Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation

Brian Hentschel, Michael S. Kester, Stratos Idreos
Harvard University
{bhentschel,kester,stratos}@seas.harvard.edu

ABSTRACT

While numerous indexing and storage schemes have been developed to address the core functionality of predicate evaluation in data systems, they all require specific workload properties (query selectivity, data distribution, data clustering) to provide good performance and fail in other cases. We present a new class of indexing scheme, termed a *Column Sketch*, which improves the performance of predicate evaluation independently of workload properties. Column Sketches work primarily through the use of lossy compression schemes which are designed so that the index ingests data quickly, evaluates any query performantly, and has small memory footprint. A Column Sketch works by applying this lossy compression on a value-by-value basis, mapping base data to a representation of smaller fixed width codes. Queries are evaluated affirmatively or negatively for the vast majority of values using the compressed data, and only if needed check the base data for the remaining values. Column Sketches work over column, row, and hybrid storage layouts.

We demonstrate that by using a Column Sketch, the select operator in modern analytic systems attains better CPU efficiency and less data movement than state-of-the-art storage and indexing schemes. Compared to standard scans, Column Sketches provide an improvement of 3×-6× for numerical attributes and 2.7× for categorical attributes. Compared to state-of-the-art scan accelerators such as Column Imprints and BitWeaving, Column Sketches perform 1.4 - 4.8× better.

ACM Reference Format:
Brian Hentschel, Michael S. Kester, Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3183713.3196911

1 INTRODUCTION

Modern Data Access Methods. Base data access and methods for predicate evaluation are of central importance to analytical database performance. Indeed, as every query requires either an index or full table scan, the performance of the select operator acts as a baseline for system performance. Because of this, there exists a myriad of methods and optimizations for enhancing predicate evaluation [14, 17, 23, 25, 28, 29, 33, 38]. Despite the large volume of work done, existing access methods each have situations in which they perform suboptimally. Figure 1 shows an example where different classes of access methods from traditional indexing such as B-trees to scan accelerators such as Zone Maps [25] and BitWeaving [23] do not bring any improvement over a plain scan. We use Figure 1 throughout this section as we discuss the performance characteristics of state-of-the-art indexing and scan accelerator methods, and use their performance to motivate Column Sketches.

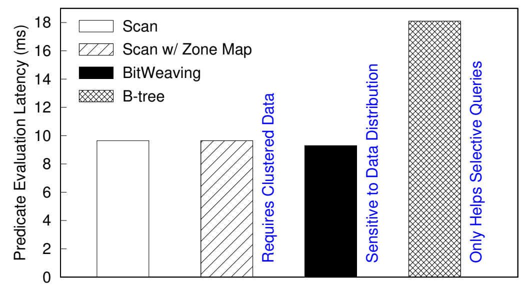


Figure 1: For certain workloads, no state of the art methods provide any significant performance benefit over a scan.

Traditional Indexes. A long term staple of database systems, traditional secondary indices such as B-trees localize data access to tuples of interest, and thus provide excellent performance for queries that contain a low selectivity predicate [8]. However, once selectivity reaches even moderate levels, the performance of B-trees is notably worse than other methods. Figure 1 shows such an example with low selectivity at 3%.

To achieve their performance on highly selective queries, B-trees introduce several inherent shortcomings. First, traditional indices look at data in the order of the domain, not in the order of the table. Thus their output leaves a choice: either sort the output of the index by the order of the table or continue through the rest of query execution looking at values out of order. Second, sorted order indices require gaps, as in the form of non-full leaf nodes for B-trees, for new insertions to amortize update costs. These gaps then require jumping around in memory during predicate evaluation, continually interrupting the processor so that it can wait for data. Both of these contrast with the modern scan, which relies on comparisons in tight for loops over contiguous data in memory, and which looks at data in the order of the table. Third, traditional indices have updates scattered throughout their domain and thus don't interact well with the append-only file systems [1, 2] that many analytic databases run over [9, 19, 36]. Additionally, recent changes such as the change in storage layout from row-oriented to column-oriented and increasing memory capacities have made traditional scans more performant relative to traditional

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196911

Workload → Technique ↓	Low Selectivity Scan	Arbitrary Scan: Skewed Data	Arbitrary Scan: Uniform Data	Arbitrary Scan: Clustered Data
B-Trees	Best	✗	✗	✗
Zone Maps	✓	✗	✗	Best
Early Pruning	✓	✗	Best	✓
Column Sketches	✓	Best	Best	✓
✗: poor performance, ✓: great performance, Best: best performance				

Table 1: Each technique has scenarios where it is the best for select performance. However, Column Sketches are the only technique which improves performance in every scenario.

indices. Currently, scans outperform B-trees for query selectivities as low as 1% [18].

Lightweight Indices. Recently, lightweight indexing techniques have made an impact on scan performance. These techniques largely work as a way to skip data while doing an in-order scan. Zone Maps are amongst the most widely used techniques today, and work by storing small amounts of metadata such as min and max for blocks of data [26]. This small amount of metadata exploits natural clustering properties in data and allows scans to skip over blocks that either entirely qualify or entirely do not qualify. Other techniques such as Column Imprints [33] or Feature Based Data Skipping [35] take more sophisticated approaches, but the high level idea is the same: they use summary statistics over groups of data to enable data-skipping. While incredibly useful in the right circumstances, the approach of using summary statistics over groups of data provides no help in the general case where data does not exhibit clustering properties [30]. Figure 1 shows such a case, where the column's values are independent of their positions; then, Zone Maps bring no advantage and the scan performance with and without Zone Maps is the same.

Early Pruning Methods. Early pruning methods such as Byte-Slicing [14], Bit-Slicing [23, 27], and Approximate and Refine [28] techniques work by bitwise-decomposing data elements. On a physical level, this means partitioning single values into multiple sub-values, either along each bit [23], each byte [14], or along arbitrary boundaries [28]. After physically partitioning the data, each technique takes a predicate over the value and decomposes the predicate into conjunctions of disjoint sub-predicates. As an example, checking whether a two byte numeric value equals 100 is equivalent to checking if the high order byte is equal to 0 and the lower order byte is equal to 100. After decomposing the predicate into disjoint parts, each technique evaluates the predicates in order of highest order bit(s) to lowest order bit(s), skipping predicate evaluation for predicates later in the evaluation order if groups of tuples in some block are all certain to have qualified or not qualified. Substantial amounts of data are skipped if the data in the high order bytes is informative, however these techniques can suffer under data skew. For instance, continuing with the above example, if a significant portion of high order bytes have value 0 then the predicate over the first byte is largely uninformative and the predicate over the second order byte is almost always evaluated. This is what happens in the case of Figure 1; the high order bits are biased towards zero, enabling very little pruning. As a result early pruning brings no significant advantage over a traditional scan.

Rapid and Robust Indexing via Column Sketches. We propose a new indexing scheme, Column Sketches, that unlike prior techniques, is robust to selectivity, data value distribution, and data clustering. A Column Sketch works by applying lossy compression on a value-by-value basis to create an auxiliary column of codes which is a "sketch" of the base data. Then, predicate evaluation is broken up into (1) a predicate evaluation over the sketched data, and, if necessary, (2) a predicate evaluation over a small amount of data from the base data.

Lossy Compression. By using lossy instead of lossless compression, Column Sketches simultaneously achieve three major goals. First, lossy compression allows for efficient algorithms to find encoding schemes that make code values always informative. Second, lossy compression guarantees that informative encodings can be achieved while being space efficient, with the resulting Column Sketch being significantly smaller than the base data. Third, lossy compression allows for faster ingestion speeds than indexing based off lossless encoding techniques. This is because the resulting codes are small, thus the structure necessary to store the mapping from values to codes is small, making the translation from value to code fast. As well, in contrast to lossless encoding such as dictionary compression, lossy compression means this mapping does not need to be injective and so new values in the domain do not cause a Column Sketch to rewrite old code values.

Technical Distinctions. Column Sketches achieve robust performance by differing from past techniques in key ways. In contrast to traditional indexing, data is seen in the order of the table, not in the order of the domain. The compressed auxiliary column lies in contiguous memory addresses and so no pointer chasing occurs. In contrast to lightweight indices, Column Sketches work on a value by value basis, allowing for data skipping even when groups of values mix values which satisfy and don't satisfy a predicate. In contrast to early pruning methods, lossy compression is tightly wedded with physical layout to enable guaranteed fast predicate evaluation and ingestion speeds. Other state-of-the-art acceleration techniques used in modern scans, such as SIMD, multi-core, and scan sharing, apply to Column Sketches as is. This is because at their core Column Sketches employ a sequential scan over a column of codes. The result is Table 1, with Column Sketches providing performance benefits for equality and range queries in all scenarios.

Contributions. The contributions of this paper are as follows:

(1) We introduce the Column Sketch, a data structure for accelerating scans which uses lossy compression to improve performance regardless of selectivity, data-value distribution, and data clustering (§2).
(2) We show how lossy compression can be used to create informative bit representations while keeping memory overhead low (§3).
(3) We provide algorithms for efficient scans over a Column Sketch (§4), give models for Column Sketch performance (§5), and show how Column Sketches can be easily integrated into modern system architectures (§6).
(4) We demonstrate both analytically and experimentally that Column Sketches improve scan performance by 3×-6× for numerical data, 2.7× for categorical data, and improve upon current state-of-the-art techniques for scan accelerators (§7).

