The Flight Database: Reflective Documentation

Schemas and Entity Relationship Diagram

A schema is a visual or textual representations of the database, including tables, columns, data types and constraints. The database forming this assignment is represented visually in Figure 1 below. Figure 1 shows the five tables forming the database, each table's column names and all primary keys (PK) and foreign keys (FK). All tables have a primary key ('ID') and the flight table has multiple foreign keys because a flight will incorporate multiple staff, one plane and two airports.
The diagram also indicated how many instances of an entity relate to other instances. For example, an airport can have only one status (one and only one), but a status may be held by several airports (one to many).

A database schema may also be represented through SQL. The Flight Database was created by executing a SQL file which effectively contains the schema for the database. This file has been included at *Appendix 1. Database Schema.* This schema fully details all table structure, columns and constraints, primary and foreign keys and also the data which has been used to demonstrate and test the database.

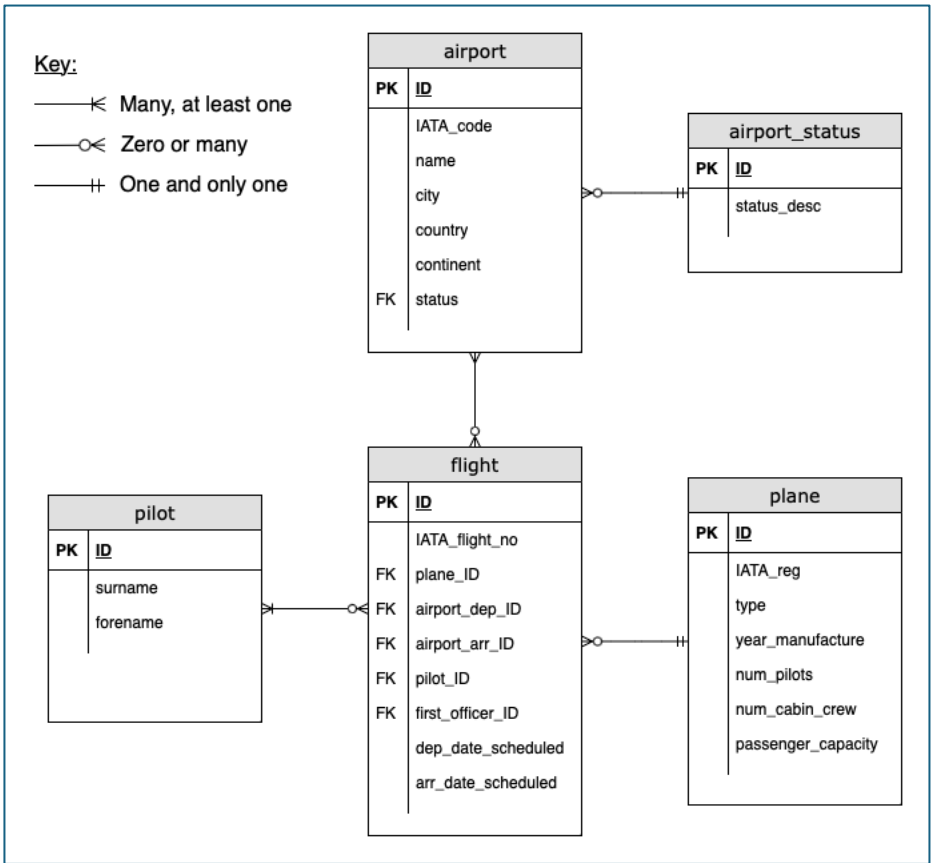

Figure 1: Database Relational Schema (Visual)

