CSE 344 – 26wi – Final Exam Review **KEY**

Last Name:		
First Name:		
UWNetID	@uw.edu	
Person to your Left | Person to your Right	@uw.edu	@uw.edu
All submitted work is my own and was completed in compliance with the below instructions. I will not share the contents of this exam with others who have not taken it yet. *I acknowledge that violation of these terms could result in a failing grade.* **(please sign)**		

Instructions:

- This exam contains **11 pages**, including this cover page. Show scratch work for partial credit, but write your final answers in the boxes and blanks provided.
- Write your UWNetID at the **top of every page**. This helps us reassemble your exam when its pages are inevitably separated and lost.
- The exam is **closed book**. Please do not use laptops, tablets, wearable devices, mobile phones, notes sheets, or books.
- Please silence and put away all noise-making devices, including mobile phones and wearable devices.
- You have 50 minutes to complete this exam.

Advice:

- Read *all* the questions before answering any questions. Start where you feel the most confident.
- Skip questions that are taking a long time. You can always come back.
- Breathe. You've got this. You belong here.

Question 1 (XXX points):

Assume we have the following relation:

```
CREATE TABLE Students(
  sid           INTEGER PRIMARY KEY,
  name          VARCHAR(100),
  email         VARCHAR(100),
  enrollment_date DATE
);

CREATE TABLE Courses(
  cid           INTEGER PRIMARY KEY,
  title         VARCHAR(100),
  department    VARCHAR(50),
);

CREATE TABLE Enrollments(
  sid           INTEGER PRIMARY KEY,
  cid           VARCHAR(100),
  grade         VARCHAR(100),
  enroll_date   DATE,
  PRIMARY KEY (sid, cid),
  FOREIGN KEY (sid) REFERENCES Students(sid),
  FOREIGN KEY (cid) REFERENCES Courses(cid)
);
```

Assume we are using a PostgreSQL database with standard SQL syntax.

Question 1.a (XXX points):

A new course called "Computer Networks" is being offered in the CSE department. Insert this course into the Courses table with cid = 461. Write a SQL query to perform this insertion.

```
INSERT INTO Courses (cid, title, department)
VALUES (461, 'Computer Networks', 'CSE');
```

Question 1.b (XXX points):

A university policy requires that all students who enrolled before January 1, 2023 and currently have no email address (NULL) must have their email set to: unknown@university.edu Write a SQL query to perform this update.

```
UPDATE Students
SET email = 'unknown@university.edu'
WHERE email IS NULL
AND enrollment_date < '2023-01-01';
```

Question 1.c (XXX points):

Due to a system error, some enrollments were recorded with an invalid grade of -1. Write a SQL query to delete all enrollments where:

- the grade is -1
- **AND** the enrollment occurred before January 1, 2024

```
DELETE FROM Enrollments
WHERE grade = -1
AND enroll_date < '2024-01-01';
```

Question 2 (XXX points):

Assume we have the following two tables:

```
Students
sid | name
-------------
1   | Alice
2   | Bob
3   | Charlie

Enrollments
sid | course
-------------
1   | CSE344
2   | CSE444
4   | CSE311
```

For each part below, write the output of the query:

Question 2.a (XXX points):

```
SELECT Students.sid, Students.name, Enrollments.course
FROM Students
INNER JOIN Enrollments
ON Students.sid = Enrollments.sid;
```

```
sid | name  | course
-----------------------
1   | Alice | CSE344
2   | Bob   | CSE444
```

Question 2.b (**XXX** points):

```
SELECT Students.sid, Students.name, Enrollments.course
FROM Students
LEFT JOIN Enrollments
ON Students.sid = Enrollments.sid;
```

```
sid | name    | course
--------------------------
1   | Alice   | CSE344
2   | Bob     | CSE444
3   | Charlie | NULL
```

Question 3 (**XXX** points):

Decompose `relation R` into BCNF (see below table). At each step of your decomposition, state which "bad FD" you are decomposing on. When you are done, circle the final relations. For each of your final relations, annotate which of the original FDs apply to it and indicate its keys.

