ByteStore: Hybrid Layouts for Main-Memory Column Stores

Pengfei Zhang  Ziqiang Feng†  Eric Lo  Hailin Qin
Chinese University of Hong Kong  †Google

Abstract—The performance of main memory column stores highly depends on the scan and lookup operations on the base column layouts. Existing column-stores adopt a homogeneous column layout, leading to sub-optimal performance on real workloads since different columns possess different data characteristics. In this paper, we propose ByteStore, a column store that uses different storage layouts and corresponding encoding methods for different columns. Extensive experiments show that ByteStore outperforms homogeneous storage engines by up to 5.2×.

I. INTRODUCTION

Main-memory column stores are popular for fast analytics on big relational data [1]–[3]. Analytical workloads are usually read-only and denormalization is often used to transform tables into one or a few outer-joined wide tables such that expensive joins and nested queries can then be flattened as simple scan-based queries [4]–[6]. Under this scan-heavy paradigm, most of the query time is spent on two operations that directly consume the base columns: scan and lookup. The scan operation on a column filters row IDs whose column values satisfy a predicate (e.g., \( \text{year} < 2018 \)). Given these row IDs, the lookup operation extracts column values into their plain form (e.g., \( \text{int32} \)) to be consumed by upstream operations, such as sorting and aggregation. The overall performance of queries thus heavily depends on the scan and lookup performance on the base columns [5], [6].

Existing main-memory column stores all encode the native column values into codes [5]–[7] and adopt a homogeneous storage layout, i.e., the same storage layout is used across all data columns. Specifically, we use the term storage layout to refer to the format of bits of codes stored in the main memory. Figure 1 hints the drawback of such a homogeneous approach, which shows the scan performance two state-of-the-art main-memory column layouts: ByteSlice [5] and PE-VBP [7] on a real dataset [8] with 23 columns. It is clear that no one size fits all: PE-VBP, which is skew-aware, outperforms ByteSlice, which is skew-agnostic, on 9 out of 23 columns. Meanwhile, ByteSlice outperforms PE-VBP for the rest. The above motivates us to design ByteStore, a new storage engine for column stores. ByteStore adopts different storage layouts for different data columns. ByteStore is different from other hybrid storage engines like HYRISE [9]. HYRISE uses a hybrid of row/column storage for HTAP (hybrid transaction-and-analytic) workloads. By contrast, ByteStore is a pure column store that focuses on OLAP (online analytical) workload but it is hybrid in terms of using different encoding and layouts to store the columns. To our best knowledge, this paper is the first to use different storage layouts for different base columns in main-memory column stores.

ByteStore is beyond a simple integration of ByteSlice and PE-VBP. First, ByteStore uses a new storage layout in place of PE-VBP for skewed columns. The problem of PE-VBP is that it actually has a sub-optimal lookup performance and long code length. Therefore, one contribution of this paper is PP-VBS (stands for Prefix Preserving Variable Byte Slice), a new storage layout that uses byte as the storage unit and a new prefix-preserving method to encode the skewed columns. Unlike ByteSlice, PP-VBS is skew aware. Different from PE-VBP, PP-VBS has balanced performance on both scan and lookup.

With ByteSlice for non-skewed data and PP-VBS for skewed data, ByteStore has to make a binary decision between the two storage layouts for a given data column. Although the decision boundary looks complicated — ByteSlice and PP-VBS outperform each other based on an array of factors beyond data skewness (e.g., selectivity, domain size), we observe that there exists a simple yet reliable decision boundary. Therefore, another contribution of this paper is an experiment-driven column-layout-advisor.

Our last contribution is a comprehensive experimental study based on not only synthetic data but also six open datasets and workloads. Experiments show that ByteStore outperforms any homogeneous storage engines.

The remainder of this paper proceeds as follows: Section II contains necessary background information; Section III presents the new storage layout PP-VBS; Section IV presents the column-layout advisor; Section V presents the experimental results. Section VI discusses related works; Section VII gives the conclusion.
II. BACKGROUND AND PRELIMINARY

A. Scan-based OLAP Framework

In modern main memory analytic databases, data is often stored in a compressed form. Dictionary encoding are mostly used compression scheme and it replaces the values of a column by a unique integer code and storing the mapping between values and codes in a separate data structure, the dictionary. Modern analytical column stores transform complex queries into scan-heavy queries on denormalized wide tables [4]–[6]. These queries typically have extensive WHERE clauses requiring scan on many columns. A scan takes as input a dictionary-encoded column, and a predicate of types $=$, $\neq$, $>$, $<$, $\leq$, $\geq$, BETWEEN. The scalar literals in the predicates (e.g., 2018 in WHERE year $< 2018$) are encoded using the same dictionary to encode the column values. By using order-preserving encoding, comparison on codes yields correct result for comparison of the original column values. For predicates involving arithmetic or similarity search (e.g., LIKE predicates on strings), codes have to be decoded before a scan is evaluated in the traditional way.

The scan operation filters all matching column codes, and outputs a result bit vector to indicate the matching row IDs. The bit vector makes it easy to combine scan results in logical expression (conjunction or disjunction), and handle NULL values and three-valued Boolean logic [6]. After scan, column codes involved in projection, aggregation or sorting may need to be retrieved and decoded into their canonical forms (e.g., int32) via a lookup operation.

Under this framework, scan and lookup are the two major operations whose performance directly depend on the base columns’ layouts. Other operations such as sorting and aggregation are independent of the base columns.

