Carnegie Mellon University

DATABASE SYSTEMS

Database Storage: Files & Pages

LECTURE #02 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #0 is due Sunday Sept 7th @ 11:59pm

Homework #1 is due Sunday Sept 7th @ 11:59pm

LAST CLASS

We now understand what a database looks like at a logical level and how to write queries to read/write data (e.g., using SQL).

We will next learn how to build software that manages a database (i.e., a DBMS).

COURSE OUTLINE

Relational Databases

Storage

Query Execution

Query Planning / Optimization

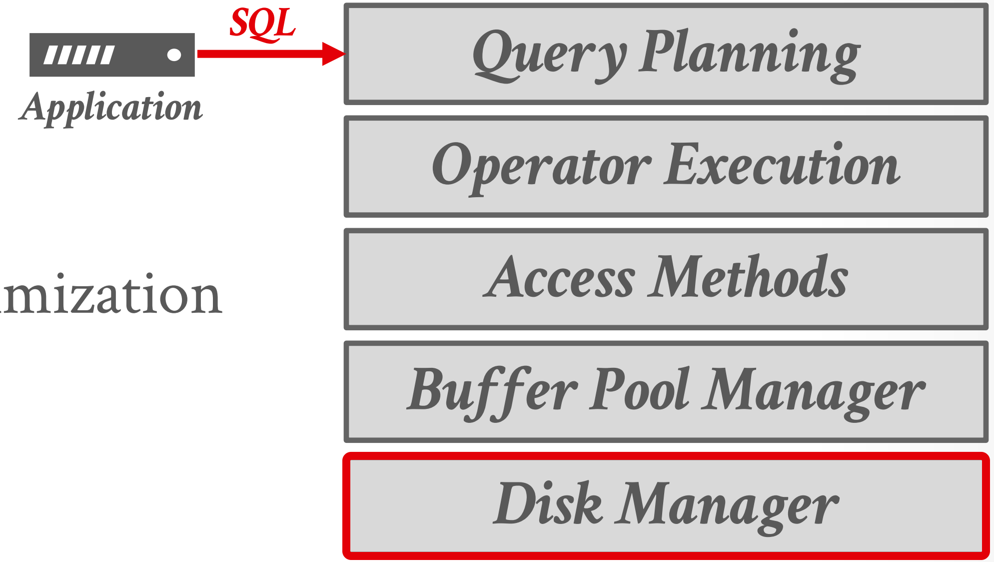
Concurrency Control

Database Recovery

Distributed Databases

COURSE OUTLINE

- Relational Databases
- Storage
- Query Execution
- Query Planning / Optimization
- Concurrency Control
- Database Recovery
- Distributed Databases



TODAY'S AGENDA

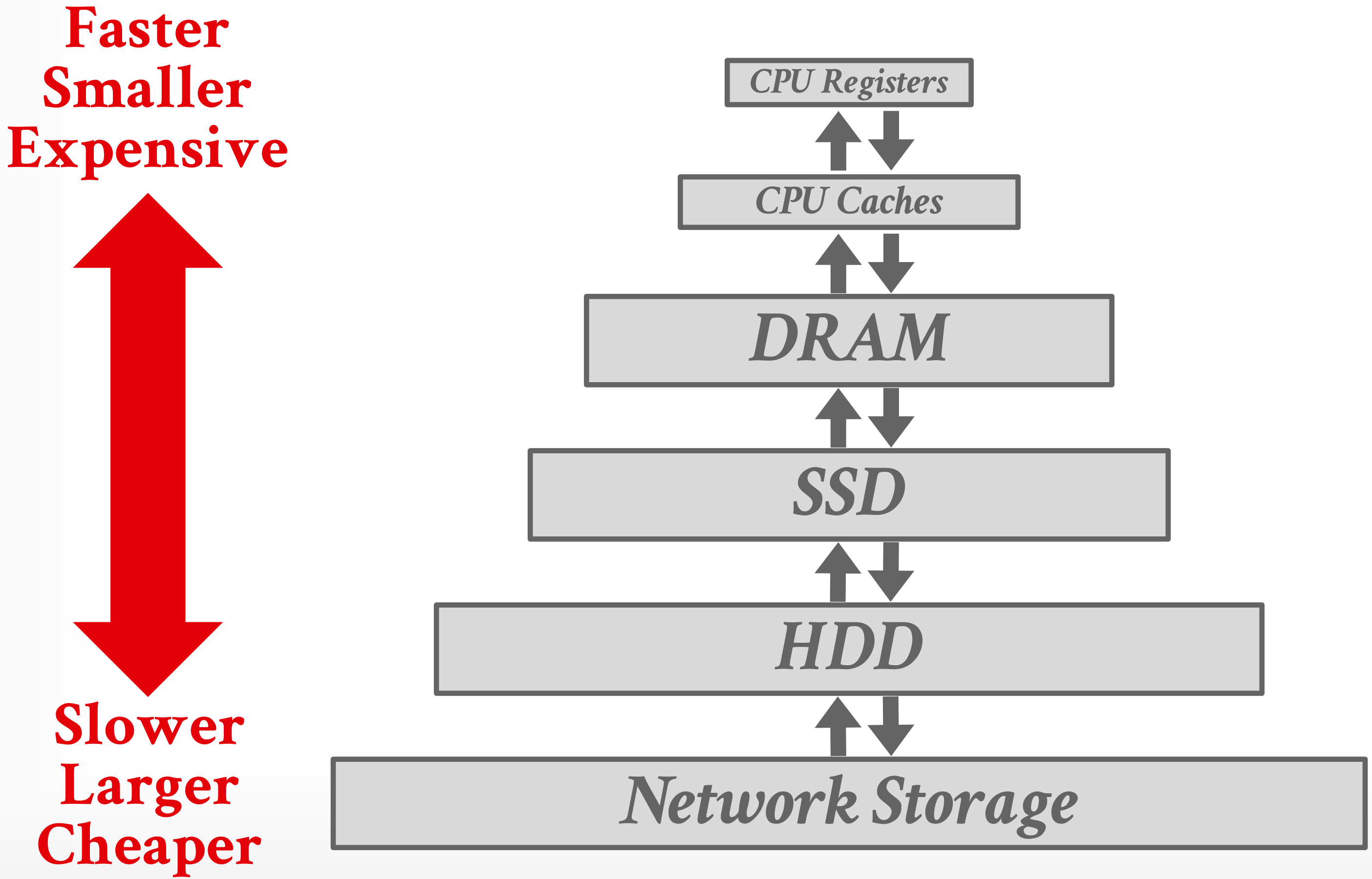
Background
File Storage
Page Layout
Tuple Layout

DISK-BASED ARCHITECTURE

The DBMS assumes that the primary storage location of the database is on non-volatile disk.

The DBMS's components manage the movement of data between non-volatile and volatile storage.

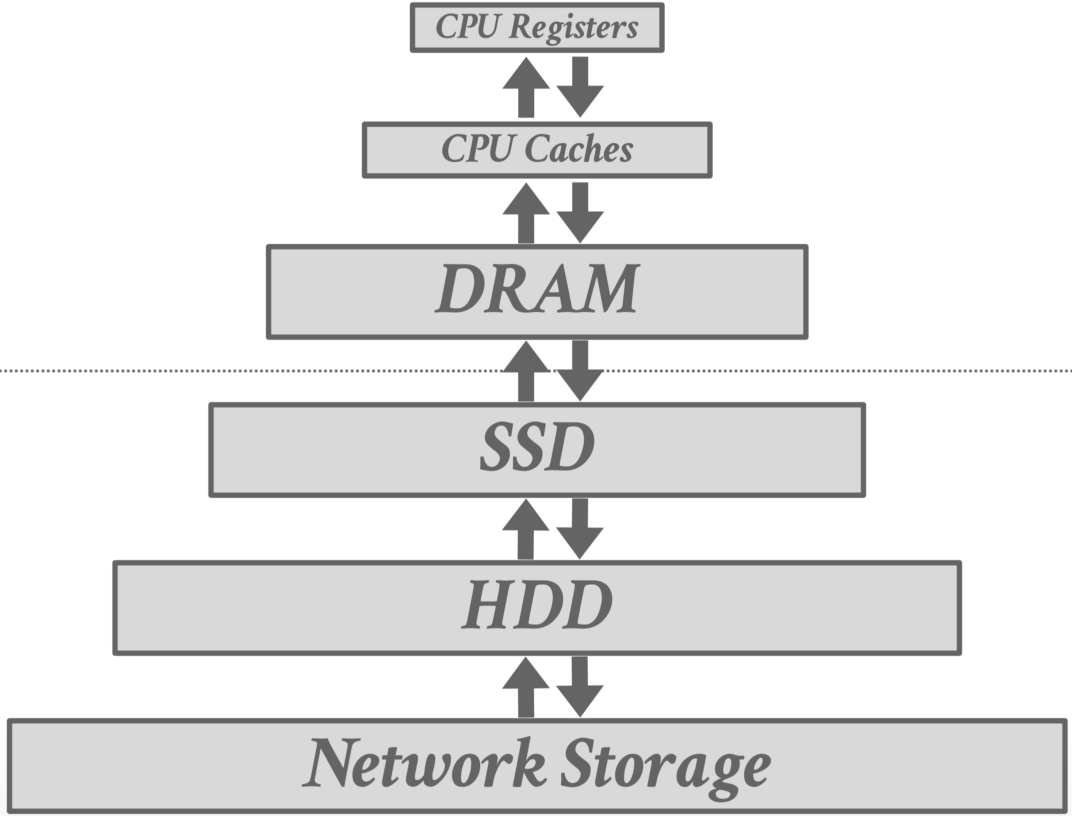
STORAGE HIERARCHY



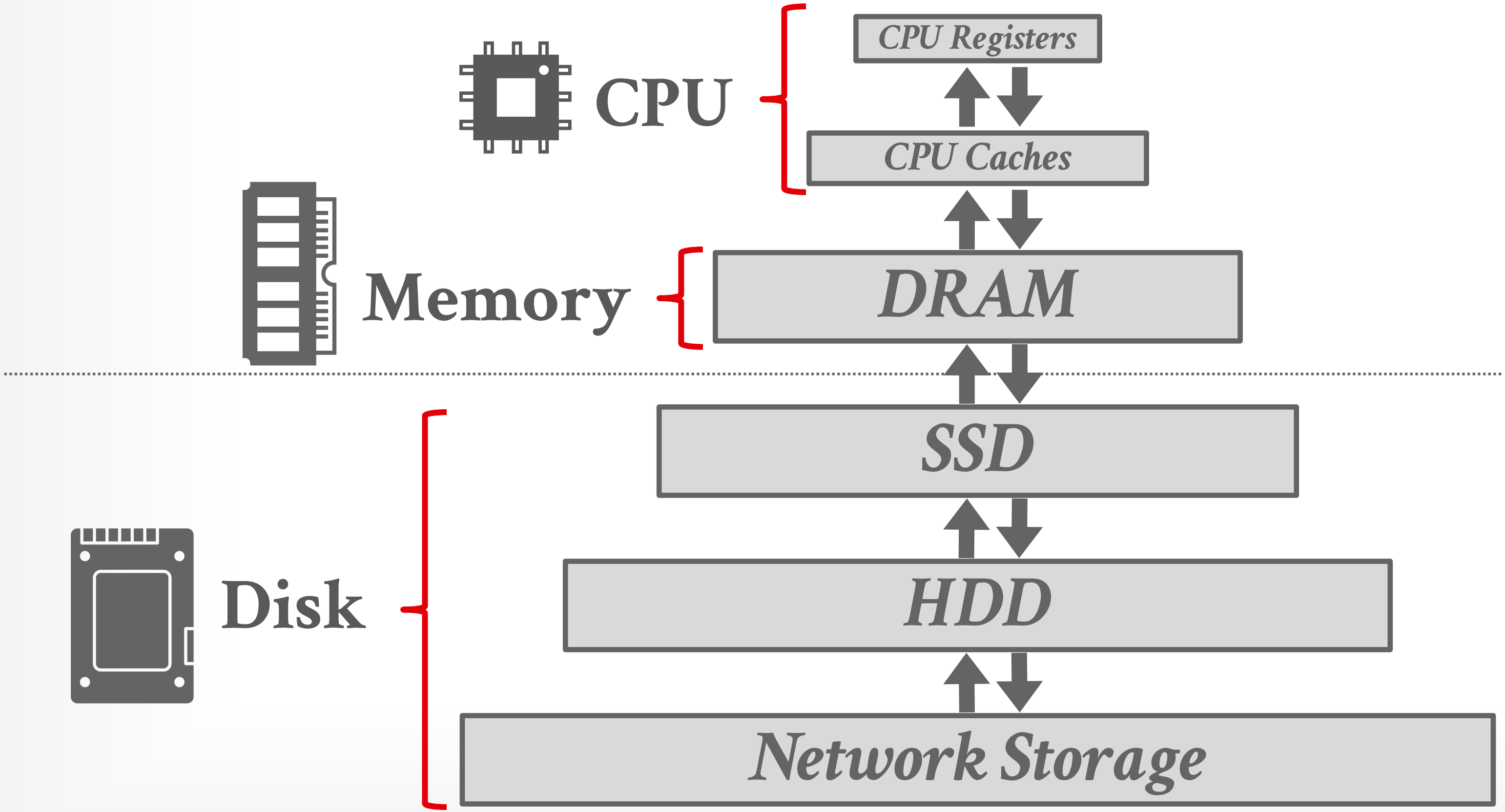
STORAGE HIERARCHY

Volatile
Random Access
Byte-Addressable

Non-Volatile
Sequential Access
Block-Addressable



STORAGE HIERARCHY



ACCESS TIMES

Latency Numbers Every Programmer Should Know

1 ns	L1 Cache Ref
4 ns	L2 Cache Ref
100 ns	DRAM
16,000 ns	SSD
2,000,000 ns	HDD
~50,000,000 ns	Network Storage
1,000,000,000 ns	Tape Archives

