

Carnegie Mellon University

# Database Systems

## Storage Models & Data Compression



15-445/645 FALL 2024 » PROF. ANDY PAVLO

# ADMINISTRIVIA

---

**Homework #2** is due Sept 22<sup>nd</sup> @ 11:59pm

**Project #1** is due Sept 29<sup>th</sup> @ 11:59pm

→ Recitation on Wed Sept 18<sup>th</sup> @ 6:00pm

# UPCOMING DATABASE EVENTS

---

## CMU-DB Industry Affiliates Visit Day

- Monday Sept 16<sup>th</sup>: Research Talks + Poster Session
- Tuesday Sept 17<sup>th</sup>: Company Info Sessions
- All events are open to the public.

**Carnegie  
Mellon  
University**  
Database Group  
Industry Affiliates

Sign-up for Company Info Sessions ([@61](#))

Add your Resume if You Want to Make \$\$\$ ([@92](#))

# LAST CLASS

---

We discussed storage architecture alternatives to tuple-oriented scheme.

- Log-structured storage
- Index-organized storage

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

But the most important query for some workloads may be read (**SELECT**) performance...

# TODAY'S AGENDA

---

Database Workloads

Storage Models

Data Compression

DB Flash Talk: **StarTree**

# 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

*Operation Complexity*

Complex

Simple

Write-Heavy

Read-Heavy

*OLTP*

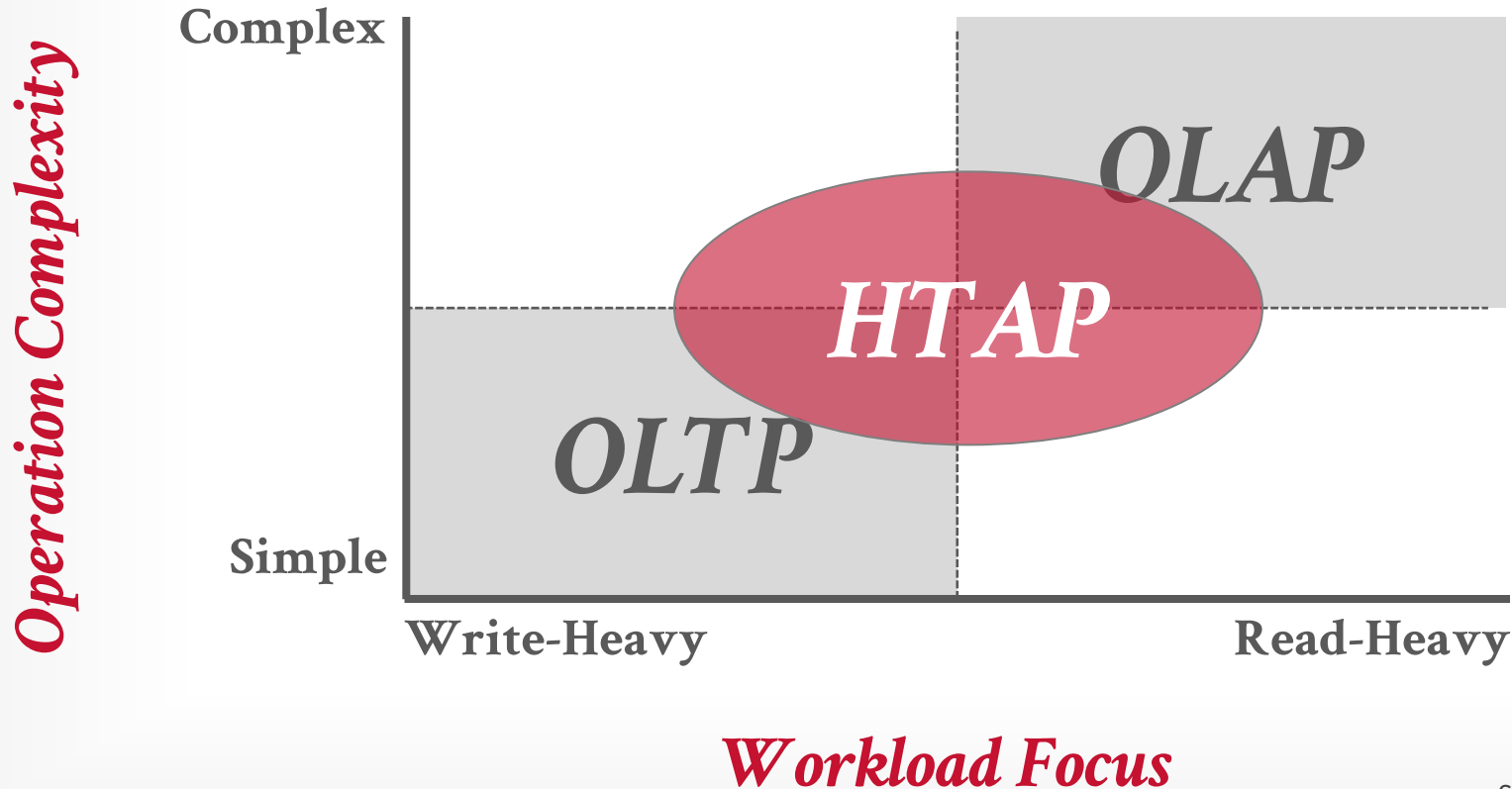
*OLAP*



*Jim Gray*

*Workload Focus*

# DATABASE WORKLOADS





# 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 not specify that the DBMS must store all a tuple's attributes together in a single page.

