Question A: SQL
(30 points)

Using the procedure outlined in class, translate the following query into a relational algebra tree or expression. Translate the query directly, without optimizing the resulting tree/expression.

```
SELECT partvalue.partkey, partvalue.supplyvalue
FROM (
  SELECT partkey, sum(supplycost * avail_qty) AS supplyvalue
  FROM partsupp ps, supplier s, nation n
  WHERE ps.suppkey = s.suppkey AND s.nationkey = n.nationkey
    AND n.name = 'ELBONIA'
  GROUP BY partkey
) partvalue,
(
  SELECT sum(supplycost * avail_qty) AS supplyvalue
  FROM partsupp ps, supplier s, nation n
  WHERE ps.suppkey = s.suppkey AND s.nationkey = n.nationkey
    AND n.name = 'ELBONIA'
) nationvalue
WHERE partvalue.supplyvalue > nationvalue.supplyvalue * 0.1
ORDER BY supplyvalue DESC
LIMIT 2;
```

There were three portions worth 10 points each. Minus 5 points if selection push down was used and minus 3 points for missing operators.

- Main projection, Minus 3 if Limit and Order By were switched.
- PARTVALUE subselect, contained group by (Many notations were accepted, must be different from NATIONVALUE). Minus 3 if group by operator is seperate from SUM and after SUM. No optimizations were used.
- NATIONVALUE subselect, no optimizations were used.

Question B: RA Equivalence
(40 points)

For each of the following pairs of *bag*-relational algebra expressions, either (1) prove that they are equivalent (using ra-equivalences), or (2) show a counter-example (input tables on which the expressions are not equivalent). Tables used have the following schemas:

$$R(A, B) \qquad S(C, D, E, F) \qquad T(C, D, E, F)$$

Part 1. $\pi_F(\sigma_{B=C \wedge D=E \wedge A=3}(R \times (\pi_{C,D}(S) \times \pi_{E,F}(T))))$
$\stackrel{?}{=}$ $\pi_F($ ($(\sigma_{A=3}(R)) \bowtie_{B=C} (\pi_{C,D}(S))$) $\bowtie_{D=E} (\pi_{E,F}(T))$) **(15 points)**.
Equivalent. Need to show: Selection pushdown, join conversion, join/cross associativity

Part 2. $\pi_A(\sigma_{A=C \wedge A=B \wedge D>3}(R \times (S \uplus T)))$
$\stackrel{?}{=}$ $\pi_A($ ($\sigma_{A=B}(R)$) $\bowtie_{A=C}$ ($\pi_C(S \uplus \sigma_{C>3}(T))$)) **(15 points)**.
Not equivalent, but sneakily so. Max 8 points if they try to prove equivalence. 10 points if they catch the conflict (selection pushes down both sides of a union) and list an S with a C ¿= 3.

Part 3. $(R \bowtie R) \stackrel{?}{=} R$ **(10 points)**.
Not equivalent. Counterexample: any R with two copies of the same row

Question C: Cost-Based Optimization
(30 points)

A database has gathered the statistics below regarding three tables in a restaurant reservation database.

Table	# of Rows	Key
Reservation (resv)	20,000,000	n/a
Customer (cust)	1,000,000	$\langle id \rangle$
Restaurant (rest)	10,000	$\langle id \rangle$

Field	# Distinct	Min	Max
resv.rest_id	1,000	1	1,000
resv.cust_id	10,000	1	1,000,000
resv.date	30	Jan 1	Jan 30
cust.id	1,000,000	1	1,000,000
cust.city	50	n/a	n/a
cust.name	800,000	n/a	n/a
rest.id	10,000	1	10,000
rest.offer_date	30	Jan 1	Jan 30

Where available, primary (clustered) indexes are built on the key columns. There are also secondary indexes built on `Customer.city`, `Reservation.cust_id` and `Reservation.rest_id`. No other information is available to the optimizer. Assume that enough memory is available for slightly more than 5,000 tuples (say 5,100 tuples), independent of relation.

Questions in this section pertain to the following query:

$\pi_{name,num}($
  Restaurant
  $\bowtie_{\texttt{rest.offer\_date<resv.date}}$
  (
    $\gamma_{\texttt{resv.date,count(*) AS num}}($
      Reservation
      $\bowtie_{\texttt{cust.id=resv.cust\_id}}$
      $\sigma_{\texttt{cust.city='Buffalo'}}($
        Customer
)
)
)
)

Part 1. Estimate the number of rows emitted by each operator in the plan above **(10 points)**.

