Documentation

Requirements Specification

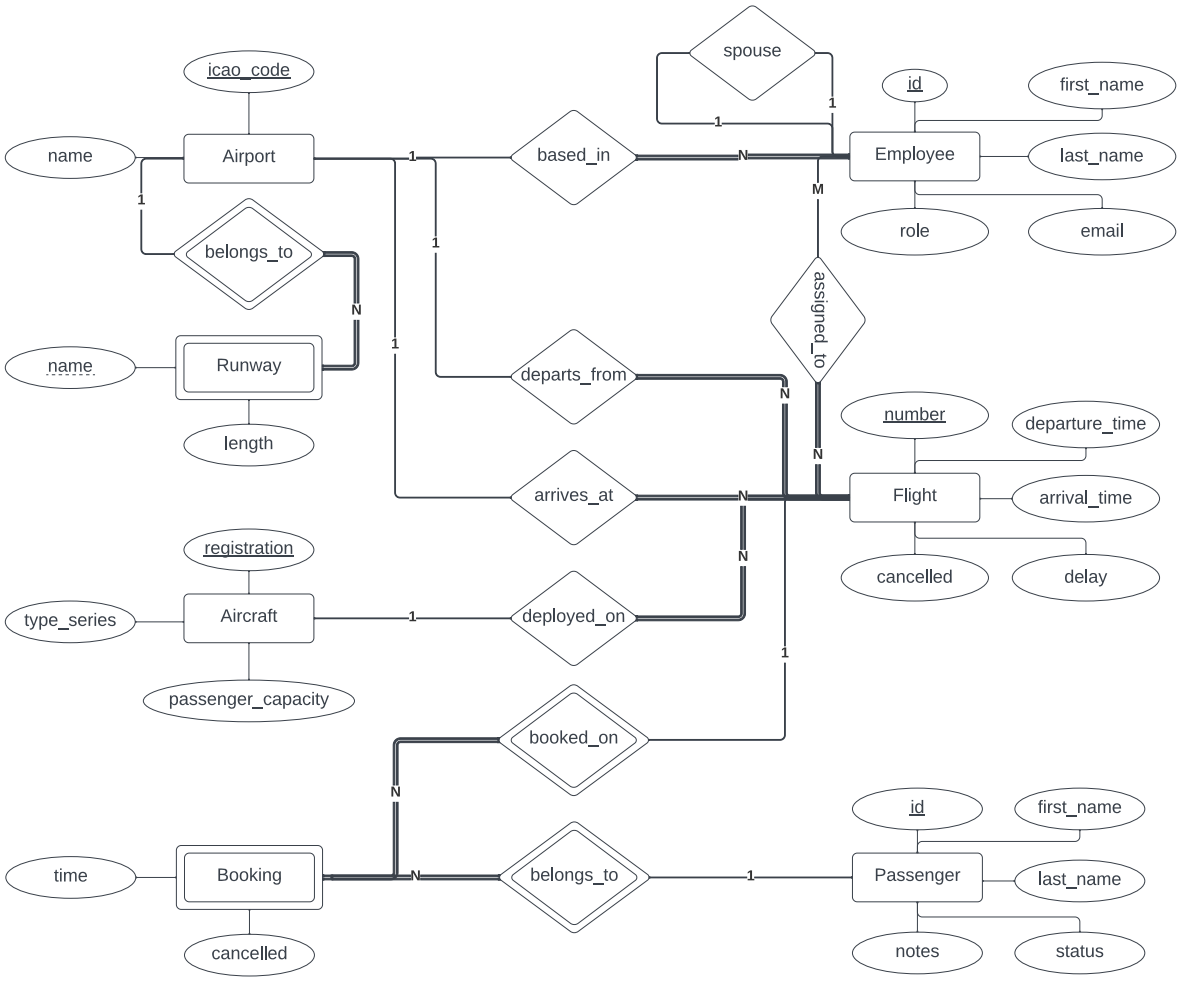
The aviation management software suite FlightX, serves as common ground flight management for airlines and airports. Its two components are AirportX (supporting the management of large aviation facilities) and AirlineX (used for flight management).

