6.830 Mid-term Report

Benchmarking In-Memory Joins

Amir Farhat David Morejon
amirf@mit.edu dmorejon@mit.edu

https://github.com/dmorejon/6.830FinalProject

Project Status

Changes from Proposal

After discussion with professor Madden, we've changed some core aspects of our project. We were interested in extending SimpleDB to include a benchmarking framework, a basis for us to implement and profile different join and sort algorithms. Since then, we've made three changes:

Firstly, we moved away from SimpleDB, opting instead for a system built ground-up in a low-level language, Rust, that we use to implement joins with more flexibility over representation, execution, and overhead. Secondly, we narrowed the design space to relational tables which 1) fit in memory, 2) contain only integers, and 3) participate in two-table equijoins. Thirdly, we chose to focus exclusively on join algorithms. The space of join algorithms and multicore exploitation over them makes for a no less interesting project and is more interesting to our team

Completed Tasks and Deliverables

Joins Algorithms

We focus only on in-memory equi-joins on one column in each table. We have implemented nested loops, block nested loops, and simple hash, where the hash is on the right, smaller table

Representation

Depending on where they are stored (disk vs program memory), our framework represents tables in two different ways. Tables on disk are stored as csv files, with a header naming each column, a comma separating column values, and a newline character separating rows. A table in memory is represented as a heap-allocated Rust vec containing stack-allocated Records, implemented as a fixed-size array of Rust i32s

Table Generator

We generate on-disk tables using two functions. The first function, *generate_table()* generates a table of size *num_rows * num_cols*, fills it with random i32s, and writes the table to disk. This is the "left" or "outer" table of the join. The second function, *generate_right_table()* produces the "right" or "inner" table of the join, and is more involved

It takes an already-constructed *left_table*, *num_rows*, *num_cols*, *join_selectivity*, and the columns to join on. We construct the right table in steps:

1) Create the right table entirely randomly as above
2) Then, to satisfy *join_selectivity*,
    a) Assign the right table's first *matches = (left_table.num_rows * join_selectivity)* columns in its join column to the same values as the left table's first *matches* columns on its join column, and
    b) Assign the remaining right table join column values to values that are not at all in the left table's join column. The Missing Value Picker (MVP) assists in value generation
3) Finally we shuffle the right table's rows

Note that *join_selectivity* is not actually a perfect indication that the join will have true selectivity *join_selectivity*, but it is a useful parameter that changes the join's selectivity in a relatively consistent way. We can verify the true selectivity by running the join and measuring selectivity after the fact

Missing Value Picker (MVP)

The MVP, initialized with a collection of integers, exposes a *next()* method which is guaranteed to produce values which are: 1) neither in the input integers, nor 2) repeated from previous calls to *next()*

Testing

We wrote unit tests for our structs. Additionally, we wrote end-to-end tests that verify end-to-end correctness of join algorithms on small tables using nested loops. We test the other joins by asserting that their result matches the nested loops result. Since they are not ordered, we sort the join results record-by-record and then rely on vector equality for checking correctness. We test block nested loops with an extensive set of different block size pairs

Open Tasks and Deliverables

More Advanced Joins

Our current plan for the next set of joins to implement is leapfrog trie join and radix join, as suggested by the staff in the proposal feedback. Depending on the results of those, we might continue implementing more unique joins or move our focus to multicore implementations of our current selection of joins in order to profile how much parallelism (SIMD or true core parallelism) we can squeeze out of them

Multicore Exploitation

We have preliminary ideas for squeezing more join performance by leveraging multiple cores and advanced CPU functions. Parallel multicore iteration in Rust via the rayon crate is an example of the former, while the slipstream crate exposes vectorized CPU operations for idiomatic Rust invocation

Cache Flushing, Hits, and Branch Misses
To complement our currently single metric for inspecting join performance, execution time, we will implement and verify more avenues: 1) flushing L* caches before join execution to simulate an empty cache, and 2) measuring cache hits to determine how much each join exploits CPU architecture, and 3) measuring branch misses to quantify how much each join utilizes the CPU's branch prediction and pipelining of predicted branches' instructions

