CS2102 Project (Part 2)

Group 093

Team Members

- Chua Shi Hong A0219821E
- Kok Chun Khai A0245797J
- Lian Guo Yang A0236453E
- Ng Shi Jun A0245615E

Responsibilities

- Chua Shi Hong: Triggers 11,13, Procedure 2 Function 3
- Kok Chun Khai: Triggers 1-7, 12 Procedure 1 Function 1
- Lian Guo Yang: Triggers 8-9, Procedure 3, Function 2
- Ng Shi Jun: Triggers 10,14

Table of Content

Triggers

Delivery_requests

Trigger Requirement:

- (1) Each delivery request has at least one package

Trigger Name: `delivery_requests`
Trigger Function: `check_delivery_requests()`

```
CREATE OR REPLACE FUNCTION check_delivery_requests()
RETURNS TRIGGER AS $$
BEGIN
IF NOT EXISTS (SELECT 1 FROM packages WHERE request_id = NEW.id) THEN
   RAISE EXCEPTION 'Each delivery request must have at least one package.';
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE CONSTRAINT TRIGGER delivery_requests
AFTER INSERT ON delivery_requests
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION check_delivery_requests();
```

Implementation Idea:

- Function check_delivery_requests() is defined to ensure that each delivery request has at least one package.
- The trigger delivery_requests is defined as a constraint trigger to execute the check_delivery_requests() function after each insert on the delivery_requests table.
- The trigger is defined as DEFERRABLE INITIALLY DEFERRED to allow for deferring constraint checks until the end of a transaction.
- The function checks if there exists at least one package with the same request_id as the new row inserted into delivery_requests.
- If there is no matching package, an exception is raised to prevent the insert.
- Otherwise, the trigger returns the new row to allow the insert to proceed.

Package

Trigger Requirement:

- (2) For each delivery request, the IDs of the packages should be consecutive integers starting from 1.

Trigger Name: `delivery_request_package`
Trigger Function: `check_delivery_request_packages()`

```
CREATE OR REPLACE FUNCTION check_delivery_request_packages()
RETURNS TRIGGER AS $$
DECLARE
   last_package_id INTEGER;
BEGIN
   SELECT MAX(package_id) INTO last_package_id
   FROM packages
   WHERE request_id = NEW.request_id;

   IF (last_package_id IS NULL) AND (NEW.package_id != 1) THEN
       RAISE EXCEPTION 'Package IDs for delivery request % must start from 1.', NEW.request_id;
```

```
    END IF;
    IF (last_package_id IS NOT NULL) AND (last_package_id != NEW.package_id - 1) THEN
        RAISE EXCEPTION 'Package IDs for delivery request % must be consecutive integers. The
latest packages ID for this delivery request is %', NEW.request_id, last_package_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER delivery_request_packages
BEFORE INSERT ON packages
FOR EACH ROW
EXECUTE FUNCTION check_delivery_request_packages();
```

Implementation Idea:

- Define the check_delivery_request_packages function with a RETURNS TRIGGER statement.
- Declare a variable last_package_id of type INTEGER.
- Query the packages table to find the maximum package_id for the request_id of the new row being inserted and store it in last_package_id.
- Check if the last_package_id is NULL and if the package_id of the new row being inserted is not 1, raise an exception with a custom error message indicating that the package IDs for the delivery request must start from 1.
- Check if the last_package_id is not NULL and if it is not equal to package_id - 1 of the new row being inserted, raise an exception with a custom error message indicating that the package IDs for the delivery request must be consecutive integers.
- Return the NEW row, allowing the trigger to execute.

Unsuccessful_pickups

Trigger Requirement:

- (3) For each delivery request, the IDs of the unsuccessful pickups should be consecutive integers starting from 1.
- (4) The timestamp of the first unsuccessful pickup should be after the submission_time of the corresponding delivery request. In addition, each unsuccessful pickup's timestamp should be after the previous unsuccessful pickup's timestamp (if any).

