Group 28

Owen, Russ, Gabrielle, Marcello

Pre. Renewed Relational Schema

Relations

Entites:

Potato(p_id, colour, quality, rotten)

Leprechaun(l_id, in_training, first_name, last_name, birthday, fatigue, s_id) - s_id is a foreign key referencing the SensitivityTraining entity

RainbowRegion(region_name)

Stand(coordinates, golden_stock, regular_stock, l_id) - l_id is a foreign key referencing the FarmstandClerk entity

SensitivityTraining(s_id, date, time)

Customer(c_id, credit_card_number, credit_card_expiry, first_name, last_name)

DeliveryOrder(d_id, regular_quantity, golden_quantity, c_id, l_id) - c_id is a foreign key referencing the Customer entity - l_id is a foreign key referencing the PotatoCourier entity

Weak Entities:

Rainbow(r_id, region_name, regular_quantity, golden_quantity, rotten_quantity) - region_name is a foreign key referencing the RainbowRegion entity

IS-A Entities:

RainbowHunter(l_id, gold_level, hit_rate) - l_id is a foreign key referencing the Leprechaun entity

PotatoDigger(l_id, latest_yield, lifetime_haul) - l_id is a foreign key referencing the Leprechaun entity

FarmstandClerk(l_id, potatoes_sold) - l_id is a foreign key referencing the Leprechaun entity

PotatoCourier(l_id) - l_id is a foreign key referencing the Leprechaun entity

Relationships:

RainbowHunting(l_id, r_id, region_name) - l_id is a foreign key referencing the RainbowHunter entity - r_id is a foreign key referencing the Rainbow entity - region_name is a foreign key referencing the RainbowRegion entity

PotatoDigging(p_id, l_id) - p_id is a foreign key referencing the Potato entity - l_id is a foreign key referencing the PotatoDigger entity

TravellingOnRainbow(l_id, r_id, region_name) - l_id is a foreign key referencing the Leprechaun entity - r_id is a foreign key referencing the Rainbow entity - region_name is a foreign key referencing the RainbowRegion entity

Purchased(p_id, c_id, cost) - p_id is a foreign key referencing the Potato entity - c_id is a foreign key referencing the Customer entity

Stocked(p_id, coordinates) - p_id is a foreign key referencing the Potato entity - coordinates is a foreign key referencing the Stand entity

AdjacentTo(region_name, region_name) - region_name is a foreign key referencing the RainbowRegion entity

Near(r_id, region_name, coordinates) - r_id is a foreign key referencing the Rainbow entity - region_name is a foreign key referencing the RainbowRegion entity - coordinates is a foreign key referencing the Stand entity

CanWorkIn(l_id, region_name) - l_id is a foreign key referencing the Leprechaun entity - region_name is a foreign key referencing the RainbowRegion entity

There are no relations that can be combined without introducing redundancy.

I. Pending Constraints

Within “AdjacentTo” relationship table:

Two tuples carrying the same meaning can exist within “AdjecentTo” table (e.g., (North America, Latin America) and (Latin America, North America) are different tuples, but has the same meaning). This may lead to some duplicates in the results of some queries.

Within tables describing working Leprechauns:

Leprechauns who are currently in training or are too tired to work can be erroneously added to tables such as “RainbowHunting”, “PotatoDigging”, “Stand”, and “TravellingOnRainbow”. This requires checking for Leprechauns’ tiredness attribute and “in_training” flag.

Within tables describing Leprechauns’ professions:

Leprechauns can be added to two or more profession tables (e.g., “RainbowHunter”, “PotatoDigger”). This requires checking for duplicate Leprechaun IDs across multiple tables — possibly upon assigning a job to a Leprechaun — which is impossible to do in the database code. Also, if ever a Leprechaun were to be assigned to two or more professions, that Leprechaun can undergo those jobs at the very same time, and consequently/possibly work in different locations.

Within “Stocked” relationship table:

There are two ways to describe, in the database server, a single potato that is not stocked anywhere: (1) either that potato is not described anywhere in the “Stocked” relationship table, (2) or that potato has a record in the “Stocked” table, but does not refer to any “Stand” entity. Enforcing a “Stocked” tuple to be created alongside the creation of a “Potato” record brings more consistency.

Within tables describing Leprechauns’ use of Rainbows:

There are no database-level constraints on who can use a certain, busy Rainbow or how many workers can use it. Certain groups of working Leprechauns should prevent some other Leprechauns with different jobs to use the same Rainbow (e.g., Potato diggers and couriers cannot use a rainbow simultaneously). Infinitely many Leprechaun workers can also be added to a single Rainbow at the database level.

II. SQL Queries

Query1

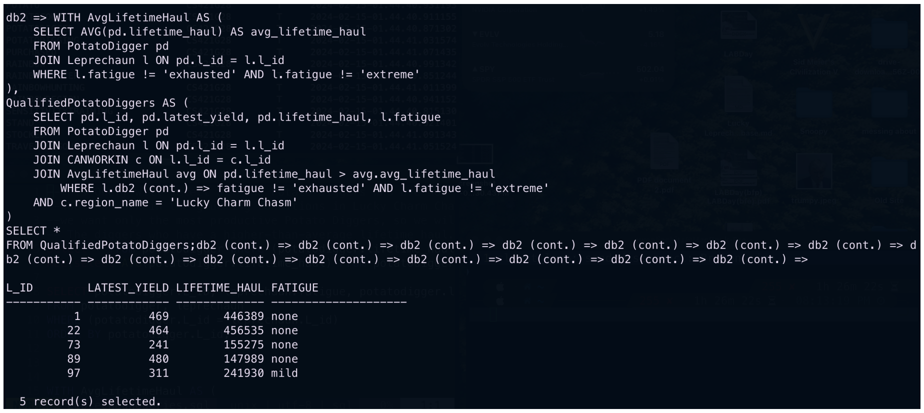
Description of what information the query returns:

This Query will select all the Potato Diggers who are not anywhere beyond mild fatigue, so they can work in the grueling conditions in Lucky Charm Chasm. They must of course, be allowed to work in Lucky Charm Chasm as well. In addition we want only the most productive Potato Diggers, so we will select only The diggers who have a higher-than-average lifetime haul.