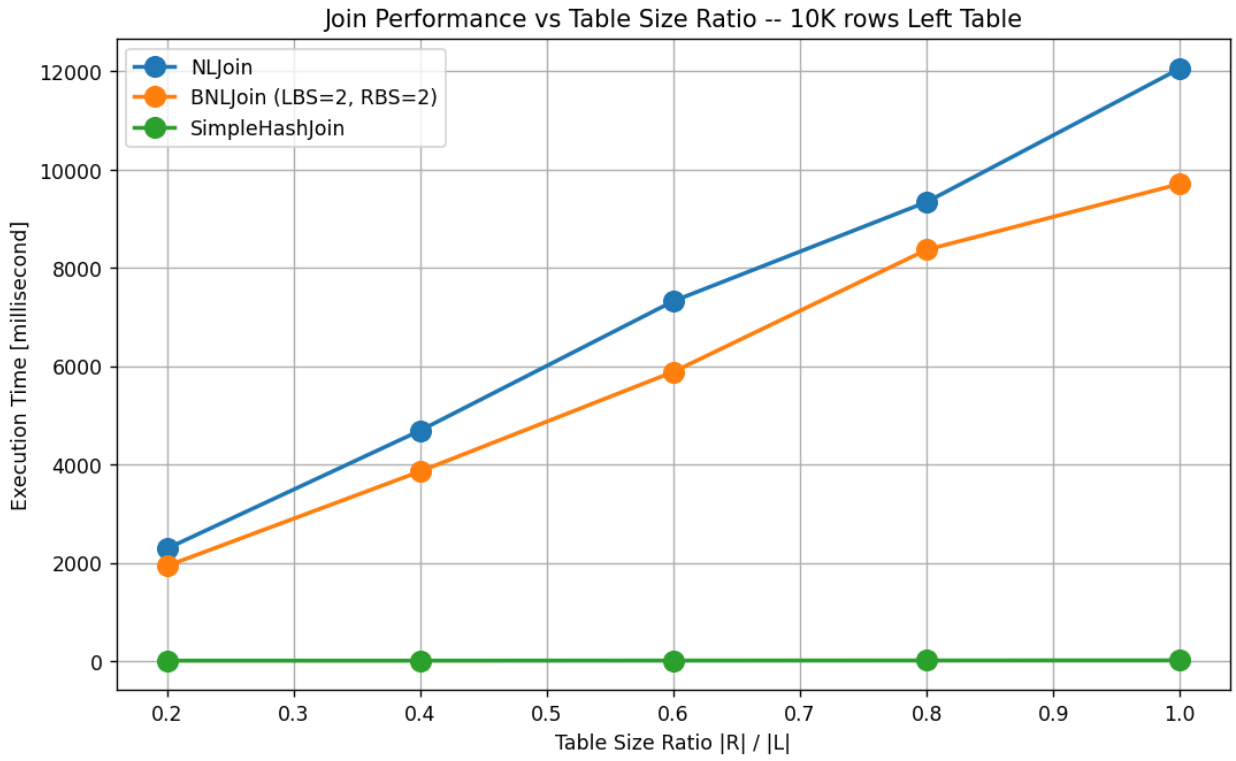
Interactive Graphs
As we measure more performance data, include more joins, and evaluate more tables, our data will have more dimensions. For ease of visualization, we will convert our currently static plots to interactive ones