This may not 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 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.

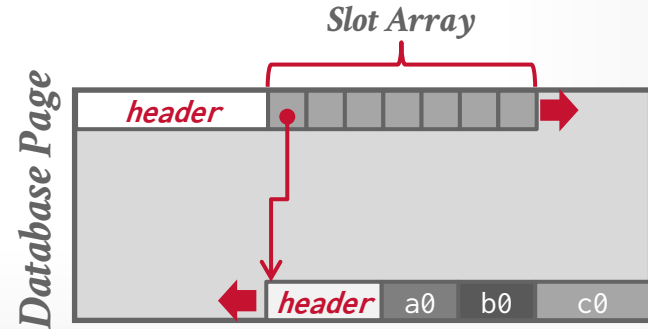
→ [See Lecture #03](#)

# 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.

	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

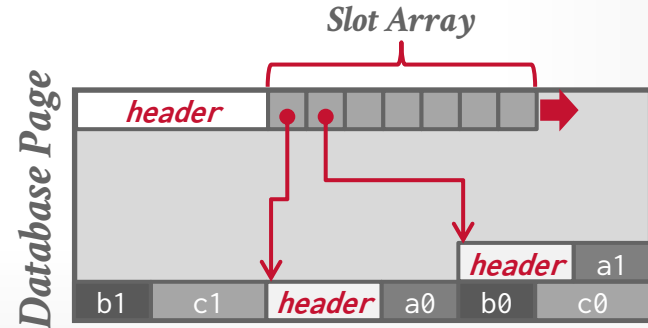


# NSM: PHYSICAL ORGANIZATION

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.

	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



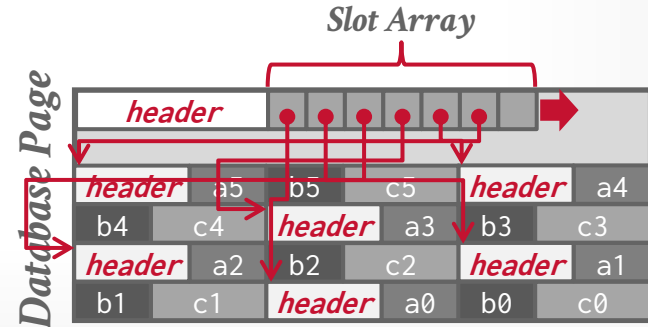


# NSM: PHYSICAL ORGANIZATION

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

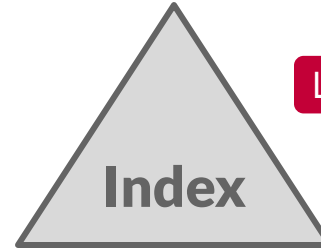
The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

	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 = ?
```

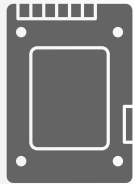


Lectures #8 + #9



*NSM Disk Page*

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	-	-	-	-	-



*Disk*

*Database File*



# NSM: OLTP EXAMPLE

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

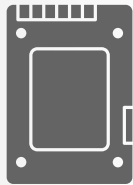
```
INSERT INTO useracct
VALUES (?, ?, ...?)
```

Lectures #8 + #9

Index

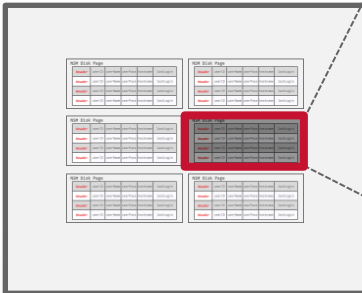
NSM Disk Page

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin



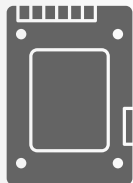
Disk

Database File



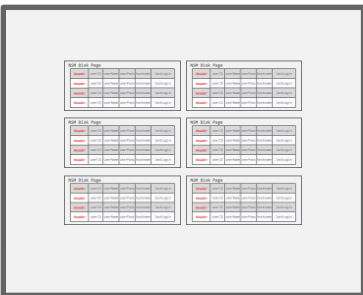
# NSM: OLAP EXAMPLE

```
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) 聚合操作
```



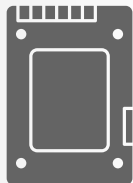
*Disk*

*Database File*



# 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)
```



*Disk*

*Database File*



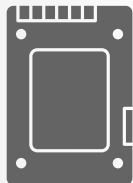
*NSM Disk Page*

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	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)
```

需要第二次扫描表中的 lastLogin 列数据。



Disk

Database File

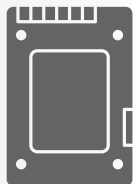


NSM Disk Page

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	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)
```



Disk

Database File



*NSM Disk Page*

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin



*Useless Data!*

行存储会产生很多无用 I/O 操作.

