CS3223 Project Report [AY20/21, Sem 2]

Saif Uddin Mahmud (A0170896N),
Tay Kang Ming (A0164811J)
Enhao (A0200239U)

Functions Implemented

Blocked Nested Loop Join (BNJ)

BNJ makes use of BatchList (ArrayList of Batches) as the “Block” for the outer relation during the join. The Block is initialized with B-2 pages, where B represents the number of buffers pages made available to the join in the Physical Plan. The tricky thing about the implementation is that the Output Batch can become full at any point in the next() function. Subsequent calls to next() have to figure out where the function last exited and continue from there. A plethora of flags and counters are used to cleverly keep track of this.

Sort Merge Join (SMJ)

Like BNJ the Output Batch can fill up anytime in the SMJ as well. It handles the issue in a similar fashion. When 2 tuples match the join condition, we add them to either the TempLeft and TempRight files respectively. After ensuring that both the left and right do not have any more tuples with the same values on their join column, we then proceed to perform the join and write them to Output. The TempLeft and TempRight files incur additional I/O cost (on top of what the lecture says) - but they’re necessary to handle the edge case where backtracking operation crosses page boundaries. This also takes additional buffer space (on top of 3). The TempLeft takes upto 1 additional buffer space, and so does the TempRight. Hence, in the plan cost, we made sure that using SMJ would not be feasible if there are less than 5 buffers available.

External Sort

The External Sort is initialized with a list of attributes to sort by, in order of priority. Then it runs the 2 stages:

1. Generate sorted runs by populating the available buffer pages and running internal sort.
2. Implement multi-way merge sort algorithm by using B-1 TupleReader instances (each run gets 1 buffer page). We then compare the head of the runs and write the winning candidate to disk (using 1 TupleWriter instance). Recursively run this stage until the last pass (i.e. 1 file is remaining)

DISTINCT Function

The DISTINCT function is run after Projection. It uses - and only incurs the cost of - the External Sort. We attempted a Hash Based Distinct Function, but it did not support recursive repartitioning in case of overflow. You can check the partial implementation in *DistinctByHash.java.*

ORDERBY Function

The ORDERBY function is run after the Joins. It uses the External Sort by passing in a *direction* which tells External Sort if it should sort the tuples in an ascending or descending order. The parser was modified following the instructions in the course webpage to enable this behaviour.

Aggregate Functions

The Aggregate functions are implemented within the *AggregateValue.java* and are utilized within *Project.java*. That is, the aggregation happens only during Projection. As such, it does not incur additional cost. Two drawbacks of this approach are:

1. You can’t mix aggregate functions with non-aggregate functions.
2. You can’t use aggregate functions anywhere except in the SELECT clause.

Another additional caveat is that the Aggregate Functions use INTEGERS instead of FLOAT. This causes truncation of certain results. This can be quickly patched by changing the data types.

Bugs In Given Implementation

Incorrect Page Nested Join Cost Calculation (Bug Fixed)

The plan cost of nested join is given as *joincost = leftpages * rightpages*. However, the correct cost is *leftpages + leftpages * rightpages* instead.

Ability to Go Over Buffer Space Limit (Limitation - Mitigated)

The given implementation disregards the number of buffers available (configuration) and lets the developer exceed the limit. As an example, we can read all the tables into memory without any explicit errors. To guard against this, we created a BatchList utility class to ensure that we abide by the number of buffers that we have. We also did a thorough analysis of the number of buffer pages used in each function. Furthermore, we used TupleWriter to ensure overflowing batches were always written to disk and TupleReader to ensure reading files was done one page at a time.

Poor Constraint Enforcement (Spotted - Not Fixed)

During Sample Data Creation, the FK constraints are not respected. Furthermore, Composite Primary Keys were not possible, and multiple primary keys in the same table were treated as separate PKey Columns. These bugs were spotted but not fixed.

