ADVANCED DATABASE SYSTEMS

Data Formats & Encoding II

03

Andy Pavlo
CMU 15-721
Spring 2024
Carnegie Mellon University

LAST CLASS

Storage Models (NSM, DSM, PAX)

Open-Source Data File Formats
→ File Meta-Data
→ Format Layout
→ Type System
→ Encoding Schemes
→ Block Compression
→ Zone Maps + Bloom Filters
→ Nested Data (Shredding vs. Presence)

NESTED DATA

Real-world data sets often contain semi-structured objects (e.g., JSON, Protobufs).

A file format will want to encode the contents of these objects as if they were regular columns.

Approach #1: Record Shredding

Approach #2: Length+Presence Encoding

DREMEL: A DECADE OF INTERACTIVE SQL ANALYSIS AT WEB SCALE
VLDB 2020

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
   Code: 'en-gb'
   Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Language.Code

value	r	d
en-us	0	2

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Language.Code

value	r	d
en-us	0	2

Name.Language.Country

value	r	d
us	0	3

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Language.Code

value	r	d
en-us	0	2
en	1	2

Name.Language.Country

value	r	d
us	0	3

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Language.Code

value	r	d
en-us	0	2
en	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Url

value	r	d
http://A	0	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
   Code: 'en-gb'
   Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1
en-gb	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
   Code: 'en-gb'
   Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1
en-gb	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1
gb	1	3

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
   Code: 'en-gb'
   Country: 'gb'
```

Shredded Columns

DocID

value	r	d
10	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1
en-gb	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1
gb	1	3

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 20
Name:
  Url: 'http://C'
```

Shredded Columns

DocID

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1
en-gb	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1
gb	1	3

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 20
Name:
  Url: 'http://C'
```

Shredded Columns

DocID

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1
en-gb	1	2

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1
gb	1	3

Source: Sergey Melnik

NESTED DATA: SHREDDING

Store paths in nested structure as separate columns with additional meta-data about paths.

Definition Level: How many optional elements are defined in the path to an attribute.

Repetition Level: How many times a structure has been repeated.

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 20
Name:
  Url: 'http://C'
```

Shredded Columns

DocID

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Name.Language.Code

value	r	d
en-us	0	2
en	1	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	1	2
NULL	1	1
gb	1	3
NULL	0	1

Source: Sergey Melnik

NESTED DATA: LENGTH+PRESENCE

Store paths in nested structure as separate columns but maintain additional columns to track the number of entries at each path level (***length***) and whether a key exists at that level for a record (***presence***).

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
   Code: 'en-gb'
   Country: 'gb'
```

