CS2102: Database System

P03: Triggers + Routines AY2023/24 Sem 2: Week 12

Objective

The objective of this team project is for you to apply what you have learned in class to design and develop a database application using PostgreSQL. The project is to be done in teams of four students. The project consists of the following four tasks:

(P01) Design an ER data model for the application. Your ER model should capture as many of the application's requirements as possible.

(P02) Translate your ER data model into a relational database schema. Your relational schema should capture as many of the application's constraints as possible.

(P03) Implement an SQL or PL/pgSQL functions/procedures for each of the functionalities listed in Application Functionalities.

(P03) Implement triggers as required by the specification.

Current Status

Now that we have a schema `DDL.ddl` and the constraints analysis `P03.docx`, we want to fully enforce the constraints and implement some functionalities. Please study the schema and constraints analysis to understand the schema better. You are guaranteed that all data used for testing will satisfy the schema.

In this last part of the project, you are allowed to use any feature of PostgreSQL as long as there are no additional libraries/frameworks used. For *DATE*, please read the following documentation (`https://www.postgresql.org/docs/current/functions-datetime.html`) to understand PostgreSQL *DATE* type.

Files

- `DDL.sql`: Schema from P02.
- `P03.docx`: Constraints analysis from P02.
- `P03.sql`: Template for routines.
- `P03.pdf`: This file.

Triggers

From the constraints analysis, there are several constraints that cannot be enforced purely by SQL DDL. In particular, those constraints require data either from multiple rows or from multiple tables (*or worse, both*). For each constraints below, you only need to consider `INSERT` trigger. You do **NOT** need to consider `DELETE` or `UPDATE` triggers, *unless* they are specifically mentioned by the question.

For each constraints below, you may use multiple triggers to enforce it. However, please be careful on the order of the triggers to ensure that the constraints are properly enforced. Your trigger should work from a database with all the data satisfying all the constraints. Furthermore, your trigger should work for any sequence of valid `INSERT` regardless of whether a procedure/transaction is used or not.

1. Drivers cannot be double-booked.
   - On insertion into `Hires`.

     By double-booked, we meant that driver (*identified by* `eid`) is hired for two different bookings (*identified by* `bid`) on overlapping hiring dates. The hiring dates is computed from `fromdate` to `todate` (*inclusive of both*).

     Consider the following examples on overlapping. We will use the notation `[fromdate, todate]` to indicate the hiring dates from `fromdate` to `todate` (*inclusive of both*).
     - `['2024-01-03', '2024-01-05']` and `['2024-01-05', '2024-01-06']` overlap.
     - `['2024-01-03', '2024-01-05']` and `['2024-01-04', '2024-01-06']` overlap.
     - `['2024-01-03', '2024-01-05']` and `['2024-01-02', '2024-01-06']` overlap.
     - `['2024-01-03', '2024-01-05']` and `['2024-01-06', '2024-01-07']` do not overlap.

     Note the special case when `todate` of one booking is equal to `fromdate` of another booking.

   Clarifications:
   Our definition using *inclusive of both* is slightly different from P00.pdf. Changes in specification is unfortunately common in larger real-world project and we are mimicking real-world project.

2. Cars (*i.e.,* `CarDetails`) cannot be double-booked.
   - On insertion into `Assigns`.

     This is similar to Trigger 1 but for `CarDetails`.

3. During handover, the employee must be located in the same location the booking is for.
   - Triggers on insertion into `Handover`.

     `Handover` relates `Bookings` (*identified by* `bid`) and `Employees` (*identified by* `eid`). The employee must work for the same location (*identified by* `zip`) as the location (*identified by* `zip`) the booking is for.

4. The car (*i.e.,* `CarDetails`) assigned to the booking must be for the car models for the booking.
    - Triggers on insertion into `Assigns`.

      When a customer initiates a booking (*identified by* `bid`), the customer selects a car model (*identified by* `(brand, model)`). When a car (*i.e.,* `CarDetails` *identified by* `plate`) is assigned to the booking after the booking is initiated, it must have the same `(brand, model)` as required by the booking.

5. The car (*i.e.,* `CarDetails`) assigned to the booking must be parked in the same location as the booking is for.
    - Triggers on insertion into `Assigns`.

