BitWeaving: Fast Scans for Main Memory Data Processing

Yinan Li
University of Wisconsin–Madison
yinan@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin–Madison
jignesh@cs.wisc.edu

ABSTRACT

This paper focuses on running scans in a main memory data processing system at "bare metal" speed. Essentially, this means that the system must aim to process data at or near the speed of the processor (the fastest component in most system configurations). Scans are common in main memory data processing environments, and with the state-of-the-art techniques it still takes many cycles per input tuple to apply simple predicates on a single column of a table. In this paper, we propose a technique called BitWeaving that exploits the parallelism available at the bit level in modern processors. BitWeaving operates on multiple bits of data in a single cycle, processing bits from different columns in each cycle. Thus, bits from a batch of tuples are processed in each cycle, allowing BitWeaving to drop the cycles per column to below one in some case. BitWeaving comes in two flavors: BitWeaving/V which looks like a columnar organization but at the bit level, and BitWeaving/H which packs bits horizontally. In this paper we also develop the arithmetic framework that is needed to evaluate predicates using these BitWeaving organizations. Our experimental results show that both these methods produce significant performance benefits over the existing state-of-the-art methods, and in some cases produce over an order of magnitude in performance improvement.

Categories and Subject Descriptors

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

Keywords

Bit-parallel, intra-cycle parallelism, storage organization, indexing, analytics

1. INTRODUCTION

There is a resurgence of interest in main memory database management systems (DBMSs), due to the increasing demand for real-time analytics platforms. Continual drop in DRAM prices and increasing memory densities have made it economical to build and deploy "in memory" database solutions. Many systems have been developed to meet this growing requirement [2, 5–7, 9, 14, 21].

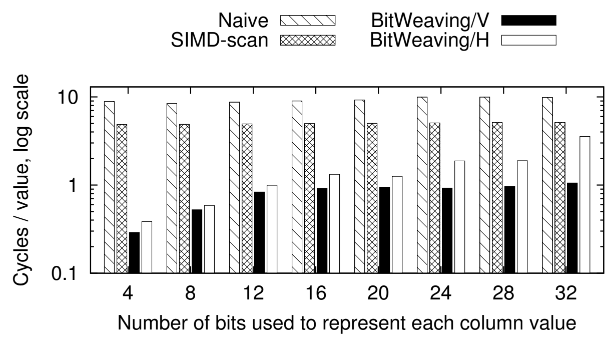


Figure 1: Performance Comparison

A key operation in a main memory DBMS is the full table scan primitive, since ad hoc business intelligence queries frequently use scans over tabular data as base operations. An important goal for a main memory data processing system is to run scans at the speed of the processing units, and exploit all the functionality that is available inside modern processors. For example, a recent proposal for a fast scan [18] packs (dictionary) compressed column values into four 32-bit slots in a 128-bit SIMD word. Unfortunately, this method has two main limitations. First, it does not fully utilize the width of a word. For example, if the compressed value of a particular attribute is encoded by 9 bits, then we must pad each 9-bit value to a 32-bit boundary (or what ever is the boundary for the SIMD instruction), wasting $32 - 9 = 23$ bits every 32 bits. The second limitation is that it imposes extra processing to align tightly packed values to the four 32-bit slots in a 128-bit SIMD word.

In this paper, we propose a set of techniques, which are collectively called BitWeaving, to aggressively exploit "intra-cycle" parallelism. The insight behind our intra-cycle parallelism paradigm is recognizing that in a single processor clock cycle there is "abundant parallelism" as the circuits in the processor core are simultaneously computing on multiple bits of information, even when working on simple ALU operations. We believe that thinking of how to fully exploit such intra-cycle parallelism is critical in making data processing software run at the speed of the "bare metal", which in this study means the speed of the processor core.

The BitWeaving methods that are proposed in this paper target intra-cycle parallelism for higher performance. BitWeaving does not rely on the hardware-implemented SIMD capability, and can be implemented with full-word instructions. (Though, it can also leverage SIMD capabilities if that is available.) BitWeaving comes in two flavors: BitWeaving/V and BitWeaving/H, corresponding to two underlying storage formats. Both methods produce as output a result bit vector, with one bit per input tuple that indicates if the input tuple matches the predicate on the column.

The first method, BitWeaving/V, uses a *bit-level* columnar data organization, packed into processor words. It then organizes the words into a layout that results in largely sequential memory address lookups when performing a scan. Predicate evaluation in the scan operation is converted to logical computation on these "words of bits" using the arithmetic framework proposed in this paper. In this organization, storage space is not wasted padding bits to fit boundaries that are set by the hardware. More importantly, in many cases, an early pruning technique allows the scan computation to be safely terminated, even before all the bits in the underlying data are examined. Thus, predicates can often be computed by only looking at some of most significant bits in each column. This scheme also naturally produces compact result bit vectors that can be used to evaluate the next stage of a complex predicate efficiently.

The second method, BitWeaving/H, uses a bit organization that is a dual of BitWeaving/V. Unlike the BitWeaving/V format, all the bits of a column value are stored together in BitWeaving/H, providing high performance when fetching the entire column value. Unlike previous horizontal bit packing methods, BitWeaving/H staggers the codes across processor words in a way that produces compact result bit vectors that are easily reusable when evaluating the next stage of a complex predicate.

Both BitWeaving methods can be used as a native storage organization technique in a column store database, or as an indexing method to index specific column(s) in row stores or column stores.

Figure 1 illustrates the performance of a scan operation on a single column, when varying the width of the column from 1 bit to 32 bits (Section 6 has more details about this experiment). This figure shows the SIMD-scan method proposed in [18], and a simple method (labeled Naive) that scans each column in a traditional scan loop and interprets each column value one by one. As can be seen in the figure, both BitWeaving/V and BitWeaving/H outperform the other methods across all the column widths. Both BitWeaving methods achieve higher speedups over other methods when the column representation has fewer number of bits, because this allows more column predicates to be computed in parallel (i.e. the intra-cycle parallelism per input column value is higher). For example, when each column is coded using 4 bits, the BitWeaving methods are **20X** faster than the SIMD-scan method. Even for columns that are wider than 12 bits, both BitWeaving methods are often more than **4X** faster than the SIMD-scan method. Note that as described in [18], real world data tends to use 8 to 16 bits to encode a column; BitWeaving is one order of magnitude faster than the SIMD-scan method within this range of code widths.

The contribution of this paper is the presentation of the BitWeaving methods that push our intra-cycle parallelism paradigm to its natural limit – i.e. to the bit level for each column. We also develop an arithmetic framework for predicate evaluation on BitWeaved data, and present results from an actual implementation.

The remainder of this paper is organized as follows: Section 2 contains background information. The BitWeaving methods and the related arithmetic framework is described in Sections 3 through 5. Section 6 contains our experimental results. Related work is covered in Section 7, and Section 8 contains our concluding remarks.

2. OVERVIEW

Main memory analytic DBMSs often store data in a compressed form [2, 4, 5, 10]. The techniques presented in this paper apply to commonly used column compression methods, including null suppression, prefix suppression, frame of reference, and order-preserving dictionary encoding [2,4,5,10]. Such a scheme compresses columns using a fixed-length order-preserving scheme, and converts the native column value to a *code*. In this paper, we use the term "code" to mean an encoded column value. The data for a column is represented using these codes, and these codes only use as many bits as are needed for the fixed-length encoding.

In these compression methods, all value types, including numeric and string types, are encoded as an unsigned integer code. For example, an order-preserving dictionary can map strings to unsigned integers [3, 10]. A scale scheme can convert floating point numbers to unsigned integers by multiplying by a certain factor [4]. These compression methods maintain an order-preserving one-to-one mapping between the column values to the codes. As a result, column scans can usually be directly evaluated on the codes.

For predicates involving arithmetic or similarity predicates (e.g. the `LIKE` predicates on strings), scans cannot be performed directly on the encoded codes. These codes have to be decoded, and then are evaluated in a conventional way.

2.1 Problem Statement

A *column-scalar scan* takes as input a list of n k-bit codes and a predicate with a basic comparison, e.g. $=, \neq, <, >, \leq, \geq,$ `BETWEEN`, on a single column. Constants in the predicate are also in the domain of the compressed codes. The column-scalar scan finds all matching codes that satisfy the predicate, and outputs an n-bit vector, called the *result bit vector*, to indicate the matching codes.

A *processor word* is a data block that can be processed as a unit by the processor. For ease of explanation, we initially assume that a processor word is an Arithmetic Logic Unit (ALU) word, i.e. a 64-bit word for modern CPUs, and in Appendix C of the extended version of this paper [12] we generalize our method for wider words (e.g. SIMD). The instructions that process the processor word as a unit of computation are called *full-word instructions*. Next, we define when a scan is a *bit-parallel method*.

DEFINITION 1. *If w is the width of a processor word, and k is the number of bits that are needed to encode a code in the column C, then a column-scalar scan on column C is a bit-parallel method if it runs in $O(\frac{nk}{w})$ full-word instructions to scan over n codes.*

A bit-parallel method needs to run in $O(\frac{nk}{w})$ instructions to make full use of the "parallelism" that is offered by the bits in the entire width of a processor word. Since processing nk bits with w-bit processor words requires at least $O(\frac{nk}{w})$ instructions, intuitively a method that matches the $O(\frac{nk}{w})$ bound has the potential to run at the speed of the underlying processor hardware.

2.2 Framework

The focus of this paper is on speeding up scan queries on columnar data in main memory data processing engines. Our framework targets the single-table predicates in the WHERE clause of SQL. More specifically, the framework allows conjunctions, disjunctions, or arbitrary boolean combinations of the following basic comparison operators: $=, \neq, <, >, \leq, \geq,$ `BETWEEN`.

