Writeup Lab4

IntHistogram

I use an array `buckets` to store each bucket representing the number of records in a fixed range of the domain of the attribute of the histogram, as README told us to do.

There are two details worth to mention in my implementation:

1. As I set the bucket's width as integer, the last bucket's width may be different from the others, so I write a private function `getBucketWidth(int)` to calculate the width.
2. In the `estimateSelectivity(Op, int)` function, for convenience, I leverage case `EQUALS`, `GREATER_THAN` and `LESS_THAN` to implement other cases. However, in case `GREATER_THAN_OR_EQ`, if I just add `GREATER_THAN` and `EQUALS` up, there will be loss of accuracy (about 3%) and will not pass TableStatsTest. So I change it to `estimateSelectivity(Predicate.Op.GREATER_THAN, v-1)` and the problem is solved. The same case is in `LESS_THAN_OR_EQ`.

TableStats

I use a HeapFile `table` to link to the table which has the given tableid, and two `ConcurrentHashMap` to store the mapping from TupleDescription's field index to the corresponding Histograms.

The constructor contains two scans:

1. In the first scan, I get every field's value range (if the type is integer) and build histogram tables.
2. In the second scan, I add values into histogram tables.

The other parts are trivial.

JoinOptimizer

If we perform a join operation, we scan the left table (need cost1), for every row in left table, we need to scan the right table once (need card1*cost2), when we scan each row in right table, we need to perform the predicate operation to check if it is satisfied (need card1*card2). Then the `estimateJoinCost` function is trivial.

For `estimateTableJoinCardinality`, we need to follow the README:

1. For equality joins, when one of the attributes is a primary key, the number of tuples produced by the join cannot be larger than the cardinality of the non-primary key attribute.
2. For equality joins when there is no primary key, we make up a simple heuristic (the larger table of the two).
3. For range scans, assume that a fixed fraction 30% is emitted.

For `orderJoins`, I just use Selinger optimizer as the README given, with the two function `enumerateSubsets` and `computeCostAndCardOfSubplan`, the implementation is trivial.