Source: Colin Scott

ACCESS TIMES

Latency Numbers Every Programmer Should Know

1 ns	L1 Cache Ref	⇐	**1 sec**
4 ns	L2 Cache Ref	⇐	**4 sec**
100 ns	DRAM	⇐	**100 sec**
16,000 ns	SSD	⇐	**4.4 hours**
2,000,000 ns	HDD	⇐	**3.3 weeks**
~50,000,000 ns	Network Storage	⇐	**1.5 years**
1,000,000,000 ns	Tape Archives	⇐	**31.7 years**

Source: Colin Scott

SEQUENTIAL VS. RANDOM ACCESS

Random access on non-volatile storage is almost always slower than sequential access.

→ Random I/O: 80–100 μs

→ Sequential I/O: 10–100 μs

DBMS will want to maximize sequential access.

→ Algorithms try to reduce number of writes to random pages so that data is stored in contiguous blocks.

→ Allocating multiple pages at the same time is called an extent.

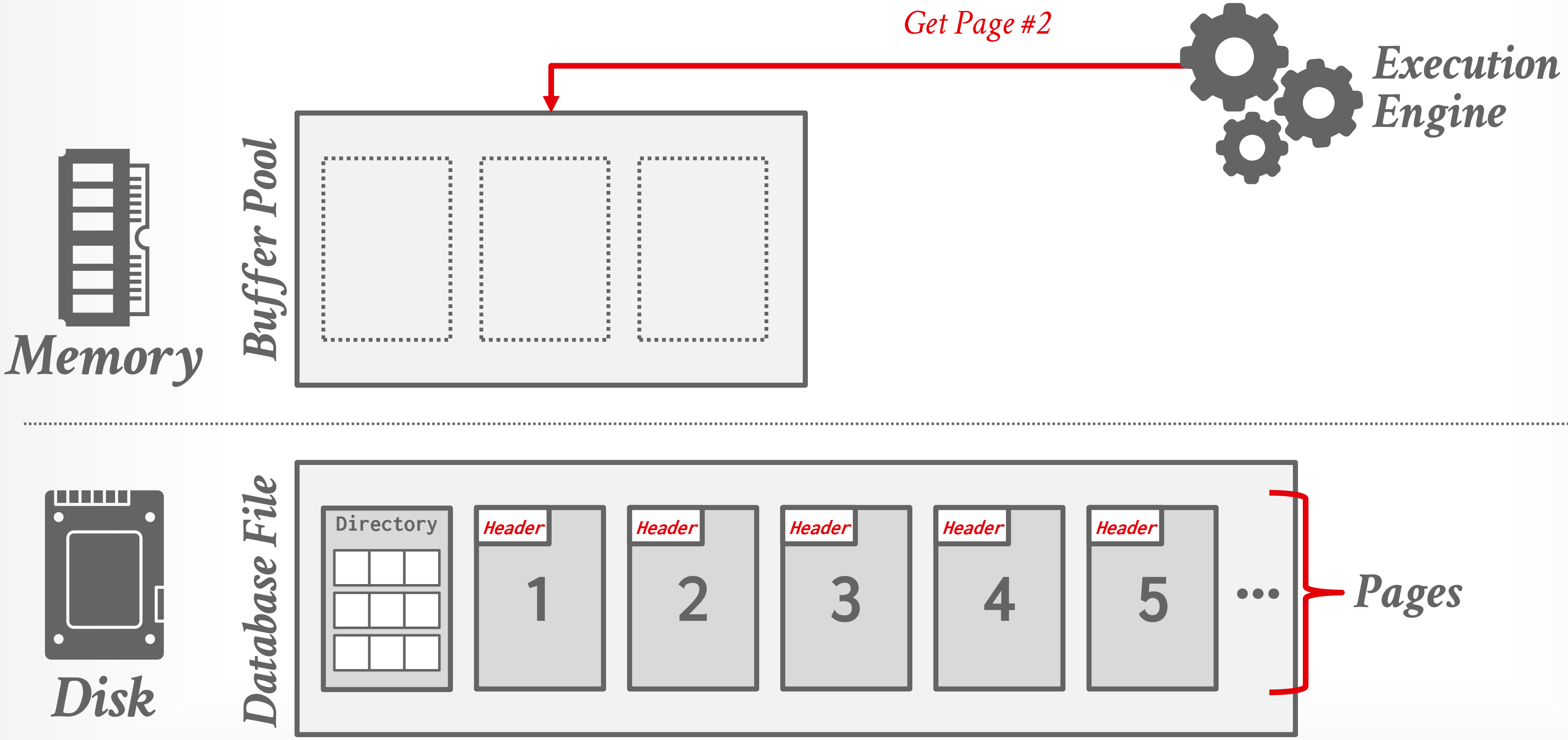
SYSTEM DESIGN GOALS

Allow the DBMS to manage databases that exceed the amount of memory available.

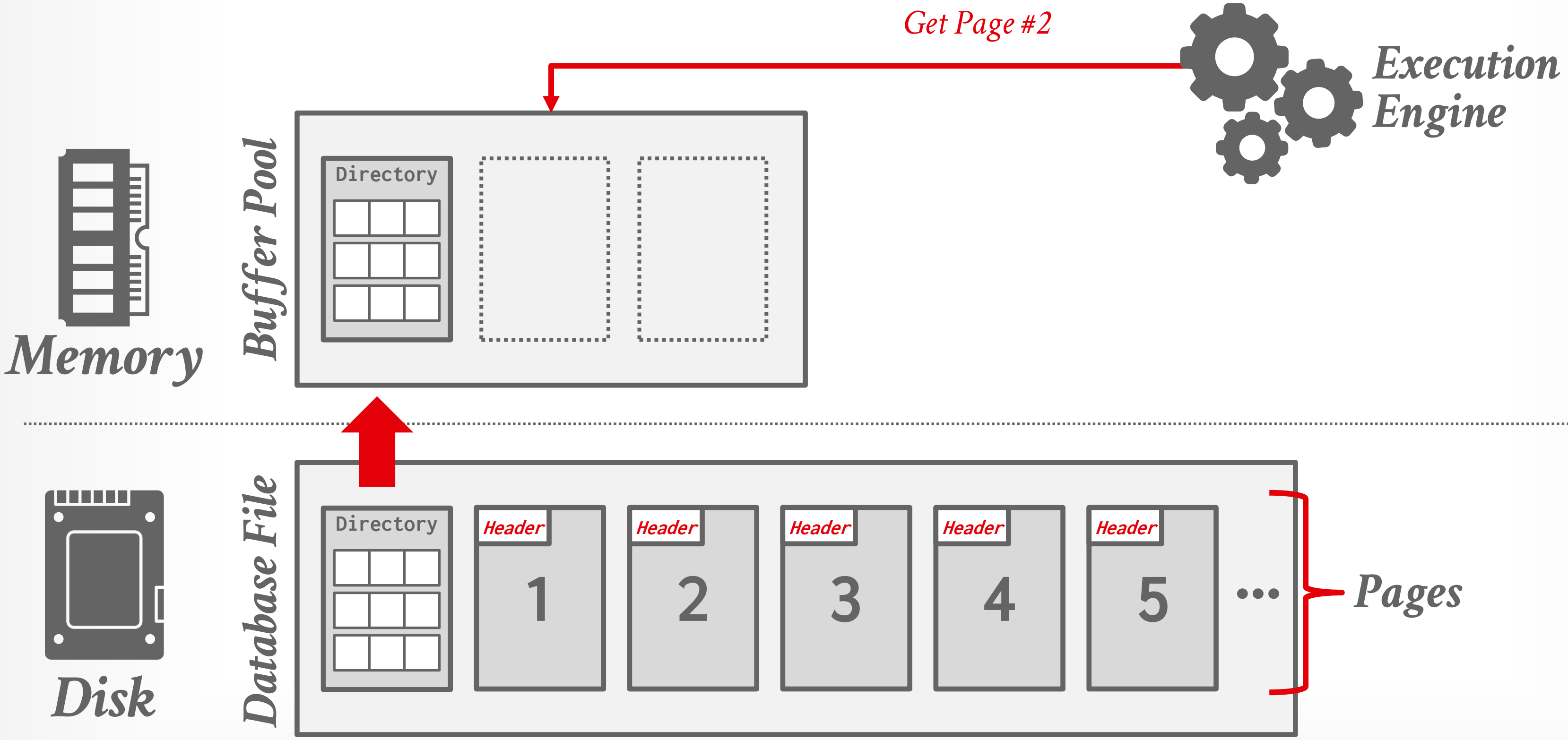
Reading/writing to disk is expensive, so it must be managed carefully to avoid large stalls and performance degradation.

Random access on disk is usually much slower than sequential access, so the DBMS will want to maximize sequential access.

