CS2102 Project: Part 2

Project Team Number: 100

Team Members:

Jung Hyunseok (Peter)
Han Xiao Guang
Nicholas Russell Saerang
Chia Jeremy

1 Responsibilities

Jung Hyunseok (Peter)

- Triggers 4 and 6
- Function 3

Han Xiao Guang

- Triggers 2 and 5
- Function 1

Nicholas Russell Saerang

- Triggers 1 and 3
- Procedures 1 and 2

Chia Jeremy

- Procedure 3
- Function 2

The team divided the questions based on difficulty so that each member could contribute equally to the project. The questions were ranked in decreasing order of difficulty: functions → procedures → triggers.

2 Triggers

2.1 Trigger 1

Trigger name: `block_user`

Trigger function: `check_user`

Idea:

The trigger function is run after an insert on Users table and checks if there has been any user which does not exist in both Backers and Creators tables. This behaviour can be manually produced by inserting a tuple to Users but neither Backers nor Creators. If the trigger occurs, an exception is raised with the message `'There exists a user who is neither a backer nor a creator'`. This encourages the user to actually use a procedure to add the user to the Users table as well as the Backers and/or Creators table. Note that the trigger is a deferrable trigger so it will not check directly after an insertion to the Users table but rather after a commit that includes an insertion to the Users table.

With the presence of the procedure `add_user`, provided that the given parameter inputs are valid, this trigger will not block the insertion to the Users table because the procedure guarantees that right after an insert to Users table is done, an insertion to either Backers or Creators is also done.

The consideration of not returning NULL and instead raising an exception when the check inside the trigger fails is just for the sake of clarity and disambiguation. Suppose we return NULL, then calling the procedure `add_user` with the incorrect kind should technically do nothing, but from our point of view it is harder to determine the reason that nothing happens, either the trigger firing, or the procedure just doing nothing but the constraints are still satisfied.

There is one alternative implementation considered with the presence of the `add_user` procedure: if the trigger is applied upon insertion to Backers and Creators (two triggers), either BEFORE or AFTER, we will not be able to catch the constraint violation due to the insertion to Users but neither Backers nor Creators.

2.2 Trigger 2

Trigger name: `block_pledge`

Trigger function: `check_amount`

Idea:

The trigger function is run before an insert on Backs and checks whether the backing amount to be inserted is strictly less than the minimum amount for the reward level. If the condition is true, the insertion is blocked. Otherwise, the insert is successful. The minimum amount is obtained from the Rewards table filtered by the Project ID and name of the Reward Level in the insertion.

2.3 Trigger 3

Trigger Name: `block_project`

Trigger Function: `check_project`

Idea:

The main idea of the implementation is the same as of Trigger 1 where after the insertion to the Projects table, we need to make sure there is at least one corresponding tuple on the Rewards table. The main check of the trigger is just whether the newly added project has no rewards attached to it, which is done by looking at the `id` column on the Rewards table. If this check fails, we shall raise an exception `'There exists a project without any reward levels'`. Note that the trigger is a deferrable trigger so it will not check directly after an insertion to the Projects table but rather after a commit that includes an insertion to the Projects table.

With the presence of the procedure `add_project`, provided that the given parameter inputs are valid, this trigger will not block the insertion to the Projects table because the procedure guarantees that right after an insert to Projects table is done, an insertion to Rewards is also done.

There is also one alternative implementation considered with the presence of the `add_project` procedure, similar to that of the insertion trigger to Users: if the trigger is applied upon insertion to Rewards, either BEFORE or AFTER, we will not be able to catch the constraint violation due to the insertion to Projects but not Rewards.

2.4 Trigger 4

Trigger name: `block_approval`

Trigger function: `check_deadline`

Idea:

The trigger function is run before an insert on Refunds and checks for two conditions: (1) whether the refund request does not exist (`r.request IS NULL`), but if it does, (2) if the refund approval status is "accepted", checks whether the refund approval has not been made within 90 days of the deadline (`New.accepted = TRUE AND 90 < r.request - r.deadline`). If either of these conditions are true, the insertion for the refund approval is rejected and a new row is not inserted into Refunds. The results that can occur from this trigger function is as follows:

Refund request made?	Accepted?	Within 90 days of the deadline?	Result
YES	YES	YES	INSERT
YES	YES	NO	NO INSERT
YES	NO	YES	INSERT
YES	NO	NO	INSERT
NO	YES	YES	NO INSERT
NO	YES	NO	NO INSERT
NO	NO	YES	NO INSERT
NO	NO	NO	NO INSERT

This ensures that for every successful insert on Refunds,

1. The refund was actually requested, and
2. The request was made within 90 days of the deadline if the request is approved.

