tests, mainTest function is created. It has numberOfPasses (correct answers) and numberOfFails (wrong answers) set to 0, this increments while function is running. At the top there is also a connection that is formed.

```
=====
# Main test
=====
```

```
def mainTest():
    print("=====")
    print("Starting unit testing")
    print("=====")
    numberOfPasses = 0
    numberOfFails = 0
    connection, _ , _ = create_server_connection("localhost","30330",
        ↪ "my_user", "my_password")
```

Test1: check valid connection to mariadb-galera-0 instance.

```
# Test1:
#     Task: check connection to mariadb-galera-0 with valid credentials
#     Expected result: 0 - connection should be successful
test1 = create_server_connection("localhost","30330", "my_user",
    ↪ "my_password")
if(test1[1] == None):
    print("Test 1 pass: Connection to mariadb-galera-0 was established")
    numberOfPasses += 1
else:
    print("Test 1 fail: Connection could not be made to mariadb-galera-0")
    numberOfFails += 1
```

Test2: check valid connection to mariadb-galera-1 instance.

```
# Test2:
#     Task: check connection to mariadb-galera-1 with valid credentials
#     Expected result: 0 - connection should be successful
test2 = create_server_connection("localhost","30331", "my_user",
    ↪ "my_password")
if(test2[1] == None):
    print("Test 2 pass: Connection to mariadb-galera-1 was established")
    numberOfPasses += 1
else:
    print("Test 2 fail: Connection could not be made to mariadb-galera-1")
    numberOfFails += 1
```

Test3: check valid connection to mariadb-galera-2 instance.

```
# Test3:
#     Task: check connection to mariadb-galera-2 with valid credentials
#     Expected result: 0 - connection should be successful
test3 = create_server_connection("localhost","30332", "my_user",
    ↪ "my_password")
if(test3[1] == None):
    print("Test 3 pass: Connection to mariadb-galera-2 was established")
    numberOfPasses += 1
else:
    print("Test 3 fail: Connection could not be made to mariadb-galera-2")
    numberOfFails += 1
```

Test4: check valid connection to mariadb-galera-3 instance.

```
# Test4:
#     Task: check connection to mariadb-galera-3 with valid credentials
#     Expected result: 0 - connection should be successful
test4 = create_server_connection("localhost","30333", "my_user",
    ↪ "my_password")
if(test4[1] == None):
    print("Test 4 pass: Connection to mariadb-galera-3 was established")
    numberOfPasses += 1
else:
    print("Test 4 fail: Connection could not be made to mariadb-galera-3")
    numberOfFails += 1
```

Test5: check invalid connection to unknown database (wrong port).

```
# Test5:
#     Task: check connection to unknown db with valid credentials
#     Expected result: 1 - connection should not be successful
test5 = create_server_connection("localhost","30336", "my_user",
    ↪ "my_password")
if(test5[1] == 1):
    print("Test 5 pass: Connection to unknown db was not established")
    numberOfPasses += 1
else:
    print(test5[2])
    print("Test 5 fail: Connection could be made to unknown db")
    numberOfFails += 1
```

Test6: check invalid connection with wrong IP.

```
# Test6:
#     Task: check connection to mariadb-galera-0 with invalid ip (not
    ↪ localhost)
#     Expected result: 1 - connection should not be successful
test6 = create_server_connection("not_localhost","30330", "my_user",
    ↪ "my_password")
if(test6[1] == 1):
    print("Test 6 pass: Connection to mariadb-galera-0 with invalid ip (not
    ↪ localhost) was not possible")
    numberOfPasses += 1
else:
    print("Test 6 fail: Connection to mariadb-galera-0 with invalid ip was
    ↪ possible")
    numberOfFails += 1
```

Test7: check invalid connection with wrong user.

```
# Test7:
#     Task: check connection to mariadb-galera-0 with invalid valid user
#     Expected result: 2 - connection should not be successful
test7 = create_server_connection("localhost","30330", "not_my_user",
    ↪ "my_password")
if(test7[1] == 2):
```

```
print("Test 7 pass: Connection to mariadb-galera-0 with invalid user
↳ was not possible")
numberOfPasses += 1
else:
print("Test 7 fail: Connection to mariadb-galera-0 with invalid user
↳ was possible")
numberOfFails += 1
```

Test8: check invalid connection with wrong password.

```
# Test8:
# Task: check connection to mariadb-galera-0 with invalid valid
↳ password
# Expected result: 0 - connection should not be successful
test8 = create_server_connection("localhost","30330", "my_user",
↳ "not_my_password")
if(test8[1] == 2):
print("Test 8 pass: Connection to mariadb-galera-0 with invalid
↳ password was not possible")
numberOfPasses += 1
else:
print("Test 8 fail: Connection to mariadb-galera-0 with invalid
↳ password was possible")
numberOfFails += 1
```

Test9: try removing employee with ID 999999999

```
# Test9:
# Task: try removing employees
# Expected result: 0 - employee 999999999 shouldn't be found
test9 = testRemoveEmployee(connection, 999999999)
if(test9 == 1):
print("Test 9 pass: No employee was removed")
numberOfPasses += 1
else:
print("Test 9 fail: Employee was removed")
numberOfFails += 1
```

Test10: searching for employee with ID 999999999

```
# Test10:
# Task: get Employee data that matches 99999 from the DB
# Expected result: None - there shouldn't be an employee with this ID
test10 = testGetEmployeeID(connection, 999999999)
if(test10 == None):
print("Test 10 pass: No employee was not found")
numberOfPasses += 1
else:
print("Test 10 fail: Employee was found")
numberOfFails += 1
```

Test11: try inserting employee

```
# Test11:
#     Task: generate an employee
#     Expected result: 1 - employee should be created
test11 = testEmployeeInsertion(dbConnection = connection, employeeID =
→ 999999999, hireDate = "2022-12-13", terminationDate = "2022-12-14",
→ title = "Test_Title", employmentType = "Test_Employment", salary =
→ 9000, supervisor = 5, addressLine1 = "Test_Address_Line_1",
→ addressLine2 = "Test_Address_Line_2", postcode = "Test_Postcode", city
→ = "Test_City", birthDate = "2022-11-14", parkingSpotFK = 17, vehicleFK
→ = 21, departmentFK = 5, manageDepartment = "null")
if(test11[0] == 1):
    print(f"Test 11 pass: {test11[1]}")
    numberOfPasses += 1
else:
    print("Test 11 fail")
    numberOfFails += 1
```

Test 12: try inserting employee with same ID

```
# Test12:
#     Task: fail to generate an employee based on duplicate ID
#     Expected result: 2 - employee should be created
test12 = testEmployeeInsertion(dbConnection = connection, employeeID =
→ 999999999, hireDate = "2022-12-13", terminationDate = "2022-12-14",
→ title = "Test_Title", employmentType = "Test_Employment", salary =
→ 9000, supervisor = 5, addressLine1 = "Test_Address_Line_1",
→ addressLine2 = "Test_Address_Line_2", postcode = "Test_Postcode", city
→ = "Test_City", birthDate = "2022-11-14", parkingSpotFK = 17, vehicleFK
→ = 21, departmentFK = 5, manageDepartment = "null")
if(test12[0] == 2):
    print(f"Test 12 pass: {test12[1]}")
    numberOfPasses += 1
else:
    print("Test 12 fail")
    numberOfFails += 1
```

Test 13: insert employee with wrong data type

```
# Test13:
#     Task: fail to generate an employee based on wrong attribute -
→ hireDate
#     Expected result: 3 - employee should be created
test13 = testEmployeeInsertion(dbConnection = connection, employeeID =
→ 999999998, hireDate = "hireDate", terminationDate = "2022-12-14", title
→ = "Test_Title", employmentType = "Test_Employment", salary = 9000,
→ supervisor = 5, addressLine1 = "Test_Address_Line_1", addressLine2 =
→ "Test_Address_Line_2", postcode = "Test_Postcode", city = "Test_City",
→ birthDate = "2022-11-14", parkingSpotFK = 17, vehicleFK = 21,
→ departmentFK = 5, manageDepartment = "null")
if(test13[0] == 3):
    print(f"Test 13 pass: {test13[1]} {test13[2]}")
    numberOfPasses += 1
```



```
else:
    print("Test 13 fail")
    numberOfFails += 1
```

Test 14: insert employee with wrong data type and duplicate ID

```
# Test14:
#     Task: fail to generate an employee based on duplicate ID and wrong
#     attribute - hireDate
#     Expected result: 4 - employee should be created
test14 = testEmployeeInsertion(dbConnection = connection, employeeID =
    ↪ 999999999, hireDate = "hireDate", terminationDate = "2022-12-14", title
    ↪ = "Test_Title", employmentType = "Test_Employment", salary = 9000,
    ↪ supervisor = 5, addressLine1 = "Test_Address_Line_1", addressLine2 =
    ↪ "Test_Address_Line_2", postcode = "Test_Postcode", city = "Test_City",
    ↪ birthDate = "2022-11-14", parkingSpotFK = 17, vehicleFK = 21,
    ↪ departmentFK = 5, manageDepartment = "null")
if(test14[0] == 4):
    print(f"Test 14 pass: {test14[1]} {test14[2]}")
    numberOfPasses += 1
else:
    print("Test 14 fail")
    numberOfFails += 1
```

Test 15: remove employee

```
# Test15:
#     Task: remove employee generated from above
#     Expected result: 1 - employee 999999999 shall be removed
test15 = testRemoveEmployee(dbConnection = connection, employeeID =
    ↪ 999999999)
if(test15 == 0):
    print("Test 15 pass: Employee was removed")
    numberOfPasses += 1
else:
    print("Test 15 fail: No employee was removed")
    numberOfFails += 1
```

Test 16: insert parking spot

```
# Test16:
#     Task: generate a parking spot
#     Expected result: 1 - parking spot should be created
test16 = testCustomerParkingInsertion(dbConnection = connection,
    ↪ parkingSpotID = 999999999, parkingType = '{"1": "Car Parking"}' ,
    ↪ pricePerHour = "6")
if(test16[0] == 1):
    print(f"Test 16 pass: {test16[1]}")
    numberOfPasses += 1
else:
    print("Test 16 fail")
    numberOfFails += 1
```

Test 17: insert parking spot based on duplicate ID

```
# Test17:
#     Task: fail to generate a customer parking spot based on duplicate
#     ↪ ID
#     Expected result: 2 - there shouldn't be an customer parking spot
#     ↪ with this ID
test17 = testCustomerParkingInsertion(dbConnection = connection,
#     ↪ parkingSpotID = 999999999, parkingType = '{"1": " Car Parking "}' ,
#     ↪ pricePerHour = "6")
if(test17[0] == 2):
    print(f"Test 17 pass: {test17[1]}")
    numberOfPasses += 1
else:
    print("Test 17 fail")
    numberOfFails += 1
```

Test 18: insert parking spot with price per hour being too low

```
# Test18:
#     Task: fail to generate a customer parking spot based on
#     ↪ pricePerHour being 3 - under the limits
#     Expected result: 3 - customer parking spot shouldn't be inserted
#     ↪ due to constraints
test18 = testCustomerParkingInsertion(dbConnection = connection,
#     ↪ parkingSpotID = 999999997, parkingType = '{"1": " Car Parking "}' ,
#     ↪ pricePerHour = "3")
if(test18[0] == 3):
    print(f"Test 18 pass: {test18[1]}")
    numberOfPasses += 1
else:
    print("Test 18 fail")
    numberOfFails += 1
```