The SQL Query in plaintext:

WITH AvgLifetimeHaul AS (SELECT AVG(pd.lifetime_haul) AS avg_lifetime_haul FROM PotatoDigger pd JOIN Leprechaun l ON pd.l_id = l.l_id WHERE l.fatigue != ‘exhausted’ AND l.fatigue != ‘extreme’), QualifiedPotatoDiggers AS (SELECT pd.l_id, pd.latest_yield, pd.lifetime_haul, l.fatigue FROM PotatoDigger pd JOIN Leprechaun l ON pd.l_id = l.l_id JOIN CANWORKIN c ON l.l_id = c.l_id JOIN AvgLifetimeHaul avg ON pd.lifetime_haul > avg.avg_lifetime_haul WHERE l.fatigue != ‘exhausted’ AND l.fatigue != ‘extreme’ AND c.region_name = ‘Lucky Charm Chasm’) SELECT * FROM QualifiedPotatoDiggers;

Screenshot of the query being executed (always include all of the SQL statement):



```
db2 => WITH AvgLifetimeHaul AS (
    SELECT AVG(pd.lifetime_haul) AS avg_lifetime_haul
    FROM PotatoDigger pd
    JOIN Leprechaun l ON pd.l_id = l.l_id
    WHERE l.fatigue != 'exhausted' AND l.fatigue != 'extreme'
),
QualifiedPotatoDiggers AS (
    SELECT pd.l_id, pd.latest_yield, pd.lifetime_haul, l.fatigue
    FROM PotatoDigger pd
    JOIN Leprechaun l ON pd.l_id = l.l_id
    JOIN CANWORKIN c ON l.l_id = c.l_id
    JOIN AvgLifetimeHaul avg ON pd.lifetime_haul > avg.avg_lifetime_haul
        WHERE l.db2 (cont.) => fatigue != 'exhausted' AND l.fatigue != 'extreme'
    AND c.region_name = 'Lucky Charm Chasm'
)
SELECT *
FROM QualifiedPotatoDiggers;db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => d
b2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>

L_ID        LATEST_YIELD LIFETIME_HAUL FATIGUE
----------- ------------ ------------- --------------------
          1          469        446389 none
         22          464        456535 none
         73          241        155275 none
         89          480        147989 none
         97          311        241930 mild

  5 record(s) selected.
```

Query2

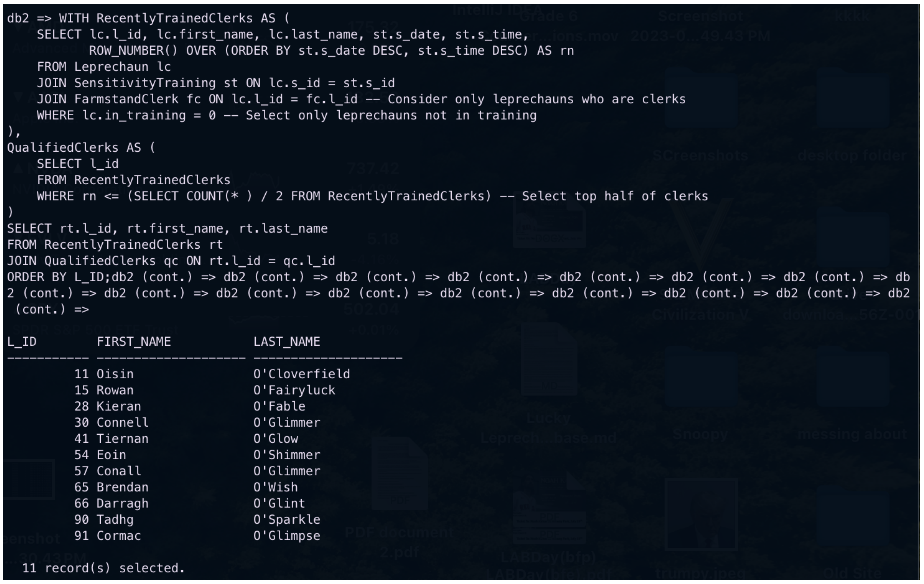
Description of what information the query returns:

WE want to make sure our Farmstands are tended by only the most up-to-date clerks who will provide the best customer service. So, we want the top half of current farmstand clerks ordered by the clerks who have most recently gone through sensitivity training. We don't want any Clerks who are still in training either.

The SQL Query in plaintext:

WITH RecentlyTrainedClerks AS (SELECT lc.l_id, lc.first_name, lc.last_name, st.s_date, st.s_time, ROW_NUMBER() OVER (ORDER BY st.s_date DESC, st.s_time DESC) AS rn FROM Leprechaun lc JOIN SensitivityTraining st ON lc.s_id = st.s_id JOIN FarmstandClerk fc ON lc.l_id = fc.l_id WHERE lc.in_training = 0), QualifiedClerks AS (SELECT l_id FROM RecentlyTrainedClerks WHERE rn <= (SELECT COUNT(*) / 2 FROM RecentlyTrainedClerks)) SELECT rt.l_id, rt.first_name, rt.last_name FROM RecentlyTrainedClerks rt JOIN QualifiedClerks qc ON rt.l_id = qc.l_id ORDER BY L_ID;

Screenshot of the query being executed (always include all of the SQL statement):



```
db2 => WITH RecentlyTrainedClerks AS (
    SELECT lc.l_id, lc.first_name, lc.last_name, st.s_date, st.s_time,
           ROW_NUMBER() OVER (ORDER BY st.s_date DESC, st.s_time DESC) AS rn
    FROM Leprechaun lc
    JOIN SensitivityTraining st ON lc.s_id = st.s_id
    JOIN FarmstandClerk fc ON lc.l_id = fc.l_id -- Consider only leprechauns who are clerks
    WHERE lc.in_training = 0 -- Select only leprechauns not in training
),
QualifiedClerks AS (
    SELECT l_id
    FROM RecentlyTrainedClerks
    WHERE rn <= (SELECT COUNT(* ) / 2 FROM RecentlyTrainedClerks) -- Select top half of clerks
)
SELECT rt.l_id, rt.first_name, rt.last_name
FROM RecentlyTrainedClerks rt
JOIN QualifiedClerks qc ON rt.l_id = qc.l_id
ORDER BY L_ID;db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db
2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2
 (cont.) =>

L_ID        FIRST_NAME           LAST_NAME
----------- -------------------- --------------------
         11 Oisin                O'Cloverfield
         15 Rowan                O'Fairyluck
         28 Kieran               O'Fable
         30 Connell              O'Glimmer
         41 Tiernan              O'Glow
         54 Eoin                 O'Shimmer
         57 Conall               O'Glimmer
         65 Brendan              O'Wish
         66 Darragh              O'Glint
         90 Tadhg                O'Sparkle
         91 Cormac               O'Glimpse

  11 record(s) selected.
```

Query3

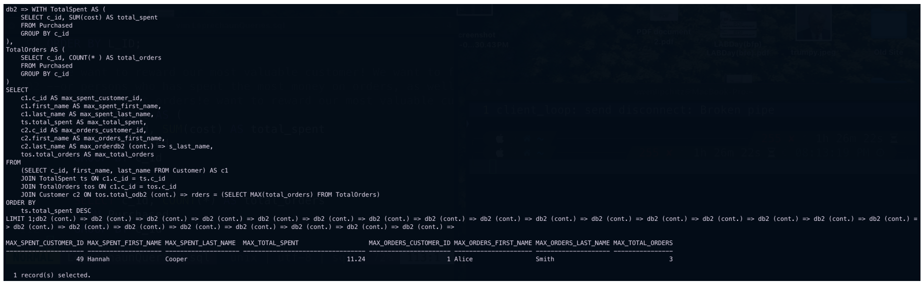
Description of what information the query returns:

We want to reward our most valuable customer(s)! We want to find the person or persons who has spent the most money on orders, as well as the person with the most orders.

