CMPS 182, Final Exam, Winter 2016, Shel Finkelstein

Student Name: ______________________________

Student ID: ______________________________

UCSC Email: ______________________________

Final Points

Part	Max Points	Points
I	25	
II	24	
III	15	
IV	36	
Total	100	

Part I: (25 points, 5 each):

For the questions in Part I, assume that there are tables with the following schemas, and that attributes except for grade can't be NULL. Underlined attributes are keys. For each question in this part, write a SQL statement that does what's requested. (You may use views that you define, if you wish.)

Student(studentID, student_name, major)
For each studentID, gives the students name and major.

Course(courseID, course_name, department, instructor)
For each courseID, gives the course's name, department and instructor.

Enroll(studentID, courseID, grade)
For each studentID and courseID where the student has taken (or is taking) that course, gives the student's grade. Grade is a float number if the student has taken the class, and is NULL if the student is still taking the class.

Question 1: Find the instructors for courses that are in either the 'BIO' or the 'CHEM' department. List just the instructors, but don't list any instructor more than once.

Answer 1:

```
SELECT DISTINCT instructor
FROM Course
WHERE.department = 'BIO' OR department = 'Chem';
```

There are many ways to write SQL queries in Part I. Answer provided gives just one of them. In some cases, answering using Views may be the easiest, but using Views is not required, except for Question 19 in Part IV, which requests use of View.

Question 2: Find the names of courses and instructors for which at least one student's grade is NULL. Don't include duplicates in your answer.

Answer 2:

```
SELECT DISTINCT course_name, instructor
FROM Course
WHERE EXISTS (
        SELECT *
        FROM Enroll
        WHERE Course.courseID = Enroll.courseID
          AND grade IS NULL );
```

Question 3: For each 'CMPS' department course that has at least 50 students enrolled, list the ID for that course and the number of students in that course.

Answer 3:

```
SELECT c.courseID, COUNT(e.studentID)
FROM Course c, Enroll e
WHERE c.courseID = e.courseID
  AND c.department = 'CMPS'
GROUP BY c.courseID
HAVING COUNT(e.studentID) >= 50;
```

SELECT DISTINCT is not needed, since courseID is the key for Course. It's extraneous, but not wrong, so no credit off.

Question 4: Find the ID and name for courses where the maximum grade in that course is more than twice the average grade in that course.

Answer 4:

```
SELECT c.courseID, c.course_name
FROM Course c
WHERE (    (  SELECT MAX(e1.grade)
              FROM Enroll e1
              WHERE c.courseID = e1.courseID )
        > 2 * (  SELECT AVG(e2.grade)
              FROM Enroll e2
              WHERE c.courseID = e2.courseID )    );
```

Could use e (or Enroll) in both cases, instead of using e1 and e2 as tuple variables. I've put in extra parentheses for clarity around the SELECT MAX and SELECT AVG, although they're not necessary.

A shorter and perhaps simpler alternative using GROUP BY and HAVING:

```
SELECT c.courseID, c.course_name
FROM Course C, Enroll e
WHERE c.courseID = e.courseID
GROUP BY c.courseID, c.course_name
HAVING MAX(e.grade) > 2 * AVG(e.grade);
```

Note that both courseID and course_name are needed in the GROUP BY, since they both appear in the SELECT.

Again, SELECT DISTINCT is not needed in either version of the answer, although it's not wrong. That's obvious for the first version of the answer, since courseID is the key for Course, but can you see why it's not needed for the answer using HAVING?

Question 5: Write a transaction that changes enrollment for the students who are enrolled in course 'CMPS 182' but who are not enrolled in 'CMPS 17', so that those students also are enrolled in 'CMPS 17', with a grade that's NULL.

For this question, assume that course name is unique.

It's okay to use programming language together with SQL to answer this question, e.g., using stored procedures, host language with SQL embedded, or connection tools/libraries such as JDBC. If you use JDBC, you can assume that connection already exists.

