Introduction

Parallel join operations remain to be one of the most expensive operations during data analysis for large datasets. A novel algorithm known as SOJA is developed to minimize memory usage while maintaining equal or better performance compared to industry implementations. This report aims to demonstrate the workability of the SOJA algorithm and its performance as a Proof of Concept application.

High-Level Design (Methodology)

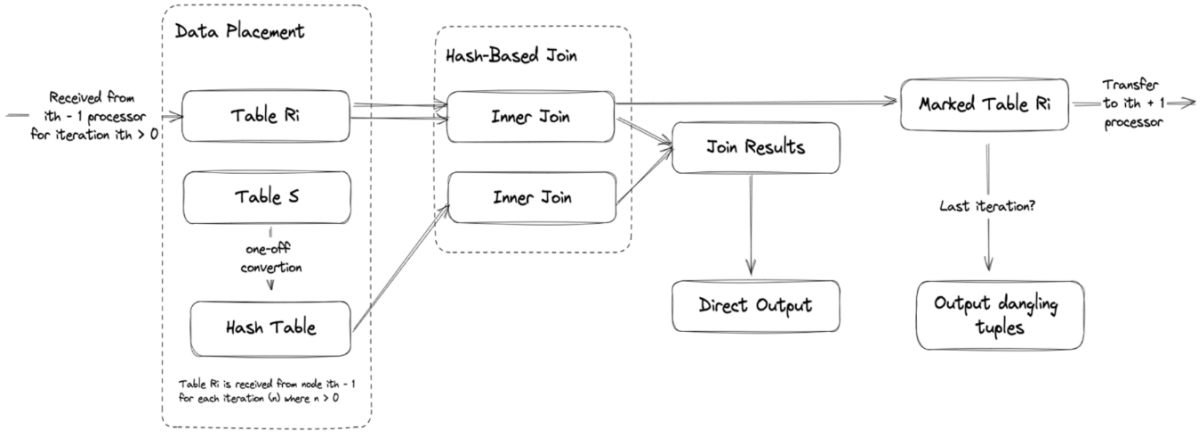

Figure 1. Architecture of SOJA algorithm.

On a high level, the SOJA algorithm is implemented according to the architecture above. The processors are arranged in a ring-topology fashion. The detailed step of the algorithm is as follows:

1. During the initial data placement (assumed round-robin), create a hash table (H) on the larger table (S_i) on each processor.
2. Execute a local hash join by looping over table R_i and performing a lookup on the hash table H. Mark dangling tuples (tuples with no join matches) in R_i. Return the intermediate join results as direct outputs.
3. Each process sends the marked table R_i to the next (i^{th} + 1) processor according to the ring topology. Note that the last processor will wrap around and send it to the first processor.
4. Repeat steps 2 and 3 N - 1 times where N is the number of processors.
5. Output marked tuples in table R_i as dangling tuples in the final iteration.

Code Implementation

One prerequisite needed for the SOJA algorithm (referenced in `soja.py`) to perform optimally is data needs to be allocated equally hence the code implemented a `roundrobin_partition()` function to process the data prior to running SOJA (lines 166-167). Additionally, a distributed processing environment is emulated by using the `multiprocessing` package to spin up multiple processors according to CPU count.

Ring Topology

The ring topology is implemented by using `multiprocessing` and queues in `soja()`. Line 174 initializes the queue data structure for each process. Then, the code initializes multiple processes through a worker target (a function which does the outer joins) with the current and next process's queue as one of their parameters (lines 180-192). This is done to set up inter-process communication without a dedicated channel. Lines 195 to 210 start the worker process, initializes the workers' queue with the respective data partition that is needed and wait until each process returns before marking the algorithm as completed.

Local Execution

This section is described through the `worker()` function implemented on line 104. As described earlier, the worker function is used as a dedicated process which does the outer joins. The local execution is implemented using a while loop which continuously polls the input queue for tasks (line 126). This action is blocking hence each process will wait until there's a task in the queue before proceeding.

On the first iteration, the worker will create the hash table for Table S through `create_hash_table()` (line 129) (Step 1). It will then run the `process()` function on line 143 to process the current table R and hash table (S). The `process()` function works by looping through each row in table R and doing a lookup in S to check if there's a match. It will also remove the row from being marked as dangling by removing it from the ID set (line 81) (Step 2). After getting the result and updated dangling tuples set, the algorithm will write the immediate results into a file to keep memory usage low (line 144) (Step 2). Then, the current process will send table R and the dangling tuple set to the next process by adding them to the ith + 1 process queue (line 146) (Step 3). This is repeated N - 1 times by keeping track of the number of iterations that the process goes through (Step 4). Upon reaching the last iteration, the block (lines 134-138) will execute to write the dangling tuples into the file and exit (Step 5).

Note: Specific implementation of each subfunction is documented in the code as docstrings.

Benchmarking

Some benchmarking code is included in the algorithm to measure execution time and memory usage. This is done by using the `tracemalloc` and `time` packages. Lines 170-171 initialize the benchmark test and lines 214-219 calculate the elapsed time and memory used.

