ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout

Ziqiang Feng† Eric Lo† Ben Kao§ Wenjian Xu†
† Department of Computing, The Hong Kong Polytechnic University
§ Department of Computer Science, The University of Hong Kong
† {cszqfeng, ericlo, cswxu}@comp.polyu.edu.hk
§kao@cs.hku.hk

ABSTRACT

Scan and lookup are two core operations in main memory column stores. A scan operation scans a column and returns a result bit vector that indicates which records satisfy a filter. Once a column scan is completed, the result bit vector is converted into a list of record numbers, which is then used to look up values from other columns of interest for a query. Recently there are several in-memory data layout proposals that aim to improve the performance of in-memory data processing. However, these solutions all stand at either end of a trade-off — each is either good in lookup performance or good in scan performance, but not both. In this paper we present ByteSlice, a new main memory storage layout that supports both highly efficient scans and lookups. ByteSlice is a byte-level columnar layout that fully leverages SIMD data-parallelism. Micro-benchmark experiments show that ByteSlice achieves a data scan speed at less than 0.5 processor cycle per column value — a new limit of main memory data scan, without sacrificing lookup performance. Our experiments on TPC-H data and real data show that ByteSlice offers significant performance improvement over all state-of-the-art approaches.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management — *systems*

Keywords

main memory; column store; storage layout; SIMD; OLAP

1. INTRODUCTION

In main-memory column stores like SAP HANA [15], MonetDB [20], Vectorwise [43], and Oracle Exalytics [1], data is memory-resident, queries are read-mostly, and the performance goal is to support real-time analytic. When the performance is not disk-bound, one key design goal of data processing algorithms is to process data at (or near) the speed of CPU by judiciously utilizing all the available parallelisms in each processing unit.

Data-level parallelism is one strong level of parallelism supported by modern processors. Such parallelism is supported by the SIMD (single instruction multiple data) instruction set, whose instructions can process multiple data elements in parallel. SIMD instruction set (e.g., SSE 128-bits, AVX2 256-bits) was originally designed for accelerating multimedia applications (e.g., to increase the video brightness when playing a video), but work including [42] pioneered the use of SIMD instructions to accelerate core database operations such as filter scans and joins. After about a decade of research, SIMD parallelism has now become the basic requirement of building many new data processing systems [24, 25, 15]; operations in Google's Supersonic library [17], an open-source library for building data processing system, are all SIMD enabled.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2747642.

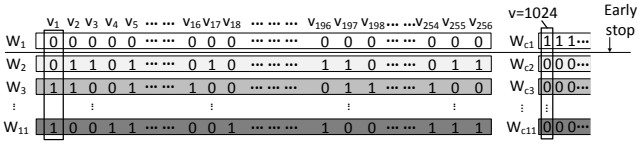


Figure 1: 11-bit column values stored in memory using VBP

SIMD instructions can execute one multi-operand operation per cycle, where each operand is b bits ($b = \{8, 16, 32, 64\}$ in AVX2). To enable SIMD parallel processing in column-oriented database, (encoded) column values need to be aligned and loaded into SIMD registers before processing. Early work in SIMD data processing [42] was not fastidious about the storage layout in the main memory and simply stored columns values using standard data type (e.g., 32-bit integer). So, in the context of today's AVX2 whose SIMD registers are 256 bits, a SIMD register is used as eight 32-bit banks (i.e., $b = 32$) and eight 32-bit values are directly loaded from memory into the register for processing. Subsequent proposals [40, 39] focused more on the memory bandwidth consumption and proposed to store the column values in memory using a tightly *bit-packed* layout, ignoring any byte boundaries. For example, to store a column of 11 bits in memory, the first value is put in the 1-st to 11-th bits whereas the second value is put in the 12-th to 22-nd bits and so on. Such bit-packed layout incurs overhead to unpack the data before processing. In the example above, several SIMD instructions have to be spent to align eight 11-bit values with the eight 32-bit banks of a register.[1] After alignment, data processing operations like filter scan can then be carried out using a series of SIMD instructions. In the example above, although 8-way data-level parallelism is achieved in data processing, many cycles are actually wasted during unpacking and alignment. Furthermore, as 0's are used to pad up with the SIMD soft boundaries, for the ex-

[1]As an 11-bit value may span across three bytes under the bit-packed format, a bank width $b = 32$ is needed because $b = 32 > 3 \times 8$). More details will be given in Section 2.1.

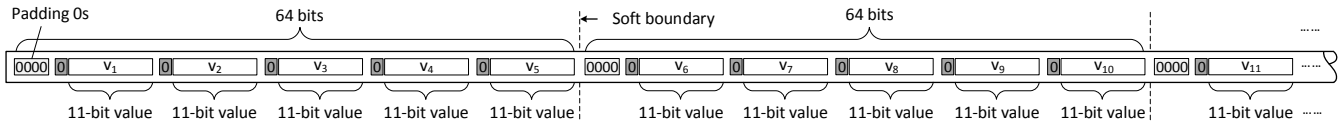


Figure 2: 11-bit column values stored in memory using HBP

ample above, any data processing operation is wasting (32-11) $\times$ 8 = 168 bits of computation power per cycle.

Recently, Li and Patel [31] observed that by changing the main memory storage layout, higher parallelism and thus better scan performance could be achieved. Inspired by bit-slice indexing [33], their first layout, Vertical Bit-Parallel (VBP), is a bit-level columnar that systematically packs values in chunks of memory called *words*. Figure 1 shows an example of VBP for an 11-bit data column given that a SIMD register is 256 bits. Specifically, the i-th bit of each k-bit value v is stored in the i-th 256-bit memory word. In the example, 256 values are packed into $k = 11$ memory words: W_1, W_2, ..., W_{11}. Under such layout, after a memory word is brought into a SIMD register, predicate evaluation in a scan operation is carried out as a series of logical computations on those "words of bits". SIMD scans on VBP formatted data can achieve significant speedup because (1) a memory word and a register are of the same length (256 bits) so register bits are fully utilized in processing (no register bits are wasted to padding), and (2) scans can **early stop**. To illustrate, consider the evaluation of the predicate "$v = 1024$" (see Figure 1). Under VBP, the bits of the constant to be compared against (i.e., $10000000000_2 = 1024_{10}$) is first transposed and vertically packed into k words (see words W_{c1}, W_{c2}, ... W_{c11} in Figure 1). Then the scan carries out predicate evaluation through k iterations. In the first iteration, it intersects word W_1 with word W_{c1}. The result of the intersection, which is a bit vector, indicates the data values whose first bits match that of 1024_{10}. In the example, after the first iteration, the scan concludes that none of the 256 values $v_1 \dots v_{256}$ satisfies the predicate. In this case, the scan stops early and skips the remaining words W_2 to W_{11}. Reconstructing/looking-up a value under the VBP layout is expensive though. That is because the bits of a value are spread across k words. Retrieving all these bits incurs many CPU instructions and possibly many cache misses. Poor lookup performance hurts the performance of operations that require the values in their plain form (e.g., joins and aggregations), resulting in poor query performance.

The Horizontal Bit-Parallel (HBP) storage layout is the other in-memory storage layout proposed in [31]. HBP supports value lookups efficiently. In HBP, all bits of a column value are packed into the same memory word, providing good lookup performance. Figure 2 shows an example of a 256-bit memory word formatted in HBP. In this example, we consider a SIMD register to operate in banks of 64 bits (i.e., $b = 64$). Hence, under HBP, a 256-bit memory word is partitioned into chunks of 64 bits with soft boundaries defined at every 64 bits of the word. That way, each 64-bit chunk can hold five 11-bit values. During processing, each 64-bit chunk (containing 5 values) is loaded into a 64-bit bank of a SIMD register. Note that the whole 256-bit memory word contains 4 chunks and thus 20 11-bit values in total, which is much more than just eight 11-bit values offered by the bit-packed format we discussed earlier [42, 40, 39]. SIMD scans on HBP formatted data can leverage two levels of parallelism: *intra-bank parallelism* and *inter-bank parallelism*. Intra-bank parallelism manages evaluation on values loaded within a bank of the register (e.g., evaluating a predicate "$v = 4$" among v_1 to v_5 in Figure 2). Inter-bank parallelism is offered by the SIMD instructions that process values in multiple banks in parallel. In our example, HBP brings 5-way intra-bank parallelism and SIMD brings 4-way inter-bank parallelism, achieving a 20-way parallelism in scans. Although efficient in lookup, HBP offers no early stopping opportunity for scans because all bits of a column value are stored in the same word. Hence, all bits of a value are always processed at the same time.

In summary, scans on VBP has excellent performance because of early stopping but lookups are poor. In contrast, lookups on HBP are excellent but scans are way poorer than on VBP. Scans on HBP and VBP are generally faster than scans on bit-packed formatted data. In this paper we present ByteSlice, a simple yet powerful storage layout with which scans perform better than on VBP and lookups perform as well as on HBP. ByteSlice has two key properties: (1) bytes of a k-bit value are spread across $\lceil k/8 \rceil$ words; (2) an S-bit memory word contains bytes from $S/8$ different values. Compared with VBP that distributes the *bits* of a value to k words, ByteSlice distributes the *bytes* of a value to only $\lceil k/8 \rceil$ words, thereby reducing the reconstruction/lookup efforts by a factor of 8. Such reduction turns out to be significant enough to make a lookup overlap with other operations in the instruction pipeline, resulting in lookup performance that is as good as on HBP. Compared with HBP that has no early stopping opportunities, ByteSlice enjoys early stopping like VBP. Recall that scans on VBP involve examining values *bit-by-bit* and that lower-order bits can be skipped when appropriate. Scans on ByteSlice examine values *byte-by-byte*, and so lower-order bytes can be skipped in a similar fashion. As an interesting observation, our analysis shows that ByteSlice provides even more effective early stopping than VBP. In short, ByteSlice combines the advantages of both VBP and HBP without inheriting their disadvantages. ByteSlice thus gives main memory analytic databases a single best storage layout, which frees the database administrators or the tuning advisors from choosing between VBP and HBP.

The contribution of this paper is ByteSlice, a new in-memory storage layout for column stores that supports both fast scans and lookups. The corresponding framework to support scans and lookups on ByteSlice formatted data is also included in this paper. Experimental results show that running TPC-H queries on ByteSlice can be up to **10X**, **5.5X**, and **3X** faster than on bit-packed, VBP, and HBP formatted data, respectively.

The remainder of this paper is organized as follows: Section 2 contains background information. Section 3 presents the ByteSlice storage layout along with the framework to support scans and lookups. Section 4 gives our experimental results. Section 5 discusses related works. Section 6 concludes the paper with a future work discussion.

2. BACKGROUND AND PRELIMINARY

SIMD instructions interact with S-bit SIMD registers as a *vector of banks*. A bank is a continuous section of b bits. In AVX2, $S = 256$ and b is 8, 16, 32, or 64. We adopt these values in this paper but remark that our techniques can be straightforwardly extended to other models (e.g., 512-bit AVX-512 [23] and Larrabee [35]). The choice of b, the *bank width*, is on per instruction basis.

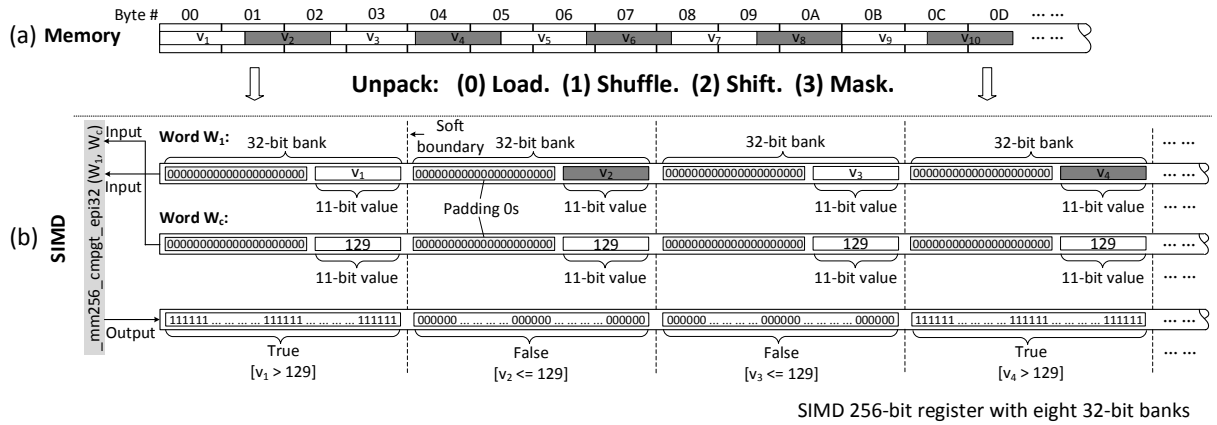


Figure 3: (a) 11-bit column values stored under bit-packed memory layout (b) Scans on bit-packed data with predicate $v > 129$

A SIMD instruction carries out the same operation on the vector of banks simultaneously. For example, the `_mm256_add_epi32()` instruction[2] performs an 8-way addition between two SIMD registers, which adds eight pairs of 32-bit integers simultaneously. Similarly, the `_mm256_add_epi16()` instruction performs 16-way addition between two SIMD registers, which adds sixteen pairs of 16-bit short integer simultaneously. The degree of such *data-level* parallelism is S/b.

