Rethinking the Encoding of Integers for Scans on Skewed Data

MARTIN PRAMMER, University of Wisconsin-Madison, USA
JIGNESH M. PATEL, Carnegie Mellon University, USA

Bit-parallel scanning techniques are characterized by their ability to accelerate compute through the process known as early pruning. Early pruning techniques iterate over the bits of each value, searching for opportunities to safely prune compute early, before processing each data value in its entirety. However, because of this iterative evaluation, the effectiveness of early pruning depends on the relative position of bits that can be used for pruning within each value. Due to this behavior, bit-parallel techniques have faced significant challenges when processing skewed data, especially when values contain many leading zeroes. This problem is further amplified by the inherent trade-off that bit-parallel techniques make between columnar scan and fetch performance: a storage layer that supports early pruning requires multiple memory accesses to fetch a single value. Thus, in the case of skewed data, bit-parallel techniques increase fetch latency without significantly improving scan performance when compared to baseline columnar implementations.

To remedy this shortcoming, we transform the values in bit-parallel columns using novel encodings. We propose the concept of forward encodings: a family of encodings that shift pruning-relevant bits closer to the most significant bit. Using this concept, we propose two particular encodings: the Data Forward Encoding and the Extended Data Forward Encoding. We demonstrate the impact of these encodings using multiple real-world datasets. Across these datasets, forward encodings improve the current state-of-the-art bit-parallel technique's scan and fetch performance in many cases by 1.4x and 1.3x, respectively.

CCS Concepts: • **Information systems** → **Data management systems**; **Data layout**; **Data scans**.

Additional Key Words and Phrases: storage organization, column store, columnar scan, bit-parallel, integer code, integer encoding

ACM Reference Format:
Martin Prammer and Jignesh M. Patel. 2023. Rethinking the Encoding of Integers for Scans on Skewed Data. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 257 (December 2023), 27 pages. https://doi.org/10.1145/3626751

1 INTRODUCTION

A critical property of an integer data type is its binary representation, which has largely remained unchanged since the start of the computing field. The natural method to represent an integer in a machine-amenable format is to use a base-2 positional representation and then interpret the number using a signed notation, usually two's complement [17]. The two's complement representation has been a crucial component of modern computing systems since it was proposed as part of the von Neumann architecture [38]. As we will see in this paper, this encoding plays a critical role in the performance of predicate-based columnar scan techniques – a common operation in data platforms that has received much attention in the community [1, 4, 9, 15, 16, 18, 22, 23, 29, 34, 40].

Authors' addresses: Martin Prammer, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI, 53706, USA, prammer@wisc.edu; Jignesh M. Patel, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, jignesh@cmu.edu.

2836-6573/2023/12-ART257
https://doi.org/10.1145/3626751

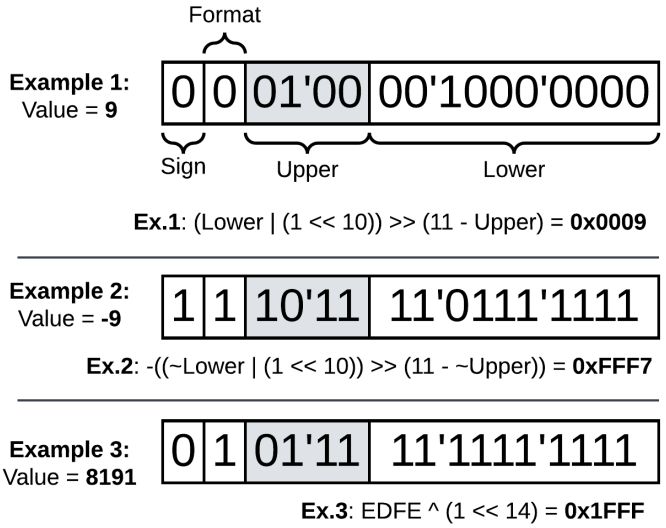

Fig. 1. The values 9 (Example 1), -9 (Example 2), and 8191 (Example 3) encoded in EDFE, along with the process to decode each value back to an INT (explained in detail later) using C program syntax. We omit masking operations applied to the upper and lower fields to make the examples more concise. The upper field of each encoded value is highlighted in light gray. In Example 2 (-9), because the sign bit is set, the upper and lower fields are inverted during the decoding process, and the final output is negated. In Example 3 (8191), because the sign bit is cleared and the format bit is set, the only step to decode is to invert the format bit.

First, we observe that numbers may have multiple machine representations. As a more complex example than the aforementioned integer representation, consider the IEEE 754 floating point number representation. When read from left to right, it defines a set of discrete fields at fixed bit positions (sign, exponent, significand) [14].

Second, building on this observation, we reimagine the encoding of integers to more resemble a floating-point representation, but without losing precision or the natural ordering of integers. Our reimagined integers are also compatible with existing binary comparison logic, allowing their use as a replacement for existing integer types without significant changes to existing data processing kernels. Due to these properties, our alternative integer encodings are a significant departure from existing floating-point formats, as they do not call for dedicated floating-point processors [14, 25, 39].

One such method we consider is illustrated in Figure 1. This new format is called the **E**xtended **D**ata **F**orward **E**ncoding (EDFE). EDFE re-encodes a two's complement integer to minimize the number of leading zeroes. Thus, EDFE is particularly amenable to bit-parallel columnar-scan methods that rely on a property called *early pruning* to efficiently evaluate a predicate scan on an integer-typed column. EDFE extends the simpler **D**ata **F**orward **E**ncoding (DFE) by adding a sign and a format bit.

Example 1 in Figure 1 illustrates the EDFE decoding process (Section 3 presents the technique in full). To decode a value in EDFE to a two's complement integer, the value in the lower field is shifted as a function of the upper field. The bit pattern in the lower field is always able to reconstruct the entirety of the original value. This simplicity of encoding allows EDFE to be efficiently processed using simple bit-manipulation operations instead of relying on dedicated hardware, facilitating a lightweight encoding and decoding process that modern CPUs can perform in a few clock cycles.

We emphasize the broad property of EDFE shifting bits towards the most significant bit (MSB). This property is common to all encodings in this family, which we identify as *forward encodings* (FEs). When using FEs instead of two's complement representations, the bits likely to distinguish between values are shifted toward the MSB. As shown in Figure 1, evaluating $-9 < 9 < 8191$ is represented in INT as `0xFFF7` < `0x0009` < `0x1FFF`, while in EDFE it becomes `0xEF7F` < `0x1080` < `0x5FFF`.

Forward encoding works particularly well with *bit-parallel columnar scan* methods [9, 16, 19, 22, 29, 34]. These methods often slice the bits in a sequence/column of integer values at the bit level and store the column by the bit position. For example, in both the classic bit-sliced index [29] and the BitWeaving/V [22] storage organizations, retrieving the first 64 bits of an integer column returns the 64 MSBs of the first 64 column values.

Predicate evaluation using bit-parallel organizations benefits from *early pruning*, which allows a scan to safely (early) terminate when sufficient bit slices have been evaluated to guarantee a correct result. The simplest case is that of an equality predicate. A bit-parallel method compares a slice of data bits with the corresponding slice in the predicate literal. If any data bits do not match, the corresponding column value (and hence the record) will not match the predicate, irrespective of additional data bits. However, integer columns often have many leading zeros, especially in the case of skewed datasets. These leading zeros delay early pruning, as more bit slices must be scanned before processing a prunable bit. Forward encodings are designed to address this weakness in two ways: First, the set bits from the original value are shifted toward the MSB. Second, the upper field captures the magnitude of a value while the remaining bits capture the details; as both fields can be used for early pruning, many scans prune early after evaluating the first few bits of the upper field.

The improved columnar scan performance of bit-parallel techniques is based on rearranging the bits of each value in a column. As the performance benefit of early pruning is based upon accessing a portion of the underlying value, this access pattern must be facilitated by the column's data layout. However, because of this modified layout, retrieving a value from the column requires additional memory reads. Broadly, bit-parallel techniques improve columnar scan performance at the cost of fetch performance.

Previous bit-parallel scan acceleration techniques have primarily been evaluated using synthetic benchmarks that usually generate uniform-random data [28, 36]. While uniform-random data is useful for comparing techniques, it obfuscates the exact origin of scan performance improvements that are influenced by the underlying data. Is a uniform distribution of bits necessary for early pruning to function effectively? Does reliance on a uniform bit distribution necessitate that bit-parallel techniques must always be used to store integers using the two's complement representation? To approach these questions, we perform our evaluation using multiple skewed, real-world datasets, which we describe during our evaluation.

The encoding method of a value is orthogonal to the storage organization of a column. One could take a column/batch of integer values and slice them at the bit level. Indeed, this is what methods like bit-sliced indices [29] and BitWeaving/V [22] do. But, one could also slice at a different "vertical" boundary – such as the byte level. This latter approach is adopted by ByteSlice [29], the current state-of-the-art bit-parallel method. We evaluate our proposed encodings using BitWeaving/V and ByteSlice to understand the impact of the choice of encoding on multiple storage organizations.

Collectively, the contributions of this paper are as follows:

(1) We propose reimagining the encoding of integer data types and present a new general method of encoding integers called forward encoding. Besides the specific use cases discussed in this paper, we intend for the paper to initiate a new way of thinking about the encoding of integer data.

(2) We propose two new forward encodings, DFE and EDFE, and explore their theoretical properties.
(3) We note the orthogonality between encoding and storage organization. In this framework, BitWeaving/V and ByteSlice use existing integer encodings with bit-sliced and byte-sliced boundaries. We provide an intuitive method to characterize different ways of organizing the bits in a column of integers. Using this characterization, we also explore each storage organization's trade-offs between columnar scan and fetch operation performance.
(4) We demonstrate the performance advantage of DFE and EDFE across multiple storage organizations using skewed, real-world data. Averaging across several cases, we observe geometric mean speed-ups of 1.47x and 1.33x for scan and fetch operations using the state-of-the-art bit-parallel technique (ByteSlice). The performance of BitWeaving/V is improved as well (1.55x and 1.19x), elevating it to be comparable with ByteSlice.

