CMSC 23530: Parallel Sort Merge Join Project

Mark Cao

May 27, 2022

1 INTRODUCTION

Parallel computing is a subdivision of high-performance computing. Its main idea is to decompose complex problems into several parts and assign each part to an independent processor (computing resource) for the calculation to improve efficiency. Join is one of the most expensive operations for modern DBMS, so applying parallel computing to join operation is always a point worth exploring.

There is two main parallel join method which is a parallel hash join and parallel sort-merge join. In this paper, we experimentally study the performance of a parallel, multi-core sort-merge join algorithm embedded in the CrustyDB [3] which is an academic Rust-based relational database management system built by ChiData at The University of Chicago as the third join algorithm besides the nested loop join and the hash join.

2 IMPLEMENTATIONS

Parallel sort-merge join has three phases which are partition phase[1], sort phase, and merge phase. In three phases, the partition phase is optional and we did not implement partition for two main reasons. First of all, the project I did last quarter built and benchmark hash join on the same machine which can be used as a comparison, and the hash join skipped the partition phase. In order to compare their throughput and latency, the parallel sort-merge join in this study has to skip the partition phase. Second, due to the late joining of this class, the real working time is shorter, so we have to do some compromises by skipping the partition phase in order to get the project done.

For the sort and merge phase, there are three levels which are In-Register Sorting, In-Cache Sorting, and Out-of-Cache Sorting [4]. In each level, we partition the data into runs that fit into CPU registers, CPU caches, and out of cache respectively, sort the runs in parallel and merge the sorted runs into the next level of sort-merge. For level 1 and level 2, we implemented the Sorting Network algorithm for sorting and the Bitonic Merge Network algorithm for merging two runs together [2]. Because of the sorting network, our sort-merge algorithm will only work for the input that size is a power of two. Level 3 will redistribute all the runs into bigger runs and the number of level 3 runs are decided by the number of physical threads in the machine, so we decided only to use that for level 1 and level 2. In this case, the input of the sort-merge algorithm will not be limited further.

As part of CrustyDB, the implementation is highly crusty-like, it contains a struct SortMergeJoin, a impl for itself, and the OpIterator. In the OpIterator, the open function will take care of all the jobs before joining, for example, partition data into level 1 runs, sort-merge level 1 runs, sort-merge level 2 runs, and sort level 3 runs based on the methods (m-pass/m-way), in parallel for both left and right child. The next function will join the sorted level 3 runs in either multi-way or multi-pass. Because the multi-pass won't redistribute data, it ends up having more level 3 runs than the m-way approach. We will discuss how this will affect their performance in the Result Analysis section.

3 EXPERIMENT DESIGN

In this study, the goal is to compare the different approaches for the m-way and m-pass parallel sort-merge join algorithms and evaluate their performance basically in two aspects: throughput and latency. In order to compare these two aspects in different variable settings, this study is using cardinality and key length as configurations. The running time is the total running time for both preparation and joining, in this case, the sum of the open function and the next function.

Based on the inspiration from CrustyDB, we create a similar equal join structure. Both relations (children) in our sort-merge join contain a vector of vector of tuple. The number of fields in the tuple is changeable, and in our experiments, we set the left child to have 2 fields in the tuples and the right child to have 3. The equal join will compare the tuple's second fields whose index is 1.

Latency and Throughput are two common metrics when evaluating algorithms' performance. Ideally, For a single machine and single thread the lower the latency, the higher the throughput. Throughput is expressed as to how much

amount of data is processed during a defined period of time. Latency is another important thing we care about. In this study, we define latency by average total time per processed tuple. We set micro-benchmarks on different key lengths and cardinalities. We will change the cardinality and key length respectively and record the average time it took, in order to explore the two approaches' performance under different circumstances. The permutations are outlined in Table 1. We set no.1 as the control group while comparing the other micro-benchmarks. The particular reason for the circumstance is that 2048 tuples are a very ideal amount of data to be tested based on the capability of our running machine, and the 0-1000 range is very common in daily practice.

We run the micro-benchmark on my laptop. The machine uses Microsoft Windows 10 Pro, X64 system, and Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz, 2601 Mhz, 4 Core(s), 8 Logical Processor(s).

Permutation List		
No.	Key Range	Cardinality per Child
1	0 - 1000	$2^{11} = 2048$
2	0 - 1000	$2^{15} = 32768$
3	0 - 1000	$2^{17} = 131072$
4	4000 - 5000	$2^{11} = 2048$
5	9000 - 10000	$2^{11} = 2048$
6	99000 - 100000	$2^{11} = 2048$

Table 1: The list of variable permutations

4 RESULTS & ANALYSIS

In this section, we will divide the micro-benchmarks results and analysis into two subsections which are cardinality and key range. In each subsection, we will discuss the results both horizontally and vertically which means comparing the differences between micro-benchmarks and between combinations in the same micro-benchmark.

4.1 Cardinality

In the cardinality subsection, we will use three different cardinality values to test the performances which are 2048, 32768, and 131072. The processing time increase exponentially. We may make the no.1 as the control group. It takes 0.266s and 0.563s to process 2k tuples for M-Way and M-Pass respectively1. And it takes 353.290s and 2163.321s to process 131k tuples. The size of tuples are around 6 times but the running time is more than 3800 times for M-Pass. As we can see from the graph2, the M-Pass is always take more time than M-Way. For the reason, we think it is because it has more runs than threads, which didn't take full advantages of multi-threading processing.

