Quantized Multi-Columnar Predicate Evaluation

Zachary Yedidia
Harvard College

Bagatur Askaryan
Harvard College

Zachary Dietz
Harvard College

Brian Hentschel
Harvard University

ABSTRACT

We demonstrate three different methods of evaluating multi-attribute predicates using quantized sketches of columns: Lookup Tables (LUT), Range Lookup Tables (Range LUT), and Quantized Arithmetic (Q-Arithmetic). We empirically measure the speedups of each of these quantized methods over the baseline and find the improvement is inline with what we would expect. Our fastest method achieves between **2.8 − 4.8X speed up** over the baseline for uniformly random signed 32-bit integer data. We explore empirically the relationship between performance and data size and show that our approaches have good scalability across many threads.

ACM Reference Format:
Zachary Yedidia, Bagatur Askaryan, Zachary Dietz, and Brian Hentschel. . Quantized Multi-Columnar Predicate Evaluation.

1 INTRODUCTION

Motivating question: Can we speed up selects in column-stores, and in particular those with multi-attribute predicates, by using quantized sketches of the data?

Many companies and researchers today have large data sets that they wish to perform timely and accurate analysis on. Internet companies are one such example, but many others exist such as in the domain of finance, automakers, meteorology, astronomy and more. Achieving state of the art analytic performance means maintaining up-to-date data structures that are fast to build and require minimal space. A common solution is to use fixed-width columnar storage which allows for easy compression and light-weight zone maps. In this paper, we consider another approach which relies on a lossily, quantized compression scheme. Using this compressed version of the data, we can answer queries in a relational database and only need to use the larger, losslessly compressed data when we cannot solve the query accurately with the lossily compressed data alone. Through this approach, we hope to develop methods to evaluate predicates much faster than could be achieved with solely operating over the base data.

1.1 Compression Scheme

In this paper we only consider predicates over two columns, x and y, although we believe the methods we explore could be easily generalized to handle predicates over an arbitrary number of columns.

Our compression scheme to send x and y into lossily compressed codes with $qbits$ bits is

$$x_i' = \lfloor \frac{x_i - x_{min}}{x_{max} - x_{min}} \cdot 2^{qbits} \rfloor , y_i' = \lfloor \frac{y_i - y_{min}}{y_{max} - y_{min}} \cdot 2^{qbits} \rfloor \ \forall i$$

Because x and y are generated from the same distribution, $x_{max} - x_{min} = y_{max} - y_{min}$ for our experiments and thus x and y are quantized in the exact same way.

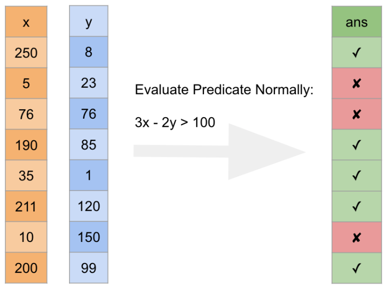
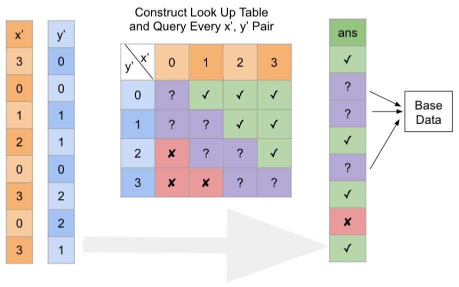


Figure 1: The basic approach to evaluate a multicolumn predicate uses both columns of the base data.

2 LOOKUP TABLES

Our first approach for a general method of doing multi-column predicate evaluation over quantized data was to generate a lookup table with an entry for every combination of quantized value. The entry specifies whether the predicate would evaluate to true, false, or uncertain if evaluated with the given quantized values. When scanning multiple quantized columns with a predicate, it is then enough to look into the table with the quantized values as an index to determine if a tuple is true, false, or uncertain.



y' \ x'	0	1	2	3
0	?	✓	✓	✓
1	?	?	✓	✓
2	✘	?	?	✓
3	✘	✘	?	?

x'	y'	ans
3	0	✓
0	0	?
1	1	?
2	1	✓
0	0	?
3	2	✓
0	2	✘
3	1	✓

Figure 2: LUT approach generates a look-up table on-the-fly for each query and uses this with the quantized data to evaluate the predicate.