The remainder of this paper is organized as follows. First, we explore key background works in Section 2 to establish a set of standard parameters that describe bit-parallel techniques (Section 2.2). Next, we use these parameters to examine runtime discovered early stopping (a more precise definition of early pruning) in Section 2.3. We discuss related work in Section 2.4. Section 3 describes the properties of forward encodings (Section 3.1) before discussing the specifics of DFE (Sections 3.2) and EDFE (3.3). Then, we introduce a new concept called FE-enabled early stopping in Section 3.4. In Sections 3.6 and 3.7, we explore usage considerations and trade-offs between our proposed and existing encodings. We evaluate our proposed encodings in Section 4 and discuss the results of our experiments in Section 5. Finally, Section 6 contains our concluding remarks.

2 BACKGROUND AND RELATED WORK

This section describes key bit-parallel techniques and identifies a set of standard parameters to compare bit-parallel methods. Using these parameters, we explore the critical components that make early pruning effective. We also explore a number of related works in this space, including hardware implementations of early pruning and alternative bit representations (codes) of integers.

2.1 Existing Bit-Parallel Techniques

Bit-parallel techniques have achieved significant reach since their inception [9, 16, 19, 22, 29, 34]. Broadly, these methods recognize that reorganizing the bits of multiple values can lead to increased storage and processing efficiency.

We consider the bit-sliced index [29] the starting point of many bit-parallel techniques. This initial technique was extended in a variety of directions [9, 15, 22, 34]. In particular, we emphasize the later contribution of *early pruning* [22]. Early pruning allows bit-parallel techniques to identify opportunities to compute results without accessing every bit of each processed value. The properties behind early pruning are crucial to the behavior of modern bit-parallel, predicate-based, columnar scan techniques.

We select two background works to emphasize fundamental mechanisms common to bit-parallel techniques: BitWeaving/V [22] and ByteSlice [9]. We choose these two particular techniques for two reasons: First, ByteSlice is the current state-of-the-art bit-parallel technique. Second, the authors of the ByteSlice paper evaluate ByteSlice against BitWeaving/V to demonstrate the differences between literal bit-parallel (bit width $b = 1$) and byte-parallel ($b = 8$) storage layers. These two techniques serve as a baseline to compare multiple forms of bit-parallelism when performing columnar scan and fetch operations.

As our proposed encodings are heavily influenced by early pruning, we explore the behavior of early pruning in a dedicated portion of this section. However, before we can thoroughly investigate

Variable Name	V.	Value	Definition
Bit Width	b	Parameter	The bit-width of each data value in the original column. This parameter is data-dependent.
Group Size	g	Parameter	The number of segments of data values that are processed in parallel. This parameter is specified by the implementation of the bit-parallel technique.
Strata Width	s	Parameter	The smallest number of actionable bits per data value. This parameter is defined by the bit-parallel technique.
Strata Count	c	$c = b/s$	The number of strata that each data value is broken into.
Parallelism Size	w	$w = g \times s$	The number of data bits that are processed in parallel. This value is influenced by the available compute resources.

Table 1. Variables used to categorize bit-parallel techniques.

early pruning in prior work, we need to establish a standard set of parameters to describe existing bit-parallel techniques. From this foundation, we can explore the impact of early pruning.

2.2 Generalization of Techniques

Prior bit-parallel works use different terms to identify their bit-parallel techniques. We unify the existing terminology using a precise set of terms, which we tabulate in Table 1.

We define a bit-parallel technique as a technique that processes s bits of g values in a combined processing step. The parameter s identifies the "strata width," which indicates the number of bits processed per value in a parallel context. A "byte-parallel" technique specifically identifies a bit-parallel method with a strata width parameter $s = 8$. The parameter g (group size) identifies how many individual segments of data values are processed in parallel. We define g as an implementation detail of each method; in many cases, the theory behind the overall technique remains the same when g is modified.

When a bit-parallel method loads a column of data, it processes values with a bit width of b. All other terms that describe bit-parallel techniques can be derived from b, g, and s.

The number of strata each value is broken into is defined as the "strata count" $c = b/s$. A higher value of c allows for more early stopping (early pruning) opportunities during processing. However, higher values of c generally incur a higher overhead cost when recombining strata to restore the original value.

The number of bits processed in parallel at each step is the parallelism size $w = g \times s$. The parallelism size of bit-parallel techniques is generally implementation dependent, influenced by not only the same considerations that influence the group size g but also the available compute and storage resources.

Using this framework, we can systematically characterize various terms used to describe bit-parallel techniques.

"Bit-parallel," "byte-parallel," and "N-bit-parallel" all refer to a bit-parallel storage technique with a strata width s denoted by the name. We still use bit-parallel and bit-stratified as general descriptors for bit-stratified storage organizations. The strata width will be explicitly stated when necessary for clarity.

"Early stopping," "early pruning," and "early termination" all refer to an algorithm that stops before processing all bits in every processed value. Because the early stopping condition is evaluated between strata to determine if a stop can occur, we describe these behaviors as "runtime discovered early stopping." In contrast, an algorithm that determines the bit-location of an early stop before scanning begins has performed a "planned early stop."

We note that a stratum is intended as the smallest width of actionable bits in the context of early stopping. Strata width is usually, but not necessarily, linked to the particular implementation of the

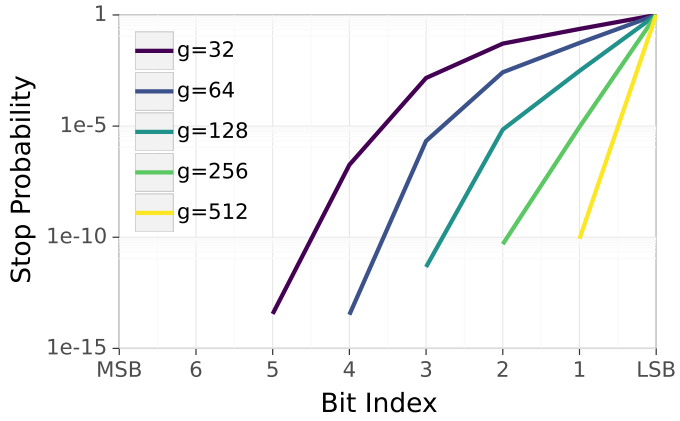

Fig. 2. Stopping probabilities when performing a predicate-based scan on the SSB column LO_QUANTITY, using a value of 15, at multiple group sizes (g). A (non-early) stop occurs when all bits of a value have been processed.

storage layer orchestrated by a bit-parallel method. For example, BitWeaving/V implemented using a "bit group size" of 4 is identified as having a strata width of $s = 4$, while ByteSlice has a strata width of $s = 8$ [9, 22].

The performance of columnar scans and fetches is significantly impacted by the strata width of a given bit-parallel technique. Reducing the strata width makes it possible to perform an early stop after processing a smaller number of bits. Depending on the group size parameter and the exact query, the average number of bits processed by the columnar scan may be reduced. However, reducing the strata width will usually increase the latency of fetching values from a bit-parallel column, as each stratum retrieved may require its own memory access.

2.3 (Runtime Discovered) Early Stopping

Having identified a standard set of parameters to describe bit-parallel techniques, we can now more deeply discuss the behavior of runtime discovered early stopping.

A bit-parallel technique processes g (group size) elements at each processing step. A discovered early stop can only occur when the early stopping condition for the entire group g is met. For all integer comparison operations, this condition is "data strata not equal to the corresponding predicate strata." A model of this behavior has been explored in previous work [9, 22], which we summarize here. Assuming a set of uniformly distributed values of b bits in length, the probability of a value to allow for early stopping after i_b bits have been evaluated is $\mathcal{P}(i_b) = 1 - (\frac{1}{2})^{i_b}$. This behavior is plainly understood, as each additional bit examined will remove half of the remaining values. This construction can be expanded to include g and a "fill factor" term f, which corresponds to the fraction of values present compared to the total cardinality of the number of bits. This expanded form expresses the previous probability as $\mathcal{P}(i_b, g, f) = (1 - (\frac{1}{2})^{i_b})^{gf}$.

While this model is accurate, its assumptions carry a significant caveat that impedes its general case usage: The fill factor reflects a random removal of elements from the overall cardinality. In practice, fill factors are regularly skewed. For example, consider the LO_QUANTITY column from the Star Schema Benchmark (SSB) [28], which contains values in the range $[1, 50]$ and is stored using 6 bits. The skewed fill reduces the value of evaluating an early stopping condition at some bit indices. To represent this impact, we replace the $\frac{1}{2}$ term with $I(b)$, where I represents a

function mapping the probability of bit b in the predicate being equal to bit b across all values in the scanned column's data. For example, given the LO_QUANTITY column and a query value of 15, $I(b) = \{0.62, 0.62, 0.48, 0.48, 0.5, 0.5\}$. After this modification, our early stopping model becomes: $\mathcal{P}(b, g) = (1 - \prod_{i=1}^{b} I(b))^g$.

Our model allows for an arbitrary extension of the bit representation of values with "leading zero" padding, which are bits that do not allow for any discovered early stopping opportunities. For example, extending the LO_QUANTITY column from 6 to 8 bits impacts the original example of a query predicate value of 15 as follows: $I(b) = \{1, 1, 0.62, 0.62, 0.48, 0.48, 0.5, 0.5\}$. These leading zeroes pose a fundamental challenge to existing bit-parallel methods, as they require processing but do not contribute to runtime discovered early stopping opportunities.

To illustrate the impact of our early stopping model, we apply the query "LESS THAN 15" to the previously discussed LO_QUANTITY column sized at 8 bits. We vary the group size g and depict the results in Figure 2. Points below 1e−15 are omitted from the figure. In general, large values of g negatively impact runtime discovered early stopping. By the next to last bit (bit index 1), only 0.3% of values have been early stopped when using a parallelism group size of $g = 128$. Once large values of g are reached, runtime discovered early stopping rarely finds opportunities to stop, significantly reducing its overall impact.

This early stopping behavior has been identified by the authors of ByteSlice and serves as part of the motivation to use byte-parallel ($b = 8$) storage instead of smaller bit-parallel ($b < 8$) storage configurations [9]. We agree that, as is, there are significant limitations to runtime discovered early stopping when working with large group sizes g. These results motivate an analysis into techniques that can complement runtime discovered early stopping.