Trigger Name: `unsuccessful_pickups`
Trigger Function: `check_unsuccessful_pickups()`

```
CREATE OR REPLACE FUNCTION check_unsuccessful_pickups()
RETURNS TRIGGER AS $$
DECLARE
    last_pickup_id INTEGER;
    last_pickup_time TIMESTAMP;
BEGIN
    SELECT MAX(pickup_id), MAX(pickup_time) INTO last_pickup_id, last_pickup_time
    FROM unsuccessful_pickups
    WHERE request_id = NEW.request_id;

    -- Check if pickup ID starts from 1
    IF (last_pickup_id IS NULL) AND (NEW.pickup_id != 1) THEN
        RAISE EXCEPTION 'Unsuccessful pickup IDs for delivery request % must start from 1.',
NEW.request_id;
    END IF;

    -- Check if the current pickup ID is consecutive
    IF (last_pickup_id IS NOT NULL) AND (last_pickup_id != NEW.pickup_id - 1) THEN
        RAISE EXCEPTION 'Unsuccessful pickup IDs for delivery request % must be consecutive
integers.', NEW.request_id;
    END IF;
```

```
   -- Check if the current pickup timestamp is after the submission_time of the corresponding
delivery request
   IF NEW.pickup_time <= (SELECT submission_time FROM delivery_requests WHERE id =
NEW.request_id) THEN
       RAISE EXCEPTION 'Unsuccessful pickup timestamps for delivery request % must be after the
submission time of the corresponding delivery request.', NEW.request_id;
   END IF;

   -- Check if the current pickup timestamp is after the previous pickup timestamp (if any)
   IF (last_pickup_time IS NOT NULL) AND (last_pickup_time <= NEW.pickup_time) THEN
       RAISE EXCEPTION 'Unsuccessful pickup timestamps for delivery request % must be after the
previous one.', NEW.request_id;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER unsuccessful_pickups
BEFORE INSERT ON unsuccessful_pickups
FOR EACH ROW
EXECUTE FUNCTION check_unsuccessful_pickups();
```

Implementation Idea:

- For consecutive unsuccessful pickup_id, obtain the current maximum value of package_id, call it max_id:
    - If NULL, there is currently no packages, add as per normal
    - If Not NULL, there is at least one package, check if the package_id of the new package is equal to the package_id of max_id.
      If it is, then the package_id is consecutive, raise an exception otherwise.
- For submission_time of delivery request to be before timestamp of unsuccessful pickup, using the request_id of the unsuccessful pickup to obtain the submission_time of said request from the delivery_requests table, and compare it to the pickup_time of the unsuccessful pickup
- For the timestamps of every subsequent unsuccessful pickups to be strictly increasing, obtain the current maximum value of pickup_time, call it max_timestamp:
    - If NULL, there is currently no unsuccessful pickups, add as per normal
    - If Not NULL, there is at least one unsuccessful pickup, check if the timestamp of the new unsuccessful pickup is greater than max_timestamp.
      If it is, then timestamps inserted are strictly increasing, raise an exception otherwise.

Legs

Trigger Requirement:

- (5) For each delivery request, the IDs of the legs should be consecutive integers starting from 1.
- (6) For each delivery request, the start time of the first leg should be after the submission_time of the delivery request and the timestamp of the last unsuccessful pickup (if any).
- (7) For each delivery request, a new leg cannot be inserted if its start_time is before the end_time of the previous leg, or if the end_time of the previous leg is NULL.

Trigger 5

Trigger Name: `leg_id`

Trigger Function:`check_leg_id()`

```
CREATE OR REPLACE FUNCTION check_leg_id()
RETURNS TRIGGER AS $$
DECLARE
   last_leg_id INTEGER;
```

```
BEGIN
   SELECT MAX(leg_id) INTO last_leg_id FROM legs WHERE request_id = NEW.request_id;

   IF (last_leg_id IS NULL) AND (NEW.leg_id != 1) THEN
       RAISE EXCEPTION 'Leg IDs for delivery request % must start from 1.', NEW.request_id;
   END IF;

   IF (last_leg_id IS NOT NULL) AND (last_leg_id != NEW.leg_id - 1) THEN
       RAISE EXCEPTION 'Leg IDs for delivery request % must be consecutive integers.',
NEW.request_id;
   END IF;

RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER leg_id
BEFORE INSERT ON legs
FOR EACH ROW
EXECUTE FUNCTION check_leg_id();
```

Trigger 6 (part 1)

Trigger Name: first_leg_start_time1

Trigger Function:`check_first_leg_start_time1()`

```
CREATE OR REPLACE FUNCTION check_first_leg_start_time1()
RETURNS TRIGGER AS $$
DECLARE
subm_time TIMESTAMP;
BEGIN
   SELECT submission_time INTO subm_time FROM delivery_requests
       WHERE (id = NEW.request_id);

   IF (NEW.leg_id = 1) THEN
     IF (NEW.start_time <= subm_time) THEN
       RAISE EXCEPTION 'Invalid start time for first leg, start_time of first leg must be after
the time the delivery request was placed';
     END IF;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER first_leg_start_time1
AFTER INSERT ON legs
FOR EACH ROW
EXECUTE FUNCTION check_first_leg_start_time1();
```

Trigger 6(part 2)