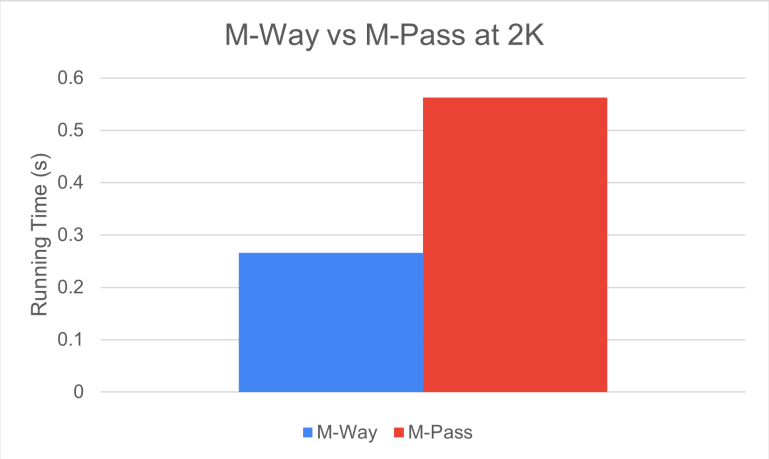

Figure 1: 2k

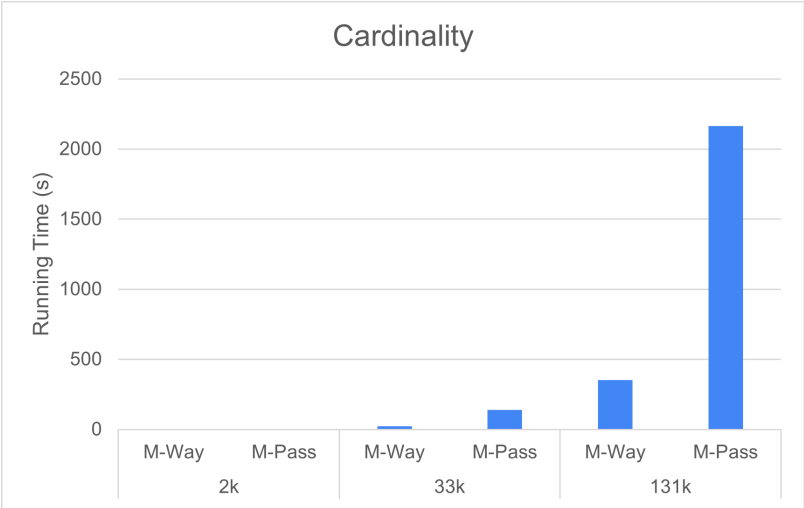

Figure 2: cardinality

4.2 Key Range

Key range is another variable we want to control. Because we would like to see how would the number affect the performance of the sort-merge join. We choose 4000-5000, 9000-10000, and 99000-100000. From the graph3, we can see that our assumption about the key range will effect the performance is incorrect. It didn't change the throughput and the latency very much while the number is larger tha before more than 10 times.

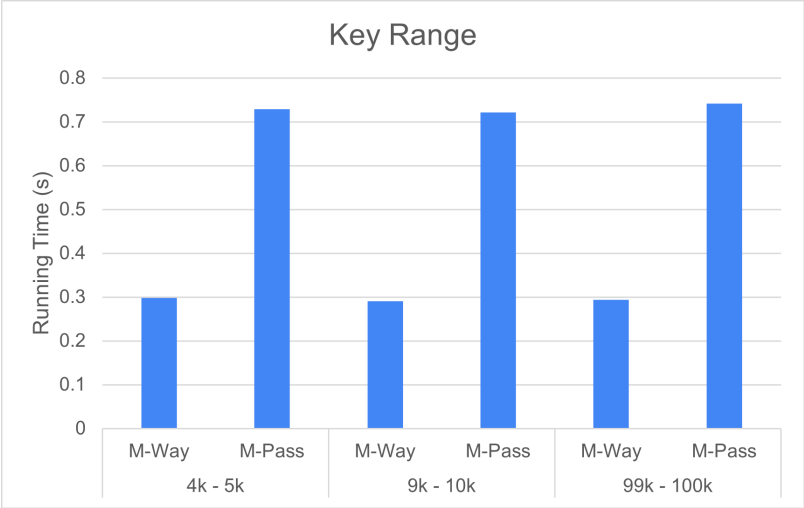

Figure 3: key range

5 FUTURE WORK

There are a lot more interesting things that we can explore, such as the distribution of the same tuple in children. And we actually implemented that micro-benchmark, but there are some errors that we couldn't fix before the deadline, so we have to delete that part from our experiments. Also, we could try recursively use bitonic merge network for level 3 which is another interesting thing to do and see how the performance different from current version.

References

[1] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDB*, 7(1):85–96, 2013.

[2] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *VLDB*, 1:1313–1324, 2008.

[3] ChiData. Crustydb-22. https://github.com/UCHI-DB-COURSES/crustydb-22, 2022.

[4] Authors: Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *VLDB*, 2(2):1378–1389, 2009.