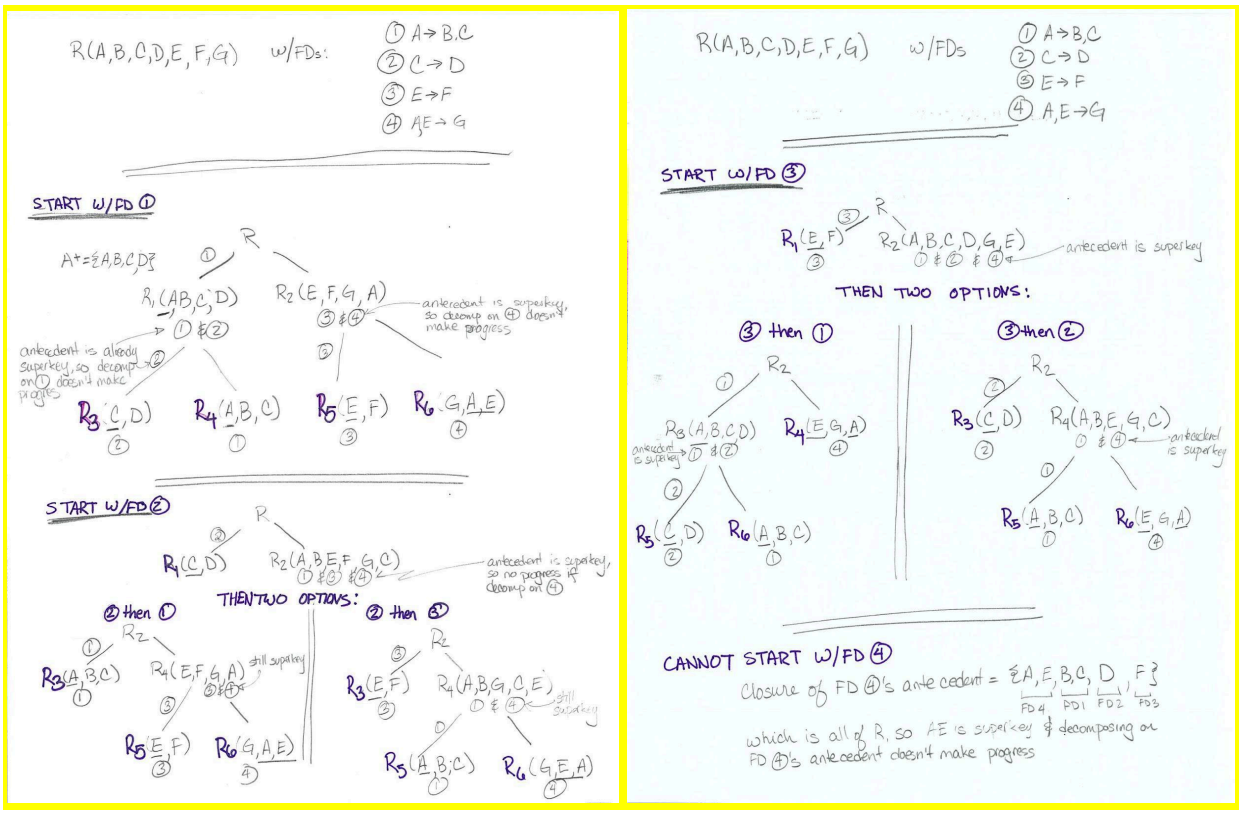
```
R(A, B, C, D, E, F, G)
   1. A  →  B, C
   2. C  →  D
   3. E  →  F
   4. A, E  →  G
```

Answer: There are 5 possible "decomposition paths" to a solution, all of which yield:

- R1(A, B, C) w/FD 1
- R2(C, D) w/FD 2
- R3(E, F) w/FD 3
- R4(A, E, G) w/FD 4

Note that we *never* chose FD 4 as a "bad FD" on which to decompose, for *any* of the decomposition paths. FD 4's antecedent – namely, (A, E) – is a superkey of R and every subset of R that appeared in the decompositions; thus, any attempt to decompose on FD 4 would yield the exact same table again.

See below for visual:



Question 4 (XXX points):

Consider the following statements about database indexing we have discussed in lecture & section.
Select **ALL** statements that are **TRUE**

