CSE 180, Final Exam, Winter 2020, Shel Finkelstein

Student Name: ______________________________________

Student ID: ______________________________________

UCSC Email: ______________________________________

Final Points

Part	Max Points	Points
I	40	
II	24	
III	36	
Total	100	

The first Section (Part I) of the Winter 2020 CSE 180 Final is multiple choice and is double-sided. Answer all multiple choice questions on your Scantron sheet. You do not have to hand in the first Section of the Exam, but you must hand in the Scantron sheet, with your Name and Student ID filled in (including marking bubbles below) on that Scantron sheet. Please be sure to use a #2 pencil to mark your choices on that Section of the Final.

This separate second Section (Parts II and III) of the Final is not multiple choice and is single-sided, so that you have extra space to write your answers. If you use that extra space, please be sure to write the number of the problem that you're solving next to your answer. Please write your Name, Email and Student ID on this second Section of the Exam, which you must hand in. You may use any writing implement on this Section of the Exam.

At the end of the Final, please be sure to hand in both your Scantron sheet for the first Section of the Exam and also **this second Section of the Exam**. You must also show your UCSC id when you hand them in.

Part II: (24 points, 6 points each)

Question 22: Here are statements creating two tables R and S:

```
CREATE TABLE R (
   a INT,
   b INT,
   PRIMARY KEY(a),
   FOREIGN KEY(b)
          REFERENCES S
             ON DELETE CASCADE,
             ON UPDATE CASCADE
    );
```

```
CREATE TABLE S (
   b INT,
   c INT,
   PRIMARY KEY(b)
    );
```

Table R(a,b) currently contains the four tuples, (0,4), (1,5), (2,4), and (3,5).
Table S(b,c) currently contains the four tuples, (2,10), (3,11), (4,12), and (5,14).

Indicate all changes that happen to both R and S when the following SQL statements are executed upon the R and S instances that are shown above. That is, you should ignore changes made by earlier parts of this question when you answer later parts. If a statement changes existing tuples, show both old and new values of those tuples.

22a): UPDATE S SET b = 9 WHERE c = 14;

Answer 22a):
Changes made to the original R: (1,5) → (1,9), (3,5) → (3,9)

Changes made to the original S: (5,14) → (9,14)

22b): UPDATE R set b = 7 WHERE a = 0;

Answer 22b):
Changes made to the original R: No changes

Changes made to the original S: No changes (Referential Integrity violation)
(You didn't need to give explanation.)

22c): DELETE FROM S WHERE b = 5;

Answer 22c):
Changes made to the original R: (1,5) and (3,5) are deleted

Changes made to the original S: (5,14) is deleted

Question 23: For the relation Employees(name, age, salary), write a Serializable SQL transaction that executes the following three SQL statements and then commits.

1. The transaction should insert an employee whose salary is 9000, whose age is 21, and whose name is John Smith.
2. After that, the transaction should update the salary of every employee that has the lowest salary value, adding 1000 to their salary.
3. Then the transaction should delete all employees who make more than the average salary and whose age is more than 70.

Answer 23:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

INSERT INTO Employees
        VALUES ('John Smith', 21, 9000);

UPDATE Employees
SET salary = salary + 1000
WHERE salary = ( SELECT MIN(salary)
                 FROM Employees );

DELETE FROM Employees
WHERE salary > ( SELECT AVG(salary)
                 FROM Employees )
   AND age > 70;

COMMIT TRANSACTION;
```

The UPDATE can also be written in other ways such as:

```
UPDATE Employees
SET salary = salary + 1000
WHERE salary <= ALL ( SELECT salary)
                      FROM Employees );

