6.830 Final Report
Benchmarking In-Memory Equi-Joins in Rust

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

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

Abstract

In this work, we design and implement a micro benchmarking framework to evaluate different join algorithms. We restrict ourselves to in-memory equi-joins of two tables that consist of signed 32 bit integers. In running different join algorithms using Rust on a powerful multicore machine, we found that, when compared with traditional algorithms and parallel radix join, leapfrog trie join emerges as the best jack of all trades

Introduction & Related Work

DB systems, On-disk vs In-memory

Join algorithms are a core part of any DBMS. Often, they are developed to squeeze as much performance as possible out of these systems. DBMSs which focus on data persistence with big tables will store tables on disk, and join algorithms must account for this. However, DBMSs which store smaller tables can fit in RAM. This changes the problem of joining tables drastically. Optimizing the join algorithm, then, focuses not on disk operations, but instead on reducing the total number of CPU operations, maximizing parallelism, and optimizing cache usage.

Our work focuses on in-memory equi-join algorithms on tables containing 32-bit signed integers. Understanding the intricacies of optimal join algorithms in a restricted setting allows us to extract useful insights about the inner workings and design considerations of join algorithms for specialized DB systems

Table Join Algorithms

Traditional join algorithms like nested loops, block nested loops, and simple hash were built accounting for the disk DBMS. Modern, multicore CPUs present a space for parallelization of these algorithms. One step further is join algorithms specifically designed for fast in-memory operations, like radix join and leapfrog trie join. We consider the above algorithms and parallel variants of them where possible

Profiling papers

Idreos et al's MonetDB we read in class discusses the importance of cache-awareness and leveraging SIMD CPU instructions. This work profiles both radix and sort-merge on multi-thousand core machines, giving interesting analysis on throughput, cache statistics, and execution time breakdown. Madden et al's VoltDB and other in-memory databases consider problem of joining tables similarly to the aforementioned

Rust

To comprehensively benchmark the above join algorithms, we found ourselves needing a high performance, high control language, with support for correctness in the face of parallelism. Our search led us to Rust, a low-level systems programming language designed to give performance similar to C and C++ but with focus on correctness. Rust's model of variable-to-value ownership is both restrictive and strict. However, it opens the way for performance optimizations, compile time alleviation of memory management bugs, and most critically, increased correctness in the face of parallelism. We did not know Rust at the outset and are happy to report that despite the unimaginably difficult ramp-up, we are now proud Rustaceans. In other words, *println!("i am rusty");*

Design

We implement a micro benchmarking framework to aid in the development and profiling of join algorithms. In particular, it is tailored for testing in-memory equi-join algorithms on variously-sized tables made up strictly of signed 32 bit integers. Below, we outline the components of the framework. Following, we discuss the choice and design of the join algorithms

Benchmarking Framework

The benchmarking framework is composed of table generation logic with configurable selectivity, different join run options exposed via a CLI, and join runtime collection with plotting in an IPython Notebook

Table Generator

Tables are generated in-memory then written to disk using the csv format due to ease of import/export into/from programs. We generate on-disk tables using two phases. While the first, generating the left table, is more straightforward, the second, which generates the right table, is less so. It uses awareness of the left table to achieve a target selectivity when the join between the left and right tables eventually runs

The first phase creates the "left/outer" table of the join via *generate_table(num_rows, num_cols).* It generates a single table of size num_rows * num_cols, populated with random i32s and is then written out to disk

The second phase, which is more involved, creates a number of "right/inner" tables with awareness of the left table using repeated calls to *generate_right_table(left_table, num_rows, num_cols, join_selectivity, left_col, right_col)*. Each call implements the following algorithm

1) Create the right table entirely randomly as above
2) Then, to satisfy *join_selectivity*,
    a) Let *matches* = *(left_table.num_rows * join_selectivity)* be the number of left table column values we expect to pass the join selectivity. Then,
    b) Assign the right table's first columns in its join column to the same values as the left table's first *matches* columns on its join column, and
    c) Assign the remaining right table join column values to ones that are entirely missing from the left table's join column. The Missing Value Picker (MVP), given the left join column, assists in finding these values
3) Shuffle the right table's rows
4) Write the right table out to disk

