**CS165 Report**
Albert Ge

## 1 Introduction

This report summarizes high-level results for the completed parts of my project. The following features are supported:

- (M1) load, select, simple aggregate queries
- (M2) shared scans, multithreaded shared scans
- (M3) sorted column indexes, btree indexes
- (M4) nested joins, hash joins, grace hash joins

### 1.1 How to interpret cache results

In each section, the cache results were obtained from Cachegrind. Cachegrind primarily produces two useful metrics: 1) references, and 2) misses.

References usually mean L1 cache references - roughly, the number of bytes accessed by the program when running a given query. This metric also gives an approximation for the amount of computation done-that is, how many instructions are being executed for a query.

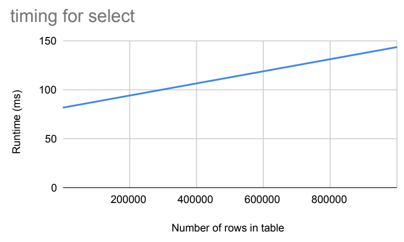
Misses usually mean L1 cache misses - the number of bytes that were accessed but not located in any L1 cacheline.

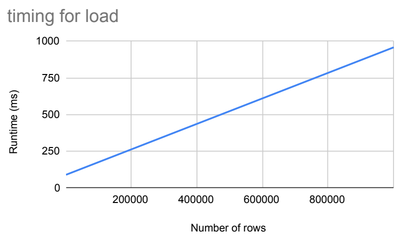
Miss rate is computed by dividing the number of misses by references.

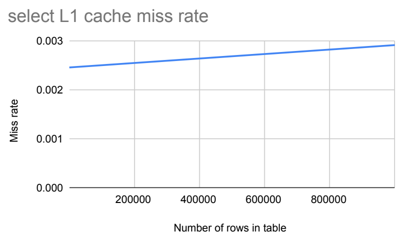
### 1.2 Machine specs

This project was run on a 2015-MBP, 2-core machine. L1 cache size is 32768, page size is 4096.

## 2 M1





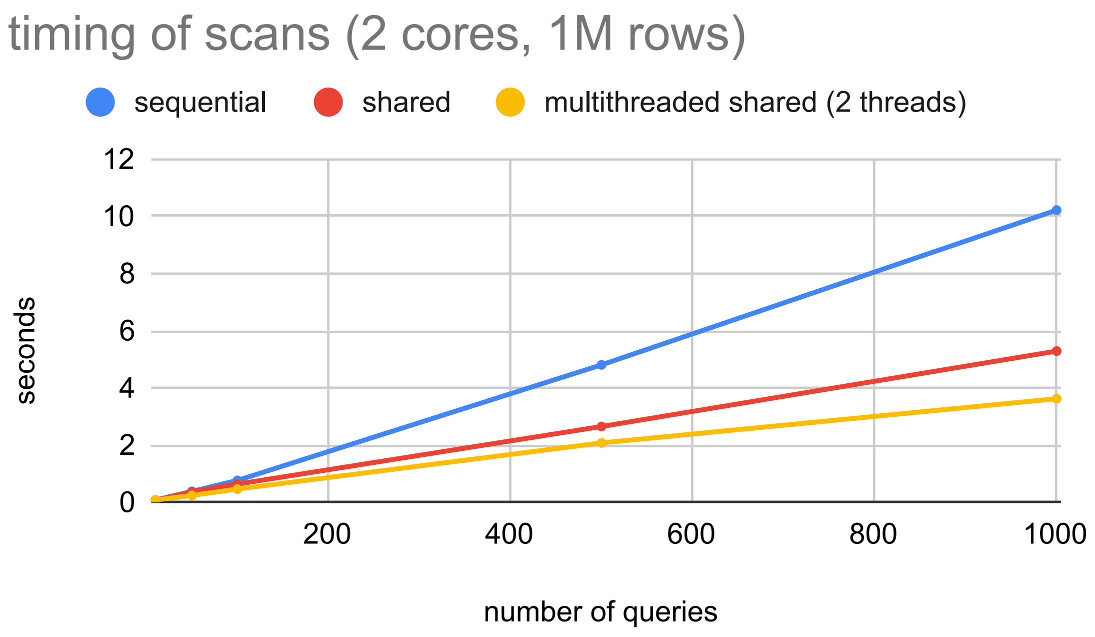


We see a linear scaling with the select and load functions. Our cache miss rate is small, and this stays relatively flat as the number of rows in the table scales.

