Vectorized Execution (Part II)

@Andy_Pavlo // 15-721 // Spring 2018
TODAY’S AGENDA

Bit-Slicing
Bit-Weaving
Relaxed Operator Fusion (Prashanth)
### BITMAP ENCODING

**Original Data**

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>M</td>
</tr>
</tbody>
</table>
## Bitmap Encoding

### Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>M</td>
</tr>
</tbody>
</table>

### Compressed Data

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
BITMAP INDEX: ENCODING

Approach #1: Equality Encoding
→ Basic scheme with one Bitmap per unique value.

Approach #2: Range Encoding
→ Use one Bitmap per interval instead of one per value.

Approach #3: Hierarchical Encoding
→ Use a tree to identify empty key ranges.

Approach #4: Bit-sliced Encoding
→ Use a Bitmap per bit location across all values.
HIERARCHICAL ENCODING

Keys: 1, 3, 9, 12, 13, 14, 38, 40
HIERARCHICAL ENCODING

Keys: 1, 3, 9, 12, 13, 14, 38, 40

Original: 8 bytes
Encoded: 4 bytes
**BIT-SLICED ENCODING**

**Original Data**

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

**Bit-Slices**
BIT-SLICED ENCODING

Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

Bit-Slices

bin(21042)→ 00101001000110010

Source: Jignesh Patel
**BIT-SLICED ENCODING**

**Original Data**

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

**Bit-Slices**

bin(21042) → 00101001000110010

Source: Jignesh Patel
**BIT-SLICED ENCODING**

**Original Data**

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

**Bit-Slices**

\[
\text{bin}(21042) \rightarrow 00101001000110010
\]

Source: Jignesh Patel
**BIT-SLICED ENCODING**

**Original Data**

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

**Bit-Slices**

```
SELECT * FROM customer_dim
WHERE zipcode < 15217
```

Source: Jignesh Patel
**BIT-SLICED ENCODING**

Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

Bit-Slices

```
```

```
```

SELECT * FROM customer_dim
WHERE zipcode < 15217

Walk each slice and construct a result bitmap.
### Original Data

<table>
<thead>
<tr>
<th>id</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21042</td>
</tr>
<tr>
<td>2</td>
<td>15217</td>
</tr>
<tr>
<td>3</td>
<td>02903</td>
</tr>
<tr>
<td>4</td>
<td>90220</td>
</tr>
<tr>
<td>6</td>
<td>14623</td>
</tr>
<tr>
<td>7</td>
<td>53703</td>
</tr>
</tbody>
</table>

### Bit-Slices

<table>
<thead>
<tr>
<th>N?</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Walk each slice and construct a result bitmap. Skip entries that have 1 in first 3 slices (16, 15, 14)

Source: Jignesh Patel
BIT-SLICED ENCODING

Bit-slices can also be used for efficient aggregate computations.

Example:  \( \text{SUM(} \text{attr} \text{)} \)
- First, count the number of 1s in \( \text{slice}_{17} \) and multiply the count by \( 2^{17} \)
- Then, count the number of 1s in \( \text{slice}_{16} \) and multiply the count by \( 2^{16} \)
- Repeat for the rest of slices...

Intel added \text{POPCNT} SIMD instruction in 2008.
Observation

The bit width of compressed data does not always fit naturally into SIMD register lanes.
→ This means that the DBMS has to do extra work to transform data into the proper format.

Just because the **lanes** are fully utilized does not mean the **bits** are fully utilized…
The bit width of compressed data does not always fit naturally into SIMD register lanes. This means that the DBMS has to do extra work to transform data into the proper format.

Just because the lanes are fully utilized does not mean the bits are fully utilized...
BITWEAVING

Alternative storage layout for columnar databases that is designed for efficient predicate evaluation on compressed data using SIMD.
→ Order-preserving dictionary encoding.
→ Bit-level parallelization.
→ Only require common instructions (no scatter/gather)

BITWEAVING

Alternative storage layout for columnar databases that is designed for efficient predicate evaluation on compressed data using SIMD:

→ Order-preserving dictionary encoding.
→ Bit-level parallelization.
→ Only require common instructions.

Implemented in Wisconsin’s QuickStep engine.