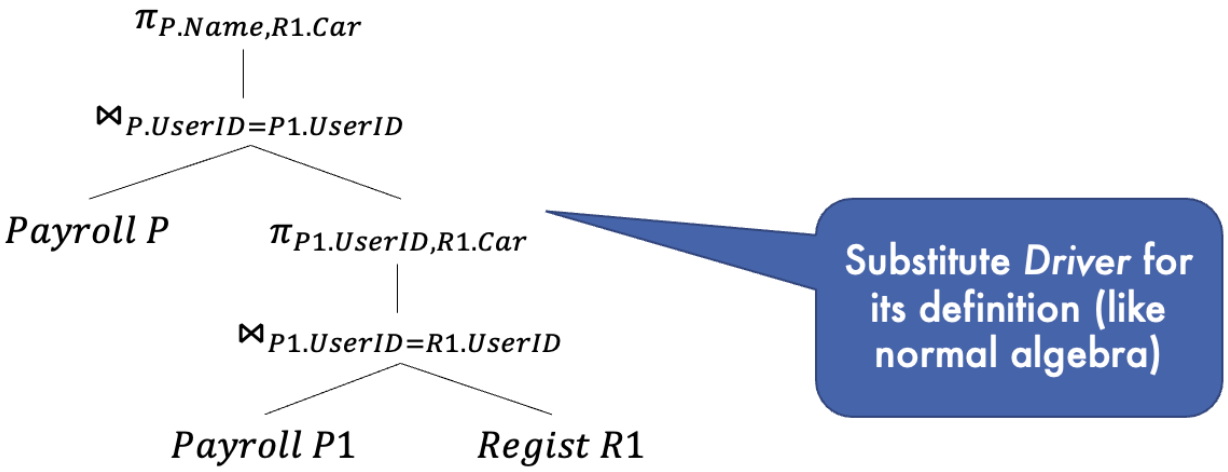
- ▢ A clustered index determines the physical order of rows in a table's underlying file
- ▢ A table can have multiple clustered indexes
- ▢ An unclustered index stores pointers to the rows of a table rather than storing the table's rows in index order
- ▢ Using an index always speeds up every type of query.
- ▢ Indexes can significantly improve the performance of queries that filter using WHERE conditions
- ▢ Indexes can make INSERT, UPDATE, and DELETE operations slower
- ▢ A primary key is often implemented using an index

Question 5 (XXX points):

```
WITH Driver AS
     (SELECT P1.UserID AS UserID, R1.Car AS Car
      FROM Payroll AS P1, Regist AS R1
```

```
      WHERE P1.UserID = R1.UserID)
SELECT P.name, D.car
  FROM Payroll AS P, Driver AS D
 WHERE P.UserID = D.UserID
```

Given the SQL query above, translate it into relational algebra:



Question 6 (**XXX** points):

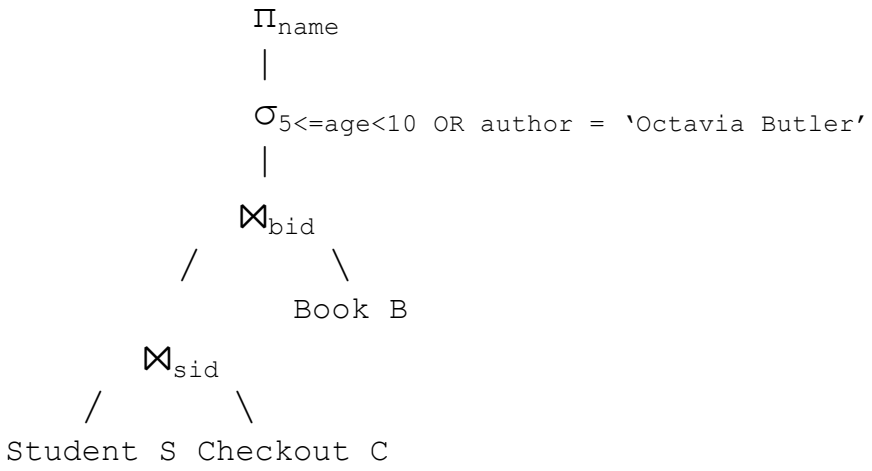
Estimate the cardinality at each stage of the query plan below based on the following information about a library system:

```
Students(sid, name, age)
Books(bid, title, author)
Checkout(sid, bid, date)

T(S) = 25,000
T(B) = 75,000
T(C) = 200,000

V(B, author)= 1000
```

- Student ages are in the range: 5 <= age < 25
- Assume that no student younger than 10 has checked out a book by Octavia Butler



After the 1st join:

- We want to take the cross product of the two tables' tuple counts (25,000 * 200,000 = 5,000,000,000).
- We want to divide this number by the maximum number of distinct sid's between the two tables we are joining. Logically, any sid listed in C must have a match in S (since sid is the PK in S), meaning that the maximum number of sid's will come from table S. By the nature of a PK, this will just be the tuple count of S.
- Computing the division, we get 200,000 tuples resulting from the first join.

After the 2nd join:

- This follows the same logic as the first join. Taking the resulting 200,000 tuples from the first join, we'll multiply it by the 75,000 tuples in Books.
- We'll divide the resulting value by 75,000 (maximum distinct bid's possible) to get a cardinality of 200,000 resulting tuples.

