Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #05

Storage Models & Compression

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel

ADMINISTRIVIA

Homework #1 is due September 15th @ 11:59pm.

Project #1 is due October 1st @ 11:59pm.

UPCOMING DATABASE TALKS

OtterTune (ML⇄DB Seminar)
→ Monday Sept 18th @4:30pm

LAST CLASS

We discussed alternatives to tuple-oriented storage scheme.
→ Log-structured storage
→ Index-organized storage

These approaches are ideal for write-heavy (**INSERT**/**UPDATE**/**DELETE**) workloads.

But the most important query for an application may be read (**SELECT**) performance…

DATABASE WORKLOADS

On-Line Transaction Processing (OLTP)

→ Fast operations that only read/update a small amount of data each time.

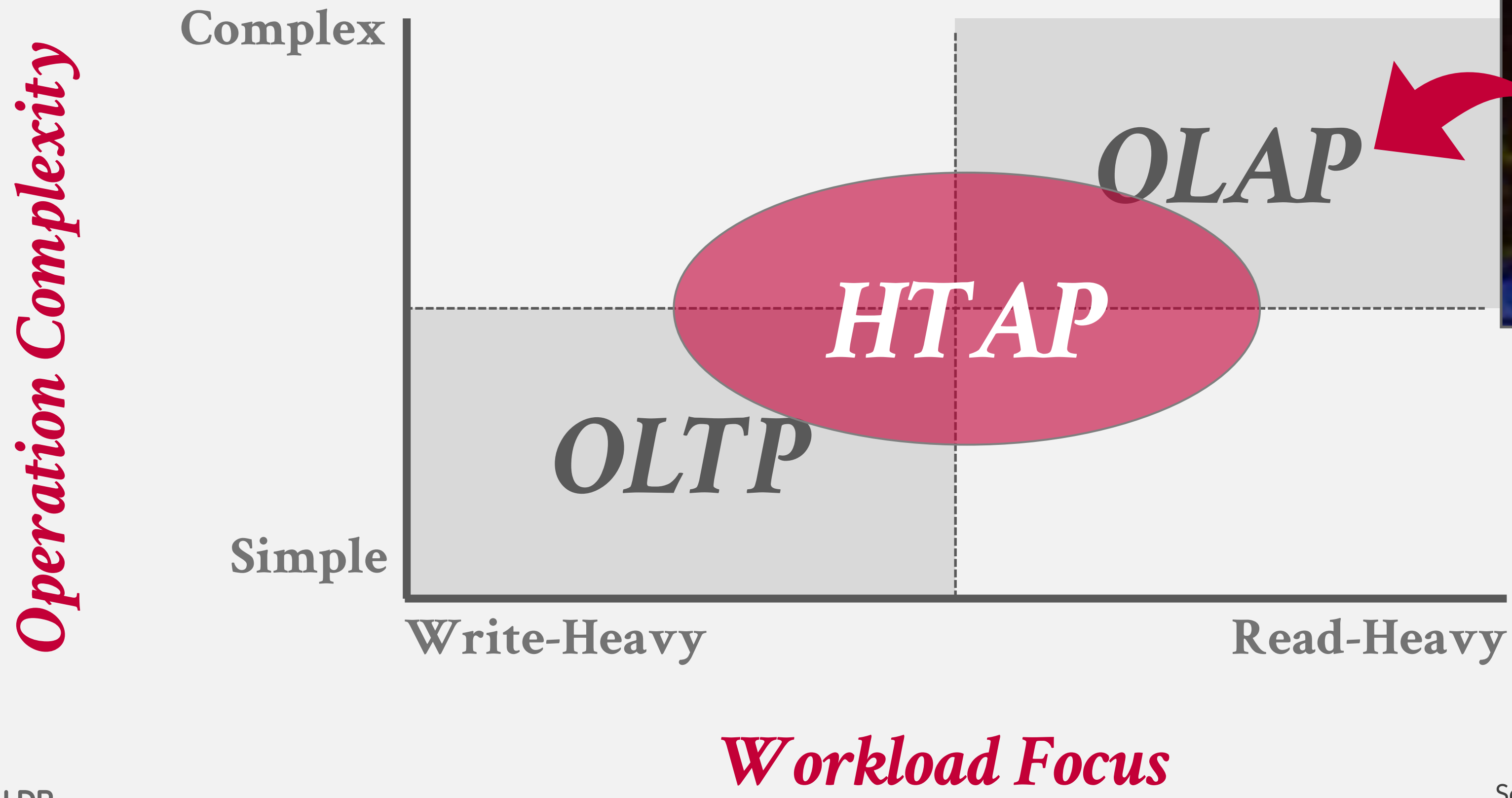
On-Line Analytical Processing (OLAP)

→ Complex queries that read a lot of data to compute aggregates.

Hybrid Transaction + Analytical Processing

→ OLTP + OLAP together on the same database instance

DATABASE WORKLOADS



Jim Gray

Source: Mike Stonebraker

WIKIPEDIA EXAMPLE

```
CREATE TABLE useracct (
  userID INT PRIMARY KEY,
  userName VARCHAR UNIQUE,
  ⋮
);
```

```
CREATE TABLE pages (
  pageID INT PRIMARY KEY,
  title VARCHAR UNIQUE,
  latest INT
  ↳REFERENCES revisions (revID),
);
```

```
CREATE TABLE revisions (
  revID INT PRIMARY KEY,
  userID INT REFERENCES useracct (userID),
  pageID INT REFERENCES pages (pageID),
  content TEXT,
  updated DATETIME
);
```

OBSERVATION

The relational model does <u>not</u> specify that the DBMS must store all a tuple's attributes together in a single page.

This may <u>not</u> actually be the best layout for some workloads…

OLTP

On-line Transaction Processing:

→ Simple queries that read/update a small amount of data that is related to a single entity in the database.

This is usually the kind of application that people build first.

```
SELECT P.*, R.*
  FROM pages AS P
 INNER JOIN revisions AS R
    ON P.latest = R.revID
 WHERE P.pageID = ?
```

```
UPDATE useracct
   SET lastLogin = NOW(),
       hostname = ?
 WHERE userID = ?
```

```
INSERT INTO revisions VALUES
(?,?…,?)
```

OLAP

On-line Analytical Processing:

→ Complex queries that read large portions of the database spanning multiple entities.

You execute these workloads on the data you have collected from your OLTP application(s).

```
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)
```

STORAGE MODELS

A DBMS's **storage model** specifies how it physically organizes tuples on disk and in memory.