This approach is quite general and can easily extend to more than two columns, and a wide variety of predicates, but has performance limitations. The main limitation is the size of the table. With an N-bit quantization, and a predicate with k variables, the table will have size $O(2^{Nk})$ bits. Each entry can have three possible values, and can therefore be encoded in 2 bits for maximum space efficiency. However packing the bits like this causes unpacking overheads during the scan. For a typical 8-bit quantization and two-column case, the table will be roughly 65kb in unpacked form. This is small enough to fit in the cache of a modern processor, but still large enough that the constant random access becomes a bottleneck.

2.1 Range Lookup Table

There is a better method which reduces the size of the lookup table, which we call the range lookup table. We found that for evaluating functions we only need to store the uncertain range for each possible quantization. For example, in a two-variable predicate $f(x, y)$, if the predicate is a mathematical function, then there must be one contiguous range of uncertain values in the y dimension for each quantized x value. The start and end of this range are each N-bit values and can be stored in a table. This time however, the table will only have size $O(2N2^N)$ bits for the two-variable case. For further variables, space in the table can be traded for restrictions to the function. For an 8-bit quantization, this is a much more reasonable 512 bytes, and packing/unpacking is not a problem since information is not encoded as trits. The range lookup table must also store one additional boolean determining the direction of the inequality (whether the predicate is true/false below/above the uncertain range)

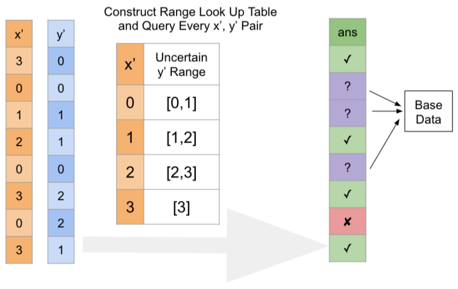


Figure 3: Range LUT follows the same approach as that of the LUT but stores the uncertain range for y' for each possible x' as to yield a smaller LUT.

2.2 Lookup Table Construction

Constructing the lookup table can present additional restrictions on the kinds of functions that can be represented. Since we are specifically analyzing predicates of the form $ax + by > c$, our lookup table construction assumes that the function is monotonic. For every combination of two quantized values it evaluates the predicate using the four edges of the ranges represented by those quantizations. If any of the four evaluations do not match, the entry is considered uncertain. In our experiments, the lookup table construction time was insignificant (on the order of microseconds).

We also thought about a more complex construction algorithm that we did not implement but that would work for arbitrary mathematical functions. The algorithm is as follows: Given an x and y range, find the minimum and maximum y values of the predicate in the x range. If this min-max range overlaps with the y range, then the entry is uncertain. Otherwise the value of the entry depends on the direction of the inequality. This method only works if the minimum and maximum can be found quickly, which may mean that the derivative of the predicate is necessary for fast construction.

3 QUANTIZED ARITHMETIC

The third approach is to perform operations directly on the quantized sketches, without any lookup tables. That is, given quantized elements we compute lower and upper bounds on the value of the desired function and use these to determine if the predicate is true, false, or uncertain. Below we show how to use this approach for predicates which use linear combinations of attributes. We demonstrate how to solve a two-variable linear combination, but our results easily generalize to any linear combination. We leave it for future work to work out the lower and upper bound functions for other classes of functions.