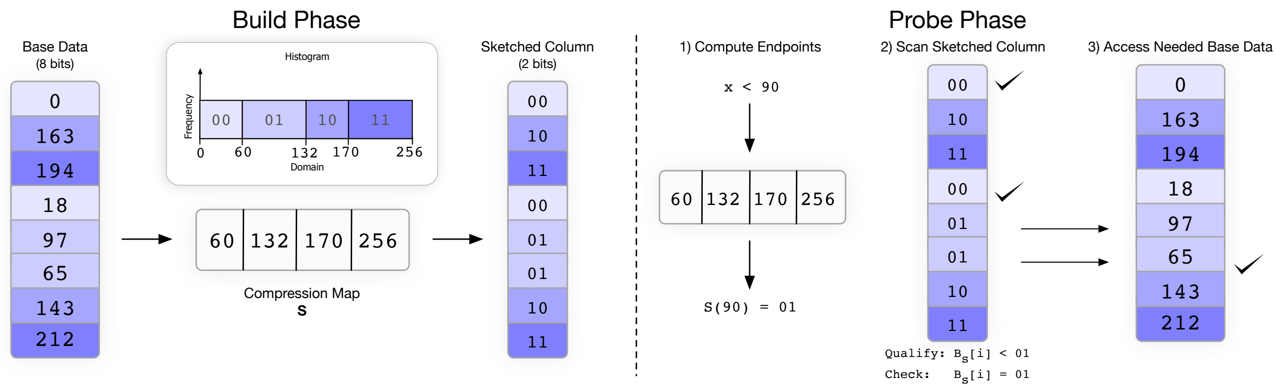


Figure 2: Column Sketches use their compression map to transform (possibly compressed) values in the base data to smaller code values in the sketched column. These codes are then used to filter most values in the base data during predicate evaluation.

2 COLUMN SKETCHES OVERVIEW

We begin with an illustrative example to describe the main idea and the storage scheme. For ease of presentation we use a simple lossy compression function and scan algorithm in the example. The rest of the paper then builds on the logical concepts covered here and shows how Column Sketches use these concepts to deliver robust, efficient performance.

Supported Base Data Format. The only requirement for the base data is that given a position i and base attribute B, that it be able to produce a value $B[i]$ for that position. Column Sketches work over row, column-group, or columnar data layouts, with the main body of this paper focusing on Column Sketches over columnar data layouts. Appendix C discusses alternative layouts. As is common in state-of-the-art analytic systems, all base columns of a table are assumed to be positionally aligned and thus positions are used to identify values of the same tuple across columns [3]. For numerical data types and dictionary encoded string data, the base data is an array of fixed-width values, with the value of position i at index i in the array. For unencoded variable length data such as strings there is one level of indirection, with an array of offsets pointing into a blob data structure containing the values.

Column Sketch Format. A Column Sketch consists of two structures. The first structure in a Column Sketch is the compression map, a function denoted by $S(x)$. The second structure is the sketched column. The compression map uses the function S to map the values in the base data to their assigned codes in the sketched column. The term Column Sketch refers to the joint pairing of both the compression map and the sketched column, and an example is shown in Figure 2.

(1) Compression Map. The compression map S is stored in one of two formats. If S is order-preserving, then we call the resulting Column Sketch order-preserving and the compression map is stored as an array of sorted values. The value in the array at position i gives the last element included in code i. For example, if position $i-1$ holds the value 1000 and position i holds the value 2400, then code i represents values between 1001 and 2400. In addition to the value at index i, there is a single bit used to denote whether the code is "unique". Unique codes are discussed in Section 3.

For non-order preserving Column Sketches, the function S is composed of a hash table containing unique codes and a hash function. In this format, frequent values are given unique codes and stored in the hash table. Infrequent values do not have their codes stored and are instead computed as the output of a (separate) hash function.

(2) Sketched Column. The sketched column B_s is a fixed width and dense array, with position i storing the output of the function S applied to the value at position i of the base data. To differentiate between values in the base data and the sketched column, we will refer to the values in the base data as simply values and the values in the sketched column as codes or code values.

Example: Building & Querying a Column Sketch. Consider the example shown in Figure 2, where we use the function S, defined by the array in the middle, to map from the 8 bit unsigned integers I_8 to the 2 bit unsigned integers I_2. S is order preserving and so it has the following properties:

(1) for $x, y \in I_8$, $S(x) \neq S(y) \Rightarrow x \neq y$
(2) for $x, y \in I_8$, $S(x) < S(y) \Rightarrow x < y$

Furthermore, S produces an output that is fixed-width (two bits) and assigns an equal number of values in the base data to each code.

We use S to build a smaller sketched column from the base data. For each position i in the base attribute B, we set position i in the sketched column to $S(B[i])$. The sketched column is $\frac{1}{4}$ the size of the original column, and thus scanning it takes less data movement. As an example, consider the evaluation of a query with the predicate WHERE $B < x$. Because S is order preserving, a Column Sketch can translate this predicate into $(B_s < S(x))$ OR $(B_s = S(x)$ AND $B < x)$.

To evaluate a predicate, the Column Sketch first computes $S(x)$. Then, it scans the sketched column B_s and checks both $B_s < S(x)$ and $B_s = S(x)$. For values less than $S(x)$, their base value qualifies. For values greater than $S(x)$, their values in the base data do not qualify. For values equal to $S(x)$, their base value may or may not qualify and so we evaluate $B < x$ using the base data. Algorithm 1 depicts this process.

Figure 2 shows an example. Positions 1 and 4 in the sketched column qualify without seeing the base data. Positions 5 and 6 need to be checked in the base data, and of these two, only position 6 qualifies. In the example, the Column Sketch needs to go to the base data twice while checking 8 values. This is explained by the

Algorithm 1 Select where $B < x$

Require: S is a function which is order preserving, B is the base attribute, B_s is the sketched column

```
for i = 0 to B.size do
    if B_s[i] < S(x) then
        write position i to result output
    else if B_s[i] == S(x) then
        if B[i] < x then
            write position i to result output
```

small number of bits in the compressed codes. In general, each code in a Column Sketch has a relatively equal number of values, and a Column Sketch needs to check the base data whenever it sees the mapped predicate value $S(x)$. As a result, we expect to need to go to the base data once for every $2^{\#\text{bits}}$ values.

Byte Alignment. Column Sketches work for any code size. However, we find that on modern hardware there exists around a 30% performance penalty for using non-byte aligned codes. Thus, we give special attention to 8 bit Column Sketches and to 16 bit Column Sketches.

3 CONSTRUCTING COMPRESSION MAPS

We now show how to construct compression maps for a Column Sketch. By definition, this map is a function from the domain of the base data to the domain of the sketched column. To go over compression maps, we discuss their objectives in Section 3.1, give guarantees of their utility in Section 3.2, and discuss how to build them for numerical and categorical attributes in Sections 3.3 & 3.4.

3.1 Compression Map Objectives

The goal of the compression map is to limit the number of times we need to access the base data, and to efficiently support data modifications. To achieve this, compression maps:

(1) Assign frequently seen values their own unique code. When checking the endpoint of a query such as $B < x$, a Column Sketch scan needs to check the base data for code $S(x)$. If x is a value that has its own code (i.e. $S^{-1}(S(x)) = \{x\}$), then we do not need to check the base data and can directly answer the query through only the Column Sketch. This property holds for both range predicates and equality predicates.

To achieve robust scan performance, we identify frequent values and give them their own unique code. As a simple example to see why this is critical for robust performance, if we have a value that accounts for 10% of tuples and it has a non-unique code, then predicates on this value's assigned code need to access the base data a significant number of times. Because accessing any item of data in a cache line brings the entire cache line to the processor, accessing 10% of tuples is likely to make performance similar to a traditional scan. Thus, we identify the frequent values so that we can limit the amount of base data we touch for any predicate.

(2) Assign non-unique codes similar numbers of values. The reasoning for this is similar to the reasoning for why frequent values need unique codes. We assign each non-unique code a relatively even and small portion of the data set so that we need only a small number of base data accesses for any scan.

(3) Preserve order when necessary. Certain attributes see range predicates whereas others do not. For attributes which see range predicates, the compression map should be order-preserving so that range queries can be evaluated using the Column Sketch.

(4) Handle unseen values in the domain without re-encoding. Re-encoding should be a rare and on-demand operation. By nature of being lossy, lossy compression means new values are allowed to be indistinguishable from already occuring values. Thus, provided we define our encoding smartly, new values do not require a Column Sketch to be re-encoded. For ordered Column Sketches to not need re-encoding, there cannot be consecutive unique codes. For instance, if S assigns the unique codes i to "gale" and $i + 1$ to "gate", then input strings such as "game" have no code value. Changing the code for "gate" to be non-unique solves this problem. For unordered Column Sketches, every unseen value has a possible value as long as there exists at least one non-unique code.

(5) Optional: Exploit Frequently Queried Values. Exploiting frequently queried values can give extra performance benefits; however, unlike frequent data values, identifying frequent query values makes query performance less robust. We focus on describing how to achieve efficient and robust performance for any query in the main part of the paper, and include details on exploiting frequently queried values in Appendix F.

3.2 Bounding Base Data Accesses

The following two theorems hold regarding how we can limit the number of values assigned to non-unique codes.

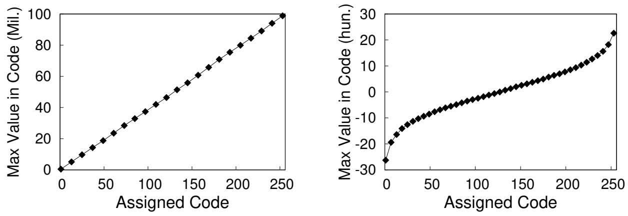
THEOREM 1. *Let X be any finite domain with elements $x_1, x_2, \ldots, x_m$ and order $x_1 < x_2 < \cdots < x_m$. Let each element x_i have associated frequency f_i with $\sum_{i=1}^{m} f_i = 1$. Let Y be a domain of size 256 and have elements $y_1, y_2, \ldots, y_n$. Then there exists an order-preserving function $S : X \to Y$ such that for each element y_i of Y, either $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{n}$ or $S^{-1}(y_i)$ is a single element of X.*

This theorem for an order preserving function then implies the result holds for a non-order preserving function as well.

COROLLARY 2. *Let X be any finite domain with elements $x_1, x_2, \ldots, x_m$ and let each element have associated frequency $f_1, f_2, \ldots, f_m$ such that $\sum_{i=1}^{m} f_i = 1$. Let Y be a domain of size n and have elements $y_1, y_2, \ldots, y_n$. Then there exists a function S such that for each element y_i of Y, either $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{n}$ or $S^{-1}(y_i)$ is a single element of X.*

The theorem and corollary prove that we can create mappings that limit the amount of values in the base data assigned to any non unique code. This directly implies that we can limit the amount of times we need to access the base data. The proof of Theorem 1 and Corollary 2 are given in Appendices A and B respectively, with both proofs giving an algorithm for the explicit construction of S. The proofs are given separately for each, as the algorithm given for creating the unordered compression map would give a map with less variance in the number of values in each non-unique code.