Answer 5:

Don't need to create View to answer this, but it's easier to read with a view.

```
CREATE View EnrolledStudents AS
        SELECT studentID, courseID
        FROM Enroll e, Course c
        WHERE e1.courseID = c.courseID;

BEGIN TRANSACTION;          // BEGIN isn't needed; transactions begin implicitly

INSERT INTO Enroll (studentID, courseID )
        SELECT s.studentID,
                ( SELECT courseID FROM Course WHERE course_name = 'CMPS 17' )
        FROM Student s
        WHERE  studentID IN
           (    SELECT es1.studentID
                FROM EnrolledStudents e1
                WHERE es1.courseID = 'CMPS 182     )
        EXCEPT
           (    SELECT es2.studentID
                FROM EnrolledStudents es2
                WHERE es2.courseID = 'CMPS 17     );

END TRANSACTION;
```

Note that grade will be NULL for tuples inserted into Enroll, which problem requests.

You could also answer this by using ability to embed SQL in a programming language, e.g., by getting the courseID for 'CMPS17' and putting it into a variable, and then using that variable in an EXEC SQL or JDBC statement.

This is a hard question, and will be graded somewhat more generously than other questions. As noted above, people who use JDBC can assume that connection already exists.

Part II: (24 Points, 4 each)

Question 6: If R(A,B) is a relation where A's domain is (a1, a2, a3) and A can also be NULL, and B's domain is (b1, b2, b3, b4, b5) but B cannot be NULL, what is the maximum number of different tuples that can be in R?

Answer 6: ______20______

*A can have 4 values (including NULL), and B can have 5 values (since it can't be NULL), so there can be 4 * 5 = 20 different tuples in R.*

Question 7: A database has the relation Employees, with primary key emp_id, and other attributes giving name, department and salary of an employee.

Employees(<u>emp_id</u>, ename, edept, esalary)

A programmer writes a SQL query intending to find the name, salary and department for employees who make the most money in their departments.

```
SELECT e1.ename, e1.salary, e1.edept
FROM Employees e1
WHERE e1.salary >=
          ( SELECT e2.salary
            FROM Employees e2
            WHERE e1.edept = e2.edept );
```

7a): What's wrong with writing the query this way?

Answer 7a):

The subquery beginning SELECT e2.salary will return more than one value if there's more than one employee in a department, so this statement will result in a runtime error when executed.

7b): How would you fix the query so that it does what the programmer intended?

Answer 7b):

One simple way to fix the query to achieve intended result is to insert the word **ALL**, so the query reads:

```
SELECT e1.ename, e1.salary, e1.edept
FROM Employees e1
WHERE e1.salary >= ALL
          ( SELECT e2.salary
            FROM Employees e2
            WHERE e1.edept = e2.edept );
```

Could also fix the query by selecting employees whose salary was greater than or equal to the MAX of the salaries for people in the same department.

```
SELECT e1.ename, e1.salary, e1.edept
FROM Employees e1
WHERE e1.salary >=
          ( SELECT MAX(e2.salary)
            FROM Employees e2
            WHERE e1.edept = e2.edept );
```

Question 8: What does the Serializability isolation level mean for transactions, and what is the benefit of Serializability?

Answer 8:

The Serializability isolation level for transactions means that although multiple transactions may be executing at the same time in a database system, the results of executing a collection of transactions is "as if" they had been executed one at a time.

A database is "consistent" if the data in the database obeys all required business rules, such as a requirement that the total money in Checking + Savings accounts for a person must not go below zero. Suppose that each transaction maintains consistency, meaning that if the database started out consistent before the transaction, then after executing that transaction by itself, the database will still be consistent. Then after executing a collection of transactions with Serializability isolation, a database that started off consistent will still be consistent.

Why? Because the database started off consistent, each transaction maintains consistency, and Serializability means that its "as if" transactions were executed one at a time.

Question 9: Assume that relation R(A,B,C,D,E) includes the row (5,4,3,2,1), and that R has Functional Dependencies A,B → C and C,D → E,

Mark each row below with a check mark if it might also be in R, and mark each row with an X if it can not also be in R.

Answer 9:

__✔__	(5,4,3,3,1)	
__✖__	(5,4,2,2,1)	because A,B → C, so 5,4 determines that C must be 3
__✔__	(1,2,3,4,5)	
__✖___	(6,5,3,2,3)	because C,D → E, so 3,2 determines that E must be 1

Question 10: The following tables should be familiar.