```
DocId: 20
Name:
  Url: 'http://C'
```

DocId

value	p
10	true
20	true

Name

len
3
1

Name.Url

value	p
http://A	true
http://B	true
	false
http://C	true

Name.Language

len
2
0
1
0

Name.Language.Code

value	p
en-us	true
en	true
en-gb	true

Name.Language.Country

value	p
us	true
	false
gb	true

Source: Sergey Melnik

CRITIQUES OF EXISTING FORMATS

Variable-sized Runs

→ Not SIMD friendly.

Eager Decompression

→ No random access if using block compression.

Dependencies Between Adjacent Values

→ Examples: Delta Encoding, RLE

Vectorization Portability

→ ISAs (versions, vendor) have different SIMD capabilities.

Source: Azim Afroozeh

TODAY'S AGENDA

BtrBlocks (TUM)

FastLanes (CWI)

BitWeaving (Wisconsin)

BTRBLOCKS

PAX-based file format with more aggressive ***nested encoding schemes*** than Parquet / ORC.

Uses a greedy algorithm to select the best encoding for a column chunk (based on sample) and then recursively tries to encode outputs of that encoding.
→ No naïve block compression (Snappy, zstd)

Store a file's meta-data separately from the data.

BTRBLOCKS: EFFICIENT COLUMNAR COMPRESSION FOR DATA LAKES
SIGMOD 2023

BTRBLOCKS: ENCODING SCHEMES

RLE / One Value

Frequency Encoding

FOR + Bitpacking

Dictionary Encoding

Pseudodecimals

Fast Static Symbol Table (FSST)

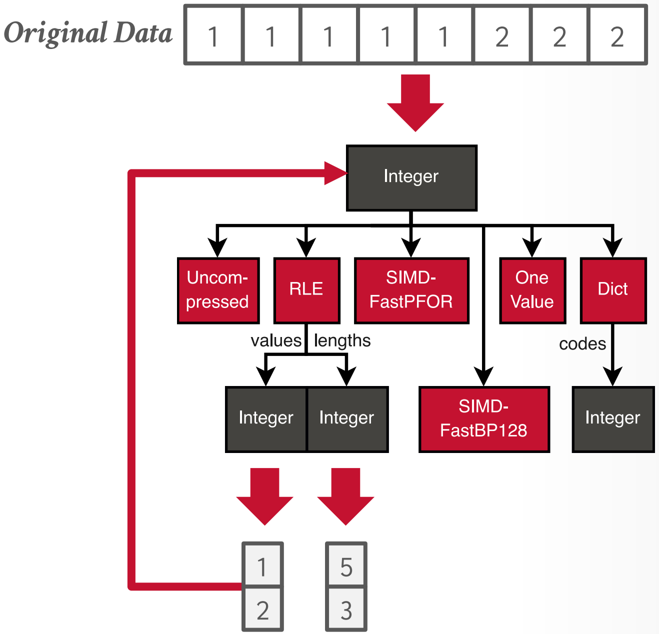
Roaring Bitmaps for NULLs + Exceptions

BTRBLOCKS: ENCODING SELECTION

Collect a sample from the data and then try out all viable encoding schemes. Repeat for three rounds.

Instead of sampling individual values, BtrBlocks selects multiple small runs from non-overlapping random positions.

→ For 64k values, it uses 10 runs of 64 values (1% sample size).

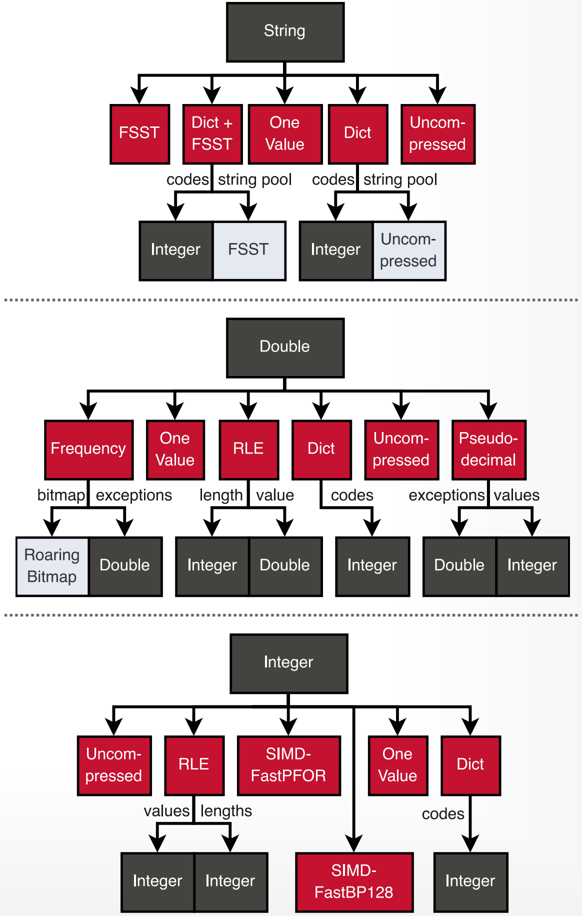


BTRBLOCKS: ENCODING SELECTION

Collect a sample from the data and then try out all viable encoding schemes. Repeat for three rounds.

Instead of sampling individual values, BtrBlocks selects multiple small runs from non-overlapping random positions.

→ For 64k values, it uses 10 runs of 64 values (1% sample size).



Source: Maximilian Kuschewski

BTRBLOCKS: ENCODING SCHEMES

RLE / One Value

Frequency Encoding

FOR + Bitpacking

Dictionary Encoding

Pseudodecimals

Fast Static Symbol Table (FSST)