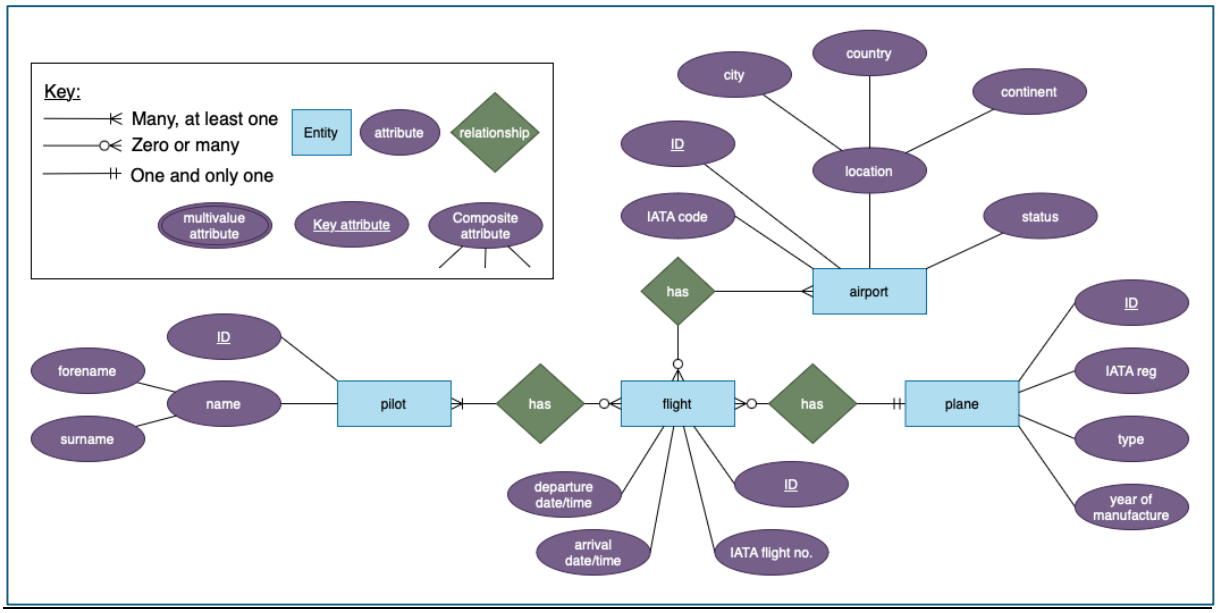

Figure 2: Entity Relationship Diagram

The Entity Relationship Diagram (Figure 2) shows the relationships between entities in the database. The central entity ‘flight’ has relationships with three entities ‘pilot, ‘airport’ and ‘plane’. These entities may have no flights at all assigned to them at any given time but they are all necessary for a flight to exist. There are several composite attributes in the database: airport location which is made up of city, country and continent, and pilot names which are comprised of forename and surname.

Database Structure

Setup

The database is imported from a SQL file 'db_setup.sql'. The file is executed in its entirety by calling a python function that reads the file, opens a database connection, creates a database object and executes it, committing the changes and then closing the connection. In the source code this can be found at file_handler.py/execute_sql_file.
The file contains multiple SQL commands which are all executed which the file is run. It deletes and recreates all tables and views (DROP TABLE/VIEW IF EXISTS and CREATE TABLE/VIEW IF NOT EXISTS respectively) and inserts the starting data (INSERT INTO table_name VALUES…)
This effectively clears the database and resets it to its original state. After the first time the program is run and the initial set up is complete, the import_db() python function in 'main.py' may be deleted or disabled (converted to a comment) using '#' and thereafter any changes made to the database will persist for subsequent runs.
This resetting was found to be useful during the database implementation as the structure of tables and views evolved following their initial set up. Tables, views, queries and values needed to be updated in line with each other as the program developed. Furthermore it was useful to be able remove 'test data' with each run of the program.
More information on the specific 'views', their structure and usage, can be found below.
All tables have an 'ID' column which contains a code unique to the organisation, generally following the structure _ _ I D [int] e.g. 'STID001' for staff IDs. These are always primary keys in each table they are stored in, and often used as foreign keys in queries, allowing detailed data from other tables to be displayed alongside the query table data.

Airport Table

The airport table stores data about the departure and arrival airports from which the fleet operate, including the three-letter IATA code, the airport name and its location (city, country, continent).
These values are generally static but the final value, status, indicates whether an airport is open, open with an alert or closed. This value is held in the airport table as an ID linked to another table, airport_status, which contains the descriptive text (e.g. 'open'). This prevents users from entering free text to represent airport status, only allowing the user to choose from a list of predefined options.
When the program displays airport information to the user, the plain text version is displayed for clarity – more details are provided as to how this is done under 'Queries: view_all_airports'. This could also be done using the enum data type, which is not available in SQLite3.

Plane Table

The plane table stores data about the planes in the fleet. It contains a unique organisation ID, the plane's registration number lodged with the relevant civilian aviation authority (CAA), type, year of manufacture and capacities.
This table feeds into any queries displaying flight information (further details below).