Test 19: insert parking spot with price per hour being too low and ID being the same

```
# Test19:
#     Task: fail to generate a customer parking spot based on
#     ↪ pricePerHour being 3 - under the limits and ID of 999999999 is taken
#     Expected result: 4 - customer parking spot shouldn't be inserted
#     ↪ due to constraints and ID
test19 = testCustomerParkingInsertion(dbConnection = connection,
#     ↪ parkingSpotID = 999999999, parkingType = '{"1": " Car Parking "}' ,
#     ↪ pricePerHour = "3")
if(test19[0] == 4):
    print(f"Test 19 pass: {test19[1]}")
    numberOfPasses += 1
else:
    print("Test 19 fail")
    numberOfFails += 1
```

Test 20: insert parking spot with price per hour being too high

```
# Test20:
#     Task: fail to generate a customer parking spot based on
#     ↪ pricePerHour being 400 - over the limits
#     Expected result: 3 - customer parking spot shouldn't be inserted
#     ↪ due to constraints
test20 = testCustomerParkingInsertion(dbConnection = connection,
    ↪ parkingSpotID = 999999997, parkingType = '{"1": "Car Parking"}' ,
    ↪ pricePerHour = "400")
if(test20[0] == 3):
    print(f"Test 20 pass: {test20[1]}")
    numberOfPasses += 1
else:
    print("Test 20 fail")
    numberOfFails += 1
```

Test 21: insert parking spot with price per hour being too high and ID being the same

```
# Test21:
#     Task: fail to generate a customer parking spot based on
#     ↪ pricePerHour being 400 - over the limits
#     Expected result: 4 - customer parking spot shouldn't be inserted
#     ↪ due to constraints
test21 = testCustomerParkingInsertion(dbConnection = connection,
    ↪ parkingSpotID = 999999999, parkingType = '{"1": "Car Parking"}' ,
    ↪ pricePerHour = "400")
if(test21[0] == 4):
    print(f"Test 21 pass: {test21[1]}")
    numberOfPasses += 1
else:
    print("Test 21 fail")
    numberOfFails += 1
```

Test 22: insert parking spot with wrong price per hour data type

```
# Test22:
#     Task: fail to generate a customer parking spot based on 6a being
#     ↪ wrong type
#     Expected result: 5 - failed to generate parking spot based on wrong
#     ↪ type
test22 = testCustomerParkingInsertion(dbConnection = connection,
    ↪ parkingSpotID = 999999997, parkingType = '{"1": "Car Parking"}' ,
    ↪ pricePerHour = "6a")
if(test22[0] == 5):
    print(f"Test 22 pass: {test22[1]}")
    numberOfPasses += 1
else:
    print("Test 22 fail")
    numberOfFails += 1
```

Test 23: insert parking spot with wrong price per hour data type and ID being the same

```
# Test23:
#     Task: fail to generate a customer parking spot based on 6a being
#     ↪ wrong type and same ID
#     Expected result: 6 - failed to generate parking spot based on wrong
#     ↪ type and same ID
test23 = testCustomerParkingInsertion(dbConnection = connection,
    ↪ parkingSpotID = 999999999, parkingType = '{"1": "Car Parking"}' ,
    ↪ pricePerHour = "6a")
if(test23[0] == 6):
    print(f"Test 23 pass: {test23[1]}")
    numberOfPasses += 1
else:
    print("Test 23 fail")
    numberOfFails += 1
```

At the end of the mainTest function, it informs us that it has finished and what are the results. It prints how many test cases were tested, what number of test cases passed and what number of test cases failed.

```
print("=====")
print("Unit testing has finished")
print(f"Number of tests executed: {numberOfPasses + numberOfFails}")
print(f"Number of passes: {numberOfPasses}")
print(f"Number of fails: {numberOfFails}")
print("=====")
```

We call the mainTest function at the bottom of our Python script.

```
#####
# Run unit tests 1 - 23
#####

mainTest()
```

If script is executed and everything works correctly, result shall be the same as in Figure 45. Green checkbox means that test passed, meaning that result of the test was as expected (for example: value 3 cannot be inserted due to constraints limits).

```
Unit testing database
Starting unit testing
Test 1 pass: Connection to mariadb-galera-0 was established
Test 2 pass: Connection to mariadb-galera-1 was established
Test 3 pass: Connection to mariadb-galera-2 was established
Test 4 pass: Connection to mariadb-galera-3 was established
Test 5 pass: Connection to unknown db was not established
Test 6 pass: Connection to mariadb-galera-0 with invalid ip (not localhost) was not possible
Test 7 pass: Connection to mariadb-galera-0 with invalid user was not possible
Test 8 pass: Connection to mariadb-galera-0 with invalid password was not possible
Test 9 pass: No employee was removed
Test 10 pass: No employee was not found
Test 11 pass: Employee successfully inserted
Test 12 pass: Failed to insert employee, there is already an employee with this ID
Test 13 pass: Failed to insert employee, one of the attributes is in wrong type Incorrect value: Incorrect date value: 'hireDate' for column 'Airport_DB`.`Employee`.`Hire_Date' at row 1
Test 14 pass: Failed to insert employee, one of the attributes is in wrong type and there is already an employee with this ID Incorrect value: Incorrect date value: 'hireDate' for column 'Airport_DB`.`Employee`.`Hire_Date' at row 1
Test 15 pass: Employee was removed
Test 16 pass: Customer parking spot successfully inserted
Test 17 pass: Failed to insert customer parking spot, there is already an customer parking spot with this ID
Test 18 pass: Failed to insert customer parking spot, one of the attributes is in wrong type, check constraint
Test 19 pass: Failed to insert customer parking spot, one of the attributes is in wrong type and there is already an customer parking spot with this ID, check constraint
Test 20 pass: Failed to insert customer parking spot, one of the attributes is in wrong type, check constraint
Test 21 pass: Failed to insert customer parking spot, one of the attributes is in wrong type and there is already an customer parking spot with this ID, check constraint
Test 22 pass: Failed to insert customer parking spot, one of the attributes is in wrong type
Test 23 pass: Failed to insert customer parking spot, one of the attributes is in wrong type and there is already an customer parking spot with this ID
Unit testing has finished
Number of tests executed: 23
Number of passes: 23
Number of fails: 0
```

Figure 45: Example of all test passes

Since the script doesn't remove customer parking spot, we can rerun the script and it will result in the Test 16 showcasing the error. Error becomes due to parking ID already be taken which is not expected for Test 16. Figure 46 shows red mark on failed test.

```
Unit testing database
Starting unit testing
Test 1 pass: Connection to mariadb-galera-0 was established
Test 2 pass: Connection to mariadb-galera-1 was established
Test 3 pass: Connection to mariadb-galera-2 was established
Test 4 pass: Connection to mariadb-galera-3 was established
Test 5 pass: Connection to unknown db was not established
Test 6 pass: Connection to mariadb-galera-0 with invalid ip (not localhost) was not possible
Test 7 pass: Connection to mariadb-galera-0 with invalid user was not possible
Test 8 pass: Connection to mariadb-galera-0 with invalid password was not possible
Test 9 pass: No employee was removed
Test 10 pass: No employee was not found
Test 11 pass: Employee successfully inserted
Test 12 pass: Failed to insert employee, there is already an employee with this ID
Test 13 pass: Failed to insert employee, one of the attributes is in wrong type Incorrect value: Incorrect date value: 'hireDate' for column 'Airport_DB`.`Employee`.`Hire_Date' at row 1
Test 14 pass: Failed to insert employee, one of the attributes is in wrong type and there is already an employee with this ID Incorrect value: Incorrect date value: 'hireDate' for column 'Airport_DB`.`Employee`.`Hire_Date' at row 1
Test 15 pass: Employee was removed
Test 16 fail
Test 17 pass: Failed to insert customer parking spot, there is already an customer parking spot with this ID
Test 18 pass: Failed to insert customer parking spot, one of the attributes is in wrong type, check constraint
Test 19 pass: Failed to insert customer parking spot, one of the attributes is in wrong type and there is already an customer parking spot with this ID, check constraint
Test 20 pass: Failed to insert customer parking spot, one of the attributes is in wrong type, check constraint
Test 21 pass: Failed to insert customer parking spot, one of the attributes is in wrong type and there is already an customer parking spot with this ID, check constraint
Test 22 pass: Failed to insert customer parking spot, one of the attributes is in wrong type
Test 23 pass: Failed to insert customer parking spot, one of the attributes is in wrong type and there is already an customer parking spot with this ID
Unit testing has finished
Number of tests executed: 23
Number of passes: 22
Number of fails: 1
```

Figure 46: Example of all test passes

Do note that images are showcasing emojis. This is due to simpler readability and is implement in the code. LaTeX is just not showcasing emojis, therefore they are removed from report (but it can be seen on GitHub (ZanZver [n.d.](#))).

8 Queries

This section showcases 3 individual queries and one optimisation technique for each team member.

8.1 Daniel

8.1.1 Query1

```
01 | Query1
```

8.1.2 Query2

```
01 | Query2
```

8.1.3 Query3

```
01 | Query3
```

8.1.4 Optimisation technique

```
01 | optimisation technique
```

8.2 Zan

8.2.1 Query1

Use case: Find a customer that has the oldest departure date and check amount that needs to be paid for parking.

To solve the problem, tables Ticket, Customer and Flight are joined. Based on that, we order departure time as ascending (oldest to newest) and limit set of results to 1. What is returned is an ID of the customer.

```
01 | SELECT Customer_ID_To_Find
02 | FROM
03 |     (
04 |         SELECT c.Customer_ID as Customer_ID_To_Find
05 |         FROM Ticket t
06 |         JOIN Customer c
07 |         ON t.Ticket_ID = c.Customer_ID
08 |         JOIN Flight f
09 |         ON t.Flight_FK = f.Flight_ID
10 |         ORDER BY f.Departure_Time ASC
11 |         LIMIT 1
12 |     ) as FoundCustomerID;
```

Listing 41: Query 1a - Zan

Figure 47 represents result of executed query in Label 41.

	Customer_ID	Token_ID	Date	Parking_Spot_ID	Parking_Type	Price_Per_Hour	ToPay
1	2,613	2,729	2023-01-03 10:37:04.000	2,729	{ "1": "Motor Parking" }	235	12,337.5

Figure 47: Query1a result - Zan

If result (ID 2613) is inserted into stored procedure, we receive information that amount due is £12K.

```
01 | CALL GetParkingToBePaid(2613);
```

Listing 42: Query 1b - Zan

Figure 48 represents result of executed query in Label 42.

	Customer_ID	Token_ID	Date	Parking_Spot_ID	Parking_Type	Price_Per_Hour	ToPay
1	2,613	2,729	2023-01-03 10:37:04.000	2,729	{ "1": "Motor Parking" }	235	12,455

Figure 48: Query1b result - Zan

8.2.2 Query2

Use case: Generate a report that shows how many people each person is supervising, what is their salary and their name. Results need to be ordered by number of people who are being supervised by employee and salary of that person (highest to lowest).