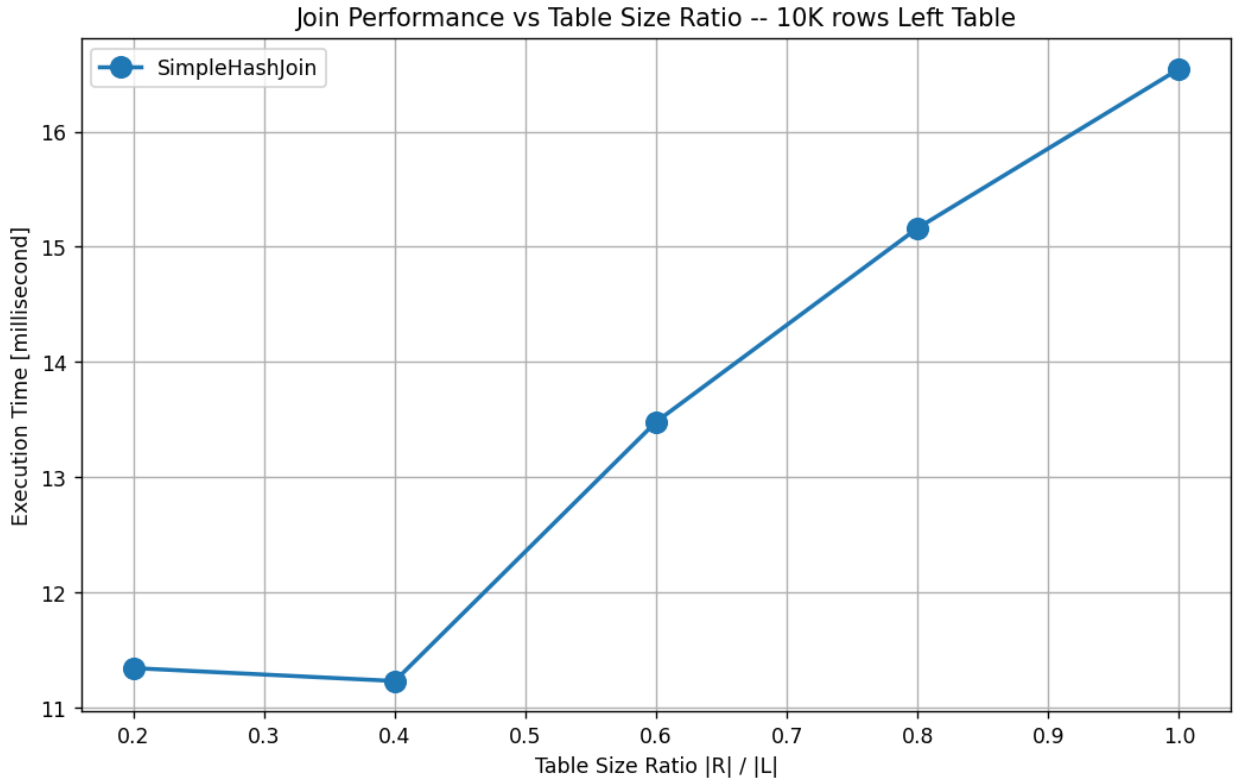
Increase Trial Runs
Currently we run one trial for each join-table1-table2 combination, but we will run multiple trials and plot interesting graphics on those collections, like CDF, slowest, fastest, and median

Clean Slate Machine
In order to get results that are as deterministic as possible, we want to run these join experiments on a dedicated machine that is not any of our laptops

Results

Performance Graphs
We compare the performance of our joins by running them on a series of five right tables and one left table. We plot performance as a function of the ratio of the cardinality of the right table over the left table. Both tables have a fixed number of columns


Join Performance vs Table Size Ratio -- 10K rows Left Table
NLJoin
BNLJoin (LBS=2, RBS=2)
SimpleHashJoin
Execution Time [millisecond]
Table Size Ratio |R| / |L|
0
2000
4000
6000
8000
10000
12000
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0


Join Performance vs Table Size Ratio -- 10K rows Left Table
SimpleHashJoin
Execution Time [millisecond]
Table Size Ratio |R| / |L|
11
12
13
14
15
16
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Problems

New to Rust

One of our biggest problems is that none of us have used Rust before, and writing high-performance system code is hard in a new language. We spent a long time learning about the Rust build system, different primitive types, mutability, ownership, and generic types

Table Generation & MVP

The MVP currently will panic on i32 overflow. This is not likely since overflow is achieved with a table of ~ 4.3 billion unique rows, while we don't exceed one million rows

Updated Timeline

- May 1-4: New joins
- May 1-9: Cache + branch stats
- May 4-9: Multicore
- May 9-13: Interactive graphing
- May 9-13: Run on dedicated machine & collect data
- May 13-18: Final Report + Presentation
- May 19: Presentations

Work Breakdown

Amir

- Design / representations / project infrastructure
- Runner configuration / CLI
- Table generator
- Unit tests
- Graphing Infrastructure

David

- Design / representations / project infrastructure
- NL, BNL, SimpleHash joins
- Table generator
- Integration tests

Lots of peer programming on Zoom and VSCode Live Share 😊

Obsolete Record Field Generics

Before our simplifying assumption of solely i32 record fields, we attempted an approach with Rust generics, so we could support many different primitive types. These include but are not limited to variously-sized signed integers, variously-sized unsigned integers, variously-sized floats, and strings. However, the Rust type system made this a challenging task. The difficulty

was further compounded by our noob status in Rust. Continuing support for generic types would have meant redesigning our entire representation and complication of simple tasks within the codebase

Questions

In-memory Table Representation

How shall we represent in-memory tables for performance? Currently Tables are on the heap while Records are on the stack. Is this reasonable? What alternatives exist?

Joins

Our open deliverables are leaning towards leapfrog trie and radix joins, then multi-core implementations. Does this seem like a reasonable plan? Any other joins we should consider first?

Graphing Tradeoffs

We fix a single left table and vary the right table to grow in size until it is of the same size as the left. What are more valuable graph formats?