Experiments

Experiment 1 - Independent Variables

We built a fairly user-friendly script to automatically run a suite of tests for us. Therefore we could collect a wide array of data points for various independent variables, as listed below:

1. SQL Query (Query 1, 2, or 3)
2. Join Algorithm used (SortMerge, BlockNested)
3. Page Size (512 Bytes to 8192 Bytes)
4. Number of Buffer Pages Available (5 to 20)
5. OS Used (MacOS, Windows 10, Ubuntu 20.04)
6. Disk Drive Used (SSD, HDD)

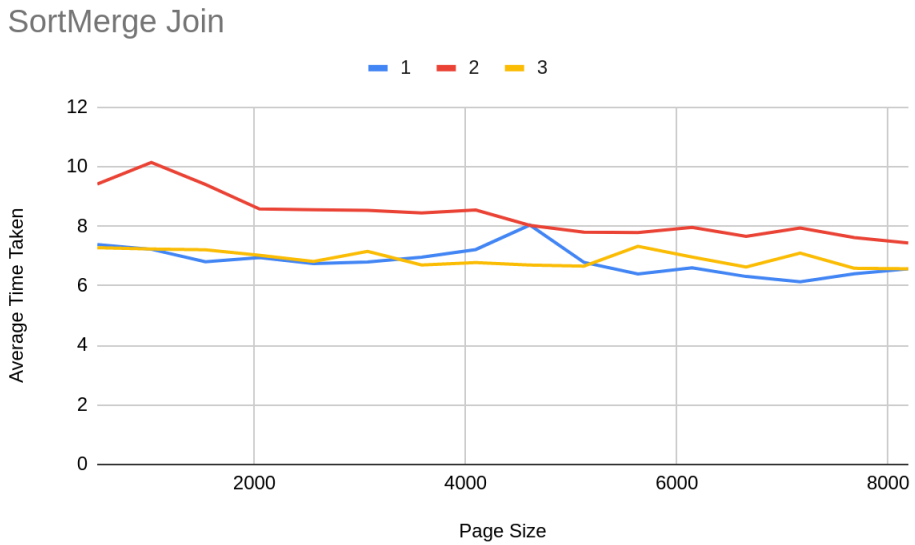
Experiment 2 - Independent Variables

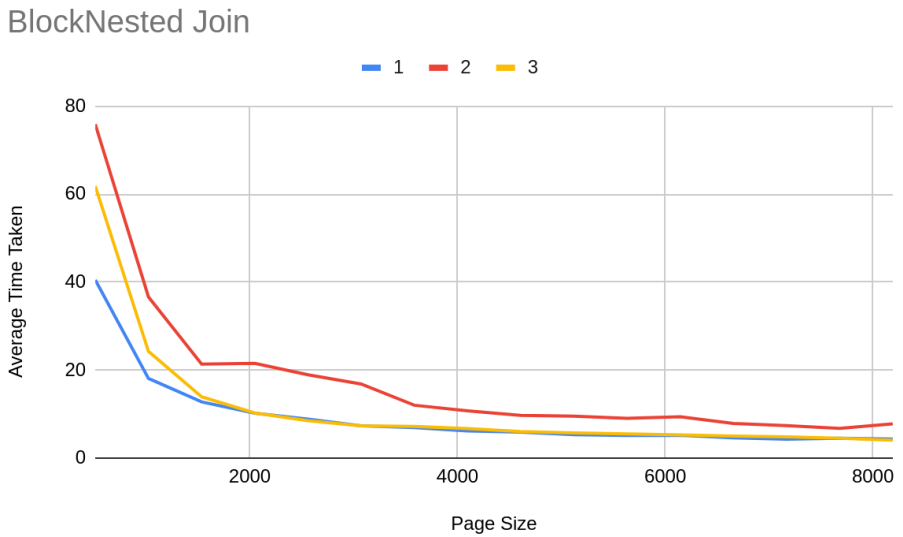
We built a fairly user-friendly script to automatically run a suite of tests for us. Therefore we could collect a wide array of data points for various independent variables, as listed below:

1. Query Plan (BNJ only, SMJ only, Mixed)
2. Number of Buffer Pages Available (20 to 50)
3. Page Size (512 Bytes to 8192 Bytes)

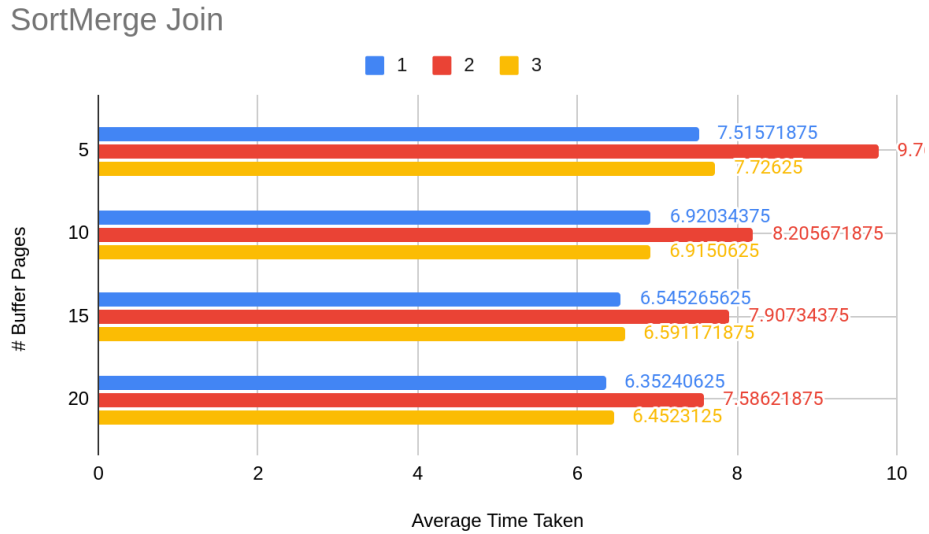
Analysis of Data - Experiment 1

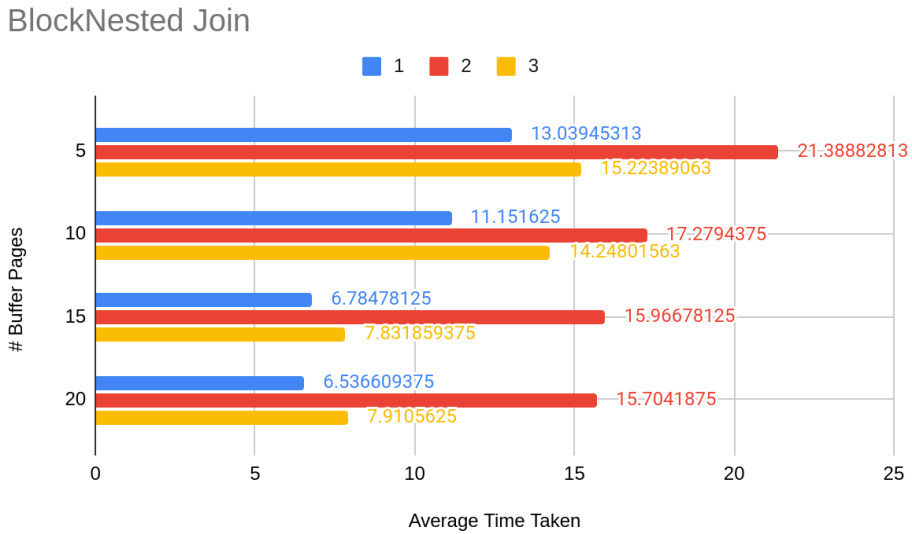
The graphs below demonstrate that the relationship between page size and average time taken is inversely proportional. This is especially prominent for BNJ.



