CS2102 Project (Part 2)

The objective of this part of the team project is for you to apply what you have learned in class to implement triggers and routines for a schema designed for the application mentioned in Part 1. In relation to this, we provide a reference ER diagram in "ER Diagram.png", as well as a relation schema in "schema.sql". Note that the schema has implemented some constraints using foreign keys and checks. You are to implement additional triggers and routines, **using PostgreSQL 14.1 or 14.2**, according to the requirements detailed in the following.

Note: Please **do not** make any changes to the schema in "schema.sql". This is because we will evaluate each team's implementation using automated testing, for which we need each team's schema to be exactly the same. (For details, you may refer to the FAQ in Section 4.)

1. Constraints to be Enforced using Triggers

Please implement appropriate triggers to enforce the following 12 constraints. For simplicity, you only need to consider INSERT triggers (i.e., triggers that are activated by insertions), and do NOT need to consider DELETE or UPDATE. For each constraint, you are allowed to use multiple triggers to enforce it.

Product related:

(1) Each shop should sell at least one product.

Order related:

(2) An order must involve one or more products from one or more shops.

Coupon related:

(3) A coupon can only be used on an order whose total amount (before the coupon is applied) exceeds the minimum order amount.

Refund related:

(4) The refund quantity must not exceed the ordered quantity.
- Note: for each order, we need to ensure that the sum of the quantifies of all refund requests (except those that have been rejected) does not exceed the ordered quantity.

(5) The refund request date must be within 30 days of the delivery date.

(6) Refund request can only be made for a delivered product.

Comment related:

(7) A user can only make a product review for a product that they themselves purchased.

(8) A comment is either a review or a reply, not both (non-overlapping and covering).

(9) A reply has at least one reply version.

(10) A review has at least one review version.

Complaint related:

(11) A delivery complaint can only be made when the product has been delivered.

(12) A complaint is either a delivery-related complaint, a shop-related complaint or a comment-related complaint (non-overlapping and covering).

2. Routines

Routines must be implemented with the same name specified in this document, following the same order of input parameters and returning the exact output parameter type (if any). You may change the names of the input parameters in your implemented routines, but not the data type of the input parameters. If a call to a routine should fail because of invalid parameters or database constraint violations, the routine should not return any values. You may raise an exception explicitly, or simply allow it to fail silently. It is important to adhere to these instructions since automated testing will be used to evaluate your implementation. Therefore, the input and output formats of routines have to be fixed.

For all routines, you may assume that the input values are all valid, i.e., you do not need to do validity checks for them in your implementation.

In addition, if you are to use arrays, you should follow PostgreSQL's default behaviour of starting the index subscript from 1. You can also use any functions available in PostgreSQL (e.g., DATEDIFF).

2.1 Procedures

The following routines in this section do not have return values, and should be implemented as PostgreSQL procedures (https://www.postgresql.org/docs/14/sql-createprocedure.html).

(1) place_order(user_id INTEGER, coupon_id INTEGER, shipping_address TEXT, shop_ids INTEGER[], product_ids INTEGER[], sell_timestamps TIMESTAMP[], quantities INTEGER[], shipping_costs NUMERIC[])
- Places an order by a user for selected product listings
- DATATYPE[] refers to an array of DATATYPE
- Each orderline will have one entry in each array, e.g. shop_ids[i] refers to the shop_id of orderline i
- The quantity sold for each product listing involved should be updated after the order is placed successfully
- The order's payment_amount should be calculated as the sum of (price * quantity + shipping_cost) for each purchased product
- The coupon_id is optional; it is NULL if the user does not use a coupon
    - If a coupon is used, the payment_amount should be adjusted

(2) review(user_id INTEGER, order_id INTEGER, shop_id INTEGER, product_id INTEGER, sell_timestamp TIMESTAMP, content TEXT, rating INTEGER, comment_timestamp TIMESTAMP)
- Creates a review by the given user for the particular ordered product

(3) reply(user_id INTEGER, other_comment_id INTEGER, content TEXT, reply_timestamp TIMESTAMP)
- Creates a reply from user on another comment

2.2 Functions

The following routines in this section have return values, and should be implemented as PostgreSQL functions (https://www.postgresql.org/docs/14/sql-createfunction.html).

