CS2102

Project 2

Team 109

Team Members:

Hoang Huu Chinh

Gupta Ananya Vikas

Tan Jing Xue Andre

Phua Anson

1. Project Responsibilities

Name	Contribution
Hoang Huu Chinh	- Created the dummy data file - Worked on Triggers 2,5,6, Procedure 3, Function 3
Gupta Ananya Vikas	- Worked on Trigger 1,3, Procedure 1 & 2, Function 2
Tan Jing Xue Andre	- Worked on drafting the report - Worked on Triggers 1, 3
Phua Anson	- Worked on Triggers 3,4, Function 1& 3

The project responsibilities were evenly distributed. We coded the triggers, functions and procedures separately, divided evenly among every member and then we came together to debug and show the logic behind our coded functions, procedures and triggers, followed by giving feedback to each other and fixing the routines with broken logic. Next, each of us tested on the other's routines to ensure that they are compiling and working with some test cases and edge cases suggested.

Files included in the zip file: DDL.sql (Schema file), data.sql (dummy data file), Proc.sql (functions, procedures and triggers)

2. Triggers

Trigger 1: To enforce the constraint that Users === {Creators, Backers}

Trigger Name: check_user_type

Function name: check_user()

Description:

This trigger checks that a user must either be a backer, creator or both. This is reinforced through the function check_user and a series of if-else statements to determine its kind. So first, the user would be checked against the records in the Creators table and then Backers table where the declared variable 'assigned' would be updated each time. The user (identified through his unique email) is first checked from the Creators' pool then subsequently, from the Backers'. If the assigned quantity remains 0, it indicates an invalid user type and that entry is deleted as it is not a user belonging to either of the tables. An exception is raised to indicate the violation and a null value is returned.

Given that the function needs to verify the user, it is deferrable in nature and therefore, called AFTER INSERT ON since the user is added to the User table and checked after the constraint is initially deferred.

Trigger 2: To enforce the constraint that a backer's pledge amount should minimally be greater or equal to the reward level

Trigger Name: trigger2

Function name: trigger2_func()

Description:

This trigger guarantees a backer's pledge is only verified if it meets the required reward level. The numeric min is first recorded. It then captures a reward's minimum amount and through an if-else statement, a new reward ID and name will only be updated if that user's min input is greater or equal to the reward level. Otherwise, a null output is returned to indicate a rejection.

Trigger 3: To enforce that each project must have at least one reward level

Trigger Name: reject_adding_project

Function name: do_not_add_project()

Description:

The function enforces that a project must have one reward level at minimum. We first declare a counter to tally the reward levels.

Should the counter have at least 1 level, we return the new ID of the reward. Otherwise, we return a null with a warning that the input project does not have any reward level.

The trigger used BEFORE INSERT as we wish to execute this trigger first to check if the conditions are met before we insert the accepted ID in.

Trigger 4: Enforce the constraint that refunds can only be approved if refunds are requested within 90 days of the deadline and that refunds not requested cannot be approved or rejected.

Trigger Name: process_refund

Function name: processing_the_refund

Description:

The function initially declares two date features; “request_date” determines if a refund is requested (it stores the attribute “requests” from Backs) and “project_deadline” captures the date of deadline from the entity Projects. A successful refund updates the primary key ID of Project, the refund’s email and back’s pid.

The validation comes in 2 if-else statement, the first checks if the request_date is null, if so, there is no request and null is return, otherwise a refund is requested. Within another nested if-else loop, this statement now checks if the requested time exceeds the Project's deadline attribute by 90, if so we will return the new tuple but indicate a "false" to signify that this refund is not approved. Otherwise, we follow through with the request.

The trigger uses BEFORE INSERT again as we wish to execute this trigger to check whether a request is successful before inserting the new approved/unapproved tuple.

Trigger 5: Enforce the constraint that backers back before deadline

Trigger Name: trigger5

Function name: trigger5_func()

Description:

We declare 2 date features: created_date (which captures the Project's date created attribute) and deadline_date (captures the deadline attribute of the same Project).

We store any update from the Project primary Key onto the new output "NEW.id".

Through an if-else statement, we will check if a backer's requested date has exceeded the existing deadline, or if a new requested date happens before the date of any project created. If so, we return a NULL value. Otherwise, we will make the update accordingly.

This trigger runs on BEFORE INSERT, before we do an insertion, we have to verify the constraint that a backer cannot back after a deadline nor back before any project is even created.

Trigger 6: Enforces the constraint that refunds can only be made for successful projects

Trigger Name: trigger6

Function name: trigger6_func

Description:

Declaring 2 numeric: “total”, “goal” features ; 1 date: “deadline” feature

In “total”, we use the aggregate function SUM() to tally up the refund value (specifically the numeric attribute amount in Backs)

In “goal” and “deadline” we store their respective attributes from the Project entity.

In an if-else statement, we verify if a request date is valid by first checking the refund date has passed the deadline. Concurrently, we also check if our input refund value exceeds or equals to that project’s goal value. If both conditions are met, we will update an existing New.id back into a pre-existing projects.id. Otherwise, we return a null to indicate an unsuccessful refund.

This trigger runs on BEFORE UPDATE as no insertions are made, but instead, we wish to fire the trigger to check if refunds can be made possible before we update the project ID.

3. Procedures

Procedure 1: Adding new user which is a creator, backer or both

Name: add_user

Description:
This function uses a nested if-else statement where first the given attributes for the new user are checked if the 'kind' of user is amongst the three possible values: BACKER, CREATOR, BOTH and if it belongs to one of them then it is added in the Users table. Within this IF statement, there is an individual check done for each of the kinds to add them to their respective tables Backer or Creator or added in both accordingly. Thai ensures that no invalid user is added in any table.