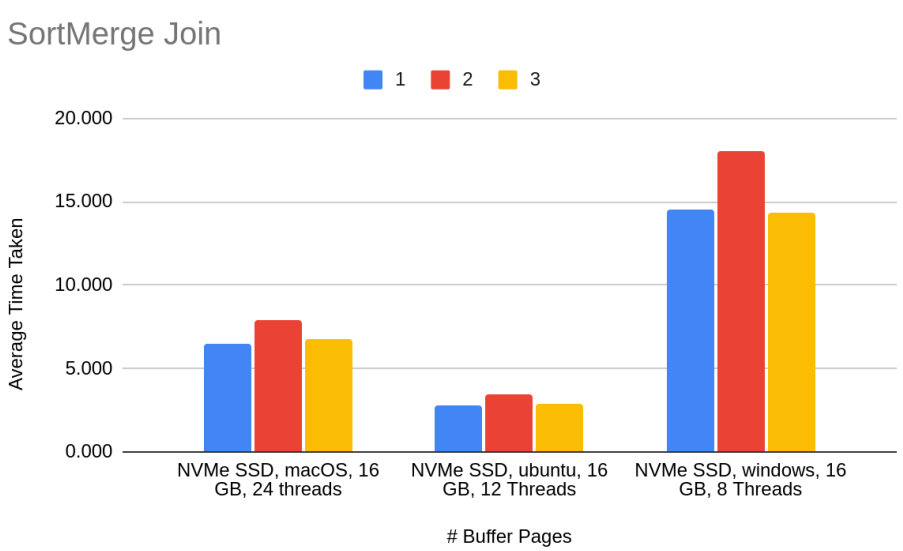


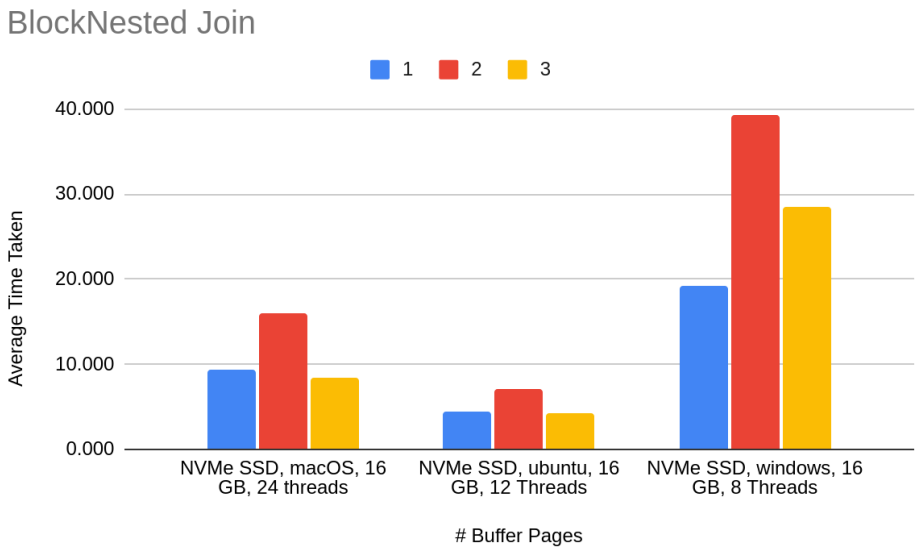
Likewise the 2 graphs below demonstrate the inversely proportional relationship between number of buffer pages and average time taken.



