CS 4321 Project 5 Performance Report

Team members: Sitian Chen (sc2294), Xinqi Lyu (xl358), Xinzhe Yang(xy269)

Abstract

In this project, we explored multiple options of performance improvements such as parallelizing block-nested loop joins and using hash join instead of sort-merge join. The different methods and different combinations of them produced various results. We will summarize the options we attempted and evaluate their performance in this report.

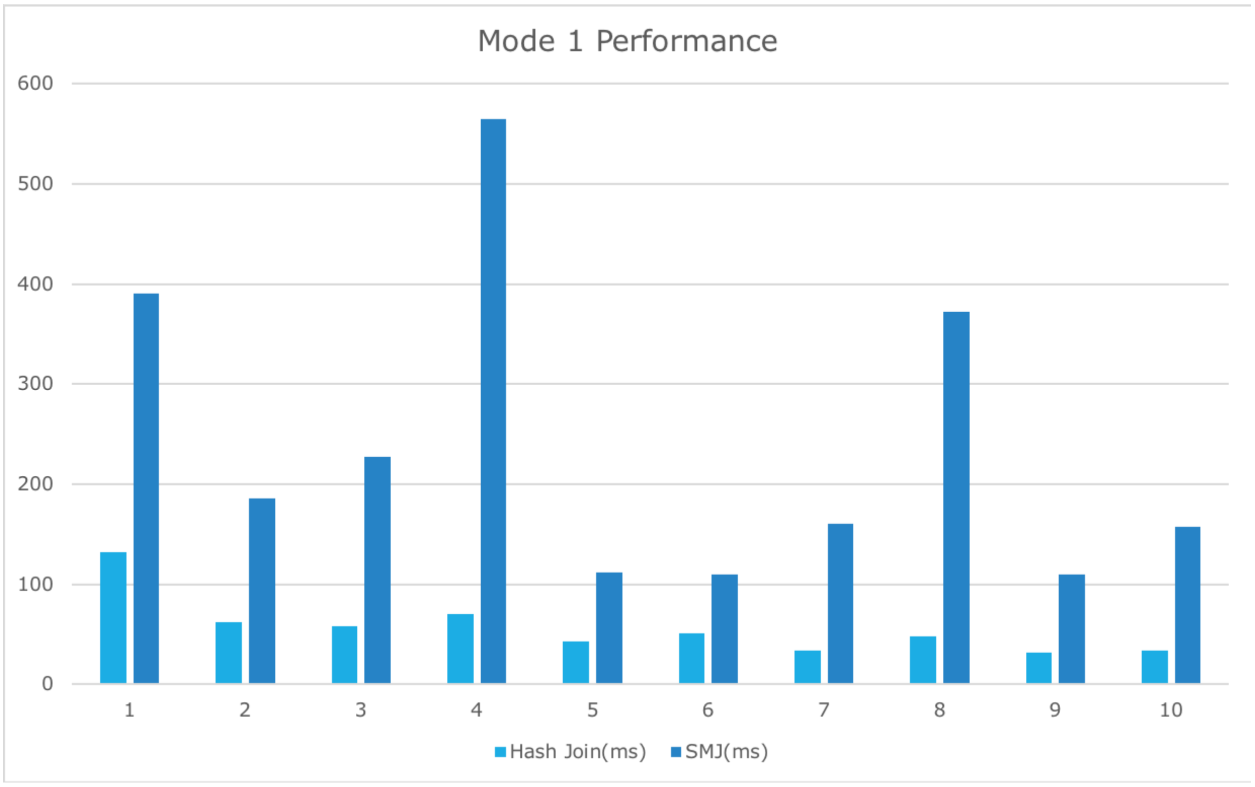
Implementing Hash Join and Attempting to Parallelize Hash Join

We implemented hash join to speed up equity joins and the hash join operator is located in physicaloperator/HashJoinOperator.java. We follow closely the algorithm discussed in class and in the paper *Parallel GRACE Hash Join on Shared-Everything Multiprocessor : Implementation and Performance Evaluation on Symmetry S81* provided on piazza, with a split phase implemented in the method splitphase() and the join phase (build + probe phase) merged into getNextTuple() so that the operator gets one joined tuple at a time.