- FlightX needs an ICAO (International Civil Aviation Organization) code to uniquely identify each managed **airport**. Additionally a human readable airport name has to be stored for a better user experience.
  - Each Airport can have multiple **runways** assigned, that have a certain length and a name (e.g. 07C). As they are only uniquely identifiable through the name in combination with the airport code, this is a weak entity type.*
- To track individual **aircraft**, the software has to keep track of each aircraft's unique registration, its series type (A380-800, A350-800, A350-900, A350-1000, A320-100, A320-200, A320neo, B777-200, B777-300, B787-8, B787-9, B747-400, B747-8I, B737-MAX10) and the available passenger capacity.
- Each **passenger** is described by their first and last name, their status (Bronze, Silver, Gold, Platinum) and optionally a short note with additional information. An artificial ID uniquely identifies each customer.
- Each **employee** is described by their first and last name, their employee email address and their role (Captain, First Officer, Second Officer, Cabin Crew). An artificial ID uniquely identifies each employee. In addition to that, each employee has a base where he is stationed (one of the airports).
  - For better employee management the system also stores marital relations between coworkers if existent (One-to-One)
- To manage **flights**, the software has to keep track of the flight number which uniquely identifies each flight, the departure airport, the destination airport, the aircraft, departure time, arrival time, potential delay time as well as whether the flight has been canceled or not.
  - A cancellation should be automatically applied to the corresponding bookings. Additionally, the employees that work on any given flight must be known (Trigger and Procedure) so that their assignments can be deleted from the database.
  - An employee can be assigned to multiple flights through an assignment consisting of the employee ID and the unique flight number (Many-to-Many).
- The **bookings** are uniquely identified by flight and a passenger and are therefor a weak entity. Bookings also contain a timestamp from when the booking was made and the cancellation status.

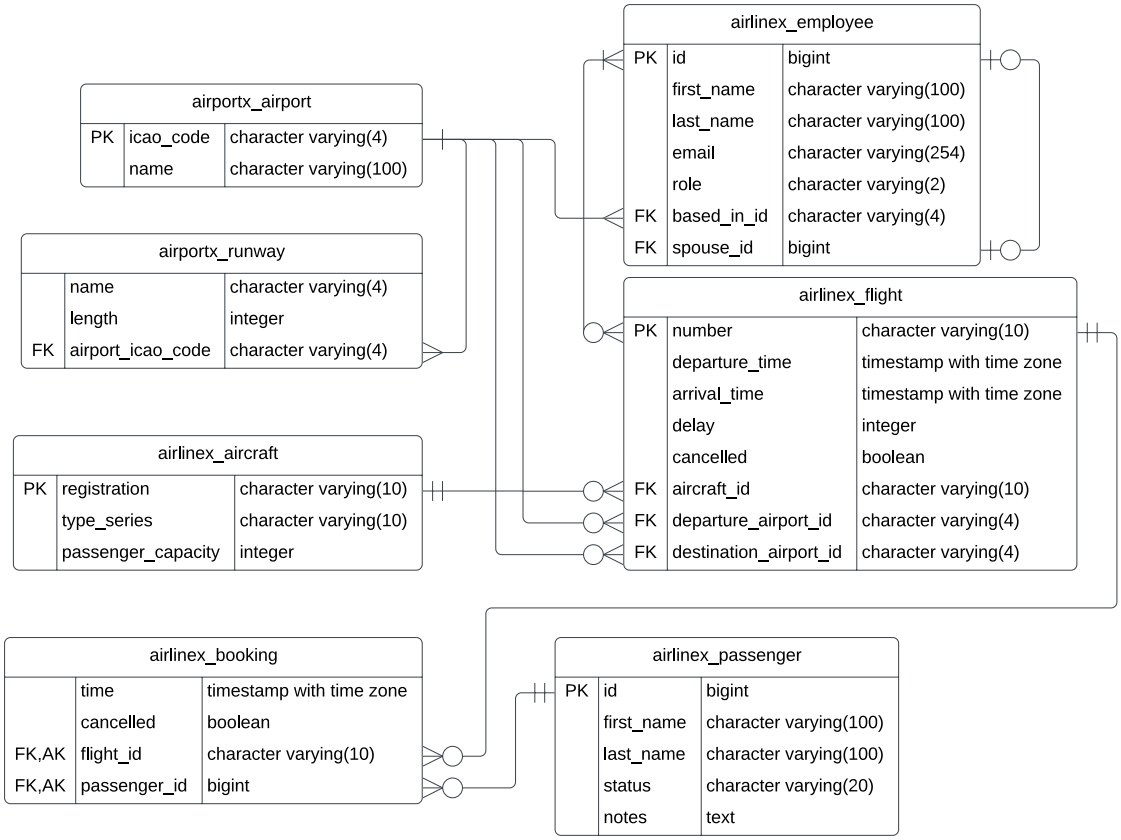
* Note: As the Runway for a flight is decided by the ATC (Air Traffic Control) just before the approach at the airport, in accordance to current weather conditions and availability, it is not part of the flight relation. Additionally it does not add any value for the systems user to know the runway of a flight. Runways are usually approachable from two sides, while one side might have a different available runways length as the other, for instance due to noise protection (e.g. London City Airport's runway). This is why each runway has two separate database entries, one for each direction. The runway name is composed of the course (e.g. 07 for 070 degrees) and the position in case of several parallel runways in the same direction (e.g. L for Left, R for Right, C for Center).

