Solution Report

1 Implementation Details

I implemented a generalized version of the **Yannakakis algorithm** to evaluate k-line join queries of the form:

$$q(A_1, \ldots, A_{k+1}) : -R_1(A_1, A_2), R_2(A_2, A_3), \ldots, R_k(A_k, A_{k+1})$$

The implementation in `Solution.js` handles arbitrary values of k (tested up to $k = 9$) using a dynamic approach rather than hardcoded steps.

1.1 Algorithm Steps

1. **Semijoin Reduction Phase**:
   To ensure **global consistency**, we perform two passes of semijoin reductions.

   - **Right-to-Left Pass**: For $i = k - 1$ down to 1, we reduce R_i by R_{i+1}:

     $$R'_i = R_i \ltimes R_{i+1}$$

     This propagates constraints from the end of the chain to the beginning.
   - **Left-to-Right Pass**: For $i = 2$ up to k, we reduce R_i by R_{i-1}:

     $$R''_i = R'_i \ltimes R'_{i-1}$$

     This propagates constraints from the beginning back to the end.

   After these two passes, all relations are fully reduced, meaning every tuple in every relation participates in at least one valid result tuple.

2. **Join Phase (DFS)**:
   We perform a Depth-First Search (DFS) to enumerate the results.

   - We start by iterating through the tuples of the first relation R_1.
   - For a tuple in R_i, we use a hash index to efficiently find matching tuples in R_{i+1}.
   - We recursively build the result tuple $(A_1, \ldots, A_{k+1})$.
   - Because of the reduction phase, we are guaranteed that every path in the DFS leads to a valid result (no dead ends).

2 Complexity Analysis

The algorithm maintains the optimal $O(N + OUT)$ time complexity for any constant k.

1. **Reduction Phase**:

   - We perform $2(k - 1)$ semijoin operations.

- Each semijoin $R \ltimes S$ takes $O(|R| + |S|)$ time using hash indices.
- Total time for reduction is proportional to the total size of all input relations: $O(N)$.

2. **Join Phase**:
   - The DFS traversal visits each result tuple exactly once.
   - Since the relations are globally consistent, there is no wasted work exploring invalid paths.
   - The time spent is proportional to the number of output tuples: $O(OUT)$.

Total Complexity: $O(N + OUT)$.

3 Execution Results

I ran the generalized algorithm with three test cases:

Case 1: k=3 (Original Example)

- **Input**: 3 relations with 5 tuples each (plus noise).
- **Result**: Found **4 tuples**.
- **Output Sample**:

Index	A1	A2	A3	A4
0	1	10	100	1000
1	2	20	200	2000
2	2	20	200	2001
3	4	40	400	4000

Case 2: k=5

- **Input**: 5 relations with 10 tuples each (generated chain with noise).
- **Result**: Found **15 tuples**.
- **Output Sample**: { `A1: 0, A2: 0, A3: 0, A4: 0, A5: 0, A6: 0` }

Case 3: k=9

- **Input**: 9 relations with 20 tuples each (generated chain with noise).
- **Result**: Found **30 tuples**.
- **Output Sample**: { `A1: 0, ..., A10: 0` }

The algorithm successfully scaled to handle longer chains, correctly filtering out dangling tuples and producing the expected join results.