→ Can have different performance characteristics based on the target workload (OLTP vs. OLAP).
→ Influences the design choices of the rest of the DBMS.

Choice #1: N-ary Storage Model (NSM)
Choice #2: Decomposition Storage Model (DSM)
Choice #3: Hybrid Storage Model (PAX)

N-ARY STORAGE MODEL (NSM)

The DBMS stores (almost) all attributes for a single tuple contiguously in a single page.
→ Also known as a "row store"

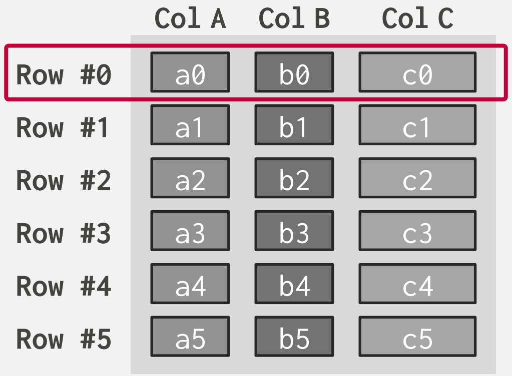
Ideal for OLTP workloads where queries are more likely to access individual entities and execute write-heavy workloads.

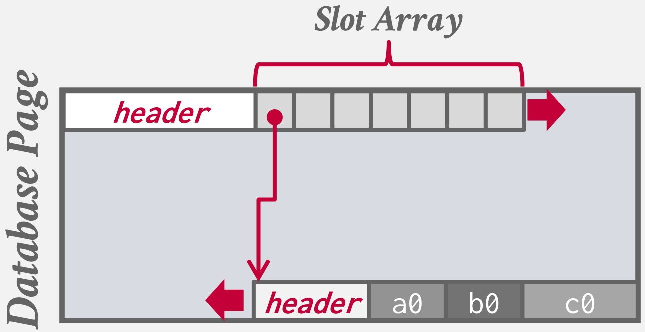
NSM database page sizes are typically some constant multiple of **4 KB** hardware pages.
→ Oracle (4 KB), Postgres (8 KB), MySQL (16 KB)

NSM: PHYSICAL ORGANIZATION

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

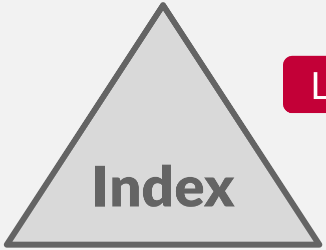
The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.