The SQL Query in plaintext:

WITH TotalSpent AS (SELECT c_id, SUM(cost) AS total_spent FROM Purchased GROUP BY c_id), TotalOrders AS (SELECT c_id, COUNT(*) AS total_orders FROM Purchased GROUP BY c_id) SELECT c1.c_id AS max_spent_customer_id, c1.first_name AS max_spent_first_name, c1.last_name AS max_spent_last_name, ts.total_spent AS max_total_spent, c2.c_id AS max_orders_customer_id, c2.first_name AS max_orders_first_name, c2.last_name AS max_orders_last_name, tos.total_orders AS max_total_orders FROM (SELECT c_id, first_name, last_name FROM Customer) AS c1 JOIN TotalSpent ts ON c1.c_id = ts.c_id JOIN TotalOrders tos ON c1.c_id = tos.c_id JOIN Customer c2 ON tos.total_orders = (SELECT MAX(total_orders) FROM TotalOrders) ORDER BY ts.total_spent DESC LIMIT 1;

Screenshot of the query being executed (always include all of the SQL statement):



```
db2 => WITH TotalSpent AS (
    SELECT c_id, SUM(cost) AS total_spent
    FROM Purchased
    GROUP BY c_id
),
TotalOrders AS (
    SELECT c_id, COUNT(* ) AS total_orders
    FROM Purchased
    GROUP BY c_id
)
SELECT
    c1.c_id AS max_spent_customer_id,
    c1.first_name AS max_spent_first_name,
    c1.last_name AS max_spent_last_name,
    ts.total_spent AS max_total_spent,
    c2.c_id AS max_orders_customer_id,
    c2.first_name AS max_orders_first_name,
    c2.last_name AS max_orderdb2 (cont.) => s_last_name,
    tos.total_orders AS max_total_orders
FROM
    (SELECT c_id, first_name, last_name FROM Customer) AS c1
    JOIN TotalSpent ts ON c1.c_id = ts.c_id
    JOIN TotalOrders tos ON c1.c_id = tos.c_id
    JOIN Customer c2 ON tos.total_odb2 (cont.) => rders = (SELECT MAX(total_orders) FROM TotalOrders)
ORDER BY
    ts.total_spent DESC
LIMIT 1;db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =
> db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>

MAX_SPENT_CUSTOMER_ID MAX_SPENT_FIRST_NAME MAX_SPENT_LAST_NAME  MAX_TOTAL_SPENT                   MAX_ORDERS_CUSTOMER_ID MAX_ORDERS_FIRST_NAME MAX_ORDERS_LAST_NAME MAX_TOTAL_ORDERS
--------------------- -------------------- --------------------  --------------------------------- ---------------------- --------------------- -------------------- ----------------
                   49 Hannah               Cooper                                           11.24                      1 Alice                 Smith                               3

  1 record(s) selected.
```

Query4

Description of what information the query returns:

We want to make sure our Farmstands are adequitely stocked on golden potatoes. Thus, we will need our best gold-finding rainbow hunters to go out and hunt us some new gold. They will be the best available rainbow hunter for the corresponding region. We don't want our poor workers to work too hard, or else the unions will come after us. So, we need to make sure the rainbow hunters who are the most effective only work in the Region they are qualified in. ### The SQL Query in plaintext: 'WITH TopRainbowHunters AS (SELECT rh.l_id, rh.hit_rate, rhg.region_name, ROW_NUMBER() OVER(PARTITION BY rhg.region_name ORDER BY rh.hit_rate DESC) AS rn FROM RainbowHunter rh, rainbowhunting rhg WHERE rh.l_id = rhg.l_id),

AverageGoldenPotatoes AS (SELECT AVG(s.golden_stock) AS avg_golden_stock, n.region_name FROM Stand s JOIN Near n ON s.x_coord = n.x_coord AND s.y_coord = n.y_coord GROUP BY n.region_name),

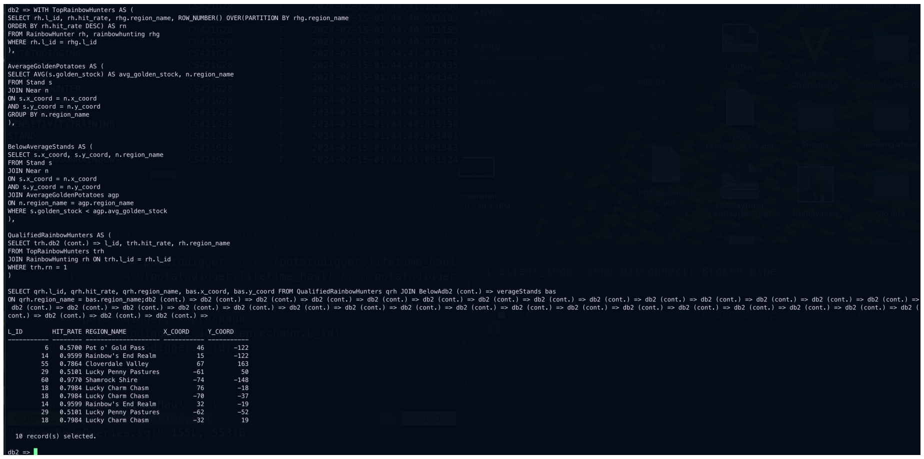
BelowAverageStands AS (SELECT s.x_coord, s.y_coord, n.region_name

FROM Stand s JOIN Near n ON s.x_coord = n.x_coord AND s.y_coord = n.y_coord JOIN AverageGoldenPotatoes agp ON n.region_name = agp.region_name WHERE s.golden_stock < agp.avg_golden_stock),

QualifiedRainbowHunters AS (SELECT trh.l_id, trh.hit_rate, rh.region_name FROM TopRainbowHunters trh JOIN RainbowHunting rh ON trh.l_id = rh.l_id WHERE trh.rn = 1)

SELECT qrh.l_id, qrh.hit_rate, qrh.region_name, bas.x_coord, bas.y_coord FROM QualifiedRainbowHunters qrh JOIN BelowAverageStands bas ON qrh.region_name = bas.region_name; ‘

