# CSC367 report 3

hongmao.wu

November 2023

The following SQL query count the occurrences of students who have served as teaching assistants (TAs) for courses and have a GPA greater than 3.0:

```
SELECT COUNT(*)
FROM Student s
JOIN TA t ON s.sid = t.sid
WHERE s.gpa > 3.0;
```

In this query, we have two relations, the `Student` and `TA` tables. The `sid` (student ID) attribute in the `Student` relation serves as the primary key, ensuring uniqueness for each student ID. The `sid` attribute in the `TA` relation acts as a foreign key, indicating that for every `sid` value in the `TA` relation, there exists a corresponding `sid` value in the `Student` relation. It's important to note that there may be duplicate `sid` values in the `TA` relation, which means a student has been a TA for multiple courses. Certainly! Let's improve the readability of the paragraph:

## Nested Join

The nested join is the simplest but the most time-consuming join. It involves looping through an outer loop and an inner loop. Assuming there are $n$ student records and $m$ TA records, this join performs $n \times m$ comparisons.

## Merge Join

The merge join exploits the assumption that `sid` values are in ascending order. It compares `sid` values of both relations one by one from beginning to end. It starts from the first tuple of the two relations. If the `sid` value of a tuple from the student relation is strictly larger than the `sid` value from the TA relation, the method selects the next tuple on the student relation and compares it with the same tuple from the TA relation again, and vice versa. Whenever `sid` values from two tuples are equal, the method increments the total number of matches by one. This method then finds all other TA tuples that have the same `sid` values as the current student tuple. This join performs $n + m$ comparisons in total.

## Hash Join

Hash join does not require the assumption of sorted `sid` values. It loops through every tuple in the student relation to create a hash value for every `sid` value. Based on the hash value, it creates a hash table. In the implementation, there are an equal number of buckets in the hash table as student records. Every bucket contains 10 spaces, and the size doubles if needed. The hash function is simply the `sid` value modulus the total number of student records. If the `sid` value is from 0 to the total number of student records minus 1, then every bucket would contain exactly one student record. Then, this method loops through TA records, calculates the hash value, and compares the TA records to

every student record inside that bucket. In the worst case, every student record hashes to the same bucket, and every TA record is at the bottom of that bucket, in which case, it is as bad as the nested join, but this is very unlikely.

### Parallel Join

This join method leverages the previously described three join methods. Instead of finding ways to join two relations, it focuses on partitioning relations into different segments. There are two ways of partitioning: fragment and replicate, and symmetric partitioning. Fragment and replicate partition the larger relation and replicate the smaller relation. Each partition of the larger relation joins with a replicate of the smaller relation. This way of partition performs relatively poorly when the number of student records and TA records are similar. Symmetric partitioning partitions both relations, and each partition of a relation is roughly the same size, except for some extreme cases, such as all TA records having the same `sid`. Symmetric partitioning needs to partition TA records such that every `sid` of a TA record is within the bounds of student records in the same partition.

## OpenMP Implementation - Fragment and Replicate Parallel Join

This section explains the OpenMP implementation details for the fragment and replicates parallel join. OpenMP creates multiple threads, each thread handles a partial join between a fragment and a replicate. To prevent overhead created by last thread, the larger relation is divided such that the last thread is assigned less or equal number of records to join than all other threads. Each thread has access to two relations and determines the starting and ending records based on its thread ID. For instance, thread with thread id 0 join the first partition of the student relation with a replicate of the TA relation, assuming there are more student records than TA records. As the nature of fragment and replicate involves duplicating the smaller relation, each thread duplicates the smaller relation while all threads share the larger relation. At the end of execution, OpenMP reduces the private variable **count** to calculate the total number of matches. The reduction process is performed after partial join has completed.

## OpenMP Implementation - Symmetric Partitioning Parallel Join

This paragraph explains the implementation details of symmetric partitioning parallel join using OpenMP. Each thread has four private variables: *start_ind*, *end_ind*, *lower_bound*, and *upper_bound*, representing the information of a partition on the student relation for a thread. The *lower_bound* and *upper_bound* variables contain the sid values stored at *start_ind* and *end_ind*, respectively.