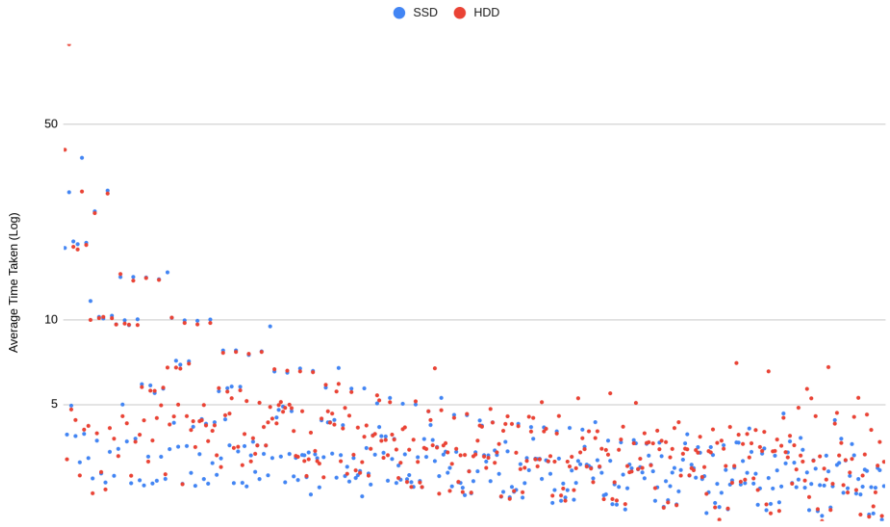


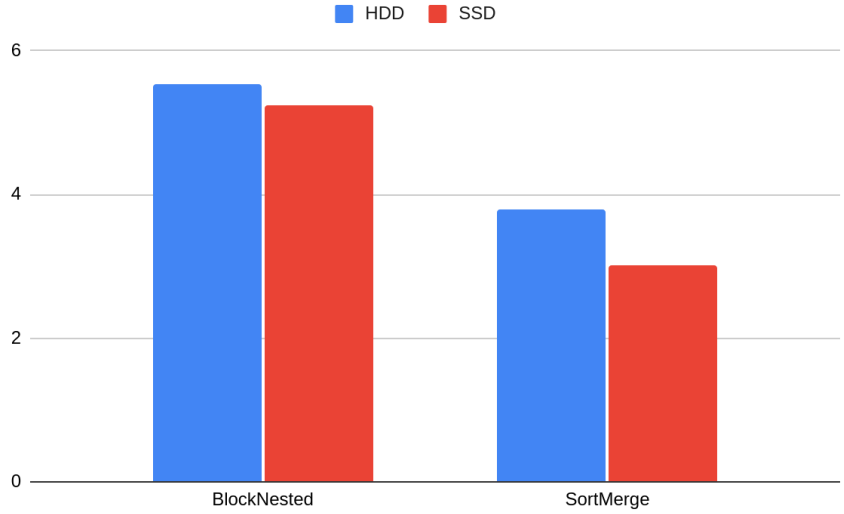
We ran the same 3 experiments on 3 separate laptops (all multi-core i7s, 16 GB RAM, NVMe SSD). The graphs below show the effect of different Operating Systems on the query performance. This was a surprising discovery and we cannot explain why.





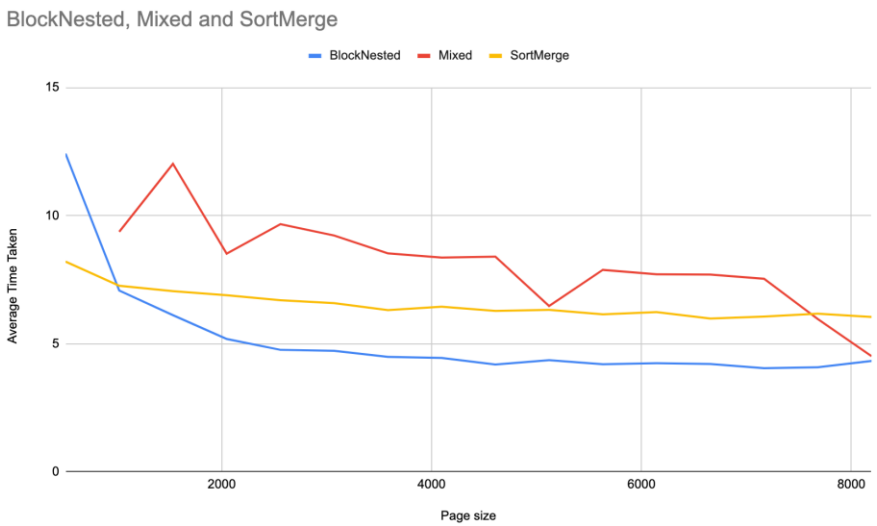
Likewise, the 2 graphs below demonstrate that using a SSD storage device could be faster than using a HDD when running database queries.

