香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF DATA SCIENCE
數據科學學院

Ack: Prof. Jignesh Patel @ CMU
Prof. Andy Pavlo @CMU

CSC3170
5: Storage Model

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

Database Workloads

- **On-Line Transaction Processing (OLTP)**

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

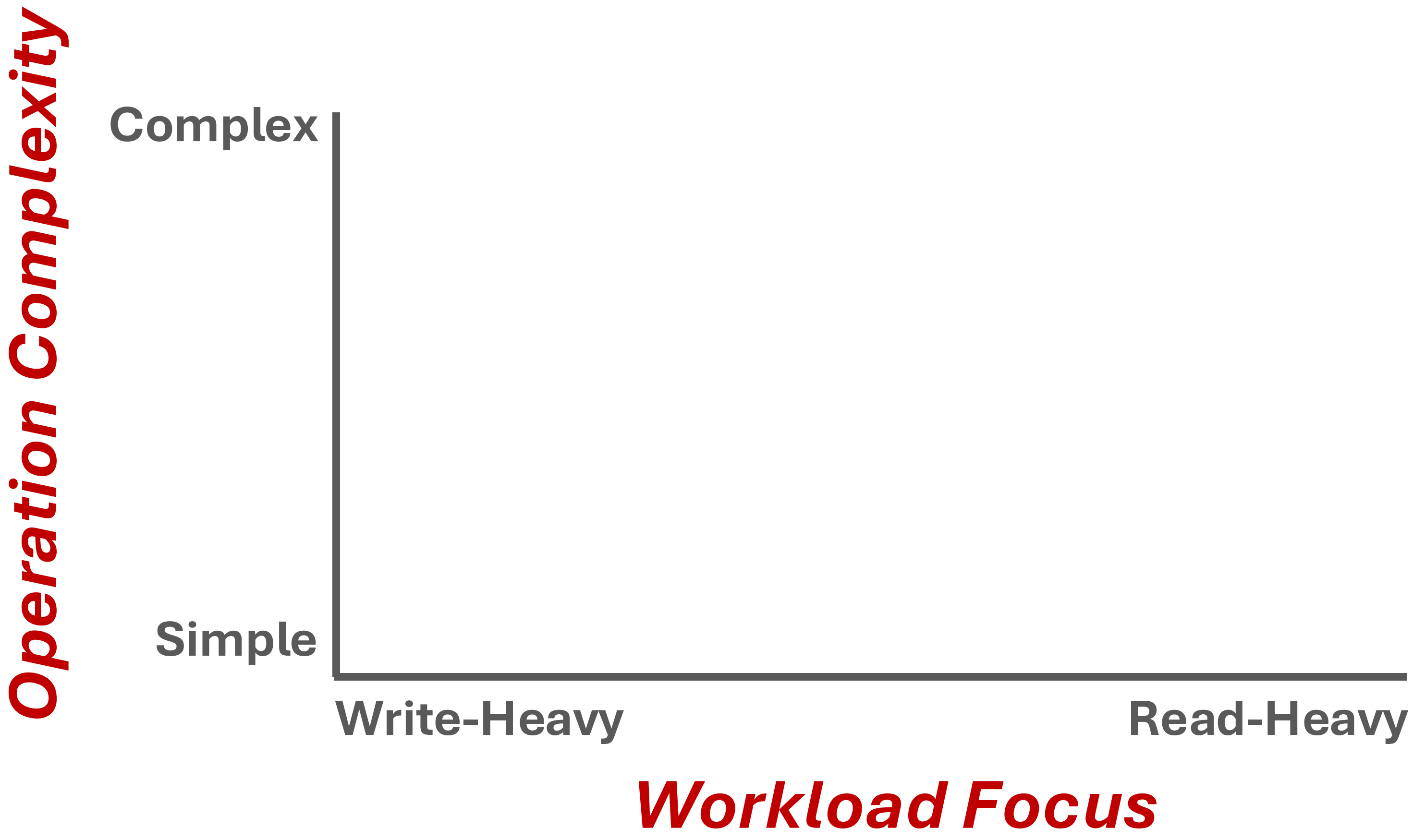
- **On-Line Analytical Processing (OLAP)**

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

- **Hybrid Transaction + Analytical Processing**

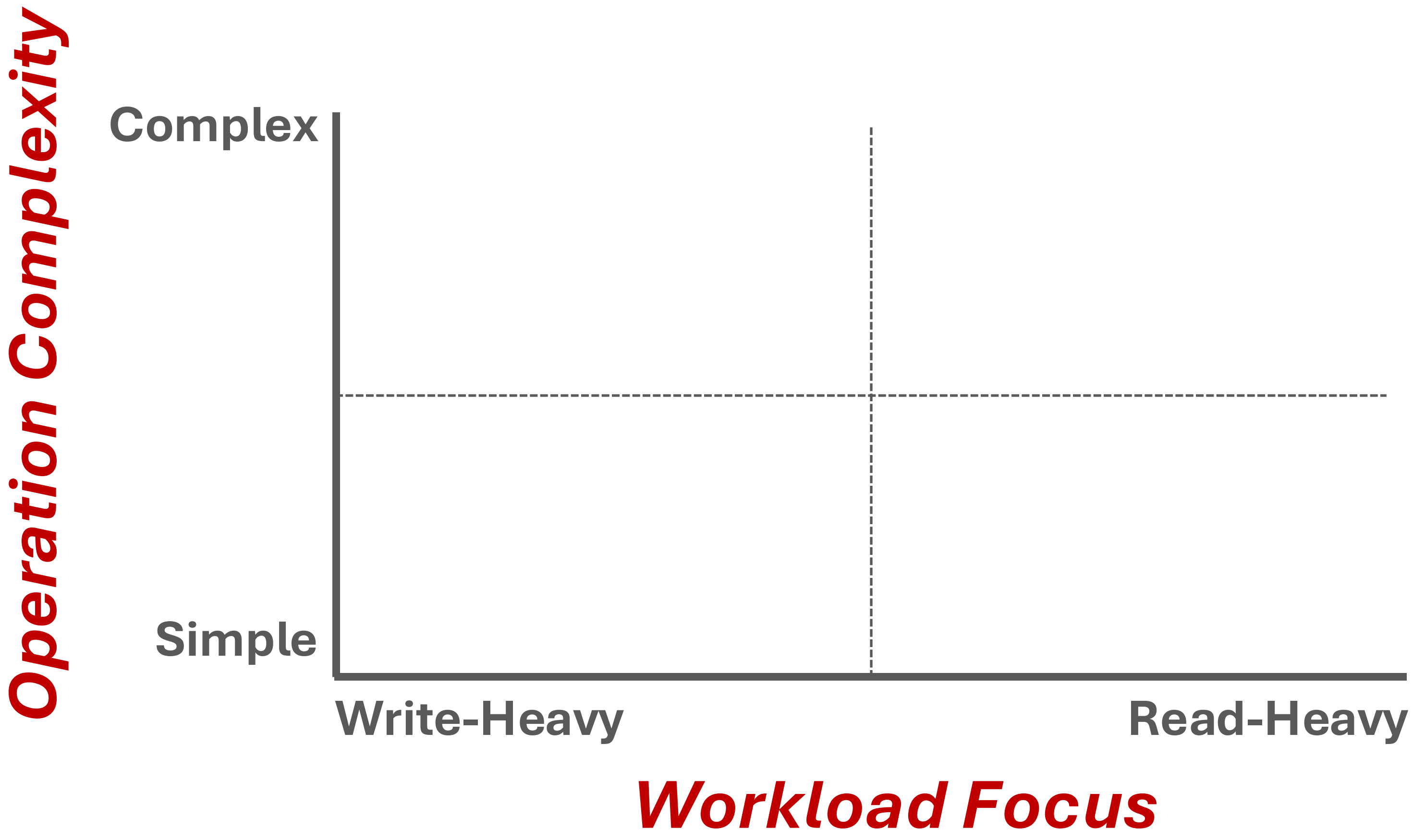
→ OLTP + OLAP together on the same database instance

Database Workloads



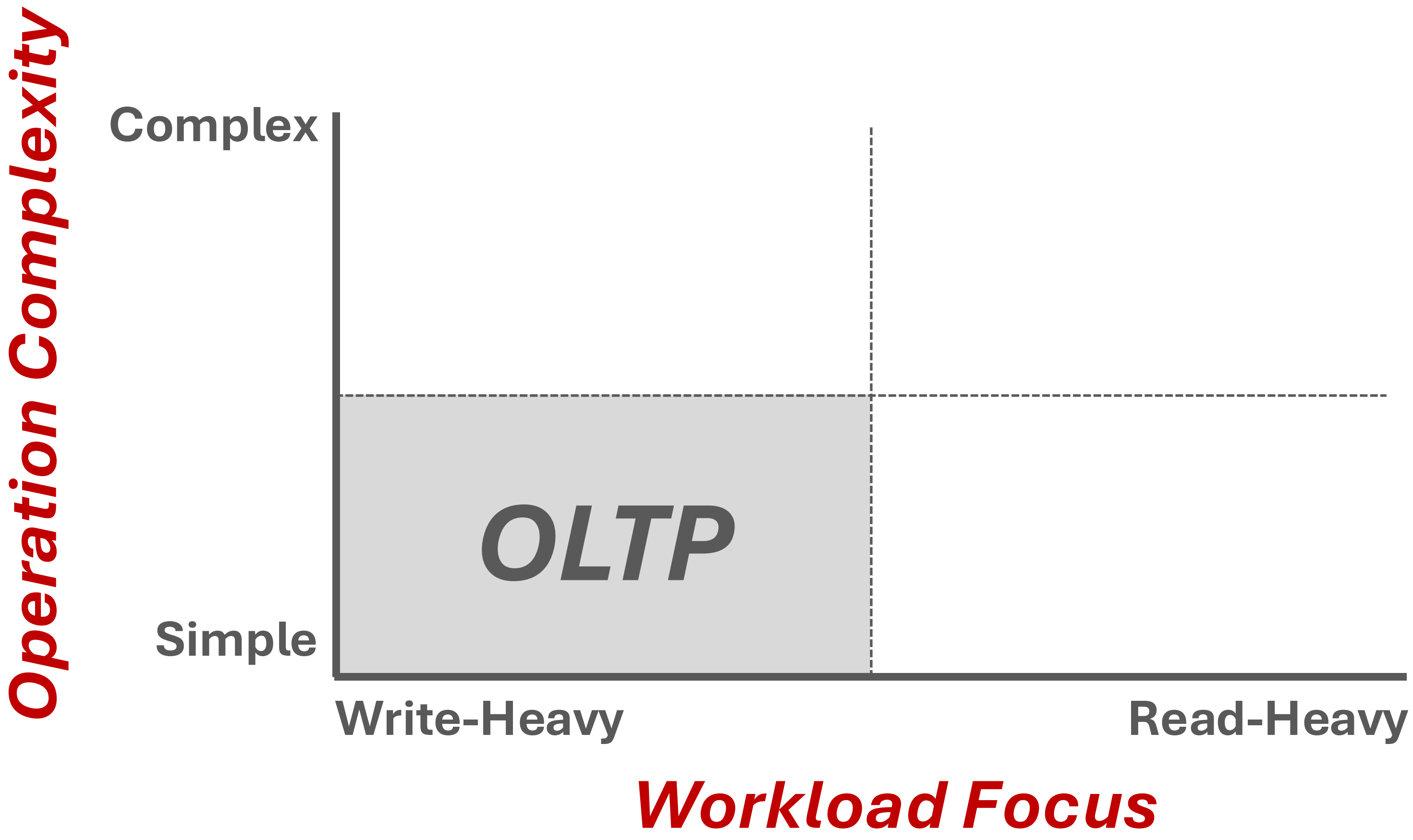
Source: Mike Stonebraker

Database Workloads



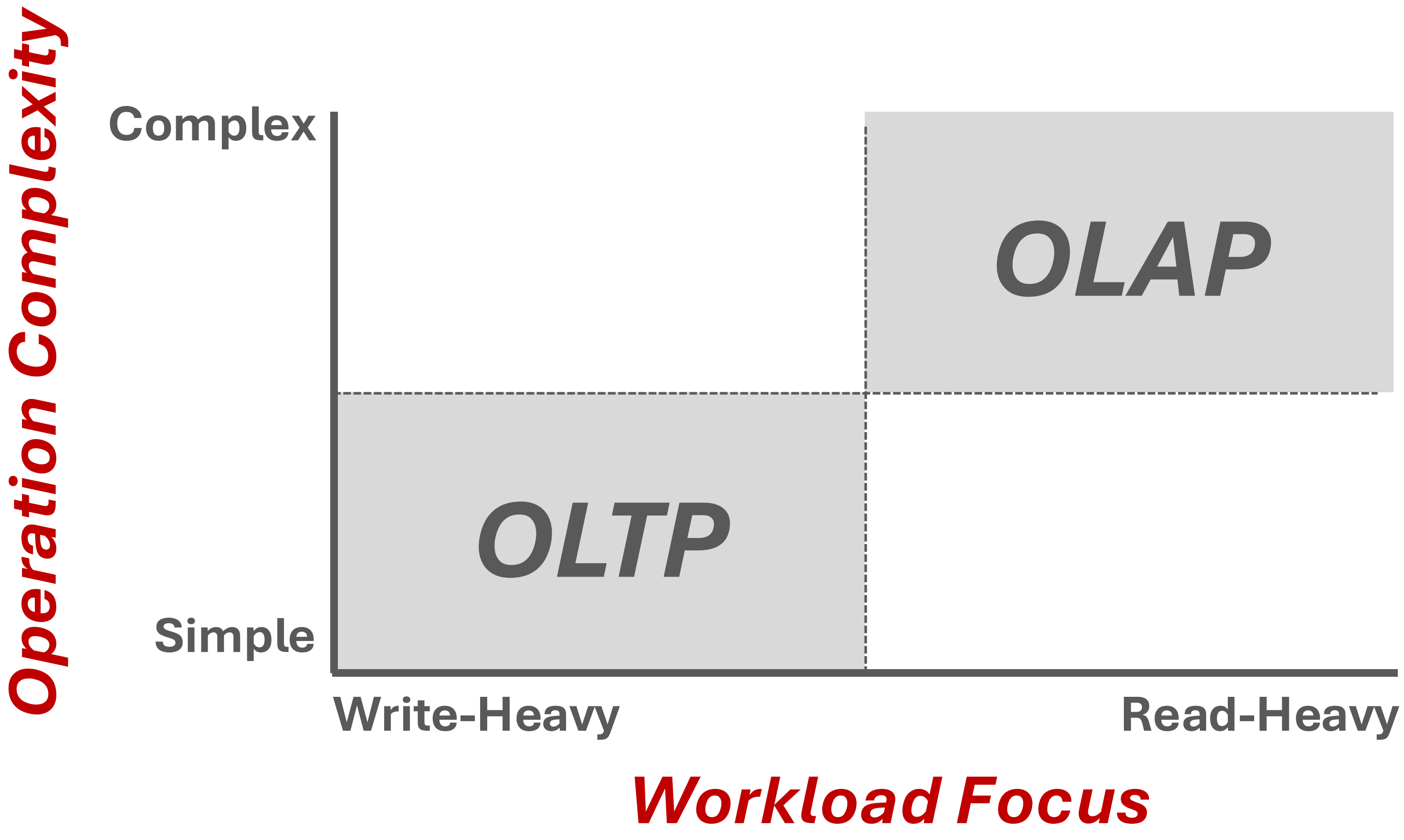
Source: [Mike Stonebraker]

Database Workloads



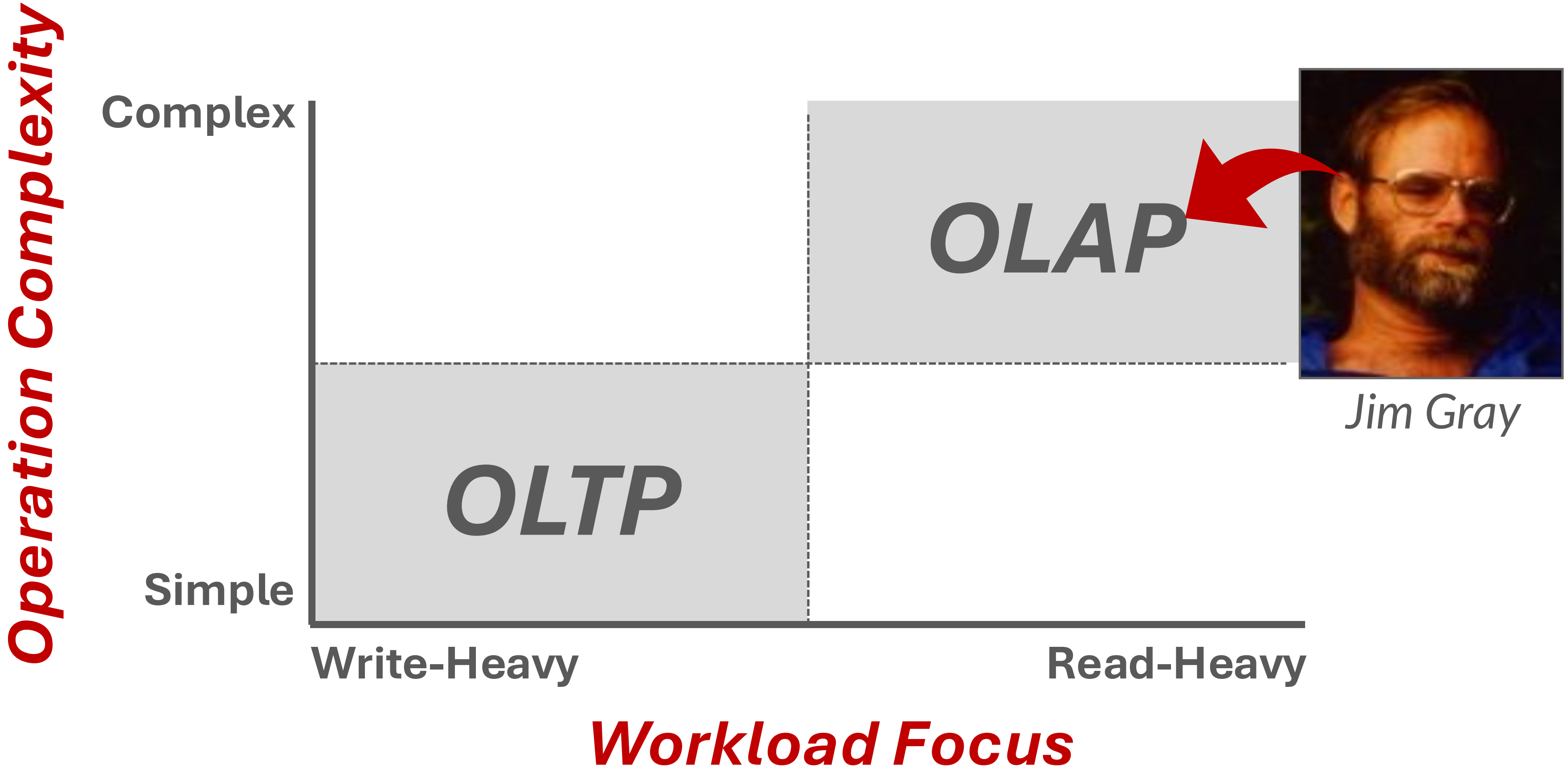
Source: [Mike Stonebraker]