In order to solve the problem, we need to start by generating a nested query that will count number of people that are supervised. This is done with aggregated function COUNT (count all) and then results are grouped (with GROUP BY) based on Supervisor ID. This query is then returned as Supervisors table. To gather more information about the supervisor, we join Supervisors, Employees and EmployeeNames together. Supervisors will give us number of people that are supervised by certain employee, Employee table will return ID and salary of employee and view EmployeeNames will return first, last and middle name of employee. At the end, results are ordered by number of people that are supervised first and then by the salary.

```
01 | SELECT e.Employee_ID, Supervisors.Supervises, e.Salary, en.
    | First_Name, en.Last_Name, en.Middle_Name
02 | FROM
03 |     (
04 |         SELECT Supervisor, COUNT(*) as Supervises
05 |         FROM Employee
06 |         GROUP BY Supervisor
07 |     ) as Supervisors
08 | JOIN Employee e
09 | ON Supervisors.Supervisor = e.Employee_ID
10 | JOIN EmployeeNames en
```

```
11 | ON Supervisors.Supervisor = en.Person_ID
12 | ORDER BY Supervisors.Supervises DESC, e.Salary DESC;
```

Listing 43: Query 2 - Zan

Figure 49 represents result of executed query in Label 43. Do note that picture is cropped (there are more results that are not seen).

	Employee_ID	Supervises	Salary	First_Name	Last_Name	Middle_Name
1	25	13	21,330.56	Ronald	Li	Vanessa
2	185	9	55,645.04	Erin	Brewer	Latoya
3	1	9	46,004.4	Martha	Small	Susan
4	41	9	4,951.85	Valerie	Small	Stephen
5	1,041	8	94,090.5	Ashley	Jackson	Tommy
6	9	8	61,434.3	Darrell	Berry	Jennifer
7	5	8	52,415.58	Heidi	Johnson	Stephen
8	117	8	44,977.73	Jon	Gardner	David
9	33	7	69,483.27	Paul	Berger	[NULL]
10	173	7	53,184.09	Mary	Hancock	Sarah
11	45	7	33,132.84	Robert	Willis	David
12	673	7	22,507.12	Mark	Giles	Howard
13	37	7	18,176.14	Kevin	Garcia	[NULL]
14	737	7	3,724.77	James	Howell	Susan
15	101	6	94,568.14	Patricia	Vasquez	Karen
16	89	6	83,439.8	Jeffery	Valdez	Cory
17	17	6	79,892.37	Christopher	Baker	Amy

Figure 49: Query2 result - Zan

8.2.3 Query3

Use case: For every airline, gather top 3 most popular destinations and airlines monthly revenue. Airlines monthly revenue needs to be in descending order.

First thing that needs to be done in this scenario is connection between Flight and Airline. Once this is done, destination, company name and revenue can be returned. In order to get top 3 use cases from each airline, ROW_NUMBER() is used with PARTITION. Essentially what SQL does at this point is it splits the data based on the destination, company name and then orders which they are ordered descending (inside the partition). At the end, filtering is done to receive top 3 rows and revenue is ordered descending since partitions have been combined.

```
01 | SELECT Destination, Company_Name, Revenue
02 | FROM (
03 |     SELECT
04 |         f.Destination as Destination,
05 |         a.Company_Name as Company_Name,
06 |         a.Revenue as Revenue,
07 |         ROW_NUMBER() OVER (PARTITION BY Company_Name Order
08 |             by Revenue DESC) AS RowID
09 |     FROM Flight f
10 |     JOIN Airline a
11 |     ON f.Airline_FK = a.Company_ID
12 | ) RNK
12 | WHERE RowID <= 3
```



```
13 | ORDER BY Revenue DESC;
```

Listing 44: Query 3 - Zan

Figure 50 represents result of executed query in Label 44. Do note that picture is cropped (there are more results that are not seen).

	ABC Destination	ABC Company_Name	123 Revenue
1	South Connor	Thai AirAsia	96,686.24
2	Kennedychester	Thai AirAsia	96,686.24
3	Englishhaven	Thai AirAsia	96,686.24
4	Lake Natalieburgh	Chengdu Airlines	91,890.67
5	Lovechester	Chengdu Airlines	91,890.67
6	Brandonborough	Chengdu Airlines	91,890.67
7	Wilsonshire	Air Canada	91,382.97
8	Angelicabury	Air Canada	91,382.97
9	New Kayla	Air Canada	91,382.97
10	New Tyler	Batik Air	87,062.39
11	Freyton	Batik Air	87,062.39
12	New Samuelshire	Batik Air	87,062.39
13	East Jenniferberg	VietJet Air	83,928.47
14	Port Brandon	VietJet Air	83,928.47
15	South Stacey	VietJet Air	83,928.47
16	New Johnfort	Frontier Airlines	81,239.23
17	East Joshuafort	Frontier Airlines	81,239.23

Figure 50: Query3 result - Zan

8.2.4 Optimisation technique

Name table has two views in order to receive limited answer (only employee or customer names). But underneath the views, it is essentially just filtering based on where condition.

Code blocks bellow (Listings 46, 48) showcase that no matter if we search for customer (Is_Employee = 0) or employee (Is_Employee = 1) full table scan (looking at every single row) is made.

```
01 | EXPLAIN FORMAT=JSON SELECT * FROM Name n WHERE Is_Employee =
    0;
```

Listing 45: Unoptimised - receive all customers query

Code in Listing 45 was used to generate results bellow (Listing 46).

```
01 | {
02 |   "query_block": {
03 |     "select_id": 1,
04 |     "nested_loop": [
05 |       {
06 |         "table": {
07 |           "table_name": "n",
08 |           "access_type": "ALL",
09 |           "rows": 4500,
10 |           "filtered": 100,
11 |           "attached_condition": "n.Is_Employee = 0"
12 |         }
13 |       }
14 |     ]
15 |   }
16 | }
```

Listing 46: Unoptimised - Receive all customers result

```
01 | EXPLAIN FORMAT=JSON SELECT * FROM Name n WHERE Is_Employee =
    | 1;
```

Listing 47: Unoptimised - receive all employees

Code in Listing 47 was used to generate results bellow (Listing 48).

```
01 | {
02 |   "query_block": {
03 |     "select_id": 1,
04 |     "nested_loop": [
05 |       {
06 |         "table": {
07 |           "table_name": "n",
08 |           "access_type": "ALL",
09 |           "rows": 4500,
10 |           "filtered": 100,
11 |           "attached_condition": "n.Is_Employee = 1"
12 |         }
13 |       }
14 |     ]
15 |   }
16 | }
```

Listing 48: Unoptimised - Receive all employees result

Since scanning the whole table is not efficient, we can split (partition) our data based on desired attributes. In this case, split is done based on customers and employees. If value is less than 1 (essentially 0) it will be in Customer_Partition section, but if value is less than 2 (essentially 1), it will be in Employee_Partition.

The goal is for database to access only that partition, so full table scan can be avoided. In the DDL below (Listing 49), it can be seen how partitioning was append to the Name table.

```
01 | -- Person Name table
02 | CREATE TABLE `Name` (
03 |     `Person_ID` int(11) NOT NULL,
04 |     `First_Name` varchar(48) NOT NULL,
05 |     `Middle_Name` varchar(48) DEFAULT NULL,
06 |     `Last_Name` varchar(48) NOT NULL,
07 |     `Is_Employee` tinyint(1) NOT NULL DEFAULT 0 COMMENT 'Data
    needs to be 0 (default) if person is not employee or 1
    if they are employee',
08 |     PRIMARY KEY (`Person_ID`,`Is_Employee`),
09 |     CHECK(`Is_Employee`<=1 or `Is_Employee`>=0)
10 | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3
11 | PARTITION BY RANGE (Is_Employee)
12 | (
13 |     PARTITION Customer_Partition VALUES LESS THAN (1),
14 |     PARTITION Employee_Partition VALUES LESS THAN (2)
15 | );
```

Listing 49: Optimised - updated name DDL

In order to test the partitioning, same queries from above are executed. Searching for only customers at first (Listing 50), will result in only 2500 entries being filtered (instead of 4500) (Listing 51). It is also marked that Customer_Partition is used.

```
01 | EXPLAIN FORMAT=JSON SELECT * FROM Name n WHERE Is_Employee =
    0;
```

Listing 50: Optimised - receive all customers query

```
01 | {
02 |   "query_block": {
03 |     "select_id": 1,
04 |     "nested_loop": [
05 |       {
06 |         "table": {
07 |           "table_name": "n",
08 |           "partitions": ["Customer_Partition"],
09 |           "access_type": "ALL",
10 |           "rows": 2500,
11 |           "filtered": 100,
12 |           "attached_condition": "n.Is_Employee = 0"
13 |         }
14 |       }
15 |     ]
16 |   }
17 | }
```

Listing 51: Optimised - Receive all customers result

Doing the same search for the employee (Listing 52), showcases that only 2000 (instead of 4500) rows were filtered and that Employee_Partition was used (Listing 53).

```
01 | EXPLAIN FORMAT=JSON SELECT * FROM Name n WHERE Is_Employee =  
    1;
```

Listing 52: Optimised - receive all employees

```
01 | {  
02 |   "query_block": {  
03 |     "select_id": 1,  
04 |     "nested_loop": [  
05 |       {  
06 |         "table": {  
07 |           "table_name": "n",  
08 |           "partitions": ["Employee_Partition"],  
09 |           "access_type": "ALL",  
10 |           "rows": 2000,  
11 |           "filtered": 100,  
12 |           "attached_condition": "n.Is_Employee = 1"  
13 |         }  
14 |       }  
15 |     ]  
16 |   }  
17 | }
```

Listing 53: Optimised - Receive all employees result

At the end, we can see that partitioning worked as intended (Pina, Sá, and Bernardino 2023). Full table scan was mitigated by SQL reading only from one partition or the other. With optimisation techniques (partitioning, indexing, clustering, caching, etc...) we can drastically improve performance of the database (Mavro 2014). But there is a point of diminishing returns, meaning that SQL is doing more things to get data which deems it inefficient.

9 Conclusion

At the end of the report, we have successfully created example of working database for airport. System itself can be scaled, meaning that we have more than one databases and traffic could be routed to other databases in order to distribute the load.

Further development improvements were mentioned in the report (like use of AWS), but one big thing that would be needed is focus on security. For now, there is only one user that can access everything, but in production we would prefer to have multiple users set with least needed privileges.

Reflecting to team work, GitHub (more information available in appendix A12 (10.13)) was used in order to share the code and Overleaf was used to create this report. From management perspective, agile methodology was used, meaning that a lot of small changes were committed to Git instead of one big one. At the beginning, work was distributed evenly, but at the end it wasn't done as intended. List of completed tasks can be seen in appendix A9 (10.10).

10 Appendixes

10.1 A0

MAG airport makes money out of parking, cuts from shops and selling runway time slots. This led to the creation of first entity, **ParkingSpot**(ParkingSpotID, Type). The ID of ParkingSpot is its unique identifier that matches the ID in parking space. Type is describing what kind of parking spot we have in mind. For example, it can be for cars, motorcycles, handicapped, etc... Since Type can have multiple attributes (such as parking for cars and handicapped), it is presented as a multivalued attribute. **ParkingSpot** itself is a super type with two subtypes, **EmployeeParkingSpot** and **CustomerParkingSpot**. **EmployeeParkingSpot** is the same as **ParkingSpot**, while **CustomerParkingSpot** includes PricePerHour attribute. Describing subtypes, we get **EmployeeParkingSpot** (ParkingSpotID, Type) and **CustomerParkingSpot** (ParkingSpotID, Type, PricePerHour).