For the methods proposed in this paper, we evaluate the complex predicate by first evaluating basic comparisons on each column, using a *column-scalar scan*. Each column-scalar scan produces a *result bit vector*, with one bit for each input column value that indicates if the corresponding column value was selected to be in the result. Conjunctions and disjunctions are implemented as logical AND and OR operations on these result bit vectors. Once the column-scalar scans are complete, the result bit vector is converted to a list of record numbers, which is then used to retrieve other columns of interest for this query. (See Appendix A in [12] for more details.) NULL values and three-valued boolean logic can be implemented in our framework using the techniques proposed in [13], and, in the interest of space, this discussion is omitted here.

We represent the predicates in the SQL WHERE clause as a binary predicate tree. A leaf node encapsulates a basic comparison operation on a single column. The internal nodes represent logical operation, e.g. AND, OR, NOT, on one or two nodes. To evaluate a predicate consisting of arbitrary boolean combinations of basic comparisons, we traverse the predicate tree in depth-first order, performing the column-scalar comparison on each leaf node, and merging result bit vectors at each internal node based on the logical operator that is represented by the internal node. Figure 8 illustrates an example predicate tree. In Section 3, we focus on single-column scans, and we discuss complex predicates in Section 4.3.

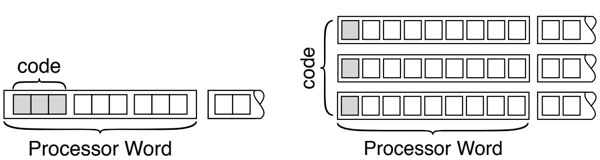
3. BIT-PARALLEL METHODS

In this section, we propose two bit-parallel methods that are designed to fully utilize the entire width of the processor words to reduce the number of instructions that are needed to process data. These two bit-parallel methods are called Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP) methods. Each method has a storage format and an associated method to perform a column-scalar scan on that storage method. In Section 4, we describe an early pruning technique to improve on the column-scalar scan for both HBP and VBP. Then, in Section 5 we describe the BitWeaving method, which combines the bit-parallel methods that are described below with the early pruning technique. BitWeaving comes in two flavors: BitWeaving/H and BitWeaving/V corresponding to the underlying bit-parallel method (i.e. HBP or VBP) that it builds on.

3.1 Overview of the two bit-parallel methods

As their names indicate, the two bit-parallel methods, HBP and VBP, organize the column codes horizontally and vertically, respectively. If we thought of a code as a tuple consisting of multiple fields (bits), HBP and VBP can be viewed as row-oriented storage and column-oriented storage *at the bit level*, respectively. Figure 2 demonstrates the basic idea behind HBP and VBP storage layouts.

Both HBP and VBP only require the following full-word operations, which are common in all modern CPU architectures (including at the SIMD register level in most architectures): logical and ($\wedge$), logical or ($\vee$), exclusive or ($\oplus$), binary addition ($+$), negation ($\neg$), and k-bit left or right shift ($\leftarrow_k$ or $\rightarrow_k$, respectively).



(a) Horizontal Bit-Parallel (b) Vertical Bit-Parallel

Figure 2: HBP and VBP layouts for a column with 3-bit codes. The shaded boxes represent the bits for the first column value.

Since the primary access pattern for scan operations is the sequential access pattern, both the CPU cost and the memory access cost are significant components that contribute to the overall execution time for that operation. Consequently, our methods are optimized for both the number of CPU instructions that are needed to process the data, as well as the number of CPU cache lines that are occupied by the underlying (HBP or VBP) data representations.

3.1.1 Running Example

To illustrate the techniques, we use the following example throughout this section. The data set has 10 tuples, and the column of interest contains the following codes: $\{1 = (001)_2, 5 = (101)_2, 6 = (110)_2, 1 = (001)_2, 6 = (110)_2, 4 = (100)_2, 0 = (000)_2, 7 = (111)_2, 4 = (100)_2, 3 = (011)_2\}$, denoted as $c1$ – $c10$ respectively. Each value can be encoded by 3 bits ($k = 3$). For ease of illustration, we assume 8-bit processor words (i.e. $w = 8$).

3.2 The Horizontal bit-parallel (HBP) method

The HBP method compactly packs codes into processor words, and implements the functionality of hardware-implemented SIMD instructions based on ordinary full-word instructions. The HBP method solves a general problem for hardware-implemented SIMD that the natural bit width of a column often does not match any of the bank widths of the SIMD processor, which leads to an underutilization of the available bit-level parallelism.

We first present the storage layout of HBP in Section 3.2.1, and then describe the algorithm to perform a basic scan on a single column over the proposed storage layout in Section 3.2.2.

3.2.1 Storage layout

In the HBP method, each code is stored in a $(k + 1)$-bit section whose leftmost bit is used as a delimiter between adjacent codes (k denote the number of bits needed to encode a code). A method that does not require the extra delimiter bit is feasible, but is much more complicated than the method with delimiter bits, and also requires executing more instructions per code [11]. Thus, our HBP method uses this extra bit for storage.

HBP tightly packs and pads a group of $(k + 1)$-bit sections into a processor word. Let w denote the width of a processor word. Then, inside the processor word, $\lfloor \frac{w}{k+1} \rfloor$ sections are concatenated together and padded to the right with 0s up to the word boundary.

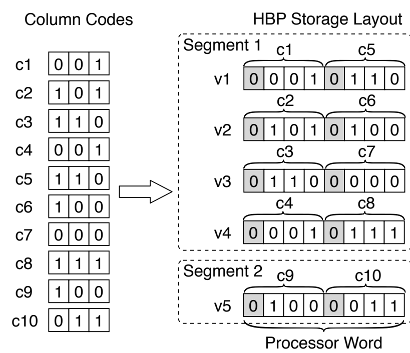


Figure 3: Example of the HBP storage layout ($k = 3, w = 8$). Delimiter bits are marked in gray.

In the HBP method, the codes are organized in a storage layout that simplifies the process of producing a result bit vector with one bit per input code (described below in Section 3.2.2). The column is divided into fixed-length *segments*, each of which contains $(k + 1) \cdot \lfloor \frac{w}{k+1} \rfloor$ codes. Each code represents $k + 1$ bits values, with k bits for the actual code and the leading bit set to the delimiter value of 0. Since a processor word fits $\lfloor \frac{w}{k+1} \rfloor$ codes, a segment occupies $k + 1$ contiguous processor words in memory space. Inside a segment, the layout of the $(k + 1) \cdot \lfloor \frac{w}{k+1} \rfloor$ codes, denoted as $c_1 \sim c_{(k+1)\cdot\lfloor \frac{w}{k+1} \rfloor}$, is shown below. We use v_i to denote the i^{th} processor word in the segment.

$$\begin{array}{lccccc}
v_1: & c_1 & c_{k+2} & c_{2k+3} & \cdots & c_{k\cdot\lfloor \frac{w}{k+1} \rfloor+1} \\
v_2: & c_2 & c_{k+3} & c_{2k+4} & \cdots & c_{k\cdot\lfloor \frac{w}{k+1} \rfloor+2} \\
\vdots & \vdots & \vdots & \vdots & & \vdots \\
v_k: & c_k & c_{2k+1} & c_{3k+2} & \cdots & c_{(k+1)\cdot\lfloor \frac{w}{k+1} \rfloor-1} \\
v_{k+1}: & c_{k+1} & c_{2k+2} & c_{3k+3} & \cdots & c_{(k+1)\cdot\lfloor \frac{w}{k+1} \rfloor}
\end{array}$$

	$v_1(c_1, c_5)$	$v_2(c_2, c_6)$	$v_3(c_3, c_7)$	$v_4(c_4, c_8)$	$v_5(c_9, c_{10})$
$X =$	$(0001\ 0110)_2$	$(0101\ 0100)_2$	$(0110\ 0000)_2$	$(0001\ 0111)_2$	$(0100\ 0011)_2$
$Y =$	$(0101\ 0101)_2$	$(0101\ 0101)_2$	$(0101\ 0101)_2$	$(0101\ 0101)_2$	$(0101\ 0101)_2$
$mask =$	$(0111\ 0111)_2$	$(0111\ 0111)_2$	$(0111\ 0111)_2$	$(0111\ 0111)_2$	$(0111\ 0111)_2$
$X \oplus mask =$	$(0110\ 0001)_2$	$(0010\ 0011)_2$	$(0001\ 0111)_2$	$(0110\ 0000)_2$	$(0011\ 0100)_2$
$Y + (X \oplus mask) =$	$(1011\ 0110)_2$	$(0111\ 1000)_2$	$(0110\ 1100)_2$	$(1011\ 0101)_2$	$(1001\ 1001)_2$
$Z = (Y + (X \oplus mask)) \wedge \neg mask =$	$(1000\ 0000)_2$	$(0000\ 1000)_2$	$(0000\ 1000)_2$	$(1000\ 0000)_2$	$(1000\ 1000)_2$

Figure 4: Evaluating a predicate $c < 5$ on the example column c

Figure 3 demonstrates the storage layout for the example column. Since each code in the example column is encoded by 3 bits ($k = 3$), we use $4 = 3 + 1$ bits to store each code and fit two codes into a 8-bit word ($w = 8$). As shown in the figure, the 10 values are divided into 2 segments. In segment 1, eight codes are packed into four 8-bit words. More specifically, word 1 contains code 1 and 5. Word 2 contains code 2 and 6. Word 3 contains code 3 and 7. Word 4 contains code 4 and 8. Segment 2 is only partially filled, and contains code 9 and code 10 that are packed into word 5.