Database Workloads



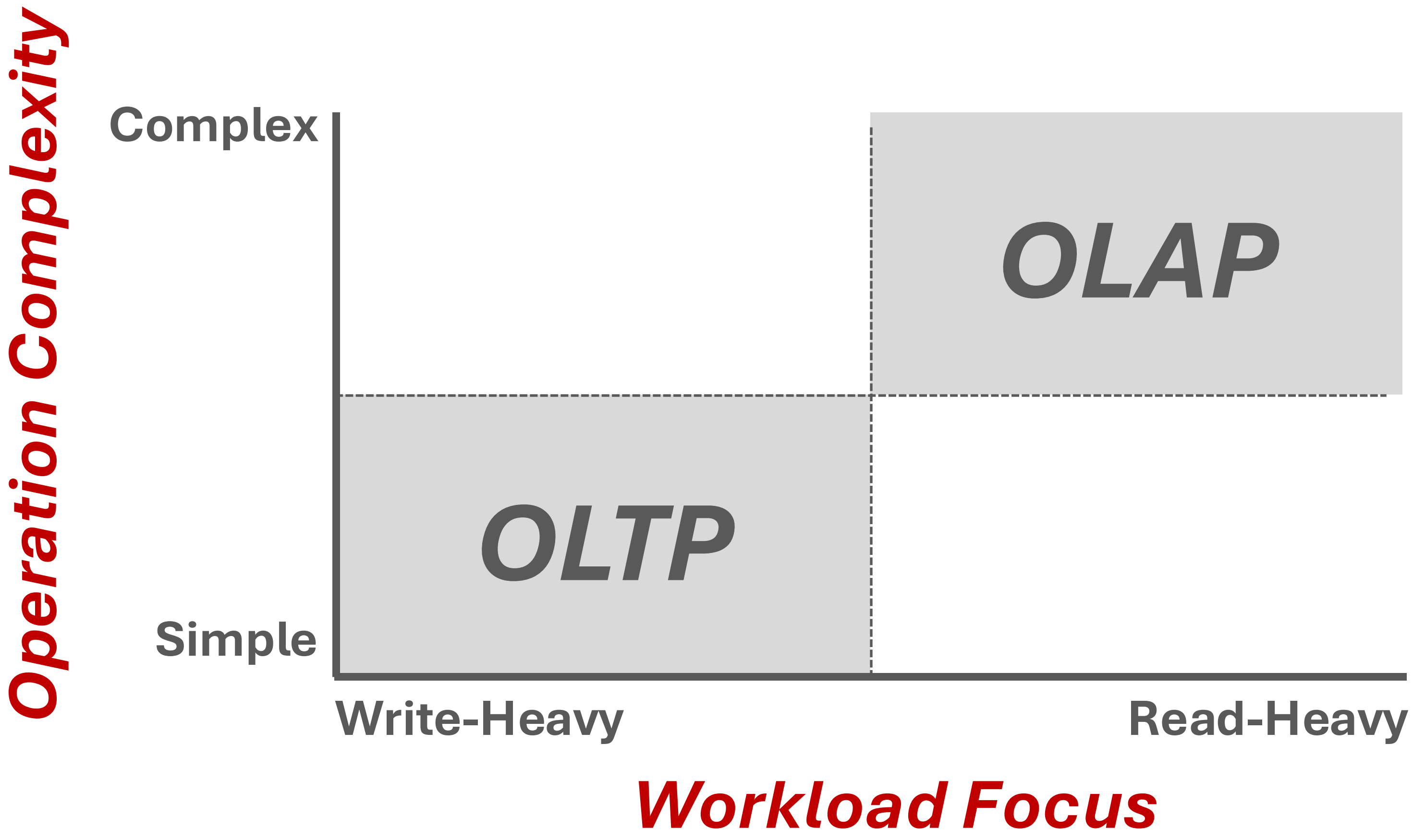
Source: [Mike Stonebraker]

Database Workloads



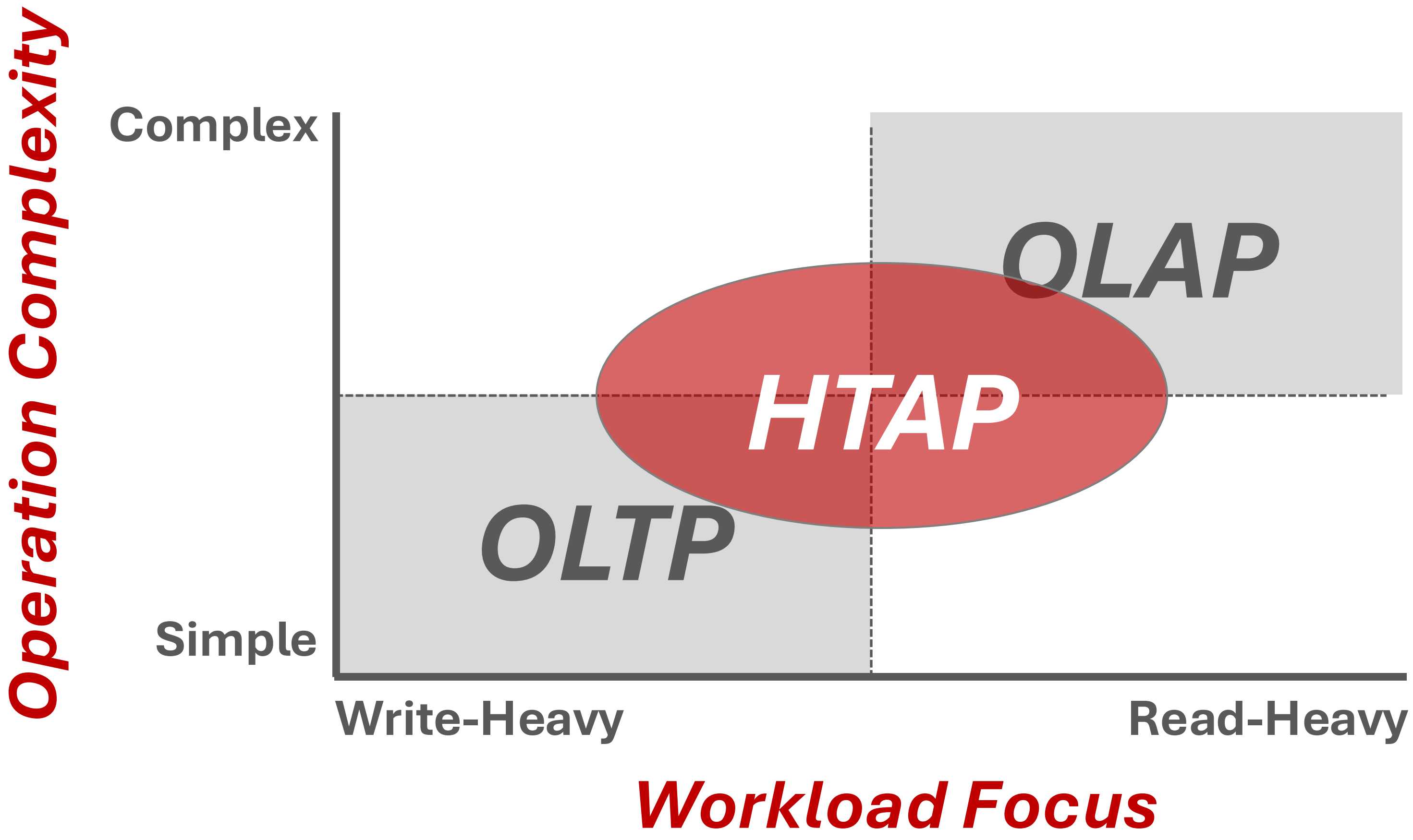
Source: [Mike Stonebraker]

Database Workloads



Source: [Mike Stonebraker]

Database Workloads



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
);
```

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
);
```

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 of a tuple's attributes together on 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 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 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)

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.

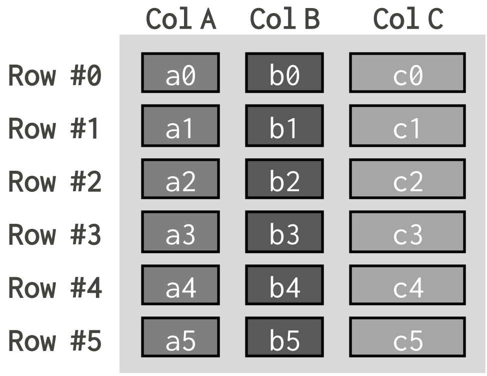
- NSM database page sizes are typically some constant multiple of 4 KB hardware pages.

→ 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.

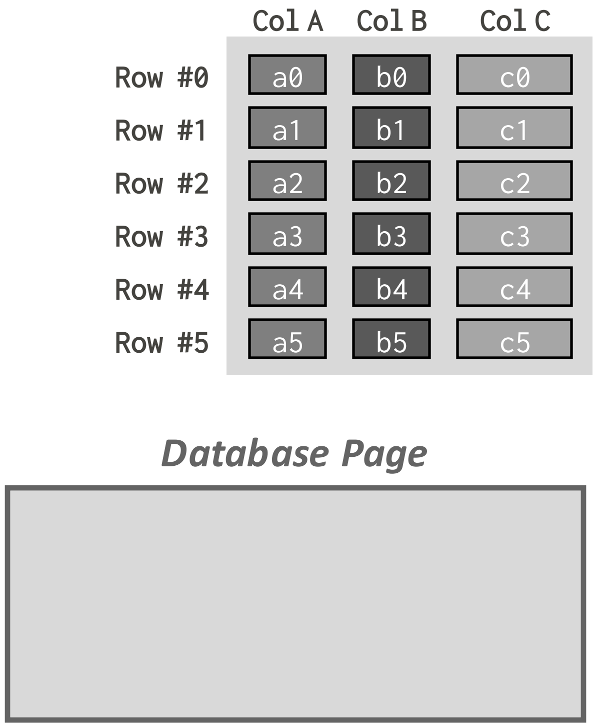


SCHOOL OF DATA SCIENCE

NSM: Physical Organization

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

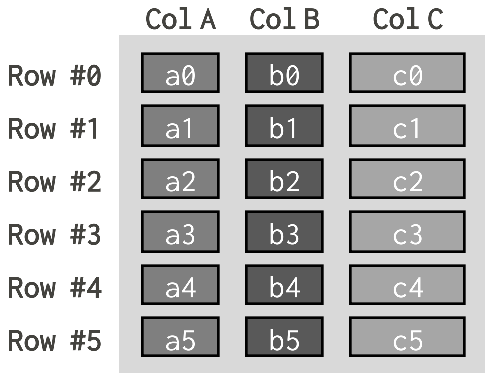
- The tuple’s **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.



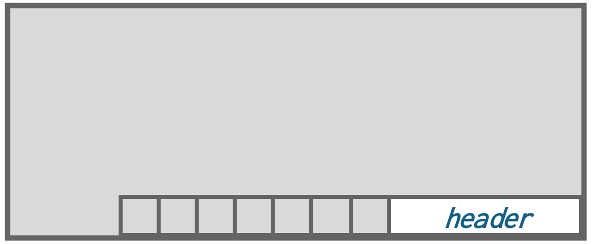
NSM: Physical Organization

- 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.

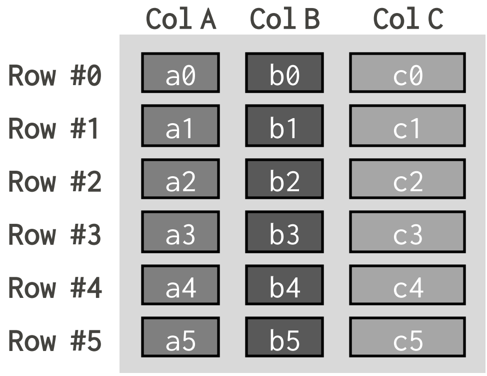


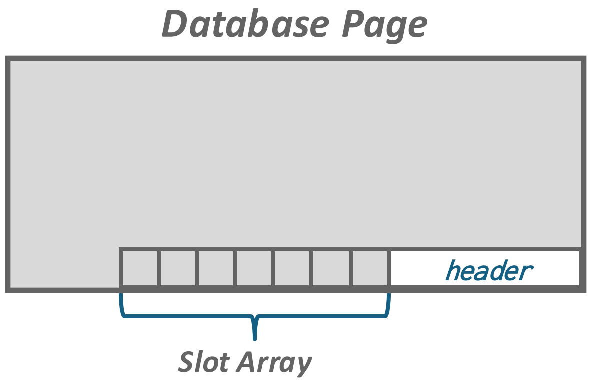
Database Page



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.

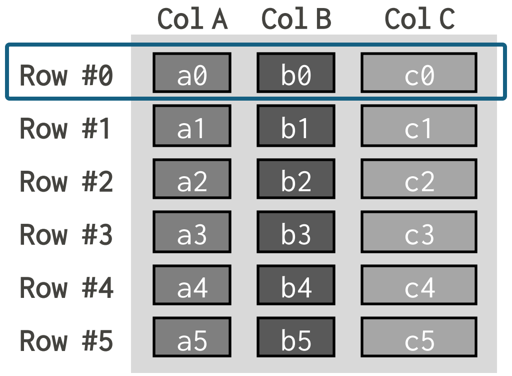


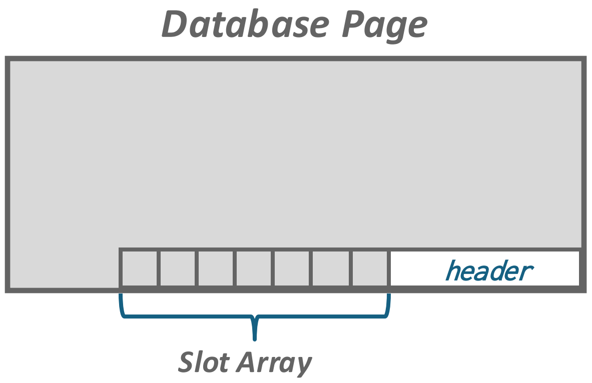


NSM: Physical Organization

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

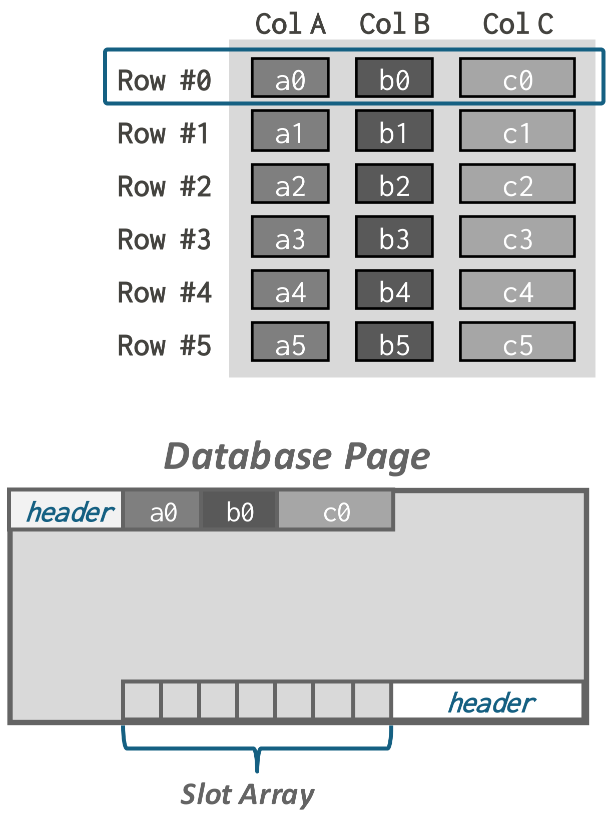
- The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.