Pilot Table

The pilot table stores data about all pilots. Forenames and surnames can be updated by the program user, but pilot IDs may not be. In a more detailed implementation that would be reserved for a user with higher access privileges.

Flight Table

The flight table stores basic direct information about the scheduled flights – flight ID, departure and arrival dates and times – and also many values which are foreign key references to other tables, such as IDs for pilot and first officer, which reference the pilot table, plane ID which references the plane table, and airport IDs which reference the airport table. As mentioned previously, this allows detailed data from other tables like the plane registration or airport name or code to be displayed alongside the query table data.

Discussion: Database Queries

Views

The program contains three SQL 'views': virtual tables that are generated when the database is imported, allowing for faster access when the program is running. The three views are fundamental queries which print all records in a table. They display selected information to the user and do not contain parameters.

view_all_pilots is the simplest view, displaying all data held in the pilot table. It is constructed as follows:

```
CREATE VIEW IF NOT EXISTS view_all_pilots AS
SELECT * FROM pilot;
```

As one might expect, CREATE VIEW IF NOT EXISTS creates the view if it doesn't already exist. If the program attempts to create a view when one already exists, an error will be thrown : "sqlite3.OperationalError: view [name] already exists". The view is assigned the identifier view_all_staff, and this will be used by the program to use the view.

The SELECT statement determines which columns will be used/displayed by the view and the FROM statement indicates the table name (shortened to alias 'a').

The view_all_flights view uses some more complex commands.

```
CREATE VIEW IF NOT EXISTS view_all_flights AS
SELECT
    f.IATA_flight_no,
    pl.IATA_reg,
    f.airport_dep_ID,
    DATE(f.date_dep_scheduled),
    TIME(f.date_dep_scheduled),
    f.airport_arr_ID,
    DATE(f.date_arr_scheduled),
    TIME(f.date_arr_scheduled),
    pt.surname,
    fo.surname
FROM flight as f
LEFT JOIN plane AS pl ON pl.ID=f.plane_ID
LEFT JOIN pilot AS pt ON pt.ID=f.pilot_ID
LEFT JOIN pilot AS fo ON fo.ID=f.first_officer_ID;
```

It displays data from other tables by listing them in the SELECT statement, such as p.registration. The LEFT JOIN command (using aliases for the plane and pilot tables) links the values needed by the SELECT statement to their location in the other tables – in this case, LEFT JOIN plane AS p ON p.ID=f.plane_ID is this returns records from the plane table, joining them where, for example, the ID in the plane table matches the plane_ID in the flight table. However, only the CAA registration code is displayed to the user (p.registration in the above query) as this is more useful identifying information for business purposes than the internal ID code.

The LEFT JOIN command ensures that all data in the original table (the left table - in this case, flight) is preserved even if there is no corresponding value in the joining table. INNER JOIN, in contrast, drops records which do not have a corresponding value.

IF INNER JOIN used with the above query to join the first officer record onto the flight table, the view would not display flights that did not have a first officer. Hence the LEFT JOIN command is essential in order to display all flights. This matters because not all flights have a first officer – it could be a NULL value.

As plane_ID and pilot_ID are NOT NULL values, the LEFT JOIN/INNER JOIN distinction is not as critical (there will always be a plane registration and pilot for every flight), but it has been implemented here nonetheless as a failsafe.

Queries

Queries were used to interact with the databases where the user wants a more complex data structure returned, such as filtering flights by pilot, departure airport or departure date, or only viewing pilots who have flights scheduled.

Queries are constructed similarly to views. Instead of the CREATE VIEW IF NOT EXISTS command and the assigning of a name to the query, the SELECT command comes first.