Roaring Bitmaps for NULLs + Exceptions

FSST

String encoding scheme that supports random access without decompressing previous entries.

Replace frequently occurring substrings (up to 8 bytes) with 1-byte codes.

Uses a "perfect" hash table scheme for fast look-up of symbols without conditionals / loops.

→ Construct table using evolutionary algorithm that simply replaces entries if occupied.

FSST: FAST RANDOM ACCESS STRING COMPRESSION
VLDB 2020

ROARING BITMAPS

Bitmap index that switches which data structure to use for a range of values based local density of bits.

→ Dense chunks are stored using uncompressed bitmaps.
→ Sparse chunks use bitpacked arrays of 16-bit integers.

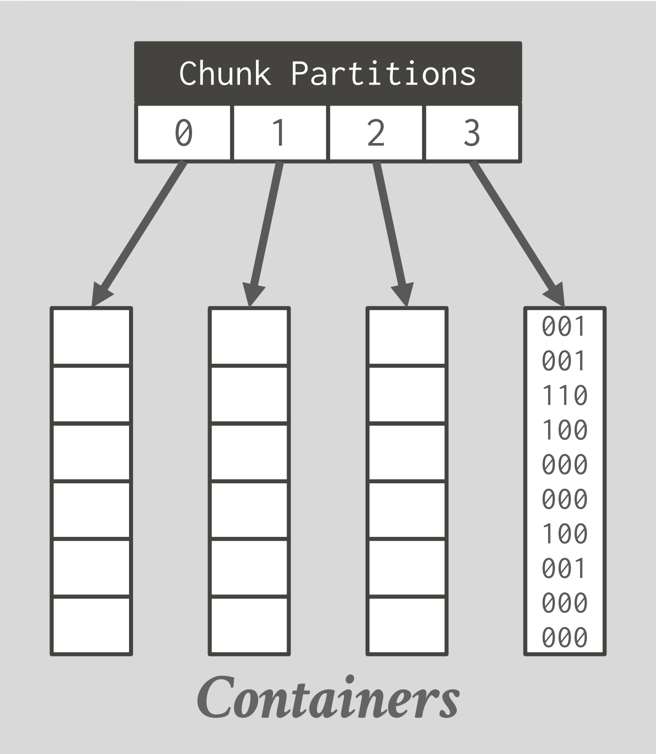
Dense chunks can be further compressed with RLE.

There are many open-source implementations that are widely used in different DBMSs.

ClickHouse, influxdb, Apache Spark, Apache Lucene, SirixDB, Weaviate, Hive, pilosa, pinot

BETTER BITMAP PERFORMANCE WITH ROARING BITMAPS
SOFTWARE: PRACTICE AND EXPERIENCE 2015

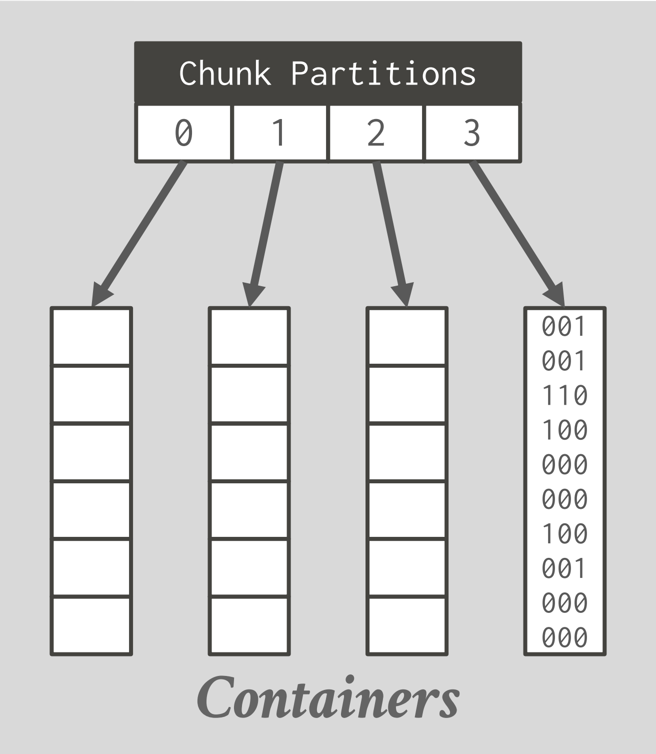
ROARING BITMAPS



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

→ Store **k** in the chunk's container.

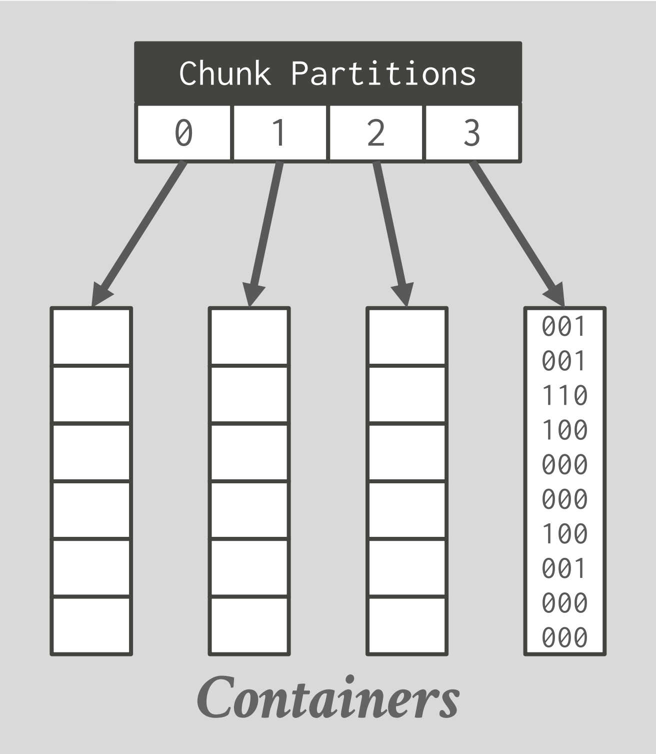
ROARING BITMAPS



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

ROARING BITMAPS

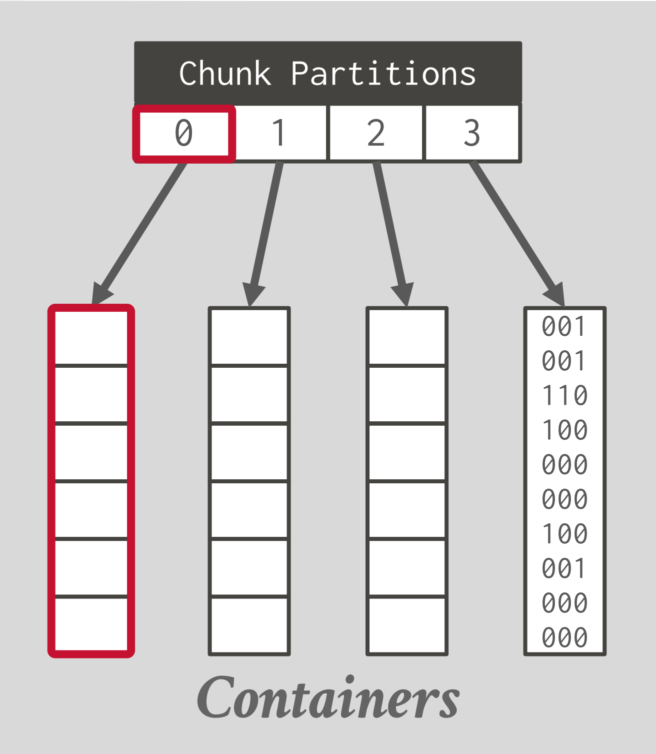


For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

ROARING BITMAPS



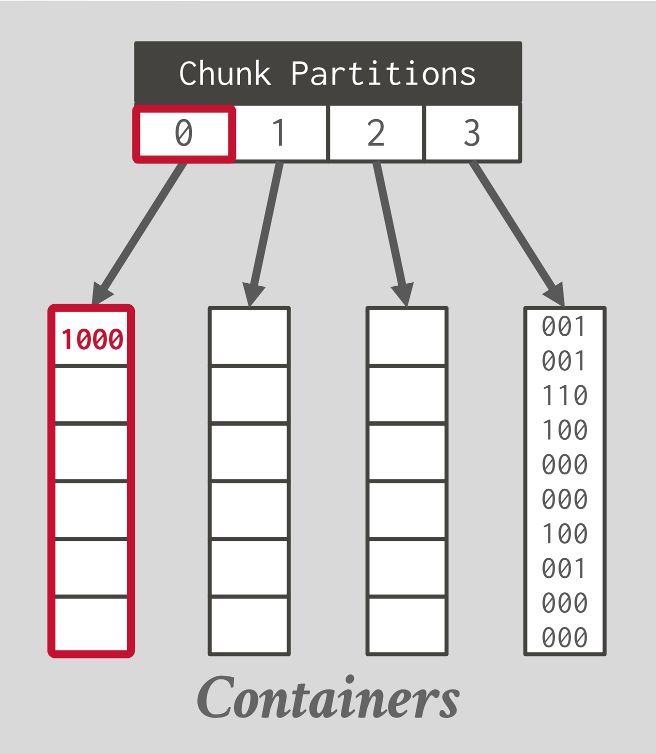
For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

$1000/2^{16}=0$

ROARING BITMAPS



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

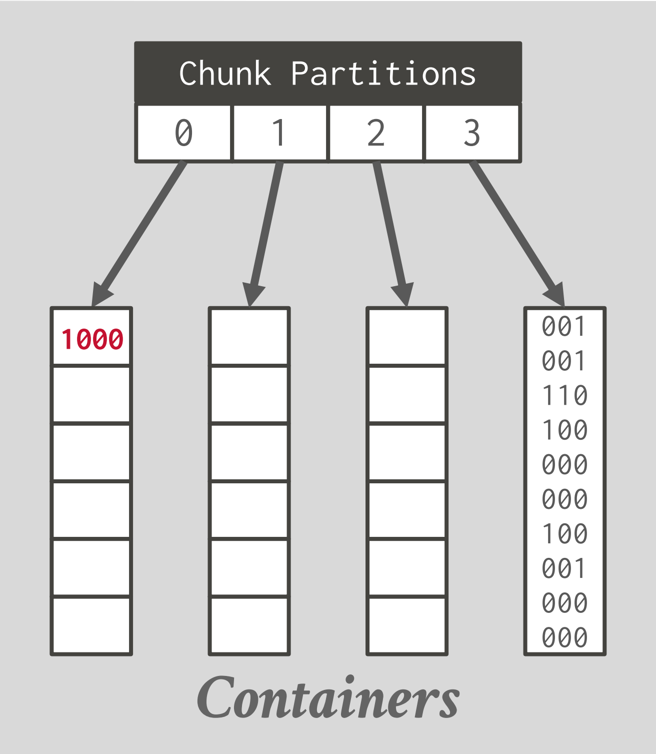
If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

$1000/2^{16}=0$

$1000\%2^{16}=1000$

ROARING BITMAPS



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

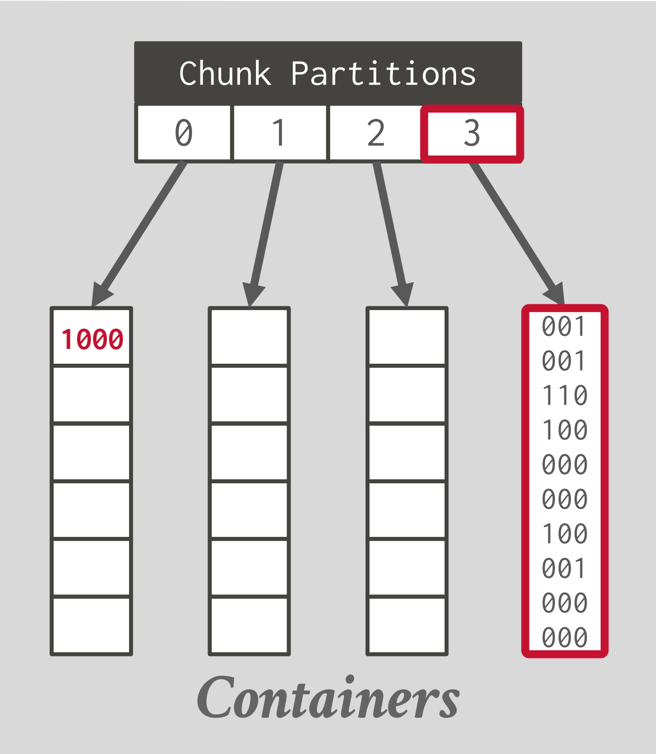
k=1000

$1000/2^{16}$=0

$1000\%2^{16}$=1000

k=199658

ROARING BITMAPS



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000

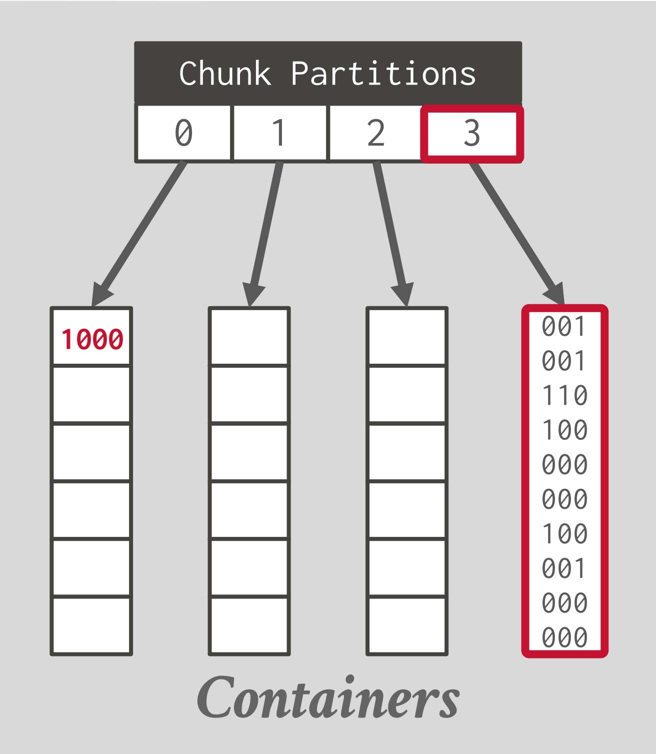
$1000/2^{16}=0$

$1000\%2^{16}=1000$

k=199658

$199658/2^{16}=3$

ROARING BITMAPS



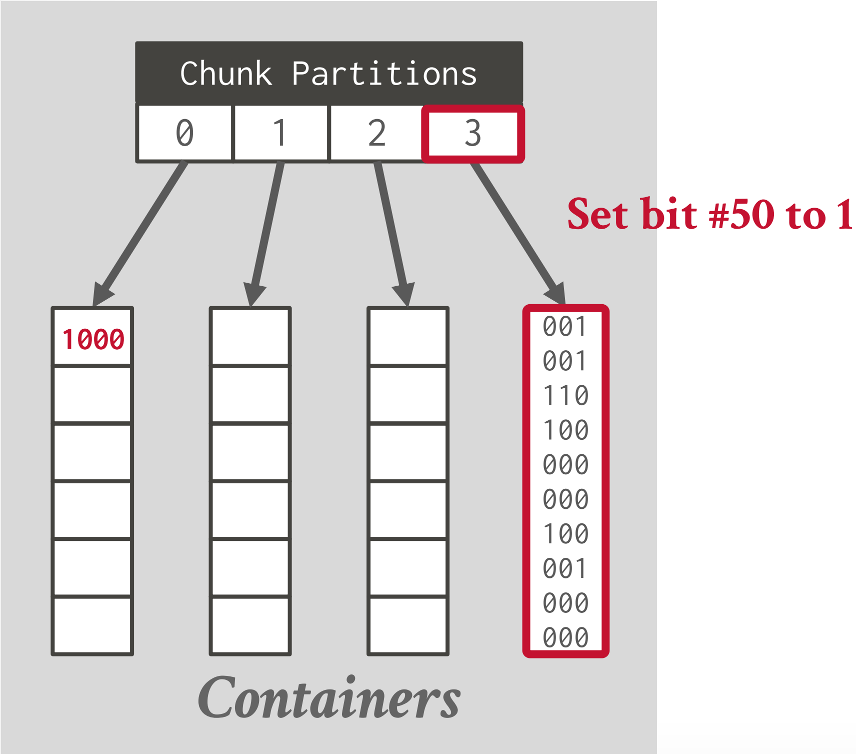
For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000
