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].

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD’13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

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 staggersthe 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 operations, 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 \((\land)\), logical or \((\lor)\), exclusive or \((\oplus)\), binary addition \((+\)\), negation \((-\)\), and \(k\)-bit left or right shift \((\leftarrow_k\) or \(\rightarrow_k\), respectively.

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

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 \(c_1 \text{–} c_{10}\) 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 \((k\) denotes 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, \(\left\lfloor \frac{w}{k+1} \right\rfloor\) 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.](image)

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 \((k + 1) \cdot \left\lfloor \frac{w}{k+1} \right\rfloor\) 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 \(\left\lfloor \frac{w}{k+1} \right\rfloor\) codes, a segment occupies \(k + 1\) contiguous processor words in memory space. Inside a segment, the layout of the \((k + 1) \cdot \left\lfloor \frac{w}{k+1} \right\rfloor\) codes, denoted as \(c_1 \sim c_{(k+1) \cdot \left\lfloor \frac{w}{k+1} \right\rfloor}\), is shown below. We use \(v_i\) to denote the \(i^{th}\) processor word in the segment.

\[
\begin{align*}
\forall i : \quad & c_1 \quad c_2 k+2 \quad c_2 k+3 \quad \cdots \quad c_{k+1} k+1+
\forall i : \quad & c_2 k+3 \quad c_2 k+4 \quad \cdots \quad c_{k+2} k+2
\vdots : \quad & \vdots \quad \vdots \quad \vdots 
\forall i : \quad & c_{k+1} k+1 \quad c_{k+2} k+2 \quad \cdots \quad c_{(k+1) \cdot \left\lfloor \frac{w}{k+1} \right\rfloor} k+1-
\end{align*}
\]
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 two 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, and 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 \( HBP \), \( \lfloor \frac{x}{2^k} \rfloor \) codes are packed into a processor word. Thus, we first introduce a function \( f_o(X, C) \) that performs simultaneous comparisons on \( \lfloor \frac{x}{2^k} \rfloor \) packed codes in a processor word. The outcome of the function is a vector of \( \lfloor \frac{x}{2^k} \rfloor \) results, each of which occupies a \((k+1)-\)bit section. The delimiter (leftmost) bit of each section indicates the comparison result.

Formally, a function \( f_o(X, C) \) takes as input a comparison operator \( o \), a comparison constant \( C \), and a processor word \( X \) that contains a vector of \( \lfloor \frac{x}{2^k} \rfloor \) codes in the form \( X = (x_1, x_2, \ldots, x_{\lfloor \frac{x}{2^k} \rfloor}) \), and outputs a vector \( Z = (z_1, z_2, \ldots, z_{\lfloor \frac{x}{2^k} \rfloor}) \), where \( z_i \) is 1 if \( x_i \circ C = \text{true} \), or \( z_i = 0 \) otherwise if \( x_i \circ C = \text{false} \). Note that in the notation above for \( z_i \), we use exponentiation to denote bit repetition, e.g. \( 1^{100} = 111100 \), \( 1^{10} = 100 \).

Since the codes are packed into processor words, the ALU instruction set cannot 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 \( \lfloor \frac{x}{2^k} \rfloor \) instances of constant \( C \), i.e. \( Y = (y_1, y_2, \ldots, y_{\lfloor \frac{x}{2^k} \rfloor}) \), 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 \( x_i \neq y_i \) if \( x_i \neq y_i \neq 0 \). Thus, we know that \( x_i \neq y_i \) if \( x_i \neq y_i + 01^k \) (we use * to represent an arbitrary bit), which is true iff \((x_i + y_i) + 01^k \) \( \land 0^k = 1^k \). We know that \((x_i + y_i) + 01^k \) is always less than \( 2^k+1 \), so overflow is impossible for each \((k+1)-\)bit section. As a result, these computations can be done simultaneously on all \( x_i \), \( y_i \) within a processor word. It is straightforward to see that \( Z = ((X \oplus Y) + 01^k 0^k \cdot \cdot \cdot 01^k) \land 0^{k+1} \cdot \cdot \cdot 10^{k} \).

- **EQUALITY \(=\)**. Equality operator is implemented by the complement of the INEQUALITY operator, i.e. \( Z = \neg((X \oplus Y) + 01^k 0^k \cdot \cdot \cdot 01^k) \land 0^{k+1} \cdot \cdot \cdot 10^{k} \).