Screenshot of the query being executed (always include all of the SQL statement):



```
db2 => WITH TopRainbowHunters AS (
SELECT rh.l_id, rh.hit_rate, rhg.region_name, ROW_NUMBER() OVER(PARTITION BY rhg.region_name
ORDER BY rh.hit_rate DESC) AS rn
FROM RainbowHunter rh, rainbowhunting rhg
WHERE rh.l_id = rhg.l_id
),

AverageGoldenPotatoes AS (
SELECT AVG(s.golden_stock) AS avg_golden_stock, n.region_name
FROM Stand s
JOIN Near n
ON s.x_coord = n.x_coord
AND s.y_coord = n.y_coord
GROUP BY n.region_name
),


BelowAverageStands AS (
SELECT s.x_coord, s.y_coord, n.region_name
FROM Stand s
JOIN Near n
ON s.x_coord = n.x_coord
AND s.y_coord = n.y_coord
JOIN AverageGoldenPotatoes agp
ON n.region_name = agp.region_name
WHERE s.golden_stock < agp.avg_golden_stock
),

QualifiedRainbowHunters AS (
SELECT trh.db2 (cont.) => l_id, trh.hit_rate, rh.region_name
FROM TopRainbowHunters trh
JOIN RainbowHunting rh ON trh.l_id = rh.l_id
WHERE trh.rn = 1
)

SELECT qrh.l_id, qrh.hit_rate, qrh.region_name, bas.x_coord, bas.y_coord FROM QualifiedRainbowHunters qrh JOIN BelowAdb2 (cont.) => verageStands bas
ON qrh.region_name = bas.region_name;db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>
 db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (
cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>

L_ID        HIT_RATE REGION_NAME          X_COORD     Y_COORD
----------- -------- -------------------- ----------- -----------
          6   0.5700 Pot o' Gold Pass              46        -122
         14   0.9599 Rainbow's End Realm           15        -122
         55   0.7864 Cloverdale Valley             67         163
         29   0.5101 Lucky Penny Pastures         -61          50
         60   0.9770 Shamrock Shire               -74        -148
         18   0.7984 Lucky Charm Chasm             76         -18
         18   0.7984 Lucky Charm Chasm            -70         -37
         14   0.9599 Rainbow's End Realm           32         -19
         29   0.5101 Lucky Penny Pastures         -62         -52
         18   0.7984 Lucky Charm Chasm            -32          19

  10 record(s) selected.

db2 => 
```

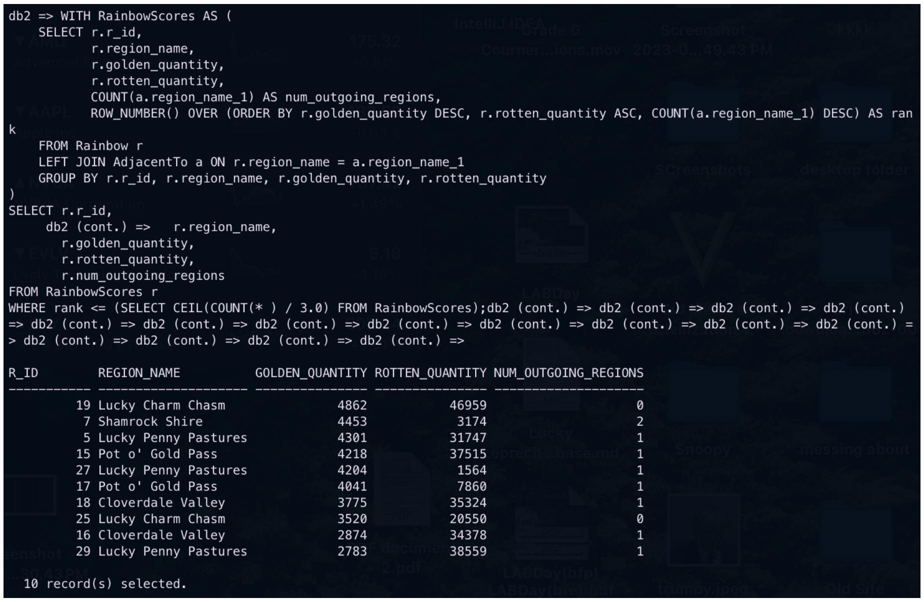
Query5

Description of what information the query returns:

We want to know the rainbows that are the “best.” This means we want to maximize the gold quantity belonging to that rainbow, minimize the rotten_quantity, and we want the rainbow to ideally have many regions which it can go to. We grab the top third of Rainbows who fit these criteria ### The SQL Query in plaintext: WITH RainbowScores AS (SELECT r.r_id, r.region_name, r.golden_quantity, r.rotten_quantity, COUNT(a.region_name_1) AS num_outgoing_regions, ROW_NUMBER() OVER (ORDER BY r.golden_quantity DESC, r.rotten_quantity ASC, COUNT(a.region_name_1) DESC) AS rank FROM Rainbow r LEFT JOIN AdjacentTo a ON r.region_name = a.region_name_1 GROUP BY r.r_id, r.region_name, r.golden_quantity, r.rotten_quantity) SELECT r.r_id, r.region_name, r.golden_quantity, r.rotten_quantity, r.num_outgoing_regions FROM RainbowScores r WHERE rank <= (SELECT CEIL(COUNT(*) / 3.0)

FROM RainbowScores);

Screenshot of the query being executed (always include all of the SQL statement):



```
db2 => WITH RainbowScores AS (
    SELECT r.r_id,
           r.region_name,
           r.golden_quantity,
           r.rotten_quantity,
           COUNT(a.region_name_1) AS num_outgoing_regions,
           ROW_NUMBER() OVER (ORDER BY r.golden_quantity DESC, r.rotten_quantity ASC, COUNT(a.region_name_1) DESC) AS ran
k
    FROM Rainbow r
    LEFT JOIN AdjacentTo a ON r.region_name = a.region_name_1
    GROUP BY r.r_id, r.region_name, r.golden_quantity, r.rotten_quantity
)
SELECT r.r_id,
     db2 (cont.) =>   r.region_name,
       r.golden_quantity,
       r.rotten_quantity,
       r.num_outgoing_regions
FROM RainbowScores r
WHERE rank <= (SELECT CEIL(COUNT(* ) / 3.0) FROM RainbowScores);db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.)
=> db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =
> db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>

R_ID        REGION_NAME          GOLDEN_QUANTITY ROTTEN_QUANTITY NUM_OUTGOING_REGIONS
----------- -------------------- --------------- --------------- --------------------
         19 Lucky Charm Chasm               4862           46959                    0
          7 Shamrock Shire                  4453            3174                    2
          5 Lucky Penny Pastures            4301           31747                    1
         15 Pot o' Gold Pass                4218           37515                    1
         27 Lucky Penny Pastures            4204            1564                    1
         17 Pot o' Gold Pass                4041            7860                    1
         18 Cloverdale Valley               3775           35324                    1
         25 Lucky Charm Chasm               3520           20550                    0
         16 Cloverdale Valley               2874           34378                    1
         29 Lucky Penny Pastures            2783           38559                    1

  10 record(s) selected.
```

