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, all of the requirements for Second Normal Form are also included in the requirements are for Third Normal Form.

b. A foreign key can contain a NULL.

True, depending on the constraints of the table.

c. The union rule allows to combine functional dependencies A→CD and B→CD into a functional dependency AB→CD.

False, the union rule tells us that if A -> C and A -> D then A -> CD. This is not an example of the union rule.

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

False, you can sort the way that query results are delivered in. Depending on which columns are selected, the query will return results in varying orders, but the primary key is not the default sorting column.

e. A table in a relational database is equivalent of a set of tuples.

True

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

True, NULL is the absence of a value, so it is not necessarily equivalent to other NULL values.

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 99.99 (column definition should enforce **both** the number precision and greater-than-0 requirement).

PurchaseAmt NUMBER(4,2) CHECK(PurchaseAmt > 0)

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 10 characters and has to start from 'rue' (hint: CHECK constraint can use LIKE operator).

Street VARCHAR2(10) 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 within the first 4 years of the program. You can assume that GradYear is an integer, but your query is not allowed to use a specific assumed year.

```
SELECT Student.StudentID, Student.Name, Department
FROM Student
INNER JOIN Grade
ON Student.StudentID = Grade.StudentID
LEFT OUTER JOIN Course
ON Grade.CName = Course.CName
GROUP BY Department
HAVING 4 >= (GradYear - MIN(GradYear));
```

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.

```
SELECT Name, Cname
From Student, Grade
Where Student.StudentID = Grade.StudentID and Name LIKE '%Muriel%';
```

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.

```
SELECT Name, GradYear
FROM Student
WHERE(Select COUNT(*)
    FROM Grade
    WHERE StudentID = StudentID) < 2;
```

Q4. Update all student records, to increase the graduation year by 1 for all students who live in Chicago

```
UPDATE Student
Set GradYear = GradYear + 1
WHERE Address LIKE '%Chicago%';
```

Q5. Modify the course table to add a Chair column that can be up to 25 characters (that question requires a DDL rather than a DML SQL statement)

```
ALTER TABLE Course
ADD Chair VARCHAR2(25);
```

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.

X	Y	Z	W	A
1	1	1000	Dog	2

1	2	5000	Cat	3
1	3	3000	Elephant	4
2	1	1000	**Cat**	3
2	2	2000	**Elephant**	4
2	3	5000	**NULL**	5

For the missing values in the table, since A leads to W, I followed the previous patterns, where the value of A led to the value of W. In this case, if A has a value of 2, then W would have a value of 'Dog', if A has a value of 3 then W has a value of 'Cat' and if A has a value of 4 then W would have a value of 'Elephant'. Since there is no pattern present for A having a value of 5, I placed a NULL in the column of W for the final row.

- 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?

Yes, through the decomposition rule, if XY determine ZA, that means that XY determines Z and XY determines A. If XY determines A and A determines W, then XY determines W through transitivity. So, in conclusion, XY does determine W through decomposition and transitivity.

- Suppose that you were also given relation S:
  S(P, Q, U, M, N)
  What functional dependencies (if any) can you assume?

Since it appears that P,U and N are all primary keys, there is more than one possibility. But one possibility is PUN -> QM. Since we must be able to identify all of the items given the primary keys.

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:

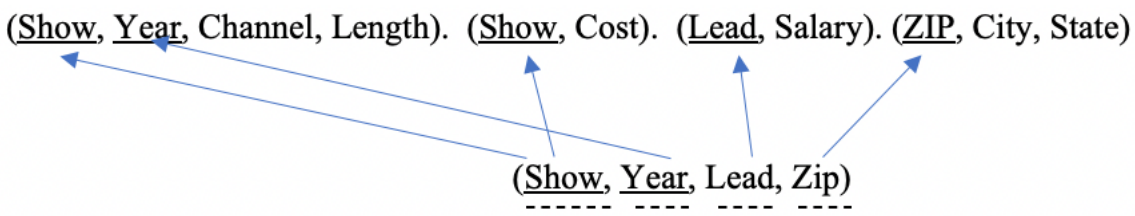
Show → Cost

Zip → City, State

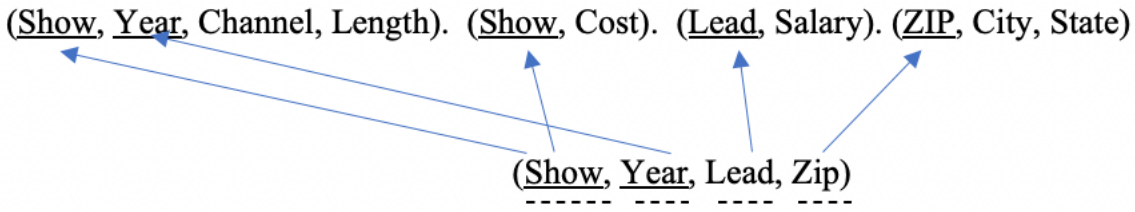
Lead → Salary

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.



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



c. (7 points)

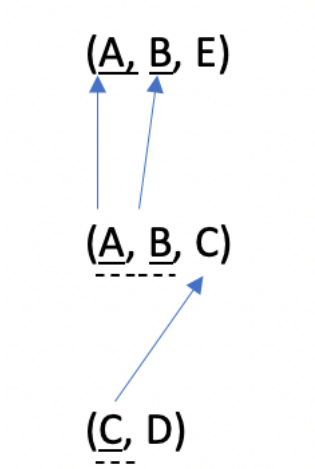
Given the schema R and the following functional dependencies:

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)

We can tell that AB determines C and D through transitivity. There is no information regarding how E is determined. Since AB determine C and D, I would make A and B composite primary keys.

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



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 9 enrollments (at least one of the students should not be enrolled in any courses and at least one course should have zero current enrollments).

```
import sqlite3

conn = sqlite3.connect('dsc450_Midterm1.db') # open the connection
cursor = conn.cursor()

createTbl1 = """CREATE TABLE Student
(
    StudentID  NUMBER(4) Primary key,
    Name VARCHAR2(15),
    Address VARCHAR2(45),
    GradYear NUMBER(4))

"""

createTbl2 = """CREATE TABLE Course
```

```
(
    CNAME VARCHAR2 (35) Primary Key,
    Department VARCHAR2 (25),
    Credits NUMBER(4))

"""

createTbl3 = """CREATE TABLE Grade
(
    CName VARCHAR2(35),
    StudentID NUMBER(4),
    CGrade NUMBER(3,2),

    PRIMARY KEY(CName, StudentID),

    FOREIGN KEY (StudentID)
      REFERENCES Student_Midterm(StudentID),

   FOREIGN KEY (CName)
      REFERENCES Course_Midterm(CName))

"""

cursor.execute(createTbl1)
cursor.execute(createTbl2)
cursor.execute(createTbl3)

inserts1 = ["INSERT INTO Student VALUES(1, 'Dan Obrien', '123 North Ave Chicago IL
60622', 2022);", "INSERT INTO Student VALUES(2, 'Sarah Obrien', '124 North Ave Chicago
IL 60622', 2021);", "INSERT INTO Student VALUES(3, 'Jack Black', '12 Division Ave Chicago
IL 60622', 2024);", "INSERT INTO Student VALUES(4, 'Jack White', '321 California Ave
Chicago IL 60622', 2022);"]
inserts2 = ["INSERT INTO Course VALUES('Intro to Data Science', 'Data Science', 4);",
"INSERT INTO Course VALUES('Intro to Machine Learning', 'Computer Science', 4);",
"INSERT INTO Course VALUES('Python Programming', 'Computer Science', 4);", "INSERT
INTO Course VALUES('Data Visualization', 'Data Science', 4);"]
inserts3 = ["INSERT INTO Grade VALUES('Intro to Data Science', 1, 3.2);", "INSERT INTO
Grade VALUES('Intro to Data Science', 2, 3.8);", "INSERT INTO Grade VALUES('Intro to
Data Science', 3, 4);", "INSERT INTO Grade VALUES('Intro to Machine Learning', 1, 3.5);",
"INSERT INTO Grade VALUES('Intro to Machine Learning', 2, 4);", "INSERT INTO Grade
```

```
VALUES('Intro to Machine Learning', 3, 3);", "INSERT INTO Grade VALUES('Python
Programming', 1, 4);", "INSERT INTO Grade VALUES('Python Programming', 2, 3.7);",
"INSERT INTO Grade VALUES('Python Programming', 3, 4);"]

for ins in inserts1:
   cursor.execute(ins)

for ins in inserts2:
   cursor.execute(ins)

for ins in inserts3:
   cursor.execute(ins)
```

a) Create a view that pre-joins the three tables, including all of the records from student and course tables

