DATABASES

ADIT

Lab Compendium – BrianAir Project

Table of Contents

Institutionen för datavetenskap (IDA), Linköpings universitet

Project: BrianAir Database

Objectives

The objectives of this project are to learn how to implement a database, get some hands-on experience of SQL stored procedures and more experience in writing SQL queries.

Background Reading

To work on this project, you need to be familiar with EER-modelling, translation of an EER diagram into a relational database schema, SQL queries, stored procedures, and transactions. Hence, the project covers many aspects of the whole course.

Estimated Project Time

This is a large project, reaching over six lab sessions. Note, however, that these lab sessions are expected to cover only the SQL-based implementation part of your project. The EER modelling and translation into a relational database schema (i.e. the first part of the project) needs to be done before the scheduled lab sessions. Check the course Website or ask your lab assistant if you have further questions about the schedule, deadlines, etc.

Exercise Background

A friend of yours, called Brian, has decided to get into the travel business and has just started up a low-price airline company called BrianAir. He is now designing the JavaScript-based booking Website and has realized that he needs a database in the backend. Knowing that you are taking a database course, he has asked you to design and implement the flight and booking database. In other words, you are supposed to plan and set up the database that he will use for his company. You think the company is a terrible idea but, as a good friend, you have agreed to help him.

The main idea of the Website is that a customer should be able to search for flights and create bookings on these flights. The booking procedure should be fairly simple where the customer creates a reservation for a number of people on a flight and get a reservation number. The customer should then be able to add passengers to the reservation as well as some contact details. Finally, the customer should be able to add payment details and pay for the reservation and, thereby, ensure that the passengers have seats on the chosen plane. When this happens, the reservation is said to become a booking. All of this can be done within one session at the Website, but the customer should also be able to create a reservation, check the price, and finish the booking procedure at a later date using the given reservation number.

Part 1, Designing the Database

1. (at home) Draw an EER diagram for the BrianAir database. Your friend has not really thought this through and will let you decide how to design the database as long as the following requirements and specifications are met. Note that you must read through the entire project description (including questions 2-10) to get a proper understanding of what the database should be able to handle and what attributes are required for the entities.

2) BrianAir uses only one type of airplane that fits 40 passengers. You do *not* need to model different airplane types.

3) BrianAir only flies between the airports Lillby and Smallville (and back) but they will soon expand. Hence, the database should be constructed such that it is easy to add new destinations and routes later.

4) BrianAir operates on a strict weekly schedule, meaning that this schedule is exactly the same every week of the year (same flights, same times, no exceptions for holidays). The weekly schedule is valid for one year and may be changed on January 1st for the next year.

a. Reservations should be possible to make on any route added to the system. A single reservation should only be for one route, i.e. one direct flight, but may contain several passengers. The reservation is confirmed by issuing a unique, unguessable reservation number that is needed to finish the booking (i.e. paying) at a later date.

5) The flight pricing depends on:

6) the route, which is characterized by its departure and destination location, where each such route has a specific price, which may be different for different routes;

7) the day of the week (i.e., for each of the seven weekdays, there is a pricing factor that is independent of the routes; e.g., factor 4.7 on Fridays and Sundays, factor 1 on Tuesdays);

8) the number of already confirmed/booked passengers on the flight (the more passengers are booked, the more expensive the flight becomes);

9) the profit that BrianAir wants to make, which is the same for all flights.

The total price is thus calculated as follows:

$$TotalPrice = RoutePrice_{to,from} \cdot WeekdayFactor_{day} \cdot (\#BookedPassengers_{flight} + 1)/40 \cdot ProfitFactor$$

The route prices and the two pricing factors (weekday factor and profit factor) can change when the schedule changes, i.e., they may be different each year.

10) The actual price for a booking is calculated at payment time and should be saved in the database for future reference. Moreover, each person in one booking has the same seat price (calculated as the first person in the booking).

