04: Data Skipping

Andrew Crotty // CS497 // Fall 2023

Observation

OLTP DBMSs use indexes to find individual tuples without performing sequential scans.

→ Tree-based indexes (B+Trees) are meant for queries with low selectivity predicates.

→ Also need to accommodate incremental updates.

But OLAP queries usually don't need to find individual tuples and data files are read-only.

How can we speed up sequential scans?

Data Skipping

Approach #1: Approximate Queries (Lossy)

→ Execute queries on a sampled subset of the entire table to produce approximate results.

→ Examples: BlinkDB, Redshift, ComputeDB, XDB, Oracle, Snowflake, Google BigQuery, DataBricks

Approach #2: Data Pruning (Lossless)

→ Use auxiliary data structures for evaluating predicates to quickly identify portions of a table that the DBMS can skip instead of examining tuples individually.

→ DBMS must consider trade-offs between ***scope*** vs. ***filter efficacy***, ***manual*** vs. ***automatic***.

Data Skipping

Approach #1: Approximate Queries (Lossy)

→ Execute queries on a sampled subset of the entire table to produce approximate results.

→ Examples: BlinkDB, Redshift, ComputeDB, XDB, Oracle, Snowflake, Google BigQuery, DataBricks

Approach #2: Data Pruning (Lossless)

→ Use auxiliary data structures for evaluating predicates to quickly identify portions of a table that the DBMS can skip instead of examining tuples individually.

→ DBMS must consider trade-offs between ***scope*** vs. ***filter efficacy***, ***manual*** vs. ***automatic***.

Data Considerations

Predicate Selectivity
→ How many tuples will satisfy a query's predicates.

Skewness
→ Whether a column has all unique values or contains many repeated values.

Clustering / Sorting
→ Whether the table is pre-sorted on the attributes accessed in a query's predicates.

Today's Agenda

Zone Maps
Bitmap Indexes
Bit-Slicing
Bit-Weaving
Column Imprints
Column Sketches

Zone Maps

Pre-computed aggregates for a block of tuples. DBMS checks the zone map first to decide whether to access the block.
→ Originally ***Small Materialized Aggregates*** (SMA)
→ DBMS automatically maintains this meta-data.

Original Data

val
100
200
300
400
400

SMALL MATERIALIZED AGGREGATES: A LIGHT WEIGHT INDEX STRUCTURE FOR DATA WAREHOUSING
VLDB 1998

Zone Maps

Pre-computed aggregates for a block of tuples. DBMS checks the zone map first to decide whether to access the block.

→ Originally ***Small Materialized Aggregates*** (SMA)

→ DBMS automatically maintains this meta-data.

Original Data

val
100
200
300
400
400

Zone Map

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

SMALL MATERIALIZED AGGREGATES: A LIGHT WEIGHT INDEX STRUCTURE FOR DATA WAREHOUSING
VLDB 1998

Zone Maps

Pre-computed aggregates for a block of tuples. DBMS checks the zone map first to decide whether to access the block.

→ Originally ***Small Materialized Aggregates*** (SMA)

→ DBMS automatically maintains this meta-data.

```
SELECT * FROM table
 WHERE val > 600
```

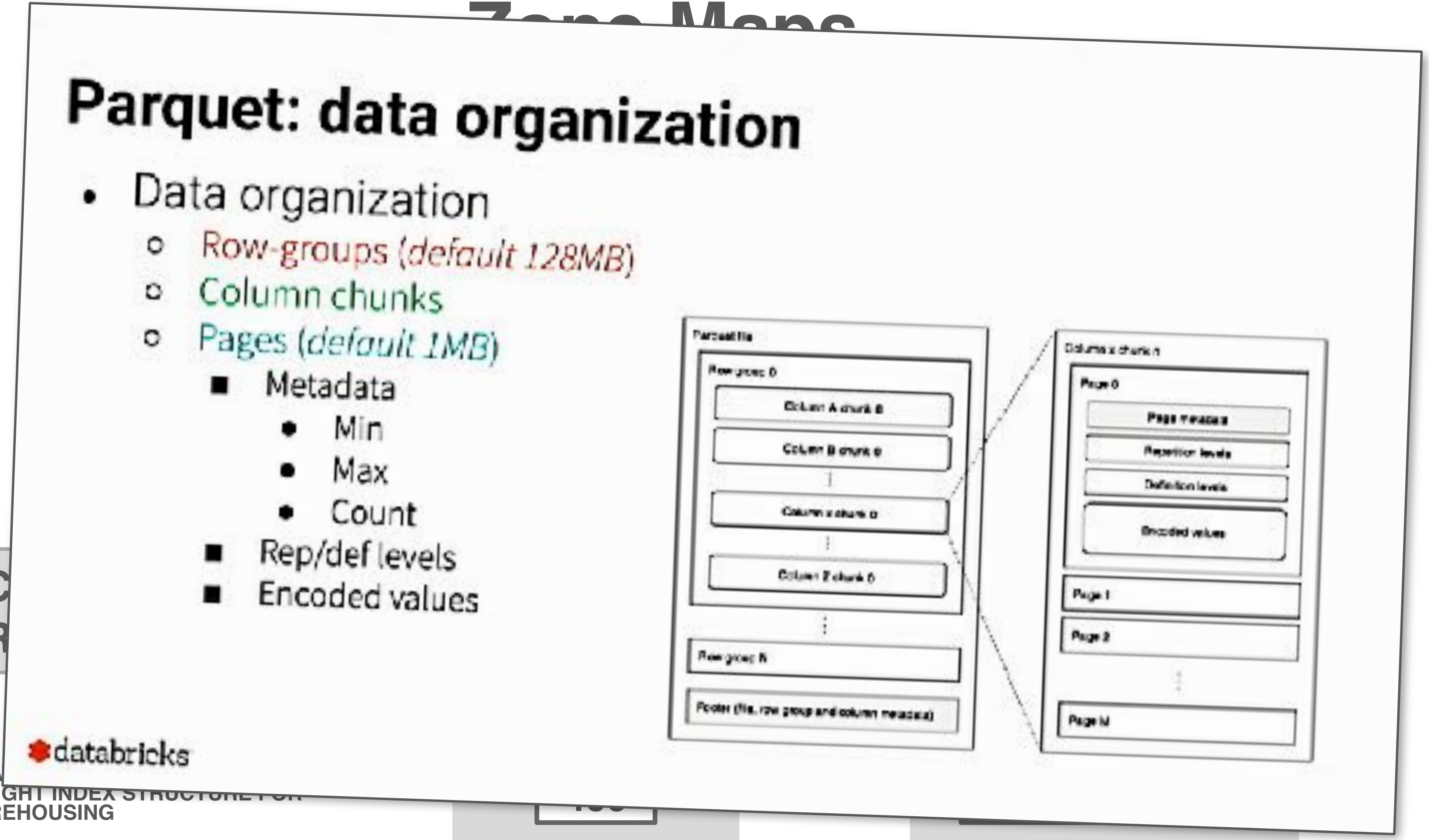
Original Data

val
100
200
300
400
400

Zone Map

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

SMALL MATERIALIZED AGGREGATES: A LIGHT WEIGHT INDEX STRUCTURE FOR DATA WAREHOUSING
VLDB 1998



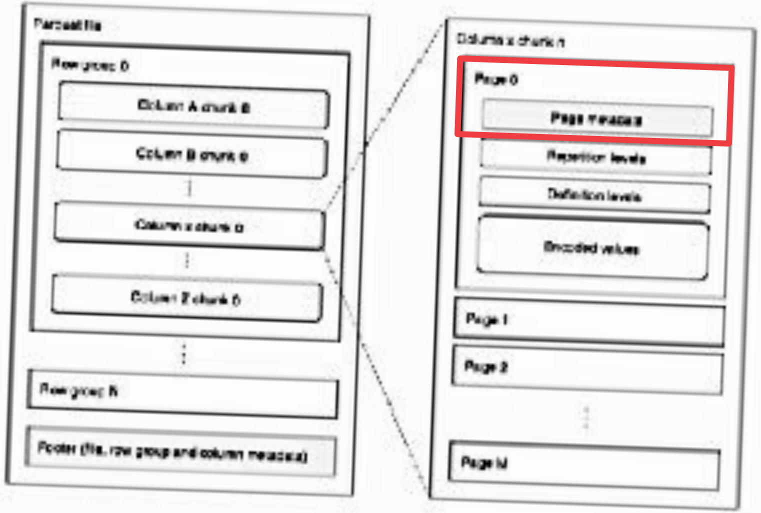
SELEC
WHER

SMALL MA
LIGHT WEIGHT INDEX STRUCTURE FOR
DATA WAREHOUSING
VLDB 1998

Zone Maps

Parquet: data organization

- Data organization
  - Row-groups (*default 128MB*)
  - Column chunks
  - Pages (*default 1MB*)
    - Metadata
      - Min
      - Max
      - Count
    - Rep/def levels
    - Encoded values



databricks

SELEC

WHER

SMALL MA

LIGHT WEIGHT INDEX STRUCTURE FOR

DATA WAREHOUSING

VLDB 1998

Observation

Trade-off between scope vs. filter efficacy.
→ If the scope is too large, then the zone maps will be useless.
→ If the scope is too small, then the DBMS will spend too much time checking zone maps.

Zone Maps are only useful when the target attribute's position and values are correlated.

Bitmap Indexes

Store a separate Bitmap for each unique value for an attribute where an offset in the vector corresponds to a tuple.

→ The i^{th} position in the Bitmap corresponds to the i^{th} tuple in the table.

Typically segmented into chunks to avoid allocating large blocks of contiguous memory.

→ Example: One per row group in PAX.

MODEL 204 ARCHITECTURE AND PERFORMANCE
HIGH PERFORMANCE TRANSACTION SYSTEMS 1987

Bitmap Indexes

Original Data

id	lit?
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Bitmap Indexes

Original Data

id	lit?
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Bitmap Indexes

Original Data

id	lit?
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

	lit?	
id	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

Bitmap Indexes

Original Data

id	lit?
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Compressed Data

id	lit?	
	Y	N
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Take the intersection of three bitmaps to find matching tuples.

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Take the intersection of three bitmaps to find matching tuples.

Assume we have 10 million tuples.
43,000 zip codes in the US.
→ **10000000 × 43000 = 53.75GB**

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Take the intersection of three bitmaps to find matching tuples.

Assume we have 10 million tuples.
43,000 zip codes in the US.
→ **10000000 × 43000 = 53.75GB**

Bitmap Indexes: Example

```
CREATE TABLE customer_dim
(
  id INT PRIMARY KEY,
  name VARCHAR(32),
  email VARCHAR(64),
  address VARCHAR(64),
  zip_code INT
);
```

```
SELECT id, email FROM customer_dim
 WHERE zip_code IN
(15216,15217,15218);
```

Take the intersection of three bitmaps to find matching tuples.

Assume we have 10 million tuples.
43,000 zip codes in the US.
→ **10000000 × 43000 = 53.75GB**

This is **wasteful** because most entries in the bitmaps will be zeros.
→ Original: **10000000 × 32-bit = 40MB**

Bitmap Indexes: Design Choices

Encoding Scheme

→ How to represent and organize data in a Bitmap.

Compression

→ How to reduce the size of sparse Bitmaps.

Bitmap Indexes: Design Choices

Encoding Scheme
→ How to represent and organize data in a Bitmap.

Compression
→ How to reduce the size of sparse Bitmaps.