In modern main memory analytic databases, data is often stored in a compressed form [6, 14, 15, 28]. ByteSlice is applicable to common compression schemes such as null suppression, prefix suppression, frame of reference and dictionary encoding [6, 14, 15, 28]. In these schemes, native column values are compressed as fixed-length order-preserving *codes*. All data types including numeric and strings are encoded as unsigned integer codes. For example, strings are encoded by building a sorted dictionary of all strings in that column [7, 28]. Floating point numbers with limited precision can be scaled to integers by multiplication with a certain factor [14]. Consequently, range-based column scans can be directly evaluated on such codes. From now on, we use the terms **code** and **value** interchangeably. 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.

In this paper we focus on the *scan* and *lookup* operations in main-memory column stores. A (column-scalar) scan takes as input a list of n k-bit codes and a predicate with a range-based comparison, e.g., $=, \neq, <, >, \leq, \geq$, `BETWEEN`, on a single column. Constants in the predicate are in the same domain of the compressed codes. A column-scalar scan finds all matching codes that satisfy a predicate, and outputs an n-bit vector, called the *result bit vector*, to indicate the matching codes. Conjunctions and disjunctions of predicates can be implemented as logical AND and OR operations on these result bit vectors. NULL values and three-valued boolean logic can be handled using the techniques proposed in [33].

Once the column-scalar scans are completed, the result bit vector is converted to a list of record numbers, which is then used to retrieve codes/values from other columns of interest for the query. A *lookup* refers to retrieving a code in a column given a record number. Depending on the storage layout, a code may have to be reconstructed from multiple memory regions during a lookup. In existing main-memory column store implementations, the results (retrieved codes) of lookups are inserted into an array of a standard data type (e.g,. `int32[]`) [26, 8, 9, 5, 4]. This array serves as an intermediate result whose content is consumed by operations like joins, aggregations and sorts. Since these operations are not directly processing data straight from the base columns, their performances are independent of the storage layout of the base columns [18]. For this reason, we focus on the basic operations *scan* and *lookup* in this paper. Nonetheless, we envision that ByteSlice has the potential of being a representation of *intermediate query results* in addition to just being a storage format for the base column values. In that case, operations like joins and sorts could potentially benefit from reading input formatted in ByteSlice. Section 6 elaborates this future work idea.

[2]Technically, it is a C functions supported by SIMD instructions. We use the C function name in place of the SIMD instructions for simplicity.

2.1 Bit-Packed (BP) Layout

The Bit-Packed layout [40, 39] aims to minimize the memory bandwidth usage when processing data. Figure 3a shows an example with 11-bit column codes. The codes are tightly packed together in the memory, ignoring any byte boundaries. In this layout, a byte may contain bits from multiple codes (e.g., Byte# 02 contains bits from both v_2 and v_3) and a code may span across multiple bytes (e.g., v_3 spans across Byte# 02 to Byte# 04).

Scan Scans on bit-packed data requires unpacking the tightly packed data into SIMD registers. In Figure 3(a), as a code may initially span 3 bytes (e.g., v_3), a bank width of $b = 32$ has to be used. Under $b = 32$, scans can be run in 8-way parallelism and thus 8 codes are loaded to a SIMD register each time. Under the bit-packed layout, that means 8×11 bits of data (i.e., $v_1 \sim v_8$) are loaded from the memory to the register. To align the 8 values into the eight 32-bit banks, three instructions are carried out: (1) Shuffle: an SIMD byte-level shuffle instruction is used to copy the bytes of each code to their destination bank. In Figure 3(a), Bytes# $00 \sim 01$ are shuffled to the first bank, as they contain bits from v_1, Bytes# $01 \sim 02$ are shuffled to the second bank, and Bytes# $02 \sim 04$ are shuffled to the third bank, so on. (2) Shift: a shift instruction is used to align the codes to their (right) bank boundaries. (3) Mask: carry out a bitwise AND instruction with a mask to clear the leading unwanted bits (of another code). After unpacking, data in the SIMD register (e.g., W_1 in Figure 3b) is ready to be processed by any database operation. Figure 3b illustrates how a scan with the predicate "$v > 129$" is evaluated against the unpacked data using AVX2's 8-way greater-than comparison instruction `_mm256_cmpgt_epi32()`. The result of the instruction is a vector of eight boolean masks. After that, the scan starts another iteration to unpack and compare the next 8 codes with W_c. From

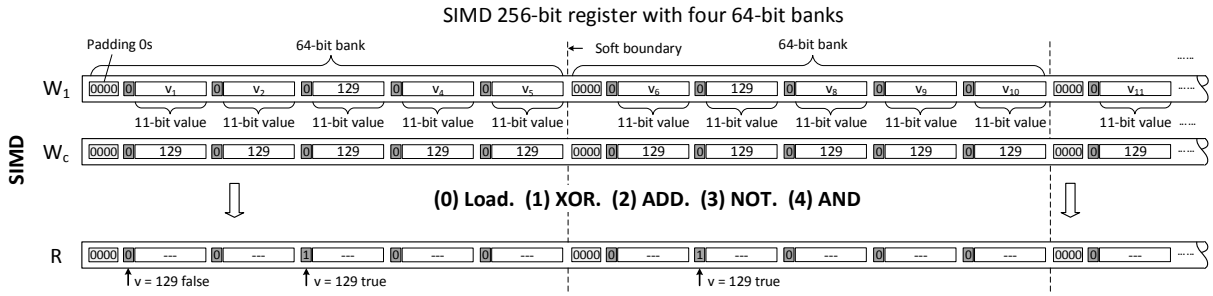


Figure 4: Evaluating $v = 129$ on 11-bit column values stored in HBP format.

the above, we see that the unpacking step consumes a number of cycles, hurting the scan performance.

Lookup To look up a code v_i in bit-packed data, one has to first compute (1) at which byte v_i starts and (2) the offset of v_i's starting bit within that byte. For example, to look up v_3 from memory in Figure 3a, we compute $\lfloor k(i-1)/8 \rfloor = \lfloor (11 \times (3-1))/8 \rfloor = 2$ and $k(i-1) \mod 8 = (11 \times (3-1)) \mod 8 = 6$ to obtain the byte number and offset, respectively. Next, three bytes starting from Byte# 02, i.e., Bytes# $02 \sim 04$, are fetched from memory. These bytes are stitched together using shifting and bitwise OR operations. Finally, the stitched value is shifted by 6 bits (the offset) and a mask is applied to retain the 11 bits of v_3. As a code may span multiple bytes under the bit-packed format, a lookup may incur multiple cache misses, particularly when those bytes span across multiple cache lines.

2.2 Vertical Bit-Parallel (VBP) Layout

The VBP storage layout vertically distributes the bits of a k-bit code across k chunks of memory called *words*. Specifically, the column of codes is broken down into fixed-length *segments*, each of which contains S codes, where S is the width of a SIMD register. The S k-bit codes in a segment are then transposed into k S-bit words, denoted as W_1, W_2, ..., W_k, such that the j-th bit in W_i equals to the i-th bit in the original code v_j.

Scan Scans on VBP formatted data are carried out segment by segment. Within a segment, a single predicate is evaluated through k iterations. As VBP allows processing n k-bit codes in S-way parallelism, its worst-case scan complexity is $O(\frac{nk}{S})$ instructions. Practically, VBP scans seldom hit that bound because of early stopping, which was illustrated in Figure 1. When early stopping occurs, the scan will proceed to the next segment. To reduce the condition-checking overhead as well as branch miss-prediction penalty, the early stopping condition is tested for every τ iterations. It has been empirically determined that τ should be set as 4 [31]. Scans on VBP has no overflow problem because the predicate evaluations only use bitwise operations AND, OR, NOT, and XOR but not addition or multiplication.

Let us illustrate VBP's early stopping with a back-of-the-envelope calculation. Consider the predicate "$v = c$". Assume the S codes of a segment are independent and that the codes and the comparison constant c are random numbers uniformly distributed in the domain of the codes. The probability of a code v matching the constant c at any particular bit position is 1/2. Note that we can conclude that the predicate evaluates to false after scanning the most significant t bits of v if any one of those t bits of v does not match the corresponding bit of c. The probability of this event is $1 - (1/2)^t$. To early stop a segment of S codes after examining the codes' most significant t bits, we need the above condition to hold for all S codes. Hence, the probability of early stop processing a segment after t bits is [31]:

$$P_{VBP}(t) = \left(1 - (\frac{1}{2})^t\right)^S \quad (1)$$

From Equation 1, we derive that VBP needs to scan, on average, 10.79 bits of each code before the processing of a segment can be early-stopped. As we will see later, we can significantly improve the early-stop probability with ByteSlice and thus lower the number of bits per code read before early stop to 8.94. With registers in future generations of SIMD architecture (e.g., 512-bit AVX-512 [23] and Larrabee [35]) having larger register width (larger S), the early-stop probability will become smaller (see Equation 1) and thus it will become harder for VBP to early stop. In this scenario, we expect ByteSlice's advantage over VBP to be even more pronounced.

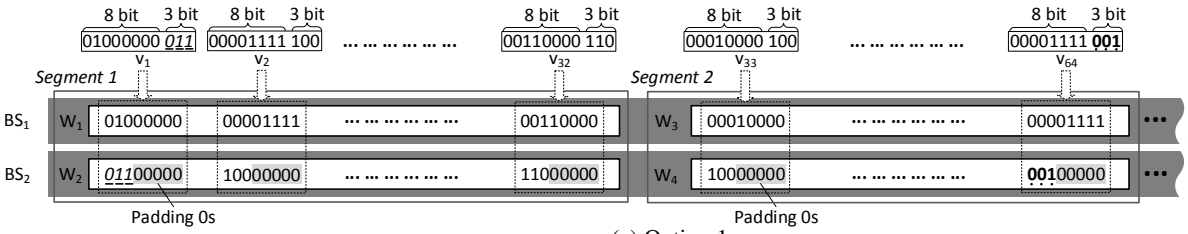
Lookup Lookups under VBP are expensive because retrieving a value requires accessing k different memory words. For example, in Figure 1, to look up v_5, we have to extract the 5-th bit of each word W_1, ..., W_{11} with a mask, shift the bit to the correct position, and merge it with the running output using a bitwise OR operation. The number of instructions involved increases linearly with k. Moreover, since the memory words can be located in different cache lines, the number of cache misses expected during a lookup also increases with k.

2.3 Horizontal Bit-Parallel (HBP) Layout

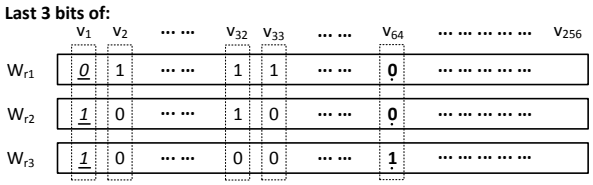
The HBP storage layout horizontally packs codes from a column into S-bit memory words. The layout is designed so that the bits of a word can be loaded directly into an S-bit register without unpacking. To maximize the number of codes that can be stored in a memory word under the constraint of bank boundaries (Figure 2), we always use the largest possible register bank width (b) e.g., in AVX2, b is set to 64.

Each k-bit code v_i is stored with an additional bit, which is a delimiter between adjacent codes of the same bank. The delimiter bit is prepended to the code and is dedicated for handling overflow and producing the result bit vector. A bank can hold $\lfloor \frac{b}{k+1} \rfloor$ values. If b is not a multiple of $k+1$, 0's are left padded up to the bank boundary.

