MLDS-413 Introduction to Databases and Information Retrieval

Homework 8: Triggers, Integrity Constraints, Transactions, Views, and Window Functions

Name 1: ____

NetID 1: ____

Name 2: ____

NetID 2: ____

Instructions

You should submit this homework assignment via Canvas. Acceptable formats are word files, text files, and pdf files. Paper submissions are not allowed and they will receive an automatic zero.

As explained during lecture and in the syllabus, assignments are done in groups. The groups have been created and assigned. Each group needs to submit only one assignment (i.e., there is no need for both partners to submit individually the same homework assignment).

Each group can submit solutions multiple times (for example, you may discover an error in your earlier submission and choose to submit a new solution set). We will grade only the last submission and ignore earlier ones.

Make sure you submit your solutions before the deadline. The policies governing academic integrity, tardiness and penalties are detailed in the syllabus.

SchoolScheduling.sqlite Database (30 points)

1. **(10 points)** Write the SQL statements that perform the following operations.
    a. **(1 point)** First, in preparation for this section of the homework, write and execute a query that inserts a new class status with ID 4 and description “Failed”.

```
INSERT INTO Student_Class_Status VALUES (4, "Failed");

-- confirm the change (you do not need to do this in your solution)
SELECT * FROM Student_Class_Status;

OUTPUT:
1      Enrolled
2      Completed
3      Withdrew
4      Failed
```

    b. **(3 points)** Start a new transaction.

```
SAVEPOINT TR1;
```

    c. **(3 points)** Update the class status in student 1001’s schedule for class 4180 to “Completed”.

```
UPDATE Student_Schedules
       SET classStatus = (SELECT ClassStatus
                          FROM Student_Class_Status
                          WHERE ClassStatusDescription = 'Completed')
WHERE StudentID=1001 AND classID=4180;

-- let’s confirm the update worked (you do not need to do this in your solution)
SELECT * FROM Student_Schedules NATURAL JOIN Student_Class_Status WHERE StudentID=1001;

OUTPUT:
1001   1000   2      99.83  Completed
1001   1168   2      70.0   Completed
1001   2907   2      67.33  Completed
1001   3085   2      87.14  Completed
1001   4180   2      0.0    Completed
1001   5917   1      0.0    Enrolled
1001   6082   1      0.0    Enrolled
```

    d. **(3 points)** The problem now is that if the administrator forgets to set 1001’s grade for class 4180, the student will have a class marked as completed with grade 0. You want to enforce a rule that a class cannot be marked completed unless the student has already received a passing grade, i.e., a grade of at least 60.0. You want to implement your integrity rules first and then do the data updates. Abort the transaction you started in part (b) in order to undo the changes in part (c). Do not undo the changes of part (a), though.

```
-- Do not use plain ROLLBACK. It will abort all outstanding transactions
-- and also undo the row insertion in question 1 (which we intend to keep)
ROLLBACK TO TR1;

-- let’s confirm transaction TR1 rolled back (you do not need to do this in your solution)
SELECT * FROM Student_Schedules NATURAL JOIN Student_Class_Status WHERE StudentID=1001;

OUTPUT:
1001   1000   2      99.83  Completed
1001   1168   2      70.0   Completed
1001   2907   2      67.33  Completed
1001   3085   2      87.14  Completed
1001   4180   2      0.0    Enrolled
1001   5917   1      0.0    Enrolled
1001   6082   1      0.0    Enrolled
```

```
   -- confirm Q1’s INSERT did not roll back (you do not need to do this in your solution)
   SELECT * FROM Student_Class_Status;

   OUTPUT:
   1     Enrolled
   2     Completed
   3     Withdrew
   4     Failed
```

2. **(10 points)** Write a query that enforces the data integrity constraint described in question Q1.d.

```
CREATE TRIGGER NoCompleteClassTrigger
BEFORE UPDATE OF ClassStatus ON Student_Schedules
FOR EACH ROW
BEGIN
   SELECT CASE WHEN (new.classStatus=(SELECT ClassStatus
                                       FROM Student_Class_Status
                                       WHERE ClassStatusDescription = 'Completed')
                       AND new.Grade < 60.0)
          THEN RAISE(FAIL, "ERROR: cannot mark a class complete without grade >= 60.0")
          END;
END;

-- verify that the update of question 2 now fails with an error
-- you do not have to do this in your solution
UPDATE Student_Schedules
       SET classStatus = (SELECT ClassStatus
                       FROM Student_Class_Status
                       WHERE ClassStatusDescription = 'Completed')
WHERE StudentID=1001 AND classID=4180;

OUTPUT:
Execution finished with errors.
Result: ERROR: cannot mark a class complete without grade >= 60.0
```

3. **(10 points)** Write a query that enforces the following data integrity constraint: when a grade changes from a non-passing grade to a passing grade, automatically set the status of the class to completed. If the grade changes to a non-passing grade, set the class status to failed.