# 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)

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.

## A DECOMPOSITION STORAGE MODEL

George P. Copeland  
Sotraj N. Khoshafian

Microelectronics And Technology Computer Corporation  
8450 Research Blvd  
Austin, Texas 78759

### Abstract

This report examines the relative advantages of a storage model based on decomposition (of community view relations into binary relations containing a surrogate and one attribute) over conventional n-ary storage models.

There seems to be a general consensus among the database community that the n-ary approach is better. This conclusion is usually based on a consideration of only one or two dimensions of a database system. The purpose of this report is not to claim that decomposition is better; instead, we claim that the consensus opinion is not well founded and that neither is clearly better until a closer analysis is made along the many dimensions of a database system. The purpose of this report is to move further in both scope and depth toward such an analysis. We examine such dimensions as simplicity, generality, storage requirements, update performance and retrieval performance.

### 1 INTRODUCTION

Most database systems use an n-ary storage model (DSM) for a set of records. This approach stores data as rows in the conceptual scheme. Also, various inverted file or cluster indexes might be added for improved access speeds. The key concept in the DSM is that all attributes of a conceptual scheme record are stored together. For example, the conceptual scheme relation

```

| name | a1 | a2 | a3 |
|-----|-----|
| a1 | v11 | v21 | v31 |
| a2 | v12 | v22 | v32 |
| a3 | v13 | v23 | v33 |

```

contains a surrogate for record identity and three attributes per record. The DSM would store a1, v11, v21 and v31 together for each record i.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires fee and/or specific permission.

© 1985 ACM 0-89791-160-1/85/005-0268 \$00.75

Some database systems use a fully transposed storage model. For example, DM (Lorie and Spence 1971), TD (Niederhaid et al 1975), MAPT (Turner et al 1978), ADM (Burnett and Thomas 1981), Delta (Shibayama et al 1982) and (Yanaka 1983). This approach stores all values of the same attribute of a conceptual scheme relation together. Several studies have compared the performance of transposed storage models with the DSM (Hoffer 1979, Batory 1979, March and Severance 1977, March and Scudder 1984). In this report, we describe the advantages of a fully decomposed storage model (DSM), which is a transposed storage model with surrogates included. The DSM pairs each attribute value with the surrogate of its conceptual scheme record in a binary relation. For example, the above relation would be stored as

```

| a1sur | val | a2sur | val | a3sur | val |
|-----|-----|-----|-----|
| a1 | v11 | a2 | v21 | a3 | v31 |
| a2 | v12 | a2 | v22 | a3 | v32 |
| a3 | v13 | a3 | v33 | a3 | v33 |

```

In addition, the DSM stores two copies of each attribute relation. One copy is clustered on the value while the other is clustered on the surrogate. These statements apply only to base (i.e., extensional) data. To support the relational model, intermediate and final results need an n-ary representation. If a richer data model than normalized relations is supported, then intermediate and final results need a correspondingly richer representation.

This report compares these two storage models based on several criteria. Section 2 compares the relative simplicity and generality of the two storage models. Section 3 compares their storage requirements. Section 4 compares their update performance. Section 5 compares their retrieval performance. Finally, Section 6 provides a summary and suggests some refinements for the DSM.

### 2 SIMPLICITY AND GENERALITY

This section compares the two storage models to illustrate their relative simplicity and generality. Others (Abram 1976, Pelissier and Komaki 1977, Komaki 1978, Gadd 1979) have argued for the semantic clarity and generality of representing each basic fact individually within the conceptual scheme as the DSM does within the storage scheme.

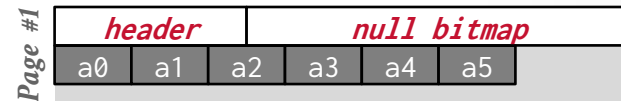
# DSM: PHYSICAL ORGANIZATION

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...

	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

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



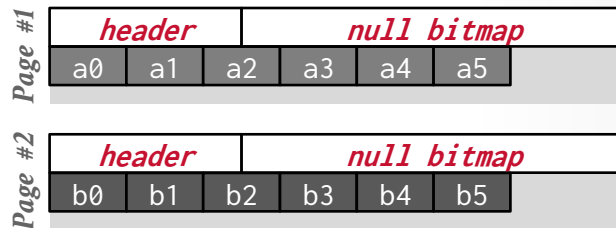
# DSM: PHYSICAL ORGANIZATION

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...

	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

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



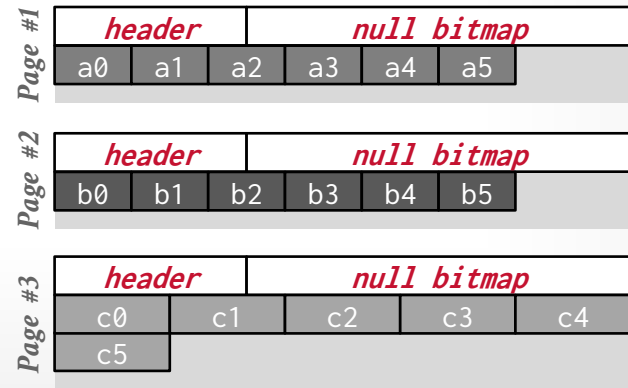
# DSM: PHYSICAL ORGANIZATION

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...

