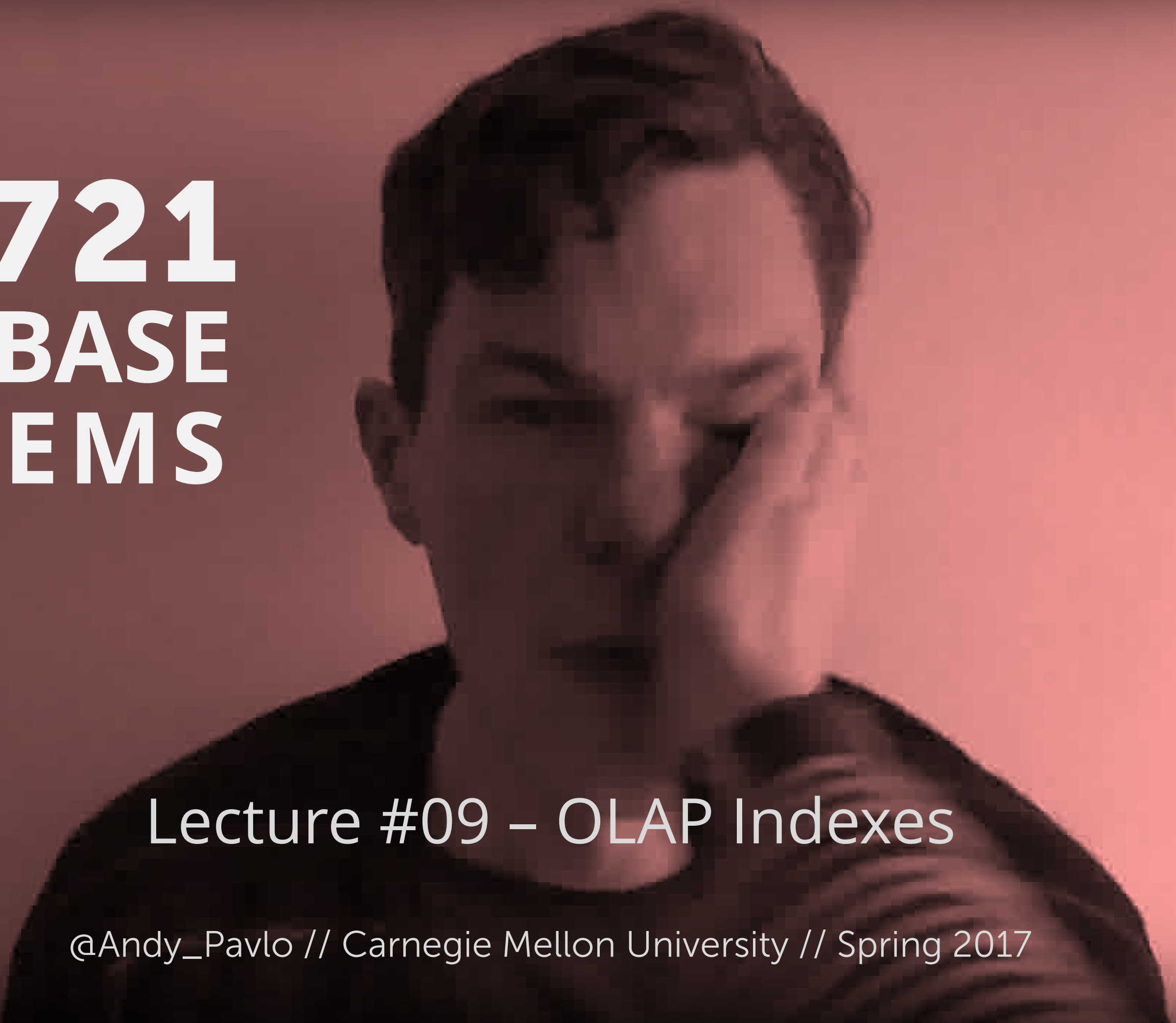
# 15-721 DATABASE SYSTEMS



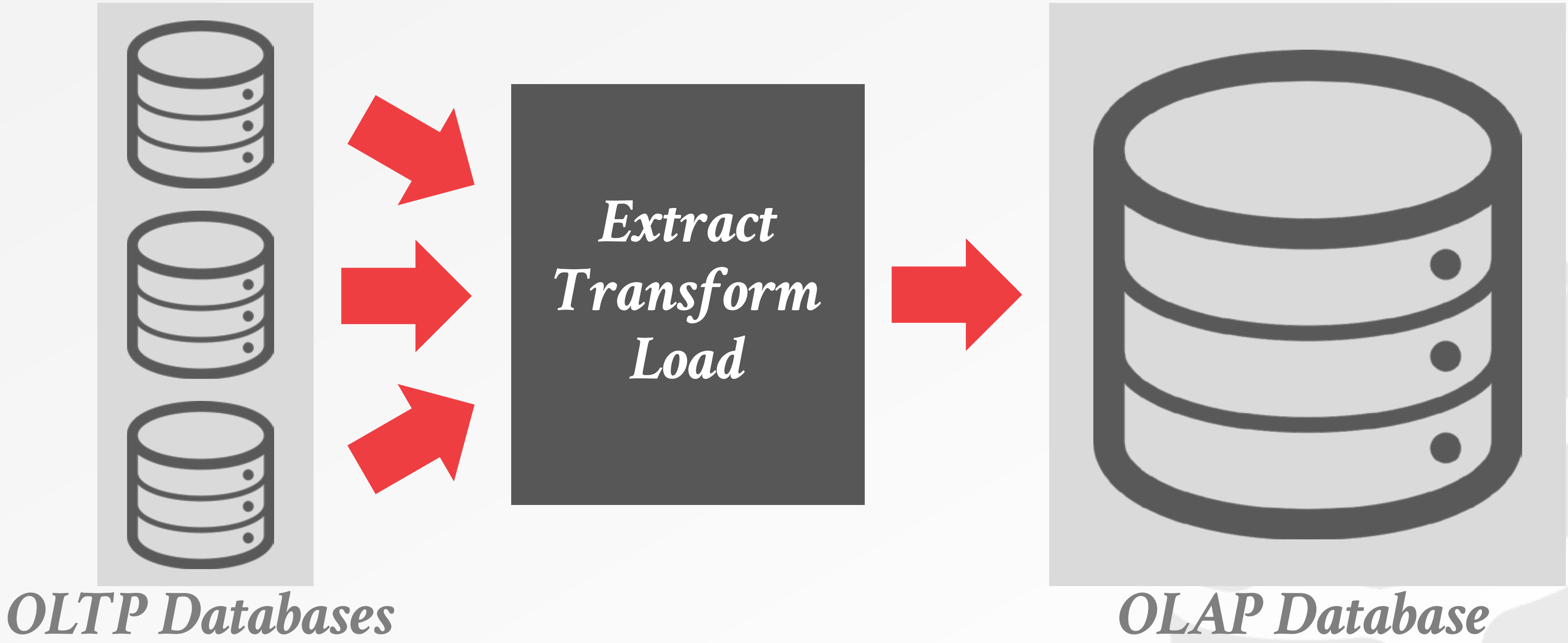
## Lecture #09 – OLAP Indexes

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

# TODAY'S AGENDA

OLAP Schemas
Projection/Columnar Indexes (MSSQL)