Doctors (doctor_id, name, location, specialty)
Medicines (medicine_id, name, price)
Patients (patient_id, name, address, email, doctor_id, admitted)
Prescriptions (prescription_id, doctor_id, medicine_id, patient_id, prescription_date)

The Prescriptions table could have been created by the statement:

```
CREATE TABLE Prescriptions (
    prescription_id INTEGER PRIMARY KEY,
    doctor_id INTEGER,
    medicine_id INTEGER,
    patient_id INTEGER,
    prescription_date CHAR(20)
  );
```

Rewrite this CREATE statement (don't do an ALTER) so that doctor_id, medicine_id and patient_id are Foreign Keys that correspond to the keys of Doctors, Medicines and Patients, respectively. The policies for referential integrity should be:

- Doctors can't be deleted if there are prescriptions from that doctor.
- Deleting a medicine deletes all the prescriptions for that medicine.
- If a patient is deleted, then the patient_id in any prescriptions that were for that patient should become NULL.

(You don't have to deal with what happens if there's an update.)

Answer 10:

```
CREATE TABLE Prescriptions (
    prescription_id INTEGER PRIMARY KEY,
    doctor_id INTEGER REFERENCES Doctors(doctor_id),
    medicine_id INTEGER REFERENCES Medicines(medicine_id) ON DELETE CASCADE,
    patient_id INTEGER REFERENCE Patients(patient_id) ON DELETE SET NULL,
    prescription_date CHAR(20)
  );
```

There is no need to specify an ON DELETE action for doctor_id, since SQL's default is to reject the deletion of a Doctors tuple if there are "child tuples" (in this case, Prescriptions) referencing that Doctors tuple.

Question 11:

11a): Explain in words exactly what the regular expression (in red) for person in the following address book DTD requires.

```
<!DOCTYPE addressbook [
  <!ELEMENT addressbook (person*)>
  <!ELEMENT person
      (name, address+, ( homephone | workphone | mobile )*, email?)>
  <!ELEMENT name           (#PCDATA)>
  <!ELEMENT address        (#PCDATA)>
  <!ELEMENT homephone   (#PCDATA)>
  <!ELEMENT workphone   (#PCDATA)>
  <!ELEMENT mobile          (#PCDATA)>
  <!ELEMENT email            (#PCDATA)>
]>
```

Answer 11a):

A person consists of exactly one name, followed by one or more addresses, followed by zero or more of any of homephone, workphone and mobile values, with those appearing in any order, followed by zero or one email values.

11b): Does the following data conform to that DTD?   (YES or NO)

```
person>
        <name> MacNiel, John </name>
        <addr> Rome, OH 98765 </addr>
        <addr> Paris, TX 75460 </addr>
        <workphone> (440) 555 1234 </homephone>
        <workphone> (903) 555 9876 </workphone>
        <mobile> (321) 555 2543 </mobile>
</person>
```

Answer 11b): _____NO_____

The data should have begun with <person>, rather than person>, so answer is NO, and everyone was told that. Everybody will get full credit for 11b).

However, assume that <person> was fixed. The answer would still be NO, because the first opening tag <workphone> has an incorrect ending tag, </homephone>. If that opening tag was <homephone> (or if the ending tag was </workphone>), then the answer would be YES>.

Part III: (15 points, 3 each):

Answer the following questions with **YES** or **NO**.

Question 12: For a database with the following relations, with primary keys underlined:

Employees(emp_id, ename, edept, esalary)
Departments(dept_id, dmanager, daddress)

Are the following two queries always equivalent?

```
SELECT ename, esalary
FROM Employees, Departments
WHERE edept = dept_id
AND esalary > 8000;
```

```
SELECT ename, esalary
FROM Employees
WHERE esalary > 8000
AND EXISTS ( SELECT *
             FROM Departments
             WHERE edept = dept_id );
```

Answer 12: ____YES______

Normally a JOIN and an Existential subquery wouldn't give you the same result, since there could be multiple matches in the second relation. But in that case case, they are the same, since dept_id is the primary key for Department, so there can be only one match (at most) in Departments for any employee in Employees. Both queries select the name and salary for employees whose salary is more than 8000 and who are in a department which appears in Departments (edept = dept_id).