2.4 Related Work

Bit-parallel computing techniques are deeply intertwined with our proposed encodings [9, 16, 19, 22, 23, 29]. These techniques have resulted in several successful implementations [3, 11, 30, 31]. While we focus on two specific implementations of bit-parallel techniques in this work (BitWeaving/V [22] and ByteSlice [9]), our generalization of bit-parallelism and the concept of a bit-stratified storage layer is a broadly applicable framework that could be applied to many existing bit-parallel implementations.

Further, recent work has investigated specialized hardware designed to perform bit-parallel compute [20, 35, 40–44]. Some of these recent works have implemented bit-parallel techniques within compute-capable memory units [20, 35, 41]. We find that our generalization of bit-parallel techniques can accurately describe these methods. As these techniques operate on parallelism sizes (w) significantly wider than those used by CPUs, our analysis of runtime discovered early stopping (Section 2.3) is also pertinent to these techniques.

Application-specific encodings have found success in a variety of environments [7, 10, 12, 21, 25, 32, 39]. These encodings were proposed to address the needs of a specific application, just as DFE and EDFE are focused on improving bit-parallel techniques when processing skewed data. Further, methods that automatically select an optimal type or encoding amplify the benefits of additional encoding options [13, 15].

3 ENCODING

In this section, we explore the motivations behind the design of forward encodings (FEs). Then, we propose two FEs: Data Forward Encoding (DFE) and Extended Data Forward Encoding (EDFE). DFE is intended as a simple implementation of a forward encoding to demonstrate the properties of an FE. In contrast, EDFE is usable as a general-purpose encoding that extends the DFE. After

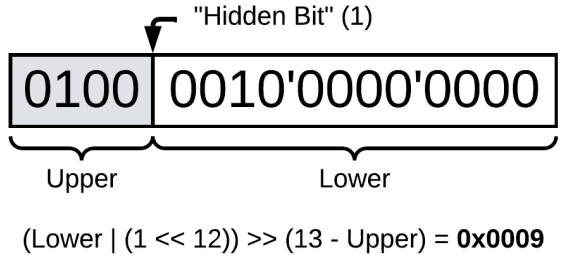

Fig. 3. The value 9 encoded in DFE. The upper field is highlighted in light gray. Note that the natural bit representation of 9 is 1001, which is present (shifted) in the lower field once the hidden bit (1) is made explicit.

describing the encodings, we explore how forward encodings enable new opportunities for early stopping. Finally, we examine the trade-offs between existing representations of integers, DFE, and EDFE.

3.1 Forward Encodings

In this section, we explore the concept of forward encodings. Then, we demonstrate how forward encodings enable a new set of early stopping opportunities.

A forward encoding (FE) is broadly intended to shift the bits that could be used for runtime discovered early stopping towards the most significant bit. Our design of forward encodings takes significant inspiration from floating-point formats that demarcate regions of a longer word as independent fields. However, in contrast to floating-point formats, FEs are still fundamentally integer encodings: forward encodings maintain full number precision and compatibility with integer comparison operations. Thus, while FEs share properties with floating-point encodings, they have a fundamentally different set of behaviors.

Previously, we demonstrated decoding several values from EDFE to their INT form (Figure 1). We use one of these values (9) as an example to showcase the decoding process of the data forward encoding (DFE). The DFE of 9 is shown in Figure 3. Note that the DFE of 9 has an upper field value of $upper = 4$.

First, the bit pattern representing 9 (1001) is always readily available once the hidden bit (1) is made explicit. This allows for a simple decoding process: $(lower|(1 \ll 12)) \gg (13 - upper) = 9$.

Second, we note that the upper field decremented by one ($upper - 1 = 3$) is the number of *salient* bits (starting from the MSB) in the lower field: Outside of these first three bits, all other bits will be

Value	INT	DFE	EDFE
8191	0001111111111111	1101111111111111	0101111111111111
2048	0000100000000000	1100000000000000	0100100000000000
2047	0000011111111111	1011111111111100	0010111111111111
9	0000000000001001	0100001000000000	0001000010000000
3	0000000000000011	0010100000000000	0000101000000000
2	0000000000000010	0010000000000000	0000100000000000
1	0000000000000001	0001000000000000	0000010000000000
0	0000000000000000	0000000000000000	0000000000000000

Table 2. Examples of integer values and their 16 bit representations using INT, DFE, and EDFE. The upper fields of DFE and EDFE are highlighted in light gray.

Algorithm 1: DFE Encoding Algorithm

input: Integer n represented using b bits.

```
1  if n = 0 then
       /* Encoding zero. Return zero.                          */
2      return 0;
3  else
4      clz ← CountLeadingZeroes(n);
5      len(u) ← ⌈log2(b)⌉;
6      len(l) ← b − ⌈log2(b)⌉;
7      if clz < len(u) − 1 then
           /* Number out of bounds for encoding.               */
8          throw ErrorOutOfBounds;
9      else
           /* Upper field records the shift.                   */
10         u ← (b − clz) ≪ len(l);
           /* Lower field preserves the value.                 */
11         l ← n ≪ (clz − len(u) + 1);
12         l_mask ← (1 ≪ len(l)) − 1;
           /* Combine fields with a bitwise OR.                */
13         return u | (l & l_mask);
14     end
15 end
```

shifted out during the decoding process and thus have no impact on the decoded value. Thus, in effect, DFE has trailing zeroes instead of leading zeroes.

The trailing zeroes of DFE are common to all forward encodings. FEs have *non-impactful trailing zeroes* instead of *impactful leading zeroes*. In the context of bit-parallel techniques, this has a profound impact: Information from the upper field can be used to set a maximum bound on the total number of strata that must be processed. Further, this maximum bound can also be used for comparisons, allowing the scan predicate to apply a maximum strata boundary on a columnar scan (described in Section 3.4). In the following sections, we explore how DFE and EDFE implement the properties of forward encodings.

3.2 Data Forward Encoding

In this section, we present the Data Forward Encoding (DFE). The algorithm to encode a value from an integer to its DFE encoded form is shown in Algorithm 1. We include examples of encoded values in Table 2.

Before we sketch our proof for Algorithm 1, we walk through an example of encoding and decoding. We also use this opportunity to link DFE to the properties of forward encodings illustrated previously.

We present the DFE representation of the value 9 in both Figure 3 and Table 2. Before we explain DFE, consider a simple idea: Imagine shifting a non-zero UINT (unsigned integer) value to the left until the MSB of the shifted value is 1. As the shifting process has guaranteed that this (new) MSB bit is always 1, we can assume this bit's value without storing it.

This process is precisely what occurs when encoding a value in DFE. The original value is shifted so that the first set bit is aligned with the MSB of the lower field. As this bit is always known to be 1, it can be omitted. Thus, for the DFE value of 9, by including the "hidden bit," we see that the base-2 representation of 9 `1001` has been shifted to the MSB of the lower field. By hiding this bit, we double the representable range.

Conceptually, the hidden bit resides between the lower and upper fields. In this context, the value stored in the upper field is similarly intuitive: the upper field records the number of salient bits in the lower field, starting with the hidden bit. Consider the example of encoding 9: the upper field has a value of 4; thus, the lower field and hidden bit together are 4 bits long. The first three bits of the lower field are `001`, which, when augmented with the hidden bit, becomes `1001`. It is trivial to decode 9 in DFE back to the original INT encoding through the use of bit manipulation: $(lower|(1 \ll 12)) \gg (13 - upper) = 9$.

To support Algorithm 1, we provide sketches of proofs that identify the key components of this alternative encoding.

Definition 3.1. A value of 0 has a DFE of 0.

Theorem 3.2. *Given a word length of b bits, DFE is a reversible encoding for all values n where $n \leq 2^{b-\lceil \log_2 b \rceil + 1} - 1$ and $n \geq 0$.*

Sketch of Proof. Given a value n that is represented using a word length of b, we note that $\lceil \log_2 b \rceil = len(u)$ is the number of bits required to represent b. In this construction, $len(u)$ bits are used to express the "upper" field (u) using a subset of fixed position bits in the existing word. The rest of the word is used for the "lower" field (l), the length of which is $b - len(u) = len(l)$. The values of u and l are stored using $len(u)$ and $len(l)$ bits from the word, respectively. We place these bits at positions relative to a word of size b using a few logical shift operations.

$u = (b - CLZ(n)) \ll len(l)$

$l = (n \ll (CLZ(n) - len(u) + 1)) \& ((1 \ll len(l)) - 1)$

$DFE(n) = u|l$

As also seen in Algorithm 1, prior to combining u and l into a single size b value, we apply a mask to l. This mask removes the leading bit of l; per Definition 3.1 and the behavior of the CLZ function, this removed leading bit is always 1. DFE can represent values of up to length $b - len(u) + 1 = len(l) + 1$, equal to a maximum value of $2^{b-\lceil \log_2 b \rceil + 1} - 1$.

To decode a value in DFE, we determine a shift and a value by reversing the process to construct the upper and lower fields.

$n_s = DFE(n) \& ((1 \ll len(l)) - 1) | (1 \ll len(l))$

$n_v = (len(l) + 1) - (DFE(n) \gg len(l))$

$n = n_v \gg n_S$

Note that n_v is restored bit pattern of the original value n: applying the shift n_s to n_v results in n.

We require that all shifts are logical shifts that shift in zeroes. If an arithmetic shift is used, then masks must be applied to set the shifted-in bits to zero.

As n has been defined to have an upper bound of $2^{b-\lceil \log_2 b \rceil + 1} - 1$, n can always be represented using $len(l) + 1$ bits.

The value n has never been lost, only shifted around within the existing b bits of the word. This behavior demonstrates that DFE is as much of a bit-representation remapping as it is a unique encoding. This mapping purposefully reduces the representable space of values to create additional early stopping opportunities closer to the MSB. As the mapping has been demonstrated to be reversible for values within the predefined bounds, DFE is a reversible encoding within the given bounds of n. □

Theorem 3.3. *Given two values x and y and the DFE encoding function $DFE()$, for any comparison operation $\bullet \in \{<, \leq, >, \geq, =, \neq\}$, $DFE(x) \bullet DFE(y) \leftrightarrow x \bullet y$.*

Sketch of Proof. First, we note that we have defined the encoded value of 0 to be 0. We identify the encoded value of 0 as our base case.