Bitmap Indexes

# BIFURCATED ENVIRONMENT

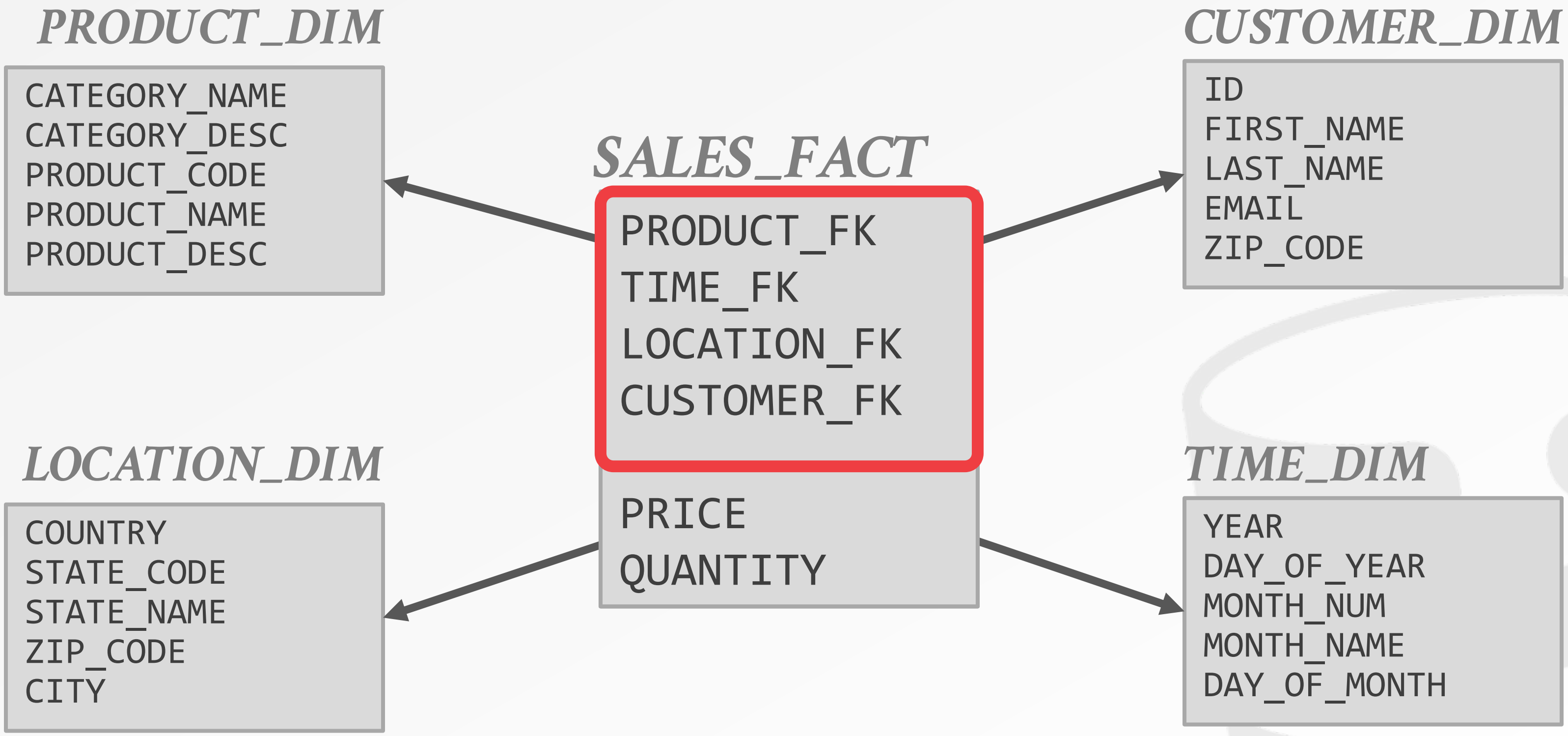


# DECISION SUPPORT SYSTEMS

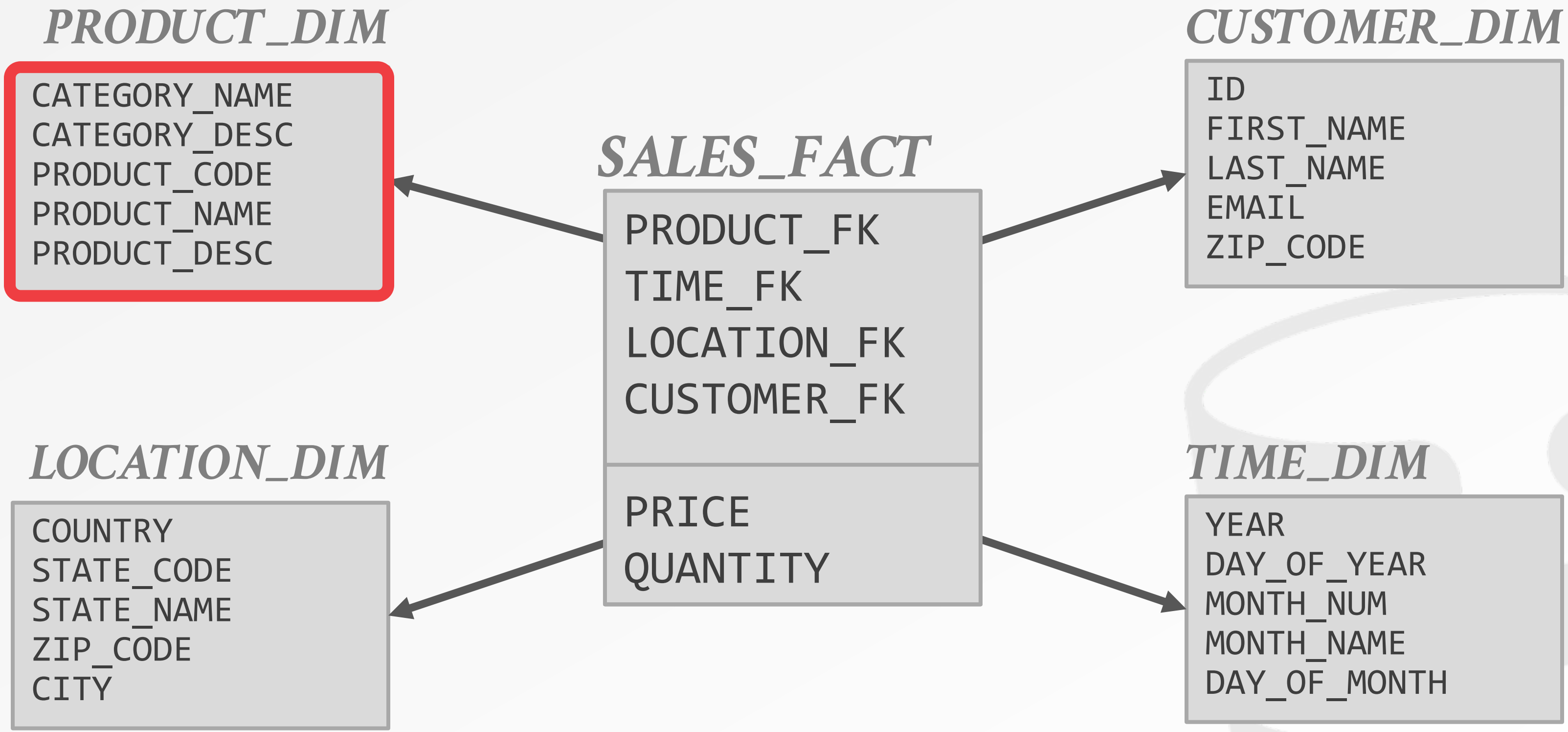
Applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analyzing historical data.