Entity Relationship Diagram

The entity relationship diagram in Chen's notation contains its relationship in the form of logical connecting words, while foreign key attributes are omitted as proposed by Peter Chen.



While the ERD in Chen's notation gives a good overview, we think that the Crow's foot notation provides additional insights over the actual implementation of the database. Hence, we are providing the ERD in Crow's foot notation as a bonus.



Normalization

This is an analyses to check whether all database relations satisfy third normal form (3NF). For this to be the case the following criteria have to be met:

- To be in 1NF the domain of each attribute needs to have atomic values and the value of any attribute in a tuple must be a single value from the domain.
- To be in 2NF a relation R has to satisfy 1NF and all non-prime attributes A in R have to be functionally dependent on any candidate key of R.
- To be in 3NF a relation R has to satisfy 2NF and all non-prime attributes of R can not be transitively dependent on any candidate key.

Check for 1NF: Every attribute in every relation only has values that cannot be divided further (no composite attributes). Therefore the database satisfies first normal form (see definition above). This includes any attributes that hold a date and time. Even though these could be stored separately as a date and a time attributes, in our case they are implemented as a datetime attribute which is atomic.

Check for 2NF: The analysis shows that all relations only include attributes that that are fully functionally dependent on the candidate keys (e.g. primary key) and first normal form is satisfied (see above). Hence all relations are in second normal form.

Check for 3NF: No relation in our database contains non-prime attributes that are transitively dependent ($\{X \to Z \land Z \to Y\} \implies X \to Y$) on any candidate key (e.g. primary key). This in addition to 2NF being satisfied, as demonstrated above, means that all relations are in third normal form.

SQL Scripts

This section describes the SQL scripts that can be found in the repository.

To comply with the point 3.1 in the project_description file, use `DDL_from_ER.sql` to setup the database tables as specified in the ERD. This file also contains all SQL statements to create procedures, functions, triggers and views. As a bonus, we used a

stored procedure and a stored function although only one was required, because it made sense to create the conditional execution of the procedure that way.

In order to populate the database tables with sample content, please use the `DML_from_ER.sql` script.

A dump of the entire Django application level database can be found in the `DDL_and_DML_from_APP.sql` file. The superuser login data is:

Username: postgres

Password: postgres

Note that the Database structure used with the application deviates slightly from the ERD as Django does not support composite primary keys and weak entities. The corresponding entities have surrogate keys generated from sequences as agreed via email before the submission of this project.

Views

The database has two views (one simple view and one materialized view).

The airport_stats view aggregates data from multiple tables to provide statistics for each airport. Specifically, it includes the following fields:

- icao_code: The unique identifier for each airport, as defined in the airportx_airport table.
- avg_delay: The average delay of all non-cancelled flights departing from or arriving at the airport, as calculated from the airlinex_flight table.
- num_flights: The total number of non-cancelled flights departing from or arriving at the airport, as counted from the airlinex_flight table.

The view uses a JOIN to link the "airportx_airport" and "airlinex_flight" tables based on the airport's ICAO code and the flight's departure or destination airport ID, and groups the results by the airport's ICAO code.

```
CREATE OR REPLACE VIEW airport_stats AS
SELECT
  airportx_airport.icao_code,
  AVG(airlinex_flight.delay) AS average_delay,
  COUNT(airlinex_flight.number) AS number_flights
FROM
  airportx_airport
  JOIN airlinex_flight
  ON airportx_airport.icao_code = airlinex_flight.departure_airport_id
  OR airportx_airport.icao_code = airlinex_flight.destination_airport_id
GROUP BY
  airportx_airport.icao_code;
```

The airport_stats view serves the purpose of allowing members of an Aviation Organization or Airline to track the utilization of air traffic capacity of each airport as well as measure their performance efficiency through the average delay of flights.

