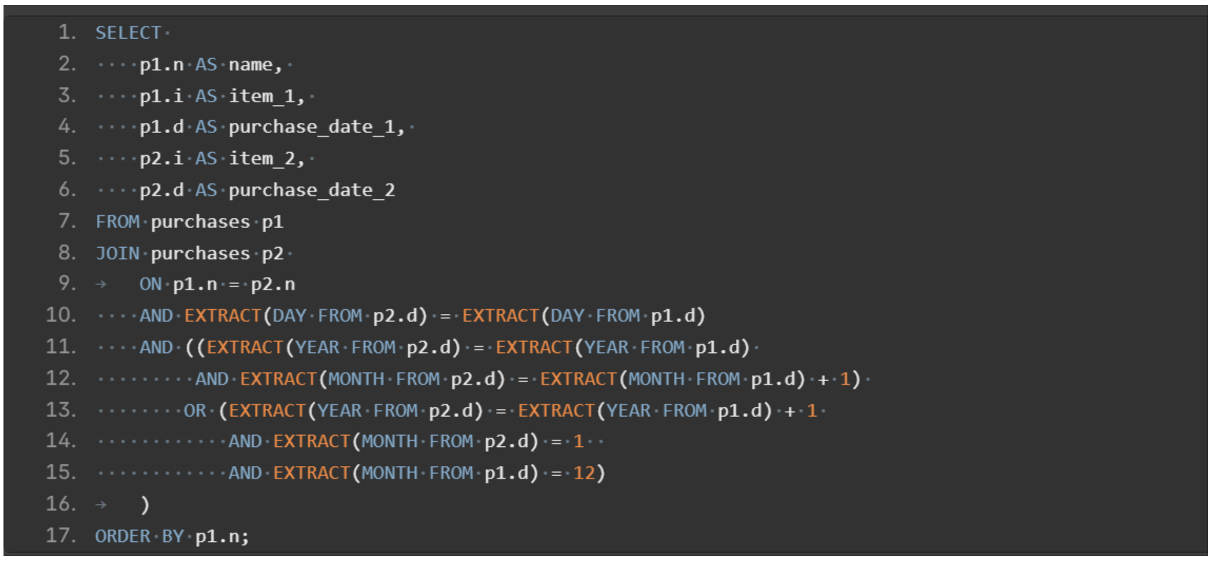
HW1 instructions: SQL queries

Completion requirements

Use the HW1.sql file given for the table creation and data insertion.
Create the SELECT queries that produce the following results:

a) For each customer: articles that the customer bought at the same day of month for two consecutive months, e.g. 16.7.2019 and 16.6.2019.
Output: The output columns should be: name | item_1 | purchase_date_1 | item_2 | purchase_date_2
and the results should be ordered by the customer name.

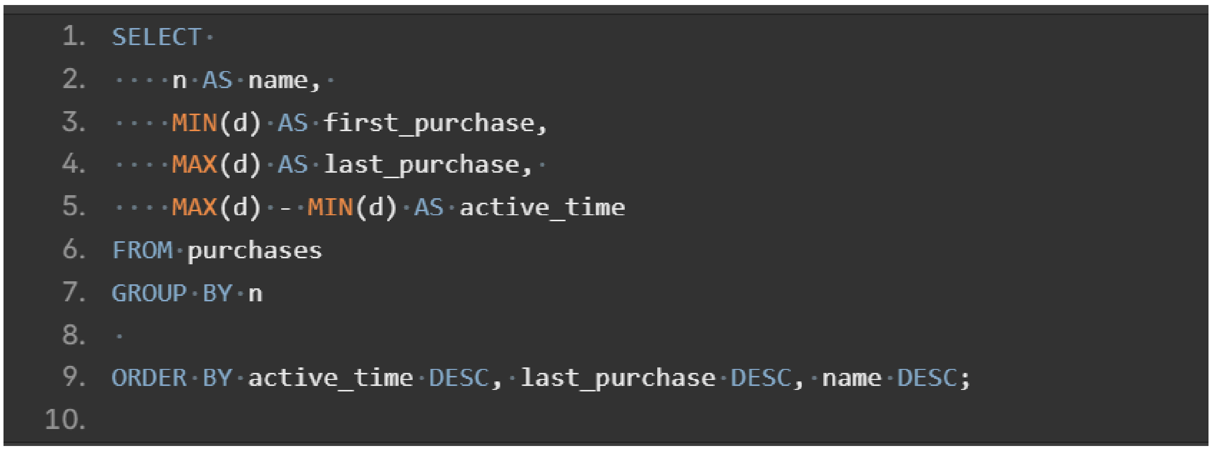


```sql
SELECT 
    p1.n AS name, 
    p1.i AS item_1, 
    p1.d AS purchase_date_1, 
    p2.i AS item_2, 
    p2.d AS purchase_date_2
FROM purchases p1
JOIN purchases p2 
    ON p1.n = p2.n
    AND EXTRACT(DAY FROM p2.d) = EXTRACT(DAY FROM p1.d)
    AND ((EXTRACT(YEAR FROM p2.d) = EXTRACT(YEAR FROM p1.d) 
         AND EXTRACT(MONTH FROM p2.d) = EXTRACT(MONTH FROM p1.d) + 1) 
        OR (EXTRACT(YEAR FROM p2.d) = EXTRACT(YEAR FROM p1.d) + 1 
            AND EXTRACT(MONTH FROM p2.d) = 1  
            AND EXTRACT(MONTH FROM p1.d) = 12)
    )
ORDER BY p1.n;
```

##

b) For each customer: timeslot in which the customer has been active,i.e. number of days between the first and the last purchases
Output: The output columns should be: name | first_purchase| last_purchase | active_time

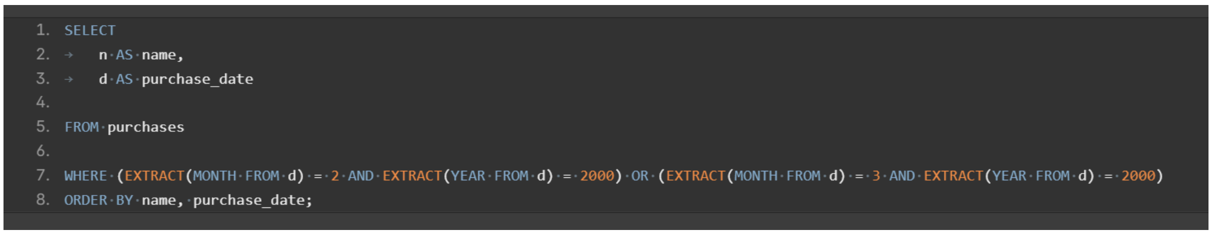
and the results should be ordered by the active time, last purchase and name in descending order.



```sql
SELECT 
    n AS name, 
    MIN(d) AS first_purchase,
    MAX(d) AS last_purchase, 
    MAX(d) - MIN(d) AS active_time
FROM purchases
GROUP BY n
 
ORDER BY active_time DESC, last_purchase DESC, name DESC;

```

###

c) All purchases of customers in February and March of leap years
Output: The output columns should be: name | purchase_date
and the results should be ordered by the customer name and purchase date.



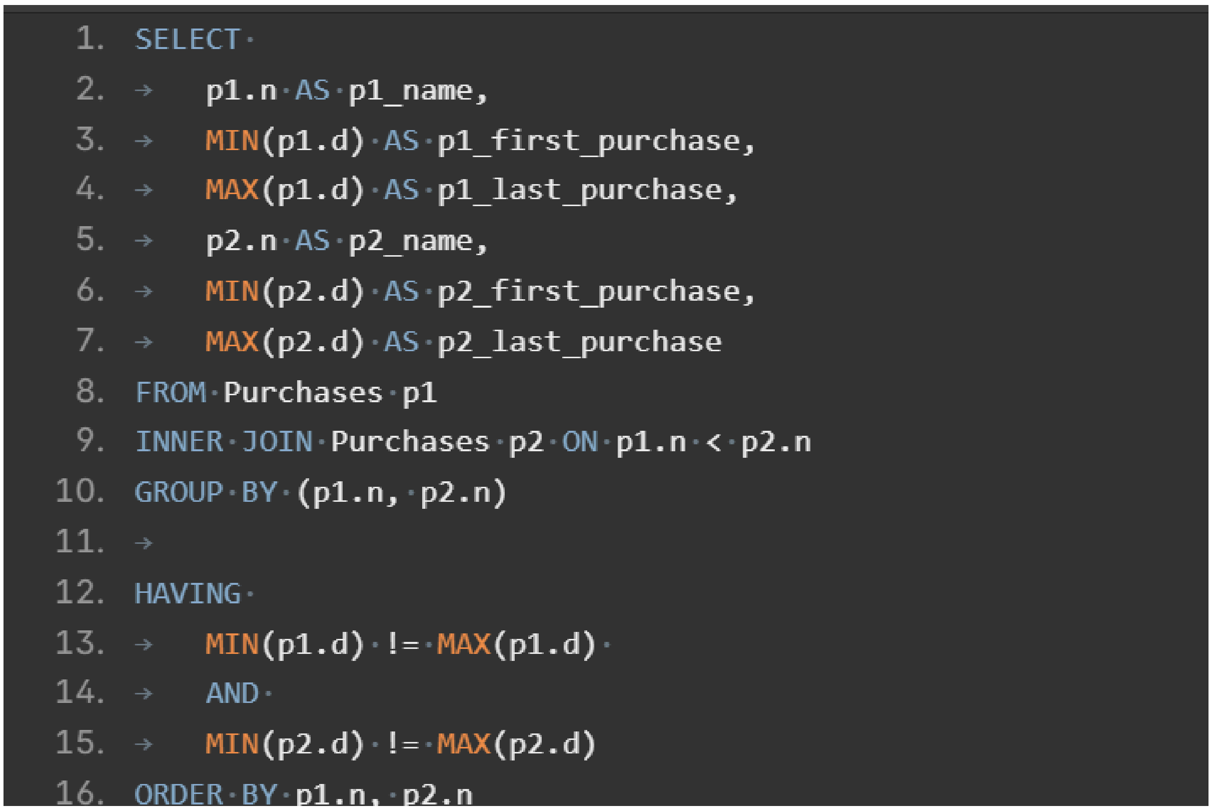
```sql
SELECT
	n AS name,
	d AS purchase_date

FROM purchases

WHERE (EXTRACT(MONTH FROM d) = 2 AND EXTRACT(YEAR FROM d) = 2000) OR (EXTRACT(MONTH FROM d) = 3 AND EXTRACT(YEAR FROM d) = 2000)
ORDER BY name, purchase_date;
```

###

d) Get pairs of customers - each with more than a single purchase - that have overlapping active purchase periods.