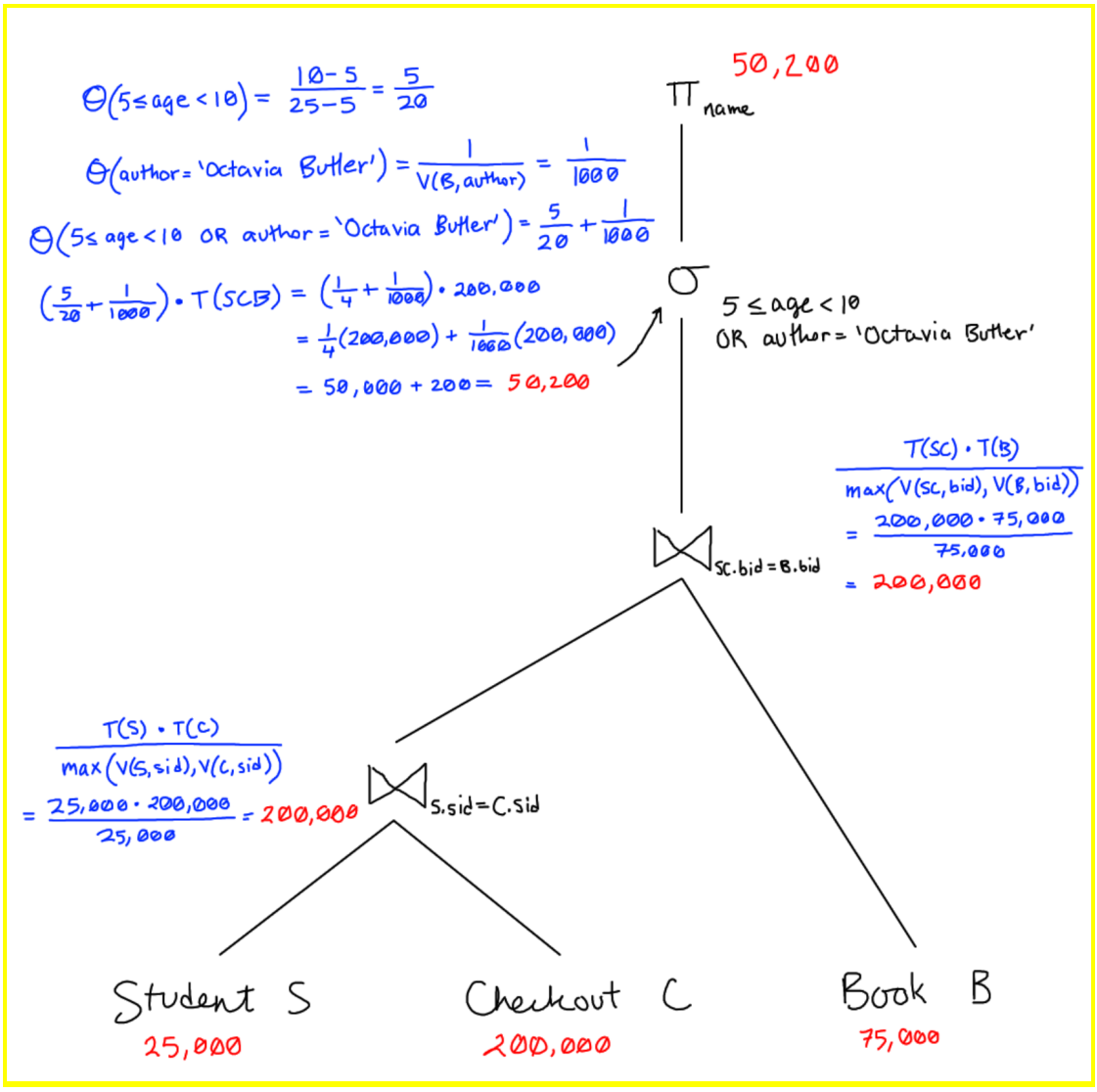
After the selection:

1. The first condition of the select generates a selectivity of 5/20 based on the age ranges.

2. The second condition, we are selecting 1 author out of 1000 for a selectivity of 1/1000.
3. Multiply the 200,000 tuples by the addition of both of these selections to get 50,200 resulting tuples.

After the projection:

- The projection is not changing the tuple count, so we once again have 50,200 tuples.



Question 7 (XXX points):

Match each ACID property with the correct definition. Write the letter of the correct property next to each definition.

