Warmup #1  (Textbook Problem 12.3)

Let relations $r_1(A,B,C)$ and $r_2(C,D,E)$ have the following properties: $r_1$ has 20,000 tuples, $r_2$ has 45,000 tuples, 25 tuples of $r_1$ fit on one block, and 30 tuples of $r_2$ fit on one block. Estimate the number of block transfers required, using each of the following join strategies for $r_1 \Join r_2$:

Answer: $r_1$ needs 800 blocks, and $r_2$ needs 1500 blocks. Let us assume $M$ pages of memory. If $M > 800$, the join can easily be done in $1500 + 800$ disk accesses, using even plain nested-loop join. So we consider on the case where $M \leq 800$ pages.

1. Nested-loop join
   Using $r_1$ as the outer relation we need $20,000 \times 1,500 + 800 = 30,000,800$ disk accesses, if $r_2$ is the outer relation we need $45,000 \times 800 + 1,500 = 36,001,500$ disk accesses.

2. Block nested-loop join
   If $r_1$ is the outer relation, $\lceil \frac{800}{M-2} \rceil \times 1500 + 800$ disk accesses are needed. If $r_2$ is the outer relation, $\lceil \frac{1500}{M-2} \rceil \times 800 + 1500$ disk accesses are needed.

3. Merge join
   Assuming that $r_1$ and $r_2$ are not initially sorted on the join key, the total sorting cost inclusive of the output is $B_s = 1500(2\lceil \log_{M-1} 1500/M \rceil + 2) + 800(2\lceil \log_{M-1} (800/M) \rceil + 2)$ disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses.

4. Hash join
   We assume no overflow occurs. Since $r_1$ is smaller, we use it as the build relation and $r_2$ as the probe relation. If $M > 800/M$, i.e. no need for recursive partitioning, then the cost is $4(1500 + 800) = 6900$ disk accesses, else the cost is $2(1500 + 800)\lceil \log_{M-1} 1500 - 1 \rceil + 1500 + 800$.

Warmup #2  (Textbook Problem 12.6)

Consider the following bank database, where the primary keys are underlined:

- branch(branch_name, branch_city, assets)
- customer(customer_name, customer_street, customer_city)
- loan(loan_number, branch_name, amount)
- borrower(customer_name, loan_number)
- account(account_number, branch_name, balance)
- depositor(customer_name, account_number)

Suppose that a B+Tree index on branch_city is available on relation branch, and that no other index is available. List different ways to handle the following selections that involve negation:

Answer:
1. $\sigma_{\text{branch.city}<\text{"Brooklyn"}}(\text{branch})$
   
   Use the index to locate the first tuple whose branch.city field has value “Brooklyn.” From this tuple, follow the pointer chains till the end, retrieving all the tuples.

2. $\sigma_{\text{branch.city}='\text{Brooklyn}'}(\text{branch})$
   
   For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose branch.city field is anything other than “Brooklyn.”

3. $\sigma_{\text{branch.city}<\text{"Brooklyn"} \lor \text{assets}<5000}(\text{branch})$
   
   This query is equivalent to the query

   $\sigma_{\text{branch.city} \geq \text{"Brooklyn"} \land \text{assets}<5000}(\text{branch})$

   Using the index on branch.city, we can retrieve all tuples with branch.city value greater than or equal to “Brooklyn” by following the pointer chains from the first “Brooklyn” tuple. We also apply the additional criteria of assets < 5000 on every tuple.

---

Warmup #3 (Textbook Problem 13.4)

Consider the relations $r_1(A,B,C)$, $r_2(C,D,E)$, and $r_3(E,F)$, with primary keys $A$, $C$, and $E$, respectively. Assume that $r_1$ has 1000 tuples, $r_2$ has 1500 tuples, and $r_3$ has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$ and give an efficient strategy for computing the join.

Answer:

1. The relation resulting from the join of $r_1$, $r_2$, and $r_3$ will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining $r_1$ with $r_2$ will yield a relation of at most 1000 tuples, since $C$ is a key of $r_2$. Likewise, joining that result with $r_3$ will yield a relation of at most 1000 tuples because $E$ is a key of $r_3$. Therefore the final relation will have at most 1000 tuples.

2. An efficient strategy for computing this join would be to create an index on attribute $C$ for relation $r_2$ and on $E$ for $r_3$. Then for each tuple in $r_1$, we do the following:
   
   (a) Use the index for $r_2$ to look up at most one tuple which matches the $C$ value of $r_1$.
   
   (b) Use the created index on $E$ to look up in $r_3$ at most one tuple which matches the unique value of $E$ in $r_2$.

---

Warmup #4 (Textbook Problem 14.12)

List the ACID properties. Explain the usefulness of each.

Answer:

1. Atomicity. Either all operations of the transaction are reflected properly in the database, or none are.

2. Consistency. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

3. Isolation. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started or $T_j$ started execution after $T_i$ finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

4. Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
Problem 5 (To Be Graded)

Consider the following query on the account(aID, name) and deposit(aID, date, amount) relations:

```
SELECT a.name, d.date, d.amount
FROM deposit AS d
INNER JOIN account AS a
  ON a.aID = d.aID
WHERE amount >= 400
```

Assume that account contains 10,000 accounts, and every account has made 50 deposits on average (the deposit table contains 500,000 deposits total). Both relations are not sorted in any particular order, and there are no indices on the relations. The in-memory buffer can hold up to 12 blocks and there is 100 tuples blocks. Let $B = \frac{\text{num.entriesperleaf}}{2}$. Now let’s add some indices to these tables. First let’s add a primary index on deposit.amount. The expected number of tuples in deposit match the selection predicate is $500,000 \times \frac{500-400}{500-0} = 125,000$. The number of blocks needed to store these tuples is $N_s = \left\lceil \frac{125,000}{100} \right\rceil = 1,250$.