Bitmap Indexes: Encoding

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

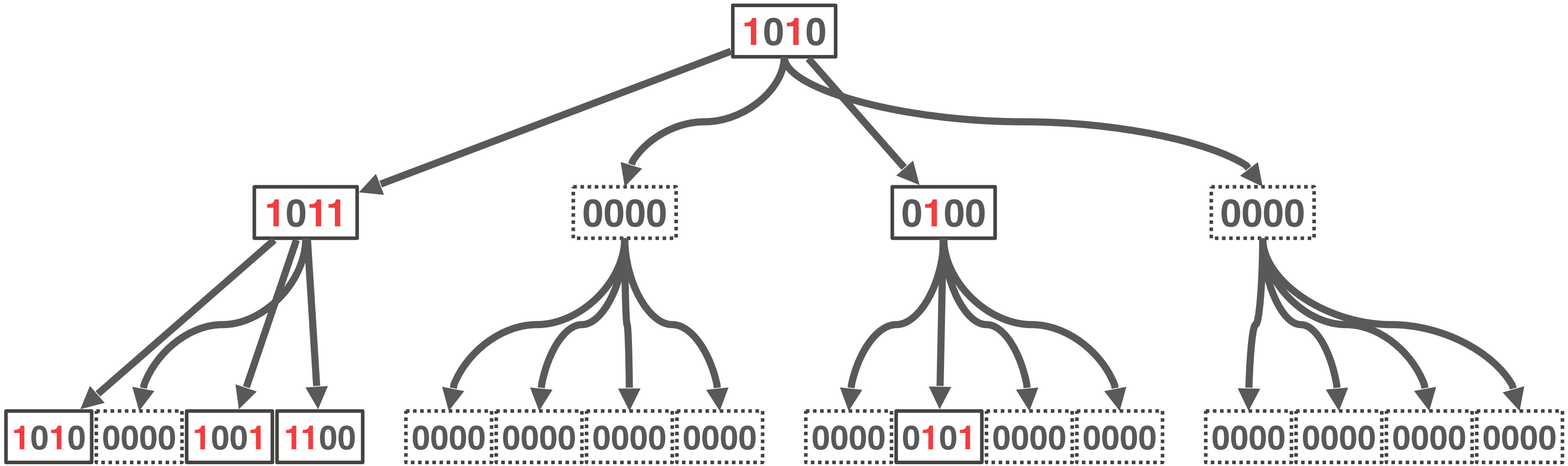
Approach #2: Range Encoding
→ Use one Bitmap per interval instead of one per value.

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

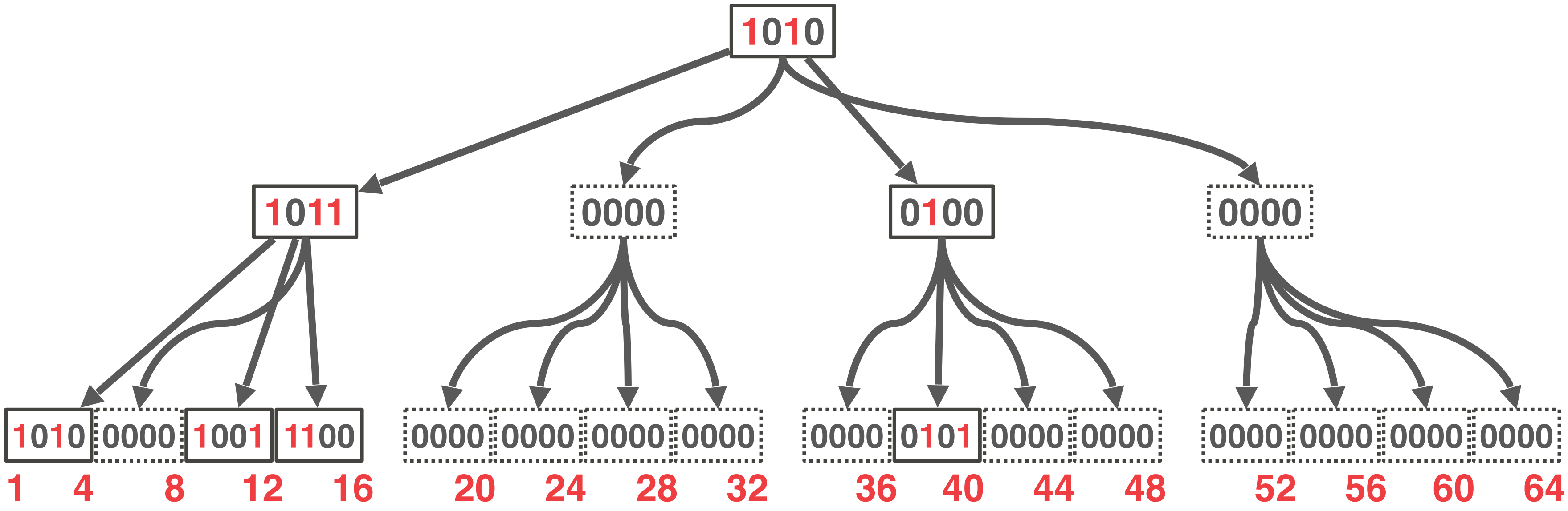
Offsets: **1, 3, 9, 12, 13, 14, 38, 40**



HIERARCHICAL BITMAP INDEX: AN EFFICIENT AND SCALABLE INDEXING TECHNIQUE FOR SET-VALUED ATTRIBUTES
ADVANCES IN DATABASES AND INFORMATION SYSTEMS 2003

Hierarchical Encoding

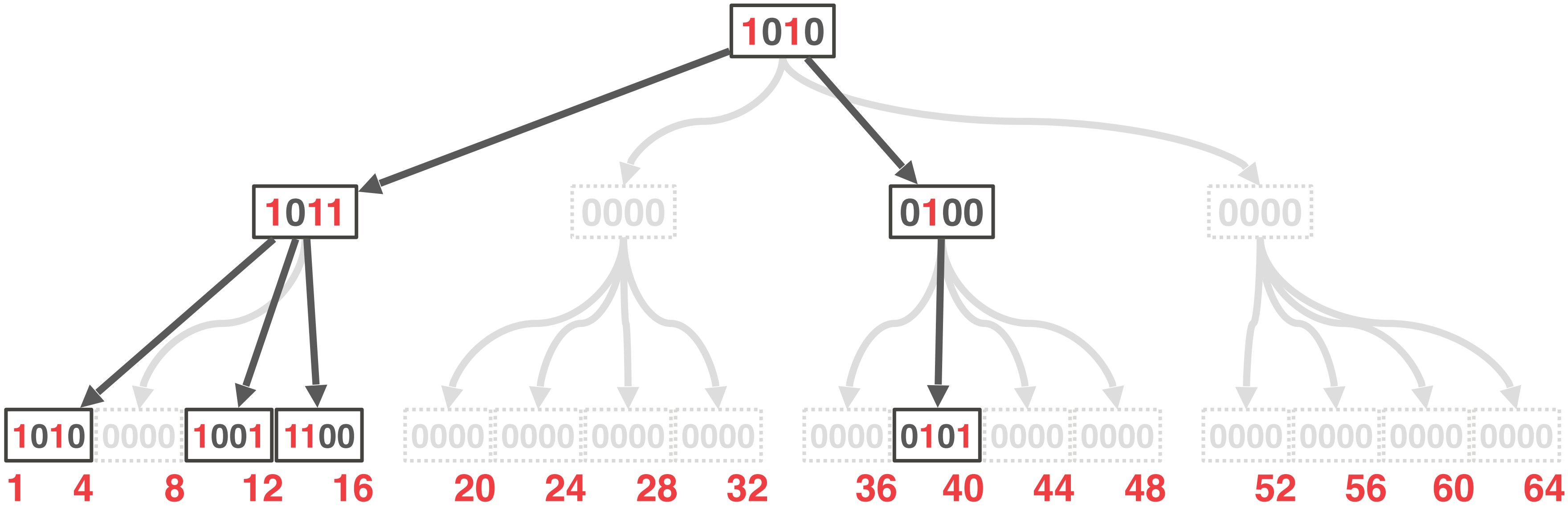
Offsets: **1, 3, 9, 12, 13, 14, 38, 40**



HIERARCHICAL BITMAP INDEX: AN EFFICIENT AND SCALABLE INDEXING TECHNIQUE FOR SET-VALUED ATTRIBUTES
ADVANCES IN DATABASES AND INFORMATION SYSTEMS 2003

Hierarchical Encoding

Offsets: **1, 3, 9, 12, 13, 14, 38, 40**



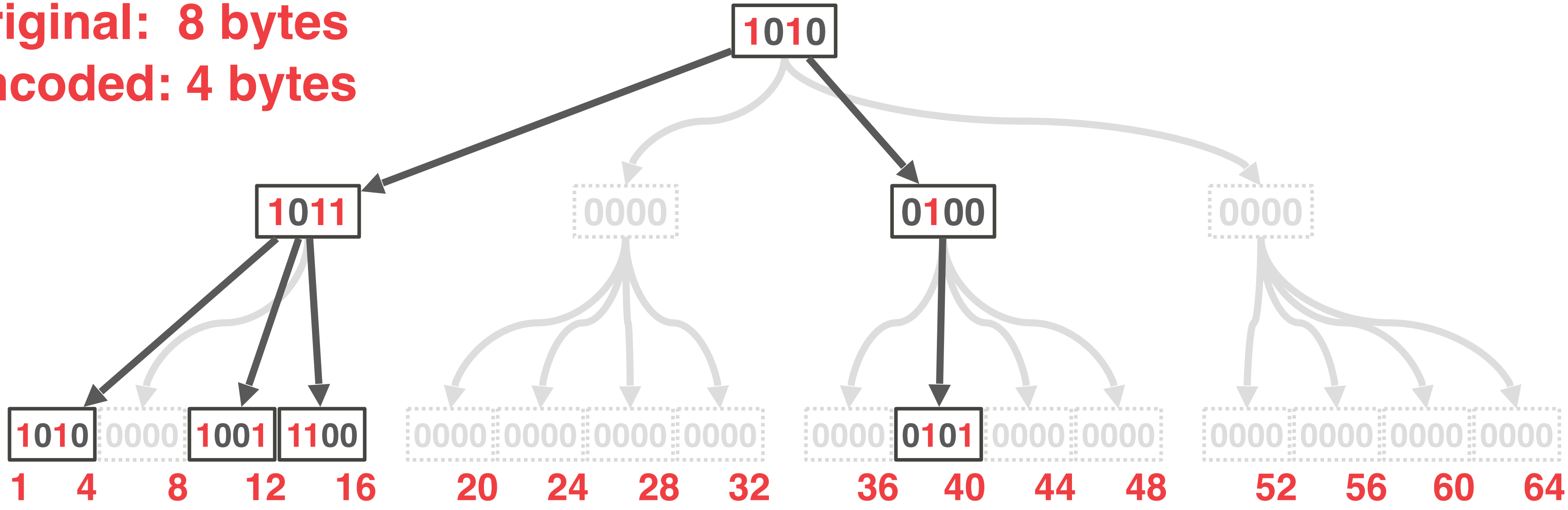
HIERARCHICAL BITMAP INDEX: AN EFFICIENT AND SCALABLE INDEXING TECHNIQUE FOR SET-VALUED ATTRIBUTES
ADVANCES IN DATABASES AND INFORMATION SYSTEMS 2003

Hierarchical Encoding

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

Original: 8 bytes

Encoded: 4 bytes



HIERARCHICAL BITMAP INDEX: AN EFFICIENT AND SCALABLE INDEXING TECHNIQUE FOR SET-VALUED ATTRIBUTES

ADVANCES IN DATABASES AND INFORMATION SYSTEMS 2003

Bit-Sliced Encoding

Original Data

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

Bit-Slices

Source: [Jignesh Patel]

Bit-Sliced Encoding

Original Data

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

Bit-Slices

Source: Jignesh Patel

Bit-Sliced Encoding

Original Data

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

Bit-Slices

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

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																	

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																	

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

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

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.