3.2.2 *Column-scalar scans*

The HBP column-scalar scan compares each code with a constant C, and outputs a bit vector to indicate whether or not the corresponding code satisfies the comparison condition.

In HBP, $\lfloor \frac{w}{k+1} \rfloor$ codes are packed into a processor word. Thus, we first introduce a function $f_\circ(X, C)$ that performs simultaneous comparisons on $\lfloor \frac{w}{k+1} \rfloor$ packed codes in a processor word. The outcome of the function is a vector of $\lfloor \frac{w}{k+1} \rfloor$ results, each of which occupies a $(k + 1)$-bit section. The delimiter (leftmost) bit of each section indicates the comparison results.

Formally, a function $f_\circ(X, C)$ takes as input a comparison operator $\circ$, a comparison constant C, and a processor word X that contains a vector of $\lfloor \frac{w}{k+1} \rfloor$ codes in the form $X = (x_1, x_2, \cdots, x_{\lfloor \frac{w}{k+1} \rfloor})$, and outputs a vector $Z = (z_1, z_2, \cdots, z_{\lfloor \frac{w}{k+1} \rfloor})$, where $z_i = 10^k$ if $x_i \circ C = \texttt{true}$, or $z_i = 0^{k+1}$ if $x_i \circ C = \texttt{false}$. Note that in the notation above for z_i, we use exponentiation to denote bit repetition, e.g. $1^4 0^2 = 111100$, $10^k = 1\underbrace{00\cdots00}_{k}$.

Since the codes are packed into processor words, the ALU instruction set can not be directly used to process these packed codes. In HBP, the functionality of vector processing is implemented using full-word instructions. Let Y denote a vector of $\lfloor \frac{w}{k+1} \rfloor$ instances of constant C, i.e. $Y = (y_1, y_2, \cdots, y_{\lfloor \frac{w}{k+1} \rfloor})$, where $y_i = C$. Then, the task is to calculate the vector Z in parallel, where each (k+1)-bit section in this vector, $z_i = x_i \circ y_i$; here, $\circ$ is one of comparison operators described as follows. Note that most of these functions are adapted from [11].

INEQUALITY ($\neq$). For the INEQUALITY, observe that $x_i \neq y_i$ iff $x_i \oplus y_i \neq 0^{k+1}$. Thus, we know that $x_i \neq y_i$ iff $(x_i \oplus y_i) + 01^k = 1{*}^k$ (we use $*$ to represent an arbitrary bit), which is true iff $((x_i \oplus y_i) + 01^k) \wedge 10^k = 10^k$. We know that $(x_i \oplus y_i) + 01^k$ is always less than 2^{k+1}, so overflow is impossible for each (k+1)-bit section. As a result, these computation can be done simultaneously on all x_i and y_i within a processor word. It is straightforward to see that $Z = ((X \oplus Y) + 01^k 01^k \cdots 01^k) \wedge 10^k 10^k \cdots 10^k$.

EQUALITY ($=$). EQUALITY operator is implemented by the complement of the INEQUALITY operator, i.e. $Z = \neg((X \oplus Y) + 01^k 01^k \cdots 01^k) \wedge 10^k 10^k \cdots 10^k$.

LESS THAN ($<$). Since both x_i and y_i are integers, we know that $x_i < y_i$ iff $x_i \leq y_i - 1$, which is true iff $2^k \leq y_i + 2^k - x_i - 1$. Observe that $2^k - x_i - 1$ is just the k-bit logical complement of x_i, which can be calculated as $x_i \oplus 01^k$. It is then easy to show that $(y_i + (x_i \oplus 01^k)) \wedge 10^k = 10^k$ iff $x_i < y_i$. We also know that $y_i + (x_i \oplus 01^k)$ is always less than 2^{k+1}, so overflow is impossible for each $(k + 1)$-bit section. Thus, we have $Z = (Y + (X \oplus 01^k 01^k \cdots 01^k)) \wedge 10^k 10^k \cdots 10^k$ for the comparison operator $<$.

LESS THAN OR EQUAL TO ($\leq$). Since $x_i \leq y_i$ iff $x_i < y_i + 1$, we have $Z = (Y + (X \oplus 01^k) + 0^k 1) \wedge 10^k 10^k \cdots 10^k$ for the comparison operator $\leq$.

Then, GREATER THAN ($>$) and GREATER THAN OR EQUAL TO ($\geq$) can be implemented by swapping X and Y for LESS THAN ($<$) and LESS THAN OR EQUAL TO ($\leq$) operators, respectively.

Thus, the function $f_\circ(X, C)$ computes the predicates listed above on $\lfloor \frac{w}{k+1} \rfloor$ codes using 3–4 instructions.

Figure 4 illustrates an example when applying $f_<(v_i, 5)$ on the words $v_1 \sim v_5$ shown in Figure 3. The i^{th} column in the figure demonstrates the steps when calculating $Z = f_<(v_i, 5)$ on the word v_i. The last row represents the results of the function. Each result word contains two comparison results. The value $(1000)_2$ indicates that the corresponding code is less than the constant 5, whereas the value $(0000)_2$ indicates that the corresponding code does not satisfy the comparison condition.

Algorithm 1 HBP column-scalar scan

Input: a comparison operator $\circ$
a comparison constant C
Output: BV_{out}: result bit vector

```
for each segment s in column c do
    m_s := 0
    for i := 1 ... k+1 do
        m_w := f_o(s.v_i, C)
        m_s := m_s ∨ →_{i-1} (m_w)
    append m_s to BV_out
return BV_out;
```

Next, we present the HBP column-scalar scan algorithm based on the function $f_\circ(X, C)$. Algorithm 1 shows the pseudocode for the scan method. The basic idea behind this algorithm is to reorganize the comparison results in an appropriate order, matching the order of the original codes. As shown in the algorithm, for each segment in the column, we iterate over the $k + 1$ words. In the inner loop over the $k + 1$ words, we combine the results of $f_\circ(v_1, C) \sim f_\circ(v_{k+1}, C)$ together to obtain the result bit vector on segment s. This procedure is illustrated below:

$f_\circ(v_1, C)$:	$R(c_1)$	0	$\cdots$	0	0	$R(c_{k+2})$	$\cdots$
$\rightarrow_1 (f_\circ(v_2, C))$:	0	$R(c_2)$	$\cdots$	0	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$		$\vdots$			
$\rightarrow_{k-1} (f_\circ(v_k, C))$:	0	0	$\cdots$	$R(c_k)$	0	0	$\cdots$
$\rightarrow_k (f_\circ(v_{k+1}, C))$:	0	0	$\cdots$	0	$R(c_{k+1})$	0	$\cdots$
$\sum_\vee$:	$R(c_1)$	$R(c_2)$	$\cdots$	$R(c_k)$	$R(c_{k+1})$	$R(c_{k+2})$	$\cdots$

In the tabular representation above, each column represents one bit in the outcome of $f_\circ(v_i, C)$. Let $R(c_i)$ denote the binary result of the comparison on $c_i \circ C$. Since $R(c_i)$ is always placed in

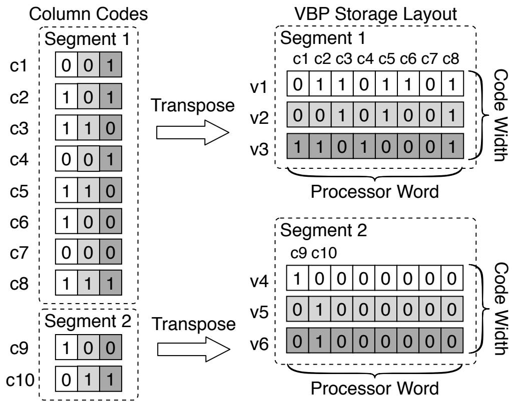


Figure 5: Example of the VBP storage layout. The middle bits of codes are marked in light gray, whereas the least significant bits are marked in dark gray.

the delimiter (leftmost) bit in a $(k+1)$-bit section, the output of $f_\circ(v_i, C)$ is in the form: $R(c_i)0^k R(c_{k+1+i})0^k \cdots$. By right shifting the output of $f_\circ(v_i, C)$, we move the result bits $R(c_i)$ to the appropriate bit positions. The OR ($\vee$) summation over the $k+1$ result words is then in the form of $R(c_1)R(c_2)R(c_3)\cdots$, representing the comparison results on the $\lfloor \frac{w}{k+1} \rfloor$ codes of segment s, in the desired result bit vector format.

For instance, to compute the result bit vector on segment 1 (v_1, v_2, v_3, and v_4) shown in Figure 3 and Figure 4, we perform $(1000\ 0000)_2 \vee \rightarrow_1 (0000\ 1000)_2 \vee \rightarrow_2 (0000\ 1000)_2 \vee \rightarrow_3 (1000\ 0000)_2 = (1001\ 0110)_2$. The result bit vector $(1001\ 0110)_2$ means that c_1, c_4, c_6, and c_7 satisfy the comparison condition.

Note that the steps above, which are carried out to produce a result bit vector with one bit per input code, are essential when using the result bit vector in a subsequent operation (e.g. the next step of a complex predicate evaluation in which the other attributes in the predicate have different code widths).

