Project 1 Report

Task 1: Run Length Encoding

Subtask 1.A: Implement a scan for RLE data (5%)

For this task, a simple solution was employed — read in all the RLE columns, decompress everything and turn columns into tuples. Return one tuple each time next() is called.

Subtask 1.B: Execution on RLE data (25%)

Subtask 1.B.i: Implement RLE primitive operators

- Decode: for each RLE entry, repeat *length* times, append to a collection of tuples and output one tuple each time.

- Reconstruct:

(i) submitted solution: Decompress everything and then reconstruct RLE entries and output them one by one.

(ii) more efficient solution but didn't manage to implement correctly in time:
While both left and right has next:

- compare the left input RLE entry *l* and the right input RLE entry r
- take *max(l.startVid, r.startVid)* as the *newStartVid*, take *min(l.endVid, r.endVid)* as the *newEndVid*.
- *newLength* = *newEndVid* – *newStartVid* +1. If *newLength* > 0*,* then we can reconstruct a new RLE entry RLEentry(newStartVid, *newLength, l.value++r.value*)
- Now, since we have compared up to the *min(l.endVid, r.endVid),* we can fetch the next from the side which has smaller end vid.s

Subtask 1.B.ii: Extend relational operators to support execution on RLE-compressed data

Similar to project 0, aggregate and join is hash table based, sort is priority queue based. The main difference is when handling a RLE entry, we often need to repeat *length* times.

- In *aggregate*, to get arguments for reducing on a group, we call getArgument() *length* times.
- In join, when joining two matching RLE entries, we return a new RLE entry with length equal to the product of the two RLE entries.
- Project and filter are very similar to project 0.

Task 2: Query Optimisation Rules (35%)

In the tree, a parent operator is constructed by passing child operator(s) to it as parameter. Once this recursive structure is understood, this part is straightforward to implement.

Task 3: Execution Models (35%)

Subtask 3.A: Enable selection-vectors in operator-at-a-time execution

Implementations of operators in this task are similar to project 0. They have the following pattern: transpose columns to tuples, process, transpose tuples back to columns. Aggregate and join are based on hash table. Sort is based on priority queue.

Subtask 3.B: Column-at-a-time with selection vectors and mapping functions

- For filter, use the mappredicate() function to produce a boolean vector. Update the new selection vector to be such that a tuple has corresponding Boolean value true iff it was active and can pass the filter.
- For project, apply each eval function on all input homogenous columns to obtain one output homogenous column.
- Other implementations are all very similar to subtask 3A, including hash based join and priority queue based sort. The only difference is conversion between column and homogeneous column.