	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

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



# DSM: OLAP EXAMPLE

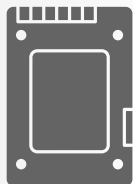
---

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin

# DSM: OLAP EXAMPLE

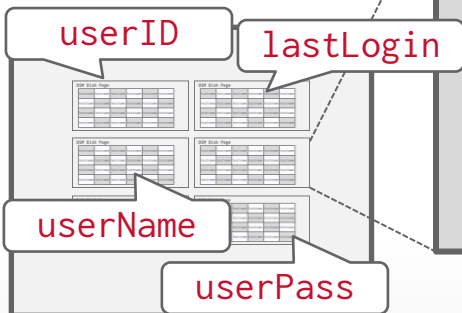
```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
FROM useracct AS U
WHERE U.hostname LIKE '%.gov'
GROUP BY EXTRACT(month FROM U.lastLogin)
```

按列拆分，将它们存储在单独的页面。



Disk

Database File

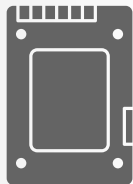


DSM Disk Page

<i>header</i>		hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname	hostname

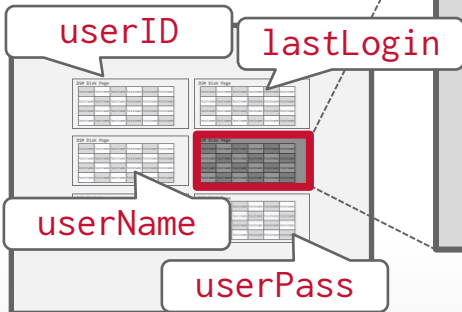
# DSM: OLAP EXAMPLE

```
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)
```



*Disk*

*Database File*

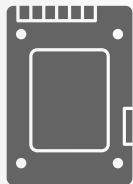


*DSM Disk Page*

<i>header</i>	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname
hostname	hostname	hostname	hostname	hostname

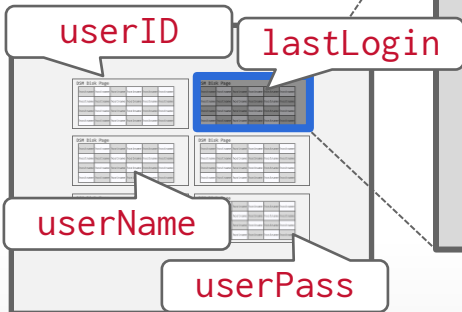
# DSM: OLAP EXAMPLE

```
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)
```



Disk

Database File



*DSM Disk Page*

<i>header</i>	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	lastLogin
lastLogin	lastLogin	lastLogin	lastLogin	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.

*Don't  
Do This!*

### Offsets

	A	B	C	D
0				
1				
2				
3				

### Embedded Ids

	A	B	C	D
0		0		0
1		1		1
2		2		2
3		3		3

# 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). 映射到字符串的字典编码.  
→ More on this later in this lecture...

# 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 (Lecture #13). CPU 顺序执行指令, 避免分支和跳转.
- 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...

列存储格式，但同时尽量让每个元组的数据物理上连续存储。

# 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.

## Weaving Relations for Cache Performance

Anastassia Ailamaki<sup>‡</sup>  
Camerge Mellon University  
anastax@cs.cmu.edu

David J. DeWitt  
Univ. of Wisconsin-Madison  
dewitt@cs.wisc.edu

Mark D. Hill  
Univ. of Wisconsin-Madison  
markhill@cs.wisc.edu

Marios Skounakis  
Univ. of Wisconsin-Madison  
marios@cs.wisc.edu

### Abstract

Relational database systems have traditionally optimized for IO performance and organized records sequentially on disk pages using the *N-ary Storage Model* (NSM) (i.e., slotted pages). Recent research, however, indicates that cache utilization and performance is becoming increasingly important on modern platforms. In this paper, we first demonstrate that in-page data placement is the key to high cache performance and that NSM exhibits low cache utilization on modern platforms. Next, we propose a new data organization model called PAX (Partition Attributes Across), that significantly improves cache performance by grouping together all values of each attribute within each page. Because PAX only differs inside the pages, it incurs no storage penalty and does not affect IO behavior. According to our experimental results, when compared to NSM (a) PAX exhibits superior cache and memory bandwidth utilization, saving at least 75% of NSM's stall time due to data cache accesses, (b) range selection queries and updates on memory-resident relations execute 17-25% faster, and (c) TPC-H queries involving IO execute 11-48% faster.

### 1 Introduction

The communication between the CPU and the secondary storage (IO) has been traditionally recognized as the major database performance bottleneck. To optimize data transfer to and from mass storage, relational DBMSs have long organized records in slotted disk pages using the *N-ary Storage Model* (NSM). NSM stores records contiguously starting from the beginning of each disk page, and uses an offset (slot) table at the end of the page to locate the beginning of each record [27].