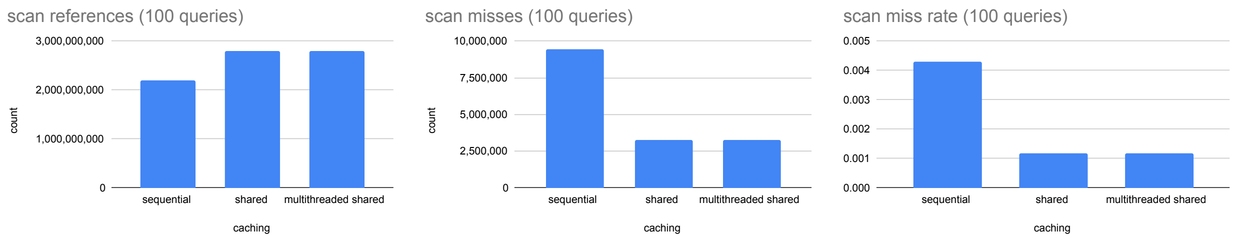
## 3 M2

This graph shows that shared scans have better scaling factor than sequential scanning. Ideally, we expect shared scans to be flat, since we only need to move the data once. In my project, this effect is not as strong, since the data is already in main memory. The relative cost of constructing the result for each query, for every row, is higher than if the data had to be moved from disk.

For the multithreaded case, I use 2 threads (since I assume optimal performance is approximately 1 thread per core). 4 threads have a similar performance to 2 threads.



Though not completely flat in their scaling, both shared and multithreaded shared scans are slightly sublinear (the more queries in the batch, the shorter the amortized time to complete each query), and definitely better than sequentially performing the queries.

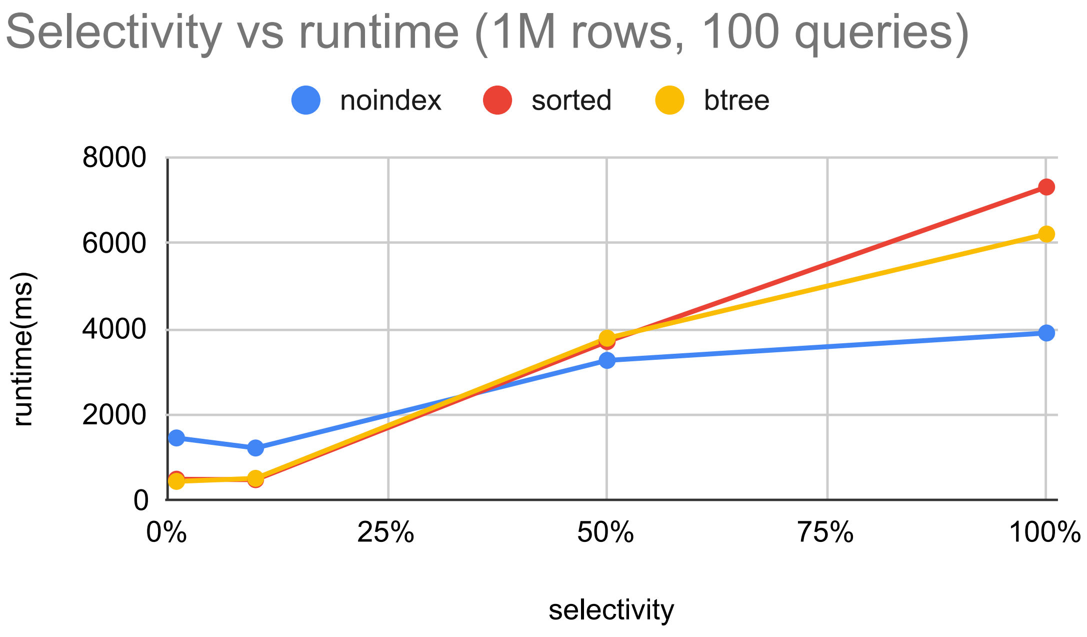


The miss rate on 100 queries shows that the shared scans is able to keep the average number of misses very low - there is not as much data movement. The sequential scans moves much more data. Compared to M1 (a single query), the shared scans are slightly better (on average, less data is moved per-query), and sequential scans are slightly worse.