Question 13: Suppose that R and S are relations, and that:

- *condR* is a condition only on attributes of R
- *condS* is a condition only on attributes of S
- *condX* is a condition on attributes of both R and S
- *cond* is (*condR* AND *condS* AND *condX*)

Is the following relational algebra equality always true?

$$\sigma_{cond} (R \times S) = \sigma_{condX} (\sigma_{condR} (R) \times \sigma_{condS} (S))$$

Answer 13: ___YES______

This is "predicate pushdown", moving predicates "down" in the execution tree if they are only on R (on only on S) so that they are applied as soon as possible.

Question 14: If R and S are union-compatible SQL tables with attributes A, B, C, are the following queries always equivalent?

```
( SELECT DISTINCT *                    ( SELECT *
  FROM R                                 FROM R
  WHERE A = 5 )                          WHERE A = 5 )

UNION ALL                              UNION

( SELECT DISTINCT *                    ( SELECT *
  FROM S                                 FROM S
  WHERE  B  = 12 )                       WHERE B = 12 )
```

Answer 14: _____NO_____

Although at first glance it may look like before left query and right query get rid of all duplicates, duplicates may appear in the left query because the same tuple might show up in the result selecting from R and in the result selecting from S. There are no duplicates for the query on the right, since UNION means UNION DISTINCT.

Question 15: Is the following relation with the specified Functional Dependencies in Third Normal Form?

Company_Info(Emp, Dept, Manager)
  Emp → Dept
  Dept → Manager

Answer 15: ___NO_______

The FD Dept → Manager doesn't meet any of the three requirements for 3NF, as we discussed in class.

Question 16: For OLAP, with a star schema, the number of facts in the Fact Table must always be less than or equal to the product that you get if you multiply together the number of values in the different Dimension Tables.

Answer 16: _____YES____

For the star schemas we discussed, the key of the Fact Table consists of the keys of the Dimension tables, so the maximum number of values in the Fact Table is the product of the sizes of the different Dimension Tables.

Part IV: (36 points, 6 each):

The questions in Part IV are about the following tables:

Sailors(<u>sid</u>, sname, age, rating) // sailor id, sailor name, age, rating
Boats(<u>bid</u>, bname, color) // boat id, boat name, color of boat
Reserves(<u>sid, bid</u>, day) // sailor id, boat id, date that sid reserved bid.

Question 17: Write a SQL statement that creates the Sailor table with sid as a primary key, sname unique, and a default rating of 'Beginner'. Age can be NULL, but the other attributes can't be NULL. sid and age are integers, and sname and rating are character strings of length 30. Also, age must be between 18 and 90.

Answer 17:

```
CREATE TABLE Sailors (
        sid             INTEGER PRIMARY KEY,
        sname           CHAR(30) UNIQUE NOT NULL,
        age             INTEGER CHECK (age >= 18 AND age <= 90),
        rating          CHAR(30) NOT NULL
   );
```

Question 18:

18a): Write a SQL statement that inserts a tuple into the Sailors table for a sailor named Henry whose rating is Shark, whose sid is 678, and whose age is 30.

Answer 18a):

```
INSERT INTO Sailors VALUES (678,  'Henry', 30, 'Shark');
```

Need to have the attributes in order, or specify where attributes are which explicitly.

18b): Write a SQL statement that deletes all the reservations made by a sailor named Carol.

Answer 18b):

```
DELETE FROM Reserves
WHERE Reserves.sid = ANY
    (   SELECT Sailors.sid
        FROM  Sailors
        WHERE Sailors.sname = 'Carol'  );
```

The ANY isn't necessary if sname is UNIQUE in Sailors. If sname weren't UNIQUE (meaning that there could be two sailors named Carol, we would need the ANY to DELETE reservations made by any of the sailors named Carol. Because of problem 17, your answer will be accepted if you omitted ANY.

Question 19:

19a): Write a View MultiBoatSailors that finds the sids of all Sailors who reserved at least two different boats.

Answer 19a):

```
CREATE VIEW MultiBoatSailors AS
        SELECT sid
        FROM Sailors
        WHERE     (   SELECT Count(DISTINCT bid)
                      FROM Reserves
                      WHERE Reserves.sid = Sailors.sid   )  >= 2;
```

Note that DISTINCT is necessary here. A Sailor who reserved the same boat on different days but didn't reserve any other boats wouldn't qualify.