Unfortunately, most queries use only a fraction of each record. To minimize unnecessary IO, the Decomposition Storage Model (DSM) was proposed in 1985 [10]. DSM partitions an *n*-attribute relation vertically into *n* sub-relations, each of which is accessed only when the corresponding attribute is needed. Queries that involve multiple attributes from a relation, however, must spend

tremendous additional time to join the participating sub-relations together. Except for Sybase-IQ [33], today's relational DBMSs use NSM for general-purpose data placement [20][29][32].

Recent research has demonstrated that modern database workloads, such as decision support systems and spatial applications, are often bound by delays related to the processor and the memory subsystem rather than IO [20][5][26]. When running commercial database systems on a modern processor, data requests that miss in any of the caches and are transferred from main memory) are a key memory bottleneck [1]. In addition, only a fraction of the data transferred to the cache is useful to the query: the item that the query processing algorithm requests and the transfer unit between the memory and the processor are typically not the same size. Loading the cache with useless data (a) wastes bandwidth, (b) pollutes the cache, and (c) possibly forces replacement of information that may be needed in the future, incurring even more delays. The challenge is to repair NSM's cache behavior without compromising its advantages over DSM.

This paper introduces and evaluates **Partition Attributes Across (PAX)**, a new layout for data records that combines the best of the two worlds and exhibits performance superior to both placement schemes by eliminating unnecessary accesses to main memory. For a given relation, PAX stores the same data on each page as NSM. Within each page, however, PAX groups all the values of a particular attribute together on a minipage. During a sequential scan (e.g., to apply a predicate on a fraction of the records), PAX fully utilizes the cache resources, because on each miss a number of a single attribute's values are loaded into the cache together. At the same time, all parts of the record are on the same page. To reconstruct a record one needs to perform a *mini-join* among minipages, which incurs minimal cost because it does not have to look beyond the page.

We evaluated PAX against NSM and DSM using (a) predicate selection queries on numeric data and (b) a variety of queries on TPC-H datasets on top of the Shore storage manager [7]. We vary query parameters including selectivity, projectivity, number of predicates, distance between the projected attribute and the attribute in the predicate, and degree of the relation. The experimental results show that, when compared to NSM, PAX (a) incurs 50-75% fewer second-level cache misses due to data

<sup>‡</sup> Work done while author was at the University of Wisconsin-Madison. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment. *Proceedings of the 27th VLDB Conference, Roma, Italy, 2001*

# PAX: PHYSICAL ORGANIZATION

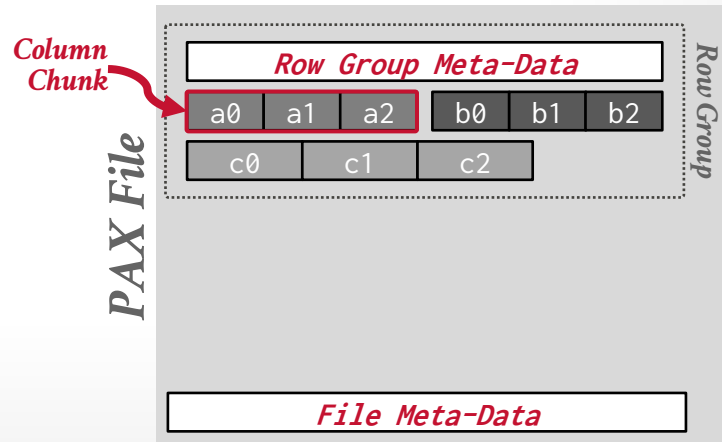
Horizontally partition data into *row groups*. Then vertically partition their attributes into *column chunks*.

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.

	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



# PAX: PHYSICAL ORGANIZATION

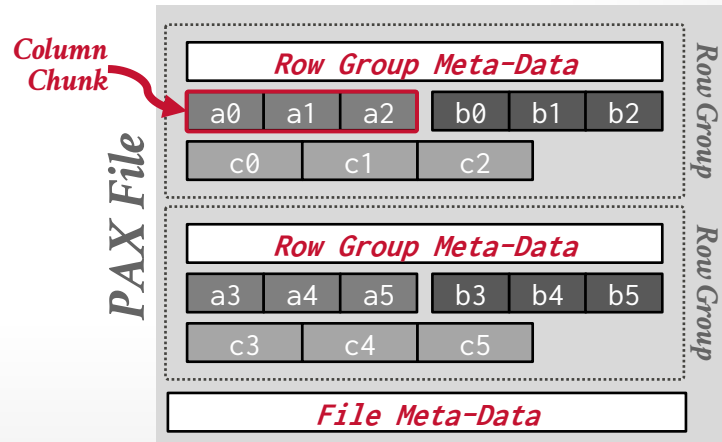
Horizontally partition data into *row groups*. Then vertically partition their attributes into *column chunks*.

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.

