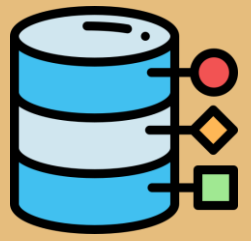


CSAI 302

Advanced Database Systems

Lec 03

Storage Management [2]

Tuple-oriented storage

- **Insert a new tuple:**
  - Check page directory to find a page with a free slot.
    - Retrieve the page from disk (if not in memory).
  - Check slot array to find empty space in page that will fit.
- **Update an existing tuple using its record id:**
  - Check page directory to find location of page.
    - Retrieve the page from disk (if not in memory).
  - Find offset in page using slot array.
    - If new data fits, overwrite existing data.
    - Otherwise, mark existing tuple as deleted and insert new version in a different page.

Problems

- Fragmentation
  - Pages are not fully utilized (unusable space, empty slots).
- Useless Disk I/O
  - DBMS must fetch entire page to update one tuple.
- Random Disk I/O
  - Worse case scenario when updating multiple tuples is that each tuple is on a separate page.
- What if the DBMS cannot overwrite data in pages and could only create new pages?
  - Examples: Some object stores, **HDFS**, **Google Colossus**

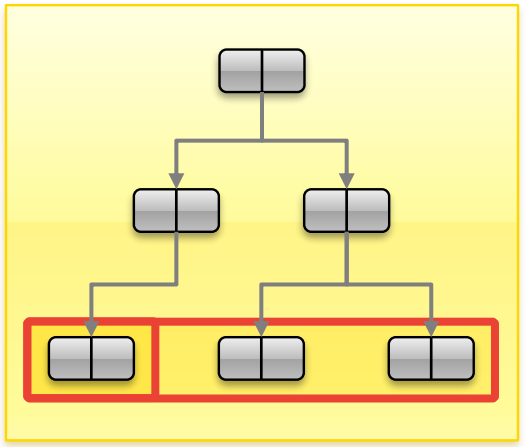
Log-structured Storage

Log-structured Storage

- Instead of storing tuples in pages and updating them in-place, the DBMS maintains a log that records changes to tuples.
  - ★ Each log entry represents a tuple **PUT/DELETE** operation.
  - ★ Originally proposed as **log-structure merge** trees (LSM Trees) in 1996.
- The DBMS applies changes to an in-memory data structure (**MemTable**) and then writes out the changes sequentially to disk (**SSTable**).

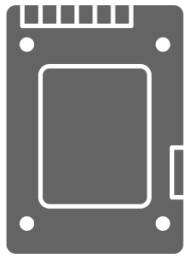
Log-structured storage

PUT (key102,b2)



MemTable

Memory



Disk

Key (Low → High)

PUT (key101,a_2)

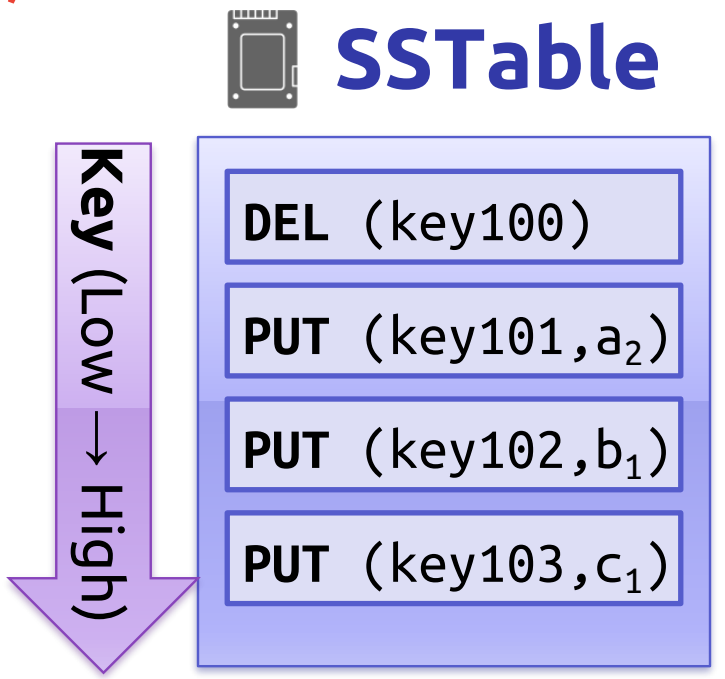
PUT (key102,b_1)

PUT (key103,c_1)

SSTable

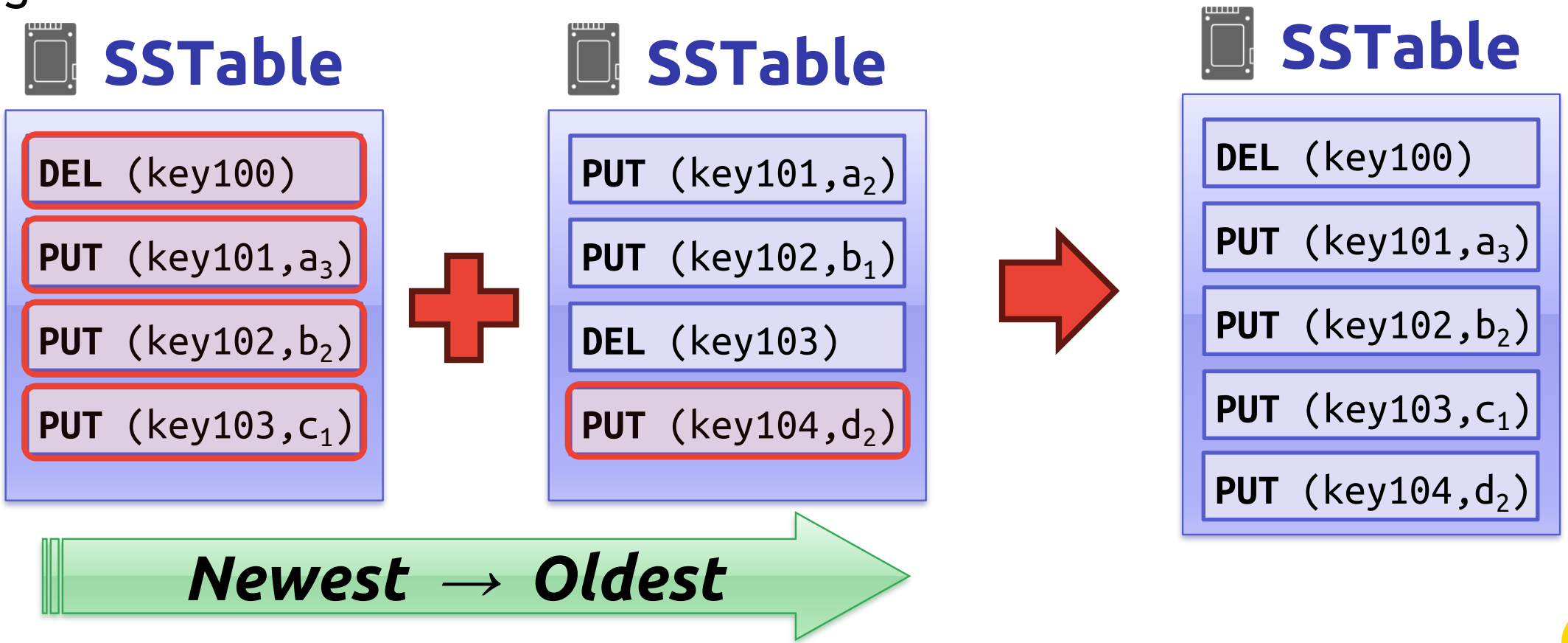
Log-structured storage

- Key-value storage that appends log records on disk to represent changes to tuples (PUT, DELETE).
  - Each log record must contain the tuple's unique identifier.
  - Put records contain the tuple contents.
  - Deletes marks the tuple as deleted.
- As the application makes changes to the database, the DBMS appends log records to the end of the file without checking previous log records.