NSM: Physical Organization

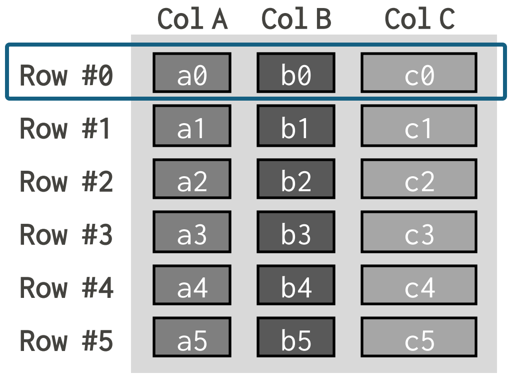
- A disk-oriented NSM system stores a tuple's fixed-length and variable-length attributes contiguously in a single slotted page.
- The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

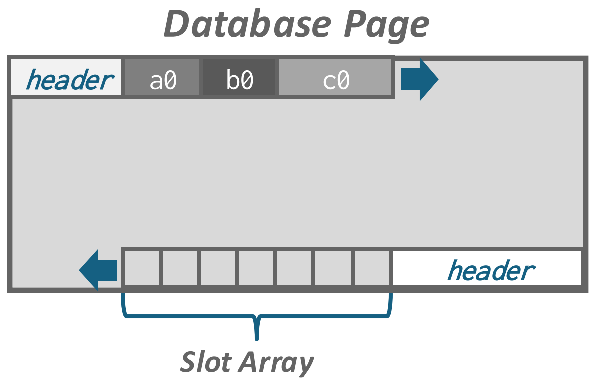


NSM: Physical Organization

- 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.

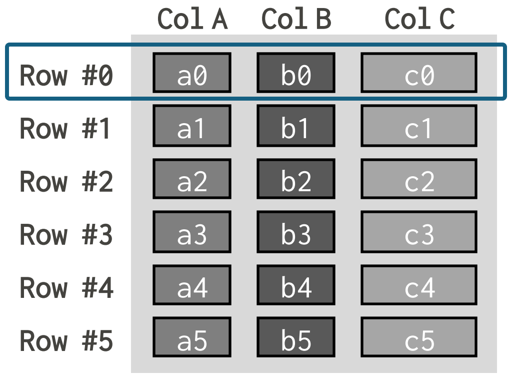


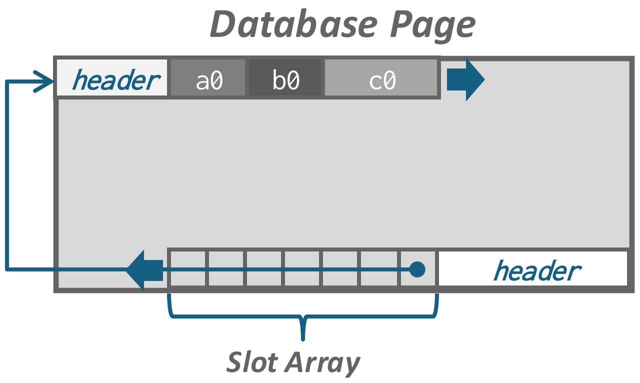


NSM: Physical Organization

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

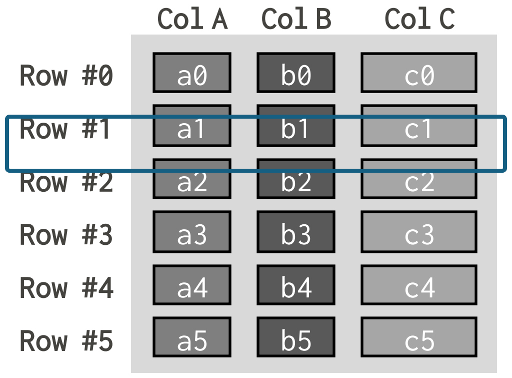
- The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

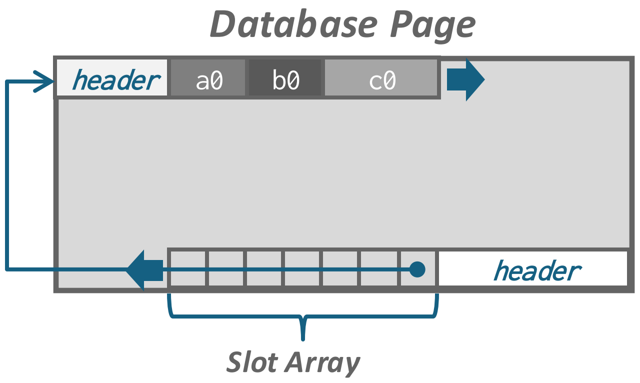




NSM: Physical Organization

- 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.

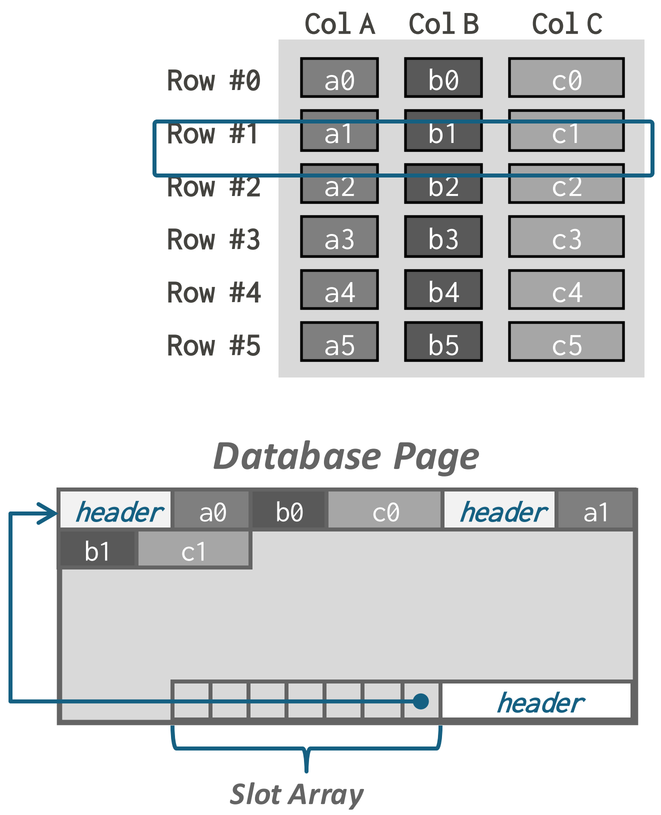




NSM: Physical Organization

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

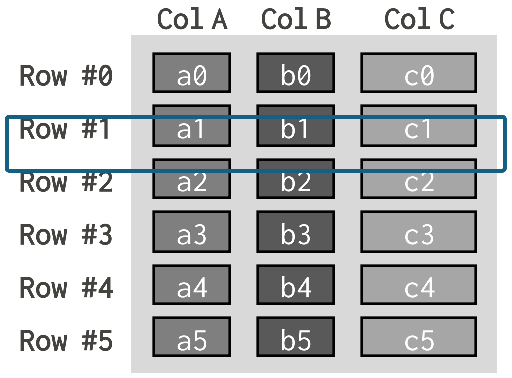
- The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.

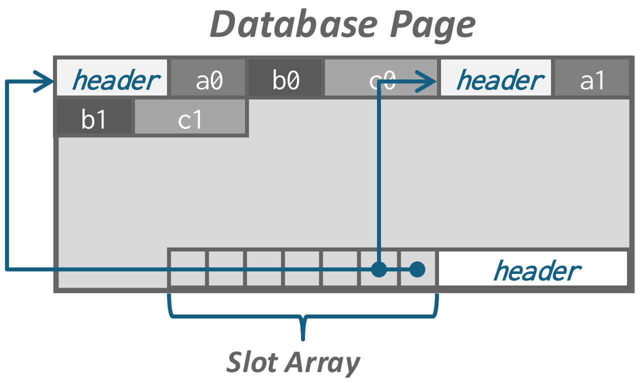


NSM: Physical Organization

- 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.

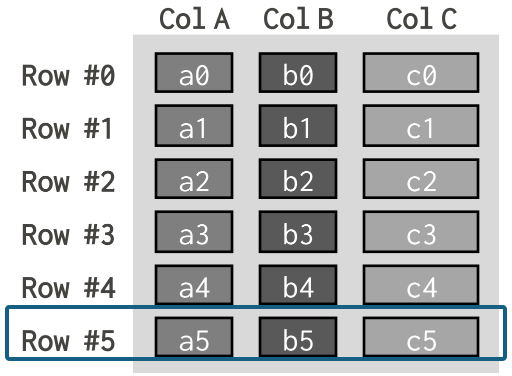


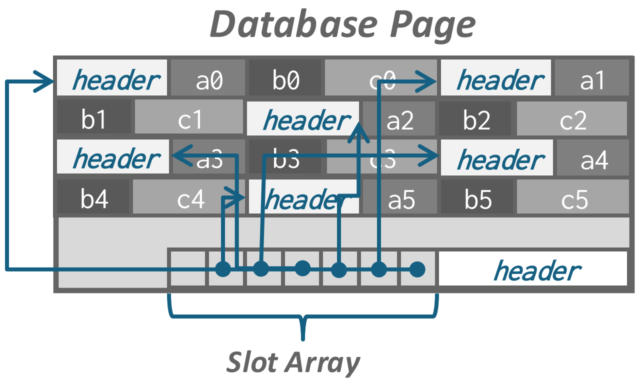


NSM: Physical Organization

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

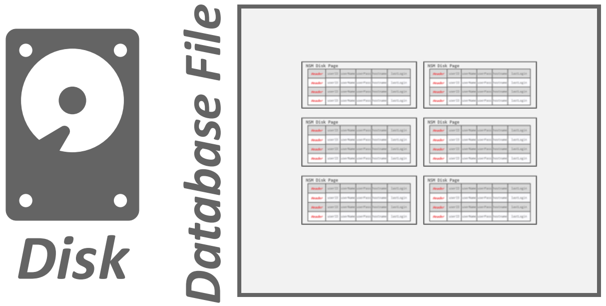
- The tuple's **record id** (page#, slot#) is how the DBMS uniquely identifies a physical tuple.





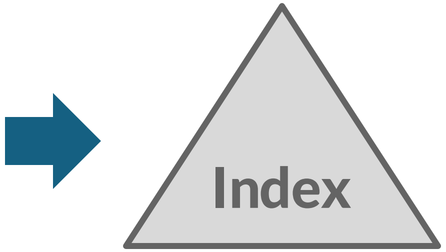
NSM: OLTP Example

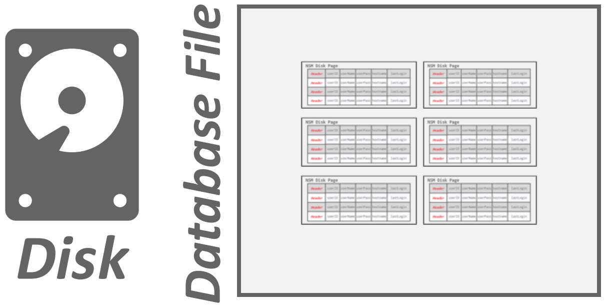
```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```



NSM: OLTP Example

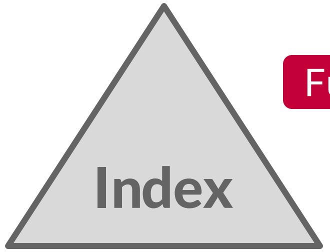
```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```



