CPS 216 Fall 2001
Homework #2
Due: Thursday, October 4

Please note: *The last problem of this homework requires working with IBM DB2. Start early and allow yourself some time to get familiarized with the system.*

Problem 1.

Once again, consider the database containing information about World War II capital ships from Problem 1 of Homework #1. It involves the following tables:[1]

```
CREATE TABLE Class(class VARCHAR(30) NOT NULL PRIMARY KEY,
                   type CHAR(2) NOT NULL CHECK(type='bb' OR type='cc'),
                   country VARCHAR(20) NOT NULL,
                   numGuns INTEGER NOT NULL,
                   bore INTEGER NOT NULL,
                   displacement INTEGER NOT NULL);
CREATE TABLE Ship(name VARCHAR(20) NOT NULL PRIMARY KEY,
                  class VARCHAR(20) NOT NULL,
                  launched DATE NOT NULL);
CREATE TABLE Battle(name VARCHAR(10) NOT NULL PRIMARY KEY,
                    date DATE NOT NULL);
CREATE TABLE Outcome(ship VARCHAR(20) NOT NULL,
                     battle VARCHAR(10) NOT NULL,
                     result CHAR(7) NOT NULL
                                   CHECK(result='sunk' OR
                                         result='damaged' OR
                                         result='ok'),
                     PRIMARY KEY(ship, battle));
```

Write SQL queries to answer the following queries. Make sure your final answers include no duplicate rows; but add the keyword `DISTINCT` *only* in cases where duplicates would otherwise be produced.

(a) List the name, displacement, and number of guns for each ship engaged in the battle of Guadalcanal.
(b) Using a subquery in `WHERE`, find all ships that were damaged in the battle of the North Atlantic. List their names, classes, and launch dates. Sort the list by class (in alphabetical order). Within the same class, sort by launch date (in chronological order).
(c) Without using subqueries, find the ships that sunk before 1943. List the name, class, and country for each ship.
(d) Find all battles fought by *both* "California" and "North Carolina."
(e) Using aggregates, find the ships that fought at least two battles.
(f) Same as above, but use a subquery in `WHERE` instead of aggregates.
(g) Same as above, but do not use subqueries or aggregates.

[1] If you wish to try it out on a DBMS, you might need to change some table and/or columns names in this schema to avoid conflicts with reserved keywords in SQL.

(h) Find ships that fought in *every* battle that "California" fought in.
(i) Find all battles involving more than ten ships. For each battle, also list the total displacement of all ships damaged or sunk in the battle.

Write SQL modification statements to complete the following tasks:

(j) If a ship has fought more than eight battles, put it on `RollOfHonor(ship, date)`. The date should be that of the eighth battle. A ship only gets on the Roll of Honor once.
(k) You noticed that in the `Outcome` table, some ships had already sunk but they were later "resurrected" to fight in more battles. Fix this inconsistency by changing their results in earlier battles from "`sunk`" to "`damaged`."

The last question is about constraints.

(l) Rewrite the `CREATE TABLE` statements to declare all referential integrity constraints that are reasonable in this schema.

Problem 2.

The *outerjoin* of tables *R* and *S* is a variation of the join of *R* and *S* that also preserves information in "dangling" rows. A row is dangling if it does not join with any row from the other table. The outerjoin result has the same columns as *R* join *S*. It includes all rows in *R* join *S*, plus all dangling rows of *R* and *S*. To match the result schema, the dangling rows are padded with `NULL` for all "missing" columns. Here is an example:

R

A	*B*
1	2
3	4

S

B	*C*
2	5
6	7

R `OUTERJOIN` *S*

A	*B*	*C*
1	2	5
3	4	`NULL`
`NULL`	6	7

Write an SQL query to compute the outerjoin of *R* and *S* (without using SQL's built-in outerjoin operator).

Problem 3.

Some schedules are not conflict-serializable, yet they are still equivalent to some serial schedules. For example, consider the following schedule:

T_1.read(A), T_2.write(A), T_2.commit, T_1.write(A), T_1.commit, T_3.write(A), T_3.commit

(a) Draw the precedence graph for this schedule and show that it is not conflict-serializable.

(b) Show a serial schedule that is equivalent to this schedule.[2] Recall that in a serial schedule, transactions are executed in some serial order, with no interleaving operations.

Problem 4.