Log-structured compaction

- Periodically compact SSTAbles to reduce wasted space and speed up reads.
  - ★Only keep the "latest" values for each key using a sort- merge algorithm.



Discussion

- Log-structured storage managers are more common today than in previous decades.
  - This is partly due to the proliferation of RocksDB.

- What are some downsides of this approach?
  - Write-Amplification.
  - Compaction is expensive.

Index-organized Storage

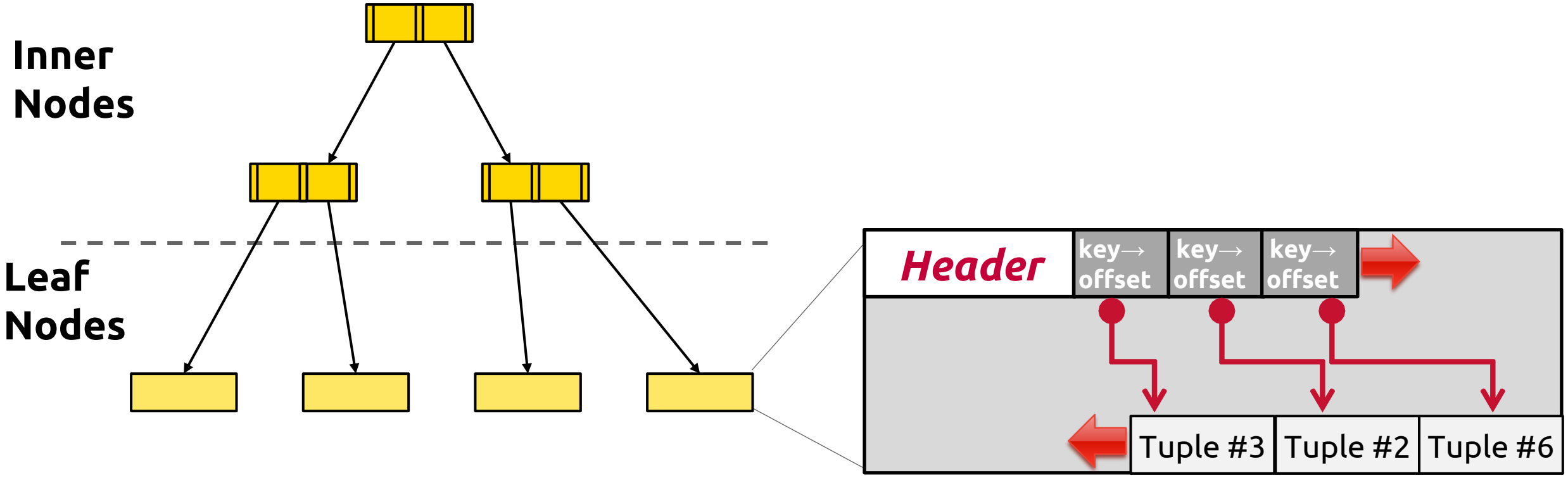
Observation

- The two-table storage approach relies on indexes to find individual tuples.
  - Such indexes are necessary because the tables are inherently unsorted.
- But what if the DBMS could keep tuples sorted automatically using an index?

Index-organized Storage

- DBMS stores a table's tuples as the value of an index data structure.
  - Still use a page layout that looks like a slotted page.
  - Tuples are typically sorted in page based on key.
- B+Tree pays maintenance costs upfront, whereas LSMs pay for it later.

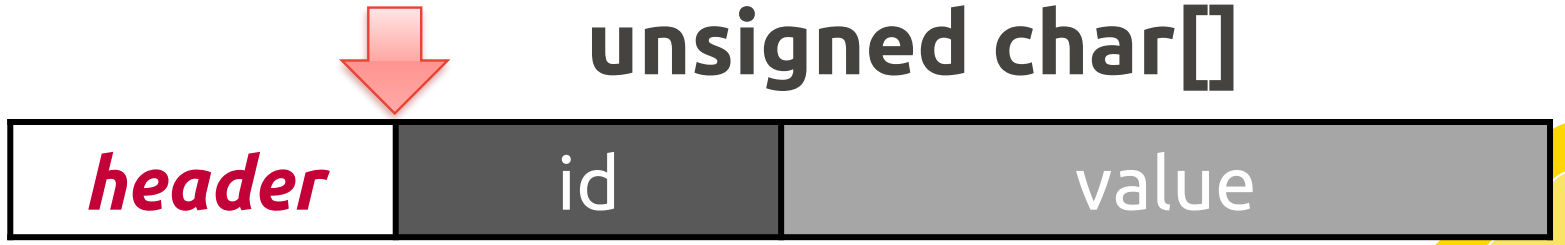
Index-organized Storage



Tuple storage

- A tuple is essentially a sequence of bytes prefixed with a header that contains meta-data about it.
- It is the job of the DBMS to interpret those bytes into attribute types and values.
- The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.

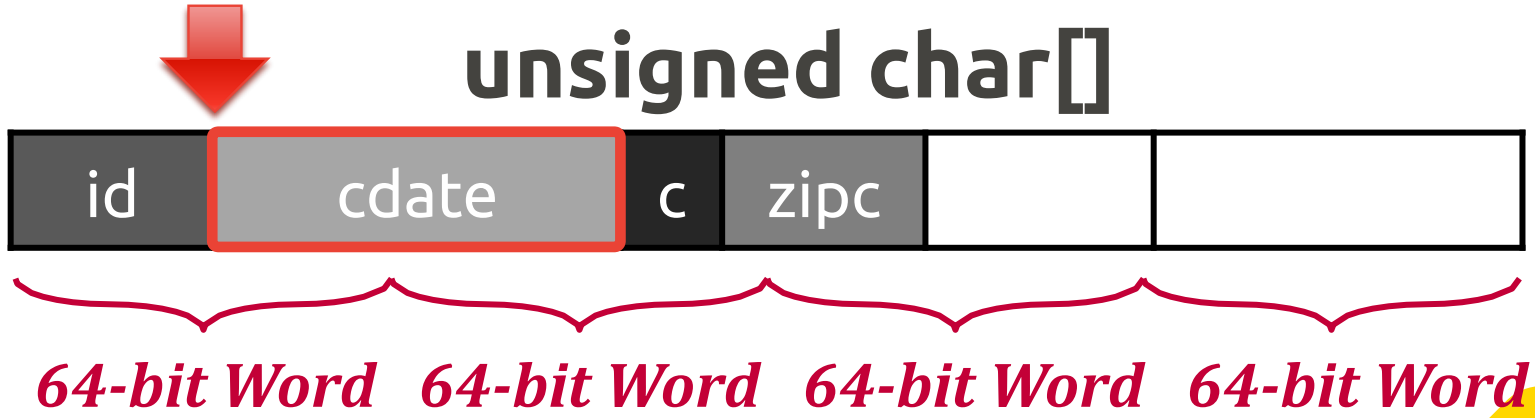
```
CREATE TABLE foo (
   id INT PRIMARY KEY,
   value BIGINT
);
```