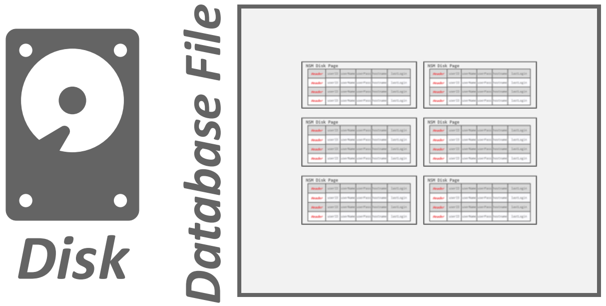


NSM: OLTP Example

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

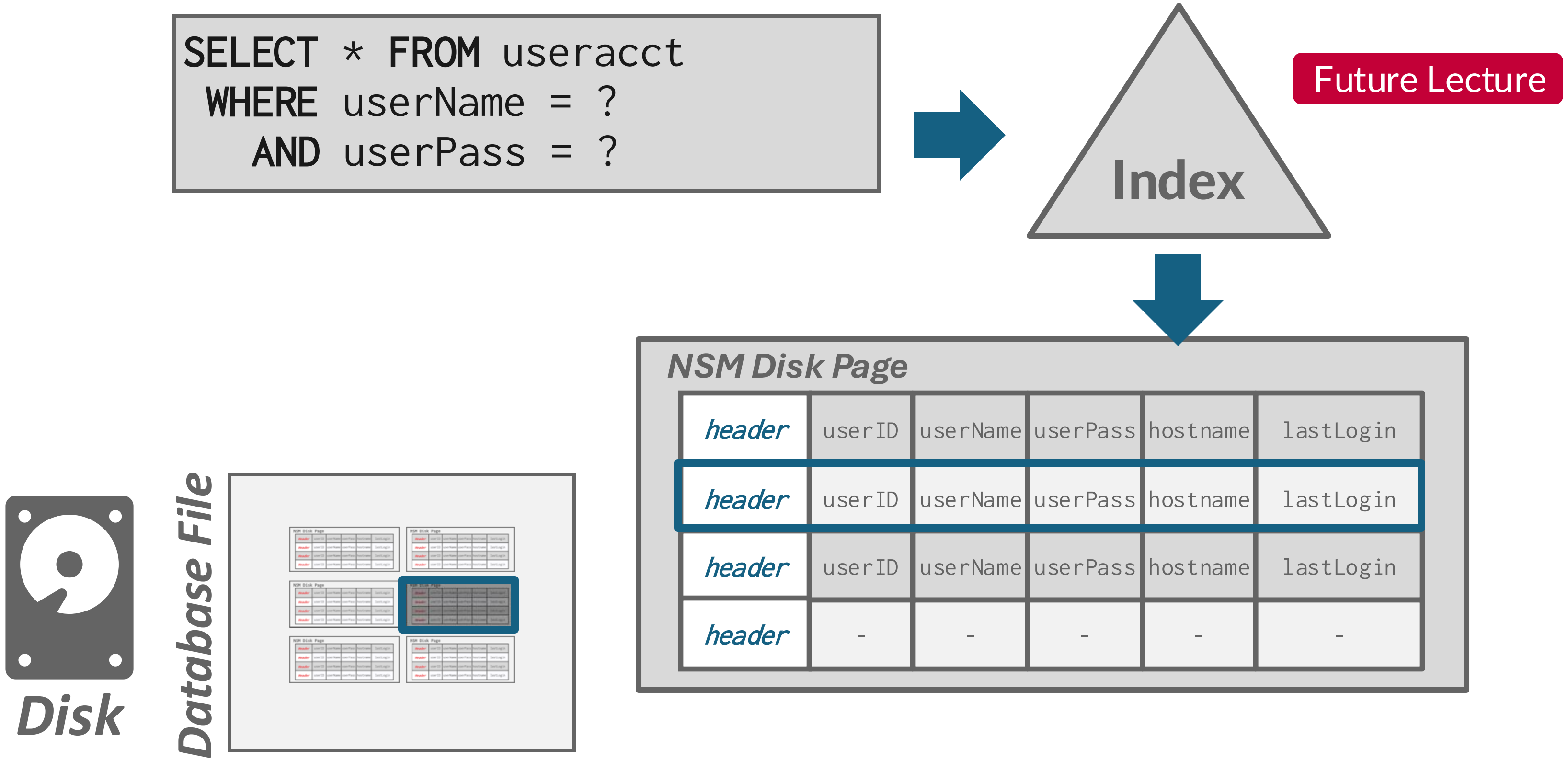


Future Lecture



NSM: OLTP Example

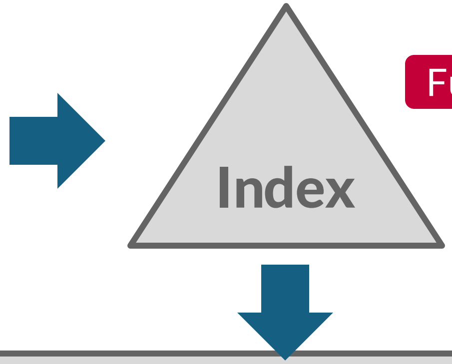
```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```



NSM: OLTP Example

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

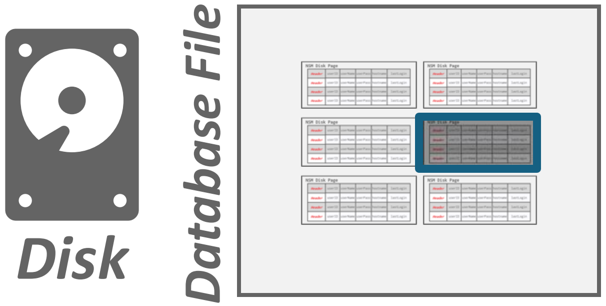
```
INSERT INTO useracct
VALUES (?,?,…?)
```



Future Lecture

NSM Disk Page

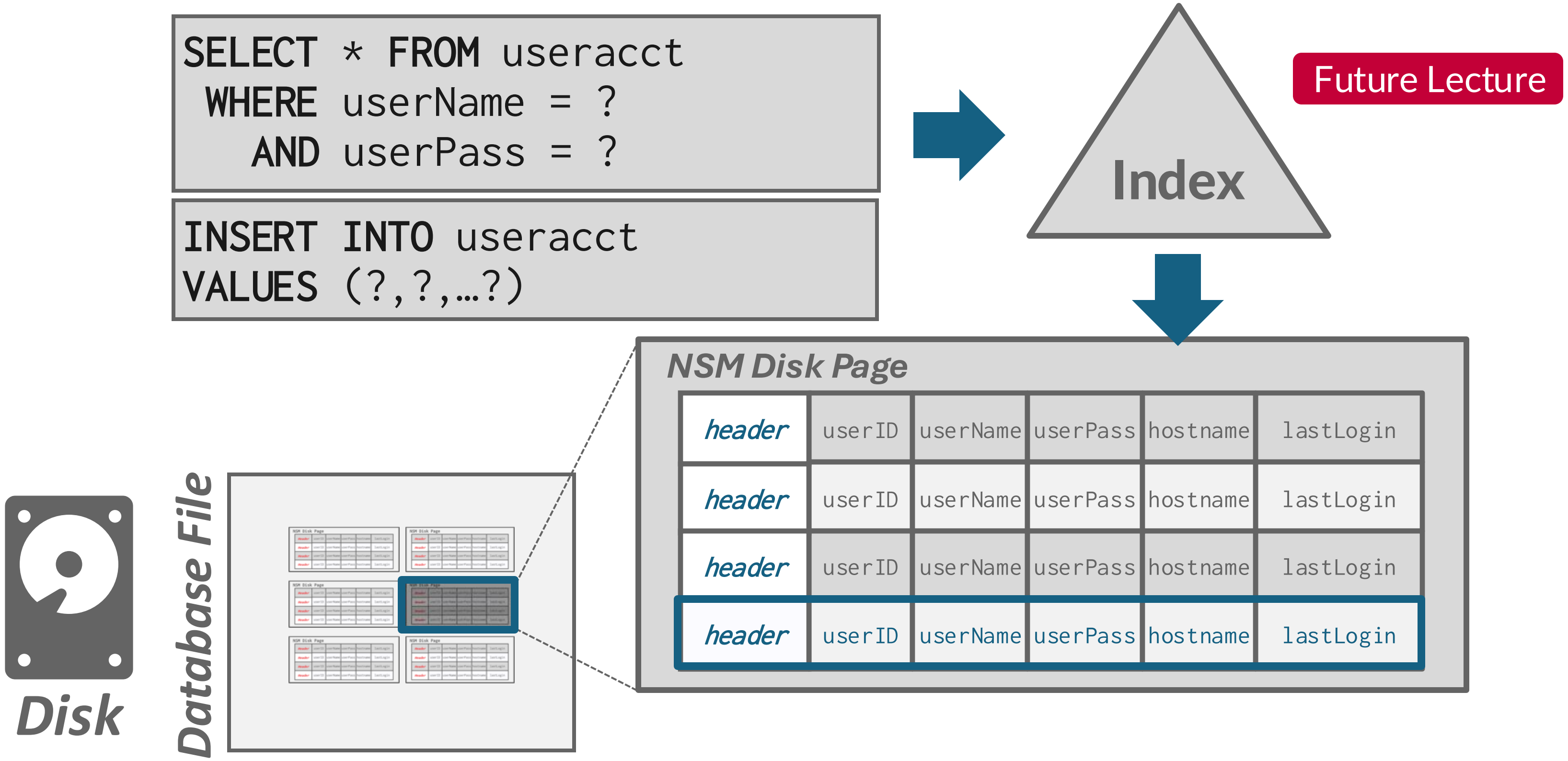
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	userID	userName	userPass	hostname	lastLogin
header	-	-	-	-	-



NSM: OLTP Example

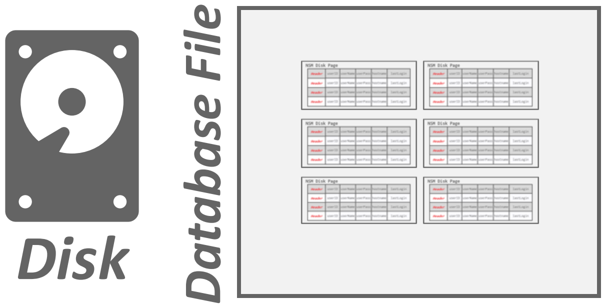
```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```

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



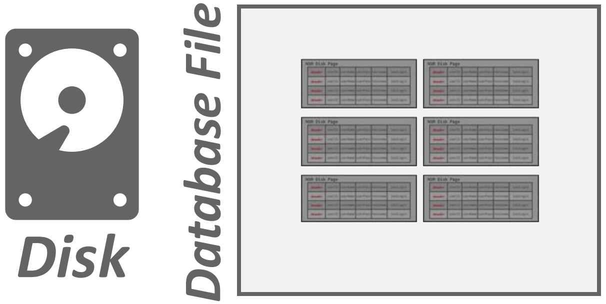
NSM: OLAP Example

```
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)
```



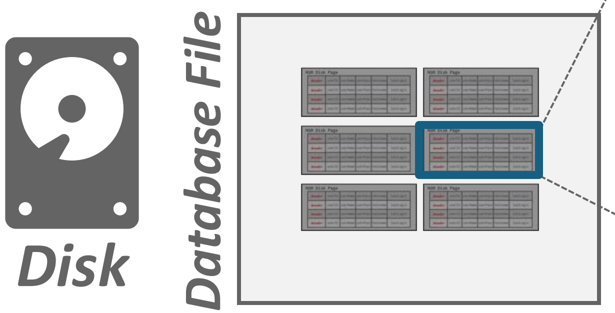
NSM: OLAP Example

```
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: 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)
```

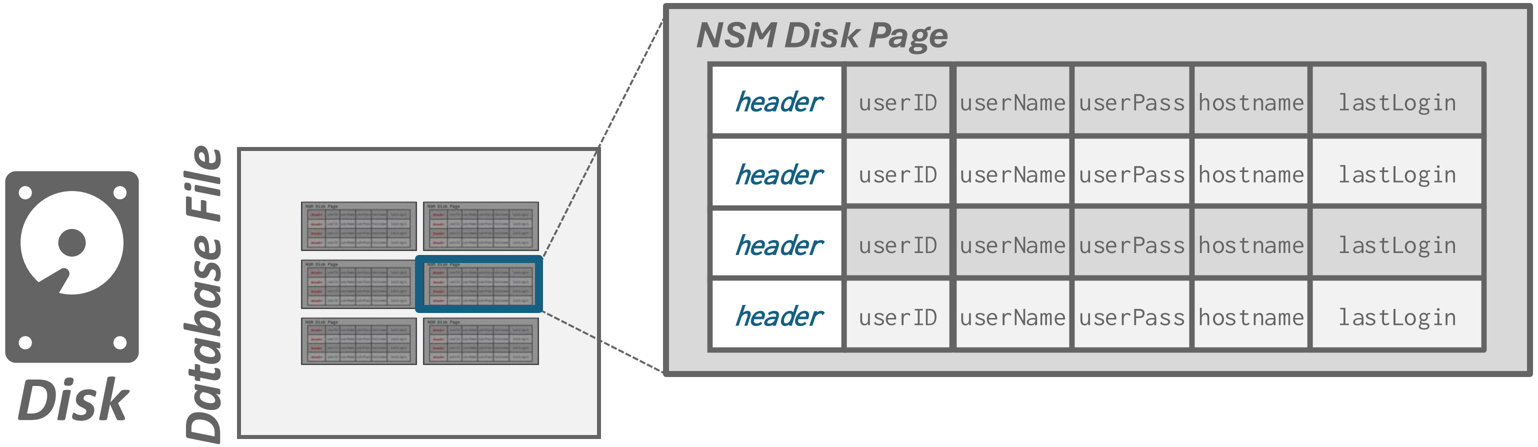


NSM Disk Page

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

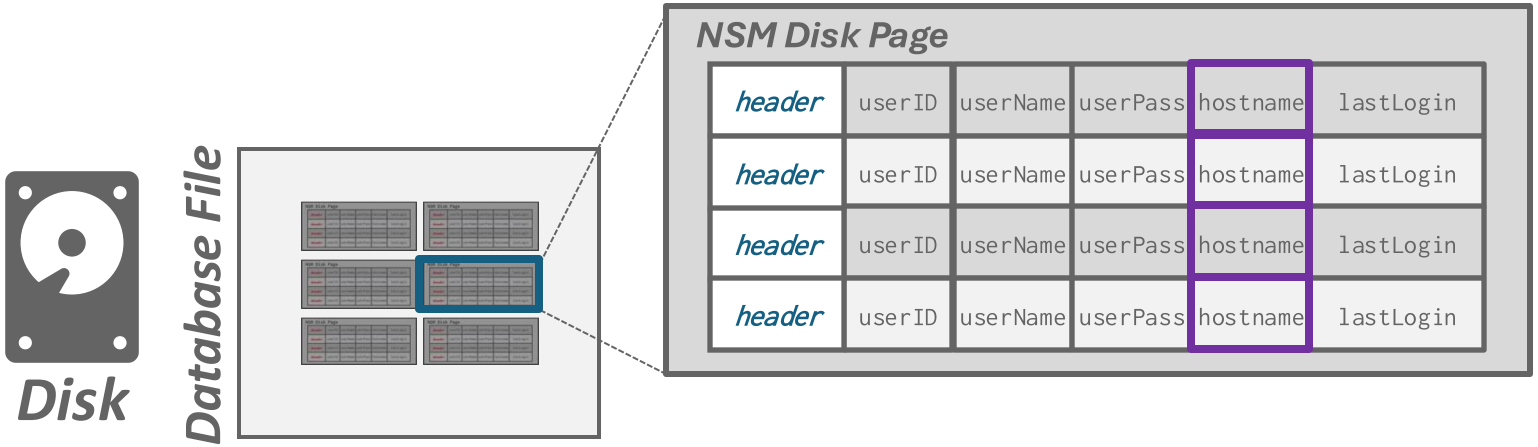
NSM: OLAP Example

```
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)
```



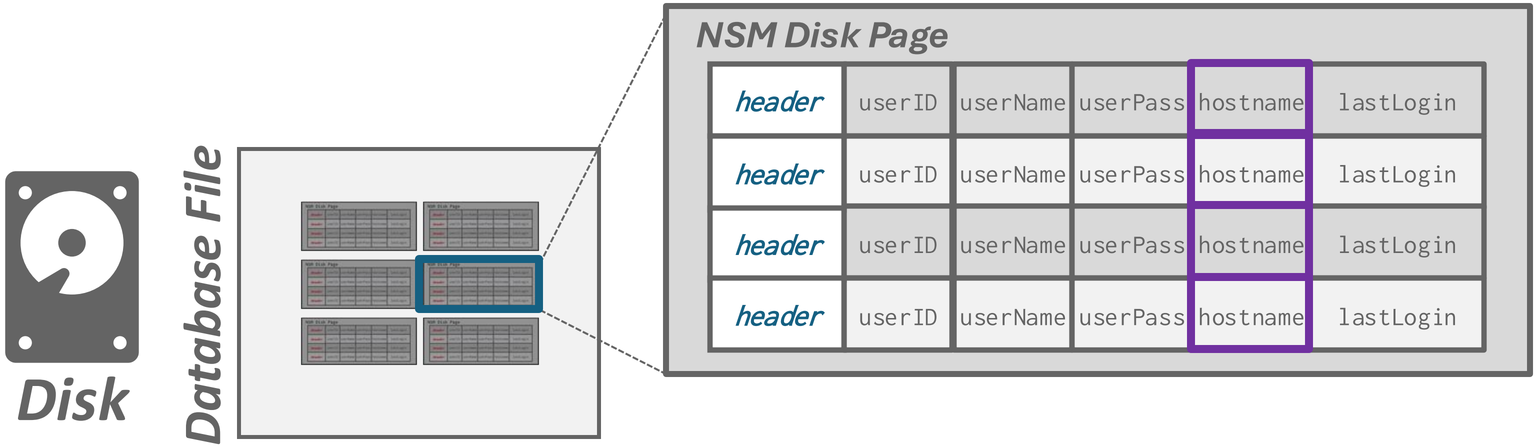
NSM: OLAP Example

```
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)
```



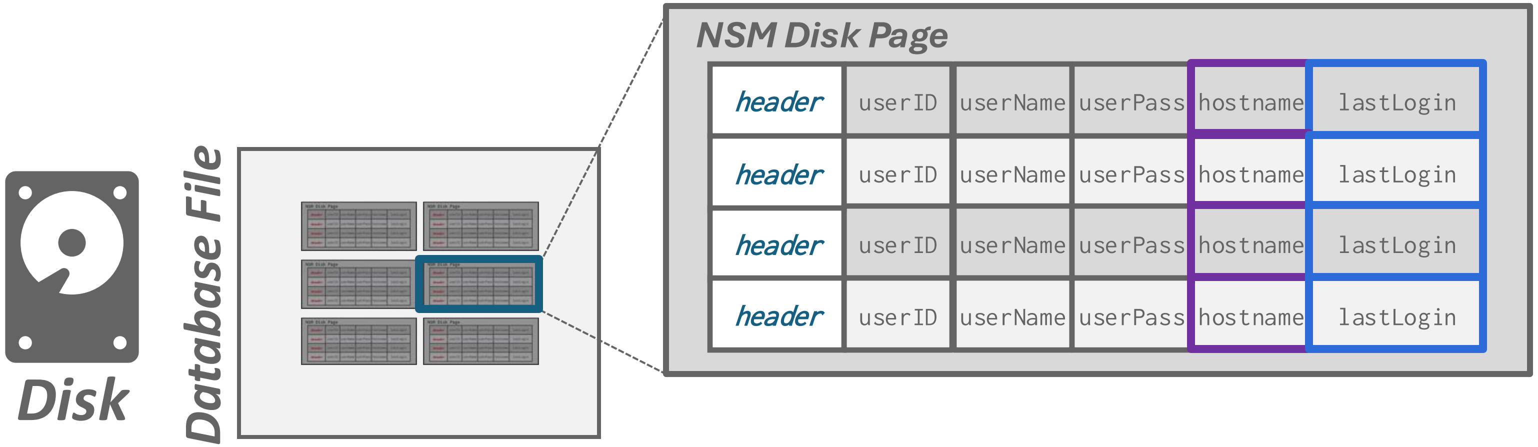
NSM: OLAP Example

```
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)
```



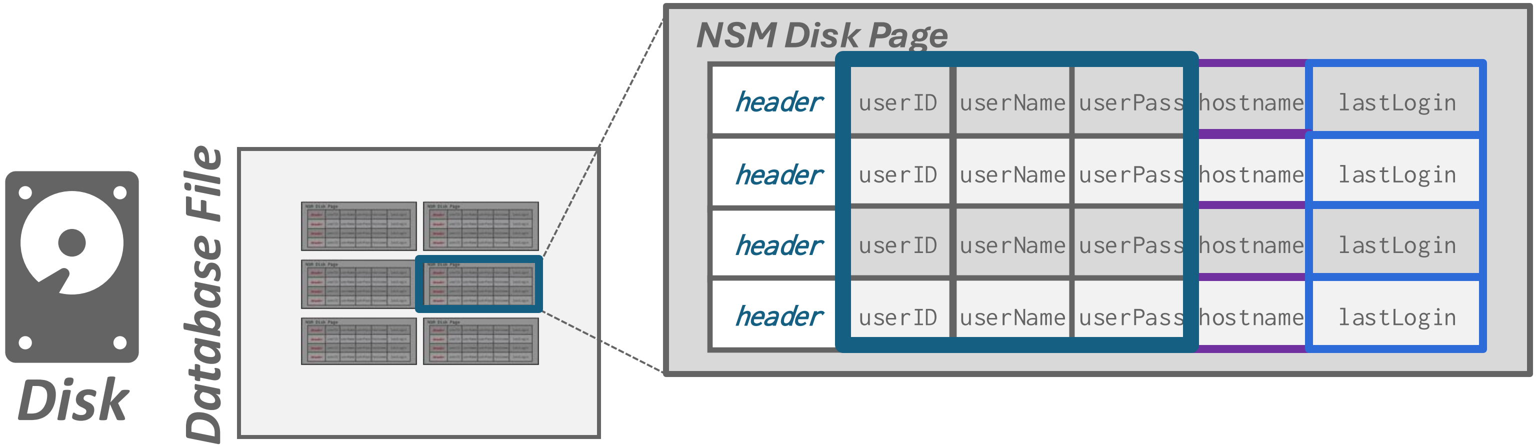
NSM: OLAP Example

```
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)
```