As u is constructed using $b - CLZ(n)$, we note that as $CLZ(n)$ decreases, u increases (Theorem 3.2). Further, l is constructed using a shift of $CLZ(n) - len(u) + 1$. By observation, incrementing n by one has one of two effects: Either u is incremented by one and $l = 0$ (as the only set bit in l is now the hidden bit) or the pre-shift value of l is incremented by one. Given these cases, we see that the sequence of possible values of $DFE(n)$ forms a sequence of monotonically increasing numbers, provided that the consecutive values of n are also monotonically increasing. Thus, for any comparison operation $\bullet \in \{<, \leq, >, \geq, =, \neq\}$, $DFE(x) \bullet DFE(y) \leftrightarrow x \bullet y$. □

We note that DFE is an encoding that can be used with various bit widths. The significant change between each width is the size of the upper field ($\lceil \log_2(b) \rceil$), as the upper field must contain enough bits to fully record any applied shift for the bit width in question. For example, DFE16, which is used in Table 2, has an upper field size of 4, while DFE32 would have an upper field size of 5.

Further, while the upper field must meet a minimum size requirement to support lower fields of increasing size, there is no requirement that the combined field size is always a machine word. This leads to the ability to form columns encoded in DFE9, where the last 7 bits of the lower field of DFE16 have been removed.

Bit packing at these exotic bit sizes is common in many storage formats, including Parquet [2], making the FE methods compatible with existing bit-based methods that store values using arbitrary bit sizes based on the representation needs of the stored column.

3.3 Extended Data Forward Encoding

As identified in Algorithm 1, DFE cannot represent values with fewer than $\lceil \log_2(b) \rceil - 1$ leading zeroes or negative numbers.

To remedy these shortcomings, we propose the "Extended Data Forward Encoding" (EDFE) as an extension to DFE. The EDFE addresses these issues by modifying the encoding process. This algorithm is shown in Algorithm 2.

Examining Algorithm 2, we see many similarities with the process to encode values in DFE (Algorithm 1). Table 2 and Figure 1 both show the result of encoding 9 in EDFE. This encoded form is intentionally similar to the encoding of 9 in DFE. EDFE includes two extensions to DFE: a "sign indicator" bit and a "format" bit.

To give an intuitive understanding of these two additions, we explore some of the values in Figure 1 (some of which are also included in Table 2). We emphasize how the additions to EDFE preserve its integer comparison functionality.

First, we examine the EDFE of 8191. For values that require more salient bits than what can be represented using the lower field, we bypass the encoding process and toggle the format bit. In this case, decoding is trivial: the format bit is inverted, restoring the integer to the original two's complement representation. As setting the format bit is based on a value threshold and the format bit is placed before the upper field, comparison functionality is preserved.

Next, we examine the values for 9 and −9. The EDFE encoded value of −9 is the bit inversion of the EDFE of 9. This is intentionally similar to the ones' complement integer representation, as the bit inversion extends the comparison properties of the positive number range to the negative number range [17].

As EDFE represents two significant changes from the DFE, it requires a separate proof sketch.

Algorithm 2: EDFE Encoding Algorithm

input: Integer n represented using b bits.

```
1  if n = 0 then
       /* Encoding zero. Return zero.                                  */
2      return 0;
3  else
4      abs ← AbsoluteValue(n);
5      clz ← CountLeadingZeroes(abs);
6      len(u) ← ⌈log2(b)⌉;
7      len(l) ← b − ⌈log2(b)⌉;
8      if clz ≤ 1 then
           /* Number out of bounds for encoding.                       */
9          throw ErrorOutOfBounds;
10     else if clz ≤ len(u) then
           /* Large number case. Invert format bit (using an XOR operation). */
11         return n ^ (1 ≪ (b − 2));
12     else
           /* Small number case. Similar to DFE.                       */
13         u ← (b − clz) ≪ (len(l) − 2);
14         l ← abs ≪ (clz − len(u) − 1);
15         l_mask ← (1 ≪ (len(l) − 2)) − 1;
16         if n < 0 then
               /* Negative Number case. Invert bits.                   */
17             return ∼(u | (l & l_mask));
18         else
19             return u | (l & l_mask);
20         end
21     end
22 end
```

Theorem 3.4. *Given a word length of b bits, EDFE is a reversible encoding for all values n where* $-1 \times (2^{b-2} - 1) \leq n \leq 2^{b-2} - 1$ *that obeys the comparison behavior of DFE (for any comparison operation* $\bullet \in \{<, \leq, >, \geq, =, \neq\}$*,* $DFE(x) \bullet DFE(y) \leftrightarrow x \bullet y$*).*

Sketch of Proof. EDFE extends DFE. This extension is implemented via a right shift by 2 on the upper and lower fields, creating space for two additional bits. The most significant bit is identified as the "sign indicator" bit. The bit following the most significant bit is identified as the "format" bit. Both the sign indicator bit and the format bit are cleared for positive values small enough to be encoded similarly to DFE.

If a value is negative, the EDFE is computed using the magnitude of the value to encode. As the last step in the encoding, all bits of the encoded value are inverted, setting the sign indicator bit. In this way, the sign indicator bit allows for compatibility with signed integer comparison hardware.

If a value's magnitude is too large for a DFE-like encoding (i.e. $CLZ(ABS(n)) \leq len(u)$), the value is encoded by inverting the format bit. As this is a magnitude-based threshold, EDFE values still form a monotonically increasing sequence, preserving the properties of the DFE for comparisons.

To prove that EDFE maintains the comparison properties of DFE, we sketch a proof by exhaustion. There are four cases, based on both the sign and the magnitude of the value to be encoded.

Case 1: Positive, inactive format bypass (Algorithm 2, lines 12, 19). This case is the DFE-like case; no behavior is changed.

Case 2: Positive, active format bypass (lines 10, 11). As the format bit is closer to the MSB than the upper field, integer comparisons interpret format-set EDFE values as larger than format-unset EDFE values. As the format bit is set by a threshold based on magnitude (via the leading zero count), it holds true that format-set EDFE values will always be larger than format-unset EDFE values.

Case 3: Negative, inactive format bypass (lines 12, 17). We note that the bit-inversion step is shared between EDFE and the process to encode values as a ones' complement negative integer. Due to this similarity, using the MSB as the sign bit allows for using existing signed integer comparison hardware.

Case 4: Negative, active format bypass (lines 10, 11). As the value is already a negative integer (MSB is set), toggling the format bit applies the same thresholding as before. This toggling still obeys integer comparison rules, as the toggling is based on the magnitude of the value: all EDFE values where the format bit has not been toggled (is set for negative values) are greater than EDFE values where the format bit has been toggled (is not set for negative values).

Thus, while the feature set of the DFE has been expanded into the EDFE, all of the previously existing properties regarding the correct semantics of comparison operations are preserved. □

3.4 FE-enabled Early Stopping

The upper field can be used to generate two kinds of stopping opportunities: planning a stop before performing a predicate-based scan and discovering a stop while fetching a value.

FEs exchange the padded leading zeroes for trailing zeroes, a behavior seen in Table 2. However, these zeroes are not equivalent: Trailing zeroes in DFE do not impact the result of comparisons. This is the foundation for our definition of FEs having *non-impactful trailing zeroes*, as opposed to the *impactful leading zeroes* present in existing integer encodings.

For any value encoded in DFE, $l_t = len(u) + u - 1$ bits, starting from the most significant bit, must be processed to reconstruct the original value. We reiterate that u contains the number of salient bits in the lower field (starting with the hidden bit). Thus, $u - 1$ is the number of bits that must be read in addition to the upper field. For any two values a and b in an FE, they can be compared using $min(l_t(a), l_t(b))$ bits. In the case of a columnar scan, l_t can be calculated for the predicate before any values are loaded, allowing a bit-parallel scan to plan how many strata must be processed. This behavior enables the planning of early stops before a scan begins, reducing the number of strata evaluated without relying on a runtime discovered early stopping condition.

While impactful leading zeroes cannot be skipped without impacting the result of a comparison, non-impactful zeroes will not change the result of a comparison, regardless of whether or not they have been processed. This property is the basis for *FE-enabled early stopping*, which is complementary to the existing early stopping technique. FE-enabled early stops can be applied to both predicate-based scans and fetch operations. First, the FE-enabled stop for scans is implemented as a *planned stop*, where the predicate value determines a stop location using l_t before the scan begins. Second, the FE-enabled stop for fetches is implemented as a runtime discovered early stop with a modified stopping condition: l_t determines the maximum number of strata required to reconstruct a value. We explore both kinds of FE-enabled early stops in this section.

First, we present an example of an FE-enabled planned stop when evaluating a predicate-based columnar scan. Consider a column containing uniform-random distributed values $\{0, 1, 2, 3, 9\}$ encoded in DFE, where we apply a filtering operation of "LESS THAN 2." The minimum number of bits to reconstruct a value in DFE is $l_t = len(u) + u - 1$, as we must read both the upper field and

all salient bits of the lower field. As seen in Table 2, our predicate value of 2 has an upper field of $u = 2$ and thus requires $len(u) + 1 = 5$ bits to represent the value fully in 16-bit form. However, for a comparison between any two DFE encoded values a and b, the number of bits required from each value is $\min(l_t(a), l_t(b))$. For example, when comparing our predicate value 2 against a data value of 9, only the first $\min(5, 7) = 5$ bits from each value are required to perform the computation. While 9 is not fully representable using only 5 bits, existing binary comparisons will still identify that `00100` < `01000`, leading to a correct comparison result. Similarly, using 7 bits to represent the value 2 does not change the comparison result (`0010000` < `0100001`). This example showcases the core property of non-impactful trailing zeroes and how the usage of DFE/EDFE allows for reducing the reliance on discovered early stopping to reduce the number of processed strata.

We now show how the properties explored by the example can be broadly applied to create additional early stops in bit-parallel predicate-based scan and fetch operations.

Given some data array d encoded in DFE16 ($b = 16$) and the query $d > c$, we iterate over each element in d. Due to the properties of non-impactful trailing zeroes, we know that the maximum number of bits required to perform all comparisons is $l_t(c)$. In the case of DFE16 ($len(u) = 4$) and $c = 2$ ($l_t = 5$), 5 bits of each 16 bit value are processed. In contrast, $c = 9$ determines that 7 bits of each value are processed. We identify this behavior as predicate-determined early stopping, where the maximum number of bits to process per value is known before the scan begins.