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().* We originally implemented the MVP such that *next()* returns the current value of an incrementer, starting from i32::MIN to i32::MAX which skips values it's seen. We observed that this led to data skew in partition and hash based join algorithms. Consequently, we changed the implementation to sample integers uniformly at random from the entire i32 range until a value not seen before is generated. Though this approach likely leads to collisions, the i32 range is on the order of 4e9, while our maximum table size is of order 1e7

Runner & Profiler
We wrote a CLI which exposes a program that executes a sequence of user-specified join algorithms on a set of table pairs. This program invokes our implementations of the join algorithms mentioned in the paper, computes the execution time, and later verifies correctness of the join result as a fail-safe for bugs. Before the join algorithm executes, the program flushes the L1 cache by writing a dummy value many times to an array. Due to the variability of execution times, the user must specify the number of trials to run each join.

The CLI passes the user arguments to the program. Upon completion of the joins, the program writes the execution time output to a JSON file containing information about each run, with schema similar to below

```
{
 join_type:{
   join_name:String,
   left_block_size:usize,
   right_block_size:usize,

 },
 outer_table:{
   table_name:String,
   num_records:usize,
   columns_per_record:usize,

 },
 inner_table:{
   table_name:String,
   num_records:usize,
   columns_per_record:usize,

 },
 execution_time_nanos:u128,
 num_emitted_records:usize,
 trial_number:i8
}
```

Notebook
We employ an IPython Notebook based on a list of the above output JSON to group joins on similar left-right table pairs and compute the execution time as the right table grows in size while the left remains constant. We use three graphs: 1) parameter sweeps of block sizes for

block-nested loops, on small tables, 2) “traditional” join algorithm performance on small tables, and 3) “novel/advanced” algorithms’ performance on increasingly large tables. The graphs are in the evaluation section

Join Algorithms

We implemented a mix of “traditional” join algorithms like nested loops, block nested loops, and simple hash. Additionally, we attempted parallelizable versions of these to measure how much parallelism contributes to reducing theoretical complexity. Finally, we implemented two “novel” algorithms, radix join and leapfrog trie join on two tables. The figure below highlights the algorithms, complexity, and optimizations we implemented for a join between two tables L and R.

Join Name	Algorithmic Complexity	Parallel	Cache-aware
Nested Loops	O(L*R)	N	N
Block Nested Loops	O(L*R)	N	Y
Simple Hash	O(L+R)	N	N
Parallel Nested Loops	O(L*R)	Y	N
Parallel Simple Hash	O(L+R)	Y	N
Radix	O(L+R)	Y	Y
Leapfrog	O(L*log(L) + R*log(R))	Y	N

Nested Loops Join

We first implemented a basic nested loops join. After completing our serial algorithms, we moved to parallelization. Initially, we had a parallel *forEach* through each element of the left table. The results were surprisingly slow. Eventually, we tracked down that there was significant synchronization overhead. We moved to parallel *chunks*, giving each thread a large chunk of the left table’s records to join with the whole right table, like a modified parallel block nested loop join. This turned out to be much faster

Block Nested Loops Join

Our second algorithm was block nested loops. The only interesting aspect of this algorithm is choosing block sizes. We tried the cartesian product of many different block sizes (the best of which are shown in the evaluation section)