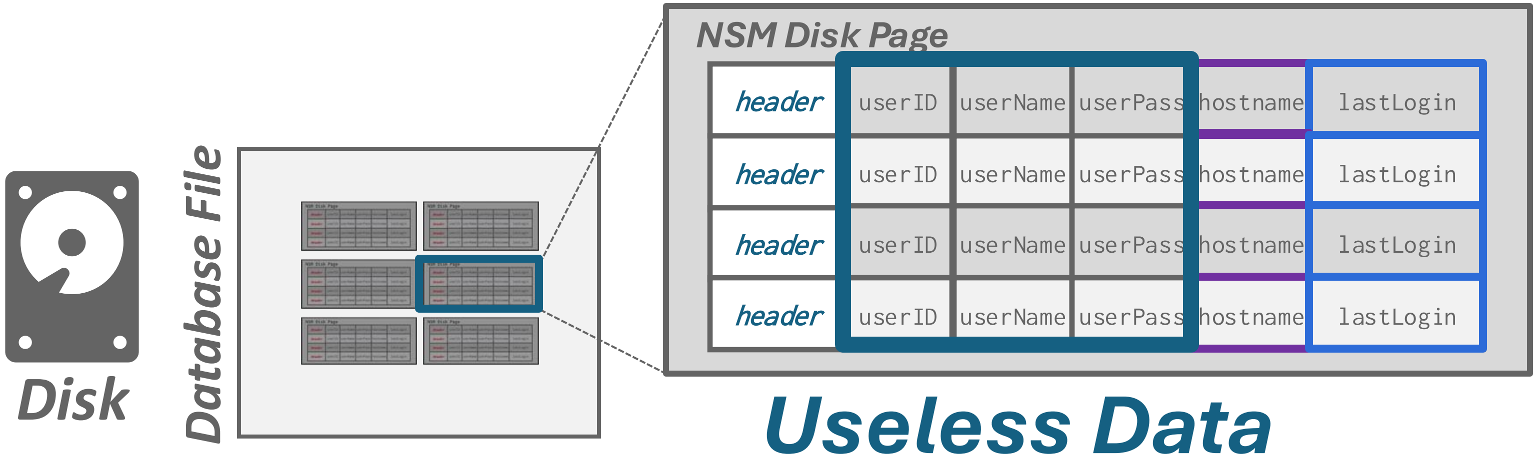
NSM: OLAP Example

```
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: 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 for OLAP access patterns.
→ Not ideal for compression because of multiple value domains within a single page.

Decomposition Storage Model (DSM)

Decomposition Storage Model

- 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.

A DECOMPOSITION STORAGE MODEL

George P Copeland
Setrag N Khoshafian

Microelectronics And Technology Computer Corporation
9430 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 (NSM) for a set of records This approach stores data as seen in the conceptual schema Also, various inverted file or cluster indexes might be added for improved access speeds The key concept in the NSM is that all attributes of a conceptual schema record are stored together For example, the conceptual schema relation

R	sur	a1	a2	a3
	s1	v11	v21	v31
	s2	v12	v22	v32
	s3	v13	v23	v33

contains a surrogate for record identity and three attributes per record The NSM would store si, v1i, v2i and v3i together for each record i

Some database systems use a fully transposed storage model, for example, RM (Lorie and Symonds 1971), TOD (Wiederhold et al 1975), RAPID (Turner et al 1979), ALDS (Burnett and Thomas 1981), Delta (Shibayama et al 1982) and (Tanaka 1983) This approach stores all values of the same attribute of a conceptual schema relation together Several studies have compared the performance of transposed storage models with the NSM (Hoffer 1976, 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 schema record in a binary relation For example, the above relation would be stored as

a1	sur	val
	s1	v11
	s2	v12
	s3	v13

a2	sur	val
	s1	v21
	s2	v22
	s3	v23

a3	sur	val
	s1	v31
	s2	v32
	s3	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 complexity 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 (Abrial 1974, Deliyanni and Kowalski 1977, Kowalski 1978, Codd 1979) have argued for the semantic clarity and generality of representing each basic fact individually within the conceptual schema as the DSM does within the storage schema

268

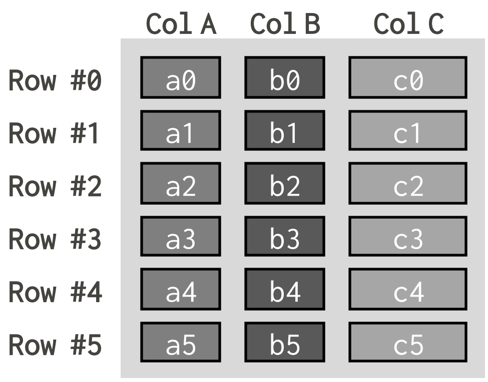
DSM: Physical Organization

- Store attributes and metadata (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 metadata about the entire column.



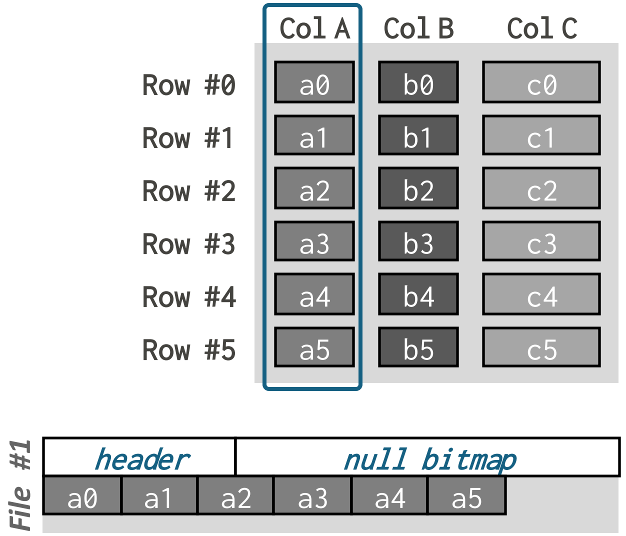
DSM: Physical Organization

- Store attributes and metadata (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 metadata about the entire column.



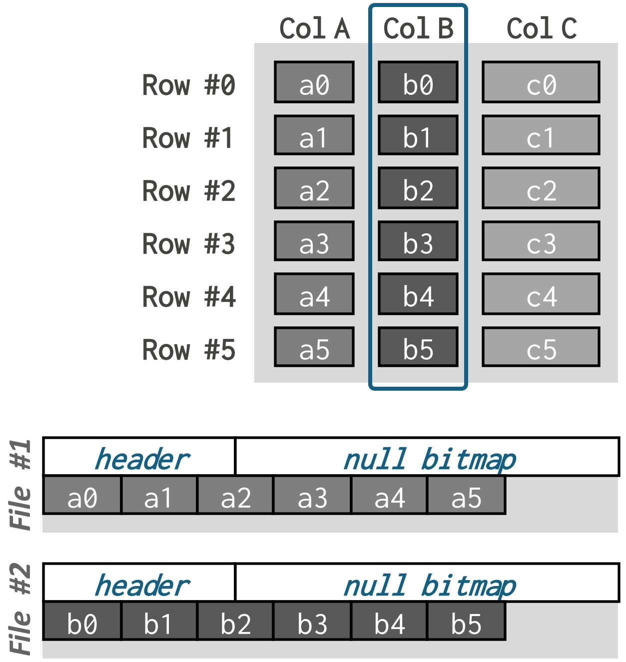
DSM: Physical Organization

- Store attributes and metadata (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 metadata about the entire column.



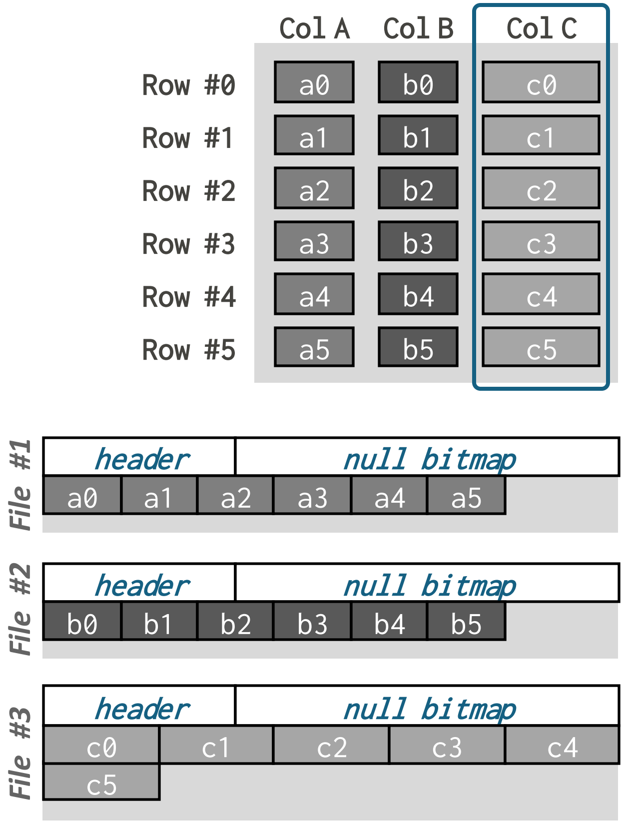
DSM: Physical Organization

- Store attributes and metadata (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 metadata about the 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.

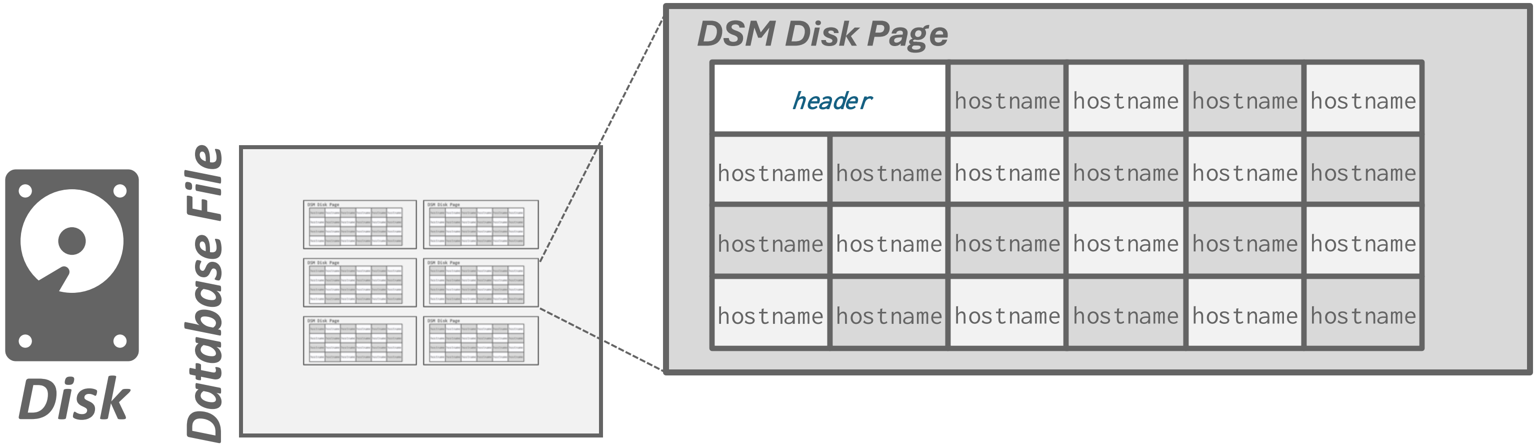
→ Also known as a "column store".

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.

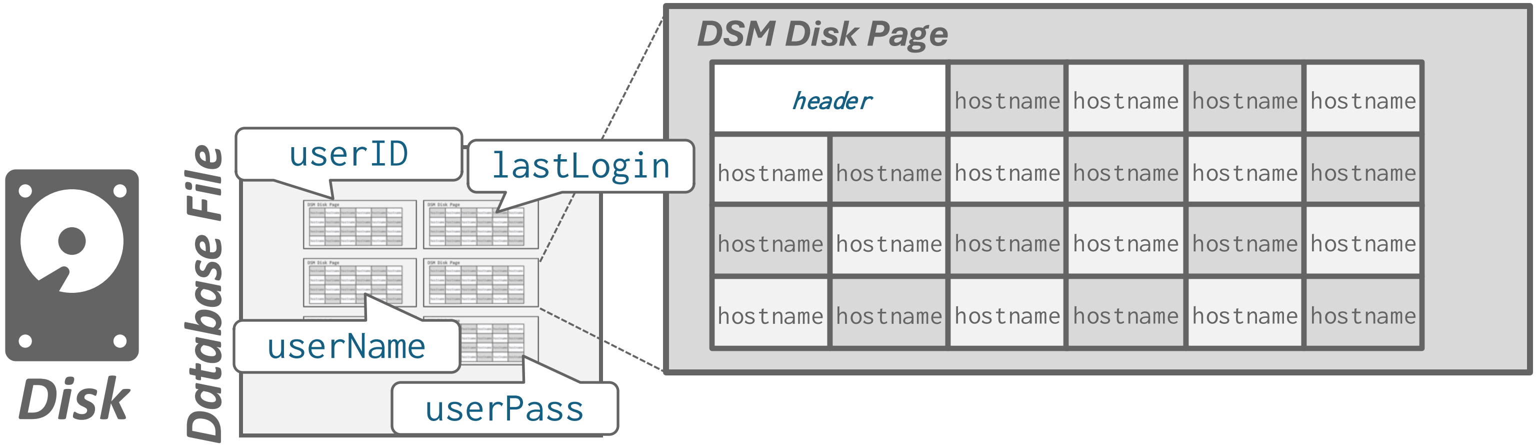
→ Also known as a “column store”.



DSM: 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: Tuple Identification

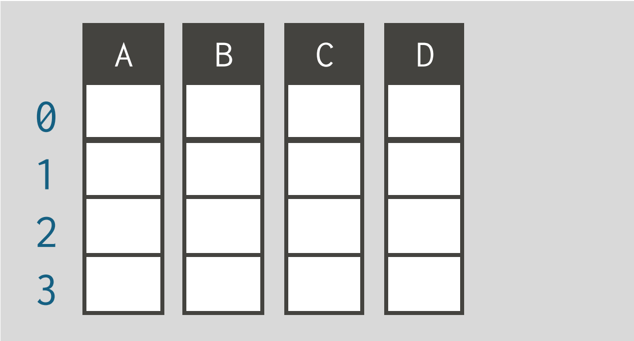
- **Choice #1: Fixed-length Offsets**

→ Each value is the same length for an attribute.

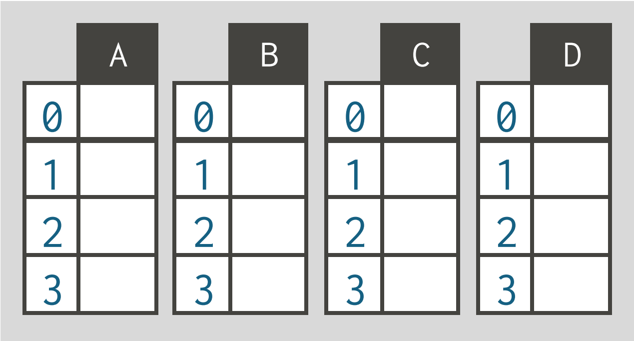
- **Choice #2: Embedded Tuple Ids**

→ 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 in a few slides.

DSM: System History

- **1970s:** Cantor DBMS
- **1980s:** DSM Proposal
- **1990s:** SybaseIQ (in-memory only)
- **2000s:** Vertica, Vectorwise, MonetDB
- **2010s:** Everyone + Parquet / ORC

DSM: System History

- **1970s:** Cantor DBMS
- **1980s:** DSM Proposal
- **1990s:** SybaseIQ (in-memory only)
- **2000s:** Vertica, Vectorwise, MonetDB
- **2010s:** Everyone + Parquet / ORC

SCHOOL OF DATA SCIENCE

Decomposition Storage Model

- **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 in a few slides).

- **Disadvantages**

→ Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching/reorganization.

Hybrid Storage Model (PAX)

SCHOOL OF DATA SCIENCE

Observation

- OLAP queries rarely 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 a 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 Parquet and Orc use.