In this program the SQL queries are contained in a 'queries.sql' file and extracted using python. They are identified by their name in a comment at the start of the query :

```
-- query_flights_by_pilot
```

and extracted by the load_query() function which takes the query name as a parameter, splits the queries.sql file into sections at each comment and returns the query if it has the correct name, returning an error message if no query could be found. The queries.sql file has to be carefully formatted to allow the individual queries to be identified. See the code file for further details – it has been fully commented.

```
-- query_flights_by_pilot
SELECT
    f.ID,
    f.plane_ID,
    f.airport_dep_ID,
    DATE(f.date_dep_scheduled) AS dep_date,
    TIME(f.date_dep_scheduled) AS dep_time,
    f.airport_arr_ID,
    DATE(f.date_arr_scheduled) AS arr_date,
    TIME(f.date_arr_scheduled) AS arr_time
FROM flight as f
LEFT JOIN pilot as p ON p.ID=f.pilot_ID
WHERE f.pilot_ID = ?
ORDER BY DATE(f.date_dep_scheduled);
```

query_flights_by_pilot returns selected flight data from the flight table, adding pilot information with a LEFT JOIN, ensuring all flight data is preserved before the filter is applied with the WHERE clause. A pilot ID is taken as a parameter and only flights to which that pilot is assigned are then returned. Results are then ordered by departure date.

```
-- query_flights_by_dep_airport
SELECT
    f.ID,
    f.plane_ID,
    f.airport_dep_ID,
    DATE(f.date_dep_scheduled) AS dep_date,
    TIME(f.date_dep_scheduled) AS dep_time,
    f.airport_arr_ID,
    DATE(f.date_dep_scheduled) AS arr_date,
    TIME(f.date_dep_scheduled) AS arr_time,
    p.surname AS pilot,
    fo.surname AS first_officer
FROM flight as f
LEFT JOIN pilot as p ON p.ID=f.pilot_ID
LEFT JOIN pilot as fo ON fo.ID=f.first_officer_ID
WHERE airport_dep_ID = ?
ORDER BY f.airport_dep_ID;
```

query_flights_by_dep_airport takes an airport ID as a parameter and displays all flights departing from that airport. It selects various columns from the flight table adding pilot and first officer data with LEFT JOINS (so as not to drop any records without staff assigned). The filter is applied via a WHERE clause returning records matching the departure airport that is passed as a parameter. The results are then ordered by ID

```
-- query_flights_by_date
SELECT
    f.ID,
    f.plane_ID,
    f.airport_dep_ID,
    DATE(f.date_dep_scheduled) AS dep_date,
    TIME(f.date_dep_scheduled) AS dep_time,
    f.airport_arr_ID,
    DATE(f.date_dep_scheduled) AS arr_date,
    TIME(f.date_dep_scheduled) AS arr_time,
    p.surname AS pilot,
    fo.surname AS first_officer
FROM flight as f
LEFT JOIN pilot as p ON p.ID=f.pilot_ID
LEFT JOIN pilot as fo ON fo.ID=f.first_officer_ID
WHERE DATE(date_dep_scheduled) = ?
ORDER BY f.airport_dep_ID;
```

query_flights_by_date takes a date as a parameter and displays all flights departing on that date. It selects various columns from the flight and pilot tables (joining the pilot data with a LEFT JOIN so all flight data is returned regardless of whether staff data is present or not). The filter is applied via a WHERE clause returning records matching the date that is passed as a parameter.

```
-- view_pilots
SELECT * FROM pilot ORDER BY ID;
```

view_pilots takes no parameters and displays all pilots.

```
-- view_pilots_with_flights_assigned
SELECT
    p.ID AS pilot_ID,
    p.surname AS Surname,
    p.forename AS Forename
FROM pilot as p
INNER JOIN flight as f ON f.pilot_ID=p.ID
GROUP BY p.ID
ORDER BY p.surname;
```

view_pilots_with_flights takes no parameters but displays all pilots who have been assigned to flights. This is done by selecting all column in the staff table and joining the flight table with an INNER JOIN for all records where the pilot ID for the flight matches the staff ID. The INNER JOIN ensures that all pilots that do not have flights assigned to them are not returned by the query. Results are ordered alphabetically by surname.

```
-- count_flights_to_destinations
SELECT
    a.airport_name,
    COUNT(*) AS flight_count
FROM flight AS f
LEFT JOIN airport AS a ON f.airport_arr_ID=a.IATA_code
GROUP BY f.airport_arr_ID
ORDER BY flight_count DESC;
```

Count_flights_to_destinations counts the number of records for each arrival airport ID in the flight table. It then groups them by that ID and rreturns the total for each destination. Finally it orders the results in descending order so the destination with the largest number of flights is at the top. It would be possible to add a join to this query to display more information, such as the airport name.

```
-- count_flights_to_destinations_greater_1
SELECT
    a.airport_name,
    COUNT(*) AS flight_count
FROM flight AS f
LEFT JOIN airport AS a ON f.airport_arr_ID=a.IATA_code
GROUP BY f.airport_arr_ID
HAVING flight_count >1
ORDER BY flight_count DESC;
```

Count_flights_to_destinations_greater_1 operates similarly to the above query but it only displays destinations with more than one flight using the HAVING > 1 statement, which operates on the query after the groupings have been made.

