“Flights Booking System” Report

Course: Data Management I, Academic Year: 2023 / 2024

Supervised by: Pr. Karima ECHIHABI

*Safae HAJJOUT, Kawtar LABZAE, Othmane AZOUBI, Mounia BADDOU*

*Mohammed VI Polytechnic University, College of Computing, School of Computer Science*

**Index:**

* ERD Changes (From **um6p-cs-introdb-project2-erd-new.docx** file);
* Schemas and minimum cover (From **um6p-cs-introdb-project2-fds.docx** file);
* The SQL DDL (From **um6p-cs-introdb-project2-ddl.sql** file);
* The SQL DCL (From **um6p-cs-introdb-project2-dcl.sql** file);
* The SQL Data population of databases (From **um6p-cs-introdb-project2-data.sql** file);
* The SQL DML (From **um6p-cs-introdb-project2-dml.sql** file);

This report is a walkthrough on our thought process working on the second deliverable of our project: **“Flights Booking System”**.

# **ERD Changes: *(Approx. 2 hours)***

In terms of ERD changes, we restarted on a fresh slate thinking about some parts of the ERD in light of the remarks previously received. We realized many relationships and entities needed revisiting, as such, here is a rundown of the total changes we made, this is an abbreviation from the **um6p-cs-introdb-project2-erd-new.docx** file.

* + Removed the weak entities **"Child"** and **"Service"** along with their relationships.
  + Split up the full names of **"User"** and **"Passenger"** entities.
  + Introduced the **"Reservation"** entity between **"User"** and **"Ticket."**
  + Excluded the **"transactionID"** attribute from the **"Ticket"** entity.
  + Divided the **"seats"** attribute in the **"Airplane"** entity into multiple class seats.
  + Renamed the **"capacity"** attribute in the **"Airplane"** entity to "maxWeight.
  + Eliminated the **"reservedSeats"** attribute from the **"Flight"** entity.
  + Renamed the ID attributes for consistency and clarity.

Overall, these changes were made in order to streamline our data model, enhance clarity, and improve the representation of relationships between entities in our system.

* In total this part of the deliverable 2 process took about a class session’s worth of work in the presence of Professor Karima Echihabi and the TA’s, followed by a few group conversations during the week.

# 

# 

# 

# **Schemas *(Approx. 4 hours)*:**

After completing the ERD, defining relations, and subsequently normalizing the database, our attention shifted towards refining functional dependencies. This iterative process demanded significant effort and careful consideration as we fine-tuned the database schema. The final result was a well-organized and normalized database structure. While a few areas for improvement were identified, overall, the refined functional dependencies contributed to a more efficient and streamlined schema.The final schema looked like the following (from the **um6p-cs-introdb-project2-fds.docx** file):

**User (**uemail: *String*, ufirstName: *String*, ulastName: *String*, ubirthDate: *Date*, passwordHash: *String***)**

**Passenger (**passportID: *String*, cin: *String*, pbirthDate: *Date*,phoneNumber: *String*, pfirstName: *String*, plastName: *String*, fcid: *String***)**

**PassengerCard (**fcid: *String*, fctype: *String***)**

*FK fcid FROM Passenger, FK fctype FROM FidelityCard*

**Reservation (**rid: *String*, rDate: *DateTime*, confirmDate: *DateTime*, uemail: *String***)**

*FK uemail FROM User*

**Ticket (**tid: *String*, seatNumber: *Integer*, class: *String*, price: *Float*, passportID: *String*, rid: *String*, fid: *String,* fctype: *String***)**

*FK passportID FROM Passenger, FK rid FROM Reservation, FK fid FROM Flight, FK fctype FROM FidelityCard*

**FidelityCard (**fctype: *String*, reduction: *Integer***)**

**Flight (** fid: *String*, arrivalTime: *DateTime*, departureTime: *DateTime*, destination: *String*, departure: *String*, registrationNumber: *Integer***)**

*FK registrationNumber FROM Airplane*

**AirplaneModel (**model: *String*, economySeats: *Integer*, premiumEconomySeats: *Integer*, businessClassSeats : *Integer*, firstClassSeats: *Integer*, maxWeight: *Integer***)**

**Airplane (**registrationNumber: *Integer*, model: *String*, airline: *String***)**

*FK model FROM AirplaneModel*

**Checks (**uemail: *String*, fid: *String***)**

*FK uemail FROM User, FK fid FROM Flight*

# **Minimal Cover *(Approx. 1 hour)*:**

The minimal cover was easily deduced from the schemas and functional dependencies, as such it did not take us as much time.

**Reservation:**

* rid —> rDate, confirmDate, uemail

**User:**

* uemail —> ufirstName, ulastName, ubirthDate, passwordHash

**Passenger:**

* passportID —> pbirthDate, passportID, phoneNumber, pfirstName, plastName, fctype, fcid
* phoneNumber —> pbirthDate, passportID, phoneNumber, pfirstName, plastName, fctype, fcid
* cin —> pbirthDate, passportID, phoneNumber, pfirstName, plastName, fctype, fcid
* fcid —> fctype

**Ticket:**

* tid —> seatNumber, class, price, passportID, rid, fid, fctype
* (fid, passportID) —> seatNumber, class, price, passportID, rid, fid, fctype
* (fid, seatNumber) —> seatNumber, class, price, passportID, rid, fid, fctype

**FidelityCard:**

* fctype —> reduction

**Flight:**

* fid —> arrivalTime, departureTime, destination, departure, registrationNumber

**Airplane:**

* registrationNumber —> economySeats, premiumEconomySeats, businessClassSeats, firstClassSeats, model, maxWeight, airline
* model —> economySeats, premiumEconomySeats, businessClassSeats, firstClassSeats, maxweight

**Checks:**

* (email, fid) —> email, fid

# 

# **The SQL DDL: *(Approx. 3 hours)***

As seen in the document **um6p-cs-introdb-project2-ddl.sql,** we opted in this part to create tables that represent the schemas found in the part above. The foreign key constraints were quite the challenge to debug after the initial code was written.

