CS2102

Team 097 Project 2

Group Member	Matriculation Number	Responsibilities
Carissa Ying Geok Teng	A0205190R	Trigger 3 Procedure 3 Function 3
Nguyen Son Linh	A0200705X	Trigger 4 Trigger 5 Trigger 6
Nicholas Tanvis	A0204738A	Trigger 1 Procedure 1 Function 1
Venkat Vishwanth Sreenivasan	A0223960H	Trigger 2 Procedure 2 Function 2

Triggers

Trigger 1

Enforce the constraint that Users can be Backer, Creator or both

Implementation:

```
CREATE OR REPLACE FUNCTION validate_user()
RETURNS TRIGGER AS $$
BEGIN
  IF NOT EXISTS (SELECT 1 FROM Backers b WHERE b.email = NEW.email) AND NOT
EXISTS (SELECT 1 FROM Creators c WHERE c.email = NEW.email) THEN
    RAISE EXCEPTION 'User is not a backer or creator';
  END IF;

  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE CONSTRAINT TRIGGER valid_user
AFTER INSERT ON Users
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE FUNCTION validate_user();
```

Explanation:

To implement a trigger that enforces that a Users entry must have an accompanying Backers or Creators entry, we can check if there exists an entry in either of these two tables when a user is inserted. However, as this trigger has to coexist with Procedure 1, it is not sufficient for this to be a simple trigger as Procedure 1 requires a Users entry to be added first before the following Backers or Creator entry. This calls for a constraint trigger that is deferred until the procedure has completed.

Trigger 2

Enforce the constraint that (backers' pledge amount) (reward level minimum amount).

Implementation:

```
CREATE OR REPLACE FUNCTION min_pledge_check()
RETURNS TRIGGER AS $$
DECLARE
      min_amount NUMERIC;
BEGIN
      SELECT min_amt INTO min_amount
      FROM rewards WHERE name = new.name AND id = new.id;

      IF (new.amount >= min_amount) THEN
            RETURN new;
      ELSE
            RAISE EXCEPTION 'Minimum Reward level not met';
      END IF;
END;
$$ language plpgsql;

CREATE TRIGGER pledge_checker
      BEFORE INSERT ON backs
      FOR EACH ROW EXECUTE FUNCTION min_pledge_check();
```

Explanation:

Each project has an ID and many reward names. Hence, we use the provided reward name and project id to get the minimum reward level amount. Using this value, we check against the pledge amount to verify if the pledge amount is larger than this minimum amount.

If the pledge is strictly smaller than the minimum reward amount, pledge < minimum amount, then the function raises an exception. This prevents the pledge entry from being added to the ‘backs’ table as well as informs the user of the issue.

Trigger 3

Enforce the constraint Projects === Has (i.e., every project has at least one reward level).

Implementation:

```
CREATE OR REPLACE FUNCTION check_project_has_level()
RETURNS TRIGGER AS $$
BEGIN
  IF NOT EXISTS (SELECT id FROM Rewards WHERE id = NEW.id) THEN
    RAISE 'Project has no reward levels';
  END IF;
  RETURN NULL;
END; $$ LANGUAGE plpgsql;

CREATE CONSTRAINT TRIGGER project_has_level_check
AFTER INSERT ON Projects
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE FUNCTION check_project_has_level();
```

Explanation:

To ensure that projects must have at least one reward level, the trigger checks that for a given project id, there is at least one entry under it in the Rewards table. Additionally, the trigger is deferred. Procedure 2 handles the adding of a project and its reward levels. However, the project must be created first since reward levels strongly depend on projects. To stop Trigger 3 from triggering before the transaction is done, we defer the trigger until the end of the transaction.

Trigger 4

Enforce the constraint that refund can only be approved for refunds requested within 90 days of the deadline. Also enforce the constraint that refund not requested cannot be approved/rejected.