For example, Customer_1 buys something on 10.1.2019 and 10.3.2019. Customer_2 buys something on 20.1.2019 and 21.1.2019, now they have overlapping purchase periods.
Output: The output columns should be: p1_name | p1_first_purchase | p1_last_purchase |

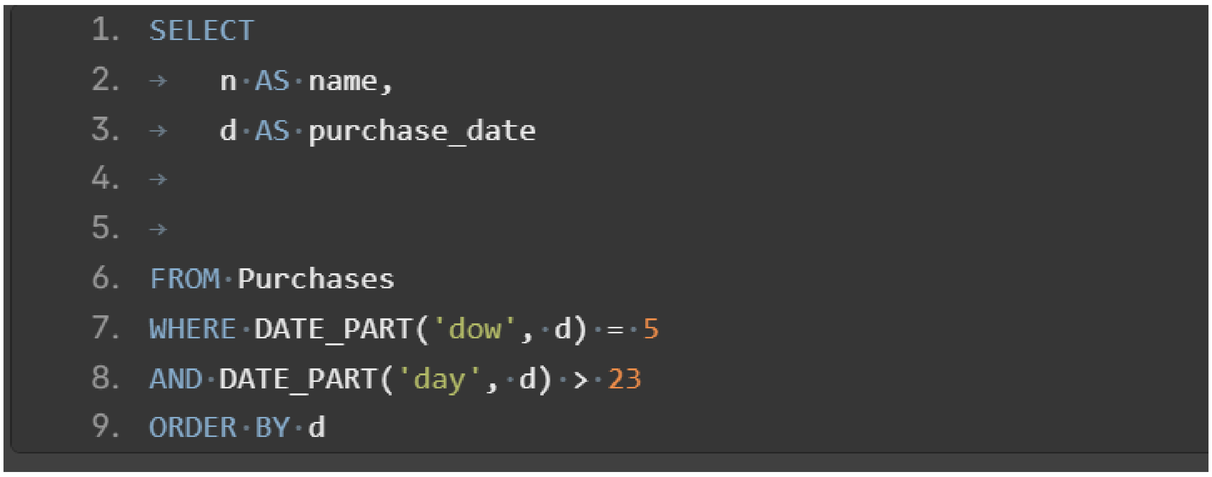
p2_name | p2_first_purchase | p2_last_purchase
and the results should be ordered by the customer 1 and customer 2 name in ascending order.



```
SELECT
    p1.n AS p1_name,
    MIN(p1.d) AS p1_first_purchase,
    MAX(p1.d) AS p1_last_purchase,
    p2.n AS p2_name,
    MIN(p2.d) AS p2_first_purchase,
    MAX(p2.d) AS p2_last_purchase
FROM Purchases p1
INNER JOIN Purchases p2 ON p1.n < p2.n
GROUP BY (p1.n, p2.n)
    
HAVING
    MIN(p1.d) != MAX(p1.d)
    AND
    MIN(p2.d) != MAX(p2.d)
ORDER BY p1.n, p2.n
```

##

e) All purchases of the last Fridays of a month.
Output: The output columns should be: name | purchase_date
and the results should be ordered by the purchase date.



```
SELECT
    n AS name,
    d AS purchase_date
    
    
FROM Purchases
WHERE DATE_PART('dow', d) = 5
AND DATE_PART('day', d) > 23
ORDER BY d
```

###

Each is 2 %, equals to 10 %

The home work will be submitted through CodeGrade in separate activities.

Table structure (you can find the SQL file in the homework section):

```
CREATE TABLE purchases (
  id SERIAL,
  n varchar(255) NOT NULL,
  d DATE default NULL,
  i varchar(50) NOT NULL,
  PRIMARY KEY (id)
);
```

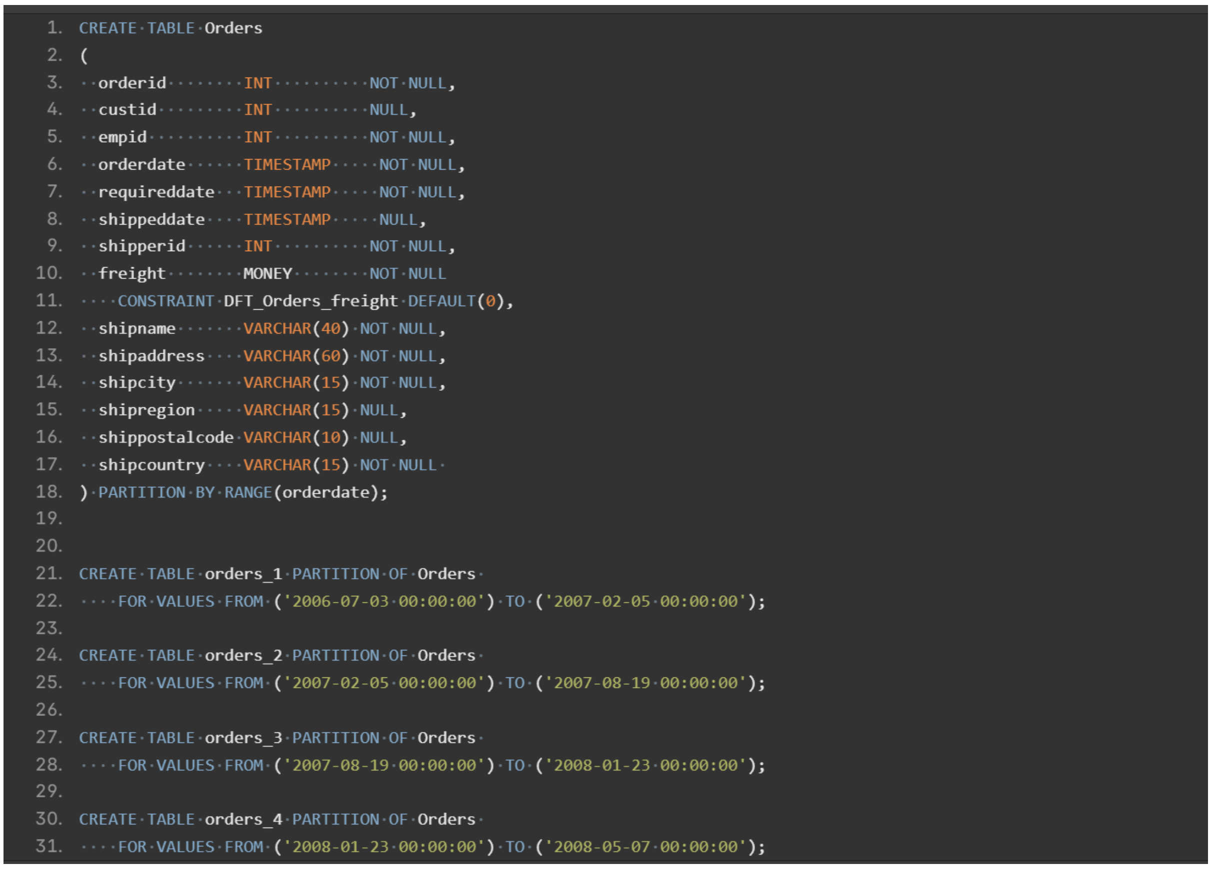
HW2 instructions: Partitions

Completion requirements

NOTE! Remember to empty your table / partitions when adding constraints when testing. So in short, create the partitions before inserting data.
Remember to use Default partitions as well.
a) Partition the Orders table using orderdate with the following constraints:

