Efficient Lightweight Compression Alongside Fast Scans

Orestis Polychroniou
Kenneth A. Ross

Columbia University
Columbia University
IN THE CITY OF NEW YORK

DaMoN 2015, Melbourne, Victoria, Australia
Databases & Compression

- Process data on **disk**
  - Nearly *unlimited* capacity
  - Affects query *optimization*
    - Minimize # of blocks fetched
    - Minimize # of random block accesses
  - Compress to improve disk speed
    - Focused on *compression rate* since disks are “slow”

![Bar chart comparing read GB/s for HDD 5400 RPM, HDD 7200 RPM, and SSD (2014-2015)]
Databases & Compression

- **Process data on disk**
  - Nearly *unlimited* capacity
  - Affects query *optimization*
    - Minimize # of blocks fetched
    - Minimize # of random block accesses
  - Compress to improve disk speed
    - Focused on *compression rate* since disks are “slow”

- **Process data on RAM**
  - Always *limited* capacity
  - Affects query *optimization* & query *execution*
    - Minimize # of accesses (e.g. column stores & late materialization)
    - Minimize # of *random* (out of CPU cache) accesses (e.g. partitioned join)
  - Compress to improve RAM speed & *avoid* disk
    - Focused on *(de-)compression efficiency* as RAM is “fast”
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group nearby similar values
    - e.g. run-length-encoding, frame-of-reference

\[ 8 \times 32 = 256 \text{ bits} \]

\[ \min = 14 \]
\[ \max = 21 \]
\[ b = \log (\max - \min + 1) = 3 \text{ bits per code} \]
\[ 2 \times 32 = 64 \text{ bits} \]
\[ + 8 \times b = 88 \text{ bits} \]
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group *nearby* similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a *symbol* to each distinct value
    - e.g. dictionary compression

![Diagram showing compression process]

original data

A C A B A D C B

$n^*W$ bits

dictionary with
$D$ distinct values

$D = \log_2 D$

$n^*b$ bits

compressed data

0 2 0 1 0 3 2 1
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group *(nearby)* similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a *symbol* to each distinct value
    - e.g. dictionary compression
  - Frequency (symbol) compression
    - Compress *(frequent)* symbols with less bits
    - e.g. Huffman coding (slow), multiple dictionaries (fast)
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group *nearby* similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a *symbol* to each distinct value
    - e.g. dictionary compression
  - Frequency (symbol) compression
    - Compress *frequent* symbols with less bits
    - e.g. Huffman coding (slow), multiple dictionaries (fast)

- DBMS integration
  - Decompress *during* execution
    - In CPU cache (non-integrated) or in registers (integrated)
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group nearby similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a symbol to each distinct value
    - e.g. dictionary compression
  - Frequency (symbol) compression
    - Compress frequent symbols with less bits
    - e.g. Huffman coding (slow), multiple dictionaries (fast)

- DBMS integration
  - Decompress during execution
    - In CPU cache (non-integrated) or in registers (integrated)
  - Process compressed data without decompressing
Bit Packing

Definition
- Input code width is hardware-supported
  - 8-bit, 16-bit, 32-bit, 64-bit
- Output code width $b$ must be (almost) constant
  - Either constant across the entire input
  - Or constant for the next group of items (e.g. frame-of-reference)
Bit Packing

- Layouts
  - *Horizontal* bit packing
    - Bits per code are *contiguous*
Bit Packing

❖ Layouts
  ❖ **Horizontal** bit packing
    ❖ Bits per code are *contiguous*
  ❖ **Vertical** bit packing
    ❖ Bits of codes are *interleaved*

\[
\begin{align*}
\text{Horizontal:} & \quad 00010000 \quad 10100000 \quad 11000000 \quad 11111000 \\
\text{Vertical:} & \quad 01010000 \quad 11001000 \quad 10001000 \quad 00100000 \\
\text{b} &= 5 \quad \cdots \quad \cdots \\
\text{k} &= 4 \quad 0111 \quad 0011 \quad 0101 \quad 1001 \quad 0001 \quad 0110 \quad 1100 \quad 0001 \quad 1000 \quad 0110 \\
\end{align*}
\]
Bit Packing