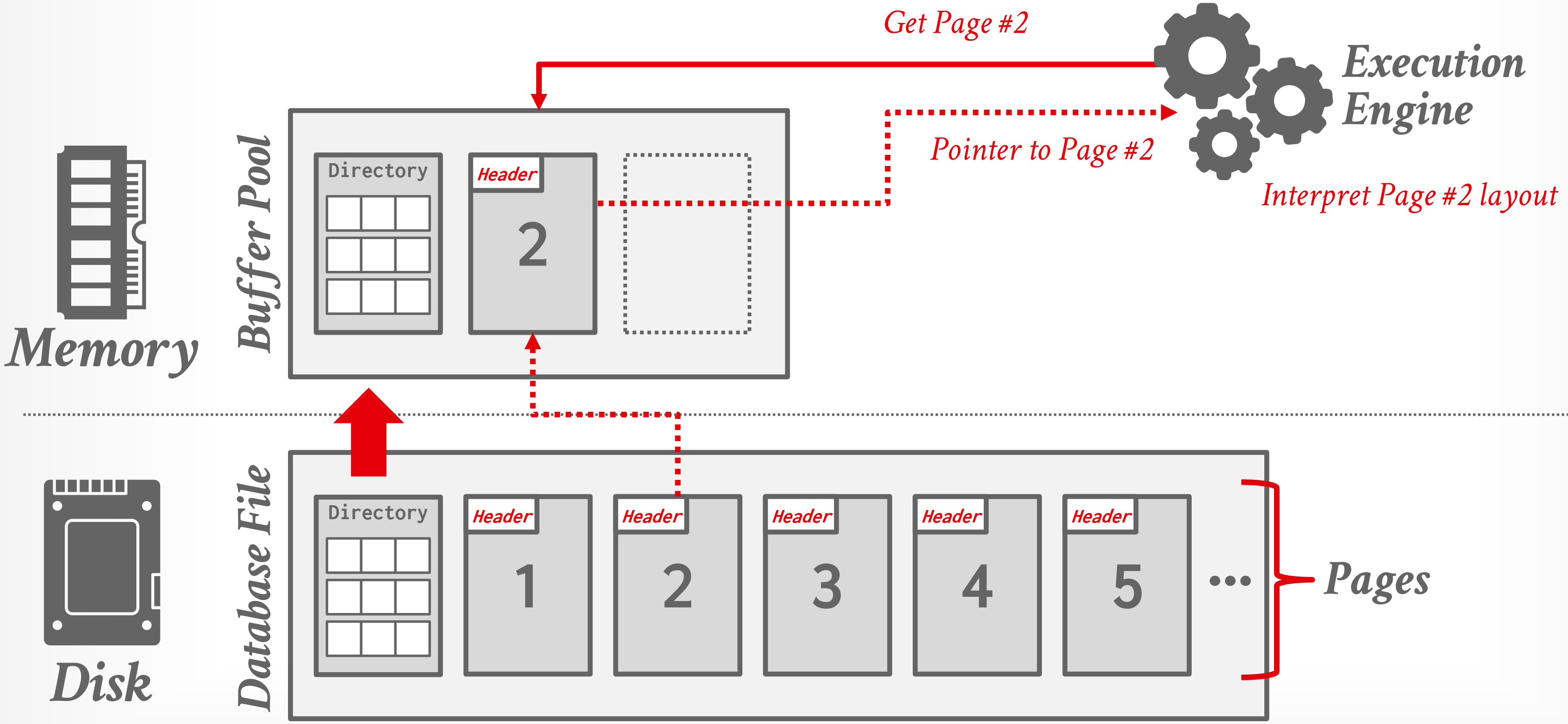
DISK-ORIENTED DBMS



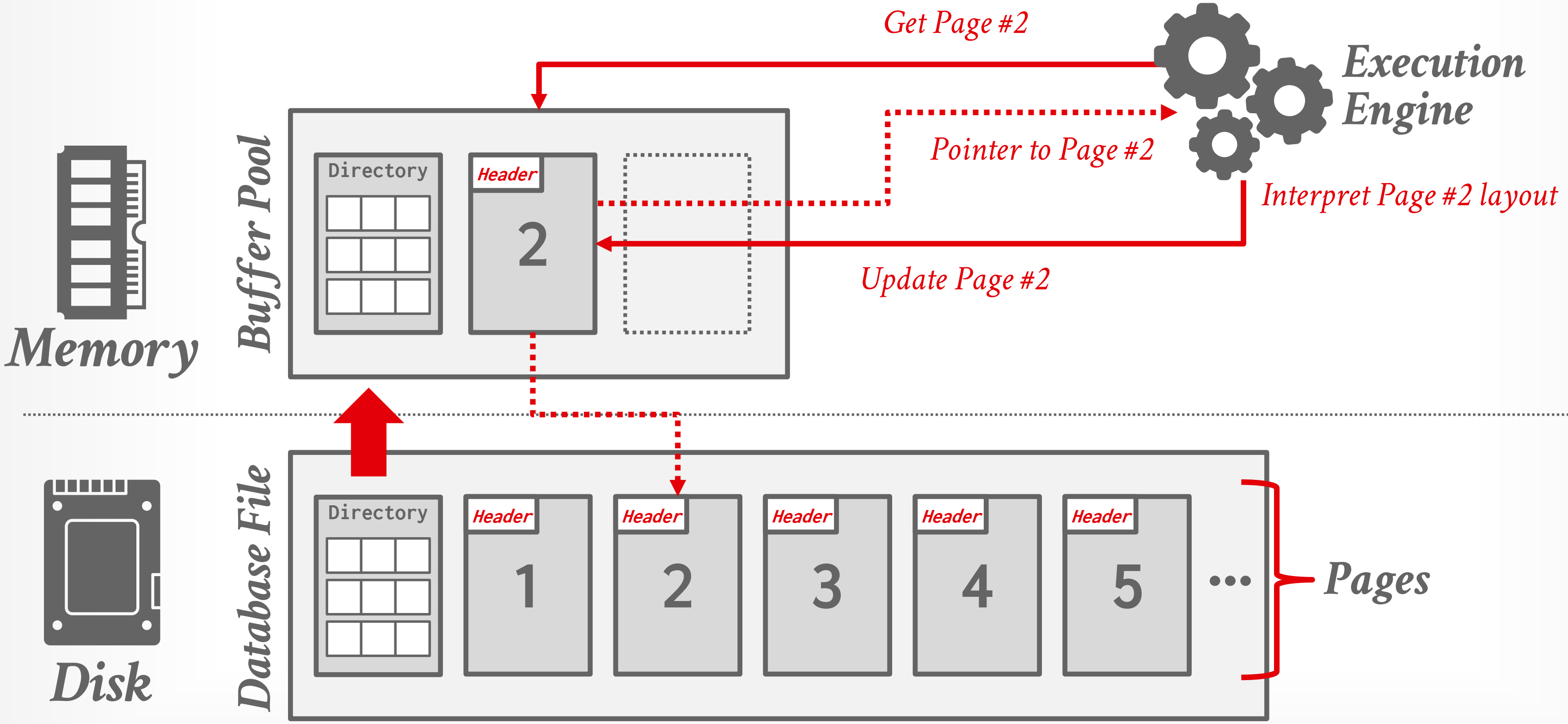
DISK-ORIENTED DBMS



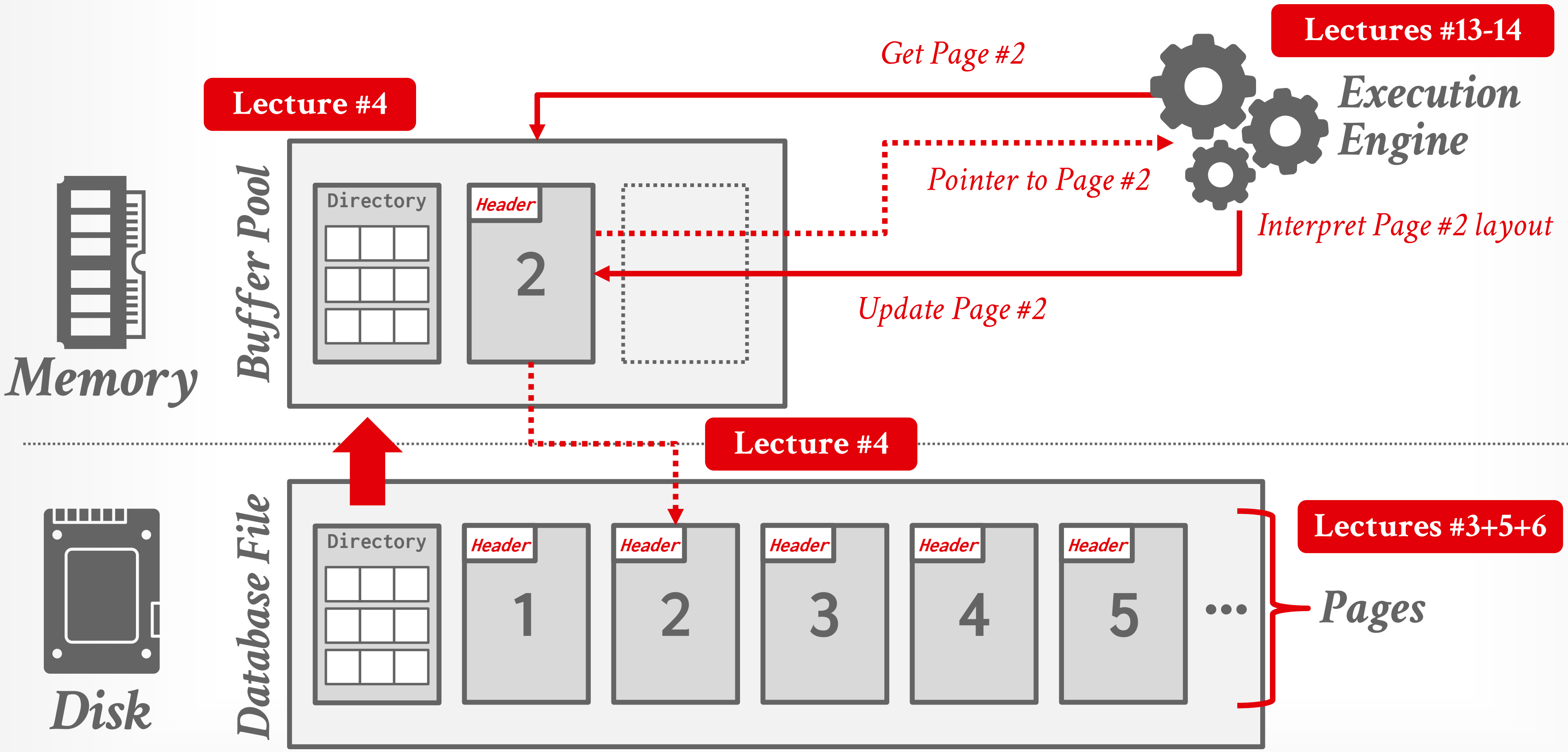
DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



DATABASE STORAGE

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

← Today

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

FILE STORAGE

The DBMS stores a database as one or more files on disk typically in a proprietary format.

→ The OS does not know anything about the contents of these files.
→ We will discuss portable file formats next week…

Early systems in the 1980s used custom filesystems on raw block storage.

→ Some enterprise DBMSs still do this (Oracle, Teradata).
→ Most newer DBMSs do not do this.

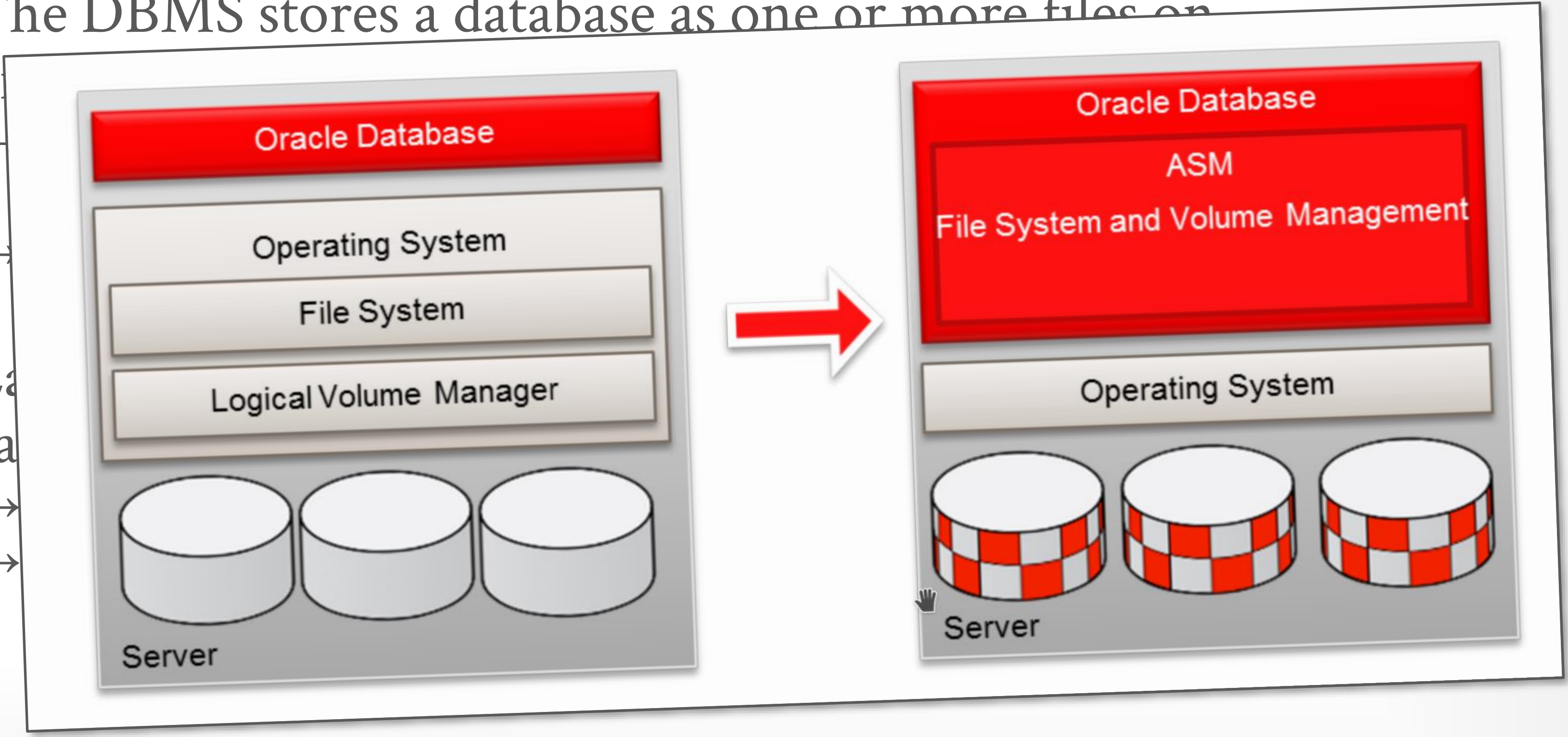
FILE STORAGE

The DBMS stores a database as one or more files on
d

→

→

Ea
ra
→
→



STORAGE MANAGER

The storage manager is responsible for maintaining a database's files.
→ Some do their own scheduling for reads and writes to improve spatial and temporal locality of pages.