B. Bit-Packed

The Bit-Packed layout [10], [11] aims to minimize the memory bandwidth usage when filtering big data. Bit-Packed is skew-agnostic and codes are fixed-length. Figure 2(a) shows an example with 11-bit column codes. Codes are tightly packed together in the memory, ignoring any byte boundaries. To be specific, the first code $v_1$ is put in the 1-st to 11-th bits whereas the second code $v_2$ is put in the 12-th to 22-nd bits and so on.

To evaluate a scan on a bit-packed column, it is necessary to unpack the tightly packed data into SIMD registers. Since a code may initially span 3 bytes (e.g., $v_3$) as shown in Figure 2(a), each code has to be unpacked into a 32-bit bank of the SIMD register. Under AVX2 architecture (i.e., the length of SIMD registers is 256-bit), scan is run in 8-way (256/32) data level parallelism. In other words, eight 11-bit codes (e.g., $v_1 \sim v_8$) are loaded from memory and aligned into eight 32-bit banks of the register. After unpacking, data in the SIMD register (e.g., $w_i$ in Figure 2(b)) is ready to be processed by a scan operation. Figure 2(b) shows how to evaluate a predicate $v > 129$ on the unpacked codes using AVX2’s 8-way greater-than comparison instruction _mm256_cmpgt_epi32(). After that, the scan starts another iteration to unpack and compare the next 8 codes with $W_c$. In the example above, although 8-way parallelism is achieved in data processing, many cycles are actually wasted during unpacking. To align the 8 codes into the eight 32-bit banks, three extra SIMD instructions (i.e., Shuffle, Shift and Mask) are carried out. Furthermore, as 0’s are used to pad up with the SIMD soft boundaries, for above example, any data processing operation is wasting $(32 - 11) \times 8 = 168$ bits of computation power per cycle.

To retrieve a code $v_i$ from bit-packed layout, one has to gather all bytes that it spans. For example, to look up $v_3$, Bytes# 02~04 are fetched from the memory. As a code may span multiple bytes under the bit-packed format, retrieving one code may incur multiple cache misses, particularly when those bytes span across multiple cache lines.

C. PE-VBP

Li et al. [7] proposed a variable-length encoding scheme called Padded Encoding (PE) that leverages data skew to accelerate scan. They combine PE with a vertical bit packing (VBP) storage layout [6] to form PE-VBP, a skew-aware column scan technique. Figure 3(a) shows how VBP (without PE) stores a block of eight 13-bit column codes $v_1 \sim v_8$ in memory. Each horizontal box $W_j$ is a contiguous memory region. The $j$-th bits of the codes are stored in $W_j$. In other words, the bits belonging to $v_1 = (1100000001111)_2$ are...
vertically distributed across different memory regions (i.e., \( W_1 \sim W_{13} \)).

The scan on VBP leverages a key insight that within each processor there is abundant “intra-cycle parallelism” as the processor’s ALU operates on multiple bits in parallel. To illustrate, consider the evaluation of predicate “\( v = 6156 \)" (Figure 3(a)). The scan algorithm compares the bits of \( v_1 \) with the bits of 6156 in a bit-by-bit fashion, from the most significant bit to the least significant bit. Since the first bits of \( v_1 \sim v_8 \) are stored together in \( W_1 \), they can be loaded into the CPU as a single memory word and processed in parallel using bitwise instructions (in real implementation, a memory word is usually a SIMD register, e.g., 256 bits). It compares the first bit of 6156 — \((1)\) — with all bits in \( W_1 = (10111110)_2 \). At this point, only \( v_2 \) and \( v_8 \) fail to pass the predicate because their first bit is 0. The other six codes are inconclusive. Thus, the scan can continue to the next (second) bit, and repeats. After scanning the 12-th bits (i.e., \( W_{12} \)), there are no codes whose first 12 bits match 6156’s first 12 bits, safely declaring all codes in this block (i.e., \( v_1 \sim v_8 \)) fail the predicate. Thus there is no need to load \( W_{13} \) into the CPU and process it. In this case, the scan is said to stop early and can proceed to the next block of codes.

PE-VBP builds on VBP by leveraging data skew to increase the chance of early stop during a scan. Similar to Huffman encoding, PE assigns shorter codes for frequent values and longer codes for infrequent values. To store PE-encoded values in VBP, shorter codes are padded with zeros to align with the longest code. Figure 3(b) shows how PE-VBP encodes and stores the same eight column values, where the grey bits are the encoded value (called code) of \( a \) cannot be a prefix of that of \( b \), or vice versa. This property increases the maximum length of PE-encoded codes, for example, from 13 to 16 bits in Figure 3.

PE-VBP achieves faster scan than VBP when the data is skewed. Figure 3(b) shows the scan can now stop early at the 6-th bits (i.e., \( W_6 \)), about \( 2 \times \) faster than VBP because frequent values \( v_1, v_3, v_4, v_5, v_6, v_7 \) are encoded with only 6 bits, as opposed to 13 bits in VBP. In general, PE-VBP increases the likelihood of early stop after scanning each bit. Unfortunately, both VBP and PE-VBP suffer from very expensive lookup operation. When reconstructing a column code, both VBP and PE-VBP must retrieve every single bit of a code from different memory words. Each bit is likely to reside in a different cache line, and incur a cache miss, costing hundreds of CPU cycles. As shown in [5], the expensive lookup often offsets the performance gain of fast scan from the whole query point of view. The problem is exacerbated in PE-VBP since the storage layout of PE-VBP is intrinsically at odds with its variable-length encoding scheme. As shown in Figure 3(b), all short codes have to be padded with zeros to align with the longest code. It retrieves 16 bits instead of 13 bits in VBP (see Figure 3(a)) to re-build a single code.