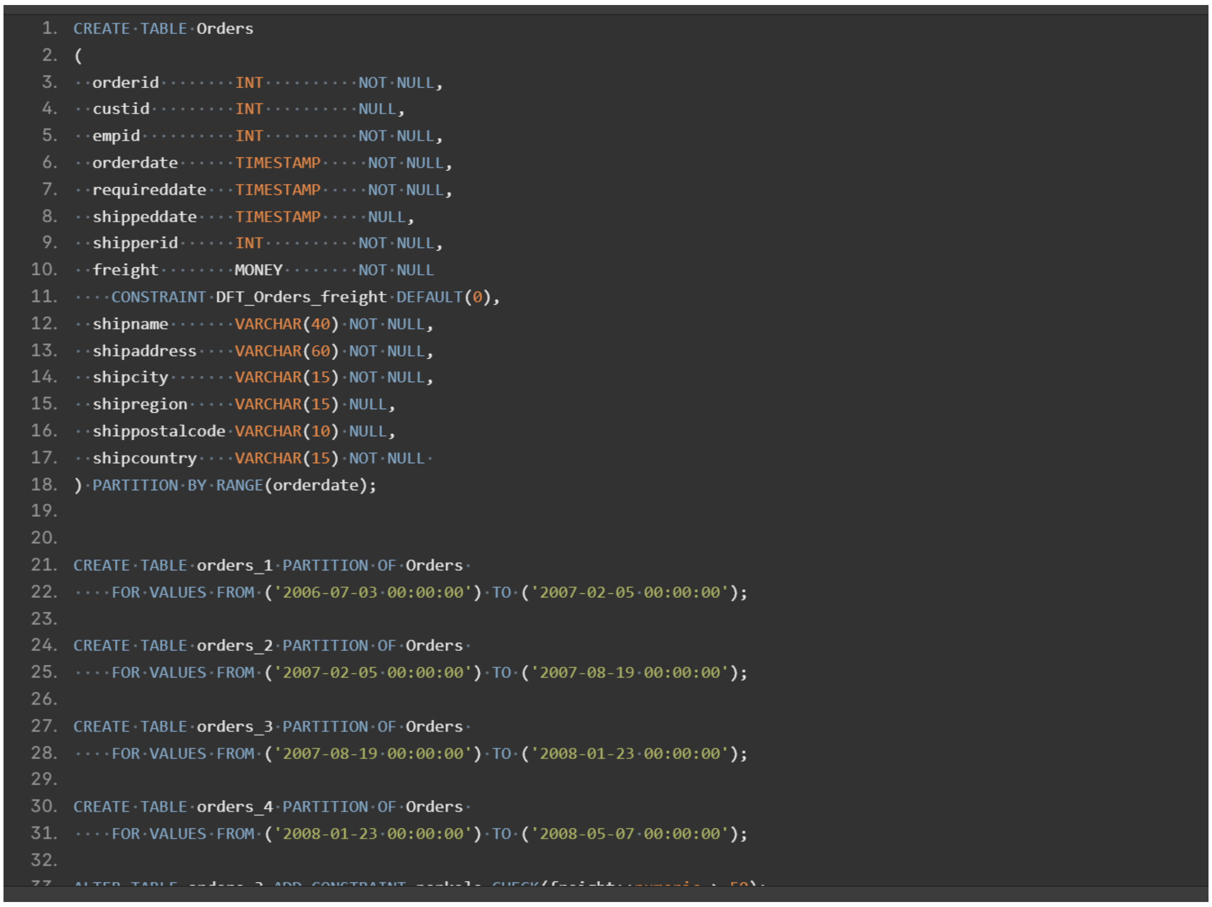
1. Orders between: 20060703 00:00:00.000 and 20070205 00:00:00.000
2. Orders between: 20070205 00:00:00.000 and 20070819 00:00:00.000
3. Orders between: 20070819 00:00:00.000 and 20080123 00:00:00.000
4. Orders between: 20080123 00:00:00.000 and 20080507 00:00:00.000

Name your partitions the following: orders_1, orders_2, orders_3, orders_4



```sql
CREATE TABLE Orders
(
  orderid        INT          NOT NULL,
  custid         INT          NULL,
  empid          INT          NOT NULL,
  orderdate      TIMESTAMP     NOT NULL,
  requireddate   TIMESTAMP     NOT NULL,
  shippeddate    TIMESTAMP     NULL,
  shipperid      INT          NOT NULL,
  freight        MONEY        NOT NULL
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
  shipname       VARCHAR(40) NOT NULL,
  shipaddress    VARCHAR(60) NOT NULL,
  shipcity       VARCHAR(15) NOT NULL,
  shipregion     VARCHAR(15) NULL,
  shippostalcode VARCHAR(10) NULL,
  shipcountry    VARCHAR(15) NOT NULL
) PARTITION BY RANGE(orderdate);


CREATE TABLE orders_1 PARTITION OF Orders
    FOR VALUES FROM ('2006-07-03 00:00:00') TO ('2007-02-05 00:00:00');

CREATE TABLE orders_2 PARTITION OF Orders
    FOR VALUES FROM ('2007-02-05 00:00:00') TO ('2007-08-19 00:00:00');

CREATE TABLE orders_3 PARTITION OF Orders
    FOR VALUES FROM ('2007-08-19 00:00:00') TO ('2008-01-23 00:00:00');

CREATE TABLE orders_4 PARTITION OF Orders
    FOR VALUES FROM ('2008-01-23 00:00:00') TO ('2008-05-07 00:00:00');
```

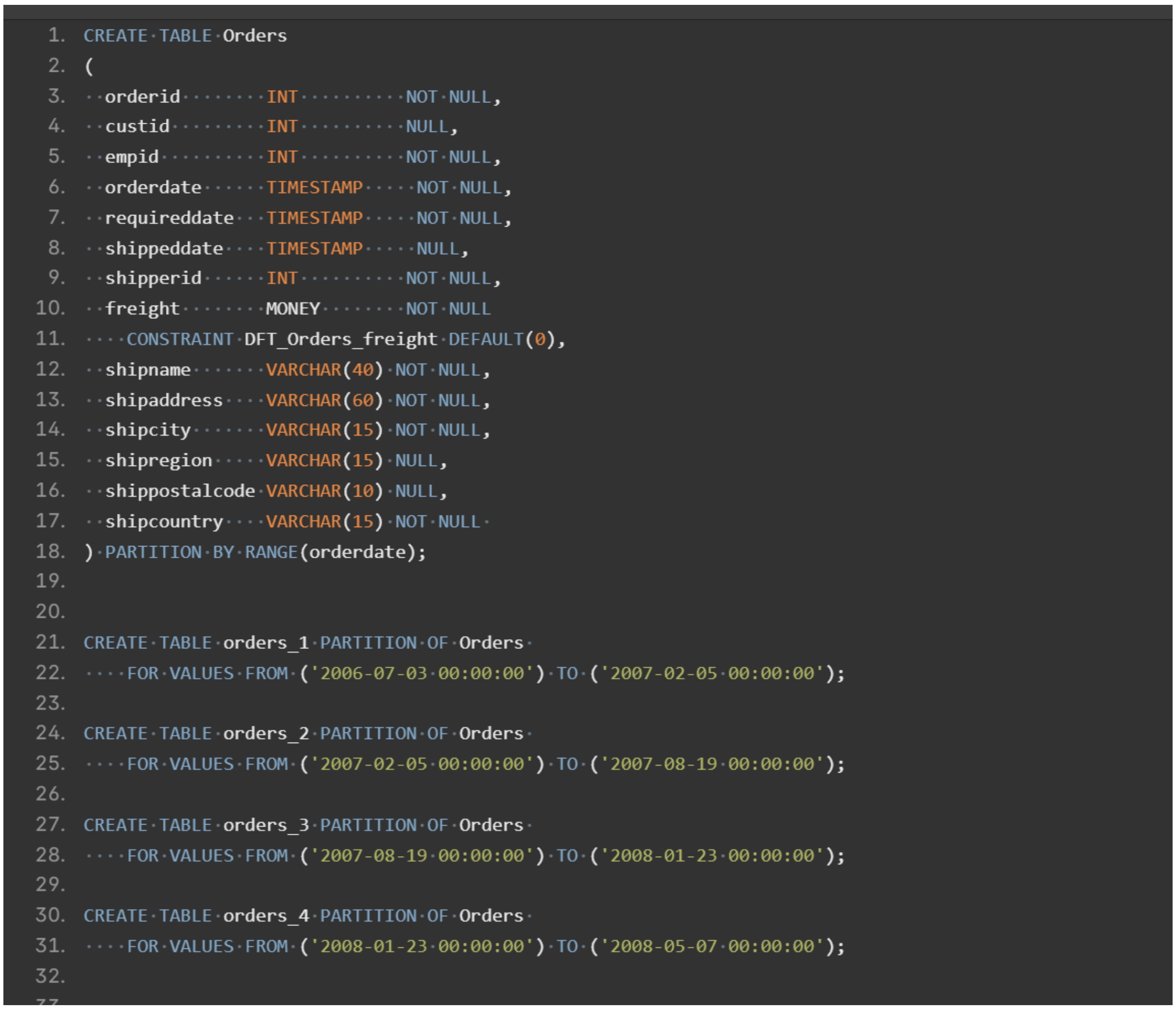
b) Alter the third partition (orders_3) and add a constraint where the freight cost is higher than 50 €



```
CREATE TABLE Orders
(
  orderid        INT          NOT NULL,
  custid         INT          NULL,
  empid          INT          NOT NULL,
  orderdate      TIMESTAMP     NOT NULL,
  requireddate   TIMESTAMP     NOT NULL,
  shippeddate    TIMESTAMP     NULL,
  shipperid      INT          NOT NULL,
  freight        MONEY        NOT NULL
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
  shipname       VARCHAR(40) NOT NULL,
  shipaddress    VARCHAR(60) NOT NULL,
  shipcity       VARCHAR(15) NOT NULL,
  shipregion     VARCHAR(15) NULL,
  shippostalcode VARCHAR(10) NULL,
  shipcountry    VARCHAR(15) NOT NULL
) PARTITION BY RANGE(orderdate);


CREATE TABLE orders_1 PARTITION OF Orders
    FOR VALUES FROM ('2006-07-03 00:00:00') TO ('2007-02-05 00:00:00');

CREATE TABLE orders_2 PARTITION OF Orders
    FOR VALUES FROM ('2007-02-05 00:00:00') TO ('2007-08-19 00:00:00');

CREATE TABLE orders_3 PARTITION OF Orders
    FOR VALUES FROM ('2007-08-19 00:00:00') TO ('2008-01-23 00:00:00');

CREATE TABLE orders_4 PARTITION OF Orders
    FOR VALUES FROM ('2008-01-23 00:00:00') TO ('2008-05-07 00:00:00');

ALTER TABLE orders_3 ADD CONSTRAINT [illegible] CHECK(freight::numeric > 50);
```