Parking can be done by two actors, employees or customers. In other words, the parking space is divided into parking spots for customers and parking spots for employees. To represent this, full completeness with disjoint can be observed in the ER diagram below.

If an employee has parked a car, this would be considered as an employee using a parking spot. An employee may park one (or none) car on one parking spot. This led to employee – parking relation being **one to one** with EmployeeParkingSpot being set as optional.

As with many companies, MAGs biggest expense is on employees. This led to the creation of our second entity, **Employee**(EmployeeID, HireDate, TerminationDate, Salary, EmploymentType, Title, BirthDate, Age, FirstName, LastName, MiddleName, AddressLine1, AddressLine2, PostCode, City, Supervisor). The unique identifier of Employee is EmployeeID. The entity **Employee** is in 2nd normal form, but if we would like to break it down further, we can take **Name**(FirstName, LastName, MiddleName) out. Age is represented as a derived attribute, meaning it can be calculated from BirthDate (therefore we are not directly storing it). The Supervisor attribute is recursively represented, meaning that an employee can be a supervisor for other employees.

All our employees have different skills that are listed under **Qualification**(QualificationId, Type, Name). The idea is for employees to have 0 or N number of qualifications (due to some professions not requiring qualifications). This is represented in ERD with an intermediate table. We will have a list of individual certificates which are unique for each employee and proves their qualifications, meaning **1 to M** (all employees are in the intermediate table) and **M to 1** (employees can have 0 or N qualifications).

Entity **Certificate**(CertificateID, Level, CertificateName) would contain a list of all the certificates that the company has on record.

Navigating back to the **Employee** entity, we are also connecting it to the **Department** entity. Each employee is working in one department and one department

only, while department has many employees. This would make the relation as 1 (1 department per employee) to M (many employees per department). Each department also has one manager that is selected from the Employee table. Not all employees need to be managers, so it is optional. Relationship is presented as one to one.

Looking deeper into **Department**(DepartmentID, DepartmentName, Location), we can see that it is build from 3 attributes and it is in 3rd normal form. Location is represented as where can it be found at airport (example: room 42). If location would be a combination of floor and room, we could break it apart and store it as separate Location entity.

Entity **Department** is connected to the **CompanyVehicle**(VehicleID, Name, DrivingLicenseRequirement) entity as well. Department can have many company vehicles or none, while company vehicle can belong to one and only one department. This is represented as M (number of vehicles per department) to 1 (one vehicle being assigned to 1 department).

Each **Employee** may be assigned a **CompanyVehicle** 1 vehicle maximum and 0 minimum (since they might not need it). This is represented as one to one relationship, while both parties are optional (a company vehicle might not be assigned to any employee).

CompanyVehicle is also connected to **EmployeeParkingSpot**. Each vehicle is parked at one employee parking spot (if vehicle is not in use). Each parking spot is assigned to one company vehicle. This is represented as one to one relationship.

Navigating back to **CustomerParkingSpot**, it is connected to **ParkingToken**. The connection is one to one but parking token is optional. One parking space has one parking token (if parking space is occupied), while parking token is connected to parking spot.

Entity **ParkingToken**(TokenID, Date). Goal of parking token is to store tokens and time when it was occupied. With this, business can calculate when the parking lot is mostly occupied, when it is empty, etc... This would allow MAG to adjust prices based on vehicles being parked.

ParkingTokens are bought by **Customers**. Each customer can have one parking token (one parking space) or none, while parking token can be allocated to one and only one customer. This is represented as one to one relationship with **ParkingToken** being optional.

In order to be profitable (which is the goal of MAG), customers are needed. This is why the **Customer**(Customer ID, ExpressLane, FirstName, LastName, MiddleName) entity is created. The entity itself is in 3rd normal form, we can split name into **Name**(FirstName, LastName, MiddleName). Doing that, we can have **Name** entity with **Customer** and **Employee** names, locating them by ID. This will later on be shared as employees and customers.

Customer is also connected to **Ticket**. One customer can have 0 tickets, in case they are just parking at the airport, they are our customer but without a ticket. The Ticket itself can be assigned to one and only one customer. This is

represented as **M** (customer having multiple tickets or none) to **1** (one ticket per customer).

Since airlines do share limited information with the airport, entity **Ticket**(**TicketID**) has only one attribute. **Ticket** and **Flight** relationship is **M** (flight having multiple or none tickets) to **1** (one ticket being valid on one flight). Entity **Ticket** can also be optional, depending what flight it is (private, cargo, test, etc...).

Entity **Flight**(**FlightID**, **Destination**, **Gate**, **PlaneModel**, **DepartureTime**). Flight is in 3rd normal form, but there are options for improvement. Gate, plane model and departure time can be its own attribute with additional information. This can be changed in future revisions.

Flight is also connected to **Runway**. The connection is **one to one**, meaning that only one flight can be scheduled at the runway and runway can have only one flight at a time.

As mentioned at the start, MAG is selling runway slots, therefore an entity is added as **Runway**(**RunwayID**, **Length**, **Width**). We are not saving time slots (when runway is occupied or not) since this data can be calculated from the **Flight** entity.

Flight has another connection to the **Airline** attribute. Connection is **M** (airline can have multiple flights) to **1** (flight has only one airline).

Entity **Airline**(**CompanyID**, **CompanyName**, **Revenue**) is representing all of the airlines our airport would have. It is in 3rd normal form. This information is needed since airports get paid from airlines as well. If airline would perform poorly at our airport, we can remove it and allocate runway slots to a new airline that would generate more money.

10.2 A1

To create the database, Docker (Docker [n.d.\[a\]](#)) is used as virtualisation platform to create docker container(s) with database(s) inside. This allows for isolation of environments and simpler replication.

The database management system used is MariaDB with an Innodb engine. MariaDB was selected since it is open source (free to use) (Mariadb [n.d.\[c\]](#)). Innodb engine is the default engine (Mariadb [n.d.\[b\]](#)) with MariaDB and is known to be reliable (Mariadb [n.d.\[a\]](#)). If the database would be dealing with big data, it would be preferred to explore Spider Storage Engine (Mariadb [n.d.\[f\]](#)) from MariaDB since its purpose is to handle this big data. Another competitor would be S3 Storage Engine (Mariadb [n.d.\[e\]](#)). This would be great if we would use S3 compatible storage (example: AWS S3 buckets) to store the data and this engine to retrieve its data.

To achieve high availability, multiple databases are needed with an option to communicate with each other. MariaDB Galera Cluster (Mariadb [n.d.\[g\]](#)) allows us to solve this problem. This is multi-primary cluster that synchronises all of the databases. VMware (Bitnami [n.d.](#)) has created a Docker image that is used in this project.

As an example, 4 Galera nodes are created and clustered together. They have an IP of localhost and ports range from 30330-30333 (being forwarded to 3306 in Docker). As soon as one operation takes place and transaction lock is released, writes are synced across the nodes. Figure 51 is showcasing proposed design for this project.

Based on the explanation above, here is how implementation could look like for MAG group (based on this experiment): Since MAG combines different airports (Manchester, London Stansted and East Midlands) (MAG [n.d.](#)), there could be one docker container per airport location. This would mean that Manchester database can be replicated to two other locations as soon as local transactions are complete (and vice versa).

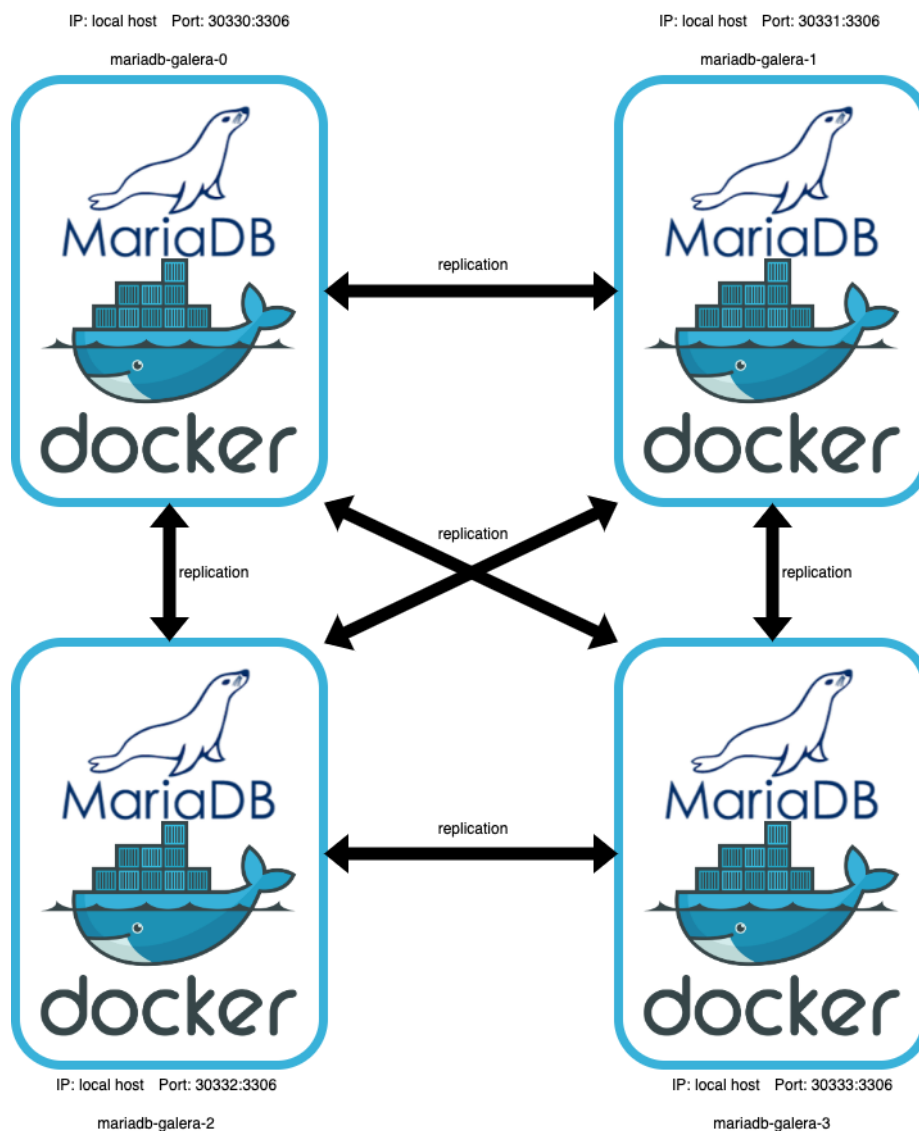


Figure 51: Physical design

To deploy the infrastructure Docker Compose (Docker [n.d.\[b\]](#)) was used. This is an extension of Docker that deploys predefined structures based on what is in the given YAML file.

