Query Processor Project: Final Report

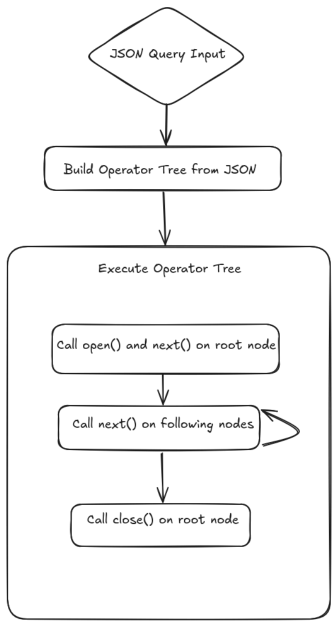
Wallace Schageman

High-Level Architecture Overview

This project includes a volcano-style iterator model query processor written in Python. This program processes JSON serialized queries from tables described in a .csv file format. This query processor is capable of Select, Project, Join, Scan, and Limit query operations. It is also capable of evaluating column references and aliases, constants, arithmetic operations, comparison operations, and logical operations. The code takes a modular approach, where the join, limit, project, scan, and select, are all subclasses of a base operation superclass allowing for further implementation of additional operations.

The system parses the JSON query input into a tree of operation objects using the python json library. Depending on what the operation is listed as in the JSON serialized query, a different operation object is created depending on the listed operation type in the query. For example, if the operation (“op”) is listed as “join”, a JoinOp operation object would be created. Each operation object is capable of open(), next(), and close() as detailed in the volcano iterator model.

After constructing the operator tree, the system executes the query by first calling open() and next() on the root node, and then by repeatedly calling next() (while loop) on the subsequent nodes while appending each result to a list of results. When all results have been compiled, the system calls close() on the root node and returns the list of results to be printed. A high level overview of the system can be visualized below.

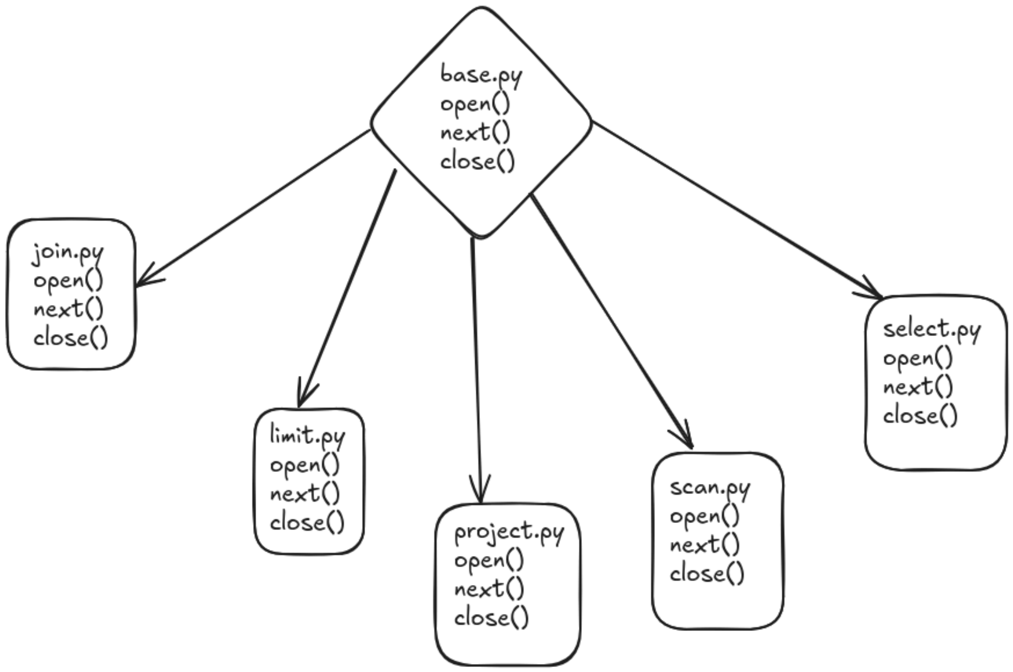

JSON Query Input
Build Operator Tree from JSON
Execute Operator Tree
Call open() and next() on root node
Call next() on following nodes
Call close() on root node

Critical Design Decisions

The first critical design decision I made was to use python instead of a low-level systems programming language like C or even C++. Python provided easier readability and allowed for fast prototyping. Python's json library also made parsing JSON-serialized queries very seamless. While the implementation was easy, Python may introduce performance tradeoffs with larger queries.

The second critical design decision that was made when designing this query processor was using in-memory processing for database queries instead of using disk to store intermediate results. This is done by running operations and sorting rows of tables in lists stored in memory within the next() computations within operation objects. This is especially relevant in the nested loop joins (default) and sort-merge joins, where one or both tables are stored in lists to be sorted and computed over. In-memory processing greatly increases performance across the board, especially for join operations.