Minimizing the number of loaded bits can also be adapted to fit fetch operations, using a runtime discovered early stop, albeit in a different form than existing runtime discovered early stops. By loading u before loading any bits in l, the number of l bits to be read can be determined during the fetch process. As $len(u)$ and $len(l)$ are determined by the type of the array, a bit-parallel fetch can be broken down into a set of required reads and optional reads depending on the strata width, where the optional reads are performed based on u. This process also functions when reading multiple u in parallel. For example, an upper bound of u across many values can either be calculated at runtime (possibly by applying a vertical bit-wise OR across a group of u values) or be stored as metadata during column construction. As only trailing zeroes reside after the salient bits of a forward encoded value, reading a few more bits of each value, while not perfectly efficient, still reduces the average number of bits processed per value.

3.5 Encoding/Decoding Implementation

The algorithms that describe the encoding process for (E)DFE are purposefully designed to map directly to modern CPU instructions. Using scalar operations, GCC 12.3 compiles decoding DFE32 to INT27 to 9 instructions (4 if the zero branch is taken). Decoding EDFE32 to INT31 is 10, 17, or 18 instructions depending on branching (format, sign). Encoding INT27 to DFE32 is at most 21 instructions, and encoding INT31 to EDFE32 is at most 26. Further, these scalar kernels can be directly mapped to SIMD; for example, the AVX512 instructions that perform "SRLV"/"SLLV" and "LZCNT" operations can be directly applied to the discussed kernels.

3.6 Usage of FEs

Forward encodings are lossless encodings. To facilitate a lossless transformation, the upper field requires allocating additional bits. For example, a column that requires 28 bits to be represented as a two's complement integer would require 32 bits in DFE and EDFE. The additional bits of overhead are used by the upper field to store the salient bit count before the data bits, which benefits bit-parallel scan and fetch operations on skewed data.

The upper field must grow to accommodate the number of salient bits (Section 3.2). However, this leads to smaller integer widths allocating a proportionally larger fraction of their overall size to

the upper field (DFE8: $len(u) = 3$), while larger integer widths require proportionally fewer upper field bits (DFE64: $len(u) = 6$). The format bit in EDFE assists in mitigating this drawback.

3.7 Trade-offs of FEs

DFE and EDFE preserve the validity of existing comparison techniques. Techniques based on these properties, such as existing bit parallel work, can perform both scan and fetch operations using DFE and EDFE without technique-level modifications.

Comparing DFE and EDFE, we find that the extensions that EDFE provides come at the cost of increasing encoding and decoding complexity. However, supporting signed values is necessary for usage outside of constrained environments. Thus, while the DFE is more performant in applicable circumstances, these situations may not align with the needs of an application looking to better leverage bit-parallel techniques. Similarly, DFE reduces the number of scanned bits compared to EDFE in situations that do not benefit from the usage of a format or sign bit. Thus, we find that both encodings have their own respective use cases depending on the needs of the target application. One could imagine a single system using existing integer encodings, DFE, and EDFE on the same platform, where the system selects the most appropriate encoding to use in each situation.

The concept of automatic typing and data encoding is well explored by other work [13, 15] and is broadly related to block-based storage layers [2, 5, 33]. There has been a recent surge in block-based storage formats, such as Parquet [2] and Arrow [33]; even one of the datasets we use as part of the evaluation of DFE and EDFE is distributed using the Parquet format [27]. One could envision that the choice of encoding (INT, DFE, or EDFE) is added as block-level metadata. As DFE and EDFE can be used with a variety of bit widths, these alternative encodings can be freely applied on top of existing data. Further, in data systems where in-place updates are allowed, a block could be converted between encodings if beneficial.

FE-native arithmetic is expensive when using integer hardware; a limitation shared by floating-point representations. However, encoding to (and decoding from) FEs is demonstrably lightweight (see Section 4.6), minimizing the need for dedicated FE compute units.

Finally, we recognize that using an alternative integer representation is a significant departure from existing computing ideas. Given how the existing integer implementation has not changed in over 50 years [38], this change opens a new set of challenges. However, we emphasize that forward encodings are intended for use when deemed beneficial by the data platform, similar to other alternative encodings used by modern systems [10, 21, 25, 39].

4 EVALUATION

Throughout this paper, we have emphasized that DFE and EDFE are intended to complement existing bit-parallel data processing techniques. We selected two background works to establish a core set of properties shared between bit-parallel techniques (Section 2.1). These background techniques use different strata widths ($s = 4$ for BitWeaving/V using a bit group size of 4, and $s = 8$ for ByteSlice). We compare these two bit-parallel methods with a columnar baseline, which we implement using a contiguous in-memory array. Consistent with existing results, ByteSlice outperforms BitWeaving/V in all cases [9]. However, as our results will demonstrate, both DFE and EDFE can significantly reduce the performance gap between these two bit-parallel techniques.

Our experiments aim to address the following questions:

Q1. What is the performance benefit of using forward encodings to accelerate bit-parallel scans? How impactful is setting an early stopping bound using l_t? (Sections 4.2 and 4.3.)

Q2. What is the fetch performance of forward encoded columns? Does the salient bit count optimization outweigh the cost of decoding values? (Section 4.4.)

Q3. What is the impact of the usage of forward encodings on the compressibility of a column? (Section 4.5.)
Q4. What is the cost to encode to or decode from an FE? (Section 4.6.)

Each experiment shares several standard parameters, which we identify in the remainder of this section.

Our experiments were run on a CloudLab c220g5 machine [6]. All experiments were run using a single CPU (an Intel Xeon Silver 4114). Unless otherwise stated, our experiments use all 20 threads of the CPU. All datasets used in our evaluation fit in the available main memory of this machine (192 GB). Each experiment was run ten times to generate an average result. The variance between experiments was negligible; thus, to aid readability, we do not include error bars in our figures.

4.1 Datasets

To evaluate the impact of forward encodings, we perform a series of microbenchmarks on selected columns from three real-world datasets: individual contributions to political campaigns in the US [8], taxi data from New York City, USA [27], and energy consumption readings from households in London, UK [37]. We refer to these datasets as the "FEC," "Taxi," and "Home" datasets respectively. We describe the selected columns from these datasets in Table 3 and depict the distribution of each column in Figure 4. These datasets represent real-world data with skewed values, which is the target for forward encodings. The seven selected columns, from across the three datasets, facilitate a quantitative evaluation of how effectively forward encodings improve bit-parallel techniques when processing skewed data. As an example of the skewed nature of the chosen columns, while many taxi rides are short trips within NYC, some riders arrange long-distance (over 100 miles) trips with drivers for negotiated rates. Similarly, of the approximately 63 million individual political campaign contributions from 2021 to 2022, there were 778 individual contributions of 1 million USD or more.

We apply a minimal amount of cleaning to the datasets. As we evaluate both DFE and EDFE, we remove all negative values from the selected columns (the Home dataset did not have negative values). We also removed all rows from the Taxi dataset that were missing values. To store the columns as unsigned integers, we apply unit transformations to fixed precision decimals (such as

Dataset	Column	Units	90%	99%	99.9%	99.99%	Max	INT	DFE	EDFE
FEC	Transaction Amount (Amt)	Dollars	112	2k	5.8k	100k	125M	27	31	32
Home	Milliwatts per Half Hour (mW)	Milliwatts	481k	1.49M	2.8M	4.67M	10.76M	24	28	30
Taxi	Trip Distance (Dist)	0.01 Miles	896	2.02k	2.92k	5.8k	38.97M	26	30	32
Taxi	Fare Amount (Fare)	Cents	3.1k	6.46k	11.85k	25k	40.11M	26	30	32
Taxi	Tip Amount (Tip)	Cents	575	1.49k	2.5k	6k	140.02k	18	22	24
Taxi	Tolls Amount (Tolls)	Cents	0	655	1.9k	2.83k	91.19k	17	21	23
Taxi	Total Amount (Total)	Cents	4.33k	8.18k	14.43k	28.44k	40.11M	26	30	32

Table 3. The evaluated columns from each dataset. The 90th, 99th, 99.9th, and 99.99th percentile values of each column and the maximum value per column are included. We also include the minimum number of bits required to represent each column using each evaluated encoding (INT, DFE, EDFE), based on each column's maximum value.

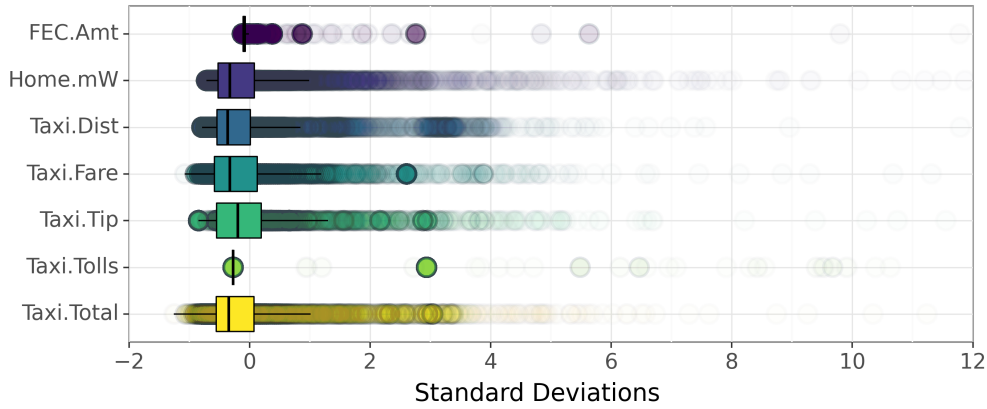

Fig. 4. The distribution of 10,000 samples from each explored column across the three datasets. Each sample is drawn as a mostly transparent circle; visible points indicate the overlap of samples.

dollars and cents to cents). We replicated each dataset multiple times, resulting in each column having a size of about 400 million elements.

Note that removing negative elements from the datasets impacts the performance of EDFE, as branch prediction is able to quickly learn that the sign bit is never set when processing an EDFE encoded value. However, this impact is minor due to the small number of instructions required to encode to and decode from (E)DFE.

