CSE 344 Final Exam  
Summer 2019  
August 23, 2019

- Please read all instructions (including these) carefully.
- **This is a closed-book exam.** You are allowed two pages of note sheets that you can write on both sides.
- Write your name and UW student number below.
- No electronic devices are allowed, including cell phones used merely as watches.
- Solutions will be graded on correctness and **clarity**. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- There are 10 pages in this exam, including this one.
- There are 5 questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 60 minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the blank pages as scratch paper. **Do not** use any additional scratch paper.

Relational algebra operators:

<table>
<thead>
<tr>
<th>Union ⋃</th>
<th>Difference —</th>
<th>Selection σ</th>
<th>Projection π</th>
<th>Join ⋈</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename ρ</td>
<td>Duplicate elimination δ</td>
<td>Grouping and aggregation γ</td>
<td>Sorting τ</td>
<td></td>
</tr>
</tbody>
</table>

By writing your name below, you certify that you have not received any unpermitted aid for this exam, and that you will not disclose the contents of the exam to anyone in the class who has not taken it.

NAME: ______________________________

STUDENT NUMBER: ___________________________

Text in red indicates clarifications post-exam

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>80</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1: Warm Up (20 points total)

Select either True or False for each of the following questions. For each question you get 2 points for answering it correctly. There is no penalty for an incorrect answer.

a) In parallel a RDBMS, horizontal partitioning means splitting a relation among different nodes by dividing its tuples between those nodes.
   True    False

b) A relation can have more than one unclustered index created on it.
   True    False

c) In transaction scheduling, the problem of phantom tuples is solved by Strict 2-phase locking.
   True    False

d) The semi-structured data model allows datasets to break the “flat table” requirement of the relational model.
   True    False

e) In E/R diagrams, the “many-to-one” property of a relationship is an example of a database constraint.
   True    False

f) In database concurrency, "Atomic" means that no matter the actual schedule executed by the system, the effect should be the same as if the transactions run one after the other.
   True    False

g) For a given SQL query, query optimizers consider multiple logical plans and multiple physical plans.
   True    False

h) In schema design, a table can have many super-keys but we only choose one primary key.
   True    False

i) MapReduce solves fault tolerance problems by saving the lineage of the query in RDDs.
   True    False

j) An index can be created on multiple attributes, for example “CREATE INDEX idx_name
   ON Person (fname, lname);”
   True    False
Problem 2: Transactions (19 points total)

a) (5 points) Draw the precedence graph of the following schedule. Is this schedule conflict serializable? If so, write YES and the order in which the transactions would have to run in the equivalent serial schedule. Otherwise write NO.

\[ R_1(A); W_3(B); R_1(B); R_3(A); W_1(A); W_2(A); W_1(B); W_2(A); R_2(B); \]

Answer: Yes, it is conflict serializable. The order is 3 -> 1 -> 2

b) (5 points) Draw the precedence graph of the following schedule. Is this schedule conflict serializable? If so, write YES and the order in which the transactions would have to run in the equivalent serial schedule. Otherwise write NO.

\[ R_3(B); R_1(A); W_3(B); R_3(A); R_1(B); W_3(A); W_2(B); R_1(A); \]

Answer: No, it is not conflict serializable.
c) (3 points) Consider the following two transactions executed under strict two-phase locking (strict 2PL).

T1: R(A), W(A), W(B) COMMIT

T2: R(A), W(A), W(B) COMMIT

Is deadlock possible between these two transactions? Circle one: YES / NO

Note: If transactions only request locks as they need them, deadlock will not happen, but if they can acquire locks any time they want then a deadlock schedule is possible, so “yes” was also accepted.

d) (6 points) For the transactions above, fill in a schedule in the table below that uses strict 2PL and does not deadlock.
Indicate when any locks are acquired (ex: L(A)) or released (ex: U(A)). Only use exclusive locks, not shared locks.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>L(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>U(A)</td>
<td></td>
</tr>
<tr>
<td>U(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L(A)</td>
</tr>
<tr>
<td></td>
<td>L(B)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>U(A)</td>
</tr>
<tr>
<td></td>
<td>U(B)</td>
</tr>
</tbody>
</table>
Problem 3: Functional Dependencies (12 points total)

Assume the following functional dependencies hold on the relation R:

\[ R( A, B, C, D, E, F) \]

\[ A \rightarrow CD \]
\[ F \rightarrow AE \]
\[ D \rightarrow B \]

a) Given the above functional dependencies, write two non-trivial FDs that also hold on the relation. (Trivial FDs are the identity FDs such as \( A \rightarrow A \))

(Many possible answers)

\[ A \rightarrow BCD \] (2 points)

\[ F \rightarrow ACDE \] (2 points)

b) The closure \( \{X\}^+ \) is the set of all attributes determined by the attribute(s) \( X \). For example, with the above FDs, \( \{D\}^+ = \{BD\} \)

Write the closures of the following attributes:

\[ \{CD\}^+ = \) (2 points) \]

\[ \{A\}^+ = \) (2 points) \]

\[ \{AE\}^+ = \) (2 points) \]

\[ \{F\}^+ = \) (2 points) \]
Problem 4: Indexing and Query Optimization (10 points total)

Assume we have relations R and S in a database, with the following attributes. The primary keys are underlined.

\[
\begin{align*}
\text{R} & \quad (a, b) \\
\text{S} & \quad (c, d)
\end{align*}
\]

(In the paper copy R and S had primary keys specified, but a correction was made during exam time to remove those.)

Say we run the following SQL query on the above tables:

```sql
SELECT R.a, S.c
FROM R, S
WHERE R.a = S.d AND
      R.b = '1234'
```

In the following problem we will estimate the number of tuples (cardinality) at each step in the query plan.