Implementation:

```
CREATE OR REPLACE FUNCTION check_refund_request()
RETURNS TRIGGER AS $$
DECLARE
  rrdate DATE;
  pdeadline DATE;
BEGIN
  SELECT request INTO rrdate
  FROM Backs
  WHERE NEW.email = Backs.email AND NEW.pid = Backs.id;

  IF rrdate IS NULL THEN
    RAISE EXCEPTION 'Refund not requested!';
  END IF;

  SELECT deadline INTO pdeadline
  FROM Projects
  WHERE NEW.pid = Projects.id;

  IF pdeadline + INTERVAL '90 days' < rrdate AND NEW.accepted = TRUE THEN
    RAISE EXCEPTION 'Cannot accept refund requested 90 days after project
deadline!';
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER enforce_refund
BEFORE INSERT ON Refunds
FOR EACH ROW EXECUTE FUNCTION check_refund_request();
```

Explanation:

One trigger is used to enforce both constraints. Since we are dealing with approval/rejection of refund requests, this trigger will run per insertion to the `Refunds` table. First, we obtain the request date (first `SELECT` clause in the trigger function), if it is

null then the refund is not requested. Therefore, there should be no insertion to the `Refunds` table as the schema implicitly enforces its status to be rejected or approved. If it is not null, we proceed to find the project deadline and check whether an approval after 90 days is being inserted to the `Refunds` table. If yes, the transaction is cancelled. An exception is raised to annul the transaction and provide clarity for both cases.

Trigger 5

Enforce the constraint that backers back before the deadline and after it has been created.

Implementation:

```
CREATE OR REPLACE FUNCTION check_backing_day()
RETURNS TRIGGER AS $$
DECLARE
  pcreation_date DATE;
  pdeadline DATE;
BEGIN
  SELECT created, deadline INTO pcreation_date, pdeadline
  FROM Projects
  WHERE NEW.id = Projects.id;

  IF NEW.backing <= pdeadline AND NEW.backing >= pcreation_date THEN
    RETURN NEW;
  ELSE
    RAISE EXCEPTION 'Must back before deadline and after creation!';
  END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER back_after_creation_before_deadline
BEFORE INSERT ON Backs
FOR EACH ROW EXECUTE FUNCTION check_backing_day();
```

Explanation:

This is a straightforward trigger to implement. Our solution simply follows the requirement verbatim, getting the creation date and deadline of the project being backed and checking if `Backs.backing` falls between the range. The trigger runs on every insertion to the `Backs` table.

Trigger 6

Enforce the constraint that refund can only be made for successful projects.

Implementation:

```
CREATE OR REPLACE FUNCTION can_request()
RETURNS TRIGGER AS $$
DECLARE
  pmoney_pledged NUMERIC;
  pdeadline DATE;
  pgoal NUMERIC;
BEGIN
  SELECT deadline, goal INTO pdeadline, pgoal
  FROM Projects
  WHERE NEW.id = Projects.id;

  SELECT COALESCE(SUM(amount), 0) INTO pmoney_pledged
  FROM Backs
  WHERE NEW.id = Backs.id;

  IF pdeadline < CURRENT_DATE AND pgoal <= pmoney_pledged THEN
    RETURN NEW;
  ELSE
    RAISE EXCEPTION 'Can only request refund for successful project!';
  END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER request_refund_on_successful_request_only
BEFORE UPDATE ON Backs
FOR EACH ROW WHEN (OLD.request IS NULL AND NEW.request IS NOT NULL) EXECUTE
FUNCTION can_request();
```

Explanation:

From the project description, it is clear that this is a `UPDATE` trigger that runs on every update on the `Backs` table when `Backs.request` is set from `NULL` to `non-NULL` values. This is reflected by our `WHEN` condition in the trigger definition. With regards to the trigger function, our solution first obtains the goal and deadline of the project. This is straightforward. After that we need to find the total amount pledged to the project, which is a bit tricky but can be handled using the `COALESCE` SQL function. With these values we can then check whether the project is successful and if yes, allow the update to proceed else annul the transaction.

Procedures

Procedure 1

Write a procedure to add a user which may be a backer, a creator, or both.