Became an Apache Incubator project in 2016.
BITWEAVING – STORAGE LAYOUTS

Approach #1: Horizontal
→ Row-oriented storage at the bit-level

Approach #2: Vertical
→ Column-oriented storage at the bit-level
## HORIZONTAL STORAGE

<table>
<thead>
<tr>
<th>Segment #1</th>
<th>Segment #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$ 0 0 1</td>
<td>$t_8$ 1 0 0</td>
</tr>
<tr>
<td>$t_1$ 1 0 1</td>
<td>$t_9$ 0 1 1</td>
</tr>
<tr>
<td>$t_2$ 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$t_3$ 0 0 1</td>
<td></td>
</tr>
<tr>
<td>$t_4$ 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$t_5$ 1 0 0</td>
<td></td>
</tr>
<tr>
<td>$t_6$ 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$t_7$ 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

Segment #1: $t_0 = 1$, $t_1 = 5$, $t_2 = 6$, $t_3 = 1$, $t_4 = 6$, $t_5 = 4$, $t_6 = 0$, $t_7 = 7$

Segment #2: $t_8 = 4$, $t_9 = 3$
HORIZONTAL STORAGE

Segment #1

$t_0: 0 0 1$
$t_1: 1 0 1$
$t_2: 1 1 0$
$t_3: 0 0 1$
$t_4: 1 1 0$
$t_5: 1 0 0$
$t_6: 0 0 0$
$t_7: 1 1 1$

$t_0$ $t_4$

$v_0: 0 0 0 1 0 1 1 0$
$v_1: 0 1 0 1 0 1 0 0$
$v_2: 0 1 1 0 0 0 0 0$
$v_3: 0 0 0 1 0 1 1 1$

Segment #2

$t_8: 1 0 0$
$t_9: 0 1 1$

$v_4: 0 1 0 0 0 0 1 1$

$t_8$ $t_9$
HORIZONTAL STORAGE

Segment #1

Segment #2

Processor Word

CMU 15-721 (Spring 2018)
HORIZONTAL STORAGE

Segment #1

Segment #2

Delimiter

Processor Word
BITWEAVING/H – EXAMPLE

SELECT * FROM table
WHERE val < 5

\[ X = \begin{array} {cccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
\end{array} \]

\[ Y = \begin{array} {cccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \]

Source: Jignesh Patel
**BITWEAVING/H – EXAMPLE**

**SQL Example:**

```sql
SELECT * FROM table
WHERE val < 5
```

**Example Values:**

- **$X = [0, 0, 0, 1, 0, 1, 1, 0]$**
- **$Y = [0, 1, 0, 1, 0, 1, 0, 1]$**
- **$mask = [0, 1, 1, 1, 0, 1, 1, 1]$**

**Calculation:**

\[
(Y + (X \oplus mask)) \land \neg mask = [1, 0, 0, 0, 0, 0, 0, 0]
\]
SELECT * FROM table
WHERE val < 5

\[
\begin{array}{c}
X = \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
t_0 & t_4 \\
5 & \text{mask} = \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
Y = \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
5 & \text{(Y + (X ⊕ mask)) ∧ ¬mask =} \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Selection Vector

Source: Jignesh Patel
**BITWEAVING/H – EXAMPLE**

```
SELECT * FROM table
WHERE val < 5
```

\[
\begin{array}{cccc}
X &=& t_0 & t_4 \\
   &=& \begin{array}{cccc}
0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0
\end{array} \\
Y &=& \begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0
\end{array} \\
\text{mask} &=& \begin{array}{cccc}
0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
1 & 1 & 1 & 1
\end{array}
\end{array}
\]

\[
(Y + (X \oplus \text{mask})) \wedge \neg \text{mask} = \begin{array}{cccc}
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array}
\]

\[
1 < 5 \quad 5 < 6
\]

Source: Jignesh Patel
SELECT * FROM table WHERE val < 5

Only requires three instructions to evaluate a single word.

Works on any word size and encoding length.

Paper contains algorithms for other operators.

\[
\begin{align*}
X &= \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
5 & 5 & & & & & & 
\end{bmatrix} \\
Y &= \begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
& & & & & & & 
\end{bmatrix} \\
mask &= \begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
& & & & & & & 
\end{bmatrix} \\
(Y + (X \oplus mask)) \land \neg mask &= \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 < 5 & 5 < 6 & & & & & & 
\end{bmatrix}
\]
**BITWEAVING/H – EXAMPLE**

```
SELECT * FROM table
WHERE val < 5
```
**BITWEAVING/H – EXAMPLE**

```
SELECT * FROM table
WHERE val < 5
```
BITWEAVING/H – EXAMPLE

SELECT * FROM table
WHERE val < 5
SELECT * FROM table
WHERE val < 5
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate. Have to convert it into offsets / positions.