The HBP storage layout is designed to make it easy to assemble the result bit vector with one bit per input code. Taking Figure 3 as an example again, imagine that we lay out all the codes in sequence, i.e. put $c1$ and $c2$ in $v1$, put $c3$ and $c4$ in $v2$, and so forth. Now, the result words from the predicate evaluation function $f_\circ(v_i, C)$ on $v1, v2, \cdots$ are $f_\circ(v_1, C) = R(c_1)000R(c_2)000$, $f_\circ(v_2, C) = R(c_3)000R(c_4)000$, $\cdots$. Then, these result words must be converted to a bit vector of the form $R(c_1)R(c_2)R(c_3)R(c_4)\cdots$, by extracting all the delimiter bits $R(c_i)$ and omitting all other bits. Unfortunately, this conversion is relatively expensive compared to the computation of the function $f_\circ(v_i, C)$ (See Appendix B in [12] for more details). In contrast, the storage layout used by the HBP method does not need to execute this conversion to produce the result bit vector. In Section 6.1.1, we empirically compare the HBP method with a method that needs this conversion.

3.3 The Vertical bit-parallel (VBP) method

The Vertical Bit-Parallel (VBP) method is like a bit-level column store, with data being packed at word boundaries. VBP is inspired by the bit-sliced method [13], but as described below, is different in the way it organizes data around word boundaries.

3.3.1 Storage layout

In VBP, the column of codes is broken down to fixed-length segments, each of which contains w codes (w is the width of a processor word). The w k-bit codes in a segment are then transposed into

Algorithm 2 VBP column-scalar comparison

Input: a predicate $C1 < c < C2$ on column c
Output: BV_{out}: result bit vector
1: **for** $i := 1 \ldots k$ **do**
2:     **if** i-th bit in C1 is on **then**
3:         $C1_i := 1^w$
4:     **else**
5:         $C1_i := 0^w$
6: **for** $i := 1 \ldots k$ **do**
7:     **if** i-th bit in C2 is on **then**
8:         $C2_i := 1^w$
9:     **else**
10:         $C2_i := 0^w$
11: **for** each segment s in column c **do**
12:     $m_{lt}, m_{gt} := 0$
13:     $m_{eq1}, m_{eq2} := 1^w$
14:     **for** $i := 1 \ldots k$ **do**
15:         $m_{gt} := m_{gt} \vee (m_{eq1} \wedge \neg C1_i \wedge s.v_i)$
16:         $m_{lt} := m_{lt} \vee (m_{eq2} \wedge C2_i \wedge \neg s.v_i)$
17:         $m_{eq1} := m_{eq1} \wedge \neg(s.v_i \oplus C1_i)$
18:         $m_{eq2} := m_{eq2} \wedge \neg(s.v_i \oplus C2_i)$
19:     append $m_{gt} \wedge m_{lt}$ to BV_{out}
20: **return** BV_{out};

k w-bit words, denoted as v_1, v_2, $\cdots$, v_k, such that the j-th bit in v_i equals to the i-th bit in the original code c_j.

Inside a segment, the k words, i.e. v_1, v_2, $\cdots$, v_k, are physically stored in a continuous memory space. The layout of the k words exactly matches the access pattern of column-scalar scans (presented below in Section 3.3.2), which leads to a sequential access pattern on these words, making it amenable for hardware prefetching.

Figure 5 illustrates the VBP storage layout for the running example shown in Section 3.1.1. The ten codes are broken into two segments with eight and two codes, respectively. The two segments are separately transposed into three 8-bit words. The word v_1 in segment 1 holds the most significant (leftmost) bits of the codes $c1 \sim c8$, the word v_2 holds the middle bits of the codes $c1 \sim c8$, and the word v_3 holds the least significant (rightmost) bits of the codes $c1 \sim c8$. In segment 2, only the leftmost two bits of the three words are used, and the remaining bits are filled with zeros.

3.3.2 Column-scalar scans

The VBP column-scalar scan evaluates a comparison condition over all the codes in a single column and outputs a bit vector, where each bit indicates whether or not the corresponding code satisfies the comparison condition.

The VBP column-scalar scan follows the natural way to compare two integers in the form of bit strings: we compare each pair of bits at the same position of the two bit strings, starting from the most significant bits to the least significant bits. The VBP method essentially performs this process on a vector of w codes in parallel, inside each segment.

Algorithm 2 shows the pseudocode to evaluate the comparison predicate `BETWEEN` $C1$ `AND` $C2$.

At the beginning of the algorithm (Lines 1–5), we create a list of words $C1_1 \sim C1_k$ to represent w instances of $C1$ in the VBP storage format. If the i-th bit of $C1$ is 1, $C1_i$ is set to be all 1s, as all the i-th bits of the w instances of $C1$ are all 1s. Otherwise, $C1_i$ is set to be all 0s. Similarly, we create $C2_1 \sim C2_k$ to represent $C2$ in the VBP storage format (in Line 6–10).

In the next step, we iterate over all segments of the column, and simultaneously evaluate the range on all the w codes in each segment. The bit vector m_{gt} is used to indicate the codes such that they are greater than the constant $C1$, i.e. if the i-th bit of m_{gt} is on, then the i-th code in the segment is greater than the constant

$C1$. Likewise, m_{lt} is used to indicate the codes that are less than the constant $C2$. m_{eq1} and m_{eq2} are used to indicate the codes that are equivalent to the constant $C1$ and $C2$, respectively.

In the inner loop (Line 14–18), we compare the codes with the constants $C1$ and $C2$ from the most significant bits to the least significant bits, and update the bit vector m_{gt}, m_{lt}, m_{eq1}, and m_{eq2}, correspondingly. The k words in the segment s are denoted as $s.v_1 \sim s.v_k$. At the i-th bit position, for a code with the i-th bit on, the code must be greater than the constant $C1$ iff the i-th bit of $C1$ is off and all bits to the left of this position between the code and $C1$ are all equal ($m_{eq1} \wedge \neg C1_i \wedge s.v_i$). The corresponding bits in m_{gt} are then updated to be 1s (Line 15). Similarly, m_{lt} is updated if the i-th bit of a code is 0, the i-th bit of $C2$ is 1, and all the bits to the left of this position are all equal (Line 16). We also update m_{eq1} (m_{eq2}) for the codes that are different from the constant $C1(C2)$ at the i-th bit position (Line 17 & 18).

After the inner loop, we perform a logical AND between the bit vector m_{gt} and m_{lt} to obtain the result bit vector on the segment (Line 19). This bit vector is then appended to the result bit vector.

Algorithm 2 can be easily extended for other comparison conditions. For example, we can modify Line 19 to "append $m_{gt} \wedge m_{lt} \vee m_{eq1} \vee m_{eq2}$ to BV_{out}" to evaluate the condition $C1 \leq c \leq C2$. For certain comparison conditions, some steps can be eliminated. For instance, Line 15 and 17 can be skipped for a LESS THAN ($<$) comparison, as we do not need to evaluate m_{gt} and m_{eq1}.

4. EARLY PRUNING

The early pruning technique aims to avoid accesses on unnecessary data at the bit level. This technique is orthogonal to the two bit-parallel methods described in Section 3, and hence can be applied to both the HBP and the VBP methods. However, as the early pruning technique is more naturally described within the context of VBP, we first describe this technique as applied to VBP. Then, in Section 5.2 we discuss how to apply this technique to HBP.

4.1 Basic idea behind early pruning

It is often not necessary to access all the bits of a code to compute the final result. For instance, to compare the code $(110\underline{1}0101)_2$ to a constant $(110\underline{0}1010)_2$, we compare the pair of bits at the same position, starting from the most significant bit to the least significant bit, until we find two bits that are different. At the 4th position (underlined above), the two bits are different, and thus we know that the code is greater than the constant. We can now ignore the remaining bits.

	Constant	VBP words	m_{lt}
1st bit	0	01101101	00000000
2nd bit	1	00101001	10010010
3rd bit	1	11010001	-

Figure 6: Evaluating $c < 3$ with the early pruning technique

It is easy to apply the early pruning technique on VBP, which performs comparisons on a vector of w codes in parallel. Figure 6 illustrates the process of evaluating the eight codes in segment 1 of the example column c with a comparison condition $c < 3$. The constant 3 is represented in the binary form $(011)_2$ as shown in the second column in the figure. The first eight codes ($1 = (001)_2$, $5 = (101)_2$, $6 = (110)_2$, $1 = (001)_2$, $6 = (110)_2$, $4 = (100)_2$, $0 = (000)_2$, $7 = (111)_2$) of column c are stored in three 8-bit VBP words, as shown in the third column in the figure.

By comparing the first bit of the constant (0) with the first bits of the eight codes (01101101), we notice that no code is guaranteed to be less than the constant at this point. Thus, the bit vector m_{lt} is all 0s to reflect this situation. Next, we expand the comparison to the second bit between the constant and the codes. Now, we know that the 1st, 4th, and 7th codes are smaller than the constant because their first two bits are less than the first two bits of the constant (01). We also know that the 2nd, 3rd, 5th, 6th, and 8th codes are greater than the constant, as their first two bits are greater than the first two bits of the constant (01). At this point, all the codes have a definite answer w.r.t. this predicate, and we can terminate the VBP column-scalar scan on this segment. The bit vector m_{lt} is updated to be 10010010, and it is also the final result bit vector.

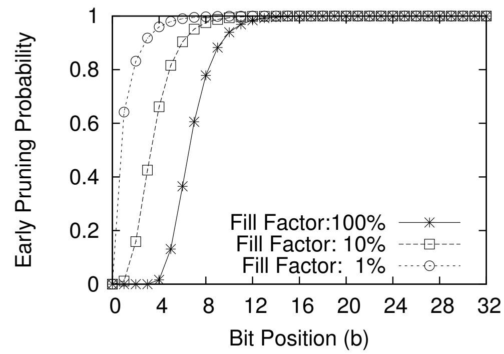


Figure 7: Early Pruning Probability $P(b)$

4.2 Estimating the early pruning probability

We first introduce the *fill factor* f of a segment, defined as the number of codes that are present over the maximum number of codes in the segment, i.e. the width of processor word w. For instance, the fill factor of the segment 1 in Figure 5 is $8/8 = 100\%$, whereas the fill factor of the segment 2 is $2/8 = 25\%$. According to this definition, a segment contains wf codes.