3.1 Linear Combinations — Bounding Functions

For any $x, y \in \mathbb{N}$, let us start with the simple quantization scheme $x' = \left\lfloor \frac{x}{n} \right\rfloor$, $y' = \left\lfloor \frac{y}{n} \right\rfloor$. For any $a, b, c \in \mathbb{N}$, we would like to evaluate the linear predicate $ax + by > c$ using only x' and y'. Note that:

$$\left\lfloor \frac{x}{n} \right\rfloor \leq \frac{x}{n} < \left\lfloor \frac{x}{n} \right\rfloor + 1$$
$$\Leftrightarrow n\left\lfloor \frac{x}{n} \right\rfloor = nx' \leq x < n(x'+1) = n\left\lfloor \frac{x}{n} \right\rfloor + n.$$

For $a > 0$ we then have:

$$anx' \leq ax < an(x'+1),$$

and for $a < 0$:

$$an(x'+1) < ax \leq anx'.$$

Combining these gives us:

$$an(x' + (1 - \mathbb{1}[a > 0])) \leq ax \leq an(x' + \mathbb{1}[a > 0]).$$

The same analysis can be done for y, meaning the linear combination is bound by:

$$\begin{aligned} &an(x' + (1 - \mathbb{1}[a > 0])) + bn(y' + (1 - \mathbb{1}[b > 0])) \\ \leq &ax + by \\ \leq &an(x' + \mathbb{1}[a > 0]) + bn(y' + \mathbb{1}[b > 0]). \end{aligned}$$

That is, we get lowerbound:

$$L = an(x' + (1 - \mathbb{1}[a > 0])) + bn(y' + (1 - \mathbb{1}[b > 0])) \leq ax + by,$$

and upperbound:

$$U = an(x' + \mathbb{1}[a > 0]) + bn(y' + \mathbb{1}[b > 0]) \geq ax + by.$$

If $L > c$ then it is certain that the predicate is true, since:

$$ax + by \geq L > c,$$

and if $U < c$ then it is certain that the predicate is false, since:

$$ax + by \leq U < c.$$

If neither condition holds then we cannot be certain about the result and must look at the base data x and y directly. [With some additional bookkeeping we could sometimes make the upper and lower bounds strict inequalities. If either a or b is positive, then the lower bound is strictly less than ax+by, and if a or b is negative than the upper bound is strictly greater.]

Thinking about implementation, multiplying our quantized values by n forces us to cast our quantized values to larger data types. A better approach would be to quantize c instead. Let $L' = \lfloor L/n \rfloor = L/n = a(x' + (1 - \mathbb{1}[a > 0])) + b(y' + (1 - \mathbb{1}[b > 0]))$ and similarly let $U' = \lfloor U/n \rfloor = U/n$ and $c' = \lfloor c/n \rfloor$. We show that $L' > c' \Rightarrow L > c$ and $U' < c' \Rightarrow U < c$:

$$\text{Lower bound: } L' > c' \Leftrightarrow \lfloor L/n \rfloor > \lfloor c/n \rfloor \Rightarrow L > c.$$
$$\text{Upper bound: } U' < c' \Leftrightarrow \lfloor U/n \rfloor < \lfloor c/n \rfloor \Rightarrow U < c.$$

Ideally these relations would go both ways so that no information is lost in this more space-efficient method. While this is the case for the lower bound, for the upper bound only the weak inequality is equivalent. This is still useful when we know U is strictly greater than $ax + by$, which happens when either a or b is positive:

$$\text{Lower bound: } L > c \Leftrightarrow L/n > c/n \Rightarrow L/n > \lfloor c/n \rfloor \Leftrightarrow L' > c'$$
$$\text{Upper bound : } U \le c \Leftrightarrow U/n \le c/n \Rightarrow \lfloor U/n \rfloor \le \lfloor c/n \rfloor \Leftrightarrow U' \le c'$$