III. SQL Modifications

Mod1

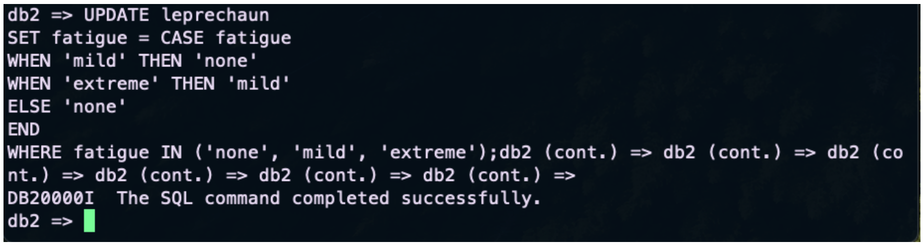
Description of what modification the statement performs:

This statement is updating the fatigue level of leprechauns. If a leprechaun is currently mildly tired, it becomes not tired. If a leprechaun is currently extremely tired, it becomes mildly tired. If a leprechaun is not tired, it remains not tired.

The SQL update/delete/insert statement in plaintext:

UPDATE leprechauns SET fatigue = CASE fatigue WHEN ‘mild’ THEN ‘none’ WHEN ‘extreme’ THEN ‘mild’ ELSE ‘none’ END WHERE fatigue IN (‘none’, ‘mild’, ‘extreme’);

Screenshot of the Mod being executed (always include all of the SQL statement):



```
db2 => UPDATE leprechaun
SET fatigue = CASE fatigue
WHEN 'mild' THEN 'none'
WHEN 'extreme' THEN 'mild'
ELSE 'none'
END
WHERE fatigue IN ('none', 'mild', 'extreme');db2 (cont.) => db2 (cont.) => db2 (co
nt.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>
DB20000I  The SQL command completed successfully.
db2 =>
```

Mod2

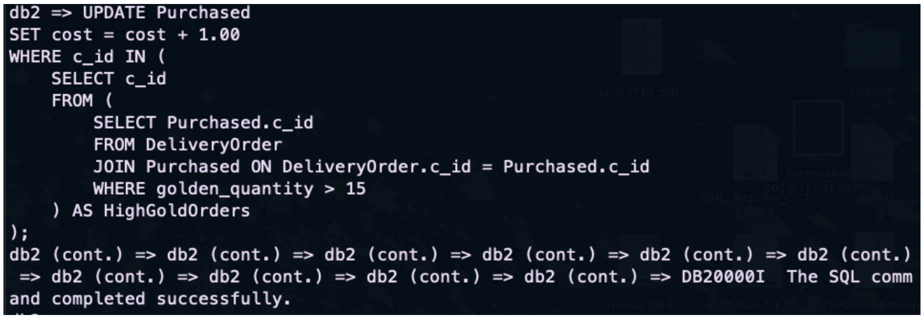
Description of what modification the statement performs:

This modification adds 1.00 to the cost of each purchase made by customers who placed orders with a golden potato quantity greater than 15.

The SQL update/delete/insert statement in plaintext:

UPDATE Purchased SET cost = cost + 1.00 WHERE c_id IN (SELECT c_id FROM (SELECT Purchased.c_id FROM DeliveryOrder JOIN Purchased ON DeliveryOrder.c_id = Purchased.c_id WHERE golden_quantity > 15) AS HighGoldOrders);

Screenshot of the Mod being executed (always include all of the SQL statement):



```
db2 => UPDATE Purchased
SET cost = cost + 1.00
WHERE c_id IN (
    SELECT c_id
    FROM (
        SELECT Purchased.c_id
        FROM DeliveryOrder
        JOIN Purchased ON DeliveryOrder.c_id = Purchased.c_id
        WHERE golden_quantity > 15
    ) AS HighGoldOrders
);
db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.)
 => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => DB20000I  The SQL comm
and completed successfully.
```

IV. Views

View1

Description of what the view represents:

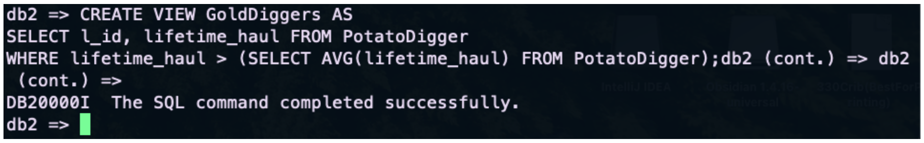
It is essentially selecting Leprechauns (identified by their “l_id”) who have collected more potatoes over their lifetime than the average potato haul among all Leprechauns in the dataset. This view provides a filtered perspective on the dataset, focusing only on those Leprechauns who are particularly proficient at digging potatoes.

The The CREATE VIEW statement:

The CREATE VIEW statement:

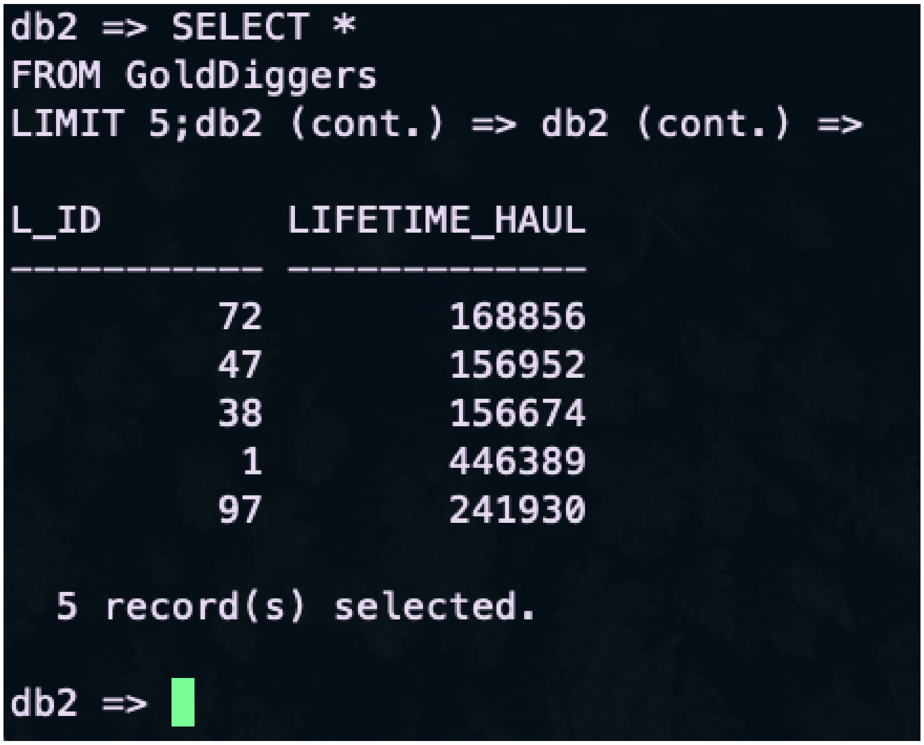
CREATE VIEW GoldDiggers AS SELECT l_id, lifetime_haul FROM PotatoDigger WHERE lifetime_haul > (SELECT AVG(lifetime_haul) FROM PotatoDigger);

