inf-2700-oblig2

Sigurd Uhre

September 2022

1 Introduction

In this assignment the db277 DBMS will be extended with a natural join operation feature. The natural join implementation will be tested by the implementations of the nested-loop join feature and the block nested-loop join feature, the results will be compared and explained. This report will also discuss B+-tree organized files and join algorithms theoretically.

2 Technical background

The Natural join operation operates on two relations and produces a relation as the result. Natural join considers only those pairs of tuples with the same value of those attributes that appear in the schemas of both relations [1].

3 Design and implementation

There are two python programs that will generate a populated table for each program. The number of rows for each table is set to 1000, but can be changed if we want to run larger tables. The rows of the tables will consist of three fields where two will be unique for the table and one will be identical (the id field) for the two tables. The natural join implementation is based upon search and finding of a common field for the two tables. This is done by iterating trough the field for each table an running a test to check if the selected fields match. If they match, the result is set as the "join schema" function. This function copies all the fields from the two tables and create a new schema. This new schema created is given as the "result" argument for the *nested-loop join* and *block nested-loop join* operations.

The *nested-loop join* implementation is done by using the programs page functionality to set up an outer and inner table iteration process. The first for-loop starts with the first value of the outer table and then iterating over all the values in the inner table while comparing the values each time to check for a match. If the values is a match, those recodrs are joined and a new table is appended. The position is updated and the outer relation will do the same process for the next value in line.

Wile the nested-loop join algorithm iterates each tuple starting from the outer relation $Tr * Ts$ (Tr = outer relation tuple, Ts = inner relation tuples). The *block nested-loop join* starts the iterations from blocks. Records per block and blocks is calculated before the iteration can start. In the first iterations, every block of the inner relation is paired and compared with every block of the outer relation. Within each pair of blocks, every tuple (records) in one block is paired with every tuple in the other block , to generate all pairs of tuples. All pairs of of tuples that satisfy the join condition are added to the result with the join records and append records functions.

When a match is found by either the *nested-loop join* algorithm or the *block nested-loop join* algorithm, each function will call the "join records" function. This function preforms the actual join of the matched records by iterating both schemas and assign the field that matches the relation. The append record function append the record to the new table file, containing the values from the matched relation as well as their unique attributes.

4 Results (Task 2)

To compare the *nested-loop join* algorithm against the *block nested-loop join* algorithm a specific querie is run: "select * from workers natural join person where income > 995;". This selection is tested on different records-sizes for the two tables to illustrate how the methods block nested-loop join and nested-loop join handles an increasingly larger record size to match-up.