$1000/2^{16}=0$
$1000\%2^{16}=1000$

k=199658
$199658/2^{16}=3$
$199658\%2^{16}=50$

ROARING BITMAPS



For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

If # of values in container is less than 4096, store as array. Otherwise, store as Bitmap.

k=1000	**k=199658**
$1000/2^{16}=0$	**$199658/2^{16}=3$**
$1000\%2^{16}=1000$	**$199658\%2^{16}=50$**

OBSERVATION

BtrBlocks + **Parquet** + **ORC** generate variable-length runs of values.

→ This wastes cycles during decoding for both scalar + vectorized operations.

Parquet + **ORC** use Delta encoding where each tuple's value depends on the preceding tuple's value.

→ This is impractical to process with SIMD because you cannot pass data between lanes in the same register.

FASTLANES

Suite of encoding schemes that achieve better data parallelism thorough clever reordering of tuples to maximize useful work in SIMD operations.

Similar nested encoding as BtrBlocks:

→ Dictionary
→ FOR
→ Delta
→ RLE

To future proof format, they define a "virtual" ISA with 1024-bit SIMD registers.

THE FASTLANES COMPRESSION LAYOUT: DECODING > 100 BILLION INTEGERS PER SECOND WITH SCALAR CODE
VLDB 2023

UNIFIED TRANSPOSED LAYOUT

Reorder values in a column in a manner that improves the DBMS's ability to process them in an efficient, vectorized manner via SIMD.

→ Relational algebra is based on unordered sets, so users should not expect data to be ordered.

Algorithms defined in FastLanes' virtual 1024-bit SIMD ISA that can be emulated on AVX512 or scalar instructions.

UNIFIED TRANSPOSED LAYOUT

Original Data

B	B	C	C	C	C	B	B	B	A	A	A	A	A	A	A
15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

Run-Length Encoding

Run Values

B	C	B	A
3	*2*	*1*	*0*

Run Lengths

2	4	3	7
3	*2*	*1*	*0*

Delta Encoding

Run Values

B	C	B	A
3	*2*	*1*	*0*

3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

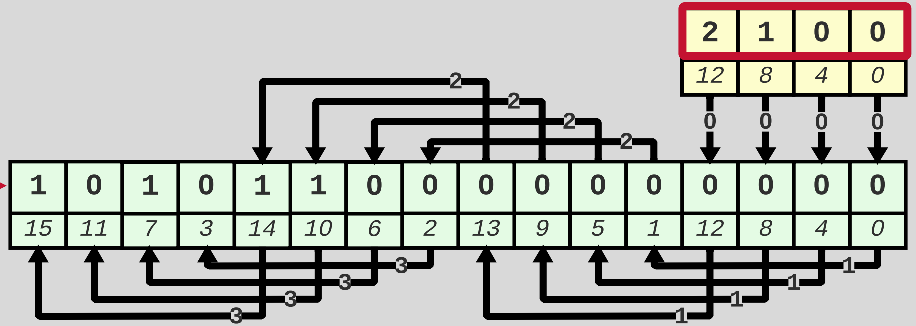
Index Vector

0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0
15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

0
0

Delta Encoded Vector

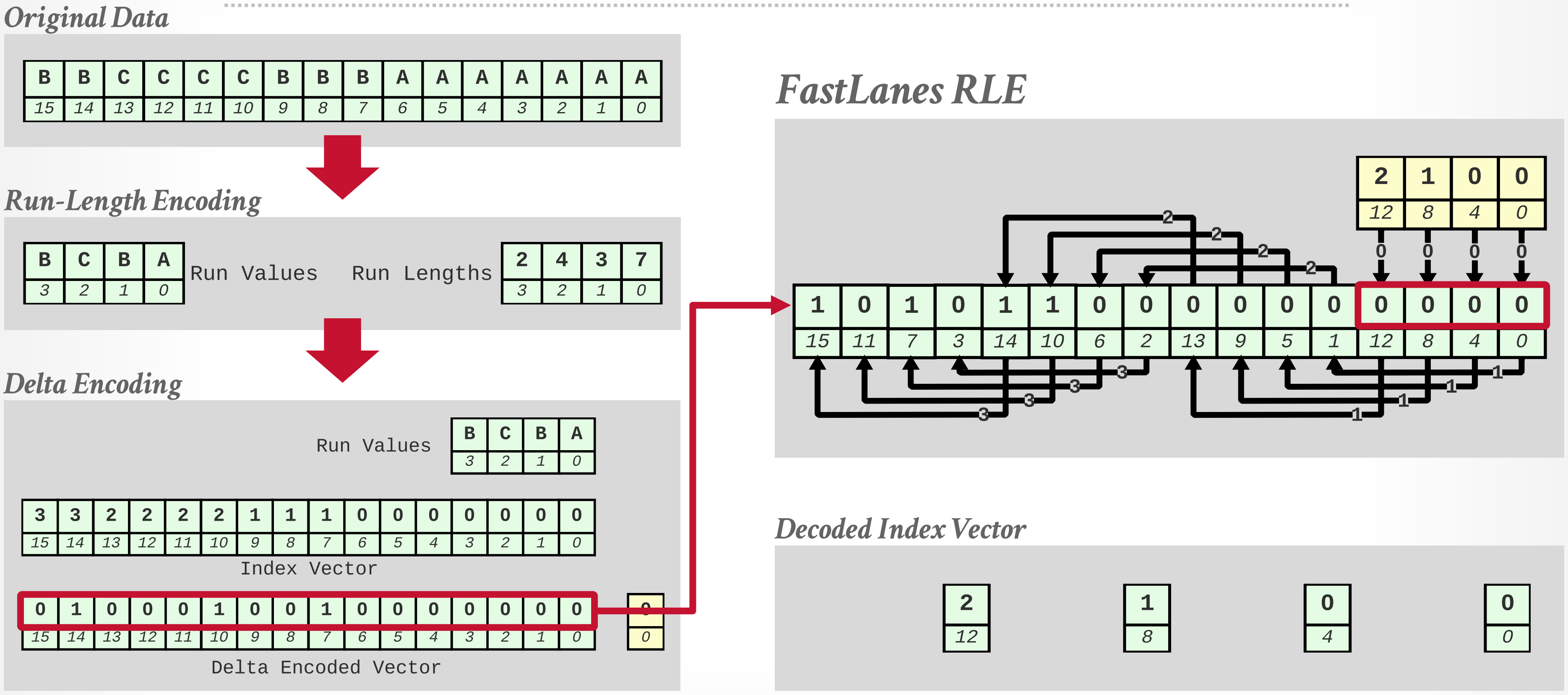
FastLanes RLE



Decoded Index Vector

Source: Azim Afroozeh

UNIFIED TRANSPOSED LAYOUT



Source: Azim Afroozeh

UNIFIED TRANSPOSED LAYOUT