Procedure 2: Adding a new project with their reward level

Name: add_project

Description:

Our procedure initially states all required attributes and their data types. An INT holder labeled 'n' is declared which will store the length of the arrays of the names of the project.

We continue if this length 'n' is more than 0 (indicating a project exists) where we will add into Projects the new id, email, ptype (Project type), created (date of creation), name, the project's deadline and goal and then add the first element of the arrays of names and minimum amounts given for the new project id into Rewards table to prevent foreign key constraint being violated.

Following this, the function uses a loop to traverse through the arrays of names and minimum amounts given for the project id from index 2. Within the loop, we will then insert into the Rewards table the name, id and pledge amount extracted from the arrays, until all n elements of reward levels for the project id are added.

Procedure 3: A procedure to help an employee with id ‘eid’ auto-reject all refund request made 90 days from the project’s deadline

Name: auto_reject

Description:

The procedure declares 2 feature data type, ‘eid’ (INT) for the employee’s ID and ‘today’ (DATE) for today’s date

We start with an UPDATE Statement as our aim is to modify existing entries in the Refunds table.

The SET Command next updates our date attribute to today and the accepted is updated as FALSE to signify a rejected request.

From the Refunds and Projects Table, we search for Projects with Refunds via their primary key (email) along with its Employee ID (eid).

We then specify the condition that a rejected request would be one whose time exceeds the project’s deadline by 90 days (Refunds.date – Projects.deadline > 90) as auto_failed (its Employer ID is also captured here).

The WHERE STATEMENT next will then filter the requests as rejected if they satisfy this condition.

4. Functions

Function 1: Write a function to find the emails and names of all superbackers for a given month and year

Name: find_superbackers

Description:

The function initially declares some datatypes, 2 Integer datatype counters for projects and types (successful_project_count, successful_project_type_counter), Booleans criterions A and B for for conditions and a cursor.

Our function utilizes this cursor to loop through the backers' email to check if each backer fulfills Condition A or B as stated. Should a backer not be verified by either conditions, then they will be assigned 'Not superbackers' to their name (Where name <> 'Not superbacker').

Function 2: Find the find the top N most successful project's ID, name,email of creator and the amount pledged based on a success metric for given date

Name: find_top_success

Description: This function returns the result as a table with the attributes ID , name , email , amount captured from the Projects table. To derive the success metric: the funding ratio, the aggregate function is used on the backer's pledge amount before dividing it by the project goal.

A WHERE Clause then filters based on our project type , it's id and a GROUP BY statement is used to group rows according to the ID. An ORDER BY keyword helps sort our attribute funding_ratio first then deadline and ID in a descending order to ensure we obtain only the most successful projects first

To ensure we obtain the most successful projects in a descending order, an ORDER BY keyword helps sort our attribute funding_ratio first then deadline and it's ID in that manner. Finally the LIMIT Clause is used to specify the number of records to return.

Function 3: Write a function to find the project's ID and name, the email of the creator and number of days it takes for the project to reach its funding goal, for the top N most popular projects based on the popularity metric for the given date.

Name: find_top_popular

Description:
In our main function : find_top_popular, we aim to return a table of different Projects with their attribute ID, name, Creator's Email and the number of days to reach the funding. We first execute the selection through the SELECT DISTINCT Statement to select required keys from their respective tables. To select the DAYS feature, we call onto a predetermined function (num_of_days_to_reach_goal). The ORDER BY… ASC Keyword then sorts every project DAYS and ID to implement the popularity metric. Finally we call the LIMIT Clause to restrict the number of returns to the top N and call another ORDER…BY DESC to sort our results table in a descending order

Function 4: subordinate function used with find_top_popular

Name: num_of_days_to_reach_goal

Description:
This function works to give the number of days to reach a goal by checking through each record in the Backs table using a cursor. First, the goal, progress, cursor and record are declared, which are updated as we loop along every record. The project goals are extracted separately into variable 'goal', which then checks for popularity within the cursor to return the number of days to reach the goal compared to the funding amount which is recorded under 'progress' variable.

5. Reflection

The experience of conducting this project was quite enriching and made us appreciate the content taught in class. The relation between a constraint and its implication while coding the triggers was something we explored thoroughly in this project. In terms of the routines, we did a lot of discussion and trial and error on how to enforce each trigger and the logic enforced in each function. Following that, we carried out several tests using the data file we created and kept trying different edge cases to ensure that each routine worked in the required manner, which highlighted new problems and we would have to fix the functions based on any failed case to check how to enforce the additional constraint or if we are missing a particular constraint. Overall, it was a lot of back and forth to finally conclude with a function that would cover as many cases as possible.

One of the challenges was that while creating a test data file, it was difficult to incorporate every kind of case so we had to take some time to manually populate the data file with each row to ensure every constraint is satisfied. Considering these cases and changing the routines in several iterations was surely challenging and made the time allocated for this project quite intensive into debugging and redefining our implementations.

In terms of group work, we split the project routines amongst ourselves. We split the routines in a way that every person had 4-5 routines to work on independently. Hence, when we discussed our logic and approaches, so that we can debug faster and clarify the logic of the routines better. Once, we discussed the entire code in our weekly meetings, we tested each other's functions and triggers for more thorough testing. It made sure that we had weekly meetings to keep up with our assigned work and make sure we are moving in the right direction for the next meeting.

Overall, we feel that this project has given us a good understanding of the fundamental concepts and implementation of them, which can be applied in future projects.