Word-aligned tuples

- All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

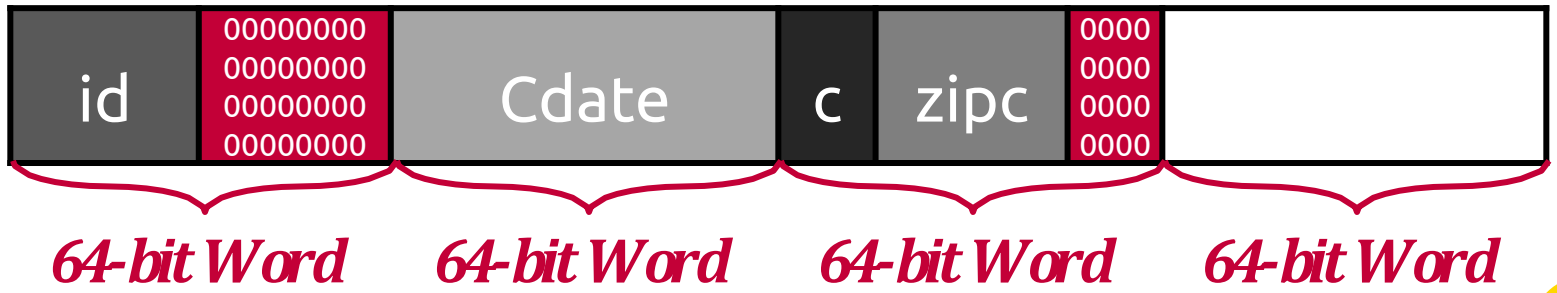
```
CREATE TABLE foo (
32-bits id INT PRIMARY KEY,
64-bits cdate TIMESTAMP,
16-bits color CHAR(2),
32-bits zipcode INT
);
```



Word-alignment: Padding

- Add empty bits after attributes to ensure that tuple is word aligned.
- Essentially round up the storage size of types to the next largest word size.

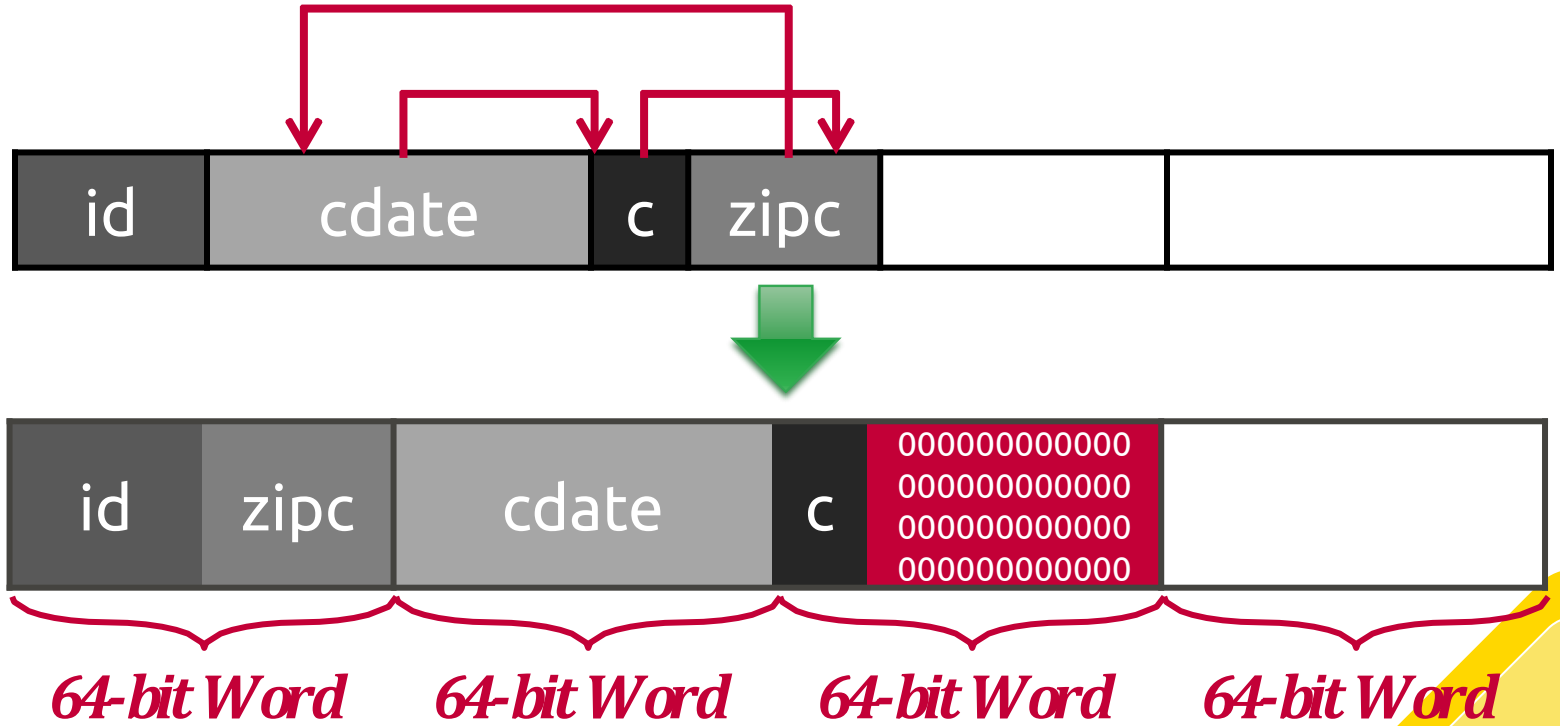
```
CREATE TABLE foo (
32-bits  id INT PRIMARY KEY,
64-bits  cdate TIMESTAMP,
16-bits  color CHAR(2),
32-bits  zipcode INT
);
```



Word-alignment: Reordering

- Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
  - ★ May still have to use padding to fill remaining space.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,      -- 32-bits
  cdate TIMESTAMP,         -- 64-bits
  color CHAR(2),           -- 16-bits
  zipcode INT              -- 32-bits
);
```



Storage models

Storage models

- A DBMS's storage model specifies how it physically organizes tuples on disk and in memory.
  - ★ N-ary Storage Model (**NSM**)
  - ★ Decomposition Storage Model (**DSM**)
  - ★ Hybrid Storage Model (**PAX**)
- Can have different performance characteristics based on the target workload (OLTP vs. OLAP).
- Influences the design choices of the rest of the DBMS.

N-ary Storage Model (NSM)

N-ary Storage Model (NSM)

- The DBMS stores (almost) all attributes for a single tuple contiguously in a single page.
  - ★ Also commonly known as a ***row store***
- Ideal for OLTP workloads where queries are more likely to access individual entities and execute write-heavy workloads.
- NSM database page sizes are typically some constant multiple of ***4 KB*** hardware pages.

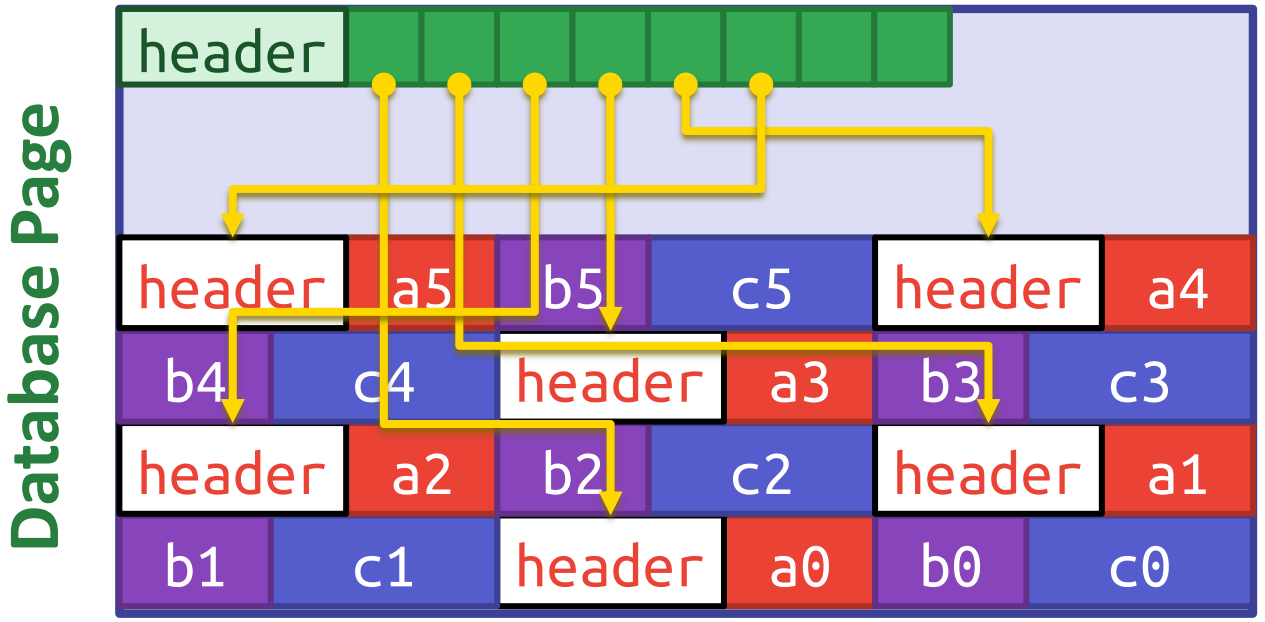
NSM: Physical Organization

- A disk-oriented NSM system stores a tuple's fixed-length and variable-length attributes contiguously in a single slotted page.

- The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

NSM: Physical Organization

	Col A	Col B	Col C
Row#0	a0	b0	c0
Row#1	a1	b1	c1
Row#2	a2	b2	c2
Row#3	a3	b3	c3
Row#4	a4	b4	c4
Row#5	a5	b5	c5



NSM: OLTP example