Trigger Name: first_leg_start_time2

Trigger Function:`check_first_leg_start_time2()`

```
CREATE OR REPLACE FUNCTION check_first_leg_start_time2()
RETURNS TRIGGER AS $$
DECLARE
   last_unsuccessful_pickup_time TIMESTAMP;
BEGIN
   SELECT MAX(pickup_time) INTO last_unsuccessful_pickup_time FROM unsuccessful_pickups WHERE
```

```
request_id = NEW.request_id;
   IF (NEW.leg_id = 1) THEN
       IF (last_unsuccessful_pickup_time IS NOT NULL) AND (NEW.start_time <
last_unsuccessful_pickup_time) THEN
           RAISE EXCEPTION 'Invalid start time for first leg, start_time of first leg cannot be
before last unsuccessful pickup time';
       END IF;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER first_leg_start_time2
AFTER INSERT ON legs
FOR EACH ROW
EXECUTE FUNCTION check_first_leg_start_time2();
```

Trigger 7

Trigger Name: leg_start_and_end_time

Trigger Function:`check_leg_start_and_end_time()`// Constraint 7

```
CREATE OR REPLACE FUNCTION check_leg_start_and_end_time()
RETURNS TRIGGER AS $$
DECLARE
   last_leg_end_time TIMESTAMP;
BEGIN
   SELECT end_time INTO last_leg_end_time FROM legs WHERE request_id = NEW.request_id AND leg_id =
NEW.leg_id - 1;

   IF (NEW.leg_id > 1) AND (last_leg_end_time IS NULL) THEN
       RAISE EXCEPTION 'Invalid leg, end time of previous leg must not be NULL';
   END IF;
   IF (NEW.leg_id > 1) AND (NEW.start_time <= last_leg_end_time) THEN
       RAISE EXCEPTION 'Invalid start time for leg, must not be before end time of previous leg';
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER leg_start_and_end_time
AFTER INSERT ON legs
FOR EACH ROW
EXECUTE FUNCTION check_leg_start_and_end_time();
```

Implementation Idea:

- (5) For consecutive leg_id, obtain the current maximum value of leg_id of that , call it max_id:
  - If NULL, there is currently no packages, add as per normal
  - If not NULL, there is at least one package, check if the package_id of the new package is equal to the package_id of max_id.
    If it is, then the package_id is consecutive, raise an exception otherwise.
- (6.1) For start_time of each first leg to be after submission_time of its delivery request, obtain the submission_time of the delivery request using the request_id form the delivery_requests table, call it req_time:
  - If not NULL, check if the leg_id is 1, if it then check if the start_time is before req_time, raise an exception if it is
- (6.2) For start_time of each first_leg to be after the last unsuccesful_pickup, obtain the maximum time_stamp of unsuccessful pickups, call it max_unsuccesful_time:
  - If NULL, there are no unsuccessful pickups, add per normal
  - If not NULL, there are unsuccessful pickups, check if the time of delivery of the first leg is after the last unsuccessful pickup timestamp, raise an exception if not
- (6.3) For timestamp of previous leg to be not NULL and and start_time of the new leg to be after end_time of previous leg, first check if the leg is a first leg

- If it's the first leg, just add as normal
- If its not the first leg, check two things:
    - If the end_time of the previous leg is NULL, raise exception if it is
    - If the start_time of the new leg is <= the end_time of the previous leg, raise exception if it is

Unsuccessful_deliveries

Trigger Requirement:

- (8) The timestamp of each unsuccessful_delivery should be after the start_time of the corresponding leg.
- (9) For each delivery request, there can be at most three unsuccessful_deliveries.

Trigger Name: `unsuccessful_deliveries`
Trigger Function: `check_unsuccessful_deliveries()`

```
CREATE OR REPLACE FUNCTION check_unsuccessful_deliveries()
RETURNS TRIGGER AS $$
DECLARE
   curr_start_time TIMESTAMP;
   unsuccessful_time TIMESTAMP;
   unsuccessful_count INTEGER;
BEGIN
   -- Get the start time of the corresponding leg
   SELECT start_time INTO curr_start_time
   FROM legs
   WHERE request_id = NEW.request_id AND leg_id = NEW.leg_id;

   -- Constraint 8: Check if the unsuccessful delivery timestamp is after the start time
   IF NEW.attempt_time < curr_start_time THEN
       RAISE EXCEPTION 'The timestamp of unsuccessful_delivery for delivery_requst % should be
after the start_time of the corresponding leg.', NEW.request_id;
   END IF;

   -- Count the number of unsuccessful deliveries for the request
   SELECT COUNT(*) INTO unsuccessful_count
   FROM unsuccessful_deliveries
   WHERE request_id = NEW.request_id;

   -- Constraint 9: Check if there are more than three unsuccessful deliveries for the request
   IF unsuccessful_count >= 3 THEN
       RAISE EXCEPTION 'For delivery request ID=%, there is currently % unsuccesful deliveries.
There can be at most 3 unsuccessful_deliveries for each delivery_request.', NEW.request_id,
unsuccessful_count;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER unsuccessful_deliveries
BEFORE INSERT ON unsuccessful_deliveries
FOR EACH ROW
EXECUTE FUNCTION check_unsuccessful_deliveries();
```

Implementation Idea:

- Constraint 8:
    - Retrieve the start time of the corresponding leg for the delivery request from the legs table using the request_id and leg_id of the new unsuccessful delivery row.