Scan Figure 4 shows an example of evaluating a predicate $v = 129$ on HBP formatted data W_1. Before the scan, the constant in the predicate (i.e., 129) is first repeatedly aligned and packed into a SIMD register W_c in HBP format. Then, a sequence of arithmetic and logic operations are carried out to evaluate the predicate. That generates a result bit vector R, where the evaluation results are stored in the delimiter bits. Scans on HBP formatted layout have no early stopping because all bits of a code are in the same word/register. The scan complexity is $\Theta(\frac{nb}{S\lfloor b/(k+1) \rfloor})$.



(a) Option 1



(b) Option 2

Figure 5: 11-bit column codes stored under ByteSlice memory layout. (a) Option 1: Storing tailing bits as bytes (b) Option 2: Storing tailing bits using VBP

Lookup Lookup in HBP formatted data is similar to that in bit-packed formatted data. We have to compute (1) which bank in a memory word a code v_i is located and (2) the offset of v_i in that bank. For example, to look up v_9 in Figure 2, we first determine that v_9 is located in the 2nd bank with a 12-bit offset from the right boundary. Shifting and masking instructions are then executed to retrieve the 11-bit value. As each code lies in one memory word, a lookup under HBP incurs at most one cache miss only.

3. BYTESLICE (BS) LAYOUT

The ByteSlice storage layout regards S contiguous bits in memory as one word. It also views a word or a SIMD register as $S/8$ 8-bit banks. ByteSlice vertically distributes the *bytes* of a k-bit code across the same bank of $\lceil k/8 \rceil$ memory words. In other words, an S-bit memory word contains bytes from $S/8$ different codes. These $S/8$ codes form a *segment*. The magic number 8 comes from the smallest bank width of SIMD multi-operand instructions. A bank width of 8 bits implies the highest degree of SIMD parallelism, i.e., $S/8$-way (e.g., $256/8 = 32$), is exploited. The choice also has the advantage of simple implementation because bytes are directly addressable.[3] Hence, bit shifting and masking operations are avoided.

Figure 5a illustrates how a segment of thirty-two 11-bit codes ($v_1 \sim v_{32}$) are formatted into two memory words W_1 and W_2 under the ByteSlice layout. In the example, the first bytes of $v_1 \sim v_{32}$ are packed into the banks of W_1. There are two options to deal with the remaining 3 bits. Option 1 is to pad 0's at the end and pack them into the banks of W_2 in a way similar to W_1 (see Figure 5a). Option 2 is to pack those remaining bits in VBP format, i.e., use three memory words W_{r1}, W_{r2}, W_{r3} to store the remaining 3 bits of each code, with the 1st remaining bit goes to W_{r1}, the 2nd remaining bit goes to W_{r2}, and the last remaining bit goes to W_{r3} (see Figure 5b)[4]. We choose Option 1 for reasons that will become clear in Section 3.1.1. Continuing with our example, the next segment of 32 values ($v_{33} \sim v_{64}$) are formatted in the same way into words W_3 and W_4. Words that contain the i-th bytes of values are stored in a contiguous memory region, which we name as ByteSlice. In Figure 5a, W_1 and W_3 are stored in ByteSlice BS_1 whereas W_2 and W_4 are stored in ByteSlice BS_2. As we will explain shortly, ByteSlice supports early stopping. It is thus likely that after word W_1 (which stores the first bytes of values in Segment 1) is processed, word W_2 can be skipped and the scan operation proceeds to processing W_3. With a machine having a cache line size of 512 bits (i.e., $2S$), by putting W_1 and W_3 into neighboring memory locations, the two words can be found in the same cache line. This arrangement also avoids bringing W_2/W_4 into the cache when early stopping can happen.

[3]In this regard, a bank width of 16 bits is also an option. Appendix A explains our choice of 8 bits bank width in more detail.

[4]Using HBP to store those tailing bits is not an option because scan algorithms on HBP cannot work on part of the codes. The two options also apply to columns whose codes use less than 8 bits (i.e., $k \leq 7$). In this case, ByteSlice is degenerated into VBP if Option 2 is used.

3.1 Scan

We now describe how to evaluate comparison predicates ($<, >, \leq, \geq, =, \neq$, `BETWEEN`) using SIMD instructions under ByteSlice. The output of such filter scans is a result bit vector R, which indicates which codes in the column satisfy the predicate. Each scan operation processes one segment of codes at a time. Without loss of generality, we assume the code width k is a multiple of full bytes. If not, we pad 0's at the end of both the codes and the comparison constant. The comparison result should remain the same, e.g., $(10000)_2 > (01000)_2 \leftrightarrow (10000\ \underline{000})_2 > (01000\ \underline{000})_2$. We use the notation $v^{[i]}$ to denote the i-th byte of v. For example, in Figure 5, $v_1^{[1]} = (01000000)_2, v_1^{[2]} = (011\ \underline{00000})_2$.

We begin with the discussion of evaluating the LESS-THAN predicate ($v < c$) in ByteSlice. To illustrate, consider the following two 11-bit values v_1 and v_2 and a comparison constant c (with padding 0's underlined):

$$v_1 = (01000000\ 011\underline{00000})_2$$
$$v_2 = (00001111\ 100\underline{00000})_2$$
$$c = (00010000\ 100\underline{00000})_2$$

The evaluation is carried out in $\lceil k/8 \rceil$ iterations, with the j-th iteration comparing the j-th most significant bytes of v's and c. For the example, after the j-th = 1st iteration, we know:

$$v_1^{[1]} > c^{[1]} \quad \text{and} \quad v_2^{[1]} < c^{[1]},$$

which allows us to conclude that (1) $v_1 > c$ and (2) $v_2 < c$. So, v_1 does not satisfy the predicate but v_2 does. In this case, we can early stop and skip the next iteration. In general, when evaluating the predicate "v `op` c" (`op` $= \{<, >, \leq, \geq, =, \neq\}$), we can early stop after the j-th iteration if $v_i^{[j]} \neq c^{[j]}$ for all v_i's.

Algorithm 1 delineates the pseudo-code of evaluating the LESS-THAN predicate under ByteSlice. The algorithm takes a predicate "$v < c$" as input and outputs a result bit vector R whose i-th bit is 1 if v_i satisfies the predicate or 0 otherwise. Initially, the bytes of the constant c are broadcast to $\lceil \frac{k}{8} \rceil$ SIMD words (Lines 1–3). Then, the algorithm scans the column codes segment-wise, with each segment containing $S/8$ (i.e., 32, when $S = 256$) codes (Lines 4–18).

For each segment, the algorithm first prepares two S-bit segment-level result masks $\mathcal{M}_{eq}$ and $\mathcal{M}_{lt}$. We interpret the bit-mask $\mathcal{M}_{lt}$ as a vector of $S/8$ banks, where all eight bits in the i-th bank are 1's if $v_i < c$ or all 0's otherwise. The bit mask $\mathcal{M}_{eq}$ is similarly interpreted for the condition $v_i = c$. The algorithm then examines the codes byte-by-byte through $\lceil \frac{k}{8} \rceil$ iterations (Lines 7–15). In the j-th iteration, it first inspects the mask $\mathcal{M}_{eq}$ to see if we can early stop (Lines 8–9).[5] If not, it loads the j-th bytes of all the codes in the segment into a SIMD register. It then executes two $S/8$-way SIMD instructions to determine the codes whose j-th bytes are either (1) $= c^{[j]}$ or (2) $< c^{[j]}$ (Lines 10–12). The comparison results are put into two local masks M_{eq} and M_{lt}, which are subsequently used to update the segment-level masks $\mathcal{M}_{eq}$ and $\mathcal{M}_{lt}$, respectively (Lines 13-14). After the iterations, the S-bit mask $\mathcal{M}_{lt}$, which contains the segment's comparison results, is condensed to a $S/8$-bit mask r. This is done by converting each bank of all 1's (0's) in $\mathcal{M}_{lt}$ into a single bit of 1 (0) in r using the SIMD `movemask` instruction (Line 16). Finally, the segment result r is appended to the final result R (Line 17) before the processing of the next segment begins. Algorithm 1 can be easily modified to handle other comparison operators (e.g., $\leq$). The details are given in Appendix B.

Algorithm 1 ByteSlice Column Scan ($<$)

Input: predicate $v < c$
Output: result bit vector R

```
1: for j = 1 ... ⌈k/8⌉ do
2:     W_cj = simd_broadcast(c^[j])          ▷ Word with c's j-th byte
3: end for
4: for every segment of S/8 codes v_{i+1} ... v_{i+S/8} do
5:     M_eq = 1^S                            ▷ a mask of S 1's
6:     M_lt = 0^S                            ▷ a mask of S 0's
7:     for j = 1 ... ⌈k/8⌉ do
8:         if simd_test_zero(M_eq) then
9:             break                         ▷ early stopping
10:        W_j = simd_load(v_{i+1}^[j] ... v_{i+S/8}^[j])   ▷ load the j-th bytes
11:        M_lt = simd_cmplt_epi8(W_j, W_cj)
12:        M_eq = simd_cmpeq_epi8(W_j, W_cj)
13:        M_lt = simd_or(M_lt, simd_and(M_eq, M_lt))
14:        M_eq = simd_and(M_eq, M_eq)
15:    end for
16:    r = simd_movemask_epi8(M_lt)          ▷ condense the mask
17:    Append r to R                         ▷ Append S/8 results to final R
18: end for
19: return R
```

3.1.1 Early Stopping

The early stopping condition (Line 8 in Algorithm 1) generally holds for all comparison conditions. Consider a t that is a multiple of 8. ByteSlice would have processed the t most significant bits of the codes in a segment after the $(t/8)$-th iteration. With $S/8$ codes in a segment, ByteSlice can early stop at that point if none of the codes in the segment matches the constant c in their $t/8$ most significant bytes. Assuming that any bit of a code matches the corresponding bit of c with a probability of 1/2, the probability, $P_{BS}(t)$, of ByteSlice early stopping after the t most significant bits are processed is given by:

$$P_{BS}(t) = \left(1 - (\frac{1}{2})^t\right)^{\frac{S}{8}} \quad (2)$$

Table 1 compares the early stopping probabilities of VBP and ByteSlice. For VBP, early stopping is checked for every $\tau = 4$ iterations even though each iteration handles one bit per code (see Section 2.2). So for VBP, only entries for t that are multiples of 4 are shown. Similarly, ByteSlice processes 1 byte (8 bits) per iteration, so only entries of t that are multiples of 8 are shown. First, from the table, we see that ByteSlice's early stopping probabilities are all larger than 0.88. That is highly desirable because it implies that the conditional branch given in Algorithm 1 Line 8 is highly predictable. Thus scans on ByteSlice incur low branch mis-prediction penalty. Second, we see that the chance of VBP early stopping at $t = 4$ is very slim (≈ 0). That means the one extra chance of early stopping with VBP at $t = 4$ (compared with ByteSlice whose first chance is at $t = 8$) is actually immaterial. Furthermore, we see that probability of ByteSlice early stopping after the first bytes (i.e., $t = 8$) is much higher than that of VBP (0.88 vs. 0.37). In fact, when $S = 256$, the expected number of bits that needed to be scanned by ByteSlice and VBP are respectively 8.94 and 10.79 per code. That is a difference of 1.85 bit per code. For the next generation SIMD whose registers are double in width ($S = 512$), the expected number bits that needed to be scanned by ByteSlice and VBP are respectively 9.78 and 11.96 per code. That translates into an even bigger difference of 2.18 bit per code.

Bit examined (t)	$P_{VBP}(t)$	$P_{BS}(t)$
4	0.0000000668	-
8	0.3671597549	0.8822809129
12	0.9394058945	-
16	0.9961013398	0.9995118342
20	0.9997558891	-
24	0.9999847413	0.9999980927
28	0.9999990463	-
32	0.9999999404	0.9999999925
Expected Value	scan 10.79 bits / code	scan 8.94 bits / code

Table 1: Early stopping probability under $S = 256$

Higher early stopping probability not only helps saving instruction executions (and thus running time), but also help reducing memory bandwidth consumption. Consider a code width of $k = 12$, by referring to Table 1, the expected bandwidth usage of ByteSlice is $0.88 \times 8 + (1 - 0.88) \times 16 \approx 8.94$ bits per code. Similarly, the expected bandwidth usage of VBP, which has two early stopping chances at $t = 4$ and $t = 8$, is ≈ 10.53 bits per code. HBP has no early stopping. It packs $\lfloor \frac{64}{k+1} \rfloor = \lfloor \frac{64}{13} \rfloor = 4$ codes per 64-bit bank, resulting in $4 \times 4 = 16$ codes per 256-bit memory word. That means it consumes 16 bits bandwidth per code. Bit-packed layout tightly packs the codes in memory. However, as scans on BP has no early stopping either, its bandwidth usage is exactly $k = 12$ bits per code.