      When a customer initiates a booking (*identified by* `bid`), the customer selects a location (*identified by* `zip`). When a car (*i.e.,* `CarDetails` *identified by* `plate`) is assigned to the booking after the booking is initiated, it must be parked the same location (*identified by* `zip`) as required by the booking.

6. Drivers must be hired within the start date and end date of a booking.
    - Triggers on insertion into `Hires`.

      In effect, the hiring date of drivers denoted by `[fromdate, todate]` must *overlap* with the booking schedule denoted by `[sdate, edate]`. Note that `edate` is derived. Fortunately, you may perform `DATE + INTEGER` to get another `DATE`.

Routines

Routines are old terminology that encapsulates both functions and procedures. Also note that the distinction between functions are procedures are getting blurred as well. In what follows, you are **NOT** allowed to change the template as they will be automatically graded. Among other things, you are **NOT** allowed to change the following. ***The only thing you can change is the name of the parameters.***

- Change the **name** of the routines.
- Change the **parameters** (*e.g.*, add/remove parameter, change type, *etc*).
    - You may change the **name** of the parameters.
- Change the **language** (*e.g.*, from `sql` to `plpgsql` or vice versa).

You may assume that the input format will be valid, but the input data may not conform to all constraints (*e.g.*, failed trigger check or failed `CHECK` from DDL). In the case of failure, the routine should **NOT** cause changes to the database. You may raise an exception explicitly or you may simply allow it to fail silently.

If an **array** (`https://www.postgresql.org/docs/current/arrays.html`) is used, you should follow PostgreSQL default behavior of starting the index from 1 (instead of the usual 0 typical in other programming languages). You may also read up on ways of looping through arrays (https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-FOREACH-ARRAY).

You may add additional functions and/or procedures to help your routines. Make sure that name of the added routines do not conflict with the name of the required routines. To simplify your task, your routines may invoke other routines specified in this document. You may also choose to create `VIEW`.

It is important to adhere to these instructions since automated testing will be used to evaluate your implementation.

Procedures

The routines in this subsection do not have return values and should be implemented as PostgreSQL procedures. The routines must be successful on valid inputs even in the presence of triggers above. You may need to use deferred triggers in cases where it applies. In the procedure failed, the database should be left unchanged.

1. Write a procedure to add employees. Note that employees may also be drivers depending on whether the `pdvl` is `NULL` or non-`NULL`.

```
CREATE OR REPLACE PROCEDURE add_employees (
  eids INT[], enames TEXT[], ephones INT[], zips INT[], pdvls TEXT[]
) AS $$
-- add declarations here
BEGIN
  -- your code here
END;
$$ LANGUAGE plpgsql;
```

   - Array assumptions:
     - You may assume that all arrays have the same number of elements (*but maybe 0*).
     - You may assume that the values on all arrays for the same index is for the same employee (*e.g.*, `eids[3]`, `enames[3]`, `ephones[3]`, `zips[3]`, and `pdvls[3]` are information for the same employee).
   - `eids` is an array of *INT* corresponding to the employee `eid`.
   - `enames` is an array of *TEXT* corresponding to the employee `ename`.
   - `ephones` is an array of *INT* corresponding to the employee `ephone`.
   - `zips` is an array of *INT* corresponding to the employee `zip`.
   - `pdvls` is an array of *TEXT* corresponding to the driver `pdvl`.
     - If the corresponding `pdvl` is `NULL`, the employee is **NOT** a driver.
     - Otherwise, the employee is a driver (*insert into* `Driver`).

2. Write a procedure to add a car model with all the corresponding car details. Recap that a single car model may have 0 or more car details.

```
CREATE OR REPLACE PROCEDURE add_car (
  brand    TEXT    , model   TEXT    , capacity INT,
  deposit NUMERIC, daily   NUMERIC,
  plates  TEXT[] , colors TEXT[] , pyears   INT[], zips INT[]
) AS $$
-- add declarations here
BEGIN
  -- your code here
END;
$$ LANGUAGE plpgsql;
```