Now suppose we are in the constrained case, where $x, y \in [D_{MIN}, D_{MAX}) \subset \mathbb{N}$ and we want $x', y' \in [0, n_{bins}) \subset \mathbb{N}$. That is, x and y are in some finite range and we want to quantize them into a fixed number of non-negative values. The simplest way to do this is to let $n = \frac{D_{MAX} - D_{MIN}}{n_{bins}}$ and $x' = \left\lfloor \frac{x - D_{MIN}}{n} \right\rfloor$.
If D_{MIN} is divisible by n, which will often be the case in practical settings where standard integer types are used, then the updated arithmetic is simple:

$$x' = \left\lfloor \frac{x - D_{MIN}}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor - \frac{D_{MIN}}{n}$$
$$\Leftrightarrow x' + \frac{D_{MIN}}{n} = \left\lfloor \frac{x}{n} \right\rfloor,$$

such that we can just use $x'' = x' + \frac{D_{MIN}}{n}$ in all of our previous results. In the more general case our updated arithmetic will look like:

$$\left\lfloor \frac{x - D_{MIN}}{n} \right\rfloor \le \left\lfloor \frac{x}{n} \right\rfloor - \left\lfloor \frac{D_{MIN}}{n} \right\rfloor \le \left\lfloor \frac{x - D_{MIN}}{n} \right\rfloor + 1$$
$$\Leftrightarrow x' + \left\lfloor \frac{D_{MIN}}{n} \right\rfloor \le \left\lfloor \frac{x}{n} \right\rfloor \le x' + \left\lfloor \frac{D_{MIN}}{n} \right\rfloor + 1$$
$$\Rightarrow x' + \left\lfloor \frac{D_{MIN}}{n} \right\rfloor \le \left\lfloor \frac{x}{n} \right\rfloor \le \frac{x}{n} < \left\lfloor \frac{x}{n} \right\rfloor + 1 \le x' + \left\lfloor \frac{D_{MIN}}{n} \right\rfloor + 2$$

Thus we can use $x'' = x' + \left\lfloor \frac{D_{MIN}}{n} \right\rfloor$ in our previous results, with the caveat that we must now multiply all of our indicator variables by 2 when computing the upper and lower bounds.

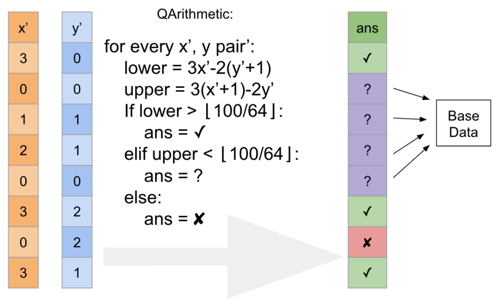


Figure 4: Quantized arithmetic approach scans the quantized data and calculates an upper and lower bound of the predicate for each x', y' pair.

3.2 Linear Combinations — Vectorized Algorithm

We present here a vectorized algorithm for performing the quantized arithmetic method with linear combination predicates. For simplicity we assume the quantization scheme is $x' = \left\lfloor \frac{x}{n} \right\rfloor$.

Algorithm 1: Vectorized Quantized Arithmetic Select

```
input  : $\vec{x_q}, \vec{y_q}, \vec{x}, \vec{y}, a, b, c$
output : $\vec{res}$
for $i \leftarrow 0$ to $n$ step $buff\_size$ do
    for $j \leftarrow 0$ to $buff\_size$ do
        $idx \leftarrow i + vec\_size * j$
        $\vec{u} = a*(\vec{x_q}[idx] + \vec{\mathbb{1}}[a > 0]) + b*(\vec{y_q}[idx] + \vec{\mathbb{1}}[b > 0])$
        $\vec{l} = \vec{u} + a*(1 - 2*\vec{\mathbb{1}}[a > 0]) + b*(1 - 2*\vec{\mathbb{1}}[b > 0])$
        $\vec{res}[idx] \leftarrow (\vec{l} > c)$
        $\vec{buff} \leftarrow$ ToPos($\vec{l} \le c$ && $\vec{u} \ge c$)
        $buff\_ct \leftarrow buff\_ct +$ Count($\vec{l} \le c$ && $\vec{u} \ge c$)
    for $k \leftarrow 0$ to $buff\_ct$ do
        $idx \leftarrow \vec{buff}[k]$
        $\vec{res}[idx] \leftarrow (a * \vec{x}[idx] + b * \vec{y}[idx] > c)$
```