We now come back to the discussion of how tailing bits are handled: Option 1 (pad them up as a byte; Figure 5a) and Option 2 (using VBP; Figure 5b). First, from our discussion above, we see that if a column is $9 < k < 16$ bits wide, there is a very high chance ($>$0.88) that a scan is early stopped after examining the first bytes of codes. The choice between the two options therefore is insignificant . In fact, we have tested the two options on TPC-H workloads and found that the overall performance difference between the two options is very minimal. Since Option 2 requires branching (to switch between ByteSlice and VBP for the last byte)[6] and it incurs a higher reconstruction cost for lookup, we recommend Option 1. Similar arguments also apply to columns whose width is 8 bits or less. In particular, scans and lookups on short codes generally consume very few cycles, rendering the choice between the

[5]The instruction `simd_test_zero()` (Line 8) is implemented by the `vptest` instruction in Intel AVX2.

[6]That would still increase the number of instructions even the branch can be eliminated using JIT query compiling [27].

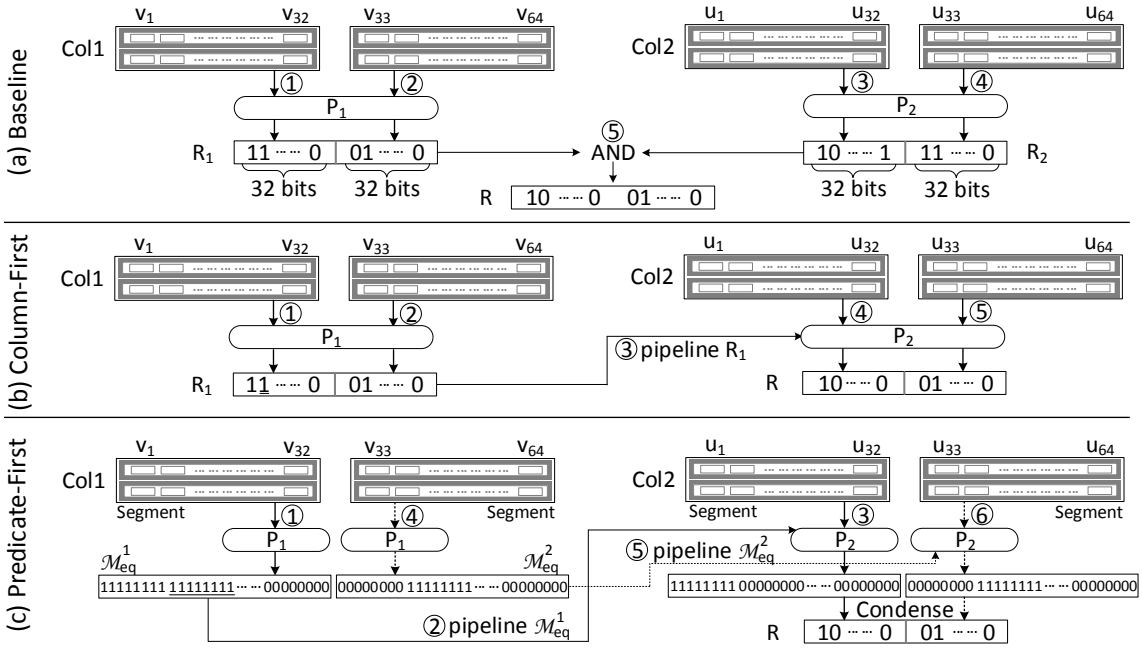


Figure 6: Three approaches to handle conjunctions. The labels ① ② ③, etc. denote the execution sequence.

two options insignificant. Moreover, our focus is actually more on columns with $k > 8$ because scans and lookups on those columns consume much more cycles than on columns with $k \leq 8$. That is, columns with $k > 8$ are the ones that throttle the overall query performance and being able to significantly reduce the cycles for them deserves more focus. Finally, Option 2 (VBP) still has the drawback of higher lookup costs when comparing with Option 1 (ByteSlice) after all. As we strive to reduce the code complexity for the various operations on top of ByteSlice data, we generally recommend the use of ByteSlice for all column widths.

3.1.2 Evaluating Complex Predicates

We now discuss how complex predicates that involve multiple columns are evaluated with ByteSlice. The baseline approach is to evaluate each predicate separately and then combine their result bit vectors. Figure 6a illustrates the baseline approach of evaluating a complex predicate `col1 < 5 AND col2 > 18`. The final result bit vector R is obtained by intersecting the result bit vector R_1 of evaluating predicate P_1: `col1 < 5` and the result bit vector R_2 of evaluating predicate P_2: `col2 > 18`.

Another approach is to *pipeline* the result bit vector of one predicate evaluation to another so as to increase the early stopping probability of the subsequent evaluations [31]. In our context, there are two possible implementations of such pipelining approach. Figure 6b shows the *column-first* implementation for the example above. After P_1 evaluation is done on the *whole* column `col1` (Steps ① and ②), its result bit vector R_1 is pipelined to the evaluation of P_2 on `col2`. This step can be implemented by modifying Algorithm 1 to (i) accept a result bit vector R_{prev} and (ii) for each $S/8$-bits r_{prev} from R_{prev}, execute an instruction that inverses "`simd_movemask`" to transform r_{prev} into a 256-bit mask $\mathcal{M}_{eq}$ (See Line 16 of Algorithm 1). However, AVX2 does not provide such an "inverse" `movemask` instruction, which has to be implemented using other instructions. To illustrate, consider a 32-bit result vector $r = 0\underline{1}00\ldots000$ whose bits except the 2nd one are all 0's (false). To expand r into a 256-bit mask $00000000\underline{111111110}\ldots0$, three instructions: `simd_shuffle_epi8`, `simd_and`, and `simd_cmpeq_epi8` are executed as illustrated in Figure 7. The overhead of executing these additional instructions, however, nullifies the benefit (e.g., efficiency obtained through early-stopping) of ByteSlice. To eliminate this overhead, we perform the following trick: Instead of "expanding" a pipelined 32-bit result vector r_{prev} into a 256-bit mask $\mathcal{M}_{eq}$, we "condense" $\mathcal{M}_{eq}$ to a 32-bit vector instead. Algorithm 2 gives the pseudo-code of this column-first implementation for the $<$ comparison. Algorithm 2 is modified from Algorithm 1. Changes are made at Lines 7, 9, and 18.

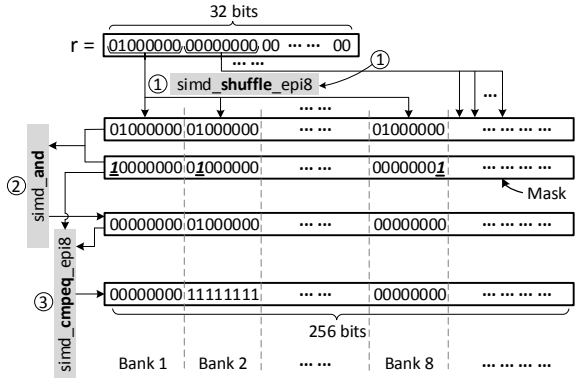


Figure 7: Simulating "inverse" `movemask` instruction.

Figure 6c shows an alternate implementation of the pipelining approach. We call this implementation *predicate-first* because it first processes all predicates for each segment of $S/8$ values before moving on to the next segment. In Figure 6c, after P_1 evaluation is done on a segment of $S/8$ codes from column `col1` (Step ①), the intermediate result $\mathcal{M}^1_{eq}$ for those $S/8$ codes is pipelined to the evaluation of P_2 on `col2` (Step ②). Unlike the column-first implementation, this predicate-first implementation does not need to movemask back-and-forth between r_{prev} and $\mathcal{M}_{eq}$ but simply pipeline $\mathcal{M}_{eq}$ until all predicates are evaluated. The column-first implementation cannot follow suit because it would have very large memory footprint to hold the $\mathcal{M}_{eq}$'s of all segments until the next

predicate is evaluated. To reduce the memory footprint, the column-first implementation must pipeline the condensed result bit vector instead.

Algorithm 2 Column-First Scan ($<$) Pipelined

Input: predicate $v < c$
Input: result bit vector R_{prev} from a previous predicate
Output: result bit vector R

1: **for** $j = 1 \dots \lceil \frac{k}{8} \rceil$ **do**
2: $W_{cj} = \texttt{simd\_broadcast}(c^{[j]})$ ▷ word with c's j-th byte
3: **end for**
4: **for** every $S/8$ values $v_{i+1} \dots v_{i+S/8}$ **do**
5: $\mathcal{M}_{eq} = 1^S$ ▷ a mask of S 1's
6: $\mathcal{M}_{lt} = 0^S$ ▷ a mask of S 0's
7: $r_{prev} = R_{prev}[i+1 \dots i+S/8]$ ▷ extract from R_{prev} the $S/8$ result bits for this segment
8: **for** $j = 1 \dots \lceil \frac{k}{8} \rceil$ **do**
9: **if** $(r_{prev}$ & $\texttt{simd\_movemask\_epi8}(M_{eq})) = 0$ **then**
10: **break** ▷ early stopping
11: $W_j = \texttt{simd\_load}(v_{i+1}^{[j]} \dots v_{i+S/8}^{[j]})$ ▷ load the j-th bytes
12: $M_{lt} = \texttt{simd\_cmplt\_epi8}(W_j, W_{cj})$
13: $M_{eq} = \texttt{simd\_cmpeq\_epi8}(W_j, W_{cj})$
14: $\mathcal{M}_{lt} = \texttt{simd\_or}(M_{lt}, \texttt{simd\_and}(\mathcal{M}_{eq}, M_{lt}))$
15: $\mathcal{M}_{eq} = \texttt{simd\_and}(\mathcal{M}_{eq}, M_{eq})$
16: **end for**
17: $r = \texttt{simd\_movemask\_epi8}(\mathcal{M}_{lt})$ ▷ condense the mask
18: Append $(r$ & $r_{prev})$ to R ▷ Append $S/8$ results to final R
19: **end for**
20: **return** R

The predicate-first implementation of the pipelining approach has its own pros and cons. This implementation needs fewer `movemask` instructions but it is more difficult for the compiler to optimize because the number of columns involved in a complex predicate is unknown until run-time. Furthermore, it switches to accessing another column for every $S/8$ values. As columns are located in different memory regions, it thus incurs more cache conflict misses (i.e., a useful cache line is evicted because another cache line is mapped to the same entry).

To evaluate disjunction, in Algorithm 2, we change r_{prev} to $\neg r_{prev}$ in Line 9. That is, only tuples that do *not* satisfy the previous predicate are considered. Next, we change r & r_{prev} to $r \mid r_{prev}$ in Line 18.

3.2 Lookup

Lookup in ByteSlice formatted memory data is simple. As each value is sliced into several bytes, we stitch the bytes back together and remove the padding zeros at the end if needed ($\ll$ and $\gg$ are left shift and right shift):

$$v_j = \Big(\sum_{i=1}^{\lceil \frac{k}{8} \rceil} \big(BS_i[j] \ll 8(\lceil \frac{k}{8} \rceil - i) \big) \Big) \gg (8\lceil \frac{k}{8} \rceil - k)$$

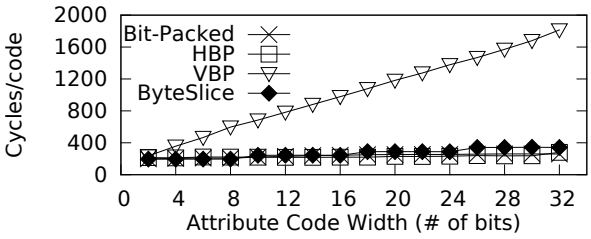
For example, looking-up (reconstructing) v_2 in Figure 5 needs:

$$\begin{aligned} v_2 &= \big(BS_1[2] \ll 8 + BS_2[2] \big) \gg 5 \\ &= \big(00001111_2 \ll 8 + 10000000_2 \big) \gg 5 \\ &= (00001111\ 10000000)_2 \gg 5 \\ &= (00001111100)_2 \end{aligned}$$

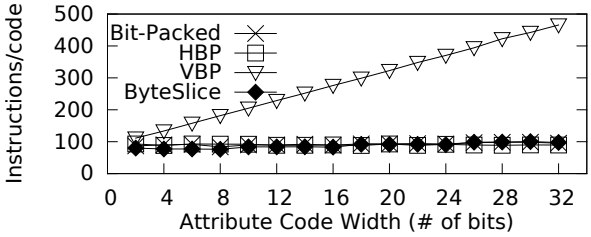
The above example involves two shifts, one addition and two memory reads. Roughly, for each *byte* involved, it requires a left shift and an addition (or bitwise OR). A right shift is probably needed at the end in order to remove padding bits. The possibly incurred cache misses are bounded by $\lceil \frac{k}{8} \rceil$. Since in TPC-H most attributes could be encoded using fewer than 24 bits, a lookup usually needs to handle no more than 3 bytes. Such a few instructions can be effectively overlapped in the processor's instruction pipeline, rendering its performance very close to that of Bit-Packed and HBP layouts.

