**DSC 450: Database Processing for Large-Scale Analytics**

**Take-home Midterm**

Do not forget to include all python code and SQL code for every question (screenshots are only required when the question specifically asked for it).

**Part 1**

**(9 points)** Please give a “True” or a “False” rating to each statement below.

**a.** A schema that is in Third Normal Form (3NF) must also be in Second Normal Form (2NF).

**True**

**b.** A foreign key can contain a NULL.

**True**

**c.** The union rule allows combining functional dependencies A🡪CD and B🡪CD into a functional dependency AB🡪CD.

**False, not the union rule specifically by itself. However, you can make this conclusion with multiple rules and axioms:**

**A=>C A=>D B=>C B=>D (decomp)**

**AB=>BC AB=>AD (aug)**

**AB=> ABCD (union)**

**AB=>CD (decomp)**

**d.** SQL query results are always sorted by the primary key.

**False, when there is an order by statement in a query for another column**

**e.** A column that appears in the WHERE clause must always appear in the SELECT clause.

**False**

**f.** A UNIQUE constraint on a column allows insertion of a NULL.

**True, since a unique constraint doesn’t refer to the primary key, and the value of Null is technically unknown.**

**Part 2**

**a. (3 points)** Write a SQL column definition for PurchaseAmt (just for one column, don’t worry about the rest of the table) that ensures that the purchase amount must be between 0 and 999.99 (column definition should enforce **both** the number precision and greater-than-0 requirement).

**Create Table Purchased (**

**PurchaseAmt Number(5,2)**

**);**

**b. (3 points)** Write a SQL column definition for Street (just for one column, don’t worry about the rest of the table) that ensures that the street address is up to 17 characters and has to start from ‘rue’ (hint: CHECK constraint can use LIKE operator).

**Create Table Road (**

**Street VarChar2(17),**

**Constraint street\_col**

**Check (Street like ‘rue%’)**

**);**

**c. (12 points)** The following set of relations records information about university students, courses and assigned grades. The Student relation contains information about the student, including the name (full name is stored in one column), address and their year of graduation. The Course relation records information about courses: course name (primary key), course department and the number of credits provided by the course. Finally, the Grade relation records information about the grades given; CName is the foreign key referring to the primary key of the Course relation and StudentID is the foreign key referring to the primary key of the Student relation. The grades are a numeric value given on a 4-point system.

**Student(StudentID, Name, Address, GradYear)**

**Grade(CName, StudentID, CGrade)**

**Course(CName, Department, Credits)**

For each part below, write a single SQL query.

**Q1.** Display the list of student IDs and names for the students who graduated in most recent two years. You can assume that GradYear is an integer, but your query is not allowed to assume any particular year.

**Query:**

**--check for students graduating in 2020 or 2021**

**select studentid, name from student**

**where gradyear in (to\_char(sysdate, 'YYYY')-1,to\_char(sysdate, 'YYYY'));**

**Q2.** Display student names and their taken course names for all students with the middle name of ‘Muriel’. You may assume that name is always written as ‘First Middle Last’. Your query output should be sorted by grade.

**Query:**

**--Output Students and Course names for students with middle name 'Muriel'**

**select name, cname from student**

**--account for students with no courses taken**

**left outer join grade on (student.studentid = grade.studentid)**

**where name like '% Muriel %'**

**order by cgrade asc;**

**Q3.** For students who are either not enrolled in any courses or are enrolled in only 1 course, list those student’s names and graduation years.

**Query:**

**--Output student names and grad years not enrolled or enrolled in 1 course**

**select studentid, gradyear from student**

**where (select count(\*) from grade**

**where student.studentid = grade.studentid)<=1;**

**Q4.** Update all student records, to increase the graduation year by 2 for all students who live in Chicago

**Query:**

**--Add two years to gradyear on students with a city of Chicago address**

**update student set gradYear = gradYear + 2**

**where address like '%Chicago%';**

**Q5.** Modify the course table to add a Chair column that can be up to 26 characters (that question requires a DDL rather than a DML SQL statement)

**Query:**

**--Add a chair col with up to 26 characters in course table**

**alter table course**

**add Chair VarChar2(26);**

**Part 3**

**a. (8 points)**

* For the table below, fill in the missing values in W column, consistent with functional dependencies: XY🡪ZA , A🡪W. You can make any necessary assumptions, but be sure to state them.