Assume we have the following statistics on the tables R and S:

<table>
<thead>
<tr>
<th>Relation</th>
<th>( T(X) )</th>
<th>( V(X, a) )</th>
<th>( V(X, b) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>10,000</td>
<td>400</td>
<td>2,000</td>
</tr>
<tr>
<td>S</td>
<td>10,000</td>
<td>500</td>
<td>1,000</td>
</tr>
</tbody>
</table>

\( T(X) \) is the number of tuples in a relation \( X \).
\( V(X, y) \) is the number of distinct values for the attribute \( y \) in the relation \( X \).
Consider this relational algebra query plan for the above query:

Compute the expected number of tuples produced by each operator in the query plan. The first two are given for you as they are the number of tuples in the base relations.

(a) __10,000__ tuples__________

(b) __10,000__ tuples__________

(c) __5__ tuples________________ (4 points)

\[ T(C) = \frac{T(A)}{V(A, b)} = \frac{10,000}{2,000} = 5 \]

(d) __50__ tuples________________ (3 points)

\[ V(C, a) = V(A, a) \times \left( \frac{T(C)}{T(A)} \right) = 400 \times \left( \frac{5}{10,000} \right) = 0.2 \] Round up to 1.

\[ T(D) = \frac{T(C) \times T(B)}{\max\{ V(C, a), V(B, d) \}} = \frac{5 \times 10,000}{\max\{ 1, 1,000 \}} = 50 \]

(e) __50__ tuples________________ (3 points)

Projection does not change the number of tuples.
(Duplicate elimination \( \delta \) (or grouping by \( a \) and \( c \), \( \gamma_{ac} \)) is a different story.)
Problem 5: Parallel Databases (19 points total)

Say you are designing a parallel relational database to store data about films. You have three tables:

- **ACTOR** (pid, fname, lname)
- **MOVIE** (mid, name, year, revenue)
- **CASTS** (pid, mid, role)

Casts.pid is a foreign key reference to Actor.pid, and Casts.mid is a foreign key reference to Movie.mid. A tuple in Casts represents an actor with pid being cast in a movie with mid.

There is a large amount of data in all tables that would have to be spread between multiple machines to fit in the entire database. All tables have about the same number of tuples in them.

As the database designer, you know that the most common query that will be run on the system is:

```
SELECT x.fname, x.lname
FROM ACTOR x, CASTS c, MOVIE y
WHERE x.pid = c.pid AND
  c.mid = y.mid AND
  y.name = [some movie name]
```

a) For each table, describe in a sentence or two how you would partition the table between machines if your goal is to maximize performance of the above query. To maximize performance, you should avoid as much intermediate shuffling of tuples as possible. (9 points)

(Replicating the any table to each machine is impossible because the data is too large, as described above)

**Basic approach:**

**ACTOR table:**

Hash partition on Actor.pid. This is to match Casts.pid for the join between them. Now Actor and Casts can be joined without reshuffling Actor.

**MOVIE table:**

Hash partition on Movie.mid. This is to match Casts.mid for the join between them. Now Movie and Casts can be joined without reshuffling Movie.

**CASTS table:**

After joining Casts to either Actor or Movie, we will have to shuffle the intermediate result to join with the third table. Thus we should hash partition either on Casts.mid
or Casts.pid so that it can be joined to Actor or Movie. We can’t avoid doing a

**Advanced approach:**

An alternative approach is to hash partition on Movie.name. This allows quick selection of only the applicable movie, which should result in a single tuple. That tuple’s mid may then be broadcasted to all machines and used to select only those tuples in Casts that match the mid. Continue on to a join with Actor, which could be a distributed hash join.

A range partition on Movie.name is also effective, though not as good as a hash partition since the query template involves a point lookup.

b) Now instead of the relational database above, you are working with a NoSQL document store version of a movies database. The general structure of the data is that each key is the unique movie identifier mid, and the value for that key is a JSON object with key-value pairs representing the information of the movie with that mid. For example, one entry is the following (key, value) pair for a movie with mid=101:

```
(101, {
    "name": "The Third Man",
    "year": 1949,
    "revenue": 105209,
    "cast": [
        {
            "pid": 237
            "fname": "Alida"
            "lname": "Valli"
            "role": "Anna Schmidt"
        },
        {
            "pid": 937257
            "fname": "Orson"
            "lname": "Welles"
            "role": "Harry Lime"
        },
        ...
    ]
})
```

You can assume that all movies have only the key-value pairs above and that there are no null values.
Attributes in the JSON object can be referenced by their names. For example, in the above example of key 101 and value as the movie JSON object, `value.year` is the year field 1949. The revenue attribute is the total amount of money made by the movie, in US dollars.

We can also use the `count()` function on an array to return the number of entries in that array, and `sum()` on an array to sum the numerical values of the array.

For each of the following MapReduce programs, describe what it calculates. This description can either be a few sentences, or an equivalent SQL query on the relational movies database.

<table>
<thead>
<tr>
<th>1.</th>
<th>map(key, value):</th>
<th>reduce(key, values):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>emit(value.year, 1)</td>
<td>emit(key, count(values))</td>
</tr>
</tbody>
</table>

Description of Output: (5 points)

Return the count of movies for each year.

SELECT year, COUNT(*)
FROM Movie
GROUP BY year

<table>
<thead>
<tr>
<th>2.</th>
<th>map(key, value):</th>
<th>reduce(key, values):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>if (value.year &gt;= 2010):</td>
<td>emit(key, sum(values))</td>
</tr>
</tbody>
</table>

p.10
| emit(value.year, value.revenue) |

Description of Output: (5 points)

Return the total revenue for movies each year, for all years 2010 and later.

SELECT year, SUM(revenue)
FROM Movie
GROUP BY year
WHERE year >= 2010