19b): Using the MultiBoatSailors View, write a query that gives the names of Sailors who reserved at least two boats, together with the names of each of the different boats that they reserved.

Answer 19b):

```
SELECT s.sname, b.bname
FROM Sailors s, Boats b, MultiBoatSailors m
WHERE s.sid = m.sid
        AND EXISTS (
                SELECT *
                FROM Reserves r
                WHERE r.sid = s.sid
                AND r.bid = b. bid      );
```

sid is unique in both Sailors and MultiBoatSailors, so joining Sailors with MultiBoatSailors is the same as checking whether a Sailors sid also appears in MultiBoatSailors. Okay to write another Existential subquery instead.

Question 20:

20a): Write a statement that creates an index on Sailors' age and rating attributes.

Answer 20a):

CREATE Index SailorAgeRating ON Sailors(age, rating);

20b): Is that the same as creating an index on Sailors' rating and age attributes? Explain your answer. (Don't bother to create another index.)

Answer 20b):

No, it's not the same. For 20a), the index helps you look up tuples in Sailors that have an age, and within that age, tuples that have an age, and within that, tuples that have a rating. Such an index would be good for finding tuples that have a specific age and rating, or that have a specific age, or that have an age greater than some value, but it wouldn't be that good at finding tuples that have a specific rating (you'd have to look through all the different ages).

An index on rating and then age would also be good at finding tuples that have a specific rating and age (same as specific age and rating) and tuples that have a specific rating. But it wouldn't be that good at finding tuples that have a specific age (you'd have to look through all the different ratings) or tuples that have an age greater than some value.

Question 21: Assume that a JDBC connection myCon has been established to our Sailors/Boats/Reserves database.

Print out all the names of all the boats that the sailor named Henry reserved. Don't bother with including libraries or variable declarations, and you can have an informal print statement if you want.

Here 's an outline of what you need to write:

```
// Execute the query
// Loop through the results
// For each value in the result, get the values of boat name, and print it
```

Answer 21:

```
// Execute the query
Statement stmt = myCon.createStatement();
resultset = stmt.executeQuery(
              "SELECT bname
              FROM Boats b
              WHERE EXISTS (
                     SELECT * FROM Sailors s, Reserves r
                     WHERE r.sid = s.sid
                       AND r.bid = b.bid
                       AND s.name = 'Henry' )  "
              );

// Loop through the results
while (resultset.next()) {
  // For each value in result, get values of bname, and print it
  System.out.println(resultset.getString(1));
}
```

Question 22: For this question only, let's assume that Sailors that also has the Functional Dependency **age → level**. Here's an instance of the Sailors table:

sid	sname	Age	Level
123	John	30	Shark
333	Sam	32	Whale
456	Mary	30	Shark
663	Alan	20	Bass
789	Bob	20	Bass
953	Linda	24	Dolphin

22a): Explain one of the Anomalies that we discussed, identifying the Anomaly and giving an example based on the above instance and Functional Dependency.

Answer 22a):

Question asks you to describe just one Anomaly, but here's an explanation of all three using the above instance and FD. All the anomalies involve redundancy in the relation, exception for the Delete Anomaly, which could lose information.

- **Insert Anomaly**: If Sarah was inserted with Age 20, she'd need to have Level Bass, or else the FD age → level would be violated.
- **Update Anomaly**: If John's age became 32, he'd need to Level updated to Whale, or else the FD age → level would be violated.
- **Delete Anomaly**: If Linda were deleted, we'd lose knowledge that people who are Age 24 should have Dolphin at their Level.

22b) Could you determine, based on an instance, whether or not the Functional Dependency **age → level** holds for all instances? Explain your answer briefly.

Answer 22b):

No, you can't tell whether an **FD holds** based on looking only at an instance, since having an FD mean that attribute (or attributes) determine some attribute in every instance that can exist. Knowing about one instance, or even a bunch of instances, doesn't tell you about what could happen in other instances.

However, by looking at one instance, you can tell that an FD does not hold for all instances if you see that an **FD doesn't hold** for that instance**.** For example, you can tell that Level doesn't functionally determine sname, since there are two tuples with the same Level but with different names.