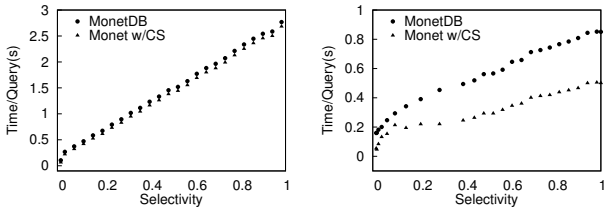

(a) TPC-H Q1 (b) TPC-H Q6

Figure 15: Column Sketches improve scan performance on TPC-H queries

Ordered Column Sketches. In the case that we have a number of consecutive codes that are non-unique, the optimal solution tries to place partition points into the sorted list of values such that

$$g(c) = q_c \left(1 - (1 - f_c)^{\frac{M_g}{B_b}}\right)$$

is relatively equal for all codes. Given a sorted list of n values with q_i, f_i values for each, this can be done in order of the values by deciding which partitions shift their endpoints over by a code. For instance, if have 256 partitions, upon reading a value v, the algorithm could decide to shift partitions 130-256 over by a single value, and leave the endpoints of partitions 101-129 unchanged. The decision of which code gains a new value is made greedily; for every new value which we examine, the code which sees the smallest increase in g by accepting a value does so.

We start the procedure for order-preserving Column Sketches by assuming all C codes are non-unique and performing this procedure, taking time $O(nC)$. We keep track of the value in each code C which has the largest result for $g(v)$. We then proceed to examine the value in each code c with largest $g(v)$. If giving v the unique code c and shifting the other values into codes $c-1$ and $c+1$ produces a decrease in

$$\sum_{c \notin U} q_c \left(1 - (1 - f_c)^{\frac{M_g}{B_b}}\right)$$

then we give v unique code c. Otherwise, we do not. Additionally, this procedure is again barred from giving consecutive codes unique values, and cannot give the first or last codes unique values.

G SHIFTING DOMAINS

In order to retain robust performance, the compression map needs to retain a relatively even number of values assigned to each code. The simplest way to do this is to count the number of values assigned to each code, and re-encode when some code is both non-unique and has too large a proportion of the dataset. Counting the number of values assigned to each code is easily done, as it can be done in one pass as data is ingested or as the Column Sketch is created. Additionally, this doesn't take too much space as tracking the number of values assigned to each code requires a single integer per code. Since the number of codes is usually quite small, this is a small memory overhead.

When re-encoding the Column Sketch, the process is similar to its original creation and involves a sampling phase followed by an encoding of the base column. Because the Column Sketch is a secondary index, this process can be done in the background and does not halt query processing. Additionally, the database can choose to use the pre-existing Column Sketch in the interim, or it can drop it immediately.

Finally, we look at the case of clustered data such as date columns and similar. First, we note that columns with clustered data usually do well with lightweight indices such as Zone Maps or Column Imprints, and so there exist prior techniques which better suit these scenarios. However, Column Sketches should retain there robust behavior in these scenarios and so we provide two ways to deal with correlated data. The first is to perform horizontal partitioning per some amount of data. The second technique is to run a regression on column order and column position at the initial creation time of a Column Sketch. If the correlation is high, then we leave some number of codes at the end of a Column Sketch empty. The number of codes to leave empty, e, is a tunable parameter. The Column Sketch scan will perform on average like each non-unique code has $\frac{1}{256-e}$ of the data, and the Column Sketch will need to re-encode the Column Sketch every time the base data reaches $\frac{256}{256-e}\times$ its prior size.

H TPC-H

To run TPC-H queries we integrated Column Sketches into MonetDB. The results support the main observations from the synthetic workload experiments, with Column Sketches providing a large boost in scan performance.

To perform the integration in MonetDB, we introduced a new select operator for Column Sketches that takes as input the same API as the standard MonetDB select, i.e., a single column and a predicate. However, instead of the usual output of MonetDB which is a position list, Column Sketches outputs a bitvector. Thus, we also wrote a new fetch operator to use instead of the original fetch operator in MonetDB that works with position lists. The rest of the operators used in the TPC-H plans are the original MonetDB operators.

Furthermore, we did not do any changes in the MonetDB optimizer to use those new operators automatically. Instead, we took the plans created from the explain command in MonetDB and edited those plans to use Column Sketches. Then we fed those plans directly into the MAL interface of MonetDB. This means our results do not include the optimizer cost, and so for fairness we also remove this cost from MonetDB.

Overall, we setup these experiments by varying selectivity and we compare plain MonetDB against MonetDB with ColumnSketches (Monet w/CS) enabled. We use TPC-H scale factor 100 and provide performance experiments against query one (Q1) and query 6 (Q6) of TPC-H. The results can be seen in Figure 15.

For both queries, Column Sketches improves on the scan performance of MonetDB. In Q1, this improvement is less apparent as the majority of the time spent in query execution is spent doing aggregation. In contrast, Q6 sees a much larger performance improvement as much more time is spent doing predicate evaluation. For both queries, this improvement is roughly constant across all selectivites. As a percentage of query time, the improvement is largest for low selectivities and smaller for higher selectivities. The improvement for Q1 ranges from 19% at 1% selectivity to 3% at 98% selectivity. The improvement for Q6 ranges from 54% at 1% selectivity to 41% at 98% selectivity.