The distributions of the skewed columns are shown in Figure 4. This figure was generated by normalizing and then sampling each column 10,000 times. We use sampling to showcase the heavily skewed nature of the entire dataset, as opposed to each column being fairly normal except for a few outliers. For example, the tolls column of the Taxi dataset (Taxi.Tolls) is skewed due to about 92% of the values in the column being equal to zero. Of the remaining 8% of values, we clearly see a number of repeated values; one such value is $6.55 (represented as 655), the current "E-ZPass" fare rate for multiple bridges and tunnels around NYC [26]. The other columns showcase similar behavior, with varying degrees of skew and multimodal-ness. As we demonstrate throughout the rest of our evaluation, forward encodings allow bit-parallel columns to process skewed data more efficiently.

The three real-world datasets we use differ from the previous evaluations of bit-parallel techniques, usually performed using TPC-H or SSB [28, 36]. Earlier, we explored the behavior of runtime discovered early stopping when processing the "LO_QUANTITY" column from SSB (Section 2.3), which contains values in the range of $[1, 50]$. As previously shown by the early stopping model of the LO_QUANTITY column, each additional bit evaluated causes many records to discover a stop, irrespective of the exact query predicate. This behavior does not extend to columns containing skewed data using the two's complement integer representation, as the large outlier values require a large representation range and thus necessitate the usage of leading zeroes in the non-outlier data values. Thus, while each column is stored using the minimum number of bits per encoding (Table 3), the representation ranges of these bit counts far exceed the median value of each column. In these cases, runtime discovered early stopping (traditional early pruning) is significantly less effective, necessitating alternative mechanisms to provide stopping opportunities for bit-parallel columns.

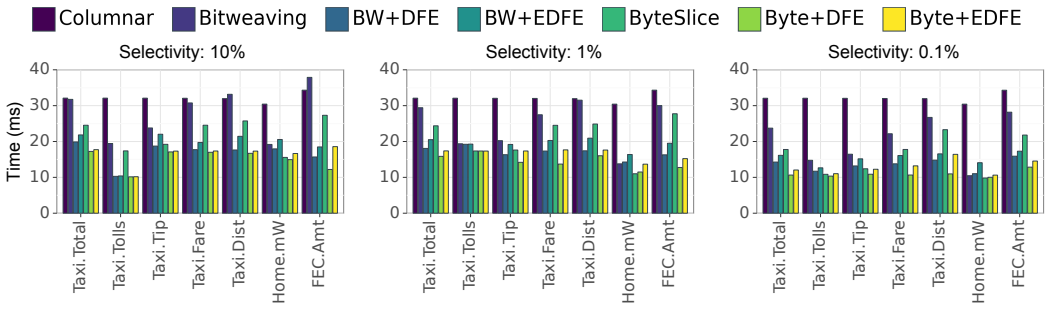

Fig. 5. Average scan performance of the "data greater than constant" query applied to each column using 20 threads, separated into individual figures by filter selectivity.

4.2 Scan Performance

To evaluate the performance impact of forward encodings on columnar scans, we perform multiple scans using the following query:

```
SELECT count(*) FROM column WHERE (column > threshold)
```

The threshold is a constant query parameter that results in a query with 10%, 1%, or 0.1% selectivity. For example, the 10%, 1%, and 0.1% selectivity thresholds for the campaign contribution transaction amount (FEC.Amt) are $112, $2000, and $5800 USD, respectively. We specifically use the "greater than" operator because we are concerned with processing the outlier data from the aforementioned datasets.

We include our overall scan results in Figure 5, where each scan was performed using 20 threads. Overall, ByteSlice with DFE is the best-performing technique. This is not particularly surprising, as it combines the state-of-the-art bit-parallel technique with the forward encoding that emphasizes performance over wider applicability. In the 20-thread configuration, ByteSlice with DFE has an average geometric mean speed-up over ByteSlice of 1.47x. In contrast, ByteSlice with EDFE has an average geo. mean speed-up of 1.29x. This is primarily due to the extra two bits, which do not benefit runtime discovered early stopping for the selected datasets.

BitWeaving/V, implemented with a strata width ("bit group size") of $s = 4$ requires more memory reads than ByteSlice ($s = 8$) to process the upper field of each value. Because of this difference in storage organization, ByteSlice+DFE usually outperforms BitWeaving/V+DFE. In addition, the smaller strata width is also more impacted by the skewed data, as the increased number of strata

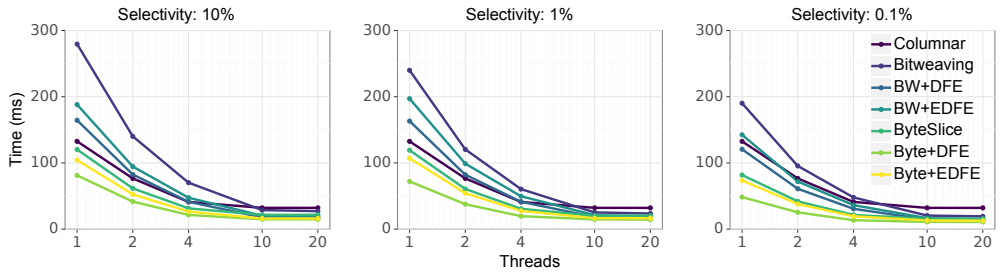

Fig. 6. Geometric mean of scan performance of all evaluated columns, across all selectivity and threading configurations

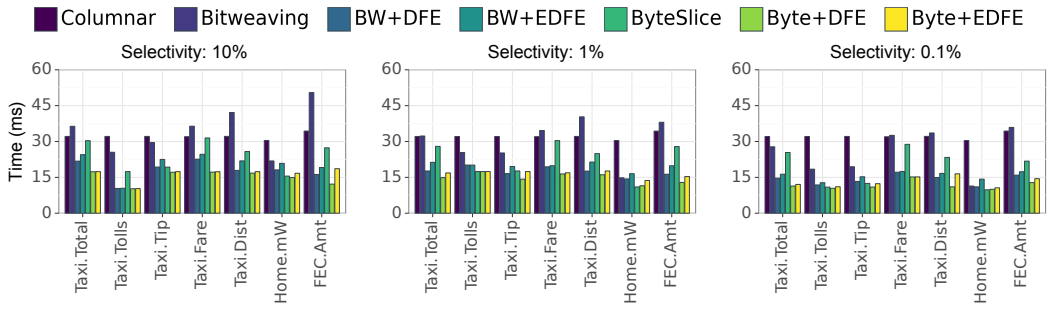

Fig. 7. Average scan performance of each column using 20 threads when the alternative "lower than" query is applied, separated into individual figures by filter selectivity.

can incur more memory accesses when relying solely on runtime discovered early stopping. BitWeaving/V+DFE has an average speed-up of 1.54x over BitWeaving/V (both using 20 threads), while BitWeaving/V+EDFE has a respective speed-up of 1.35x. These results match our expectations; due to BitWeaving/V's smaller strata width, the planned stop skips more BitWeaving/V strata than ByteSlice strata.

We also vary the number of threads used to perform each scan-based filter. The results from these experiments are shown in Figure 6, which depicts the geometric mean performance of each columnar scan across multiple thread configurations at each selectivity threshold. The performance stagnation between 10 and 20 threads is due to the used CPU only having 10 physical cores with two logical cores each; using the second logical core of each physical core does not improve the scan performance of the evaluated techniques. This result is expected, as scan operations are generally limited more by memory performance than CPU performance. Further, this behavior results in "free" CPU cycles that can be used for tasks such as decoding while waiting for memory transfers. We measure the utilization of these otherwise unused cycles in Section 4.6.

We note that the evaluated scans selected values above the 90%, 99%, and 99.9% percentiles, as opposed to queries that select beneath the 10%, 1%, and 0.1% percentiles. We depict the results of these "lower than" queries in Figure 7. While smaller values have fewer salient bits, the skewed distributions of the columns result in many of these small values being present in a tight cluster. This clustering of values significantly impacts the performance of existing bit-parallel techniques as

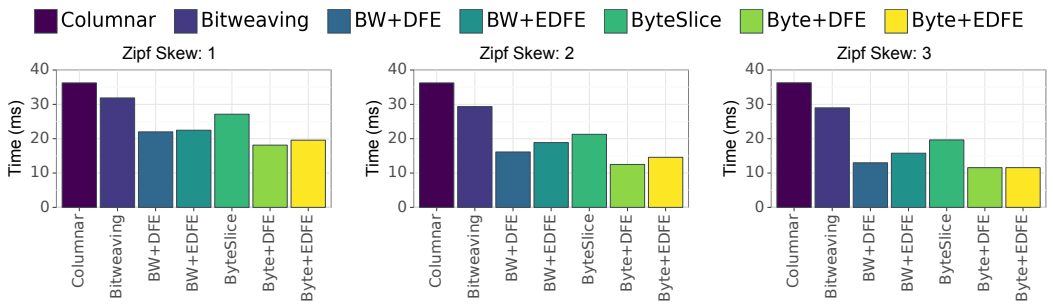

Fig. 8. Average scan performance of each skewed, synthetic dataset using 20 threads, separated into individual figures by the skew parameter for each Zipf distribution.

it impairs traditional runtime discovered early stopping. In contrast, the salient bit count prevents a lack of discovered early stopping opportunities from significantly decreasing scan performance.

4.3 Synthetic Data Scan Performance

We also perform scans on synthetic columns with skewed distributions. We generate columns of 400M elements using one uniform-random and three Zipf (skew of 1, 2, and 3) distributions of values in the range [1,1M]. We perform a scan-based filter operation on the column with the query "greater than 100."

The results from the experiments on synthetic, skewed columns are shown in Figure 8. The speed-up of using (E)DFE increases as the skew increases. Across all storage configurations, DFE speed-ups ranged from 1.4x to 2.2x, while EDFE speed-ups ranged from 1.4x to 1.8x. These results align with our expectations due to the impact of parallelism group size g on scans: the less skewed distributions have a larger percentage of distribution occupied by values equal to the scan predicate (100), and thus require a larger fraction of the groups to be evaluated without the benefit of runtime discovered early stopping. However, while the speed-up of applying (E)DFE to the uniform random column was lower (about 1.7x for BitWeaving/V and 1.5x for ByteSlice), the time to complete the query for FE-encoded columns containing the uniform-random distribution was similar to the Zipf-3 distribution, at around 12ms; the speed-up was reduced because BitWeaving/V and ByteSlice performed significantly better when the columns were not skewed (20ms and 17ms vs. 32ms and 27ms for BitWeaving/V and ByteSlice, respectively). These results mirror the initial exploration of runtime discovered early stopping, as illustrated by our original model (Section 2.3).

