BitWeaving: Fast Scans for Main Memory Data Processing

Yinan Li and Jignesh M. Patel
University of Wisconsin-Madison
Motivation
- Example TPC-H Query 6

```
SELECT SUM(l_extendedprice * l_discount)
FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
  AND l_discount BETWEEN Discount - 0.01
    AND Discount + 0.01
  AND l_quantity < Quantity
```
Motivation
- Example TPC-H Query 6

Main memory analytics DBMSs convert native column values to codes.

```sql
SELECT SUM(l_extendedprice * l_discount)
FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
  AND l_discount BETWEEN Discount - 0.01
    AND Discount + 0.01
  AND l_quantity < Quantity
```
Motivation
- Example TPC-H Query 6

Main memory analytics DBMSs convert native column values to codes.

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

12 bits
4 bits
6 bits
Main memory analytics DBMSs convert native column values to codes.

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

CPU register

- Code size: 4-12 bits
- Word size: 64 bits
Motivation
- Example TPC-H Query 6

Main memory analytics DBMSs convert native column values to codes.

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

CPU register

- SIMD word size: 256 bits
- Code size: 4-12 bits
- Word size: 64 bits
Motivation
- Example TPC-H Query 6

Main memory analytics DBMSs convert native column values to codes.

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

- Code size: 4-12 bits
- Word size: 64 bits
- SIMD word size: 256 bits
- Underutilizes the processor word!
Motivation
- Example TPC-H Query 6

Main memory analytics DBMSs convert native column values to codes.

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

CPU register

- SIMD word size: 256 bits
- Word size: 64 bits
- Code size: 4-12 bits
Motivation
- Example TPC-H Query 6

Main memory analytics DBMSs convert native column values to codes.

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

Intra-cycle parallelism!
BitWeaving
BitWeaving

• In this talk, we introduce BitWeaving
  – A fast \textit{scan} method
  – for \textit{column-oriented} databases
BitWeaving

• In this talk, we introduce BitWeaving
  – A fast scan method
  – for column-oriented databases
• Fully exploits intra-cycle parallelism
BitWeaving

• In this talk, we introduce BitWeaving
  – A fast scan method
  – for column-oriented databases
• Fully exploits intra-cycle parallelism
• How: By “gainfully” using every bit in every processor word.
BitWeaving: Two Flavors

BitWeaving/H (Horizontal bit organization)

<table>
<thead>
<tr>
<th>Code</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 0 1 0 1 0</td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
</tr>
</tbody>
</table>

BitWeaving/V (Vertical bit organization)

<table>
<thead>
<tr>
<th>Code</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 0 0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1 0 1 0 0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0 0 0 1 0 0 1</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
Framework
Framework

• Targets single-table scans
Framework

• Targets single-table scans

• Column-scalar scan: scan on a single column
  – produce a result bit vector, with one bit for each input tuple to indicate the matching tuples
Framework

- Targets single-table scans

- Column-scalar scan: scan on a single column
  - produce a result bit vector, with one bit for each input tuple to indicate the matching tuples

- Complex predicates in the scan: logical AND and OR operations on these result bit vectors
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
   AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
   AND l_quantity < Quantity
Framework – Example

```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
    AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
    AND l_quantity < Quantity
```
Framework – Example

```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
    AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
    AND l_quantity < Quantity
```
Framework – Example

```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity
```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
   AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
   AND l_quantity < Quantity
Framework – Example

SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
  AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
  AND l_quantity < Quantity

Convert to a RID List
RID List: 9, 15

Result bit vector

Result bit vector

Result bit vector

Result bit vector

Result bit vector

AND

l_quantity
Framework – Example

```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
  AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
  AND l_quantity < Quantity
```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity
Framework – Example

```sql
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity
```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity

Result bit vector
RID List: 9, 15

Mixing of BitWeaving/V BitWeaving/H columns

l_price -> Aggregation
l_discount

Result bit vector
l_shipdate

Result bit vector

AND

Result bit vector

AND

Result bit vector

l_discount

Result bit vector

l_quantity
Framework – Example

```sql
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity
```
Framework – Example

```
SELECT SUM(l_discount * l_price) FROM lineitem
WHERE l_shipdate BETWEEN Date AND Date + 1 year
AND l_discount BETWEEN Discount - 0.01 AND Discount + 0.01
AND l_quantity < Quantity
```
Outline

• Motivation & Overview
• BitWeaving/V
• BitWeaving/H
• Evaluations
• Conclusion
BitWeaving/V

- **Storage layout**
  - Bit-level columnar data organization, i.e. it's like a *bit-level columnar store*.