The airport_and_based_crew view is a materialized view. This means it is stores the query result persistently and has to be refreshed to incorporate updates in underlying tables. It is derived from the tables airportx_airport and airlinex_employee and calculates the number of employees based at each airport in the database. The view is created by joining the "airportx_airport" and "airlinex_employee" tables on the airport's icao_code and the employee's based_in_id. The result is then grouped by the airport's icao_code:

```
CREATE MATERIALIZED VIEW airport_and_based_crew
AS SELECT airportx_airport.*,
COUNT(airlinex_employee.based_in_id) AS num_employees
FROM airportx_airport
LEFT JOIN airlinex_employee
ON airportx_airport.icao_code = airlinex_employee.based_in_id
GROUP BY airportx_airport.icao_code;
```

The airport_and_based_crew view can help Airlines with their flight scheduling since for each flight they have to check if there is enough staff at the relevant Airport. Because of the frequency with which this information is required as well as the fact that the base airport for an employee only changes very infrequently, it makes sense to increase performance by creating a materialized view for this task.

Secondary Indices

We implemented a secondary index for the airport name, as airports might be frequently searched for using their name because most people do not have all the airport ICAO codes in mind. Analyzing the performance difference using a Django filter query, we found out that:

	Without Secondary Index	With Secondary Index
Execution time 5 Airports	0.026ms	0.025ms
Execution time 27.170 Airports	2.882ms	0.015ms

Conclusion: A secondary index is justified, when handling a large number of airports as expected by the business use case.

Interestingly, the execution time with secondary index is even lower for the large number of airports compared to filtering 5 airports. Double checking the application & database and repeating the experiment confirmed the above results.

Further candidates for a secondary index would be the last name of passengers as well as employees. This has been implemented accordingly.

Stored procedure

A good use case for a SQL stored procedure in a flight management system is to handle the process of canceling a flight. The stored procedure takes the flight ID as an input parameter and then performs/triggers the following actions:

1. Check if there are any bookings for the flight. If there are, the procedure cancels the bookings.
2. Remove the assignments for the crew members on the flight.
3. Mark the flight as cancelled in the flights table. (Performed automatically by Django)

This process involves updating multiple tables and handling business logic, making it a good candidate for a stored procedure. The benefits of using a stored procedure for this process include:

1. Improved performance: Since the stored procedure is executed on the database server, it can perform the necessary actions more efficiently than executing multiple separate queries from the application.
2. Data integrity: The stored procedure can ensure that all necessary updates are performed, and that the data is kept consistent.
3. Reusability: The same stored procedure can be used for cancelling flights across different parts of the application, ensuring consistency and reducing code duplication.

```
-- Stored Procedure
CREATE OR REPLACE PROCEDURE cancel_flight(flight_number VARCHAR)
LANGUAGE plpgsql
AS $$
BEGIN
  -- Update bookings
  UPDATE airlinex_booking SET cancelled = true WHERE flight_id = flight_number;
  -- Remove crew assignments for the flight
  DELETE FROM airlinex_assignment WHERE flight_id = flight_number;
END;$$;

-- Trigger function to trigger and check for cancellation
CREATE OR REPLACE FUNCTION cancel_flight_trigger_function()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.cancelled THEN
    CALL cancel_flight(NEW.number);
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

The trigger function first performs a check whether the update includes the flight cancellation and calls the stored procedure accordingly.

The trigger triggering the above trigger function is implemented by:

```
CREATE TRIGGER cancel_flight_trigger
AFTER UPDATE ON airlinex_flight
FOR EACH ROW
EXECUTE FUNCTION cancel_flight_trigger_function();
```

Advanced SQL queries

We are presenting several queries for each use case to point out multiple ways of solving each problem. This is to demonstrate our ability to implement different approaches in SQL, every one of them having its advantages and disadvantages to it.

1. Shared flights

A general rule implemented by all airlines is that cockpit crews should always be well mixed. This is because if two pilots fly together too often, it can no longer be ensured that unbiased and professional behavior prevails at all times.

To check whether the two pilots Jürgen Raps and Joong Gi Joost have already flown together, we want to retrieve all flight numbers that both were assigned to using the following query:

```
-- Option 1: Works, but would also retrieve flights that the hypothetical pilots Jürgen Joost and Joong Gi Raps would be on.
SELECT DISTINCT f.number AS "Shared flights"
FROM airlinex_flight AS f
JOIN airlinex_assignment AS a ON a.flight_id = f.number
WHERE a.employee_id IN
(SELECT id FROM airlinex_employee AS e
WHERE e.first_name IN ('Jürgen', 'Joong Gi')
AND e.last_name IN ('Raps', 'Joost'));