We also scan a column of uniform-random distributed values in the range $[1, 50]$ using the same experimental setup as before (Section 4.2). This scan is similar to experiments performed in existing work [9, 22]. The distribution of this column is the same as the previously explored SSB column "LO_QUANTITY" (Section 2.3). We perform a scan-based filter operation at 10% selectivity (column > 45). This result is depicted in Figure 9. We note that the bit width of the column is 6, 9, and 8 for INT, DFE, and EDFE, respectively.

ByteSlice and ByteSlice+EDFE have similar performance, as both methods use the same stratification (one 8-bit stratum), in contrast to the two strata that ByteSlice+DFE uses. As DFE requires an additional stratum to be processed when using both BitWeaving/V and ByteSlice, it cannot outperform the INT encoding. Further, BitWeaving/V+EDFE incurs significant overhead when

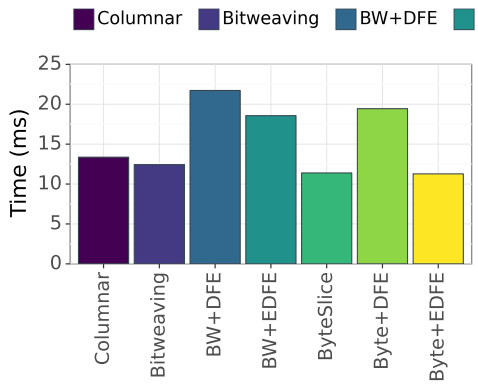

Fig. 9. Average scan performance of the synthetic uniform-random column using each storage format.

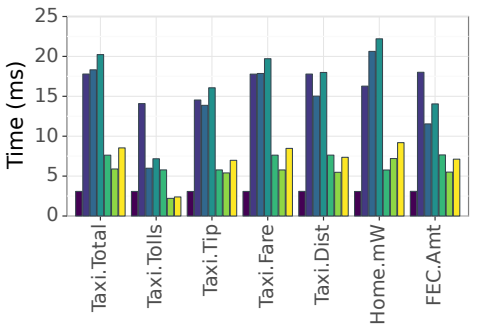

Fig. 10. Average time to perform one million random fetch operations on each column using each storage format.

testing the format bit to determine whether the second stratum should be loaded. This behavior incurs branch misses and randomizes the strata access pattern, resulting in additional overhead that does not improve performance over streaming the full 8 bits of all values from memory. Prior work has identified this behavior as a source of decreased performance [22].

Broadly, we include these results because forward encodings are designed for the specific case of processing skewed datasets with bit-parallel techniques. On a uniform-random dataset, ByteSlice using EDFE offers no performance improvement over ByteSlice using the existing integer encoding, though we note that it does not decrease performance either. Existing bit-parallel techniques have contributed numerous ways to process uniform-random datasets; bit-parallel techniques currently lack the ability to efficiently process skewed columns, as showcased by our other results. These results on a uniform-random dataset demonstrate that our encodings improve the performance of scanning and fetching from skewed datasets without a significant decrease in performance when applying the same operations to columns without skew.

4.4 Fetch Performance

Our fetch microbenchmark is implemented using the following steps: randomly generate a row ID, fetch the corresponding element, and then decode the value to INT (if necessary). We randomly fetch one million elements from each column to evaluate the fetch performance of forwarded encoded columns. This benchmark poses a significant challenge for bit-parallel storage formats: by rearranging bits in memory to be more amenable to columnar scans, additional memory reads must be performed to reconstruct the original values. Thus, storage formats that use more strata have a higher average fetch latency. However, as we will demonstrate, the usage of the salient bit count allows for some strata to be bypassed during the fetch operation, as those strata do not contain salient bits for the particular fetched value. Explicitly flushing the CPU cache did not significantly impact the fetch benchmark results.

The results of our fetch microbenchmark are depicted in Figure 10. We emphasize the fetch results from the "Taxi.Tip" column. Note that 99% of the values in this column are under approximately $15 (Table 3, 1493), which requires 11, 15, or 17 bits to represent using INT, DFE, or EDFE respectively. Thus, while the INT column requires fewer bits to represent all values (18 vs. 22/24 for DFE/EDFE), 99% of fetches using DFE/EDFE require only 15/17 bits instead of 18 bits. However, because BitWeaving/V and ByteSlice use strata widths of 4 and 8, respectively, increasing the bits fetched from 15 to 17 causes an additional stratum to be accessed. Thus, DFE, but not EDFE, improves the fetch performance of the Taxi.Tip column.

While ByteSlice using DFE is always the best-performing fetch configuration, the exact results depend on the average number of strata required to reconstruct the original value. Broadly, the salient bit count of forward encodings can significantly improve the fetch performance of skewed

Col.	BL	b.INT	b.DFE	b.EDFE	B.INT	B.DFE	B.EDFE
F.Amt	4.25	4.14	3.58	3.67	4.27	**4.42**	4.19
H.mW	**2.34**	2.01	1.83	1.82	2.11	2.03	2.07
T.Dist	2.41	**3.09**	2.53	2.55	3.02	3.02	2.86
T.Fare	**3.60**	2.79	2.54	2.53	3.27	3.29	3.40
T.Tip	3.10	3.38	2.81	2.87	**3.66**	3.42	3.33
T.Tolls	**24.11**	11.06	9.36	10.67	12.29	17.76	17.23
T.Total	2.45	2.54	2.27	2.29	**2.62**	2.61	2.50

Table 4. The compression ratio of each evaluated column, using each storage format when compressed using zstd. "BL" is the baseline columnar method, "b" refers to BitWeaving/V-stored columns, and "B" refers to columns stored in the ByteSlice format. The best compression technique for each column is in bold.

bit-parallel columns. DFE provides a geometric mean fetch performance improvement of 1.33x and 1.19x for ByteSlice and BitWeaving/V, respectively. While EDFE does not offer a significant performance improvement (2% and 4% for the respective techniques), these results are expected due to the cost of retrieving the sign and format bit.

4.5 Compressiblity

We evaluate the compressibility of the proposed encodings when using both bit-parallel and byte-parallel storage formats. We compress each of the used dataset columns using Zstandard (zstd) at compression level 3 [24]. These results are shown in Table 4. There is no clear-cut choice of storage format most suitable for compression when using zstd alone. Note that we calculate our compression ratio against the baseline columnar representation, which advantages the columns that use fewer bits to represent the uncompressed values. Overall, the best choice of compression format relies on the complex interactions between a column's data, bit-parallel storage format, and encoding.

As a second experiment, we also compress the columns by applying either run-length (RLE) or delta encoding to the columns before zstd, the results of which are shown in Tables 5 and 6, respectively. Applying run-length or delta encoding before compression does not significantly impact the results. We note that the evaluated datasets are fairly antagonistic to RLE due to the large number of unique values in each dataset. For example, the average run length of the Taxi.Distance column is about 1.01, which effectively doubles the column size by repeatedly storing runs of length 1. Both RLE and Delta compression demonstrate two related behaviors of alternative integer encodings. First, RLE and other techniques that compress a series of integers as a stream of symbols (dictionary, move-to-front, etc.) are not impacted by the exact value of the underlying integer but rather by the repetition of the symbol. Thus, changing the bit representation of a value does not significantly impact the overall performance of RLE compression. Second, while delta compression is impacted by the distance between values, the recorded distances can be interpreted as a pattern of symbols, regardless of the exact bit representation of each delta.

Col.	BL	b.INT	b.DFE	b.EDFE	B.INT	B.DFE	B.EDFE
F.Amt	**6.88**	6.24	5.59	5.49	6.69	6.02	6.10
H.mW	**4.52**	3.52	3.33	3.33	3.33	3.25	3.26
T.Dist	4.31	**5.44**	4.59	4.51	4.92	4.53	4.68
T.Fare	5.36	4.87	4.56	4.51	4.85	4.92	**5.37**
T.Tip	4.76	**5.52**	4.86	4.84	5.30	5.32	5.34
T.Tolls	10.86	7.93	14.78	16.03	8.49	**34.87**	29.52
T.Total	4.28	**4.51**	4.16	4.18	4.21	4.16	4.10

Table 5. The compression ratio of each column after compressing using run-length encoding then zstd.

Col.	BL	b.INT	b.DFE	b.EDFE	B.INT	B.DFE	B.EDFE
F.Amt	3.44	2.46	3.33	3.30	2.78	3.70	**3.76**
H.mW	**2.32**	1.83	1.84	1.75	2.09	1.98	1.97
T.Dist	2.19	2.19	2.51	2.43	2.35	**2.68**	2.61
T.Fare	**3.15**	2.07	2.48	2.32	2.59	2.79	3.00
T.Tip	2.45	2.61	2.46	2.38	**2.76**	2.73	2.75
T.Tolls	**22.66**	6.34	6.34	6.14	11.51	16.65	16.00
T.Total	1.91	1.95	2.27	2.12	2.14	2.35	**2.29**

Table 6. The compression ratio of each column after compressing using delta encoding then zstd.

Overall, applying run-length and delta encodings to each of the evaluated columns before compressing with zstd does not lead to one technique being more compressible than the others.

4.6 Encode and Decode Performance

We compare our proposed FEs with existing integer encoding techniques to better frame the performance cost of (E)DFE encode and decode operations. We evaluate DFE and EDFE against a unary code, the Elias δ, γ, and ω codes [7], as well as Rice codes with turntable parameters of 4 and 8 [32]. We evaluate a uniform random distribution with values in the range $[1, 50]$, similar to the previously described LO_QUANTITY column. First, we record the time it takes to encode the entire column. Then, we randomly fetch 10M (2.5% of the column) values.

For lightweight encodings like (E)DFE, the encoding and decoding process can often be "free" when it is performed in otherwise wasted cycles, such as during the time spent waiting for memory. Compared to an unencoded baseline, DFE and unary encodings incurred no significant amount of overhead during decoding. γ and Rice 4/8 both had decoding overheads in the range of 2.5% to 3.5%. δ, EDFE, and ω had the largest encoding overheads, of 12%, 35%, and 108% respectively. Encoding performance is similar, where γ, unary, and Rice 8 encodings did not incur a significant overhead. δ, DFE, and Rice 4 added an additional 5%, 6%, and 9% encoding overhead, respectively. EDFE and ω incurred the largest encoding overheads of 19% and 55%. Overall, while encoding and decoding DFE is faster than EDFE, both are lightweight codes with costs comparable to other integer codes.