11) Information about each passenger participating in any BrianAir flight must be stored in the database as detailed by the flight safety standard. This information includes passport number as well as the passenger's full name.

12) Each booking should contain one passenger that is the *contact* of that booking and must supply phone number and e-mail address. Note that this contact has to be added before the booking is paid.

13) Only credit cards can be used to pay for the flights and the necessary credit card information, such as card number and credit card holder, should be stored. It is possible that the person who pays for the flight (i.e., the credit card holder) is not one of the passengers.

14) The payment of the booking is confirmed by issuing a unique, unguessable, ticket number per passenger, which the passengers need to bring to the airport instead of paper tickets.

15) Overbookings are not allowed but overreservations are. This means that the sum of the number of passengers in different ongoing reservations may exceed the number of the still unbooked seats, but at the point when a reservation is paid for, and thus becomes an actual booking, there must still be enough unbooked seats to fit all passengers of that reservation. If there are not enough unbooked seats when the payment for a reservation happens, then the whole booking-process for the reservation is aborted and the reservation should be removed from the system.

Other requirements are up to your assumptions. State them in the EER diagram.

Some hints and further requirements:

A simple solution would be to model flights as an entity type with attributes such as cities of departure and arrival, day of the year, time of departure, etc. However, this is not acceptable because the resulting table may contain a lot of duplicated information. Hence, we require you to include the following types of entities instead (note that you will need additional entities beside these ones):

16) *Route*, which contains all the routes that BrianAir flies. Each such route is characterized by the cities of departure and arrival.

17) *Weekly flight*, which contains all the weekly-recurring flights that are part of the weekly schedule for a given year. Such a weekly flight is characterized by a unique flight number, the route, the year, the day of the week, and the time of departure.

18) *Flight,* which contains all the 52 week-specific instantiations of each of the weekly flights. Such a flight is characterized by a unique ID, the specific week in which it happens, and the corresponding weekly flight that it instantiates.

Handing In

For the first hand-in of the project (Assignment 4a), you need to hand in an EER diagram for the BrianAir database. Once this diagram has been approved by your lab assistant, the second hand-in (Assignment 4b) will be a relational database schema that is the result of translating the approved version of your EER diagram. This schema needs to be approved before you start the implementation in part 2 of the project.

Part 2, Implementation of the Database

Once your EER diagram and relational database schema are approved, it is time for the implementation. Your friend has no opinion on how this should be done, as long as the database contains a set of stored procedures that works as an interface to your friend's JavaScript frontend.

To avoid unnecessary work, **read through all the questions, hints, etc., before starting with the implementation.**

Hint

For this assignment it is strongly recommended that you write your SQL code in a script to allow you to easily recreate the database from scratch. This is simply done by writing all SQL statements sequentially in a text file and save it in your home directory. The script can then be run by using the SQL command "*SOURCE name_of_text_file.txt*". In the beginning of the file, you should drop all tables and procedures of the database before you "recreate" them (using the commands "*DROP TABLE IF EXISTS table_name" resp. "DROP PROCEDURE IF EXISTS proc_name")*.

Testing and Expected Output

The assignments page of the course Website provides links to download several SQL scripts that allow you to test your implementation, to see that it gives the correct output given different input. These scripts also contain additional information about the input-output relation, such as error messages etc., that are not given in the questions below. **Note that your implementation should give the correct output for all scripts before you hand in.** Also note that these scripts are by no means complete and it is not enough to only give the correct output without using the correct way of "finding" it.

Questions:

2. Create the tables (with primary keys and foreign keys) in your database using CREATE TABLE statements, and if necessary ALTER TABLE statements. Once you are done, the database should have the same structure as shown in your relational database schema. Also, read up on how attributes can be automatically incremented and implement where appropriate.