        - Compare the start time with the attempt_time of the new unsuccessful delivery row.
        - If the attempt_time is before the start time, raise an exception with a message indicating that the timestamp of the unsuccessful delivery should be after the start time of the corresponding leg.
- Constraint 9:
    - Count the number of unsuccessful deliveries for the delivery request from the unsuccessful_deliveries table using the request_id of the new unsuccessful delivery row.
    - If the count is greater than or equal to 3, raise an exception with a message indicating that there are currently more than 3 unsuccessful deliveries for the delivery request.

Cancelled_requests

Trigger Requirement:

- (10) The cancel_time of a cancelled request should be after the submission_time of the corresponding delivery request.

Trigger Name: `cancelled_requests`
Trigger Function:

```
CREATE OR REPLACE FUNCTION check_cancelled_requests()
RETURNS TRIGGER AS $$
DECLARE
   sub_time TIMESTAMP;
BEGIN
   SELECT submission_time INTO sub_time
   FROM delivery_requests
   WHERE delivery_requests.id = NEW.id;
   IF (sub_time IS NOT NULL) AND (sub_time >= NEW.cancel_time) THEN
       RAISE EXCEPTION 'For request ID=%, the cancel_time should be after the submission_time
of the corresponding delivery request.', NEW.id;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER cancelled_requests
BEFORE INSERT ON cancelled_requests
FOR EACH ROW
EXECUTE FUNCTION check_cancelled_requests();
```

Implementation Idea:

- Create a function named "check_cancelled_requests"
- Declare a variable "sub_time" of type timestamp
- Select the "submission_time" from the "delivery_requests" table where the id matches the "id" of the new row being inserted and assign it to "sub_time"
- Check if "sub_time" is not null and greater than or equal to the "cancel_time" of the new row being inserted
- If the condition is true, raise an exception with a custom error message, otherwise, return the new row.

Return_legs

Trigger Requirement:

- (11) For each delivery request, the first return_leg's ID should 1, the second return_leg's ID should be 2, and so on.
- (12) For a delivery request, the first return_leg cannot be inserted if
    - (i) there is no existing leg for the delivery request or
    - (ii) the last existing leg's end_time is after the start_time of the return_leg. In addition, the return_leg's start_time should be after the cancel_time of the request (if any).
- (13) For each delivery request, there can be at most three unsuccessful_return_deliveries.

Trigger 11

Trigger Name: `check_return_leg_id`
Trigger Function: `return_leg_id()`

```
CREATE OR REPLACE FUNCTION return_leg_id()
RETURNS TRIGGER AS $$
DECLARE
   max_return_leg_id INTEGER;
BEGIN
   SELECT MAX(leg_id) INTO max_return_leg_id
   FROM return_legs
   WHERE return_legs.request_id = NEW.request_id;

   IF max_return_leg_id IS NULL THEN
       IF NEW.leg_id <> 1 THEN
           RAISE EXCEPTION 'First return_leg ID must be 1';
       END IF;
   END IF;

   IF max_return_leg_id IS NOT NULL THEN
       IF NEW.leg_id <> (max_return_leg_id + 1) THEN
           RAISE EXCEPTION 'Every new return_leg ID has to be exactly one more than the
previous one, the latest return_leg ID for delivery_request ID=% is %', NEW.request_id,
max_return_leg_id;
       END IF;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_return_leg_id
BEFORE INSERT ON return_legs
FOR EACH ROW
EXECUTE FUNCTION return_leg_id();
```