These variables guide a thread directly to the designated partition. During execution, each thread loops through TA records to find sid value larger than the lower bound and smaller than the upper bound, incrementing the count whenever there is a match. OpenMP then sums up the count at the end of execution. In the worst-case scenario, if the TA relation only contains sid values within a partition of student records, only one thread performs meaningful work, causing the master thread to wait for that thread to finish and incurring overhead.

## Message Passing Implementation - Fragment and Replicate Parallel Join with MPI

This paragraph explains the implementation details of fragment and replicate parallel join using MPI. MPI distributes work across different nodes/lab machines. Each node starts with a partition of the student and TA relations. An assumption is made that any TA record within a TA relation partition will not match with any student record from another partition. Without this assumption, every node would require a complete student relation regardless of the total size of student and TA relations.

If the total number of student records is greater than the total number of TA records, due to the nature of fragment and replicate, each node needs to collect every partition of the TA relation from other nodes and merge them into a complete TA relation. Subsequently, every node must compare its partition of the student relation with the complete TA relation and sum up the total count by calling **MPI_Allreduce**. If the total number of TA records exceeds the total number of student records, nodes do not need to merge partitions of student records, due to the assumption. Hence, each node only compares its partition of the student relation with the assigned partition of the TA relation.

## Message Passing Implementation - Symmetric Partitioning Parallel Join with MPI

This paragraph provides explains the implementation details of symmetric partitioning parallel join using MPI. The same assumption is made here, allowing each node to directly compare its partition of the TA relation with the partition of the student relation. Subsequently, each node can call **MPI_Allreduce** to contribute its local result and receive the globally summed-up result.

## Analysis

Note that my mpi implementation cannot run on multiple machines because it encounters a TCP connection error.

To generate all the graphs discussed below, please execute the command "./generate_graphs" in the command line. This command should produce five

graphs. The execution of this command may take some time, but it is expected to complete within 1 minute. It is important to note that, in addition to sequential execution, only merge join and hash join are tested in the experiment for OpenMP and MPI. This decision is made because the nested join would take a huge amount of time.

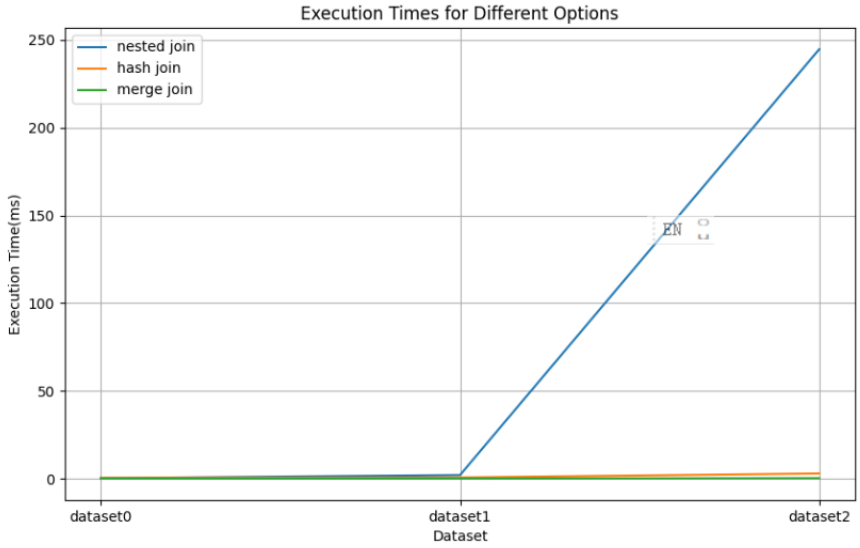


Figure 1: seq graph

By examining Figure 1, it is evident that the nested join is significantly less efficient, and hash join is slightly less efficient than merge join. Datasets 3, 4, and 5 are excluded from the graph since the observed patterns are already obvious, and their execution times are too long.

As the sequential join does not exploit any parallelization, it is expected that both OpenMP and Message Passing would show shorter execution times than sequential join. This conclusion can be made because nested join take much more execution time than both hash join and merge join when processing dataset3.