We use a buffer of size $buff\_size$ to store the positions of uncertain elements and periodically flush the buffer by going to the base data. $buff\_size$ is an important parameter to tune. The *ToPos* function takes a bit vector and converts it to a position list and the *Count* function counts the number of 1s in a bit vector. When computing the lower bound vector we take advantage of the fact that L can be expressed in terms of U, a and b, without needing to go to the quantized data again: $L = U + a(1 - 2*\mathbb{1}[a > 0]) + b(1 - 2*\mathbb{1}[b > 0])$.

4 IMPLEMENTATION DETAILS

In order to get maximum performance from the system and reach memory bandwidth (or close to it), the select implementation must be multithreaded and vectorized. We implement the three methods (lookup table, range lookup table, and quantized arithmetic) using AVX2 instructions, relying on the Vectorclass library [1] by Agner Fog for a more portable and maintainable implementation. However, large lookup tables (frequent random accesses), the need to avoid overflow in arithmetic evaluation, and uncertain values cause complications in implementing a SIMD selection.

4.1 Memory Access

Evaluating the predicate on quantized data with a lookup table approach requires frequent memory accesses. In the standard lookup table implementation the index is a 16-bit value comprised of the two quantized values of x and y. Generating this index involves using a 256-bit vector which can hold 16, 16-bit indices. Then to perform the memory access, there are two methods: 1) perform a sequential memory access, or 2) use the *gather* intrinsic. At first blush, the gather intrinsic might appear to be the more efficient solution, but unfortunately it must fetch 32-bit data, meaning that 24 of the fetched bits will be wasted. Additionally, the gather intrinsic is not particularly well-optimized, especially on older architectures. We found that it was more efficient to use sequential access and build a SIMD vector from that for both lookup table methods.

We also found that the performance of the range lookup table increased significantly when compiling with `clang++` rather than `g++`. This is likely because clang was able to better optimize the sequential memory access pattern.

4.2 Overflow

During the quantized arithmetic method, to avoid overflow the computation of the lower and upper bounds must be computed in a higher bit precision. This means that the data loaded from memory must be extended to a larger bitwidth. In our implementation these values are computed with 16 bits of precision, which allows the results to be stored in a 256-bit AVX2 register of 16 shorts. However, this limits the possible values of a, b, and c to 8 bits. Supporting larger constants would extend to 32 bits, requiring larger/more registers which is costly on a machine without AVX512 support. However, overflow in the lookup table evaluation is not a problem because the bounds are computed during construction.

4.3 Checking Uncertain Data and Prefetching

Since checking uncertain data is rare, we found that using a position list to maintain a list of uncertain values and periodically check them was more efficient than populating a bit vector of uncertain values to check afterward. Every time a data point is determined to be uncertain the position is added to the “position buffer.” After a number of iterations, all values in the position buffer are checked by going to the base data. A small optimization we made was to emit a prefetch instruction when adding a position to the position buffer. This requests that the processor fetch the base data that will be needed ahead of time, and hopefully the data will have arrived by the time it is needed during the later pass. Since checking uncertain values is the bottleneck (as we show in the experimental evaluation), even small optimizations to this part of the algorithm can give speed-ups.

5 EXPERIMENTAL EVALUATION

To test our predicate evaluation approaches, we used a personal computer with an AMD Ryzen 5 1600 processor which has 16 GB RAM, 16 MB L3 cache, and six cores with two threads per core. We used a uniformly random distribution to generate ~268 million signed, 32-bit integers per column in $[-2147483648, 2147483648)$. We chose this number so that our data would be memory-resident