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.

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)

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…

Intel added **`POPCNT`** SIMD instruction in 2008.

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)

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

Alternative storage layout for colu
databases that is designed for effi
predicate evaluation on compress
using SIMD.
→ Order-preserving dictionary encoding
→ Bit-level parallelization.
→ Only require common instructions (n

Implemented in Wisconsin's Quic
→ Became an Apache Incubator projec
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

Horizontal Storage

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_8	1	0	0
t_9	0	1	1

Horizontal Storage

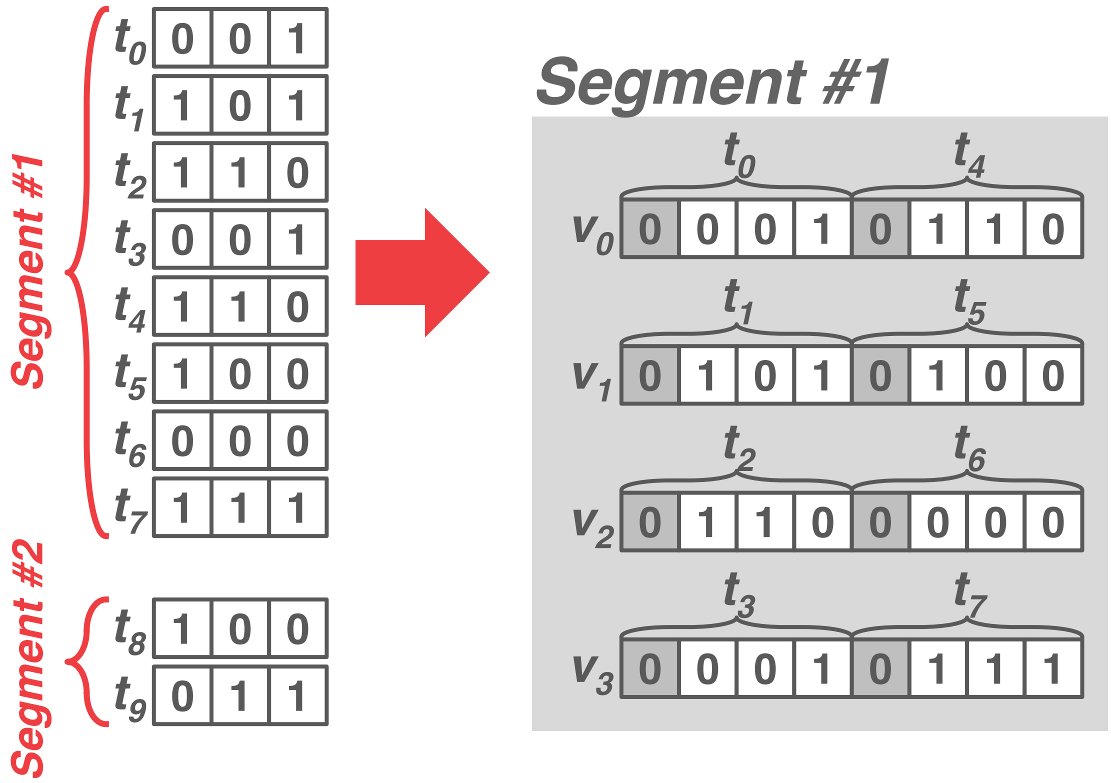
t_0	0	0	1	=1
t_1	1	0	1	=5
t_2	1	1	0	=6
t_3	0	0	1	=1
t_4	1	1	0	=6
t_5	1	0	0	=4
t_6	0	0	0	=0
t_7	1	1	1	=7

