Final Exam CS 5300

Name

**Question 1: Relational Algebra Equivalence**

To determine which expressions are equivalent to R–(S):

Given: R–(S)

**Correct Option: e. None of the above**

a. **(R)-S :** This is not equivalent because it selects from (R) first and then subtracts (S) .

b.**(R-S):** This is not equivalent because it subtracts ( S ) from ( R ) first and then applies the selection.

c. **(S-R) :** This is not equivalent because it subtracts ( R ) from ( S ) first and then applies the selection.

d. **(R)** - **(S)**: This is not equivalent because it applies the selection to both ( R ) and ( S ) separately and then subtracts the results.

e. **None of the above:** This is correct because none of the given options are equivalent to **(R)**.

**Question 2: Query Tree Construction**

Given SQL query:

SELECT P.Age, AVG(PP.Score) AS AverageScore

FROM PLAYERS AS P, GAMES AS G, PLAYER\_PERFORMANCE AS PP

WHERE P.PlayerID = PP.PlayerID AND

      G.GameID = PP.GameID AND

      G.GameName = 'Tug of War' AND

      PP.Status = 'Passed' AND

      P.Gender = 'Female'

GROUP BY P.Age

HAVING AVG(PP.Score) > 8.0;

### Steps to Construct the Query Tree

1. **Cross Product**: Combine the three tables (PLAYERS, GAMES, PLAYER\_PERFORMANCE).
2. **Selection**: Apply the selection predicates.
   1. P.PlayerID = PP.PlayerID
   2. G.GameID = PP.GameID
   3. G.GameName = 'Tug of War'
   4. PP.Status = 'Passed'
   5. P.Gender = 'Female'
3. **Projection**: Project the required columns.
4. **Grouping and Aggregation**: Group by P.Age and compute the average score.
5. **Selection on Aggregation**: Apply the HAVING clause to filter groups with AVG(PP.Score) > 8.0.

### Query Tree in Canonical Form

Here is the canonical query tree:

π\_{P.Age, AVG(PP.Score)}

  σ\_{AVG(PP.Score) > 8.0}

    γ\_{P.Age, AVG(PP.Score)}

      σ\_{P.Gender = 'Female' AND PP.Status = 'Passed' AND G.GameName = 'Tug of War' AND P.PlayerID = PP.PlayerID AND G.GameID = PP.GameID}

        (PLAYERS × GAMES × PLAYER\_PERFORMANCE)

**Visual Representation**

π\_{P.Age, AVG(PP.Score)}

  |

  σ\_{AVG(PP.Score) > 8.0}

    |

    γ\_{P.Age, AVG(PP.Score)}

      |

      σ\_{P.Gender = 'Female' AND PP.Status = 'Passed' AND G.GameName = 'Tug of War' AND P.PlayerID = PP.PlayerID AND G.GameID = PP.GameID}

        |

        (PLAYERS × GAMES × PLAYER\_PERFORMANCE)

In this tree:

* **π** stands for projection.
* **σ** stands for selection.
* **γ** stands for grouping and aggregation.

**Question 3: Optimized Query Tree**

**Heuristic Optimizations:**

**Push Selections Down:**

* 1. Apply selections as early as possible to reduce the size of intermediate relations.

**Join Order:**

* 1. Join smaller relations first to reduce the size of intermediate results.

**Combine Selections:**

* 1. Combine multiple selections into a single operation if possible.

**Optimized Query Tree:**

**Selections Pushed Down:**

* 1. σG.GameName=′TugofWar′(G)
  2. σPP.Status=′Passed′(PP)
  3. σP.Gender=′Female′(P)

**Optimized Join Order:**

* 1. σP.Gender=′Female′(P)⋈P.PlayerID=PP.PlayerIDσPP.Status=′Passed′(PP)
  2. Result⋈G.GameID=PP.GameIDσG.GameName=′TugofWar′(G)

**Grouping and Aggregation:**

* 1. Group by P.AgeP.AgeP.Age
  2. Aggregate AVG(PP.Score)
  3. Selection on the aggregated result: σAVG(PP.Score)>8.0​

**Optimized Query Tree:**

π\_{P.Age, AVG(PP.Score)}

  σ\_{AVG(PP.Score) > 8.0}

    γ\_{P.Age, AVG(PP.Score)}

      (σ\_{P.Gender = 'Female'}(P) ⨝\_{P.PlayerID = PP.PlayerID} σ\_{PP.Status = 'Passed'}(PP))

      ⨝\_{G.GameID = PP.GameID} σ\_{G.GameName = 'Tug of War'}(G)

**Question 4: Cost Estimation for Selection Operations**

**Given:**

- PLAYERS table: 2000 records, 100 disk blocks.