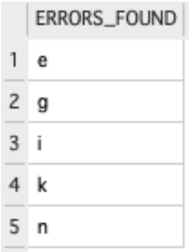
```
CREATE TRIGGER UpdateGradeTrigger
AFTER UPDATE OF Grade ON Student_Schedules
FOR EACH ROW
WHEN old.Grade != new.Grade
BEGIN
   UPDATE Student_Schedules SET classStatus =
          (CASE WHEN new.Grade >= 60.0 THEN (SELECT ClassStatus
                                             FROM Student_Class_Status
                                             WHERE ClassStatusDescription = 'Completed')
          ELSE (SELECT ClassStatus
                 FROM Student_Class_Status
                 WHERE ClassStatusDescription = 'Failed')
          END)
   WHERE StudentID = new.StudentID AND ClassID = new.ClassID;
END;

-- to verify the trigger works as expected, update a student’s grades and
-- verify that student’s class status for these classes has been automatically updated
-- you do not have to do this in your solution

-- set student’s 1001 grade for class 4180 to 98.8 and for class 5917 to 59.9
UPDATE Student_Schedules SET grade=98.8 WHERE StudentID=1001 AND classID=4180;
UPDATE Student_Schedules SET grade=59.9 WHERE StudentID=1001 AND classID=5917;

-- verify that student’s status for these classes has been automatically updated
-- to Completed and Failed respectively
SELECT * FROM Student_Schedules NATURAL JOIN Student_Class_Status WHERE StudentID=1001;
```

```
OUTPUT:
1001     1000  2     99.83 Completed
1001     1168  2     70.0  Completed
1001     2907  2     67.33 Completed
1001     3085  2     87.14 Completed
1001     4180  2     98.8  Completed
1001     5917  4     59.9  Failed
1001     6082  1     0.0   Enrolled
```

Homework 5 Question 6 Solution Database (10 points)

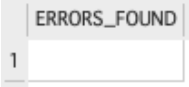
4. **(10 points)** Sometimes you want to only check integrity constraints, not enforce them. One way to do that is to create a view that you examine whenever you want to verify data integrity. One such example are the constraints (e), (g), (i), (k), and (n) in Homework 5 Question 6.

For this assignment, you will use the Homework 5 Question 6 solution database. Your goal is to create a view named `check_db` that checks if the database violates any of the constraints (e), (g), (i), (k), and (n). The view should return a table that lists all the violated constraints, or an empty row if there are no violations. For example, if constraints e, g, i, k, and n are all violated, the view will be the table below (do not worry about the row order):



	ERRORS_FOUND
1	e
2	g
3	i
4	k
5	n

Note that it is OK if some of the rows of your result are empty rows. Similarly, if no constraints are violated, the view could simply return a table with an empty row:



	ERRORS_FOUND
1	

To check that your view works properly, you can execute the following deletions on the Homework 5 Question 6 solution database and check the output of your query after each deletion set, as the comments and the SQL queries below show. The queries should be executed in the exact order below to achieve each of the stated results.