We compared the results for running the first ten tuples of mode 1 with running with our previous sort-merge join method from project 4. SMJ is run with 10 buffers for external sort, and hash join is run with 12 main memory buffers hardcoded. All queries are executed with the same logical and physical query plan. We can see from the results and conclude that the hash join method performs much better than the sort-merge join method for queries with all-equity queries. We believe that this is because the hash join method reduces the I/O costs significantly.

Query	Hash Join(ms)	SMJ(ms)
1	132	391
2	62	186
3	58	227
4	70	565
5	43	112
6	51	110
7	34	161

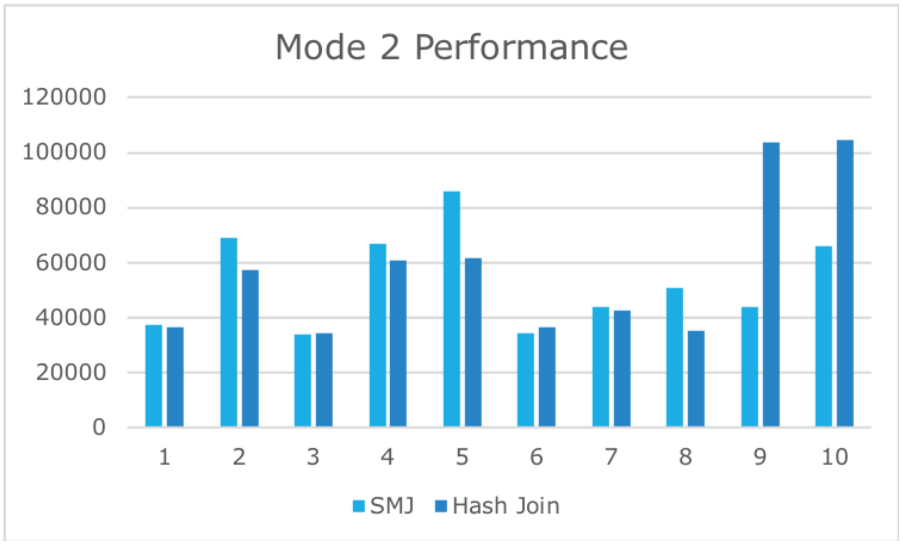
8	48	372
9	32	110
10	34	158



We also attempted to parallelize the split phase of the hash join and the implementation is in the method splitPhaseParallelized(). We parallelize the implementation of applying the hash function over tuples by using Java Streams: we first store pages of tuples from the relation to be partitioned in a buffer in main memory and then parallelize the partition of the buffer by the hash function, the partitions are also stored first in main memory and then each partition of tuples is written to the buffer of its corresponding TupleWriter. However, when benchmarking on queries from mode 1 we found that the performance were worsened. We were not sure about the underlying reason, but it’s probably from the overhead of storing and moving buffer of tuples and partitions in main memory. In addition, the execution time is mainly from I/O costs, and the cost of CPU time is rather marginal, so it’s easy to generate extra overhead from parallelizing CPU executions. Since parallelism is not performing so well in the split phase and hash join is already very fast for equity joins, we choose to not parallelize hash join and use single-threaded hash join instead.

We also tested performance of Hash Join against SMJ on queries from mode 2, on which Hash Join / SMJ is combined with BNLJ in query processing.