----

c) Alter the fourth partition (orders_4) and add a constraint that the shipped date should not be null


```
CREATE TABLE Orders
(
  orderid        INT          NOT NULL,
  custid         INT          NULL,
  empid          INT          NOT NULL,
  orderdate      TIMESTAMP    NOT NULL,
  requireddate   TIMESTAMP    NOT NULL,
  shippeddate    TIMESTAMP    NULL,
  shipperid      INT          NOT NULL,
  freight        MONEY        NOT NULL
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
  shipname       VARCHAR(40) NOT NULL,
  shipaddress    VARCHAR(60) NOT NULL,
  shipcity       VARCHAR(15) NOT NULL,
  shipregion     VARCHAR(15) NULL,
  shippostalcode VARCHAR(10) NULL,
  shipcountry    VARCHAR(15) NOT NULL
) PARTITION BY RANGE(orderdate);


CREATE TABLE orders_1 PARTITION OF Orders
    FOR VALUES FROM ('2006-07-03 00:00:00') TO ('2007-02-05 00:00:00');

CREATE TABLE orders_2 PARTITION OF Orders
    FOR VALUES FROM ('2007-02-05 00:00:00') TO ('2007-08-19 00:00:00');

CREATE TABLE orders_3 PARTITION OF Orders
    FOR VALUES FROM ('2007-08-19 00:00:00') TO ('2008-01-23 00:00:00');

CREATE TABLE orders_4 PARTITION OF Orders
    FOR VALUES FROM ('2008-01-23 00:00:00') TO ('2008-05-07 00:00:00');
```

----

d) Create two partitions of the first partition (so a partition of orders_1) using shipcountry so that:

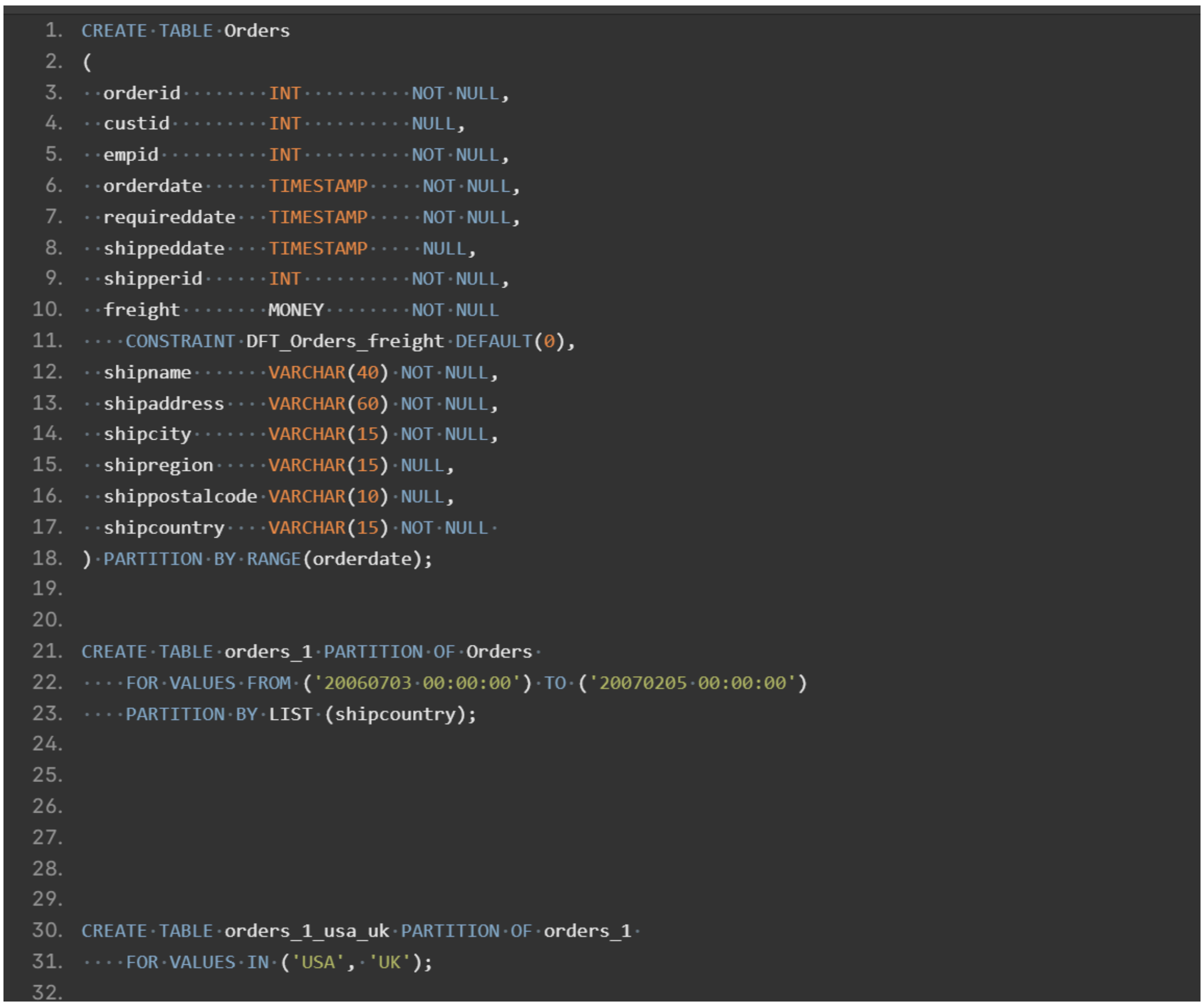
1. Orders shipped to USA and UK are in one

2. Orders shipped to Germany and Finland are in another

Name your partitions the following:

orders_1_usa_uk

orders_1_ger_fin



```sql
CREATE TABLE Orders
(
  orderid        INT          NOT NULL,
  custid         INT          NULL,
  empid          INT          NOT NULL,
  orderdate      TIMESTAMP    NOT NULL,
  requireddate   TIMESTAMP    NOT NULL,
  shippeddate    TIMESTAMP    NULL,
  shipperid      INT          NOT NULL,
  freight        MONEY        NOT NULL
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
  shipname       VARCHAR(40) NOT NULL,
  shipaddress    VARCHAR(60) NOT NULL,
  shipcity       VARCHAR(15) NOT NULL,
  shipregion     VARCHAR(15) NULL,
  shippostalcode VARCHAR(10) NULL,
  shipcountry    VARCHAR(15) NOT NULL
) PARTITION BY RANGE(orderdate);


CREATE TABLE orders_1 PARTITION OF Orders
    FOR VALUES FROM ('20060703 00:00:00') TO ('20070205 00:00:00')
    PARTITION BY LIST (shipcountry);






CREATE TABLE orders_1_usa_uk PARTITION OF orders_1
    FOR VALUES IN ('USA', 'UK');

```

Use the HW2.sql given in Moodle for creating (and inserting) data.

Last modified: Thursday, 7 March 2024, 8:05 PM

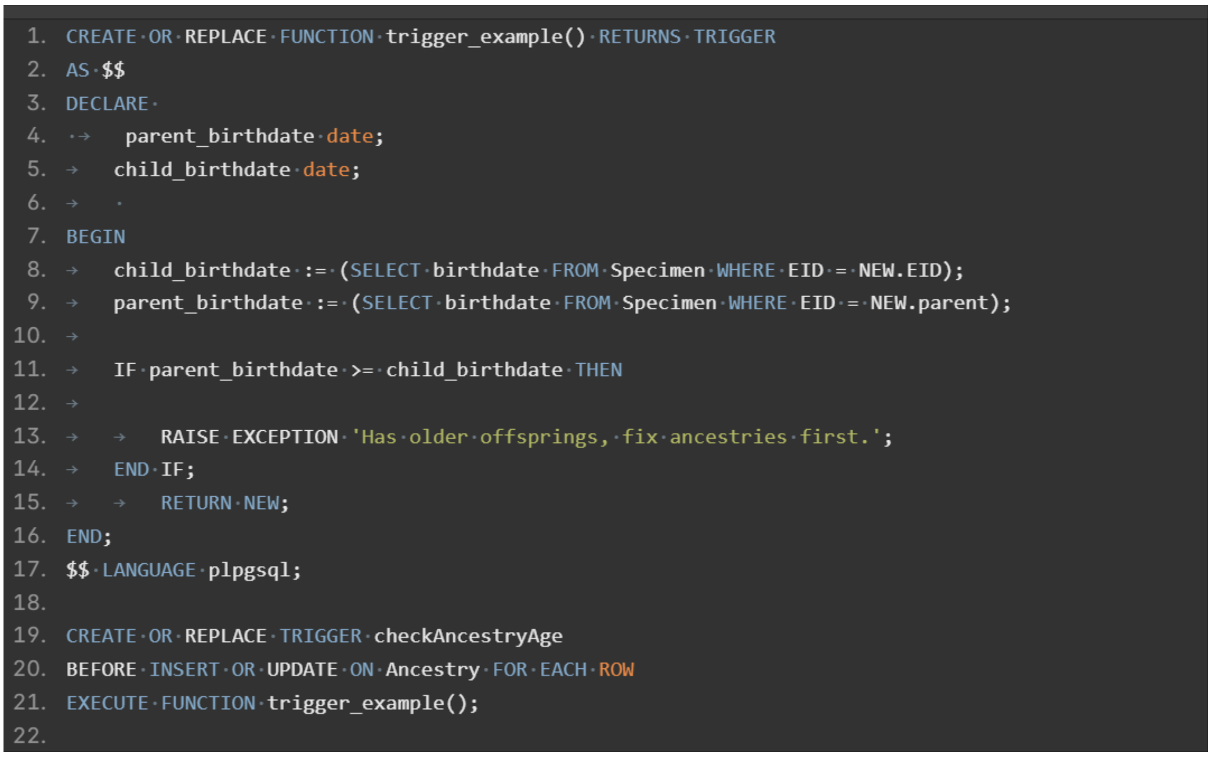
HW3 instructions: Triggers

Completion requirements

NOTE: To test your triggers, you need to try insert new data into the tables that go against the rules.
I recommend A and C be done first as they are the easier ones. B and D are more difficult.