It organizes the files as a collection of pages.
→ Tracks data read/written to pages.
→ Tracks the available space.

A DBMS typically does not maintain multiple copies of a page on disk.
→ Assume this happens above/below storage manager.

DATABASE PAGES

A page is a fixed-size block of data.

→ It can contain tuples, meta-data, indexes, log records…

→ Most systems do not mix page types.

→ Some systems require a page to be self-contained.

Each page is given a unique identifier (**page ID**).

→ A page ID could be unique per DBMS instance, per database, or per table.

→ The DBMS uses an indirection layer to map page IDs to physical locations.

DATABASE PAGES

There are three different notions of "pages" in a DBMS:
→ Hardware Page (usually 4KB)
→ OS Page (usually 4KB, x64 2MB/1GB)
→ Database Page (512B-32KB)

A hardware page is the largest block of data that the storage device can guarantee failsafe writes.

Default DB Page Sizes

4KB: SQLite, ORACLE, IBM DB2, RocksDB, WIREDTIGER

8KB: Microsoft SQL Server, INGRES, PostgreSQL, Informix

16KB: MySQL

DATABASE PAGES

Optimal database page size depends on environment, database contents, and expected workload.

DBMSs specializing in read-heavy workloads tend to have larger page sizes (1 MB or larger).

→ Fetching a single page brings in many tuples that are needed for a query.

DBMSs specializing in write-heavy workloads tend to have smaller page sizes (4-16 KB).

→ The system must write entire page to disk even if only a small portion of it is modified.

PAGE STORAGE ARCHITECTURE

Different DBMSs manage pages in files on disk in different ways.

→ Heap File Organization
→ Tree File Organization
→ Sequential / Sorted File Organization (ISAM)
→ Hashing File Organization

At this point in the hierarchy, we do <u>not</u> need to know anything about what is inside of the pages.

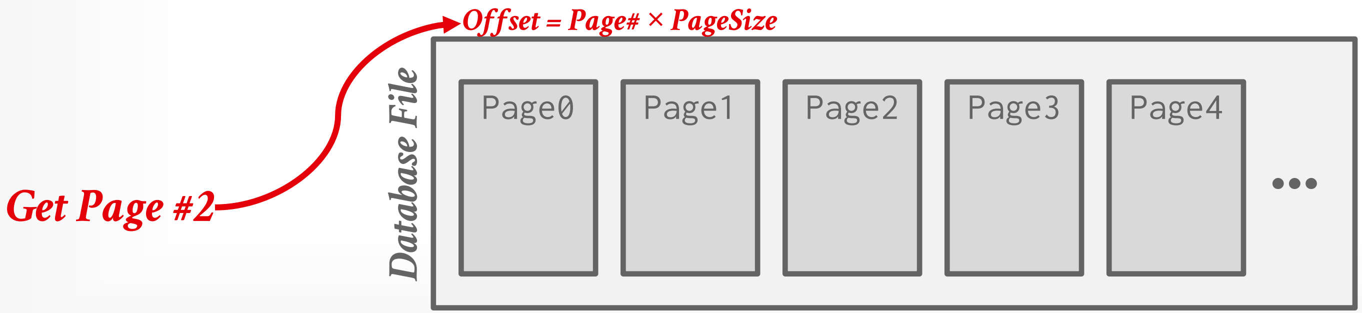
HEAP FILE

A heap file is an unordered collection of pages with tuples that are stored in random order.

→ Create / Get / Write / Delete Page

→ Must also support iterating over all pages.

Need additional meta-data to track location of files and free space availability.



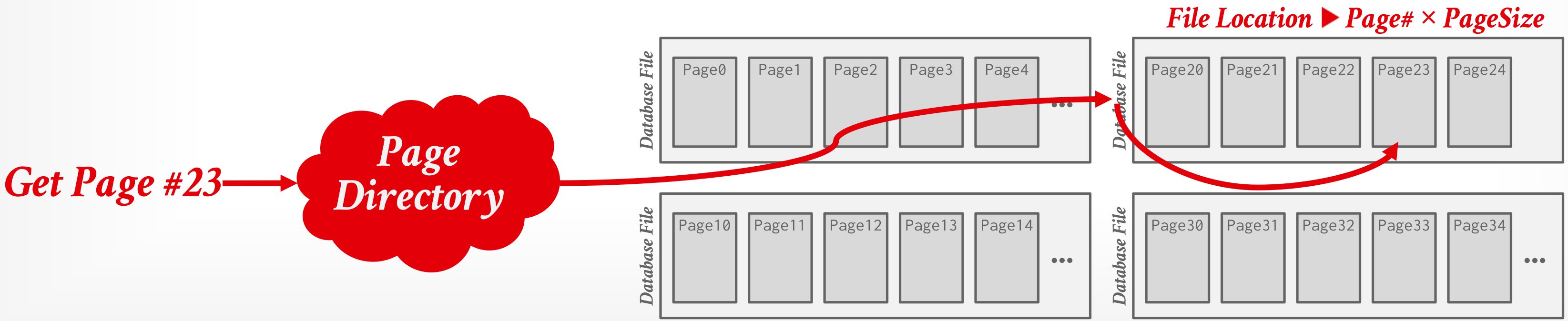
HEAP FILE

A heap file is an unordered collection of pages with tuples that are stored in random order.

→ Create / Get / Write / Delete Page

→ Must also support iterating over all pages.

Need additional meta-data to track location of files and free space availability.



HEAP FILE: PAGE DIRECTORY

The DBMS maintains special pages that tracks the location of data pages in the database files.

→ One entry per database object.

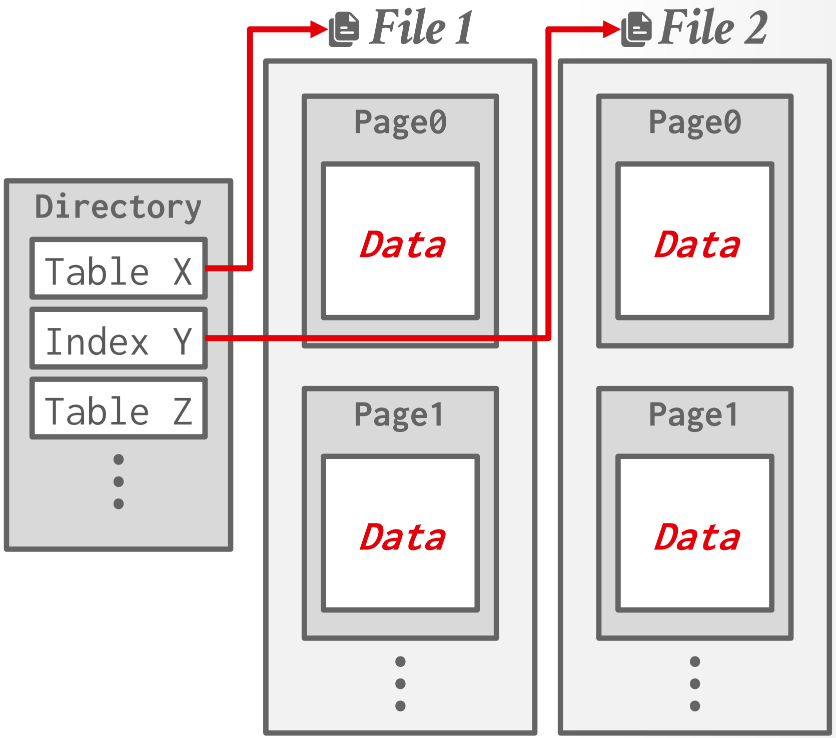
→ Must make sure that the directory pages are in sync with the data pages.

DBMS also keeps meta-data about pages' contents:

→ Amount of free space per page.

→ List of free / empty pages.

→ Page type (data vs. meta-data).



TODAY'S AGENDA

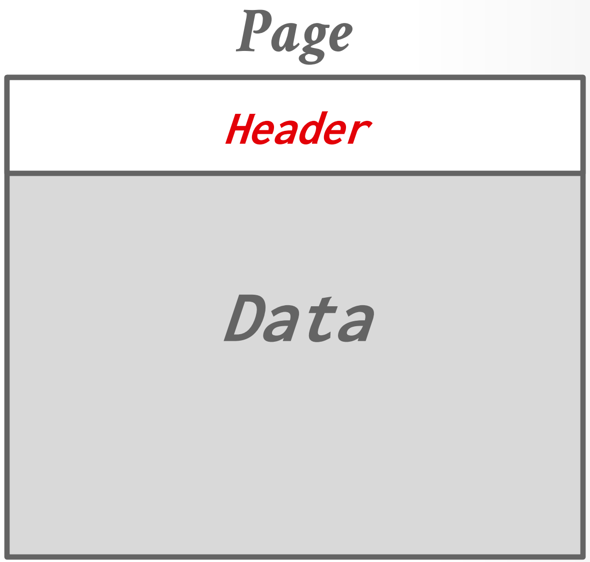
File Storage
Page Layout
Tuple Layout

PAGE HEADER

Every page contains a header of meta-data about the page's contents.

- → Page Size
- → Checksum
- → DBMS Version
- → Transaction Visibility
- → Compression / Encoding Meta-data
- → Schema Information
- → Data Summary / Sketches

Some systems require pages to be self-contained (e.g., Oracle).



PAGE LAYOUT

For any page storage architecture, we now need to decide how to organize the data inside of the page.

→ We are still assuming that we are only storing tuples in a row-oriented storage model.

→ We will also assume that an each tuple fits in a single page.

Approach #1: Tuple-oriented Storage

Approach #2: Log-structured Storage

Approach #3: Index-organized Storage

PAGE LAYOUT

For any page storage architecture, we now need to decide how to organize the data inside of the page.

→ We are still assuming that we are only storing tuples in a row-oriented storage model. Lecture #6

→ We will also assume that an each tuple fits in a single page.

Approach #1: Tuple-oriented Storage ← Today

Approach #2: Log-structured Storage

Approach #3: Index-organized Storage

PAGE LAYOUT

For any page storage architecture, we now need to decide how to organize the data inside of the page.

→ We are still assuming that we are only storing tuples in a row-oriented storage model.

Lecture #6

→ We will also assume that an each tuple fits in a single page.

Approach #1: Tuple-oriented Storage

Approach #2: Log-structured Storage

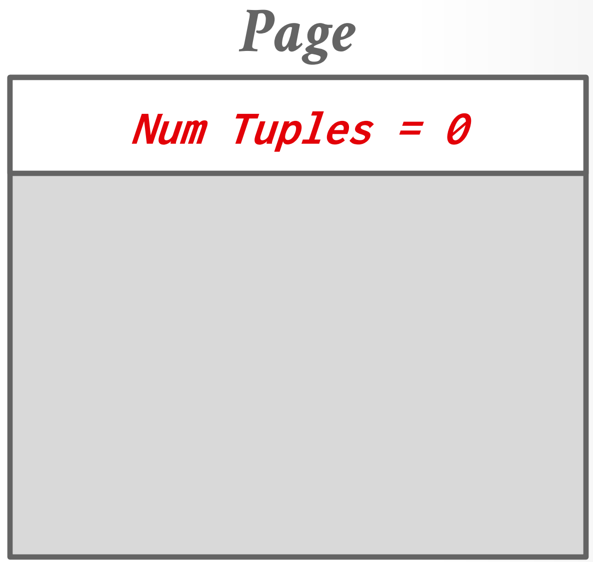
Approach #3: Index-organized Storage

Lecture #5

TUPLE-ORIENTED STORAGE

How to store tuples in a page?