NSM: OLTP EXAMPLE

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



Lecture #8

NSM Disk Page

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

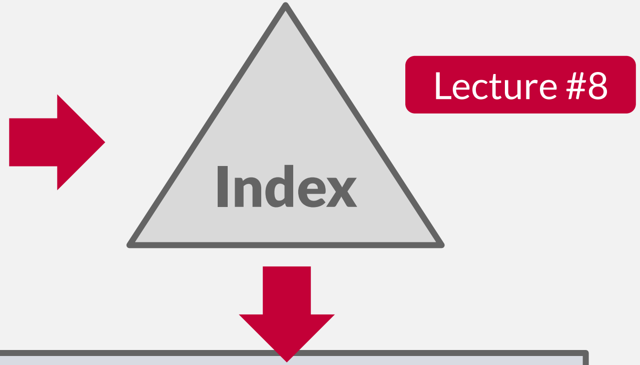
Disk

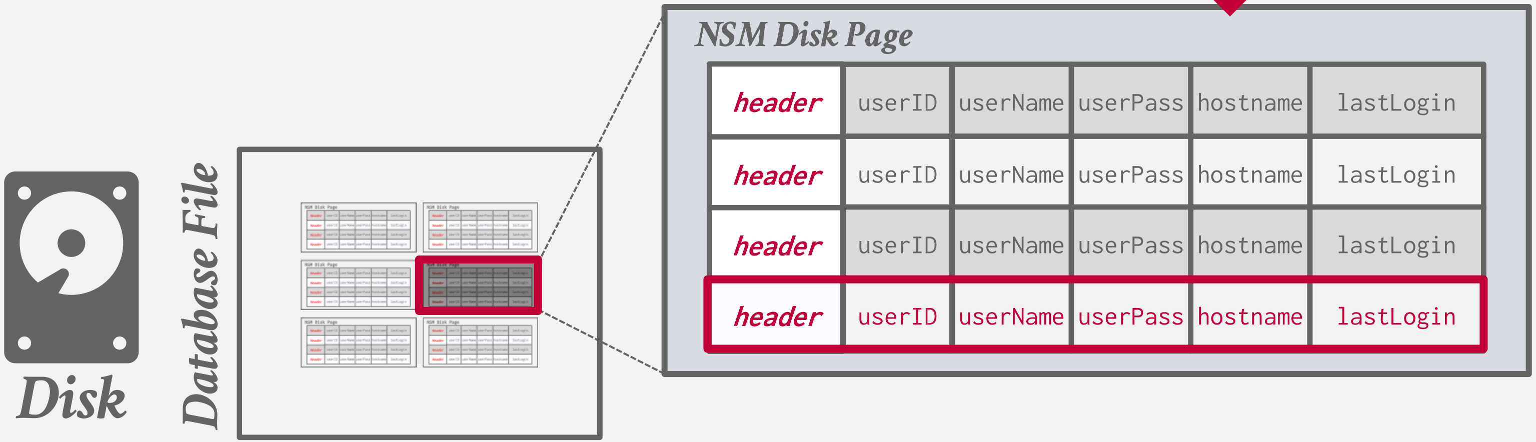
Database File

NSM: OLTP EXAMPLE

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

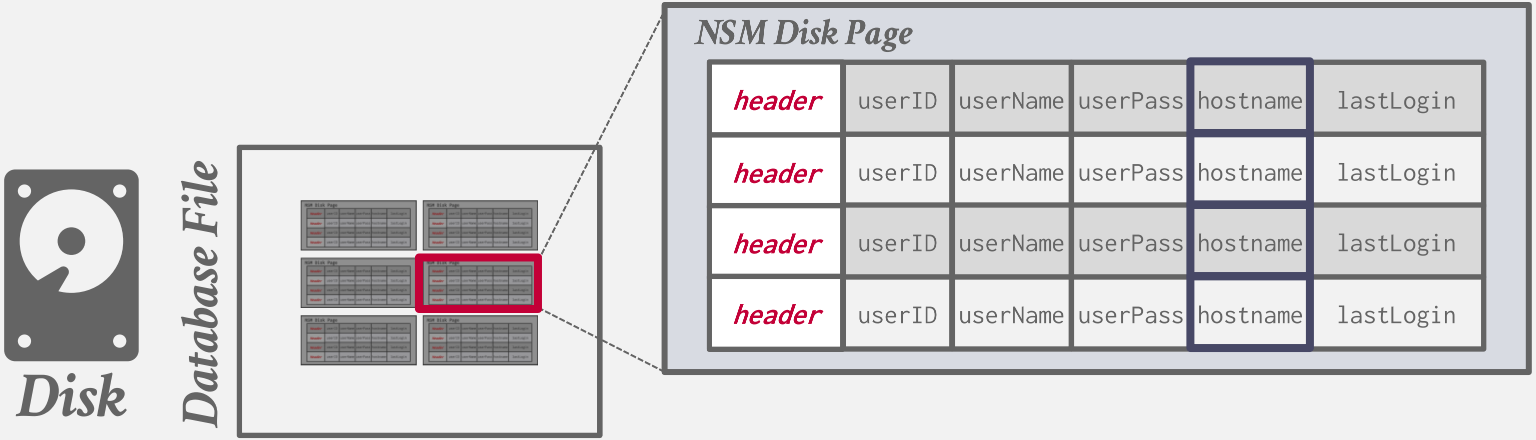
```
INSERT INTO useracct
VALUES (?,?,…?)
```





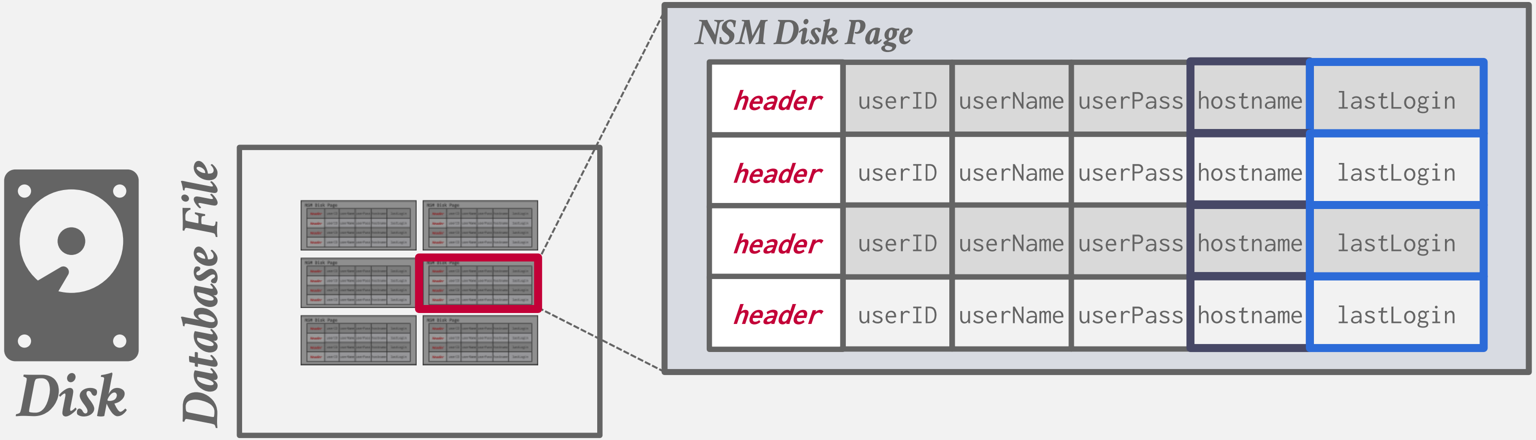
NSM: OLAP EXAMPLE

```sql
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)
```