	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



# PAX: PHYSICAL ORGANIZATION

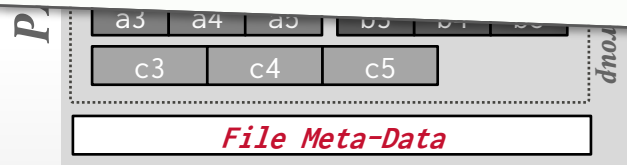
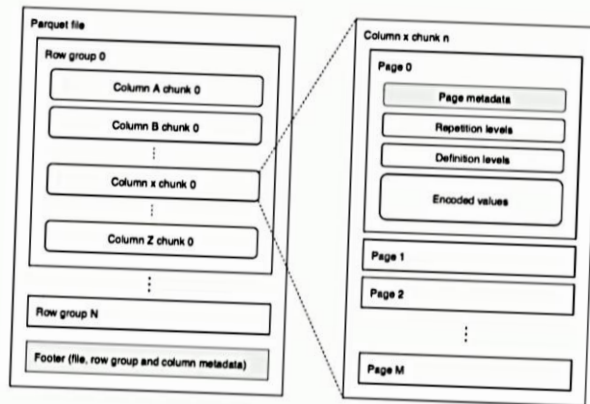
Horizontally partitioned into **row groups**. Then vertically partitioned into **column chunks**.

Global meta-data dictionary (offsets to the file's row groups) → This is stored in the footer (immutable (Parquet, ...)

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

## Parquet: data organization

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





# 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.

→ People (typically) don't like losing data.

→ Any lossy compression must be performed by application.

# 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

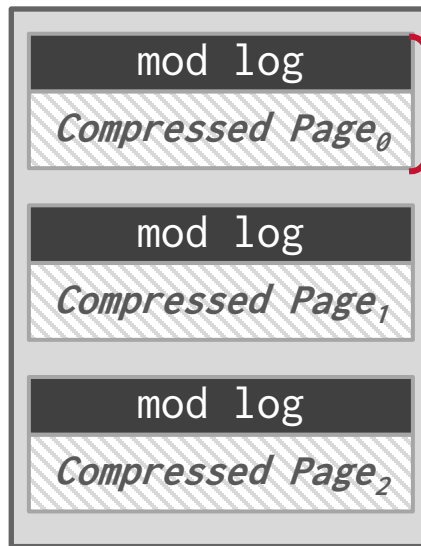
- Computational overhead
- Compress vs. decompress speed.