(1) view_comments(shop_id INTEGER, product_id INTEGER, sell_timestamp TIMESTAMP)
- Output: TABLE (username TEXT, content TEXT, rating INTEGER, comment_timestamp TIMESTAMP)
- Retrieves info about all comments related to a product listing (an instance of the Sells relation)

    - This includes reviews, and also replies to the reviews for that product listing, as well as replies to replies (i.e., the whole reply chain should be returned).
    - You may consider using loops or recursive queries (see https://www.postgresql.org/docs/current/queries-with.html, Section 7.8.2).
- If the comment is a reply, the rating should be NULL for that row
- If a comment has multiple versions, return only the latest version
- If a comment belongs to a deleted user, display their name as 'A Deleted User' rather than their original username
- Results should be ordered ascending by the timestamp of the latest version of each comment
    - In the case of a tie in comment_timestamp, order them ascending by comment_id

(2) get_most_returned_products_from_manufacturer(manufacturer_id INTEGER, n INTEGER)

- Output: TABLE (product_id INTEGER, product_name TEXT, return_rate NUMERIC(3, 2))
- Obtains the N products from the provided manufacturer that have the highest return rate (successfully refunded)
    - Products are only successfully refunded if the refund_request status is 'accepted'
    - The output table may have fewer than N records if the manufacturer has produced fewer than N products
- Return rate for a product is calculated as (sum of quantity successfully returned across all orders) / (sum of quantity delivered across all orders)
    - The return rate should be a numeric value between 0.00 and 1.00, rounded off to the nearest 2 decimal places
    - If a product has never been ordered, its return_rate should default to 0.00
- Results should be ordered descending by return_rate
    - In the case of a tie in return_rate, order them ascending by product_id

(3) get_worst_shops(n INTEGER)

- Output: TABLE(shop_id INTEGER, shop_name TEXT, num_negative_indicators INTEGER)
- Finds the N worst shops, judging by the number of negative indicators that they have
- Each ordered product from that shop which has a refund request (regardless of status) is considered as one negative indicator
    - Multiple refund requests on the same orderline only count as one negative indicator
- Each shop complaint (regardless of status) is considered as one negative indicator
- Each delivery complaint (regardless of status) for a delivered product by that shop is considered as one negative indicator
    - Multiple complaints on the same orderline only count as one negative indicator
- Each 1-star review is considered as one negative indicator
    - Only consider the latest version of the review
    - i.e., if there is a previous version that is 1-star but the latest version is 2-star, then we do not consider this as a negative indicator
- Results should be ordered descending by num_negative_indicators (the total number of all negative indicators listed above)
    - In the case of a tie in num_negative_indicators, order them ascending by shop_id

3. Deadline and Deliverables

By 6pm, April 8 (Friday), 2022, each team is to upload a zip file named teamNN.zip, where NN is the team number, to the LumiNUS file folder named Part_2_submissions. For late submissions, upload the zip file to

the LumiNUS file folder named Late_Part_2_model_submissions. 5 marks (out of 18) will be deducted for submissions up to one day's late; submissions late for more than one day will receive zero marks and will not be graded.

The submission teamNN.zip should contain the following two files:

- report.pdf: project report in pdf format
- proc.sql: the triggers and routines of your implementation

The project report (up to a maximum of 20 pages in pdf format with at least 10-point font size) should include the following contents:

- Names and students numbers of all team members and project team number (on the first page).
- A listing of the Project (Part 2) responsibilities of each team member.
    - Team members who did not contribute a fair share of the project work will not receive the same marks awarded to the team.
- For each trigger:
    - Provide the name of the trigger.
    - Explain the basic idea of the trigger implementation.
- For each routine:
    - Explain the basic idea of the routine implementation.
- A summary of any difficulties encountered and lessons learned from the project.

4. FAQ

(1) "Why are we not allowed to make any changes to the schema provided or the routine names and parameters given?"

- This is because we will evaluate each submission using automatic testing. That is, we will prepare some test data following the schema provided to you, and then combine it with your proc.sql and run some queries. We will examine the database state after each query to check whether it is correct; if it is, then you get some marks associated with the query. To facilitate such automatic testing, it is important that every team's implementation is based on exactly the same schema, and uses the exact routine interface that we provided.

(2) "Will the test data be provided?"

- Unfortunately no. But you may generate your own test data to check your implementation.