Finally, we note that Theorem 1 and Corollary 2 apply when the domain X is a compound space. For instance, X could be the domain of (country, city, biological sex, marital status, employment status) and the theorem would still apply. Appendix C includes further discussion and experiments with multi-column Column Sketches.



(a) Uniform Distribution **(b) Normal Distribution**

Figure 3: Bucket sizes track the distribution of the data.

3.3 Numerical Compression Maps

Lossless Numerical Compression. For numeric data types, lossless compression techniques such as frame-of-reference (FOR), prefix suppression, and null suppression work by storing the value as a relative value from some offset [11, 37, 42]. All three techniques support operations in compressed form; in particular, they can execute equality predicates, range predicates, and aggregation operators without decompressing. However, to support aggregation efficiently, each of these techniques conserves differences; that is, given base values a and b, their encoded values e_a and e_b satisfy $e_a - e_b = a - b$. This limits their ability to change the entropy of high order bits, and these bits can only be truncated if every value in a column has all 0's or all 1's on these high order bits.

Constructing Numerical Compression Maps. In contrast to lossless techniques, lossy compression is focused only on maximizing the utility of the bits in the sketch. The simplest way to do this while preserving order is to construct an equi-depth histogram which approximates the CDF of the input data, and then to create codes based on the endpoints of each histogram bucket. When given a value in our numerical domain, the output of the map is then simply the histogram bucket that a value belongs to. We create approximately equi-depth histograms by sampling values uniformly from the base column, sorting these values, and then generating the endpoints of each bucket based off of this sorted list.

Because histogram buckets are contiguous, storing the endpoint of each bucket i is enough to know the range that histogram bucket covers. Figures 3a and 3b show examples of mappings using histograms for two different data sets. In both graphs, we use 200,000 samples to create 256 endpoints. The uniform distribution goes from 0 to 10,000,000 and the normally distributed data is of mean 0 and variance 1000. The codes for the uniform distribution are evenly spaced throughout, and the codes for the normal distribution are farther apart towards the endpoints of the distribution and closer together towards the mean. The histograms capture the distributions of both functions and evenly space the values in the base data across the codes.

Handling Frequent Values. We define a frequent value to be a value that appears in more than $\frac{1}{z}$ of the base data values. To handle these frequent values, we first perform the same procedure as before and create a sorted list of sampled values. If a value represents more than $\frac{1}{z}$ of a sample of size n, then one of the values in the sorted list at $(\frac{n}{z}, \frac{2n}{z}, \ldots, \frac{(z-1)n}{z})$ must be that value. Thus, for each of these z values we can search for its first and last occurrence to check if it represents more than $\frac{1}{z}$ of the sample. If so, mark the middle position of that value in the list and give the value the unique code $c * \frac{midpoint}{n}$ (rounded to nearest integer), where c is the number of codes in the Column Sketch. In the case that $z > c$ and that two values would be given the same unique code c, the more frequent value is given that unique code. In this paper, we use $z = c$. Though a larger value of z can create faster average query times, we chose $z = c$ so that making a code unique does not increase the proportion of values in non-unique codes.

After finding the values that deserve a unique code and giving them associated code values, we equally partition the sorted lists between each unique code and assign the remaining code values accordingly. The identification of unique codes is in the worst case comparable to a single pass over the sample, and the partitioning of non-unique codes is then a constant time operation.

To make it so that updates cannot force a re-encoding, we do not allow unique codes to occupy subsequent positions. If in the prior procedure values v_i and v_{i+1} would be given unique code i and $i + 1$ respectively, only the more frequent value is given a unique code. For values to be assigned subsequent codes, the less frequent code can contain no more than $\frac{2}{c}$ of the sampled values, and so our previous robustness results for no non-unique code having too many values still holds. Additionally, we do not allow the first and last codes in the compression map to be unique.

Estimating the Base Data Distribution. For the compression map to have approximately equal numbers of values in each code, the sampled histogram created from the empirical CDF needs to closely follow the distribution of the base data. The Dvoretzky-Kiefer-Wolfowitz inequality provides bounds on the convergence of the empirical CDF F_n of n samples towards the true CDF F, stating: $P(\sup_{x \in \mathbb{R}} \|F_n(x) - F(x)\| \geq \epsilon) \leq 2e^{-n\epsilon^2}$ [10, 24]. In this equation we can treat F, the true distribution, as an unknown quantity and the column as an i.i.d. sample of F, or we can treat the column as a discrete distribution having CDF exactly equal to the CDF of the base data. In both cases, sampling from the base data n times gives the required result on the distance of our sampled data's empirical CDF F_n from the true CDF F[1]. We prove in Section 7 that for a one byte Column Sketch, any column with less than $\frac{4}{256}$ of the base data provides 2× performance benefit over a plain scan. Since a Column Sketch mapping never assigns a single non-unique code any proportion of values that it estimates as over $\frac{2}{256}$, we targeted $\epsilon = \frac{2}{256}$. With 200,000 samples as in Figure 3, the chance of an error of this amount is less than 10^{-5}. Both the number of samples n and the desired ϵ are tunable.

In Appendix G, we provide ways to deal with columns whose value distribution shifts over time.

3.4 Categorical Compression Maps

Categorical Data and Dictionary Encoding. Unlike numerical distributions, categorical distributions often have values which take up significant portions of the dataset. Furthermore, certain categorical distributions have no need for ordering.

Traditionally, categorical distributions have been encoded using (optionally order preserving) fixed width dictionary encoding. Dictionary encoding works by giving each unique value its own numerical code. A simple example is the states in the United States.

[1]Under the case that our column is a considered an i.i.d. sample from F, this should be without replacement. For the case that our columns CDF is our desired CDF, this should be with replacement.

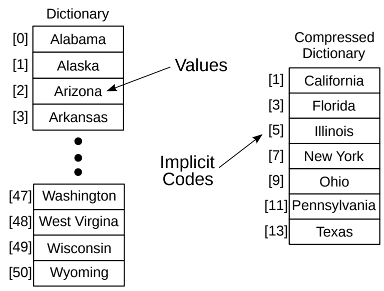


Figure 4: Lossy dictionaries trade uniqueness for better compression.

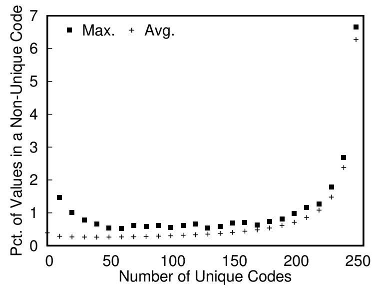


Figure 5: The number of unique codes should be neither too high nor too low.

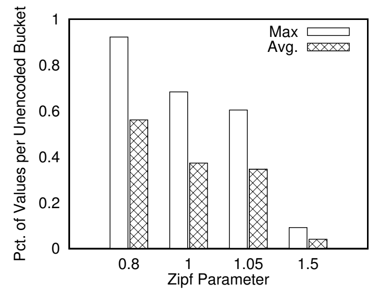


Figure 6: Regardless of the level of skew, the number of values in non-unique codes is low.

While this might be declared as a varchar column, there will be only 50 distinct values and so each state can be represented by a number between 0 and 49. Since each distinct value requires a distinct code, the number of bits needed to store a dictionary encoded value is $\lceil \log_2 n \rceil$, where n is the number of unique values.

Lossy Dictionaries. The compression maps for categorical distributions look similar to dictionary encoding, except that rare codes have been collapsed in on each other, making the number of code values smaller. The primary benefit of this collapsing is that a scan of the sketched column reads less memory. However, there is also a processing benefit as we can choose the number of code values in a non-injective encoding so that codes are of fixed byte length. For instance, if we look at a dataset with 1200 unique values, then a dictionary encoded column needs 11 bits per value. If these codes are packed densely one after the other, they will not begin at byte boundaries and the CPU will need to unpack the codes to align them on byte boundaries. If they are not packed densely, then the codes are padded to 16 bits, which in turn brings higher data movement costs. With Column Sketches, the lossy encoding scheme can choose the number of bits to be a multiple of 8, saving data movement without creating the need for code unpacking.

Shown in Figure 4 is an example comparing an order-preserving dictionary to an order-preserving lossy dictionary for states in the United States. Only the unique codes are shown, with the non-unique codes in the implied gaps. Though this is a simplified example, it shows various properties that we wish to hold for lossy dictionaries. The most frequent values, in this case the most populous states, are given unique codes, whereas rarer values share codes. California is given the unique code 1 whereas Wyoming shares code 14 with Wisconsin, West Virginia and several other states. The 7 unique codes cover nearly 50% of the population of the U.S. The other 50% of the population are divided amongst the 8 non-unique codes, with each non-unique codes having an expected 6.25% of the data. However, this is due to the low number of non-unique codes. For instance, if we were to change this to cities in the United States, of which there are around 19,000, and have 128 unique codes and 128 non-unique codes, then each non-unique code would have only 0.6% of the data in expectation.

Unordered Categorical Data. We first discuss assigning codes for categorical distributions that have no need for data ordering. Without the need for ordered code values, we are free to assign any value to any code value. This freedom of choice makes the space of possible compression maps very large, but also gives rise to fairly good intuitive solutions. We have three major design decisions:

(1) How many values should be given unique codes?
(2) Which values do we give unique codes?
(3) How do we distribute values amongst the non-unique codes?

(1) Assigning Unique Codes. The simplest approach to assigning unique codes is to give the most frequent values unique codes. This is robust in that it bounds the amount of times we access the base data for any predicate. More aggressive (but potentially less robust) approaches which analyze query history to assign unique code values are presented in Appendix F.

(2) Number of Unique Codes. The choice of how many unique codes to create is a tunable design decision depending on the requirements of the application at hand. We describe here two ways to make this decision. One way is to give every value that occurs with more than some frequency z in our sample a unique code value, leaving the remaining codes to be distributed amongst all values with frequency less than the specified cutoff. This parameter z has the same tradeoffs as in the ordered case, and tuning it to workload and application requirements is part of future work. In this paper, z is set to 256, for analogous reasons to the ordered case. The second way of assigning unique codes is to set a constant value for the number of codes that are unique. The second method works particularly well for certain values. For instance, if exactly half the assigned codes are unique codes, then we can use the first or last bit of code values to delineate unique and non-unique codes.