- **Column-scalar scan**
  - Predicate evaluation is converted to logical computation on these “words of bits”
BitWeaving/V

• Storage layout
  – Bit-level columnar data organization, i.e. its like a bit-level columnar store.

• Column-scalar scan
  – Predicate evaluation is converted to logical computation on these “words of bits”

• Based on the idea of Bit-sliced index*. Two differences:
  – Segmented storage layout
  – Early pruning technique

BitWeaving/V – Storage Layout

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
<th>c8</th>
<th>c9</th>
<th>c10</th>
<th>c11</th>
<th>c12</th>
<th>c13</th>
<th>c14</th>
<th>c15</th>
<th>c16</th>
<th>c17</th>
<th>c18</th>
<th>c19</th>
</tr>
</thead>
</table>


BitWeaving/V – Storage Layout

Segment 1

| c1 | c2 | c3 | c4 | c5 | c6 | c7 | c8 | c9 | c10 | c11 | c12 | c13 | c14 | c15 | c16 | c17 | c18 | c19 |
BitWeaving/V – Storage Layout

Segment 1

Code size (4 bits)
BitWeaving/V – Storage Layout

Segment 1

Code size (4 bits)

<table>
<thead>
<tr>
<th>Segment 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>c2</td>
</tr>
<tr>
<td>c3</td>
<td>c4</td>
</tr>
<tr>
<td>c5</td>
<td>c6</td>
</tr>
<tr>
<td>c7</td>
<td>c8</td>
</tr>
<tr>
<td>c9</td>
<td>c10</td>
</tr>
<tr>
<td>c11</td>
<td>c12</td>
</tr>
<tr>
<td>c13</td>
<td>c14</td>
</tr>
<tr>
<td>c15</td>
<td>c16</td>
</tr>
<tr>
<td>c17</td>
<td>c18</td>
</tr>
<tr>
<td>c19</td>
<td></td>
</tr>
</tbody>
</table>

Word 1

<table>
<thead>
<tr>
<th>Word 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Word 2

<table>
<thead>
<tr>
<th>Word 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Word 3

<table>
<thead>
<tr>
<th>Word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Word 4

<table>
<thead>
<tr>
<th>Word 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
BitWeaving/V – Storage Layout

Segment 1: Code size (4 bits)

Word 1: 1 1 0 0 1 0 0 0
Word 2: 0 1 0 1 0 1 0 0
Word 3: 1 0 1 1 0 1 0 0
Word 4: 0 0 1 0 1 1 1 0

The first (most significant) bits of the 8 codes
BitWeaving/V – Storage Layout

Segment 1

Code size (4 bits)

The first (most significant) bits of the 8 codes

The second bits of the 8 codes

Word 1

1 1 0 0 1 0 0 0

Word 2

0 1 0 1 0 1 0 0

Word 3

1 0 1 1 0 1 0 0

Word 4

0 0 1 0 1 1 1 0
BitWeaving/V – Storage Layout

Segment 1

1. The first (most significant) bits of the 8 codes
2. The second bits of the 8 codes
3. The third bits of the 8 codes
4. The last (least significant) bits of the 8 codes
BitWeaving/V – Storage Layout

Codes: 10, 12, 3, 6, 9, 7, 1, 0

The first (most significant) bits of the 8 codes

The second bits of the 8 codes

The third bits of the 8 codes

The last (least significant) bits of the 8 codes
BitWeaving/V – Column-scalar Scan

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8  Literal
10, 12, 3, 6, 9, 7, 1, 0  5

Segment 1

code < 5?

Literal

Code

0

1

0

1

0

1

0

1

0

1

0

1

0

1

0

1
BitWeaving/V – Column-scalar Scan

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8  Literal
10, 12, 3, 6, 9, 7, 1, 0

code < 5?