Simple Hash Join

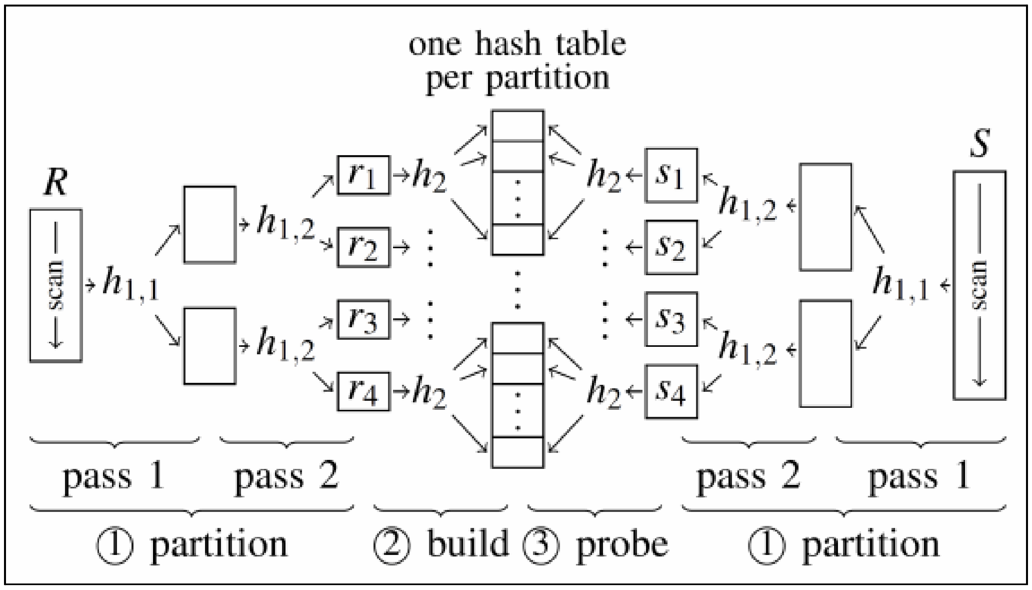
The last algorithm from class was simple hash. We iterate through the smaller table and build a hash table on the join attribute. The hash table is a map from *i32* to *Vec<Record>*, containing all Records that have the key attribute. As we iterate through the left table, we simply probe the hash table on the left join attribute and merge the record with each element in the table's value. This significantly outperformed the previous algorithms' performance and took some work to beat

Parallelization could target one or both of the stages of hash join: build and probe. Considering that the build phase touches only the right table (smaller table) and that concurrently populating a hash table likely harms performance, we chose not to parallelize the build phase. However, we can easily parallelize the probing phase. Initially we did a parallel *forEach*, but like nested loops, we found there was less synchronization overhead by using *chunks* instead.

Radix Join

For our first advanced algorithm, we implemented radix join. Radix join is similar to hash join, but differs in its aim to achieve fast join performance by focusing on data locality rather than the total number of CPU operations. The idea is to divide both tables into small corresponding partitions, then build a hash table on the smaller of each pair of partitions (that ideally both fit in memory). Then, each core can load a separate partition, each of which are small and should fit in the cache. The overall join result is the concatenation of each partition's join

Partitioning is done by some 'radix' of the join attribute, some function of the bits of a record's join attribute. We employ two radix hash functions, *h1,1(x)* and *h1,2(x)*, that deterministically partition the tables. Both functions are applied to each table in the same order so that pairs of partitions are guaranteed to hold all possible records that can be joined. The diagram below outlines the general architecture we followed



https://db.in.tum.de/teaching/ws1819/seminarHauptspeicherdbs/slides/05-main-memory-hash-joins.pdf?lang=de

The literature did not contain concrete implementation details, so we had to make some design choices. We will discuss the overall algorithm before mentioning the different decisions. The most obvious is the choice of *h1,1* and *h1,2*. Without even worrying about the internals of these functions, the number of distinct output values determines the number of partitions at each stage. This is absolutely critical to radix join's performance. A bad choice of *h1,1* and *h1,2* with 2 distinct values each would only split the data into 4 large partitions, effectively losing all locality benefits. We found that 32 partitions at each stage ($2^5 * 2^5$ total) worked well. *h1,1(x)* returns the lowest 5 bits of x and *h1,2(x)* returns the second lowest 5 bits of x

In considering how to parallelize radix join, a simple starting place is to execute the partitioning stages in parallel. For the join phase, since each partition pair is independent, those can all be run in parallel as well, aggregating the results at the end

Parallel Two-table-unary-Leapfrog-trie Join
(or parallel leapfrog for short)