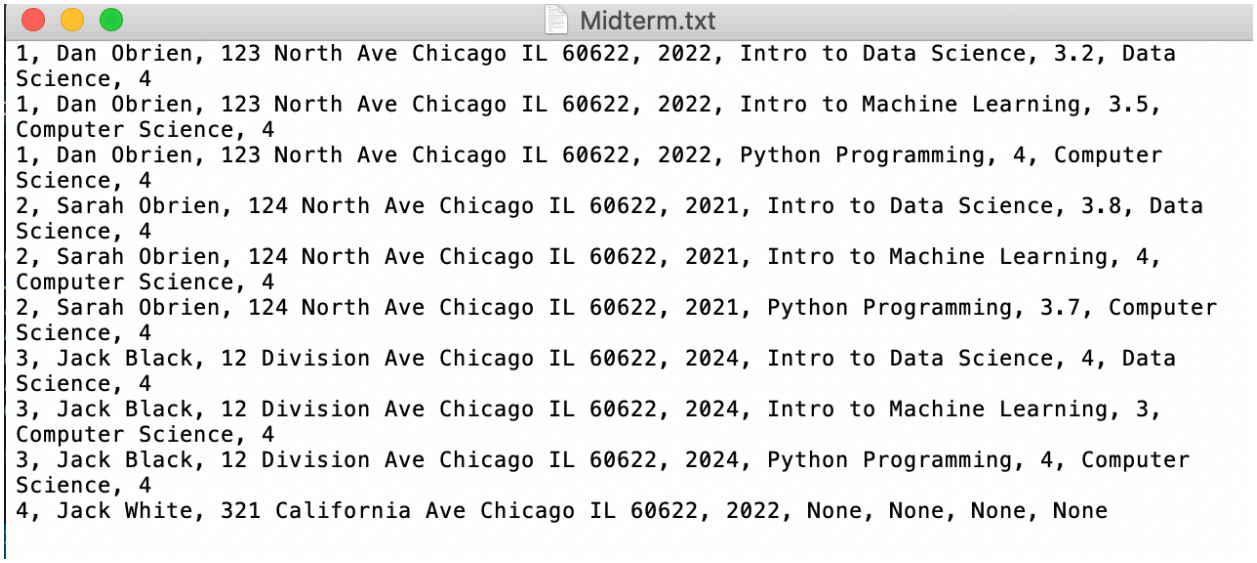
```
CreateView = """CREATE VIEW AllValues AS
SELECT Student.StudentID, Name, Address, GradYear, Course.CName, CGrade, Department,
Credits
FROM Student
LEFT OUTER Join Grade
ON Student.StudentID = Grade.StudentID
LEFT OUTER JOIN Course
ON Course.CName = Grade.CName;

"""
cursor.execute(CreateView)
```

b) 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.

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

```
x = cursor.execute("Select * FROM AllValues")
outputFile = open("Midterm.txt", 'w')
for items in x:
   items = ', '.join(map(str, items))
   outputFile.write('%s\n' % items)
conn.commit()
```

outputFile.close()