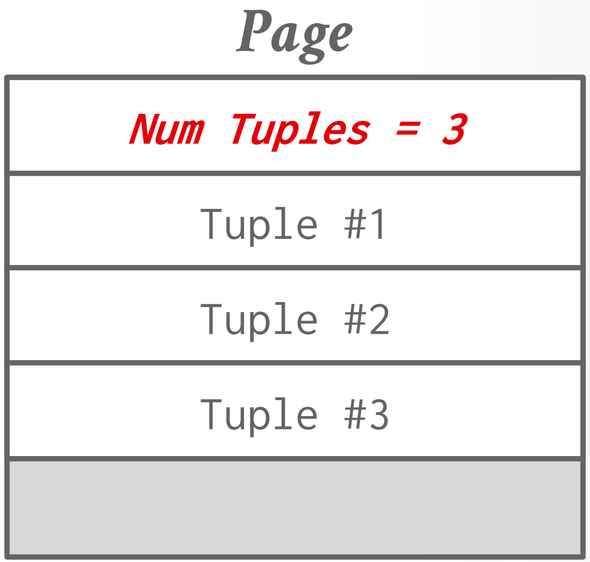
Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.



TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

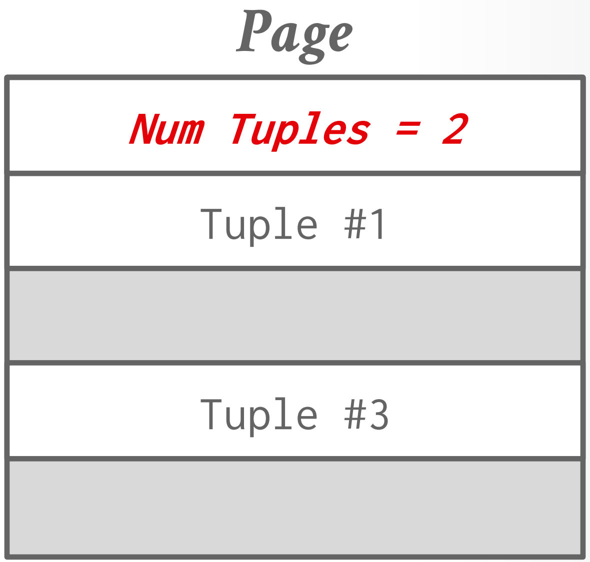


TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

→ What happens if we delete a tuple?

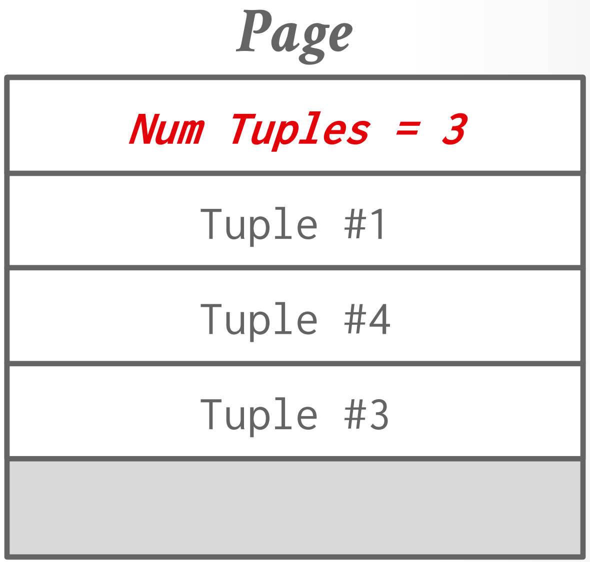


TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

→ What happens if we delete a tuple?



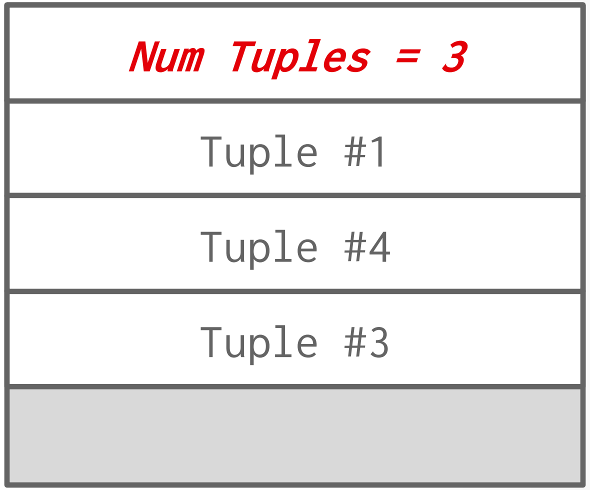
TUPLE-ORIENTED STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

→ What happens if we delete a tuple?
→ What happens if we have a variable-length attribute?

Page

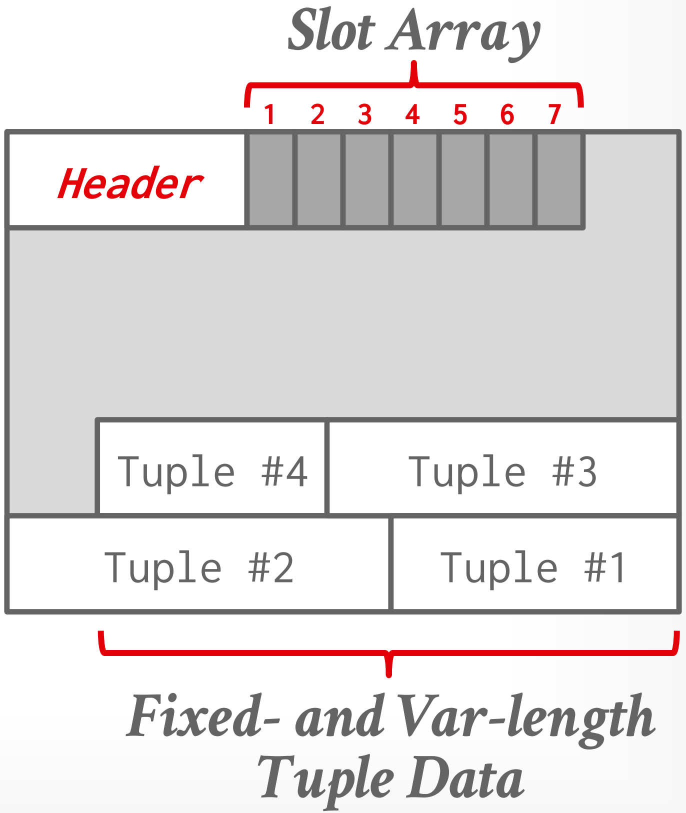


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.



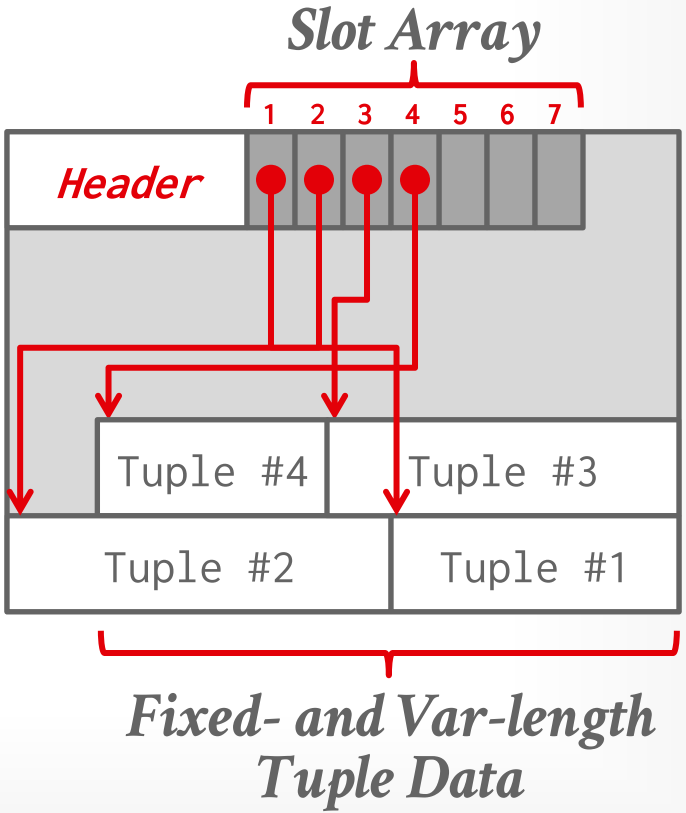
SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

→ The # of used slots
→ The offset of the starting location of the last slot used.

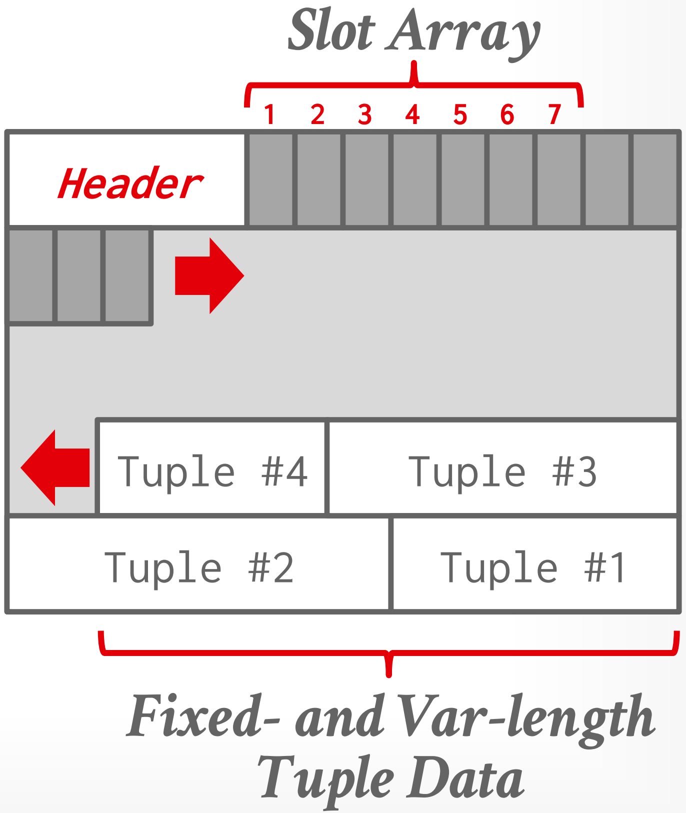


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

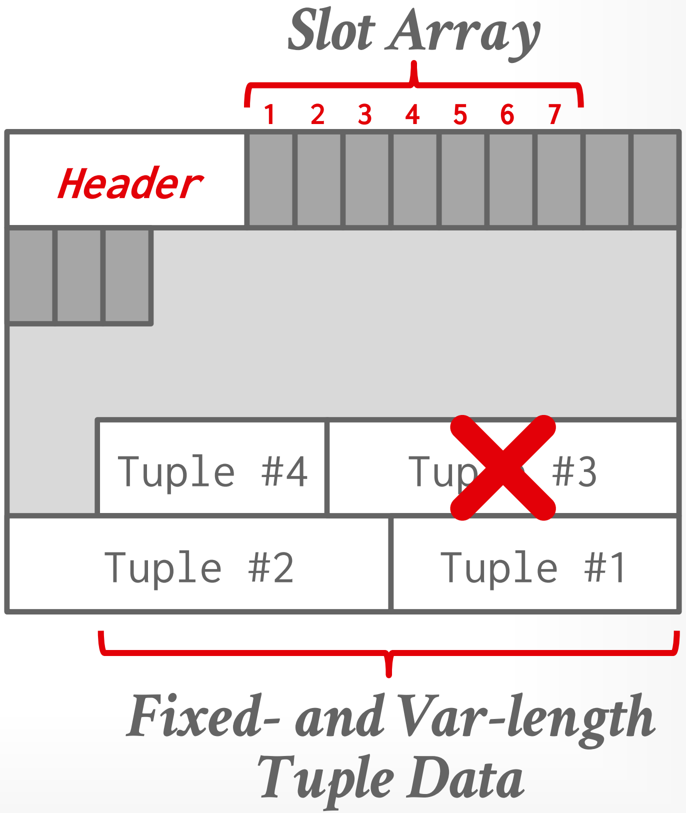


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

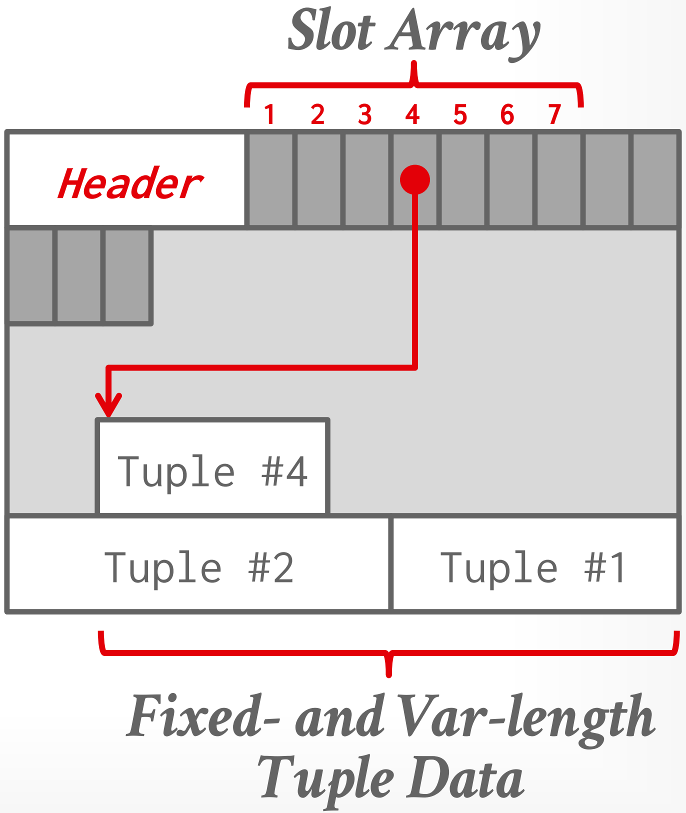


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.