Develop triggers that support the following integrity constraints using the HW3.sql:

a) Parents are not younger than their offsprings
When inserting or updating specimen or ancestry, have trigger(s) raise an exception when the rule is broken: '*Has older offsprings, fix ancestries first.*'.

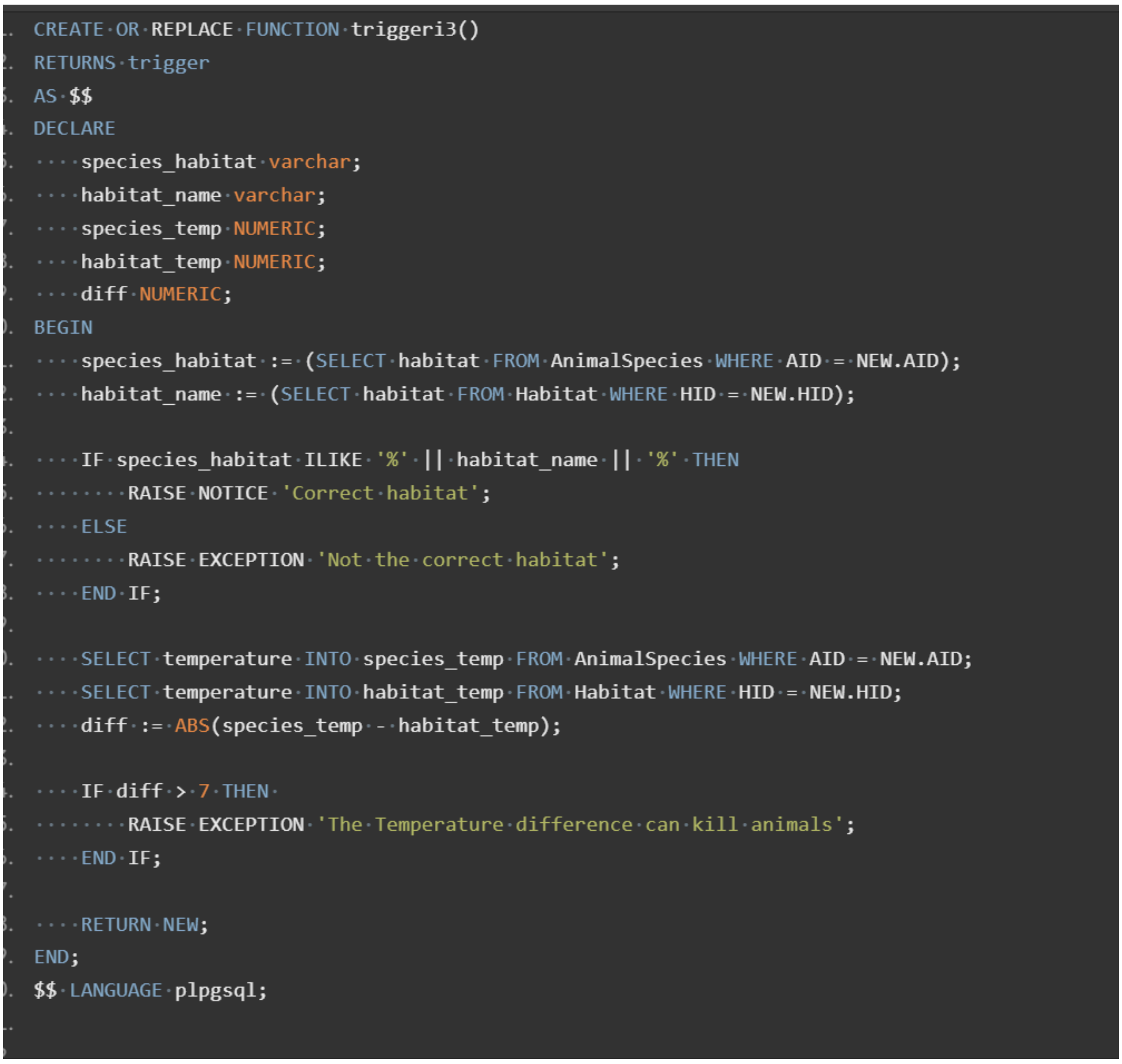


```
CREATE OR REPLACE FUNCTION trigger_example() RETURNS TRIGGER
AS $$
DECLARE
     parent_birthdate date;
    child_birthdate date;
    
BEGIN
    child_birthdate := (SELECT birthdate FROM Specimen WHERE EID = NEW.EID);
    parent_birthdate := (SELECT birthdate FROM Specimen WHERE EID = NEW.parent);

    IF parent_birthdate >= child_birthdate THEN

        RAISE EXCEPTION 'Has older offsprings, fix ancestries first.';
    END IF;
        RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER checkAncestryAge
BEFORE INSERT OR UPDATE ON Ancestry FOR EACH ROW
EXECUTE FUNCTION trigger_example();
```

b) AnimalSpecies Habitat information must coincide with the Habitat table.

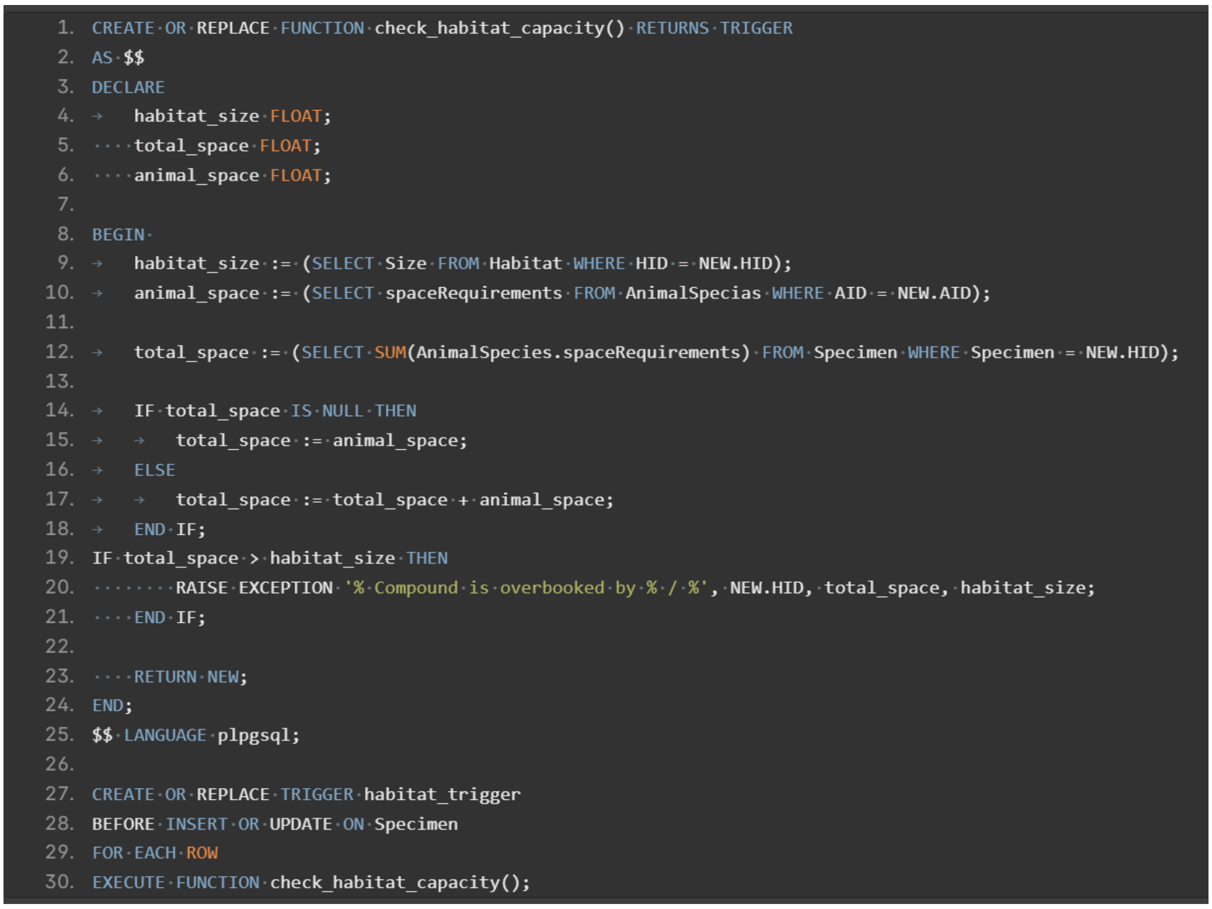
AnimalSpecies Habitat name (*Habitat column*) must coincide with the Habitat table (*Habitat column*) (partial match of words). Note that AnimalSpecies may have a longer list of possible habitats.
Temperature should not differ more than 7 degrees of what the species needs.
When inserting specimen or updating habitat information, have trigger(s) do the following:

1. If habitat name does not match, raise an exception: *Not the correct habitat*
2. If habitat name does match, raise a notice: *Correct habitat*
3. If temperature is over 7 degrees different, raise an execption: *The Temperature difference can kill animals*



```
CREATE OR REPLACE FUNCTION triggeri3()
RETURNS trigger
AS $$
DECLARE
    species_habitat varchar;
    habitat_name varchar;
    species_temp NUMERIC;
    habitat_temp NUMERIC;
    diff NUMERIC;
BEGIN
    species_habitat := (SELECT habitat FROM AnimalSpecies WHERE AID = NEW.AID);
    habitat_name := (SELECT habitat FROM Habitat WHERE HID = NEW.HID);

    IF species_habitat ILIKE '%' || habitat_name || '%' THEN
        RAISE NOTICE 'Correct habitat';
    ELSE
        RAISE EXCEPTION 'Not the correct habitat';
    END IF;

    SELECT temperature INTO species_temp FROM AnimalSpecies WHERE AID = NEW.AID;
    SELECT temperature INTO habitat_temp FROM Habitat WHERE HID = NEW.HID;
    diff := ABS(species_temp - habitat_temp);

    IF diff > 7 THEN
        RAISE EXCEPTION 'The Temperature difference can kill animals';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

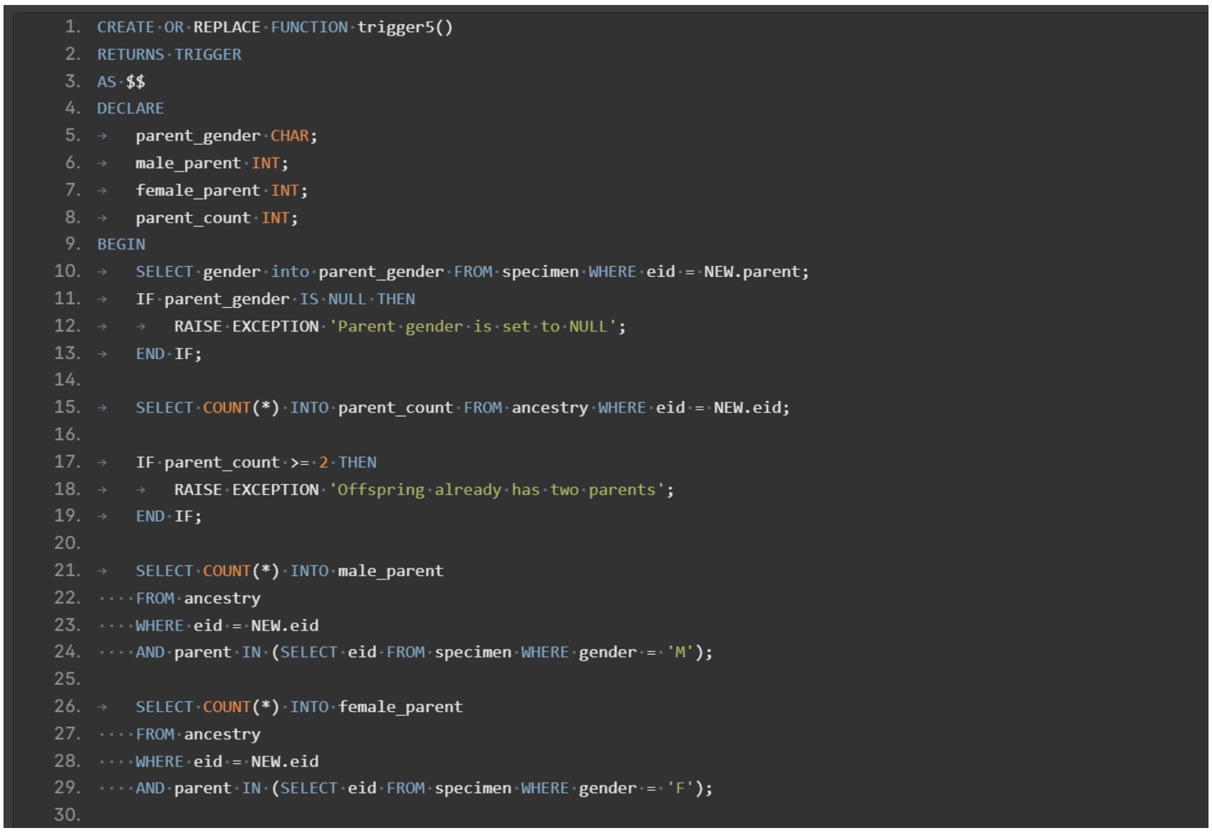
c) If a Habitat becomes overbooked then we need a warning. Compare Habitat size to the SpaceRequirement of AnimalSpecies.
When inserting or updating specimen, have trigger(s) raise a warning: *% Compound is overbooked by % / %* if exceeding the size of the habitat.
Note: % marks in the trigger code act as placeholders for variables (similarly to what Python used with ? and SQL).
So the three % marks correspond to three variables in the trigger code that should be: HID number, total size that would be in compound after the addition and finally the actual compound size.
For example: hid = 100, habitat size = 120, total space requirements = 150.
% Compound is overbooked by % / %   == 100 compound is overbooked by 150/120.
The variables are added after the string and separating them with a comma (as with Python and ? ? ? placeholders).



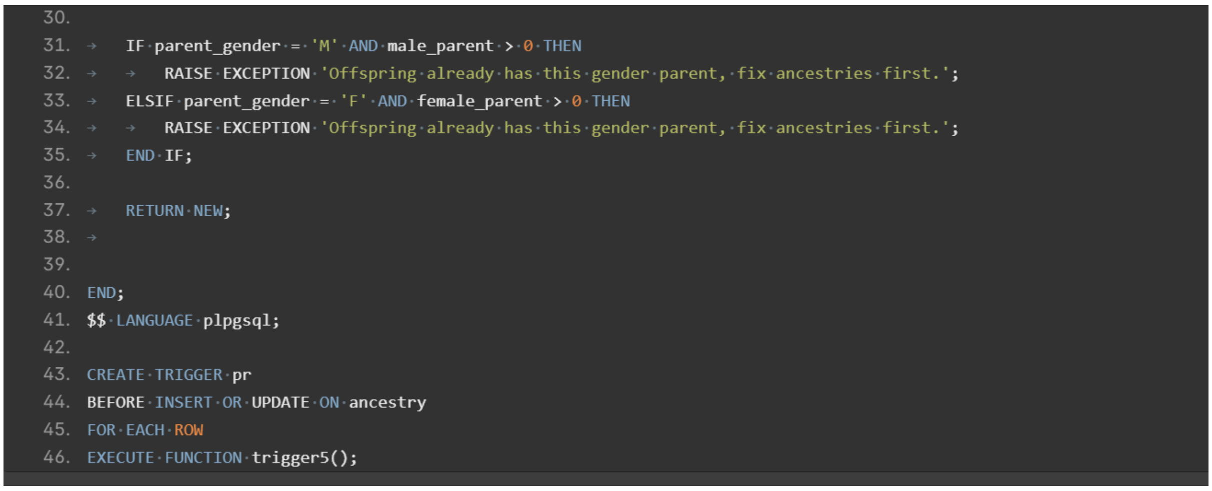
```sql
CREATE OR REPLACE FUNCTION check_habitat_capacity() RETURNS TRIGGER
AS $$
DECLARE
    habitat_size FLOAT;
    total_space FLOAT;
    animal_space FLOAT;

BEGIN 
    habitat_size := (SELECT Size FROM Habitat WHERE HID = NEW.HID);
    animal_space := (SELECT spaceRequirements FROM AnimalSpecias WHERE AID = NEW.AID);

    total_space := (SELECT SUM(AnimalSpecies.spaceRequirements) FROM Specimen WHERE Specimen = NEW.HID);

    IF total_space IS NULL THEN
        total_space := animal_space;
    ELSE
        total_space := total_space + animal_space;
    END IF;
IF total_space > habitat_size THEN
        RAISE EXCEPTION '% Compound is overbooked by % / %', NEW.HID, total_space, habitat_size;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER habitat_trigger
BEFORE INSERT OR UPDATE ON Specimen
FOR EACH ROW
EXECUTE FUNCTION check_habitat_capacity();
```

d) Offsprings have at MOST one male, one female parent. Consider NULLS.
When inserting or updating specimen or ancestry, have trigger(s) raise the following exceptions:

1. If parent gender is null: *Parent gender is set to NULL.*
2. If the offspring already has two parents: *Offspring already has two parents.*
3. If the offspring has the same gendered parent: *Offspring already has this gender parent, fix ancestries first.*



```sql
CREATE OR REPLACE FUNCTION trigger5()
RETURNS TRIGGER
AS $$
DECLARE
    parent_gender CHAR;
    male_parent INT;
    female_parent INT;
    parent_count INT;
BEGIN
    SELECT gender into parent_gender FROM specimen WHERE eid = NEW.parent;
    IF parent_gender IS NULL THEN
        RAISE EXCEPTION 'Parent gender is set to NULL';
    END IF;

    SELECT COUNT(*) INTO parent_count FROM ancestry WHERE eid = NEW.eid;

    IF parent_count >= 2 THEN
        RAISE EXCEPTION 'Offspring already has two parents';
    END IF;