```
SELECT * FROM useracct
 WHERE userName = ?
 AND userPass = ?
```

```
INSERT INTO useracct
 VALUES (?,?,...?)
```

Index

Disk

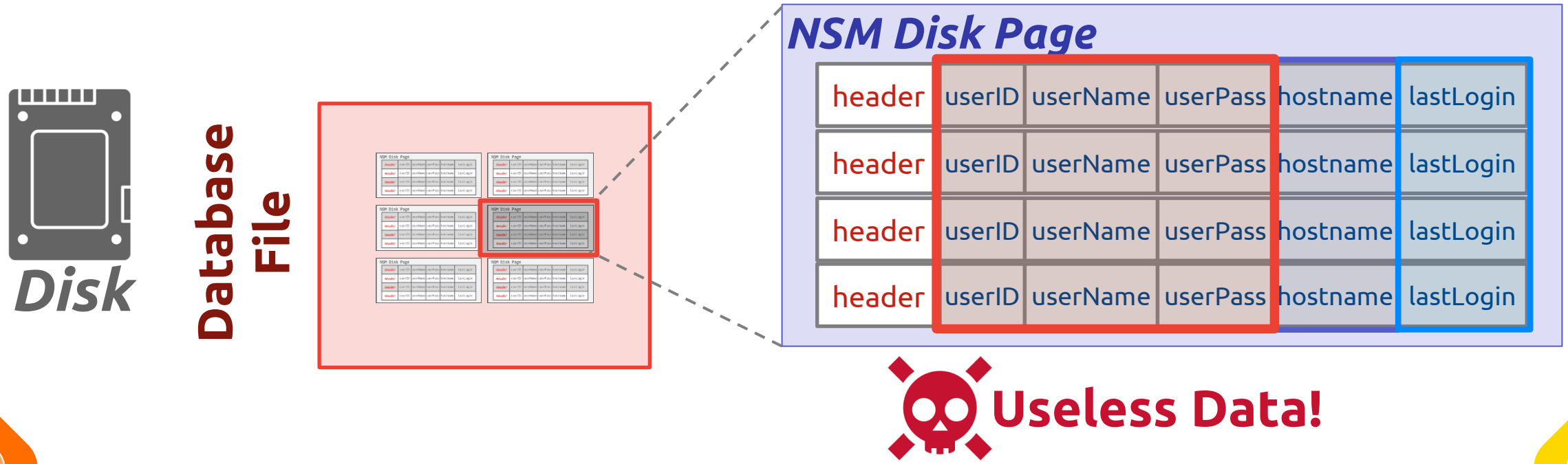
Database File

NSM Disk Page

header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	-	-	-	-	-

NSM: OLAP example

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
  WHERE U.hostname LIKE '%.gov'
  GROUP BY EXTRACT(month FROM U.lastLogin)
```



NSM: summary

- Advantages
  - Fast inserts, updates, and deletes.
  - Good for queries that need the entire tuple (OLTP).
  - Can use index-oriented physical storage for clustering.
- Disadvantages
  - Not good for scanning large portions of the table and/or a subset of the attributes.
  - Terrible memory locality in access patterns.
  - Not ideal for compression because of multiple value domains within a single page.

Decomposition Storage Model (DSM)

Decomposition Storage Model (DSM)

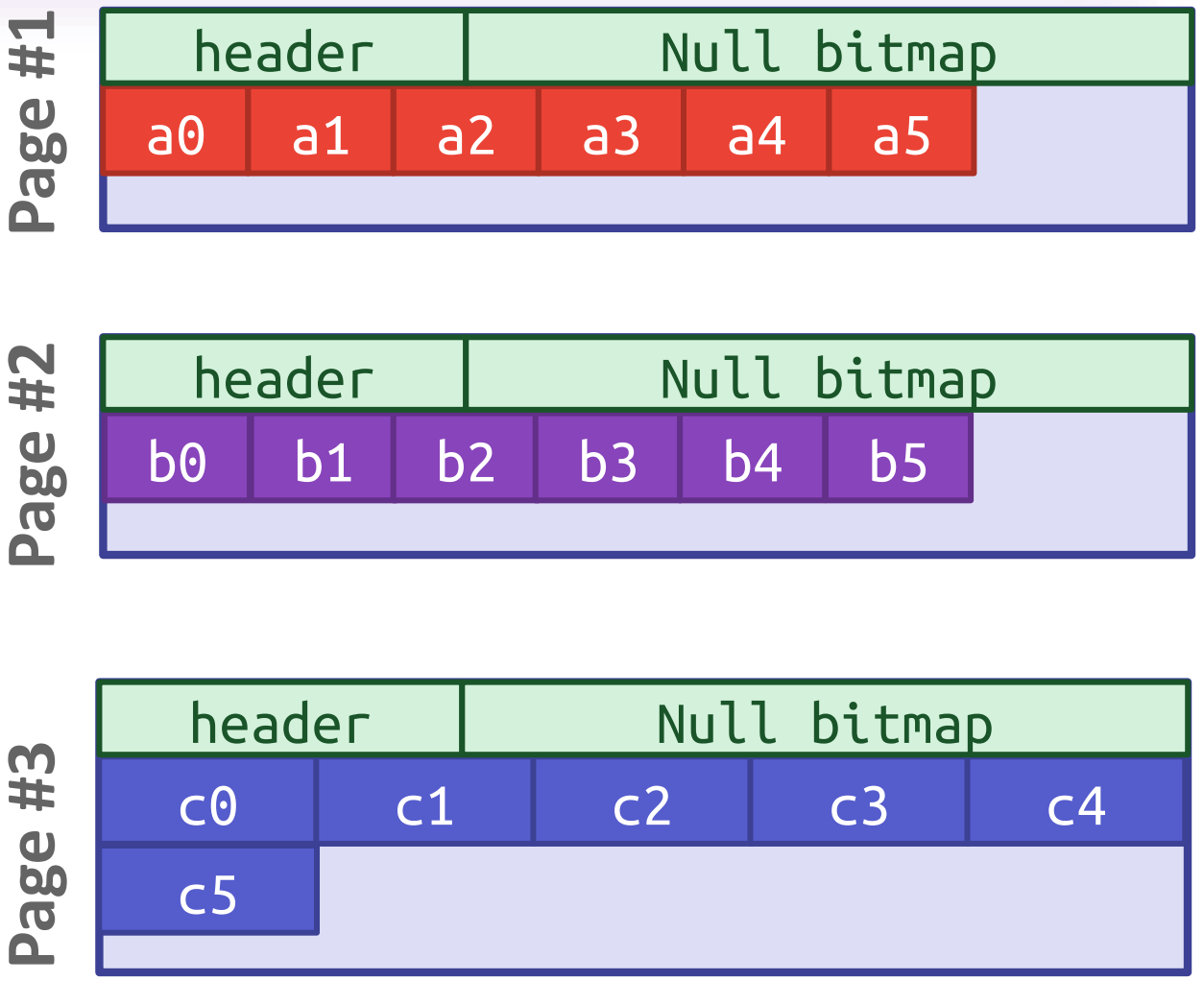
- Store a single attribute for all tuples contiguously in a block of data.
  - Also known as a "***column store***"
- Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.
- DBMS is responsible for combining/splitting a tuple's attributes when reading/writing.