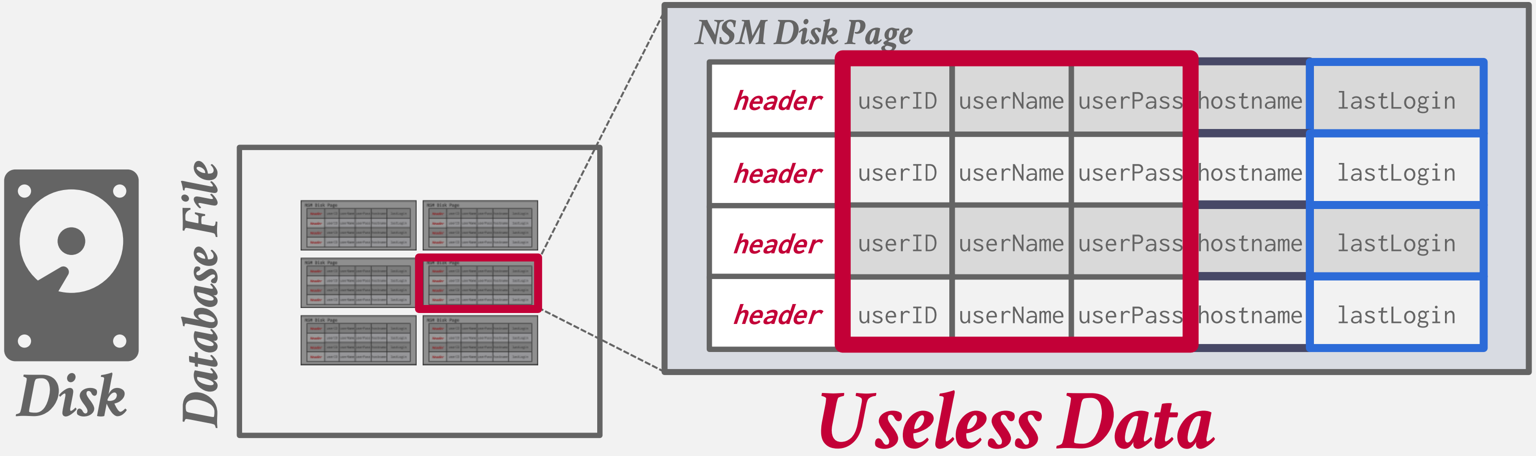
NSM: OLAP EXAMPLE

```
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: 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)

The DBMS stores 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.

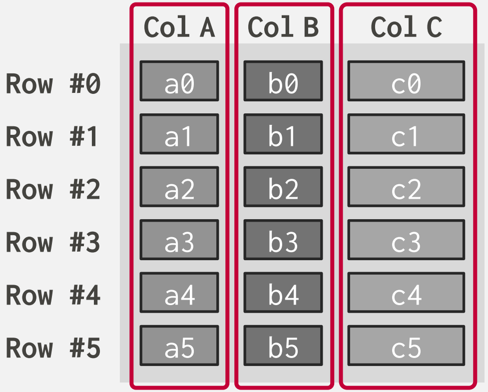
DBMS is responsible for combining/splitting a tuple's attributes when reading/writing.

DSM: PHYSICAL ORGANIZATION

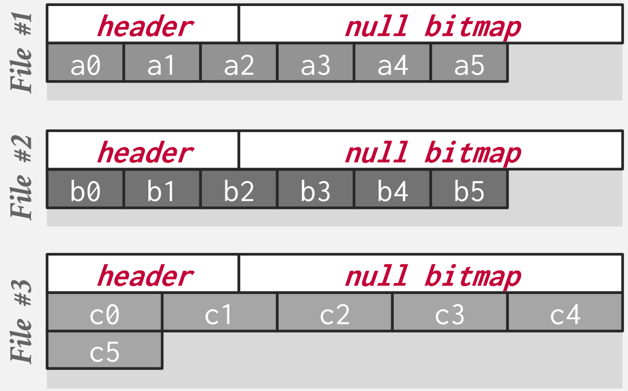
Store attributes and meta-data (e.g., nulls) in separate arrays of **fixed-length** values.

→ Most systems identify unique physical tuples using offsets into these arrays.

→ Need to handle variable-length values…



Maintain a separate file per attribute with a dedicated header area for meta-data about entire column.



DSM: DATABASE EXAMPLE

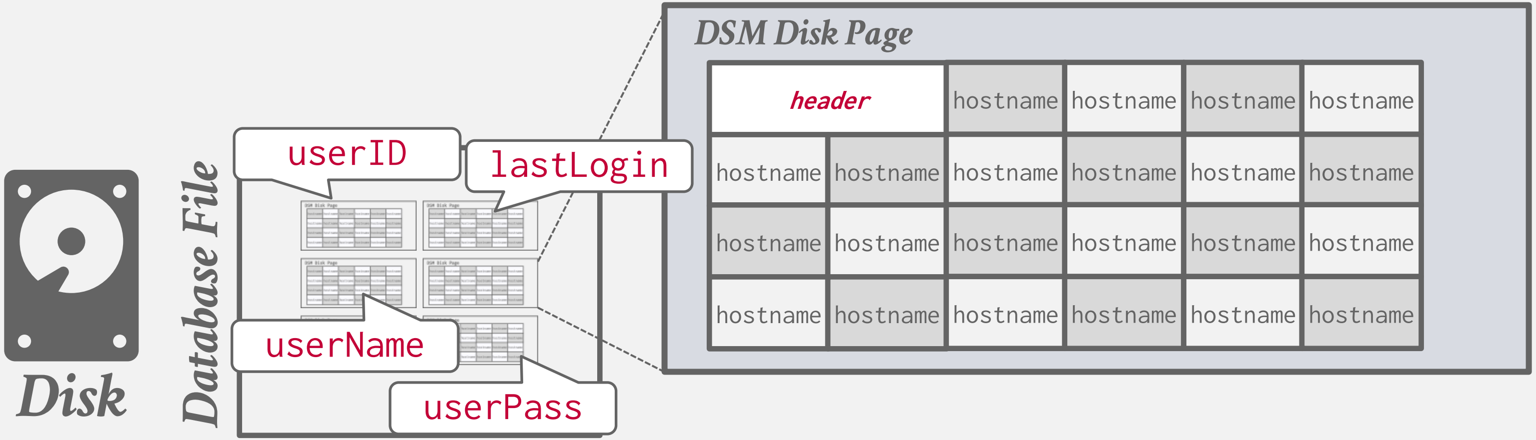
The DBMS stores the values of a single attribute across multiple tuples contiguously in a page.

→ Also known as a "column store".

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

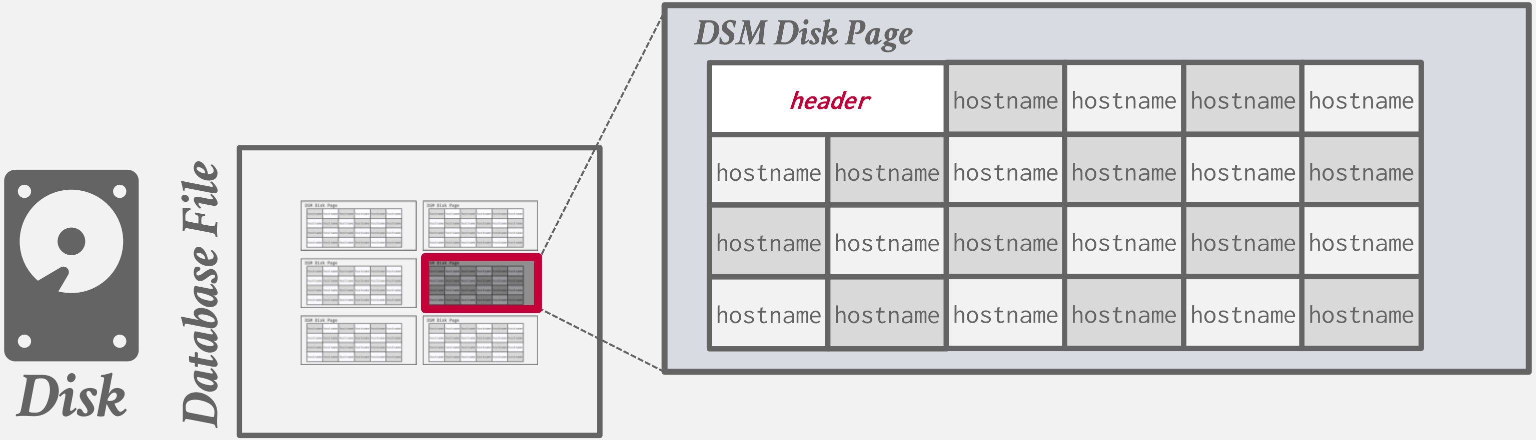
DSM: DATABASE EXAMPLE

The DBMS stores the values of a single attribute across multiple tuples contiguously in a page.
→ Also known as a "column store".



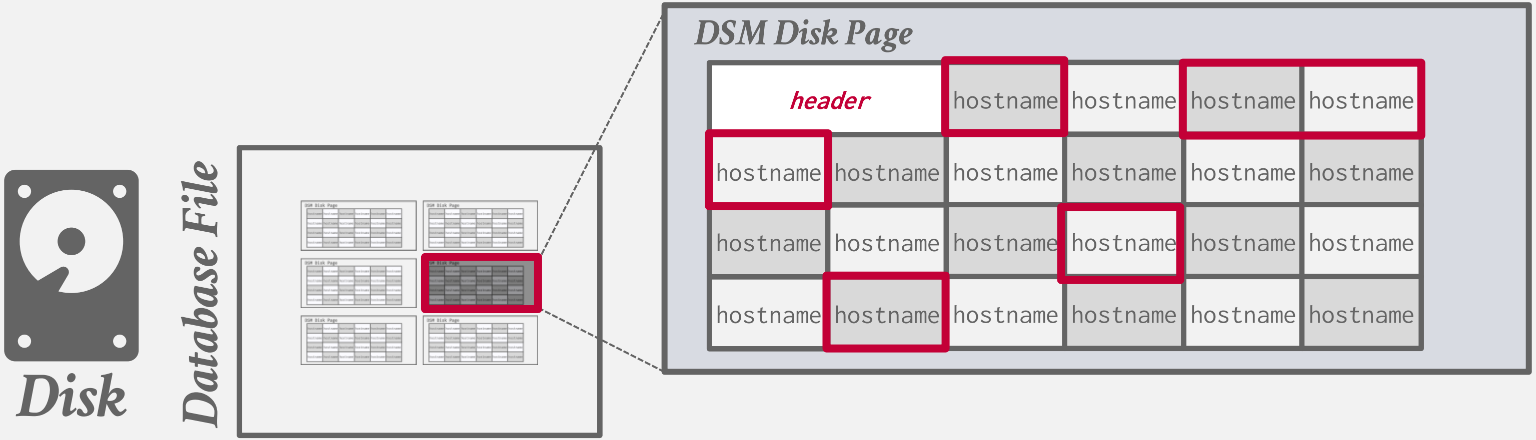
DSM: OLAP EXAMPLE