2.5 Trigger 5

Trigger name: `bef_deadline`

Trigger function: `check_backing_date`

Idea:

The trigger function is run before an insert on Backs and checks whether the backing date to be inserted is before or on the deadline of the Project. It also checks whether the backing is after the creation of the project. If both conditions are true, the insertion is successful. Otherwise, the trigger function blocks the insertion. The date of the creation and deadline of the project are obtained from the Projects table filtered by the Project ID in the insertion.

2.6 Trigger 6

Trigger name: `block_request`

Trigger function: `check_request`

Idea:

The trigger function is run before an update on Backs and checks whether a project is successful or not before processing a refund request. That is, with respect to the definition of a successful project, the function verifies that (1) the total pledged amount from all backers meets or exceeds the goal and (2) the deadline has passed. This is achieved by joining Projects and Backs based on the project ID (`WHERE B.id = P.id`), isolating rows with the project that is supported by the backing using the project ID (`AND P.id = New.id`), verifying that the refund request date occurs after the project deadline (`AND New.request > P.deadline`), and grouping all remaining rows by the project ID (`GROUP BY P.id`) to use with the aggregate function `SUM` to confirm that the total pledged amount from all backers is equal to or greater than the

project goal (`SUM(B.amount) >= P.goal`). If this condition holds true, the update is executed successfully. Otherwise, the update is not performed.

3 Procedures

3.1 Procedure 1

Procedure name: `add_user`

Idea:

We use multiple if-else statements to determine what INSERT statements to execute depending on the kind of the user: creator, backer, or both. The insertion into Users must be done before the insertion to either Creators or Backers as both Creators and Backers have a foreign key constraint where the email attribute must reference the one at the Users table.

3.2 Procedure 2

Procedure name: `add_project`

Idea:

We first add the single row into the Projects table, and then using the UNNEST built-in function on the names and amounts array, we can insert the pairwise elements into the Rewards table. Note that this should not be a problem since the given assumption is that the two arrays are of the same length so there should not be any null value generated from the unnesting. Since we want to force the constraint that a reward must have at least one reward level, an additional IF conditional is inserted within the procedure.

3.3 Procedure 3

Procedure name: `auto_reject`

Idea:

Considering that when a refund request is rejected, it has to be inserted into the Refunds table, the basic idea is to use a cursor to loop over a database and insert it

into Refunds whenever rejected. Using a selection query, we find all refund requests that haven't been approved or rejected yet, and whose date of request has passed 90 days since the project deadline. The table returns all refund requests that have passed the 90-day deadline and are to be rejected. The cursor will then iterate over this table and insert each tuple into the Refunds table with the accepted attribute set to `False`.

4 Functions

4.1 Function 1

Function name: `find_superbackers`

Idea:

The first CTE created (`successful_projs`) keeps track of the list of relevant successful projects. This is done by grouping the projects in Backs and Projects by their id and summing up their backing amounts. A HAVING clause is used to obtain only projects where the sum of backing amounts is greater than or equal to the goal of the project. An extra condition is added to ensure that the deadline is before today and within 30 days of today.

The second CTE created (`superbackers`) would keep track of the superbackers using the 2 conditions given. As it is a disjunction, UNION is used to join 2 queries. The first query looks for backers who satisfy the first condition. This is checked by counting the IDs and distinct types of project the Backer has backed from an inner join of Backs and Projects, and checking if they are more than or equal to 5 and 3 respectively. The first CTE is used here to filter out the irrelevant projects. The second query looks for backers who have satisfied the second condition. This is checked by summing up the amount the Backer has placed into relevant projects (again filtered by the first CTE) and checking if it is greater than or equal to 1500. Then, the request column for the backer is checked to ensure that there are no non-NULL entries between today and 30 days before inclusive. In both queries, the existence of the email in the Verifies table is checked and the verification date must be on or before today.

The final query obtains the email and name from Users whose email exists in the second CTE, ordered by email.

4.2 Function 2

Function name: `find_top_success`

Idea:

The implemented function differs from the template in that the name of the input parameter `ptype` has been changed to `projectType`. As the Projects table also contains an attribute named `ptype`, this avoids confusion and errors. Without renaming, if I wish to use the input parameter's `ptype` within a selection query that uses the Projects table, Projects' `ptype` will be used instead. Renaming the Projects' `ptype` using an inner query could remedy this, however, this would be redundant since renaming the input parameter would accomplish the same thing.