Original Data

B	B	C	C	C	C	B	B	B	A	A	A	A	A	A	A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Run-Length Encoding

Run Values

B	C	B	A
3	2	1	0

Run Lengths

2	4	3	7
3	2	1	0

Delta Encoding

Run Values

B	C	B	A
3	2	1	0

3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

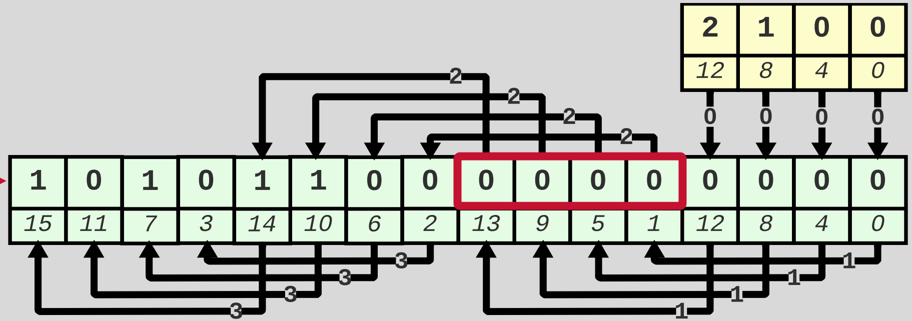
Index Vector

0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0
0

Delta Encoded Vector

FastLanes RLE



Decoded Index Vector

2	2
13	12

1	1
9	8

0	0
5	4

0	0
1	0

Source: Azim Afroozeh

UNIFIED TRANSPOSED LAYOUT

Original Data

B	B	C	C	C	C	B	B	B	A	A	A	A	A	A	A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Run-Length Encoding

Run Values

B	C	B	A
3	2	1	0

Run Lengths

2	4	3	7
3	2	1	0

Delta Encoding

Run Values

B	C	B	A
3	2	1	0

3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

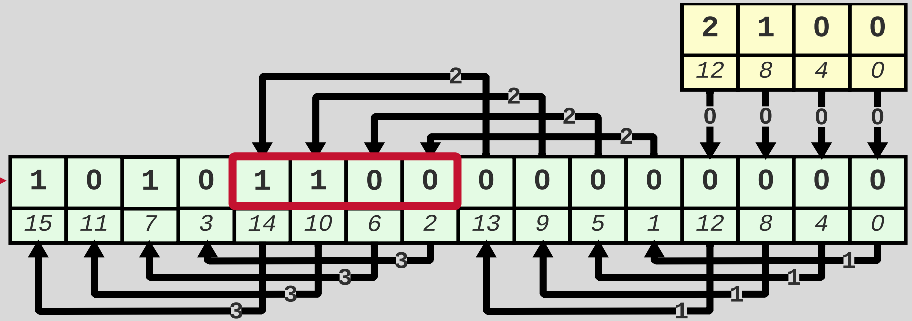
Index Vector

0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0
0

Delta Encoded Vector

FastLanes RLE



Decoded Index Vector

3	2	2
14	13	12

2	1	1
10	9	8

0	0	0
6	5	4

0	0	0
2	1	0

Source: Azim Afroozeh

UNIFIED TRANSPOSED LAYOUT

Original Data

B	B	C	C	C	C	B	B	B	A	A	A	A	A	A	A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Run-Length Encoding

Run Values

B	C	B	A
3	2	1	0

Run Lengths

2	4	3	7
3	2	1	0

Delta Encoding

Run Values

B	C	B	A
3	2	1	0

Index Vector

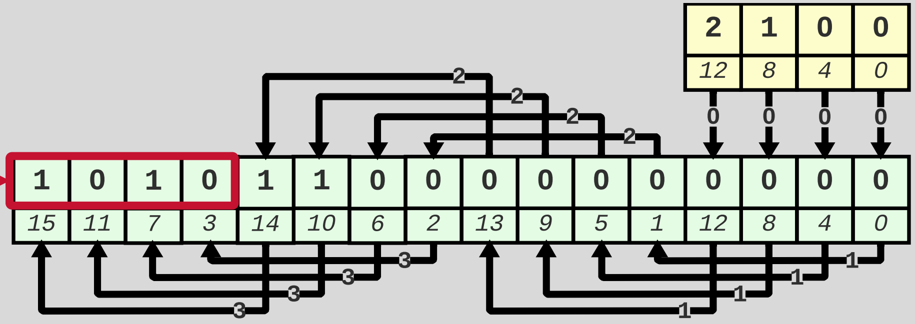
3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Delta Encoded Vector

0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0
0

FastLanes RLE



Decoded Index Vector

3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Source: Azim Afroozeh

OBSERVATION

The previous encoding schemes scan data by examining the entire value of each attribute (i.e., all the bits at the same time).

→ The DBMS cannot "short-circuit" comparisons integer types because CPU instructions operate on entire words.

OBSERVATION

The previous encoding schemes scan data by examining the entire value of each attribute (i.e., all the bits at the same time).

→ The DBMS cannot "short-circuit" comparisons integer types because CPU instructions operate on entire words.

What if a DBMS could scan a **subset** of each value's bits and then only check the rest bits if needed?