-- Option 2: A bit longer but prevents risk from option 1
SELECT DISTINCT f.number AS "Shared flights"
FROM airlinex_flight AS f
JOIN airlinex_assignment AS a ON a.flight_id = f.number
WHERE a.employee_id IN
(SELECT id FROM airlinex_employee AS e
WHERE (e.first_name = 'Jürgen' AND e.last_name = 'Raps')
OR (e.first_name = 'Joong Gi' AND e.last_name = 'Joost'));

-- Option 3: More efficient solution as it does not need a subquery
-- The improvement in execution time is roughly 4x
SELECT DISTINCT f.number AS "Shared flights"
FROM airlinex_flight AS f
JOIN airlinex_assignment AS a ON a.flight_id = f.number
JOIN airlinex_employee AS e ON e.id = a.employee_id
WHERE (
  (e.first_name = 'Jürgen' AND e.last_name = 'Raps')
  OR (e.first_name = 'Joong Gi' AND e.last_name = 'Joost')
);
```

2. Finding the all time passenger

The following query intends to find the passenger(s) with status “Platinum” that is booked on every flight. To accomplish that it (Option 1) retrieves the first and last names of passengers (p) who have a 'Platinum' status. To check if the corresponding passenger is booked on all flights, it joins the airlinex_booking table (b) on the airlinex_passenger table (p) on passenger_id and the airlinex_flight table (f) on flight_id and number. It then groups by the passenger id in order to count all flights of a passenger and compare this number with the total number of flights.

```
-- Option 1
SELECT p.first_name, p.last_name
FROM airlinex_booking b
INNER JOIN airlinex_passenger p ON b.passenger_id = p.id
INNER JOIN airlinex_flight f ON b.flight_id = f.number
WHERE p.status = 'P'
GROUP BY p.id
HAVING COUNT(DISTINCT f.number) = (SELECT COUNT(*) FROM airlinex_flight);
```

```
-- Option 2: Better efficiency due to omitted Group By
SELECT distinct p.first_name, p.last_name
FROM airlinex_booking b
INNER JOIN airlinex_passenger p ON b.passenger_id = p.id
WHERE p.status = 'P'
AND NOT EXISTS (
    SELECT 1
    FROM airlinex_flight f
    WHERE NOT EXISTS (
        SELECT 1
        FROM airlinex_booking b2
        WHERE b2.flight_id = f.number
        AND b2.passenger_id = p.id
    )
);
```

3. Top pilot after number of flights

To retrieve the name of the captain, first officer or second officer that is assigned to the most flights, we need to join the employee and assignment tables and filter the employees for roles C, FO and SO (Captain, First Officer, Second Officer). Then, we group the results by employee and count the number of assignments for each employee. Finally, we select the first name and last name of the employee with the maximum count.

```
-- Option 1
SELECT (e.first_name ||' '|| e.last_name) as "Name", COUNT(a.id) as "Number of flights"
FROM airlinex_employee as e
JOIN airlinex_assignment as a ON e.id = a.employee_id
WHERE e.role IN ('C', 'FO', 'SO')
GROUP BY e.id
ORDER BY COUNT(a.id) DESC
LIMIT 1;

-- Option 2: More efficient through renunciation of Group By
SELECT (e.first_name ||' '|| e.last_name) as "Name",
    (SELECT COUNT(a2.id) FROM airlinex_assignment a2 WHERE a2.employee_id = e.id) as "Number of flights"
FROM airlinex_employee as e
WHERE e.role IN ('C', 'FO', 'SO')
ORDER BY "Number of flights" DESC
LIMIT 1;
```

4. Captains of top 3 most delayed flights

The query selects the Flight Number and assembles the full name of the Captains for display. The relevant captains are selected by joining the flights table with the assignments and the employees table. The joined table is filtered for Captains only and ordered by delay of the flight in descending manner. As we want to extract the top 3, there is a limit.