We based our implementation on ideas from https://arxiv.org/abs/1210.0481. This paper is far more explicit in implementation details than any of the radix resources, but still requires some modification to fit our use case. First, leapfrog trie join is used to merge *n* tables at once, which is far different from most of the joins and query plans we studied in class that assume a tree-like structure. Our use case is *two-table* only. Second, leapfrog trie join allows for multiple join predicates. Our use case is *unary* predicates only. Thirdly, leapfrog assumes sorted input relations. Since none of our other algorithms rely on this assumption, we treat sorting time as part of this algorithm's cost. After the relations are sorted, we start at the minimum join attribute value and "jump" (or leap) over consecutive runs of values that do not match, similar to a sort-merge join. We found no easy way to parallelize the merge step, but we perform a parallel sort of each table concurrently

Rust Implementation Details

We highlight various Rust features that we exploited to optimize our algorithms

First, and easiest, was building the *--release* version of our code, a way of building for production, which immediately improved our results by **two orders of magnitude**. Secondly, we initially attempted to use generics in order to support record columns of any type, aiming to explore how different data affected performance. Rust's trait-based polymorphism made this very difficult, so we had to narrow our scope to tables of *i32s*. This likely made our algorithms much faster than if we had used arbitrary length types like String. Thirdly, Rust *requires* declaration of mutability/immutability on all variables. Because of this, we can safely run concurrent operations on immutable data. For example, as we probe the hash table in parallel simple hash join, there are no concerns about data races because the compiler guarantees and enforces that the data is immutable. What enables the compiler to do this is Rust's ownership model

Rust's ownership model is different from that of, say, C. By enforcing that mutable variables have at most one owner, the Rust compiler can enforce at most one writer on variables at compile time. The compiler also allows multiple readers of strictly immutable data. In particular, given an immutable variable, the read operation in Rust is called a "borrow", which is a reference to an immutable data value. Similarly, a write in Rust is a "move"

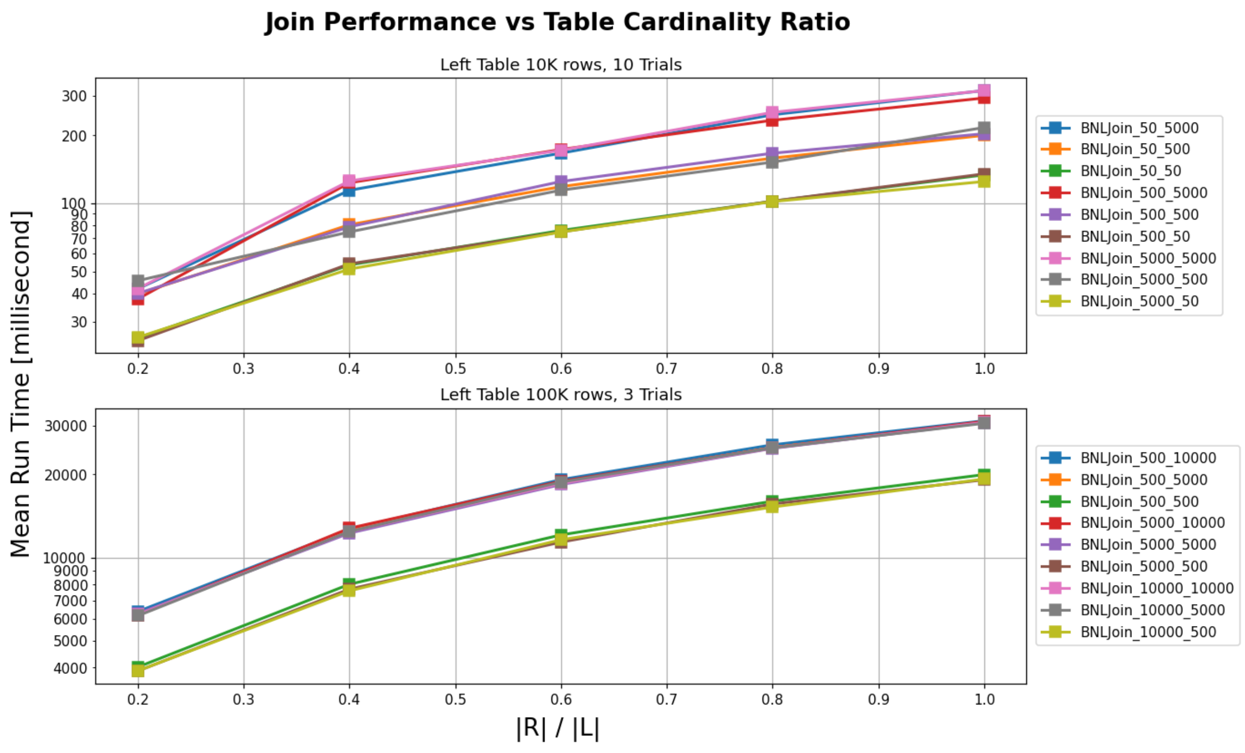
The Rayon concurrency package depends on these properties of data to make concurrency easy. We refactored our serial algorithms to use Rust iterators over the data, rather than our custom table iterators. From there, Rayon allows us to substitute the original Rust iterator with a parallel iterator that automatically assigns threads from a fixed global thread pool to different elements in the iterator so that computation happens in parallel. We can use *map*, *forEach*, and other functional programming primitives to specify what needs to happen for each record, followed by a combination step which *collect*s merged the results into a single output vector