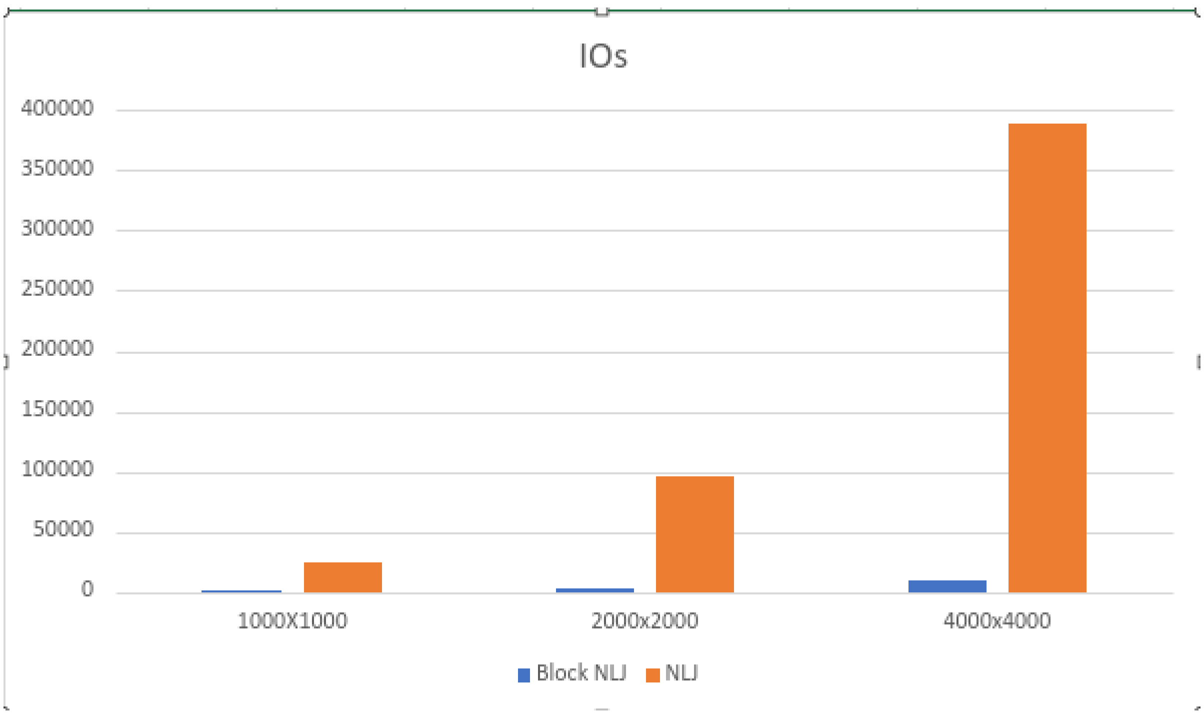


Figure 1: IOs operations for nested-loop join (orange) and Block nested-loop join (Blue). Y-graph is IOs operations and X-graph is record sizes.

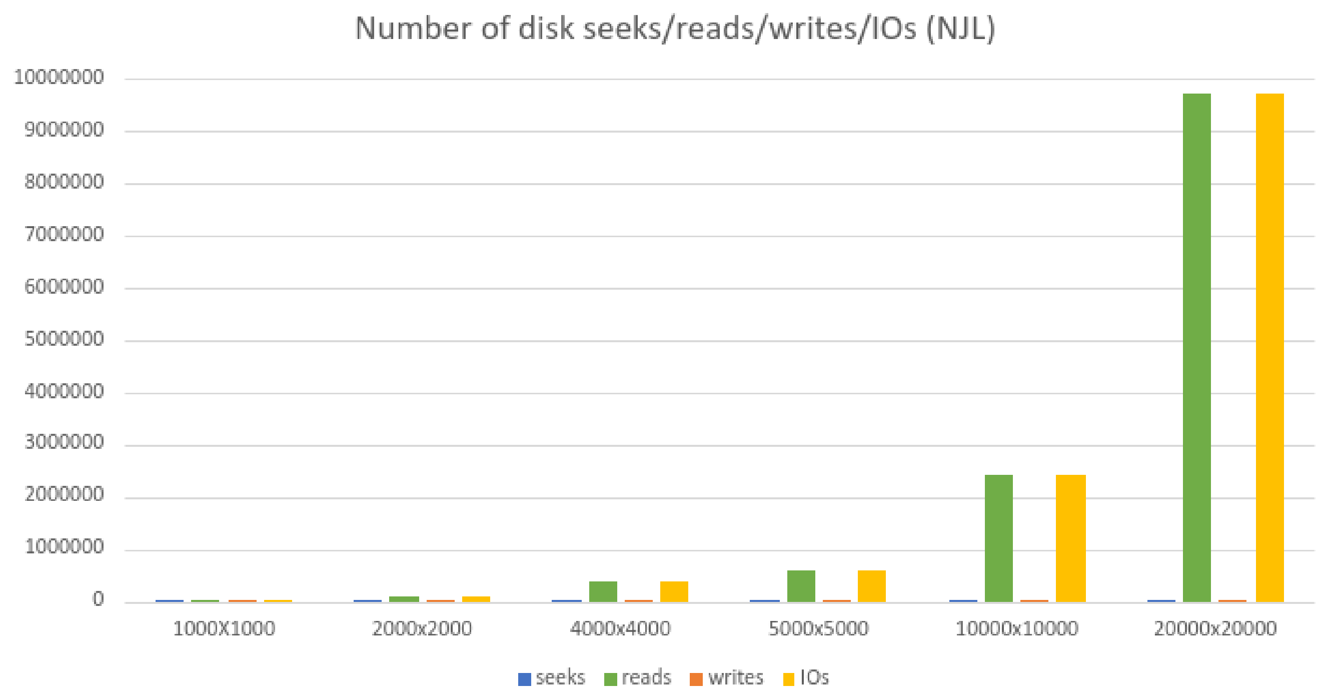


Figure 2: Number of disk operations for nested-loop join . Y-graph is number of operations and X-graph is record sizes.