```
-- Option 1
SELECT f.number as "Flight number", (e.first_name ||' '|| e.last_name) as " Captains name"
FROM airlinex_flight f
INNER JOIN airlinex_assignment a ON f.number = a.flight_id
INNER JOIN airlinex_employee e on a.employee_id = e.id
WHERE e.role = 'C'
ORDER BY f.delay DESC
LIMIT 3;

-- Option 2: Using a subquery - about the same efficiency
SELECT f.number AS "Flight number", (e.first_name ||' '|| e.last_name) AS " Captains name"
FROM airlinex_flight f
INNER JOIN airlinex_assignment a ON f.number = a.flight_id
INNER JOIN airlinex_employee e ON a.employee_id = e.id
WHERE e.role = 'C' AND f.number IN (
  SELECT number FROM airlinex_flight
  WHERE cancelled = false
  ORDER BY delay DESC
  LIMIT 3
);
```

5. Top captain after number of hours in the air

This query returns the captain with the most hours in the air. The query selects the first and last name of the employee with the most hours in the air based on flight information (airlinex_flight), employee assignments to flights (airlinex_assignment), and

employee information (airlinex_employee). It joins the three tables and filters by a specific role ('C') to only include employees who accumulate hours in the air. The results are grouped by first and last name and ordered by total hours in descending order, with a limit of one row to return the employee with the most hours in the air.

```
-- Option 1
SELECT airlinex_employee.first_name, airlinex_employee.last_name, SUM(EXTRACT(epoch FROM (airlinex_flight.arrival_time - airlinex_flig
FROM airlinex_assignment
JOIN airlinex_flight ON airlinex_flight.number = airlinex_assignment.flight_id
JOIN airlinex_employee ON airlinex_employee.id = airlinex_assignment.employee_id
WHERE airlinex_employee.role = 'C'
GROUP BY airlinex_employee.first_name, airlinex_employee.last_name
ORDER BY total_hours DESC
LIMIT 1;

-- Option 2: Shifting to pkey operations to increase efficiency
SELECT e.first_name, e.last_name, SUM(EXTRACT(epoch FROM (f.arrival_time - f.departure_time)) / 3600.0) AS total_hours
FROM airlinex_employee e
JOIN airlinex_assignment a ON a.employee_id = e.id
JOIN airlinex_flight f ON f.number = a.flight_id
WHERE e.role = 'C'
GROUP BY e.id
ORDER BY total_hours DESC
LIMIT 1;
```

6. & 7.

We decided to use advanced queries for our “normal” and materialized view. See the View section for detailed descriptions and SQL queries.

Bonus: 8. Number of bookings grouped by status

This query returns the number of bookings made for each passenger class (Bronze, Silver, Platinum). It uses a JOIN statement to link the airlinex_passenger and airlinex_booking tables by the passenger_id and id columns, respectively, which represent a foreign key relationship between the tables. The GROUP BY statement is used to group the results by passenger status, while the COUNT function is used to count the number of bookings for each group. Finally, an alias is assigned to the COUNT function to give the column a more descriptive name (num_bookings).

```
-- Option 1
SELECT p.status, COUNT(b.id) AS num_bookings
FROM airlinex_passenger p
JOIN airlinex_booking b ON b.passenger_id = p.id
GROUP BY p.status;
```

Bonus: 9. Aircraft on the ground for each Airport

This query is implemented twice: Once it returns the number of aircraft per airport and once a comma separated list of the aircraft ids per airport.

It filters out flights that have not yet arrived at their destination airports or have departed for other flights after arriving at the current airport. Finally, it groups the remaining flights by the destination_airport_id column and either calculates the number of distinct aircraft or a list of the aircraft’s ids, that satisfy the filtering conditions for each airport .

```
-- Option 1: Number of aircrafts
SELECT f.destination_airport_id AS airport_id, COUNT(DISTINCT f.aircraft_id) AS num_aircrafts_on_ground
FROM airlinex_flight f
WHERE f.arrival_time <= NOW() AND NOT EXISTS (
  SELECT 1 FROM airlinex_flight f2 WHERE f2.aircraft_id = f.aircraft_id AND f2.departure_time > f.arrival_time
)
GROUP BY airport_id;

-- Option 2: List of aircrafts ids
SELECT f.destination_airport_id AS airport_id, STRING_AGG(DISTINCT f.aircraft_id::text, ',' ORDER BY f.aircraft_id::text) AS aircraft_
FROM airlinex_flight f
WHERE f.arrival_time <= NOW() AND NOT EXISTS (
  SELECT 1 FROM airlinex_flight f2 WHERE f2.aircraft_id = f.aircraft_id AND f2.departure_time > f.arrival_time
)
GROUP BY airport_id;
```

NOTE: Use of advanced queries in the application

Not all advanced queries were usable in the application in a meaningful way. However, we incorporated the queries 2, 3 and 5 by adding badges to list records. These badges enrich the table by adding information on which captain flies the most (by hours and number of flights), as well as whether there is an “all time passenger” (someone booked on all flights).