	Bit-packed	VBP	HBP	**ByteSlice**
Scan Complexity	$\Theta(\frac{nb}{S})$	$O(\frac{nk}{S})$	$\Theta(\frac{nb}{S\lfloor b/(k+1) \rfloor})$	$O(\frac{8n\lceil k/8 \rceil}{S})$
Early Stop	No	Good	No	**Strong**
Lookup	**Good**	Poor	**Good**	**Good**

Table 2: Summary of Comparison



(a) **Cycles**



(b) **Instructions**

Figure 8: Lookup Performance

4. EXPERIMENTAL EVALUATION

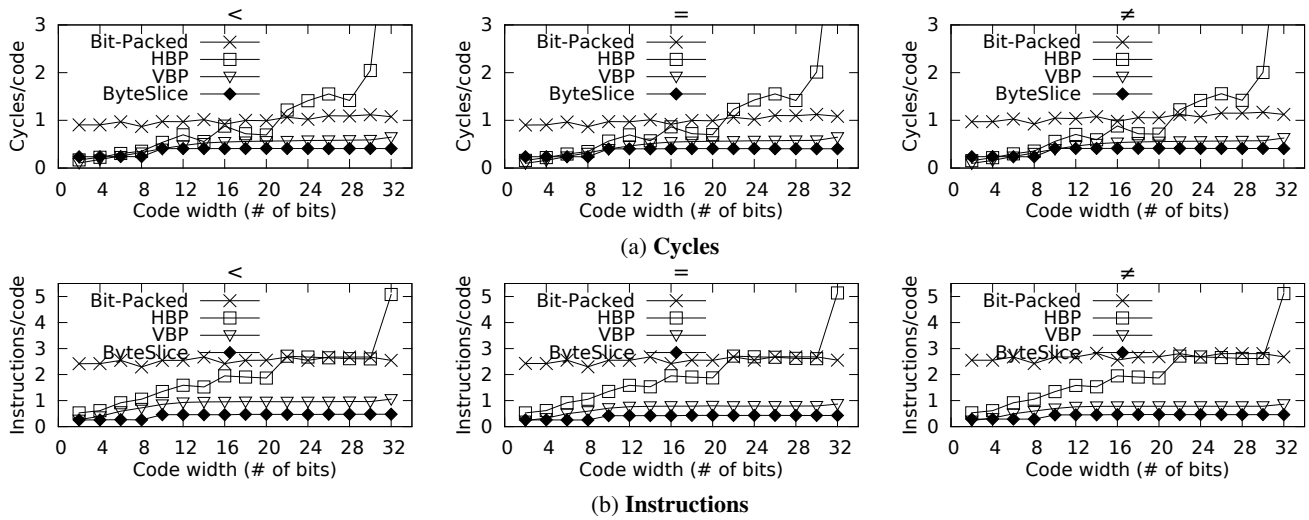
We run our experiments on a personal computer with a 3.40GHz Intel i7-4770 quad-core CPU, and 16GB DDR3 memory. Each core has 32KB L1i cache, 32KB L1d cache and 256KB L2 unified cache. All cores share an 8MB L3 cache. The CPU is based on Haswell microarchitecture which supports AVX2 instruction set. In the experiments, we compare ByteSlice with Bit-packed, VBP, and HBP. Table 2 summarizes their properties. Collectively, these competitors represent the state-of-the-art main memory storage layouts that support fast scans and lookups. We implement all methods in C++. All implementations are optimized using standard techniques such as prefetching. The programs are compiled using g++ 4.9 with optimization flag -O3. We use Intel Performance Counter Monitor [22] to collect the performance profiles. Unless stated otherwise, all experiments are run in a single process with a single thread.

4.1 Micro-Benchmark Evaluation

For micro-benchmarking, we create a table T with one billion tuples. Values in each column are by default uniformly distributed integer values between $[0, 2^k)$, where k is the width of the column, and is equal to 12 by default.

4.1.1 Lookup

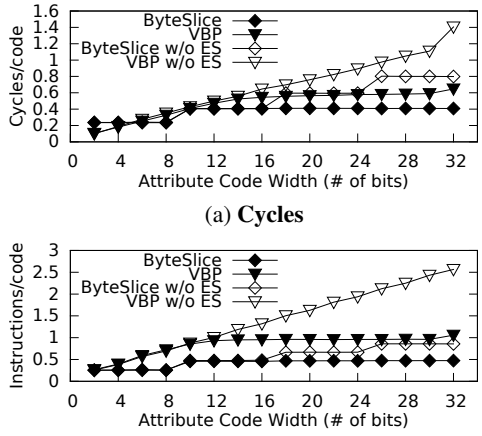
In this experiment, we compare the lookup performance on all layouts by varying the code width k. We perform one million random lookups and report (1) the average processor cycles per code



(a) **Cycles**

(b) **Instructions**

Figure 9: Scan Performance: (a) Execution Time and (b) Number of Instructions versus Code Width



(a) **Cycles**

(b) **Instructions**

Figure 10: Effectiveness of Early Stopping (ES) on Scans: VBP and ByteSlice (`T.v < c`)

and (2) the average number of instructions per code. The results are shown in Figure 8. We can see that lookups on VBP are significantly more expensive, in terms of both cycles and instruction, than the other methods. The lookup cost of VBP increases linearly with k, because every single bit of a VBP code is stored in a different memory word. A single VBP lookup can cost up to 1800 cycles for large k values.

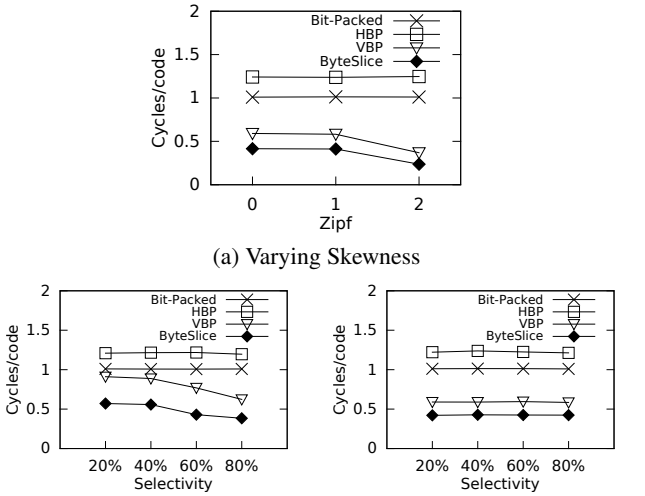
Lookups on Bit-packed, HBP, and ByteSlice data have comparable performance. Although the cost of a lookup on ByteSlice data shall increase piecewise linearly with $\lceil k/8 \rceil$, that is almost not noticeable in the experimental result because (1) the code-reconstruction process under ByteSlice is so lightweight that overlaps with other instructions in the instruction pipeline and (2) the incurred cache misses are bounded by $\lceil \frac{k}{8} \rceil$.

4.1.2 Scan

In this experiment, we evaluate column scan performance on all layouts by varying the code width k. The benchmark query is in the form of:

```
SELECT COUNT(*) FROM T WHERE T.v OP c
```

We present experimental results for `OP` as $<$, $=$, and $\neq$. Similar results are obtained for $>$, $\leq$, $\geq$ and we put them in Appendix C. The constant `c` in the WHERE clause is used to control the selectivity. By default we set the selectivity as 10%, i.e., 10% of the input tuples match the predicate. We put the results of other selectivity values in Appendix D.

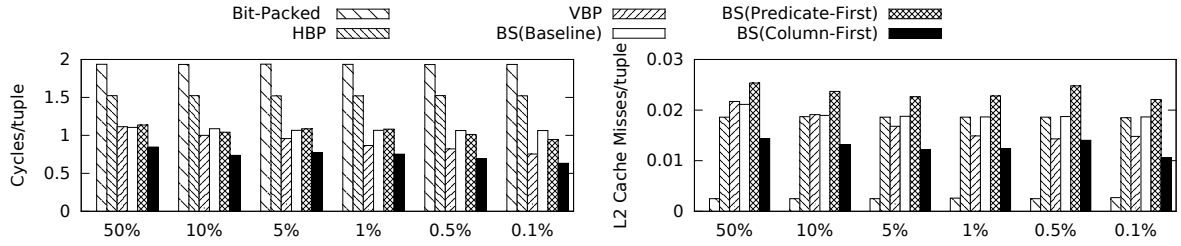


(a) Varying Skewness

(b) Varying Selectivity ($zipf = 1$) (c) Varying Selectivity (uniform)

Figure 11: Scan Performance (`T.v < c`)

Figure 9a and Figure 9b report the scan cost in terms of processor cycles per code and number of instructions spent per code respectively, when varying the column code width k. As can be seen in Figure 9b, scans on ByteSlice outperform all the other methods across all code widths in terms of the number of instructions. This translates into ByteSlice excellent performance in terms of cycles per code in Figure 9a. Scans on ByteSlice and VBP outperform scans on HBP and Bit-packed because of early stopping. For the same reason, increasing the code width does not significantly increase the scan costs on ByteSlice and VBP because scans are usually early stopped after examining early bytes for ByteSlice and early bits for VBP. When the code width is $22 \leq k \leq 30$, scans on HBP have the worst performance because starting from there, a



(a) **Cycles** (b) **L2 Cache Misses**

Figure 12: Evaluation of Complex Predicate (Conjunction)

64-bit SIMD bank can hold only two codes. The benefit of 2-way intra-bank parallelism turns out to be outweighed by the penalty of executing instructions that are required by HBP's scan algorithm. When the code width $k = 32$, a 64-bit SIMD bank can hold only one code (because HBP requires one extra delimiter bit per code). Consequently, a scan on HBP gains no advantage but overhead, explaining its abrupt rise in running time.

Figure 10 shows the effectiveness of early stopping when performing scans on VBP and ByteSlice. From Figure 10a, we see that (1) early stopping indeed bring performance improvement to both VBP and ByteSlice, and (2) without early stopping, scans on ByteSlice still perform better than scans on VBP because the former consumes fewer instructions than the latter (Figure 10b).

As the example given in Section 3.1 illustrated, given a scan query such as $v < c$, early stopping in ByteSlice (as well as in VBP) is most effective if the data values of a given segment are different from the query constant c in their more-significant bytes. That is, when the data values are not *close* to the query constant. Let us define *data density at a value* c to be the fraction of the column data values that are close to c. Skewness in data distribution affects the effectiveness of early stopping in VBP and ByteSlice scan because a skewed data distribution implies regions of high data density and regions of low density. More specifically, given a query constant c, if the data values are *skewed towards* c (i.e., the data density at c is high), then early stopping will be less effective. On the other hand, if data values are *skewed away* from c (i.e., the data density at c is low), early stopping is most effective.

