Performance Report

What we have tried:

In this project, we mainly consider the following two approaches to optimize this database system:

- **Hash Join**

(1) The first idea we have is to use parallel join, that is, replace SMJ with Hash Join firstly and add multiple threads to it.
Based on what we have learned from courses and online materials. If we hope to perform a join between two relations on multiple threads simultaneously to speed up operation, there are two main approaches to consider: Sort-Merge Join or Hash Join. Given that we have already implemented SMJ and observed better performance when encountering equality condition, we hope Hash Join can perform better since it can use multiple threads in at least two phases.

We choose the following way to implement Hash Join:

* (1) Partitioning Phase: given two tables R and S, partitioning two tables separately and save temporary partition file on disk.
————————using multiple threads in the following two phases, every thread is responsible for one partition————————
* (2) Building Phase: building hash table according to the hash key of small table, say R table, for each partition.
* (3) Probing Phase: probing S table using hash function and adding concatenated tuple to message queue, when using getNextTuple() method, get the next tuple from message queue.

In this implementation, we assume that the memory is enough such that every partition in Building phase can be read into memory.

- **SMJ parallel external sort**

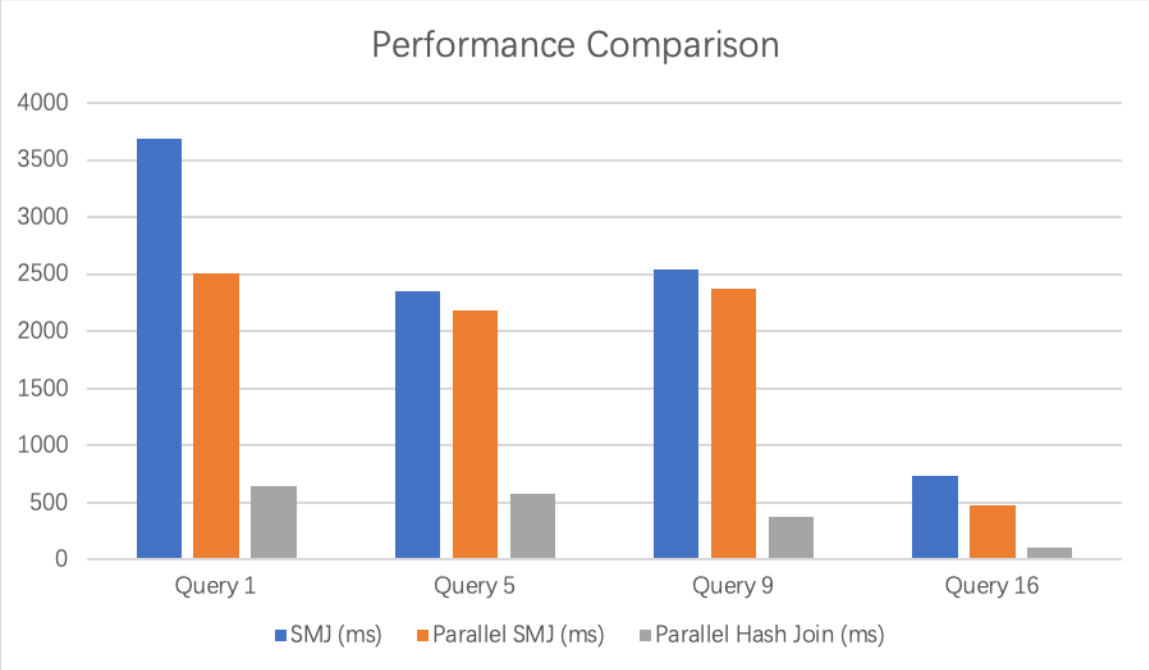
SMJ (Sort Merge Join) Operator, which is also a constantly used join operator in practice case, plays a significant role in dealing with equality constraints. Comparing the time cost for SMJ Operator and for BNLJ Operator, it is apparent that SMJ Operator takes advantages when constraints are of equality.
The main cost of Sort Merge Join Operator is due to the external sort, and the bottle neck for external sort efficiency is the bounded buffer memory. Because of the limited

amount of buffer pages, the processor has to sort a file pass by pass, and during one pass, the database has to load the limited pages of file partitions into memory. To accelerate this process, we use multiple threads to load the files into buffer memory, each thread gets the charge of part of the buffer pages, merge them, and writes them back to the disk.

PS: We also fixed bugs and optimize our join order plan in project 4, which also improves the performance of all queries.

We then use 16 queries and data provided from bench mark to compare the performance among SMJ without multiple threads, SMJ with multiple threads and Hash Join with multiple threads. From the plot, we found that parallel hash join has best performance among three approaches.

	SMJ(ms)	Parallel SMJ(ms)	Parallel Hash Join(ms)
Query 1	3687	2504	648
Query 2	1668	851	212
Query 3	5807	3482	585
Query 4	3399	3027	691
Query 5	2357	2181	579
Query 6	1249	1087	306
Query 7	3616	2770	492
Query 8	3313	2680	643
Query 9	2542	2376	371
Query 10	3112	2252	255
Query 11	3567	2643	428
Query 12	1210	1011	244
Query 13	2442	1863	680
Query 14	3908	3049	349
Query 15	1311	1034	171
Query 16	731	481	102



What we haven't implemented:

- **Parallel BNLJ**

We plan to use multiple threads in BNLJ, for example, use different threads to deal with different blocks. But considering that most of queries have equality condition instead of inequality conditions, we think this optimization may not a good direction. And in BNLJ, we still need to handle cross-product join in every block, which does not have advantages comparing to SMJ with sorted tuples or Hash Join with hash tables.

- **Concurrent Query**

We also consider using concurrent queries to accelerate query speed, however, we have already implemented multiple threads in specific operators in every query while the maximum thread which can be used have already been fixed. In this case, we do not think this optimization approach can help a lot. Therefore, we hope to focus on optimizing operators when processing only one query.