Screenshot the statement showing both input and output statement):



```
db2 => CREATE VIEW GoldDiggers AS
SELECT l_id, lifetime_haul FROM PotatoDigger
WHERE lifetime_haul > (SELECT AVG(lifetime_haul) FROM PotatoDigger);db2 (cont.) => db2
 (cont.) =>
DB20000I  The SQL command completed successfully.
db2 =>
```

A Screenshot of an SQL query that selects everything from the view truncated to 5 records):



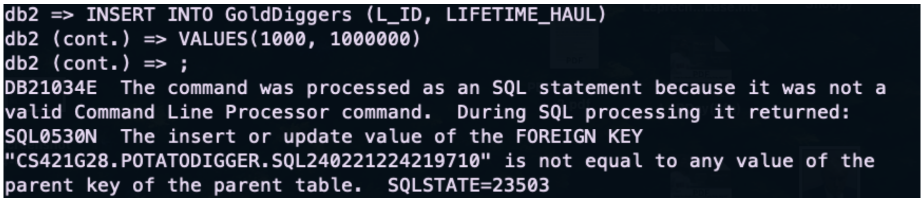
```
db2 => SELECT *
FROM GoldDiggers
LIMIT 5;db2 (cont.) => db2 (cont.) =>

L_ID        LIFETIME_HAUL
----------- -------------
         72        168856
         47        156952
         38        156674
          1        446389
         97        241930

  5 record(s) selected.

db2 =>
```

A Screenshot of when you insert a new, valid record, and result:



```
db2 => INSERT INTO GoldDiggers (L_ID, LIFETIME_HAUL)
db2 (cont.) => VALUES(1000, 1000000)
db2 (cont.) => ;
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0530N  The insert or update value of the FOREIGN KEY
"CS421G28.POTATODIGGER.SQL240221224219710" is not equal to any value of the
parent key of the parent table.  SQLSTATE=23503
```

An Explanation of what the result means and the DB2 manual description associated with it:

This failed insert operation involved inserting a value into a column that has a foreign key constraint, but the value being inserted does not exist in the referenced parent table's primary key column. This violates referential integrity because the foreign key value is attempting to reference a non-existent primary key value.

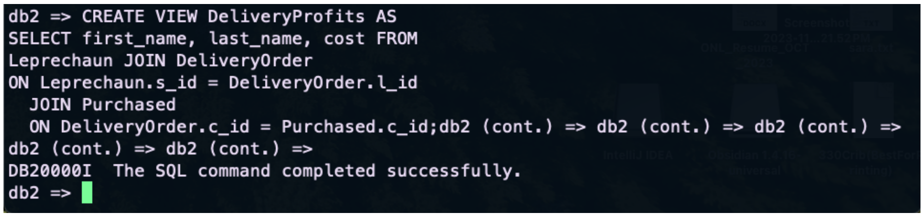
View2

Description of what the view represents:

This view provides a list of leprechauns' names alongside the cost of the delivery orders they are associated with, by joining data from the Leprechaun, DeliveryOrder, and Purchased tables. ### The CREATE VIEW statement:

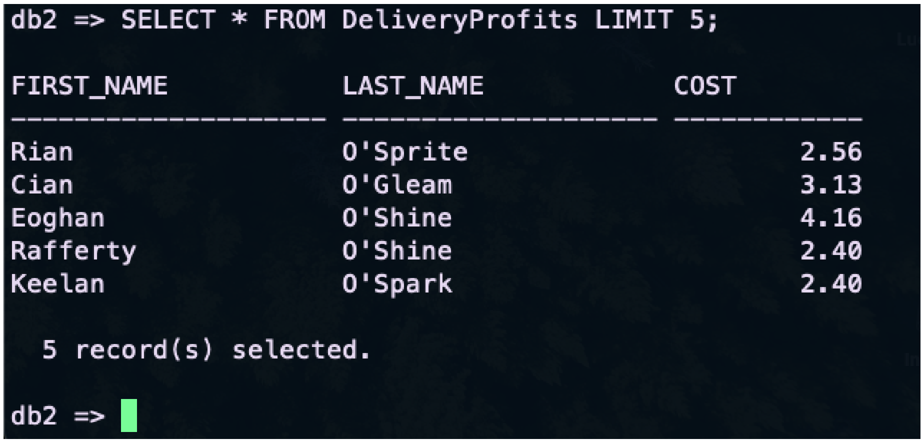
CREATE VIEW DeliveryProfits AS SELECT first_name, last_name, cost FROM Leprechaun JOIN DeliveryOrder ON Leprechaun.s_id = DeliveryOrder.l_id JOIN Purchased ON DeliveryOrder.c_id = Purchased.c_id;

Screenshot the statement showing both input and output statement):



```
db2 => CREATE VIEW DeliveryProfits AS
SELECT first_name, last_name, cost FROM
Leprechaun JOIN DeliveryOrder
ON Leprechaun.s_id = DeliveryOrder.l_id
  JOIN Purchased
  ON DeliveryOrder.c_id = Purchased.c_id;db2 (cont.) => db2 (cont.) => db2 (cont.) =>
db2 (cont.) => db2 (cont.) =>
DB20000I  The SQL command completed successfully.
db2 =>
```

A Screenshot of an SQL query that selects everything from the view truncated to 5 records):



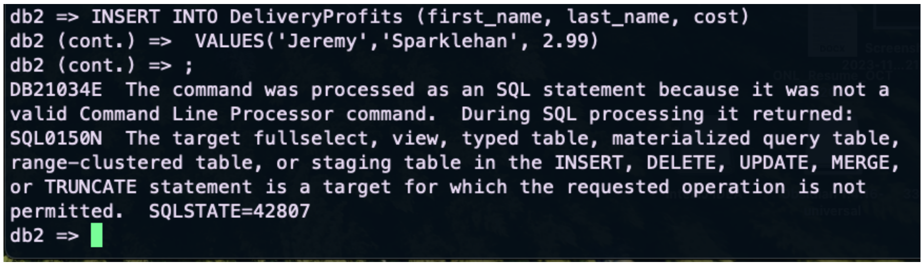
```
db2 => SELECT * FROM DeliveryProfits LIMIT 5;

FIRST_NAME           LAST_NAME            COST
-------------------- -------------------- ------------
Rian                 O'Sprite                     2.56
Cian                 O'Gleam                      3.13
Eoghan               O'Shine                      4.16
Rafferty             O'Shine                      2.40
Keelan               O'Spark                      2.40

  5 record(s) selected.

db2 =>
```