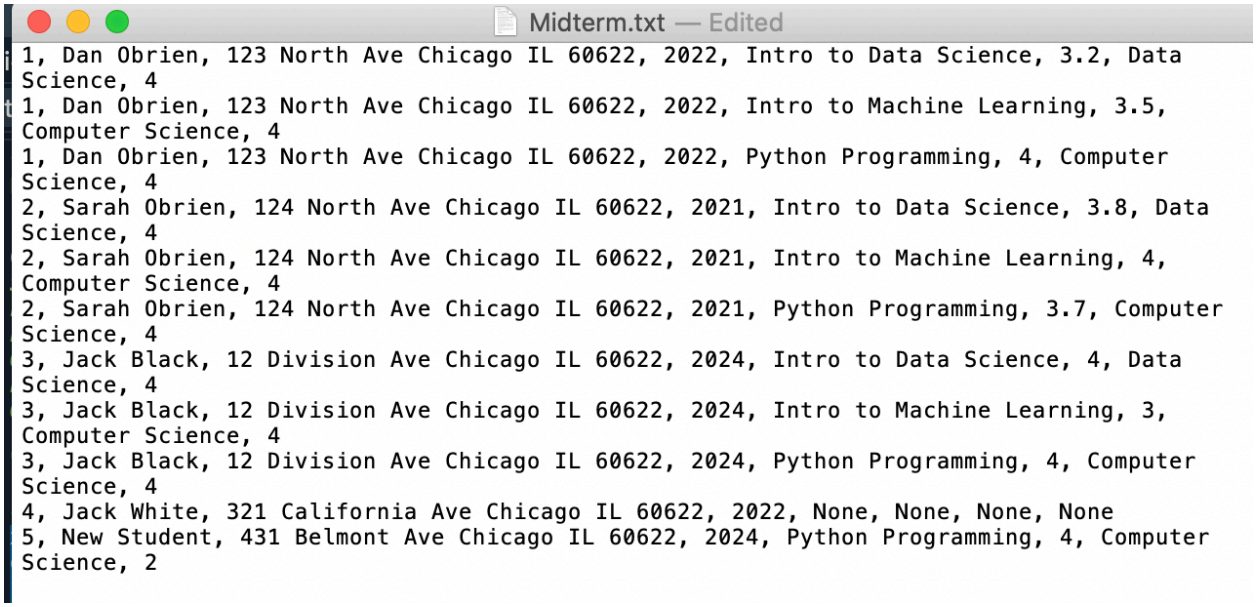


c) Add a new row to the de-normalized .txt file (you can manually edit the .txt file from part b) that violates the following functional dependency:

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)

The last entry is the new entry. New student is enrolled in Python Programming, but the course is given 2 credit hours instead of 4 credit hours.



d) 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.

```
f = open('Midterm.txt', 'r')
lines = f.readlines()
lines = ([s.replace('\n', '') for s in lines])
print(lines)

newList = []
for line in lines:
    line = line.split(',')
    newList.append(line)
print(newList)

newerList = []
for lists in newList:
    lists = [x.strip(' ') for x in lists]
    newerList.append(lists)
print(newerList)

keys = []
values = []
for lists in newerList:
    keys.append(lists[4])
    values.append(lists[7])

from collections import defaultdict
myDict = defaultdict(set)
for c, i in zip(keys, values):
    myDict[c].add(i)


problemList = []
for key in myDict:
    value = myDict[key]
    if len(value) > 1:
```

```
    problemList.append(key)

print('Functional dependency violated with the following course(s):{}'.format(problemList))
```

e) Suppose I have a new query: For every department, display the average graduation year.

1. Use the view from Part 4-a to re-write query Q1 (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).

```
queryE1 = """
SELECT Department, AVG(GradYear)
FROM ALLValues
Group By Department;
"""
cursor.execute(queryE1)
print(cursor.fetchall())
```

2. Use your .txt file containing de-normalized data from part-b to answer Q1 with python instead of SQL. Note that this solution should not use SQLite or SQL, just python.

```
f = open('Midterm.txt', 'r')
lines = f.readlines()
lines = ([s.replace('\n', '') for s in lines])
print(lines)

newList = []
for line in lines:
   line = line.split(',')
   newList.append(line)
print(newList)

newerList = []
for lists in newList:
   lists = [x.strip(' ') for x in lists]
   newerList.append(lists)
print(newerList)
```

```
DataScience = []
ComputerScience = []
Other = []
for line in newerList:
    if line[6] == 'Data Science':
        DataScience.append(line[3])
    if line[6] == 'Computer Science':
        ComputerScience.append(line[3])
    else:
        Other.append(line[3])

DataScience = [int(i) for i in DataScience]
ComputerScience = [int(i) for i in ComputerScience]
Other = [int(i) for i in Other]

AvgYearDS = sum(DataScience)/len(DataScience)
AvgYearCS = sum(ComputerScience/len(ComputerScience))
AvgYearOther = sum(Other)/len(Other)
```