For Docker containers to communicate with each other, they need to be on the same network. The code below (Listing 54) shows how the custom intranet is created ("galera-cluster-net") so the Docker Containers can communicate with each other.

```
01 | docker network create -d bridge galera-cluster-net
```

Listing 54: Docker Network

Next, we need to create the Docker Compose script (seen in the Docker Compose database creation). The network is passed in alongside with services (databases). Databases mariadb-galera-1 (MG1), mariadb-galera-2 (MG2) and mariadb-galera-3 (MG3) have the same structure while mariadb-galera-0 (MG0) is unique. This is due to MG0 acting as a primary only when system is starting. At first, MG0 is setup, then MG1, MG2 and MG3 follow afterwards. Once done, the nodes can communicate with each other and are in sync.

```

_____ Docker Compose database creation _____
1  version: '2.1'
2
3  name: MariadbCluster
4
5  networks:
6    app-tier:
7      driver: galera-cluster-net
8
9  services:
10   mariadb-galera-0:
11     container_name: mariadb-galera-0
12     image: 'bitnami/mariadb-galera:latest'
13     environment:
14       - MARIADB_GALERA_CLUSTER_BOOTSTRAP=yes
15       - MARIADB_GALERA_CLUSTER_NAME=my_galera
16       - MARIADB_GALERA_MARIABACKUP_USER=my_mariabackup_user
17       - MARIADB_GALERA_MARIABACKUP_PASSWORD=my_mariabackup_password
18       - MARIADB_ROOT_PASSWORD=my_root_password
19       - MARIADB_USER=my_user
20       - MARIADB_PASSWORD=my_password
21       - MARIADB_DATABASE=Airport_DB
22     ports:
23       - "30330:3306"
24     healthcheck:
25       test: ["CMD-SHELL", "mysqladmin ping"]
26       interval: 30s
```

```
27     timeout: 30s
28     retries: 3
29
30 mariadb-galera-1:
31     container_name: mariadb-galera-1
32     image: 'bitnami/mariadb-galera:latest'
33     links:
34         - 'mariadb-galera-0:mariadb-galera'
35     environment:
36         - MARIADB_GALERA_CLUSTER_NAME=my_galera
37         - MARIADB_GALERA_CLUSTER_ADDRESS=gcomm://mariadb-galera
38         - MARIADB_GALERA_MARIABACKUP_USER=my_mariabackup_user
39         - MARIADB_GALERA_MARIABACKUP_PASSWORD=my_mariabackup_password
40     ports:
41         - "30331:3306"
42     depends_on:
43         mariadb-galera-0:
44             condition: service_healthy
45
46 mariadb-galera-2:
47     container_name: mariadb-galera-2
48     image: 'bitnami/mariadb-galera:latest'
49     links:
50         - 'mariadb-galera-0:mariadb-galera'
51     environment:
52         - MARIADB_GALERA_CLUSTER_NAME=my_galera
53         - MARIADB_GALERA_CLUSTER_ADDRESS=gcomm://mariadb-galera
54         - MARIADB_GALERA_MARIABACKUP_USER=my_mariabackup_user
55         - MARIADB_GALERA_MARIABACKUP_PASSWORD=my_mariabackup_password
56     ports:
57         - "30332:3306"
58     depends_on:
59         mariadb-galera-0:
60             condition: service_healthy
61
62 mariadb-galera-3:
63     container_name: mariadb-galera-3
64     image: 'bitnami/mariadb-galera:latest'
65     links:
66         - 'mariadb-galera-0:mariadb-galera'
67     environment:
68         - MARIADB_GALERA_CLUSTER_NAME=my_galera
69         - MARIADB_GALERA_CLUSTER_ADDRESS=gcomm://mariadb-galera
```

```

70     - MARIADB_GALERA_MARIABACKUP_USER=my_mariabackup_user
71     - MARIADB_GALERA_MARIABACKUP_PASSWORD=my_mariabackup_password
72     ports:
73     - "30333:3306"
74     depends_on:
75     mariadb-galera-0:
76         condition: service_healthy

```

To start the docker compose, the code below (Listing 55) needs to be executed.

```
01 | docker-compose up -d
```

Listing 55: Docker Compose start

Once everything is done, all 4 of the containers can be seen (in terminal or in Docker desktop). Figure 52 represents all available containers as a cluster.





























<input type="checkbox"/>	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	 mariadbcluster	-	Running (4/4)			  
<input type="checkbox"/>	 mariadb-galera-0 24d8ed7eca5b 	bitnami/mariadb-galera:10.9	Running	30330:3306 	47 seconds ag	  
<input type="checkbox"/>	 mariadb-galera-1 895f01cbcd5 	bitnami/mariadb-galera:10.9	Running	30331:3306 	16 seconds ag	  
<input type="checkbox"/>	 mariadb-galera-3 acff1caef155 	bitnami/mariadb-galera:10.9	Running	30333:3306 	16 seconds ag	  
<input type="checkbox"/>	 mariadb-galera-2 4b86d02f999 	bitnami/mariadb-galera:10.9	Running	30332:3306 	16 seconds ag	  

Figure 52: Docker showcase

10.3 A2

A Python script was written to automate dummy data generation. This way, a lot of data can be generated at once because the INSERT INTO statements are not written manually anymore. This also allows us to specify how much data we want in the database.

At the beginning of the script, libraries are imported. The Faker library is used to generate data, random is used to generate random numbers and "mysql.connector" is used to connect to the database.

```

print("=====")
print("Generating fake data")
print("=====")

#=====
# Import external libraries
#=====
from faker import Faker

```

```
from faker_airtravel import AirTravelProvider
from faker_vehicle import VehicleProvider
#pip install faker
#pip install faker_airtravel
#pip install faker_vehicle

import random
import string
import mysql.connector
from mysql.connector import Error
```

In next section, we specify how much data we want for each table. Based on this, the program knows how many things to generate (example: only 5 employees). Department names and employment types have been manually created since nothing similar could be found in any of the Python Faker providers.

```
#####
# Define data
#####
no_runways = 10
no_airlines = 50
no_flights= 80
no_qualifs = 30
no_tickets = 80
no_customers = 2500
no_employees = 2000
no_customer_parking_spots= 900
no_employee_parking_spots= 70
no_tokens = 1500
no_tickets = 2000
no_vehicles = 80
no_certs = 50
driving_cats= ["A","B","C","D", "H","M"]
parking_types = ["Car Parking", "Motor Parking", "Handicap Parking"]
department_names = ["Landside operations", "Airside operations", "Billing and
↪ invoicing", "Information management"]
employment_types = ["Full-time", "Part-time", "Casual", "Fixed term",
↪ "Contract", "Apprentice", "Trainee", "Commission", "Piece rate"]
```

Now we initialize faker and attach extra providers to it (air travel and vehicle). These providers allow us to generate more specialised data. Based on that, fake data can be generated.

```
#####
# Initialize faker
#####

fake = Faker()
fake.add_provider(AirTravelProvider)
fake.add_provider(VehicleProvider)
```

In order to connect to the database, the `create_server_connection` function is defined. The function accepts four arguments (host name, port, user name, user password) in order for the connection to be formed. We try to establish a Connection, if it fails it prints out an error. Otherwise, it confirms that the connection was made. At the end, the function returns the connection (if it was successful) or None if the connection process failed.

```
#####  
# SQL Functions  
#####  
  
def create_server_connection(host_name, port, user_name, user_password):  
    connection = None  
    try:  
        connection = mysql.connector.connect(  
            host=host_name,  
            user=user_name,  
            passwd=user_password,  
            port= port  
        )  
        print("MySQL Database connection successful")  
    except Error as err:  
        print(f"Error: '{err}'")  
  
    return connection
```

Function `execute_query` runs a given query on a specified connection. Based on connection, cursor is generated and tries to execute the given query. If the query cannot be executed, the error is printed with the query that failed as well.

```
def execute_query(connection, query):  
    cursor = connection.cursor()  
    try:  
        cursor.execute(query)  
        connection.commit()  
    except Error as err:  
        print(f"Error: '{err}'")  
        print(f"Query: '{query}'")
```

The `get_ids` function accepts connection and query (that shall be ID related) as inputs. Based on connection, it makes sure "Airport_DB" is in use and then it generates the cursor. Afterwards, the cursor tries to execute the query and it fetches the response from it. Then, only the first attribute is returned if everything worked successfully, otherwise an error is printed with the query that failed as well.

```
def get_ids(connection, IDquery):  
    execute_query(connection, "USE Airport_DB;")  
    cursor = connection.cursor()  
    try:  
        cursor.execute(IDquery)
```

```

        table = cursor.fetchall()
        #print("Query successful")
        return([x[0] for x in table])
    except Error as err:
        print(f"Error: '{err}'")
        print(f"Query: '{IDquery}'")

```

To generate supervisor for the employee, a list of employees is needed at first. Function `generate_supervisor` receives connection to the database as an input, then it generates a query to select employee IDs from the employee table. Afterwards, the query is executed with the help of `get_ids` function. If the returned result has more than 0 employee IDs, then one of the IDs is picked at random and returned. Otherwise 'null' is returned as there are no employees in the database.

```

def generate_supervisor(dbConnection):
    employee_query = "SELECT Employee_ID FROM Employee;"
    employee_ids = get_ids(dbConnection, employee_query)
    if(len(employee_ids) != 0):
        return random.choice(employee_ids)
    else:
        return 'null'

```

Function `get_data` receives entry as an input. Entry is a dictionary type, which will be split into two tuples, one that contains the names of the columns and one that contains the values for those columns. This transformation is done because tuples, when printed as strings have round braces, which happens to be the exact syntax SQL uses.

```

def get_data(entry):
    output = (str(tuple(entry.keys())).replace("'", ""),
    ↪ str(tuple(entry.values())))
    return output if output[0][-2] != "," else
    ↪ (output[0][:-2]+output[0][-1], output[1][:-2]+output[1][-1])

```

To insert the data into the database, the `populate` function is called. It accepts three parameters (connection, list of entities and target table). At the start, the function iterates through the list and calls `get_data` function in order to retrieve columns and values (based on every item in the list). Next it generates a query for every item which is then executed with `execute_query` function.

```

#puts the dummy data into the database
def populate(connection, lst, target):
    for entry in lst:
        columns, values= get_data(entry)
        query = ("INSERT INTO " + target+ " " + columns+ " VALUES " +
        ↪ values).replace("'null'", "null")
        execute_query(connection, query)
    #print("Populated " + target + "\n")

```

In order to generate data for every table, 15 functions were created. With this approach, each table can be filled independently. If there is only one table that data needs to be generated for, only one function can be called. As seen on the ERD, there are 14 tables, but here are 15 functions. The reason for this is that Name table has two functions, one for employee names and one for customer names. The reasoning behind it is the same as above, the ability to fill each table independently. If we only want to generate one type of names, this is now possible. All of the functions work in a similar way, they accept one argument (database connection and return none. After they have been executed, it is printed that execution of the function is complete.