- 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 [‡]
Carnegie Mellon University
natassa@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 optimzed for I/O performance and organized records sequentially on disk pages using the N-ary Storage Model (NSM) (a.k.a., 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 affects layout inside the pages, it incurs no storage penalty and does not affect I/O 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 I/O execute 11-48% faster.

1 Introduction

The communication between the CPU and the secondary storage (I/O) 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 I/O, 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 I/O [20][5][26]. When running commercial database systems on a modern processor, data requests that miss in the cache hierarchy (i.e., requests for data that are not found 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 record), 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

[‡] Work done while author was at the University of Wisconsin-Madison.

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

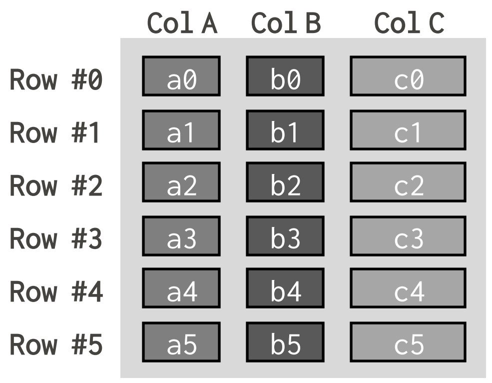
PAX: Physical Organization

- 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 metadata header about its contents.



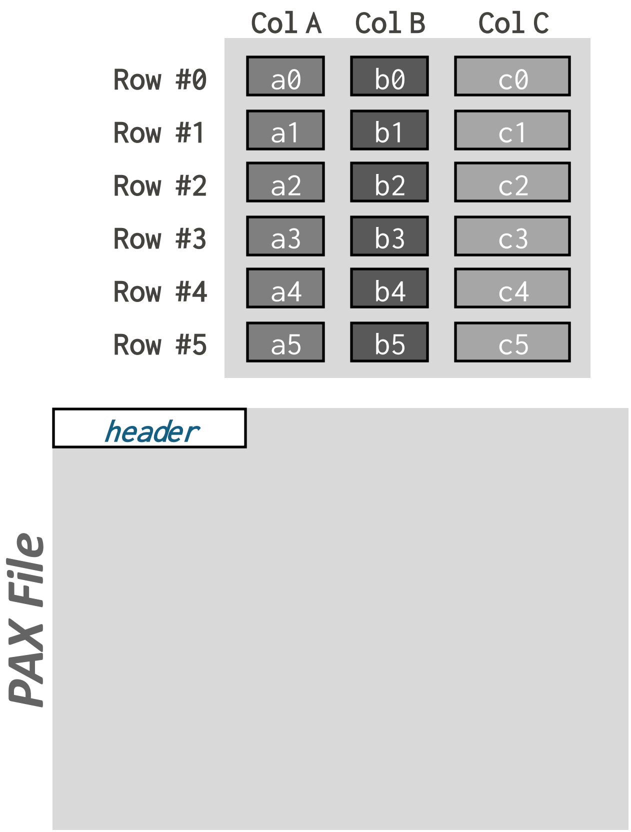
PAX: Physical Organization

- 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 metadata header about its contents.

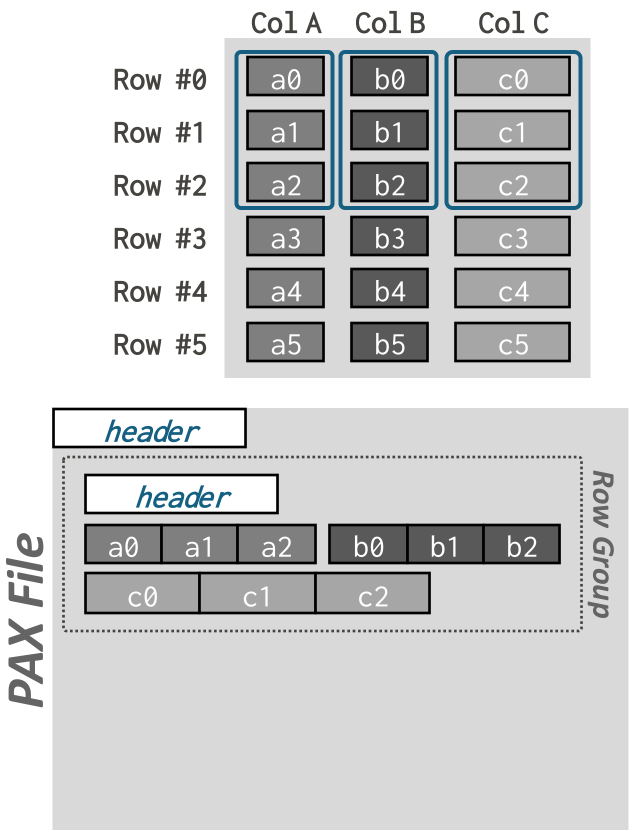


PAX: Physical Organization

- 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 metadata header about its contents.

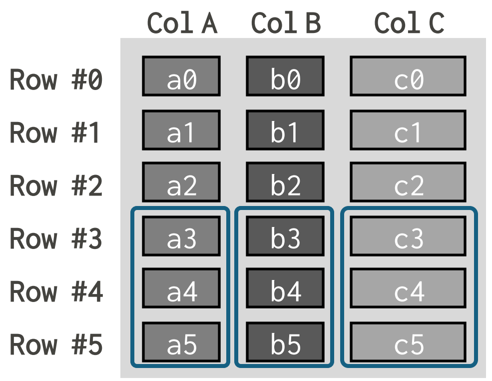


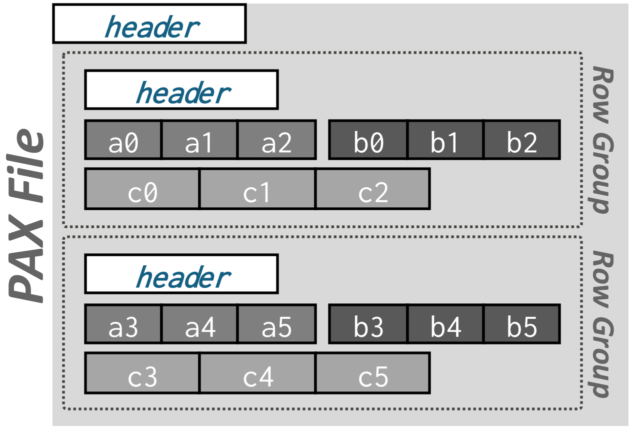
PAX: Physical Organization

- 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 metadata header about its contents.





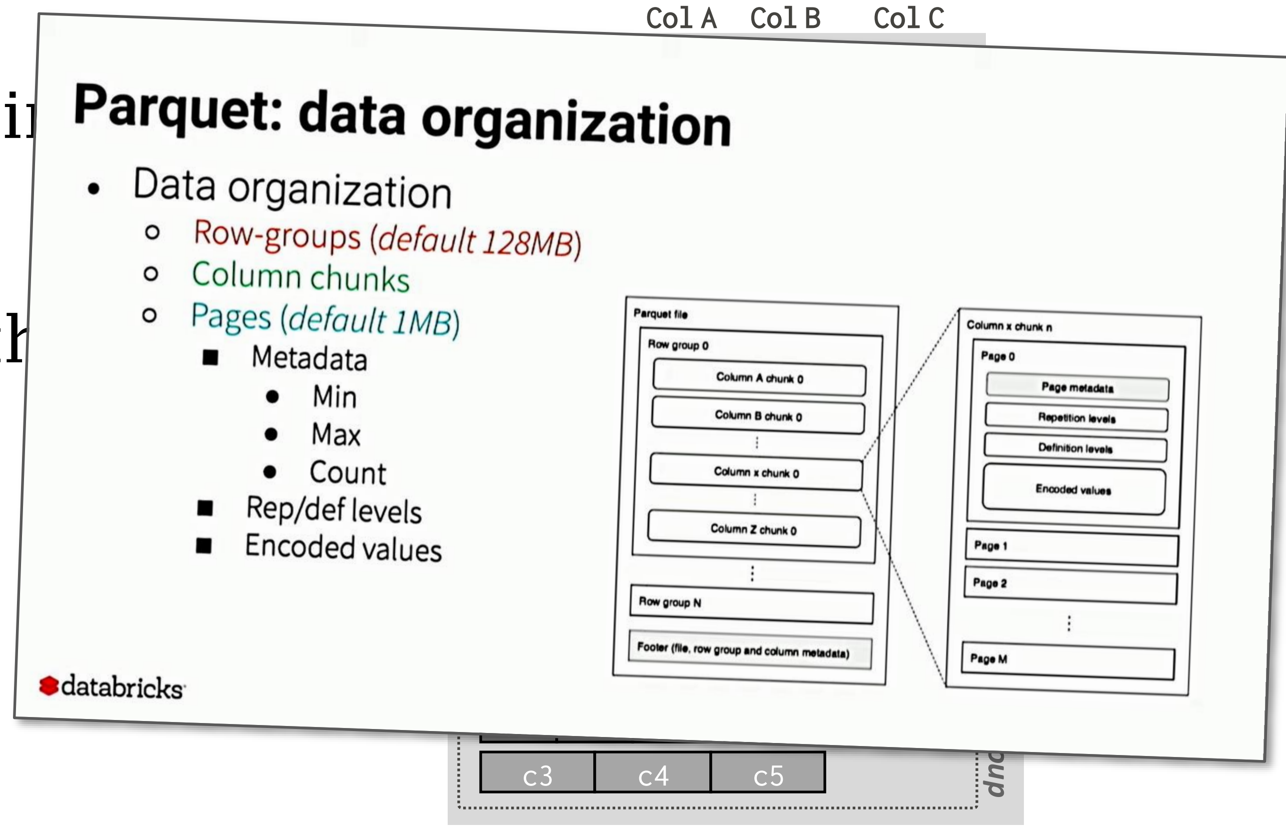
PAX: Physical Organization

- 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 metadata header about its contents.



Compression

SCHOOL OF DATA SCIENCE

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 <u>late materialization</u>.

- **Goal #3:** Must be a <u>lossless</u> scheme.

Lossless v.s. 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, row-store).

- **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, column store).

SCHOOL OF DATA SCIENCE

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, row-store).

- **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, column store).

SCHOOL OF
DATA SCIENCE

Naïve Compression

- Compress data using a general-purpose algorithm. The scope of compression is only based on the data provided as input.

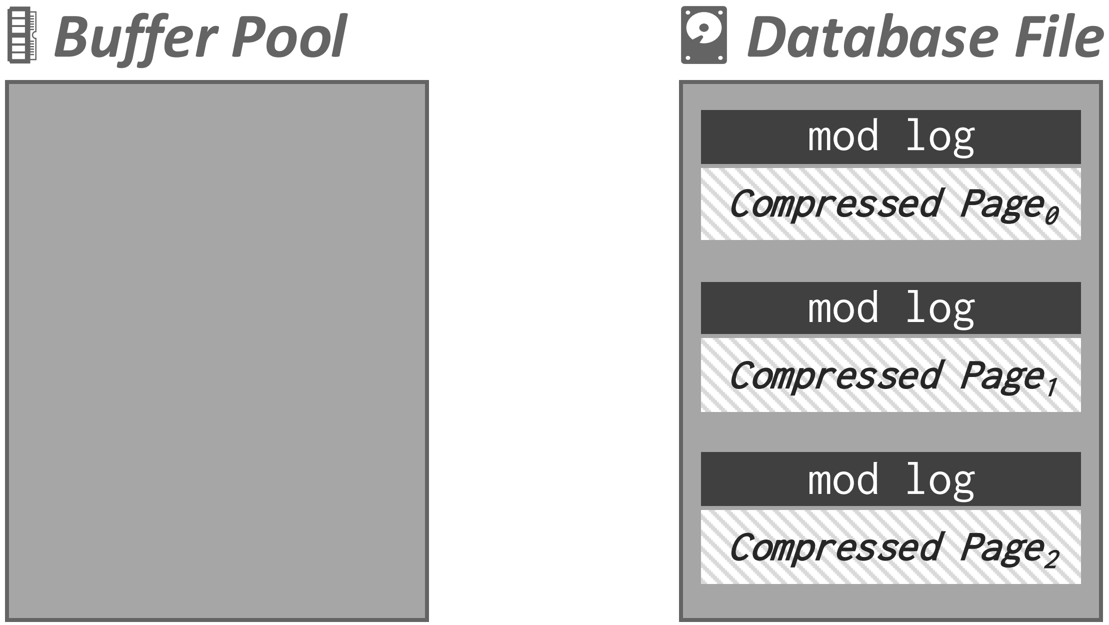
→ LZO (1996), LZ4 (2011), Snappy (2011), Oracle OZIP (2014), Zstd (2015)

- Considerations

→ Computational overhead

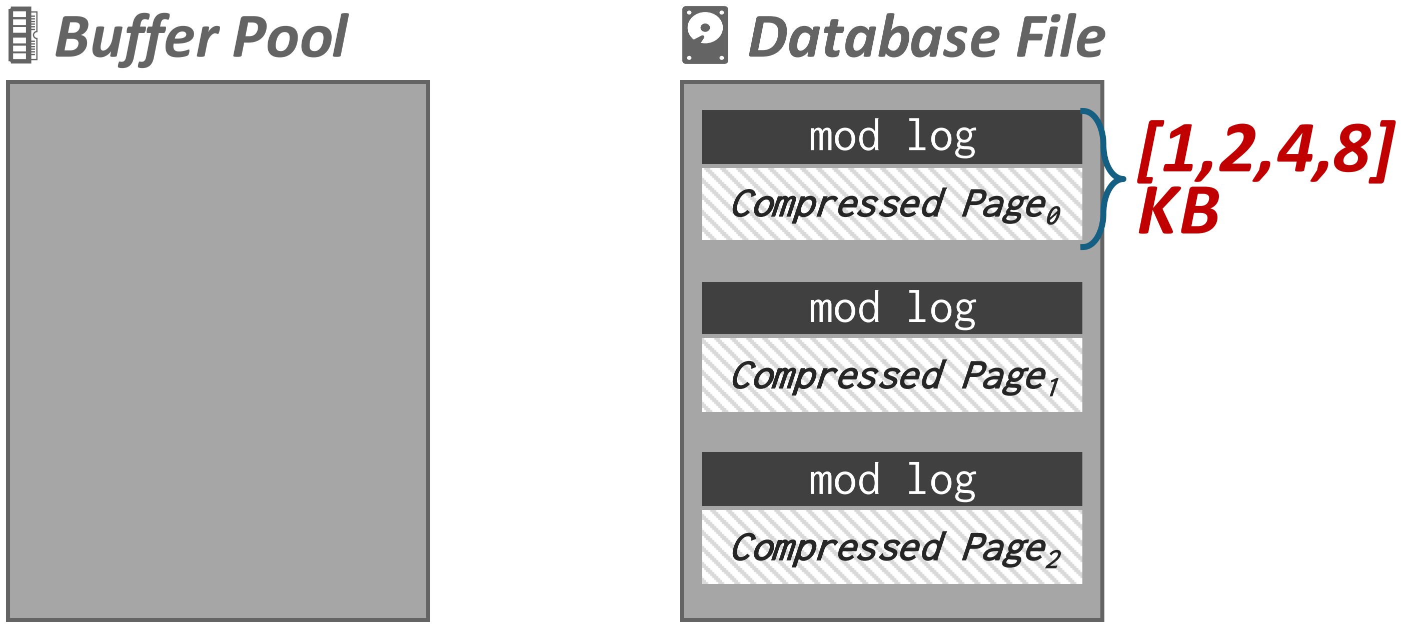
→ Compress vs. decompress speed.

MySQL InnoDB Compression



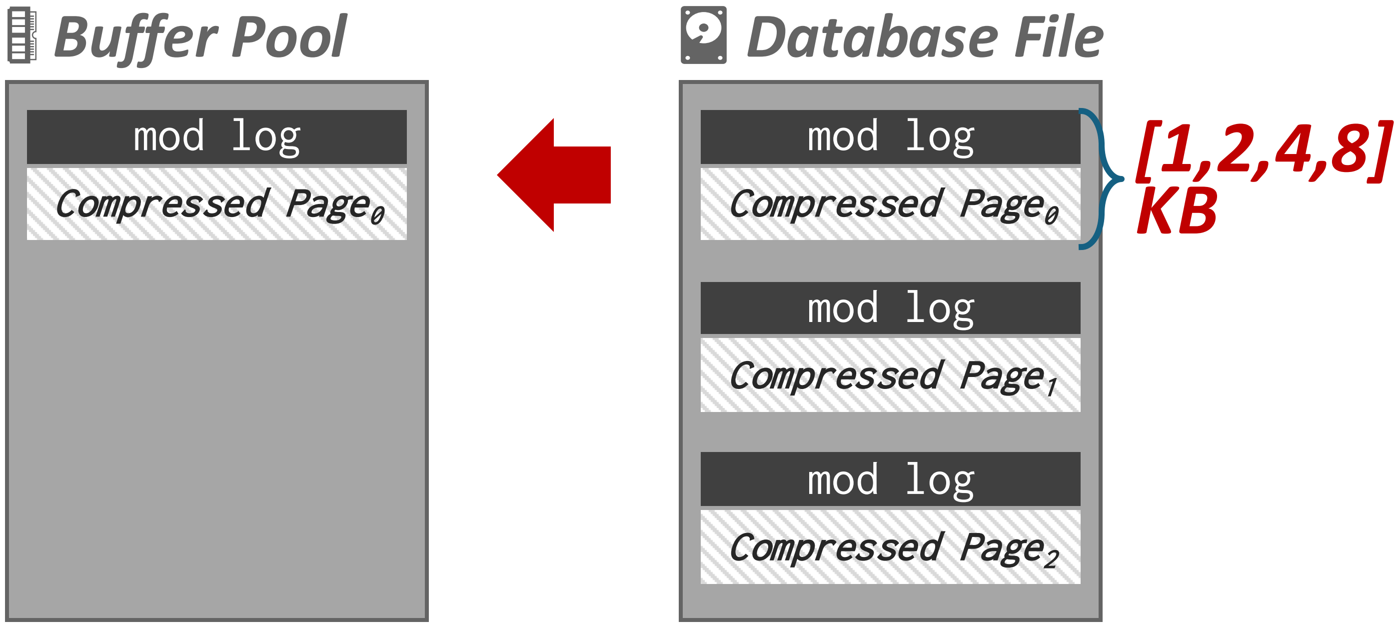
Source: [MySQL 5.7 Documentation]

MySQL InnoDB Compression



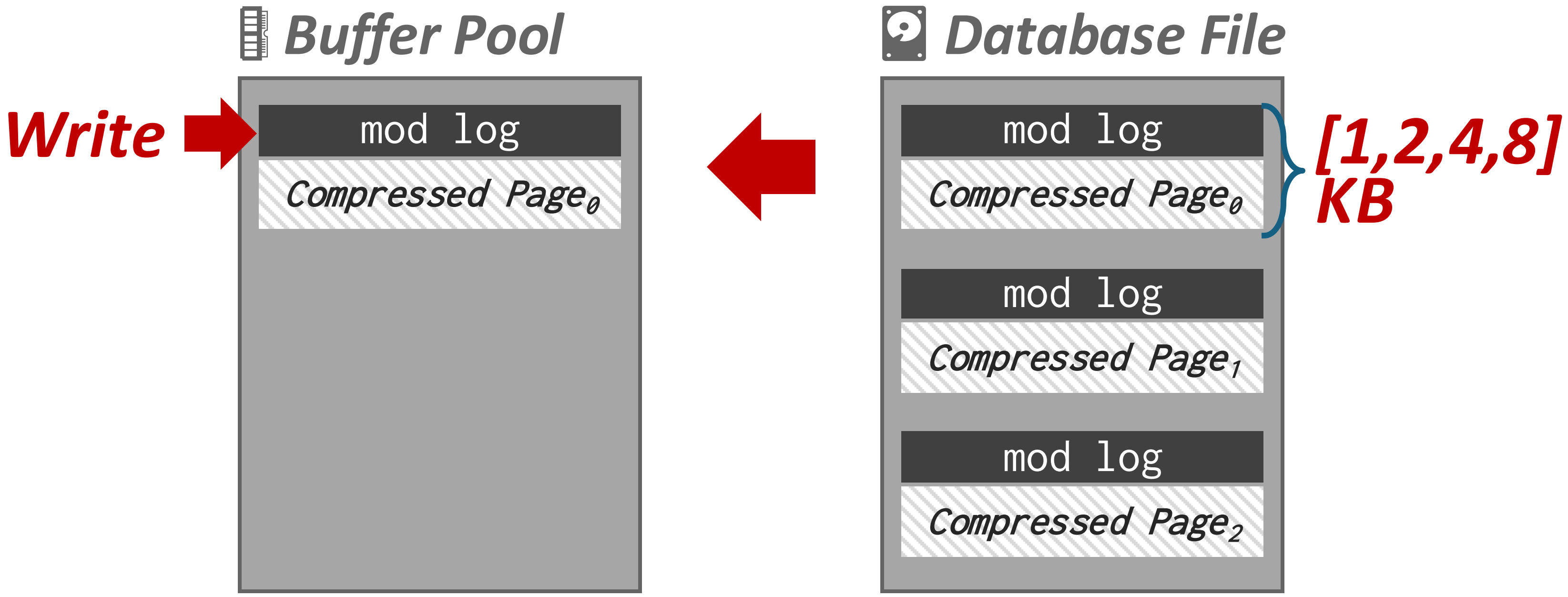
Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



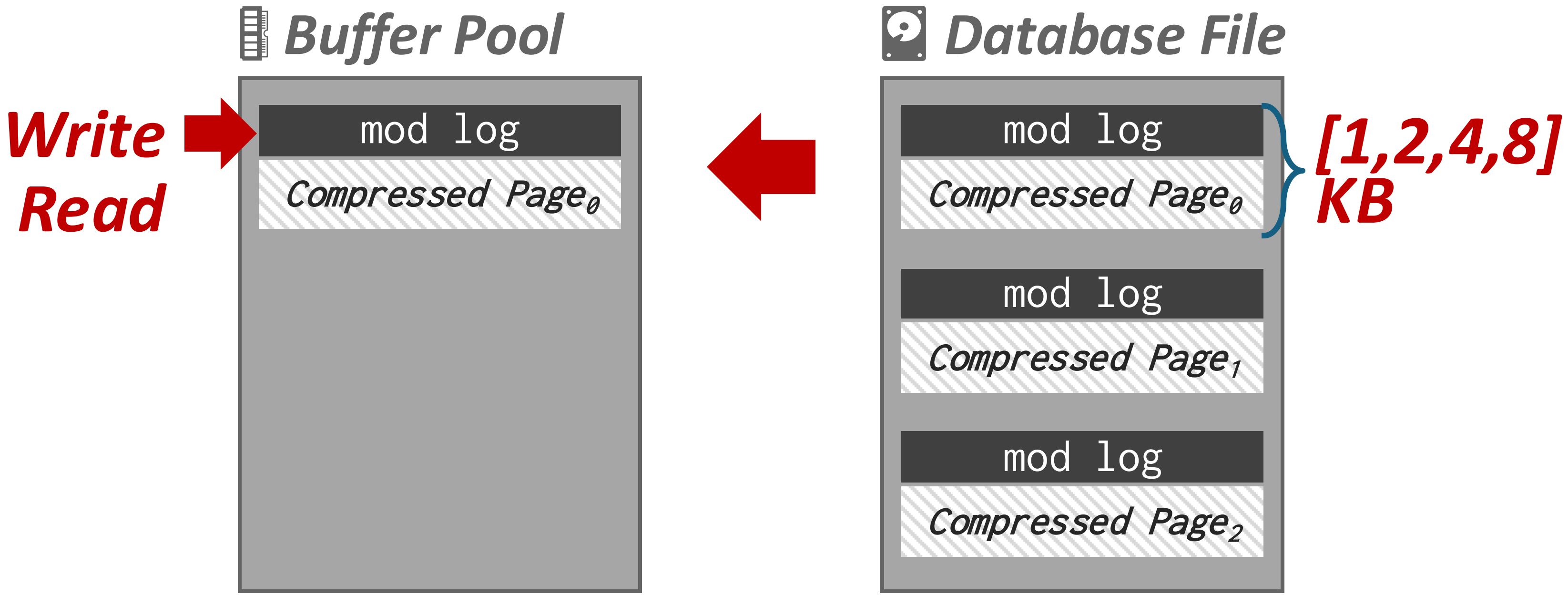
Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



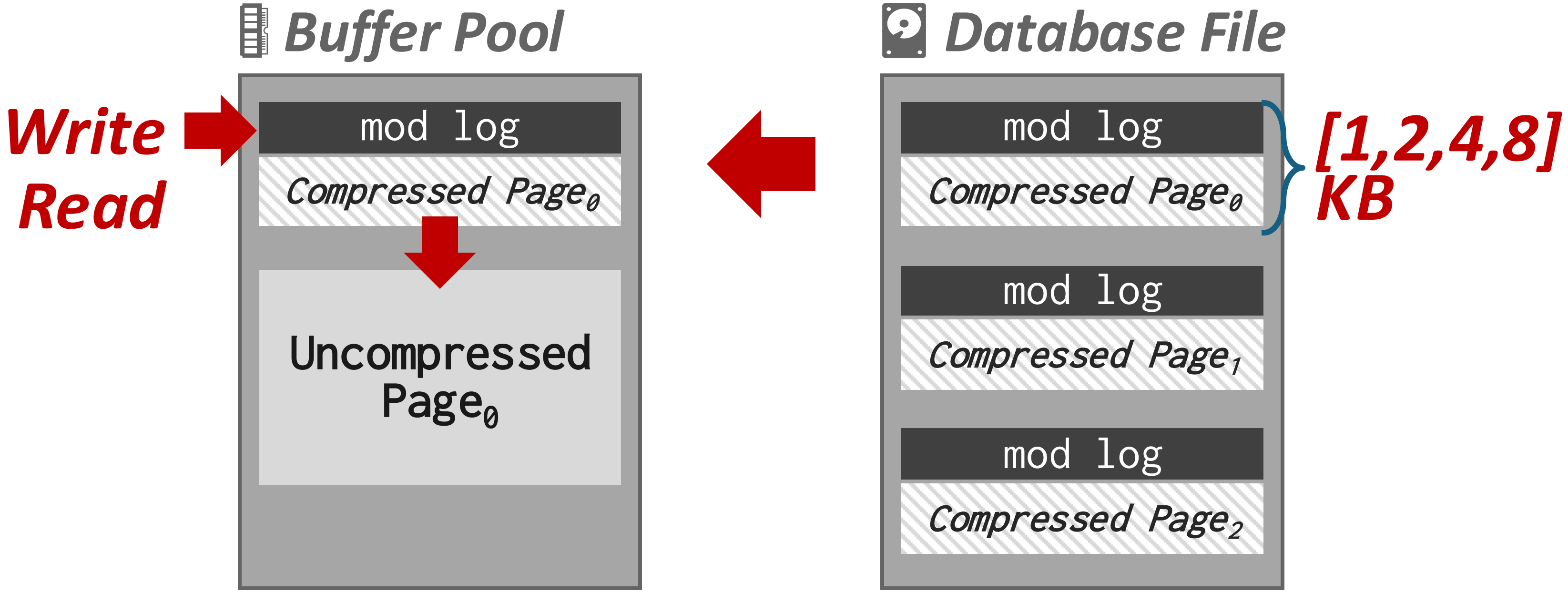
Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



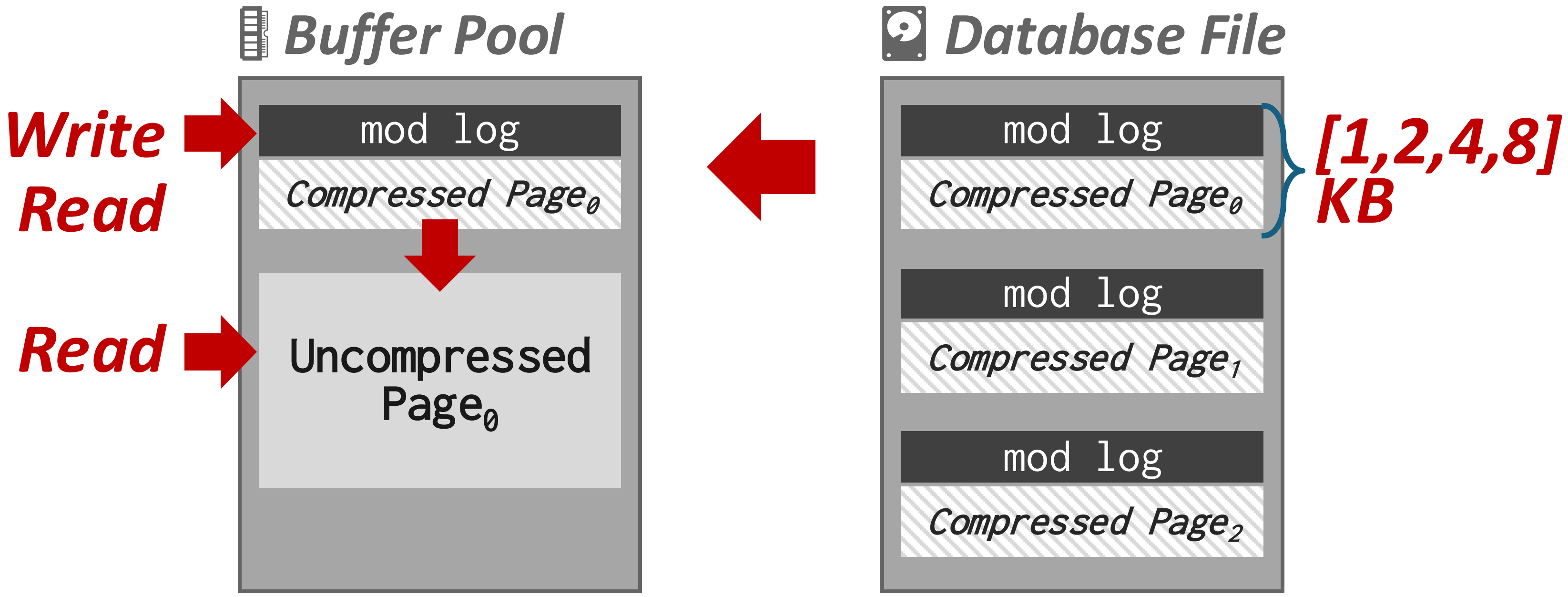
Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



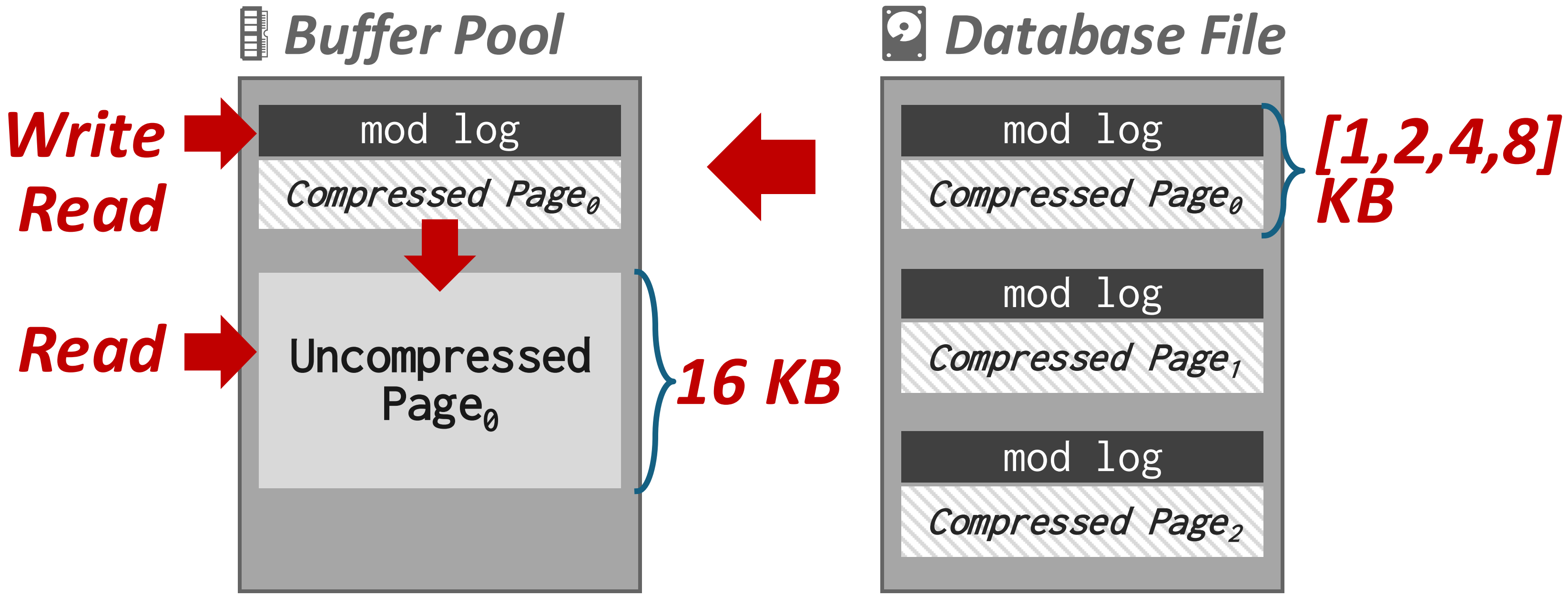
Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



Source: MySQL 5.7 Documentation

MySQL InnoDB Compression



Source: MySQL 5.7 Documentation

SCHOOL OF DATA SCIENCE

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.

Observation

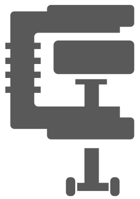
- Ideally, we want the DBMS to operate on compressed data without decompressing it first.

NAME	SALARY
Andy	99999
Jane	88888

Observation

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

NAME	SALARY
Andy	99999
Jane	88888



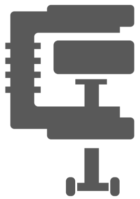
NAME	SALARY
XX	AA
YY	BB

Observation

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

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

NAME	SALARY
Andy	99999
Jane	88888



NAME	SALARY
XX	AA
YY	BB

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
```