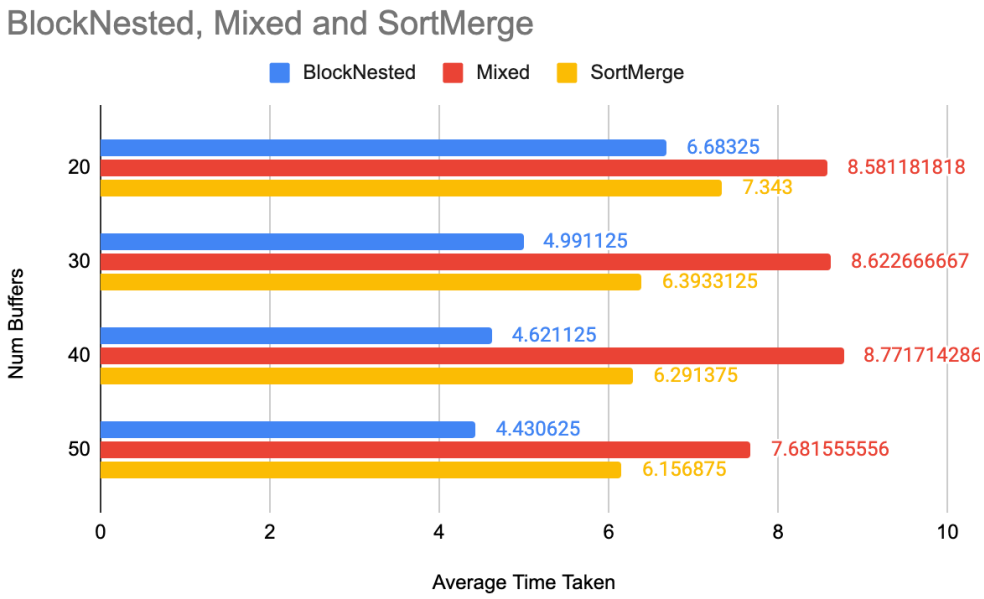




Analysis of Data - Experiment 2

Similar to experiment 1 the graphs below demonstrate that the relationship between pageSize (or NumBufferPages) and average time taken is inversely proportional. Also, from the results below, pure BNJ runs much faster than pure SMJ. In our experiments, even with increased buffer size and page size, the initial number of sorted runs do not change exponentially - which means that the number of passes are generally not changing significantly.





Discussion

In real world test, PostgreSQL takes 0.35ms (planning), 794.32ms (execution) for experiment 2 - which is almost 10x faster than us (even when restricted to BNJ and SMJ). In our tests, the Mixed Query Plans showed little difference when different orders were used. However, this is possibly due to the small size of the dataset. In real world datasets (with millions or even billions of rows), we expect the ordering differences to have a significant impact on query cost - and thus significant impact on total query time. For our implementation, we can have further optimizations done:

1. Using other query optimization techniques such as exhaustive search algo could make the query selection better in some cases.
2. Running Joins in Parallel (if possible)
3. Using more realistic Buffer Size and Page Size (Postgres default is 8GB and 8192B respectively)
4. Tweaking input and output buffer size in SortMerge (currently 1 for each)
5. Experimenting with Replacement Selection Algorithm for generating initial sorted runs.
6. Implementing Hash Join / Index-Based Join to give a bigger search space to the query planner.

Appendix

Experiment 1 Result:
https://docs.google.com/spreadsheets/d/1Bw0FcTWGoKu-LUrXWteUi6lPoxfH9rE0Qrzksd7o5vQ/edit?usp=sharing

Experiment 2 Result:
https://docs.google.com/spreadsheets/d/1U_Q5pWAWe-j7vfd9Rzgx0bPr8ZmnkgZYqbiblIPeCOg/edit?usp=sharing

Query Plan for Experiment 2 (as ran in PostgreSQL, without restrictions on Join Algorithm)