BIT-SLICED ENCODING

Original Data

id	zipcode
1	21042
2	15217
3	02903
4	90220
6	14623
7	53703

Bit-Slices

null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0																	

bin(21042)→ 00101001000110010

Source: Jignesh Patel

BIT-SLICED ENCODING

Original Data

id	zipcode
1	21042
2	15217
3	02903
4	90220
6	14623
7	53703

Bit-Slices

null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0

bin(21042)→ 00101001000110010

Source: Jignesh Patel

BIT-SLICED ENCODING

Original Data

id	zipcode
1	21042
2	15217
3	02903
4	90220
6	14623
7	53703

Bit-Slices

null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0
0	0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1
0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	1	1	1
0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	1	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	1	1
0	0	1	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1

Source: Jignesh Patel

BIT-SLICED ENCODING

Original Data

id	zipcode
1	21042
2	15217
3	02903
4	90220
6	14623
7	53703

Bit-Slices

null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0
0	0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1
0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	1	1	1
0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	1	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	1	1
0	0	1	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1

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

Walk each slice and construct a result bitmap.

0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1

Source: Jignesh Patel

BIT-SLICED ENCODING

Original Data

id	zipcode
1	21042
2	15217
3	02903
4	90220
6	14623
7	53703

Bit-Slices

null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0
0	0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1
0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	1	1	1
0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	1	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	1	1
0	0	1	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1

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

Walk each slice and construct a result bitmap.

Skip entries that have **1** in first 3 slices (16, 15, 14)

0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1