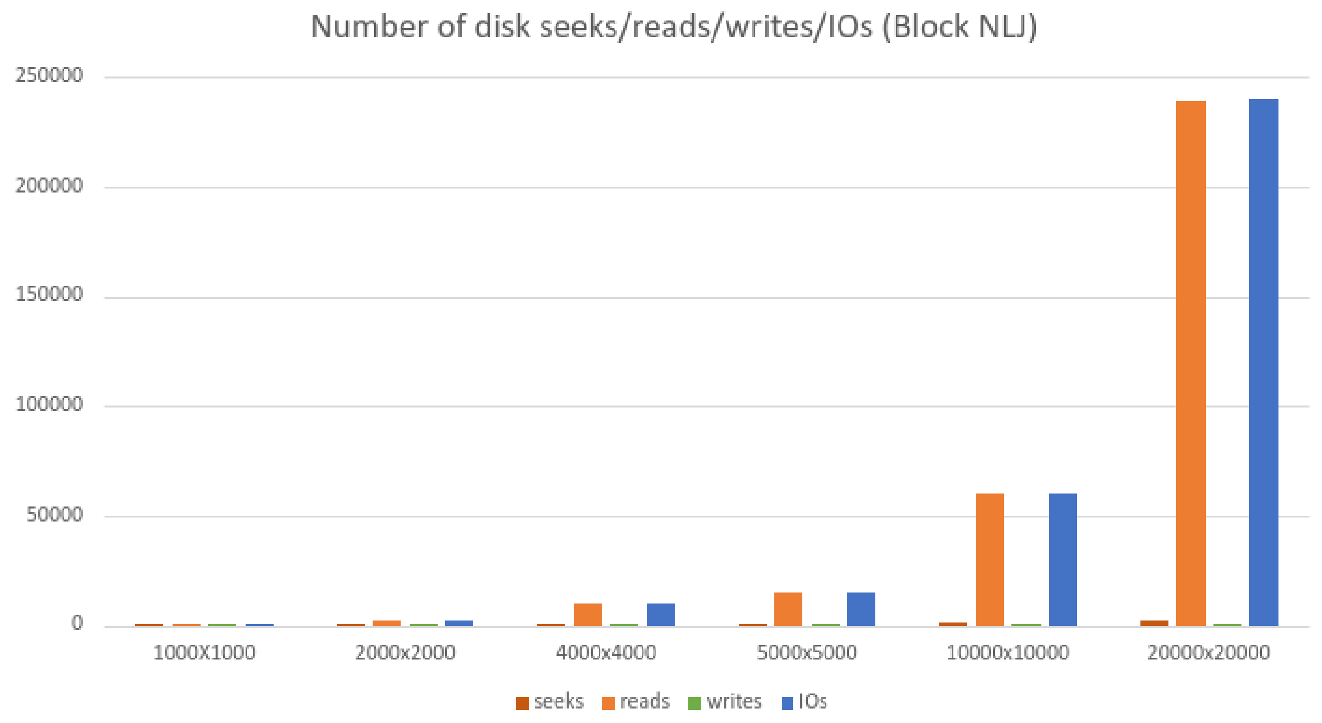
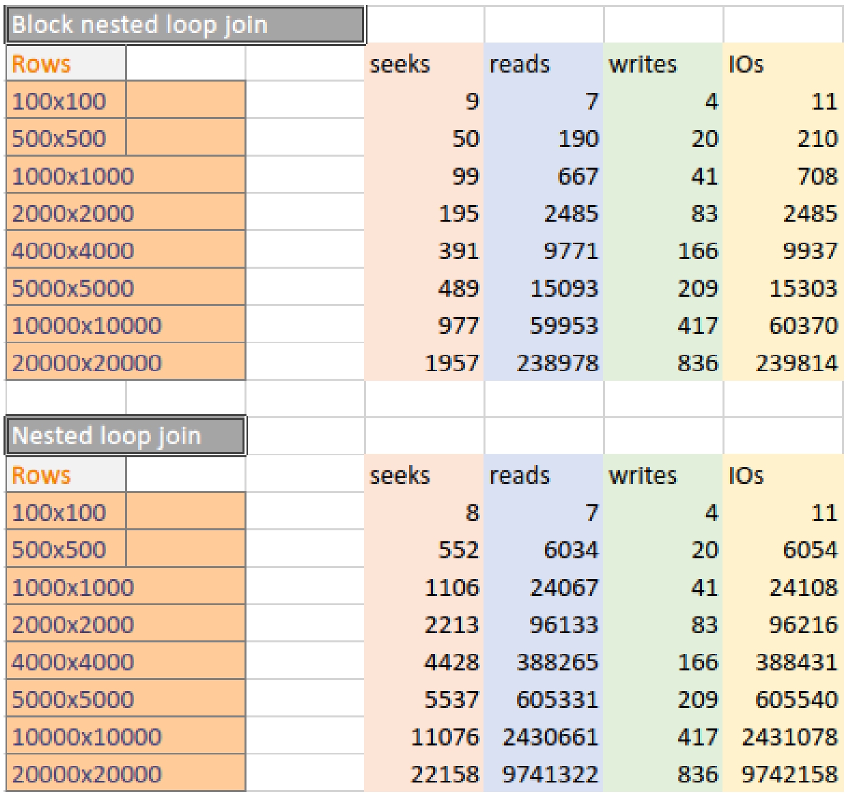