Implementation:

```
CREATE OR REPLACE PROCEDURE add_user(
  email TEXT, name    TEXT, cc1  TEXT,
  cc2   TEXT, street  TEXT, num  TEXT,
  zip   TEXT, country TEXT, kind TEXT
) AS $$
DECLARE
  is_backer BOOLEAN;
  is_creator BOOLEAN;
  is_none BOOLEAN;
BEGIN
    INSERT INTO Users VALUES (email, name, cc1, cc2);

    is_backer := kind = 'BACKER' OR kind = 'BOTH';
    is_creator := kind = 'CREATOR' OR kind = 'BOTH';
    is_none := NOT (is_backer OR is_creator);

    IF is_none THEN
      RAISE EXCEPTION 'Invalid user kind';
    END IF;

    IF is_backer THEN
      INSERT INTO Backers VALUES (email, street, num, zip, country);
    END IF;

    IF is_creator THEN
      INSERT INTO Creators VALUES (email, country);
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Explanation:

First, a user is created. This is because Backers and Creators have a foreign key constraint, which means the user needs to exist first. (Although this violates what

Trigger 1 enforces, the trigger is deferred.) Then, we eliminate the possibility of the ‘kind’ variable being invalid. Finally, according to the ‘kind’ variable, we insert the corresponding entries to Backers, Creators or both.

Procedure 2

Write a procedure to add a project with all the corresponding reward levels.

Implementation

```
CREATE OR REPLACE PROCEDURE add_project(
  id        INT,  email TEXT,   ptype     TEXT,
  created DATE,   name  TEXT,   deadline DATE,
  goal      NUMERIC, names TEXT[],
  amounts NUMERIC[]
) AS $$
DECLARE
      num_reward_levels INT;
      num_amounts INT;
BEGIN
      SELECT cardinality(names) INTO num_reward_levels;
      SELECT cardinality(amounts) INTO num_amounts;

      IF (num_reward_levels = num_amounts) THEN
            INSERT INTO projects VALUES (add_project.id, add_project.email,
      add_project.ptype, add_project.created, add_project.name,
      add_project.deadline, add_project.goal);
            FOR index IN 1..(num_reward_levels) LOOP
                  INSERT INTO rewards VALUES (names[index], add_project.id,
      amounts[index]);
            END LOOP;
      END IF;
END;
$$ LANGUAGE plpgsql;
```

Explanation

First input checks are done to ensure that the number of reward level names and amounts provided is the same. If they are, we first add the project details into the ‘project’ tables. As the rewards level can exist only when the project exists. Then, we insert the respective reward level name and its value into the ‘rewards’ table.

In the event any of the insertions cause a trigger to raise an exception, the insert is automatically aborted and this entire transaction would be cancelled. Hence, we don't need to explicitly remove the previous values inserted into the table during this transaction.

Procedure 3

Write a procedure to help an employee auto-reject all refund requests made after 90 days from the deadline.

Implementation:

```
CREATE OR REPLACE PROCEDURE auto_reject(
  eid INT, today DATE
) AS $$
DECLARE
  -- select all backing that requests a refund and the refund has not been
accepted/rejected
  curs CURSOR FOR (SELECT * FROM Backs WHERE request IS NOT NULL
    AND NOT EXISTS
      (SELECT * FROM Refunds WHERE Backs.email = email AND Backs.id = pid));
  r RECORD;
  deadline DATE;
BEGIN
  OPEN curs;
  LOOP
    FETCH curs INTO r;
    EXIT WHEN NOT FOUND;
    SELECT Projects.deadline INTO deadline FROM Projects WHERE id = r.id;

    IF deadline + INTERVAL '90 days' < r.request THEN
      INSERT INTO Refunds VALUES (r.email, r.id, eid, today, FALSE);
    END IF;
  END LOOP;
  CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

Explanation:
First, the procedure looks for Backings that have requested a refund but are still pending. This is indicated by a non null value in the request field but no entry in the

Refunds table. For each of these requests, if the date of request is more than 90 days after the project deadline, a corresponding rejected entry will be inserted into the Refunds table using the given Employee id. Else, nothing is inserted into the Refunds table, leaving the request pending.

Functions

Function 1

Find all superbackers sorted by email.