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

**Algorithm 1 HBP column-scalar scan**

**Input**: a comparison operator \( o \), a comparison constant \( C \)

**Output**: \( BV_{out} \): result bit vector

1. for each segment \( s \) in column \( c \) do
   2. \( m_s := 0 \)
   3. for \( i = 1 \ldots k + 1 \) do
      4. \( m_w := f_o(s.v_i, C) \)
      5. \( m_z := m_s \land \neg m_s \land m_w \)
   6. append \( m_z \) to \( BV_{out} \)
   7. return \( BV_{out} \)

Next, we present the HBP column-scalar scan algorithm based on the function \( f_o(v_i, C) \). 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 \( f_o(v_i, C) \) to \( f_o(v_{k+1}, C) \) together to obtain the result bit vector on segment \( s \). This procedure is illustrated below:

\[
\begin{align*}
&f_o(v_i, C) = R(c_i) \quad 0 \quad \cdots \quad 0 \quad 0 \quad R(c_{i+1}) \quad \cdots \\
&\land \quad (f_o(v_{k+1}, C)) = 0 \quad 0 \quad \cdots \quad 0 \quad R(c_{k+1}) \quad 0 \quad \cdots \\
&\sum_{c} c_i = R(c) \quad \cdots \quad R(c_{k+1}) \quad R(c_{k+1}) \quad \cdots \\
\end{align*}
\]

In the tabular representation above, each column represents one bit in the outcome of \( f_o(v_i, C) \). Let \( R(c_i) \) denote the binary result of the comparison on \( c_i \). Since \( R(c_i) \) is always placed in
the delimiter (leftmost) bit in a \((k + 1)\)-bit section, the output of 
\(f_c(v_i, C)\) is in the form: 
\[ R(c_i)0^k R(c_{k+i+1})0^k \ldots \] 
By right shifting the output of \(f_c(v_i, C)\), we move the result 
bits \(R(c_i)\) to the appropriate bit positions. The \(OR\) (v) summation over the \(k + 1\) 
result words is then in the form of 
\[ R(c_{1})R(c_{2})R(c_{3})\ldots \] 
representing the comparison results on the \([n+1]\) codes of segment \(s\), 
in the desired result bit vector format.

For instance, to compute the result bit vector on segment \(1\) 
\((v_1, v_2, v_3, \text{and } v_4)\) shown in Figure 3 and Figure 4, we perform 
\[ (1000 \ 0000)_2 \to v \to (0000 \ 1000)_2 \to v \to (0000 \ 1000)_2 \to v \to 3 \] 
\[ (1001 \ 0110)_2 \] 
The result bit vector \((1001 \ 0110)_2\) means that \(c_1, c_4, c_6, c_9\) 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 VBP 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 \(c_1\) and \(c_2\) in \(v_1\), put \(c_3\) and \(c_4\) in \(v_2\), and so forth. Now, 
the result words from the predicate evaluation function 
\(f_c(v_i, C)\) on \(v_1, v_2, \ldots\) are \(f_c(v_1, C) = R(c_1)000R(c_2)000, \) 
\(f_c(v_2, C) = R(c_3)000R(c_4)000, \ldots\). Then, these result words 
must be converted to a bit vector of the form 
\(R(c_{1})R(c_{2})R(c_{3})R(c_{4})\ldots\) 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 \(f_c(v_i, C)\) (See Appendix B in [12] 
for more details). In contrast, the storage layout used by the VBP 
method does not need to execute this conversion to produce the 
result bit vector. In Section 6.1.1, we empirically compare the VBP 
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

\[ k \] 
\[ w \] 
\[ \text{bits words, denoted as } v_1, v_2, \ldots, v_k, \text{ such that the } j\text{-th bit in } v_i \text{ equals to the } i\text{-th bit in the original code } c_j. \]

Inside a segment, the \(k\) words, i.e. \(v_1, v_2, \ldots, v_k\), are physically 
stored in a continuous memory space. The layout of the \(k\) words 
effectively 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 
\(v_1, v_2, \ldots, v_k\) in segment 1 holds the most significant (leftmost) bits of the 
codes \(c_1 \sim c_8\), the word \(v_2\) holds the middle bits of the codes \(c_1 \sim c_8\), and 
the word \(v_3\) holds the least significant (rightmost) bits of the 
codes \(c_1 \sim c_8\). 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 
basically performs this process on a vector of \(w\) codes in parallel, 
inside each segment.

Algorithm 2 shows the pseudocode to evaluate the comparison 
predicate \(C1 < c < C2\) on column \(c\).

\[
\text{Output: } BV_{out}: \text{ bit vector }
\]

\begin{align*}
1: & \text{for } i := 1 \ldots k \text{ do } \\
2: & \text{if } i\text{-th bit in } C1 \text{ is on then } \\
3: & C1_t := 1 \\
4: & \text{else } \\
5: & C1_t := 0^w \\
6: & \text{for } i := 1 \ldots k \text{ do } \\
7: & \text{if } i\text{-th bit in } C2 \text{ is on then } \\
8: & C2_{t} := 1 \\
9: & \text{else } \\
10: & C2_{t} := 0^w \\
11: & \text{for each segment } s \text{ in column } c \text{ do } \\
12: & m_{it}, m_{gt} := 0 \\
13: & m_{eq1}, m_{eq2} := 1^w \\
14: & \text{for } i := 1 \ldots k \text{ do } \\
15: & m_{gt} := m_{gt} \lor (m_{eq1} \land \sim C1_t \land s.v_i) \\
16: & m_{it} := m_{it} \lor (m_{eq2} \land C2_t \land s.v_i) \\
17: & m_{eq1} := m_{eq1} \land \sim (s.v_i \oplus C1_t) \\
18: & m_{eq2} := m_{eq2} \land (s.v_i \oplus C2_t) \\
19: & \text{append } m_{gt} \land m_{it} \text{ to } BV_{out} \\
20: & \text{return } BV_{out};
\end{align*}