D. ByteSlice

ByteSlice is a technique that achieves balanced high performance of scan and lookup. Figure 4 shows how ByteSlice [5] arranges 64 11-bit column codes \( v_1 \sim v_{64} \) in main memory. Each code is chopped into separate bytes. The \( j \)-th byte of all codes are stored in a continuous memory region. In other words, the bytes of a code \( v_i \) are vertically distributed in different memory regions. Each of these regions is called a byte slice. At query time, a 256-bit memory word such as \( W_1 \) is loaded into the CPU and processed with byte-level AVX2 SIMD instructions, achieving 32-way parallelism. Similar to VBP, ByteSlice enjoys the benefit of early stop during scan. For example, the scan on Block 1 (\( v_1 \sim v_{32} \)) in Figure 4 may stop early after processing \( W_1 \), without needing to load and process \( W_2 \).

ByteSlice is also efficient at lookup. It distributes a \( k \)-bit code across \( \lfloor k/8 \rfloor \) memory words. In Figure 4, a lookup on \( v_i \) will incur at most 2 memory accesses, as opposed to 11 in VBP. The number \( \lfloor k/8 \rfloor \) is typically small (between 1 to 3), so it can be overlapped with other instructions in the CPU’s instruction pipeline [5]. Nonetheless, ByteSlice is skew-agnostic. All codes in Figure 4 are encoded into two bytes regardless.

III. PP-VBS: A NEW STORAGE FOR SKewed COLUMNS

PE-VBP is good for skewed data but suffers from poor lookup performance. ByteSlice has an excellent balance between scan and lookup, but on non-skewed columns only. In this section, we present PP-VBS (Prefix Preserving Variable Byte Slice). PP-VBS aims to accelerate scans on skewed data without jeopardizing the efficiency of lookups. Similar to ByteSlice and PE-VBP, PP-VBS is a suite of techniques to encode column values into integer codes (Section III-A), store the codes’ bytes in main memory using a specialized layout (Section III-B), and perform efficient scan and lookup operations on top (Section III-C).

A. Prefix Preserving Encoding (PPE)

We begin by arguing that the prefix-free property, commonly found in variable-length encoding [12]–[15] and used in PE-VBP, is actually unnecessary in the context of main-memory data processing but reduces the performance of both scans and lookups. Traditionally, variable-length encoding schemes were designed for reducing the communication cost when sending a message across the network [13]. In that context, codes are often concatenated as a sequential byte stream, so the prefix-free property is crucial for ensuring the message can be decoded without ambiguity. Concretely, a prefix-free encoding...
PPE is based on building a prefix-preserving $M$-way encoding tree. In Figure 6, $M = 4$. Each node in the tree contains $M - 1$ slots. Each slot from the left to the right represents a sub-code from 1 to $M - 1$, e.g., from (01)$_2$ to (11)$_2$ in Figure 6. We call it as slot sub-code, marked in a dashed box. Each non-leaf node has $M$ pointers connecting to $M$ child nodes. Each pointer from the left to the right carries a sub-code from 0 to $M - 1$, e.g., from (00)$_2$ to (11)$_2$ in Figure 6. We call it pointer sub-code, marked in a dashed circle. Note that slot sub-codes do not start from 0 for the sake of disambiguation, as explained later.

Given an encoding tree, we obtain the encoding of a value $v$ by concatenating all pointer sub-codes on the path from the root to $v$, and $v$’s own slot sub-code. In Figure 6, the code of value ‘C’ is simply its slot sub-code (01)$_2$, since it is in the root node; the code of ‘A’ is (0001)$_2$, which is formed by concatenation of pointer sub-code (00)$_2$ and slot sub-code (01)$_2$. Figure 6 also illustrates the intuition of our encoding method: we encode high-frequency values (‘C’, ‘G’, ‘K’) with shorter codes, and low-frequency values with longer codes.

1) Encoding Numerical Columns (Order-preserving): In addition to being prefix-preserving, the encoding tree in Figure 6 is also order-preserving. That is, the tree’s in-order traversal gives the ordered sequence of the encoded values. It enables the ordinal comparison of two column values (e.g., ‘A’ vs. ‘C’) by comparing their codes directly (e.g., (0001)$_2$ vs. (01)$_2$). When comparing two integer codes of different lengths, we treat them as if the shorter code were padded zeros at the end to align with the longer one. Thus, (0001)$_2 < (0100)$_2 ⇔ ‘A’ < ‘C’. For this reason, we do not use 0 as slot sub-codes. Otherwise, it cannot differentiate between a short code with padding (01 00)$_2$ and a long code ( 01 00)$_2$. Notice this is different from prefix-free, as there are still prefix-sharing codes (e.g., ‘C’ and ‘E’). Order-preserving is crucial for numerical columns as it enables efficient evaluation of all types of predicates ($=, \neq, <, >, \leq, \geq$) without decoding codes into their represented values.

To ensure that we do not underutilize the available short codes, all slots of each non-leaf node in a prefix-preserving encoding tree should be filled with higher-frequency values. For example, the root node in Figure 6 is full of values. Therefore, we build the encoding tree in a depth-first search manner and fill all non-leaf nodes in the paths from the root node to leaf nodes.