```
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)
```



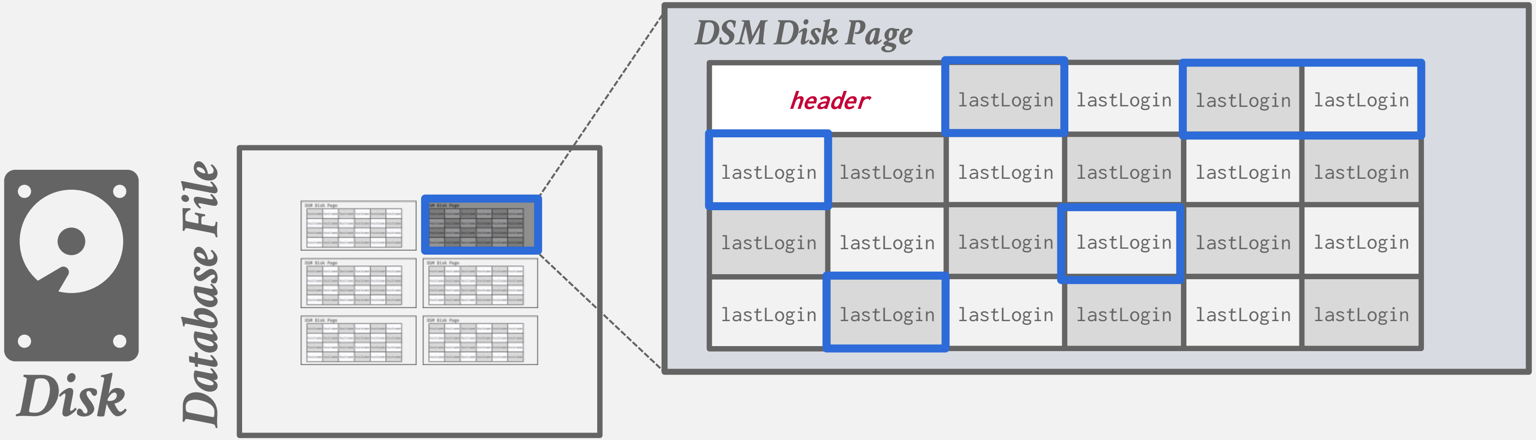
DSM: OLAP EXAMPLE