→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate.
Have to convert it into offsets / positions.
→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate. Have to convert it into offsets / positions.

→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table

```
tuples = [ ]
for (i=0; i<n; i++) {
  if sv[i] == 1
    tuples.add(i);
}
```
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate.
Have to convert it into offsets / positions.
→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table

Selection Vector: 

<table>
<thead>
<tr>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate. Have to convert it into offsets / positions.

→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table

Selection Vector

<table>
<thead>
<tr>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
<th>t₅</th>
<th>t₆</th>
<th>t₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Positions Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{0,3,5,6}</td>
</tr>
</tbody>
</table>
VERTICAL STORAGE

Segment #1

\[
\begin{array}{c}
  t_0 & 0 & 0 & 1 \\
  t_1 & 1 & 0 & 1 \\
  t_2 & 1 & 1 & 0 \\
  t_3 & 0 & 0 & 1 \\
  t_4 & 1 & 1 & 0 \\
  t_5 & 1 & 0 & 0 \\
  t_6 & 0 & 0 & 0 \\
  t_7 & 1 & 1 & 1 \\
\end{array}
\]

Segment #2

\[
\begin{array}{c}
  t_8 & 1 & 0 & 0 \\
  t_9 & 0 & 1 & 1 \\
\end{array}
\]
## VERTICAL STORAGE

<table>
<thead>
<tr>
<th>Segment #1</th>
<th>Segment #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$t_8$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$t_9$</td>
</tr>
<tr>
<td>$t_2$</td>
<td></td>
</tr>
<tr>
<td>$t_3$</td>
<td></td>
</tr>
<tr>
<td>$t_4$</td>
<td></td>
</tr>
<tr>
<td>$t_5$</td>
<td></td>
</tr>
<tr>
<td>$t_6$</td>
<td></td>
</tr>
<tr>
<td>$t_7$</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
VERTICAL STORAGE

Segment #1

\[
\begin{array}{c}
\begin{array}{c}
 t_0 \\
 t_1 \\
 t_2 \\
 t_3 \\
 t_4 \\
 t_5 \\
 t_6 \\
 t_7 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
 1 \\
 0 \\
 1 \\
 1 \\
 0 \\
 1 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
 v_0 \\
 v_1 \\
 v_2 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
 1 \\
\end{array}
\begin{array}{c}
 1 \\
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 1 \\
 1 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
 1 \\
\end{array}
\begin{array}{c}
 1 \\
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 1 \\
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 1 \\
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
 0 \\
\end{array}
\end{array}
\]

Processor Word

Segment #2

\[
\begin{array}{c}
\begin{array}{c}
 t_8 \\
 t_9 \\
\end{array}
\begin{array}{c}
 1 \\
 0 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\begin{array}{c}
 0 \\
 1 \\
\end{array}
\end{array}
\]

CMU 15-721 (Spring 2018)
Bitweaving/V – Example

**Segment #1**

```
<table>
<thead>
<tr>
<th></th>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v_2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**SELECT * FROM table WHERE key = 2**
### BITWEAVING/V – EXAMPLE

**Segment #1**

```plaintext
SELECT * FROM table
WHERE key = 2

\[
\begin{array}{cccccccc}
  t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \\
  v_0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
  v_1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
  v_2 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
```
**BITWEAVING/V – EXAMPLE**

**Segment #1**

```
SELECT * FROM table
WHERE key = 2
```

```
010

0000000000
```

**SIMD Compare**

```
01101101
v0
```

```
00101001
v1
```

```
11010001
v2
```

```
10010010
```

```
t0 t1 t2 t3 t4 t5 t6 t7
01101101
```

```
```
Can perform early pruning just like in BitMap indexes.

The last vector is skipped because all bits in previous comparison are zero.
EVALUATION

Single-threaded execution of a single query derived from TPC-H benchmark.
→ Selectivity: 10%

10GB TPC-H Database
→ 1 billion tuples
→ Uniform distribution

SELECT COUNT(*)
FROM R
WHERE R.a < C

Source: Jignesh Patel
EVALUATION

TPC-H Aggregation Query
Intel Xeon X5650 @ 2.66 GHz

- Naïve
- SIMD Scan
- BitWeaving/V
- BitWeaving/H

Fewer Cache Misses
SIMD Parallelization
Early Pruning

Source: Jignesh Patel
PARTING THOUGHTS

Just like in query compilation, getting the best performance with vectorization requires the DBMS to store data in a way that is best for the CPU and not the best for humans’ understanding.