Suggestion: use transactions when you are experimenting in this question. This way, when things don't work, you can simply rollback the changes. Note that if you rollback once you will need to start a new transaction again to be able to rollback your changes a second time (`ROLLBACK` will undo your changes AND terminate the transaction). So, it is better to use `SAVEPOINT X` and `ROLLBACK TO X`. This way you can issue `ROLLBACK TO X` as many times as you want without taking a new savepoint (`ROLLBACK TO X` will undo the changes but the transaction remains active, so there is no need to remember to start a new one each time).

```
-- check that your view works by examining it on the HW5 Q6 solution database before any deletions
-- your view should be just an empty row (i.e., there are no violations)
SELECT * FROM check_db;

-- performing the following deletions would violate the following constraint
-- k. Each invoice has at least one invoice item
-- your view should contain a row for k
DELETE FROM invoice_items WHERE invoiceId IN (2, 3);
SELECT * FROM check_db;

-- performing the following additional deletions would violate the following additional constraint
-- e. Each album has at least one track
-- your view should contain rows for e, k
DELETE FROM tracks WHERE albumId=2;
SELECT * FROM check_db;

-- performing the following additional deletions would violate the following additional constraint
-- g. Each genre is represented by at least one track
-- your view should contain rows e, g, k
DELETE FROM tracks WHERE trackId=3451;
SELECT * FROM check_db;

-- performing the following additional deletions would violate the following additional constraint
-- i. Each media type is used by at least one track
-- your view should contain rows for e, g, i, k
DELETE FROM invoice_items WHERE trackId IN (SELECT trackId FROM tracks WHERE mediaTypeId=4);
DELETE FROM tracks WHERE mediaTypeId=4;
SELECT * FROM check_db;

-- performing the following additional deletions would violate the following additional constraint
-- n. Each customer has been issued at least one invoice
-- your view should contain rows for e, g, i, k, n
DELETE FROM invoice_items WHERE invoiceId IN (SELECT invoiceId FROM invoices WHERE customerId=20);
DELETE FROM invoices WHERE customerId=20;
SELECT * FROM check_db;
```

```
CREATE VIEW check_db AS
--- Each album has at least one track
    SELECT CASE WHEN (
            SELECT count(*)
            FROM albums
            LEFT NATURAL JOIN tracks
            WHERE tracks.albumId IS NULL) != 0
        THEN "e" ELSE "" END
    AS ERRORS_FOUND
    UNION
    --- Each genre is represented by at least one track
    SELECT CASE WHEN (
            SELECT count(*)
            FROM genres
            LEFT JOIN tracks on genres.GenreId=tracks.GenreId
            WHERE trackid IS NULL) != 0
        THEN "g" ELSE "" END
    UNION
    --- Each media type is used by at least one track
    SELECT CASE WHEN (
            SELECT count(*)
            FROM media_types
            LEFT JOIN tracks on media_types.MediaTypeId=tracks.MediaTypeId
            WHERE trackid IS NULL) != 0
        THEN "i" ELSE "" END
    UNION
    --- Each invoice has at least one invoice item
    SELECT CASE WHEN (
            SELECT count(*)
            FROM invoices
            LEFT JOIN invoice_items on invoices.invoiceId=invoice_items.invoiceId
            WHERE invoice_items.invoiceId IS NULL) != 0
        THEN "k" ELSE "" END
    UNION
    --- Each customer has been issued at least one invoice
    SELECT CASE WHEN (
            SELECT count(*)
            FROM customers
            LEFT JOIN invoices on customers.customerId= invoices.customerId
            WHERE invoices.customerId IS NULL) != 0
        THEN "n" ELSE "" END;
```

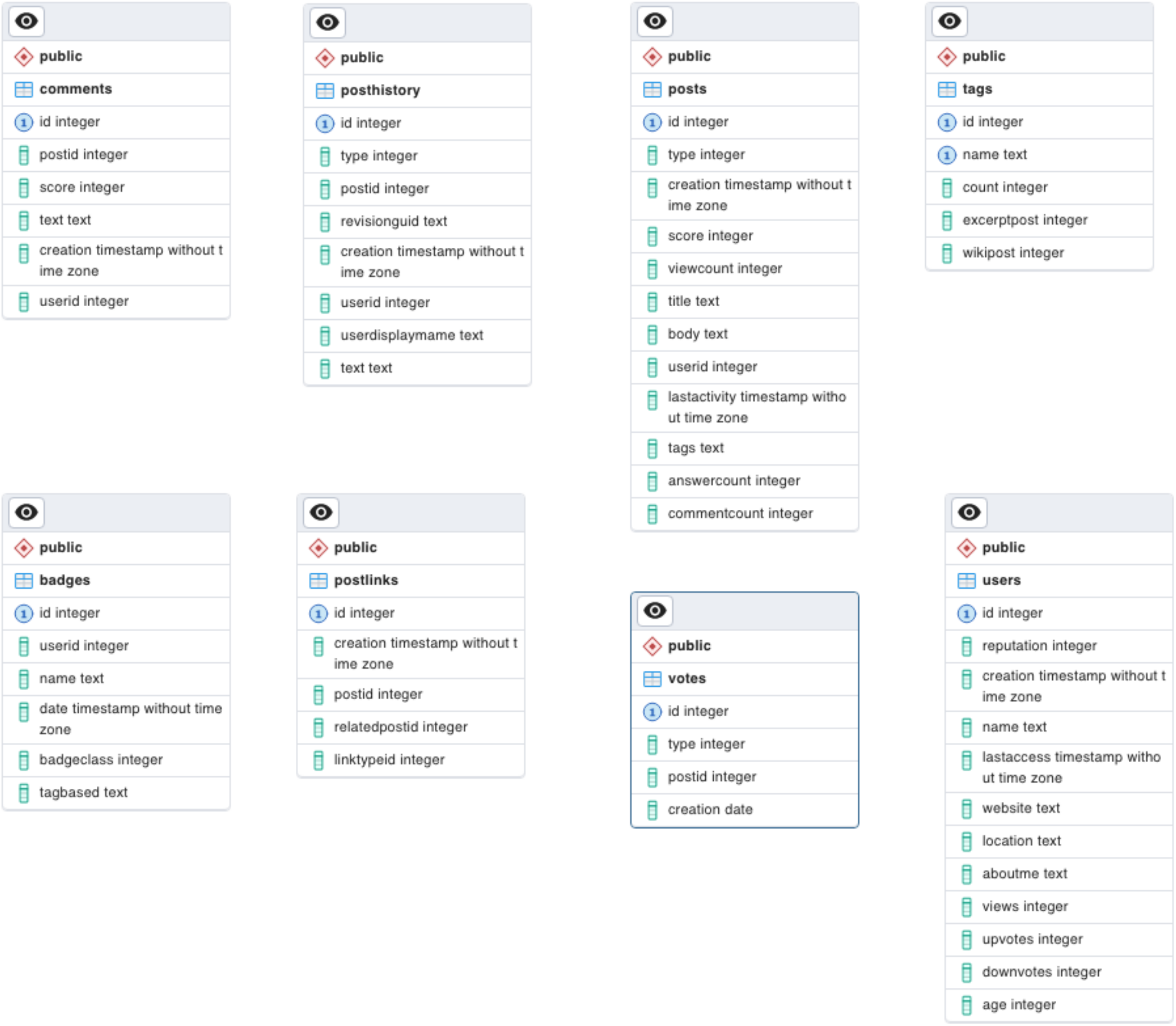
SalesOrders.sqlite Database (10 points)