Adding a new record

```
sql = f"INSERT INTO {dict['table_name']} ({columns_str}) VALUES ({placeholders})"
```

After the program extracts new values from the user via python (see source code for details), those values can be input into the database by executing the above query.

Following the SOLID, DRY (Don't Repeat Yourself) principles, the python code enables the above query to input different volumes of data into different tables

⇒ dict['table_name'] will get the relevant table name ('airport' for example) from a dictionary and pass it into the query, depending on what data the user is trying to input. This determines the table name for the purposes of the query.

⇒ 'columns_str' is constructed in the python code , for example:
['ID', 'IATA_code', 'airport_name', 'city', 'country', 'continent', 'status']

⇒ 'placeholders' constructs the required number of ?s to be included in the query as a string:

"?, ?, ?, ?, ?, ?, ?"

⇒ The final SQL query might look like this:
"INSERT INTO airport (ID, IATA_code, airport_name, city, country, continent, status) VALUES (?, ?, ?, ?, ?, ?, ?)"

⇒ It is then executed using the values taken from the user as a parameter :
('APID009', 'BRI', 'Bristol Airport', 'Bristol', 'UK', 'Europe', '1')

And this will enter the values into the airport table of the database in a secure manner. Parameterised queries as a means of SQL injection attacks as user inputs are not executed when the query is.

Deleting a record

```
cursor.execute(f"DELETE FROM {dict['table_name']} WHERE id = ?", (record_id,))
```

The program uses python to interact with the user and determine which record ID needs to be deleted (see source code). The above query will remove the record represented by it's primary key ID (passed in as a parameter record_id) from the table.

Updating a record

The query below operates in a similar manner to the 'Deleting a record' query above.

```
sql = f"UPDATE {dict['table_name']} SET {column_to_update} = ? WHERE ID = ?"
```

dict['table_name] returns the table containing the record to be updated, 'column_to_update' points to the column name of the value to be updated. The first ? is where is new value is passed in to the query and the second is where the record ID (primary key column) is passed in, determining which record is updated.

Challenges and Solutions: Reflection on the Assignment

At the planning stages I considered features such as using actual departure and arrival times which, when compared to scheduled times, would indicate the 'on time'/delayed status of a plane. A plane's flight history could also give an indication of its current location – similarly for pilots. Unfortunately time constraints and the complexity of the implementation of such features resulted in the scope being strictly limited to the requirements set out in the assignment brief, and it is my hope that this has resulted in the project being small but well executed.

One of the biggest challenges was the console-based menu system. In order for the user to be presented with a simple, easy to navigate menu system, consideration had to be given to factors such as user input validation and exception handling. It has been possible to constrain user inputs in some areas of the console application, allowing for error handling before data reaches the database. However, with more time a robust implementation would have included data input constraints at the point of entry to the database as well, to protect the integrity of the database and its functions that relying on the data being in a particular format.

From a security perspective, the database program has a simple username/password login system with verified user login details hardcoded into the application. This was simply to demonstrate that security has been considered in the program's design. Naturally in a fuller implementation or real-world scenario verified user details are not held within the program in plain text format, they would be hashed or salted and stored in a different location.

Care has been taken not to offer the user excessive access to the database. For example, a user may change a staff member's name or role, but not their staff number. They may update a flight's departure or arrival time, but not the flight number or plane ID. A 'next step' for this project would be another layer of access privileges for administrative users with the option to make more sensitive changes to the database.

Parameterised queries were used in all cases where user inputs were included in a SQL query string in order to protect against SQL injection attacks.

Time was spent formatting the printing of query results to the console in order to produce well formatted tables, but it was difficult to achieve the desired result and in some cases, while the data return is correct, the column headers are mismatched. The issue lies with the formatting functions in python that rely on predetermined headers being passed in for formatting. These were created for some of the larger queries, returning most or all of the columns in a table, but it has not been done for the smaller queries due to a lack of time. In the future I would simply use the

```
results = cursor.fetchall()
```

command and print using a python print() statement as appropriate. The results are not as good but given that the assignment related to databases and not python programming this was not a great use of time in respect of this assignment.