Implementation:

```
CREATE OR REPLACE FUNCTION find_successful_projects_on(
  today DATE
) RETURNS TABLE(id INT, ptype TEXT) AS $$
BEGIN
  RETURN QUERY SELECT p.id, p.ptype FROM Projects p
  JOIN Backs b ON b.id = p.id
  WHERE p.deadline <= today AND today <= p.deadline + INTERVAL '30 days'
  GROUP BY p.id
  HAVING sum(b.amount) >= p.goal;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION find_successful_projects_backed_by(
  today DATE, email_ TEXT
) RETURNS TABLE(id INT, ptype TEXT, amount NUMERIC) AS $$
BEGIN
  RETURN QUERY SELECT p.id, p.ptype, b.amount FROM
find_successful_projects_on(today) p, Backs b
  WHERE p.id = b.id AND b.email = email_;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION find_superbackers(
  today DATE
) RETURNS TABLE(email_ TEXT, name_ TEXT) AS $$
DECLARE
  backer RECORD;

  criteria_one BOOLEAN := FALSE;
  at_least_5_projects BOOLEAN;
  at_least_3_types BOOLEAN;

  criteria_two BOOLEAN := FALSE;
  at_least_1500_sgd BOOLEAN;
  no_refunds BOOLEAN;
```

```
BEGIN
  FOR backer IN
    SELECT b.email, u.name FROM Backers b, Users u, Verifies v
    WHERE b.email = u.email AND b.email = v.email AND v.verified <= today
    ORDER BY b.email ASC
  LOOP
    email_ := backer.email;
    name_ := backer.name;

    at_least_5_projects := (SELECT COUNT(id) FROM
find_successful_projects_backed_by(today, email_)) >= 5;
    at_least_3_types := (SELECT COUNT(DISTINCT ptype) FROM
find_successful_projects_backed_by(today, email_)) >= 3;
    criteria_one := at_least_5_projects AND at_least_3_types;

    at_least_1500_sgd := (SELECT sum(amount) FROM
find_successful_projects_backed_by(today, email_)) >= 1500;
    no_refunds := NOT EXISTS (SELECT 1 FROM Backs WHERE email = email_ AND
request <= today + INTERVAL '30 days');
    criteria_two := at_least_1500_sgd AND no_refunds;

    IF criteria_one OR criteria_two THEN
      RETURN NEXT;
    END IF;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Explanation:
The function requires two helper functions to fetch the list of all successful projects as well as to fetch the same list but filtered by backer. With these helper functions, the criteria can be computed easily. We can make use of the second helper function (which uses the first helper function) to help compute the criteria. The function begins with a loop over the verified backers and if any of the criteria is true, then a superbacker is found and will be added to the result.

Function 2

Find top N projects based on success metric. Success metric may be summarized as sorting in the following order:

- (total money funded) / (project goal) descending.
- (project deadline) descending.
- (project id) ascending.

Implementation

Note: Parameter name ptype changed to projtype to improve code readability

```
CREATE OR REPLACE FUNCTION find_top_success(
  n INT, today DATE, projtype TEXT
) RETURNS TABLE(id INT, name TEXT, email TEXT,
                  amount NUMERIC) AS $$
      -- Compute the amount
      -- Filter projects deadline before given date and same proj type
      -- Order them
      WITH TotalFunds AS
      (SELECT backs.id AS fundsId, sum(backs.amount) AS fundsTotal
            FROM backs
            GROUP BY backs.id),
      ValidProjects AS
      (SELECT projects.id AS validId, projects.name AS validName,
projects.email AS validEmail, projects.deadline AS validDeadline, fundsTotal
AS validTotal, fundsTotal/projects.goal AS validRatio
            FROM projects INNER JOIN TotalFunds
                  ON projects.id = fundsId
            WHERE projects.deadline < today AND projects.ptype = projtype
            ORDER BY validRatio DESC, validDeadline DESC, validId
            LIMIT n)
      SELECT validId AS id, validName AS name, validEmail AS email,
validTotal AS amount
      FROM ValidProjects
$$ LANGUAGE sql;
```

Explanation

Through the use of CTE, we generate two tables:

- 'TotalFunds' table is used to compute the total amount funded for each project.
    1. It groups the data by project ID, which is saved as backs.id, and runs the sum on each group.
- 'ValidProjects' table is used to get projects that meet the success matrix and returns them in a specified order.
    1. We join the 'projects' and 'TotalFunds' tables to merge them and add another column indicating the total funding generated.
    2. We filter the projects by the project type and deadline where if the project type is the intended type and its deadline occurred before the date given, it will be kept in the filtered table.

3. We make the projection through the select command and generate the percentage of goal reached and name it as ‘validratio’. This ratio will be used later on for ordering purposes.
4. We order the table by the validratio, followed by the deadline then finally the id.
5. Then we impose a limit on the number of tuples in the table.

With the ‘ValidProject’ table generated, We make a final projection to match the output type.

Function 3

Find top N projects based on popularity metric Popularity metric may be summarized as sorting in the following order:

- (# days for funding goal to be reached) ascending.
- (project id) ascending.

Implementation:

Assume that projects that do not meet the goal at all are not considered popular and will not be returned in the function.