Operator	Estimated Cardinality and Reason
$\sigma_{\texttt{cust.city='Buffalo'}}$	Assuming uniform distribution: $\frac{1{,}000{,}000}{50} = 200{,}000$.
$\bowtie_{\texttt{cust.id=resv.cust\_id}}$	Join with a key: # of rows == # of rows in resv = 20,000,000
$\gamma_{\texttt{resv.date,count(*) AS num}}$	Given exactly in the statistics: 30 tuples.
$\bowtie_{\texttt{rest.offer\_date<resv.date}}$	Assuming a uniform distribution across dates, 50% chance of passsing. Total number of tuples passed $= 10{,}000 \cdot 30 \cdot 0.5 = 150{,}000$

Part 2. For each of the operators listed in part 1, pick the algorithms that minimize the total the total IO required to execute the plan as a whole. *Justify your answer and be sure to state any assumptions you make.* **(20 points)**.

In general, 2 points were awarded for picking a good algorithm, and 3 points were awarded for a good justification. In a few cases, faulty assumptions led to wrong answers, in which case 3 points were awarded if the answer fit the assumptions.

1. $\sigma_{\texttt{cust.city='Buffalo'}}$: To optimize *the plan as a whole*, this operator may be evaluated as a secondary index scan, which reduces the number of tuples by a factor of $\frac{1}{50}$. 5/5 points were awarded for observing this. However, since this still means roughly 20000 tuples read from essentially random locations in the Customer table, the total number of IOs might still be comparable to a full table scan. Full credit was also awarded for making this observation.

2. $\bowtie_{\texttt{cust.id=resv.cust\_id}}$: In memory algorithms (e.g., 1-pass hash) were not acceptable for this, as neither input relation fits in memory. Candidate algorithms for this join included...

   - **Index Nested Loop Join**. This was the obvious choice, and a baseline 4/5 points were awarded to anyone who observed this and provided a reasonable justification (e.g., There's a 2ndary index on `rsrv.cust_id`). However, even filtered, there are an estimated 20,000 tuples on the other side of the join. That means a baseline of 20,000 index lookups (each of which adds a fixed number of IOs based on index type). It also means that each lookup requires either 2000 or 20 tuples worth of IO, depending on whether you estimated selectivity using $\frac{1}{\#UNIQ}$ or using $\frac{1}{Max-Min}$. The result would be somewhere on the order of either 400 million or 400 thousand tuples worth of IOs (plus 20k index lookups), respectively. In the former case, INLJ is outperformed by BNLJ, SMJ, and surprisingly 2P-HJ. In the latter case INLJ is optimal. Full points for observing this fact.
   - **Sort-Merge Join**. A common, although incorrect, argument in favor of sort merge join was that the index on `[cust_]id` meant that the data was in sorted order. This is not applicable here because the index on Reservation is a *secondary* index (i.e., the actual data is not stored in sorted order). Hence, at a minimum, Reservation needs to be sorted. A further concern is that using an index scan for the first operator may not emit results in sorted order, although the cost of sorting 20,000 records is comparatively small. Nevertheless, sorting 20 million tuples with roughly 5k tuples of memory available requires $\log_{5000} 20000000 = 2$ passes, or a total of $4 \cdot 20000000 = 80{,}000{,}000$ tuples worth of IO. This is still better than the worse estimate for INLJ. 2 points for choosing sort-merge join, and a full 5 for comparing it to any other join algorithm in terms of IOs.

- **Block Nested Loop Join**. With roughly 5k tuples worth of memory and customer as the outer table, there would be $\frac{20000}{5000} = 4$ outer blocks, or roughly 80,000,000 tuples worth of IOs as we first write, and then repeatedly read the contents of reservation. Going the other way, we'd have $\frac{20000000}{5000} = 4000$ blocks, or roughly the same amount. 2 points for choosing sort-merge join, and a full 5 for comparing it to any other join algorithm.
- **2-Pass Hash Join**. Under a pessimistic assumption for the performance of index-nested loop join, this was the best alternative, as it required only 40m+40k tuples worth of IO.

3. $\gamma_{\texttt{resv.date,count(*) AS num}}$: Full points for mentioning that, with 30 groups in the output, this operator could be evaluated in memory.

4. $\bowtie_{\texttt{rest.offer\_date<resv.date}}$: 0 points for trying to use an equi-join algorithm (Hash join, SMJ), or Index-Nested-Loop join. Block or regular nested loop join were preferred. Kudos to those who noticed that one side of the join fit entirely in memory.