- Array assumptions:
  - You may assume that all arrays have the same number of elements (*but maybe 0*).
  - You may assume that the values on all arrays for the same index is for the same car details (*similar to Procedure 1 array assumption but for car details*).
- `brand` is a *TEXT* corresponding to car model `brand`.
- `model` is a *TEXT* corresponding to car model `model`.
- `capacity` is an *INT* corresponding to car model `capacity`.
- `deposit` is an *INT* corresponding to car model `deposit`.
- `daily` is an *INT* corresponding to car model `daily`.
- `plates` is an array of *TEXT* corresponding to the car details `plate`.
- `colors` is an array of *TEXT* corresponding to the car details `color`.
- `pyears` is an array of *INT* corresponding to the car details `pyear`.
- `zips` is an array of *TEXT* corresponding to the car details `zip`.

Clarifications:
From `https://canvas.nus.edu.sg/courses/52825/discussion_topics/188067`.
You do not have to account for the case where a car model already exists. If the insertion failed, then the entire procedure should fail.

3. Write a procedure to handle the return of a car. What is needed from this procedure are

   (a) the computation of `cost` using the formula `(daily * days) - deposit`.

   (b) retrieval of `ccnum` from the `ccnum` used to initiate the booking ~~if it is not given (*i.e.*, the argument corresponding to this is `NULL`) and it is needed (*i.e.*, the `cost` is positive).~~

```sql
CREATE OR REPLACE PROCEDURE return_car (
  bid INT, eid INT
) AS $$
-- add declarations here
BEGIN
  -- your code here
END;
$$ LANGUAGE plpgsql;
```

   - `bid` is an *INT* corresponding to booking `bid`.
   - `eid` is an *INT* corresponding to employee `eid`.
   - If a car is returned not using this procedure, we assume that the `cost` is manually computed correctly.

Clarifications:
By returning a car, we meant that an entry should be inserted into the `Returned` table with the correct values. Additionally, condition (b) is simplified there is a problem with the template. In particular, there is no input for `ccnum`. `ccnum` should always be added regardless as it is allowed by the schema.

4. Write a procedure to auto assign cars to bookings.

   We will use the following rule to perform the auto assignment to ensure *fairness*.

   (a) Find all bookings without assigned car details.

   (b) Sort the bookings found in Step 4a in ascending order of `bid` (*i.e.*, smallest on top).

   (c) Sort all car details in ascending order of `plate`.

   (d) Starting from the smallest booking `bid`, assign the car with the smallest `plate` that satisfies its booking requirement.

   Recap the booking requirement:

   - The brand and model match.
   - The location (*identified by* `zip`) match.
   - The car is not double booked.
     – Note that we may auto assign the same car multiple times as we assume the car will be returned on time.

```sql
CREATE OR REPLACE PROCEDURE auto_assign() AS $$
-- add declarations here
BEGIN
  -- your code here
END;
$$ LANGUAGE plpgsql;  -- the template at P03.sql is already correct
```

Functions

The routines in this subsection have return values and should be implemented as PostgreSQL functions. The routines must be successful on valid inputs.

1. Write a function to compute *revenue* from the given `sdate` to the given `edate` (*inclusive of both*). In other words, in the range `[sdate, edate]`.

   The revenue is computed as follows:

   (a) For each booking `B` that are assigned a car details (*i.e.*, ignore bookings that are not assigned any car details) such that `[B.sdate, B.edate]` overlaps with `[sdate, edate ]`, we compute the revenue as `daily * days`.

   (b) For each driver `D` hired such that `[D.fromdate, D.todate]` overlaps with `[sdate, edate]`, we compute the revenue as `(D.todate - D.fromdate + 1) * 10`.
   - Note the `+1` due to the date being inclusive of both ends.

   (c) For each distinct car details `C` such that the car is assigned to a booking `B` where `[B.sdate, B.edate]` overlaps with `[sdate, edate]`, we *subtract* 100 as a cost.

   The total revenue is then given as (a) + (b) - (c) where (a), (b), and (c) are computed as non-negative values.

   For instance, if within a given date range we have
   - a total of 5 bookings where each booking is for 3 days and the daily rate of 200,
   - 3 drivers hired where each driver is hired for 2 days, and
   - 2 distinct cars used for the 5 bookings above.