```
CREATE OR REPLACE FUNCTION get_goal_reach_days(pid INT)
  RETURNS INT AS $$
DECLARE
  curs CURSOR FOR (SELECT * FROM Backs WHERE Backs.id = pid ORDER BY backing
ASC);   r RECORD;
  cum_sum INT;
  goal INT;
  start_date DATE;
BEGIN
  cum_sum := 0;
  SELECT Projects.goal INTO goal FROM Projects WHERE Projects.id = pid;
  SELECT Projects.created INTO start_date FROM Projects WHERE Projects.id =
pid;
  OPEN curs;
  LOOP
    FETCH curs INTO r;
    EXIT WHEN NOT FOUND;
    cum_sum := cum_sum + r.amount;
    IF cum_sum >= goal THEN
      CLOSE curs;
      RETURN r.backing - start_date;
    END IF;
```

```
  END LOOP;
  CLOSE curs;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION find_top_popular(
  n INT, today DATE, type TEXT
) RETURNS TABLE(id INT, name TEXT, email TEXT,
                days INT) AS $$
BEGIN
  RETURN QUERY SELECT p.id, p.name, p.email, get_goal_reach_days(p.id) days
                -- rename columns to prevent ambiguity
                FROM Projects p
                WHERE get_goal_reach_days(p.id) IS NOT NULL
                  AND p.created < today
                  AND p.ptype = type
                ORDER BY days ASC, id ASC
                LIMIT n;
END;
$$ LANGUAGE plpgsql;
```

Explanation:

First, a helper function is used to find the number of days needed for a given Project to reach the goal. This is done by first finding all Backings to said project and arranging them by ascending dates. If we take the cumulative sum starting from the earliest date, we can find the day when the Project has enough fundings to meet or exceed its goal. The function then returns the difference between this day and the starting day.

The function find_top_popular then applies the helper function to all projects to gauge their popularity. The smaller the number of days needed to reach the goal, the more popular. Since we only want the top n most popular projects, the function also limits the output to n selections.

Summary

Difficulties encountered and lessons learned:

Carissa Ying Geok Teng
Many of these required many steps and spanned across different tables. It was difficult to envision how to even reach the desired output. I learnt that it is good practice to break down the function into smaller, more manageable steps, then testing to see if those functions work as expected.

For example, for function 3, I had a hard time trying to get the number of days needed to reach a funding goal. However, I realized that if I could get the total funding up to each date, I could just find the first date where the total funding met the goal. I tested the cumulative sum function on a small table of numbers before implementing it for the final function.

Nguyen Son Linh
I learnt that these functions and triggers often contain minor details that need to be tested carefully using multiple test cases. For example, I forgot to test for the second condition in Trigger 4 at first and missed that employees can still reject after 90 days, just not accept refund requests.

Nicholas Tanvis
I learned that deferred triggers made the first trigger and the first procedure able to coexist together which seemed impossible at first as they require opposite insertion orders into the Users and Backers/Creators table.

I could not find a way to store query results into a variable. So, I had to resort to calling the helper function 3 times to compute the superbacker criteria for the first function.

Venkat Vishwanth Sreenivasan
One crucial lesson I learned through this project was that in a procedure call if any insert, update and etc statements fail, the entire procedure will be reverted. Initially for the add_project procedure, after all the insertions, I calculated the num of inserts made to the table to check if all inserts has been successfully processed. If not, I created delete statements to remove them all. My initial mindset was that each insert, update and etc statements were individual transactions.

However, thanks to my team, I learnt that this checking and deletion is not required as the entire procedure call is taken as 1 transaction and not just the individual insert, update and etc statements. Hence, if any statement fails in the procedure call, the entire procedure will be reverted.