Query	SMJ	Hash Join
1	37205	36444
2	68794	57425
3	33828	34257
4	66841	60800
5	85995	61714
6	34125	36597
7	43954	42469
8	50807	35173
9	43643	103844
10	66133	104460
AVG	**53132.5**	**57318.3**



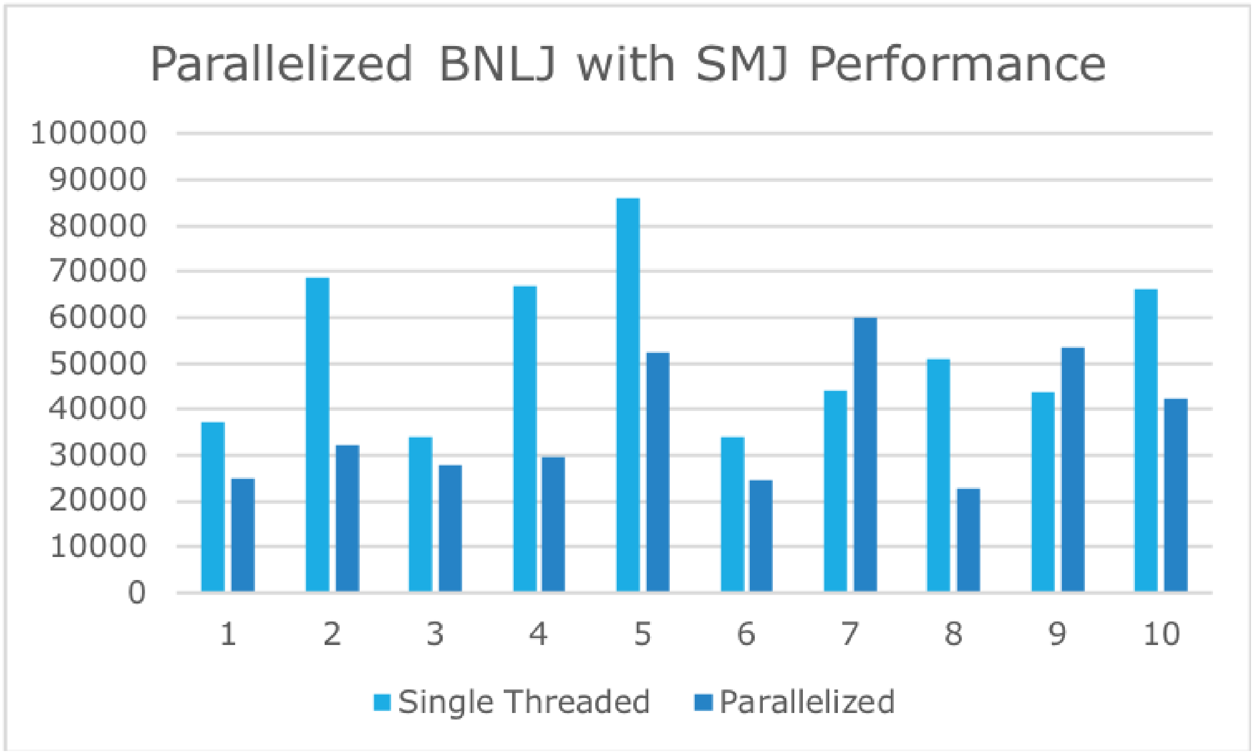
As we can see from the results, the performance of Hash Join against SMJ combined with BNLJ is mostly similar on queries, and sometimes Hash Join is even slower than SMJ on queries such as query 9 and query 10.

Parallelizing BNLJ

We used parallel streams in the Java standard library to optimize the BNLJ.
The BNLJ algorithm uses a block of pages for one relation and one buffer page for the other relation. In the previous, we use while loops to repeatedly checking matched tuples through each block. In this phase, however, we parallelize the tuple matching for all pages in the block, by applying a stateless lambda expression to the parallel stream.

We tested the performance of parallelized against single-threaded BNLJ on the first ten tuples of mode 2, and the results seem effective - the parallelized BNLJ seems to perform faster than the single-threaded BNLJ in most cases except query 7 and query 9, and the average query run time is also much faster. All equity joins in query plan are joined with SMJ. The buffer size for both single-threaded and parallelized BNLJ is 10.