The early pruning probability $P(b)$ is defined as the probability that the wf codes in a segment are all different from the constant in the most significant b bits, i.e. it is the probability that we can terminate the computation at the bit position b.

We analyze the early pruning probability $P(b)$ on a segment containing wf k-bit codes. We assume that a code and the constant have the same value at a certain bit position with a probability of 1/2. Thus, the probability that all of the leading b bits between a code and the constant are identical is given by $(\frac{1}{2})^b$. Since a segment contains wf codes, the probability that these codes are all different from the constant in the leading b bits, i.e. the early pruning probability $P(b)$, is:

$$P(b) = (1 - (\frac{1}{2})^b)^{w \cdot f}$$

Figure 7 plots the early pruning probability $P(b)$ with a 64-bit processor word ($w = 64$) by varying the bit position b. We first look at the curve with a 100% fill factor. The early pruning probability increases as the bit position number increases. At the bit position 12, the early pruning probability is already very close to 100%, which indicates that in many cases we can terminate the scan after looking at the first 12 bits. If a code is a 32-bit integer, VBP with early pruning potentially only uses 12/32 of the memory bandwidth and the processing time that is needed by the base VBP method (without early pruning).

In Figure 7, at the lower fill factors, segments are often "cut-off" early. For example, for segments with fill factor 10%, we can prune the computation at bit position 8 in most (i.e. 97.5%) cases. This cut-off mechanism allows for efficient evaluation of conjunction/disjunction predicates in BitWeaving, as we will see next in Section 4.3.

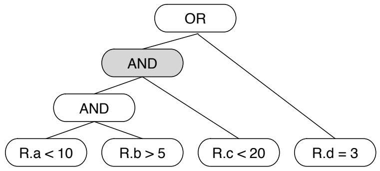


Figure 8: An example predicate tree for the expression `R.a < 10 AND R.b > 5 AND R.c < 20 OR R.d = 3`

4.3 Filter bit vectors on complex predicates

The early pruning technique can also be used when evaluating predicate clauses on multiple columns. Predicate evaluation on a single column can be pruned as outlined above in Section 4.1. But, early pruning can also be used when evaluating a series of predicate clauses with the result vector from the first clause being used to "initialize" the pruning bit vector for the second clause.

The result bit vector that is produced from a previous step is called the *filter bit vector* of the current column-scalar scan. This filter bit vector is used to filter out the tuples that do not match the predicate clauses that were examined in the previous steps, leading to a lower fill factor on the current column. Thus, the filter bit vector further reduces the computation on the current column-scalar scan (note that at the lower fill factors, predicate evaluation are often "cut-off" early, as shown in Figure 7).

As an example, consider the complex predicate: `R.a < 10 AND R.b > 5 AND R.c < 20 OR R.d = 3`. Figure 8 illustrates the predicate tree for this expression. First, we evaluate the predicate clause on column `R.a`, using early pruning. This evaluation produces a result bit vector. Next, we start evaluating the predicate clause on the column `R.b`, using early pruning. However, in this step we use the result bit vector produced from the previous step to seed the early pruning. Thus, tuples that did not match the predicate clause $R.a < 10$ become candidates for early pruning when evaluating the predicate clause on `R.b`, regardless of the value of their `b` column. As a result, the predicate evaluation on column `b` is often "cut-off" even earlier. Similarly, the result bit vector produced at the end of evaluating the AND node (the white AND node in the figure) is fed into the scan on column `R.c`. Finally, since the root node is an OR node, the complement of the result bit vector on the AND node (the gray one) is fed into the final scan on column `R.d`.

5. BIT WEAVING

In this section, we combine the techniques proposed above, and extend them, into the overall method called BitWeaving. BitWeaving comes in two flavors: BitWeaving/H and BitWeaving/V corresponding to the underlying bit-parallel storage format (i.e. HBP or VBP described in Section 3) that it builds on. As described below, BitWeaving/V also employs an adapted form of the early pruning technique described in Section 4.

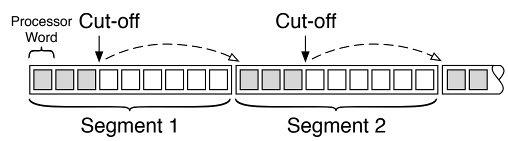
We note that the BitWeaving methods can be used as a base storage organization format in column-oriented data stores, and/or as indices to speedup the scans over some attributes. For ease of presentation, below we assume that BitWeaving is used as a storage format. It is straightforward to employ the BitWeaving method as indices, and in Section 6.2 we empirically evaluate the performance of the BitWeaving methods when used in both these ways.

5.1 BitWeaving/V

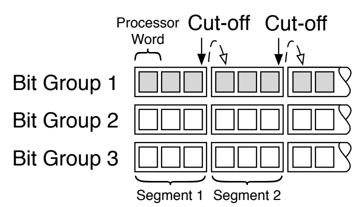
BitWeaving/V is a method that applies the early pruning technique on VBP. BitWeaving/V has three key features: 1) The early pruning technique skips over pruned column data, thereby reducing the total number of bits that are accessed in a column-scalar scan operation; 2) The storage format is not a pure VBP format, but a *weaving* of the VBP format with horizontal packing into *bit groups* to further exploit the benefits of early pruning, by making access to the underlying bits more sequential (and hence more amenable for hardware prefetching); 3) It can be implemented with SIMD instructions allowing it to make full use of the entire width of the (wider) SIMD words in modern processors.

5.1.1 Storage layout

In this section, we describe how the VBP storage layout is adapted in BitWeaving/V to further exploit the benefits of the early pruning technique. In addition to the core VBP technique of vertical partitioning the codes at the bit level, in BitWeaving/V the codes are also partitioned in a horizontal fashion to provide better CPU cache performance when using the early pruning technique. This combination of vertical and horizontal partitioning is the reason why the proposed solution is called BitWeaving.



(a) VBP



(b) BitWeaving/V

Figure 9: Early pruning on VBP and BitWeaving/V

The VBP storage format potentially wastes memory bandwidth if we apply the early pruning technique on the base VBP storage format. (See Section 3.3 for details.) Figure 9(a) illustrates a scan on a column stored in the VBP format. Suppose that with the early pruning technique (described in Section 4.1), the outcome of comparisons on all the codes in a segment is determined after accessing the first three words. Thus, the 4th to the 9th words in segment 1 can be skipped, and the processing can move to the next segment (as shown by the dashed arrow). Suppose that a CPU cache line contains 8 words. Thus, the six words that are skipped occupy the same CPU cache line as the first three words. Skipping over the content that has already been loaded into the CPU cache results in wasted memory bandwidth, which is often a critical resource in main memory data processing environments.

We solve this problem by dividing the k words in a segment into fixed sized *bit groups*. Let B denote the size of each bit group. The words in the same bit group are physically stored in continuous space. Figure 9(b) illustrates how the storage layout with the bit grouping reduces the amount of data that is loaded into the CPU cache. As shown in this figure, the nine words in each segment are divided into three bit groups, each containing three words per segment. Suppose that with the early pruning technique, the outcome of the comparisons on all the codes in a segment is determined after accessing the first three words. In this case, we only need to access the first three words of each segment, which are all laid out continuously and compactly in the first bit group. Consequently, the bit

Algorithm 3 BitWeaving/V column-scalar scan

```
Input: a predicate C1 < c < C2 on column c
       BV_in: filter bit vector
Output: BV_out: result bit vector
 1: initialize C1 and C2 (same as Lines 1-10 in Algorithm 2)
 2: for each segment s in column c do
 3:     m_lt, m_gt := 0
 4:     m_eq1, m_eq2 := BV_in.s                              ◁
 5:     for g := 1 ... ⌊k/B⌋ do                              □
 6:         if m_eq1 == 0 and m_eq2 == 0 then                □
 7:             break                                        □
 8:         for i := gB + 1 ... min(gB + B, k) do            □
 9:             m_gt := m_gt ∨ (m_eq1 ∧ ¬C1_i ∧ s.w_i)
10:             m_lt := m_lt ∨ (m_eq2 ∧ C2_i ∧ ¬s.w_i)
11:             m_eq1 := m_eq1 ∧ ¬(s.w_i ⊕ C1_i)
12:             m_eq2 := m_eq2 ∧ ¬(s.w_i ⊕ C2_i)
13:     append m_gt ∧ m_lt to BV_out
14: return BV_out;
```

grouping technique uses memory bandwidth more judiciously, and results in a more sequential access pattern.

In the example above, if the early pruning triggers at two bits (instead of three bits), then we still save on memory accesses over a method that does not use bit groups. However, we will likely waste memory bandwidth bringing in data for the third bit. Picking an optimal bit group size is an interesting direction for future work. In Appendix D.2 of the full-length version of this paper [12], we empirically demonstrate the impact of the bit group size.

5.1.2 Column-scalar scans

We apply the early pruning technique on the VBP column-scalar scan, in two ways. First, when evaluating a comparison condition on a vector of codes, we skip over the least significant bits as soon as the outcome of the scan is fully determined. Second, a filter bit vector is fed into the scan to further speedup comparisons. This bit vector is used to filter out unmatched tuples even before the scan starts. This technique reduces the number of available codes in each segment, and thus speedups the scan (recall that early pruning technique often runs faster on segments with a lower fill factor as shown in Section 4.2).