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

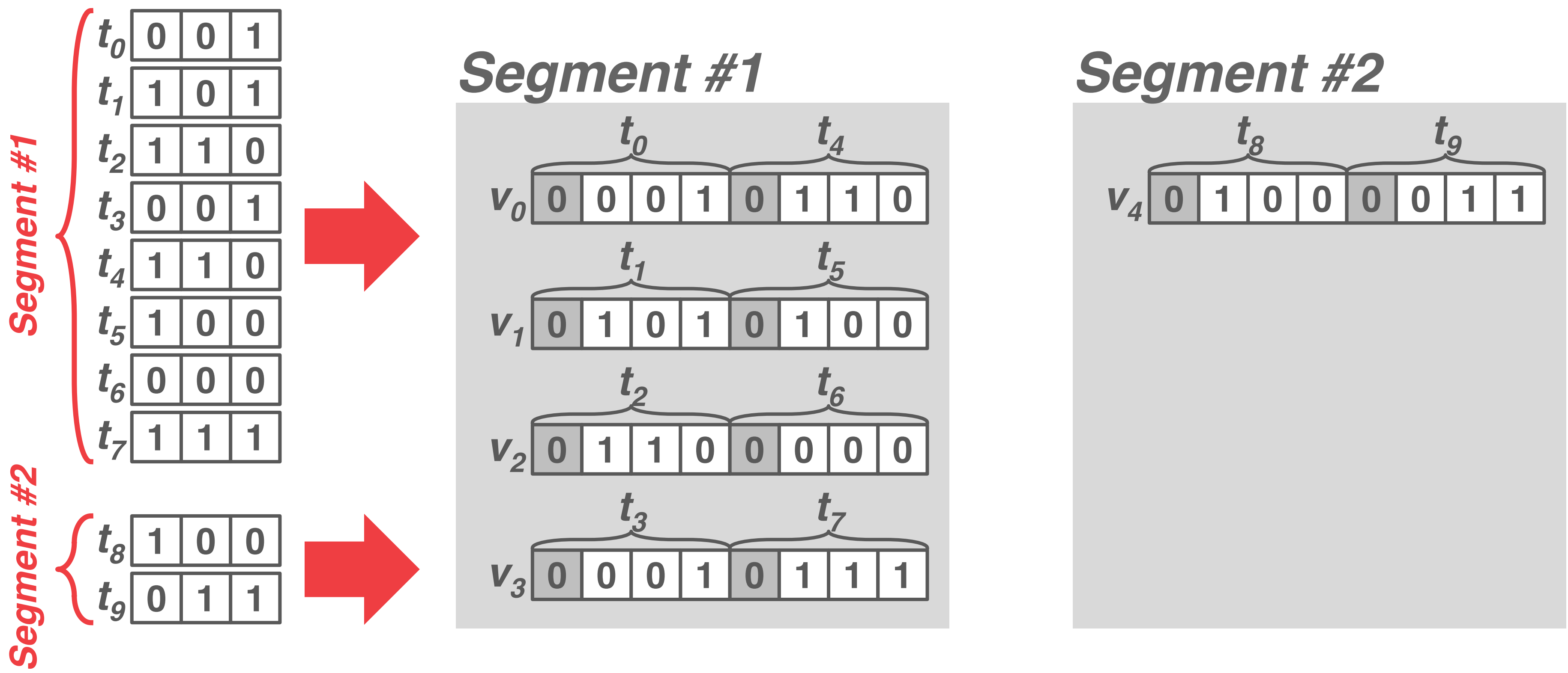
Horizontal Storage

Segment	Tuple				Value
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

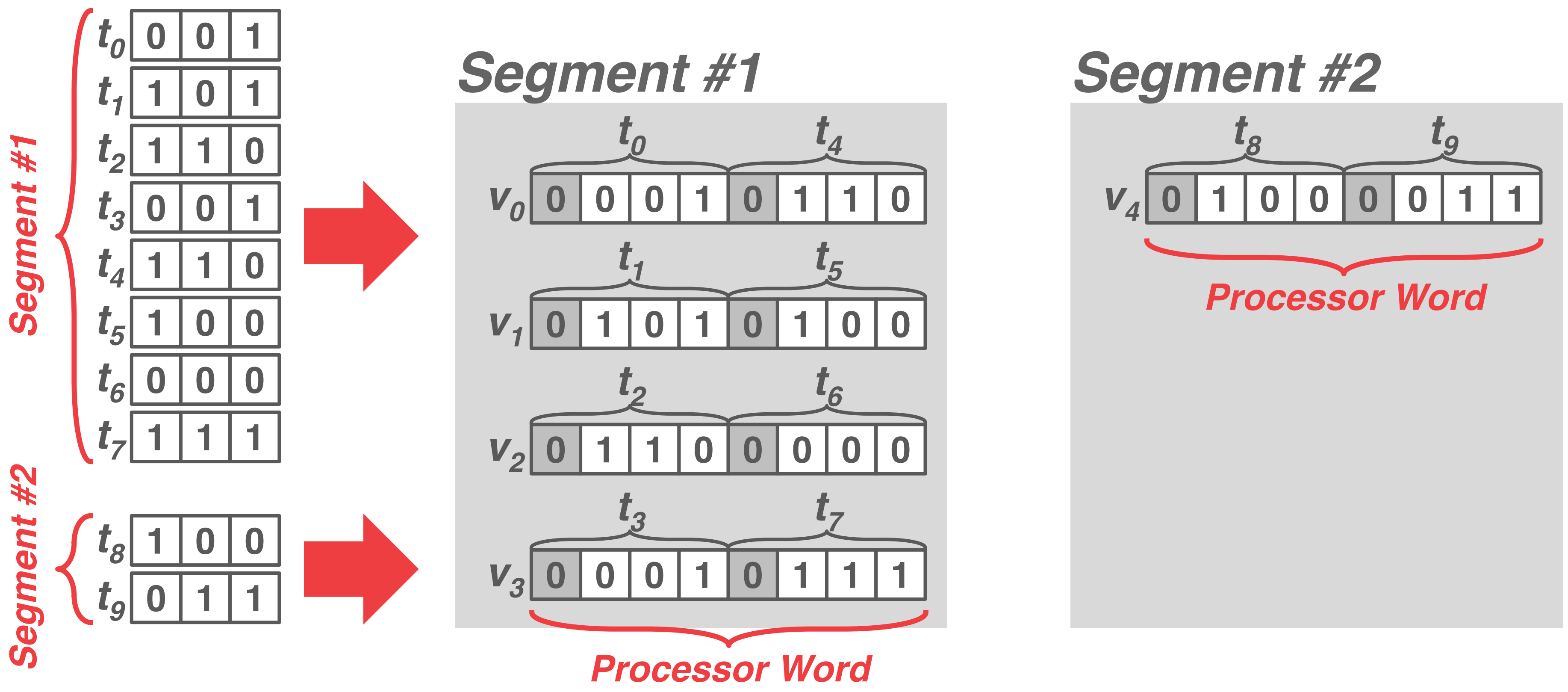
Horizontal Storage



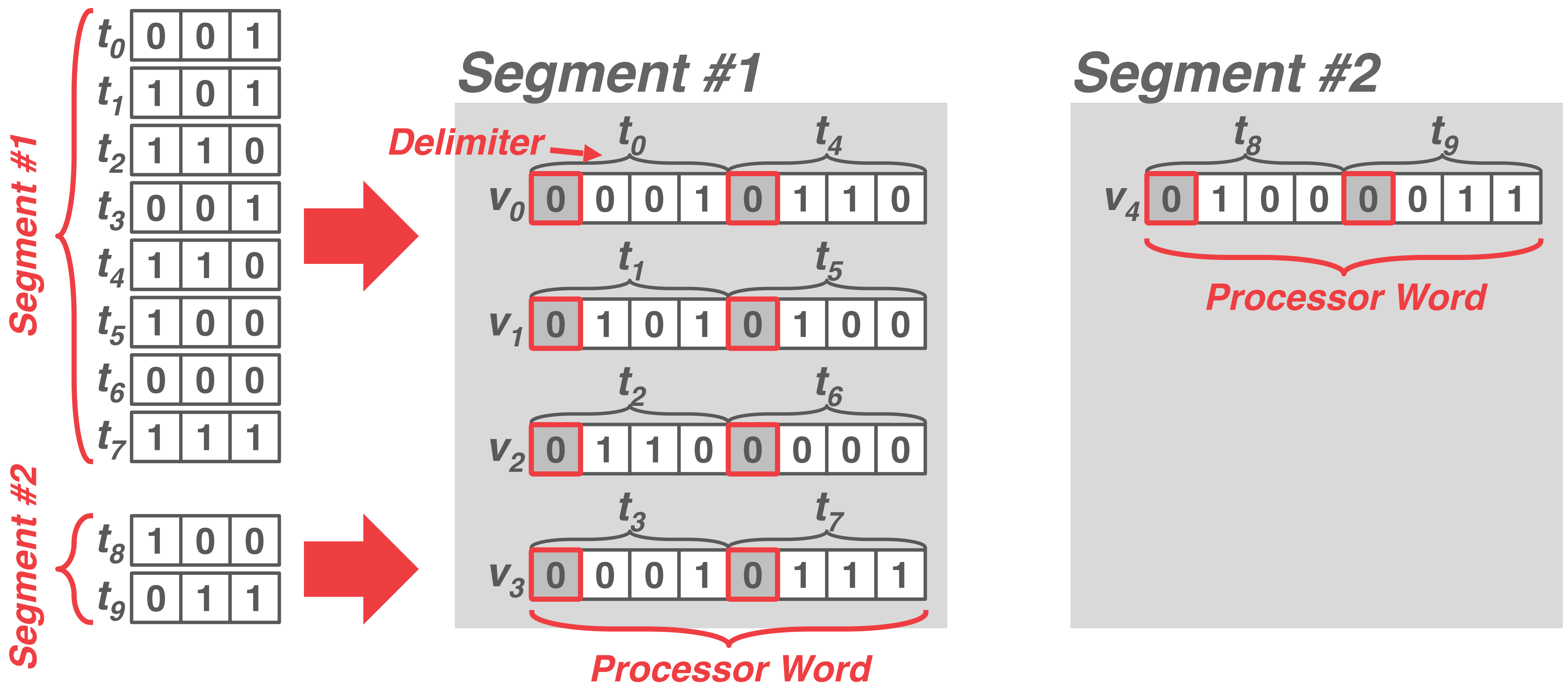
Horizontal Storage



Horizontal Storage



Horizontal Storage



Bitweaving/H – Example

SELECT * FROM table
WHERE val < 5

X =	0	0	0	1	0	1	1	0
Y =	0	1	0	1	0	1	0	1

Source: [Jignesh Patel]

Bitweaving/H – Example

SELECT * FROM table
WHERE val < 5

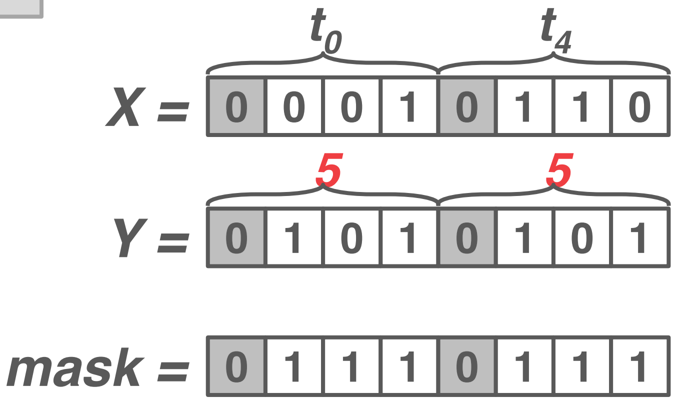
$$X = \overbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{1}}^{t_0}\overbrace{\boxed{0}\boxed{1}\boxed{1}\boxed{0}}^{t_4}$$

$$Y = \overbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}}^{5}\overbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}}^{5}$$

Source: [Jignesh Patel]

Bitweaving/H – Example

SELECT * FROM table
WHERE val < 5



Source: [Jignesh Patel]

Bitweaving/H – Example

SELECT * FROM table
WHERE val < 5

$$X = \overbrace{0\;0\;0\;1}^{t_0}\;\overbrace{0\;1\;1\;0}^{t_4}$$

$$Y = \overbrace{0\;1\;0\;1}^{5}\;\overbrace{0\;1\;0\;1}^{5}$$

$$mask = 0\;1\;1\;1\;0\;1\;1\;1$$

$$(Y+(X \oplus mask)) \wedge \neg mask = 1\;0\;0\;0\;0\;0\;0\;0$$

Source: [Jignesh Patel]

Bitweaving/H – Example

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

	t_0				t_4			
$X =$	0	0	0	1	0	1	1	0
	5				5			
$Y =$	0	1	0	1	0	1	0	1
$mask =$	0	1	1	1	0	1	1	1
$(Y+(X \oplus mask)) \wedge \neg mask =$	1	0	0	0	0	0	0	0

Selection Vector

Source: Jignesh Patel

Bitweaving/H – Example

SELECT * FROM table WHERE val < 5

$$X = \overbrace{0\ 0\ 0\ 1}^{t_0}\ \overbrace{0\ 1\ 1\ 0}^{t_4}$$

$$Y = \overbrace{0\ 1\ 0\ 1}^{5}\ \overbrace{0\ 1\ 0\ 1}^{5}$$

$$mask = 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1$$

$$(Y+(X\oplus mask))\wedge\neg mask = \underbrace{\boxed{1}}_{1<5}\ 0\ 0\ 0\ \underbrace{\boxed{0}}_{5<6}\ 0\ 0\ 0$$

Source: [Jignesh Patel]

Bitweaving/H – Example

SELECT * FROM table WHERE val < 5

	t_0				t_4			
$X =$	0	0	0	1	0	1	1	0
	5				5			
$Y =$	0	1	0	1	0	1	0	1
$mask =$	0	1	1	1	0	1	1	1
$(Y+(X \oplus mask)) \wedge \neg mask =$	1	0	0	0	0	0	0	0
	$1 < 5$				$5 < 6$			

Only requires four instructions to evaluate a single word.

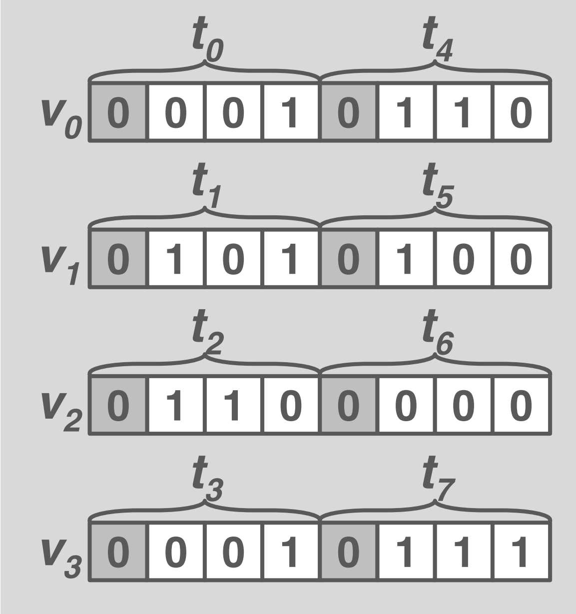
Works on any word size and encoding length.

Paper contains algorithms for other operators.

Source: Jignesh Patel

Bitweaving/H – Example

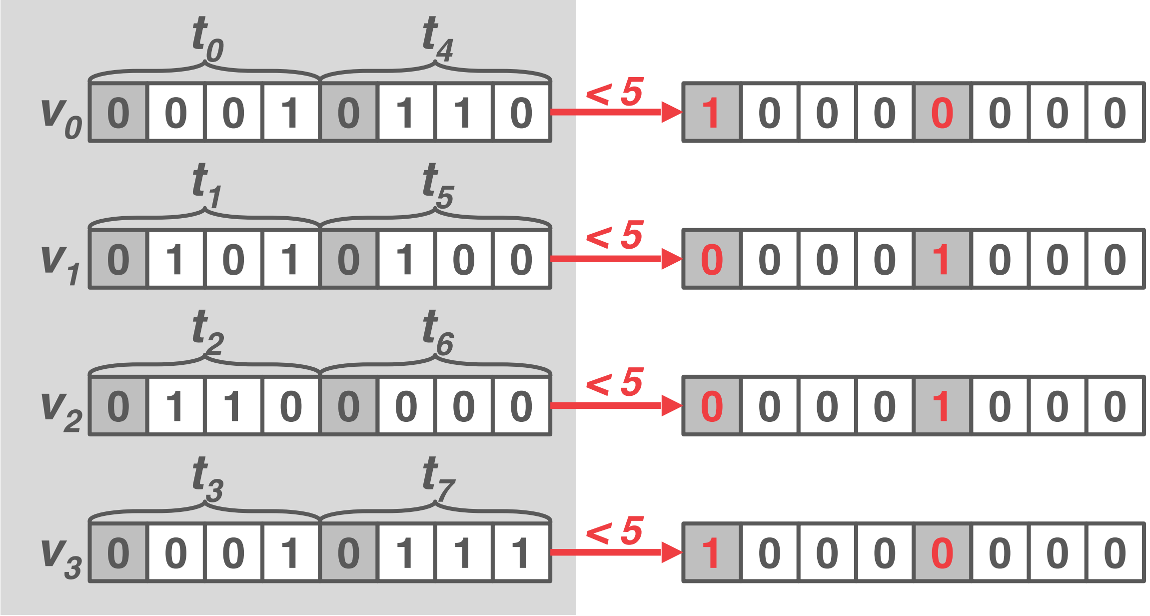
SELECT * FROM table
WHERE val < 5



Source: [Jignesh Patel]()