Trigger 12
Trigger Name: `consistency_return_legs_insertion()`
Trigger Function: `consistency_return_legs_insertion()`

```
CREATE OR REPLACE FUNCTION check_consistency_return_legs_insertion()
RETURNS TRIGGER AS $$
DECLARE
   existing_request_id INTEGER;
   last_existing_leg_end_time TIMESTAMP;
   existing_leg_id INTEGER;
   existing_cancel_time TIMESTAMP;
BEGIN

   -- There are no existing legs for this delivery_request_ID
   SELECT request_id INTO existing_request_id
   FROM legs
   WHERE legs.request_id = NEW.request_id;

   IF existing_request_id IS NULL THEN
       RAISE EXCEPTION 'There is no existing leg for delivery request ID=%', NEW.request_ID;
   END IF;
```

```
   -- Last existing leg's end_time should not be after the start_time of the return_leg
   SELECT end_time INTO last_existing_leg_end_time
   FROM legs
   WHERE request_id = NEW.request_id
   ORDER BY leg_id DESC LIMIT 1;

   IF (last_existing_leg_end_time IS NOT NULL) AND (NEW.start_time <=
last_existing_leg_end_time) THEN
       RAISE EXCEPTION 'The start_time of a return leg cannot be earlier than the end_time of
the last leg.';
   END IF;

   -- The return_leg's start_time should be after the cancel_time of the request (if any).
   SELECT cancel_time INTO existing_cancel_time
   FROM cancelled_requests
   WHERE cancelled_requests.id = NEW.request_id;

   IF existing_cancel_time IS NOT NULL THEN
       IF NEW.start_time <= existing_cancel_time THEN
           RAISE EXCEPTION 'The start_time of a return_leg must be after the cancel time of the
delivery request with ID=%', NEW.request_ID;
       END IF;
   END IF;

   RETURN NEW;

END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER consistency_return_legs_insertion
BEFORE INSERT ON return_legs
FOR EACH ROW
EXECUTE FUNCTION check_consistency_return_legs_insertion();
```

Trigger 13

- Trigger Name: `check_at_most_three_unsuccessful_return_deliveries`
- Trigger Function: `at_most_three_unsuccessful_return_deliveries()`

```
CREATE OR REPLACE FUNCTION check_at_most_three_unsuccessful_return_deliveries()
RETURNS TRIGGER AS $$
DECLARE
   unsuccessful_count INTEGER;
BEGIN
-- Count the number of unsuccessful deliveries for the request
   SELECT COUNT(*) INTO unsuccessful_count
   FROM unsuccessful_return_deliveries
   WHERE unsuccessful_return_deliveries.request_id = NEW.request_id;

   IF unsuccessful_count >= 3 THEN
       RAISE EXCEPTION 'For delivery request ID=%, there can be at most 3
unsuccessful_return_deliveries.', NEW.request_id;
   END IF;
   RETURN NEW;

END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER at_most_three_unsuccessful_return_deliveries
BEFORE INSERT ON return_legs
FOR EACH ROW
EXECUTE FUNCTION check_at_most_three_unsuccessful_return_deliveries();
```

Implementation Idea:

- Constraint 11
  - Define a variable max_return_leg_id to store the maximum leg ID for the given request ID in the return_legs table.
  - Use a SELECT statement to retrieve the maximum leg ID value for the given request ID and store it in the max_return_leg_id variable.
  - If the max_return_leg_id is NULL, then the new leg ID should be 1. Raise an exception if this is not the case.
  - If the max_return_leg_id is not NULL, then the new leg ID should be exactly one more than the previous maximum leg ID. Raise an exception if this is not the case, else return the new row.
- Constraint 12
  - First check whether the delivery request exists in the table legs
  - Then, we check whether the last existing leg(if it exists) end time is before the start time of this return leg. If not, we raise an exception
- Constraint 13
  - From unsuccessful_return_deliveries table, we count how many rows are there for this delivery request, if there are more than 3 unsuccessful return deliveries, we raise an exception .

Unsuccessful_return_deliveries

Trigger Requirement:

- (14) The timestamp of each unsuccessful_return_delivery should be after the start_time of the corresponding return_leg.

Trigger Name: `unsuccessful_return_deliveries`
Trigger Function: `check_unsuccessful_return_deliveries()`

```
CREATE OR REPLACE FUNCTION check_unsuccessful_return_deliveries()
RETURNS TRIGGER AS $$
DECLARE
   s_time TIMESTAMP;
BEGIN
   SELECT start_time INTO s_time
   FROM return_legs
   WHERE return_legs.request_id = NEW.request_id;

   IF (s_time IS NOT NULL) AND (s_time >= NEW.attempt_time) THEN
       RAISE EXCEPTION 'For unsuccessful_return_deliveries ID=%, the attempt_time should be
after the start_time of corresponding return_leg.', NEW.request_id;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER unsuccessful_return_deliveries
BEFORE INSERT ON unsuccessful_return_deliveries
FOR EACH ROW
EXECUTE FUNCTION check_unsuccessful_return_deliveries();
```

Implementation Idea:

