Solution Report

1 Implementation Details

I implemented a **Sequential Hash Join** (Left-Deep) to evaluate the k-line join query. This approach builds the result incrementally by joining the accumulated result with the next relation in the sequence. The code can be found in `Solution.js`.

1.1 Algorithm Steps

The algorithm proceeds in steps to process a query involving k relations:

$$R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k$$

1. **Step 1**: Join R_1 and R_2 on attribute A_2.

   $$R_{1-2} = R_1 \bowtie_{A_2} R_2$$

   I used the Hash Join implementation from Problem 1: Hash R_2 on A_2, then probe with R_1.

2. **Step 2**: Join the intermediate result R_{1-2} with R_3 on attribute A_3.

   $$R_{1-3} = R_{1-2} \bowtie_{A_3} R_3$$

   Hash R_3 on A_3, then probe with R_{1-2}.

3. **Step** i: Join the intermediate result R_{1-i} with R_{i+1} on attribute A_{i+1}.

   $$R_{1-(i+1)} = R_{1-i} \bowtie_{A_{i+1}} R_{i+1}$$

This "left-deep" strategy means we always join the current intermediate result (which can be large) with a base relation.

2 Comparison with Yannakakis Algorithm

Unlike the Yannakakis algorithm (Problem 2), this approach **does not** guarantee global consistency before joining.

- **Intermediate Results**: We might generate intermediate tuples that are later discarded because they don't match with subsequent relations.
- **Complexity**: In the worst case, the size of intermediate results can be much larger than the input or output, potentially leading to exponential complexity $O(N^k)$ if the join graph is cyclic or if selectivity is poor. In contrast, Yannakakis guarantees $O(N + OUT)$.

3 Execution Results

I ran the sequential algorithm with the same datasets used in Problem 2.

Case 1: $k = 3$

- **Input**: 3 relations (same dataset as Problem 2).
- **Result**: Found **4 tuples**.
- **Output**: Matches the output from the Yannakakis implementation.

Case 2: $k = 5$

- **Input**: 5 relations (chain with noise).
- **Result**: Found **15 tuples**.
- **Output**: Matches the output from the Yannakakis implementation.

The sequential hash join correctly computes the result, but it constructs full intermediate tuples at each step.