To illustrate, we generate data values following a Zipfian distribution and change the skew factor from $zipf = 0$ (uniform distribution) to $zipf = 2$ (heavily skewed distribution). Figure 11a shows the running time for the predicate `T.v < c` with `c` set as 0.1×2^k by default. We see that ByteSlice consistently outperforms all other methods under all skewness. Moreover, as the skew factor increases, the running times by ByteSlice and VBP decrease. This is because for the Zipf distribution, increasing the skew factor has the effect of shifting the data density to the small values of the domain. With the fixed value of `c` (which is 10% of the value domain) chosen in the experiment, increasing the skewness causes the data density at c to become smaller, which makes early stopping more effective. This explains why the running time of ByteSlice (and VBP) improves when the data is getting more skewed. Figure 11b shows that the running times of ByteSlice and VBP decrease when we vary `c` under $zipf = 1$ skewed data. When `c` is small (e.g., when the selectivity of the query $v < c$ is 20%), the query constant lies in the dense region of the zipfian curve. Hence, early stopping is less effective, resulting in higher running times of ByteSlice and VBP. When `c` is large (e.g., when selectivity is 80%), the query constant lies in the sparse region of the zipfian curve, resulting in very effective early stopping and very low running times. Figure 11c shows that the running times of ByteSlice and VBP does not vary when we vary `c` under uniformly distributed data. This is because the data density is uniform across the domain and hence the effectiveness of early stopping stays constant.

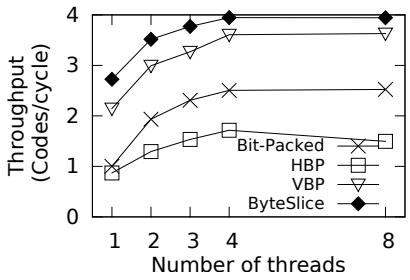


Figure 13: Scaling Scan on Multi-core CPU

4.1.3 Complex Predicate Evaluation on ByteSlice

In this experiment, we study the performance of the two implementations of the pipeline approach (column-first and predicate-first) and the baseline approach (i.e., no pipeline) when evaluating complex predicates on ByteSlice data. The WHERE clause of the benchmark query is in the form of:

```
WHERE T.col1 < c1 AND T.col2 > c2
```

For both implementations of the pipeline approach, the predicate `T.col1 < c1` is evaluated first and its result is pipelined to the predicate `T.col2 > c2`. In the experiment, we fix the constant `c2` to a value so to let the predicate `col2 > c2` to have a selectivity of 50%. We then vary the value of `c1` in order to control the selectivity of the predicate `col1 < c1`. Figure 12a shows the results in terms cycles per tuple.[7] We see that the column-first pipeline implementation, BS(Column-First), is the most efficient method of supporting conjunction evaluation, under all selectivities. When the predicate `T.col1 < c1` becomes more selective (i.e., from 50% to 0.1%), the running time of evaluating the whole query decreases because the evaluation of predicate `T.col2 > c2` runs faster by a higher early stopping chance. As expected, the predicate-first pipeline implementation, BS(Predicate-First), is not fruitful because its code path involves more branches. Furthermore, Figure 12b shows that the predicate-first pipeline implementation has more cache misses, due to accessing different memory regions more frequently than the column-first approach (Section 3.1.2). We have got similar results when evaluating disjunctions (Appendix E). We thus suggest the use of column-first pipeline implementation when evaluating complex predicates under ByteSlice.

[7]The results of Bit-Packed, VBP, and HBP are also included in the figure as a reference. Bit-Packed has the lowest cache miss (Figure 12a) merely because it is the slowest — as it processes the data slowly, it leaves sufficient time for the memory subsystem to (pre)-fetch the next item into the cache/instruction pipeline.

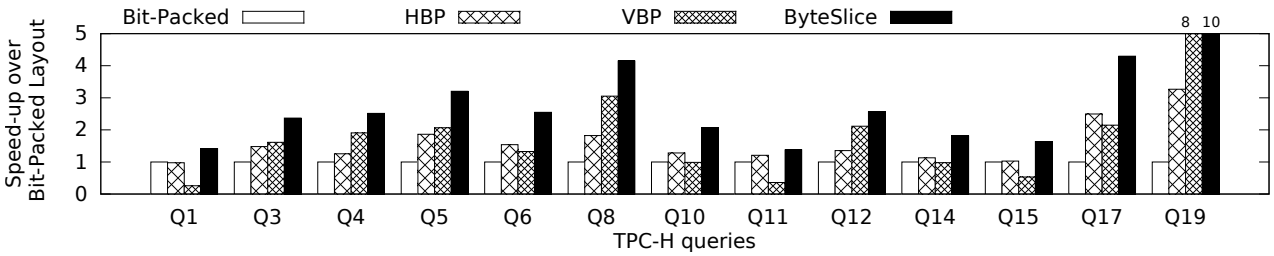


Figure 14: Speed-up over Bit-Packed on TPC-H Queries

4.1.4 Multi-Threading

In this experiment, we study the performance of using multiple threads to carry out scans on all layouts. Parallelizing data scans on multi-core is known to be straightforward. We simply partition data into chunks and assign chunks to the threads. Our CPU is quad-core, so we turn on simultaneous multi-threading (SMT) and vary the number of threads from one to eight.

Figure 13 shows the average scan throughput (number of codes processed per cycle) under the 4 memory layout schemes. The numbers shown are obtained by averaging the throughputs measured when the code width k is varied from 1 to 32 bits. From the figure, we can see the throughputs of all the schemes increase when more threads are used. The four schemes allow efficient data processing and thus they are utilizing the memory bandwidth very effectively. Computations become memory-bound when four threads are used. Among the four schemes, ByteSlice and VBP are extremely effective in using the memory bandwidth. In our experiment, we measured the bandwidth utilization under different scenarios. For example, we found that HBP, VBP and ByteSlice used more than 50% of the available memory bandwidth with a single thread. When two threads are used, HBP and VBP used more than 70% of the bandwidth and ByteSlice used more than 98% of the bandwidth. This explains the scale up behaviors of all schemes as shown in the figure. As mentioned in [31], effective bandwidth utilization is a key advantage of sophisticated storage layout schemes because one can fully exploit the potentials of high-end memory subsystems (which the in-memory appliances have) effectively. From our results, we see that ByteSlice takes this advantage to the next level.

When data processing has reached a state of memory-bound (such as when we are running many threads), ByteSlice still exhibits clear advantage over other schemes in throughput. As we have analyzed in Section 3.1.1, The early stopping properties of ByteSlice and VBP allow them to process codes without processing all the bits. This greatly improves their throughputs, especially for ByteSlice. HBP and Bit-packed offer no early stops and so their bandwidth usages increase with the code width k. Moreover, HBP uses delimiters and padding bits, it thus consumes even more bandwidth than Bit-packed in general. Consequently HBP gives the lowest throughput among the 4 schemes. When SMT threads are used, HBP's throughput further drops due to resource contention.

4.2 TPC-H and Real Data

We have also evaluated the overall performance of the methods using TPC-H benchmark. The experiments are run on a TPC-H dataset at scale factor 10. To focus only on scans and lookups, we follow [32] to materialize the joins and execute the selection-projection components of the queries. Queries that involve string similarity comparison `LIKE` are discarded.

Figure 14 shows the experimental results. The results are presented as speed-up over the bit-packed layout. The time breakdown of all queries is presented in Appendix F. When queries are not highly selective (e.g., Q1, Q14, Q15; whose selectivity $\geq 1\%$), ByteSlice layout yields the best performance for all tested queries because of its excellence on both scans and lookups. VBP performs worst and even worse than bit-packed because the lookup time dominates the query time, which unveils the poor lookup issue of VBP.

We have also repeated the experiments by using skewed TPC-H data (Appendix G) and real data (Appendix H). In those experiments, ByteSlice still consistently outperforms the others. Overall, this set of experiments shows the consistent high performance of ByteSlice over all state-of-the-art in-memory storage layout.

5. RELATED WORK

The PAX (Partition Attributes Across) layout [2] was one of the first studies on in-memory storage layout, which underlines the importance of processor architecture (e.g., cache and memory bandwidth utilization) in main memory processing. The motivation behind PAX is to keep the attribute values of each record in the same memory page as in traditional N-ary Storage Model (NSM), while using a cache-friendly algorithm for placing the values *inside* the page. PAX vertically partitions the records within each page, storing together the values of each attribute in "minipages", combining inter-record spatial locality and high data cache performance at no extra storage overhead. The PAX idea has been generalized in Data Morphing [19], which partitions the data based on the query load.

PAX and Data Morphing were storage layouts developed for row-stores. In the past decade, the analytical market has been dominated by pure column-store systems like Vertica [30], MonetDB/X100 [10], and SAP HANA [15]. In current main-memory column store implementations like Vectorwise [43] and SAP HANA, base column data by default are stored in standard data array [10]. Recently, the Bit-Packed storage layout [40, 39] was proposed to store the base column data in a tightly bit-packed manner in memory. By doing so, the memory bandwidth usage can be reduced when scanning (filtering) the base data columns. Li and Patel [31] proposed the Vertical Bit Packing (VBP) and Horizontal Bit Packing (HBP) layouts, which store the base column data in a way that fully exploits the intra-cycle parallelism in modern processors to accelerate scans. Later on, aggregations that leverage intra-cycle parallelism on VBP and HBP are also developed [16].

In current main-memory column store implementations, when an operator reads input produced from another operator, the input is assumed to be formatted using standard data array [26, 8, 9, 5, 4, 36, 13, 41, 21, 12, 34]. Under that implementation, after a lookup operation has retrieved a code from the base column, the code (together with its record number) is inserted into an array of, say, `struct {int32; int32;}`. Subsequent operations like joins, aggregations and sorts would then take the array as inputs. Since these operators are not directly processing data straight from the base columns, their performances are independent

of the storage layout of the base columns. Take join as an example. Existing hardware-conscious join algorithm [26, 8, 9, 5] all assume that their input is an array of <RecordID, JoinKey> pairs. Both RecordID and JoinKey are 32-bit integers. When a join is carried out, their major concern is to avoid excessive TLB misses and cache misses, and to reduce synchronization overheads among multi-threads. To this end, many works have engineered efficient partitioning algorithms. Kim et al. [26] proposed a lock-free histogram-based partitioning scheme that aims to parallelize on multi-core architectures. Moreover, they limit the partitioning's fan-out to the number of TLB entries so as to avoid TLB misses. They also exploit SIMD to accelerate hash computation. [5, 34] increased the partitioning fan-out without sacrificing performance by using an in-cache write buffer. Existing hardware-conscious aggregation algorithms [36, 13, 41] also assume input arrays of the form <GroupByKey, Value>. Since the the aggregate value (e.g., min or sum) would be updated by multiple threads concurrently, their major concern is to reduce the locking and contention overheads. A variety of approaches are investigated in [36, 13, 41]. For example, one can allocate a private aggregate buffer to each thread and merge them in the end. Likewise, existing hardware-conscious sorting algorithms [21, 12] also assume an array of <SortKey, RecordID> as input. When the sorting is carried out, the main concern is to exploit on-chip parallelism (e.g., SIMD) and to minimize cache misses and memory accesses. All these works are orthogonal to ByteSlice because they mostly read intermediate data (arrays) generated at runtime but not the base column data in ByteSlice format. Nonetheless, we envision that ByteSlice has the potential of being a representation of intermediate data as well. In that case, operations like partitioning and sorting could potentially benefit from ByteSlice. We will further elaborate this idea in Section 6.

Finally, a latest trend of main memory column stores is to pre-join tables upfront and materialize join results as one or more *wide tables* [43, 32]. Queries on the original database, even complex join queries, can then be handled as simple scans on wide tables. Such a denormalization approach would not incur much storage overhead in the context of column stores because of the various effective encoding schemes enabled by columnar storage. In [32], it was shown that such a "Denormalization + Columnar + VBP/HBP scans" combo can outperform MonetDB and Vectorwise on TPC-H without using much extra storage. The idea of wide table (denormalization) can also be applied to us, resulting in a "Denormalization + Columnar + ByteSlice scan/lookup" combo. We plan to investigate the performance of this combo in our future work.

6. CONCLUSION AND FUTURE WORK

Recently, there is a resurgence of interest in main memory analytic databases because of the large RAM capacity of modern servers (e.g., Intel Xeon E7 v2 servers can support 6TB of RAM) and the increasing demand for real-time analytic platforms. Existing in-memory storage layouts for columnar either accelerate scans at the cost of slowing down lookups or preserving good lookup performance with less efficient scans. This paper proposes ByteSlice, a new in-memory storage layout that supports both fast scans and fast lookups. Our experiments show that scans on ByteSlice can achieve 0.5 process cycle per column value, a new limit of main memory data scan, without sacrificing lookup performance. Experiments on TPC-H workload shows that ByteSlice outperforms all state-of-the-art approaches and can be up to 3X to 10X faster than existing approaches.