However, these alternative encodings are not FEs. The Elias codes do not preserve compatibility with integer comparison hardware. The unary component of Rice codes limits their applicability to columns with skewed values; while using large divisors limits the size of the unary component, the increased number of remainder bits results in more integer-like early stopping behavior. Thus, while we compare the cost to encode and decode (E)DFE against a number of existing encodings, we also note that each is tailored for its own specific use cases.

5 DISCUSSION

Before we performed our evaluation, we established four questions that our experiments were intended to explore:

A1. Forward encodings limit the maximum number of strata that must be evaluated when performing a scan-based filter operation using a constant predicate. When the filtering predicate is small compared to the maximum value that can be represented by the bit width of the column, many strata can be skipped.

A2. The salient bit count is applicable to both scan and fetch operations. As l_t controls the number of strata retrieved, the memory accesses required to perform a fetch on a forward encoded column are based on the value fetched and the strata width of the column rather than the overall strata count. In the case of significantly skewed datasets, the additional strata that non-FE columns must process harm performance, a penalty that is not incurred by (E)DFE-encoded columns.

A3. Forward encodings are not significantly more or less compressible than existing integer encodings.

A4. Encoding and decoding (E)DFE has a performance cost similar to other integer codes.

As the scan-related results demonstrate, forward encodings improve the performance of bit-parallel scanning techniques by reducing their data sensitivity. The previously explored early stopping model (Section 2.3) predicts that early pruning may be ineffective depending on the bit-distributions of values in a column. These predictions held when exploring our selected skewed datasets (both real-world and synthetic), as the bits composing each value did not equally contribute

opportunities for runtime discovered early stopping. Forward encodings address this data sensitivity by expanding early stopping to include planned early stops. The salient bit count l_t sets an upper bound on the number of strata that must be evaluated, reducing the average number of bits examined during the scan. Beyond reducing the execution time of scan-based filtering, this upper bound also helps stabilize the performance of scans on bit-parallel columns by reducing predicate-based variability in query execution time.

By selecting a strata size to use when splitting data vertically, bit-parallel techniques introduce a complex relationship between the bit-representation of the data and the performance of database operators. Forward encodings reduce the variability of many bit-parallel storage configurations by using the salient bit count, providing a speed-up by elevating the performance of non-ideal cases. While we focus on skewed distributions, our previously explored early stopping model is an effective tool when exploring other data distributions.

6 CONCLUSIONS AND FUTURE WORK

Runtime discovered early stopping is a core component of modern bit-parallel techniques. In this work, we explored a new family of integer encodings (forward encodings) that shift the salient bits of existing integer representations closer to the MSB, which enables more efficient early stopping in bit-parallel methods. Further, forward encodings (FEs) also enable additional opportunities for early stopping, such as planning a stop before performing a predicate-based scan or discovering an early stop during a fetch operation. These alternative stopping methods significantly improve the performance of bit-parallel techniques when processing skewed data, where the existing runtime discovered early stopping technique (known as early pruning) is ineffective.

We have also proposed a systematic way to think about encodings and storage organization. Using this framework, we proposed two FEs: DFE and EDFE. These forward encodings allow for a new set of interactions with bit-parallel techniques while preserving compatibility with existing integer comparison operations. Further, all optimizations performed by existing techniques are applicable when using FEs, allowing for DFE and EDFE's use as replacements for existing integer representations in bit-stratified methods.

In our evaluation, we demonstrated how forward encodings improve the performance of two existing bit-parallel storage formats when processing skewed data. While the research directions of encoding and bit-parallel techniques are orthogonal, the choice of encoding profoundly impacts bit-parallel methods due to the influence encodings have on early stopping.

Besides the two FEs we introduce, it seems possible to design additional forward encodings. By rethinking the encoding of integer types, we hope to encourage future research into reimagining how FE-based methods might be applicable in speeding up computations in data applications beyond just the predicate-based columnar scans and fetch operations that we consider. Given the complex interactions between encoding, storage format, and data distribution, there is ample opportunity for database engines to create significant performance improvements through automated optimization techniques that select the best combination of these orthogonal, yet connected, parameters.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under grants OAC-1835446 and SHF-2312739. Additional support was provided by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. The "CloudLab c6420" machine utilized was provided as part of the CloudLab service [6].

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[2] Apache Parquet. 2022. Apache Parquet Documentation. https://parquet.apache.org/docs/

[3] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. 2012. Business Analytics in (a) Blink. *IEEE Data Engineering Bulletin* 35, 1 (2012), 9–14.

[4] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (dec 2008), 77–85. https://doi.org/10.1145/1409360.1409380

[5] Craig Chasseur and Jignesh M. Patel. 2013. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *Proc. VLDB Endow.* 6, 13 (aug 2013), 1474–1485. https://doi.org/10.14778/2536258.2536260

[6] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1–14.

[7] P. Elias. 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory* 21, 2 (1975), 194–203. https://doi.org/10.1109/TIT.1975.1055349

[8] Federal Election Commission. 2022. Individual Contributions. https://www.fec.gov/introduction-campaign-finance/how-to-research-public-records/individual-contributions/

[9] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 31–46. https://doi.org/10.1145/2723372.2747642

[10] Ivan Flores. 1956. Reflected Number Systems. *IRE Transactions on Electronic Computers* EC-5, 2 (June 1956), 79–82. https://doi.org/10.1109/TEC.1956.5219803

[11] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database - An architecture overview. *IEEE Data Eng. Bull.* 35 (03 2012), 28–33.

[12] S. Golomb. 1966. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory* 12, 3 (1966), 399–401. https://doi.org/10.1109/TIT.1966.1053907

[13] Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César Hidalgo. 2019. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1500–1508. https://doi.org/10.1145/3292500.3330993

[14] IEEE. 2019. IEEE Standard for Floating-Point Arithmetic. https://doi.org/10.1109/ieeestd.2019.8766229

[15] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 843–856. https://doi.org/10.1145/3448016.3457283

[16] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-Wise Parallel Predicate Evaluation. *Proc. VLDB Endow.* 1, 1 (aug 2008), 622–634. https://doi.org/10.14778/1453856.1453925

[17] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, Chapter 4.1: Positional Number Systems.

[18] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proc. VLDB Endow.* 5, 1 (sep 2011), 61–72. https://doi.org/10.14778/2047485.2047491

[19] Leslie Lamport. 1975. Multiple Byte Processing with Full-Word Instructions. *Commun. ACM* 18, 8 (aug 1975), 471–475. https://doi.org/10.1145/360933.360994

[20] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 556–569. https://doi.org/10.1109/HPCA47549.2020.00052

[21] Yinan Li, Craig Chasseur, and Jignesh M. Patel. 2015. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1509–1524. https://doi.org/10.

1145/2723372.2737787
[22] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast Scans for Main Memory Data Processing *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/2463676.2465322
[23] Yinan Li and Jignesh M. Patel. 2014. WideTable: An Accelerator for Analytical Data Processing. *Proc. VLDB Endow.* 7, 10 (jun 2014), 907–918. https://doi.org/10.14778/2732951.2732965
[24] Meta Platforms Inc. [n. d.]. Zstandard. https://facebook.github.io/zstd/
[25] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. https://doi.org/10.48550/ARXIV.2209.05433
[26] New York City Metropolitan Transportation Authority. 2021. Car Toll Rates. https://new.mta.info/fares-and-tolls/bridges-and-tunnels/tolls-by-vehicle/cars
[27] New York City Taxi and Limousine Commission. 2022. TLC Trip Record Data - Yellow Taxi Trip Records, 2022. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[28] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–252.
[29] Patrick O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (Tucson, Arizona, USA) *(SIGMOD '97)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/253260.253268
[30] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-up Approach. *Proc. VLDB Endow.* 11, 6 (oct 2018), 663–676. https://doi.org/10.14778/3184470.3184471
[31] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-Time Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*. 60–69. https://doi.org/10.1109/ICDE.2008.4497414
[32] R. Rice and J. Plaunt. 1971. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communication Technology* 19, 6 (1971), 889–897. https://doi.org/10.1109/TCOM.1971.1090789
[33] Neal Richardson, Ian Cook, Nic Crane, Dewey Dunnington, Romain François, Jonathan Keane, Dragoș Moldovan-Grünfeld, Jeroen Ooms, and Apache Arrow. 2022. arrow: Integration to 'Apache' 'Arrow'. https://github.com/apache/arrow/, https://arrow.apache.org/docs/r/.
[34] Denis Rinfret, Patrick O'Neil, and Elizabeth O'Neil. 2001. Bit-Sliced Index Arithmetic. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/375663.375669
[35] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 273–287. https://doi.org/10.1145/3123939.3124544
[36] The Transaction Processing Council. 2022. TPC-H Benchmark (Version 3.0.1). http://www.tpc.org/tpch/
[37] UK Power Networks. 2014. SmartMeter Energy Consumption Data in London Households. https://data.london.gov.uk/dataset/smartmeter-energy-use-data-in-london-households
[38] J. von Neumann. 1993. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing* 15, 4 (1993), 27–75. https://doi.org/10.1109/85.238389 (Note: This is a reprint of the original 1945 document, digitized in 1992 by Michael D. Godfrey and published in 1993.).
[39] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus
[40] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (aug 2009), 385–394. https://doi.org/10.14778/1687627.1687671
[41] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. 2021. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 251–264. https://doi.org/10.1109/ISCA52012.2021.00028
[42] Lingxi Wu, Rasool Sharifi, Ashish Venkat, and Kevin Skadron. 2022. DRAM-CAM: General-Purpose Bit-Serial Exact Pattern Matching. *IEEE Computer Architecture Letters* 21, 2 (2022), 89–92. https://doi.org/10.1109/LCA.2022.3201168
[43] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/564691.564709

[44] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. 1349–1350. https://doi.org/10.1109/ICDE.2012.148

Received April 2023; revised July 2023; accepted August 2023