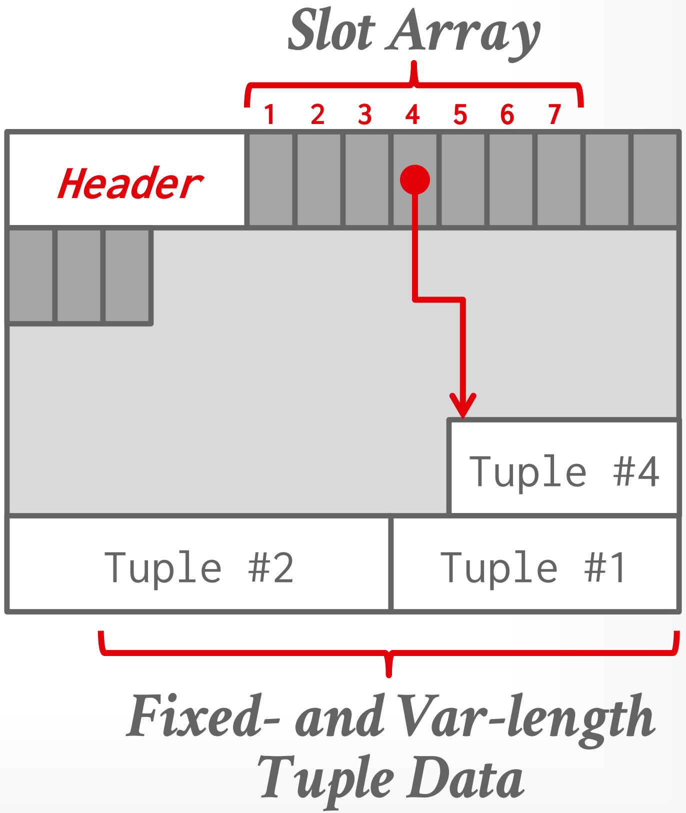


SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:
→ The # of used slots
→ The offset of the starting location of the last slot used.



RECORD IDS

The DBMS assigns each logical tuple a unique **record identifier** that represents its physical location in the database.

→ Example: File Id, Page Id, Slot #
→ Most DBMSs do not store ids in tuple.
→ SQLite uses **ROWID** as the true primary key and stores them as a hidden attribute.

Applications should never rely on these IDs to mean anything.

Record Id Sizes

INGRES	TID	*4-bytes*
PostgreSQL	CTID	*6-bytes*
SQLite	ROWID	*8-bytes*
Microsoft SQL Server	%%physloc%%	*8-bytes*
Firebird	RDB$DB_KEY	*8-bytes*
ORACLE	ROWID	*10-bytes*

TODAY'S AGENDA

File Storage
Page Layout
Tuple Layout

TUPLE LAYOUT

A tuple is essentially a sequence of bytes prefixed with a **header** that contains meta-data about it.

It is the job of the DBMS to interpret those bytes into attribute types and values.

The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.

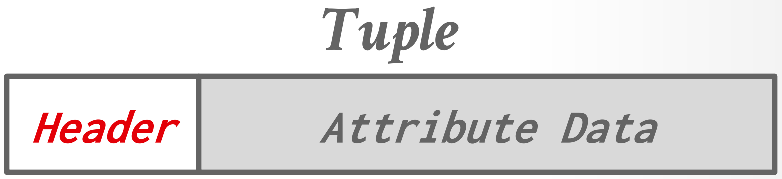
TUPLE HEADER

Each tuple is prefixed with a header that contains meta-data about it.

→ Visibility info (concurrency control)

→ Bit Map for **NULL** values.

We do not need to store meta-data about the schema.



TUPLE DATA

Attributes are typically stored in the order that you specify them when you create the table.

This is done for software engineering reasons (i.e., simplicity).

However, it might be more efficient to lay them out differently.

Tuple

Header	a	b	c	d	e

```
CREATE TABLE foo (
  a INT PRIMARY KEY,
  b INT NOT NULL,
  c INT,
  d DOUBLE,
  e FLOAT
);
```

DATA LAYOUT

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

unsigned char[]

Header	id	value

DATA LAYOUT

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

unsigned char[]

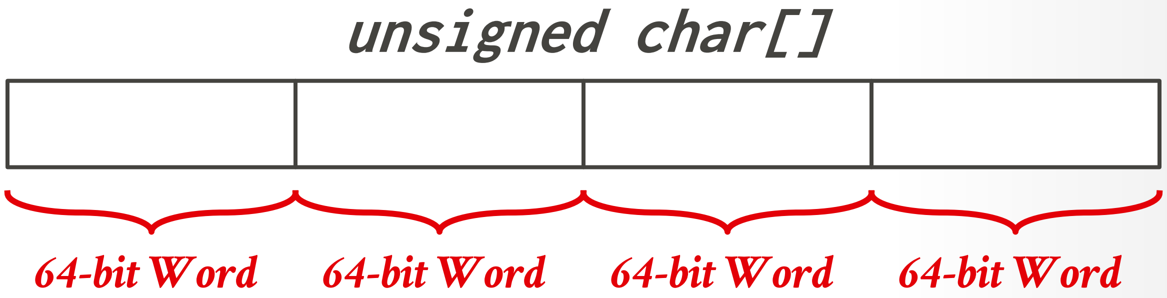
Header	id	value

```
reinterpret_cast<int32_t*>(address)
```

WORD-ALIGNED TUPLES

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

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

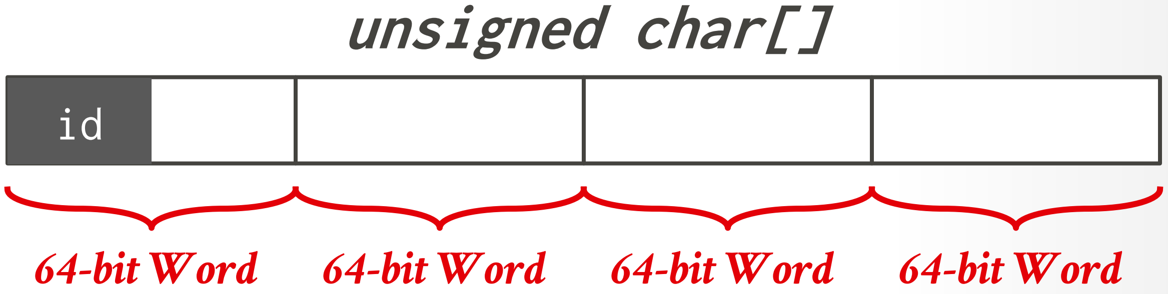


WORD-ALIGNED TUPLES

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

32-bits

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```



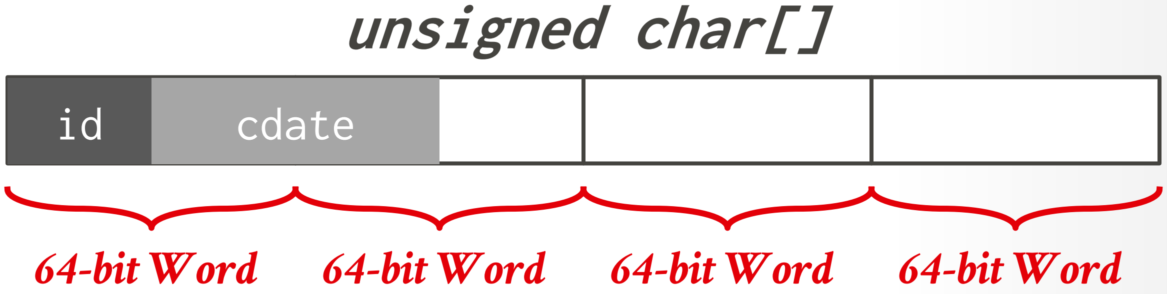
WORD-ALIGNED TUPLES

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

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

32-bits

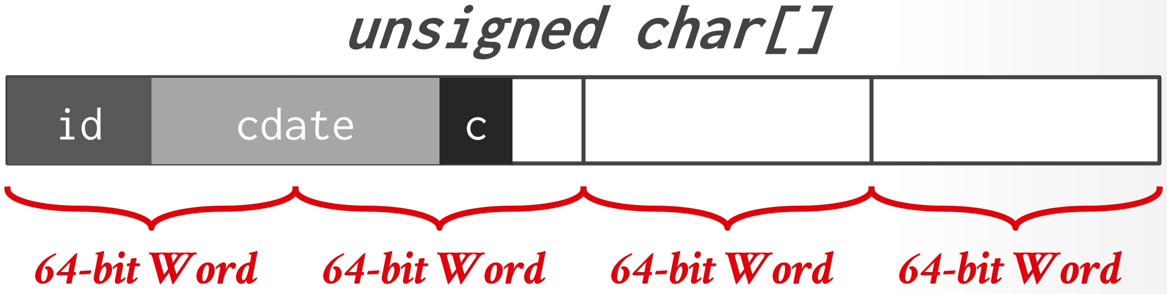
64-bits



WORD-ALIGNED TUPLES

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

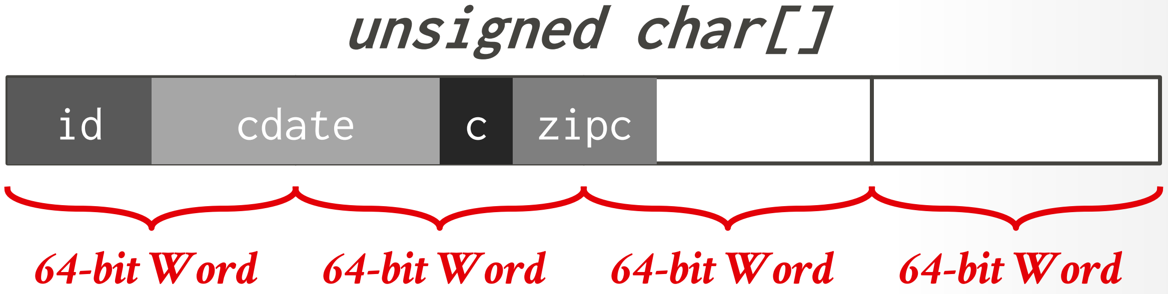
```
        CREATE TABLE foo (
32-bits   id INT PRIMARY KEY,
64-bits   cdate TIMESTAMP,
16-bits   color CHAR(2),
          zipcode INT
);
```