Star Schema vs. Snowflake Schema

# STAR SCHEMA

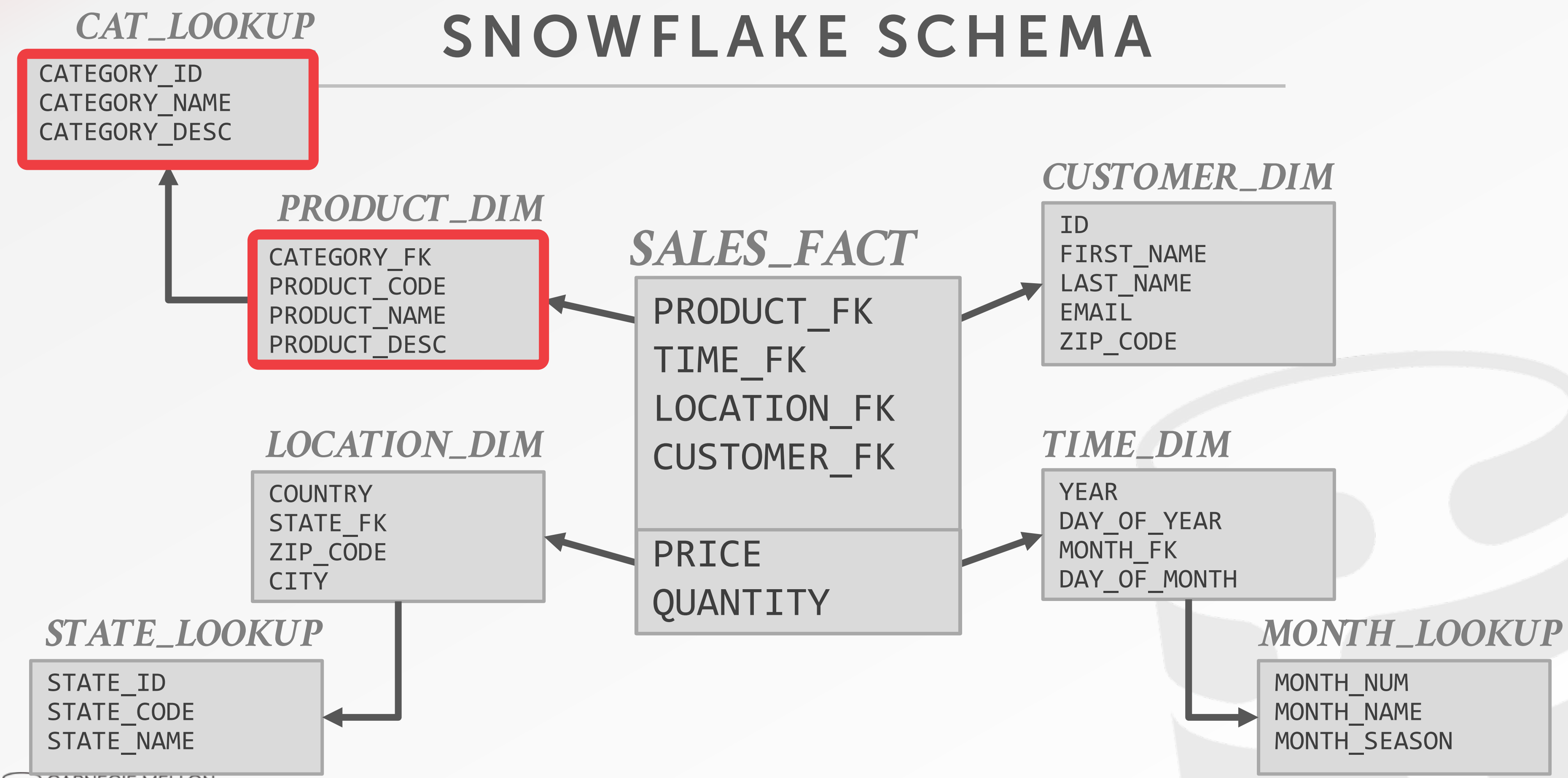

PRODUCT_DIM
CATEGORY_NAME
CATEGORY_DESC
PRODUCT_CODE
PRODUCT_NAME
PRODUCT_DESC
CUSTOMER_DIM
ID
FIRST_NAME
LAST_NAME
EMAIL
ZIP_CODE
SALES_FACT
PRODUCT_FK
TIME_FK
LOCATION_FK
CUSTOMER_FK
PRICE
QUANTITY
LOCATION_DIM
COUNTRY
STATE_CODE
STATE_NAME
ZIP_CODE
CITY
TIME_DIM
YEAR
DAY_OF_YEAR
MONTH_NUM
MONTH_NAME
DAY_OF_MONTH

# STAR SCHEMA



PRODUCT_DIM
CATEGORY_NAME
CATEGORY_DESC
PRODUCT_CODE
PRODUCT_NAME
PRODUCT_DESC
CUSTOMER_DIM
ID
FIRST_NAME
LAST_NAME
EMAIL
ZIP_CODE
SALES_FACT
PRODUCT_FK
TIME_FK
LOCATION_FK
CUSTOMER_FK
PRICE
QUANTITY
LOCATION_DIM
COUNTRY
STATE_CODE
STATE_NAME
ZIP_CODE
CITY
TIME_DIM
YEAR
DAY_OF_YEAR
MONTH_NUM
MONTH_NAME
DAY_OF_MONTH

# SNOWFLAKE SCHEMA



# STAR VS. SNOWFLAKE SCHEMA

**Issue #1: Normalization**

→ Snowflake schemas take up less storage space.
→ Denormalized data models may incur integrity and consistency violations.