(3) Assigning Values to Non-Unique Codes. The fastest method for ingesting data is to use a hash function to relatively evenly distribute the values amongst the non-unique codes. If there are c codes and u unique codes, we assign the unique codes to codes $0, 1, \ldots, u-1$. When encoding an incoming value, we first check a hash table containing the frequent values to see if the incoming value is uniquely encoded. If the value is uniquely encoded, its code is written to the sketched column. If not, the value is then encoded as $u + [h(x)\%(c-u)]$.

Analysis of Design Choices. By far the most important characteristic for performance is making sure that the most frequent values are given unique codes. Figures 5 and 6 show the maximum number of data items given to any non-unique code as well as the average across all non-unique codes. In both figures, we have 100,000 tuples given 10,000 unique values, with the frequency with which we see each value following a Zipfian distribution. The rare values are distributed amongst the non-unique codes by hashing. In the first figure, we keep the skew parameter at 1 and vary the number of unique codes. In the second graph, we use 128 unique codes and change the skew of the dataset. As seen in Figure 5, choosing a moderate number of unique codes guarantees each non-unique code has a reasonable number of values in the base data. Figure 6 shows that for datasets with both high and low skew, the number of tuples in each non-unique code is a small proportion of the data.

Ordered Categorical Data. Ordered categorical data shares properties of both unordered categorical data and of numerical data. Like numerical data, we expect to see queries that ask questions about some range of elements in the domain. Like unordered categorical data, we expect to see queries that predicate on equality comparisons. Spreading values in the domain evenly across codes achieves the properties needed by both. Thus the algorithm given for identifying frequent values in numerical data works well for ordered categorical data as well.

4 PREDICATE EVALUATION OVER COLUMN SKETCHES

For any predicate that a Column Sketch evaluates, we have codes which can be considered the endpoint of the query. For instance, the comparison $B < x$ given as an example in Section 2 has the endpoint $S(x)$. For range predicates with both a less than and greater than clause, such as $x_1 < B < x_2$, the predicate has two endpoints: $S(x_1)$ and $S(x_2)$. And while technically an equality predicate has no endpoint since it isn't a range, for notational consistency we can think of $S(x)$ as an endpoint of the predicate $B = x$.

SIMD Instructions. SIMD instructions provide a way to achieve data-level parallelism by executing one instruction over multiple data elements at a time. The instructions look like traditional CPU instructions such as addition or multiplication, but have two additional parameters. The first is the size of the SIMD register in question and is either 64, 128, 256, or 512 bits. The second parameter is the size of the data elements being operated on, and is either 8,16,32, or 64. For example, the instruction _mm256_add_epi8 (__m256i a, __m256i b) takes two arrays, each with 32 elements of size 8 bits, and produces an array of thirty-two 8 bit elements by adding up the corresponding positions in the input in one go.

Scan API. A Column Sketch scan takes in the Column Sketch, the predicate operation, and the values of its endpoints. It can output a bitvector of matching positions or a list of matching positions, with the default output being a bitvector. In general, for very low selectivities a position list should be used and for higher selectivities a bitvector should be used. This is because at high selectivities the position list format requires large amounts of memory movement.

Scan Procedure. Algorithm 2 depicts the SIMD based Column Sketch scan for a one byte Column Sketch. It uses Intel's AVX instruction set and produces bitvector output. For space reasons, we omit the setup of several variables and use logical descriptions instead of physical instructions for lengthier operations.

The inner part of the nested loop is responsible for the logical computation of which positions match and which positions possibly match. In the first line, we load the 16 codes we need before performing the two logical comparisons we need. For the less than case, our only endpoint is $S(x)$, and we check for this value using the equality predicate on line 10. For each position matching this predicate, we will need to go to the base data.

Algorithm 2 Column Sketches Scan
Select where $B < x$, Column Sketch of one byte values

```
Require: S is a function which is order preserving
1:  repeat_sx = _mm128_set1_epi8(S(x))
2:  for each segment b in sketched column do
3:      /* Work over the code one block of data at a time */
4:      position= b× segment size
5:      for number of simd iterations in segment size do
6:          /* Perform logical comparisons necessary for predicate evaluation
            on each code. The 1,2 denote that this is done twice */
7:          codes1,2 = _mm_load_si128(codes_address)
8:          definite1,2 = _mm_cmplt_epi8(codes,repeat_sx)
9:          possible1,2 = _mm_cmpeq_epi8(codes,repeat_sx)
10:         bitvector_def1,2 = _mm128_movemask_epi8(definite1)
11:         bitvector_possible1,2 = _mm128_movemask_epi8(possible1)
12:         /* Store results we are certain of */
13:         bitvector_def = (bitvector_def1 « 16) | bitvector_def2
14:         store(bitvector_def) into result
15:         /* Check if the boundary values have any matching tuples and
            store qualifying positions. */
16:         conditional store bitvector_possible1,2 into temp_result.
17:         position + = 32
18:     /* Check all tuples that we are uncertain about */
19:     for position in temp_result do
20:         if B[position] == x then
21:             set bit position + beginning position in result
```

After these comparisons, we translate the definitely qualifying positions into a bitvector and store these immediately. For the possibly matching positions, we perform a conditional store. Left out of our code for reasons of brevity, the conditional store first checks if its result bitvector is all zeros. If it is not, it translates the conditional bitvector into a position list and stores the results in a small buffer on the stack. The result bitvector for possibly matching values will usually be all zeros as the Column Sketch is created so that no code holds too many values, and so the code to translate the bitvector into a position list and store the positions is executed infrequently. As a small detail, we found it important that the temporary results be stored on the stack. Storing these temporary results on the heap instead was found to have a 15% performance penalty.

The Column Sketch scan is divided into a nested loop over smaller segments so the algorithm can patch the result bitvector using the base data while the result bitvector remains in the high levels of CPU cache. If we check the possibly matching positions all at the end, we see a minor performance degradation of around 5%.

Unique Endpoints. Unique endpoints make the scans more computationally efficient. If the code $S(x)$ is unique, there is no need to keep track of positions and no need for conditional store instructions. Furthermore, the algorithm only needs a single less than comparison. After that comparison, it immediately writes out the bitvector. More generally, given a unique code, a scan over a Column Sketch completely answers the query without referring to the base data, and thus looks exactly like a normal scan but with less data movement.

Equality and Between Predicates. Equality predicates and between predicates are processed similarly to Algorithm 2. For equality predicates, the major difference is that, depending on whether the code is unique, the initial comparison only needs to store the partial result or the definite result. It can drop the other store instruction. For two-sided ranges with two-non unique endpoints, we have both > and < comparisons and an equality check on both endpoints. The list of possible matches is the logical or of the two equality checks, and the list of definite matches is the logical and of the two inequality comparisons. The rest of the algorithm is identical. If an endpoint is unique, then similar to the one sided case, the equality comparison for that endpoint can be removed.

Two Byte Column Sketches. Previous descriptions are based on single byte Column Sketches. In case we have a two byte representation, the logical steps of the algorithm remain the same. The only change is replacing the 8 bit SIMD banks with 16 bit SIMD banks.

5 PERFORMANCE MODELING

The performance model for Column Sketches assumes that performance depends on data movement costs. This assumption is justified in our experiments for byte-aligned Column Sketches, where we show that a Column Sketch scan saturates memory bandwidth.

Notation. Let B_b be the size of each value in the base data in bytes and let B_s be the size of the codes used in the sketch (both possibly non-integer such as 7/8 for a 7 bit Column Sketch). Let n be the total number of values in the column, and let M_g be the granularity of memory access. Because the modeling in this section is aimed at main memory, we use $M_g = 64$. The analysis of stable storage based systems is given in Appendix D.

Model: Bytes Touched per Value. Let us assume that the Column Sketch has no unique codes and consider a cache line of data in the base data. This cache line is needed by the processor if at least one of the corresponding codes in the sketched column matches the endpoint of the query. If we assume that there is only one endpoint of the query, then the probability that any value takes on the endpoint code is $\frac{1}{2^{8B_s}}$. Therefore, the probability that no value in the cache line takes on the endpoint code is approximately $1-(\frac{1}{2^{8B_s}})^{\lceil \frac{M_g}{B_b} \rceil}$, with the ceiling coming from values which have part of their data in the cache line. The chance we touch the cache line is the complement of that number, and so the total number of bytes touched per value is

$$B_s + B_b[1-(1-\frac{1}{2^{8B_s}})^{\lceil \frac{M_g}{B_b} \rceil}] \tag{1}$$

Plugging in 4 for B_b, 1 for B_s, and 64 for M_g, we get the value 1.24 bytes. If we use 8 for B_b, this remains at 1.24 bytes. If we change this to a query with two endpoints, the $\frac{1}{2^{*B_s}}$ term becomes $\frac{2}{2^{*B_s}}$ and so the equation becomes $B_s + B_b[1-(1-\frac{2}{2^{8B_s}})^{\lceil \frac{M_g}{B_b} \rceil}]$ From here, if we keep $B_s = 1$ and $M_g = 64$, then $B_b = 4$ gives an estimated cost of 1.47 bytes. Again, using $B_b = 8$ gives 1.47 bytes as well. Thus, for both one and two endpoint queries, and for both 4 byte and 8 byte base columns, a Column Sketch scan has significantly less data movement than a basic table scan.

We now take into account unique codes. Assume we follow the technique for deciding on unique codes from Section 3.3, where unique codes are given to values that take more than $\frac{1}{c}$ of the sample. Since the codes partition the dataset, the non-unique codes then contain less than $\frac{1}{c}$ of the dataset on average. Following similar logic to above, the result is that creating unique codes decreases the expected cost of a Column Sketch scan in terms of bytes touched for non-unique codes. For unique codes the number of bytes touched per value is 1. More detail is given on how unique codes affect the model in Appendix E.