# MYSQL INNODB COMPRESSION

 *Buffer Pool*

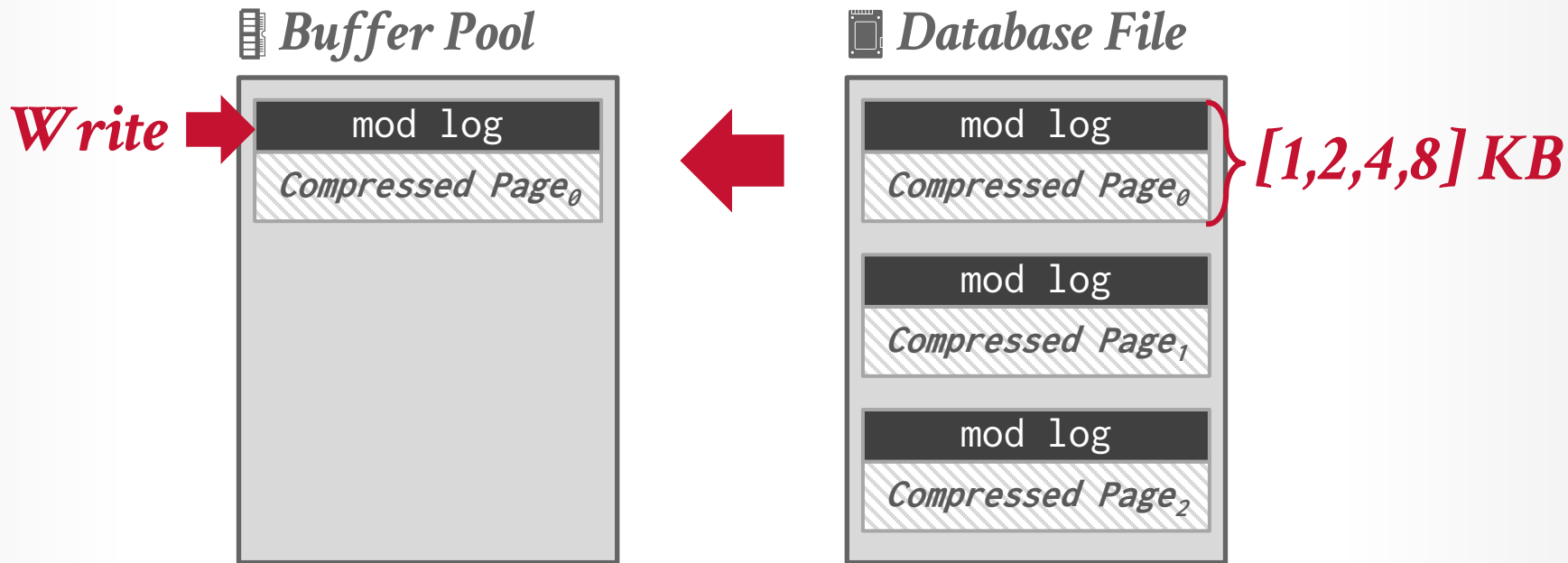


 *Database File*

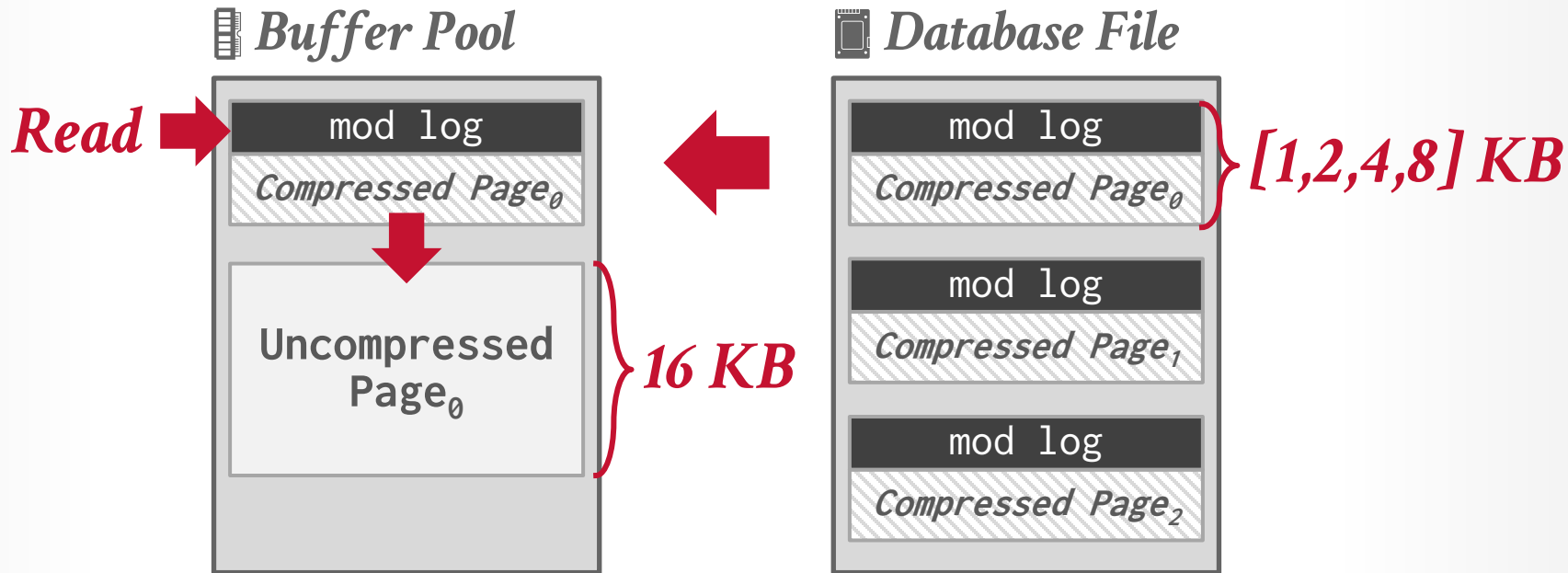


**[1,2,4,8] KB**

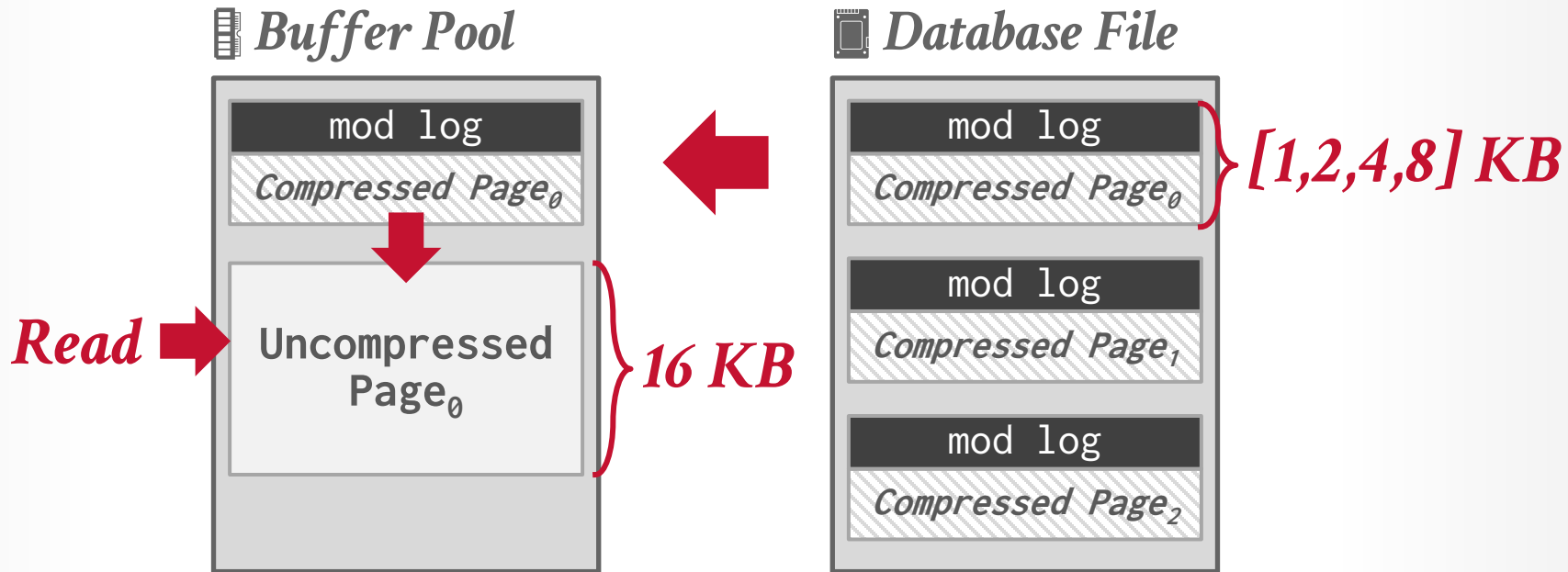
# MYSQL INNODB COMPRESSION



# MYSQL INNODB COMPRESSION



# 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.

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

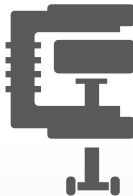
*Database Magic!*



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

直接在压缩数据上进行操作。

NAME	SALARY
Andy	99999
Jignesh	88888



NAME	SALARY
<b>XX</b>	<b>AA</b>
<b>YY</b>	<b>BB</b>

# 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 / Frame-of-Reference Encoding

Incremental Encoding

Dictionary Encoding

# RUN-LENGTH ENCODING

---

Compress runs of the same value in a single column into triplets: 三元组: (value, offset, #)

- 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
```

根据列 isDead 做聚合运算.



## *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	<i>RLE Triplet</i> - Value - Offset - Length
8	
9	

# RUN-LENGTH ENCODING

*Original Data*

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

数据交叉存储.



*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,6,2)
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*

*Original:  
 $8 \times 32\text{-bits} =$   
 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 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*

*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

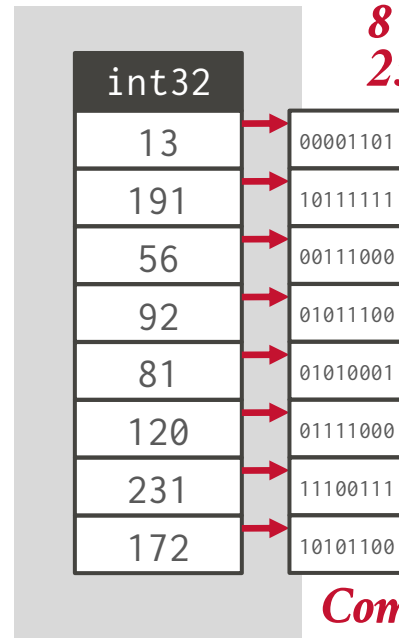


# BIT PACKING

If the values for an integer attribute is 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*



*Original:*  
 $8 \times 32\text{-bits} = 256 \text{ bits}$

仅存储有效位。

*Compressed:*  
 $8 \times 8\text{-bits} = 64 \text{ bits}$

# 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.

*Original Data*

int32
13
191
99999999
92
81
120
231
172

**Original:**  
 $8 \times 32\text{-bits} =$   