**Issue #2: Query Complexity**

→ Snowflake schemas require more joins to get the data needed for a query.
→ Queries on star schemas will (usually) be faster.

# OBSERVATION

Using a B+tree index on a fact table results in a lot of wasted storage if the values are repetitive and the cardinality is low.

```
CREATE TABLE sales_fact (
  id INT PRIMARY KEY,
  ⋮
  location_fk INT
   REFERENCES location_dim (id)
);
```

```
CREATE TABLE location_dim (
  id INT PRIMARY KEY,
  ⋮
  zip_code INT
);
```

```
SELECT COUNT(*)
  FROM sales_fact AS S
  JOIN location_dim AS L
    ON S.location_fk = L.id
 WHERE L.zip_code = 15217
```

# MSSQL: COLUMNAR INDEXES

Decompose rows into compressed column segments for single attributes.

→ Original data still remains in row store.
→ No way to map an entry in the column index back to its corresponding entry in row store.

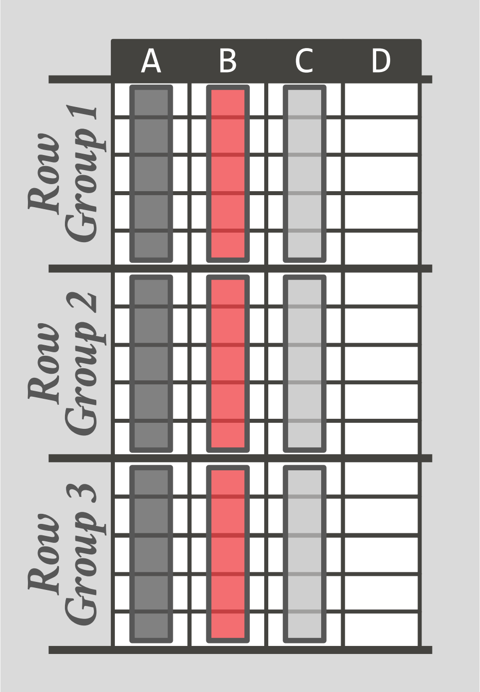
Use as many existing components in MSSQL.

Original implementation in would force a table to become read-only.

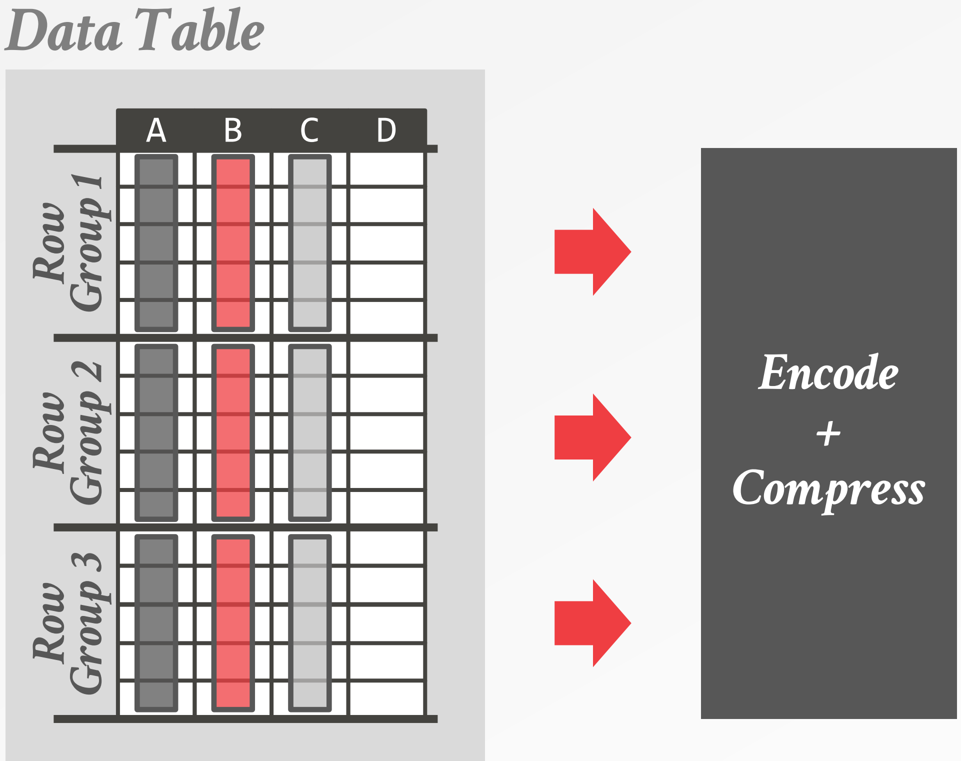
SQL SERVER COLUMN STORE INDEXES
*SIGMOD 2010*

