Catherine Horng ch756, Alison Duan ad734, Kristin Lee kjl92
CS 4321, Project 5

Parallelization Implementation and Experimentation

Implementation:

ScanOperator: ScanOperator parallelizes reading in the tuple files by turning the read into a Stream, then parallelizing the printing of that stream in the dump() function.

IndexScan: Parallelized the generation of TupleIdentifiers for faster search. For every key position, all of the potential tuple positions from 0 to the maximum number of entries have TupleIdentifiers that are found and then pushed to the stack. Like ScanOperator, IndexScan also parallelizes the dump() function in the same way.

SelectOperator: SelectOperator maintains an internal buffer of at most $n=100$ tuples that only contain tuples that satisfy the SelectOperator's expression. When getNextTuple() is called, it pops the first tuple from the buffer. If the buffer is empty, then it is replenished by getting n tuples from the child and adding each tuple to the buffer if it satisfies the condition by means of parallel streams. If the child runs out of tuples, after the last tuple in the buffer is popped, getNextTuple() returns null. This way, checking whether each tuple satisfies the condition is done in parallel.

ProjectOperator: ProjectOperator maintains an internal buffer of n tuples that only contain columns selected by the ProjectOperator. When getNextTuple() is called, it pops the first tuple from the buffer. If the buffer is empty, then it is replenished by getting n tuples from the child and projecting each of the tuples in parallel by means of parallel streams. If the child runs out of tuples, after the last tuple in the buffer is popped, getNextTuple() returns null.This way, projecting each tuple to only selected columns is done in parallel. We chose $n = 10$.

BlockNestedJoin: BlockNestedJoin maintains an internal buffer of at most $n=100$ tuples that only contain tuples that satisfy the BNLJ Operator's expression. When getNextTuple() is called, it pops the first tuple from the buffer. If the buffer is empty, then it is replenished by getting n tuples from the right child and adding each tuple to the buffer if it satisfies the condition by means of parallel streams. If the right child runs out of tuples, after the last tuple in the buffer is popped, the right child is reset before starting the loop again. If the end of the outer loop is reached, getNextTuple() returns null. This way, checking whether each tuple satisfies the condition and can be joined is done in parallel.

SortMergeJoin: SortMergeJoin is parallelized by parallelizing the joinCmp class within the SMJ Operator. The joinCmp object maintains two lists of schema orders; one for the left tuple and one for the right tuple. These are populated by adding the join columns of interest to the lists. This is done through a parallel stream over the range of columns.

ExternalSort: We parallelize ExternalSort by parallelizing the runs in each $pass_i$ where $i > 0$. On every $pass_i$, there are a number of runs that need to be merged by means of k-way merge. Because these runs do not rely on other runs in the same pass, they can be done in parallel by means of parallel stream. Additionally, the merge of the mergeSort algorithm is done in parallel.

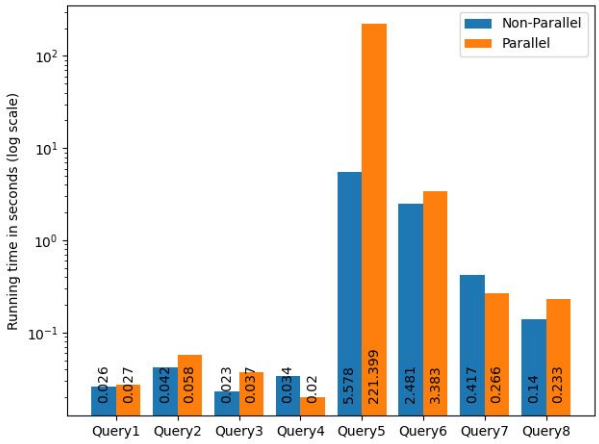
DuplicateEliminationOperator: DuplicateEliminationOperator maintains an internal buffer of at most $r \times n$ tuples that only contain unique values. When getNextTuple() is called, it pops the first tuple from the buffer. If the buffer is empty, then it is replenished by getting r runs of n tuples, removing duplicates from each run in parallel, and merging each run while checking for duplicates on the edge of each run. If the child runs out of tuples, after the last tuple in the buffer is popped, getNextTuple() returns null.This way, removing duplicate tuples is done in parallel. We chose $r = 10$ and $n = 10$.

Note: We did not parallelize physical operators SortOperator or JoinOperator as they are no longer used in our implementation.

Performance Comparison: Data was generated with the following description:

Relation:	*# Tuples:*	*Attributes:*	*Range (chosen uniformly at random):*
Candy	5000	A	0-10
		B	0-100
		C[1]	0-1000
Pudding	5000	D	0-100
		E	0-10
		F	0-1000
Berries	5000[2]	G	0-10
		H	0-100

Queries:	*Parallel PhysicalOperators Used:*
SELECT * FROM Candy;	ScanOperator
SELECT Candy.A, Candy.B FROM Candy;	ProjectOperator
SELECT * FROM Candy WHERE Candy.A < 9;	SelectOperator
SELECT * FROM Candy WHERE Candy.C > 999;	IndexScan
SELECT * FROM Candy, Pudding WHERE Candy.A < Pudding.E;	BlockNestedJoin
SELECT * FROM Candy, Pudding WHERE Candy.A = Pudding.E;	SortMergeJoin, ExternalSort
SELECT * FROM Candy ORDER BY Candy.B, Candy.A;	ExternalSort
SELECT DISTINCT * FROM Berries;	DuplicateEliminationOperator, ExternalSort



Queries were chosen such that they only test different individual (or at most two) operators at once. In almost all cases, parallelization leads to an increase in time taken per query.

Query 1 tests the ScanOperator and shows a slight increase in time taken. This is likely due to some overhead in utilizing streams and experimental error.

Query 4 tests the IndexScan and shows a decrease in time taken, since it reduces sequential time needed to calculate TupleIdentifiers.

Queries 2, 3, 5, and 8 test the ProjectOperator, SelectOperator, BlockNestedJoin, and DuplicateEliminationOperator and show an increase in time taken. Because these operators maintain an internal buffer and are required to check the buffer's condition in addition to adding and removing from the buffer, there is an increased overhead for parallelization and thus an increase in time taken. For DuplicateEliminationOperator, there is also overhead in merging runs of unique tuples but benefits from the use of ExternalSort.

Query 6 tests SortMergeJoin and ExternalSort. For SortMergeJoin, there is an increased overhead in maintaining the lists of column orders. Additionally there is overhead in getting the range of indexes over which to get the join columns. However it does benefit from the use of ExternalSort.

Query 7 tests the ExternalSort in which there is a decrease in time taken. This is because the bulk of the algorithm spends time in the sorting of each run as well as in merge-sort. Because the runs are now done in parallel and there are multiple runs per pass, there is a decrease in time taken.

[1]*There is an available clustered index on Candy.C.*
[2]*Berries was generated with 2500 tuples with the above descriptions then duplicated for a total of 5000 tuples to ensure duplicates.*