Solution Report

1 Execution Instructions

Since a MySQL server is not available in the current environment, I have generated the necessary SQL scripts to run the experiment on any MySQL instance. The code to generate these scripts can be found in `Solution.js`.

1. `Solution_dataset.sql`: Contains the SQL commands to create tables `R1`, `R2`, `R3` and insert the "stress test" dataset generated in Problem 5.
2. `Solution_query.sql`: Contains the 3-line join query.

To execute, run the following commands in your terminal:

```
mysql -u username -p < Solution_dataset.sql
mysql -u username -p < Solution_query.sql
```

2 Performance Analysis

2.1 Verification

If executed, the MySQL query will return the same **1001 tuples** as the Yannakakis and Sequential implementations. Relational databases guarantee correctness for standard join operations.

2.2 Expected Performance

Is the running time closer to Yannakakis (Problem 2) or Sequential Join (Problem 3)?

The running time will be closer to **Yannakakis (Problem 2)**, i.e., very fast (∼milliseconds).

Why? Modern relational database management systems (RDBMS) like MySQL use a **Cost-Based Optimizer (CBO)**.

1. **Statistics**: The database maintains statistics about the distribution of values in columns (histograms).
2. **Plan Selection**:
   - The optimizer will analyze the query $R_1 \bowtie R_2 \bowtie R_3$.
   - It will estimate the selectivity of joining R_1 and R_2 (which produces a huge result) versus joining R_2 and R_3 (which produces a tiny result).
   - It will likely choose to join R_2 **and** R_3 **first** (or filter R_2 based on R_3's values if using a hash join variant), because that operation is highly selective and produces a small intermediate result.
   - Alternatively, if it uses a **Hash Join**, it might build a hash table on the smaller relation (R_3) and probe with R_2, effectively filtering R_2 early.

2.3 Conclusion

A naive "Left-Deep" plan (like Problem 3) is exactly what a good optimizer tries to **avoid** when it detects poor selectivity. By reordering the joins or using efficient filtering (similar to the semijoin reduction in Yannakakis), MySQL will avoid the "explosion" of intermediate results that plagued the Sequential Join implementation. Thus, its performance will be comparable to the optimized Yannakakis algorithm.