# MSSQL: COLUMNAR INDEXES

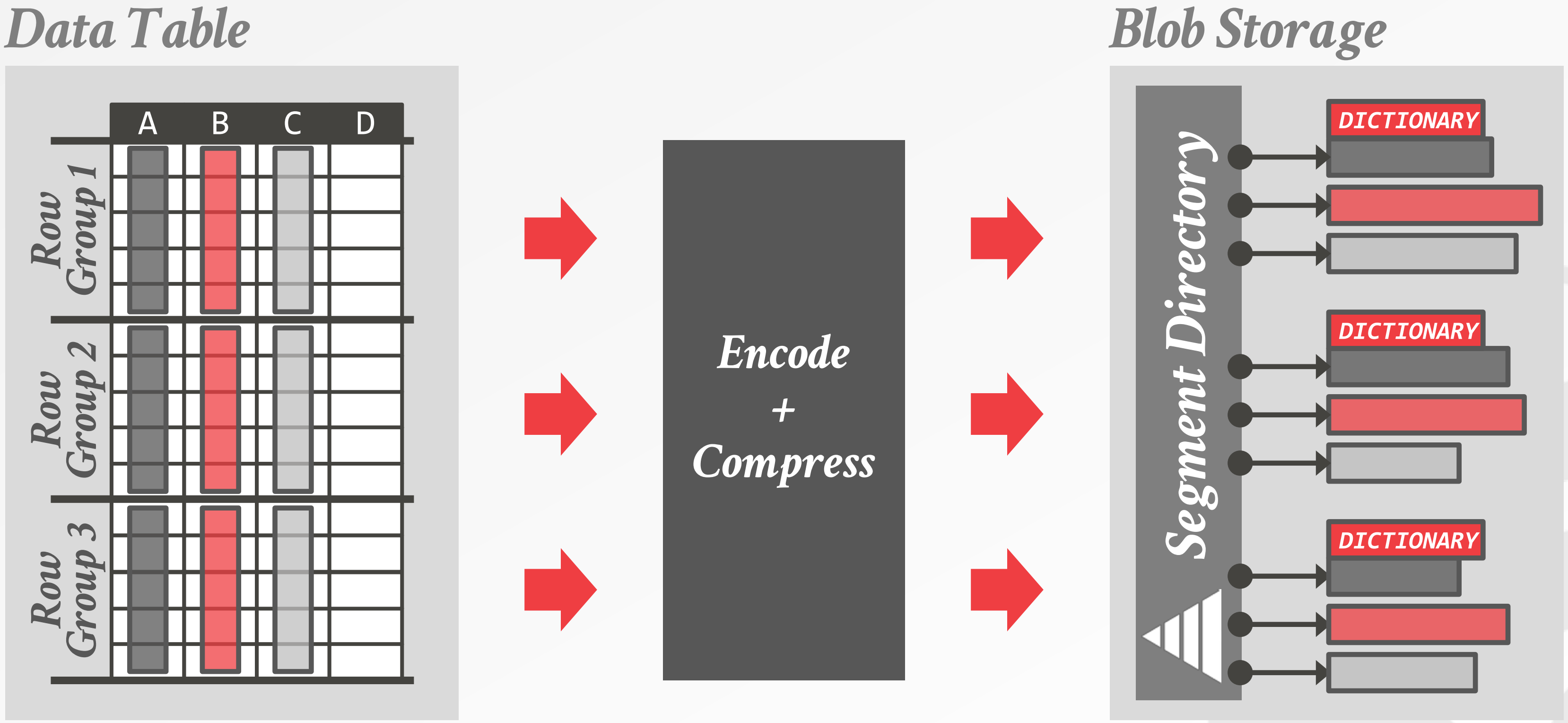
*Data Table*



# MSSQL: COLUMNAR INDEXES



# MSSQL: COLUMNAR INDEXES



# MSSQL: INTERNAL CATALOG

**Segment Directory:** Keeps track of statistics for each column segments per row group.

→ Size
→ # of Rows
→ Min and Max key values
→ Encoding Meta-data

**Data Dictionary:** Maps dictionary ids to their original values.

# MSSQL: DICTIONARY ENCODING

Construct a separate table that maps unique values for an attribute to a dictionary id.

→ Can be sorted by frequency or lexicographical ordering.

For each tuple, store the 32-bit id of its value in the dictionary instead of the real value.

# MSSQL: DICTIONARY ENCODING

*Original Data*

| id | city |
|---|---|
| 1 | New York |
| 2 | Chicago |
| 3 | New York |
| 4 | New York |
| 6 | Pittsburgh |
| 7 | Chicago |
| 8 | New York |
| 9 | New York |

# MSSQL: DICTIONARY ENCODING

*Original Data*

| id | city |
|---|---|
| 1 | New York |
| 2 | Chicago |
| 3 | New York |
| 4 | New York |
| 6 | Pittsburgh |
| 7 | Chicago |
| 8 | New York |
| 9 | New York |

*Compressed Data*

| id | city |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 0 |
| 4 | 0 |
| 6 | 2 |
| 7 | 1 |
| 8 | 0 |
| 9 | 0 |

| ***DICTIONARY*** |
|---|
| 0→(New York,5) |
| 1→(Chicago,2) |
| 2→(Pittsburgh,1) |

# MSSQL: VALUE ENCODING

Transform the domain of a numeric column segment into a set of distinct values in a smaller domain of integers.

Allows the DBMS to use smaller data types to store larger values.

Also sometimes called **delta encoding**.

# MSSQL: RUN-LENGTH ENCODING

Compress runs of the same value in a single column into triplets:

→ The value of the attribute.
→ The start position in the column segment.
→ The # of elements in the run.

Requires the columns to be sorted intelligently to maximize compression opportunities.

# MSSQL: RUN-LENGTH ENCODING

*Original Data*

| id | sex |
|---|---|
| 1 | M |
| 2 | M |
| 3 | M |
| 4 | F |
| 6 | M |
| 7 | F |
| 8 | M |
| 9 | M |

# MSSQL: RUN-LENGTH ENCODING

*Original Data*

| id | sex |
|---|---|
| 1 | M |
| 2 | M |
| 3 | M |
| 4 | F |
| 6 | M |
| 7 | F |
| 8 | M |
| 9 | M |

*Compressed Data*

| id | sex |
|---|---|
| 1 | (M,0,3) |
| 2 | (F,3,1) |
| 3 | (M,4,1) |
| 4 | (F,5,1) |
| 6 | (M,6,2) |
| 7 | |
| 8 | |
| 9 | |

*RLE Triplet*
- *Value*
- *Offset*
- *Length*

# MSSQL: RUN-LENGTH ENCODING

*Sorted Data*

| id | sex |
|---|---|
| 1 | M |
| 2 | M |
| 3 | M |
| 6 | M |
| 8 | M |
| 9 | M |
| 4 | F |
| 7 | F |

*Compressed Data*

| id | sex |
|---|---|
| 1 | (M,0,6) |
| 2 | (F,7,2) |
| 3 | |
| 6 | |
| 7 | |
| 9 | |
| 4 | |
| 7 | |

*RLE Triplet*
- *Value*
- *Offset*
- *Length*

# MSSQL: QUERY PROCESSING

Modify the query planner and optimizer to be aware of the columnar indexes.

Add new vector-at-a-time operators that can operate directly on columnar indexes.

Compute joins using Bitmaps built on-the-fly.

# MSSQL: UPDATES SINCE 2012

Clustered column indexes.

More data types.

Support for **INSERT**, **UPDATE**, and **DELETE**:

→ Use a **delta store** for modifications and updates. The DBMS seamlessly combines results from both the columnar indexes and the delta store.

→ Deleted tuples are marked in a bitmap.

ENHANCEMENTS TO SQL SERVER COLUMN STORES
*SIGMOD 2013*

# BITMAP INDEXES

Store a separate Bitmap for each unique value for a particular 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.