Without loss of generality, given a sequence of distinct values in ascending order $A_0 < A_1 < \cdots < A_{n-1}$, and their corresponding frequencies $Q = \{Q_0, Q_1, \cdots, Q_{n-1}\}$, Algorithm 1 shows the pseudocode of constructing a prefix-preserving encoding (PPE) for numerical columns. The outputs of Algorithm 1 are the codes $C$ and the corresponding code lengths $L$. Algorithm 1 builds a 256-way tree, here $M$ equals to 256($=2^8$), since the code lengths are multiple of 1 byte (8 bits) in our setting. At first, we initialize the elements of $C$ and $L$ as 0’s (Lines 22-23). Then Algorithm 1 invokes a function PPE_NUMERICAL recursively to construct the encoding tree.
Algorithm 1: PPE for Numerical Columns

\[ A \sim A_\text{min}, \]  
\[ A \sim A_\text{max}, \]

**Input:** \( n \) distinct values \( A = \{A_0 < A_1 < \ldots < A_{n-1}\} \), value weights \( Q = \{Q_0, Q_1, \ldots, Q_{n-1}\} \).

**Output:** \( n \) value codes \( C = \{C_0, C_1, \ldots, C_{n-1}\} \), code lengths \( L = \{L_0, L_1, \ldots, L_{n-1}\} \).

- \( b \): the level of the parent node
- \( s \): the start index
- \( e \): the end index

1. **Function** PPE\_NUMERICAL\((C, L, b, s, e)\):

   // two terminating conditions
   2. if \( e - s < 256 \) OR \( b \geq 2 \) then
   3. \( \beta = \log_2(e - s + 1); \)
   4. for \( i = s, \ldots, e - 1 \) do
      5. \( C_i = (C_i < b \times s + 1) + (i - s); \)
      6. \( L_i = b + \beta; \)
   7. else
      8. Find the largest 255 frequencies in \( Q_s \sim Q_{e-1} \), store their indices in an ascending array \( I \).
      9. for \( t = 0, \ldots, 254 \) do
         10. \( L_t = b + 1; \)
         11. \( C_{I_t} = (C_{I_t} < b \times s + 8) + t + 1 \).
      12. for \( t = 0, \ldots, 253 \) do
         13. for \( i = I_t + 1, \ldots, I_{t+1} - 1 \) do
            14. \( C_i = (C_i < b \times s + 8) + t + 1; \)
            15. PPE\_NUMERICAL\((C, L, b + 1, I_t + 1, I_{t+1}); \)
      16. for \( i = s, \ldots, I_0 - 1 \) do
         17. \( C_i = C_i < b \times s; \)
         18. PPE\_NUMERICAL\((C, L, b + 1, s, I_0); \)
      19. for \( i = I_{254} + 1, \ldots, e - 1 \) do
         20. \( C_i = (C_i < b \times s + 8); \)
         21. PPE\_NUMERICAL\((C, L, b + 1, I_{254} + 1, e); \)
      22. for \( i = 0, \ldots, n - 1 \) do
         23. PPE\_NUMERICAL\((C, L, 0, 0, n); \)

Among the parameters passed to \texttt{PPE\_NUMERICAL}, \( s \) and \( e \) represent the start and end index, which indicate that values \( A_s \sim A_{e-1} \) will be encoded in a sub-tree by this call; \( b \) indicates the level of its parent node. At the first round, \texttt{PPE\_NUMERICAL} selects the 255 largest frequencies from \( Q_0 \sim Q_{n-1} \) and stores their corresponding array indices in an ascending array \( I \) (Line 8), i.e., \( I_0 < I_1 < \ldots < I_{254} \). These most frequent 255 values will reside in the root node and be encoded as 1-byte codes (Lines 9-11), i.e., \( (1)_{10} \sim (255)_{10} \). This ensures the most frequent values are encoded with the shortest 1-byte codes. After that, the values between adjacent slots in the root node will be encoded by building a sub-tree, e.g., the values in the positions \( \{I_1, I_2\} \) of the sequence \( A \), by calling \texttt{PPE\_NUMERICAL} recursively (Lines 12-15). Note the extra efforts to deal with the leftmost and rightmost sub-trees (Lines 16-21). Since a non-leaf node must be full before we create any child nodes by recursion, we can see that all (non-leaf) internal nodes in a prefix-preserving encoding tree are full.

The recursion has two terminating conditions (Line 2). When \( e - s < 256 \), it means a leaf node has enough slots to hold all the values \( A_s \sim A_{e-1} \) pending to be encoded. Then their slot sub-code will be assigned \((1)_{10} \sim (e-s)_{10}\) (Line 5), and their code length will be \( b + 1 \) bytes (Line 6), where \( b \) is the level of its parent node. There is no need to compare the frequencies of these values.

When \( b \geq 2 \), it is a corner case that the encoding tree will become unbalanced, which has a negative impact on scan and lookup performance. In order to reduce the maximum code length and alleviate the tree unbalance, after 2 levels, the values in the positions \( [s,e] \) are packed into one leaf node even though the number of values is larger than 255 (Lines 3-6). Since in real-world datasets, the number of distinct values of columns (aka domain size) is mainly in the range of \((2^7,2^{10})\) [6], [10], we believe that the value 2 is a reasonable threshold.

2) **Encoding Categorical Columns (Non Order-preserving):**

While all previous works achieve order-preserving for all columns, we observe extra opportunity by forgoing this property for categorical columns. Typically, categorical columns (e.g., city name, brand name, color) are only involved in equality predicates (=, ≠), but not range predicates (<, >, ≤, ≥). By omitting the need to handle range predicates, we can build an encoding tree that is more balanced and remove exceptionally long codes.