Bitweaving/H – Example

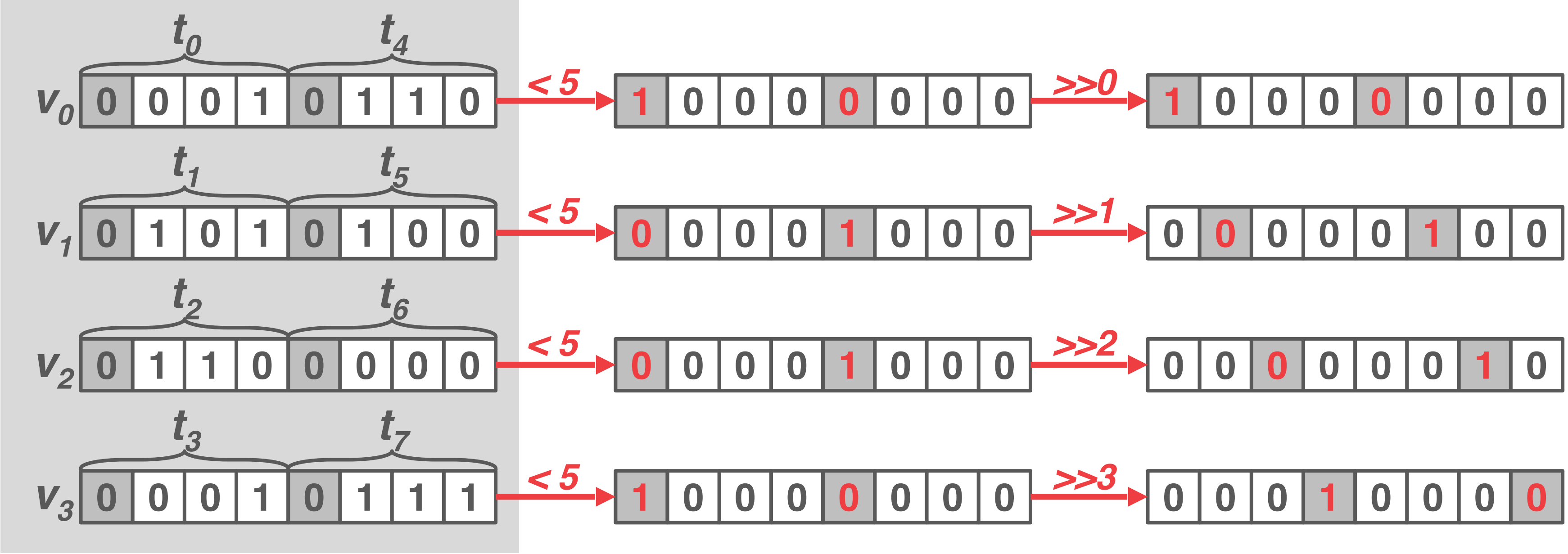
SELECT * FROM table
WHERE val < 5



Source: [Jignesh Patel]

Bitweaving/H – Example

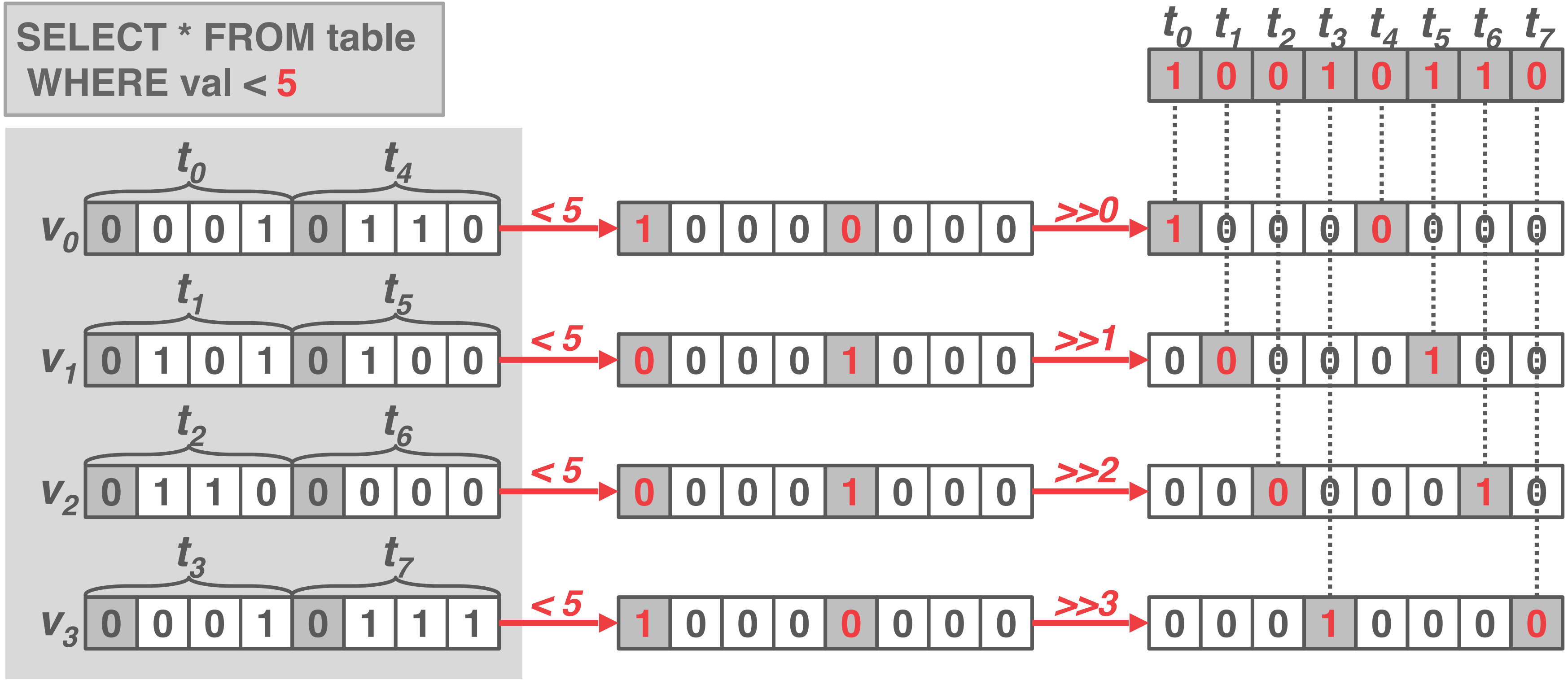
SELECT * FROM table
WHERE val < 5



Source: Jignesh Patel

Bitweaving/H – Example

SELECT * FROM table
WHERE val < 5



Source: Jignesh Patel

Selection Vector

SIMD operators produce a bitmask specifying which tuples satisfy a predicate.
→ DBMS must convert it into column offsets.

Approach #1: Iteration

Approach #2: Pre-computed Positions Table

Selection Vector

SIMD operators produce a bitmask specifying 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

Selection Vector

SIMD operators produce a bitmask specifying which tuples satisfy a predicate.
→ DBMS must convert it into column offsets.

Approach #1: Iteration

Approach #2: Pre-computed Positions Table

Selection Vector

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
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 operators produce a bitmask specifying which tuples satisfy a predicate.
→ DBMS must convert it into column offsets.

Approach #1: Iteration

Approach #2: Pre-computed Positions Table

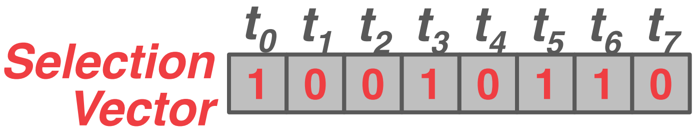
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Selection Vector	1	0	0	1	0	1	1	0

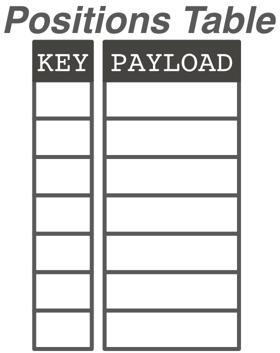
Selection Vector

SIMD operators produce a bitmask specifying which tuples satisfy a predicate.
→ DBMS must convert it into column offsets.

Approach #1: Iteration

Approach #2: Pre-computed Positions Table



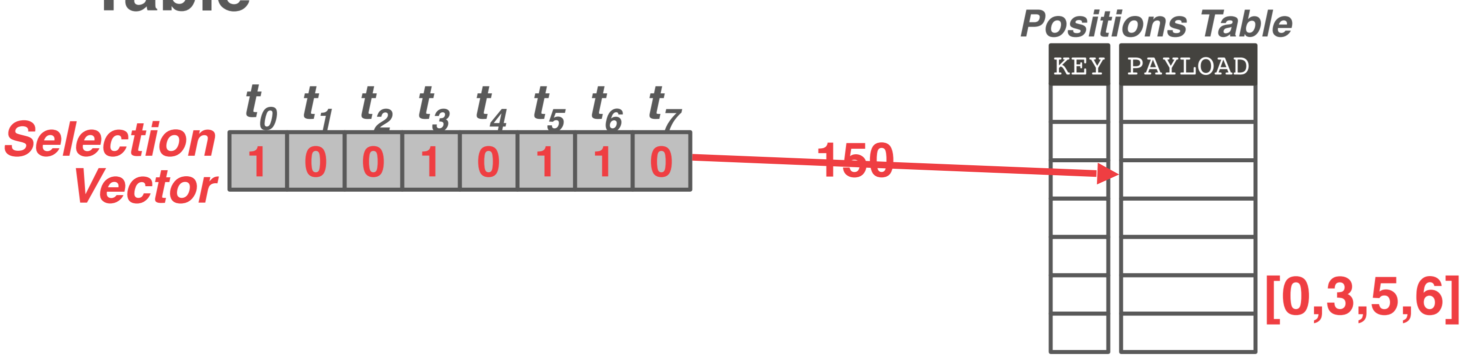


Selection Vector

SIMD operators produce a bitmask specifying 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