MODEL 204 ARCHITECTURE AND PERFORMANCE
*High Performance Transaction Systems 1987*

# BITMAP INDEXES

## *Original Data*

| id | sex |
|---|---|
| 1 | M |
| 2 | M |
| 3 | M |
| 4 | F |
| 6 | M |
| 7 | F |
| 8 | M |
| 9 | M |

# BITMAP INDEXES

*Original Data*

| id | sex |
|---|---|
| 1 | M |
| 2 | M |
| 3 | M |
| 4 | F |
| 6 | M |
| 7 | F |
| 8 | M |
| 9 | M |

*Compressed Data*

| id | sex: M | sex: F |
|---|---|---|
| 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
);
```

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

# BITMAP INDEXES: EXAMPLE

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

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

Every time a txn inserts a new tuple, we have to extend 43,000 different bitmaps.

# BITMAP INDEX: DESIGN CHOICES

Encoding Scheme

Compression

# 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**

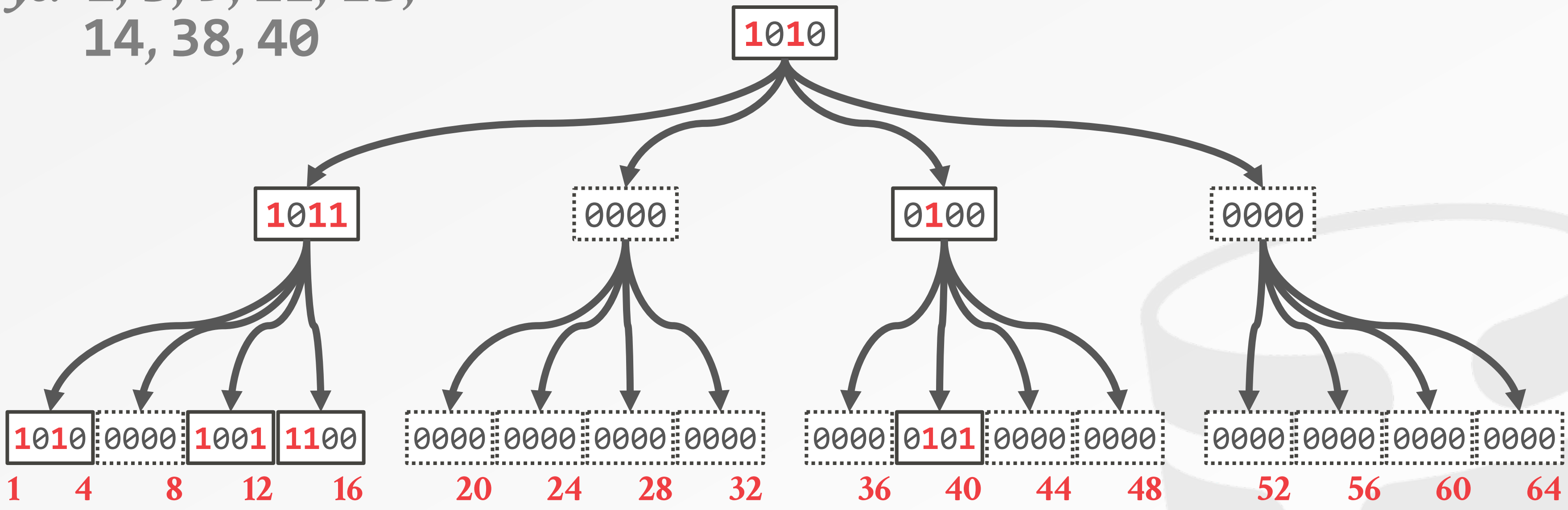
→ Use a tree to identify empty key ranges.

**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**

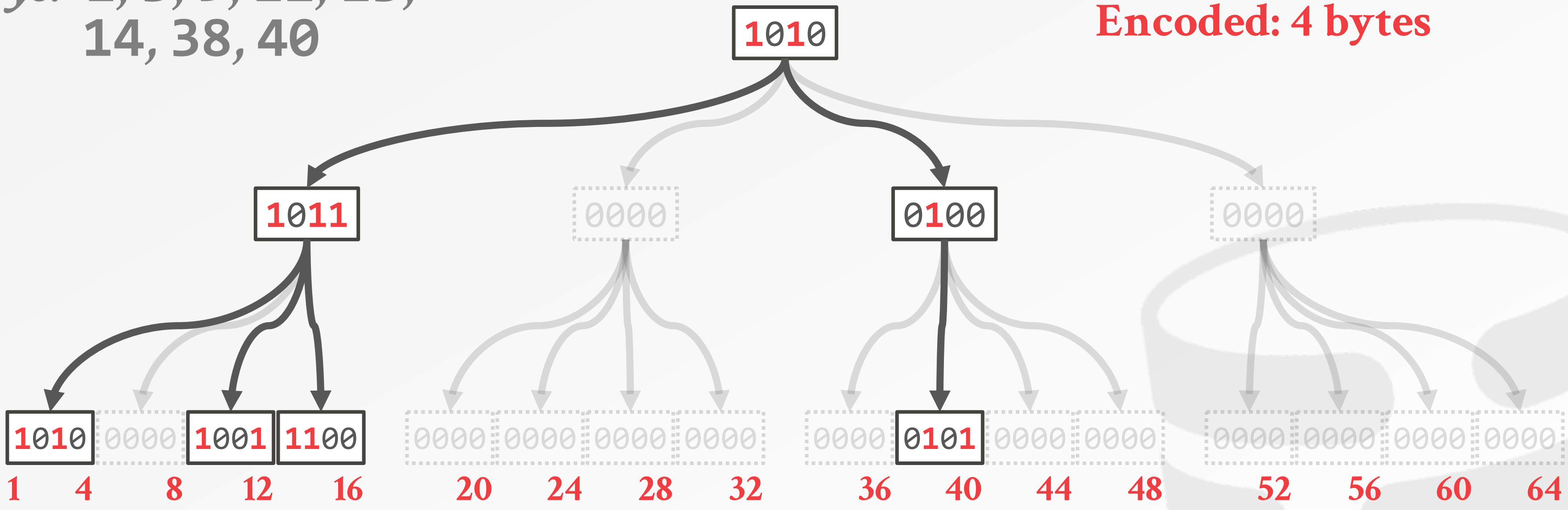



HIERARCHICAL BITMAP INDEX: AN EFFICIENT AND SCALABLE INDEXING TECHNIQUE FOR SET-VALUED ATTRIBUTES
*Advances in Databases and Information Systems 2003*

# HIERARCHICAL ENCODING

*Keys:* **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*

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*

| N? | 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*

| N? | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | | | | | | | | | | |

bin(21042)→ 00101001000110010

# BIT-SLICED ENCODING

*Original Data*

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

*Bit-Slices*

| N? | 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*

| N? | 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*

| N? | 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.

# BIT-SLICED ENCODING

## *Original Data*

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

## *Bit-Slices*

| N? | 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)`

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

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

→ Repeat for the rest of slices…

# BITMAP INDEX: COMPRESSION

**Approach #1: General Purpose Compression**

→ Use standard compression algorithms (e.g., LZ4, Snappy).

→ Have to decompress before you can use it to process a query. Not useful for in-memory DBMSs.

**Approach #2: Byte-aligned Bitmap Codes**

→ Structured run-length encoding compression.

**Approach #3: Roaring Bitmaps**

→ Modern hybrid of run-length encoding and value lists.`

# BYTE-ALIGNED BITMAP CODES

Divide Bitmap into chunks that contain different categories of bytes:

→ **Gap Byte**: All the bits are **0**s.
→ **Tail Byte:** Some bits are **1**s.

Encode each **chunk** that consists of some **Gap Bytes** followed by some **Tail Bytes**.

→ Gap Bytes are compressed with RLE.
→ Tail Bytes are stored uncompressed unless it consists of only 1 byte or has only 1 non-zero bit.

BYTE-ALIGNED BITMAP COMPRESSION
*Data Compression Conference 1995*

# BYTE-ALIGNED BITMAP CODES

## *Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

## *Compressed Bitmap*

# BYTE-ALIGNED BITMAP CODES

*Bitmap*

*Gap Bytes* *Tail Bytes*