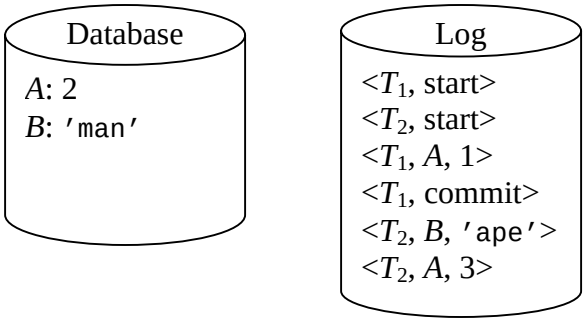
For each schedule below, tell whether it is conflict-serializable. If yes, also tell:

- Whether it is recoverable;
- Whether it avoids cascading rollbacks;
- Whether it is possible under *strict* 2PL.

(a) T_1.write(B), T_2.read(A), T_2.write(A), T_1.read(A), T_1.write(A), T_1.commit, T_2.commit
(b) T_1.write(B), T_2.read(A), T_2.write(A), T_1.read(A), T_1.write(A), T_2.commit, T_1.commit
(c) T_1.write(B), T_2.read(A), T_2.write(A), T_2.commit, T_1.read(A), T_1.write(A), T_1.commit
(d) T_1.write(B), T_2.read(A), T_1.read(A), T_2.write(A), T_1.write(A), T_2.commit, T_1.commit
(e) T_2.write(B), T_2.read(A), T_2.write(A), T_1.write(B), T_2.commit, T_1.read(A), T_1.commit

Problem 5.

Suppose a DBMS restarts after a crash and finds the following information on disks:



(a) If the system uses *undo logging*, what were the initial values of A and B on the disk before T_1 and T_2 started?
(b) If the system uses *undo logging*, what will be the final values of A and B on the disk after the recovery process is finished?
(c) If the system uses *redo logging*, what were the initial values of A and B on the disk before T_1 and T_2 started?
(d) If the system uses *redo logging*, what will be the final values of A and B on the disk after the recovery process is finished?

---

[2] A schedule like this one is said to be *view-serializable* because it is *view-equivalent* to a serial schedule. A schedule S_1 is view-equivalent to another schedule S_2 if the following conditions hold:

- If T reads the initial value of X in S_1, then T must also read the initial value of X in S_2.
- If T_i reads the value of X written by T_j in S_1, then T_i must also read the value of X written by T_j in S_2.
- If T performs the final write on X in S_1, then T must also perform the final write on X in S_2.

Although more general than conflict-serializability, view-serializability is much more expensive to enforce. It also turns out that every view-serializable schedule that is not conflict-serializable must contain some blind write (writing an object without reading it first), which is rare in practice. Therefore, view-serializability is only of theoretical interest.

Problem 6.

You and your CPS 216 project partner are working on a transaction processing system that stores both old and new values in its undo/redo log. In this system, log records typically are 250 bytes long, with the old value of the modified record taking up 100 bytes, and the new value taking another 100 bytes.

Your partner suggested the following idea for reducing the size of log records. Say a transaction updates a record, originally with value v_{old}, to new value v_{new}. Instead of logging both v_{old} and v_{new}, the system can log the value $v_{XOR} = v_{old}$ XOR v_{new}, i.e., the bit-wise exclusive-or of v_{old} and v_{new}. This logging scheme reduces the log record size from 250 bytes to 150 bytes.

If after a crash we wish to undo an action, we take the new value v_{new} found in the appropriate record in the database and do an XOR with v_{XOR} (found in log) to obtain v_{old}, which is then stored in the database. Similarly, if we wish to perform a redo, we read the old value v_{old} from the database and do an XOR with to get v_{new}.

(a) Unfortunately, the scheme proposed by your partner does not quite work correctly. Please explain what the problem is.
(b) Suggest a fix!

Problem 7.

Recall Problem 5 of Homework #1.

(a) Take your relational design (or the one in the sample solution) and create the schema in IBM DB2 using `CREATE TABLE` statements. For instructions on how to use DB2 (running on `rack40.cs.duke.edu`), please refer to the link "Getting Started with IBM DB2" on the course Web page. Choose appropriate data types for your columns, and remember to declare keys and referential integrity constraints. Then, populate your tables with some data using `INSERT` statements—a couple of rows in each table would be sufficient. When you are happy with your database, run the command "`/home/cps216/submit/hw2-7a > hw2-7a.txt`" in shell (on `rack40.cs.duke.edu`). Print out the file named `hw2-7a.txt`, and turn it in with the rest of your homework.
(b) The goal of the following exercise is to reverse engineer the view modification algorithm used by DB2. With the schema you created, try different types of views (selection, join, selection with subqueries, aggregates, etc.) using `CREATE VIEW` statements and see if they are modifiable using `INSERT`, `DELETE`, and `UPDATE`. Based on your experiments, answer the following questions:
    - What types of views are modifiable in DB2?
    - How does DB2 translate a modification on the view to modifications on base tables? If you discover anything unexpected or peculiar, describe it.