NAME	SALARY
Andy	99999
Jane	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, row-store).

- **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, column store).

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, row-store).

- **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, column store).

SCHOOL OF
DATA SCIENCE

Columnar Compression

- Run-length Encoding
- Bit-Packing Encoding
- Bitmap Encoding
- Delta 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.

SCHOOL OF DATA SCIENCE

Run-Length Encoding

Original Data

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

SCHOOL OF DATA SCIENCE

Run-Length Encoding

Original Data

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

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***

SCHOOL OF
DATA SCIENCE

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***

SCHOOL OF
DATA SCIENCE

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

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 <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
191
56
92
81
120
231
172

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	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.

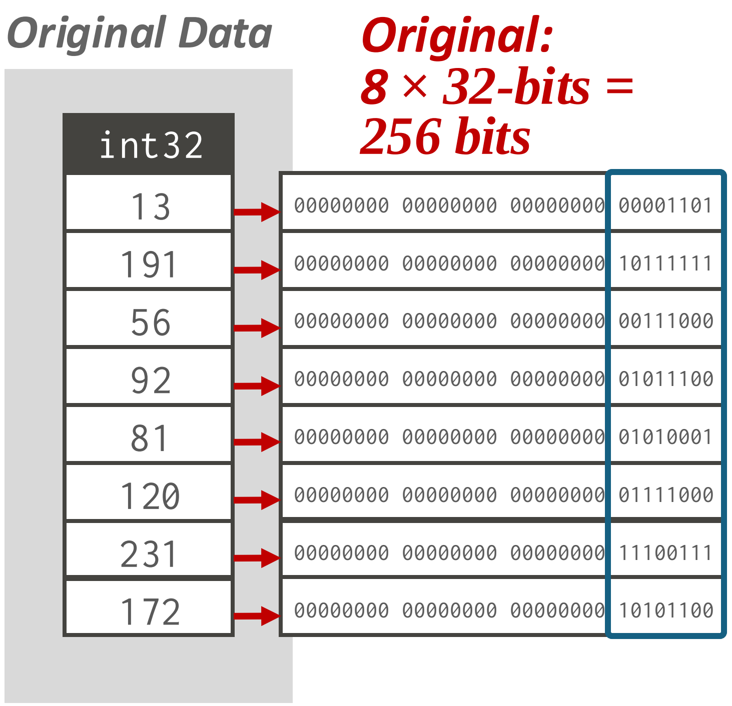
Original Data

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

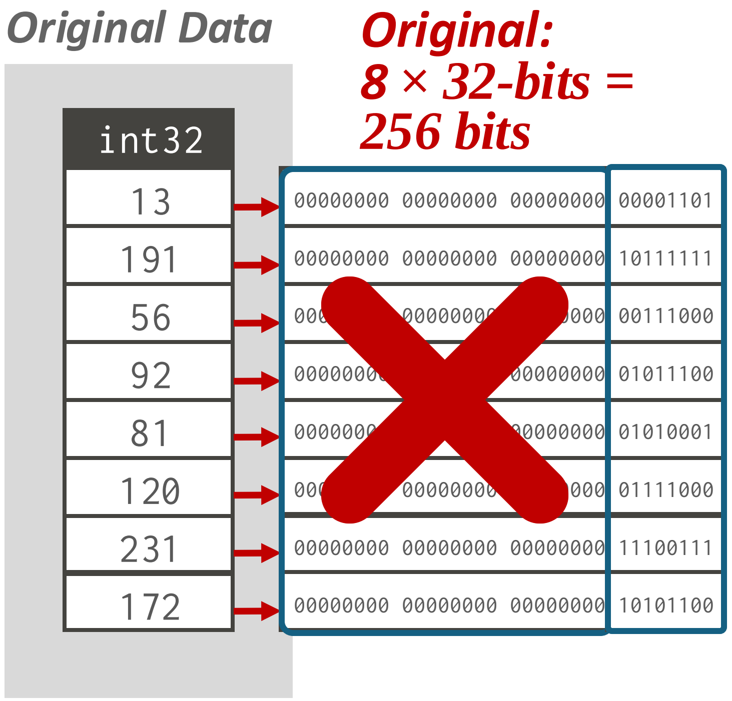
Bit Packing

- 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.