- Primary index on PlayerID: 2 levels.

- Clustering index on Gender: 4 levels, 3 distinct values.

- Bitmap index on Age: 2 disk blocks, 10 distinct values.

**OP1:** σPlayerID=007(PLAYERS)\sigma\_{PlayerID = 007}(PLAYERS)σPlayerID=007​(PLAYERS)

**Brute Force:**

- Scan all 100 blocks.

**Using Primary Index:**

- 2 I/O operations (index lookup) + 1 I/O (data retrieval) = 3 I/Os.

**Cost-effective method:** Using the primary index (3 I/Os).

**OP2:** σGender≠′Female′(PLAYERS)\sigma\_{Gender ≠ 'Female'}(PLAYERS)σGender=′Female′​(PLAYERS)

**Brute Force:**

- Scan all 100 blocks.

**Using Clustering Index:**

- Scan 4 levels (index lookup) + scanning non-'Female' records.

- Since there are 3 distinct values, about 2/3 \* 100 = 67 blocks.

**Cost-effective method:** Using the clustering index.

**OP3:** σGender=′Female′ AND Age∈(20,30,40)(PLAYERS)

**Brute Force:**

- Scan all 100 blocks.

**Using Clustering Index + Bitmap Index:**

- Clustering index: 4 levels (index lookup) + 1/3 of 100 blocks = 34 blocks for 'Female'.

- Bitmap index: 2 blocks for age + 34 blocks = 36 blocks.

**Cost-effective method:** Using the clustering index + bitmap index.

**Question 5: Join Cost Estimations**

**Given:**

- PLAYERS: 2000 records, 100 disk blocks.

- PLAYER\_PERFORMANCE: 5300 records, 300 disk blocks.

- 20 main memory buffers.

- Blocking factor: 37 records/block.

**a. Simple Nested-Loop Join (PLAYERS as outer)**

- Outer: PLAYERS (100 blocks).

- Inner: PLAYER\_PERFORMANCE (300 blocks).

- Cost: 100 + 100 \* 300 = 30100 I/Os.

**b. Simple Nested-Loop Join (PLAYER\_PERFORMANCE as outer)**

- Outer: PLAYER\_PERFORMANCE (300 blocks).

- Inner: PLAYERS (100 blocks).

- Cost: 300 + 300 \* 100 = 30300 I/Os.

**c. Index-Based Nested-Loop Join**

- Outer: PLAYERS (100 blocks).

- Inner: Indexed PLAYER\_PERFORMANCE.

- Index lookup: 2 (P.PlayerID) + 3 (PP composite key) = 5 I/Os per match.

- Cost: 100 + 2000 \* 5 = 10100 I/Os.

**d. Sort-Merge Join**

- Sort both relations on PlayerID (assuming pre-sorted).

- Cost: 100 + 300 = 400 I/Os.

**e. Partition-Hash Join**

- Partition both relations.

- Cost: Similar to sorting.

- Cost: 100 + 300 = 400 I/Os.

**f. Optimal Join Method**

Sort-Merge Join or Partition-Hash Join (400 I/Os).

**Question 6: False Statements on Multi-Relation Queries**

a. **FALSE** - The number of equivalent query trees grows rapidly as the number of join operators increases, not aggregation operators.

b. **FALSE** - A query that joins n relations will have at most (n-1) join operations, but the number of different join orders is much more complex than n!.

c. **TRUE** - A left-deep join tree is a binary tree in which the left child of each non-leaf node is always a base relation.

d. **TRUE** - It is easier to apply pipelining on a left/right-deep join tree than a bushy join tree.

e. **FALSE** - The order of joins is a significant concern for the database’s query optimizer.