Given a sequence of distinct values \( A = \{A_0, A_1, \ldots, A_{n-1}\} \) with descending frequencies \( Q = \{Q_0 > Q_1 > \ldots > Q_{n-1}\} \). Algorithm 2 shows the pseudocode of PPE for categorical columns. Similarly, the outputs of Algorithm 2 are the codes \( C \) and corresponding code lengths \( L \). The algorithm builds the encoding tree in a breadth-first search manner. Intuitively, it starts by assigning the 255 most frequent values to the top-level slots (i.e., root node). Then, it assigns the next 256 × 255 (pointers × slots) most frequent values to the second level. If there are still values to encode, it assigns them to the third level, for up to 256² × 255 (pointers² × slots), and so forth. Therefore, Algorithm 2 bounds the height of the tree by a balanced tree that can hold all values (Line 1). It first fills in the non-leaf nodes level-by-level (Lines 3–8). When filling the possibly non-full last level, it evenly distributes the remaining values among the last level nodes to maximize entropy (Lines 9–15). The order does not matter between codes, assuming no range comparison is needed in the workload. Furthermore, a PPE tree for categorical columns is balanced.

**String columns:** It is worth noting that certain string columns will encounter range predicates (e.g.,<,>) in analytical queries. We regard such string columns as semi-categorical and treat them as same as the numerical columns. In other words, we construct an order-preserving PPE for a semi-categorical column using Algorithm 1. In addition, since it is impossible to carry out a similarity search (LIKE predicate) on the codes directly, in our implementation, the codes are decoded to original strings before a similarity search is evaluated in the traditional way.

**B. Variable Byte Slice (VBS)**

PPE encodes column values into variable-length integer codes logically. Now, we describe how to store PPE values in the main memory physically in a Variable Byte Slice (VBS).
Without loss of generality, we explain with the predicate in the form of $v B$.

\[
B = \left\lfloor \log_{2^{56}} (n + 1) \right\rfloor;
\]

\[
\alpha = 0;
\]

For $b = 1, \ldots, B - 1$

\[
\text{for } c = 0, \ldots, 2^{56b-1} - 1 \text{ do}
\]

\[
\text{for } i = 1, \ldots, 255 \text{ do}
\]

\[
C = c < 8 + i;
\]

\[
L_0 = b;
\]

\[
\alpha = \alpha + 1;
\]

\[
\text{if } \alpha = n \text{ then}
\]

\[
\text{exit(0)};
\]

VBS further divides a column into many blocks. Each block contains data belonging to 32 consecutive codes, under the current implementation using 256-bit AVX2 instructions.

Figure 8 illustrates a VBS column having three blocks (96 codes), with maximum code length of 3 bytes. Each horizontal line in the figure represents a contiguous memory region. The dashed line outlines all data belonging to Block 1. Not surprisingly, Block 1 takes up 32 bytes in $BS_1$. Among the 32 codes in Block 1, only 10 have the second byte. So Block 1 takes up only 10 bytes in $BS_2$. Ditto for $BS_3$. In general, $BS_j$ always has fewer bytes than $BS_{j+1}$. When data is moderately and highly skewed, a large portion of codes are one byte, and $BS_{[\geq 2]}$ have very few bytes. A segment of 32 bytes (4 bytes) from the bitmasks $M_2$ and $M_3$ are used to store this sparsity information of Block 1. In summary, Block 1 in VBS uses $4 \times 2 + 32 + 10 + 6 = 56$ bytes, whereas it would have used $32 \times 3 = 96$ bytes in the original ByteSlice.

As discussed in [5], [6], [16], [17], scan is a memory-bound operation. Thus, VBS offers at least two advantages: 1) It reduces memory bandwidth consumption between the memory and the CPU during scan, making it scale on many-core architectures more efficiently. 2) The CPU cache can contain data of more codes, which increases the cache hit rate.

C. Scan and Lookup

A PP-VBS scan takes in the VBS column, the predicate operation, and the literal code. It outputs a result bit vector, which indicates the matching rows.

1) Scan: Without loss of generality, we explain with the GREATER-THE-N predicate in the form of $v > c$. Here, $c$ is the code of the scalar literal in a predicate. Other predicate types follow suit. We use the notation $v_j$ to denote the $j$-th byte of code $v$. For example, in Figure 7, $v_1 = (11000001)_2$, $v_2 = (01111101)_2$. Suppose we have four column codes and a predicate literal $c$ as below:

\[
\tau_1 = (10000001)_2
\]
\[
\tau_2 = (11000000)_2
\]
\[
\tau_3 = (01110111 10101001)_2
\]
\[
\tau_4 = (10000001 10001001)_2
\]
\[
c = (10000001)_2
\]

The goal of the scan is to determine whether each $\tau$ passes or fails the predicate $\tau > c$. Recall that when comparing two codes of different length, we treat the short code as if it were padded zeros at the end. PP-VBS starts by comparing the first byte of $c$ with the first byte of all $\tau_i$:

$\tau_i^{[1]} > c^{[1]}$ and $\tau_i^{[1]} < c^{[1]}$ and $\tau_i^{[1]} = c^{[1]}$.

At this point, we can safely conclude that $\tau_2$ passes the predicate, while $\tau_3$ fails. There is no need to examine $\tau_2$'s and
τ_3”的第二字节。对于τ_1, 我们可以对它进行检查，因为它的字节长度为10000000 01010000，所以该字节有第二字节。我们已经知道对于τ_1来说，它没有第二字节。

要说明，考虑一个不同的存储单位。如果没有第二字节。这是因为，如在第III-A节中所述，其字节长度为（00000000 00000000）的字节不能为零。因此，

τ_1 = 10000000 01010000

In summary, scans on all above codes stop early after the first byte, even though the maximum code length is two. More formally, scan cost on PP-VBS can be bounded by the following lemma (proof in [18]):

**Lemma 1:** Let l(v) be the byte-length of code v. For all predicate types op ∈ {<, >, ≤, ≥, =, ≠}, the evaluation of predicate v_1 op v_2 conclusively stops after examining the m-th byte, where m ≤ min {l(v_1), l(v_2)}.