The pseudocode for a VBP column-scalar scan with early pruning technique is shown in Algorithm 3. The algorithm is based on the VBP column-scalar scan shown in Algorithm 2. The modified lines are marked with ◁ and □ at the end of lines.

The first modification over the VBP scan method is to skip over the least significant bits once the outcome of the scan is fully determined (marked with □ at the end of lines). In the BitWeaving/V storage layout, k words representing a segment are divided into fixed-size bit groups. Each bit group contains B words in the segment. Predicate evaluation is also broken into a group of small loops to adapt to the design of bit groups. Before working on each bit group, we check the values of the bit masks m_{eq1} and m_{eq2}. If both bit masks are all 0s, then the leading bits between the codes and the constant are all different. Thus, the outcome of the scan on the segment is fully determined. As a result, we terminate the evaluation on this segment, and move to the next one.

We check the cut-off condition (in Line 6) in one of every B iterations of processing the k words of a segment. The purpose of this design is to reduce the cost of checking the condition as well as the cost of CPU branch mispredictions that this step triggers. If the cut-off probability at a bit position is neither close to 0% nor 100%, it is difficult for the CPU to predict the branch. Such branch misprediction can significantly slows down the overall execution. With the early pruning technique, checking the cut-off condition in one of every B iterations reduces the number of checks at the positions where the cut-off probability is in the middle range. We have observed that without this attention to branch prediction in the algorithm, the scans generally run slower by up to 40%.

The second modification is to feed a filter bit vector into the column-scalar comparisons. In a filter bit vector, the bits associated with the filtered codes are turned off. Filter bit vectors are typically the result bit vectors on other predicates in a complex WHERE clause (see Section 4.3 for more details).

To implement this feature, the bit masks m_{eq1} and m_{eq2} are initialized to the corresponding segment in the filter bit vector (marked with ◁ at the end of the line). During the evaluation on a segment, the bit masks m_{eq1} and m_{eq2} are updated by $m_{eq1} := m_{eq1} \wedge \neg(s.w_i \oplus C1_i)$ and $m_{eq2} := m_{eq2} \wedge \neg(s.w_i \oplus C2_i)$, respectively. Thus, the filtered codes remain 0s in m_{eq1} and m_{eq2} during the evaluation. Once the bits associated with the unfiltered codes are all updated to 0s, we terminate the comparisons on this segment following the early pruning technique. The filter bit vector potentially speedups the cut-off process.

5.2 BitWeaving/H

It is also feasible to apply early pruning technique on data stored in the HBP format. The key difference is that we store each bit group in the HBP storage layout (described in Section 3.2). For a column-scalar scan, we evaluate the comparison condition on bit groups starting from the one containing the most significant bits. In addition to the result bit vector on the input comparison condition, we also need to compute a bit vector for the inequality condition in order to detect if the outcome of the scan is fully determined. Once the outcome is fully determined, we skip the remaining bit groups (using early pruning).

However, the effect of early pruning technique on HBP is offset by the high overhead of computing the additional bit vector, and has an overall negative impact on performance (see Section 6.1.2). Therefore, the BitWeaving/H method is simply the HBP method.

5.3 BitWeaving/H and BitWeaving/V

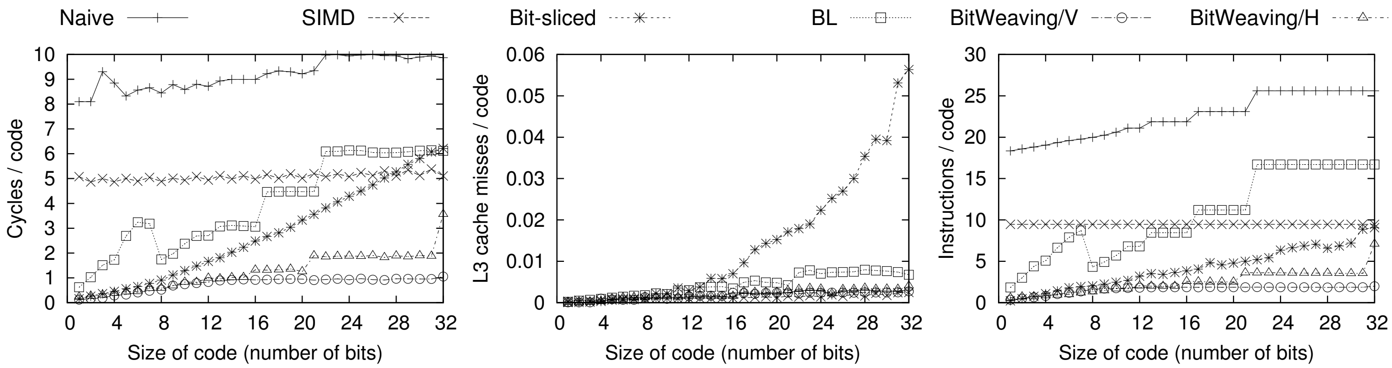
In this section, we compare the two BitWeaving methods, in terms of performance, applicability, as well as ease of implementation. The summary of this comparison is shown in Table 1.

	BitWeaving/H	**BitWeaving/V**
Scan Complexity	$O(\lfloor \frac{n(k+1)}{w} \rfloor)$	$O(\frac{nk}{w})$
SIMD Implementation	Limited	Good
Early Pruning	No	Yes
Lookup Performance	Good	Poor

Table 1: Comparing BitWeaving/H and BitWeaving/V

Scan Complexity. BitWeaving/H uses $k+1$ bits of processor word to store a k-bit code, while BitWeaving/V requires only k bits. As both methods simultaneously process multiple codes, the CPU cost of BitWeaving/H and BitWeaving/V are $O(\lfloor \frac{n(k+1)}{w} \rfloor)$ and $O(\frac{nk}{w})$, respectively; i.e., both are bit-parallel methods as per Definition 1. Both BitWeaving methods are generally competitive to other methods. However, in the extreme cases, BitWeaving/V could be close to 2X faster than BitWeaving/H due to the overhead of the delimiter bits (in BitWeaving/H). For instance, BitWeaving/H fits only one 32-bit code (with an addition delimiter bit) in a 64-bit process word, whereas BitWeaving/V fits two codes.

SIMD Implementation. The implementation of BitWeaving/H method relies on arithmetic and shift operations, which is generally not supported on an entire SIMD word today. Thus, BitWeaving/H



(a) Query Execution Time (b) Memory Performance (c) CPU Performance

Figure 10: Performance on query Q1.

has to pad codes to the width of banks in the SIMD registers, rather than the SIMD word width. This leads to underutilization of the full width of the SIMD registers. In contrast, BitWeaving/V method achieves the full parallelism that is offered by SIMD instructions. Appendix C in [12] describes how our methods can be extended to work with larger SIMD words.

Early Pruning. Applying early pruning technique on HBP requires extra processing that hurts the performance of HBP. As a result, BitWeaving/H does not employ the early pruning technique. In contrast, in BitWeaving/V, the early pruning technique works naturally with the underlying VBP-like format with no extra cost, and usually improves the scan performance.

Lookup Performance. With the BitWeaving/H layout, it is easy to fetch a code as all the bits of the code are stored contiguously. In contrast, for BitWeaving/V, all the bits of a code are spread across various bit groups, distributed over different words. Consequently, looking up a code potentially incurs many CPU cache misses, and can thus hurts performance.

To summarize, in general, both BitWeaving/H and BitWeaving/V are competitive methods. BitWeaving/V outperforms BitWeaving/H for scan performance whereas BitWeaving/H achieves better lookup performance. Empirical evaluation comparing these two methods is presented in the next section.

6. EVALUATION

We ran our experiments on a machine with dual 2.67GHz Intel Xeon 6-core CPUs, and 24GB of DDR3 main memory. Each processor has 12MB of L3 cache shared by all the cores on that processor. The processors support a 64-bit ALU instruction set as well as a 128-bit Intel SIMD instruction set. The operating system is Linux 2.6.9.

In the evaluation below, we compare BitWeaving to the SIMD-scan method proposed in [18], the Bit-sliced method [13], and a method based on Blink [8]. Collectively, these three methods represent the current state-of-the-art main memory scan methods.

To serve as a yardstick, we also include comparison with a naive method that simply extracts, loads, and then evaluates each code with the comparison condition in series, without exploiting any word-level parallelism. In the graphs below the tag *Naive* refers to this simple scan method.

Below, the tag *SIMD-scan* refers to the technique in [18] that uses SIMD instructions to align multiple tightly packed codes to SIMD banks in parallel, and then simultaneously processes multiple codes using SIMD instructions.

Below, the tag *Bit-sliced* refers to the traditional bit-sliced method proposed in [13]. This method was originally proposed to index tables with low number of distinct values; it shares similarities to the VBP method, but does not explore the storage layout and early pruning technique. Surprisingly, previous recent work on main memory scans have largely ignored the bit-sliced method.

In the graphs below, we use the tag *BL* (Blink-Like) to represent the method that adapts the Blink method [8] for column stores (since we focus on column stores for this evaluation). Thus, the tag *BL* refers to tightly (horizontally) packed columns with a simple linear layout, and without the extra bit that is used by HBP (see Section 3.2). The *BL* method differs from the BitWeaving/H method as it does not have the extra bit, and it lays out the codes in order (w.r.t. the discussion in the last paragraph in Section 3.2, the layout of the codes in *BL* is $c1$ and $c2$ in $v1$, $c3$ and $c4$ in $v2$, and so on in Figure 3).

Below, the tags BitWeaving/H (or BW/H) and BitWeaving/V (or BW/V) refer to the methods proposed in this paper. The size of the bit group is 4 for all experiments. The effect of the other bit group sizes on scan performance is shown in Appendix D.2 in [12].