Appendix 1. Database Schema

```
DROP TABLE IF EXISTS flight;
DROP TABLE IF EXISTS airport;
DROP TABLE IF EXISTS airport_status;
DROP TABLE IF EXISTS plane;
DROP TABLE IF EXISTS pilot;

DROP VIEW IF EXISTS view_all_flights;
DROP VIEW IF EXISTS view_all_airports;
DROP VIEW IF EXISTS view_all_pilots;


CREATE TABLE IF NOT EXISTS flight (
    ID VARCHAR(20) NOT NULL,
    IATA_flight_no VARCHAR(20) NOT NULL,
    plane_ID VARCHAR(20) NOT NULL,
    airport_dep_ID VARCHAR(3) NOT NULL,
    airport_arr_ID VARCHAR(3) NOT NULL,
    pilot_ID VARCHAR(20) NOT NULL,
    first_officer_ID VARCHAR(20),
    date_dep_scheduled DATETIME NOT NULL,
    date_arr_scheduled DATETIME NOT NULL,
    FOREIGN KEY (plane_ID) REFERENCES plane(ID),
    FOREIGN KEY (airport_dep_ID) REFERENCES airport(ID),
    FOREIGN KEY (airport_arr_ID) REFERENCES airport(ID),
    FOREIGN KEY (pilot_ID) REFERENCES pilot(ID),
    FOREIGN KEY (first_officer_ID) REFERENCES pilot(ID),
    PRIMARY KEY (ID)
    );

CREATE TABLE IF NOT EXISTS airport (
    ID VARCHAR(20) NOT NULL,
    IATA_code  VARCHAR(3) NOT NULL,
    airport_name VARCHAR(255) NOT NULL,
    city VARCHAR(255) NOT NULL,
    country VARCHAR(255) NOT NULL,
    continent VARCHAR(255) NOT NULL,
    status INT NOT NULL,
    FOREIGN KEY (status) REFERENCES airport(ID),
    PRIMARY KEY (ID)
    );

CREATE TABLE IF NOT EXISTS airport_status (
    ID int NOT NULL,
    status_desc VARCHAR(20) NOT NULL,
    PRIMARY KEY (ID)
    );

CREATE TABLE IF NOT EXISTS plane (
    ID VARCHAR(20) NOT NULL,
    IATA_reg VARCHAR(20) NOT NULL UNIQUE,
```

```
    type VARCHAR(255),
    year_manufacture YEAR,
    num_pilots INT,
    num_cabin_crew INT,
    passenger_capacity INT,
    PRIMARY KEY (ID)
    );

CREATE TABLE IF NOT EXISTS pilot (
    ID VARCHAR(20) NOT NULL,
    surname VARCHAR(255) NOT NULL,
    forename VARCHAR(255) NOT NULL,
    PRIMARY KEY (ID)
    );

INSERT INTO flight VALUES ('FLID001', 'US1549', 'PLID001', 'LGA', 'SEA', 'STID001', 'STID002',
'2025-06-10 07:20:00', '2025-06-10 08:30:00');
INSERT INTO flight VALUES ('FLID002', 'AFR447', 'PLID002', 'GIG', 'CDG', 'STID003', 'STID004',
'2025-06-11 22:29:00', '2025-06-12 09:03:00');
INSERT INTO flight VALUES ('FLID003', 'MH370', 'PLID003', 'KUL', 'PEK', 'STID005', 'STID006',
'2025-06-15 00:30:00', '2025-06-15 06:55:00');
INSERT INTO flight VALUES ('FLID004', '4U9525', 'PLID004', 'BCN', 'DUS', 'STID007', 'STID008',
'2025-07-01 10:00:00', '2025-07-01 11:30:00');
INSERT INTO flight VALUES ('FLID005', 'JT610', 'PLID005', 'CGK', 'PGK', 'STID009', 'STID010',
'2025-07-15 06:20:00', '2025-07-15 06:33:00');
INSERT INTO flight VALUES ('FLID006', 'UA232', 'PLID006', 'DEN', 'SUX', 'STID011', 'STID012',
'2025-07-20 14:09:00', '2025-07-20 16:00:00');
INSERT INTO flight VALUES ('FLID007', 'AA1234', 'PLID001', 'LGA', 'SEA', 'STID001', 'STID002',
'2025-06-10 15:30:00', '2025-06-10 18:30:00');