Figure 3: Number of disk operations for Block nested-loop join . Y-graph is number of operations and X-graph is record sizes.



Block nested loop join				
Rows	seeks	reads	writes	IOs
100x100	9	7	4	11
500x500	50	190	20	210
1000x1000	99	667	41	708
2000x2000	195	2485	83	2485
4000x4000	391	9771	166	9937
5000x5000	489	15093	209	15303
10000x10000	977	59953	417	60370
20000x20000	1957	238978	836	239814

Nested loop join				
Rows	seeks	reads	writes	IOs
100x100	8	7	4	11
500x500	552	6034	20	6054
1000x1000	1106	24067	41	24108
2000x2000	2213	96133	83	96216
4000x4000	4428	388265	166	388431
5000x5000	5537	605331	209	605540
10000x10000	11076	2430661	417	2431078
20000x20000	22158	9741322	836	9742158

Figure 4: Number of disk operations adjusted for record size for Block nested-loop join (top) and Nested-loop join (bottom)

5 Discussion

Figure 1 illustrates the difference in performance between nested-loop join and block nested-loop join for IOs operations at different record sizes. The difference in performance is significant already from the lowest record size of 1000x1000. Here, block nested-loop performs significantly better than nested-loop join with 708 operations versus 24,108 operations respectively. At this number of records, nested-loop join has 34 times more IOs operations compared to block nested-loop join.

This trend continues when we double the number of record sizes for each of the tables and at 4000x4000 record size, block nested-loop join has 9,937 operations against nested-loop join 388,431 operations. At this number of records, nested-loop join has 39 times more IOs operations compared to block nested-loop join. This shows that although the difference is large from the smallest record size, the distance in operations increases as the number of records increases.

Nested-loop join is an expensive algorithm since it examines every pair of tuples in the two relations. The number of pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r and n_s denotes the number of tuples in s. Relation r being the outer relation and relation s being the inner relation of the join. For each record in r, we have to performe a complete scan on s [1].

For nested-loop join the best case scenario is if there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once: hence, only $b_r + b_s$ block transfers would be required, along with two seeks. In table 4, we can see how nested-loop join performs with a small number of records (100x100), illustrating a best case scenario. In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block transfers wold be required, where b_r and b_s denote the number containing tuples of r and s, respectively [1].

Figure 2 shows how the number of disk seeks/reads/writes and IOs evolves as the number of record sizes increases for the next-loop join. The number of seeks and writes also increases, but does not appear clearly in the figure as they increase considerably less than reads and IOs. The figure shows that reads and IOs increase in step with each other as the record size increases. Figure 3 shows the same pattern for the block nested-loop join, only that here the number of operations is significantly smaller for the same record size.