- Layouts
  - **Horizontal** bit packing
    - Bits per code are *contiguous*
  - **Vertical** bit packing
    - Bits of codes are *interleaved*

\[
\begin{array}{cccccccc}
00010 & 000 & 10100 & 000 & 11000 & 000 & 1111 & 000 \\
& & & & & & & \\
\end{array}
\begin{array}{cccccccc}
01010 & 000 & 11001 & 000 & 10001 & 000 & 00100 & 000 \\
& & & & & & & \\
\end{array}
\]

\[b = 5\]
\[k = 4\]

\[
\begin{array}{cccccccc}
0111 & 0011 & 0101 & 1001 & 0001 \\
\end{array}
\begin{array}{cccccccc}
0110 & 1100 & 0001 & 1000 & 0110 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
00100 & 000 & 10100 & 000 & 11000 & 000 & 1111 & 000 \\
& & & & & & & \\
\end{array}
\begin{array}{cccccccc}
01010 & 000 & 11001 & 000 & 10001 & 000 & 00100 & 000 \\
& & & & & & & \\
\end{array}
\]

\[b = 5\]
\[k = 8\]

\[
\begin{array}{cccccccc}
01110110 & 00111100 & 01010001 & 10011000 & 00010110 \\
\end{array}
\]
Outline

❖ Operations
  ❖ Packing
  ❖ Unpacking
  ❖ Scanning
Outline

❖ Operations
  ❖ Packing
  ❖ Unpacking
  ❖ Scanning

❖ Horizontal layouts
  ❖ Fully packed
    ❖ Fast unpacking & scanning
  ❖ Word aligned
    ❖ Faster scanning
Outline

❖ Operations
  ❖ Packing
  ❖ Unpacking
  ❖ Scanning

❖ Horizontal layouts
  ❖ Fully packed
    ❖ Fast unpacking & scanning
  ❖ Word aligned
    ❖ Faster scanning

❖ Vertical layout
  ❖ Known traits
    ❖ Fastest scanning
❖ New traits
  ❖ Fast packing & unpacking
Horizontal Layout

- Fully packed
  - No space wasted
    - Codes can *span* across 2 packed words
Horizontal Layout

- **Fully** packed
  - No space wasted
    - Codes can *span* across 2 packed words
  - Packing
    - Process 1 unpacked *code* per iteration
    - Branch to store output packed *word*
  - Unpacking
    - Process 1 output *code* per iteration
    - Branch to load input packed *word*

![Graph showing throughput vs. number of bits for packing and unpacking](image-url)
**Fully packed**
- No space wasted
- Codes can span across 2 packed words

**Packing**
- Process 1 unpacked code per iteration
- Branch to store output packed word

**Unpacking**
- Process 1 output code per iteration
- Branch to load input packed word
- Can be written in SIMD!

Based on paper by T. Willhalm et al. @ VLDB 2009 (& improved using latest SIMD ISA)
Horizontal Layout

- **Fully** packed
  - No space wasted
    - Codes can *span* across 2 packed words
  - Packing
    - Process 1 unpacked *code* per iteration
    - Branch to store output packed *word*
  - Unpacking
    - Process 1 output *code* per iteration
    - Branch to load input packed *word*
    - Can be written in *SIMD*!

![Unpacking throughput graph](image)

*up to 7X improvement from SIMD*
Horizontal Layout

- **Fully** packed
  - No space wasted
    - Codes can span across 2 packed words
  - Packing
    - Process 1 unpacked code per iteration
    - Branch to store output packed word
  - Unpacking
    - Process 1 output code per iteration
    - Branch to load input packed word
    - Can be written in SIMD!
  - Scanning
    - Unpack the codes in CPU registers
    - Evaluate selective predicates and append to bitmap
    - Must unpack first thus bounded by $O(n)$
Horizontal Layout