A Screenshot of when you insert a new, valid record, and result:



```
db2 => INSERT INTO DeliveryProfits (first_name, last_name, cost)
db2 (cont.) =>  VALUES('Jeremy','Sparklehan', 2.99)
db2 (cont.) => ;
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0150N  The target fullselect, view, typed table, materialized query table,
range-clustered table, or staging table in the INSERT, DELETE, UPDATE, MERGE,
or TRUNCATE statement is a target for which the requested operation is not
permitted.  SQLSTATE=42807
db2 =>
```

An Explanation of what the result means and the DB2 manual description associated with it:

As explained in the DB2 manual, the error was caused by the fact that “The view definition contains a join, a GROUP BY, or a HAVING clause.” and our View clearly has a JOIN clause in it. Thus our insert fails.

V. Check Constraints

Check1

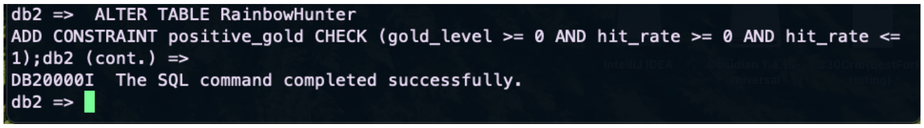
Describe what the constraint achieves

The “positive_gold” constraint ensures that the “RainbowHunter” table only contains data where both the gold level and hit rate are non-negative values, and the hit rate does not exceed 1 (or 100%). This helps maintain data integrity and consistency within the table.

Show the constraint statement :

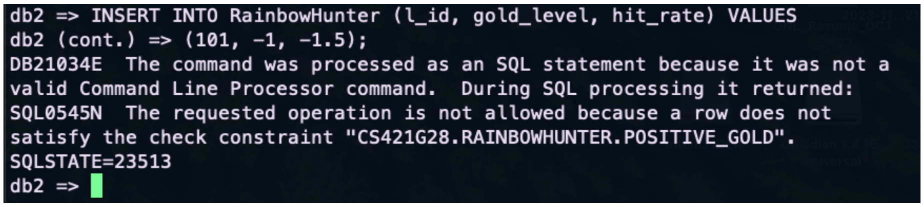
ALTER TABLE RainbowHunter ADD CONSTRAINT positive_gold CHECK (gold_level >= 0 AND hit_rate >= 0 AND hit_rate <= 1);

A Screenshot of command running on the database:



```
db2 =>  ALTER TABLE RainbowHunter
ADD CONSTRAINT positive_gold CHECK (gold_level >= 0 AND hit_rate >= 0 AND hit_rate <=
1);db2 (cont.) =>
DB20000I  The SQL command completed successfully.
db2 =>
```

Screenshot of the execution trying to insert a record that violates the constraint, and resulting error:



```
db2 => INSERT INTO RainbowHunter (l_id, gold_level, hit_rate) VALUES
db2 (cont.) => (101, -1, -1.5);
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0545N  The requested operation is not allowed because a row does not
satisfy the check constraint "CS421G28.RAINBOWHUNTER.POSITIVE_GOLD".
SQLSTATE=23513
db2 =>
```

Check2

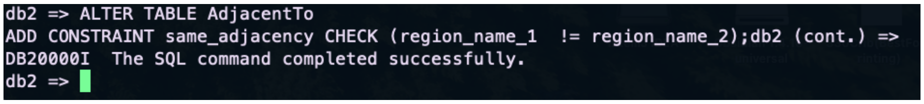
Describe what the constraint achieves

This constraint guarantees that each adjacency relationship defined in the “AdjacentTo” table links two distinct regions. It prevents scenarios where a region is mistakenly considered adjacent to itself, which would not make logical sense in the context of adjacency between regions.

Show the constraint statement :

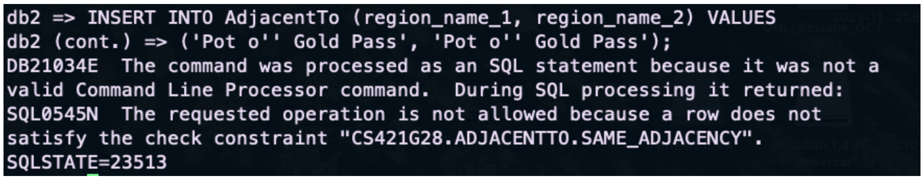
ALTER TABLE AdjacentTo ADD CONSTRAINT same_adjacency CHECK (region_name_1 != region_name_2);

A Screenshot of command running on the database:



```
db2 => ALTER TABLE AdjacentTo
ADD CONSTRAINT same_adjacency CHECK (region_name_1  != region_name_2);db2 (cont.) =>
DB20000I  The SQL command completed successfully.
db2 =>
```

Screenshot of the execution trying to insert a record that violates the constraint, and resulting error:



```
db2 => INSERT INTO AdjacentTo (region_name_1, region_name_2) VALUES
db2 (cont.) => ('Pot o'' Gold Pass', 'Pot o'' Gold Pass');
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0545N  The requested operation is not allowed because a row does not
satisfy the check constraint "CS421G28.ADJACENTTO.SAME_ADJACENCY".
SQLSTATE=23513
```

VI. Creativity

REAL DATA SETS:

Description

The Leprechaun table was populated with 100 leprechauns with suitable Irish names generated by chatGPT, eg. Paddy McShenanigans, Lorcan O'Fiddlesticks, Liam Leprechaunson, and Lucky O'Charms. Their birthdays were randomly generated between the years 749-1100 as only the wisest and most experienced leprechauns can work for Lucky Leprechaun's rainbow potato business. The fatigue attribute was generated to be a 70-20-10 split between 'none', 'mild', and 'extreme' and 10% of leprechauns are in training. The Leprechaun entity is related to the RainbowHunter, PotatoDigger, FarmstandClerk, and PotatoCourier entities through an ISA relationship and thus the four tables representing these entities contain a foreign key to the leprechaun table. The SensitivityTraining, TravellingOnRainbow, and CanWorkIn tables also have a foreign key to the Leprechaun table. The leprechauns in training have the most recent sensitivity training dates and none of them are assigned a job.