**Since A🡪W, we know that each unique value of A defines a W value. Therefore the 4th row for col W has to be Dog, and the 5th row has to be Elephant due to the data given in the previous rows. Since we don’t have a value for A historically established in the table, the 6th row is unknown or Null.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **Y** | **Z** | **W** | **A** |
| 1 | 1 | 1000 | Cat | 2 |
| 1 | 2 | 4000 | Dog | 3 |
| 1 | 3 | 3000 | Elephant | 4 |
| 2 | 1 | 1000 | **Dog** | 3 |
| 2 | 2 | 2000 | **Elephant** | 4 |
| 2 | 3 | 5000 | **(Null)** | 5 |

* Given the schema R and the following functional dependencies:

R(X, Y, Z, W, A) with XY🡪ZA , A🡪W

does XY determine W? (XY🡪A?) Why or why not?

**Since we’re given XY🡪ZA, we know that XY🡪Z and XY🡪A due to the decomposition rule. In addition, we can conclude that XY🡪W from the other given statement (A🡪W) and from the transitivity rule (XY🡪A, A🡪W, XY🡪W).**

* Suppose that you were also given relation S:

S(P, Q, U, M, N)

What functional dependencies (if any) can you assume?

**We can define the functional dependency of the primary key (PUN), which determines each other attribute in the table:**

**PUN🡪 QM**

**PUN🡪 Q, PUN🡪M (decomposition rule)**

**b. (8 points)** Consider a TVShows table that keeps track of different TV shows. The table stores the show name and the year to which the entry refers. Additionally, each row stores channel name, length of the show, the average cost of an episode, and the filming location’s zip, city and state. Moreover, each entry contains the name of the lead actor and their salary.

The table is already in First Normal Form, and its primary key is (Show, Year).

The schema for the TVShows table is:

**(Show, Year, Channel, Length, Cost, Zip, City, State, Lead, Salary)**

You are given the following functional dependencies:

Show 🡪 Cost

Zip 🡪 City, State

Lead 🡪 Salary

Show, Year 🡪 Channel, Cost

* Remove any existing partial dependencies to create a set of linked relational schema (copying functional dependencies does not define a schema, be sure to include primary/foreign keys here) in Second Normal Form.

**To get rid of any partial dependencies, we have to get rid of all dependencies that are a subset of the primary key. In this case, we have Show => Cost. The arrow below reflects the foreign key:**

**(Show, Year, Channel, Length, Zip, City, State, Lead, Salary)**

**(Show, Cost)**

* Remove any existing transitive dependencies to create a set of linked relational schemas in Third Normal Form.

**To get rid of any transitive dependencies, we need to get rid of any dependencies where the determinant is not a part of the primary key. Zip=> City,State and Lead=> Salary. The arrows reflect foreign keys:**

**(Show, Year, Channel, Length, Cost, Zip, Lead)**

**(Show, Cost)**

**(Zip, City, State)**

**(Lead, Salary)**

**c. (7 points)**

Given the schema R and the following functional dependencies:

R(A, B, C, D, E) with AB🡪C, C🡪D

* Describe how to identify a primary key for relation R (the primary key is a minimal set of columns that determines all columns in the relation such that **?** 🡪 ABCDE )

**To find the primary key, we start with AB => C (given) and AB => D (transitivity). Additionally, ABE => CE, ABE => DE due to augmentation. So, ABE => C and ABE=> D (decomposition). Finally, since ABE => ABE, we can conclude that ABE => A, ABE=>B, ABE=>E (reflexivity), and therefore our primary key is ABE.**

* Decompose relation R into a relational schema in third normal form

**Since AB=>CD, this is a partial dependency that needs to be decomposed in order to get to 2NF. Then, since C=>D, this is a transitive dependency that needs to be decomposed in order to get 3NF. The arrows below denote foreign keys:**

**(A, B, E)**

**(A, B, C, D)**

**(C, D)**

**Part 4**

**(50 points)**

Create the schema from Part 2-c in SQLite and populate it with data of at least 4 students, 4 courses, and 8 enrollments (at least one of the students should not be enrolled in any courses and at least one course should have zero current enrollments).

1. Create a view that pre-joins the three tables, including all of the records from student table (i.e., including the non-enrolled students).

**PYTHON CODE: (also in .py file)**

**def createView():**

**'''creates a view that prejoins 3 tables from part 2c'''**

**#setup connection to sqlite**

**conn = sqlite3.connect('dsc450.db')**

**cursor = conn.cursor()**

**#setup view query**

**query1 = '''create view summary as**

**select grade.cname,grade.studentid,cgrade,name,address,gradyear,department,credits from student**

**left outer join grade on (grade.studentid = student.studentid)**

**left outer join course on (grade.cname = course.cname);'''**

**#drop view if already created, otherwise create view**

**try:**

**cursor.execute(f'Drop view {query1}')**

**except:**

**pass**

**cursor.execute(query1)**

**#close/commit db**

**conn.commit()**

**conn.close()**

1. Write and execute python code that uses that view to export all data into a single .txt file (that is a “de-normalized” 1NF file with some redundancy present). This code should include NULLs due to the non-enrolled students.