Performance Tradeoffs in Code Size. Assume for the moment that non-byte aligned scans are memory-bound. Equation (1) gives a simple optimization problem which gives to the approximate optimal # of bits per code for a Column Sketch scan, with more bits per code increasing the memory bandwidth needed to perform the Column Sketch scan but decreasing the number of base data accesses. After optimizing this value, we have various tradeoffs. First and most apparent, more bits per code creates a larger memory footprint for the Column Sketch. Second, more bits per code means a larger dictionary, which slows down ingestion for ordered Column Sketches as more comparisons are needed. Finally, more bits per code makes the performance of the Column Sketch scan more robust. This is because the left side of equation (1) is constant, whereas the right side is variable ($\frac{1}{2^{8B_s}}$ was the expected proportion of codes for an endpoint). By increasing the number of bits per code, we reduce the influence of the right side of the cost equation and therefore reduce the variance of equation (1). Currently, we find that equation (1) only holds for byte-aligned code sizes with the scan performance being worse by around 30% for non-byte aligned codes, and so the tradeoffs mostly favor $B_s = 1$ or $B_s = 2$.

6 SYSTEM INTEGRATION AND MEMORY OVERHEAD

System Integration. Many components of Column Sketches already exist partially or completely in mature database systems. Creating the compression map requires sampling and histograms, which are supported in nearly every major system. The SIMD scan given in Section 4 is similar to optimized scans which already exist in analytic databases, and Zone Maps over the base data can filter out corresponding positionally aligned sections of a Column Sketch. Adding data and updating data in a Column Sketch are similar to data modifications for attributes which are dictionary encoded. In Section 7, we show that a Column Sketch scan is always faster than a traditional scan. Thus, optimizers can use the same selectivity based access path selection between traditional indices and a Column Sketch, with a lower crossover point. As well, Column Sketches work naturally over any ordered data type that supports comparisons. This contrasts with related techniques such as early pruning techniques, which need to modify the default encodings of various types such as floating point numbers to make them binary comparable. Finally, Column Sketches makes no change to the base data layout and so all other operators except for select can be left unchanged.

Memory Overhead. Let b_s be the number of bits per element in the Column Sketch. Then we need $b_s \times n$ bits of space for the sketched column. If we let b_b be the number of bits needed for a base data element, then each dictionary entry needs $b_b + 1$ bits of space, where the extra bit comes from marking whether the value for that code is unique. The size of the full dictionary is then $(b_b + 1) \times 2^b$ bits. Notably, b is usually quite small (we use $b = 8$ at all points in this paper to create byte alignment) and so the dictionary is also usually quite small. Additionally, the size of the dictionary is independent

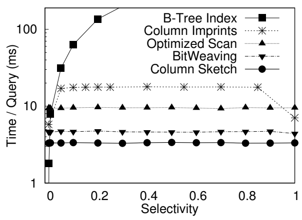


Figure 7: Column Sketches outperform other access methods in all but the most selective queries.

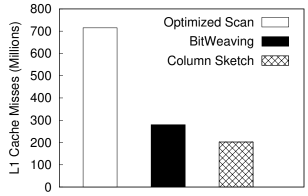


Figure 8: Column Sketches incur fewer cache misses than competing techniques.

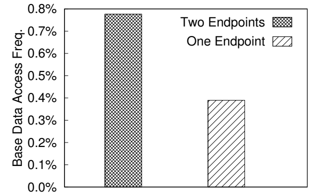


Figure 9: Column Sketches need to access the base data very infrequently.

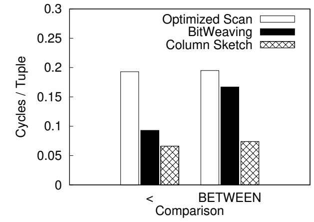


Figure 10: Column Sketches need fewer cycles to evaluate single and double sided predicates.

of n, the size of the column, and so the overhead of the Column Sketch approaches $b_s \times n$ bits as n grows. Additionally, we note that a Column Sketch works best with compression techniques on the base column that allow efficient positional access. This is normally the case for most analytical systems when data is in memory, as data is usually compressed using fixed width encodings [3].

7 EXPERIMENTAL ANALYSIS

We now demonstrate that, contrary to state of the art predicate evaluation methods, Column Sketches provide an efficient and robust access method regardless of data distribution, data clustering, or selectivity. We also show that Column Sketches efficiently ingest new data of all types, with order of magnitude speedups for categorical domains.

Competitors. We compare Column Sketches against an optimized sequential scan, BitWeaving/V (noted from here on out as just BitWeaving), Column Imprints and a B-tree index. The scan, termed FScan, is an optimized scan over numerical data which utilizes SIMD, multi-core, and zone-maps. For BitWeaving and Column Imprints, we use the original code of the authors [23, 33], with minor modifications to Column Imprints to adapt it to the AVX instruction set. We additionally compared against a SIMD version of BitWeaving [29], but found the SIMD version performed slightly less efficiently. The B-tree utilizes multi-core and has a fanout which is tuned specifically for the underlying hardware. In addition, for categorical data we compare Column Sketches against BitWeaving and "SIMD-Scan" from [38, 39], which is a SIMD scan that operates directly over bit-packed dictionary compressed data. Since we use SIMD at various points that are not referring to "SIMD-Scan", we refer to this technique as CScan. All experiments are in-memory; they include no disk I/O.

Scan API. The outputs of the scan procedure for a Column Sketch, BitWeaving, Column Imprints and FScan are identical. As an input the scan takes a single column and as an output it produces a single bitvector. The B-tree index scan takes as input a single column and outputs a list of matching positions sorted by position. This is because B-trees store their leaves as position lists and so this optimizes the B-tree performance.

Infrastructure. We run our experiments on a machine with 4 sockets, each equipped with an Intel Xeon E7-4820 v2 Ivy Bridge processor running at 2.0GHz with 16MB of L3 cache. Each processor has 8 cores and supports hyper-threading for a total of 64 hardware threads. The machine includes 1TB of main memory distributed evenly across the sockets and four 300GB 15K RPM disks configured in a RAID-5 array. We run 64-bit Debian "Wheezy" version 7.7 on Linux 3.18.11. To eliminate the effects of NUMA on performance, each of the experiments is run on a single socket. We give performance measurements in terms of cycles per tuple. For this machine and using a single socket, a completely memory bound process achieves a maximum possible performance of 0.047 cycles per byte touched by the processor.

Experimental Setup. We use a single byte Column Sketch for each experiment and unless otherwise noted, the table consists of 100 million tuples. When conducting predicate evaluation, each method is given use of all 8 cores. The numbers reported are the average performance across 100 experimental runs.

7.1 Uniform Numerical Data

Fast and Robust Data Scans. Our first experiment demonstrates that Column Sketches provide efficient performance regardless of selectivity. We test over numerical data of element size four bytes, distributed uniformly throughout the domain, and we vary selectivity from 0 to 1. The predicate is a single sided < comparison, with the endpoint of the query being a non-unique code of the Column Sketch. For this experiment only, we report performance as milliseconds per query, as the metric cycles/tuple is not very informative for the B-Tree.

Figure 7 shows the results. It depicts the response time for all five access methods as we vary selectivity. For very low selectivities below 1%, the B-tree outperforms all other techniques. However, once selectivity reaches even moderate levels, the B-tree performance significantly degrades. This is because the B-tree index needs to traverse the leaves at the bottom level of the tree, which stalls the processor and so wastes cycles. In contrast to the B-tree, the Column Sketch, Column Imprints, BitWeaving, and FScan view data in the order of the base data and consistently feed data to the processor.

During predicate evaluation, BitWeaving, Column Imprints, the Column Sketch, and the optimized scan all continually saturate memory bandwidth. However, the Column Sketch performs the best by reading the fewest number of bytes, outperforming the optimized scan by 2.92×, Column Imprints by 1.8 − 4.8× and BitWeaving by 1.4×. Figure 8 breaks down these results, showing the number of L1 Cache Misses for the Column Sketch against its two closest competitors, FScan and BitWeaving. The results align closely with the performance numbers, with the Column Sketch seeing 1.39× fewer L1 cache misses than BitWeaving and 3.54× fewer L1 cache misses than the optimized scan.

In performing predicate evaluation, the optimized scan and Column Imprints see nearly every value, leading to their high data

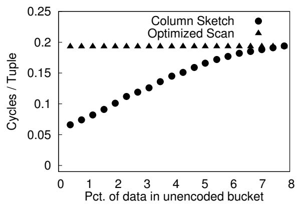


Figure 11: Column Sketches perform well even in the case of a bad compression map.

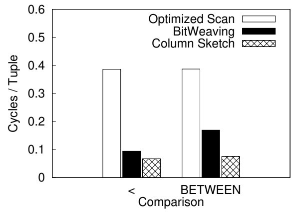


Figure 12: Column Sketches retain nearly identical performance with larger base data.

movement costs. This is because the Zone Map and Column Imprint work best over data which is clustered; when data isn't clustered, as is the case here, these techniques provide no performance benefit. BitWeaving and the Column Sketch also see every value, but decrease data movement by viewing fewer bytes per value. BitWeaving achieves this via early pruning, but this early pruning tends to start around the 12th bit. Additionally, even if BitWeaving eliminates all but one item from a segment by the 12th bit, it may need to fetch that group for comparison multiple times to compare the 13th, 14th, and so on bits until the final item has been successfully evaluated. In contrast to BitWeaving, the Column Sketch prunes most data by the time the first byte has been observed, with Figure 9 showing that a single endpoint query accesses around 0.4% of tuples in the base data. As well, in the rare case an item hasn't been resolved by the first byte, the Column Sketch goes directly to the base data to evaluate the item. This makes sense, once early pruning has reached a sparse stage where most tuples have or have not qualified, query execution should directly evaluate the small number of values left over.

Figure 10 shows the performance of the Column Sketch, BitWeaving, and the optimized scan in terms of cycles/tuple across single comparison and between predicates. In both cases, the Column Sketch performs significantly better than FScan and BitWeaving. For Fscan, its performance across both types of predicates is completely memory bandwidth bound and constant at 0.195 cycles/tuple. For BitWeaving, its performance on the between predicate is nearly half of it single comparison performance, going from 0.093 cycles/tuple to 0.167. However, this is a side effect of the distribution code we were given, which evaluates the < predicate completely before evaluating the > predicate after. If the two predicates were evaluated together, we would expect to see only a small decrease in performance. For the Column Sketch, it sees a minor drop in performance from 0.066 cycles/tuple to 0.074 cycles/tuple. This small drop in performance of the Column Sketch comes from having two non-unique endpoints, and not from increased computational costs. In the case one of the two endpoints is unique, the performance stays at 0.066 cycles/tuple. If both endpoints are unique, the performance is 0.053 cycles/tuple.