The first function generates data for the airline table. It creates a list of dictionaries with fake company name and revenue that is between £10000, £100000. Afterwards we make sure that "Airport_DB" is used and then populate function is called in order to insert the data (connection, dictionary of the airlines and argument "Airline" are passed as arguments).

```
#####  
# Dummy data generation for every table  
#####  
def generate_airlines(dbConnection):  
    airlines = [  
        {"Company_Name":fake.airline(),  
         "Revenue":random.uniform(10000,100000)  
        } for x in range(no_airlines)]  
  
    execute_query(dbConnection, "USE Airport_DB;")  
    populate(dbConnection, airlines, "Airline")  
  
    print("=====  
    print("Airline table finished")  
    print("=====
```

Function generate_certificates starts with building two queries that receive IDs (from Employee and Qualification table). Afterwards the queries are executed with the get_IDs function. Once done, a list of dictionaries is built with Employee_FK being a random employee ID (from the query above), Certificate_Name and Achievement_Level are random sentences, and Qualification_FK is random qualification ID (from the query above). Once dictionary is done, script makes sure that it uses "Airport_DB" and then sends connection, dictionary and "Certificate" to populate function in order for data to be inserted.

```
def generate_certificates(dbConnection):  
    employee_query = "SELECT Employee_ID FROM Employee;"  
    qualification_query = "SELECT Qualification_ID FROM Qualification;"  
    employee_ids = get_ids(dbConnection, employee_query)  
    qualification_ids = get_ids(dbConnection, qualification_query)  
  
    certificates = [{
```



```

    "Employee_FK":random.choice(employee_ids),
    "Certificate_Name":fake.sentence(5),
    "Achievement_Level":fake.sentence(3),
    "Qualification_FK": random.choice(qualification_ids)
}for x in range(no_certs)]

execute_query(dbConnection, "USE Airport_DB;")
populate(dbConnection, certificates, "Certificate")

print("=====")
print("Certificates table finished")
print("=====")

```

To generate company vehicles, this function first generates a query that will receive all of the department IDs. Then query is executed with the help of `get_ids` function. Next, a list of dictionaries is built that represents `Vehicle_Name` (a fake name), `Vehicle_Driving_License_Requirement` (one of the categories from above) and `Department_FK` (one random department). Once done, the function makes sure "Airport_DB" is in use and the `populate` function is called (with the generated parameters).

```

def generate_company_vehicles(dbConnection):
    department_query = "SELECT Department_ID FROM Department;"
    department_ids = get_ids(dbConnection, department_query)
    vehicles = [{
        "Vehicle_Name":fake.machine_year_make_model(),
        "Vehicle_Driving_License_Requirement":random.choice(driving_cats),
        "Department_FK":random.choice(department_ids)
    } for x in range(no_vehicles)
    ]
    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection,vehicles,"CompanyVehicle")

    print("=====")
    print("CompanyVehicle table finished")
    print("=====")

```

In order to generate a customer, we firstly generate and execute a query that will return parking spot IDs available to customers. Next we build list of dictionaries with `Express_Lane` (it can be 0 or 1 at random) and `Token_FK` (randomly allocated parking token). Afterwards the `populate` function is called to insert the data.

```

def generate_customers(dbConnection):
    parking_spot_query = "SELECT Parking_Spot_ID FROM CustomerParkingSpot;"
    parking_spot_ids = get_ids(dbConnection, parking_spot_query)
    customers = [{
        "Express_Lane":0 if (random.randrange(1,10)<9) else 1, #90% chance to
        ↪ not have express lane
        "Token_FK":random.choice(parking_spot_ids)
    }

```

```

    } for x in range(no_customers)]

execute_query(dbConnection, "USE Airport_DB;")
populate(dbConnection, customers, "Customer")

print("=====")
print("Customer table finished")
print("=====")

```

The generate_customer_parking_spots function starts of by generating a list of parking types at random (example: car parking and handicapped) for every parking spot that will be available (to the customer). This list contains strings in json format. Next step, list of dictionaries is created with Parking_Type (inserted from the list above) and Price_Per_Hour (randomly assigned between 5,300). Once done, the data is inserted into the CustomerParkingSpot table.

```

def generate_customer_parking_spots(dbConnection):
    stri=[]
    for x in range(no_customer_parking_spots):
        stri.append("{}")
        parking_spot = random.sample(parking_types,random.randrange(1,4))
        for y in range(len(parking_spot)):
            stri[x]+= "'"+str(y+1)+"': '"+parking_spot[y]+'", '
        stri[x] = stri[x][::-2]+"}"

    customer_parking_spots =[
        {"Parking_Type": stri[x],
        "Price_Per_Hour":random.randrange(5,300)
        }for x in range(no_customer_parking_spots)]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection,customer_parking_spots , "CustomerParkingSpot")

    print("=====")
    print("CustomerParkingSpot table finished")
    print("=====")

```

To generate departments, this function creates a list of dictionaries with department_names (one of the names from the list above) and Department_Location (random location). Once done, it is inserted into the "Department" table.

```

def generate_departments(dbConnection):
    departments =[
        {"Department_Name":department_names[x],
        "Department_Location":fake.street_name()
        } for x in range(len(department_names))]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection, departments, "Department")

    print("=====")

```

```
print("Department table finished")
print("=====")
```

The Employee table has 16 attributes (+ ID of employee). Therefore, the function to generate them is a bit more complicated. It starts by declaring three queries that return IDs (from EmployeeParkingSpot, CompanyVehicle and Department table). Afterwards queries are executed and returned with getIDs function. Once done, a list of dictionaries with employee data is generated. As seen, most of the data is generated by Faker, except Supervisor (which is random employee ID), Parking_Spot_FK (random ID from parking spot), Vehicle_FK (random vehicle ID), Department_FK (random department ID) and Manage_Department (random department ID). Since Supervisor is recursive, inserting employees cannot be done once. This means that every single employee (aka their dictionary) needs to be inserted into the database individually. First employee will always have 'null' (aka nobody) as a supervisor, since they are the only one present in the table. Then once the second employee is in the DB, they will have employee 1 as supervisor. Third employee has a 50% chance of having employee 1 or 2 as a supervisor. As more employees are generated, it becomes more likely for employees who were inserted earlier to be set as supervisors.

```
def generate_employees(dbConnection):
    employee_parking_spot_query = "SELECT Parking_Spot_ID FROM
    ↪ EmployeeParkingSpot;"
    company_vehicle_query = "SELECT Vehicle_ID FROM CompanyVehicle;"
    department_query = "SELECT Department_ID FROM Department;"
    employee_parking_spot_ids = get_ids(dbConnection,
    ↪ employee_parking_spot_query)
    company_vehicle_ids = get_ids(dbConnection, company_vehicle_query)
    department_ids = get_ids(dbConnection, department_query)

    for x in range(no_employees):
        employees =[{
            "Hire_Date":str(fake.date_this_decade()),
            "Termination_Date":str(fake.date_this_decade()) if
            ↪ (random.randrange(1,10)<2) else "null", #20% turnover,
            "Title": fake.job(),
            "Employment_Type":random.choice(employment_types),
            "Salary":random.uniform(3000,100000),
            "Supervisor":generateSupervisor(dbConnection),
            "Address_Line_1":fake.street_name(),
            "Address_Line_2":fake.building_number(),
            "Postcode": fake.postcode(),
            "City": fake.city(),
            "Birth_Date":str(fake.date_this_century()),
            "Parking_Spot_FK":random.choice(employee_parking_spot_ids),
            "Vehicle_FK":random.choice(company_vehicle_ids),
            "Department_FK":random.choice(department_ids),
            "Manage_Department":random.choice(department_ids)
        }]
```

```
execute_query(dbConnection, "USE Airport_DB;")
populate(dbConnection,employees,"Employee")

print("=====")
print("Employee table finished")
print("=====")
```

The generate_employee_parking function works the same as generate_customer_parking_spots. The only difference is that employees are not charged for parking, therefore attribute "Price_Per_Hour" is not present in this function.

```
def generate_employee_parking(dbConnection):
    strii=[]
    for x in range(no_employee_parking_spots):
        strii.append("{}")
        parking_spot = random.sample(parking_types,random.randrange(1,4))
        for y in range(len(parking_spot)):
            strii[x]+= "'"+str(y+1)+"':"'+parking_spot[y]+'", '
        strii[x] = strii[x][:-2]+"}"

    employee_parking_spots =[
        {"Parking_Type": strii[x]
        }for x in range(no_employee_parking_spots)]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection,employee_parking_spots ,"EmployeeParkingSpot")

    print("=====")
    print("EmployeeParking table finished")
    print("=====")
```

In order to generate the flight, two queries are defined that receive IDs (from Runway and Airline table). Afterwards, queries are executed with the help of getIDs function. Once done a list of dictionaries is created with Plane_Model (fake model name), Departure_Time (fake departure time), Destination (fake city), Gate (random numbers between 10 and 30), Runway_FK (random ID generated above) and Airline_FK (random ID generated above) attributes is build. Once done data is inserted.

```
def generate_flight(dbConnection):
    runwayQuery = "SELECT Runway_ID FROM Runway;"
    airlineQuery = "SELECT Company_ID FROM Airline;"
    runwayIDs = get_ids(dbConnection, runwayQuery)
    airlineIDs = get_ids(dbConnection, airlineQuery)

    flights = [{
        "Plane_Model":''.join(random.choice(string.ascii_uppercase) for _ in
        ↪ range(5)),
        "Departure_Time":str(fake.date_time_this_decade()),
```

```
"Destination":fake.city(),
"Gate":random.randrange(10,30),
"Runway_FK":random.choice(runwayIDs),
"Airline_FK":random.choice(airlineIDs),
} for x in range(no_flights)]

execute_query(dbConnection, "USE Airport_DB;")
populate(dbConnection, flights, "Flight")
```

Parking tokens are generated only for customers. Firstly, the script gets all of the IDs from the CustomerParkingSpot table. Afterwards, a list of dictionaries with "Date" (random date) and Parking_Spot_FK (random parking spot ID) attributes is created. Once completed, the data is inserted into the ParkingToken table.

```
def generate_parking_tokens(dbConnection):
    parkingSpotQuery = "SELECT Parking_Spot_ID FROM CustomerParkingSpot;"
    parkingSpotIDs = get_ids(dbConnection, parkingSpotQuery)
    parking_tokens = [{
        "Date":str(fake.date_time_this_month()),
        "Parking_Spot_FK":random.choice(parkingSpotIDs),
    } for x in range(no_tokens)]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection, parking_tokens, "ParkingToken")

    print("=====")
    print("ParkingToken table finished")
    print("=====")
```

Qualification function starts by creating a dictionary with two attributes (Qualification_Type and Qualification_Name). Both attributes are random sentences from Faker (so they might not make much sense. But as far as dummy data is concerned, this is a good placeholder). A list of dictionaries is then inserted into the database.

```
def generate_qualification(dbConnection):
    qualifications = [{
        "Qualification_Type":fake.sentence(5),
        "Qualification_Name":fake.sentence(5)
    } for x in range(no_qualifs)]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection, qualifications, "Qualification")

    print("=====")
    print("Qualification table finished")
    print("=====")
```

For runways to be generated, function first creates a list of dictionaries with two attributes (Length and Width). Both have been given specific range in order

to pass constraints. Afterwards, data is inserted into Runway table. The length takes a random value between 2000 and 4000 and the width takes a random value between 8 and 80.

```
def generate_runway(dbConnection):
    runways = [{
        "Length":random.uniform(2000,4000),
        "Width":random.uniform(8,80)
    } for x in range(no_runways)]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection, runways, "Runway")

    print("=====")
    print("Runway table finished")
    print("=====")
```