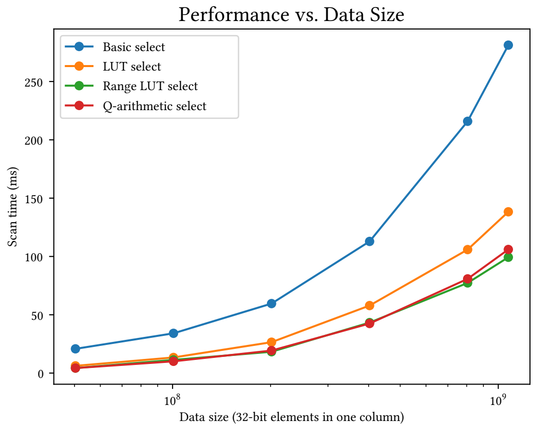


Figure 5: Evaluating $2x - y > 0$.

while far exceeding the size of L3 cache. We ran 10 trials per experiment and simdized all scans over the data with AVX2. Unless otherwise specified, we used 12 threads with simultaneous multithreading enabled, a position buffer of 8192 indices, and the evaluated the predicate $2x - y > 0$.

Figure 5 shows the speed of each approach as a function of data size. We observe that all methods which use the quantized data outperform the basic select, which we would expect since these methods incur one-fourth of the data movement cost, ignoring the uncertain values that must be dealt with. LUT, Range LUT, and Q-Arithmentic exhibit speed ups of 3.35, 4.84 and 4.84 respectively over the basic select on the smallest data size (~100 MB quantized data) and these speed ups fall to 2.03, 2.83 and 2.65 respectively at the largest data size (~540 MB quantized data). Our best explanation for this decrease in speed-up is that the scan is too fast to have fine-grained results for the smaller data sizes. In particular, at small data sizes there may be additional cache effects that come into play. Regardless, every quantized evaluation method shows significant speed up over the basic select, although none reach the ~4x goal predicted by the data movement. Of course, resolving uncertain values accounts for part of this slow down.

We examine the slow down as a result of resolving uncertain values in Figure 6. First, we observe that by calling GCC’s `__builtin_prefetch` intrinsic on any positions which we plan to resolve by going to the base data ahead of time, we are able to speed up each approach relative to the basic select baseline. We see between a $0 - .2$X speed up relative to the baseline for each method. Then, if we ignore resolving uncertain values, we see a further speed up of between $.2 - .7$x depending on the approach. For Q-arithmetic, we achieve the 4x speed up we expect.

We can further understand these results by examining Figure 7 L1 and last-level cache (LLC) misses for each approach. Observe that the LUT approach has far more L1 cache misses than the Range LUT approach and it even has more misses than the basic select. We believe this is because non-insignificant space in the L1 cache is occupied by the LUT throughout this approach. Furthermore, we see that all approaches have fewer LLC misses than the basic select,

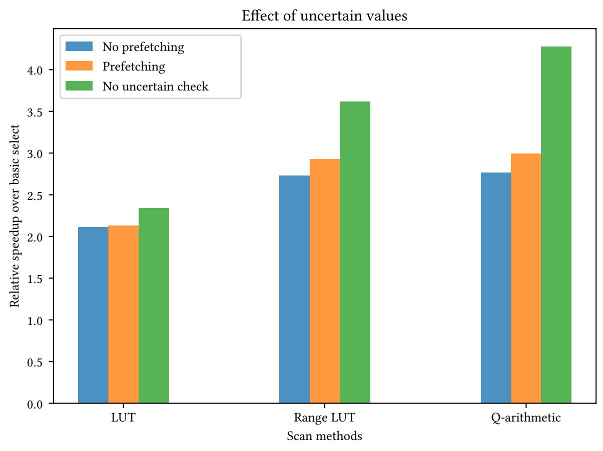


Figure 6: Speed ups relative to the basic select from (left to right) LUT, Range LUT, and QArithmetic methods when we include prefetching or ignore resolving uncertain values.