PP-VBS inherits the early stop capability of ByteSlice, even when both v and c are long. To illustrate, consider a different predicate literal c′ for the above example:

\[ c′ = (10000000 01010000)_2 \]

In this case, after comparing the first byte, we obtain:

\[ \tau_1^{[1]} > c^{[1]} \text{ and } \tau_2^{[1]} > c^{[1]} \text{ and } \tau_3^{[1]} < c′^{[1]} \text{ and } \tau_4^{[1]} > c′^{[1]} \]

which suffices to conclude that \( \tau_1, \tau_2, \tau_3, \tau_4 \) pass the predicate, whereas \( \tau_2 \) fails. There is no need to compare \( c′^{[2]} \) with any code.

To handle complex predicates that involve multiple columns, we follow ByteSlice [5] to pipeline the result bit vector of one predicate evaluation to another so as to increase the early stop probability of the subsequent evaluation.

2) Lookup: The idea of lookup is relatively straightforward. To construct a code, we retrieve the bytes from corresponding byte slices and then concatenate them in order. For example, in Figure 7, to obtain 2-byte \( v_2 \), bytes \( BS_{v_2}^{[2]} \) and \( BS_{v_2}^{[1]} \) will be retrieved. When the column is more skewed, the probability is higher that the length of a code retrieved is short so that fewer bytes will be retrieved from the memory. To the limit of the pages, please refer to our technical report [18] for more details on lookup operation.

IV. COLUMN-AYOUT ADVISOR

As our experiments will show, PE-VBP and ByteSlice will dominate scan/lookup on skewed and non-skewed data, respectively. Therefore, given a data column, ByteStore only has to make a decision to store it in ByteSlice or PP-VBS format. Since both ByteSlice and PP-VBS leverage byte as the storage unit, their lookup performances are equally good. Therefore, the decision mainly lies on which one yields better performance on scan.

There are a variety of factors that influence the scan performances of ByteSlice and PP-VBS. Data-related factors include the data distribution, the skewness, the domain size, and the value type (e.g., numeric or categorical). Query-related factors include the predicate type (e.g., >, =) and its selectivity (percentage of codes that satisfy the predicate).

Although modeling the relationship between the scan performance and the factors above is challenging, ByteStore does not need do so because the data columns are actually given. Therefore, the column-layout advisor of ByteStore goes for an experiment-driven approach [19] that chooses the best storage layout for each column based on running (scan) experiments on them. Specifically, given a data column, we first encode it twice: one using ByteSlice and one using PP-VBS; and then we generate and execute scan queries with different selectivities on top to get a profile. Finally, we choose the storage layout of a column based on its profile.

Figure 9 depicts the profiles of three real columns from the TAXI_TRIP [8] real dataset obtained from Google BigQuery.

![Fig. 9: Scan Profiles for 3 Real Data Columns from the TAXI_TRIP [8]](image)

For numeric columns, we use < c as the profiling predicate because from Section III-C we know that the scan implementations of other operators (e.g., =) are largely similar and thus their scan performance is also similar (our experiments also confirm this). For categorical columns, we use c as the profiling predicate. Each profile is obtained by generating and executing queries with 100 predicate literals from the column that span across the entire feasible selectivity spectrum. For example, Figure 9c shows that the most unselective value in the categorical column “Pickup Census Tract” would retrieve 9.9% of the column. All other values have lower selectivity than that.

After profiling, our column-layout advisor computes and picks the one with a smaller area under curve (AUC). For example, it is obvious that for the column “Seconds” in the TAXI_TRIP dataset (Figure 9a), the AUC of PP-VBS is smaller than ByteSlice (because that column is skewed). Therefore, the column layout advisor would retain that column encoded using PP-VBS and discard the one encoded using ByteSlice. For the column “Total” in the TAXI_TRIP dataset (Figure 9b), our column-layout advisor would also retain the one encoded using PP-VBS because it outperforms ByteSlice for a wide range of selectivities. Of course, if the (range of...
the) selectivity of a predicate is known, it is straightforward for our advisor to include that factor into account.

V. EXPERIMENTAL EVALUATION

We run our experiments on a rack server with a 2.1GHz 8-core Intel Xeon CPU E5-2620 v4, and 64GB DDR4 memory. Each core has 32KB L1i cache, 32KB L1d cache and 256KB L2 unified cache. All cores share a 20MB L3 cache. The CPU is based on Broadwell microarchitecture and supports AVX2 instruction set. We compare PP-VBS with Bit-Packed, PE-VBP and ByteSlice to show that PP-VBS is dominating the skewed columns. In the real data evaluation, we show that queries on ByteStore outperform any homogeneous storage engines on 6 real datasets. Unless stated otherwise, all experiments are run using one core.

A. Micro-Benchmark Evaluation

For the first experiment, we create a column with one billion numeric values. The column values are integer numbers in the range of \([0, 2^d]\). Following PE-VBP [7], we generate the column values from the Zipf distribution.