For the database to properly work with the frontend, the attributes should be of the following types:

Attribute	Type
Year	INTEGER
Weekday	VARCHAR(10)
Airport code	VARCHAR(3)
Airport name	VARCHAR(30)
Country	VARCHAR(30)
Departure time	TIME
Profit factor	DOUBLE
Route price	DOUBLE
Weekday factor	DOUBLE
Flight ID	INTEGER
Reservation number	INTEGER
Name	VARCHAR(30)
Passport number	INTEGER

Email	VARCHAR(30)
Phone number	BIGINT
Creditcard number	BIGINT

3. Write **procedures** for populating the database with flights, etc. These procedures will work as an interface towards the frontend.
   a) Inserting a year via the following procedure call:
      addYear(year, factor);
      where *factor* is the profit factor for the given *year*
   b) Inserting a weekday with the weekday factor for a particular year:
      addDay(year, weekday, factor);
   c) Inserting a potential destination airport, including its city and country:
      addDestination(airport_code, name, country);
   d) Inserting a route from a departure airport to an arrival airport, including a year-specific price for flights on that route:
      addRoute(departure_airport_code, arrival_airport_code, year, routeprice);
   e) Inserting a weekly flight on a particular weekday and departure time for a particular year, including the corresponding 52 flights for the given year:
      addFlight(departure_airport_code, arrival_airport_code, year, weekday, departure_time);
      This procedure should not only add the information for the weekly flight but also for each of the 52 flights that are the concrete instances of the weekly flight for each of the 52 weeks of the specified year.

4. Write two helper **functions** that do some of the calculations necessary for the booking procedure:
   a) Calculating the number of available seats for a certain flight:
      calculateFreeSeats(flight_ID);
      The output of this function should be the number (i.e., an integer) of free seats on the flight with the given ID, where seats are considered as free if they have not yet been paid for; i.e., reservations for which there is no payment yet can be ignored when counting the free seats.
   b) Calculating the price of the next seat on a flight:
      calculatePrice(flight_ID);
      The output of this function is the price (i.e., a double) of the next seat on the flight with the given ID, calculated as specified in point 1e above.

5. Create a **trigger** that issues a unique, unguessable ticket number (of type integer) for every passenger on a reservation once it is paid. An appropriate SQL function to obtain an unguessable number is *rand()*.

6. Now, it is time to write the stored procedures necessary for creating and handling a reservation from the frontend. In addition to the input and output detailed below, see the test files for appropriate error messages to return in case of unsuccessful payments etc.
   a) Creating a reservation on a specific flight:
      addReservation(departure_airport_code, arrival_airport_code, year, week, day, time, number_of_passengers, output_reservation_nr);
      where *number_of_passengers* is the number of passengers the reservation is for (and only used to check that enough unpaid seats are available at his point when the reservation is created) and *output_reservation_nr* is an output

variable to which the procedure should assign the reservation number that it has created for the new reservation.

    b) Adding a passenger to a reservation:

       a*ddPassenger(reservation_nr, passport_number, name);*

    c) Specifing the contact passenger for a reservation:

       addContact(reservation_nr, passport_number, email, phone);

    where the contact must have been added already as a passenger to the reservation.

    d) Adding a payment and, thereby, turning a reservation into a booking:

       addPayment (reservation_nr, cardholder_name, credit_card_number);

    This procedure should, if the reservation has a contact and there are enough unpaid seats on the plane, add payment information to the reservation and save the amount to be drawn from the credit card in the database. If the conditions above are not fulfilled, a relevant error message should be shown (as indicated in the aforementioned test scripts).

7. Create a view called *allFlights* that contains all flights in your database with the following information: *departure_city_name, destination_city_name, departure_time, departure_day, departure_week, departure_year, nr_of_free_seats, current_price_per_seat.* See the testcode for an example of how it can look like.

8. Answer the following theoretical questions:

    a) How can you protect the credit card information in the database from hackers?

19) Give three advantages of using stored procedures in the database (and, thereby, execute them on the server) instead of writing the same functions in the frontend of the system (for example, in JavaScript on a Web page)?