The research of main memory column stores is still ongoing. In current main memory column stores like Vectorwise and SAP HANA, operations other than scan and lookup do not expect their input data being formatted using any specialized layout other than arrays. We envision that ByteSlice could be used as *a layout for intermediate result* as well. That is, the lookup operations retrieve codes from ByteSlice-formatted column and construct intermediate results in ByteSlice format. In that case, operations such as partitioning, sorting and searching could potentially benefit from ByteSlice. In the following, we briefly outline our idea:

Partitioning Partitioning data is an essential step of many operations in main memory database including joins and aggregations. The state-of-the-art partitioning method is multi-pass hash radix partitioning [26, 8, 5, 4, 34]. During each pass, it partitions a key k based on R of the radix bits of k's hash value and assigns k to one of the 2^R partitions. When partitioning a chunk of data D in parallel (e.g., latching a partition and chaining), the partitioning first scans the data once, computes the hash values of keys, and builds a histogram of the partitions' sizes. Based on the histogram and the degree of parallelism, a write cursor over the buffer is designated for each partition. Next, the partitioning operation scans the data the second time in parallel, computes the hash values of keys again, and inserts the keys into corresponding positions of the output buffer. In the process, hash values computation are done twice. Hash values computation can be vectorized using SIMD. In [26], keys are represented using standard data types (32-bit integers) and thus the hash value computation can be run in $S/32 = 8$-way parallelism on AVX2. If the input data of a partition operation are formatted using (8-bit) ByteSlice, the parallelism of hash value computation can be improved to $S/8 = 32$-way on AVX2. To do so, we simply need to devise hash functions that take as input the bytes of a code and return a byte-wide hash value.

Sorting One powerful sorting algorithm for main-memory databases is radix sort [38, 34]. For example, consider sorting 16-bits (2-byte) codes. If the input data are formatted using ByteSlice, we shall have two ByteSlices BS_1 and BS_2. In this case, we can employ a least-significant-*byte* radix sort. We sort on ByteSlice BS_2 in the first iteration and sort on BS_1 in the second iteration. By having data stored in the ByteSlice format, after sorting a ByteSlice in an iteration (e.g., sorting BS_2 in the first iteration), that ByteSlice does not need to stay in the working set anymore. We can therefore progressively reduce the memory footprint of radix sort as we proceed through the iterations.

Searching Searching is to find all matches of a search key in a list of values. It is a basic operation employed by many other operations like nested loop joins and is used in the probe phase of hash joins. Native SIMD-accelerated searching algorithms are introduced in [42], where keys and values are stored using standard data type (32-bit integers), resulting in a $S/32 = 8$-way SIMD search in AVX2. If the input data are formatted using (8-bit) ByteSlice, search can enjoy $S/8 = 32$-way parallelism with early stopping offered by ByteSlice.

Acknowledgements

This work is partly supported by the Research Grants Council of Hong Kong (GRF PolyU 520413, 521012 and GRF HKU 712712E) and a research gift from Microsoft Hong Kong.

We would like to thank Professor Jignesh Patel and Mr. Yinan Li for their insightful advices. We would also thank the anonymous reviewers for their comments and suggestions.

7. REFERENCES

[1] Oracle Exalytics In-memory Machine. http://www.oracle.com/us/solutions/ent-performance-bi/business-intelligence/exalytics-bi-machine/overview/index.html.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] K. Bache and M. Lichman. UCI machine learning repository, 2013. http://archive.ics.uci.edu/ml.

[4] C. Balkesen, G. Alonso, and M. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 2013.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[6] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, et al. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 2012.

[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.

[9] S. Blanas and J. M. Patel. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *SoCC'13*, 2013.

[10] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[11] S. Chaudhuri and V. Narasayya. Program for TPC-D data generation with skew, 2012.

[12] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 2008.

[13] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *PVLDB*, 2007.

[14] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 2010.

[15] 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.

[16] Z. Feng and E. Lo. Acclerating Aggregation using Intra-cycle Parallelism. In *ICDE*, 2015.

[17] Google. Supersonic library. https://code.google.com/p/supersonic/.

[18] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, 2010.

[19] R. A. Hankins and J. M. Patel. Data morphing: an adaptive, cache-conscious storage technique. In *VLDB*, 2003.

[20] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 2012.

[21] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, 2007.

[22] Intel. Intel Performance Counter Monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor/.

[23] Intel. Intel architecture instruction set extentions programming reference, 2013.

[24] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 2014.

[25] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[26] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2009.

[27] I. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. In *VLDB*, 2014.

[28] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *PVLDB*, 2011.

[29] S. Lahman. Baseball database. http://www.seanlahman.com/baseball-archive/statistics/.

[30] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 2012.

[31] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, 2013.

[32] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 2014.

[33] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *ACM SIGMOD Record*, 1997.

[34] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *SIGMOD*. ACM, 2014.

[35] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, 2008.

[36] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, 1995.

[37] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal*, pages 1–26, 2013.

[38] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par 2011 Parallel Processing*, pages 160–169. Springer, 2011.

[39] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing database column scans with complex predicates. In *ADMS Workshop*, 2013.

[40] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2009.

[41] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DeMoN*, 2011.

[42] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, 2002.

[43] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, 2012.

APPENDIX

A. (8-BIT) BYTESLICE VS. 16-BIT-SLICE

We select 8 instead of 16 as the bank width because attributes in real-world workloads are usually encoded using 24 bits or fewer. For example, we found that 90% columns in TPC-H are shorter than 24 bits after encoding. All columns in the two real datasets that we used in the experiments are shorter than 20 bits after encoding. When column widths fall into that range:

1. Using 16-bit bank width could consume more storage space than 8-bit bank. For example, a 20-bit attribute consumes $3 \times 8 = 24$ bits/code under ByteSlice but $2 \times 16 = 32$ bits/code if 16-bit banks are used.

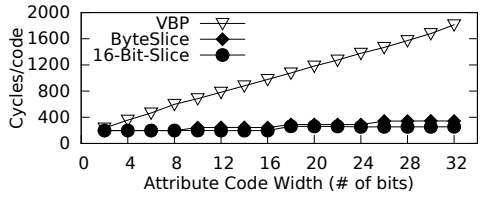
2. Using 16-bit bank width could only leverage 16-way (if $S = 256$) parallelism while using 8-bit bank width could leverage 32-way (if $S = 256$) parallelism.

3. Using 16-bit bank would not reduce lookup overhead much comparing with using 8-bit bank. For example, looking up a

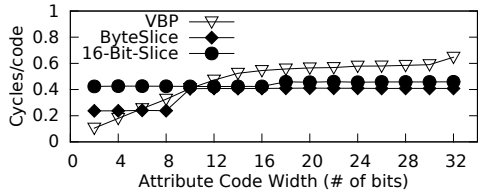
20-bit attribute under 16-bit bank requires accessing 2 memory words whereas looking up a 20-bit attribute under 8-bit bank requires accessing 3 memory words. That difference could easily be overlapped in the instruction pipeline.

Moreover, using 32 as the bank width is meaningless because it simply degrades to the naive SIMD approach.

We have verified our claims by implementing 16-bit-slice and compared its scan and lookup performance with (8-bit) ByteSlice and VBP. Figure 15 shows that (8-bit) ByteSlice always outperforms 16-bit-slice in scans and have very close performance in lookup. Based on our empirical evaluation we use 8 as the bank width in this paper.



(a) **Lookup**



(b) **Scan**

Figure 15: Lookup (a) and Scan (b) performance of using 16 bits as the bank width.

B. EVALUATING OTHER COMPARISON OPERATORS

In the following, we present how to extend Algorithm 1 to handle other comparison operators:

GREATER-THAN ($>$) Replace the instruction `simd_cmplt_epi8()` by `simd_cmpgt_epi8()` and rename the variable $\mathcal{M}_{lt}$ to $\mathcal{M}_{gt}$.

EQUAL ($=$) Remove Line 11 (M_{lt}) and Line 13 ($\mathcal{M}_{lt}$) from Algorithm 1. Use the mask $\mathcal{M}_{eq}$ instead of $\mathcal{M}_{lt}$ in Line 16.

NOT-EQUAL ($\neq$) To evaluate NOT-EQUAL ($\neq$), further negate r before appending it to R in Line 17.

LESS-THAN-OR-EQUAL-TO ($\leq$) and GREATER-THAN-OR-EQUAL-TO ($\geq$) Change Line 16 to

$$r = \texttt{simd\_movemask\_epi8}(\texttt{simd\_or}(\mathcal{M}_{lt}, \mathcal{M}_{eq})).$$

Ditto for GREATER-THAN-OR-EQUAL-TO ($\geq$).

BETWEEN The `BETWEEN` predicate "$c_1 \leq v \leq c_2$" is evaluated by the conjunction of two predicates: $c_1 \leq v$ and $v \leq c_2$.

C. SCAN PERFORMANCE OF OTHER PREDICATES

We report the scan performance of ByteSlice with other predicate types: GREATER-THAN($>$), GREATER-THAN-OR-EQUAL-TO ($\geq$) and LESS-THAN-OR-EQUAL-TO ($\leq$), in Figure 16. As we can see, the results are similar to the previous predicates we reported in Figure 9.

D. SCAN PERFORMANCE OF OTHER SELECTIVITY

We also run experiments with other selectivity. Figure 17 shows the results of the benchmark query with selectivity 90%. Figure 18 shows the results of the benchmark query with selectivity 1%. They show that the scan cost is not affected by the selectivity of the query under uniform data.

E. EVALUATION OF DISJUNCTION PREDICATES

We study the performance of the two implementations of the pipeline approach and the baseline approach with a *disjunction* predicate in the form of:

```
WHERE T.col1 < c1 OR T.col2 > c2
```

Similarly to Section 4.1.3, we fix the selectivity of `col2` and vary the selectivity of `col1`. Slightly unlike conjunction, in disjunction, a predicate only considers those tuples that did *not* pass the previous predicate. Therefore, a high selectivity of `col1` helps to increase the early stopping probability of `col2`, and thus reducing execution time. This is confirmed by the results reported in Figure 19. Again, we observe that the column-first pipeline implementation outperforms the other alternates. We conclude that it is the consistently best choice of complex predicate evaluation approach. Experimental results for Bit-Packed, HBP and VBP in Figure 19 are also similar to conjunction results in Figure 12.

F. TIME BREAKDOWN OF TPC-H EXPERIMENTS

We report the execution time breakdown of TPC-H queries in Figure 20. 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 that TPC-H benchmark has both scan-dominant (e.g., Q4 and Q19) and lookup-dominant (e.g., Q1) queries. A couple of queries sit in the middle with different weights on the two operations. The results show that ByteSlice strikes an excellent balance between scan and lookup across industrial-strength TPC-H workload.

G. EVALUATION USING TPC-H SKEW DATA

We use the data generator from [11] to generate Zipfian skewed TPC-H data. We generate skewed data with skew factor $zipf = 1$ and 2. Both skewed data sets are of scale factor 10GB.

The results are shown in Figure 21. The experimental results are still consistent — TPC-H queries on ByteSlice data outperform the other methods across the whole workload under different degrees of skewness.

H. EVALUATION USING REAL DATA

We also carry out experiments using two real data sets, ADULT and BASEBALL. ADULT [3] is a single-relation demographic data set extracted from the 1994 Census database. BASEBALL [29] is a multi-relation data set that contains statistics for Major League Baseball from 1871 to 2013. The queries on that two real datasets are extracted from [37]. We discard queries that have no selection clauses.

Figure 22 shows the experimental results on that two datasets. Again, we see that ByteSlice outperforms all competitors.