ACID PROPERTIES

A. Atomicity
B. Consistency
C. Isolation
D. Durability

DEFINITIONS

(D) 1. Once a transaction commits, its changes will persist even if the system crashes

(A) 2. A transaction is treated as an all-or-nothing operation; either all of its changes occur or none do

(C) 3. Concurrent transactions should behave as if they executed one at a time, without interfering with each other

(B) 4. The database must move from one valid state to another, preserving all defined rules and constraints

Question 8 (XXX points):

Instead of proceeding from an ER diagram to a SQL table, we're going to work backwards from the table back to an ER diagram! Consider a hypothetical `Paintings` table with the following FDs:

```
artist_name  →  birth_country
museum_id  →  location
pid  →  (title, artist_name, birth_country, museum_id, location)
```

Some example values might be as follows:

pid	title	artist_name	birth_country	museum_id	location
1	The Last Supper	Leonardo da Vinci	Italy	M001	Milan, Italy
2	The Last Supper	Salvador Dalí	Spain	M002	Washington, USA
3	Lady with an Ermine	Leonardo da Vinci	Italy	M003	Kraków, Poland
4	The Birth of Venus	Sandro Botticelli	Italy	M004	Florence, Italy
5	Woman with a Parasol	Claude Monet	France	M004	Florence, Italy
6	Madonna of the Goldfinch	Raphael	Italy	M005	Florence, Italy

Imagine that you represent Paintings as **two or more entity sets**: Artists, Museums, and potentially a third entity set. How would you represent the following?

Question 8.a (XXX points):

birth_country

- An attribute on Artists
- An attribute on Museums
- An attribute on the hypothetical 3rd entity set
- The hypothetical 3rd entity set itself

An artist is born in a specific country, so it's an attribute on artists

Question 8.b (XXX points):

Paintings

- An attribute on Artists
- An attribute on Museums
- An attribute on the hypothetical 3rd entity set
- The hypothetical 3rd entity set itself

From our example data, we can see that there's a many-to-many relationship between artists and the museums which store their paintings. So Paintings will have to be represented as an entity set.

Question 8.c (**XXX** points):

Title

- An attribute on Artists
- An attribute on Museums
- An attribute on the hypothetical 3rd entity set
- The hypothetical 3rd entity set itself

A painting's title doesn't fit cleanly into an artist nor into a museum. Regardless of whether Paintings is your 3rd entity set, the Title is clearly not an attribute of Artists or Museums. You could consider Title to be its own entity set (which would have a relationship to a Paintings-like entity set), but it's probably cleanest to represent Title as an attribute on the Paintings entity set you created in the earlier question.

Question 9 (**XXX** points):

Consider the relationship between conflict serializability and Strict 2PL. Determine whether the following statement is True or False:

"Strict 2PL guarantees conflict serializability, but there exist conflict-serializable schedules that cannot occur under Strict 2PL"

Is this statement True or False? Circle One:

(True) (False)

Question 10 (XXX points):

Consider the following transaction schedule:

T_1	T_2	T_3
R(A)		
	R(A)	
W(A)		
		R(B)
	R(C)	
	W(A)	
R(B)		
		W(C)
W(A)		
		R(C)
R(C)		
	W(C)	
		R(B)

The schedule is shown in tabular format where each column represents a transaction (T1, T2, T3) and each row represents the next operation in time.

Question 10.a (XXX points):

Is this schedule conflict-serializable? Explain your answer using a precedence graph

Answer: this isn't conflict serializable due to:

- The cycle between T1 and T2 on A
- The cycle between T2 and T3 on C
- The cycle from T1 -> T2 -> T3 (on A, then C, then C)

Question 10.b (XXX points):

Is this schedule serializable? Briefly explain your reasoning

Answer: This might be serializable, but we don't have the starting values of A, B, and C – nor do we know what the writes consist of (so we don't have the ending values of A, B, and C)

Question 11 (XXX points):

Select **ALL** statements that are **TRUE** about binary locks:

- ▢ Binary locks allow multiple transactions to read a data item simultaneously
- ▢ Binary locks have only two states: locked or unlocked
- ▢ Binary locks distinguish between shared and exclusive access
- ▢ Binary locks allow writes but prevent reads

Question 12 (XXX points):

Select **ALL** statements that are **TRUE** about B+ tree indexes and hash indexes.

Note: If you select "None of the above", you may NOT select any other option.

- ▢ Hash indexes are best for range queries
- ▢ Hash indexes are efficient for equality searches, while B+ trees are more efficient for range queries
- ▢ B+ trees only store keys in the root node
- ▢ None of the above