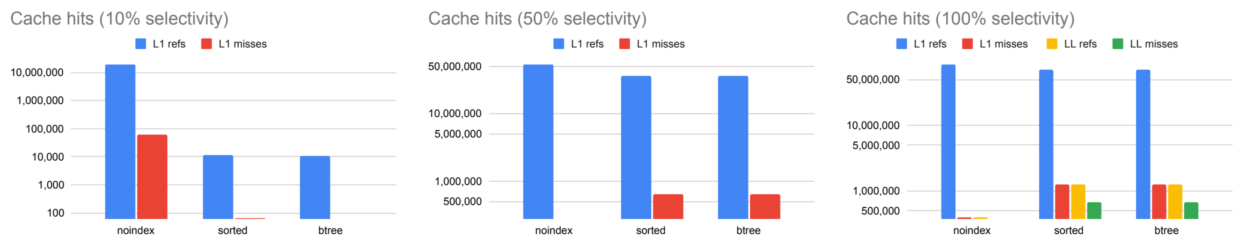
## 4 M3

The below figure demonstrates how varying selectivity will result in different query plans to be used (not using any index, using the sorted column, or using the btree). For low selectivity (only selecting a few tuples), the sorted and btree indexes perform the best. This makes sense, since they avoid scanning the entire table. For high selectivity (selecting many tuples), not using an index outperforms all other indexes, since much of the indexing work is made redundant by the number of tuples that qualify.

Sorted indexes and btrees obtain very similar performance, since the data is already in memory. We would expect a greater performance difference if the data were loaded from disk (since more pages would be accessed in the sorted index). At 1% selectivity, the btree performs slightly better. While not shown, the sorted index could perform better when it accesses fewer pages than the btree. This could happen if the sorted index operates on a column of unique values, and is able to ”hit” the right value when performing



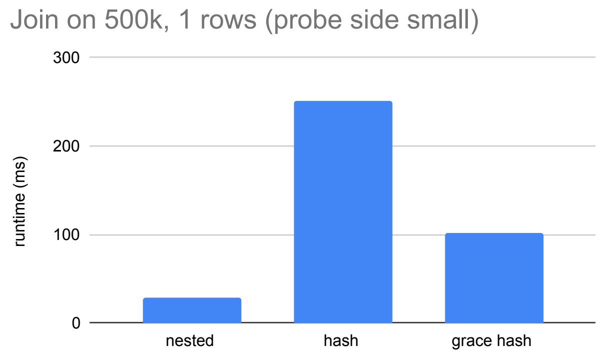
binary search, and can terminate early. Since my project assume there are duplicates in every column, sorted indexes in general should be slower than btrees.

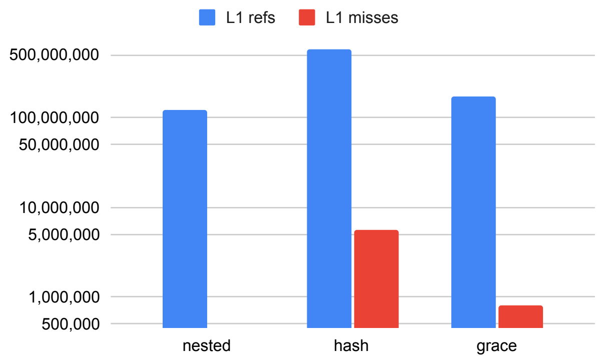


These figures break down the cache hits/misses for given selectivity levels, and explain the runtime performance that we see in the previous graph.

- For 1% selectivity, we observe that the no index plan has many more cache references, since it needs to scan the entire column. Both sorted and btree indexes have much fewer cache references, and also fewer misses (since the data is sorted, so all the qualifying tuples are packed closely together).
- For 50% selectivity, we observe that the no index plan has slightly more cache references, since it still needs to scan the entire column. However, sorted and btree indexes have much higher L1 cache misses. The indexes need to jump between different pages of the data in order to find the appropriate range of values, and this results in many misses. Much of this work becomes redundant since a large portion of the data needs to be scanned anyway, and hence we can see why no index has slightly better performance.
- For 100% selectivity, we observe a similar trend. There is now little difference between cache references for all three query plans. However, both sorted and btree indexes still have many L1 misses, and in particular, the LL (L3) cache also has a nontrivial number of hits and misses as well. There are virtually no LL hits/references for no indexes. This tells a complete story of the performance differences; the indexes are constantly moving new pages into the cache, even going as far down to the L3 cache to bring pages in.