1. Using a Block Nested-Loop Join, compute the number of block accesses that will be required to perform the operation.

Assume the WHERE clause is evaluated when tuples are joined. Assume the outer relation is account. The number of block accesses is $M + \lceil \frac{M}{B} \rceil \times N = 100 + \lceil \frac{1000}{12} \rceil \times 5,000 = 50,100$.

Alternatively, assume WHERE clause is evaluated before tuples are joined. To find $N_s$ blocks it costs $N + N_s$ disk transfers. The total cost is $(N + N_s) + M + \lceil \frac{M}{B} \rceil \times N_s = (5,000 + 1,250) + 100 + 10 \times 1,250 = 18,850$.

2. Compute the block accesses again using a Merge-Join instead. Both the deposit and account relations remain unordered.

Assume the WHERE clause is evaluated when tuples are joined. The cost of sorting account and deposit is $(2M + 2M \times \lceil \log_B 1 M/B \rceil) + (2N + 2N \times \lceil \log_B 1 N/B \rceil) = 400 + 40,000 = 40,400$. The cost of joining the sorted account and deposit is $M + N = 100 + 5,000 = 5,100$. The total cost is $40,400 + 5,100 = 45,500$.

Alternatively, assume WHERE clause is evaluated before tuples are joined. To find $N_s$ blocks it costs $N + N_s$ disk transfers. The total cost is the sum of the costs of selection, merge-sort and join, or $(N + N_s) + (2M + 2M \times \lceil \log_B 1 M/B \rceil) + (2N_s + 2N_s \times \lceil \log_B 1 N/B \rceil) + (M + N_s) = (5,000 + 1,250) + (2 \times 100 + 2 \times 100 \times \lceil \log_{12} 100/12 \rceil) + (2 \times 1,250 + 2 \times 1,250 \times \lceil \log_{12} 1,250/12 \rceil) + (100 + 1,250) = 15,500$.

3. Now let’s add some indices to these tables. First let’s add a primary index on deposit.amount. The account relation remains unordered. Again, using a Block Nested-Loop Join, compute the number of block accesses that will be required to perform the operation. Assume that the WHERE clause is evaluated before the JOIN, with an index fan out of 100.

The data entry in the tree index contains a key and an actual tuple. The height of the tree is $H = \lceil \log_{\text{fanout}} 500,000/(\text{num.entriesperleaf}) \rceil = \lceil \log_{10} 500,000/100 \rceil = 2$. To evaluate WHERE clause, all $N_s$ blocks in deposit can be retrieved through a leaf-scan. The cost is $N_i = H + N_s = 1,252$.

The WHERE clause is evaluated before tuples are joined. The first way is to use account as the outer relation. We need to materialize $N_s$ blocks first. To find $N_s$ blocks it costs $N_i + N_s$ disk transfers. The total cost is $(N_i + N_s) + (M + \lceil M/(B-2) \rceil \times N_s) = (1,252 + 1,250) + (100 + \lceil 100/(12-2) \rceil) \times 1,250 = 15,102$.

An alternative way is to use deposit as the outer relation, which does not require materialization. The total cost is $N_i + \lceil N_s/(B-2) \rceil \times M = 1,252 + \lceil 1,250/(12-2) \rceil \times 100 = 13,752$. 
(Note: If the WHERE clause is evaluated when tuples are joined, the total cost is \( M + \lceil M/(B-2) \rceil \ast N_i = 100 + \lceil 100/(12-2) \rceil \ast 1,252 = 12,620. \) This is not the assumption of this problem, though.)

4. Assume that the primary index is now changed to \( \text{deposit.aID} \), rather than \( \text{deposit.amount} \). The \( \text{account} \) relation remains unordered. Using a **Indexed Nested-Loop Join**, compute the number of block accesses that will be required to perform the operation.

With the same tree index format assumption, the tree height is the same as above, \( H = 2 \). For \( \text{deposit} \), the expected cost of looking up a tuple through the tree index is \( C_s = H + 1 = 3 \). Assume \( \text{account} \) is the outer relation. The expected total cost of the join is \( M + |\text{deposit}| \ast C_s = 100 + 10,000 \ast 3 = 30,100 \).

5. A primary index is added on \( \text{account.aID} \). Now that both relations are sorted, recompute the number of block accesses using a **Merge-Join**.

Assume the WHERE clause is evaluated when tuples are joined. Because we have primary indices on both tables, a merge join costs \( M + N = 5,100 \).

Alternatively, assume WHERE clause is evaluated before tuples are joined. To find \( N_s \) blocks it costs \( \log_2 N + N_s \) disk transfers. The total cost is \( (\log_2 N + N_s) + M = ([\log_2 5,000] + 1,250) + 100 = 1,363 \).

**Problem 6 (To Be Graded)**

Consider the following transactions. Operations are to be executed in order from top to bottom.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(G)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>write(H)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>read(G)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>write(H)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(C)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(G)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(H)</td>
<td>-</td>
<td>write(B)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>write(B)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>write(C)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(A)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>write(D)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>read(B)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>write(E)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(D)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(E)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>read(H)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>write(C)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>read(B)</td>
</tr>
</tbody>
</table>

Answer:
Is this schedule conflict serializable? Prove your answer by building a precedence graph. If serializable, provide a possible serialized order of transactions.

Yes. Please see the attached figure. The possible serialized orders are

1. T1, T4, T5, T2, T3, T6.

2. T1, T4, T2, T5, T3, T6.