ADVANCED SQL FEATURES:

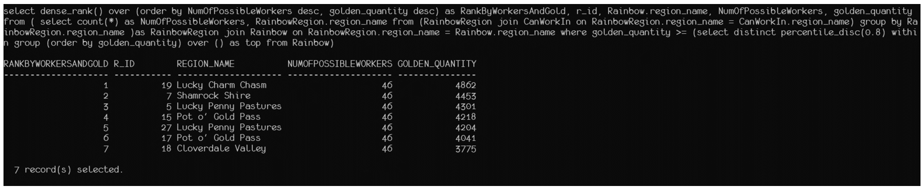
Description

For occasions when we need to have the biggest golden potato yield possible, we need to find not only the Rainbow regions that are rich in golden potatoes, but also have a high number of workers that can work in those regions. This will be done using the DENSE_RANK() OLAP function to rank the records respectively by their number of golden potatoes and by the number of Leprechauns that can harvest them. Additionally, using the PERCENTILE_DISC() OLAP function, we can obtain the desired percentile among all the Rainbow Regions in order to ultimately identify the regions of the highest grade.

SQL statements

```
select dense_rank() over (order by NumOfPossibleWorkers desc, golden_quantity desc) as RankByWorkersAndGold, r_id, Rainbow.region_name, NumOfPossibleWorkers, golden_quantity from ( select count(*) as NumOfPossibleWorkers, RainbowRegion.region_name from (RainbowRegion join CanWorkIn on RainbowRegion.region_name = CanWorkIn.region_name) group by RainbowRegion.region_name )as RainbowRegion join Rainbow on RainbowRegion.region_name = Rainbow.region_name where golden_quantity >= (select distinct percentile_disc(0.8) within group (order by golden_quantity) over () as top from Rainbow);
```

Screenshots:



```
select dense_rank() over (order by NumOfPossibleWorkers desc, golden_quantity desc) as RankByWorkersAndGold, r_id, Rainbow.region_name, NumOfPossibleWorkers, golden_quantity
from ( select count(*) as NumOfPossibleWorkers, RainbowRegion.region_name from (RainbowRegion join CanWorkIn on RainbowRegion.region_name = CanWorkIn.region_name) group by Ra
inbowRegion.region_name )as RainbowRegion join Rainbow on RainbowRegion.region_name = Rainbow.region_name where golden_quantity >= (select distinct percentile_disc(0.8) withi
n group (order by golden_quantity) over () as top from Rainbow)

RANKBYWORKERSANDGOLD R_ID        REGION_NAME          NUMOFPOSSIBLEWORKERS GOLDEN_QUANTITY
-------------------- ----------- -------------------- -------------------- ---------------
                   1          19 Lucky Charm Chasm                      46            4862
                   2           7 Shamrock Shire                         46            4453
                   3           5 Lucky Penny Pastures                   46            4301
                   4          15 Pot o' Gold Pass                       46            4218
                   5          27 Lucky Penny Pastures                   46            4204
                   6          17 Pot o' Gold Pass                       46            4041
                   7          18 Cloverdale Valley                      46            3775

  7 record(s) selected.
```

COMPLEX ANALYTICAL QUERIES:

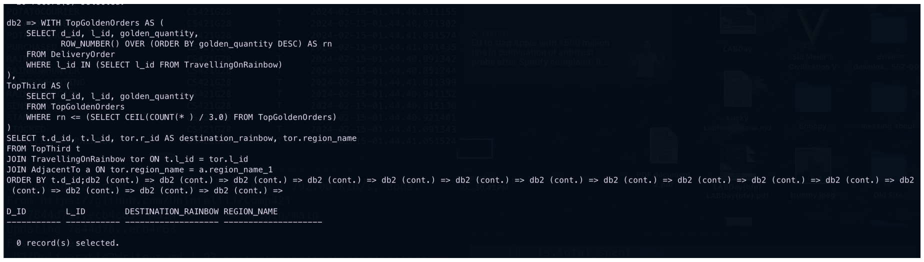
Description

Where's the gold going: We want the top third of all delivery leprechauns, by Golden Quantity, who are actively traveling on rainbows, and where they are traveling to based on their current rainbow. This will allow us to live-track where our most valuable orders are going to, and in the case of a loss or failed delivery, we can have precise information to give the Leprechaun CIA. Fortunately for our current staff of orders and delivery leprechauns, nobody who is in the upper echelon seems to be traveling on a rainbow at the moment... ITS A CONSPIRACY!!!!

SQL statements

WITH TopGoldenOrders AS (SELECT d_id, l_id, golden_quantity, ROW_NUMBER() OVER (ORDER BY golden_quantity DESC) AS rn FROM DeliveryOrder WHERE l_id IN (SELECT l_id FROM TravellingOnRainbow)), TopThird AS (SELECT d_id, l_id, golden_quantity FROM TopGoldenOrders WHERE rn <= (SELECT CEIL(COUNT(*) / 3.0) FROM TopGoldenOrders)) SELECT t.d_id, t.l_id, tor.r_id AS destination_rainbow, tor.region_name FROM TopThird t JOIN TravellingOnRainbow tor ON t.l_id = tor.l_id JOIN AdjacentTo a ON tor.region_name = a.region_name_1 ORDER BY t.d_id;

Screenshots:



```
db2 => WITH TopGoldenOrders AS (
    SELECT d_id, l_id, golden_quantity,
           ROW_NUMBER() OVER (ORDER BY golden_quantity DESC) AS rn
    FROM DeliveryOrder
    WHERE l_id IN (SELECT l_id FROM TravellingOnRainbow)
),
TopThird AS (
    SELECT d_id, l_id, golden_quantity
    FROM TopGoldenOrders
    WHERE rn <= (SELECT CEIL(COUNT(* ) / 3.0) FROM TopGoldenOrders)
)
SELECT t.d_id, t.l_id, tor.r_id AS destination_rainbow, tor.region_name
FROM TopThird t
JOIN TravellingOnRainbow tor ON t.l_id = tor.l_id
JOIN AdjacentTo a ON tor.region_name = a.region_name_1
ORDER BY t.d_id;db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) => db2
 (cont.) => db2 (cont.) => db2 (cont.) => db2 (cont.) =>

D_ID        L_ID        DESTINATION_RAINBOW REGION_NAME
----------- ----------- ------------------- --------------------

  0 record(s) selected.
```

VIII. Group Paragraph

As a group, we worked together very well. We had 2 different Zoom meetings to discuss our workload separation, group progess, and load management. These meetings were easily facilitated by constant and open communication between all group members. We assigned each group member sections of the project in equal point proportion and kept in constant communication to be able to relay the parts one after another. This made the work fairly easy and straightforward, as it put little pressure on any one individual. We were easily able to submit a polished Deliverable before the deadline. This project was made slightly more challenging by the illness of one group member, however we were able to overcome through this adversity.Every group member reviewed and approved the Deliverable before submission.