   Changes: Computation for (b) is corrected.
   The components are (a) `5 * 3 * 200 = 3000`, (b) `3 * 2 * 10 = 60`, and (c) `2 * 100 = 200`. The total revenue is then (a) + (b) - (c) = `3000 + 60 - 200 = 2860`.

```sql
CREATE OR REPLACE FUNCTION compute_revenue (
  sdate DATE, edate DATE
) RETURNS NUMERIC AS $$
  -- your code here
$$ LANGUAGE plpgsql;
```

- `sdate` is a *DATE* corresponding to `sdate` in the range `[sdate, edate]` from the description.
- `edate` is a *DATE* corresponding to `edate` in the range `[sdate, edate]` from the description.

Clarifications:
The computation of revenue is logically as follows when given `[sdate, edate]`:

(a) Find all Bookings.

(b) Keep only Bookings B where `[B.sdate, B.edate]` overlaps with `sdate, edate`.

(c) Compute the revenue for all Bookings B that overlaps *regardless of the actual* `B.sdate` *and* `B.edate`.

(d) Find all Drivers.

(e) Keep only Drivers D where `[D.fromdate, D.todate]` overlaps with `sdate, edate`.

(f) Compute the revenue for all Drivers D that overlaps *regardless of the actual* `D.fromdate` *and* `D.todate`.

(g) Find all *distinct* Car Details.

(h) Keep only *distinct* Car Details C where the car is assigned to a Booking B where `[B.sdate, B.edate]` overlaps with `sdate, edate`.

(i) Compute the cost for all Car Details C where the car is assigned to a Booking B where `[B.sdate, B.edate]` overlaps *regardless of the actual* `B.sdate` *and* `B.edate`.

2. Write a function to return *up to* the top `n` performing locations from the given `sdate` to the given `edate` (*inclusive of both*) in ascending order of rank (*i.e.*, rank 1 [*if any*] on top). In other words, in the range `[sdate, edate]`.

We rank the performance of a location `L` by the revenue generated by the location. Please see Function 1 on the computation of the revenue and we make the following selection for each part of revenue computation:

(a) Select only bookings for the location `L`.

(b) Select only drivers worked at the location `L`.

(c) Select only car details parked at the location `L`.

We then find the top `n` performing location by using "1334" ranking (`https://en.wikipedia.org/wiki/Ranking#Modified_competition_ranking_(%221334%22_ranking)`). In 1334 ranking, each location ranking is equal to the number of locations ranked equal to it or above it.

It is easier to illustrate this with an example. The table below shows the 1334 ranking.

Location	**Revenue**	**1334 Rank**
LA	2000	1
LB	1500	3
LC	1500	3
LD	1000	4

Notice how LB and LC have the same rank because they have the same revenue. However, they are both rank 3 instead of rank 2. Additionally, there is a gap between rank 1 and rank 3 (*i.e.*, missing rank 2). This is because there is a tie for rank 2 and we create a gap before putting in a rank.

Given the ranking above, top 2 locations will return only LA with revenue of 2000. This is because there is no actual rank 2 location. Top 3 locations will return LA, LB, and LC with their respective revenues. The output are sorted in ascending order of rank (*i.e.*, rank 1 at the top) following by ascending order of location name (*i.e.*, for locations with the same rank, order them by location name such that lexicographically smaller location name is on top). If there are fewer than n locations, output as many as you can. Note that even if there are more than n locations, the output may still contain fewer than n rows.

```
CREATE OR REPLACE FUNCTION top_n_location (
  n INT, sdate DATE, edate DATE
) RETURNS TABLE (lname TEXT, revenue NUMERIC, rank INT) AS $$
  -- your code here
$$ LANGUAGE plpgsql;
```

- n is an *INT* to find the top n location (*i.e.*, the cutoff rank).
- sdate is a *DATE* corresponding to sdate in the range [sdate, edate] from the description.
- edate is a *DATE* corresponding to edate in the range [sdate, edate] from the description.

Clarifications:
Given the table with the 1334 ranking above, we get the following results given the appropriate value sdate and edate.