**Include a screenshot of the output .txt file (in addition to the python code)**

Text

Description automatically generated

**PYTHON CODE: (also in .py file)**

**def txtfile():**

**'''takes the view and writes it into a .txt file'''**

**#setup connection to sqlite**

**conn = sqlite3.connect('dsc450.db')**

**cursor = conn.cursor()**

**#open write file**

**of = open('summary.txt', 'w')**

**#grab all view data**

**query1 = 'select \* from summary'**

**content = cursor.execute(query1)**

**#place data into txt file**

**for row in content:**

**for i in range(len(row)):**

**if i == 7:**

**of.write(str(row[i]))**

**else:**

**of.write(str(row[i])+', ')**

**of.write('\n')**

**#close/commit db and txt file**

**conn.commit()**

**conn.close()**

**of.close()**

1. Add a new row to the de-normalized .txt file (you can manually edit the .txt file from part b) that violates the following functional dependency:

CName 🡪 Credits

(you can do so by creating a new record that repeats the course name but does not repeat the number of credits associated with this course name)

**I added the following row to the .txt file:**

**(Healing and Potions, 1, 1, Ash Ketchum, Pallet Town, 2024, Beginner, 4)**

**Where healing and potions typically has 2 credits, I have it as 4 credits for this entry.**

1. Write python code that will identify the values for which functional dependency was violated in your .txt file (hint: when the functional dependency is valid, there is only one unique value of Credits for each CName). Your solution should detect any violation of the CName 🡪 Credits functional dependency, not just your example. Keep in mind that functional dependency is violated only in the pre-joined .txt file, not in the SQLite database, so this solution must read data from .txt file.

**PYTHON CODE: (also in .py file)**

**def catchFD():**

**'''finds any FD violations of cname => credits and states them'''**

**#open file, store content**

**infile = open('summary.txt','r')**

**content = infile.read().replace('\n',', ').split(', ')**

**content = content[:-1]**

**#put content into a list**

**lst = []**

**ans = []**

**for i in range(len(content)):**

**if i > 0 and (i+1) % 8 == 0:**

**ans.append(content[i])**

**lst.append(ans)**

**ans = []**

**else:**

**ans.append(content[i])**

**#go through each row and determine if the cname matches but credits don't**

**res = []**

**for i in range(len(lst)):**

**for j in range(i,len(lst)):**

**if lst[i][0] == lst[j][0] and lst[i][-1] != lst[j][-1]:**

**res.append([i,j,lst[i][0]])**

**#return result check**

**for i in res:**

**print(f'Lines {i[0]} and {i[1]} have a cname => credits FD error for course "{i[2]}"')**

1. Suppose I have a new query: Qe: For every department, display the average graduation year.
   1. Use the view from Part 4-a to re-write query Qe (i.e., replace the tables in the query’s FROM clause by the view and rewrite the rest of the query accordingly to produce an answer).

**SQL Code:**

**--SQL query to grab average grad year from view table**

**select department, avg(gradyear) as AVG\_STUDENT\_GRAD\_YEAR from summary**

**group by department;**

**A picture containing table

Description automatically generated**

* 1. Use your .txt file containing de-normalized data from part-b to answer Qe with python instead of SQL. Note that this solution should not use SQLite or SQL, just python.

**PYTHON CODE: (also in .py file)**

**def avgGradYear():**

**'''finds the average grad year by department in summary table'''**

**#open text file, store content**

**infile = open('summary.txt','r')**

**content = infile.read().replace('\n',', ').split(', ')**

**content = content[:-1]**

**#put content into a list**

**lst = []**

**ans = []**

**for i in range(len(content)):**

**if i > 0 and (i+1) % 8 == 0:**

**ans.append(content[i])**

**lst.append(ans)**

**ans = []**

**else:**

**ans.append(content[i])**

**#store averages by department in dictionary**

**res = {}**

**for row in lst:**

**if row[-2] in res:**

**res[row[-2]].append(int(row[-3]))**

**else:**

**res[row[-2]] = [int(row[-3])]**

**#print statement with averages**

**for key in res.keys():**

**print(key, sum(res[key]) / len(res[key]))**

**Text

Description automatically generated with medium confidence\*\*Python screenshot different due to the .txt file having an additional entry\*\***