Evaluation & Discussion

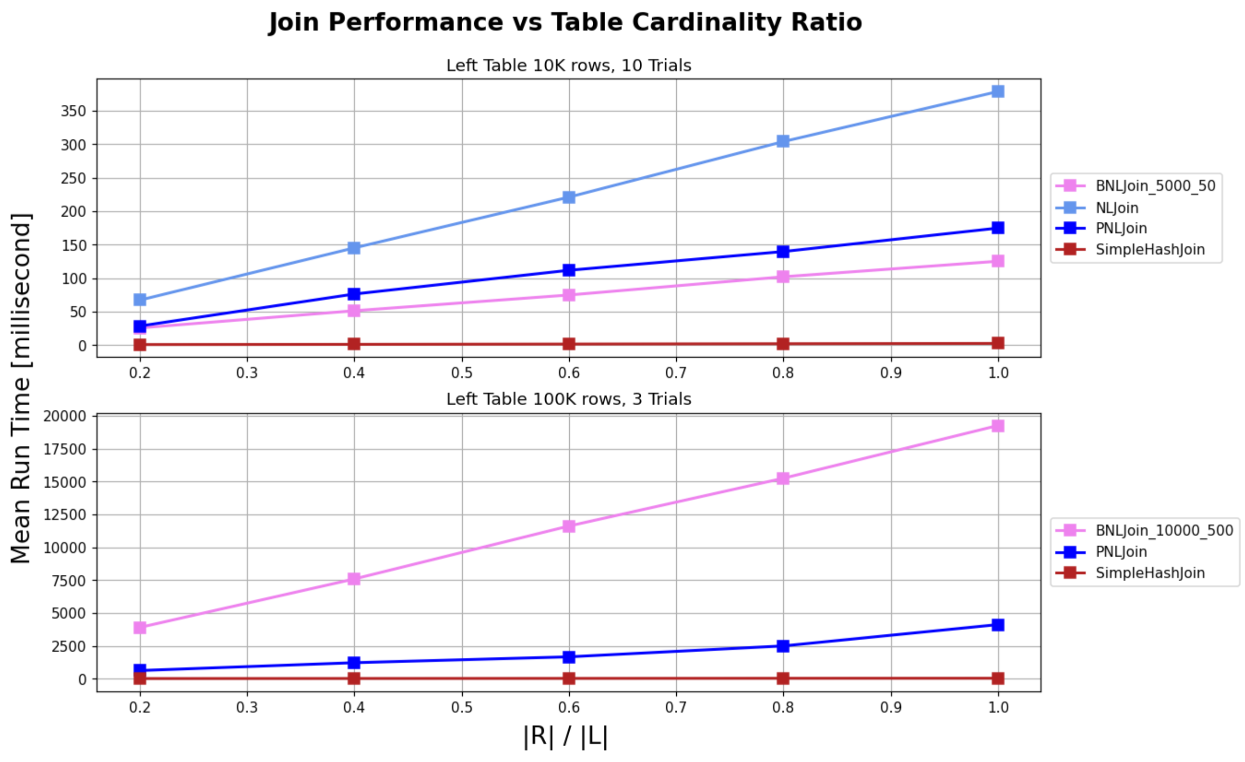
We ran the above join algorithms against various left and right table sizes as in the table below