- **Fully packed**
  - No space wasted
    - Codes can span across 2 packed words
  - Packing
    - Process 1 unpacked code per iteration
    - Branch to store output packed word
  - Unpacking
    - Process 1 output code per iteration
    - Branch to load input packed word
    - Can be written in SIMD!
  - Scanning
    - Unpack the codes in CPU registers
    - Evaluate selective predicates and append to bitmap
    - Must unpack first thus bounded by \( O(n) \)
    - Can also be written in SIMD via SIMD unpacking

```
select ... where column < C ...
```

```
<table>
<thead>
<tr>
<th>00010101</th>
<th>00110001</th>
<th>11110101</th>
<th>01100110</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010000</td>
<td>10100000</td>
<td>11000000</td>
<td>11110000</td>
</tr>
<tr>
<td>01100000</td>
<td>01100000</td>
<td>01100000</td>
<td>01100000</td>
</tr>
<tr>
<td>00000000</td>
<td>01111111</td>
<td>11111111</td>
<td>00000000</td>
</tr>
</tbody>
</table>

compare with C

extract
```

```
0110
```
Horizontal Layout

- **Fully** packed
  - No space wasted
    - Codes can span across 2 packed words
  - Packing
    - Process 1 unpacked code per iteration
    - Branch to store output packed word
  - Unpacking
    - Process 1 output code per iteration
    - Branch to load input packed word
    - Can be written in SIMD!
  - Scanning
    - Unpack the codes in CPU registers
    - Evaluate selective predicates and append to bitmap
    - Must unpack first thus bounded by $O(n)$
    - Can also be written in SIMD via SIMD unpacking

```
C1 <= column <= C2
```

Throughput (GB/s)

```
slower than unpacking
```

Number of bits
Horizontal Layout

- **Word aligned**
  - Waste space to get alignment
  - Pack $b' = w / (b+1)$ codes per processor word
  - Extra bit per word used for *scanning*

```plaintext
fully packed

01 10 11 00

word aligned

unused high order bits per word

010 100 00

unused extra bit per code
```
Word *aligned*
- Waste space to get alignment
  - Pack $b' = \frac{w}{b+1}$ codes per processor word
  - Extra bit per word used for *scanning*
- Packing
  - 1 packed word at a time
  - Nested loop to pack $b'$ codes
Word *aligned*

- Waste space to get alignment
  - Pack $b' = \frac{w}{(b+1)}$ codes per processor word
  - Extra bit per word used for *scanning*

- Packing
  - 1 packed word at a time
  - Nested loop to pack $b'$ codes

- Unpacking
  - 1 packed word at a time
  - Nested loop to unpack $b'$ codes

Unpacking throughput (GB/s)

- Fully packed (scalar)
- Fully packed (SIMD)
- Word aligned (scalar)

*slower than SIMD*
Horizontal Layout

- Word aligned
  - Waste space to get alignment
    - Pack $b' = w / (b+1)$ codes per processor word
    - Extra bit per word used for scanning
  - Packing
    - 1 packed word at a time
    - Nested loop to pack $b'$ codes
  - Unpacking
    - 1 packed word at a time
    - Nested loop to unpack $b'$ codes
  - Scanning
    - Evaluate predicates without unpacking
    - Works with simple order predicates: $<,=,>$
    - Boolean result in overflow bit of $b$-bit arithmetic
    - Executing $< O(n)$ operations

```
select ... where column < C ... 

01

set constant C
010 010 00

invert code bits
010 100 00 ^ 110 110 00 = 100 010 00

add constant
100 010 00 + 010 010 00 = 110 001 00

extract sign
110 001 00 —> 01
```

Based on paper by Leslie Lamport @ CACM 1975
Horizontal Layout

- **Word aligned**
  - Waste space to get alignment
  - Pack $b' = w / (b+1)$ codes per processor word
  - Extra bit per word used for *scanning*
- Packing
  - 1 packed word at a time
  - Nested loop to pack $b'$ codes
- Unpacking
  - 1 packed word at a time
  - Nested loop to unpack $b'$ codes
- Scanning
  - Evaluate predicates *without* unpacking
  - Works with *simple* order predicates: $<$, $<=$, $>$
  - Boolean result in *overflow* bit of $b$-bit arithmetic
  - Executing $< O(n)$ operations
Vertical Layout

- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
Fully packed & word aligned

- Interleave bits of $k$ codes
  - $k$ divides the processor word
- Scanning
  - Evaluate without unpacking
  - Can skip words early

```
00010110  00000000  11101001  ___0_00_
10011000  11111111  100_1__0  _110_001
01010001  11111111  0____0____  11101001
00111100  00000000  
01110110  00000000
```

"=" $X \&= \sim (column \wedge C)$ ........