WORD-ALIGNED TUPLES

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

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

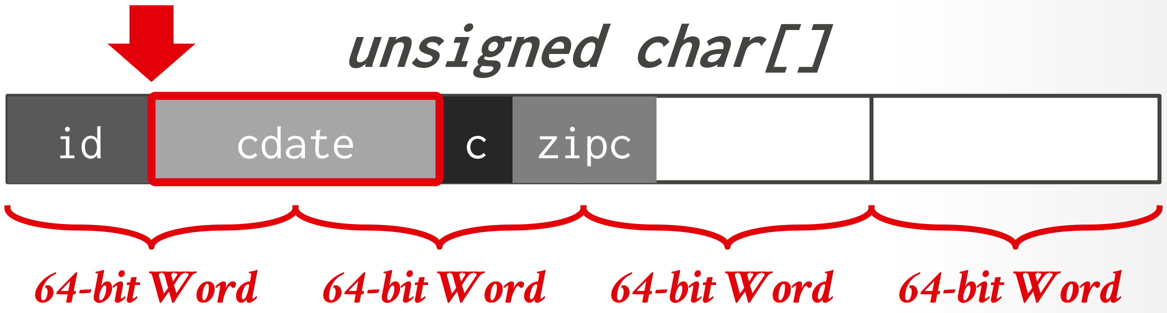


WORD-ALIGNED TUPLES

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

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

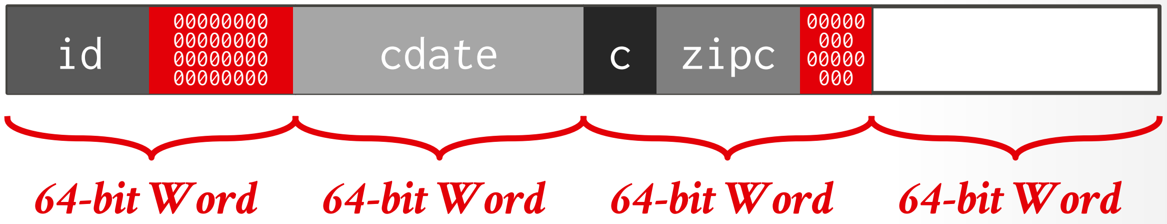
32-bits
64-bits
16-bits
32-bits



WORD-ALIGNMENT: PADDING

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

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

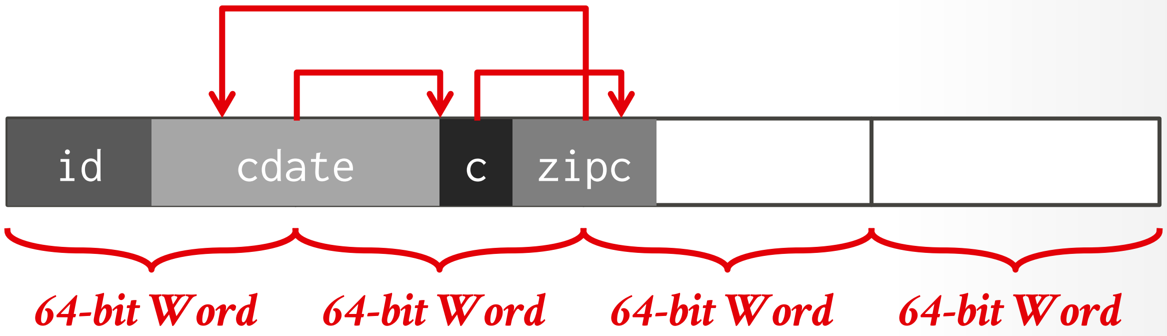


WORD-ALIGNMENT: REORDERING

Switch the order of attributes in the tuples' physical layout to make sure they are aligned.

→ May still have to use padding to fill remaining space.

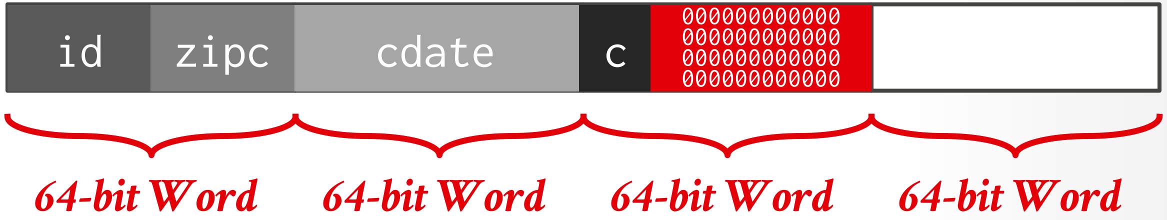
```
CREATE TABLE foo (
  id INT PRIMARY KEY,       -- 32-bits
  cdate TIMESTAMP,          -- 64-bits
  color CHAR(2),            -- 16-bits
  zipcode INT               -- 32-bits
);
```



WORD-ALIGNMENT: REORDERING

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

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



DATA REPRESENTATION

INTEGER/BIGINT/SMALLINT/TINYINT
→ Same as in C/C++.

FLOAT/REAL vs. **NUMERIC/DECIMAL**
→ IEEE-754 Standard / Fixed-point Decimals.

VARCHAR/VARBINARY/TEXT/BLOB
→ Header with length, followed by data bytes **OR** pointer to another page/offset with data.
→ Need to worry about collations / sorting.

TIME/DATE/TIMESTAMP/INTERVAL
→ 32/64-bit integer of (micro/milli)-seconds since Unix epoch (January 1st, 1970).

VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the "native" C/C++ types.

Store directly as specified by IEEE-754.
→ Example: **FLOAT**, **REAL**/**DOUBLE**

These types are typically faster than fixed precision numbers because CPU ISA's (Xeon, Arm) have instructions / registers to support them.

But they do not guarantee exact values…

VARIABLE PRECISION NUMBERS

Rounding Example

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %f\n", x+y);
    printf("0.3 = %f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

VARIABLE PRECISION NUMBERS

Rounding Example


```
#include <stdio.h>

in
}
```


```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

```
x+y = 0.30000001192092895508
0.3 = 0.29999999999999998890
```

FIXED PRECISION NUMBERS

Numeric data types with (potentially) arbitrary precision and scale. Used when rounding errors are unacceptable.

→ Example: **NUMERIC**, **DECIMAL**

Many different implementations.

→ Example: Store in an exact, variable-length binary representation with additional meta-data.

→ Can be less expensive if the DBMS does not provide arbitrary precision (e.g., decimal point can be in a different position per value).

POSTGRES: NUMERIC

of Digits

Weight of 1st Digit

Scale Factor

Positive/Negative/NaN

Digit Storage

```
typedef unsigned char NumericDigit;
typedef struct {
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
```

POSTGRES: NUMERIC

of Digits

Weight of 1st Digit

Scale Factor

Positive/Negative/NaN

Digit Storage

```
typedef unsigned char NumericDigit;
typedef struct {
  int ndigits;
  int weight;
  int scale;
  int sign;
  NumericDigit *digits;
} numeric;
```

...

Weight of...

Sca...

Positive/Negat...

Digi...

NumericDigit;

```
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * ----------
```

NULL DATA TYPES

Choice #1: Null Column Bitmap Header

→ Store a bitmap in a centralized header that specifies what attributes are null.
→ This is the most common approach in row-stores.

Choice #2: Special Values

→ Designate a placeholder value to represent **NULL** for a data type (e.g., **INT32_MIN**). More common in column-stores.

Don't Do This!

Choice #3: Per Attribute Null Flag

→ Store a flag that marks that a value is null.
→ Must use more space than just a single bit because this messes up with word alignment.

NULL DATA TYP

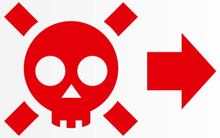
Choice #1: Null Column Bitmap

→ Store a bitmap in a centralized header t attributes are null.

→ This is the most common approach in

Choice #2: Special Values

→ Designate a placeholder value to repre (e.g., **INT32_MIN**). More common in c

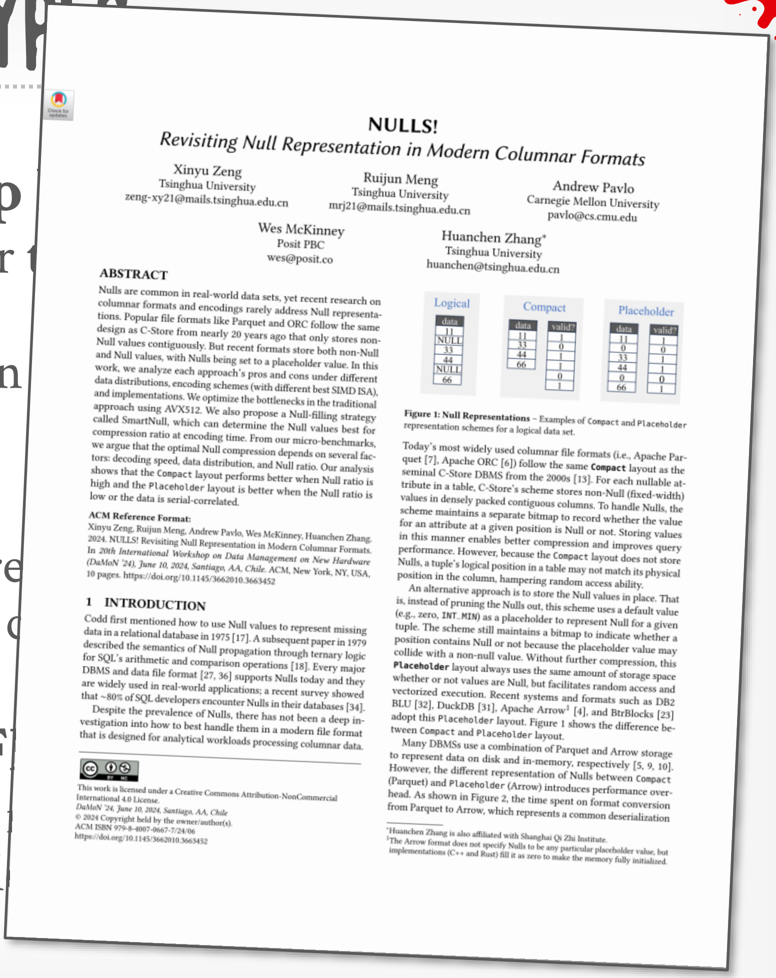


Don't Do This!

Choice #3: Per Attribute Null Fl

→ Store a flag that marks that a value is

→ Must use more space than just a singl up with word alignment.



NULLS!
Revisiting Null Representation in Modern Columnar Formats

Xinyu Zeng
Tsinghua University
zeng-xy21@mails.tsinghua.edu.cn

Ruijun Meng
Tsinghua University
mrj21@mails.tsinghua.edu.cn

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Wes McKinney
Posit PBC
wes@posit.co

Huanchen Zhang*
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT
Nulls are common in real-world data sets, yet recent research on columnar formats and encodings rarely address Null representations. Popular file formats like Parquet and ORC follow the same design as C-Store from nearly 20 years ago that only stores non-Null values contiguously. But recent formats store both non-Null and Null values, with Nulls being set to a placeholder value. In this work, we analyze each approach's pros and cons under different data distributions, encoding schemes (with different best SIMD ISA), and implementations. We optimize the bottlenecks in the traditional approach using AVX512. We also propose a Null-filling strategy called SmartNull, which can determine the Null values best for compression ratio at encoding time. From our micro-benchmarks, we argue that the optimal Null compression depends on several factors: decoding speed, data distribution, and Null ratio. Our analysis shows that the `Compact` layout performs better when Null ratio is high and the `Placeholder` layout is better when the Null ratio is low or the data is serial-correlated.

ACM Reference Format:
Xinyu Zeng, Ruijun Meng, Andrew Pavlo, Wes McKinney, Huanchen Zhang. 2024. NULLS! Revisiting Null Representation in Modern Columnar Formats. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3662010.3663452

1 INTRODUCTION

Codd first mentioned how to use Null values to represent missing data in a relational database in 1975 [17]. A subsequent paper in 1979 described the semantics of Null propagation through ternary logic for SQL's arithmetic and comparison operations [18]. Every major DBMS and data file format [27, 36] supports Nulls today and they are widely used in real-world applications; a recent survey showed that ~80% of SQL developers encounter Nulls in their databases [34].

Despite the prevalence of Nulls, there has not been a deep investigation into how to best handle them in a modern file format that is designed for analytical workloads processing columnar data.

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663452

Figure 1: Null Representations – Examples of `Compact` and `Placeholder` representation schemes for a logical data set.

Today's most widely used columnar file formats (i.e., Apache Parquet [7], Apache ORC [6]) follow the same **`Compact`** layout as the seminal C-Store DBMS from the 2000s [13]. For each nullable attribute in a table, C-Store's scheme stores non-Null (fixed-width) values in densely packed contiguous columns. To handle Nulls, the scheme maintains a separate bitmap to record whether the value for an attribute at a given position is Null or not. Storing values in this manner enables better compression and improves query performance. However, because the `Compact` layout does not store Nulls, a tuple's logical position in a table may not match its physical position in the column, hampering random access ability.

An alternative approach is to store the Null values in place. That is, instead of pruning the Nulls out, this scheme uses a default value (e.g., zero, `INT_MIN`) as a placeholder to represent Null for a given tuple. The scheme still maintains a bitmap to indicate whether a position contains Null or not because the placeholder value may collide with a non-null value. Without further compression, this **`Placeholder`** layout always uses the same amount of storage space whether or not values are Null, but facilitates random access and vectorized execution. Recent systems and formats such as DB2 BLU [32], DuckDB [31], Apache Arrow[1] [4], and BtrBlocks [23] adopt this `Placeholder` layout. Figure 1 shows the difference between `Compact` and `Placeholder` layout.

Many DBMSs use a combination of Parquet and Arrow storage to represent data on disk and in-memory, respectively [5, 9, 10]. However, the different representation of Nulls between `Compact` (Parquet) and `Placeholder` (Arrow) introduces performance overhead. As shown in Figure 2, the time spent on format conversion from Parquet to Arrow, which represents a common deserialization

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.
[1]The Arrow format does not specify Nulls to be any particular placeholder value, but implementations (C++ and Rust) fill it as zero to make the memory fully initialized.

LARGE VALUES

Most DBMSs do not allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

- → Postgres: TOAST (>2KB)
- → MySQL: Overflow (>½ size of page)
- → SQL Server: Overflow (>size of page)

Lots of potential optimizations:

- → Overflow Compression, German Strings

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```

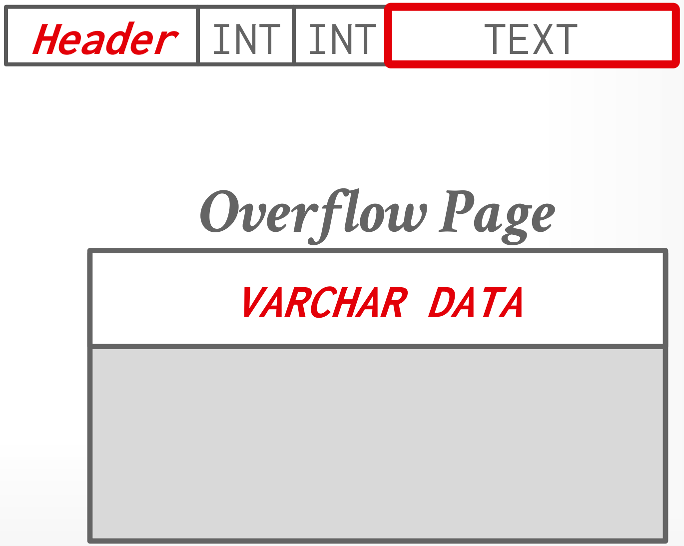
Header	INT	INT	TEXT

LARGE VALUES

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```

Most DBMSs do not allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.
→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)



Lots of potential optimizations:
→ Overflow Compression, German Strings

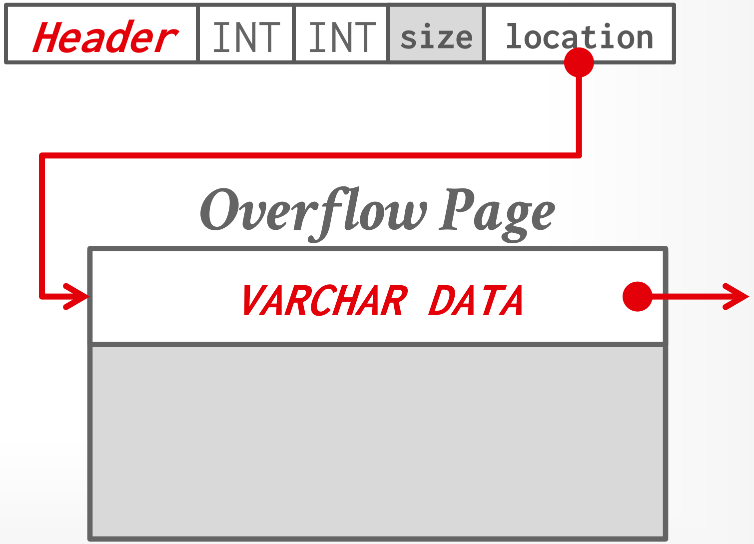
LARGE VALUES

Most DBMSs do not allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.
→ Postgres: TOAST (>2KB)
→ MySQL: Overflow (>½ size of page)
→ SQL Server: Overflow (>size of page)

Lots of potential optimizations:
→ Overflow Compression, German Strings

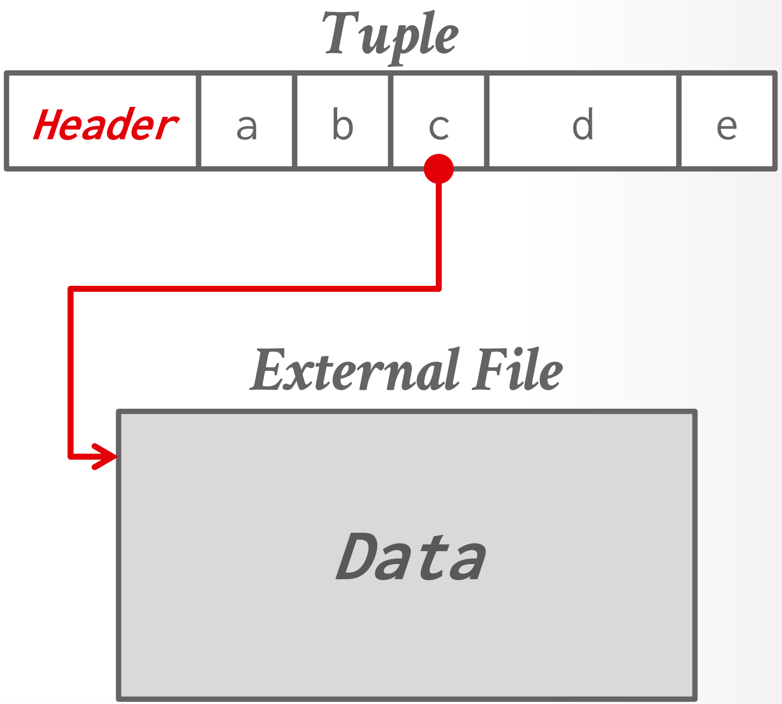
```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```