Segment 1

Literal 5
BitWeaving/V – Column-scalar Scan

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8
10, 12, 3, 6, 9, 7, 1, 0

Literal 5

code < 5?

Segment 1
## BitWeaving/V – Column-scalar Scan

Column *codes*:
- c1, c2, c3, c4, c5, c6, c7, c8
- Literal: 10, 12, 3, 6, 9, 7, 1, 0
- Literal: 5

<table>
<thead>
<tr>
<th>Segment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>0 1 0 1 0 1 0 0</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0 1 1 1 0</td>
</tr>
</tbody>
</table>

*code* < 5?

- 1: [X X √ ? X ? √ √ √]
- 1: [X X √ √ X X √ √ √]
BitWeaving/V – Column-scalar Scan

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8  Literal
10, 12, 3, 6, 9, 7, 1, 0

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exploits intra-cycle parallelism!
Uses every bit in every process word!
BitWeaving/V – Column-scalar Scan

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8

Literal values:
10, 12, 3, 6, 9, 7, 1, 0

5

code < 5?

Segment 1
BitWeaving/V – Column-scalar Scan

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8

Literal
10, 12, 3, 6, 9, 7, 1, 0

5

code < 5?

| 1 | ×× ✓ ? × ? ? ? ✓ ✓ ✓ |
| 0 | ×× ✓ ✓ × × ✓ ✓ ✓ ✓ ✓ |
| 1 | ×× ✓ ✓ ✓ × ✓ ✓ ✓ ✓ ✓ |

Segment 1

Memory space
**BitWeaving/V – Column-scalar Scan**

Column *codes*:

c1, c2, c3, c4, c5, c6, c7, c8

Literal:

10, 12, 3, 6, 9, 7, 1, 0

<table>
<thead>
<tr>
<th>Code</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

*The layout of the segment exactly matches the access pattern of column-scalar scans*
**BitWeaving/V – Early Pruning**

Column *codes*:
- c1, c2, c3, c4, c5, c6, c7, c8
- Literal
- 10, 12, 3, 6, 9, 7, 1, 0

<table>
<thead>
<tr>
<th>Segment 1</th>
<th>Literal</th>
<th>code &lt; 5?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 0 0 0</td>
<td>0</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>0 1 0 1 0 1 0 0</td>
<td>1</td>
<td>✗ ✗ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔</td>
</tr>
<tr>
<td>1 0 1 1 0 1 0 0</td>
<td>0</td>
<td>✗ ✗ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔</td>
</tr>
<tr>
<td>0 0 1 0 1 1 1 0</td>
<td>1</td>
<td>✗ ✗ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔</td>
</tr>
</tbody>
</table>

**University of Wisconsin-Madison**
BitWeaving/V – Early Pruning

Column codes:
c1, c2, c3, c4, c5, c6, c7, c8
10, 12, 3, 6, 9, 7, 1, 0

Literal 5

code < 5?

Segment 1

 Early Pruning: terminate the predicate evaluation on a segment, when all results have been determined.
Outline

• Motivation & Overview
• BitWeaving/V
• BitWeaving/H
• Evaluations
• Conclusion
BitWeaving/H

- **Storage layout**
  - Packs codes “horizontally” into processor words
- **Column-scalar scan**
  - Parallel predicate evaluation on packed codes
BitWeaving/H

• Storage layout
  – Packs codes “horizontally” into processor words

• Column-scalar scan
  – Parallel predicate evaluation on packed codes

• Shares similar basic idea with IBM Blink method*. Two differences:

BitWeaving/H

- **Storage layout**
  - Packs codes “horizontally” into processor words

- **Column-scalar scan**
  - Parallel predicate evaluation on packed codes

- Shares similar basic idea with IBM Blink method*. Two differences:
  - Uses an extra bit in each code

---

BitWeaving/H

• Storage layout
  – Packs codes “horizontally” into processor words

• Column-scalar scan
  – Parallel predicate evaluation on packed codes

• Shares similar basic idea with IBM Blink method*. Two differences:
  – Uses an extra bit in each code
  – Staggers codes across words inside a segment