 $256\text{ bits}$



*Compressed Data*

mostly8	offset	value
13	3	99999999
181		
xxx		
92		
81		
120		
231		
172		

**Compressed:**  
 $(8 \times 8\text{-bits}) +$   
 $16\text{-bits} + 32\text{-bits}$   
 $= 112\text{ bits}$

# 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



*Compressed Data*

id	isDead	
	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:  
8 × 8-bits =  
64 bits*

*Compressed Data*

id	isDead	
	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

*Compressed:  
16 bits + 16 bits =  
32 bits*

*2 × 8-bits =  
16 bits*

*8 × 2-bits =  
16 bits*



# BITMAP ENCODING: EXAMPLE

Assume we have 10 million tuples.

43,000 zip codes in the US.

→  $10000000 \times 32\text{-bits} = 40 \text{ MB}$

→  $10000000 \times 43000 = 53.75 \text{ GB}$

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  
);
```

There are compressed data structures for sparse data sets:

→ Roaring Bitmaps

在稀疏数据集上有较好的压缩效果.

# 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

# 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

# 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.

Frame-of-Reference Variant: Use global min value.

*Original Data*

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

$5 \times 64\text{-bits}$   
= 320 bits

*Compressed Data*

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

$64\text{-bits} + (4 \times 16\text{-bits})$   
= 128 bits

*Compressed Data*

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

$64\text{-bits} + (2 \times 16\text{-bits})$   
= 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.

- **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.

*Original Data*

name
Andrea
Mr.Pickles
Andy
Jignesh
Mr.Pickles



*Compressed Data*

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

*Sorted  
Dictionary*



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

# 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