Figure 10(a) reports the scan and lookup cost of different layouts in terms of processor cycles spent per code(Cycles/code), when varying skew factor. The results are averages from queries with 100 different selectivities. In this experiment, following [7], we set the domain size as \(2^{d}\) (i.e., \(d = 12\)). It is reasonable since in real-world datasets, the number of distinct values of columns (aka domain size) is mainly in the range of \((2^{7}, 2^{16})\) [6], [10]. As shown in Figure 10(a), ByteSlice is dominant on scan operation when the skew factor is less than 0.5; PP-VBS starts to dominate when the skew factor increases. PP-VBS, ByteSlice and PE-VBP achieve way better scan performance than Bit-Packed layout because of early stop. In terms of lookup, PP-VBS and ByteSlice perform as well as Bit-Packed and outperform PE-VBP in all cases because they do not scatter the bits into so many different words as PE-VBP does. Lookup performance on PP-VBS improves mildly when the data is getting more skewed because averagely each block of a column contains more short-length codes under higher skew. It decreases the number of bytes read from the memory, which benefits the memory-bound lookup operation.

Figure 10(b) reports the scan and lookup performance of different layouts when varying the domain size (i.e., \(2^1 \sim 2^{24}\)). We fix the skew factor as 1.0 in this experiment. On skewed data, PP-VBS outperforms Bit-Packed, PE-VBP and ByteSlice under a wide range of domain size in terms of scan operation. The cost of scan increases with the domain size because generally more bits of a code are retrieved from the memory for a predicate. Lookup time also increases with the domain size because the average code length increases with domain size.

Figure 10(c) reports the scan and lookup performance of different layouts when varying selectivities. We fix skew factor as 1.0 and domain size as \(2^{12}\) (i.e., \(d = 12\)) in this experiment. PP-VBS dominates all the other storage layouts on scan for all selectivities. Both PP-VBS and ByteSlice also have as excellent performance as Bit-Packed and outperform PE-VBP on lookup operation under all selectivities.

In addition, we evaluated our experiments of multiple threads. Figure 11 reports the scan performance of different layouts when varying skew factor. In this experiment, we fixed the domain size as \(2^{12}\) (i.e., \(d = 12\)) and the number of threads as 8, which means all CPU cores are used. It is clear to see that we can draw similar conclusion as above when using multi-threading.

We also have carried out experiments of (i) varying the cardinalities of the columns, (ii) using data generated by Gaussian distribution of different variances instead of using Zipf distribution, (iii) using other operators (e.g., \(\ge\)), and (iv) using AVX-512 instead of AVX2. Since those experiments draw similar conclusions as the above, we do not present them here for space interest.

Memory Usage: Figure 12 reports the average number...
of bits used per code under different storage layouts. In Figure 12(a), we fixed the domain size as $2^{12}$ and varied the skew factor. Not surprisingly, the average number of bits per code under PP-VBS reduces as the skew factor increases and PP-VBS starts to dominate other storage layouts when the skew factor is larger than 0.8. Notably, we also counted the memory usage of the bitmasks $M$. The memory usage of PE-VBP increases along with the skew factor since the maximum code length increases with the skew factor. For ByteSlice, even though the code length is 12-bit, one code must take up 16 bits under ByteSlice because of padding zeros, which will waste memory. However, we focus on scan and lookup performance in this paper and select ByteSlice as a candidate since it dominates scan and lookup performance on uniform to lightly skewed data columns. As opposite to Figure 12(a), we fixed the skew factor as 1 and varied the domain factor in Figure 12(b). PP-VBS uses the least memory when columns are skewed under varying domain factor.

B. Real Data Evaluation

This set of experiments aims to evaluate ByteStore as a whole and compare it with homogenous storage engines that use only Bit-Packed, use only ByteSlice or only PE-VBP.

The experiments are done using 6 real datasets downloaded from Google BigQuery in 2019 May [25]. Table I shows their details as well as the results of dataset ingestion by ByteStore’s column layout advisor. It clearly shows that different columns in a dataset require different storage layouts. The column layout advisor does not use much time to do the profiling and layout selection. The offline data ingestion time is mainly spent on encoding the columns, but half of that time is indispensable as the column has to be encoded in one of the two storage layouts anyway. To focus only on scans and lookups, we follow [4] to materialize the joins and execute the selection-projection components of the queries. Same as [5], we discard the queries that have no selection clause and queries that involve string similarity comparison LIKE.

Figure 13 compares ByteStore with using ByteSlice only, using PE-VBP only, and using Bit-Packed only across different queries and different datasets. In Figure 13(a), we report the execution time breakdown of all queries. All the queries selected (T1~P1) involve PP-VBS columns and ByteSlice columns in ByteStore. The run time of each query is dissected into scan cost and lookup cost. The reported numbers have been normalized on a per tuple basis. We could see there are both scan-dominant (e.g., E1 and P1) and lookup-dominant (e.g., F1) queries. We can see that ByteStore outperforms all homogeneous storage schemes. Figure 13(b) reports the query speed-up over PE-VBP only. Overall, ByteStore brings up to $4.0 \times$, $1.7 \times$ and $5.2 \times$ speedup to query performance when comparing with using PE-VBP only, ByteSlice only and Bit-Packed only respectively.

VI. RELATED WORK

There have been several hybrid data engines developed for HTAP workloads. Hybrid storage engines mainly focus on mixing the row and column representations [26]–[30].

The experiment-driven approach [19] has been used to tune database systems [31], batch systems [32], and machine learning systems [33]. In this paper, we also use the experiment-driven approach to design our column-layout-advisor. The advantage of experiment-driven approach is that its results are highly accurate at the cost of relatively longer tuning time. Nonetheless, tuning is an offline process and the tuning time is not a crucial factor.