Left Table Records	Left Table Size	Right Table Records / Size				
10K	1.1 MB	2K	4K	6K	8K	10K
		216 KB	432 KB	644 KB	860 KB	1.1 MB
100K	11 MB	20K	40K	60K	80K	100K
		2.1 MB	4.2 MB	6.3 MB	8.4 MB	11 MB
1M	105 MB	200K	400K	600K	800K	1M
		21 MB	42 MB	63 MB	84 MB	105 MB
10M	1.1 GB	2M	4M	6M	8M	10M
		210 MB	419 MB	629 MB	838 MB	1.1 GB
100M	11 GB	20M	40M	60M	80M	100M
		2.1 GB	4.1 GB	6.2 GB	8.2 GB	11 GB

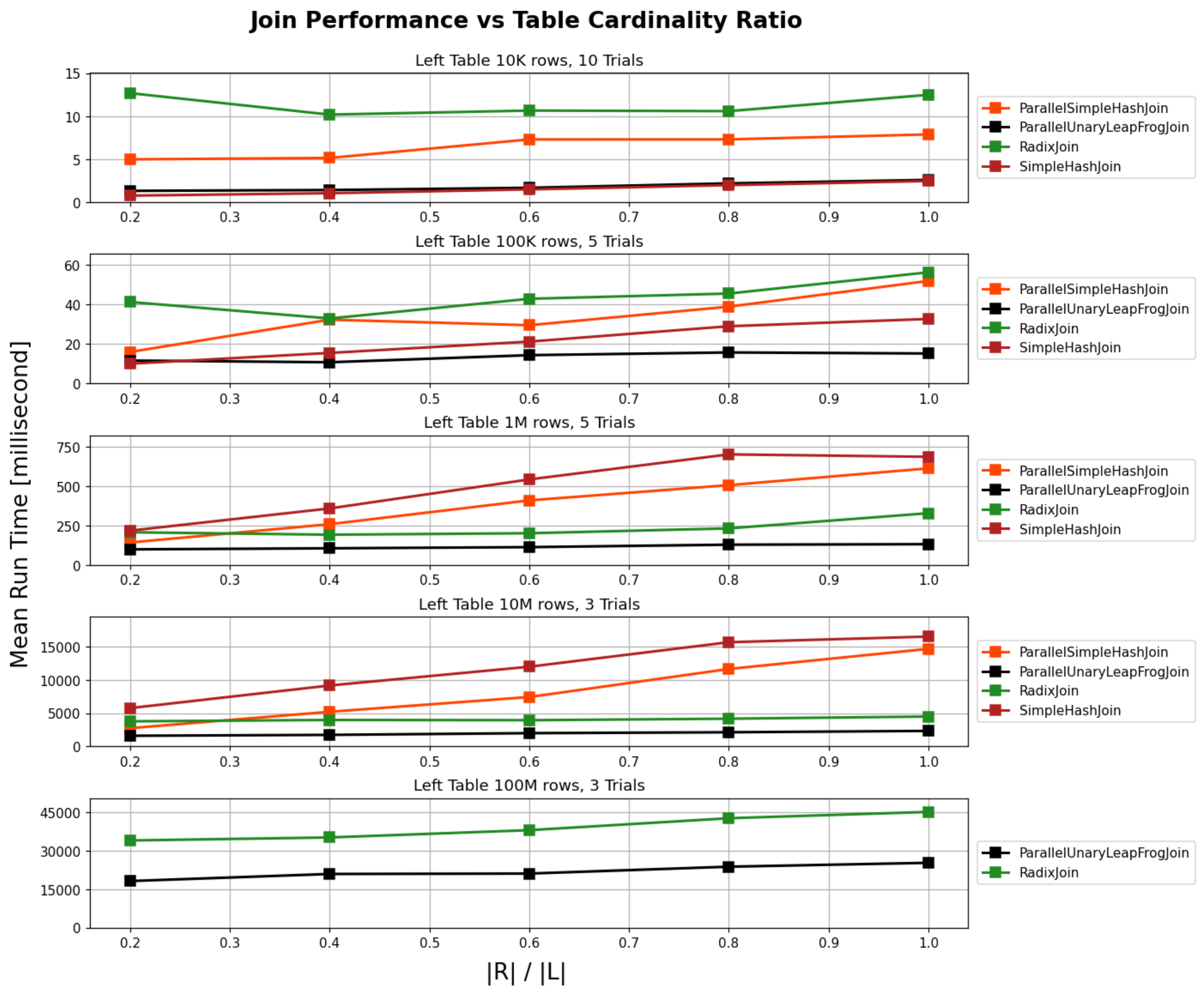
We measure mean execution time of join algorithms across multiple trials as our primary metric of performance for join algorithms. We orchestrated table generation, join profiling, and experiment plotting on a powerful multicore machine. It is a 24 core CSAIL OpenStack cloud machine with 96 GB RAM running Intel(R) Xeon(R) CPU E5-2630 v4 CPUs. It has 32 KB L1d & L1i cache, 4096 KB L2 cache, with one thread per core, and KVM hypervisor. The results of the experiment are below