## 5 M4



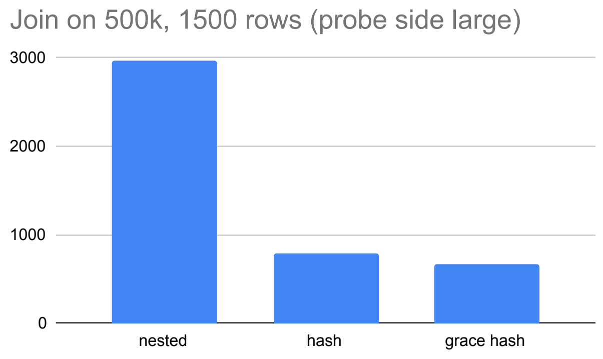


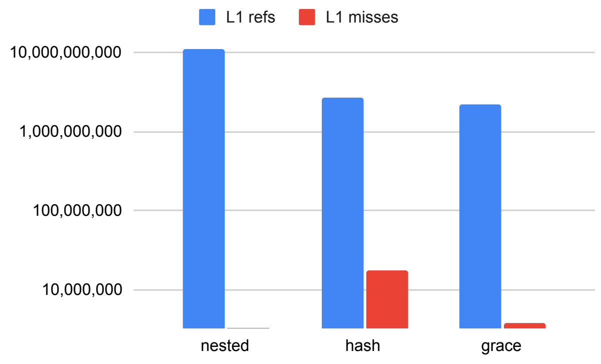
The left figure demonstrates that nested loop joins are the most efficient when the probe side is small (one column can completely fit in 1 page). The nested loop query plan becomes essentially a single loop over the larger column, and thus can complete in a single pass over the columns.

For both hash joins, there is additional overhead of constructing a hash table, querying its values, and also deallocating it. In my hash join implementation, the left column is always used to construct the hash table - thus, a hash table needs to be constructed for all of the larger implementation. Additionally, a new hash table is created for every L1 cache size - this ensures that the data for the hash table can completely fit within the L1 cache. This leads to the worst performance for all join plans - the hash table needs to be created, destroyed, and recreated again.

In my grace hash implementation, the smaller of the two columns is used to build a hash table, so that the larger column can just be streamed. In this case, grace hash is still much better than the hash join. However, for grace hash there is still extra overhead in building and writing the partitions, and thus is still slower than nested loop.

The cache counts in the right figure corroborate this story. For hash joins, there are significantly more cache hits and misses from the hash table constructions. There is much less computation for grace hash joins, but still suffers from some cache misses, from the partitioning overhead. Nested loops have not only the least cache references, but virtually no cache misses.



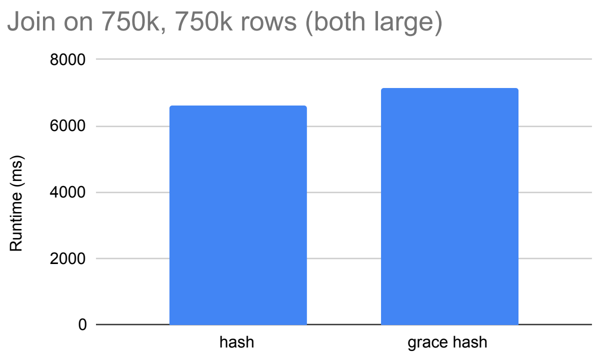


This figure demonstrates that grace hash joins are the most efficient when the probe side is slightly larger (but can still completely fit in L1 cache). This time, grace

This time, because the probe side is larger, the nested loop performs many more comparisons. This results in quadratic scaling, since every row in the first column must be compared to the second column, and thus performs the worst.

In this case, both hash joins are much better in runtime. Both hash joins save a lot of comparison work since the hash enables that only the keys that match will be compared in the join. This leads to much fewer cache references.

Grace hash joins are much better than hash joins on cache efficiency - there are much fewer cache misses in grace hash than for hash joins. This difference mainly occurs because the probe side can still fit in the L1 cache. The grace hash will, for each partition, choose the side that can fit in L1 cache, and thus the other column can just be streamed. This results in a slightly better runtime for grace hash compared to hash joins (most of the work is still from the constructing the result for the joins).



This final figure shows that on very large tables, hash joins are slightly better. This time, both of the columns will not fit in the L1 cache - even after partitioning for grace hash. Hash joins can still take advatange of cache locality, since only L1-cache-size elements are added to the hash table at any given time.

For grace hash, each partition will have more than L1-cache-size number of elements. Since we construct a hash table for each partition, this table will not be able to fit in the cache. We can implement a recursive scheme to break up the partition into smaller pieces such that each sub-partition can fit in the cache, but this incurs a lot of expensive overhead from splitting and rewriting new partitions.

Nested joins are not included in this graph, simply because it takes too long to run.