Source: Jignesh Patel

BIT-SLICED ENCODING

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

Example: **SUM(attr)** using Hamming Weight

→ First, count the number of **1**s in **slice**$_{17}$ and multiply the count by 2^{17}

→ Then, count the number of **1**s in **slice**$_{16}$ and multiply the count by 2^{16}

→ Repeat for the rest of slices…

Use the **POPCNT** instruction to efficiently count the number of bits set to **1** in a register.

BITWEAVING

Alternative encoding scheme for columnar databases that supports efficient predicate evaluation on compressed data using SIMD.

→ Order-preserving dictionary encoding.
→ Bit-level parallelization.
→ Only require common instructions (no scatter/gather)

Implemented in Wisconsin's QuickStep engine.

→ Became an Apache Incubator project in 2016 but then died in 2018.

BITWEAVING: FAST SCANS FOR MAIN MEMORY DATA PROCESSING
SIGMOD 2013

BITWEAVING STORAGE LAYOUTS

Approach #1: Horizontal

→ Row-oriented storage at the bit-level

Approach #2: Vertical

→ Column-oriented storage at the bit-level.
→ Similar to Bit-Slicing but with SIMD support.

HORIZONTAL STORAGE

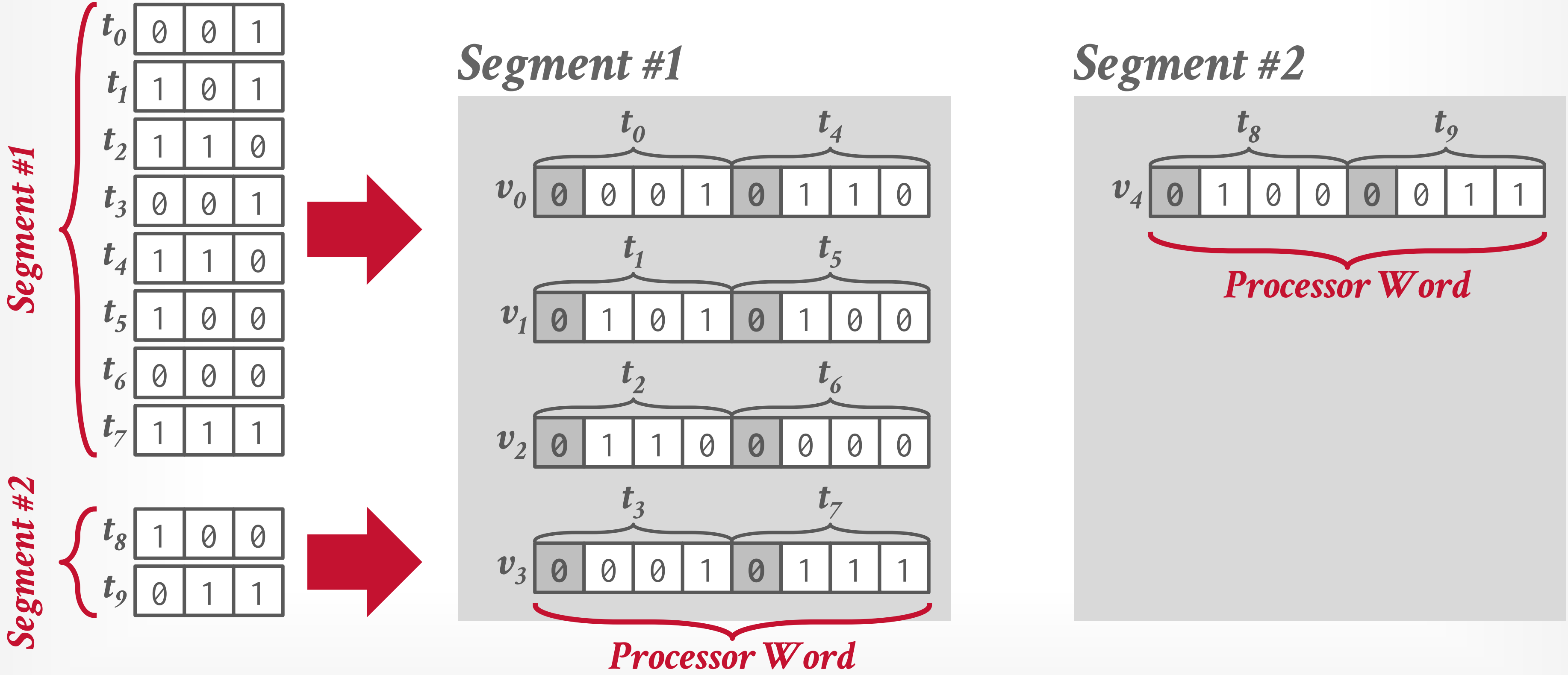
Segment #1

t_0	0	0	1	=1
t_1	1	0	1	=5
t_2	1	1	0	=6
t_3	0	0	1	=1
t_4	1	1	0	=6
t_5	1	0	0	=4
t_6	0	0	0	=0
t_7	1	1	1	=7

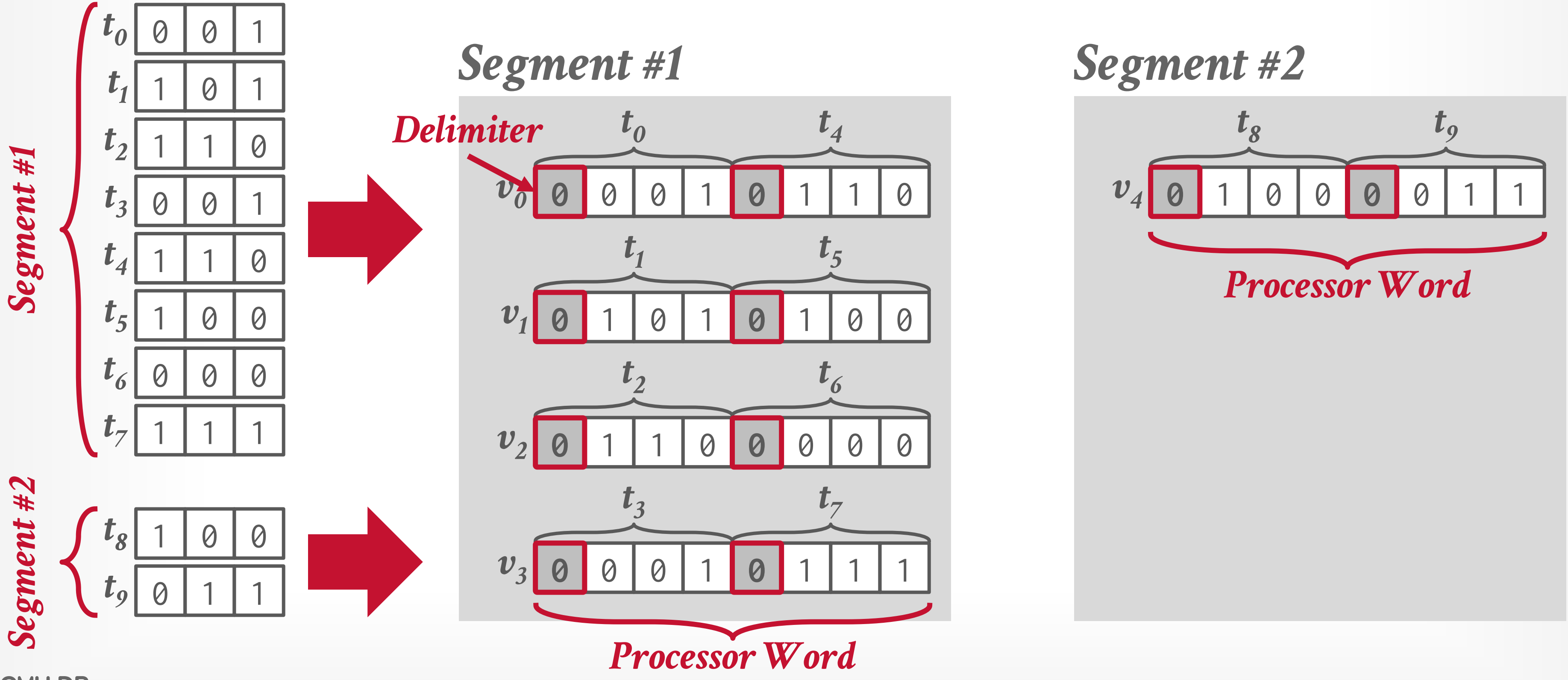
Segment #2

t_8	1	0	0	=4
t_9	0	1	1	=3

HORIZONTAL STORAGE

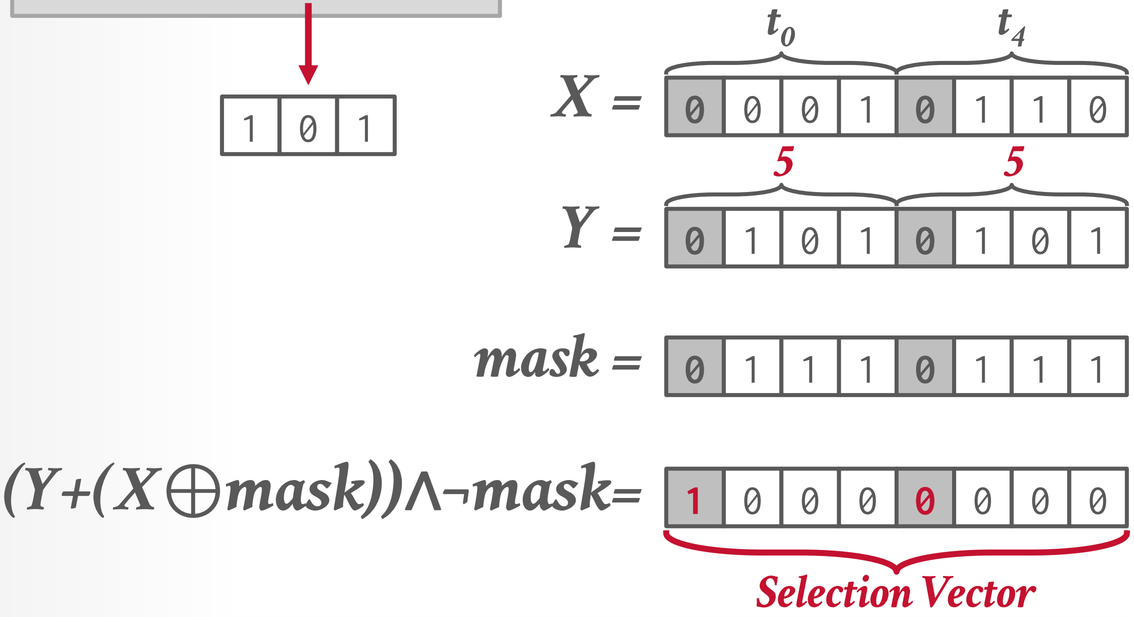


HORIZONTAL STORAGE



BITWEAVING/H: EXAMPLE

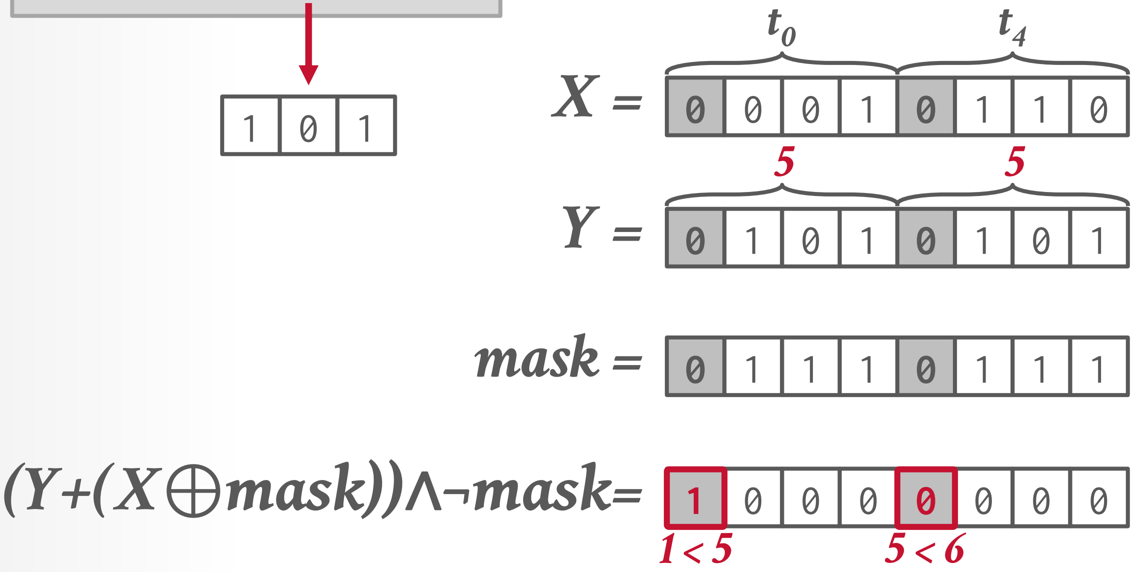
```
SELECT * FROM table
 WHERE val < 5
```



Source: Jignesh Patel

BITWEAVING/H: EXAMPLE

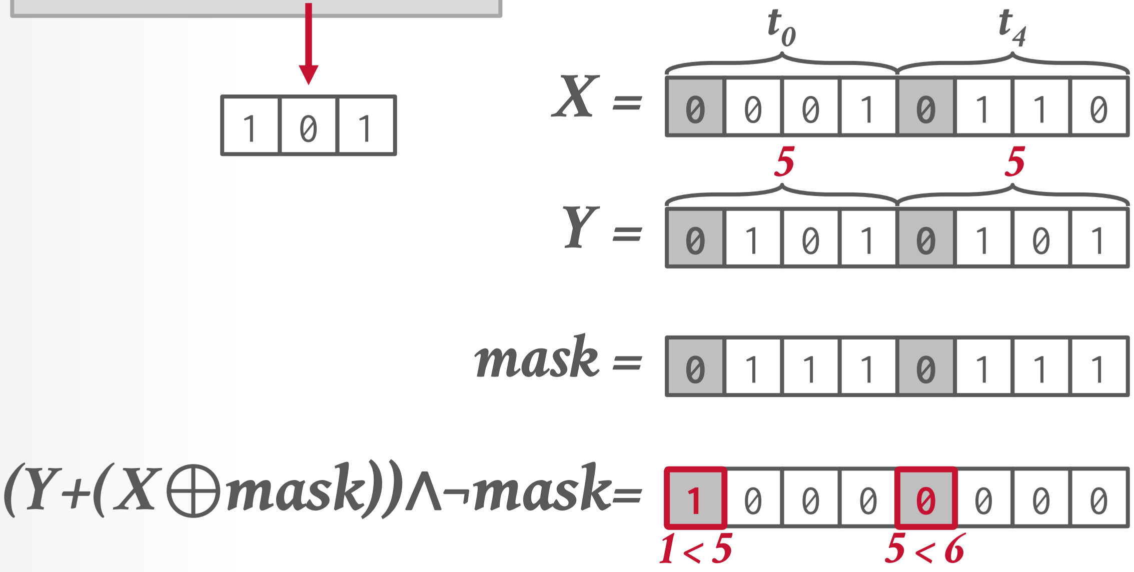
```
SELECT * FROM table
 WHERE val < 5
```



Source: Jignesh Patel

BITWEAVING/H: EXAMPLE

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



Only requires three instructions to evaluate a single word.

W

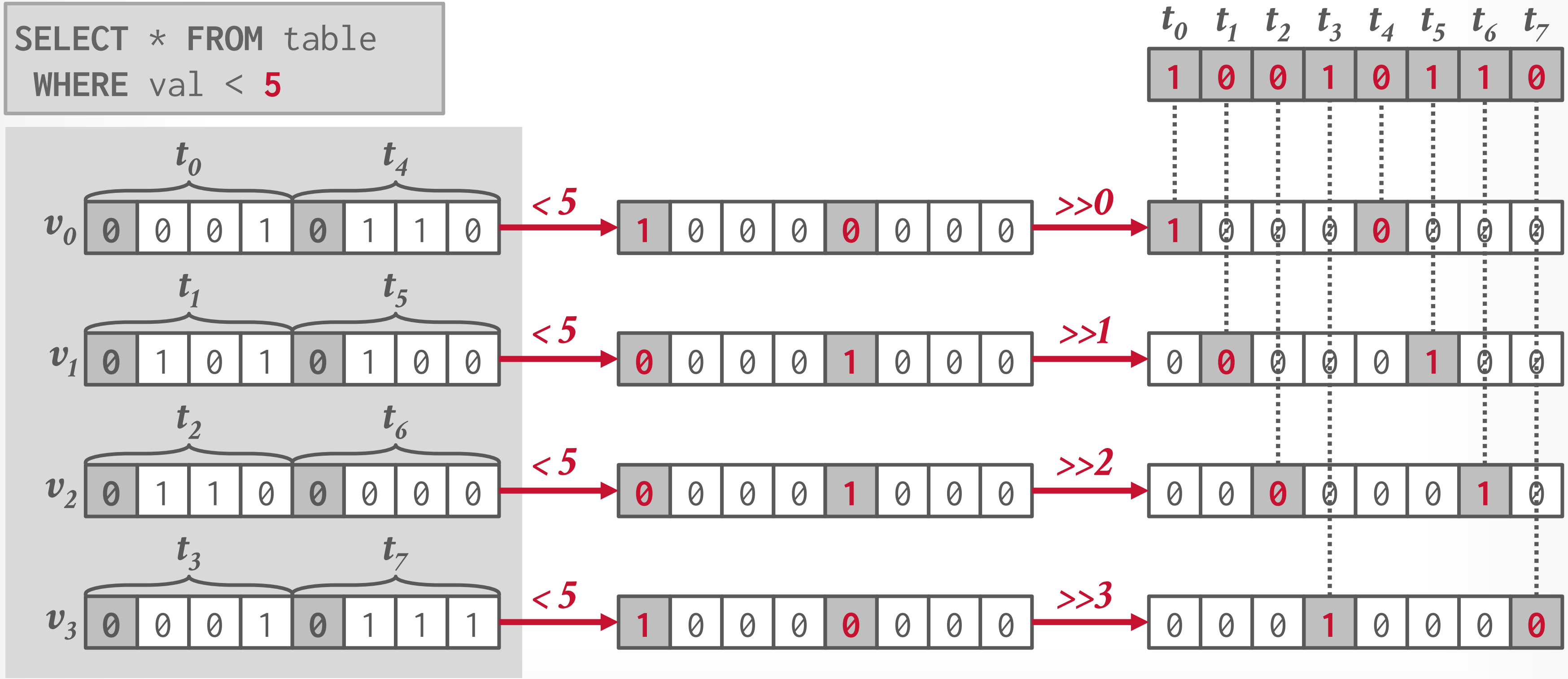
orks on any word size and encoding length.

Paper contains algorithms for other operators.

Source: Jignesh Patel

BITWEAVING/H: EXAMPLE

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



Source: Jignesh Patel

SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate.

→ DBMS must convert it into column offsets.