- Retrieve the start_time of the corresponding return leg for the delivery request specified in the NEW row.
- If the start_time exists and is greater than or equal to the attempt_time specified in the NEW row, raise an exception with a message indicating that the attempt_time should be after the start_time of the corresponding return leg, else return NEW

Procedures

submit_request

Procedure:

```
CREATE OR REPLACE PROCEDURE submit_request(
   customer_id INTEGER, evaluator_id INTEGER,
   pickup_addr TEXT, pickup_postal TEXT,
   recipient_name TEXT, recipient_addr TEXT,
   recipient_postal TEXT, submission_time TIMESTAMP,
   package_num INTEGER, reported_height INTEGER[],
   reported_width INTEGER[], reported_depth INTEGER[],
   reported_weight INTEGER[], content TEXT[],
   estimated_value NUMERIC[]
) AS $$

DECLARE
   curr_request_id INTEGER;
   package_id INTEGER;
BEGIN
   -- Insert delivery request
   INSERT INTO delivery_requests (
       customer_id, evaluater_id, status,
       pickup_addr, pickup_postal, recipient_name,
       recipient_addr, recipient_postal,
       submission_time)
   VALUES (
       customer_id, evaluator_id, 'submitted',
       pickup_addr, pickup_postal, recipient_name,
       recipient_addr, recipient_postal,
       submission_time
   ) RETURNING id INTO curr_request_id;

   -- Insert packages for the delivery request
   FOR i IN 1..package_num LOOP
       INSERT INTO packages (
           request_id, package_id, reported_height,
           reported_width, reported_depth,
           reported_weight, content, estimated_value)
       VALUES (
           curr_request_id, i, reported_height[i],
           reported_width[i], reported_depth[i],
           reported_weight[i], content[i],
           estimated_value[i]);
   END LOOP;

   -- Set actual dimensions to NULL for each package
   UPDATE packages
   SET
       actual_height = NULL,
       actual_width = NULL,
       actual_depth = NULL,
       actual_weight = NULL
   WHERE
```

```
        packages.request_id = curr_request_id;

    -- Set pickup_date, num_days_needed, and price to NULL for the delivery request
    UPDATE delivery_requests
    SET
        pickup_date = NULL,
        num_days_needed = NULL,
        price = NULL
    WHERE
        id = curr_request_id;
END;
$$ LANGUAGE plpgsql;
```

Implementation Idea:

- Insert a new row into the delivery_requests table with the given input parameters, setting the status to 'submitted' and returning the ID of the new request into a variable curr_request_id.
- Insert package_num rows into the packages table for the new delivery request, with each row corresponding to a package and containing the reported height, width, depth, weight, content, and estimated value.
- Use a loop to iterate over the packages and insert them into the packages table.
- Set the actual dimensions of each package in the new request to NULL in the packages table.
- Set the pickup_date, num_days_needed, and price columns of the new request to NULL in the delivery_requests table.

resubmit_request

Procedure:

```
CREATE OR REPLACE PROCEDURE resubmit_request(request_id INTEGER, evaluator_id INTEGER, submission_time
TIMESTAMP, reported_height INTEGER[] , reported_width INTEGER[], reported_depth INTEGER[], reported_weight
INTEGER[])
AS $$
DECLARE
    count INTEGER;
    r_id INTEGER;
    cus_id INTEGER;
    pu_addr TEXT;
    pu_postal TEXT;
    reci_name TEXT;
    reci_addr TEXT;
    reci_postal TEXT;
    con TEXT;
    est_value NUMERIC;
BEGIN
    SELECT customer_id, pickup_addr, pickup_postal, recipient_name, recipient_addr, recipient_postal INTO
cus_id, pu_addr, pu_postal, reci_name, reci_addr, reci_postal
    FROM delivery_requests
    WHERE delivery_requests.id = resubmit_request.request_id;

    SELECT COUNT(*) INTO count
    FROM packages
    WHERE packages.request_id = resubmit_request.request_id;

    INSERT INTO delivery_requests (customer_id, evaluater_id, status, pickup_addr, pickup_postal,
recipient_name, recipient_addr, recipient_postal, submission_time)
    VALUES (cus_id, evaluator_id, 'submitted', pu_addr, pu_postal, reci_name, reci_addr, reci_postal,
submission_time)
    RETURNING id INTO r_id;

    FOR i IN 1..count LOOP
        INSERT INTO packages (request_id, package_id, reported_height, reported_width, reported_depth,
reported_weight, content, estimated_value)
```

```
        SELECT r_id, package_id, reported_height, reported_width, reported_depth, reported_weight, content,
estimated_value
        FROM packages
        WHERE request_id = request_id AND package_id = i;

        UPDATE packages
        SET reported_height = resubmit_request.reported_height[i],
            reported_width = resubmit_request.reported_width[i],
            reported_depth = resubmit_request.reported_depth[i],
            reported_weight = resubmit_request.reported_weight[i],
            actual_height = NULL,
            actual_width = NULL,
            actual_depth = NULL,
            actual_weight = NULL
        WHERE request_id = r_id AND package_id = i;
    END LOOP;

    -- Set pickup_date, num_days_needed, and price to NULL for the delivery request
    UPDATE delivery_requests
    SET pickup_date = NULL, num_days_needed = NULL, price = NULL
    WHERE id = r_id;

    END;
$$ LANGUAGE plpgsql;
```