The final critical design decision that was made was creating a modular, object-oriented design for all query operations. All operation objects (join, scan, limit, etc.) inherit from a common base operation object. This represents an "is a(n)" relationship. For example, a join operation "is an" operation. The base operation class includes the open() next(), and close() functions which are required to be implemented in every object. This design decision makes the system extremely modular, allowing new operations to be developed seamlessly. An overview of the class structure can be visualized below.



Additional Features

I implemented two additional join operation features, and one general optimization feature.

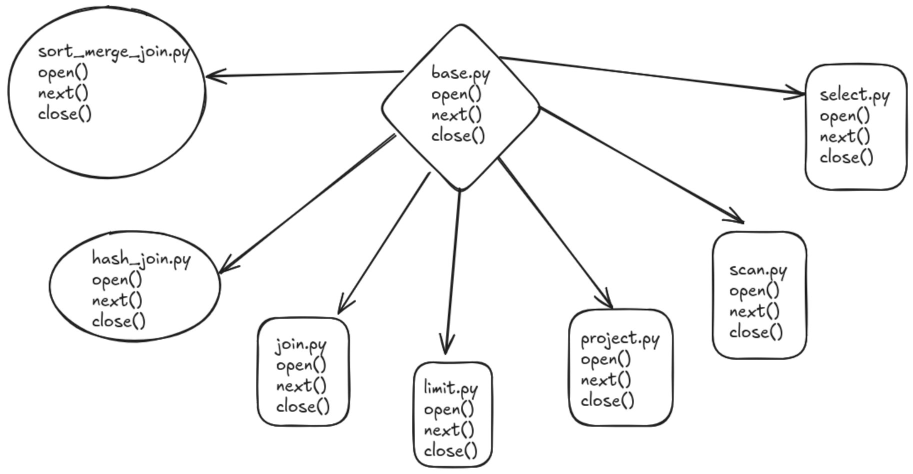
Join Operations

Hash Join

My hash join implementation offers significant efficiency for equi-joins on unsorted data, achieving average O(n + m) time complexity by building a hash table on the right input for fast lookups during left-side probing, making it ideal for large datasets where one table fits in memory. It handles duplicate keys and requires no pre-sorting or indexes. One of the drawbacks of this implementation is that the entire join must fit in-memory.

Sort-Merge Join

My sort-merge join implementation achieves O((n + m) log (n + m) + n + m) time complexity, making it superior to nested loops for big tables and beneficial when memory for hashing is limited. However, it requires loading both entire inputs into memory for sorting, risking out-of-memory errors on very large datasets without spilling, and incurs upfront sorting costs that can be prohibitive for small or unsorted data compared to hash joins.

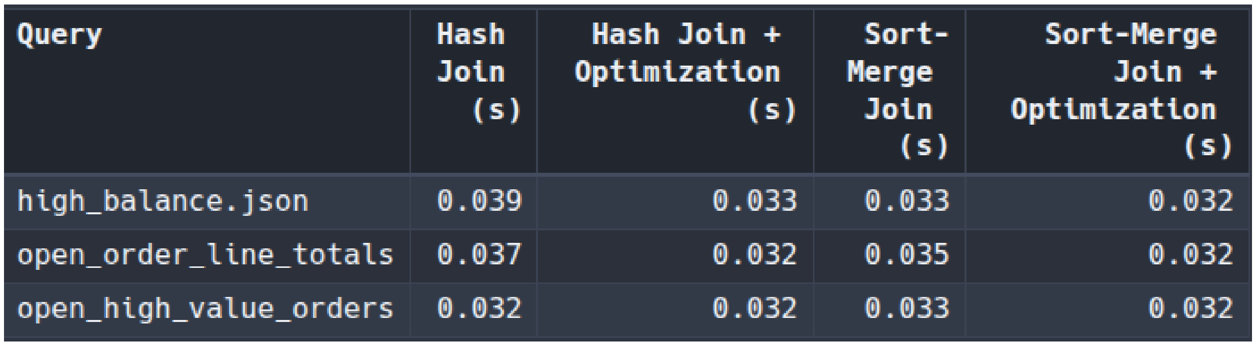


Predicate Pushdown Optimization

In order to filter data as early as possible, my predicate pushdown optimization moves Select operations closer to Scan nodes while traversing a query plan tree. It separates predicates for Joins to push conditions to the left or right input subtrees, and it rewrites and pushes predicates through Project nodes. By keeping predicates that cannot be moved below their current location, the recursive optimization maintains accuracy. By lowering intermediate result sizes early in processing, this increases query efficiency.

Benchmark Methodology

For testing, I used three of the queries included in the original example. Specifically, I used 'example1/query_high_balance.json', 'example2/query_open_order_line_totals.json', and 'example2/query_open_high_value_orders.json'. I used their respective included datasets as well. I had ChatGPT develop a handy benchmarking script for my use. The benchmarking script runs all three queries using hash joins and sort-merge joins. It does this with and without predicate pushdown optimization. It records the time to complete (s) for each test and outputs an easy to read table. Below is an example of a table generated using the benchmarking script.