"<" $Y \mid= C \& (\sim X)$ ..........................

stop if $X = 0$

Based on paper by Y. Li et al. @ SIGMOD 2013
Vertical Layout

- Fully packed & word aligned
  - Interleave bits of $k$ codes
  - $k$ divides the processor word
- Scanning
  - Evaluate without unpacking
  - Can skip words early

Scanning throughput (GB/s)

- Vertical $k = 64$ (scalar)
- Horizontal full (SIMD)
- Horizontal word (scalar)

Fastest in most cases
**Vertical Layout**

- *Fully* packed & word *aligned*
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetched

$k = 64$

$b = 5$

$k = 256$

$b = 5$
Vertical Layout

- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetcheds

### Graph

- Vertical $k =$ 64 (scalar)
- Horizontal full (SIMD)
- Horizontal word (scalar)
- Vertical $k =$ 8192 (SIMD)

Scanning throughput (GB/s)

Number of bits

*Faster due to cache line skip*
- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetches
  - Packing
    - Transfer $nb$ bits across registers

```
00100000 00000000 00000000
10000000 10000000 10000000
10001000 00000000 01100000
11001000 00000000 00101100
01010000 00000000 00010110
11110000 00000000 00000000
```

```
00100000 00000000 00000000
10000000 10000000 10000000
10001000 00000000 01100000
11001000 00000000 00101100
01010000 00000000 00010110
11110000 00000000 00000000
```
**Vertical Layout**

- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
  - Increase $k$ to minimize false (pre)fetched
- **Packing**
  - Transfer $nb$ bits across registers
  - Can be written in *SIMD*

![Diagram of bit manipulation and shift operations]

- Extract & shift
  - Shift
  - Extract & shift
  - Extract & shift
  - Extract & shift
- **Fully** packed & word *aligned*
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetcheds
  - **Packing**
    - Transfer $nb$ bits across registers
    - Can be written in *SIMD*!
    - Extract bits *per byte* not per int

<table>
<thead>
<tr>
<th>Pack</th>
<th>Extract &amp; Shift</th>
<th>Shift &amp; Pack</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001010</td>
<td>00101000</td>
<td>0000111100</td>
</tr>
<tr>
<td>00001010</td>
<td>0011001</td>
<td>0000100100</td>
</tr>
<tr>
<td>01010000</td>
<td>10011000</td>
<td>00010110</td>
</tr>
<tr>
<td>00101000</td>
<td>01101000</td>
<td>10011000</td>
</tr>
<tr>
<td>00010100</td>
<td>01010000</td>
<td>00010110</td>
</tr>
<tr>
<td>00010111</td>
<td>01001000</td>
<td>00010110</td>
</tr>
<tr>
<td>00010110</td>
<td>01010000</td>
<td>00010110</td>
</tr>
<tr>
<td>00010111</td>
<td>01010000</td>
<td>00010110</td>
</tr>
<tr>
<td>00010111</td>
<td>01010000</td>
<td>00010110</td>
</tr>
<tr>
<td>00010111</td>
<td>01010000</td>
<td>00010110</td>
</tr>
<tr>
<td>00010111</td>
<td>01010000</td>
<td>00010110</td>
</tr>
</tbody>
</table>
Vertical Layout

- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetched
- **Packing**
  - Transfer $nb$ bits across registers
  - Can be written in *SIMD*!
  - Extract bits *per byte* not per int

**Graph:**
- Packing throughput (GB/s) [logarithmic scale]
- Number of bits
- *SIMD* vs *Scalar*

*up to 27X improvement!*
Vertical Layout

- **Fully packed & word aligned**
  - Interleave bits of \( k \) codes
  - \( k \) divides the processor word
- Scanning
  - Evaluate *without* unpacking
  - Can *skip* words early
  - Increase \( k \) to minimize false (pre)fetches
- Packing
  - Transfer \( nb \) bits across registers
  - Can be written in *SIMD*!
  - Extract bits *per byte* not per int
- Unpacking
  - Transfer \( nb \) bits across registers
Vertical Layout

- **Fully** packed & word *aligned*
  - Interleave bits of *k* codes
    - *k* divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase *k* to minimize false (pre)fetcheds