INSERT INTO flight VALUES ('FLID008', 'AFR123', 'PLID002', 'CDG', 'SEA', 'STID003', 'STID004',
'2025-06-10 23:00:00', '2025-06-11 10:15:00');
INSERT INTO flight VALUES ('FLID009', 'MH999', 'PLID003', 'KUL', 'SEA', 'STID005', 'STID006',
'2025-06-10 08:00:00', '2025-06-10 15:00:00');
INSERT INTO flight VALUES ('FLID010', 'UA5678', 'PLID004', 'DUS', 'BCN', 'STID007', 'STID008',
'2025-06-12 12:45:00', '2025-06-12 14:00:00');
INSERT INTO flight VALUES ('FLID011', 'JT123', 'PLID005', 'CGK', 'PGK', 'STID009', 'STID010',
'2025-07-05 06:00:00', '2025-07-05 06:20:00');
INSERT INTO flight VALUES ('FLID012', 'UA777', 'PLID006', 'DEN', 'SUX', 'STID011', 'STID012',
'2025-07-05 12:30:00', '2025-07-05 13:45:00');
INSERT INTO flight VALUES ('FLID013', 'BA5390', 'PLID005', 'BIR', 'MAL', 'STID003', 'STID006',
'2025-06-10 07:20:00', '2025-06-10 10:15:00');


INSERT INTO airport VALUES ('APID001', 'LGA', 'LaGuardia Airport', 'New York City', 'USA', 'North
America', 1);
INSERT INTO airport VALUES ('APID002', 'SEA', 'Seattle Tacoma International Airport', 'Seattle',
'USA', 'North America', 1);
INSERT INTO airport VALUES ('APID003', 'GIG', 'Rio de Janeiro Galeão International Airport', 'Rio
de Janeiro', 'Brazil', 'South America', 1);
INSERT INTO airport VALUES ('APID004', 'CDG', 'Charles de Gaulle Airport', 'Paris', 'France',
'Europe', 2);
```

```
INSERT INTO airport VALUES ('APID005', 'KUL', 'Kuala Lumpur International Airport', 'Kuala Lumpur',
'Malaysia', 'Asia', 1);
INSERT INTO airport VALUES ('APID006', 'PEK', 'Beijing Capital International Airport', 'Beijing',
'China', 'Asia', 3);
INSERT INTO airport VALUES ('APID007', 'BCN', 'Barcelona El Prat Airport', 'Barcelona', 'Spain',
'Europe', 1);
INSERT INTO airport VALUES ('APID008', 'DUS', 'Düsseldorf Airport', 'Düsseldorf', 'Germany',
'Europe', 1);
INSERT INTO airport VALUES ('APID009', 'CGK', 'Soekarno Hatta International Airport', 'Jakarta',
'Indonesia', 'Asia', 1);
INSERT INTO airport VALUES ('APID010', 'PGK', 'Depati Amir Airport', 'Pangkal Pinang', 'Indonesia',
'Asia', 3);
INSERT INTO airport VALUES ('APID011', 'DEN', 'Denver International Airport', 'Denver', 'USA',
'North America', 1);
INSERT INTO airport VALUES ('APID012', 'SUX', 'Sioux Gateway Airport', 'Sioux City', 'USA', 'North
America', 1);
INSERT INTO airport VALUES ('APID013', 'BIR', 'Birmingham Airport', 'Birmingham', 'UK', 'Europe',
1);
INSERT INTO airport VALUES ('APID014', 'MAL', 'Malaga Airport', 'Malaga', 'Spain', 'Europe', 1);

INSERT INTO airport_status VALUES (1, 'open');
INSERT INTO airport_status VALUES (2, 'open: alert');
INSERT INTO airport_status VALUES (3, 'closed');

INSERT INTO plane VALUES ('PLID001', 'N106US', 'Airbus A320-214', 1984, 2, 3, 150); -- US Airways
Flight 1549 (Airbus A320)
INSERT INTO plane VALUES ('PLID002', 'F-GZCP', 'Airbus A330-203', 2005, 3, 9, 216); -- Air France
Flight 447 (Airbus A330)