The code used respects the norms of MySQL, and the tables created are the following:

* User\_;
* AirplaneModel;
* Airplane;
* Reservation;
* Flight;
* FidelityCard;
* PassengerCard;
* Passenger;
* Ticket;
* Checks

In this part, indexes and triggers were also added to the code.

For the indexes:

* uchecks *on Checks(email)*;
* fticket *on Ticket(fid)*;
* ureservation *on Reservation(email)*;
* dep\_dest *on Flight(departure, destination, departureTime)*;
* airline\_airplane *on Airplane(airline)*

And for the triggers:

* checkValidReservation;
* user\_age\_check

These DDL components were proved functional in our “.sql” file, and proved logged in the “.txt” file.

# 

# 

# **The SQL DCL: *(Approx. 10 hours)***

At this phase, as we thought the project deliverable was nearly done,was definitely an obstacle. Perhaps we may have provided much more than what was asked, as we automated the process of user account addition with the role and necessary grants on a user’s addition to the database (to the **“User\_”** entity), and in a similar manner applied the same logic for the airlines.

In total there are three main roles within our DCL:

* The **‘admin’** role:

the application owner has read-write access to all tables in the database.

* The **‘user**’ role:

the users logged in to the database have read access to the Flights table and individual access to their own reservations and registered passengers (as table views).

* The **‘airline’** role:

the airlines partnering with the application have read-write access to the Airplane and AirplaneModel entities.

The first difficulty was learning about how Procedures and Security Access work in mysql in order to perform exactly what we wanted to achieve: automated user account creation and deletion (with grants and revokes). We discovered this could not entirely be feasible only in mysql because of security issues linked to calling such operations within a trigger.

However we elected to keep the Procedures, they work using dynamic sql, simply meaning that the commands are written in string format for extra customization *(e.g: assigning each user view a name according to the user owning it)*.

The second difficulty was related to debugging our code as we made our initial dcl logs, this took quite some time but thanks to work being shared it was accomplished readily.

# 

# 

# **SQL Data Population *(Approx. 8 hours)*:**

After encountering difficulties with the provided websites for data population, we chose a simpler path by employing Python to generate the needed data.

Drawing from Kaggle datasets, we compiled lists of airlines, airplane models and airport names. Subsequently, we utilized the Faker library and custom generator functions to create phone numbers, passports, names, IDs and emails.

We systematically filled our database with Python-generated entities. This sequence, commencing with **"airplanemodel"** and concluding with **"ticket"**, was meticulously orchestrated to align with foreign key dependencies. This method not only facilitated the establishment of entity relationships but also ensured a smooth and error-free database population.

After creating the lists, we utilized the CSV library to convert our data into CSV files for quick portability. Finally, we developed a **"CSV"** to **"SQL Insert"** converter to generate all the INSERT QUERIES found in the **"um6p-cs-introdb-project2-data.sql"** file.

The source code for the generator is included in the **"DataPopulatingBookingSystem.zip"** within the Git repository. To run the code, ensure you have the following dependencies installed:

* Python 3 (We operated on version 3.11).
* Install the Faker library using the following command: pip install Faker.
* All other necessary files are included in the ZIP archive.

# 

# **The SQL DML: *(Approx. 10 hours)***

a brief explanation for each of the queries in the dml file (**um6p-cs-introdb-project2-dml.sql)**:

***1. Retrieve upcoming flights with available seats in economy class:***

- This query selects upcoming flights along with the count of booked seats and available seats in the economy class.

***2. Retrieve top 5 destinations based on the number of tickets:***

- This query identifies the top 5 destinations with the highest ticket counts by joining the Flight and Ticket tables.

***3. Retrieve users who have made reservations for more than one destination:***

- This query finds users who have made reservations for more than one destination by joining the User, Reservation, Ticket, and Flight tables.

***4. Retrieve Airlines that have the same departure and destination:***

- This query uses UNION to retrieve flights with the specified departure and destination for two different airlines.

***5. Find Passenger Informations for a specific Reservation:***

- This query retrieves detailed passenger information for a specific reservation ID.

***6. Research-bar-like query for Flight:***

- This query performs a search-bar-like query for flights based on the destination and departure locations.

***7. Informations about Flight from Airline and Model:***

- This query retrieves flight information along with the associated airline and airplane model.

***8. Retrieve 5 flights with the biggest luggage weight for a passenger:***

- This query selects flights with the highest luggage weight, limiting the result to 5.

***9. Total Price of a given reservation:***

- This query calculates the total price of a given reservation by summing the prices of associated tickets.

***10. Passengers in a specific flight:***

- This query retrieves passenger information for a specific flight.

***11. 2 Inserts:***

- These are two INSERT statements adding a new user and a new record to the checks table.

***12. 2 Updates:***

- These are two UPDATE statements modifying the first name and password hash for a specific user.

***13. 2 Deletions:***

- These are two DELETE statements removing a user and a passenger based on their respective identifiers.

NB:

* Each query serves a specific purpose, retrieving or modifying information based on the specified conditions.
* Each query has used a combination of what was demanded, so keeping track of the

commands asked for in the project requirements document was hard to do . However, the instructions were respected