CPSC 304 Solutions To Sample Final Exam Questions
December 15, 2009

1. a) T, b) T, c) F, d) T, e) F, f) F, g) T, h) T, i) T, j) T (but some DBMSs may allow a duplicate row)

2. Table is created ... playerID is a field/column of NHL_STATS and is a foreign key, referencing a candidate key by the same name in the NHL_PLAYERS table ... if a row in NHL_PLAYERS is deleted, then all rows with the same playerID are deleted from the NHL_STATS ... and what is perhaps the most important referential integrity reason: that every row in NHL_STATS must have a row with the same playerID already in NHL_PLAYERS (for INSERTs).

3. a) One solution is:

```
SELECT      GolferID, count(*), avg(Score)
FROM        Scores
GROUP BY    GolferID
ORDER BY    2 desc;
```

b) One solution is:

```
rho(VANGOLFCOURSES, pi_{CourseID} (sigma_{location = ‘Vancouver’} GolfCourses))

rho(SCORING, pi_{GolferID, CourseID} (sigma_{DatePlayed >= 2000-01-01} AND
    DatePlayed <= 2000-12-31} Scores))

SCORING / VANGOLFCOURSES
```

4. a) 15
   b) A+ = A, B+ = B, C+ = C, D+ = D,
   AB+ = ABCD, AC+ = AC, AD+ = AD,
   BC+ = BCD, BD+ = BD, CD+ = CD,
   ABC+ = ABCD, ABD+ = ABCD, ACD+ = ACD,
   BCD+ = BCD, ABCD+ = ABCD
   c) only AB
   d) No, B,C->D violates BCNF ...
   R=(A,B,C,D) decomposes (using lossless-join) into R1=(A,B,C) and R2=(B,C,D)

5. Entities: Judge, Panel, Courtroom, Appeal
   Relationships: Sits_On (between Judge & Panel) and a ternary relationship called Hearing (among Panel, Courtroom, and Appeal)

6.

a) During Analysis:

- start with LSN 00, begin checkpoint
- assume Transaction Table (TT) and Dirty Page Table (DPT) are empty to begin
- LSN 10: no action needed
- LSN 20: add (T1,20) to TT; add (P61,20) to DPT
- LSN 30: add (T2,30) to TT; add (P72,30) to DPT
- LSN 40: add (T3,40) to TT; add (P95,40) to DPT
- LSN 50: no action needed (although to be technically correct we can update the status of T1 to "commit" in the TT; sometimes, to simplify things, we don't bother with the "end" records in class or the text)
- LSN 60: remove T1 from TT
- LSN 70: no action needed (can update status of T1 to commit in TT)
- LSN 80: update (T3,80) in TT; add (P77,80) to DPT
- LSN 90: remove T2 from TT
- LSN 100: add (T4, 100) to TT; no change to P72 in DPT
- Note that, at this point, only (T3,80) and (T4,100) exist in the TT; and (P61,20), (P72,30), (P95,40), and (P77,80) exist in the DPT.

b) During Redo:

- start redo at smallest recLSN in DPT
- LSN 20: redo changes to P61
- LSN 30: redo changes to P72
- LSN 40: redo changes to P95
- LSN 50: no action
- LSN 60: no action
- LSN 70: no action
- LSN 80: redo change to P77
- LSN 90: no action
- LSN 100: redo change to P72

c) During Undo:

- start at highest (lastLSN) in TT: 100
- ToUndo = {100, 80}—called the "loser set" that applies to all in-flight transactions; we'll work backwards
    - i. So, for LSN 100, undo changes to P72
    - ii. add CLR to log: Undo T4 LSN 100, nextUndoLSN is null
    - iii. add "T4 End" to log

b. ToUndo = {80}
    - i. So, for LSN 80, undo changes to P77
    - ii. add CLR to log: Undo T3, LSN 80, nextUndoLSN = 40
    - iii. add {40} to loser set

c. ToUndo = {40}
    - i. So for LSN 40, undo changes to P95
    - ii. add CLR to log: Undo T3 LSN 40, nextUndoLSN = null
    - iii. add "T3 end" to log

d. ToUndo = {}. Done. So, take new checkpoint (begin checkpoint, end checkpoint).

e. DBMS is back up and running for users.

7. Crash recovery means the whole DBMS went down, possibly due to a power outage, a DBMS bug/crash, an OS bug/crash, etc. All in-flight transactions die, the buffer pool's contents are lost, and some tables may be left in a partial state of update. A rollback is the SQL term for abort; they mean the same thing. When we rollback a transaction, we undo it's updates (as per the events recorded on the log). After the rollback, the transaction's effect are no longer seen in the database; it's as if that transaction never occurred. Unlike crashes, a rollback can be initiated by the user or by the DBMS. Rollbacks tend to be pretty simple compared to crash recoveries. The former may take a second or less; the latter several minutes, perhaps. Many users are inconvenienced during crashes; few users are affected by a rollback—and the user is often happy that the transaction rolled back (depending on the circumstances).

8. As discussed in class, multiple granularity locks allow different intent locks (or warning locks) to be placed on a hierarchy of objects. If we take an S lock on a particular row, then another transaction cannot take an X lock on that same row. If we take an S lock on a particular table, then another transaction cannot take an X lock on that same table. The first case is fine: many times, dozens of users are simultaneously reading and writing rows in a table, and they are going after different rows. So, no one gets in each other's way. The second case is not good, because it means that if someone takes an S lock on the *whole* table (and doesn't release it), then someone else cannot take an X lock on the table (or actually, even on any part of it). Why not? Because the first user is effectively saying, "I'm reading all of the rows in this table over the next few units of time." This shuts out any transaction that wants to change any row. (Recall that this is called a "conflict" in our Transaction Management unit.)

   What kind of transaction might read the whole table?
   ```
   SELECT      *
   FROM        Sailors;
   ```

   What kind of transaction might write the whole (or most of the) table?
   ```
   UPDATE      Sailors
   SET         Rating = Rating + 1
   WHERE       Rating < 10;
   ```

   What kind of transaction might read one row?
   ```
   SELECT      *
   FROM        Sailors
   WHERE       sid = 101;
   ```

   What kind of transaction might write that same row?
   ```
   UPDATE      Sailors
   SET         Rating = 10
   WHERE       sid = 101;
   ```

9. We are better off using Optimistic Concurrency Control when there are few users that interfere with one another. Then, we don't bother with locking ... and just assume that everybody is going to be

conflict-free (most of the time). This yields good performance gains. Later, at validation time, we just check the timestamps (and the read and write sets, i.e., objects) to see if there is any possibility of two transactions interfering with one another. If there isn't any overlap/intersection, then we're done, and both transactions can commit. If there is overlap/intersection, then we'll roll back one of the transactions.

Optimistic concurrency control won't work well on a system where there is heavy contention for the same objects (e.g., booking Olympic tickets). There, you're probably much better off using locking (e.g., Strict 2PL).

Locking assumes the worst. Optimistic assumes the best; hence, the name.