DSM: Physical Organization

- Store attributes and meta-data (e.g., nulls) in separate arrays of fixed- length values.
  - Most systems identify unique physical tuples using offsets into these arrays.
  - Need to handle variable-length values.
- Maintain separate pages per attribute with a dedicated header area for meta-data about entire column.

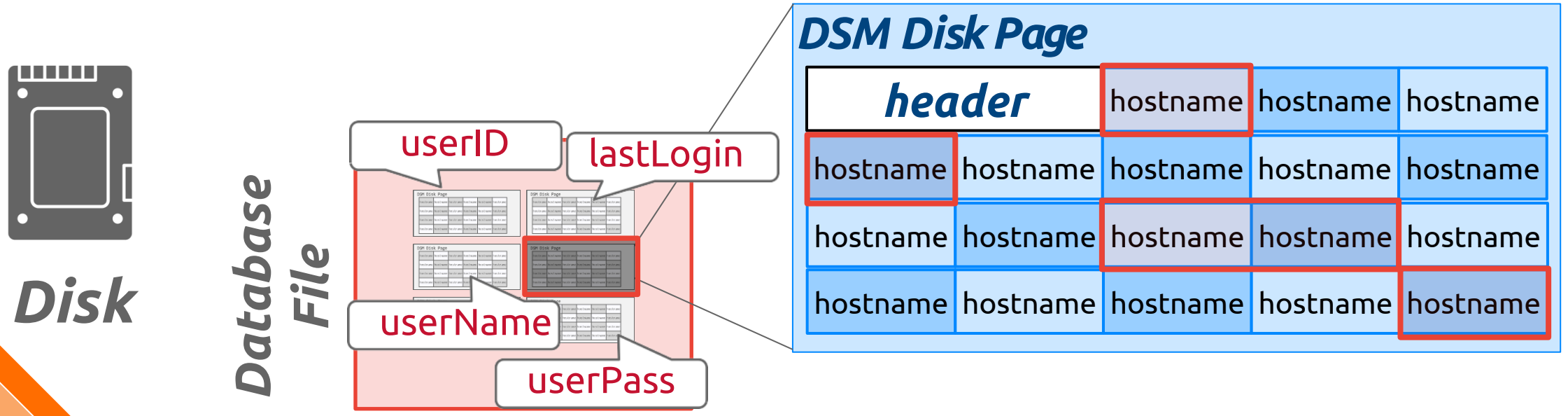
DSM: Physical Organization

	Col A	Col B	Col C
Row#0	a0	b0	c0
Row#1	a1	b1	c1
Row#2	a2	b2	c2
Row#3	a3	b3	c3
Row#4	a4	b4	c4
Row#5	a5	b5	c5



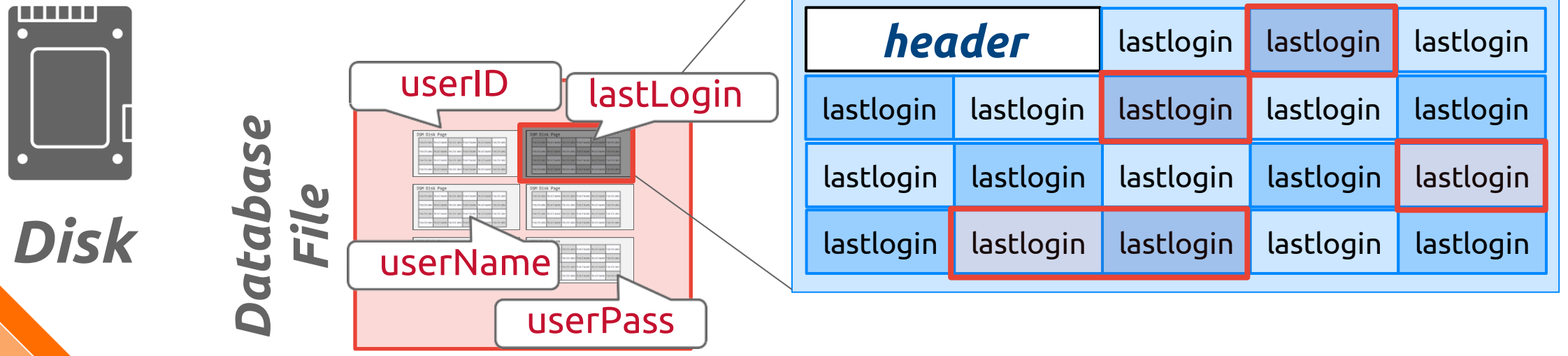
DSM: OLAP example

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



DSM: OLAP example

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



DSM: Variable-length Data

- Padding variable-length fields to ensure they are fixed-length is wasteful, especially for large attributes.
- A better approach is to use ***dictionary compression*** to convert repetitive variable-length data into fixed- length values (typically 32-bit integers).

DSM: Summary

- Advantages
  - Reduces the amount wasted I/O per query because the DBMS only reads the data that it needs.
  - Faster query processing because of increased locality and cached data reuse.
  - Better data compression.
- Disadvantages
  - Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching/reorganization.

OBSERVATION

- OLAP queries almost never access a single column in a table by itself.
  - At some point during query execution, the DBMS must get other columns and stitch the original tuple back together.
- But we still need to store data in a columnar format to get the storage + execution benefits.
- We need columnar scheme that still stores attributes separately but keeps the data for each tuple physically close to each other.