- top_n_location(1, sdate, edate)

lname	revenue	rank
LA	2000	1

- top_n_location(2, sdate, edate)

lname	revenue	rank
LA	2000	1

- top_n_location(3, sdate, edate)

lname	revenue	rank
LA	2000	1
LB	1500	3
LC	1500	3

Deliverables

Each team is to upload an **sql** file named “`teamNNN.sql`” where “`NNN`” is the three digit team number according to your project group number. You should add leading zeroes to your team number (*e.g.*, `team005.sql`). Submit your **sql** file on Canvas assignment named “Project 03” (https://canvas.nus.edu.sg/courses/52825/assignments/111108). Only the **latest** submission that is submitted **before the deadline** will be marked.

Your **sql** file should contain the triggers and routines needed for the project. At the top of your **sql**, write as comments, your group number as well as the contribution of each team member in the following format:

```
/*
Group #
1. Name 1
- Contribution A
- Contribution B
2. Name 2
- Contribution A
- Contribution B
  :
*/
```

Everyone in the group is expected to contribute equally to the project.

Deadlines

The deadline for the submission is **12:00** of ~~**Saturday** of Week 12 (*i.e.*, **13 April 2024**)~~ **Wednesday** of Week 13 (*i.e.*, **17 April 2024**). Only submissions through Canvas will be accepted. Submissions through other means (*e.g.*, emails) will not be accepted. Any extensions will be announced on Canvas.

Late Submissions

6 marks (*out of 18*) will be deducted for submissions up to **1 day late**. Submissions late for more than 1 day will receive **0 mark** and will not be graded.

Grading

The grading will be done automatically. Private test cases will be set up to check for properties of each triggers and routines (*e.g.*, for `top_n_locations`, the properties may be (a) can handle fewer than `n` locations, (b) can handle ties correctly, *etc*). Each trigger carries at most 1 mark and each procedure carries at most 2 marks.

Clarifications:
The steps in the grading will be done as follows:

1. Reset the database.
   - This guarantees any previous triggers and/or routines will not interfere with your code.
2. Run `DDL.sql` to initialize the schema.
3. Insert preliminary ***valid*** data.
   - This is done before running your triggers in-case your triggers are faulty and causes certain insertion to fail.
   - We try to avoid double-penalizing your code, but this is not fully guaranteed especially if your code has syntax errors.
4. Run your solution.
   - This will insert all triggers and routines into the database.
5. Run test cases.

Guides:

- Arrays in postgresql starts from index 1. The notation is similar to other languages `arr[i]`.
- Arrays can be created using `ARRAY[val, val, val]`.
- Looping through array can be done as follows:
```
FOR i IN 1..array_upper(arr, 1) LOOP
  -- your code
END LOOP;
```
- `DATE` can be operated on rather easily in postgresql.
  - `date1 + num` will produce the date `num` days after `date1`. If `date1` is `01-01-2024` and `num` is `3`, then the result will be `04-01-2024`.
  - Dates can be compared with `date1 < date2`, or other relational operator.
- To test your code, be careful with the format of date.
  - You should use the ISO standard of `DATE('yyyy-mm-dd')`.
  - For instance, to create the date `01-02-2024` (*i.e.*, 2nd of January 2024), you should use `DATE('2024-01-02')`.
  - Non-standard usage may rely on local machine setting (*e.g.*, US vs UK date setting).

Reminder:

- Do not forget to `CLOSE` cursor after you finish using it.
  - This is to avoid memory leaks and other complications especially if running your routines multiple times.
  - Since the problem may arise only when the routine is invoked multiple times, you may not realize that there is a problem.
- Please be careful when looking at helps online as
  - they may use outdated and/or deprecated features.
  - they may use additional libraries, frameworks, extensions, *etc.*
  - they may not fully conform to PostgreSQL standard (*e.g.*, may be for MySQL, *etc*).
- Please ensure that your code can be run from blank database state. Common mistake includes (*but not limited to*) using `DROP TABLE` instead of `DROP TABLE IF EXISTS ... CASCADE`.

FAQ:

- You may use all features of postgresql unless the feature requires additional libraries, frameworks, extensions, etc. This includes:
  - You may use CTE.
  - You may use `RANK`.
  - You may use `OVERLAPS`.
- We will not be specifying how to use them and/or the assumptions made by the features not discussed.
  - Please make sure you understand their assumptions.
  - Please check corner cases.