The block nested-loop algorithm have a worst case of $b_r * b_s + b_r$ which differ from nested-loop algorithm because each block in the inner relation s is read once for each bock in the outer relation, instead of once for each tuple in the outer relation. This makes the block nested-loop algorithm to perform much better when the number of relational records is to high to be held in memory [1].

5.1 Task 3

In a $B^{+}tree$ file organization, the leaf nodes of the tree store records, instead of storing pointers to records. The $B^{+}tree$ file organization is sorted and one algorithm that can make use of this file organization is the merge-join algorithm (sort merge join). This algorithm can be used to compute natural joins.

There are two stages to the merge-join algorithm, the first stage is to sort both relations R and S by join key (the common attribute of the tables). This is going to set all the tuples in each relation (with this key) together in consecutive order. The second stage is a join pass, this is a merge-scan of the two sorted relations to be matched (which is already sorted on disc) and then merge them together emitting tuples that match.

Once the relations are in sorted order, tuples with the same value on the join attribute are in consecutive order, thereby, each tuple in the sorted order needs to be read only once, and, as a result, each block is also read only once. If all the sets S_s fits in memory, the algorithm will only make a single pass trough both files, making it a efficient algorithm. The number of blocks transfers is equal to the sum of the number of blocks in both files $b_r + b_s$ [1].

For the best case scenario for block nested-loop where the inner relation fits in memory, there will be also be a $b_r + b_s$ block transfer for this algorithm. Thus, in the worst case scenario, there will be a total of $b_r * b_r + b_s$, block transfers, where b_r and b_s , denote the number if blocks containing records of r and s, respectively [1].

What is important to take into account when discussing the cost and performance of merge-join is if the merging relationships are sorted in advance, if not, then they must be sorted, which entails additional costs. The cost may be something like this: Sort Cost ($\mathbf{R}$) : $2M * (1 + (log_B - 1(M/B)))$ and ($\mathbf{S}$) : $2N * (1 + (log_B 1)(N/B)))$. The merge cost will then be $M + N$ and total cost will be $Sort + Merge$ [2]. We can use an example to illustrate this. Lets say the database of table R and S where:

Table R: M = 1000 pages, m = 100,000 tuples
Table S: N = 500 pages, n = 40,000 tuples

- With B=100 buffer pages, both R and S can be sorted in two passes

Sort Cost (R) $= 2000x(1 + (log_{99}1000/100) = 4000IOs$
Sort Cost (S) $= 1000x(1 + (log_{99}500/100) = 2000IOs$
Merge Cost $= (1000 + 500) = 1500IOs$
Total cost $= 4000 + 2000 = 7500IOs$[2]

The worst case for the merging phase is when the join attribute of all of the tuples in both relations contain the same value. The cost will then be $(M + N)+$ Sort cost [2]. If we compare this data with Figure 4, we can see that the block nested loop at 1000x1000 records (pages) has 708 IOs operations. This is significantly less than the merge-join algorithm both with and without sorting costs and fewer pages.

If we assume that the relations are already sorted and that they are organized in a $B^{+}tree$ file system. The block nested-loop algorithm still needs to compare every record in the inner relation with the outer relation. The merge-join algorithm does not need to sort the relations first. When the number of records (pages) extend a certain size, we would expect merge-join algorithm to perform better given that both algorithms are testes where the relations are already sorted and that they are organized in a B^+tree file system.

6 Conclusion

This report describes the implementation of the natural join with nested-loop join and block nested-loop join methods. Two programs are used to generate different sized databases and the different methods are tested with different sized databases. The results are illustrated, discussed and debated in the report together with a theoretical discussion of how merge-join will perform compared to block nested-loop join in a B^+tree organized file system.

7 References

1. Silberschatz, A., Korth, H.F. and Sudarshan, S. (2020) "Chapter 15 Query Processing," in Database system concepts. New York: McGraw-Hill Education, pp. 704–709.

2. Arulraj, J. (no date) Homepage, GT 4420/6422. Available at: https://faculty.cc.gatech.edu/ jarulraj/courses/4420-f20/ (Accessed: November 1, 2022).