```
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: 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)
```

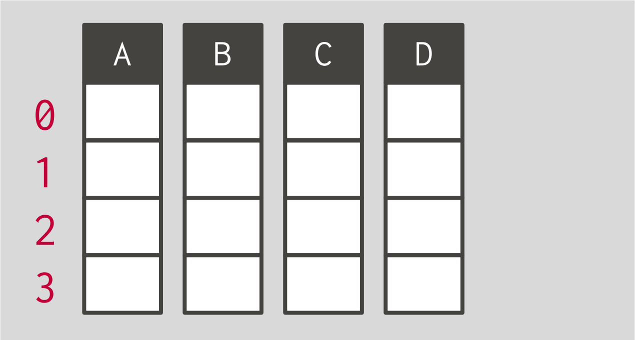


DSM: TUPLE IDENTIFICATION

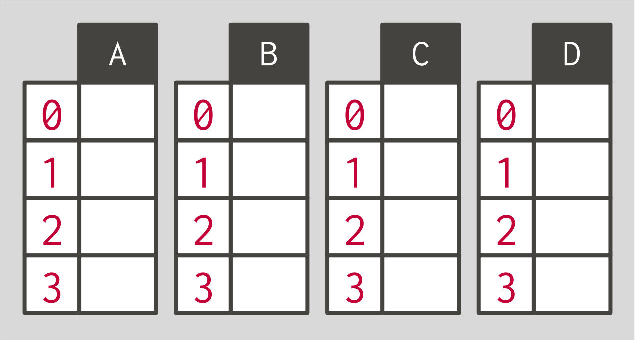
Choice #1: Fixed-length Offsets
→ Each value is the same length for an attribute.

Choice #2: Embedded Tuple Ids
→ Each value is stored with its tuple id in a column.

Offsets



Embedded Ids



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).

→ More on this next week.

DSM: SYSTEM HISTORY

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, Vectorwise, MonetDB

2010s: Everyone + Parquet / ORC

DECOMPOSITION STORAGE MODEL (DSM)

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 (more on this later)

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…

PAX STORAGE MODEL

Partition Attributes Across (PAX) is a hybrid storage model that vertically partitions attributes within a database page.

→ This is what Paraquet and Orc use.

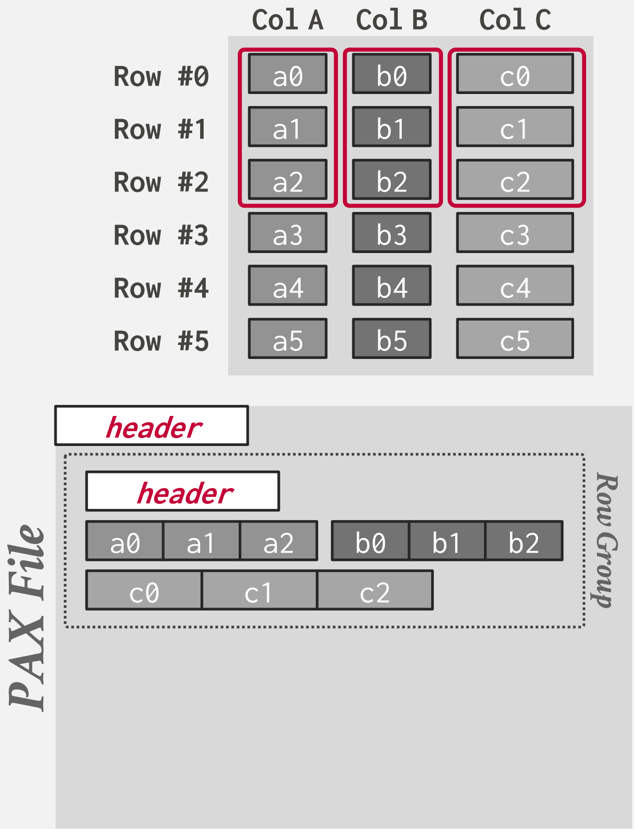
The goal is to get the benefit of <u>faster processing</u> on columnar storage while retaining the <u>spatial locality</u> benefits of row storage.

PAX: PHYSICAL ORGANIZATION

Horizontally partition rows into groups. Then vertically partition their attributes into columns.

Global header contains directory with the 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.

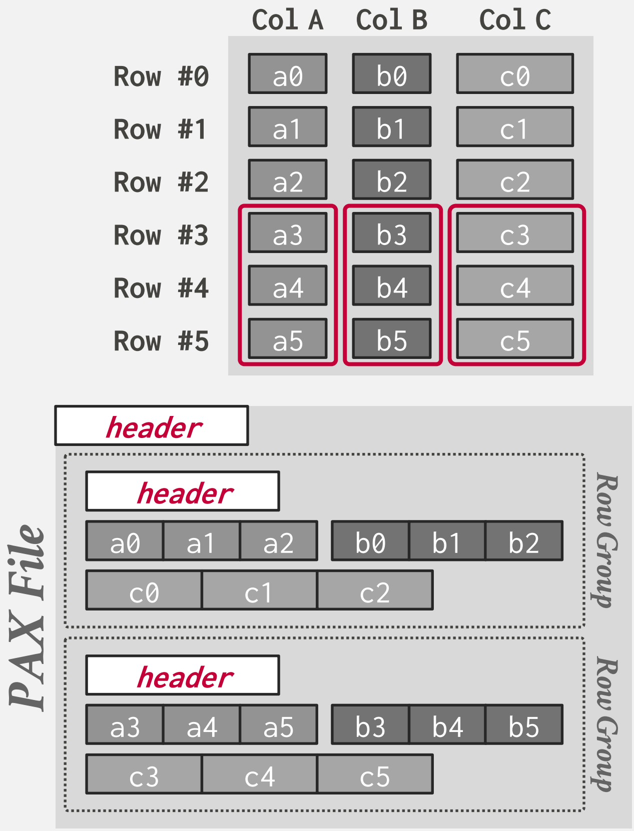


PAX: PHYSICAL ORGANIZATION

Horizontally partition rows into groups. Then vertically partition their attributes into columns.