    SELECT COUNT(*) INTO male_parent
    FROM ancestry
    WHERE eid = NEW.eid
    AND parent IN (SELECT eid FROM specimen WHERE gender = 'M');

    SELECT COUNT(*) INTO female_parent
    FROM ancestry
    WHERE eid = NEW.eid
    AND parent IN (SELECT eid FROM specimen WHERE gender = 'F');

```



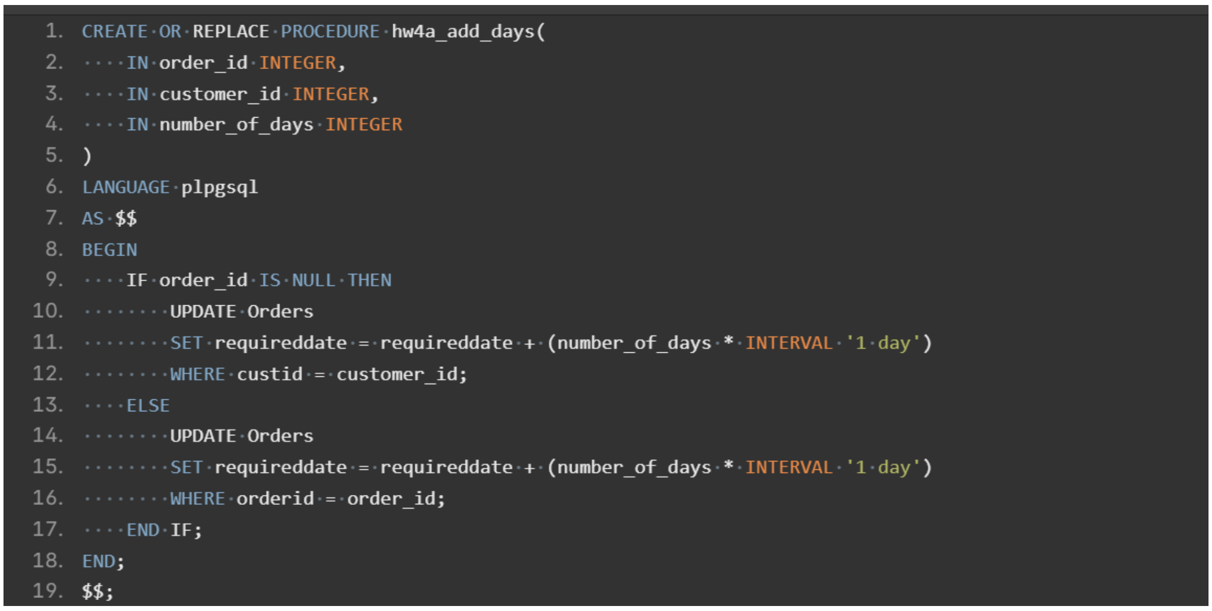
```sql

    IF parent_gender = 'M' AND male_parent > 0 THEN
        RAISE EXCEPTION 'Offspring already has this gender parent, fix ancestries first.';
    ELSIF parent_gender = 'F' AND female_parent > 0 THEN
        RAISE EXCEPTION 'Offspring already has this gender parent, fix ancestries first.';
    END IF;

    RETURN NEW;
    

END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER pr
BEFORE INSERT OR UPDATE ON ancestry
FOR EACH ROW
EXECUTE FUNCTION trigger5();
```

HW4 instructions Procedures

Completion requirements

NOTE: Procedures need to be called explicitly after they are created so to test your procedures, remember to call them.

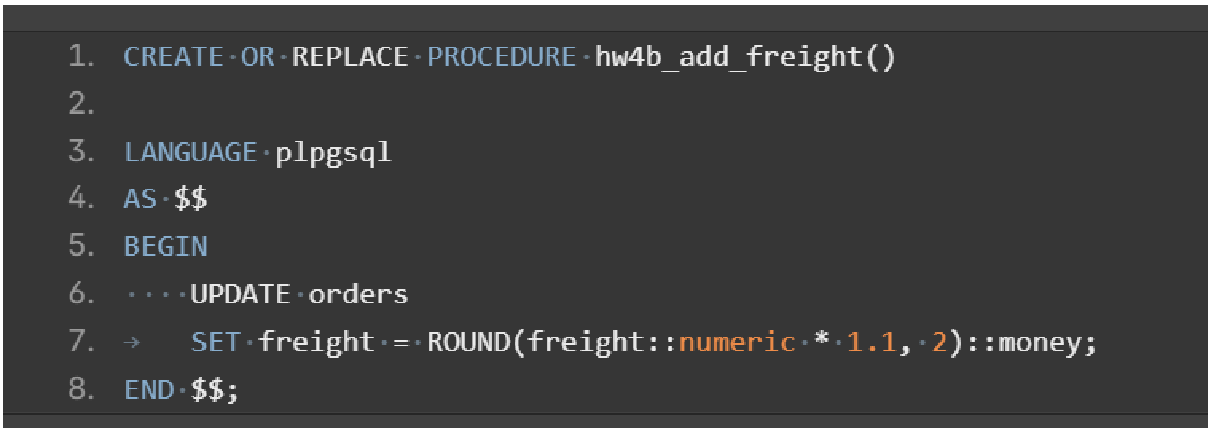
a) Create a procedure that adds X amount of days (given by the user) to the "requireddate" value based on custid or orderid. If orderid given is NULL, the procedure runs based on custid. The procedure takes three arguments (order id, customer id, number of days).
Name the procedure: hw4a_add_days()



```
CREATE OR REPLACE PROCEDURE hw4a_add_days(
    IN order_id INTEGER,
    IN customer_id INTEGER,
    IN number_of_days INTEGER
)
LANGUAGE plpgsql
AS $$
BEGIN
    IF order_id IS NULL THEN
        UPDATE Orders
        SET requireddate = requireddate + (number_of_days * INTERVAL '1 day')
        WHERE custid = customer_id;
    ELSE
        UPDATE Orders
        SET requireddate = requireddate + (number_of_days * INTERVAL '1 day')
        WHERE orderid = order_id;
    END IF;
END;
$$;
```

b) Create a procedure that adds 10 % to the freight money.

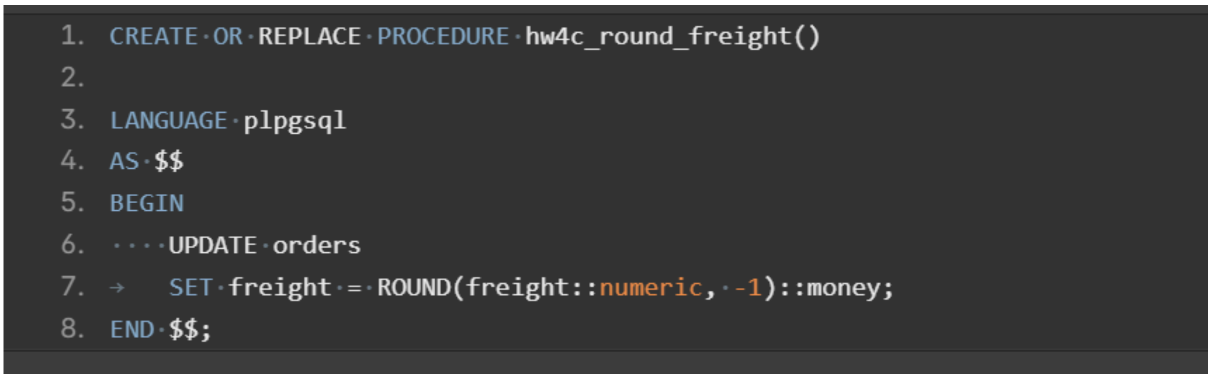
Name the procedure: hw4b_add_freight()



```sql
CREATE OR REPLACE PROCEDURE hw4b_add_freight()

LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE orders
	SET freight = ROUND(freight::numeric * 1.1, 2)::money;
END $$;
```

c) Create a procdeure that rounds the freight costs to nearest 10.

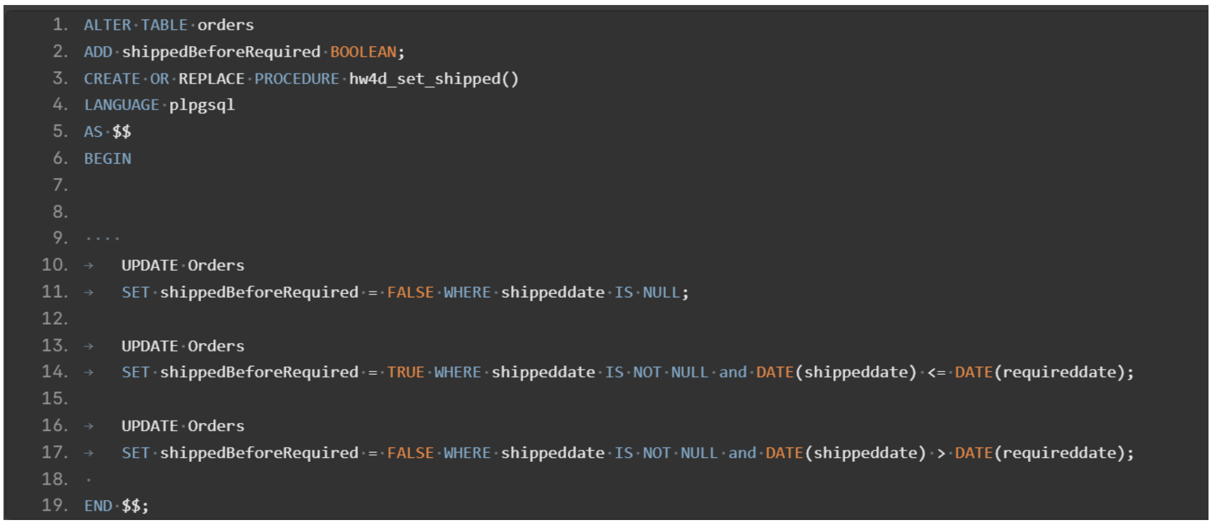
Name the procedure: hw4c_round_freight()