First, a selection query is used to find the combinations of all Projects and Backs that have the same project IDs, the same project types, and the deadline is before the `today` date. It satisfies one of the two conditions for a successful project, which is that the deadline has passed. Our next step is to group them by project ID using the GROUP BY clause since the current combination of values can return multiple rows of the same Project with different Backs. Our next step is to use the aggregate function SUM over Back's attribute `amount` to ensure that we satisfy the second condition of a successful project, which is to reach the funding goal. We next order the successful projects using the ORDER BY clause, based on the ratio of total amount pledged to the funding goal, followed by deadlines with later dates, and finally, smaller project IDs. As a final step, we extract only the top *N* rows as given by the input.

4.3 Function 3

Function name: `find_top_popular`

Idea:

The solution function (`find_top_popular`) makes use of a helper function (`projects_with_days`) to first retrieve the number of days it took for each project (with a total pledged amount greater than its goal) to reach its funding goal, and uses this information to find the top *N* most popular projects, returned in descending order of popularity metric.

The helper function performs the aforementioned retrieval by using a cursor to iterate row by row down a table of projects and their respective backings, ordered by the project ID and subsequently by the backing date.

```
SELECT P.id, P.email, P.ptype, P.created, P.name, goal,
backing, amount
    FROM Projects P, Backs B
    WHERE P.id = B.id
    ORDER BY P.id, B.backing
```

This way, the cursor encounters rows in groups that represent a single project and within each group, iterates down each row in chronological order of backing date. The cursor is accompanied by three initialised variables, `prev_id = -1` (the project ID of the previous row), `cum_sum = 0` (the cumulative sum to the date at which the funding goal was reached), and `added = FALSE` (a flag to determine whether a new row with the number of days it took to reach the funding goal has already been added for a group of rows, i.e. for a project).

For every row the cursor encounters, if the project ID of the current row is different from that of the previous row (`r.id <> prev_id`), then it acknowledges that the cursor is entering a section of rows for a new project and must reset the variables (`prev_id := r.id; cum_sum := 0; added := FALSE;`).

If a new row with the number of days it took to reach the funding goal for the current project in which the cursor operates has not been added yet (`added = FALSE`), the cumulative sum is increased by the pledged amount of the current row (`cum_sum := cum_sum + r.amount;`). Then, if the cumulative sum is greater than or equal to the funding goal (`cum_sum >= r.goal`), a new row with the number of days it took to reach that goal is added to the return table and `added` is set to `TRUE`. This process repeats down every row and upon reaching the end of the Projects × Backs table, the resultant table is returned with information about projects that reached their funding goal and the number of days it took to reach said goal.

`find_top_popular` uses the table returned from `projects_with_days` and picks projects that were (1) created before today (`P.created < today`) and (2) are of the specified project type (`P.type = ptype`), ordered by the number of days it took for a project to reach its funding goal, i.e. its popularity metric (`ORDER BY days`), and limited to *N* projects only (`LIMIT n`).

5 Additional Routines

The only routine that uses additional sub-routines is `find_top_popular` (function 3) with additional routine `projects_with_days`. As for the explanation of the mechanism behind the additional routine as well as its relationship with the main routine, it has been discussed above in Section 4.3.

6 Summary

The team greatly enjoyed working on the project as it helped us improve our understanding of database management system (DBMS) concepts and how they are actualised in real-life systems. Although things did work out well for the most part, we did encounter some challenges and in turn, learn valuable lessons along the way.

For the second part of the project, one particularly challenging concept to grasp and apply was cursors. The idea of having to manually iterate down the rows of a table and perform operations seemed rather straightforward, but with the incorporation of variables, i.e. state, that were maintained across each cursor iteration, it was quite difficult to understand how to manipulate these variables and make effective use of them. For example, for function 3 (`find_top_popular`), an additional routine (`projects_with_days`) that involved a cursor and multiple variables was implemented to facilitate computation of a partial cumulative sum for a group of backings. It was not easy to keep track of all the variables and identify which ones mattered at which point in time of the iteration, but now having completed the implementation of the function, the team realises just how powerful cursors paired with variables can be to fine-tune table operations.

Another difficulty encountered was with testing the functions and triggers, as we had to formulate our own test cases. This was particularly difficult to achieve when aggregation was involved (superbackers and popular projects) as it required multiple entries per table. This challenge was only aggravated by foreign key constraints which meant that we had to insert across many tables. We were also prone to formulating biassed test cases which would potentially result in missing out on edge cases; this problem is mitigated however by internal testing within the team, with members testing each others' routines and triggers. This approach highlights the importance of teamwork in highlighting and accounting for edge cases.