Global header contains directory with the offsets to the file's row groups.
→ This is stored in the footer if the file is immutable (Parquet, Orc).

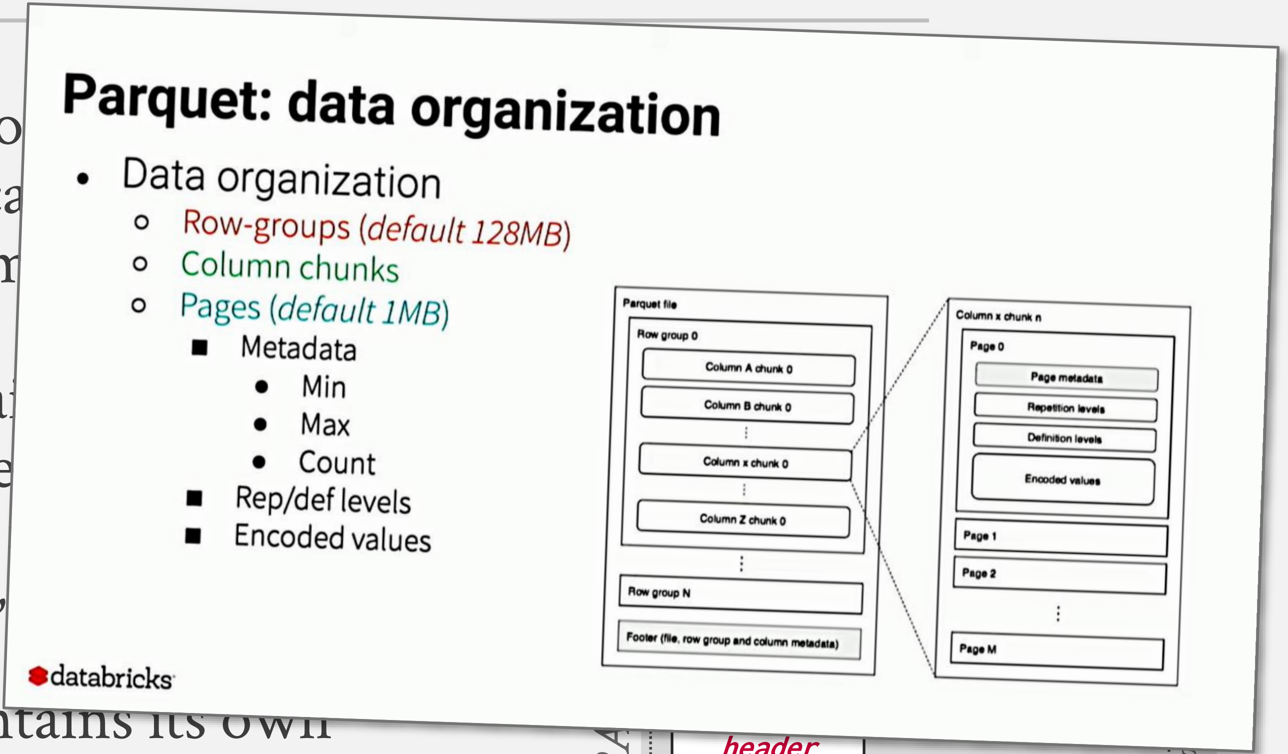
Each row group contains its own meta-data header about its contents.



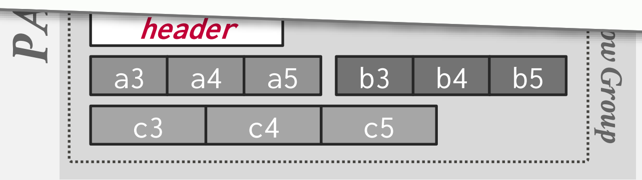
PAX: PHYSICAL ORGANIZATION

Horizontally partitio
groups. Then vertica
attributes into colum

Global header conta
the offsets to the file
→ This is stored in the
immutable (Parquet,



Each row group contains its own
meta-data header about its contents.



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

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.

LOSSLESS VS. LOSSY COMPRESSION

When a DBMS uses compression, it is always **lossless** because people don't like losing data.

Any kind of **lossy** compression must be performed at the application level.

COMPRESSION GRANULARITY

Choice #1: Block-level

→ Compress a block of tuples for the same table.

Choice #2: Tuple-level

→ Compress the contents of the entire tuple (NSM-only).

Choice #3: Attribute-level

→ Compress a single attribute within one tuple (overflow).

→ Can target multiple attributes for the same tuple.

Choice #4: 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.

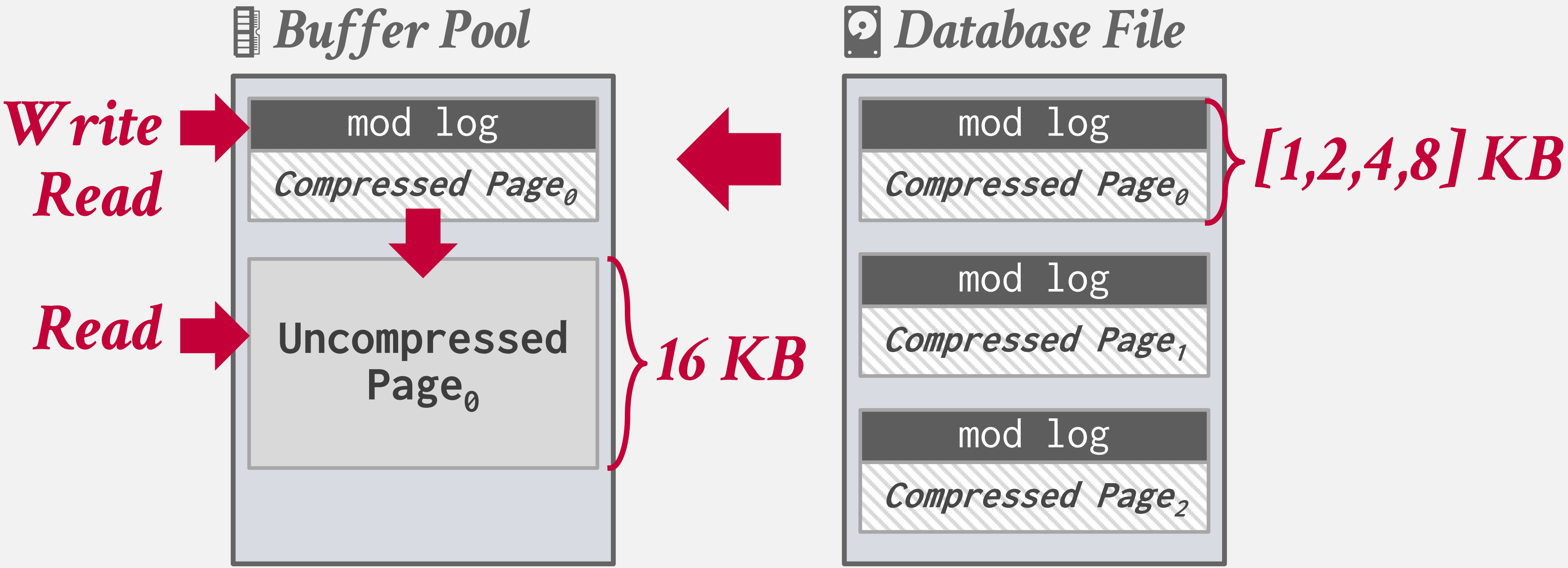
→ LZO (1996), LZ4 (2011), Snappy (2011), Oracle OZIP (2014), Zstd (2015)

Considerations

→ Computational overhead

→ Compress vs. decompress speed.

MYSQL INNODB COMPRESSION



Source: MySQL 5.7 Documentation

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.

Database Magic!