- **Packing**
  - Transfer *nb* bits across registers
  - Can be written in *SIMD*!
  - Extract bits *per byte* not per int

- **Unpacking**
  - Transfer *nb* bits across registers
  - Can be written in *SIMD*!
**Vertical Layout**

- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetcheds
  - Packing
    - Transfer $nb$ bits across registers
    - Can be written in SIMD!
    - Extract bits *per byte* not per int
  - Unpacking
    - Transfer $nb$ bits across registers
    - Can be written in SIMD!
    - Insert bits *per byte* not per int

---

**Convert bit to byte & add**

```
0000 0000 0000 1000 0000 1000 1000 0000
```

**Up-convert & add**

```
00000000 00000000 00000000 10000000
00000000 10000000 10000000 00000000
```

---

**Convert bit to byte & add**

```
1001 1000
```

**Up-convert & add**

```
0001 1010 1100 1111 0101 1100 1000 0010
```

---

**Convert bit to byte & add**

```
0001 0100 1100 1111 0101 1100 1000 0010
```

**Up-convert & add**

```
00010000 10100000 11000000 11110000
01010000 01001000 10001000 00100000
```
**Vertical Layout**

- **Fully packed & word aligned**
  - Interleave bits of $k$ codes
    - $k$ divides the processor word
  - Scanning
    - Evaluate *without* unpacking
    - Can *skip* words early
    - Increase $k$ to minimize false (pre)fetchedes
  - Packing
    - Transfer $nb$ bits across registers
    - Can be written in **SIMD**!
    - Extract bits *per byte* not per int
  - Unpacking
    - Transfer $nb$ bits across registers
    - Can be written in **SIMD**!
    - Insert bits *per byte* not per int

![Unpacking throughput (GB/s)](image)

Unpacking throughput (GB/s)

- **11–20X improvement!**

Number of bits

- **SIMD**
- **Scalar**
Scalar to \textit{SIMD} for packing & unpacking ($k = 64$)

* Scalar scan

\textit{too slow!}
Vertical Layout

- Increasing $k$ to the L1 cache size ($k = 8192$)
- SIMD scanning

---

**Throughput (GB/s)**

- Pack
- Unpack
- Scan

---

- **faster scan**
- **slightly slower pack & unpack**

---

$k = 64$  Number of bits  $k = 8192$
Vertical Layout

- If not memory *bound*
  - Using 1 thread

![Graph showing scanning throughput (GB/s) vs. number of bits for different configurations:

- **Vertical (K = 8192)**
- **Horizontal full**
- **Horizontal word**
- **Uncompressed**

**Multi-threaded**

**Single-threaded**

*uncompressed as fast if not bound*
Conclusions

❖ Horizontal layouts
  ❖ Fully packed
    ❖ No wasted space but somewhat slow
    ❖ Can optimize unpacking & scanning with SIMD
Conclusions

❖ Horizontal layouts
  ❖ Fully packed
    ❖ No wasted space but somewhat slow
    ❖ Can optimize unpacking & scanning with SIMD
  ❖ Word aligned
    ❖ Fast scalar scans but not optimal due to wasted space
Conclusions

❖ Horizontal layouts
  ❖ Fully packed
    ❖ No wasted space but somewhat slow
    ❖ Can optimize unpacking & scanning with SIMD
  ❖ Word aligned
    ❖ Fast scalar scans but not optimal due to wasted space

❖ Vertical layout
  ❖ Known techniques
    ❖ Fast scalar scans without wasting space
    ❖ Very slow scalar packing & unpacking
Conclusions

❖ Horizontal layouts
  ❖ Fully packed
    ✦ No wasted space but somewhat slow
    ✦ Can optimize unpacking & scanning with SIMD
  ❖ Word aligned
    ✦ Fast scalar scans but not optimal due to wasted space

❖ Vertical layout
  ❖ Known techniques
    ✦ Fast scalar scans without wasting space
    ✦ Very slow scalar packing & unpacking
  ❖ New techniques
    ✦ Fast packing & unpacking using SIMD
    ✦ Maximize bit transfers by using the smallest SIMD lanes
    ✦ Increase \( k \) to skip cache lines effectively
Questions ?