Partition Attributes Across (PAX)

PAX Storage Model

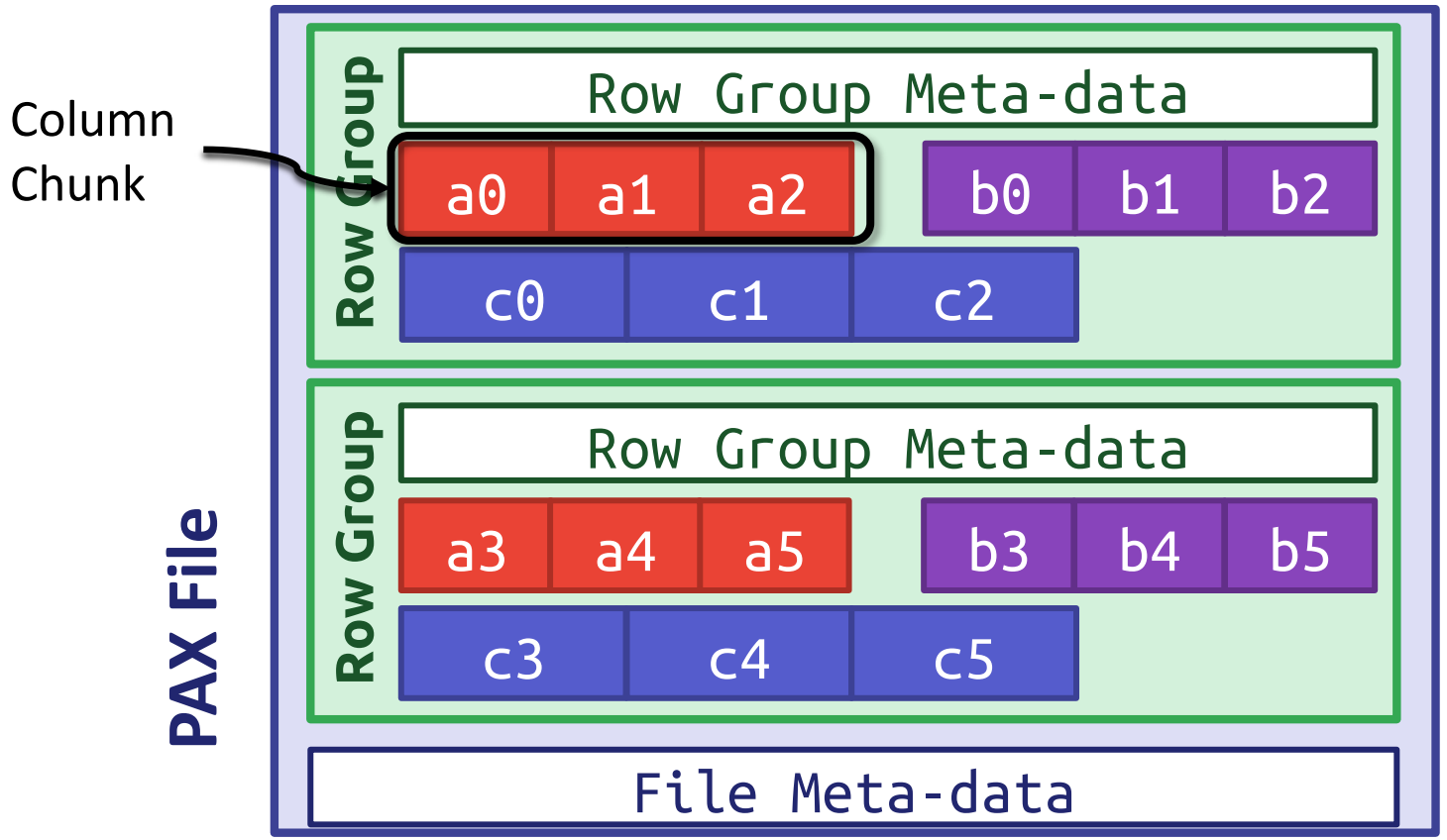
- Partition Attributes Across (PAX) is a hybrid storage model that vertically partitions attributes within a database page.
  - ★ Examples: Parquet, ORC, and Arrow.
- The goal is to get the benefit of ***faster processing*** on columnar storage while retaining the ***spatial locality*** benefits of row storage.

PAX: Physical Organization

- Horizontally partition data into ***row groups***.
- Then vertically partition their attributes into ***column chunks***.
- Global meta-data directory contains offsets to the file's row groups.
  - ★ This is stored in the footer if the file is immutable (Parquet, Orc).
- Each row group contains its own meta-data header about its contents.

PAX: Physical Organization

	Col A	Col B	Col C
Row#0	a0	b0	c0
Row#1	a1	b1	c1
Row#2	a2	b2	c2
Row#3	a3	b3	c3
Row#4	a4	b4	c4
Row#5	a5	b5	c5



Observation

- I/O is the main bottleneck if the DBMS fetches data from disk during query execution.
- The DBMS can ***compress*** pages to increase the utility of the data moved per I/O operation.
- Key trade-off is **speed** vs. **compression ratio**
  - Compressing the database reduces DRAM requirements.
  - It may decrease CPU costs during query execution.

Database Compression

Database Compression

- Goal #1: Must produce fixed-length values.
  - Only exception is var-length data stored in separate pool.
- Goal #2: Postpone decompression for as long as possible during query execution.
  - Also known as ***late materialization***.
- Goal #3: Must be a ***lossless*** scheme.
  - People (typically) don't like losing data.
  - Any ***lossy*** compression must be performed by application.

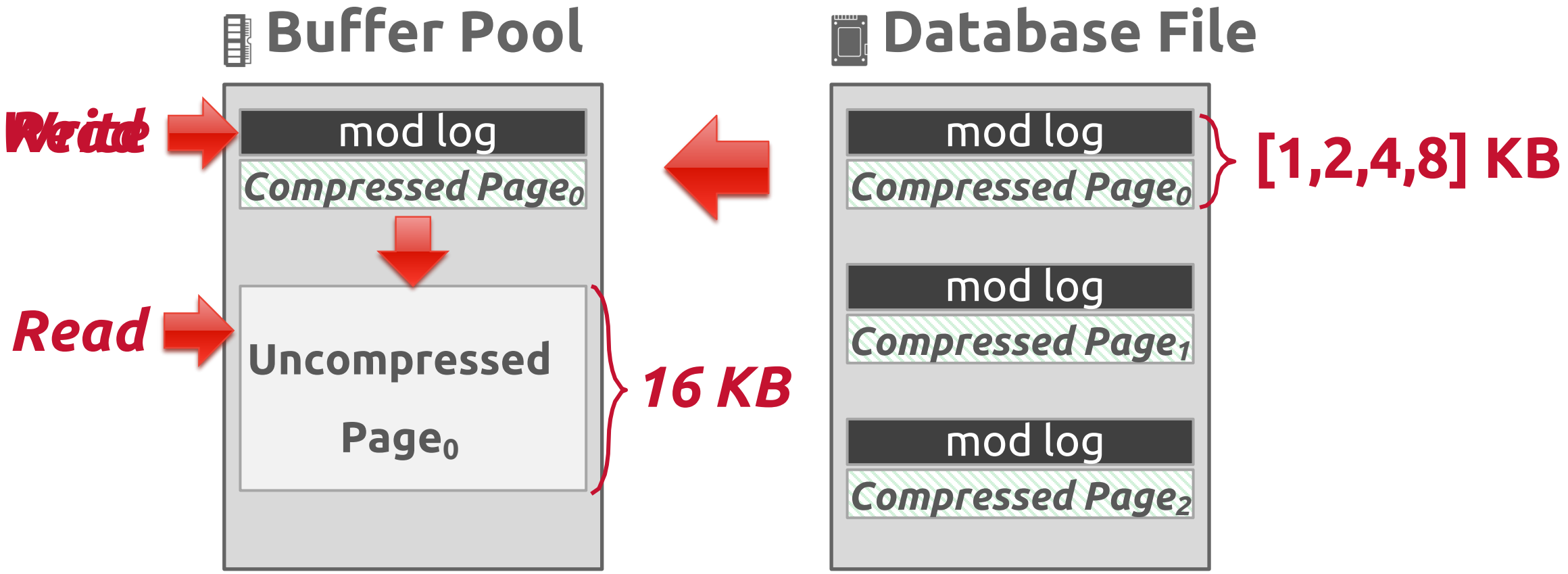
Compression Granularity

- Block-level
  - Compress a block of tuples for the same table.
- Tuple-level
  - Compress the contents of the entire tuple (NSM-only).
- Attribute-level
  - Compress a single attribute within one tuple (overflow).
  - Can target multiple attributes for the same tuple.
- Column-level
  - Compress multiple values for one or more attributes stored for multiple tuples (DSM-only).