Implementation Idea:

- Insert a new record into the delivery_requests table with the provided parameters.
- Retrieve the new id value from the delivery_requests table into the curr_request_id variable using RETURNING clause.
- Insert package details for the new delivery request into the packages table with curr_request_id and package details provided in the procedure parameters.
- Set the actual dimensions to NULL for each package associated with the curr_request_id in the packages table.
- Set pickup_date, num_days_needed, and price to NULL for the delivery request with id equal to curr_request_id in the delivery_requests table.

insert_leg

Procedure: `insert_leg`

```
CREATE OR REPLACE PROCEDURE insert_leg(request_id INTEGER, handler_id INTEGER, start_time
TIMESTAMP, destination_facility INTEGER) AS $$
DECLARE
    curr_leg_id INTEGER;
BEGIN
    SELECT COALESCE(MAX(legs.leg_id), 0) + 1 INTO curr_leg_id
    FROM legs
    WHERE legs.request_id = insert_leg.request_id;

    INSERT INTO legs (request_id, leg_id, handler_id, start_time, destination_facility,
end_time)
    VALUES (request_id, curr_leg_id, handler_id, start_time, destination_facility, NULL);
END;
$$ LANGUAGE plpgsql;
```

Implementation Idea:

- Define a new procedure called insert_leg that takes four input parameters: request_id, handler_id, start_time, and destination_facility.
- Declare a local variable curr_leg_id of type INTEGER to store the ID of the new leg that will be inserted.
- Use a SELECT statement with the MAX function to find the highest leg_id for the given request_id in the legs table, and store the result in curr_leg_id.
- Increment curr_leg_id by one to get the ID for the new leg.
- Use an INSERT statement to add a new row to the legs table with the values of the input parameters and the new leg_id.

- Set the end_time column to NULL.

Functions

view_trajectory

Function: `view_trajectory`

```
CREATE OR REPLACE FUNCTION view_trajectory (request_id INTEGER)
RETURNS TABLE (source_address TEXT, destination_address TEXT, start_time TIMESTAMP, end_time
TIMESTAMP)
AS $$
BEGIN
   RETURN QUERY
   WITH return_legs_path AS (
       SELECT
       l1_f.address as source_address,
       COALESCE(l2_f.address, (SELECT pickup_addr FROM delivery_requests WHERE
delivery_requests.id = view_trajectory.request_id)) as destination_address,
       l1.start_time,
       l1.end_time
       FROM return_legs as l1
           LEFT OUTER JOIN return_legs as l2 ON l1.request_id = l2.request_id AND l1.leg_id =
l2.leg_id - 1
           FULL OUTER JOIN facilities as l2_f ON l2_f.id = l2.source_facility
           FULL OUTER JOIN facilities as l1_f ON l1_f.id = l1.source_facility
       WHERE l1.request_id = view_trajectory.request_id
   ), legs_path AS (
       SELECT
       COALESCE(l1_f.address, (SELECT pickup_addr FROM delivery_requests WHERE
delivery_requests.id = view_trajectory.request_id)) as source_address,
       COALESCE(l2_f.address, (SELECT recipient_addr FROM delivery_requests WHERE
delivery_requests.id = view_trajectory.request_id)) as destination_address,
       l2.start_time,
       l2.end_time
       FROM legs as l1
           FULL OUTER JOIN legs as l2 ON l1.request_id = l2.request_id AND l1.leg_id =
l2.leg_id - 1
           FULL OUTER JOIN facilities as l2_f ON l2_f.id = l2.destination_facility
           FULL OUTER JOIN facilities as l1_f ON l1_f.id = l1.destination_facility
       WHERE l2.request_id = view_trajectory.request_id
   )

   (SELECT *
   FROM (
       (SELECT * FROM legs_path)
       UNION
       (SELECT * FROM return_legs_path)) t
   ORDER BY start_time ASC);
END
$$ LANGUAGE plpgsql;
```

Implementation Idea:

- The function takes a single argument: request_id of type INTEGER.
- The function returns a table with columns source_address (of type TEXT), destination_address (of type TEXT), start_time (of type TIMESTAMP), and end_time (of type TIMESTAMP).
- The function queries the legs and return_legs tables to retrieve information about the legs and return legs of the delivery request.