```
00000000 00000000 00010000   #1
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

*Compressed Bitmap*

Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

*Bitmap*

*Gap Bytes* *Tail Bytes*

```
00000000 00000000 00010000  #1
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000  #2
00000000 00000000 00000000
00000000 01000000 00100010
```

*Compressed Bitmap*

Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

*Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

*Compressed Bitmap*

**#1** (010)(1)(0100)
1-3 4 5-7

**Chunk #1 (Bytes 1-3)**

Header Byte:

→ Number of Gap Bytes (Bits 1-3)
→ Is the tail special? (Bit 4)
→ Number of verbatim bytes (if Bit 4=0)
→ Index of **1** bit in tail byte (if Bit 4=1)

No gap length bytes since gap length < 7

No verbatim bytes since tail is special

Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

## *Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

## *Compressed Bitmap*

**#1** `(010)(1)(0100)`

**#2** `(111)(0)(0010) 00001101`
`01000000 00100010`

**Chunk #2 (Bytes 4-18)**

Header Byte:
- → 13 gap bytes, two tail bytes
- → # of gaps is > 7, so have to use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

*Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

*Compressed Bitmap*

```
#1 (010)(1)(0100)

#2 (111)(0)(0010) 00001101
   01000000 00100010
```

**Chunk #2 (Bytes 4-18)**

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so have to use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.



Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

*Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

*Compressed Bitmap*

**#1** (010)(1)(0100)

*Gap Length*

**#2** (111)(0)(0010) 00001101
01000000 00100010

**Chunk #2 (Bytes 4-18)**

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so have to use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

# BYTE-ALIGNED BITMAP CODES

***Bitmap***

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

***Compressed Bitmap***

**#1** (010)(1)(0100)

**#2** (111)(0)(0010) 00001101
01000000 00100010

**Chunk #2 (Bytes 4-18)**

Header Byte:

→ 13 gap bytes, two tail bytes

→ # of gaps is > 7, so have to use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

### *Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

### *Compressed Bitmap*

**#1** (010)(1)(0100)

**#2** (111)(0)(0010) 00001101
01000000 00100010

## Chunk #2 (Bytes 4-18)

Header Byte:
- → 13 gap bytes, two tail bytes
- → # of gaps is > 7, so have to use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

Source: Brian Babcock

# BYTE-ALIGNED BITMAP CODES

*Bitmap*

```
00000000 00000000 00010000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 00000000
00000000 01000000 00100010
```

*Compressed Bitmap*

**#1** (010)(1)(0100)

**#2** (111)(0)(0010) 00001101
01000000 00100010

*Verbatim Tail Bytes*

**Chunk #2 (Bytes 4-18)**

Header Byte:
- → 13 gap bytes, two tail bytes
- → # of gaps is > 7, so have to use extra byte

One gap length byte gives gap length = 13

Two verbatim bytes for tail.

**Original: 18 bytes**

**BBC Compressed: 5 bytes.**

# OBSERVATION

Oracle's BBC is an obsolete format

→ Although it provides good compression, it is likely much slower than more recent alternatives due to excessive branching.

→ Word-Aligned Hybrid (WAH) is a patented variation on BBC that provides better performance.

None of these support random access.

→ If you want to check whether a given value is present, you have to start from the beginning and uncompress the whole thing.

# ROARING BITMAPS

Store 32-bit integers in a compact two-level indexing data structure.

→ Dense chunks are stored using bitmaps

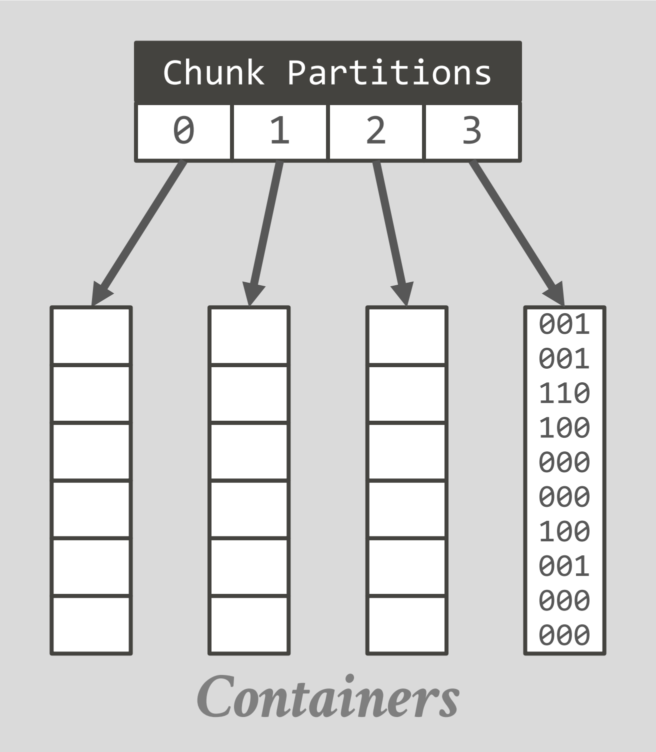
→ Sparse chunks use packed arrays of 16-bit integers.