Encoding techniques have been extensively used for main memory analytical databases in both the research community [6], [34], [35], and in the industry, e.g., SAP HANA [1]. Works listed above all use the fixed length encoding and do not leverage column skew. IBM Blink [36] and its commercial successor IBM DB2 BLU employ a proprietary encoding technique called frequency partitioning. Based on the frequency of data, a column is divided into multiple partitions, each of which uses an independent fixed-length encoding. The code lengths of different column partitions are different. Thus, that technique can be viewed as a hybrid between the fixed-length encoding and variable-length encoding. By contrast, PPE is a pure variable length encoding scheme and it should work with a distinct storage layout VBS in tandem.

Lightweight indexes are techniques that skip data processing by using summary statistics over the base column. Such techniques include Bitmap Index [37], Zone Maps [38], Column Imprints [16], Column Sketches [17] and BinDex [39]. These lightweight indexing techniques are complementary with storage layouts and can be used together.
VII. CONCLUSION

Choosing the optimal layout for individual columns in scan-based OLAP systems is non-trivial because it must balance between scan and lookup performance and account for the column data characteristics. In this paper, we first presented a new layout, PP-VBS, that achieves both fast scan and fast lookup on skewed data. We then described ByteStore, a hybrid column store using an experiment-driven approach to select the best column layout for each individual column. Experiments on real and synthetic datasets and workloads show that our hybrid column store significantly outperforms previous works in terms of end-to-end query performance.

VIII. ACKNOWLEDGEMENTS

This work is partially supported by Hong Kong General Research Fund (14200817), Hong Kong AoE/P-404/18, Innovation and Technology Fund (ITS/310/18, ITP/047/19LP) and Centre for Perceptual and Interactive Intelligence (CPI) Limited under the Innovation and Technology Fund.

REFERENCES

[1] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, “The SAP HANA Database—An Architecture Overview.” IEEE Data Eng. Bull., 2012.
[2] P. A. Boncz, M. Zukowski, and N. N., “MonetDB/X100: Hyper-Pipelining Query Execution,” in CIDR, 2005.
[3] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, “Column-oriented database systems,” PVLDB, 2009.
[4] Y. Li and J. M. Patel, “WideTable: An accelerator for analytical data processing,” PVLDB, 2014.
[5] Z. Feng, E. Lo, B. Kao, and W. Xu, “Byteslice: Pushing the envelop of main memory data processing with a new storage layout,” in SIGMOD, 2015.
[6] Y. Li and J. M. Patel, “Bitweaving: Fast scans for main memory data processing,” in SIGMOD, 2013.
[7] J. M. P. Yinan Li, Craig Chasseur, “A padded encoding scheme to accelerate scans by leveraging skew,” in SIGMOD, 2015.
[8] City of Chicago, “Chicago Taxi Trips,” https://console.cloud.google.com/marketplace/details/city-of-chicago-public-data/chicago-taxi-trips.
[9] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, “Hyrise: A main memory hybrid storage engine,” PVLDB, 2010.
[10] A. Kemper and T. Neumann, “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots,” in ICDE, 2012.
[11] R. Ramamurthy, D. J. DeWitt, and Q. Su, “A case for fractured mirrors,” VLDB J., 2003.
[12] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund, “High-performance transaction processing in SAP HANA,” IEEE Data Eng. Bull., 2013.
[13] H. Lang, T. Muhlbaier, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper, “Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation,” in SIGMOD, F. Özcan, G. Koutrika, and S. Madden, Eds., 2016.
[14] S. Duan, V. Thummalala, and S. Babu, “Tuning database configuration parameters with ituned,” PVLDB, 2009.
[15] S. Babu, “Towards automatic optimization of mapreduce programs,” in SoCC, 2010.
[16] C. Liu, P. Zhang, B. Tang, H. Shen, L. Zhu, Z. Lai, and E. Lo, “Towards self-tuning parameter servers,” CoRR, 2018.
[17] D. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in SIGMOD, 2006.
[18] W. Fang, B. He, and Q. Luo, “Database compression on graphics processors,” PVLDB, 2010.
[19] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Ideeos, M.-S. Kim, O. Koeth, J.-G. Lee et al., “Business analytics in (a) blink,” IEEE Data Eng. Bull., 2012.
[20] C. Y. Chan and Y. E. Ioannidis, “Bitmap index design and evaluation,” in SIGMOD, 1998.
[21] G. Moerkotte, “Small materialized aggregates: A light weight index structure for data warehousing,” in VLDB, 1998.
[22] L. Li, K. Zhang, J. Guo, W. He, Z. He, Y. Jing, W. Han, and X. S. Wang, “Bindex: A two-layered index for fast and robust scans,” in SIGMOD, 2020.

| Dataset | # of Rows | # of Columns (ByteSlice/PP-VBS) | Encoding Time | Profiling & Layout Selection Time | # of Public Queries |
|---------|-----------|---------------------------------|---------------|----------------------------------|---------------------|
| TAXI [8] | 185,666k  | 23 (6:17)                       | 180.21 min    | 12.25 min                        | 3                   |
| HEALTH [20] | 2,784k    | 6 (4:2)                         | 1.12 min      | 0.15 min                         | 1                   |
| NYC [21] | 146,113k  | 19 (13:6)                       | 117.16 min    | 8.03 min                         | 3                   |
| EDUCATION [22] | 5,082k | 6 (3:3)                       | 1.98 min      | 0.21 min                         | 1                   |
| FEC [23] | 20,557k   | 35 (20:15)                      | 30.59 min     | 2.58 min                         | 2                   |
| NPPES [24] | 5,943k | 480 (353:127) | 123.88 min | 15.92 min | 2 |