The Ticket generation function starts with generating two queries that return IDs (one from Customer, one from Flight). Once done, queries are executed with getIDs function and IDs are returned. Afterwards, a list of dictionaries is built with Customer_FK and Flight_FK being random variables based on return IDs. Once the list of dictionaries is built, it is passed to the populate function to insert the data.

```
def generate_ticket(dbConnection):
    customer_query = "SELECT Customer_ID FROM Customer;"
    flight_query = "SELECT Flight_ID FROM Flight;"
    customer_ids = get_ids(dbConnection, customer_query)
    flight_ids = get_ids(dbConnection, flight_query)
    tickets = [{
        "Customer_FK":random.choice(customer_ids),
        "Flight_FK":random.choice(flight_ids)
    } for x in range(no_tickets)]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(dbConnection, tickets, "Ticket")

    print("=====")
    print("Ticket table finished")
    print("=====")
```

To generate employee names, the function first receives IDs from the Employee table. Then a list of dictionaries is built with Person_ID (ID of employee), First_Name (random first name), Middle_Name (random middle name or null), Last_Name (random last name) and Is_Employee (set as 1 since this is employee). Once the list is built, it gets inserted into the Name table.

```
def generate_employee_names(dbConnection):
    employee_query = "SELECT Employee_ID FROM Employee;"
    employee_ids = get_ids(dbConnection, employee_query)
```

```
names = [{
    "Person_ID":employee_ids[x],
    "First_Name":fake.first_name(),
    "Middle_Name":fake.first_name() if (random.randrange(1,10)<8) else
    ↪ "null", #some people do not have a middle name. chance: 20%
    "Last_Name":fake.last_name(),
    "Is_Employee": 1
}for x in range(len(employee_ids))]

execute_query(dbConnection, "USE Airport_DB;")
populate(connection, names, "Name")

print("=====")
print("Name table - employees finished")
print("=====")
```

The generate_customer_names function works similar to the generate_employee_names function. The difference is at the start; it gets IDs from the Customer table. Another difference it has is that Is_Employee is set to 0 since we are inserting customers and not employees. Other than that, both functions work the same.

```
def generate_customer_names(dbConnection):
    customer_query = "SELECT Customer_ID FROM Customer;"
    customer_ids = get_ids(dbConnection, customer_query)

    names = [{
        "Person_ID":customer_ids[x],
        "First_Name":fake.first_name(),
        "Middle_Name":fake.first_name() if (random.randrange(1,10)<8) else
        ↪ "null", #some people do not have a middle name. chance: 20%
        "Last_Name":fake.last_name(),
        "Is_Employee": 0
    }for x in range(len(customer_ids))]

    execute_query(dbConnection, "USE Airport_DB;")
    populate(connection, names, "Name")

    print("=====")
    print("Name table - customers finished")
    print("=====")
```

At the bottom of the script, the functions are called to insert dummy data. At the start, the connection is made to the server. After that each table has its own function that generates fake data. This helps if the script needs to run again to increase number of data in one specific table.

```
#####
# Insert dummy data
#####
```

```
connection = create_server_connection("localhost", "30330", "my_user",  
    ↪ "my_password")  
  
generate_runway(connection)  
generate_airlines(connection)  
generate_flight(connection)  
generate_qualification(connection)  
generate_customer_parking_spots(connection)  
generate_employee_parking(connection)  
generate_parking_tokens(connection)  
generate_customers(connection)  
generate_ticket(connection)  
generate_departments(connection)  
generate_company_vehicles(connection)  
generate_employees(connection)  
generate_certificates(connection)  
generate_employee_names(connection)  
generate_customer_names(connection)
```

10.4 A3

10.4.1 A3.1

The Airline table features constraints in terms of data types. Company ID can have 11 numerical values as a maximum (99 billion would be max), ID is auto incremented upon insertion to database and cannot be null. The company name cannot be more than 64 characters. Revenue can only have 15 digits, meaning their monthly profit is not expected to exceed 999 trillion (this could be lower in production). Decimal places are limited to 2 since we are dealing with currency. There is also a check in place for revenue to be more than 10000 per month. Code below (Listing 1) is presenting DDL of the Airline table.

10.4.2 A3.2

The Certificate table (Listing 2) has similar character limitations as Airline table. Certificate name is not expected to be longer than 64 characters while Achievement level is capped at 32 characters. Qualification FK can be null since we are not expecting all of the employees to have qualifications.

10.4.3 A3.3

Company vehicle (Listing 3) has a character limit on vehicle name as 48 characters max and driving license is capped at 24 characters max.

10.4.4 A3.4

Customer table (Listing 4) has a limit on express lane. Value is set as Boolean, meaning it can only be 0 (no express lane) or 1 (having express lane). MariaDB

interpreters Booleans as tinyint (it can only have 1 numerical value). To limit the numbers, check was done for number to be only 0 or 1.

10.4.5 A3.5

Customer parking spot (Listing 5) has multi value attribute of parking type. Since parking can have more than one type (VIP and handicapped), JSON object is stored as multi valued item. MariaDB interpreters JSON as longtext, meaning we can have complex objects stored in this attribute. Price per hour is limited from 5 to 300 since this is a range of parking that is acceptable.

10.4.6 A3.6

Department table (Listing 6) has a limit on name of the department (32 characters) and name of the location (24 characters).

10.4.7 A3.7

Employee table (Listing 7) has the most attributes of all. Hire date is set as not null (since there should be a start date) while termination date is unknown. Titles of employees can be long, therefore it is set as 200 characters maximum. Salary of the employee has a big range, but that is subject of change (in real world). Address line 1 is mandatory and has the limit of 48 characters max, while address line 2 is the same it can be empty (null). All of the FKs are set to null as default since employee can not have his parking spot, they can not have company vehicle, they can not be managers of the department. Department FK is the only one that is set as not null since they need to be assigned to one department. Supervisor attribute is a recursive attribute, meaning that this would be an ID of another employee (from the same table). There is a check in place that makes sure that person is paid minimum of £2160 per year in order to be considered as employee. In future version, it might be better to separate salary to another table that also contains employees working schedule.

10.4.8 A3.8

Employee parking spot (Listing 8) is the same as Customer parking spot (Listing 5). The only difference is that employees are not charged for parking.

10.4.9 A3.9

Flight table (Listing 9) contains varchar (plane model, destination and gate), int (flight ID, runway FK, airline FK) and datetime (departure time) attributes. Departure time has date (as 22-8-2022) and time (as 16:30) therefore attribute date was not sufficient.

10.4.10 A3.10

Name table (10) will store names of customers and employees. Therefore we have a composite key as person ID and is employee. Person ID is int value, same as in customer and employee table (their ID). Attribute is employee is a Boolean that represents if someone is employee (marked as 1) or not (marked as 0). Combining person ID and is employee, we will have unique entry for every customer and every employee.

10.4.11 A3.11

Parking token table (11) is providing tokens for parking spaces. Idea is that when customer parks their vehicle, parking token is assigned to them and once they pay parking fee, token is removed from database. Based on the date and time parking token was assigned, calculation can be done how much they need to pay for parking.

10.4.12 A3.12

Runway table (12) stores runways of the airport. It is not expected to have much data inside, since there is a limited number of runways per airport but there is always at least one. Checks are done for length (needs to be between 10m and 3km) and width (between 10m and 500m). At the moment there aren't any helipads stored in the table, but rule can be created if length and width are equal (example: 15m x 15m), this would be represented as a helipad.

10.4.13 A3.13

Last table is for qualifications (13). It limits qualification type and name to have maximum of 48 characters.

10.5 A4

AN AIRLINE provides a minimum of 1 FLIGHT
AN AIRLINE provides a maximum of N FLIGHTs
A FLIGHT is provided by a minimum of 1 AIRLINE
A FLIGHT is provided by a maximum of 1 AIRLINE

A FLIGHT is assigned a minimum of 1 RUNWAY
A FLIGHT is assigned a maximum of 1 RUNWAY
A RUNWAY is assigned to a minimum of 1 FLIGHT
A RUNWAY is assigned to a maximum of 1 FLIGHT

A CUSTOMER is assigned a minimum of 1 NAME
A CUSTOMER is assigned a maximum of 1 NAME

NAME is assigned to a minimum of 1 A CUSTOMER
NAME is assigned to a maximum of 1 A CUSTOMER

A FLIGHT is booked for minimum of 0 CUSTOMERs
A FLIGHT is booked for a maximum of N CUSTOMERs
A CUSTOMER books a minimum of 0 FLIGHTs
A CUSTOMER books a maximum of N FLIGHTs

A CUSTOMER buys a minimum of 0 PARKING TOKENs
A CUSTOMER buys a maximum of 1 PARKING TOKEN
A PARKING TOKEN is bought by a minimum of 1 CUSTOMER
A PARKING TOKEN is bought by a maximum of 1 CUSTOMER

A PARKING TOKEN matches a minimum of 1 CUSTOMERPARKINGSPOT
A PARKING TOKEN matches a maximum of 1 CUSTOMERPARKINGSPOT
A CUSTOMERPARKINGSPOT matches a minimum of 0 PARKING TOKENs
A CUSTOMERPARKINGSPOT matches a maximum of 1 PARKING TOKEN

AN EMPLOYEE parks on a minimum of 0 EMPLOYEEPARKINGSPOTs
AN EMPLOYEE parks on a maximum of 1 EMPLOYEEPARKINGSPOT
AN EMPLOYEEPARKINGSPOT is used by a minimum of 0 EMPLOYEEs
AN EMPLOYEEPARKINGSPOT is used by a maximum of 1 EMPLOYEE

AN EMPLOYEE uses a minimum of 0 COMPANYVEHICLEs
AN EMPLOYEE uses a maximum of 1 COMPANYVEHICLE
A COMPANYVEHICLE is used by a minimum of 0 EMPLOYEEs
A COMPANYVEHICLE is used by a maximum of 1 EMPLOYEE

A COMPANYVEHICLE is parked on a minimum of 0 EMPLOYEEPARKINGSPOTs
A COMPANYVEHICLE is parked on a maximum of 1 EMPLOYEEPARKINGSPOT
AN EMPLOYEEPARKINGSPOT is used by a minimum of 0 COMPANYVEHICLEs
AN EMPLOYEEPARKINGSPOT is used by a maximum of 1 COMPANYVEHICLE

A COMPANYVEHICLE belongs to a minimum of 1 DEPARTMENT
A COMPANYVEHICLE belongs to a maximum of 1 DEPARTMENT
A DEPARTMENT owns a minimum of 0 COMPANYVEHICLEs
A DEPARTMENT owns a maximum of N COMPANYVEHICLEs

AN EMPLOYEE is assigned a minimum of 1 NAME
AN EMPLOYEE is assigned a maximum of 1 NAME
NAME is assigned to a minimum of 1 EMPLOYEE

NAME is assigned to a maximum of 1 EMPLOYEE

AN EMPLOYEE works at a minimum of 1 DEPARTMENT
AN EMPLOYEE works at a maximum of 1 DEPARTMENT
A DEPARTMENT has a minimum of 1 EMPLOYEE
A DEPARTMENT has a maximum of N EMPLOYEEs

AN EMPLOYEE manages a minimum of 0 DEPARTMENTs
AN EMPLOYEE manages a maximum of 1 DEPARTMENTs
A DEPARTMENT is managed by a minimum of 1 EMPLOYEE
A DEPARTMENT is managed by a maximum of 1 EMPLOYEE