Installation

The algorithm can be installed by cloning the git repository or through the zip file attached (requires unzipping). To run the algorithm, you need python3.9 and above installed. No libraries or dependencies are needed to be installed.

Code Execution

Run the following command to execute the algorithm.

```
python soja.py --R-file benchmark/source/movies.csv --S-file
benchmark/source/ratings_1000000.csv --concurrency-count 4
```

Optionally, you can specify the arguments to `--R-file`, `--S-file`, and `--concurrency-count`. Note that, for this POC, the algorithm expects the join attribute to be the first column in both CSV files and are both integers. The output of the parallel join operation can then be seen in `output-soja.csv`.

You can also run `make benchmark` to run the benchmark test (which may take some time).

Test results

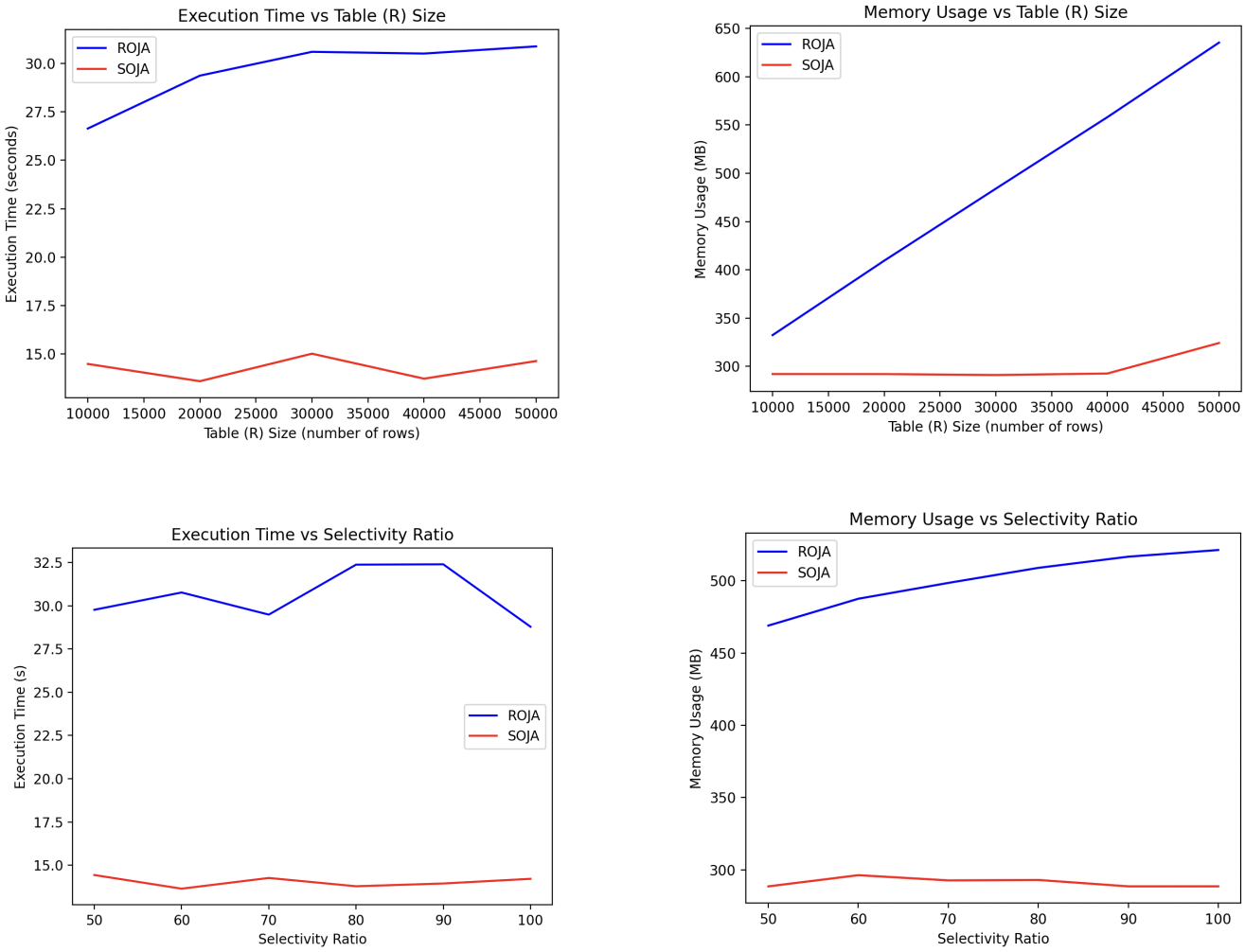

Figure 2. Benchmark Test Results between SOJA and ROJA.

Overall, the benchmark test executed against SOJA shows promising results in terms of the dimension investigated (table size & selectivity ratio). Both aspects outperformed the ROJA algorithm which is commonly used in the industry.

Conclusion

The SOJA algorithm offers a promising solution for optimizing parallel join operations in data analysis for large datasets. This report shows the overall architecture of the SOJA algorithm which is based on the concept of ring topology while minimizing communication costs. The report also describes the code implementation in detail and guides users on executing the algorithm.