NAÏVE Compression

- Compress data using a general-purpose algorithm.
- Scope of compression is only based on the data provided as input.
  - LZO (1996), LZ4 (2011), Snappy (2011), Oracle OZIP (2014), Zstd (2015)
- Considerations
  - Computational overhead
  - Compress vs. decompress speed.

MYSQL Innodb Compression

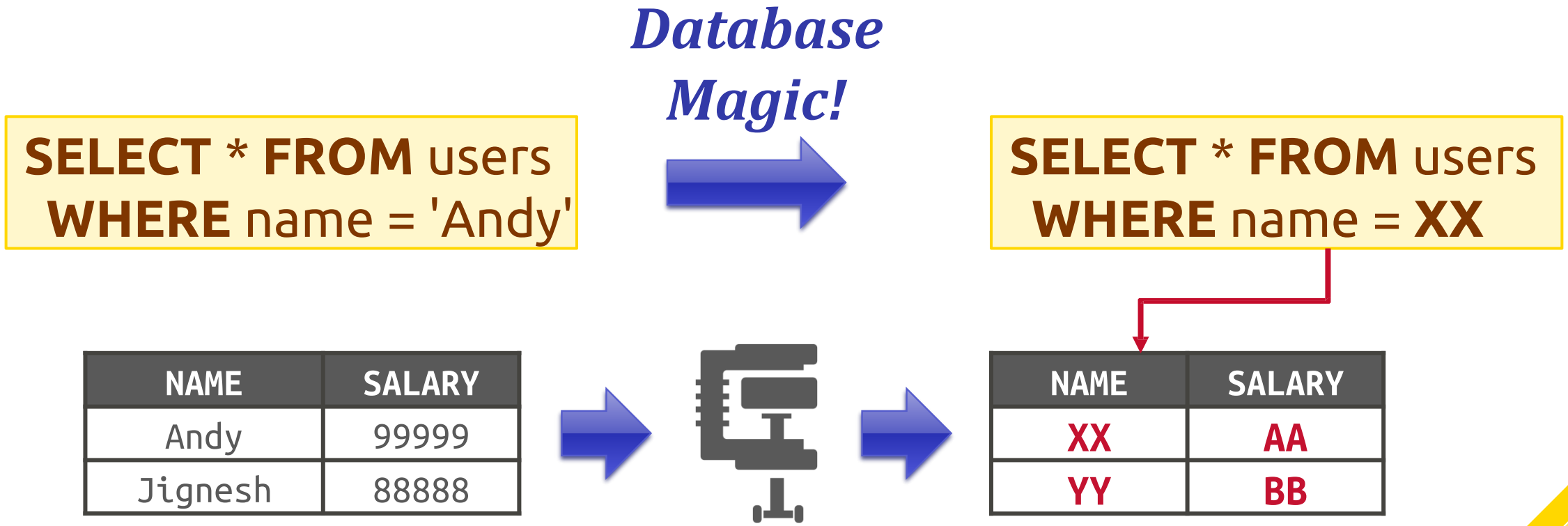


NAÏVE Compression

- The DBMS must decompress data first before it can be read and (potentially) modified.
  - This limits the "scope" of the compression scheme.
- These schemes also do not consider the high-level meaning or semantics of the data.

Observation

- Ideally, we want the DBMS to operate on compressed data without decompressing it first.



Compression Granularity

- Block-level
  - Compress a block of tuples for the same table.
- Tuple-level
  - Compress the contents of the entire tuple (NSM-only).
- Attribute-level
  - Compress a single attribute within one tuple (overflow).
  - Can target multiple attributes for the same tuple.
- Column-level
  - Compress multiple values for one or more attributes stored for multiple tuples (DSM-only).

Columnar Compression

Columnar Compression

- Run-length Encoding
- Bit-Packing Encoding
- Bitmap Encoding
- Delta / Frame-of-Reference Encoding
- Incremental Encoding
- Dictionary Encoding

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.

Run-length Encoding

```
SELECT isDead, COUNT(*)
FROM users
GROUP BY isDead
```

Compressed Data

id	isDead
1	(Y,0,3)
2	(N,3,1)
3	(Y,4,1)
4	(N,5,1)
6	(Y,6,2)
7	
8	
9	

RLE Triplet
- Value
- Offset
- Length

Original Data

id	isDead
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Run-length Encoding

Sorted Data

id	isDead
1	Y
2	Y
3	Y
6	Y
8	Y
9	Y
4	N
7	N

Compressed Data

id	isDead
1	(Y,0,6)
2	(N,7,2)
3	
6	
8	
9	
4	
7	

Bit Packing

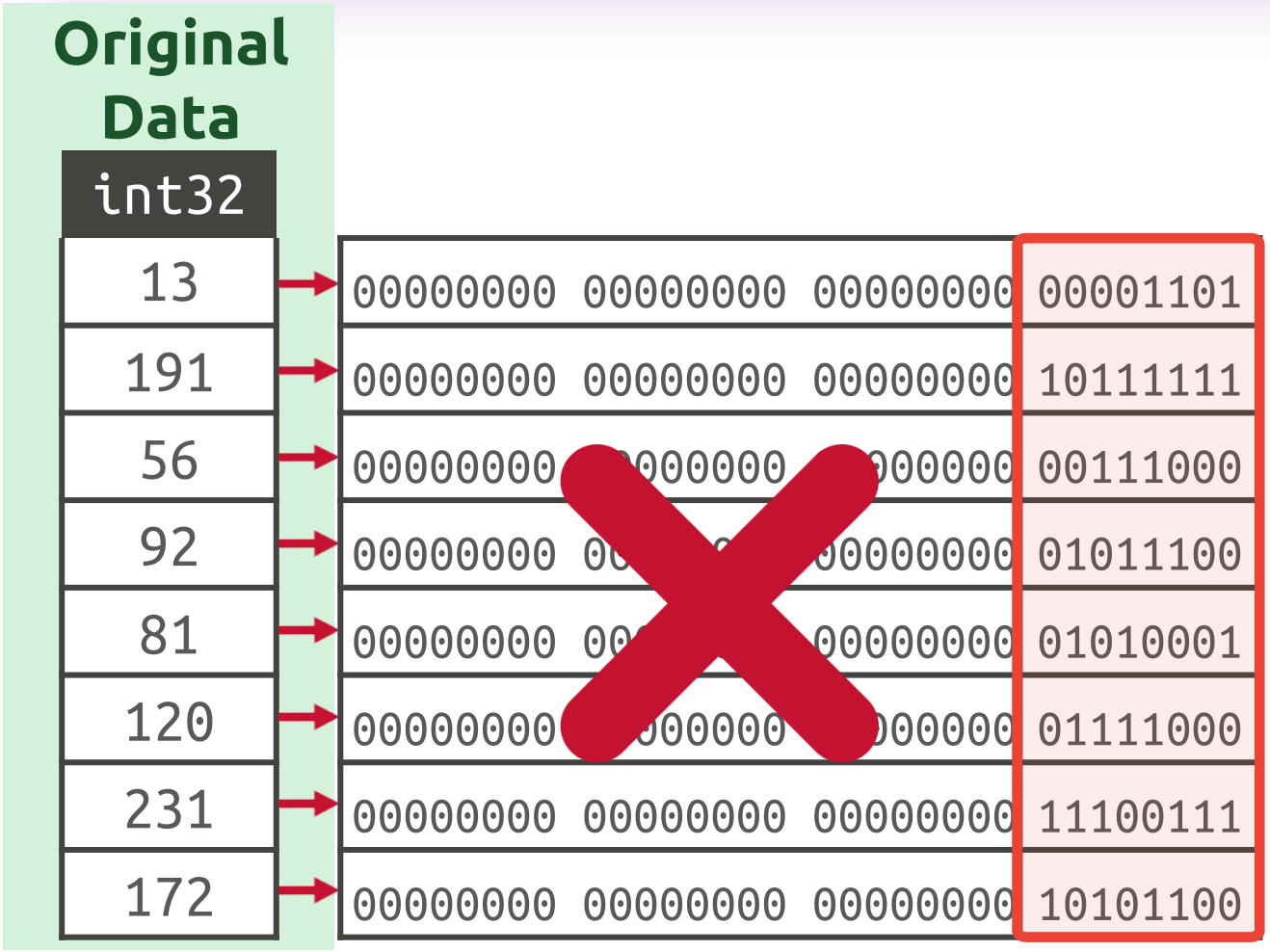
- If the values for an integer attribute is ***smaller*** than the range of its given data type size, then reduce the number of bits to represent each value.