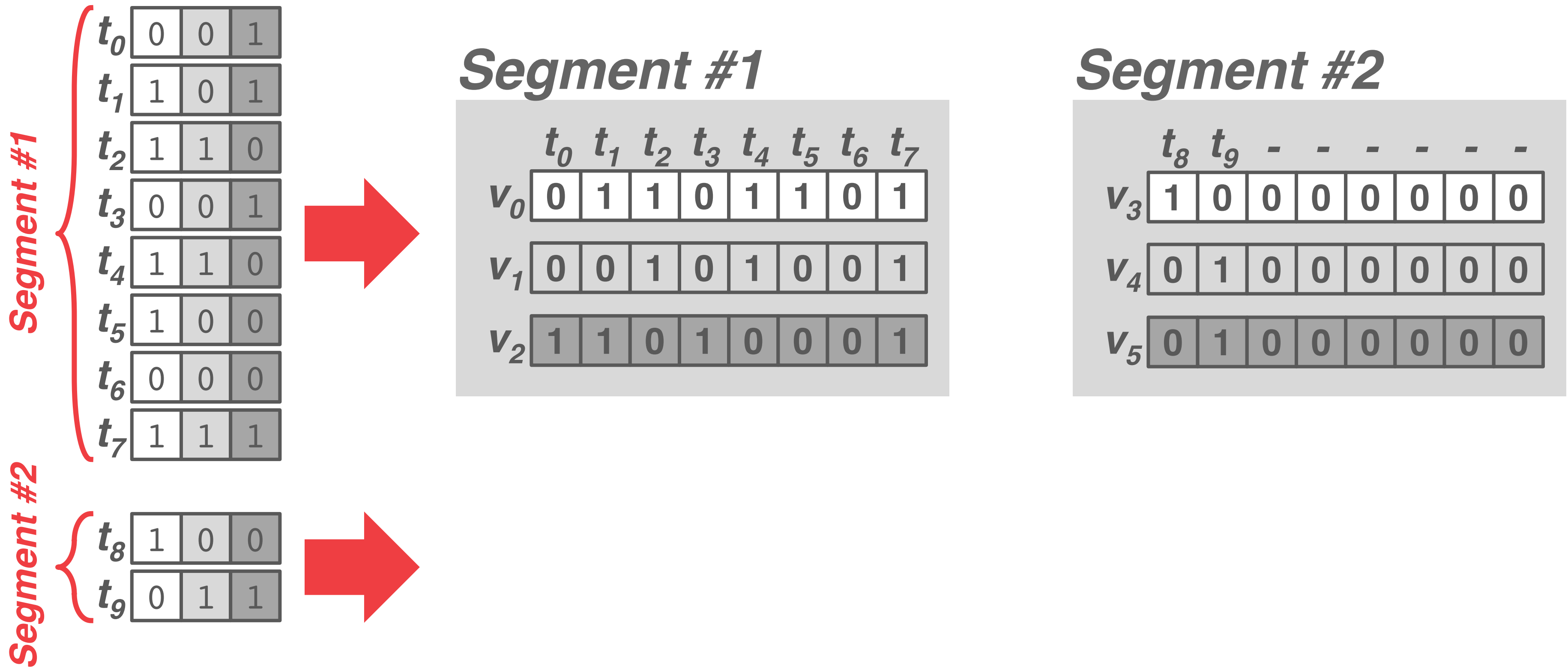
Segment #1

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

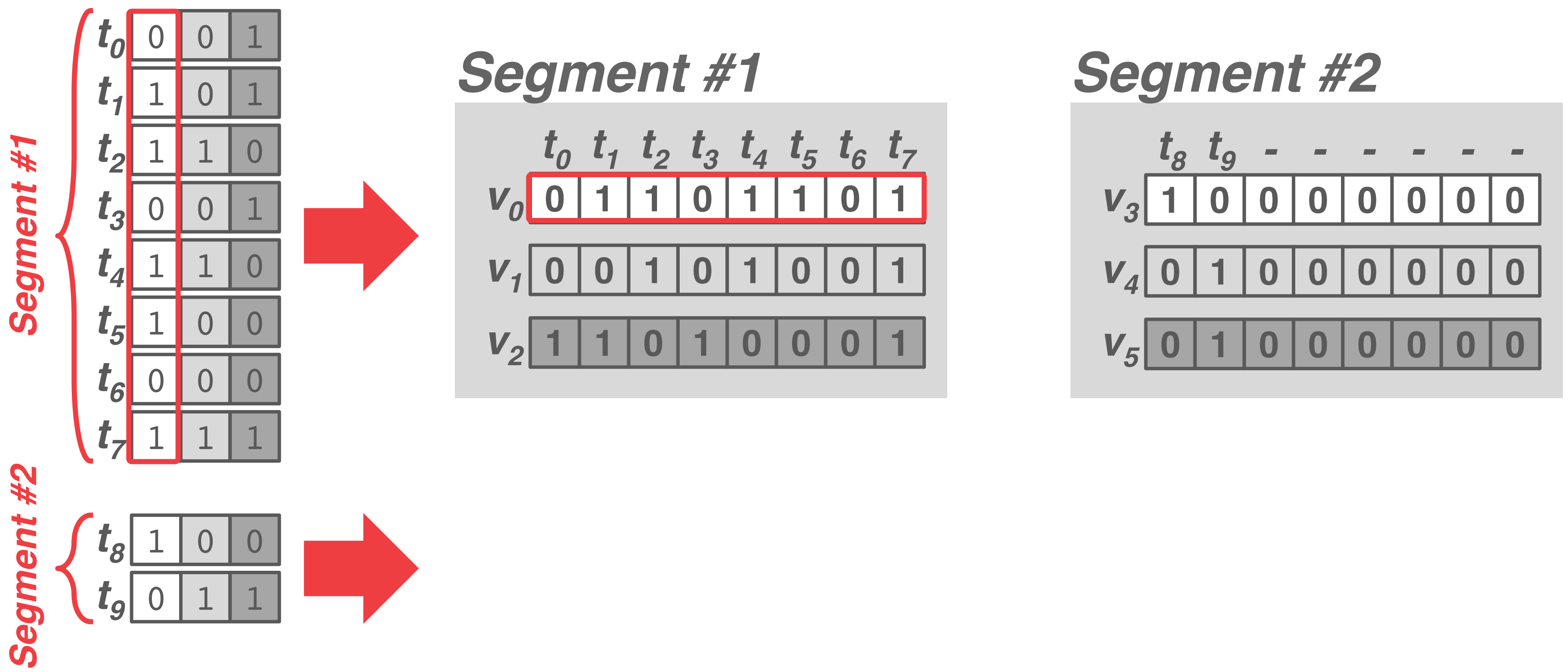
Segment #2

t_8	1	0	0
t_9	0	1	1

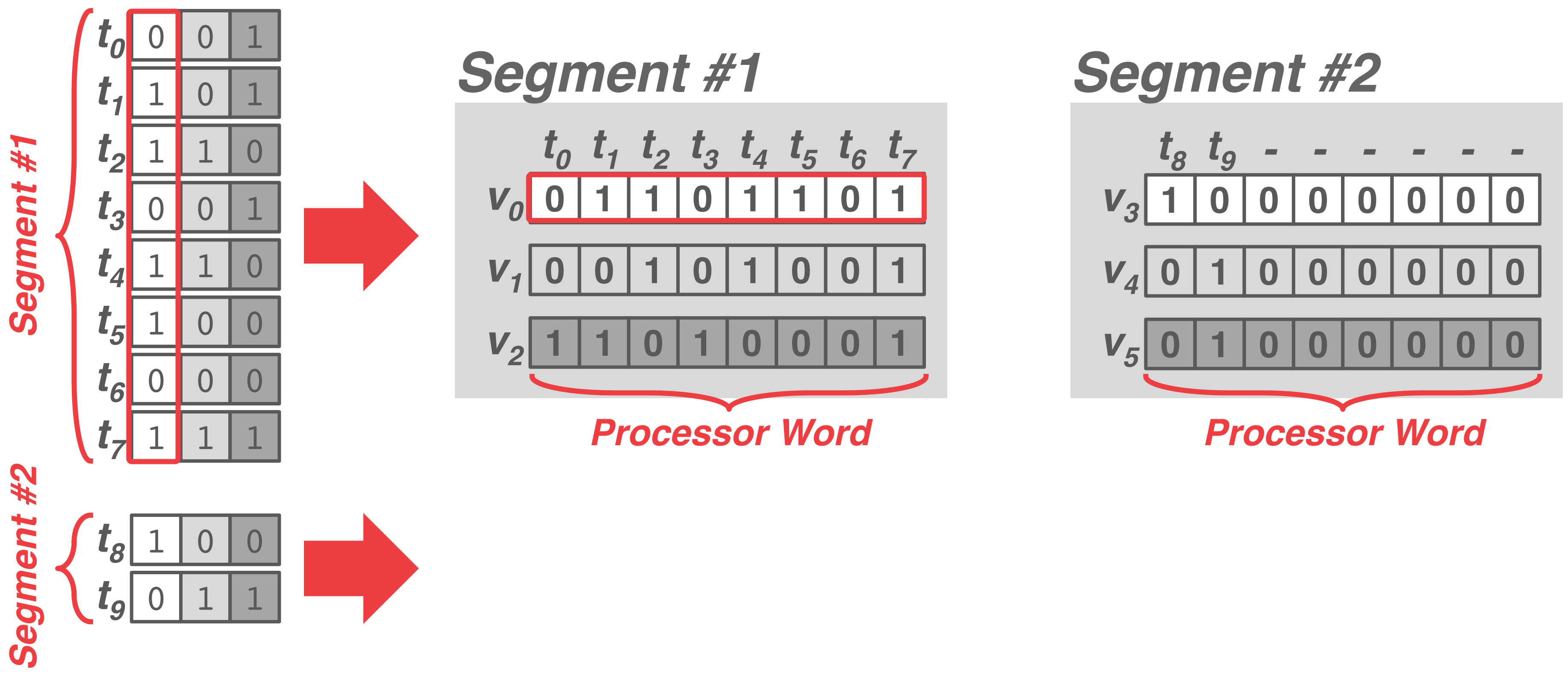
Vertical Storage



Vertical Storage



Vertical Storage



Bitweaving/V – Example

SELECT * FROM table
WHERE val = 2

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
```

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

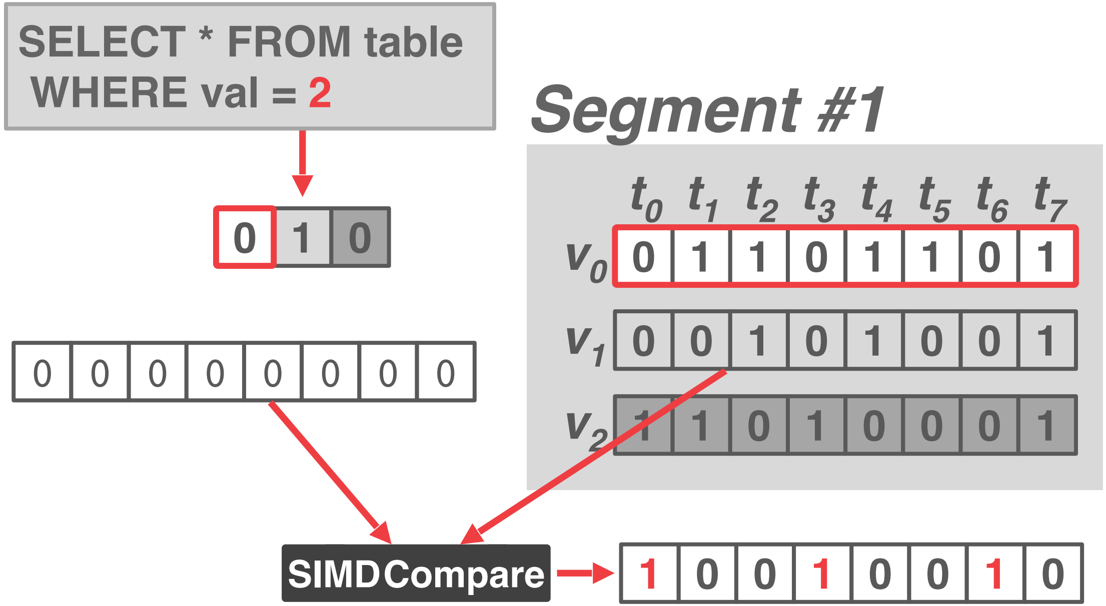
```
SELECT * FROM table
 WHERE val = 2
```

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



Bitweaving/V – Example

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

0	1	0

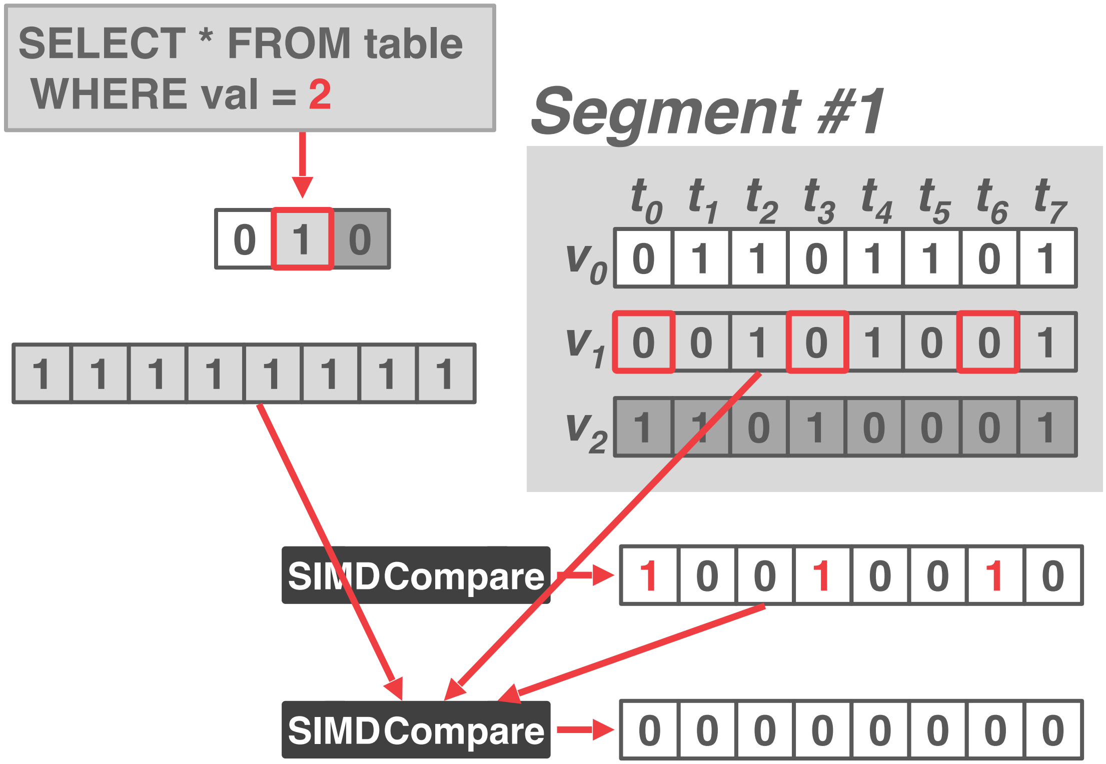
Segment #1

	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

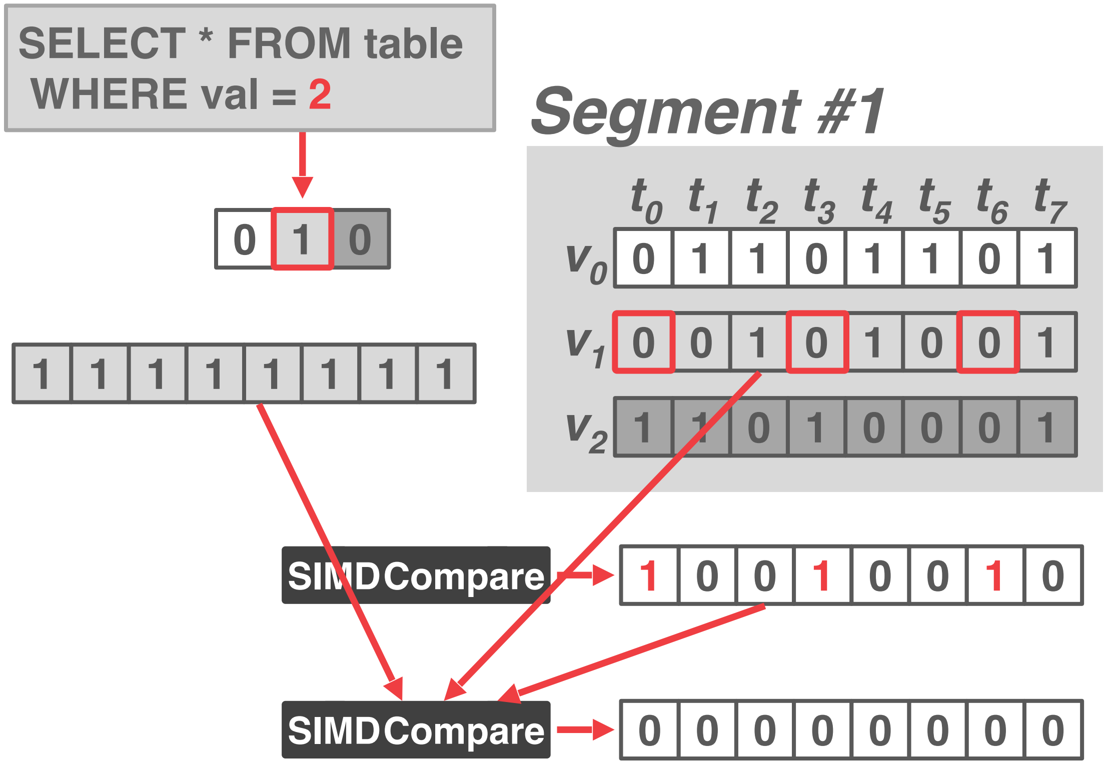
SIMDCompare →

1	0	0	1	0	0	1	0

Bitweaving/V – Example



Bitweaving/V – Example



Can perform early pruning just like in BitMap indexes.

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

Today's Agenda

~~Zone Maps~~
~~Bitmap Indexes~~
~~Bit-Slicing~~
~~Bit-Weaving~~
Column Imprints
Column Sketches

Observation

All the previous Bitmap schemes were about storing exact/lossless representations of columnar data.

The DBMS could trade off accuracy for faster evaluation in the common case.
→ It still must always check the original data to avoid false positives.

Column Imprints

Store a bitmap that indicates whether there is a bit set at a bit-slice of cache-line values.

Column Imprints