Model Verification. The performance of the Column Sketch nearly exactly matches what our model predicts. The model predicts that we would touch 1.24 bytes for a single non-unique endpoint, and 1.47 bytes for two endpoints. Taking into account the result bitvector and multiplying this by our saturated bandwidth performance of 0.047 cycles/byte, we get an expected performance from our model of 0.064 for a single endpoint and 0.075 for two endpoints.

Beta	BitWeaving (cycles/code)	Column Sketch (cycles/code)
1 (Uniform)	0.092	0.066
5	0.112	0.066
10	0.118	0.066
50	0.139	0.066
500	0.152	0.066
5000	0.188	0.066

Table 2: Scan Performance under Skewed Datasets

Robustness to a Bad Compression Map. Figure 11 shows the performance of the Column Sketch as more and more data is put into the single non-unique endpoint of our < comparison. The leftmost data point in the graph shows the expected performance of the Column Sketch, i.e. when the non-unique code has $\frac{1}{256}$ of the data, and each subsequent data point has an additional $\frac{1}{256}$ of the data assigned to that code. Notably, our analysis from Section 3.3 says that data points beyond the first 4 points will occur with probability less than $\frac{1}{10^5}$. The eventual crossover point when the performance of a Column Sketch degrades to worse than the performance of a basic scan is when the single endpoint holds $\frac{20}{256}$ of the data.

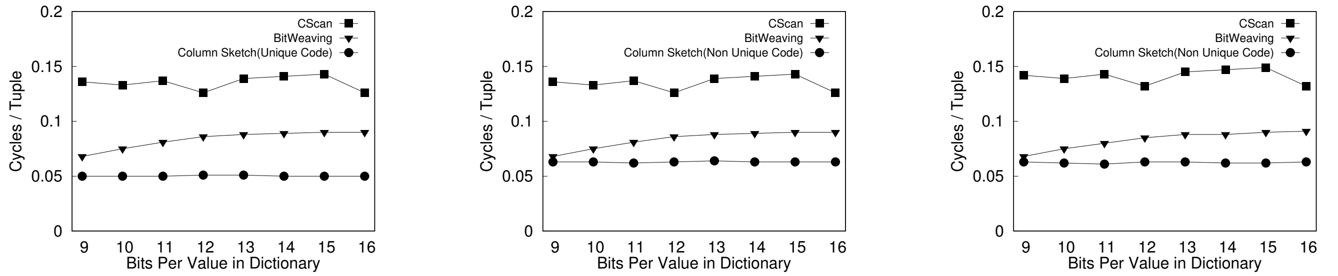
Larger Element Sizes. Our second experiment shows how performance changes for traditional scans, BitWeaving, and Column Sketch scans as the element size increases from four to eight bytes. The setup of the experiment is the same as before, i.e., we observe the response time for queries over uniformly distributed data. We do not use the index or Column Imprints from now on, as across all experiments the two closest competitors are FScan and BitWeaving. The results are shown in Figure 12.

For FScan, the larger element size means a proportional decrease in scan performance, with codes/cycle going from 0.193 to 0.386. However, for a Column Sketch, we can control the given code size independently of the element size in the base data. Thus, since we aim the Column Sketch at data in memory, we keep the code size at one byte. The scan is then evaluated nearly identically to the scan with base element size four bytes, and has nearly identical performance (0.067 instead of 0.066 cycles/code). Similarly, BitWeaving prunes almost all data by the 16th bit and so sees a negligible performance increase of 0.01 cycles/code. The overall performance increase from using the Column Sketch is 5.76× over the sequential scan and 1.4× over BitWeaving.

7.2 Skewed Numerical Data

Experiment Setup. For skewed data we use the Beta distribution scaled by the maximum value in the domain. The Beta distribution, parameterized by α and β is uniform with $\alpha = \beta = 1$ and becomes more skewed toward lower values as β increases with respect to α. We use this instead of the more commonly seen zipfian distribution as the zipfian distribution is more readily applied to categorical data with heavy skew, whereas the Beta distribution is continuous and better captures numerical skew. In the experiment, the element size is 4 bytes. We keep α at 1, and vary the β parameter. All queries are a range scan for values less than $2^8 - 1 = 255$. Table 2 depicts the results (FScan is not included as its performance is identical to Figure 7).

Robustness to Data Distribution. As the distribution gets more skewed towards low values, the high order bits tend to be mostly 0s. For BitWeaving, this means the high order bits don't give much data



(a) Equality, Unique Code **(b) Equality, Non-Unique Code** **(c) Less Than, Non-Unique Code**

Figure 13: Column Sketches is the fastest scan across unique and non-unique codes, and across equality and range comparisons

pruning, and as a result the performance of BitWeaving tends to look more like the full column scan. This is a weakness of using encoding which preserves differences. If there are even a trace amount of high values, then all code values need to keep the high order bits, which will be nearly all zero. For BitWeaving, we see performance degradation fairly quickly, as even $\beta = 5$ produces a notable performance drop. As β increases, the performance degrades more. In contrast, the performance of the Column Sketch is stable, with the ability to prune data unaffected by the data distribution change.

7.3 Categorical Data

Categorical Attributes. Our second set of experiments verifies that Column Sketches provide performance benefits over categorical attributes as well as numerical attributes. Unlike the numerical attributes seen in the previous experiments, categorical data consists of a significant number of frequent items. The Column Sketch was encoded with values taking up more than $\frac{1}{256}$ of the data being given unique codes. The resulting data set had 65 unique codes and 191 non-unique codes, with unique codes accounting for 50% of the data and non-unique codes accounting for 50% of the data.

For the dictionary compressed columns, we vary the number of unique elements so that the value size in the base data is between 9 and 16 bits. This matches the size of dictionary compressed columns which take up the majority of execution time in industrial workloads [39]. For columns whose base data takes less than 9 bits, Column Sketches provide no benefit and systems engineers should use other techniques.

In Figure 13a, we see that using a Column Sketch to perform equality predicate evaluation is faster than BitWeaving and CScan. Surprisingly, the improvement is more more than a 12% improvement over BitWeaving, even for 9 bit values, and sees considerably more improvement against CScan. For Column Sketches, the performance improvement increases up to bit 12 against BitWeaving, at which point the performance of BitWeaving becomes essentially constant. For the CScan, the improvement varies based off the base column element size. Due to word alignment, CScan does better on element sizes that roughly or completely align with words (such as 12 and 16 bits). However, for all element sizes, the computational overhead of unpacking the codes and aligning them with SIMD registers makes it so CScan is never completely bandwidth bound.

For non-unique codes, the performance of the Column Sketch is only slightly worse. The performance drops by 15% as compared to unique codes, with the Column Sketch remaining more performant than BitWeaving and CScan across all base element sizes. This is again due to the very limited frequency with which the Column Sketch looks at the base data; in this case, the Column Sketch is expected to view the base data for one out of every 256 values.

Technique	Numerical Data	Categorical Data (No New Values)	Categorical Data (New Values)
BitWeaving	50.823	26.139	80.155
Column Sketches	5.483	5.379	5.531

Table 3: Time to load all batches (sec.)

We additionally conduct range comparisons on categorical data, as shown in Figure 13c. The results are similar to Figure 13b. For Column Sketches we observe similar performance both when we stored the the base data as a text blob and when it was stored using non order-preserving dictionary encoded values. This is notable, since both blob text formats and non-order preserving dictionary encoded values are easier to maintain during periods in which new domain values appear frequently.

7.4 Load Performance

Experiment Setup. We test load performance for both numerical and categorical data, showing that Column Sketches achieve fast data ingestion regardless of data type. For both data ingestion experiments, we load 100 million elements in five successive runs. Thus at the end, the table has 500 million elements. In the numerical experiment, the elements are 32 bits and the Column Sketch contains single byte codes. In the categorical experiment, the elements are originally strings. For BitWeaving, we turn these strings into 15 bit order preserving dictionary encoded values, so that the BitWeaved column can efficiently conduct range predicates. For the Column Sketch, we encode the strings in the base data as non-order preserving dictionary encoded values, and use an order-preserving Column Sketch. As shown in the prior section, this is efficient at evaluating range queries. The time taken to perform the dictionary encoding for the base data is not counted for BitWeaving or Column Sketches. The time taken to perform encoding for the Column Sketch is counted.

The categorical ingestion experiment is then run under two different settings: in the first setting, each of the five successive runs sees some new element values, and so elements can have their encoded values change from run to run. Because there are new values, the order preserving dictionary needs to re-encode old values, and so previous values may need to be updated. In the second setting, there are no new values after the first batch.

Fast Ingestion via Column Sketches. In Table 3, we see that Column Sketches outperform BitWeaving in data ingestion by 5× - 16×. For Column Sketches, they have fast data loading performance as the only transformation needed on each item is a dictionary lookup. The data can then be written out as contiguous byte aligned codes. As well, regardless of the new values in each run, the Column Sketch always has a non-unique code for each value and thus never needs to re-encode its code values. Thus, a Column Sketch is particularly well suited to domains that see new values frequently, with the Column Sketch allowing for efficient range scans without requiring the upkeep of a sorted dictionary. In contrast to Column Sketches, BitWeaving tends to have a high number of CPU operations to mask out each bit and writes to scattered locations. More importantly, new element values can cause prior elements to need to be re-encoded. Notably, this isn't particular to BitWeaving, but an inherent flaw in any lossless order-preserving dictionary encoding scheme, with the only solution to include a large number of holes in the encoding [7].

8 RELATED WORK

Compression and Optimized Scans. The tight integration of compression and execution into scans in column-oriented databases started in the mid-2000's with MonetDB and C-Store [4, 42, 42]. Since then work has been done integrating numerous types of compression into scans, notably dictionary compression, delta encoding, frame-of-reference encoding, and run-length encoding [16, 43]. Nowadays, mixing compression and execution is standard and is seen in most commercial DBMSs [6, 12, 20, 32]. Recently, IBM created Frequency Compression [31, 32], exploiting data reordering for better scan performance.

Each of these techniques is lossless and designed to be used for base data. Thus, these techniques could achieve higher compression ratios through the use of lossy instead of lossless compression, reducing memory and disk bandwidth during scans. The potential of lossy compression for predicate evaluation has been hypothesized in the past but no solution has been presented [28].