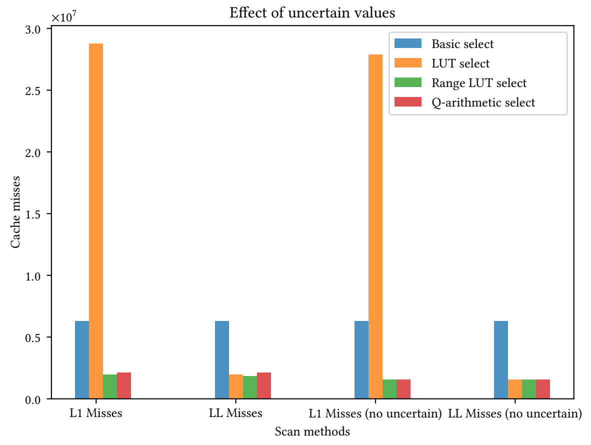


Figure 7: Simulated (from left to right) L1 cache misses, LL cache misses, L1 cache misses ignoring uncertain values, LL cache misses ignoring uncertain values. Data recorded by the *Cachegrind* tool [3].

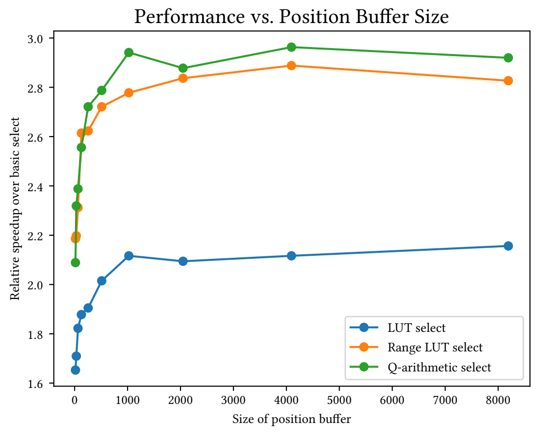


Figure 8: Relative speed up over basic select of each quantized approach as function of position buffer size (# of indices buffer can hold).

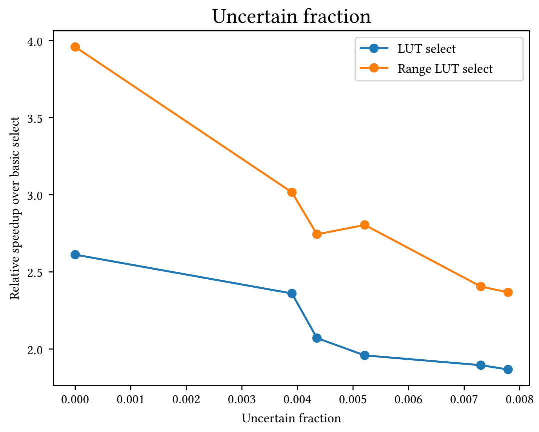


Figure 9: Relative speed up over basic select vs. fraction of uncertain positions which required going to the base data to resolve.

which confirms our understanding of the speed up we observe in Figure 5. Lastly, although the log scale makes it difficult to see, note that the LLC misses when we ignore uncertain values is exactly 4x less than the basic select for all three quantized approaches. This confirms our intuition.

Next, in Figure 8 we examine the relative speed up as a function of the size of the position buffer. We see that increasing the position buffer size, and hence decreasing the trips to the base data to resolve uncertain values, leads to faster predicate evaluation up until a certain size. Note that each thread has its own position buffer and each position is four bytes. Hence, we believe that the diminishing returns of increasing the position buffer size may be a result of limited L1 cache space. Or, we hypothesize that buffering to avoid many trips to the base data yields a speed up due to faster sequential reads of the base data, but the effect of this loses significance once the reads have become sequential enough. We believe that even faster predicate evaluation may be achieved if position buffer size were increased further and resolving of uncertain values was done via SIMD gather and scatter instructions, but it is unclear if this would improve overall performance by any meaningful amount.