We implemented each method in C++, and compiled the code using g++ 3.4.6 with optimization flags (O3).

In all the results below, we ran experiments using a single process with a single thread. We have also experimented using multiple threads working on independent data partitions. Since the results are similar to that of a single thread (all the methods parallelize well assuming that each thread works on a separate partition), in the interest of space, we omit these results.

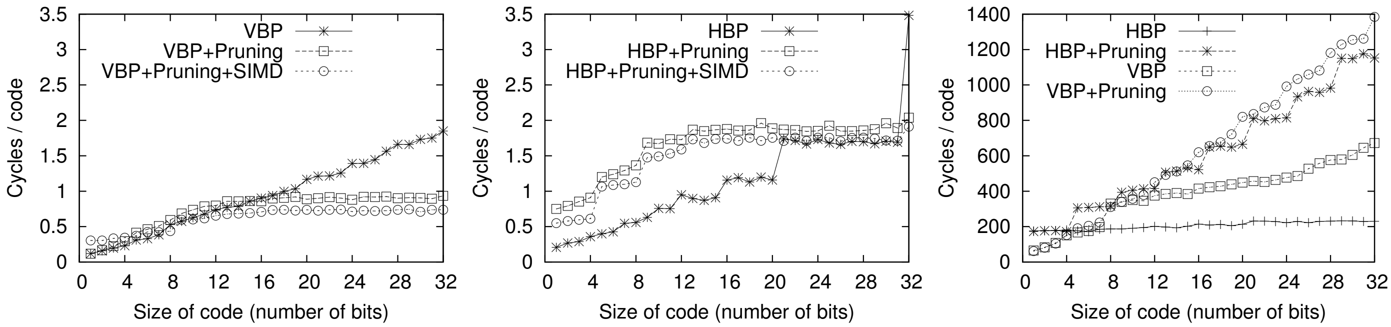
6.1 Micro-Benchmark Evaluation

For this experiment, we created a table R with a single column and one billion uniformly distributed integer values in this column. The domain of the values are $[0, 2^d)$, where d is the width of the column that is varied in the experiments. The query (Q1), shown below, is used to evaluate a column-scalar scan with a simple LESS THAN predicate. The performance on other predicates is similar to that on the LESS THAN predicate. (See Appendix D.1 in [12] for more details.) The constants in the WHERE clause are used to control the selectivity. By default, the selectivity on each predicate is set to 10%, i.e. 10% of the input tuples match the predicate. Note, we also evaluate the impact of different selectivity (see Appendix D.3 in [12]), but by default use a value of 10%.

```
Q1:  SELECT COUNT(*) FROM R WHERE R.a < C1
```

6.1.1 *BitWeaving v/s the Other Methods*

In the evaluation below, we first compare BitWeaving to the



(a) VBP-related methods: Execution Time (b) HBP-related methods: Execution Time (c) Lookup Performance

Figure 11: Performance comparison between the HBP and the VBP related methods on Query Q1.

Naive, the SIMD-scan [18], the Bit-sliced [13], and the BL methods. Figure 10(a), Figure 10(b), and Figure 10(c) illustrate the number of cycles, cache misses, and CPU instructions for the six methods for Q1 respectively, when varying the width of the column code from 1 bit to 32 bits. The total number of cycles for the query is measured by using the RDTSC instruction. We divide this total number of cycles by the number of codes to compute the cycles per code, which is shown in Figure 10(a).

As can be observed in Figure 10(a), not surprisingly, the Naive method is the slowest. The Naive method shifts and applies a mask to extract and align each packed code to the processor word. Since each code is much smaller than a processor word (64-bit), it burns many more instructions than the other methods (see Figure 10(c)) on every word of data that is fetched from the underlying memory subsystem (with L1/L2/L3 caches buffering data fetched from main memory). Even when most of the data is served from the L1 cache, its CPU cost dominates the overall query execution time.

The SIMD-scan achieves 50%–75% performance improvement over the Naive method (see Figure 10(a)), but it is still worse compared to the other methods. Even though a SIMD instruction can process four 32-bit banks in parallel, the number of instructions drops by only 2.2-2.6X (over the Naive method), because it imposes extra instructions to align packed codes into the four banks before any computation can be run on that data. Furthermore, we observe that with SIMD instructions, the CPI (Cycles Per Instructions) increases from 0.37 to 0.56 (see Figure 10(c)), which means that a single SIMD instructions takes more cycles to executed than a ordinary ALU instruction. This effect further dampens the benefit of this SIMD implementation.

As can be seen in Figure 10(a), the Bit-sliced and the BL methods shows a near linear increase in run time as the code width increases. Surprisingly, both these methods are almost uniformly faster than the SIMD-scan method. However, the storage layout of the Bit-sliced method occupies many CPU cache lines for wider codes. As a result, as can be seen in Figure 10(b), the number of L3 cache misses quickly increases and hinders overall performance.

In this experiment, the BitWeaving methods *outperform all the other methods across all the code widths* (see Figure 10(a)). Unlike the Naive and the SIMD-scan methods, they do not need to move data to appropriate positions before the predicate evaluation computation. In addition, as shown in Figure 10(b) and 10(c), the BitWeaving methods are optimized for both cache misses and instructions due to their storage layouts and scan algorithms. Finally, with the early pruning technique, the execution time of BitWeaving/V (see Figure 10(a)) does not increase for codes that are wider than 12 bits. As can be seen in Figure 10(a), for codes wider than 12 bits, both BitWeaving methods are often more than 3X faster than the SIMD-scan, the Bit-sliced and the BL methods.

6.1.2 *Individual BitWeaving Components*

In this experiment, we compare the effect of the various techniques (VBP v/s HBP, early pruning, and SIMD optimizations) that have been proposed in this paper. Figure 11(a) and 11(b) plot the performance of these techniques for VBP and HBP for query Q1, respectively.

First, we compare the scan performance of the HBP and the VBP methods for query Q1. From the results shown in Figure 11(a) and 11(b), we observe that at certain points, VBP is up to 2X faster than HBP. For example, VBP is 2X faster than HBP for 32-bit codes, because HBP has to pad 32-bit code to a entire 64-bit word to fit both the code and the delimiter bit. In spite of this, HBP and VBP generally show a similar performance trend as the code width increases. This empirical results matches our analysis that both methods satisfy the cost-bound for bit-parallel methods.

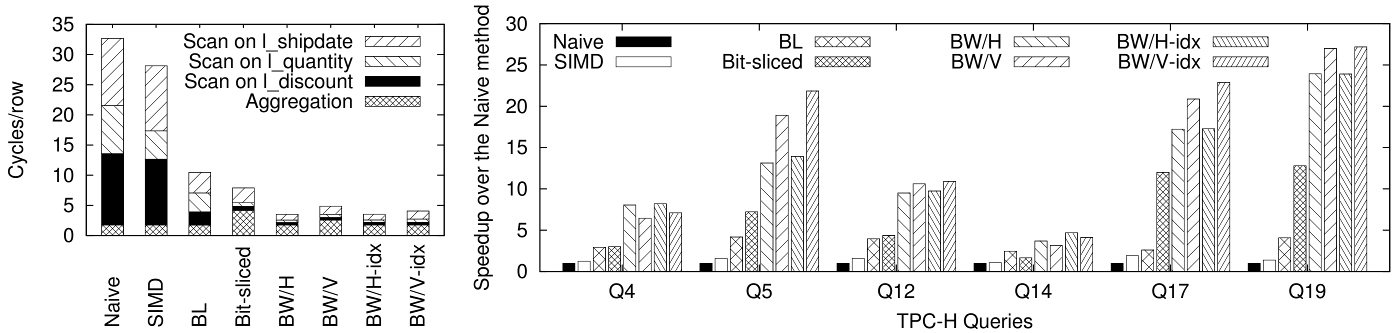
Next, we examine the effects of the early pruning technique on both the VBP and the HBP methods. As can be seen in Figure 11(a), for wider codes, the early pruning technique quickly reduces the query execution time for VBP, and beyond 12 bits, the query execution time with early pruning is nearly constant. Essentially, as described in Section 4.2, for wider codes early pruning has a high chance of terminating after examining the first few bits.

In contrast, as can be seen in Figure 11(b), the effect of the early pruning technique on the HBP method is offset by the high overhead of computing the additional masks (see Section 5.2). Consequently, the HBP method (which is the same as BitWeaving/H, as discussed in Section 5.2) is uniformly faster than "HBP + Pruning".

Applying SIMD parallelism achieves marginal speedups for both the HBP and the VBP methods (see Figures 11(a) and 11(b)). Ideally, the implementation with a 128-bit SIMD word should be 2X faster than that with 64-bit ALU word. However, by measuring the number of instructions, we observed that the SIMD implementation reduces the number of instructions by 40%, but also increase the CPI by 1.5X. Consequently, the net effect is that the SIMD implementation is only 20% and 10% faster than the ALU implementation, for VBP and HBP respectively.

Next, we evaluate the performance of a lookup operation. A lookup operation is important to produce the attributes in the projection list after the predicates in the WHERE clause have been applied. In this experiment, we randomly pick 10 million positions in the column, and measure the average number of cycles that are needed to fetch (and assemble) a code at each position. The results for this experiment are shown in Figure 11(c).