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	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

Patching / Mostly Encoding

- A variation of bit packing when attribute's values are "mostly" less than the largest size, store them with the smaller data type.

→ The remaining values that cannot be compressed are stored in their raw form.

Original Data

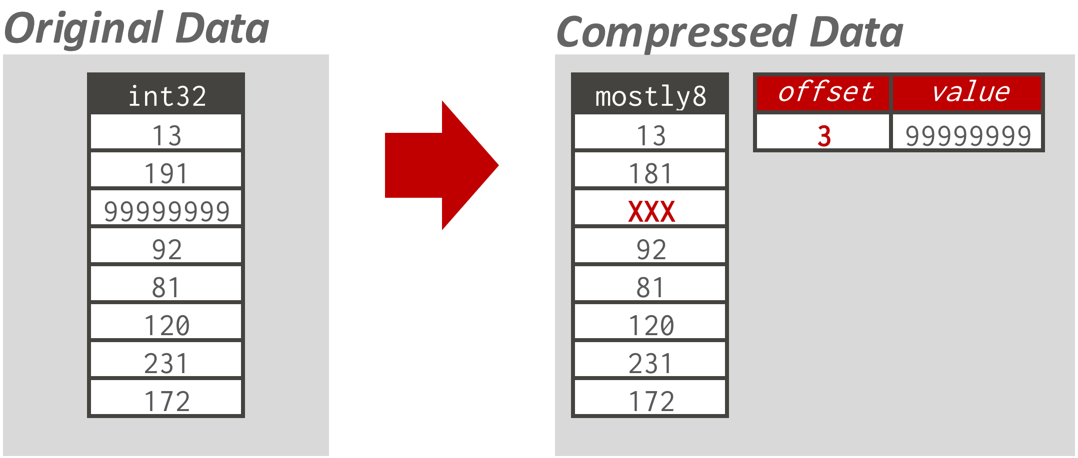
int32
13
191
99999999
92
81
120
231
172

Source: Redshift Documentation

Patching / Mostly Encoding

- A variation of bit packing when attribute's values are "mostly" less than the largest size, store them with the smaller data type.

→ The remaining values that cannot be compressed are stored in their raw form.



Source: Redshift Documentation

SCHOOL OF DATA SCIENCE

Patching / Mostly Encoding

- A variation of bit packing when attribute's values are "mostly" less than the largest size, store them with the smaller data type.

→ The remaining values that cannot be compressed are stored in their raw form.

Original:
8 × 32-bits = 256 bits

Original Data

int32
13
191
99999999
92
81
120
231
172

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

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

SCHOOL OF DATA SCIENCE

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:
8 × 8-bits =
64 bits

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:
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

SCHOOL OF DATA SCIENCE

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

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
);
```

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.
→ 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

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

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

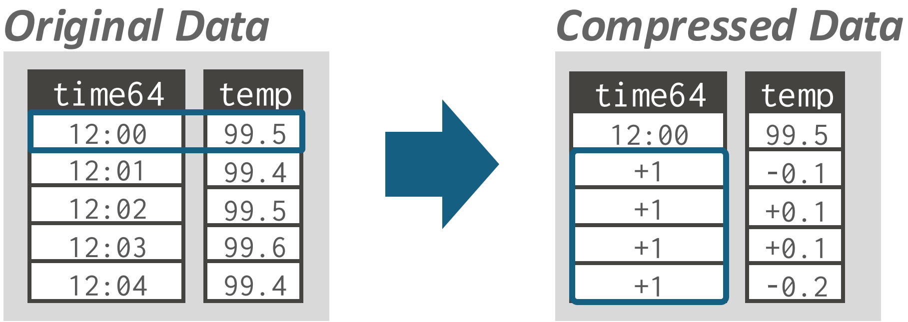
Compressed Data

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.

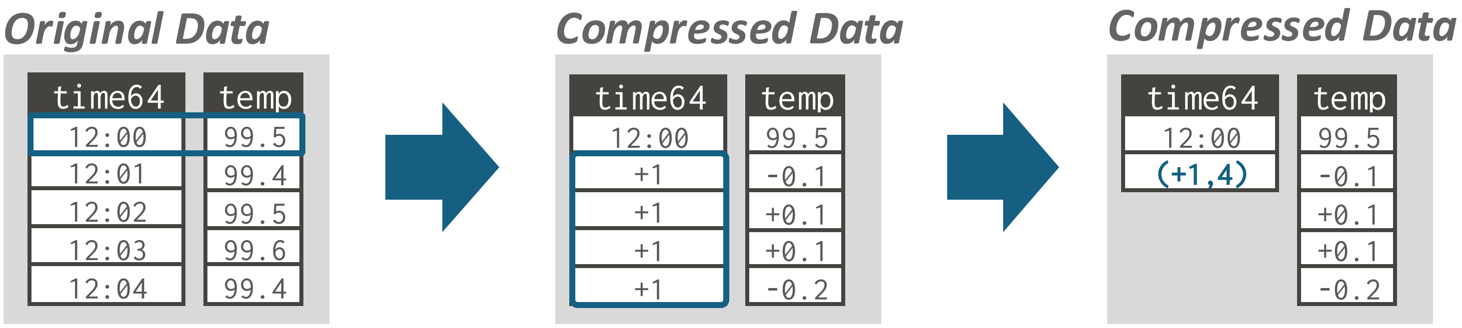
→ Store base value in-line or in a separate look-up table.
→ Combine with RLE to get even better compression ratios.



Delta Encoding

- Recording the difference between values that follow each other in the same column.

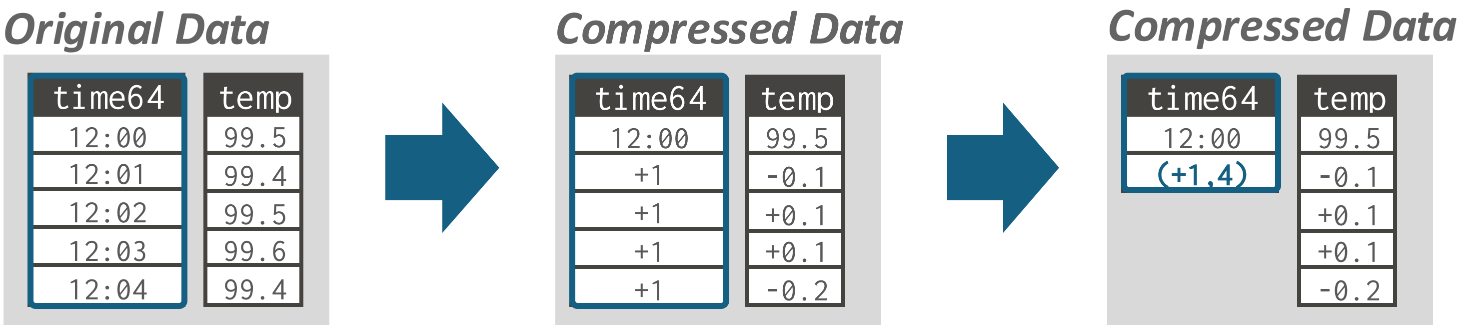
→ Store base value in-line or in a separate look-up table.
→ Combine with RLE to get even better compression ratios.



Delta Encoding

- Recording the difference between values that follow each other in the same column.

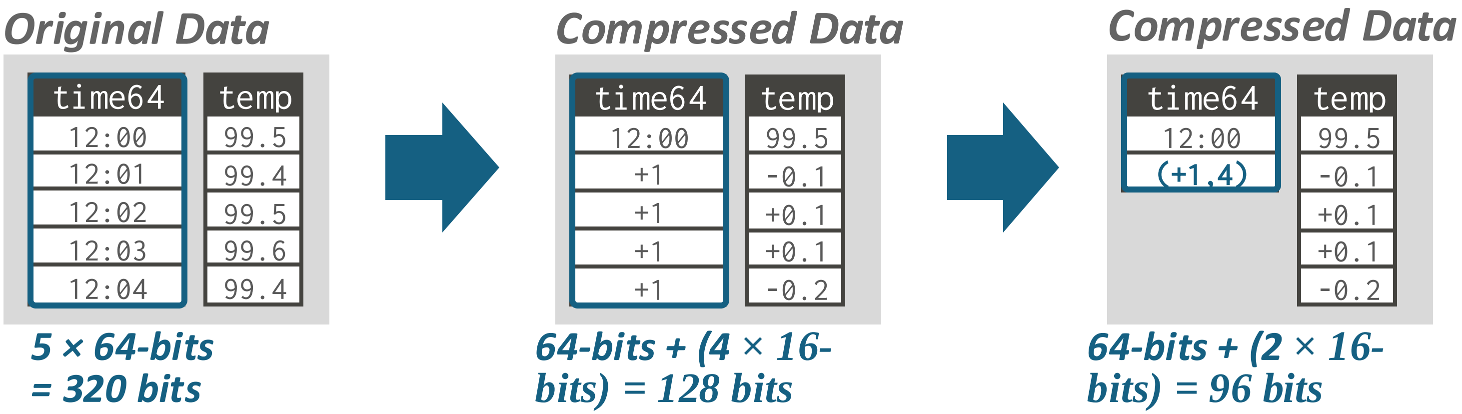
→ Store base value in-line or in a separate look-up table.
→ Combine with RLE to get even better compression ratios.



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.



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

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth

Dictionary: Example

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: Example

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: Example

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

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth

Compressed Data

name
10
20
30
40
20

Dictionary

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

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.

Original Data

name
Andrea
Prashanth
Andy
Matt
Prashanth

Dictionary: Order-Preserving

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

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

Dictionary: Order-Preserving

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

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: Order-Preserving

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

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

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: 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

Sorted Dictionary

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

Order-Preserving Encoding

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

SCHOOL OF DATA SCIENCE

Order-Preserving Encoding

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

Still must perform scan on column

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.

SCHOOL OF
DATA SCIENCE

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.

Original Data

name
Andrea
Wan
Andy
Matt

SCHOOL OF
DATA SCIENCE

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.

Original Data

name
Andrea
Wan
Andy
Matt

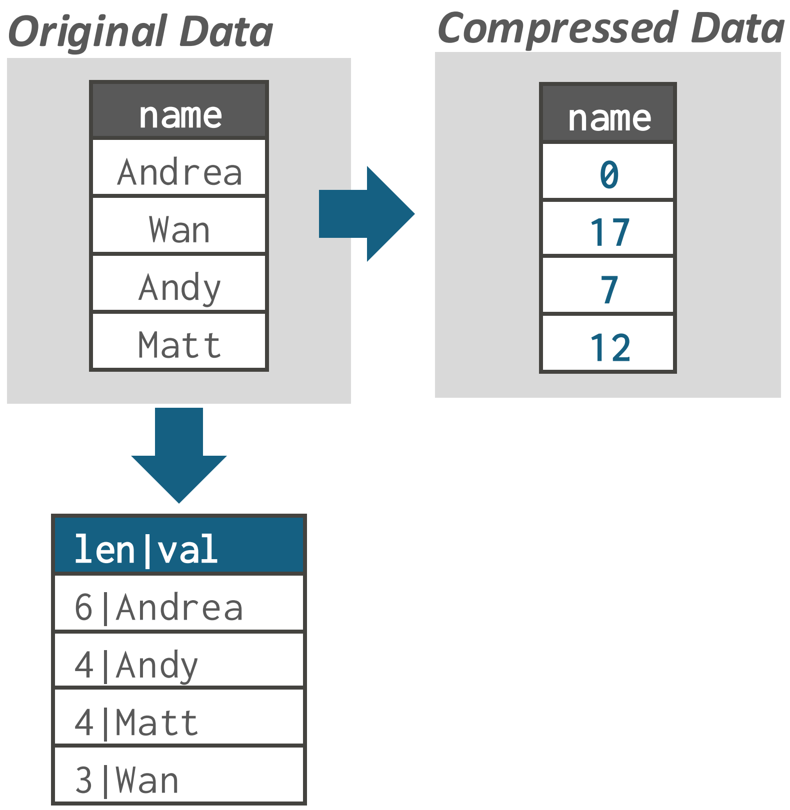
len|val
6|Andrea
4|Andy
4|Matt
3|Wan

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.

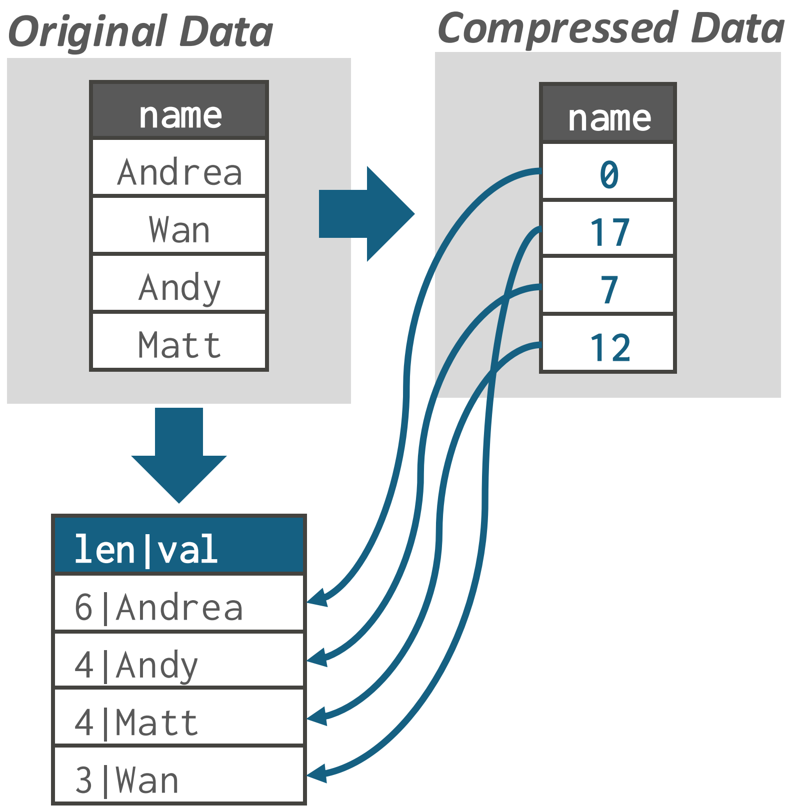


Dictionary: Array

- 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.

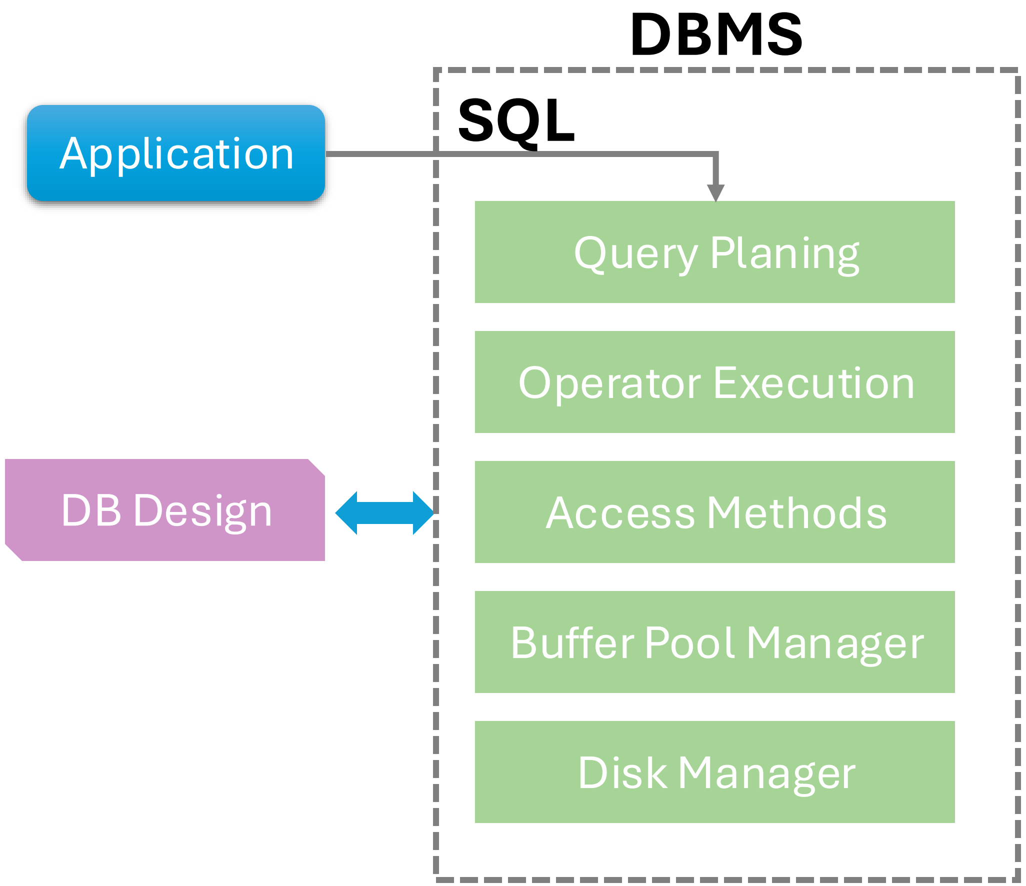
- Dictionary encoding is probably the most useful scheme because it does not require pre-sorting.

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



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