Store a bitmap that indicates whether there is a bit set at a bit-slice of cache-line values.

Original Data

value
1
8
4

COLUMN IMPRINTS: A SECONDARY INDEX STRUCTURE
SIGMOD 2013

Column Imprints

Store a bitmap that indicates whether there is a bit set at a bit-slice of cache-line values.

Original Data

value
1
8
4

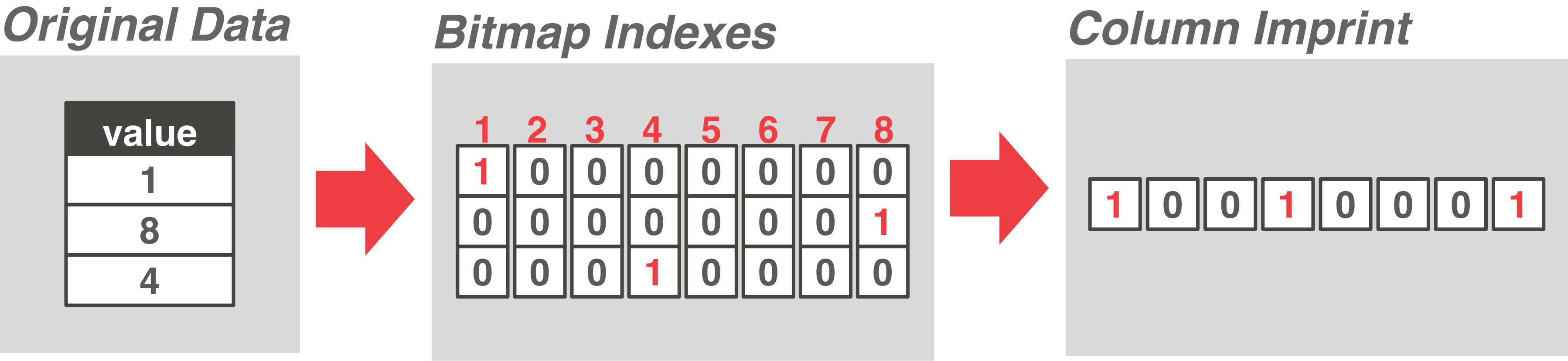
Bitmap Indexes

1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0

COLUMN IMPRINTS: A SECONDARY INDEX STRUCTURE
SIGMOD 2013

Column Imprints

Store a bitmap that indicates whether there is a bit set at a bit-slice of cache-line values.



COLUMN IMPRINTS: A SECONDARY INDEX STRUCTURE
SIGMOD 2013

Column Sketches

A variation of range-encoded Bitmaps that uses smaller “sketch” codes to indicate that a tuple's value exists in a range.

DBMS must automatically figure out the best mapping of codes.
→ Trade-off between distribution of values and compactness.
→ Assign unique codes to frequent values to avoid false positives.

COLUMN SKETCHES: A SCAN ACCELERATOR FOR RAPID AND ROBUST PREDICATE EVALUATION
SIGMOD 2013

Column Sketches

Original Data

val
13
191
56
92
81
120
231
172

8-bits

Source: Brian Hentschel

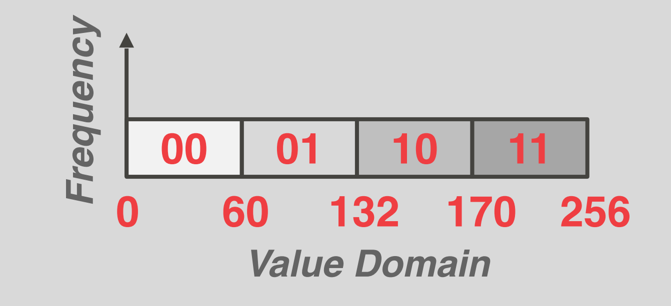
Column Sketches

Original Data

val
13
191
56
92
81
120
231
172

8-bits

Histogram



Source: [Brian Hentschel]()

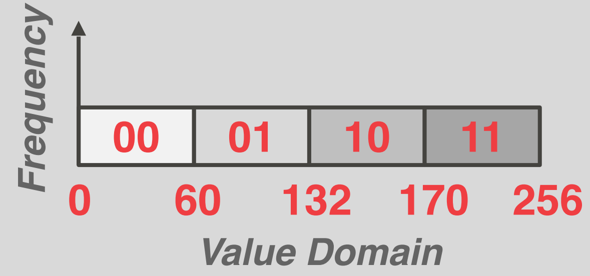
Column Sketches

Original Data

val
13
191
56
92
81
120
231
172

8-bits

Histogram

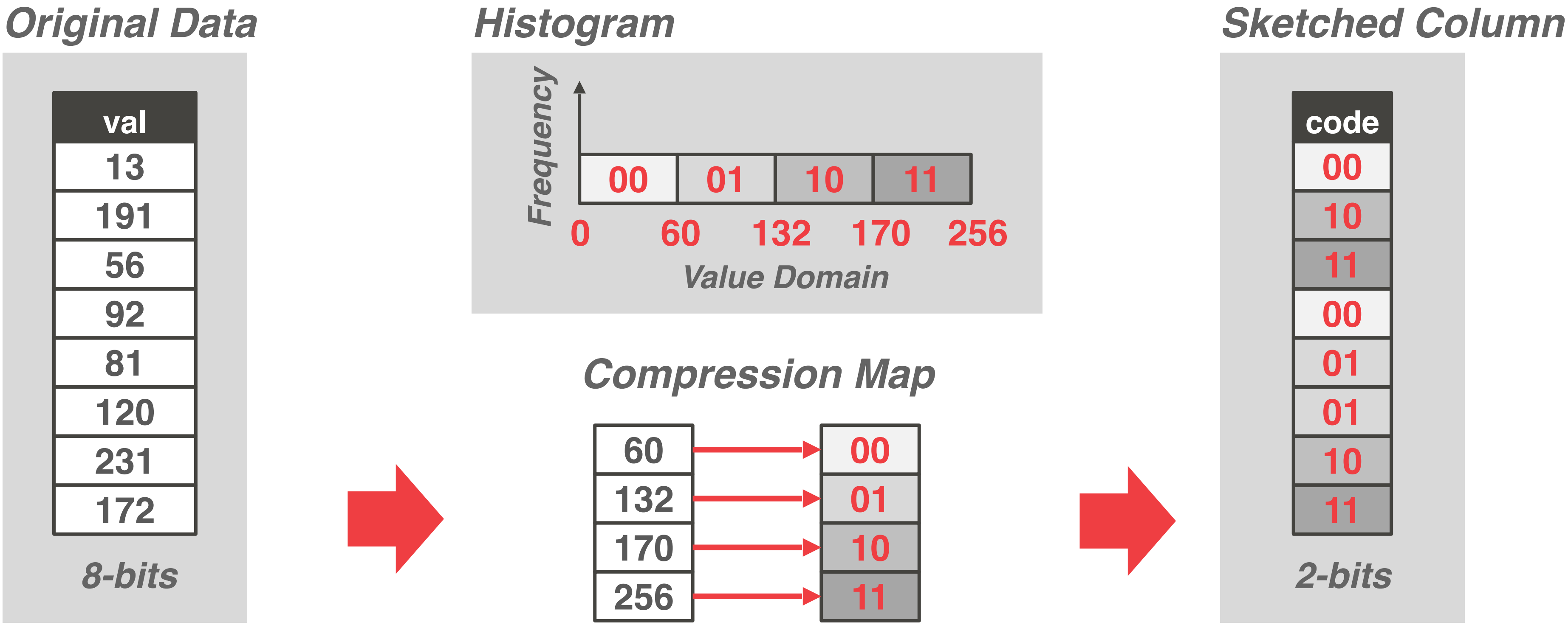


Compression Map

60	00
132	01
170	10
256	11

Source: [Brian Hentschel]