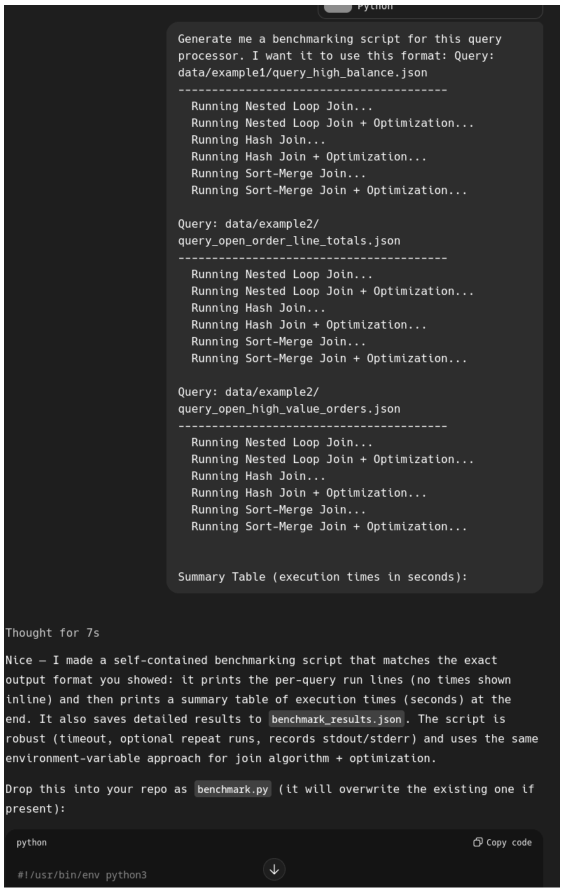


Query	Hash Join (s)	Hash Join + Optimization (s)	Sort-Merge Join (s)	Sort-Merge Join + Optimization (s)
high_balance.json	0.039	0.033	0.033	0.032
open_order_line_totals	0.037	0.032	0.035	0.032
open_high_value_orders	0.032	0.032	0.033	0.032

As expected, predicate-pushdown-optimized versions of both tests ran faster than without predicate pushdown optimization. I did find it strange, however, that sort-merge join ran noticeably slower than hash join. As described earlier, my hash join implementation theoretically runs in O(n + m) time, whereas sort-merge runs in O((n + m) log (n + m) + n + m) time. This is likely due to the small datasets and non-complex queries used in my experiments. Unfortunately, I did not have time to find or create new datasets and queries. For future work, I will test this system on extremely large datasets and queries to see how it performs.

Generative AI Reflection

Since I had no idea where to start this project, I asked ChatGPT to generate a folder structure and some boilerplate code for this project. The ideas for the modular approach came from ChatGPT. I filled in the rest of the details. I also had ChatGPT generate the benchmarking script, as well as generate most of the predicate pushdown and sort-merge join code. Here is the prompt I used to generate the benchmarking script.



```
Generate me a benchmarking script for this query
processor. I want it to use this format: Query:
data/example1/query_high_balance.json
----------------------------------------
  Running Nested Loop Join...
  Running Nested Loop Join + Optimization...
  Running Hash Join...
  Running Hash Join + Optimization...
  Running Sort-Merge Join...
  Running Sort-Merge Join + Optimization...

Query: data/example2/
query_open_order_line_totals.json
----------------------------------------
  Running Nested Loop Join...
  Running Nested Loop Join + Optimization...
  Running Hash Join...
  Running Hash Join + Optimization...
  Running Sort-Merge Join...
  Running Sort-Merge Join + Optimization...

Query: data/example2/
query_open_high_value_orders.json
----------------------------------------
  Running Nested Loop Join...
  Running Nested Loop Join + Optimization...
  Running Hash Join...
  Running Hash Join + Optimization...
  Running Sort-Merge Join...
  Running Sort-Merge Join + Optimization...


Summary Table (execution times in seconds):
```

Thought for 7s

Nice — I made a self-contained benchmarking script that matches the exact output format you showed: it prints the per-query run lines (no times shown inline) and then prints a summary table of execution times (seconds) at the end. It also saves detailed results to `benchmark_results.json`. The script is robust (timeout, optional repeat runs, records stdout/stderr) and uses the same environment-variable approach for join algorithm + optimization.

Drop this into your repo as `benchmark.py` (it will overwrite the existing one if present):

```python
#!/usr/bin/env python3
```

I usually don’t think AI is very helpful when it comes to programming, but in this case I found it very helpful. When I don’ t have AI actually creating code for the logic of the program, it works great. In this case, I mostly used it for bootstrapping and creating boilerplate template code.