UPDATE Employees
SET salary = salary + 1000
WHERE NOT EXISTS ( SELECT *
                   FROM Employees e2
                   WHERE e2.salary < salary );
```

Question 24: Codd's relational algebra for sets included only 5 operators (σ , π , x , U , and -).

Given the following relations:

Sailors(sid, sname, rating, age) // sailor id, sailor name, rating, age
Boats(bid, bname, color) // boat id, boat name, color of boat
Reserves(sid, bid, day) // sailor id, boat id, date that sailor sid
// made a reservation for boat bid

Write the following query using Relational Algebra (not SQL).

Find the names of sailors who reserved a red boat or reserved a green boat (or both).

To simplify notation, you may write SIGMA for σ and PI for π. You may also use <-- for Assignment and RHO for ρ (Rename). Also, although you may use subscripts, you may also use square brackets, for example writing:

PI[Sailors.name] (SIGMA[Sailors.age > 18] (Sailors)) instead of

$\pi_{\text{Sailors.name}} (\sigma_{\text{Sailors.age > 18}} (\text{Sailors}))$

PI[Sailors.name] (SIGMA[Sailors.sid =Reserves.sid AND Reserves.bid = Boats.bid
AND (BOAT.color = 'red' OR BOAT.color = 'green')]
(Sailors X Reserves X Boats))

or

$\pi_{\text{Sailors.name}}$
$(\sigma_{\text{Sailors.sid =Reserves.sid AND Reserves.bid = Boats.bid AND (BOAT.color = 'red' OR BOAT.color = 'green')}}$
$(\text{Sailors X Reserves X Boats}))$

Question 25: Suppose that you have a relation:

Products(ProductID, ProductName, Component, Quantity)

that has the following non-trivial Functional Dependencies:

ProductID, Component → Quantity
ProductName, Component → Quantity
ProductName → ProductID
ProductID → ProductName

25a) Is the Products relation in Boyce-Codd Normal Form? Give a clear detailed proof of your answer.

Answer 25a):

No, the Products relation is not in BCNF. For Products to be in BCNF, every Functional Dependency (FD) has to either a) be trivial, or b) have a left-hand side that's a superkey.

The FD ProductName → ProductID is non-trivial, since the right-hand side is not a subset of the left-hand side.

The left-hand side of that FD is not a superkey. Why?

If we compute ProductName$_+$ using the Attribute Closure Algorithm, then we start with {ProductName}, then get {ProductName, ProductID} using the third FD, and then stop, since there are no other FDs that we can apply to get additional attributes. Hence the Products relation in not in BCNF.

25b) Is the Products relation in Third Normal Form? Give a clear detailed proof of your answer.

Answer 25b):

Yes, the Products relation is in 3NF. For Products to be in 3NF, every Functional Dependency (FD) has to either a) be trivial, or b) have a left-hand side that's a superkey or c) have a right-hand side that's part of a key.

The trivial FDs on Products are trivial. But the 4 FDs specified above are not trivial. Let's look at them one by one to see if they all satisfy b) or c).

1. For ProductID, Component → Quantity, the left-hand side is a superkey. {ProductID, Component}$^+$ includes {ProductID, Component}, then {ProductID, Component, Quantity, ProductName} using the first and fourth FDs, and that's all the attributes of Product, so the left-hand side is a superkey.

2. For ProductName, Component → Quantity, the left-hand side is a superkey. {ProductName, Component}$^+$ includes {ProductName, Component}, then {ProductName, Component, Quantity, ProductID} using the second and third FDs, and that's all the attributes of Product, so the left-hand side is a superkey.

3. For ProductName → ProductID, the right-hand side is part of a key, namely {ProductID, Component}. Why? We've already seen (in #1) that {ProductID, Component}$^+$ equals all the attributes of Products, so {ProductID, Component} is a superkey for Products.

But if we "put that superkey on a diet" by getting rid of either attribute, it's no longer a superkey: ProductID$^+$ is {ProductID, ProductName}, and Component$^+$ is {Component}. Hence {ProductID, Component} is a key for Products.

4. For ProductID → ProductName, , the right-hand side is part of a key, namely {ProductName, Component}. Why? We've already seen (in #2) that {ProductName, Component}$^+$ equals all the attributes of Products, so {ProductName, Component} is a superkey for Products.

But if we "put that superkey on a diet" by getting rid of either attribute, it's no longer a superkey: ProductName$^+$ is {ProductName, ProductID}, and Component$^+$ is {Component}. Hence {ProductName, Component} is a key for Products.

Part III: (36 points, 9 points each)

Some familiar tables appear below, with Primary Keys underlined. **These tables also appear on the last page of the Final, which you can tear off to help you do questions in Part III of the Final.** If you tear off the page, you don't have to turn it at the end of the Exam.

Movies(movieID, name, year, rating, length, totalEarned)

Theaters(theaterID, address, numSeats)

TheaterSeats(theaterID, seatNum, brokenSeat)

Showings(theaterID, showingDate, startTime, movieID, priceCode)

Customers(customerID, name, address, joinDate, status)

Tickets(theaterID, seatNum, showingDate, startTime, customerID, ticketPrice)

Assume that no attributes can be NULL, and that there are no UNIQUE constraints.

You may assume Referential Integrity constraints as follows:
- Each theaterID in TheaterSeats appears as a Primary Key in Theaters.
- Each theaterID in Showings appears as a Primary Key in Theaters.
- Each movieID in Showings appears as a Primary Key in Movies.
- Each customerID in Tickets appears as a Primary Key in Customers.
- Each (theaterID, seatNum) in Tickets appears as a Primary Key in TheaterSeats.
- Each (theaterID, showingDate, startTime) in Tickets appears as a Primary Key in Showings.

Write legal SQL queries for Questions 26-29. If you want to create and then use views to answer these questions, that's okay, but views are not required unless the question asks for them.

Don't use DISTINCT in your queries unless it's necessary, 1 point will be deducted if you use DISTINCT when you don't have to do so. (Of course, points will also be deducted if DISTINCT is needed and you don't use it.) And some points may be deducted for queries that are very complicated, even if they are correct.

Question 26: In the TheaterSeats table, brokenSeat is true if a theater seat is broken; otherwise it's false.

Write a SQL query that finds the theaterID and seatNum for all theater seats that aren't broken, and for which no tickets have been sold. Result tuples which have a larger theaterID value should appear before result tuple that have a smaller theaterID value. If two result tuples have the same theaterID value, tuples with a smaller seatNum should appear before tuples with a larger seatNum.

No duplicates should appear in your result.

Answer 26:

```
SELECT Th.theaterID, Th.seatNum
FROM TheaterSeats Th
WHERE NOT Th.brokenSeat
    AND NOT EXISTS ( SELECT *
                     FROM Tickets T
                     WHERE T.theaterID = Th.theaterID
                         AND T.seatNum = Th.seatNum )
ORDER BY Th.theaterID DESC, Th.seatNum;
```

DISTINCT is not needed because (theaterID, seatNum) is the Primary Key of TheaterSeats, so each theater seat can appear in the result of this query at most once

Instead of "NOT Th.brokenSeat", you could write "Th.brokenSeat = FALSE" or "Th.brokenSeat IS FALSE".

Subquery could have any attributes in SELECT clause; doesn't have to be SELECT *.

There's no reason to mention Showings table in this query.

Could write "ORDER BY Th.theaterID DESC, Th.seatNum ASC", although the ASC is not needed, since that's the default.

This query can also be written using "NOT IN" (or "<> ANY"), e.g.:

```
SELECT Th.theaterID, Th.seatNum
FROM TheaterSeats Th
WHERE NOT Th.brokenSeat
    AND Th.seatNum NOT IN ( SELECT T.seatNum
                            FROM Tickets T
                            WHERE T.theaterID = Th.theaterID )
ORDER BY Th.theaterID DESC, Th.seatNum;
```

Question 27: Find the customerID, name and address of each customer whose address has the string 'Cruz' (with that exact capitalization) appearing anywhere in it, and who bought tickets to at least 8 different movies.

Your result should have attributes cID, cName and cAddress. No duplicates should appear in your result.

Answer 27:

```
SELECT C.customerID AS cID, C.name AS cName, C.address AS caddress
FROM Customers C, Tickets T, Showings S
WHERE C.customerID = T.customerID
    AND C.name LIKE '% Cruz %'
    AND T.theaterID = S.theaterID
    AND T.showingDate = S.showingDate
    AND T.startTime = S.startTime
GROUP BY C.customerID, C.name, C.address
HAVING COUNT(DISTINCT S.movieID) >= 8;
```

DISTINCT is not needed because there will be only one tuple for the GROUP BY attributes.

The query would also be correct if it just had GROUP BY C.CustomerID. That's because C.CustomerID is the Primary Key of Customers, so the name and address for a particular customer is determined by the CustomerID. And DISTINCT is still not needed because there can only be one tuple in the result with a particular CustomerID, since we GROUP BY CustomerID.

Question 28: Showings has an attribute startTime, the time at which a showing starts. The endTime (that is, the time at which a showing ends) is its startTime plus the length of that showing's movie. Two showings S1 and S2 **conflict** if they are in the same theater on the same showingDate, S1 starts before S2 starts, but S2 starts before S1 ends.

For each pair of showings S1 and S2 that conflict, your SQL query should output theaterID, showingDate, the startTime of S1, the endTime of S1, and the startTime of S2. The attributes in your result should appear as theaterID, showingDate, firstStart, firstEnd and secondStart.

No duplicates should appear in your result.

Answer 28:

```
SELECT S1.theaterID, S1.showingDate, S1.startTime as firstStart,
                    S1.startTime + M.length AS firstEnd, S2.startTime AS secondStart
FROM Movies M, Showings S1, Showings S,
WHERE M.movieID = S1. movieID
    AND S1.theaterID = S2.theatherID
    AND S1.showingDate = S2.showingDate
    AND S1.startTime < S2.startTime
    AND S2.startTime < S1.startTime + M.length;
```

Strictly speaking, M.length should be converted to an INTERVAL, but we haven't gone over methods to do that, so we'll be sloppy and just add a length to a time.

DISTINCT is <u>not needed</u> because there can't be more than one Showing S1 in a particular theater on a particular date at a particular start time, and there can't more than one conflicting showing S2 in the same theater on the same date at a particular second start time.

Question 29: This question has two parts; be sure to answer both.

29a) Define a view SingleTheaterMovies that finds the movies that had showings in exactly one theater.

The attributes of the view should be movieID and theaterID. No duplicates should appear in the SingleTheaterMovies view.

29b) Using the SingleTheaterMovies view, define another view, GreatSingleTheaterMovies. A GreatSingleTheaterMovies is a movie that had showings in exactly one theater, is rated 'G', and earned more than 2000.00. GreatSingleTheaterMovies should have attributes movieID, name and totalEarnings.

Be sure to use the SingleTheaterMovies view to do this; you may also use the original tables if necessary. No duplicates should appear in the GreatSingleTheaterMovies view.

Answer 29a): Use of GROUP BY using movieID fails, because theaterID needs to appear in the SELECT clause.

Use of GROUP BY using both movieID and theaterID fails, because we need to determine movies appearing in only one theater.

Here are some correct solutions, all of which need DISTINCT.

```
CREATE VIEW SingleTheaterMoviesAS
        SELECT DISTINCT S.movieID, S.theaterID
        FROM Showings S,
        WHERE  S.theaterID = ALL ( SELECT S2.theaterID
                                   FROM Showings S2
                                   WHERE S.movieID = S2.movieID );


CREATE VIEW SingleTheaterMoviesAS
        SELECT DISTINCT S.movieID, S.theaterID
        FROM Showings S
        WHERE NOT EXISTS
              ( SELECT *
                FROM Showings S2
                WHERE S.movieID = S2.movieID
                   AND S.theaterID <> S2.theaterID );
```

```
CREATE VIEW SingleTheaterMoviesAS
        SELECT DISTINCT S.movieID, S.theaterID
        FROM Showings S
        WHERE  ( SELECT COUNT( DISTINCT S2.theaterID )
                  FROM Showings S2
                  WHERE S2.movieID = S.movieID ) = 1;


CREATE VIEW SingleTheaterMoviesAS
        SELECT DISTINCT S.movieID, S.theaterID
        FROM Showings S
        WHERE S.movieID IN ( SELECT S2.movieID
                             FROM Showings S2
                             GROUP BY S2.movieID
                             HAVING COUNT(DISTINCT S2.theaterID ) = 1 );
```

Answer 29b):

```
CREATE VIEW GreatSingleTheaterMovies AS
        SELECT M.movieID, M.name, M.totalEarned AS totalEarnings
        FROM Movies M, SingleTheaterMovies STM
        WHERE M.movieID = STM.movieID
            AND M.rating = 'G'
            AND M.totalEarned > 2000.00;
```

This question asked about Total Earnings, not the Computed Earnings that was in Lab3.

DISTINCT is not needed, since each movieID appears only once in Movies and at most once in SingleTheaterMovies, and a MovieID appears in the result of this view only if those values match.