EXTERNAL VALUE STORAGE

Some systems allow you to store a large value in an external file.
Treated as a **BLOB** type.
→ Oracle: **BFILE** data type
→ Microsoft: **FILESTREAM** data type

The DBMS **cannot** manipulate the contents of an external file.
→ No durability protections.
→ No transaction protections.



EXTERNAL VALUE STORAGE

Some systems allow you to store a large value in an external file. Treated as a **BLOB** type.

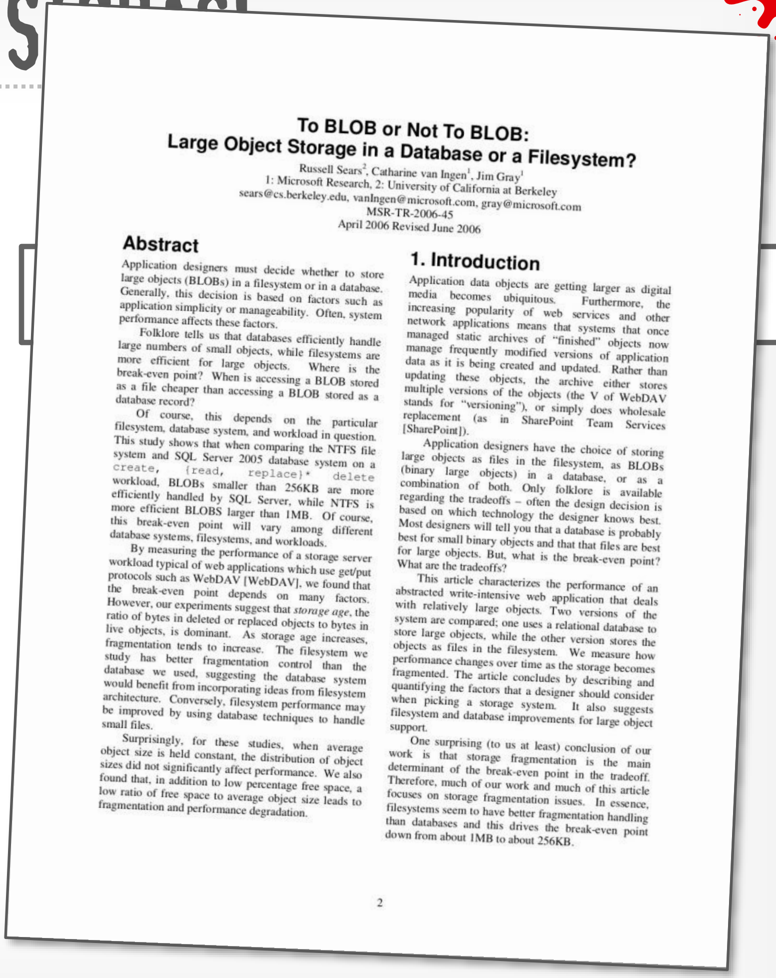
→ Oracle: **BFILE** data type

→ Microsoft: **FILESTREAM** data type

The DBMS **cannot** manipulate the contents of an external file.

→ No durability protections.

→ No transaction protections.



To BLOB or Not To BLOB:
Large Object Storage in a Database or a Filesystem?

Russell Sears[2], Catharine van Ingen[1], Jim Gray[1]
1: Microsoft Research, 2: University of California at Berkeley
sears@cs.berkeley.edu, vanIngen@microsoft.com, gray@microsoft.com
MSR-TR-2006-45
April 2006 Revised June 2006

Abstract

Application designers must decide whether to store large objects (BLOBs) in a filesystem or in a database. Generally, this decision is based on factors such as application simplicity or manageability. Often, system performance affects these factors.

Folklore tells us that databases efficiently handle large numbers of small objects, while filesystems are more efficient for large objects. Where is the break-even point? When is accessing a BLOB stored as a file cheaper than accessing a BLOB stored as a database record?

Of course, this depends on the particular filesystem, database system, and workload in question. This study shows that when comparing the NTFS file system and SQL Server 2005 database system on a `create, {read, replace}* delete` workload, BLOBs smaller than 256KB are more efficiently handled by SQL Server, while NTFS is more efficient BLOBS larger than 1MB. Of course, this break-even point will vary among different database systems, filesystems, and workloads.

By measuring the performance of a storage server workload typical of web applications which use get/put protocols such as WebDAV [WebDAV], we found that the break-even point depends on many factors. However, our experiments suggest that *storage age*, the ratio of bytes in deleted or replaced objects to bytes in live objects, is dominant. As storage age increases, fragmentation tends to increase. The filesystem we study has better fragmentation control than the database we used, suggesting the database system would benefit from incorporating ideas from filesystem architecture. Conversely, filesystem performance may be improved by using database techniques to handle small files.

Surprisingly, for these studies, when average object size is held constant, the distribution of object sizes did not significantly affect performance. We also found that, in addition to low percentage free space, a low ratio of free space to average object size leads to fragmentation and performance degradation.

1. Introduction

Application data objects are getting larger as digital media becomes ubiquitous. Furthermore, the increasing popularity of web services and other network applications means that systems that once managed static archives of "finished" objects now manage frequently modified versions of application data as it is being created and updated. Rather than updating these objects, the archive either stores multiple versions of the objects (the V of WebDAV stands for "versioning"), or simply does wholesale replacement (as in SharePoint Team Services [SharePoint]).

Application designers have the choice of storing large objects as files in the filesystem, as BLOBs (binary large objects) in a database, or as a combination of both. Only folklore is available regarding the tradeoffs – often the design decision is based on which technology the designer knows best. Most designers will tell you that a database is probably best for small binary objects and that that files are best for large objects. But, what is the break-even point? What are the tradeoffs?

This article characterizes the performance of an abstracted write-intensive web application that deals with relatively large objects. Two versions of the system are compared; one uses a relational database to store large objects, while the other version stores the objects as files in the filesystem. We measure how performance changes over time as the storage becomes fragmented. The article concludes by describing and quantifying the factors that a designer should consider when picking a storage system. It also suggests filesystem and database improvements for large object support.

One surprising (to us at least) conclusion of our work is that storage fragmentation is the main determinant of the break-even point in the tradeoff. Therefore, much of our work and much of this article focuses on storage fragmentation issues. In essence, filesystems seem to have better fragmentation handling than databases and this drives the break-even point down from about 1MB to about 256KB.

2

CONCLUSION

Database is organized in pages.

Different ways to track pages.

Different ways to store pages.

Different ways to store tuples.

NEXT CLASS

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.

← Today