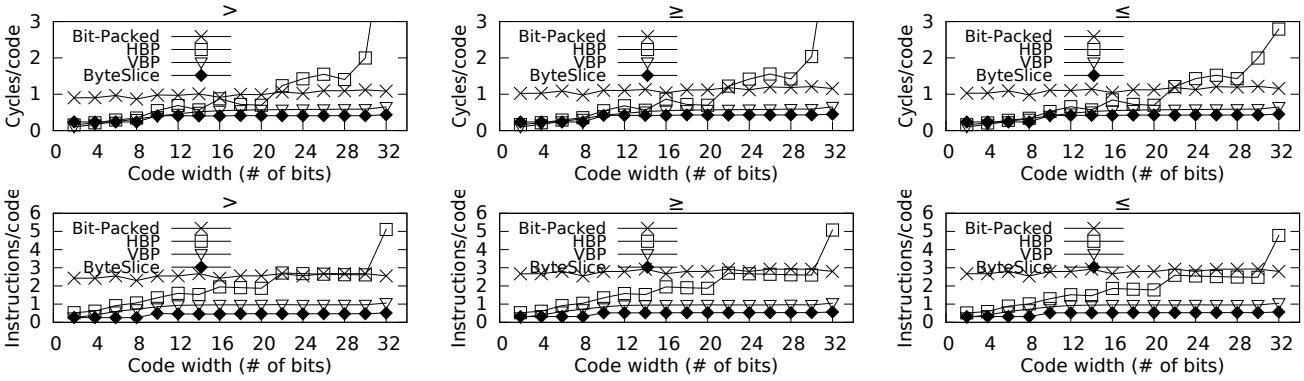


Figure 16: Scan Performance: Other Predicates

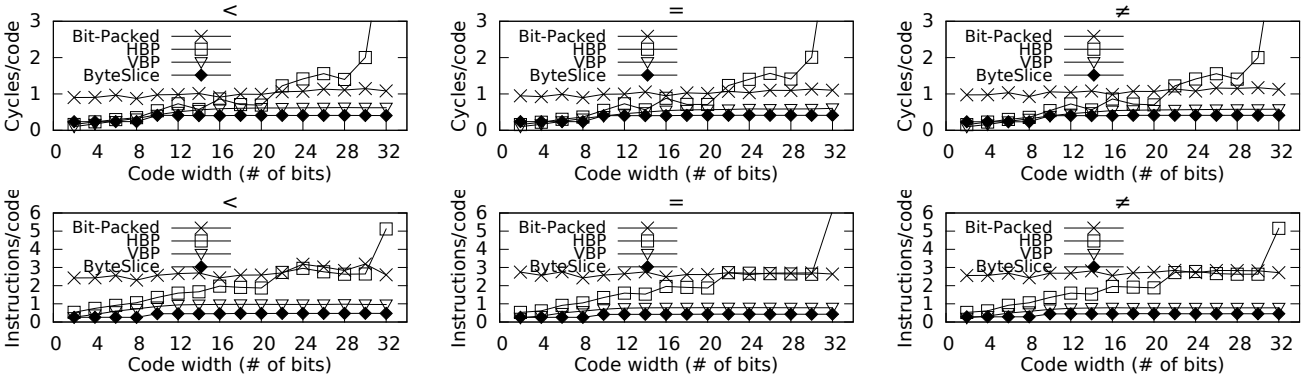


Figure 17: Scan Performance: Selectivity = 90%

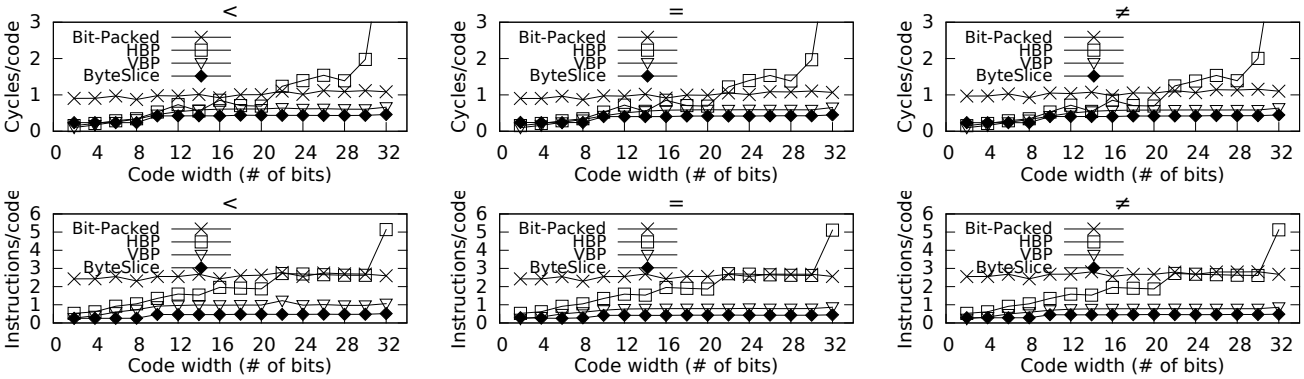
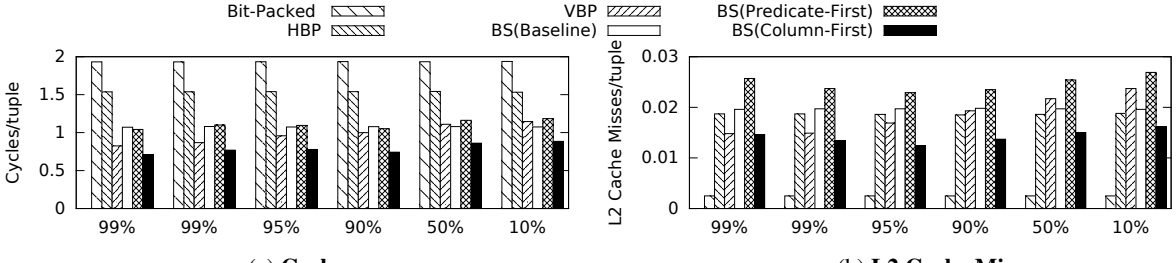


Figure 18: Scan Performance: Selectivity = 1%



(a) **Cycles** (b) **L2 Cache Misses**

Figure 19: Evaluation of Complex Predicate (Disjunction)

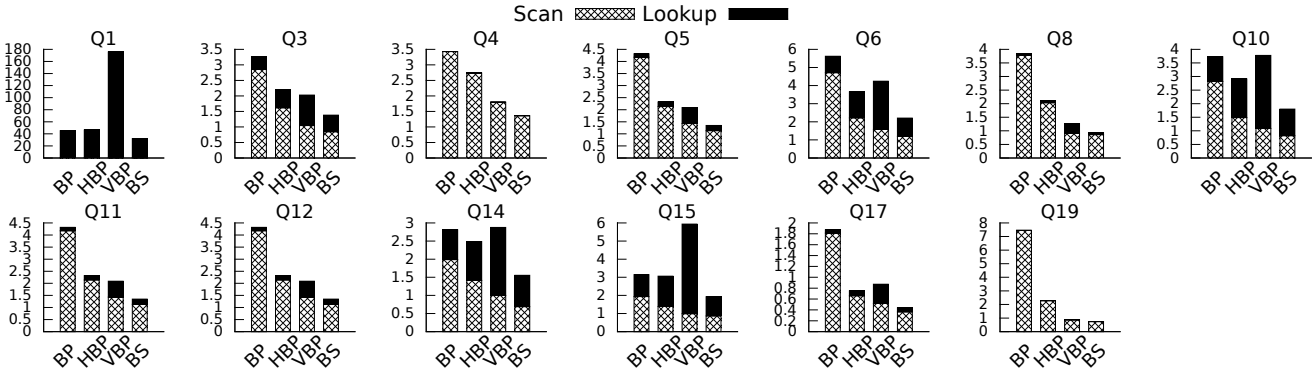
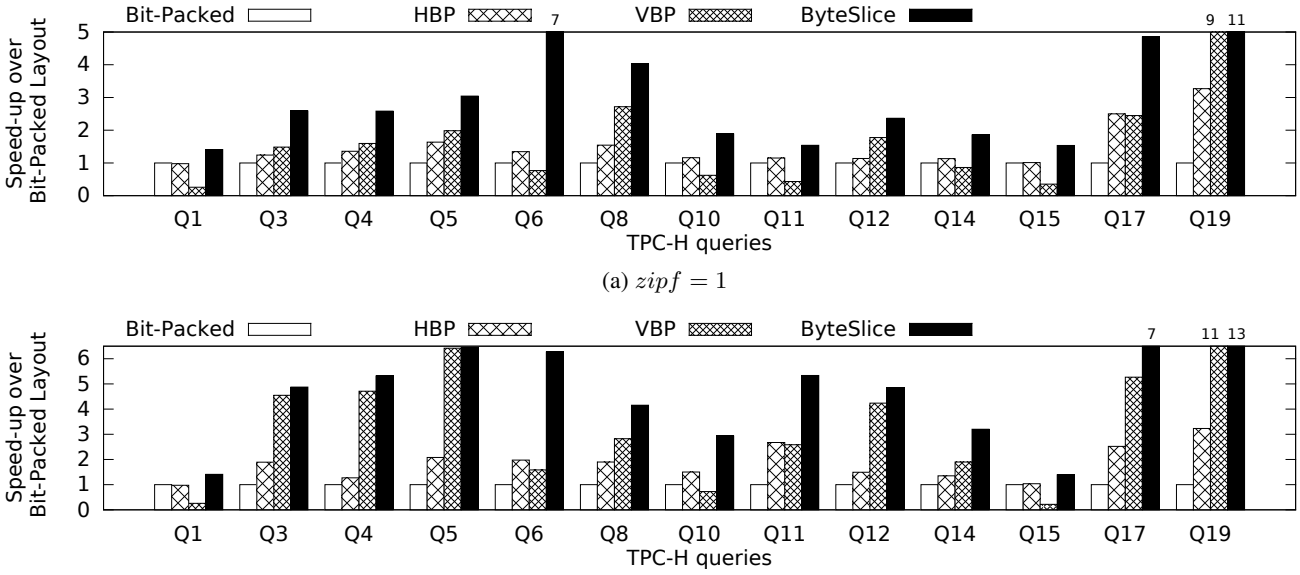


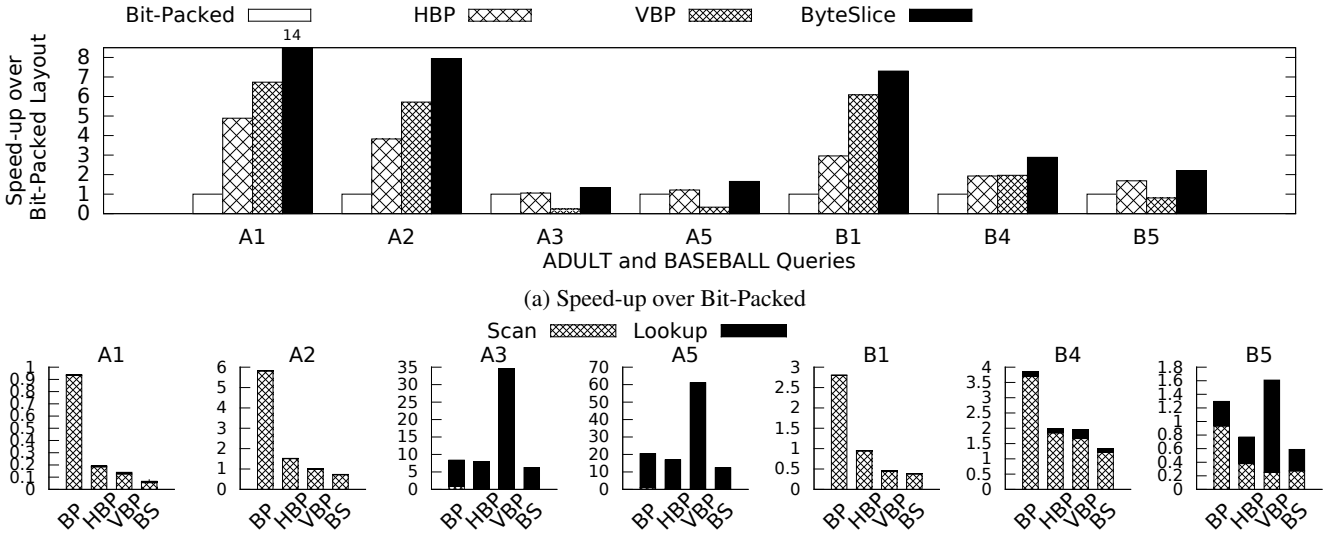
Figure 20: Execution Time Breakdown for TPC-H Queries. Y-axis reports cycles per tuple.



(a) $zipf = 1$

(b) $zipf = 2$

Figure 21: Speed-up over Bit-Packed on TPC-H Queries with Zipfian Data.



(a) Speed-up over Bit-Packed

(b) Execution Time Breakdown (Y-axis reports cycles per tuple)

Figure 22: Performance of Different Layouts on Two Real Data Sets: ADULT (Queries A*) and BASEBALL (Queries B*).