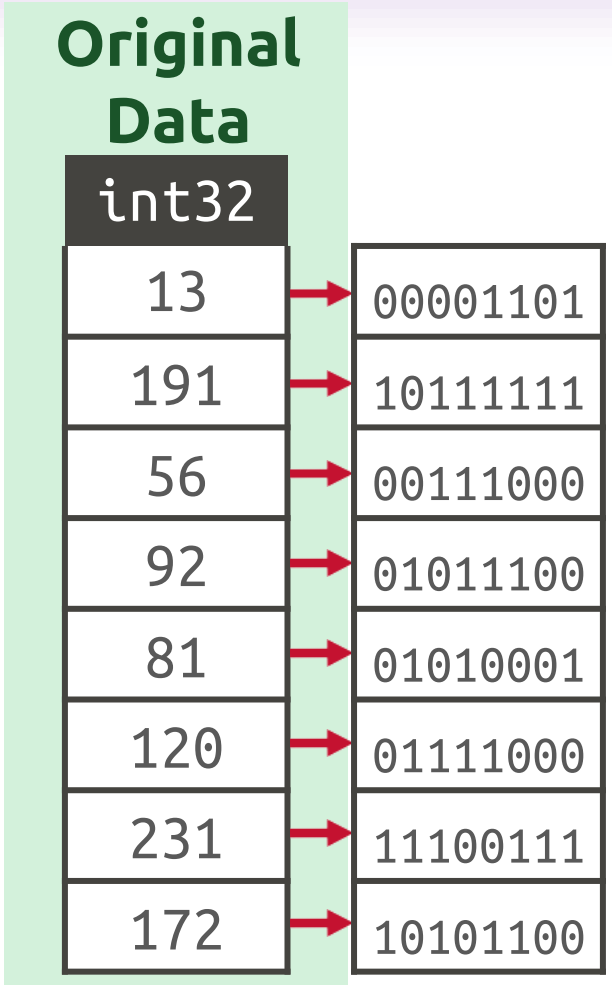
- Use bit-shifting tricks to operate on multiple values in a single word.

Original Data

int32
13
191
56
92
81
120
231
172

Bit Packing





Original:

8 × 32-bits = 256 bits

Compressed:

8 × 8-bits = 64 bits

Patching / mostly Encoding

- A variation of bit packing for when an attribute's values are "mostly" less than the largest size, store them with smaller data type.
  - The remaining values that cannot be compressed are stored in their raw form.

Patching / mostly Encoding

Original Data

int32
13
191
999999999
92
81
120
231
172

Original:

8 × 32-bits = 256 bits

Compressed Data

mostly8
13
191
XXX
92
81
120
231
172

offset	*value*
3	99999999

Compressed:

(8 × 8-bits) +16-bits + 32-bits

= 112 bits

Bitmap Encoding

- 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.
- Only practical if the value cardinality is low.
- Some DBMSs provide ***bitmap indexes***.

Bitmap encoding

Original Data

id	isDead
1	Y
2	Y
3	Y
4	N
6	Y
7	N
8	Y
9	Y

Original:

8 × 8-bits = 64 bits

Compressed Data

id	isDead: Y	isDead: 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

2 × 8-bits
= 16-bits

8 × 2-bits
= 16-bits

Compressed:

16-bits +16-bits = 32-bits

Bitmap encoding: Example

- Assume we have 10 million tuples. 43,000 zip codes in the US.
  - 10000000 × 32-bits = 40 MB
  - **10000000 × 43000 = 53.75 GB**
- Every time the application inserts a new tuple, the DBMS must extend 43,000 different bitmaps.
- There are compressed data structures for sparse data sets:
  - **Roaring Bitmaps**

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

Delta Encoding

- Recording the difference between values that follow each other in the same column.
  - Store base value in-line or in a separate look-up table.
  - Combine with RLE to get even better compression ratios.
- Frame-of-Reference Variant: Use global min value.

Delta Encoding

Original Data

time64	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

Compressed Data

time64	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

Compressed Data

time64	temp
12:00	99.5
(+1,4)	-0.1
	+0.1
	+0.1
	-0.2

5 × 64bits
= 320 bits

64bits + (4 × 16bits)
= 128 bits

64bits + (2 × 16bits)
= 96 bits

Dictionary Compression

- Replace frequent values with smaller fixed-length codes and then maintain a mapping (dictionary) from the codes to the original values
  - Typically, one code per attribute value.
  - Most widely used native compression scheme in DBMSs.
- The ideal dictionary scheme supports fast encoding and decoding for both point and range queries.
  - **Encode/Locate**: For a given uncompressed value, convert it into its compressed form.
  - **Decode/Extract**: For a given compressed value, convert it back into its original form.

Dictionary: Order-preserving

- The encoded values need to support the same collation as the original values.

```
SELECT * FROM users
  WHERE name LIKE 'And%'
```

```
SELECT * FROM users
  WHERE name BETWEEN 10 AND 20
```

Original Data

name
Andrea
Mr.Pickles
Andy
Jignesh
Mr.Pickles

Compressed Data

name
10
40
20
30
40

value	*code*
Andrea	10
Andy	20
Jignesh	30
Mr.Pickles	40

Sorted Dictionary

Order-preserving encoding

```
SELECT name FROM users
  WHERE name LIKE 'And%'
```

Still must perform scan on column

```
SELECT DISTINCT name FROM users
  WHERE name LIKE 'And%'
```

Only need to access dictionary

Original Data

name
Andrea
Mr.Pickles
Andy
Jignesh
Mr.Pickles

Compressed Data

name
10
40
20
30
40

value	*code*
Andrea	10
Andy	20
Jignesh	30
Mr.Pickles	40

Sorted Dictionary

Conclusion

- It is important to choose the right storage model for the target workload:
  - OLTP → Row Store
  - OLAP → Column Store
- DBMSs can combine different approaches for even better compression.
- Dictionary encoding is probably the most useful scheme because it does not require pre-sorting.

THANK YOU