Parameter sweep of block nested loops on small tables



Performance of traditional algorithms on small tables



Performance of the more novel algorithms on all table sizes

Block Nested Loops

Starting from the block nested loops graph downwards, we observe that not all block nested loops algorithms are born equal. The worst performing variant uses a small left block size and a large right block size. Conversely, a large left block size and a small right block size is the best performing, with all other combinations' performance roughly uniformly distributed between the two aforementioned. Empirically, a small right block size is the most significant factor. We conjecture that this is due to interaction with the cache. While all sufficiently large block sizes will have a high cache hit rate, there is less of a need to load values into the cache with a smaller right block size

Traditional Joins

In the traditional algorithms graph, we observe that, as expected, nested loops is the worst performing join, with block nested loops and parallel nested loops following, and simple hash emerging as the clear winner. Interestingly, parallel nested loops outperforms block nested loops on the larger table. This is likely due to the small return on investment for parallelization on a small table. The reverse effect is observed on the larger table, namely the overhead of parallelization becomes worth the multicore exploitation on larger tables. Note that despite an efficient implementation of parallel nested loops join, the number of cores, a constant, is insufficient to fight the quadratic increase in runtime of the traditional nested loops join algorithm. That is, *O(L*R/C)* will still scale quadratically (and inconveniently so) even with large values of C

Advanced Joins

Now, we discuss the more complex join algorithms. Radix performs surprisingly badly on small tables, even though it should have better data locality and cache awareness than simple hash. One explanation of this is that since we use *h1,1* and *h1,2* based on a radix of the join attributes, the number of partitions can be suboptimal for small tables. In particular, we might end up with many partitions that are empty or have very few values. The parallelization still assigns one thread per partition, so we have high synchronization overhead for a small amount of work per thread

We might expect that a parallel implementation of simple hash would improve performance. In fact, for left table size <= 100K records, the overhead of parallelizing simple hash harms performance, but at the largest table sizes, parallel simple hash does outperform the sequential simple hash as expected

Leapfrog join is the most surprising result. It efficiently leverages multicore sorting, done in parallel on two separate tables, concurrently with one another, but does not parallelize the two-finger merging step. Despite this, and that run lengths are expected to be small on average in our workloads, *leap frog is the best performing join algorithm that we have observed, on all table sizes*. We are unsure of the exact reason why the observed performance is better than that of the theoretical O(N*logN) complexity

We invested a significant amount of manual work to tune chunk sizes of records per core and the block sizes for block nested, but these numbers must likely change based on workload, table value distribution, and other factors to be near optimal. Therefore, the more scalable algorithms are likely to be those which do little manual tuning of this sort, or none at all

Conclusion

We implemented various join algorithms in Rust and profiled their performance. In doing so, we've found that leap frog join is the best performing on all table sizes. The use of Rust greatly complicated the task initially, but enabled fearless concurrency later on, via e.g., rayon. We empirically confirmed that the key to getting good performance is striking the right balance between multicore dispatching of subtasks and cache-aware algorithms. Finally, the more complex algorithms turned out to be more underspecified, which allowed for some of our own optimizations