Now used in Lucene, Hive, Spark.

BETTER BITMAP PERFORMANCE WITH ROARING BITMAPS

*Software: Practice and Experience 2015*

# ROARING BITMAPS

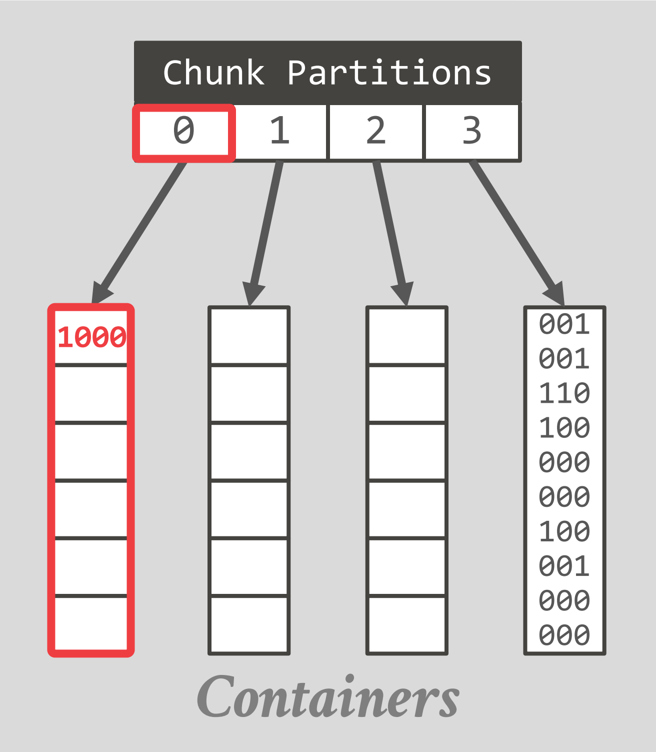



For each value **k**, assign it to a chunk based on **k/2$^{16}$**.

Only store **k%2$^{16}$** in container.

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

# ROARING BITMAPS



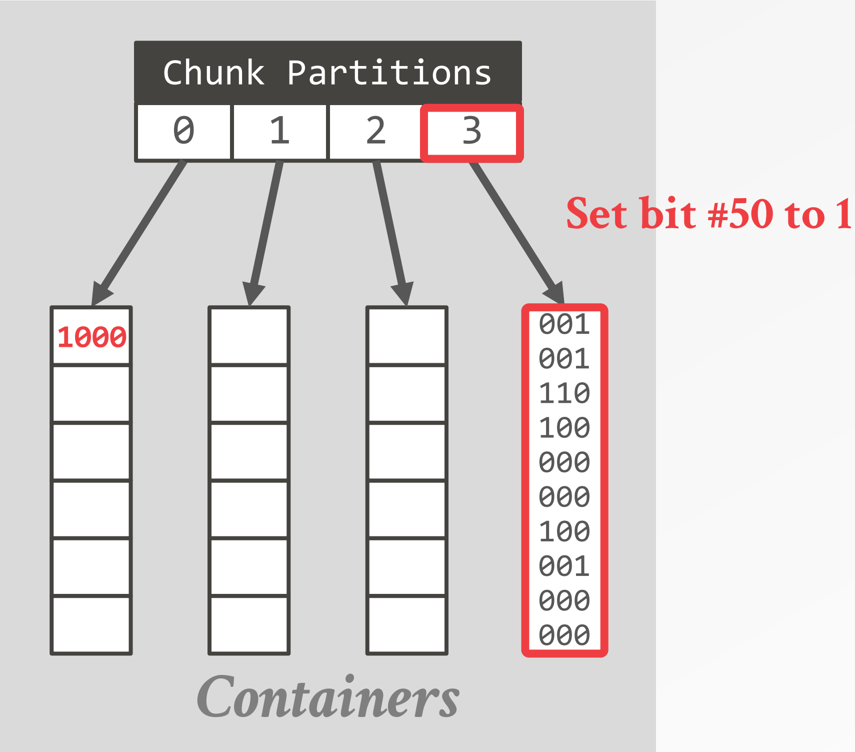

For each value **k**, assign it to a chunk based on **$k/2^{16}$**.

Only store **$k\%2^{16}$** in container.

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^**.

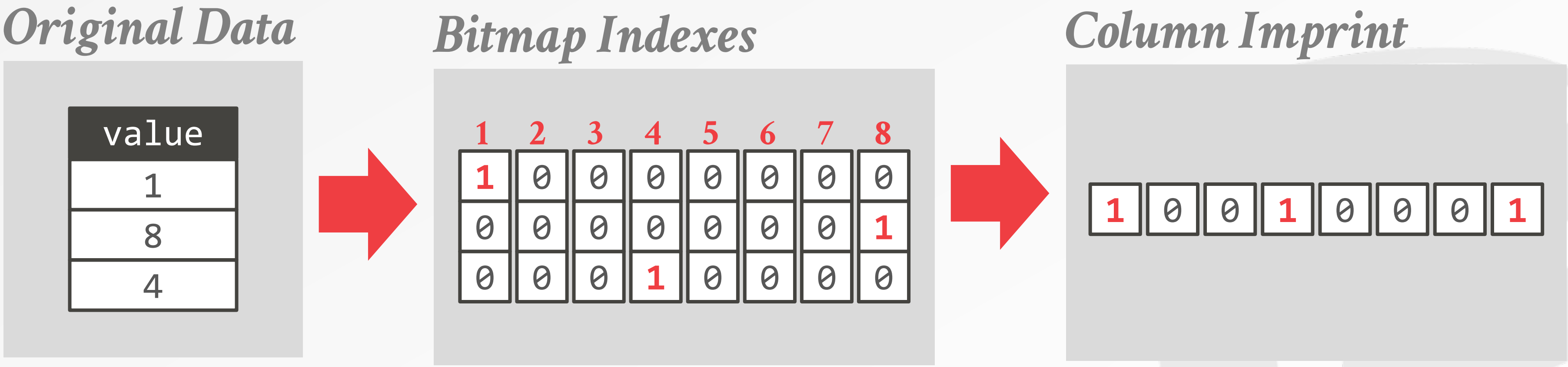
Only store **k%2^16^** in container.

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

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

# 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*

# PARTING THOUGHTS

These require that the position in the Bitmap corresponds to the tuple's position in the table.

→ This is not possible in a MVCC DBMS using the **Insert Method** unless there is a look-up table.

Maintaining a Bitmap Index is wasteful if there are a large number of unique values for a column and if those values are ephemeral.

We're ignoring multi-dimensional indexes…

# NEXT CLASS

Data Layout

Storage Models