- It joins the legs and return_legs tables with the facilities table twice, once for the source facility and once for the destination facility, to get the source and destination addresses for each leg.
- If there are no return legs for the delivery request, the function only queries the legs table.
- If there are return legs for the delivery request, the function queries both the legs and return_legs tables, and then combines the results using a UNION.
- The function orders the combined results by start_time in ascending order.
- Finally, the function returns the combined results as the output of the function.

get_top_delivery_person

Function:

```
CREATE OR REPLACE FUNCTION get_top_delivery_persons(k INTEGER)
RETURNS TABLE (
   employee_id INTEGER
)
AS $$
BEGIN
   RETURN QUERY
       SELECT delivery_staff.id as employee_id
       FROM (
           SELECT handler_id
           FROM legs
           UNION ALL
           SELECT handler_id
           FROM return_legs
           UNION ALL
           SELECT handler_id
           FROM unsuccessful_pickups
       ) trips
       RIGHT JOIN delivery_staff ON trips.handler_id = delivery_staff.id
       GROUP BY delivery_staff.id
       ORDER BY COALESCE(COUNT(trips.handler_id), 0) DESC, delivery_staff.id ASC
       LIMIT k;
END;
$$ LANGUAGE plpgsql;
```

Implementation Idea:

- The get_top_delivery_persons function takes one integer parameter k as the number of top delivery persons to return.
- The function returns a table with one column named employee_id and integer data type.
- In the function body, a RETURN QUERY statement is used to return a query result set.
- The query retrieves all handler_ids from the legs, return_legs, and unsuccessful_pickups tables using UNION ALL.
- The trip's result set is then right-joined with the delivery_staff table on the handler_id column to retrieve all delivery staff.
- The result set is grouped by delivery_staff.id and ordered first by the count of handler_ids (i.e., the number of trips) in descending order, and then by delivery_staff.id in ascending order.
- Finally, the result set is limited to k rows and only the employee_id column is returned.

get_top_connections

Function:

```
CREATE OR REPLACE FUNCTION get_top_connections(k INTEGER)
RETURNS TABLE (
   source_facility_id INTEGER,
   destination_facility_id INTEGER
) AS $$
BEGIN
```

```
    RETURN QUERY
    SELECT r2.source_facility_id, r2.destination_facility_id
    FROM (
        SELECT r.source_facility_id, r.destination_facility_id, COUNT(*) as occur
        FROM (
            SELECT
            A.destination_facility as source_facility_id,
            B.destination_facility as destination_facility_id
            FROM legs A, legs B
            WHERE A.request_id = B.request_id
            AND A.leg_id = (B.leg_id - 1)
            UNION ALL

            SELECT
            A.source_facility as source_facility_id,
            B.source_facility as destination_facility_id
            FROM return_legs A, return_legs B
            WHERE A.request_id = B.request_id
            AND A.leg_id = (B.leg_id - 1)
        ) as r
        WHERE r.source_facility_id IS NOT NULL AND r.destination_facility_id IS NOT NULL
        GROUP BY r.source_facility_id, r.destination_facility_id
        ORDER BY occur DESC, r.source_facility_id ASC, r.destination_facility_id ASC
        LIMIT k
    ) as r2;
END;
$$ LANGUAGE plpgsql;
```

Implementation Idea:
First, we find all the connections between legs. The destination facility of leg 1 is the source facility of leg 2, so we try to find all these occurrences in every delivery request. We do the same thing on return_legs and UNION ALL them. Then we count all the occurrences of (S, T), and rank them by the occurrence in descendant order.

Difficulties Encountered

1. Code testing: it is difficult to test the trigger, procedures and functions separately. As we need to initialize the database from scratch, insert dummy data subsequently in order to test the trigger. Even if we did the test, we might miss some edge cases that we weren't aware of.
2. NULL cases: when there are some cases where the data inserted is NULL, we are not sure whether the NULL case would fail the trigger or not, as the questions did not explicitly state whether the trigger should happen on a certain NULL case.

Lessons Learned

1. We should keep the trigger or function simple: It is important to keep the trigger or function as simple as possible. This will make it easier to debug and maintain these trigger functions in the future.

2. We should test the trigger or function thoroughly: It is important to test the trigger or function as thoroughly as possible. This will help to ensure that it works as expected and does not cause any unintended consequences.

3. We should follow good practices on naming trigger names and trigger function names: when writing triggers and functions, we should use descriptive names for variables and functions, comment the code for easier understanding , and use consistent format of the code.