Early Pruning Extensions. In [22], Li, Chasseur, and Patel look into lossless variable-length encoding schemes for Bit-Sliced Indices that are aimed at making the high-order bits informative in query processing. This solves the problem of data value skew and [22] additionally exploits more frequently queried predicate values. If skew is heavy enough such that frequent values or frequently queried values would require less than 8 bits, than the resulting bit-sliced index would be faster for querying those values than a Column Sketch. Furthermore, as in traditional bit-sliced indices, the resulting index is helpful even when the column's code values are smaller than 8 bits.

However, [22] does not consider lossy encoding schemes. By keeping the encoding schemes lossless, the padded variable length encoding schemes have larger memory footprint and have more expensive write times. As well, the resulting padded variable length encoding schemes are expensive to generate, with substantial run time spent on generating coding schemes for code sizes of 24 bits or less. An interesting line of future work is mixing the techniques from [22] with Column Sketches, and using padded bitweaved columns inside the sketched column of a Column Sketch.

Lightweight Indexing. Lightweight data skipping techniques such as Zone Maps and their equivalent provide a way to skip over large blocks of data by keeping simple metadata such as min and max for each column. They are included in numerous recent systems [15, 20, 31, 36, 40] and how to best organize both the data and metadata is an area of ongoing research [25, 30, 33–35]. Amongst recent approaches, Column Imprints stands out as similar in nature to Column Sketches[33]. Column Imprints also use histograms to better evaluate predicates, but do so for groups of values at a time instead of single values at at time.

For datasets with clustering properties, data skipping techniques notice that entire groups of values would evaluate the predicate to be either true or false, and so provide incredible speedup in these scenarios. For datasets that are not clustered, lightweight indices can't evaluate groups of data at a time and so scans need to check each value individually. In contrast, Column Sketches is able to handle these queries as it already works on an element by element basis. Thus, lightweight indices should be used in tandem with Column Sketches, as both have low update costs and target different scenarios.

Other Operators over Early Pruning Techniques. The use of early pruning techniques has been generalized to other operators beyond predicate evaluation [13, 28]. Most of this work can be applied to Column Sketches. For instance, a MAX operator can look at b high order bits and prune all records which are less than the maximum value seen using only those b bits, as they are certainly not the max. This is similar to Column Sketches, where any value that is not the maximum on the Column Sketch bytes is clearly not the maximum value in the column.

SIMD Scan Optimizations. There is a recent flurry of research on how to best integrate SIMD into column scans [14, 21, 29, 41]. The unpacking methods in this paper are based off work by Willhalm et al. where they use SIMD lanes to unpack codes one code per SIMD lane [38, 39]. Improvements in the computational performance of scans is complementary to Column Sketches. Column Sketches are designed to be a scannable dense array structure, and so improvements in evaluating predicates apply equally to Column Sketches.

9 CONCLUSION

In this paper, we show that neither traditional indexing nor lightweight data skipping techniques provide performance benefits for queries with moderate selectivity over unclustered data. To provide performance improvements for this large class of queries, we introduce a new indexing technique, Column Sketches, which provides better scan performance regardless of data ordering, data distribution and query selectivity. Compared to state-of-the-art approaches for scan accelerators, Column Sketches are significantly easier to update and are more performant on scans over a range of differing data distributions. Possible extensions of Column Sketches include usage for operators other than equality and range predicates, such as aggregations, set inclusion predicates, and approximate query processing.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This work is supported by the National Science Foundation under grant IIS-1452595.

REFERENCES

[1] Amazon simple storage service (s3). https://aws.amazon.com/s3/, 2017.
[2] Apache hadoop. http://hadoop.apache.org/, 2017.
[3] D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.
[4] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.
[5] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.
[6] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business Analytics in (a) Blink. *IEEE Data Engineering Bulletin*, 35(1):9–14, 2012.
[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 283–296, 2009.
[8] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
[9] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–226, 2016.
[10] A. Dvoretzky, J. Kiefer, and J. Wolfowitz. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *The Annals of Mathematical Statistics*, 1956.
[11] W. Fang, B. He, and Q. Luo. Database Compression on Graphics Processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.
[12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.
[13] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 291–302, 2015.
[14] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015.
[15] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.
[16] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *Proceedings of the ACM/IEEE-CS Symposium On Applied Computing (SAC)*, pages 22–27, 1991.
[17] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise Parallel Predicate Evaluation. *Proceedings of the VLDB Endowment*, 1(1):622–634, 2008.
[18] M. S. Kester, M. Athanassoulis, and S. Idreos. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 715–730, 2017.
[19] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.
[20] A. Lamb, M. Fuller, and R. Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.
[21] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Journal of Software: Practice and Experience*, 45(1):1–29, 2015.
[22] Y. Li, C. Chasseur, and J. M. Patel. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1509–1524, 2015.
[23] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.
[24] P. Massart. The tight constant in the dvoretzky-kiefer-wolfowitz inequality. *The Annals of Probability*, 1990.
[25] J. K. Metzger, B. M. Zane, and F. D. Hinshaw. Limiting scans of loosely ordered and/or grouped relations using nearly ordered maps, 2005.
[26] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998.
[27] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, 1997.
[28] H. Pirk, S. Manegold, and M. Kersten. Waste not… Efficient co-processing of relational data. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 508–519, 2014.
[29] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.
[30] W. Qin and S. Idreos. Adaptive Data Skipping in Main-Memory Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2255–2256, 2016.
[31] V. Raman, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, L. Zhang, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, and S. Liu. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.
[32] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 60–69, 2008.
[33] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–904, 2013.
[34] D. Slezak, J. Wroblewski, V. Eastwood, and P. Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proceedings of the VLDB Endowment*, 1(2):1337–1345, 2008.
[35] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1115–1126, 2014.
[36] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 996–1005, 2010.
[37] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *ACM SIGMOD Record*, 29(3):55–67, 2000.
[38] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing Database Column Scans with Complex Predicates. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2013.
[39] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.
[40] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2013.
[41] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002.
[42] M. Zukowski, P. A. Boncz, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.
[43] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 59, 2006.

A PROOF OF THEOREM 1

Theorem 1. *Let X be any finite domain with elements $x_1, x_2, \ldots, x_m$ and order $x_1 < x_2 < \cdots < x_m$. Let each element x_i have associated frequency f_i with $\sum_{i=1}^{m} f_i = 1$. Let Y be a domain of size n and have elements $y_1, y_2, \ldots, y_n$. Then there exists an order-preserving function $S : X \rightarrow Y$ such that for each element y_i of Y, either $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{n}$ or $S^{-1}(y_i)$ is a single element of X.*

Proof. We consider the greedy strategy for assigning codes to values of X. Assign $S(x_1), S(x_2), \ldots, S(x_{c-1}) = y_1$, where c is the first number such that $\sum_{i=1}^{c} f_i > \frac{2}{n}$. Then let $S(x_c) = y_2$. Now by definition, $\sum_{x \in S^{-1}(y_1)} f_x \leq \frac{2}{n}$ and $S^{-1}(y_2)$ is a single element of X. Furthermore, $\sum_{i=c+1}^{m} f_i \leq \frac{n-2}{n}$. Now, let us assume we continue assigning y_3 and y_4, y_5 and $y_6, \ldots, y_{n-1}$ and y_n in a similar fashion, assigning codes to the odd y_i until the next code would make $\sum_{x \in S^{-1}(y_i)} f_x \geq \frac{2}{n}$ or until we have assigned x_m a code. Then its clear that for all odd i, $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{n}$ and for all even i, $S^{-1}(y_i)$ is a single element. Furthermore, by y_n we are guaranteed to have seen all x_m since for each pair (y_i, y_{i+1}) up until we reach x_m, we have $\sum_{x_j \in S^{-1}(y_i) \cup S^{-1}(y_{i+1})} f_j > \frac{2}{n}$. □

B PROOF OF COROLLARY 2

Corollary 2 could be proven simply by creating any ordering on the domain X and then using Theorem 1. However, the separate proof of Corollary 2 is quite short and the proof gives an explicit construction of S that produces buckets with less variance in the number of values per non-unique code.

Corollary 2. *Let X be any finite domain with elements $x_1, x_2, \ldots, x_m$ and let each element have associated frequency $f_1, f_2, \ldots, f_m$ such that $\sum_{i=1}^{m} f_i = 1$. Let Y be a domain of size n and have elements $y_1, y_2, \ldots, y_n$. Then there exists a function S from X to Y such that for each element y_i of Y, either $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{n}$ or $S^{-1}(y_i)$ is a single element of X.*

Proof. Let $X_u = \{x_i | f_i \geq \frac{1}{n}\}$. Since $\sum_{i=1}^{n} f_i = 1$, it follows that there are less than n elements in X_u. Give each value in X_u a unique code and let $u = |X_u|$. Then there are $n - u$ codes left and $\sum_{\{i: x_i \notin X_u\}} f_i \leq 1 - \frac{u}{n}$.

Without loss of generality, let the values that have frequency less than $\frac{1}{n}$ be $x_1, x_2, \ldots, x_{m-u}$. We can map the elements $x_{m-u+1}, \ldots, x_m$ to codes $y_{n-u+1}, \ldots, y_n$, so it suffices to show that there exists a map S which maps $x_1, x_2, \ldots, x_{m-u}$ to $y_1, y_2, \ldots, y_{n-u}$ in a way that $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{n}$ for $1 \leq i \leq n-u$.

Let y_j be a code such that $\sum_{x \in S^{-1}(y_j)} f_x \geq \frac{2}{n}$. As each element in S has $f < \frac{1}{n}$, y_j must be composed of multiple elements. Let p be one of those elements. By the law of averages, there exists a bucket y_k with $\sum_{x \in S^{-1}(y_k)} f_x < \frac{1}{n}$. Then we can change S from mapping p to y_j to mapping p to y_k and $\sum_{x \in S^{-1}(y_k)} f_x < \frac{2}{n}$. Because X is a finite space, repeated application of this argument will eventually make $\sum_{x \in S^{-1}(y_j)} f_x < \frac{2}{n}$ for all j. □

C MULTI-COLUMN SKETCHES

Overview. Multi-Column Sketches take as input one or more columns for which the Column Sketch will be non-order preserving, and (optionally) a single column for which order is preserved. We expect Multi-Column Sketches to be of use in combining categorical variables that are queried frequently together, with perhaps a single numerical attribute. Like single column Column Sketches, we expect Multi-Column Sketches to be benficial for columnar data when the domain of the Column Sketch is larger than 256; however, the size of the domain is measured in terms of the possible joint values of the column, and so Multi-Column Sketches are useful in combining various categorical attributes which individually require less than 9 bits. Because a single encoding cannot support order on multiple attributes, we do not expect Multi-Column Sketches to be of use for multiple numerical attributes which are queried frequently together.