BitWeaving/H

• Storage layout
  – Packs codes “horizontally” into processor words

• Column-scalar scan
  – Parallel predicate evaluation on packed codes

• Shares similar basic idea with IBM Blink method*. Two differences:
  – Uses an extra bit in each code
  – Staggers codes across words inside a segment


More details about BitWeaving/H are in the paper!
Outline

• Motivation & Overview
• BitWeaving/V
• BitWeaving/H
• Evaluations
• Conclusion
Evaluation – System

• Intel Xeon X5650
  – 64 bits ALU
  – 128 bits SIMD
  – 12MB L3 Cache
• 24GB memory
Evaluation - Micro-benchmark

• Query:
  
  ```sql
  SELECT COUNT(*)
  FROM R
  WHERE R.a < C
  ```

• 1 billion tuples
• Uniform distribution
• Selectivity: 10%
• Single thread execution
Evaluation - Micro-benchmark

Cycles / code vs. Size of code (# bits)

Naive
Evaluation - Micro-benchmark

SIMD Paper: T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. PVLDB‘09
Evaluation - Micro-benchmark

SIMD Paper: T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. PVLDB‘09
Evaluation - Micro-benchmark

Evaluation - Micro-benchmark

Improved query performance with variant indexes. SIGMOD‘97
Evaluation: Micro-benchmark

![Graph showing the evaluation results for different code sizes and implementations.](image)
Evaluation: Micro-benchmark

3X-4X speedup over BL:
1) Use the extra (delimiter) bit
2) Staggered vertical layout
Evaluation - Micro-benchmark

Cycles / code vs. Size of code (# bits)

- Naive
- SIMD
- Bit-sliced
- BL
- BitWeaving/V
- BitWeaving/H
Evaluation - Micro-benchmark

![Graph showing cycles per code size for different techniques with 2X speedup due to early pruning.]

- Naive
- SIMD
- Bit-sliced
- BL
- BitWeaving/V
- BitWeaving/H

2X speedup: Early pruning
Evaluation - Micro-benchmark

- 3X-4X over Bit-sliced: Fewer cache misses
- 2X speedup: Early pruning
Evaluation - TPC-H Query 6

- TPC-H Query 6:

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

- Scale factor 10 (~10GB)
- Selectivity: ~2%
Evaluation - TPC-H Query 6

- TPC-H Query 6:

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

- Scale factor 10 (~10GB)
- Selectivity: ~2%
Evaluation - TPC-H Query 6

TPC-H Query 6:

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

- Scale factor 10 (~10GB)
- Selectivity: ~2%
Evaluation - TPC-H Query 6

- **Cycles / tuple**
- **Naive**
- **SIMD**
- **BL**
- **Bit-sliced**
- **BW/H**
- **BW/V**

Legend:
- Aggregation
- Scan on `l_shipdate`
- Scan on `l_quantity`
- Scan on `l_discount`
Evaluation - TPC-H (Denormalized)

- TPC-H Q4, Q5, Q12, Q14, Q17, Q19
- Materialized primary-key foreign-key joins in these queries
- Statistics
  - Code size (in selection): 2 – 12 bits
  - Code size (in projection): 3 – 24 bits
  - Predicates: between, less than, equality, inequality, in
  - # predicates (in selection): 1 – 18
Evaluation - TPC-H (Denormalized)

![Graph showing speedup over the Naive method for TPC-H Queries Q4, Q5, Q12, Q14, Q17, and Q19. The graph compares different methods: Naive, SIMD, BL, Bit-sliced, BW/H, and BW/V. Speedup values range from 0 to 30.]
Outline

• Motivation & Introduction
• BitWeaving/V
• BitWeaving/H
• Evaluations
• Conclusions
Conclusions
Conclusions

BitWeaving: A new method to use all the bits in a processor word gainfully.
Conclusions

BitWeaving: A new method to use all the bits in a processor word gainfully.

Two flavors: BitWeaving/H and BitWeaving/V.
Conclusions

BitWeaving: A new method to use all the bits in a processor word gainfully.

Two flavors: BitWeaving/H and BitWeaving/V.

BitWeaving are faster than state-of-the-art scan methods, in some cases by an order of magnitude.
Q & A

• Thanks