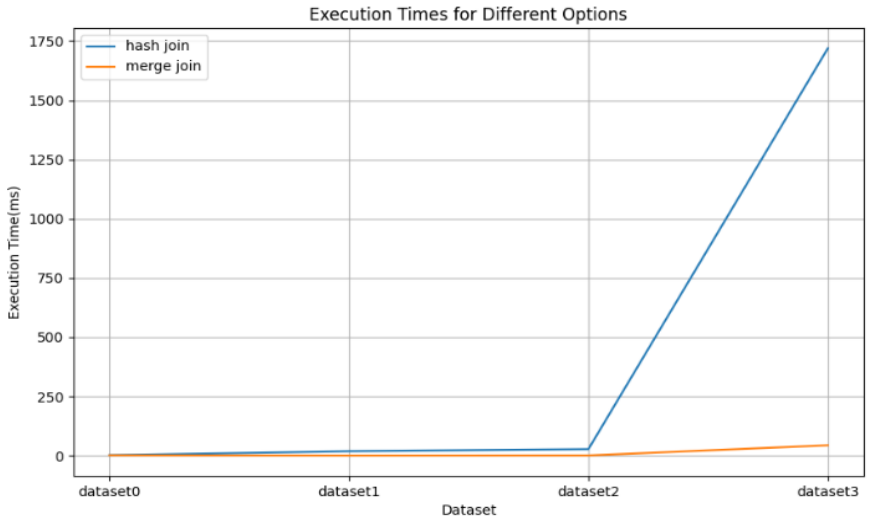


Figure 2: Omp fragment and replicate graph

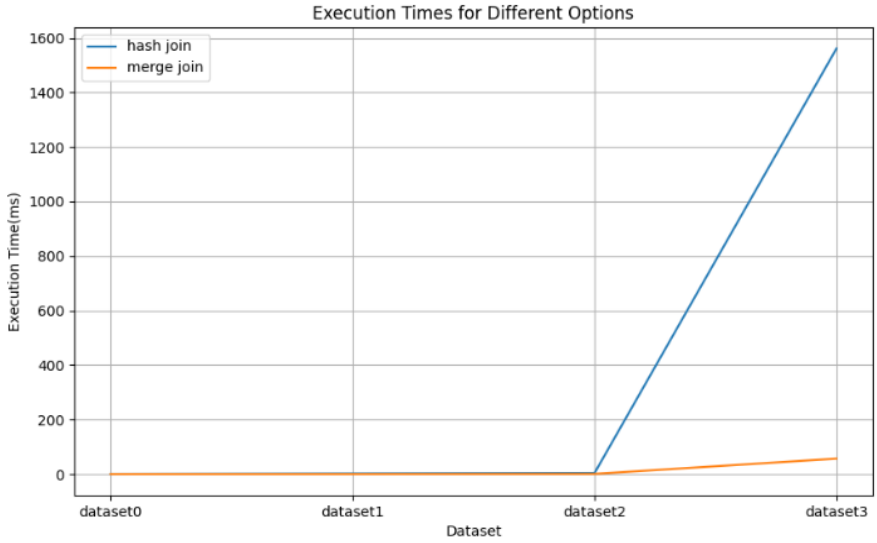


Figure 3: Omp symmetric partitioning graph

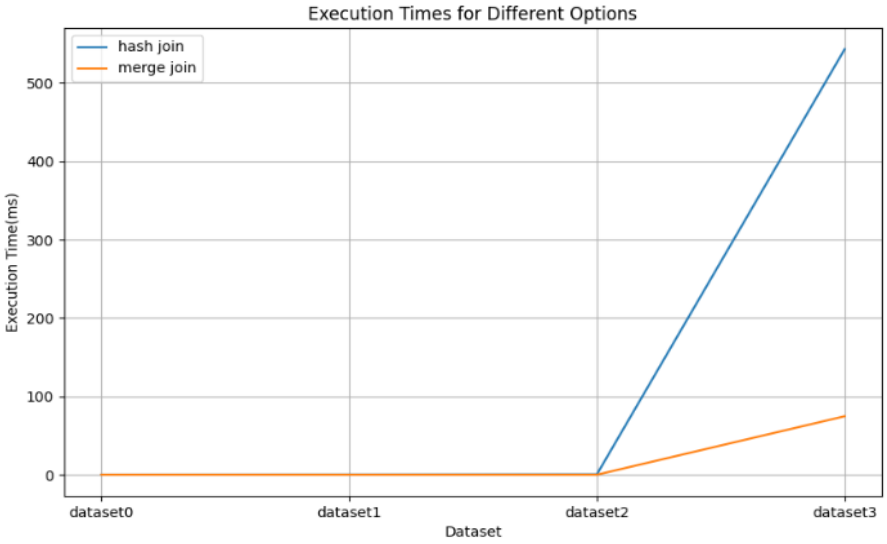


Figure 4: MPI fragment and replicate graph

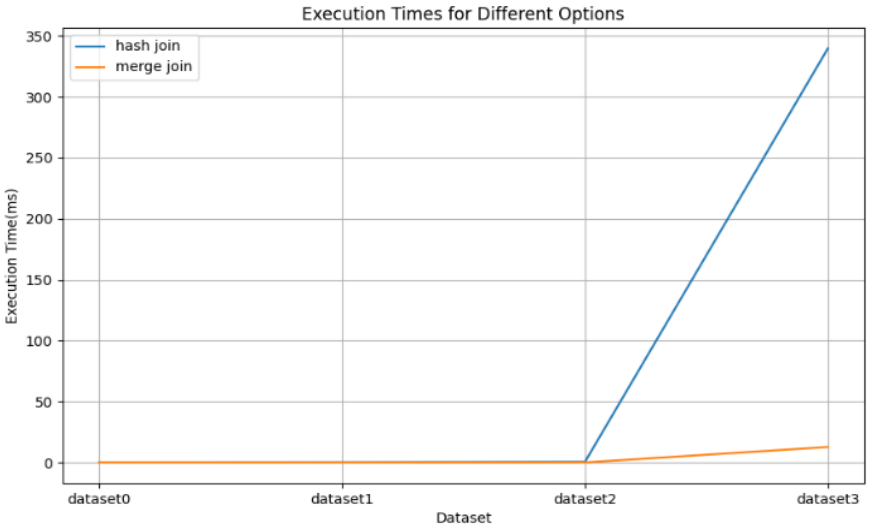


Figure 5: MPI symmetric partitioning graph

Figures 2, 3, 4, and 5 reveal that hash join consistently performs much slower than merge join. Several reasons may contribute to this observation. Merge join compares records sequentially, and these records are stored near each

other in memory, allowing the program to exploit spatial locality and efficiently reduce latency. In contrast, hash join stores data randomly, making it more challenging for the program to exploit cache locality. This contributes to the huge difference between execution time between hash join and merge join for dataset3, as depicted in Figures 2, 3, 4, and 5.

This paragraph examines factors influencing performance by comparing shared address space and distributed address space. Firstly, it is observed that in both shared and distributed address spaces, fragment and replicate take more time than symmetric partitioning. In the case of shared memory address, the difference between the same join method in Figure 2 and 3 seems less obvious than in Figure 4 and 5. This is attributed to both methods requiring the initial partitioning of two relations. The partitioning step involves identifying TA records that should be partially joined with a certain range of student records for each thread. This aspect also contributes to the longer execution time of the OpenMP implementation compared to the message passing implementation, regardless of the chosen join method.

In the context of distributed address space, symmetric partitioning proves to be much more efficient due to the assumption. However, two aspects may increase execution time. Firstly, to align with the assumption mentioned earlier, TA records could be unevenly distributed. This could lead to significant idle time for some processes as they may be blocked by an MPI routine for an extended period while waiting for other processes. Secondly, if there are more students than TAs, fragment and replicate would need to fragment the student relation and replicate the TA relation. In other words, fragment and replicate would have to merge TA records from every other process, essentially combining already partitioned data. This introduces a substantial amount of communication overhead between processes before partial join takes place, which is the reason hash join exhibits around 200 ms more execution time in fragment and replicate than in symmetric partitioning. This outcome is evident by comparing the blue line in Figures 4 and 5.