5. **(10 points)** Monthly revenue growth is defined as the percent of revenue change of a month relative to the previous month, i.e., $(M_i - M_{i-1}) / M_{i-1}$. Write a query that will return the revenue growth of the sales in the SalesOrders database and provide your query's output. This should be a single query (CTE, windowing allowed).

```
WITH monthly_sales(Month, MonthlyTotal) AS (
    SELECT strftime('%Y-%m', OrderDate), SUM (OrderTotal)
    FROM Orders
    GROUP BY DATE(OrderDate, 'start of month'))
SELECT Month, 100.0 * (MonthlyTotal - LAG(MonthlyTotal, 1, 0) OVER win)
                   / LAG(MonthlyTotal, 1, 0) OVER win AS RevenueGrowth
FROM monthly_sales
WINDOW win AS (ORDER BY Month);

OUTPUT:
2012-09   NULL
2012-10   -11.4051029579918
2012-11   4.16176772211653
2012-12   -18.2580662671738
2013-01   51.316975769029
2013-02   -17.6799170116063
```

Stackoverflow Database (50 points)

Please follow the instructions from Homework 6 to connect to the Stackoverflow (so) database on MLDS's Postgres server. The database schema is provided below:



public
comments
id integer
postid integer
score integer
text text
creation timestamp without time zone
userid integer

public
posthistory
id integer
type integer
postid integer
revisionguid text
creation timestamp without time zone
userid integer
userdisplaymame text
text text

public
posts
id integer
type integer
creation timestamp without time zone
score integer
viewcount integer
title text
body text
userid integer
lastactivity timestamp without time zone
tags text
answercount integer
commentcount integer

public
tags
id integer
name text
count integer
excerptpost integer
wikipost integer

public
badges
id integer
userid integer
name text
date timestamp without time zone
badgeclass integer
tagbased text

public
postlinks
id integer
creation timestamp without time zone
postid integer
relatedpostid integer
linktypeid integer

public
votes
id integer
type integer
postid integer
creation date

public
users
id integer
reputation integer
creation timestamp without time zone
name text
lastaccess timestamp without time zone
website text
location text
aboutme text
views integer
upvotes integer
downvotes integer
age integer

Please note that the Stackoverflow database does not give any information about the relationship between different entities. You need to analyze each table and sample some data to **infer yourself** the relationships between tables. Unfortunately, the real world is often messy.

You will use this database to answer the following questions. Please make sure that your queries in this homework are read-only.