In the next two questions you will see how the MariaDB database server handles concurrency. By default, MariaDB commits every single SQL statement as a transaction, which means that, as soon as a statement is executed, its effects are also committed to the database. However, MariaDB also supports multiple statements to be bundled up in a single transaction, as described during the lecture.

To test this, open two terminals on the same computer, both connecting to the same database. Then, use the statement **START TRANSACTION** to start a transaction that is finished by either **COMMIT** or **ROLLBACK**. Note that, once the transaction is finished, MariaDB goes back to the default setting of committing every statement, unless a new transaction is started. The main statements to handle transactions are *START TRANSACTION, COMMIT*, *ROLLBACK*, *LOCK TABLES, UNLOCK TABLES*, *SAVEPOINT* and *SELECT...FOR UPDATE*. Check the MariaDB documentation for further description and examples of these statements.

Play around with the transaction-related statements above in the two terminals and try to anticipate what will happen. One important difference between MariaDB and what is described during the lecture is that MariaDB has the transaction isolation level *Repeatable-Read* as default to maintain consistency of the database for the transactions. This means that MariaDB implicitly adds a write lock on every tuple that is inserted, or updated, in any table. It also adds read locks on the tuples when they are used, for example in IF-statements. For further motivation of this, how it works, and how it can be changed, check the MariaDB documentation regarding *transaction isolation level*.

9. Open two SQL sessions in two terminals. We call one of them A and the other one B. Write **START TRANSACTION;** in both terminals.

   a) In session A, add a new reservation.

   b) Is this reservation visible in session B? Why? Why not?

   c) What happens if you try to modify the reservation from A in B? Explain what happens and **why this happens** and how this relates to the concept of isolation of transactions.

10. Is your BrianAir implementation safe when handling multiple concurrent transactions? Let two customers try to simultaneously book more seats than what are available on a flight and see what happens. This is tested by executing the test scripts available on the course Website using two different SQL sessions. **Note that you should not use explicit transaction control unless this is your solution on 10c.**

   a) Did overbooking occur when the scripts were executed? If so, why? If not, why not?

   b) Can an overbooking theoretically occur? If an overbooking is possible, in what order must the lines of code in your procedures/functions be executed.

20) Try to make the theoretical case occur in reality by simulating that multiple sessions call the procedure at the same time. To specify the order in which the lines of code are executed use the SQL statement *SELECT sleep(5);* which makes the session sleep for 5 seconds. Note that it is not always possible to make the theoretical case occur; if not, motivate why.

21) Modify the **test scripts** so that overbookings are no longer possible using (some of) the commands START TRANSACTION, COMMIT, LOCK TABLES, UNLOCK TABLES, ROLLBACK, SAVEPOINT, and SELECT...FOR UPDATE. Motivate why your solution solves the issue, and test that this also is the case using the sleep implemented in 10c. Note that it is not okay if one of the sessions ends up in a deadlock. Also, try to hold locks on the common resources for as short time as possible to allow multiple sessions to be active at the same time.

> *Note that depending on how you have implemented the project it may be very hard to block the overbooking due to how transactions and locks are implemented in MariaDB. If you have a good idea of how it should be solved but are stuck on getting the SQL statements right, talk to your lab assistant and he/she might help you get it right or allow you to hand in the exercise with pseudocode and a theoretical explanation.*

Handing in

For the final hand-in of the project the following should be handed in:

a) *Approved* EER diagram and relational database schema. If minor changes have been made during the implementation, these should be reflected here.

b) Answers (including SQL code) for questions 2 to 10. Note that the code should have been tested on the available test scripts and give the correct output *before* you hand in.

- Identify one case where a secondary index would be useful. Design the index, describe and motivate your design. (Do not implement this.)

Note that, after your lab assistant has approved your final report, you also need to email this approved version of your report to Urkund, as detailed on the course Website. To this end, put everything together into one big .pdf or .txt file that you attach to the email.