Building Multi-Column Sketches. Multi-Column Sketches are constructed via sampling much like a regular Column Sketch, with the sample being sorted if order is necessary. For the purposes of frequent values and unique codes, values are considered frequent only if the joint tuple of values from each column is above the threshold $\frac{1}{b}$. For example, a Multi-Column Sketch on (Job, City) would consider the joint value ("Software Engineer", "San Francisco") for frequent item status but not "Software Engineer" or "San Francisco" individually.

Modeling Multi-Column Sketches. The derivation of the model for Multi-Column Sketches is similar to the analysis for single column Column Sketches. Let $b_1, b_2, \ldots, b_m$ be the base columns with base value element sizes of $B_{b_1}, \ldots, B_{b_m}$. If the columns are stored in columnar fashion, with each column in a separate memory region, then the bytes touched per value is:

$$C_{col} = B_s + \sum_{i=1}^{m} B_{b_i} [1 - (1 - \frac{1}{2^{B_s}})^{\frac{M_g}{B_{b_i}}}]$$

For data which is stored in a column group, let $B_G = \sum_{i=1}^{m} B_{b_i}$ be the data size of a column group value. This value is strictly larger than the size of each individual attribute's data value, and so any given memory region of size M_g contains fewer tuples. In particular, for a region size M_g, we have $\lceil \frac{M_g}{B_G} \rceil$ tuples per region. It follows that the bytes touched per value is

$$C_{cg} = B_s + B_G [1 - (1 - \frac{1}{2^{B_s}})^{\frac{M_g}{B_G}}]$$

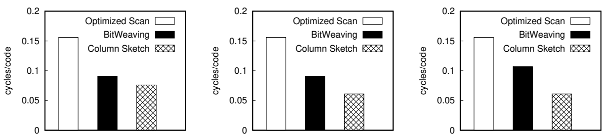
Since $\frac{M_g}{B_G} < \frac{M_g}{B_{b_i}}$ for all i, then $C_{cg} < C_{col}$ and so a Multi-Column sketch over a column group strictly dominates the performance of a Multi-Column group over columns which are stored separately.

Experimental Setup. In all experiments for Multi-Column Column Sketches the column group being sketched consists of three columns of 8 bit dictionary encoded values. The values in each column are generated independently of one another, and each column is generated using one of two distributions. In the first, the values in each column are generated using the uniform distribution. In the second, the values in each column are generated using the zipfian distribution with $z = 1.0$.

We compare the performance on equality predicates of the Multi-Column Sketch to BitWeaving and an optimized scan. For the optimized scan and the Multi-Column Sketch, the base data in the three columns is stored in either columnar format or group of columns format. For BitWeaving, the data is always stored bit-sliced.

Experimental Results. Figure 14a shows the performance of the Column Sketch, BitWeaving, and the optimized scan over uniformly distributed data and over columnar base data. The optimized scan is bandwidth bound and gets a performance of 0.156 cycles per code. The BitWeaving scan is similar to a BitWeaving scan over a single column with 24 bits, and so the performance is 0.091 cycles/code, matching Section 7. The Multi-Column Sketch scan sees some overhead compared to the equality predicates conducted in Section 7.3. This is expected; the small value sizes mean there are more values per cacheline, and so a greater chance of some code in any cacheline matching the query endpoint. Then, for each matching code, we need to go check each value in the base data. Still, the Multi-Column Sketch is faster than BitWeaving by 20% and faster than the optimized scan by 2×.

When the base data is changed to being in column group format, the number of values per cacheline of base data decreases and so more data is skipped. The result is that the performance of the Multi-Column Sketch rises by 25%, as seen in Figure 14b, bringing its speedup over BitWeaving to 50% and over the optimized scan to 2.8×. Finally, Figure 14c shows the same robustness to data distribution holds over Multi-Column Sketches, with the Multi-Column Sketch having the same performance over the zipfian dataset.



(a) Uniform Columnar **(b) Uniform Column Group** **(c) Zipfian Column Group**

Figure 14: Column Sketches retain an edge over the state-of-the-art across different data layouts.

D PERSISTENT STORAGE SYSTEMS

We can use the modeling of Section 5 to model how a Column Sketch would perform over systems aimed at persistent storage. Compared to in-memory systems, the main change is the increase in M_g from $M_g = 64$ to $M_g = 4K$.

We start with the model for performance for a single endpoint query from Section 5: $Cost = B_s + B_b[1 - (1 - \frac{1}{2^{8B_s}})^{\frac{M_g}{B_b}}]$ Previously, for $M_g = 64$ we had that with $B_s = 1, B_b = 4$, the amount of bytes touched was 1.24. With those parameters, the value of $1 - (1 - \frac{1}{2^{8B_s}})^{\frac{M_g}{B_b}}$ is 0.12. By moving to disk based page sizes, this value is 0.98, and thus we have almost no chance of skipping data pages. However, for two byte Column Sketches, i.e. $B_s = 2$, the probability of accessing any data page is only 0.02, and the expected number of bytes touched per value is 2.08. For element size $B_b = 8$, the chance of accessing any disk page is 0.992 and the number of bytes touched per value is 2.06. Thus, we expect two byte Column Sketches to be of use for persistent storage based systems.

The previous analysis assumes that the data is in columnar format and laid out entirely sequentially in persistent storage. If data is laid out in persistent storage in some other format, such as in row-store or PAX format [5], then the effective size of B_b may be much larger. In this case, the term $1 - (1 - \frac{1}{2^{8B_s}})^{\frac{M_g}{B_b}}$ becomes larger, and a Column Sketch skips more data. In particular, for large enough B_b, a single byte Column Sketch has a good chance of skipping data and is be useful for data in persistent storage. Assuming that the model above is perfect, we find that using 4K pages the effective base value size needed is 7.7 (with larger gains as the base value size rises further).

E PERFORMANCE MODEL: ACCOUNTING FOR UNIQUE CODES

In Section 5, we explained that uniquely encoding all values which appear with frequency greater than $\frac{1}{2^{B_b}}$ improved the expected select performance over both uniquely encoded and non-uniquely encoded endpoints. Here, we go into more detail about how uniquely encoding values affects Column Sketch performance.

The performance model for unique codes is trivial and says that we simply touch B_s bytes. For non-unique codes, the logic is similar to Section 5. Under the assumption that there are no unique codes, the chance any value takes on the given non-unique endpoint code is $1/2^{8B_s}$. To account for unique codes, we introduce the variables n and f_n, with n the number of non-unique codes, and f_n the portion of values which are encoded by non-unique codes. For n, f_n we have $0 \leq f_n \leq 1$ and $1 \leq n \leq 2^{8B_s}$.

If we assume that each non-unique code contains roughly the amount of values as its expectation, then the chance that any given value takes on a given non-unique endpoint is simply $\frac{f_n}{n}$. The chance we skip any given cacheline is then $1 - (1 - \frac{f_n}{n})^{\frac{M_g}{B_b}}$, and that the number of bytes touched per value is

$$B_s + B_b[1 - (1 - \frac{f_n}{n})^{\frac{M_g}{B_b}}]$$

F FREQUENTLY QUERIED VALUES

To model the cost of frequently queried values, we continue from the work presented in Appendix E. The eventual optimization problem is provably NP-hard, and so we provide solutions to this problem based on heuristics.

Let U be thet set of unique codes, and C be the total number of codes we want to assign. For each code $c \in C$, we have the cost of querying c as

$$\begin{cases} B_s & \text{if} \quad c \in U \\ B_s + B_b[1 - (1 - f_c)^{\frac{M_g}{B_b}}] & \text{if} \quad c \notin U \end{cases}$$

To optimize the cost of a given workload over a dataset, we start by incorporating into our model the variable q_c, with q_c representing the portion of queries that query on code c. Our total workload cost is then:

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

Assuming our goal is to minimize this value over all assignments of our codes, then we can simplify this equation to:

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

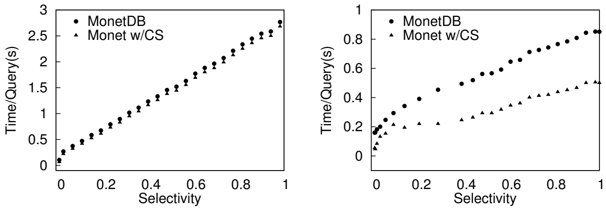
We now provide initial steps toward optimizing this assignment, with further optimization part of future work. Before delving into detail, we note the following: first, an analytical solution to this process depends on a product of terms including estimates of q_v and f_v, the proportion of all queries and data values that are on some value v. To properly estimate these terms to sufficient accuracy such that their product is accurate, significantly more samples will be needed than in Section 3. Second, if we let $\frac{M_g}{B_b} = 1$, then we have as a subproblem the optimization form of the partitioning problem, which is known to be NP-hard.

Unordered Column Sketches. In unordered Column Sketches, we have no dependencies between values and their given codes. Thus, we start by taking the codes which cost the most according to our cost model and making them unique. To do so, sort the values on $q_v(1 - (1 - f_v)^{\frac{M_g}{B_b}})$, which we will denote as expression $g(v)$. An optimal solution to the problem will assign unique codes to some m values which have the largest result on g. Let $q_N = 1 - \sum_{c \in U} q_c$ and $f_N = 1 - \sum_{c \in U} f_c$. Then, to estimate the number m which is optimal, we assign values to the set of unique codes U as long as the expression

$$q_N(1 - (1 - \frac{f_N}{C - |U|})^{\frac{M_g}{B_b}})$$

decreases. After, we continue along the list of values sorted on $g(v)$ and greedily assign the remaining values to the non-unique code which has the lowest value of

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



(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.