As can be seen in Figure 11(c), amongst the four methods, the



(a) Q6: Execution Time Breakdown

(b) Performance on TPC-H Queries with Materialized Join Inputs

Figure 12: Performance comparison with the TPC-H Queries (BW=BitWeaving).

lookup performance of the HBP method is the best, and its performance is stable across the code widths. The reason for this behavior is because all the bits of a code in the HBP method are located together. For the VBP method, all the bits of a code are stored in continuous space, and thus it is relatively fast to access all the bits and assemble the code. For the methods with the early pruning technique, the bits of a code are distributed into various bit groups. Assembling a code requires access to data across multiple bit groups at different locations, which incurs many CPU cache misses, and thus significantly hurts the lookup performance.

6.2 TPC-H Evaluation

In this experiment, we use seven queries from the TPC-H benchmark [17]. These experiments were run against a TPC-H dataset at scale factor 10. The total size of the database is approximately 10GB. First, we compare the performance of the various methods on the TPC-H scan query (Q6). This query is shown below:

```
SELECT sum(l_extendedprice * l_discount)
FROM lineitem
WHERE l_shipdate BETWEEN Date and Date + 1 year
 and l_discount BETWEEN Discount - 0.01
 and Discount + 0.01 and l_quantity < Quantity
```

As per the TPC-H specifications for the domain size for each of the columns/attributes in this query, the column `l_shipdate`, `l_discount`, `l_quantity`, `l_extendedprice` are encoded with 12 bits, 4 bits, 6 bits, and 24 bits, respectively. The selectivity of this query is approximately 2%.

Figure 12(a) shows the time breakdown for the scan and the aggregation operations for the BitWeaving and the other methods. Not surprisingly, the Naive method is the slowest. The SIMD-scan method only achieves about 20% performance improvement over the Naive method, mainly because the SIMD-scan method performs relatively poorly when evaluating the BETWEEN predicates (see Appendix D.1 in [12]). Evaluating a BETWEEN predicate is complicated/expensive with the SIMD-scan method since the results of the SIMD computations are always stored in the original input registers. Consequently, we have to make two copies for each attribute value, and compare each copy with the lower and upper bound constants in the BETWEEN predicate, respectively.

The BL method runs at a much higher speed compared to the Naive and the SIMD methods. However, compared to BitWeaving/H, the BL method uses more instructions to implement its functionality of parallel processing on packed data and the conversion process to produce the result bit vector, which hinders its scan performance.

Note that both the BitWeaving methods (BW/H and BW/V) outperform all existing methods. As the column `l_extendedprice` is fairly wide (24 bits), BitWeaving/V spends more cycles extracting the matching values from the aggregation columns. As a result, for this particular query, BitWeaving/H is faster than BitWeaving/V.

We also evaluated the effects of using the BitWeaving methods as indices. In this method, the entire WHERE clause is evaluated using the corresponding BitWeaving methods on the columns of interest for this WHERE clause. Then, using the method described in Appendix A in [12] , the columns involved in the aggregation (in the SELECT clause of the query) are fetched from the associated column store(s) for these attributes. These column stores use a Naive storage organization.

In Figure 12, these BitWeaving index-based methods are denoted as BW/H-idx and BW/V-idx. As can be seen in Figure 12(a), BW/H-idx and BW/H have similar performance. The key difference between these methods is whether the aggregation columns are accessed from either the BW/H format or from the Naive column store. However, since using BW/H always results in accessing one cache line per lookup, its performance is similar to the lookup with the Naive column store organization (i.e the BW/H-idx case). On the other hand, the BW/V-idx method is about 30% faster than the BW/V method. The reason for this behavior is that the vertical bit layout in BW/V results in looking up data across multiple cache lines for each aggregate column value, whereas the BW/V-idx method fetches these attribute values from the Naive column store, which requires accessing only one cache line for each aggregate column value.

Next, we selected six TPC-H join queries (Q4, Q5, Q12, Q14, Q17, Q19), and materialized the join component in these queries. Then, we ran scan operations on the pre-joined materialized tables. Here, we report the results of these scan operations on these materialized tables. The widths of the columns involved in the selection operations (i.e. the WHERE clause in the SQL query on the pre-joined materialized tables) ranges from 2 bits to 12 bits. All these queries, except for query Q19, contain a predicate clause that is a conjunction of one to four predicates. Query Q19 has a more complex predicate clause, which includes a disjunction of three predicate clauses, each of which is a conjunction of six predicates. These queries contain a variety of predicates, including $<, >, =, <>$, BETWEEN, and IN. Some queries also involve predicates that perform comparisons between two columns. The projection clauses of these six queries contain one to three columns with widths that vary from 3 to 24 bits.

Figure 12(b) plots the speedup of all the methods over the Naive method for the six TPC-H queries. For most queries, the BitWeaving methods are over one order of magnitude faster than the Naive method.

By comparing the performance of the BW/H and the BW/V methods, we observe that the answer to the question of which BitWeaving method has higher performance depends on many query characteristics. For Q4 and Q14, the BW/H method is slightly faster than the BW/V method, because the BW/H method performs better on the BETWEEN predicate on relative narrow columns (see Appendix D.1 in [12]). Query Q4 contains two predicate clauses, one of which is a BETWEEN predicate. In contrast, Query Q14 contains only one predicate clause, which is a BETWEEN predicate. For queries Q5, Q12, Q17, and Q19, the BW/V method outperforms the BW/H method as these four queries contain more than three predicate clauses. Although some of these queries also contain the BETWEEN predicate(s), the early pruning technique (of BW/V) speedups the scans with on BETWEEN predicates when performed at the end of a series of column-scalar scans. In general, we observe that the BW/V method has higher performance for queries with predicates that involve many columns, involve wider columns, and have highly selective predicates.

Using the BW/V method as an index improves the performance by about 15% for Q5 and Q17 (both queries contain wider columns in their projection lists), and has no significant gain for the other queries. In general, we observe that it is not very productive to use the BW/H method as an index, since it already has a low lookup cost as a base storage format. For the BW/V method, using it as an index improves the performance for some queries by avoiding the slow lookups that are associated with using the BW/V method as the base storage format.

We note that there are interesting issues here in terms of how to pick between BitWeaving/H vs. BitWeaving/V, and whether to use the BitWeaving methods as an index of for base storage. Building an accurate cost model that can guide these choices based on workload characteristics is an interesting direction for future work.

7. RELATED WORK

The techniques present in this paper are applicable to main-memory analytics DBMSs. In such DBMSs, data is often stored in compressed form. SAP HANA [5], IBM Blink [2,15], and HYRISE [10] use sorted dictionaries to encode values. Dynamic order-preserving dictionary was proposed to encode strings [3]. Other light-weight compression schemes can also be used for main-memory column-wise databases, such as [1, 4, 20].

SIMD instructions can be used to speed up database operations [19], and the SIMD-scan [18] method is the state-of-the-art scan method that uses SIMD. In this paper we compare BitWeaving with this scan method. We note that BitWeaving can also be used in processing environments that don't support SIMD instructions.

The BitWeaving/V methods shares similarity to the bit-sliced index [13], but the storage layout of a bit-sliced index is not optimized for memory access, as well as our proposed early pruning technique. A follow-up work presented the algorithms that perform arithmetic on bit-sliced indices [16]. Some techniques described in that paper are also applicable to our BitWeaving/V method.

The BitWeaving/H method relies on the capability to process packed data in parallel. This technique was first proposed by Lamport [11]. Recently, a similar technique [8] was used to evaluate complex predicates in IBM's Blink System [2].

8. CONCLUSIONS AND FUTURE WORK

With the increasing demand for main memory analytics data processing, there is an critical need for fast scan primitives. This paper proposes a method called BitWeaving that addresses this need by exploiting the parallelism available at the bit level in modern processors. The two flavors of BitWeaving are optimized for two common access patterns, and both methods match the complexity bound for bit-parallel scans. Our experimental studies show that the BitWeaving techniques are faster than the state-of-the-art scan methods, and in some cases by over an order of magnitude. For future work, we plan to explore methods to use BitWeaving effectively in other database operations, such as joins and aggregates. We also plan to study how to apply the BitWeaving technique within the context of broader automatic physical database design issues (e.g. replication, and forming column groups [8] automatically), multi-threaded scans, concurrent scans, other compression schemes, and considering the impact of BitWeaving on query optimization.

Acknowledgments

We would like to thank Jeffrey Naughton, Karu Sankaralingam, Craig Chasseur, Qiang Zeng, and the reviewers of this paper for their insightful feedback on an earlier draft of this paper. This work was supported in part by the National Science Foundation under grants IIS-1110948 and IIS-1250886, and a gift donation from Google.

9. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) Blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[3] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD Conference*, pages 283–296, 2009.

[4] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 3(1):670–680, 2010.

[5] 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.*, 35(1):28–33, 2012.

[6] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. Hyrise - a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[7] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[8] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[9] A. Kemper, T. Neumann, F. Funke, V. Leis, and H. Mühe. Hyper: Adapting columnar main-memory data management for transactional and query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[10] J. Krüger, 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*, 5(1):61–72, 2011.

[11] L. Lamport. Multiple byte processing with full-word instructions. *Commun. ACM*, 18(8):471–475, 1975.

[12] Y. Li and J. M. Patel. BitWeaving: Fast scans for main memory data processing. http://research.cs.wisc.edu/quickstep/pubs/bitweaving-extended.pdf.

[13] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[14] *Oracle Exalytics In-Memory Machine: A Brief Introduction*, October 2011. Oracle White Paper.

[15] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[16] D. Rinfret, P. E. O'Neil, and E. J. O'Neil. Bit-sliced index arithmetic. In *SIGMOD*, pages 47–57, 2001.

[17] Transaction Processing Performance Council. *TPC Benchmark H. Revision 2.14.3*. November 2011.

[18] 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*, 2(1):385–394, 2009.

[19] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[20] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, page 59, 2006.

[21] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, pages 1349–1350, 2012.