Unless otherwise noted, for each question please provide:

- The query you constructed
- The output of that query
- Any other information requested by the question

6. **(10 points)** How many posts are there that have never been edited after creation. Please provide **two different solutions** for this question. Hint: You can use many operations such as LEFT JOIN, EXCEPT, and EXIST.

```
SELECT COUNT(p.id)
FROM posts p
LEFT JOIN posthistory ph ON p.id = ph.postid
WHERE ph.postid IS NULL;

WITH unedited_post AS
   (SELECT DISTINCT p.id
   FROM posts p
   EXCEPT
   SELECT DISTINCT ph.postid
   FROM posthistory ph)
SELECT
   COUNT(*)
FROM unedited_post;

Output:
525
```

7. **(10 points)** Write a SQL query to count the number of posts that were created on Christmas Day (December 25th) for each year. Present the results in ascending years.

```
SELECT EXTRACT(YEAR FROM p.creation), count(p.id)
FROM posts p
WHERE EXTRACT(MONTH FROM p.creation) = 12
AND EXTRACT(DAY FROM p.creation) = 25
GROUP BY EXTRACT(YEAR FROM p.creation)
ORDER BY EXTRACT(YEAR FROM p.creation);

Output:
"date_part"     "count"
2008            545
2009            1670
2010            2861
2011            3848
2012            6590
2013            7625
2014            6358
2015            6554
2016            5786
2017            5841
2018            5696
2019            6215
2020            5918
2021            4332
2022            3945
```

8. **(10 points)** Rank users by their reputation and assign a percentile rank. Print the id, name, reputation, and the percentile rank of user 19787814 (user id).

```
WITH percentile_RK(id, name, reputation, percentile_rank) AS
   (SELECT id, name, reputation,
         NTILE(100) OVER (ORDER BY reputation DESC) AS percentile_rank
   FROM public.users)
SELECT *
FROM percentile_RK
WHERE id = 19787814;

Output:
   "id"         "name"       "reputation"       "percentile_rank"
   19787814     "Nova"       406                3
```

9. **(10 points)** For the post with ID 7518463 in postlinks (i.e., postlinks.postid = 7518463), find the related post (directly or **indirectly**) with the highest number of answers. In this question, you should only consider the postlink with linktypeid = 1. Hint: You need to use recursive query in this question. The directly related posts can be found in the table postlinks. Note: This is a prime example where real-world data are messy! There is a postlinks.postid = 7518463 but not a posts.id = 7518463. Most likely the user removed the post before the database dump, and now we have dangling references in the linking table, because the database designer did not enforce integrity constraints. It seems the database designer didn't take MLDS-413!

```
WITH RECURSIVE RelatedPosts(postid, relatedPostId, answerNum) AS (
   SELECT postlinks.postid, postlinks.relatedpostid, posts.answercount
      FROM postlinks
         JOIN posts ON postlinks.relatedpostid = posts.id
      WHERE postlinks.linktypeid = 1 AND postlinks.postid = 7518463
   UNION
   SELECT RelatedPosts.postid, pl.relatedpostid, p.answercount
      FROM RelatedPosts
         JOIN postlinks pl ON RelatedPosts.relatedPostId = pl.postid
         JOIN posts p ON pl.relatedpostid = p.id
      WHERE pl.linktypeid = 1
   )
SELECT MAX(answerNum) as max_answer
FROM RelatedPosts;

Output:
"max_answer"
407
```

10. **(10 points)** Find the month-over-month percentage growth in new posts in the year of 2022. Hint1: You may need to create some CTEs first. Hint2: You need to protect against the "divide by zero" error; divide only when it is safe to do so, otherwise set the corresponding percentage growth to NULL.

```
WITH monthly_posts AS (
   SELECT DATE_TRUNC('month', creation) AS month,
         COUNT(*) AS post_count
   FROM posts
   GROUP BY DATE_TRUNC('month', creation)
),
growth AS (
   SELECT month,
         post_count,
         LAG(post_count) OVER (ORDER BY month) AS previous_month_count
   FROM monthly_posts
)
SELECT month,
      CASE WHEN previous_month_count = 0 THEN NULL
         ELSE (post_count - previous_month_count) * 100.0 / previous_month_count
      END AS growth_percentage
FROM growth
WHERE EXTRACT(YEAR FROM month) = 2022
ORDER BY month;

Output:
   "month"                    "growth_percentage"
   "2022-01-01 00:00:00"     6.3879536822317583
   "2022-02-01 00:00:00"     -5.4174617370492539
   "2022-03-01 00:00:00"     7.1625724009356134
   "2022-04-01 00:00:00"     -6.5166524368679201
   "2022-05-01 00:00:00"     1.8361630497502742
   "2022-06-01 00:00:00"     -3.2969094757065878
   "2022-07-01 00:00:00"     -0.05548958891139515572
   "2022-08-01 00:00:00"     2.7123461056129957
   "2022-09-01 00:00:00"     0.89548628042442042873
   "2022-10-01 00:00:00"     1.8773555923542062
   "2022-11-01 00:00:00"     3.0783770763691993
   "2022-12-01 00:00:00"     -11.8776311095255084
```