Approach #1: Iteration

Approach #2: Pre-computed Positions Table

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Selection Vector	1	0	0	1	0	1	1	0

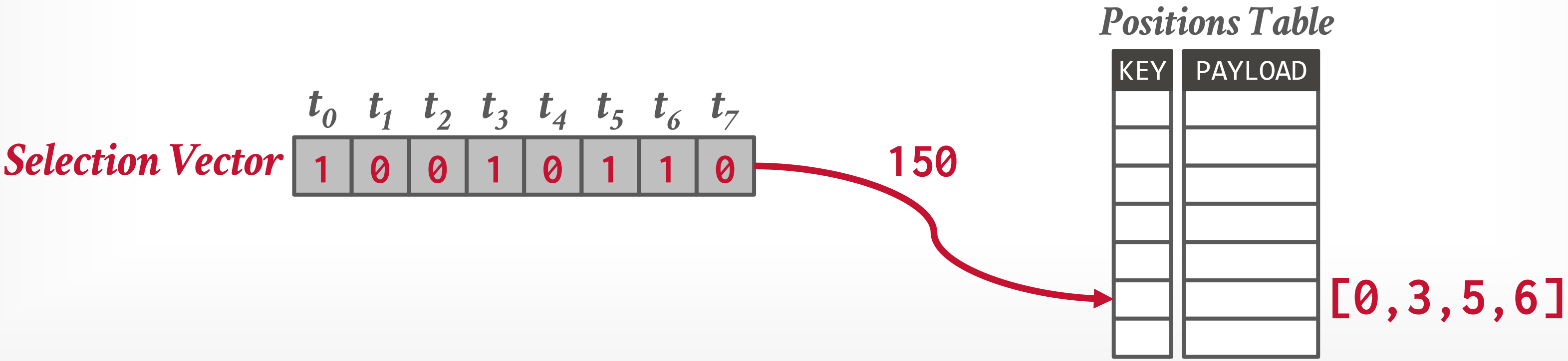
```
tuples = [ ]
for (i=0; i<n; i++) {
  if sv[i] == 1
    tuples.add(i);
}
```

SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate.
→ DBMS must convert it into column offsets.

Approach #1: Iteration

Approach #2: Pre-computed Positions Table



VERTICAL 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
Segment #2	t_8	1	0	0
	t_9	0	1	1

VERTICAL 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

Segment #2

t_8	1	0	0
t_9	0	1	1

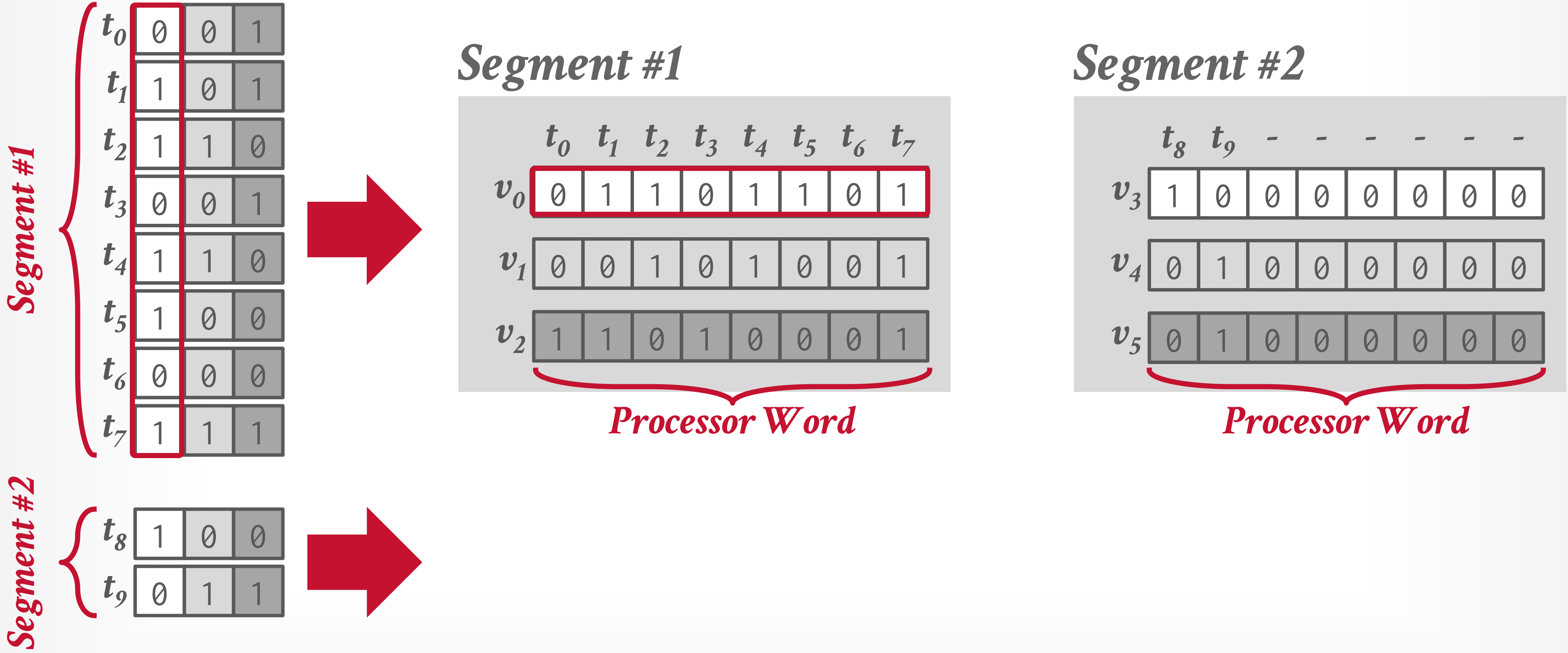
Segment #1

	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

Segment #2

	t_8	t_9	-	-	-	-	-	-
v_3	1	0	0	0	0	0	0	0
v_4	0	1	0	0	0	0	0	0
v_5	0	1	0	0	0	0	0	0

VERTICAL STORAGE



BITWEAVING/V: EXAMPLE

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

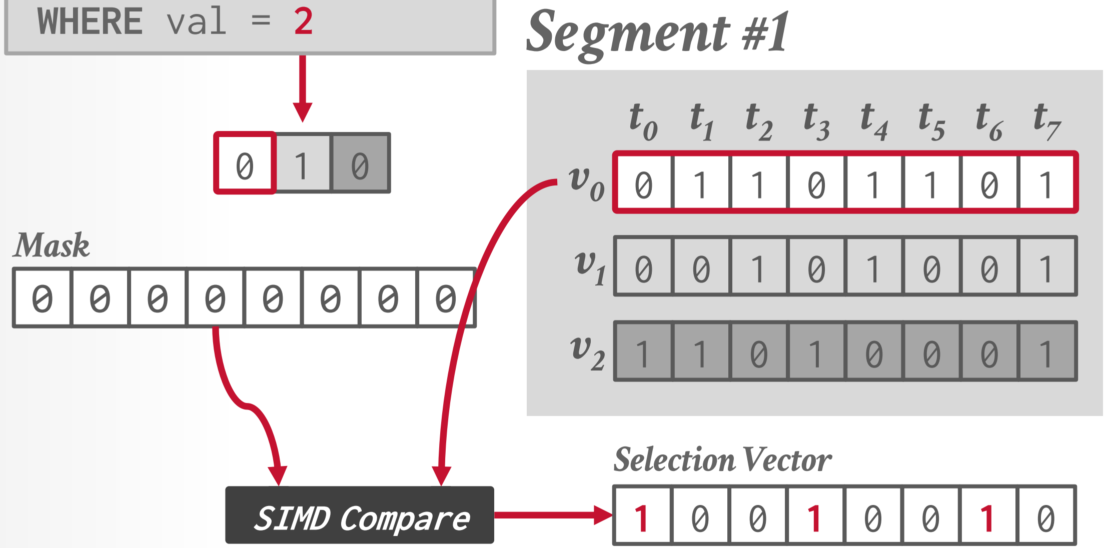
0	1	0

Segment #1

	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

BITWEAVING/V: EXAMPLE

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



BITWEAVING/V: EXAMPLE

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

0	1	0

Segment #1

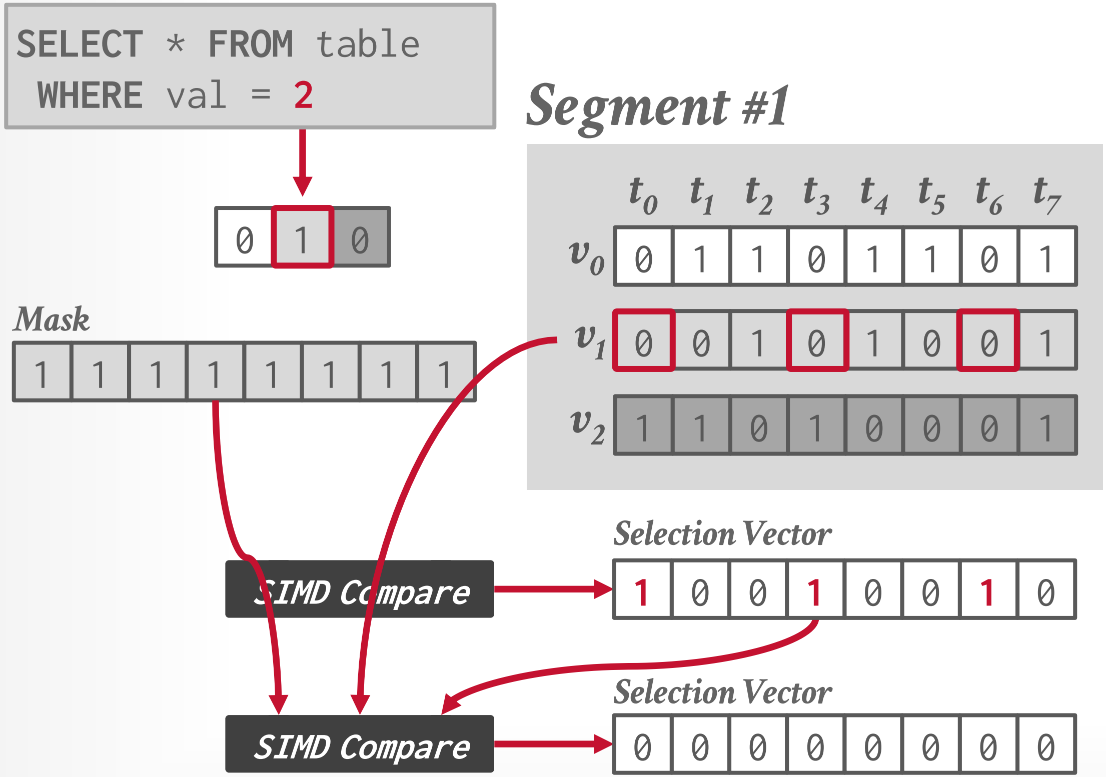
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	0	1	1	0	1	1	0	1
v_1	0	0	1	0	1	0	0	1
v_2	1	1	0	1	0	0	0	1

SIMD Compare

Selection Vector

1	0	0	1	0	0	1	0

BITWEAVING/V: EXAMPLE



DBMS can perform early pruning like Bit-Slicing.

S

kip the last vector because all bits in previous comparison are zero.

PARTING THOUGHTS

The last two lectures show why ***logical-physical data independence*** is one of the best parts of the relational model.

→ There are many strategies for representing data with unique compute-vs-storage trade-offs.
→ Applications can remain (mostly) oblivious to the low-details.

Data parallelism via SIMD is going be an important tool for us the entire semester.

NEXT CLASS

Project Proposals (5 minutes)

→ The two groups for each project topic will present one after the other.

→ The liaisons for each project topic should also present the proposed API separately.

Email me PDF of your slides + proposal documents before class.