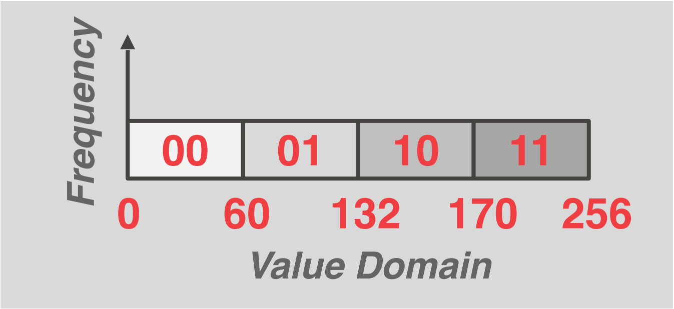
Column Sketches



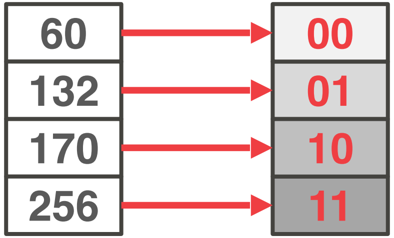
Source: [Brian Hentschel]

Column Sketches

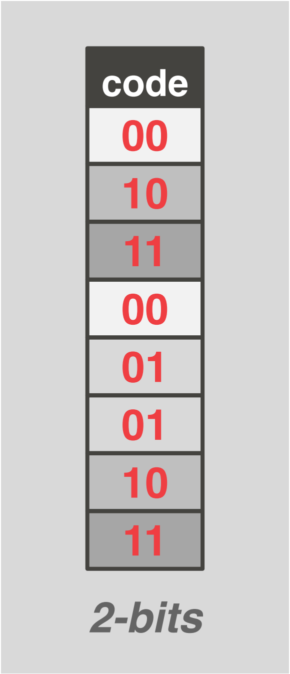
Histogram



Compression Map



Sketched Column

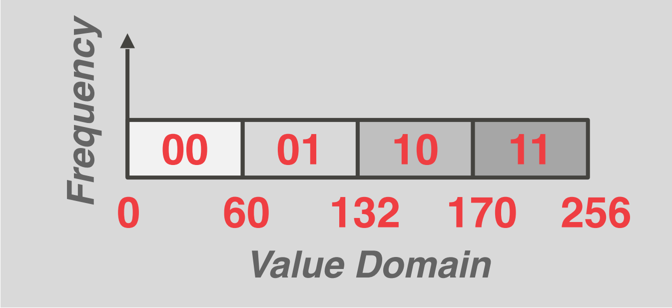


SELECT * FROM table WHERE val < 90

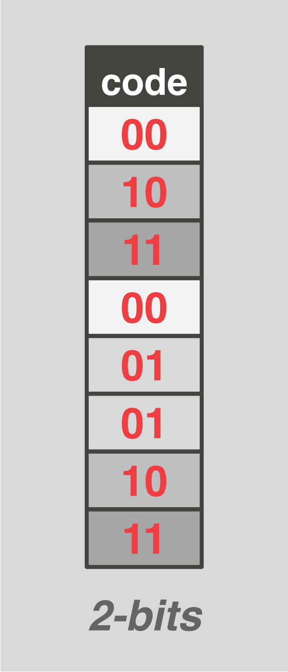
Source: Brian Hentschel

Column Sketches

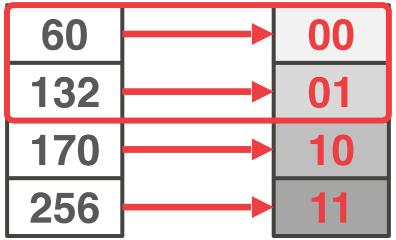
Histogram



Sketched Column



Compression Map

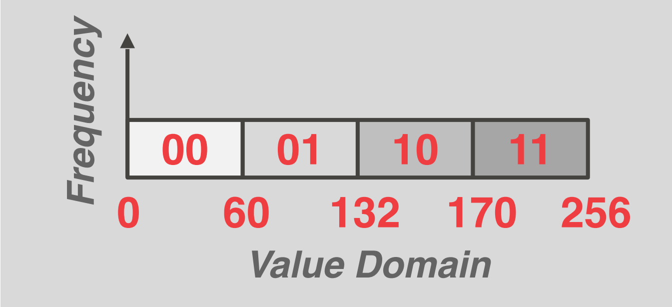


SELECT * FROM table
WHERE val < 90

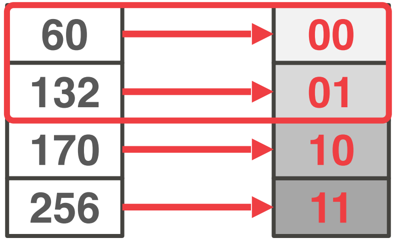
Source: [Brian Hentschel]

Column Sketches

Histogram



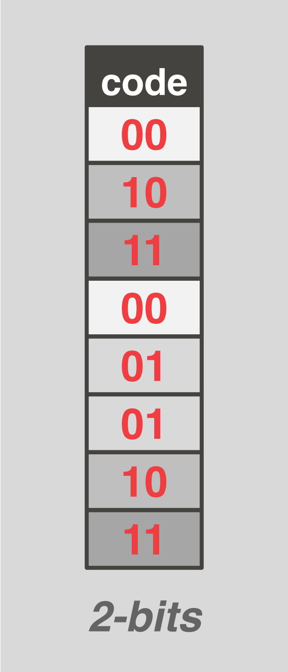
Compression Map



SELECT * FROM table
WHERE val < 90

map(90) = 01

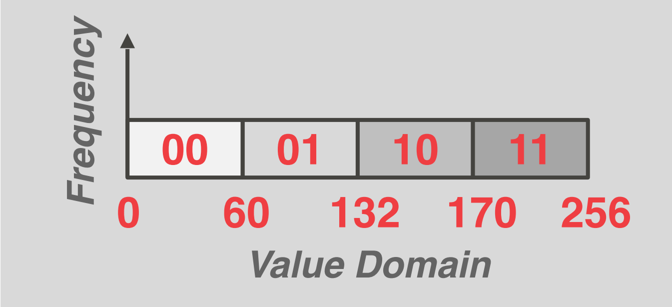
Sketched Column



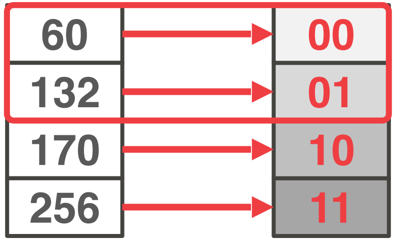
Source: [Brian Hentschel]

Column Sketches

Histogram

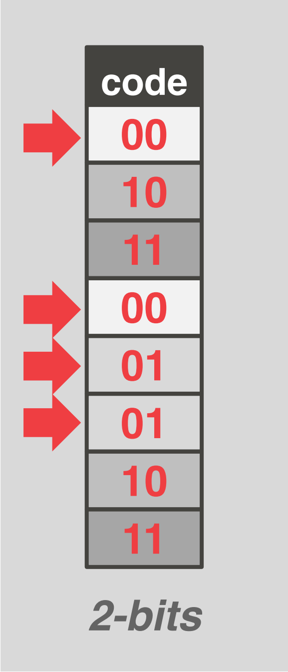


Compression Map



map(90) = 01

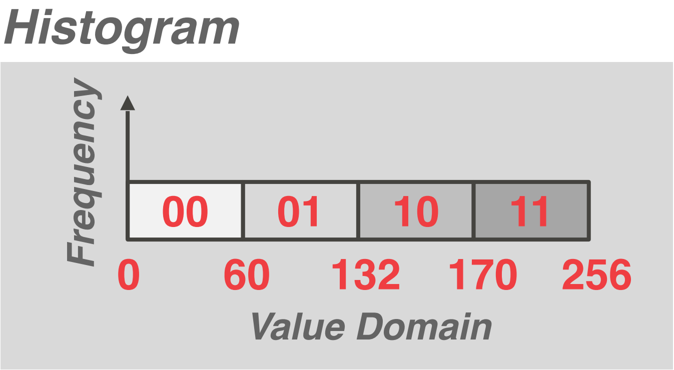
Sketched Column

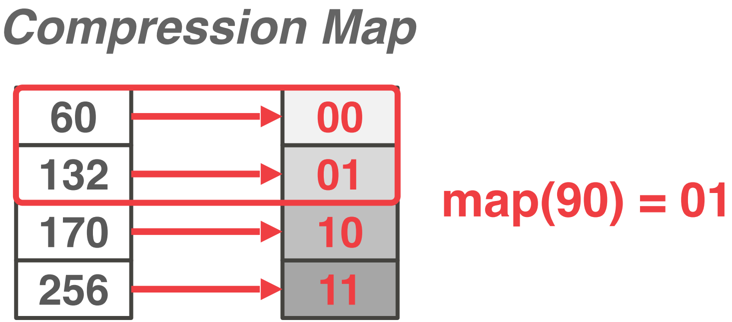


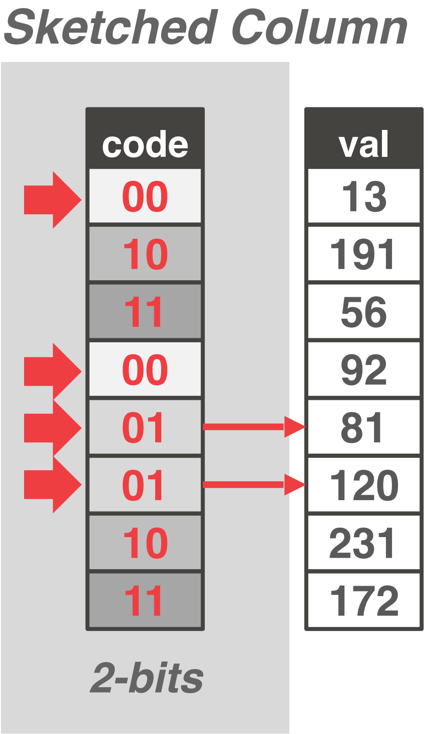
SELECT * FROM table
WHERE val < 90

Source: Brian Hentschel

Column Sketches







SELECT * FROM table
WHERE val < 90

Source: [Brian Hentschel]

Parting Thoughts

Zone Maps are the most widely used method to accelerate sequential scans.

Bitmap indexes are more common in NSM DBMSs than columnar OLAP systems.

We’re ignoring multi-dimensional and inverted indexes…

Next Class

Data Compression (Tuples, Indexes)