Next, in Figure 9, we examine the relative speed up of both LUT approaches against the fraction of uncertain values, i.e. those that necessitated going to the base data. We achieved various selectivities by varying a, b, and c in $ax + by < c$. We only include the LUT approaches because our current implementation of Q-arithmetic requires that a and b be 8-bit integers whereas the LUT approaches do not have this constraint. As expected, the relative speed up is higher when there are fewer uncertain values. This motivates

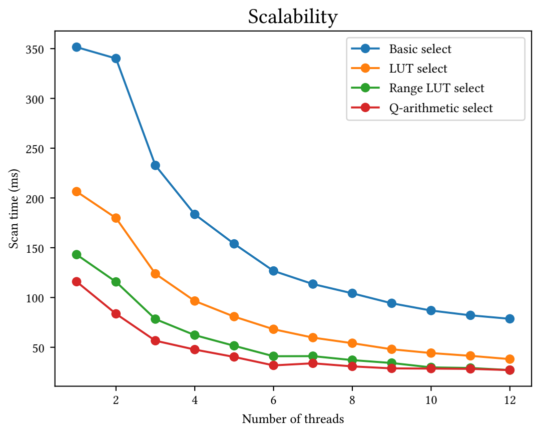


Figure 10: Speed of predicate evaluation as a function of number of threads, with simultaneous multithreading enabled.

the need for a smart quantization scheme which distributes an even number of data points in each bucket or even a quantization scheme which is optimized to have minimal uncertain values for the most common predicates, but such methods may complicate the Q-arithmetic or LUT approaches. However, due to the way the LUT is built, we believe that a LUT approach can be compatible with a more complicated quantization scheme such as a histogram, and we leave this for future work.

Lastly, we examine the scalability of each solution in Figure 10. We see similar increases in performance for all methods, although the basic select appears to improve by the greatest proportion. This is likely because of the minimal overhead of the basic select, whereas the other methods incur more overhead from additional position buffers as more threads are added. Note that SMT is enabled because we found that it improves performance slightly.

6 RELATED WORK & NEXT STEPS

This work builds off of **Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation** [2]. We used the single-column predicate evaluation code from this paper as the starting point to write our multithreaded, simdized multi-column predicate evaluation methods.

Some next steps have been touched on throughout this paper. We believe the most promising ones are to experiment with data sets other than uniformly random ones as well as real world data sets. Our simple quantization scheme will perform poorly for some predicates on these data sets because certain lossily compressed codes will represent much more data points than others. Hence, other quantization schemes such as histograms have potential to perform well here. We believe such schemes can be combined successfully with our lookup table approaches. We also believe that developing our lookup table approaches to handle an arbitrary number of columns would be a useful next project.

Another next step could be to explore the ideas pursued in this project with those of **Data Canopy: Accelerating Exploratory Statistical Analysis** [4]. Specifically, we believe predicates involving means and correlations could be answered quickly and accurately using lossily compressed data, with only a need to go to base data when higher precision is necessary.

It's also possible that the lookup table idea can extend well to floating point data where it is less clear how one might quantize down to 8 bits while using an arithmetic method. The lookup table implementation remains largely the same as the version for integer data, and with more complex floating point predicates, lookup tables will provide even larger benefits.

7 CONCLUSION

Light compression for multi-columnar predicate evaluation is a viable method for improving scan performance, but comes with extra complications that single column range queries do not have. We examined three methods for doing quantized evaluation quickly, with both lookup tables and direct arithmetic. Lookup table methods must keep the table small enough to avoid degrading performance. The arithmetic method is very efficient because it avoids frequent cache access, but incurs slightly more uncertain values, must cope with overflow, and is not as extensible to arbitrary predicates. Ultimately we found that uncertain values are the bottleneck and prevent these methods from reaching 4X speedups with 4X compression. With multi-column data each uncertain value is a random access that loads k cache blocks where k is the number of columns referenced by the predicate.

8 ACKNOWLEDGMENTS

Big thank you to Professor Stratos Idreos and the rest of the CS 265 teaching staff for help with this project and for two great semesters.

REFERENCES

[1] Agner Fog. Vectorclass library.
[2] Brian Hentschel, Michael S Kester, and Stratos Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 857–872, 2018.
[3] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.
[4] Abdul Wasay, Wei Xinding, Niv Dayan, and Stratos Idreos. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 International Conference on Management of Data*, pages 557–572, 2017.