```
SELECT * FROM users
 WHERE name = 'Andy'
```

```
SELECT * FROM users
 WHERE name = XX
```

NAME	SALARY
Andy	99999
DJ 2PL	88888

NAME	SALARY
XX	AA
YY	BB

COMPRESSION GRANULARITY

Choice #1: Block-level

→ Compress a block of tuples for the same table.

Choice #2: Tuple-level

→ Compress the contents of the entire tuple (NSM-only).

Choice #3: Attribute-level

→ Compress a single attribute within one tuple (overflow).
→ Can target multiple attributes for the same tuple.

Choice #4: Column-level

→ Compress multiple values for one or more attributes stored for multiple tuples (DSM-only).

COLUMNAR COMPRESSION

Run-length Encoding

Bit-Packing Encoding

Bitmap Encoding

Delta 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

Original Data

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

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*

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***

RUN-LENGTH ENCODING

Original Data

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

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*

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

If the values for an integer attribute is <u>smaller</u> 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

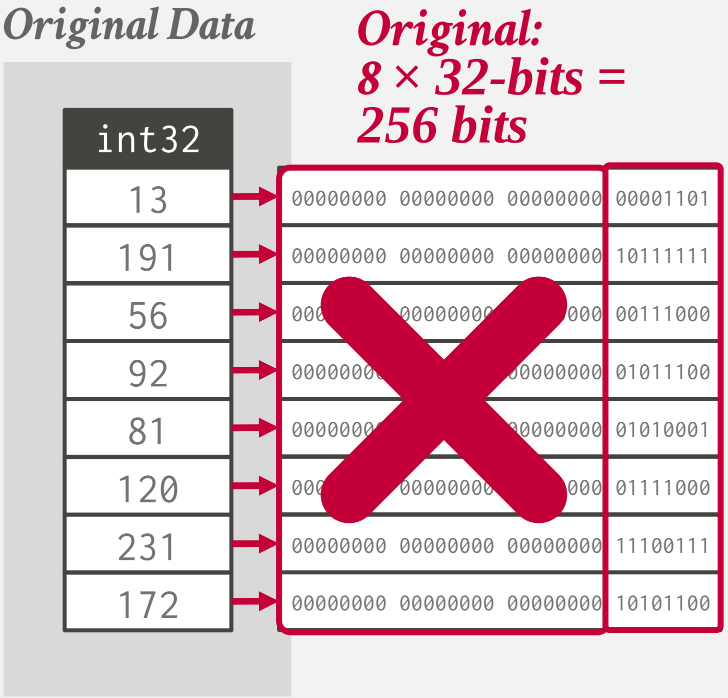
Original:
8 × 32-bits =
256 bits

int32	
13	00000000 00000000 00000000 00001101
191	00000000 00000000 00000000 10111111
56	00000000 00000000 00000000 00111000
92	00000000 00000000 00000000 01011100
81	00000000 00000000 00000000 01010001
120	00000000 00000000 00000000 01111000
231	00000000 00000000 00000000 11100111
172	00000000 00000000 00000000 10101100

BIT PACKING

If the values for an integer attribute is <u>smaller</u> 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.



BIT PACKING

If the values for an integer attribute is <u>smaller</u> 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	00001101
191	10111111
56	00111000
92	01011100
81	01010001
120	01111000
231	11100111
172	10101100

Original:
8 × 32-bits = 256 bits

Compressed:
8 × 8-bits = 64 bits

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.

Original Data

int32
13
191
99999999
92
81
120
231
172

Original:
8 × 32-bits = 256 bits

Compressed Data

mostly8
13
181
XXX
92
81
120
231
172

offset	*value*
3	99999999

Compressed:
(8 × 8-bits) + 16-bits + 32-bits = 112 bits

Source: Redshift Documentation

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

BITMAP ENCODING

Original Data

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

Compressed Data

	isDead	
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 ENCODING

Original Data

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

Original:
9 × 8-bits =
72 bits

Compressed Data

Compressed:
16 bits + 18 bits =
34 bits

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

9 × 2-bits =
18 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.

```
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.

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

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.

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

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.

Original Data

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

5 × 64-bits
= 320 bits

Compressed Data

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

64-bits + (4 × 16-bits)
= 128 bits

Compressed Data

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

64-bits + (2 × 16-bits)
= 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.

DICTIONARY: EXAMPLE

```
SELECT * FROM users
 WHERE name = 'Andy'
```

```
SELECT * FROM users
 WHERE name = 30
```

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth

Compressed Data

name
10
20
30
40
20

value	code
Andrea	10
Prashanth	20
Andy	30
Matt	40

Dictionary

DICTIONARY: ENCODING / DECODING

A dictionary needs to support two operations:

→ **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.

No magic hash function will do this for us.

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
Prashanth
Andy
Matt
Prashanth

Compressed Data

name
10
40
20
30
40

value	*code*
Andrea	10
Andy	20
Matt	30
Prashanth	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
Prashanth
Andy
Matt
Prashanth

➡

Compressed Data

name
10
40
20
30
40

value	*code*
Andrea	10
Andy	20
Matt	30
Prashanth	40

Sorted Dictionary

DICTIONARY: DATA STRUCTURES

Choice #1: Array

→ One array of variable length strings and another array with pointers that maps to string offsets.
→ Expensive to update so only usable in immutable files.

Choice #2: Hash Table

→ Fast and compact.
→ Unable to support range and prefix queries.

Choice #3: B+Tree

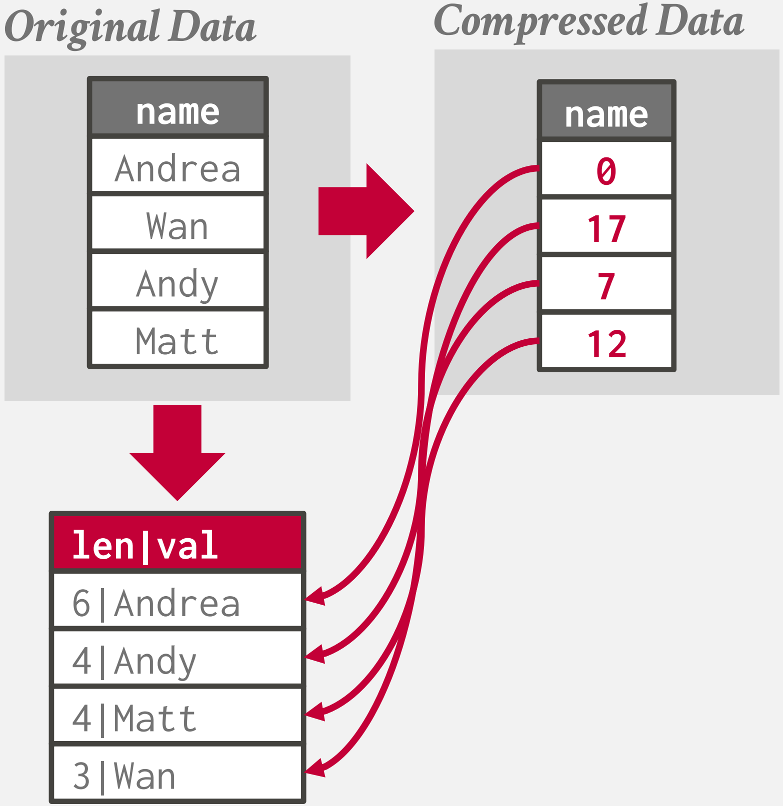
→ Slower than a hash table and takes more memory.
→ Can support range and prefix queries.

DICTIONARY: ARRAY

First sort the values and then store them sequentially in a byte array.

→ Need to also store the size of the value if they are variable-length.

Replace the original data with dictionary codes that are the (byte) offset into this array.



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.

DATABASE STORAGE

Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and moves data back-and-forth from disk.

← Next