

# 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 Procedure 1

## Table of Content

<b>Triggers.....</b>	<b>2</b>
Delivery_requests.....	2
Package.....	2
Unsuccessful_pickups.....	3
Legs.....	4
Unsuccessful_deliveries.....	7
Cancelled_requests.....	8
Return_legs.....	8
Unsuccessful_return_deliveries.....	11
<b>Procedures.....</b>	<b>12</b>
submit_request.....	12
resubmit_request.....	13
insert_leg.....	14
<b>Functions .....</b>	<b>15</b>
view_trajectory.....	15
get_top_delivery_person.....	16
get_top_connections.....	16
<b>Difficulties Encountered.....</b>	<b>18</b>
<b>Lessons Learned.....</b>	<b>18</b>

# 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 unsuccessful\_pickup, obtain the maximum time\_stamp of unsuccessful pickups, call it max\_unsuccessful\_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_request % 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 % unsuccessesful 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, evaluator_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, evaluator_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 =
12.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 =
12.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.