Solutions to 2004 final

1 Multiple Choice on Indexing and Query Processing(24)

1. True. Different orders of insertion could result in different splitting. Consider this leaf in a B+ tree where n=3. Say we want to insert 4 and 6.

Insert 4, then insert 6:

```
2 3 8
```

Insert 6, then insert 4:

```
2 3 4 6 8
```

2. False. Not true for queries that select a certain value. B+ tree works better than hash with range queries.

3. False. B is primary key of S, so \( V(S, B) = T(S) = 3000 \).

\[
T(R \bowtie S) = \frac{|R| \times \frac{V(S, B)}{|S|}}{\max (V(R, A), V(S, B))} = \frac{5000 \times 3000}{\max (V(R, A), V(S, B))} \leq \frac{5000 \times 3000}{3000} = 5000
\]

4. True. According to the definition of foreign key and primary key, for each tuple \( r \) in \( R \), there is a unique tuple \( s \) in \( S \) such that \( r.A = s.A \). So the size of natural join is the size of \( R \).

5. False. If the relation is not stored in an order on attribute \( A \), then a sparse first level index for \( A \) is useless. So two separate sparse first level indexes on different keys do not make sense unless the storage preserves order for both attributes.

6. True. The relation could be stored without preserving order of either attribute.

7. False. The first level should be dense in case the relation is not stored in order of the attribute. Also a dense second level index is useless.

8. False. Counter example:
Consider these two tables

<table>
<thead>
<tr>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROJECT₂E₁</th>
<th>PROJECT₂E₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \text{PROJECT}_2(E1 - E2) = \neq \text{(PROJECT}_2E₁ - \text{PROJECT}_2E₂) \]

2. Midterm problem revisited (80 points)
1.2. 3. (To simplify, here we omit the equivalent plans of exchanging the join order of the two lowest level relations)
4. 
select Nation: $B(\text{Nation}) = 1$
select Line: $B(\text{Line}) = 2 \times 10^7 \times 20/2000 = 2 \times 10^5$

Plan 3 is looking most promising:
. join Order, RI: $T(\cdot)^*(1+1) = 4000$
. join Customer, RI: $T(\cdot)^*(1+1) = 4000$
. join Nation, M: $2B(\cdot) = 4000 \times 52/2000 = 104$

Plan 1, 4 exceed Plan 1 because of sorting Customer.
Plan 2, 5 exceed Plan 1 because of reading Order.

3 Reducing SQL Queries to Bag Algebra, Part I (20 points)
1. SELECT department, SUM(salary) FROM Employee GROUP BY department

\[
\gamma_{\text{department}}, \sum \text{salary} \mid \text{Employee}
\]

or

\[
\gamma_{\text{department}} \mid \sum \text{salary} \mid \text{Employee}
\]

2. SELECT department, SUM(salary) FROM Employee GROUP BY department
HAVING AVERAGE(salary) > 100

\[
\pi_{\text{department}, \sum \sigma_{\text{Ave} > 100} (\gamma_{\text{department}}, \sum \text{salary}) \mid \text{Employee}}
\]

or

\[
\pi_{\text{department}, \sum} \mid \sigma_{\text{Ave} > 100} \mid \gamma_{\text{department}}, \sum \text{salary} \mid \sum \text{salary = Ave} \mid \text{Employee}
\]
3. SELECT department, SUM(salary) FROM Employee GROUP BY department HAVING AVERAGE(salary) > 100 AND COUNT(*) > 10

\[
\pi_{\text{department}}, \sum_{\text{salary}} \sigma_{\text{salary} > 100} \left( \gamma_{\text{department}}, \text{SUM(salary)}, \text{salary}, \text{COUNT(*)} \rightarrow \text{AC Employee} \right)
\]

\[
\pi_{\text{department}}, \sum
\]

\[
\sigma_{\text{salary} > 100} \rightarrow \text{AC}
\]

\[
\gamma_{\text{department}}, \text{SUM(salary)}, \text{salary}, \text{COUNT(*)} \rightarrow \text{AC Employee}
\]

4 Concurrency Control and Recovery (18 points)

1. Write a 2PL schedule that is not recoverable

\(T_2\) and \(T_3\) read \(a, b, c\), which \(T_1\) writes. Therefore to create a not recoverable schedule, we could simply put \(T_1\) before \(T_2 T_3\) and put \(\text{Commit}_1\) after \(\text{Commit}_2 \text{Commit}_3\).

\(T_1 T_2 T_3 \text{Commit}_2 \text{Commit}_3 \text{Commit}_1\)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Commit2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Commit3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Commit1</td>
</tr>
</tbody>
</table>

2. Write a 2PL and recoverable schedule that does not avoid cascading rollback.

We can make a small change to the previous schedule by moving \(\text{Commit}_1\) before \(\text{Commit}_2 \text{Commit}_3\). So \(T_2 T_3\) commit after \(T_1\) commits (recoverable) yet \(T_2 T_3\) read values written by \(T_1\), which is not committed yet (not ACR).

\(T_1 T_2 T_3 \text{Commit}_2 \text{Commit}_2 \text{Commit}_3\)
3. Write a strict 2PL schedule that is not serial.

\( r_1(a), \text{Commit}_2, T_1, \text{Commit}_1, T_3, \text{Commit}_3 \)

5 Concurrency Control (10 points)
In every serial schedule that is conflict equivalent to S, T2 precedes T1 =>

\[
T_1 \rightarrow T_3 \rightarrow T_2
\]

Therefore we need to insert an action of T3 in or before T1 (T1→T3) and an action of T3 in or after T2 (T3→T2) without generating other arrows:

\[
w_3(A)r_2(A)r_1(C)w_1(A)r_2(C)r_2(B)w_2(B)w_3(B)
\]

(Other options are replacing \( w_3(B) \) with \( r_3(B) \) or inserting \( r_3(A) \) anywhere before \( w_1(A) \) instead of inserting \( w_3(A) \))