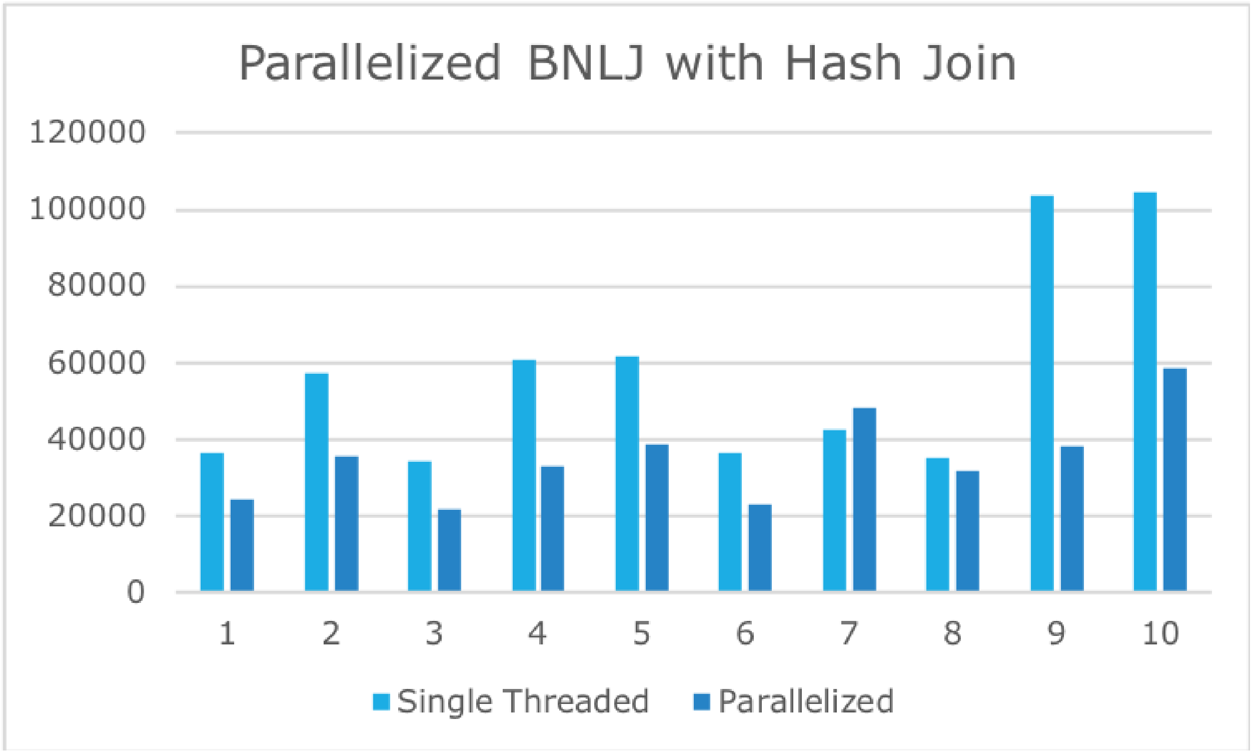
Query	Single Threaded	Parallelized
1	37205	25003
2	68794	32028
3	33828	27963
4	66841	29756
5	85995	52329
6	34125	24392
7	43954	60094
8	50807	22894
9	43643	53345
10	66133	42184
AVG	**53132.5**	**36998.8**



We also tested the performance of parallelized BNLJ combined with Hash Join. They produce optimistic results. After parallelizing, the performance improved significantly.

Query	Single Threaded	Parallelized
1	36444	24074
2	57425	35478
3	34257	21506
4	60800	33028
5	61714	38575
6	36597	22831
7	42469	48275
8	35173	31675

9	103844	37958
10	104460	58600
	57318.3	**35200**


Parallelized BNLJ with Hash Join
Single Threaded
Parallelized