INSERT INTO plane VALUES ('PLID003', '9M-MRO', 'Boeing 777-200ER', 1997, 3, 10, 282); -- Malaysia
Airlines Flight MH370 (Boeing 777)
INSERT INTO plane VALUES ('PLID004', 'D-AIPX', 'Airbus A320-211', 1990, 2, 4, 150); -- Germanwings
Flight 9525 (Airbus A320)
INSERT INTO plane VALUES ('PLID005', 'PK-LQP', 'Boeing 737-8U3', 2015, 2, 3, 189); -- Lion Air
Flight 610 (Boeing 737 Max)
INSERT INTO plane VALUES ('PLID006', 'N181UA', 'McDonnell Douglas DC-10-10', 1970, 3, 6, 270); --
United Airlines Flight 232 (McDonnell Douglas DC-10)
INSERT INTO plane VALUES ('PLID007', 'G-BGJL', 'Boeing 737-436', 1989, 2, 2, 148); -- Boeing 737-
400 (used for BA5390)

INSERT INTO pilot VALUES ('STID001', 'Sullenberger', 'Chesley'); -- US Airways Flight 1549
(Captain)
INSERT INTO pilot VALUES ('STID002', 'Skiles', 'Jeffrey'); -- US Airways Flight 1549 (First
Officer)
INSERT INTO pilot VALUES ('STID003', 'Chavarria', 'Marc'); -- Air France Flight 447 (Captain)
INSERT INTO pilot VALUES ('STID004', 'Pinto', 'David'); -- Air France Flight 447 (First Officer)
INSERT INTO pilot VALUES ('STID005', 'Zaharie', 'Shah'); -- Malaysia Airlines Flight MH370
(Captain)
INSERT INTO pilot VALUES ('STID006', 'Fariq', 'Abdul'); -- Malaysia Airlines Flight MH370 (First
Officer)
INSERT INTO pilot VALUES ('STID007', 'Lubitz', 'Andreas'); -- Germanwings Flight 9525 (Co-pilot)
INSERT INTO pilot VALUES ('STID008', 'Wunderlich', 'Patrick'); -- Germanwings Flight 9525 (Captain)
```

```sql
INSERT INTO pilot VALUES ('STID009', 'Sari', 'Raimond'); -- Lion Air Flight 610 (Captain)
INSERT INTO pilot VALUES ('STID010', 'Supriatna', 'Hanafi'); -- Lion Air Flight 610 (First Officer)
INSERT INTO pilot VALUES ('STID011', 'McDaniel', 'Dennis'); -- United Airlines Flight 232 (Captain)
INSERT INTO pilot VALUES ('STID012', 'Campbell', 'Michael'); -- United Airlines Flight 232 (First
Officer)
INSERT INTO pilot VALUES ('STID013', 'Lancaster', 'Timothy'); -- Captain Timothy Lancaster
INSERT INTO pilot VALUES ('STID014', 'Atchison', 'Alastair'); -- First Officer Alastair Atchison

CREATE VIEW IF NOT EXISTS view_all_flights AS
SELECT
    f.IATA_flight_no,
    pl.IATA_reg,
    f.airport_dep_ID,
    DATE(f.date_dep_scheduled),
    TIME(f.date_dep_scheduled),
    f.airport_arr_ID,
    DATE(f.date_arr_scheduled),
    TIME(f.date_arr_scheduled),
    pt.surname,
    fo.surname
FROM flight as f
LEFT JOIN plane AS pl ON pl.ID=f.plane_ID
LEFT JOIN pilot AS pt ON pt.ID=f.pilot_ID
LEFT JOIN pilot AS fo ON fo.ID=f.first_officer_ID
ORDER BY DATE(f.date_dep_scheduled);

CREATE VIEW IF NOT EXISTS view_all_airports AS
SELECT
    a.IATA_code,
    a.airport_name,
    a.city,
    a.country,
    a.continent,
    st.status_desc
FROM airport as a
LEFT JOIN airport_status AS st ON st.ID=a.status
ORDER BY a.IATA_code;

CREATE VIEW IF NOT EXISTS view_all_pilots AS
SELECT * FROM pilot
ORDER BY ID;
```