AN EMPLOYEE supervises a minimum of 0 EMPLOYEEs
AN EMPLOYEE supervises a maximum of N EMPLOYEEs
AN EMPLOYEE is supervised by a minimum of 0 EMPLOYEEs
AN EMPLOYEE is supervised by a maximum of 1 EMPLOYEE

AN EMPLOYEE achieves a minimum of 0 QUALIFICATIONs
AN EMPLOYEE achieves a maximum of N QUALIFICATIONs
A QUALIFICATION is achieved by a minimum of 0 EMPLOYEEs
A QUALIFICATION is achieved by a maximum of N EMPLOYEEs

10.6 A5

Entity: ParkingSpot
Attributes: ParkingSpotID, Type

Entity: CustomerParkingSpot
Attributes: ParkingSpotID, Type, PricePerHour

Entity: EmployeeParkingSpot
Attributes: ParkingSpotID, Type

Entity: ParkingToken
Attributes: ParkingSpotID, Type

Entity: Customer
Attributes: CustomerId, ExpressLane

Entity: Flight
Attributes: FlightId, Destination, Gate, PlaneModel, DepartureTime

Entity: Runway
Attributes: RunwayId, Length, Width

Entity: Airline

Attributes: CompanyId, CompanyName, Revenue

Entity: Vehicle

Attributes: VehicleId, DrivingLicenseReq, Name

Entity: Department

Attributes: DepartmentId, DepartmentName, Location

Entity: Employee

Attributes: EmployeeId, Address(address line 1, address line 2, Postcode, City),
BirthDate, Age, TerminationDate, Salary, EmploymentType, Title, HireDate

Entity: Qualification

Attributes: QualificationId, Type, Name

Entity: Name

Attributes: PersonID, FirstName, MiddleName, LastName, IsEmployee

10.7 A6

[Airline] 1 <provides> M [Flight]

[Flight] 1 <isAssigned> 1 [Runway]

[Flight] M <isBooked> M [Customer]

[Customer] 1 <buys> 1 [ParkingToken]

[Customer] 1 <has> 1 [Name]

[ParkingToken] 1 <Matches> 1 [CustomerParkingSpot]

[Employee] 1 <parks> 1 [EmployeeParkingSpot]

[CompanyVehicle] 1 <isParked> 1 [EmployeeParkingSpot]

[CompanyVehicle] 1 <isUsed> 1 [Employee]

[CompanyVehicle] M <belongsTo> 1 [Department]

[Employee] 1 <manages> 1 [Department]

[Employee] M <worksAt> 1 [Department]

[Employee] 1 <supervises> M [Employee]

[Employee] M <achieves> M [Qualification]

[Employee] 1 <has> 1 [Name]

10.8 A7

Creating a view for customer names (Listing 21) starts with naming the view. In this case, we have used CustomerNames as the name. What will be returned in the view is ID of the person (Person_ID), their first name (First_Name), middle name (Middle_Name) and last name (Last_Name). Attribute Is_Employee is not needed since we are only returning customers (where Is_Employee equals 0).

Generating employee view (Listing 23) is the same as for customer view (Listing 21). We name the view as EmployeeNames and select the same attributes as in CustomerNames view. The only difference is in WHERE condition, Is_Employee must equal to 1 (meaning is employee).

10.9 A8

For our procedure (Listing 25), we have assigned it to the user 'my_user'. We could lock it down (Wei, Muthuprasanna, and Kothari 2006) and only limit it to certain users (for example: only API calls). In next step, we assign procedure to Airport_DB and name it as GetParkingToBePaid. Since there will be some filtering taking place, an argument of Filter_Customer_ID is accepted.

Now we can start writing the procedure "body". This is SQL code that can work outside of procedure itself. We want for procedure to return ID of the customer (Customer_ID), ID of parking token (Token_ID), when they parked (Date), how much they need to pay (ToPay). ToPay attribute is generated as what is the price of parking spot per hour (Price_Per_Hour) multiplied by difference of when they have parked and current day (in hours). In order for this to work, join is created between Customer, ParkingToken, CustomerParkingSpot table. At the end, we limit the result by ID of the customer that we are searching for.

To execute (call) the procedure (Listing 26), we need to specify what database are we using and procedure name. Since our procedure accepts attributes, we need to insert ID of the customer that we want calculation to be done in the brackets (369 is the ID in this case).

10.10 A9

Work completed by team members:

- Introduction
 - Written by: Zan
- Domain of interest
 - Written by: Daniel & Zan
- Database analysis
 - List of Entity / Attributes
 - * Written by: Daniel & Zan
 - Simple Relationships
 - * Written by: Daniel & Zan
 - Connectivities, Cardinalities and Participation
 - * Written by: Daniel & Zan
- ERD Mapping
 - Mapping 1:1 Relationships
 - * Written by: Daniel & Zan
 - Mapping 1:M Relationships
 - * Written by: Daniel & Zan
 - Mapping N:M Relationships
 - * Written by: Daniel & Zan
 - Mapping A Three-way Relationship
 - * Written by: Daniel & Zan
- ERD
 - Written by: Daniel & Zan
- Database implementation
 - Physical Design
 - * Written by: Zan
 - * Code written by: Zan
 - Create tables
 - * Written by: Zan
 - * Code written by: Zan
 - Assign Foreign Keys

- * Written by: Zan
 - * Code written by: Zan
 - Create views
 - * Written by: Zan
 - * Code written by: Zan
 - Relationship Schema
 - * Written by: Zan
 - Create Procedure
 - * Written by: Zan
 - * Code written by: Zan
 - Dummy Data Insertion
 - * Written by: Daniel & Zan
 - * Code written by: Daniel & Zan
- Testing of the database
 - Written by: Zan
 - Code written by: Zan
- Queries
 - Daniel
 - * Query1
 - Written by: Daniel
 - Code written by: Daniel
 - * Query2
 - Written by: Daniel
 - Code written by: Daniel
 - * Query3
 - Written by: Daniel
 - Code written by: Daniel
 - * Optimisation technique
 - Written by: Daniel
 - Code written by: Daniel
 - Zan
 - * Query1
 - Written by: Zan
 - Code written by: Zan

- * Query2
 - Written by: Zan
 - Code written by: Zan
- * Query3
 - Written by: Zan
 - Code written by: Zan
- * Optimisation technique
 - Written by: Zan
 - Code written by: Zan
- Conclusion
 - Written by: Zan
- Appendixes
 - Written by: Zan

Note: whole report was proofread by both of the team members.

10.11 A10

To test the database, Python script was created to act as unit test (Tuya, Suárez-Cabal, and De La Riva 2006). With the script, we can make sure that future changes to the database will not change any other parameters. With CI/CD pipeline (Poth, Werner, and Lei 2018), this could be automated as well. Do note that in this case we are specifying ID of the employee / parking space due to convince. In real scenario IDs would not be specified in insertion, but they would be assigned automatically.

Alternative approach would be to use MySQL Test Framework (Mysql n.d.). The framework is compatible with MariaDB (Mariadb n.d.[d]) and it was original choice, but use of it was not possible (due to limited knowledge / technical limitations). This is why Python script was created.

10.12 A11

Software used in this project:

- Docker - version 20.10.21, build baeda1f
 - used for vitalising databases
- MariaDB - version 10.9.3

- used for storing data, in order to achieve high availability Galera cluster was added
- Python - version 3.9
 - used for creating data and testing database
- Dbeaver - version 22.3.1
 - user interface for connection to the database(s)

10.13 A12

All of the code is visible on GitHub (ZanZver [n.d.](#)). Seen there is also project description, team, software that was used and environment setup (as step by step guide).

GitHub was proven to be useful tool in for collaboration. By doing small commits, agile approach can be achieved for simpler code moderation and faster development / testing.

Commit section also shows what each team member did, which can be useful in production so work can be monitored.

11 References

References

- Amazon (n.d.). *Amazon RDS High Availability*. Last accessed 8 Jan 2023. URL: <https://aws.amazon.com/rds/ha/>.
- Bitnami (n.d.). *Mariadb-galera container*. Last accessed 8 Jan 2023. URL: <https://hub.docker.com/r/bitnami/mariadb-galera#!>.
- Docker (n.d.[a]). *Docker*. Last accessed 8 Jan 2023. URL: <https://www.docker.com>.
- (n.d.[b]). *Docker Compose*. Last accessed 8 Jan 2023. URL: <https://docs.docker.com/compose/>.
- Harrison, Guy and Steven Feuerstein (2006). *MySQL stored procedure programming*. ” O’Reilly Media, Inc.”
- MAG (n.d.). *Our airports*. Last accessed 8 Jan 2023. URL: <https://www.magairports.com/about-us/our-airports/>.
- Mariadb (n.d.[a]). *Choosing the Right Storage Engine*. Last accessed 8 Jan 2023. URL: <https://mariadb.com/kb/en/choosing-the-right-storage-engine/>.
- (n.d.[b]). *Innodb*. Last accessed 8 Jan 2023. URL: <https://mariadb.com/kb/en/innodb/>.
- (n.d.[c]). *Mariadb*. Last accessed 8 Jan 2023. URL: <https://mariadb.com/products/community-server/>.
- (n.d.[d]). *MySQL-test*. Last accessed 8 Jan 2023. URL: <https://mariadb.com/kb/en/mysqltest/>.
- (n.d.[e]). *S3 Storage Engine*. Last accessed 8 Jan 2023. URL: <https://mariadb.com/kb/en/s3-storage-engine/>.
- (n.d.[f]). *Spider*. Last accessed 8 Jan 2023. URL: <https://mariadb.com/kb/en/spider/>.
- (n.d.[g]). *What is MariaDB Galera Cluster?* Last accessed 8 Jan 2023. URL: <https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/>.
- Mavro, Pierre (2014). *MariaDB High Performance*. Packt Publishing Ltd.
- Mysql (n.d.). *The MySQL Test Framework*. Last accessed 8 Jan 2023. URL: https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html.
- Pina, Eduardo, Filipe Sá, and Jorge Bernardino (2023). “NewSQL Databases Assessment: CockroachDB, MariaDB Xpand, and VoltDB”. In: *Future Internet* 15.1, p. 10.
- Poth, Alexander, Mark Werner, and Xinyan Lei (2018). “How to deliver faster with ci/cd integrated testing services?” In: *European Conference on Software Process Improvement*. Springer, pp. 401–409.
- Sequeda, Juan F, Rudy Depena, and Daniel P Miranker (2009). “Ultrawrap: Using sql views for rdb2rdf”. In: *8th International Semantic Web Conference (ISWC2009), Washington DC, USA*.

- Tuya, Javier, M José Suárez-Cabal, and Claudio De La Riva (2006). “A practical guide to SQL white-box testing”. In: *ACM SIGPLAN Notices* 41.4, pp. 36–41.
- Wei, Kei, Muthusrinivasan Muthuprasanna, and Suraj Kothari (2006). “Preventing SQL injection attacks in stored procedures”. In: *Australian Software Engineering Conference (ASWEC'06)*. IEEE, 8–pp.
- ZanZver (n.d.). *Advanced Databases Coursework*. Last accessed 8 Jan 2023. URL: <https://github.com/ZanZver/AdvancedDatabasesCoursework>.