```sql
CREATE OR REPLACE PROCEDURE hw4c_round_freight()

LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE orders
	SET freight = ROUND(freight::numeric, -1)::money;
END $$;
```

d) Add a new column 'shippedBeforeRequired' to Orders table (using ALTER command) of boolean type. Create a procedure that sets 'shippedBeforeRequired' to true if shippeddate is smaller than requireddate and false if vice-versa (remember NULLs)

Name the procedure: hw4d_set_shipped()



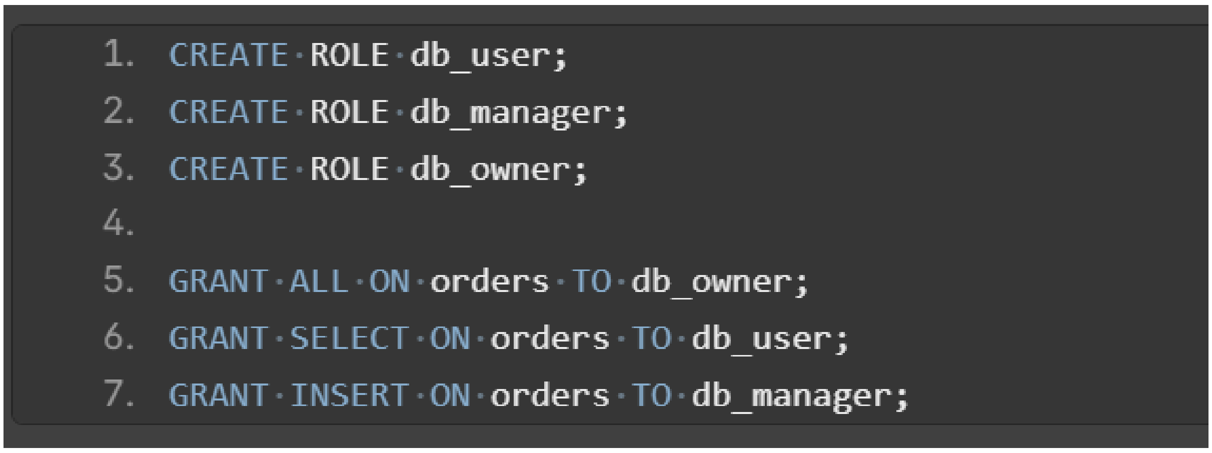
```
ALTER TABLE orders
ADD shippedBeforeRequired BOOLEAN;
CREATE OR REPLACE PROCEDURE hw4d_set_shipped()
LANGUAGE plpgsql
AS $$
BEGIN


    
	UPDATE Orders
	SET shippedBeforeRequired = FALSE WHERE shippeddate IS NULL;

	UPDATE Orders
	SET shippedBeforeRequired = TRUE WHERE shippeddate IS NOT NULL and DATE(shippeddate) <= DATE(requireddate);

	UPDATE Orders
	SET shippedBeforeRequired = FALSE WHERE shippeddate IS NOT NULL and DATE(shippeddate) > DATE(requireddate);
 
END $$;
```

Each is worth 2.5 %

HW5 instructions: Roles and functions

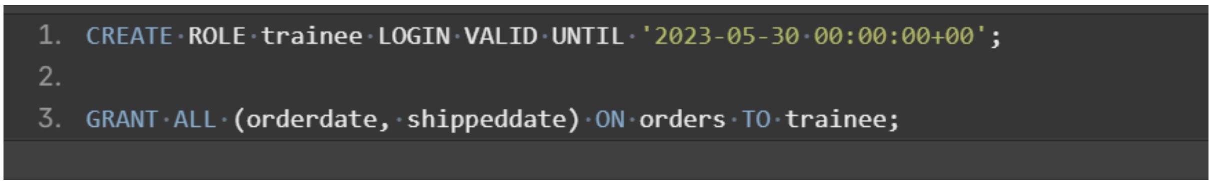
Completion requirements

a) Create the following roles: db_user, db_manager, db_owner. Grant all privileges to owner, read privileges to user, and insert privileges to manager.



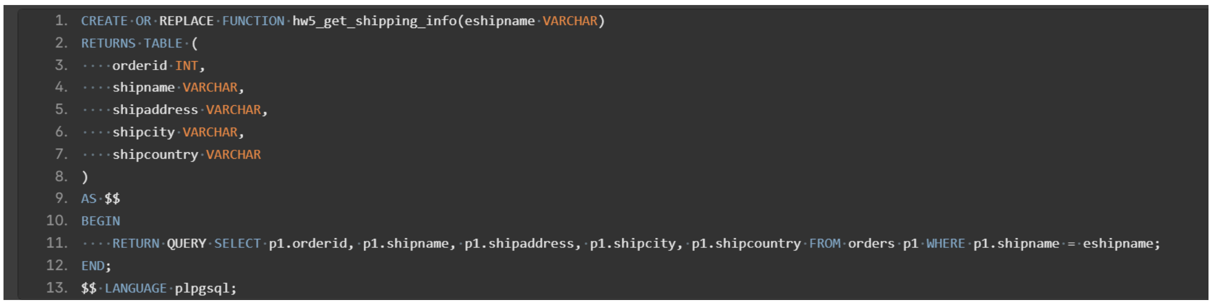
```sql
CREATE ROLE db_user;
CREATE ROLE db_manager;
CREATE ROLE db_owner;

GRANT ALL ON orders TO db_owner;
GRANT SELECT ON orders TO db_user;
GRANT INSERT ON orders TO db_manager;
```

b) Create a new role: trainee. Grant privileges only to columns *orderdate* and *shippeddate* to trainee and set the role valid until 30.5.2023.



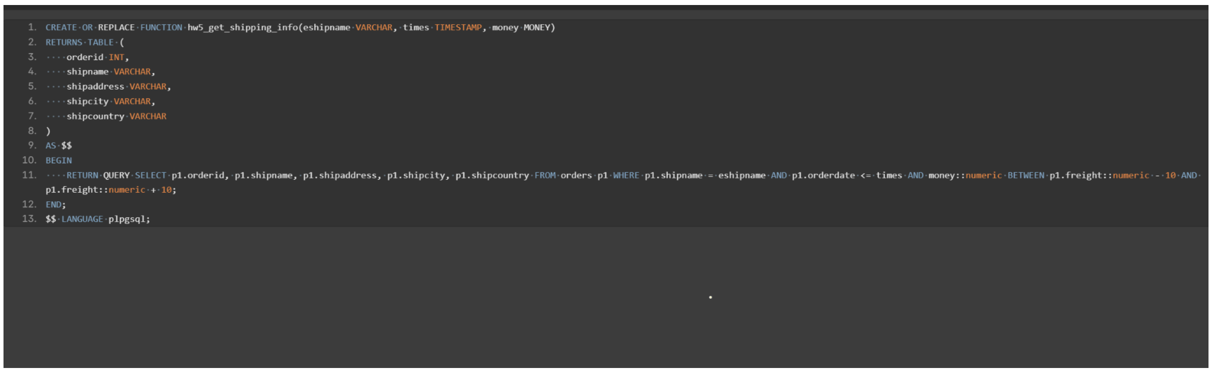
```sql
CREATE ROLE trainee LOGIN VALID UNTIL '2023-05-30 00:00:00+00';

GRANT ALL (orderdate, shippeddate) ON orders TO trainee;
```

c) Create a function hw5_get_shipping_info(varchar) that returns a table. The table should have the following columns *orderid*, *shipname*, *shipaddress*, *shipcity*, shipcountry. The function should return orders where the shipname matches the given string.



```sql
CREATE OR REPLACE FUNCTION hw5_get_shipping_info(eshipname VARCHAR)
RETURNS TABLE (
    orderid INT,
    shipname VARCHAR,
    shipaddress VARCHAR,
    shipcity VARCHAR,
    shipcountry VARCHAR
)
AS $$
BEGIN
    RETURN QUERY SELECT p1.orderid, p1.shipname, p1.shipaddress, p1.shipcity, p1.shipcountry FROM orders p1 WHERE p1.shipname = eshipname;
END;
$$ LANGUAGE plpgsql;
```

d) Extend the function in c) so that it accepts three parameters: hw5_get_shipping_info(varchar, timestamp, money). The function should return orders where the shipname matches the given string, orderdate is equal or earlier than the given timestamp and finally, freight cost is +-10 from the given money.



```sql
CREATE OR REPLACE FUNCTION hw5_get_shipping_info(eshipname VARCHAR, times TIMESTAMP, money MONEY)
RETURNS TABLE (
    orderid INT,
    shipname VARCHAR,
    shipaddress VARCHAR,
    shipcity VARCHAR,
    shipcountry VARCHAR
)
AS $$
BEGIN
    RETURN QUERY SELECT p1.orderid, p1.shipname, p1.shipaddress, p1.shipcity, p1.shipcountry FROM orders p1 WHERE p1.shipname = eshipname AND p1.orderdate <= times AND money::numeric BETWEEN p1.freight::numeric - 10 AND
p1.freight::numeric + 10;
END;
$$ LANGUAGE plpgsql;
```