Algorithm 2 VBP column-scalar comparison

- Input: a predicate \(C1 < c < C2\) on column \(c\)
- Output: \(BV_{out}\): bit vector

\[
\text{for } i := 1 \ldots k \text{ do } \\
\text{if } i\text{-th bit in } C1 \text{ is on then } \\
C1_t := 1 \\
\text{else } \\
C1_t := 0^w \\
\text{for } i := 1 \ldots k \text{ do } \\
\text{if } i\text{-th bit in } C2 \text{ is on then } \\
C2_{t} := 1 \\
\text{else } \\
C2_{t} := 0^w \\
\text{for each segment } s \text{ in column } c \text{ do } \\
m_{it}, m_{gt} := 0 \\
m_{eq1}, m_{eq2} := 1^w \\
\text{for } i := 1 \ldots k \text{ do } \\
m_{gt} := m_{gt} \lor (m_{eq1} \land \sim C1_t \land s.v_i) \\
m_{it} := m_{it} \lor (m_{eq2} \land C2_t \land s.v_i) \\
m_{eq1} := m_{eq1} \land \sim (s.v_i \oplus C1_t) \\
m_{eq2} := m_{eq2} \land (s.v_i \oplus C2_t) \\
\text{append } m_{gt} \land m_{it} \text{ to } BV_{out} \\
\text{return } BV_{out};
\]
Early pruning probability $P(b)$ is used to indicate the codes that are less than the constant $C_2$. $m_{eq1}$ and $m_{eq2}$ are used to indicate the codes that are equivalent to the constant $C_1$ and $C_2$, respectively.

In the inner loop (Line 14–18), we compare the codes with the constants $C_1$ and $C_2$ 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 $s.v_k ∼ s.v_1$. At the $i$-th bit position, for a code with the $i$-th bit on, the code must be greater than the constant $C_1$ if the $i$-th bit of $C_1$ is off and all bits to the left of this position between the code and $C_1$ are all equal ($m_{eq1} \land \neg C_1 \land s.v_i$). 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 $C_2$ 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 $C_1$($C_2$) 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 $m_{gt} \land m_{lt} \lor m_{eq1} \lor m_{eq2}$ to $BV_{out}$” to evaluate the condition $C_1 \leq c \leq C_2$. 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 (11010110) to a constant (11001010), 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.

\[
\begin{array}{ccc}
\text{Constant} & \text{VBP words} & m_{lt} \\
1st bit & 0 & 01101101 00000000 \\
2nd bit & 1 & 00101001 10010010 \\
3rd bit & 1 & 11010001 - \\
\end{array}
\]

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 $s$ 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.

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 $5/8 = 100\%$, whereas the fill factor of the segment 2 is $2/8 = 25\%$. According to this definition, a segment contains $w f$ codes.

The early pruning probability $P(b)$ is defined as the probability that the $w f$ 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 $w f$ $b$-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 $(1/2)^b$. Since a segment contains $w f$ 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:

\[
P(b) = (1 - (1/2)^b)^{w f}
\]

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.
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: \( R.a < 10 \) \( \text{AND} \ R.b > 5 \) \( \text{AND} \ R.c < 20 \) \( \text{OR} \ R.d = 3 \). 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 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 WEA VING

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.
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 $\odot$ at the end of the line). During the evaluation on a segment, the bit masks $m_{eq1}$ and $m_{eq2}$ are updated by $m_{eq1} := m_{eq1} \land \neg(s.w_i \oplus C_1)$ and $m_{eq2} := m_{eq2} \land \neg(s.w_i \oplus C_2)$, 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.

<table>
<thead>
<tr>
<th></th>
<th>BitWeaving/H</th>
<th>BitWeaving/V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan Complexity</td>
<td>$O(\frac{n(k+1)}{w})$</td>
<td>$O(\frac{n}{w})$</td>
</tr>
<tr>
<td>SIMD Implementation</td>
<td>Limited</td>
<td>Good</td>
</tr>
<tr>
<td>Early Pruning</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Lookup Performance</td>
<td>Good</td>
<td>Poor</td>
</tr>
</tbody>
</table>

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 $O(\frac{n(k+1)}{w})$ and $O(\frac{n}{w})$, 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...
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 [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 $c_1$ and $c_2$ in $r_1$, $c_3$ and $c_4$ in $r_2$, 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 vs the Other Methods**

In the evaluation below, we first compare BitWeaving to the
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 instruction takes more cycles to execute than a normal ALU instruction. This effect further damps the benefit of this SIMD implementation.

As can be seen in Figure 10(a), the Bit-sliced and the BL methods show 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
look up 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 stored in continuous space, and thus it is relatively fast to access all the bits and assemble the code. For the VBP method, the all 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_discoun 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 meth-
ods, 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.