

DBS101 Database Systems Fundamentals



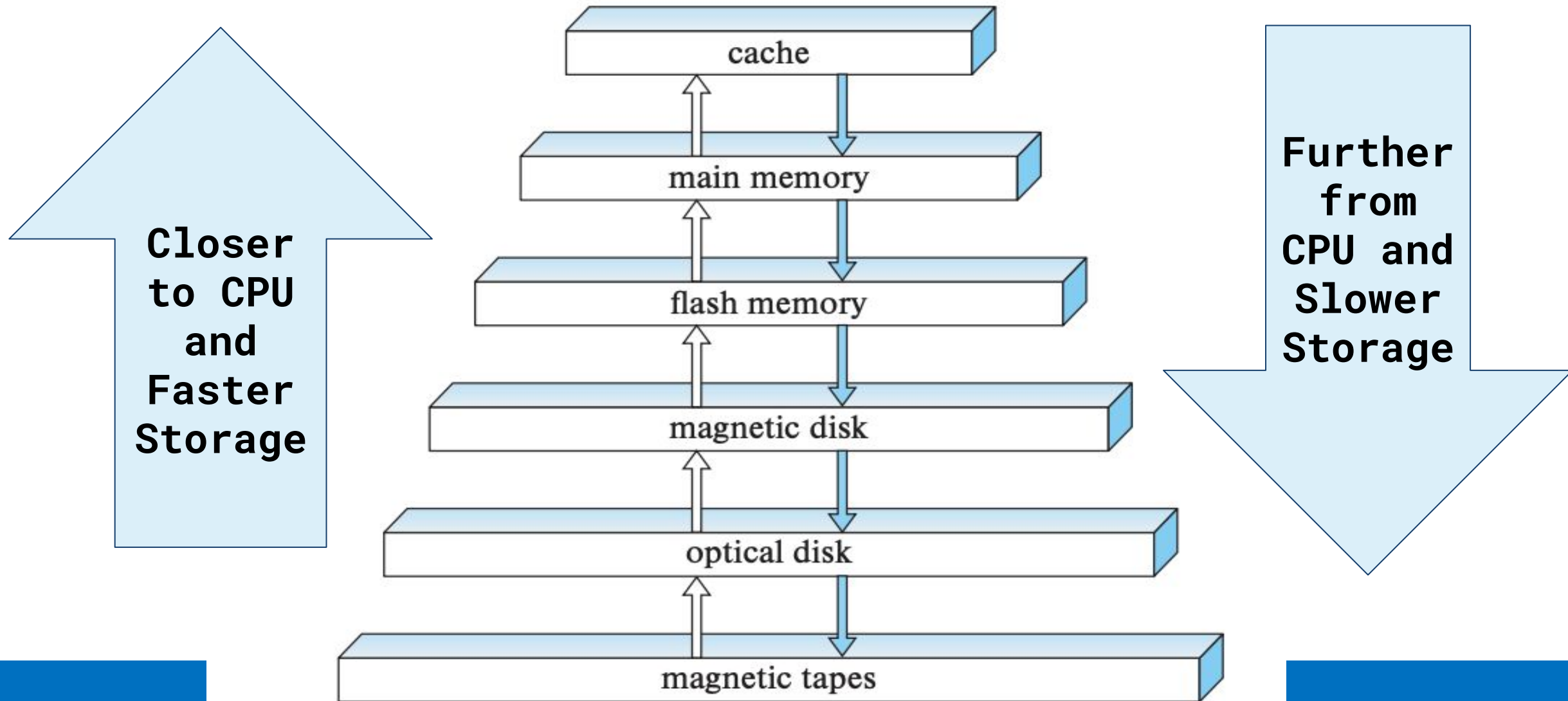
Royal University of Bhutan

Lesson 18

Learning Outcomes

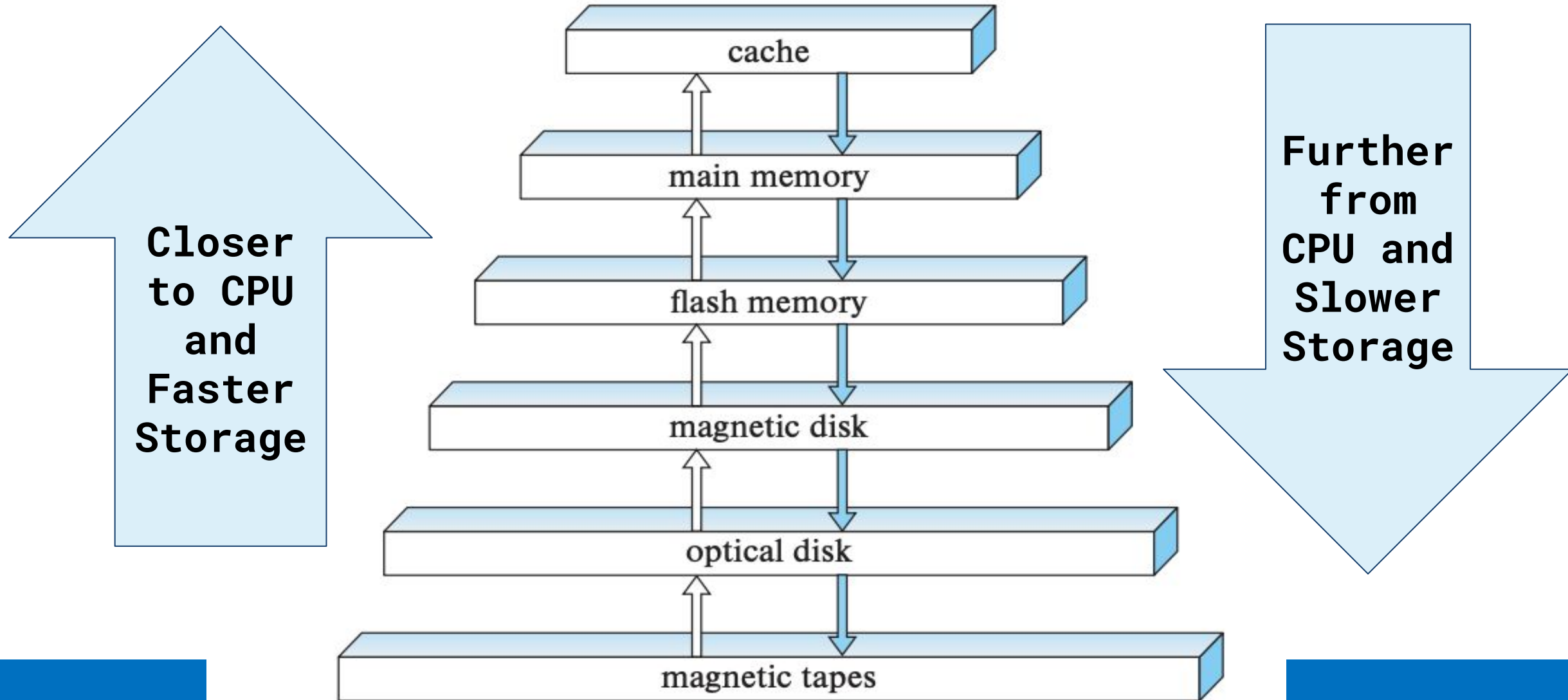
1. Understand the physical storage structures used in databases.
2. Explain the database storage architecture and its components.
3. Describe the different file organization techniques used in databases.

Physical Storage Media

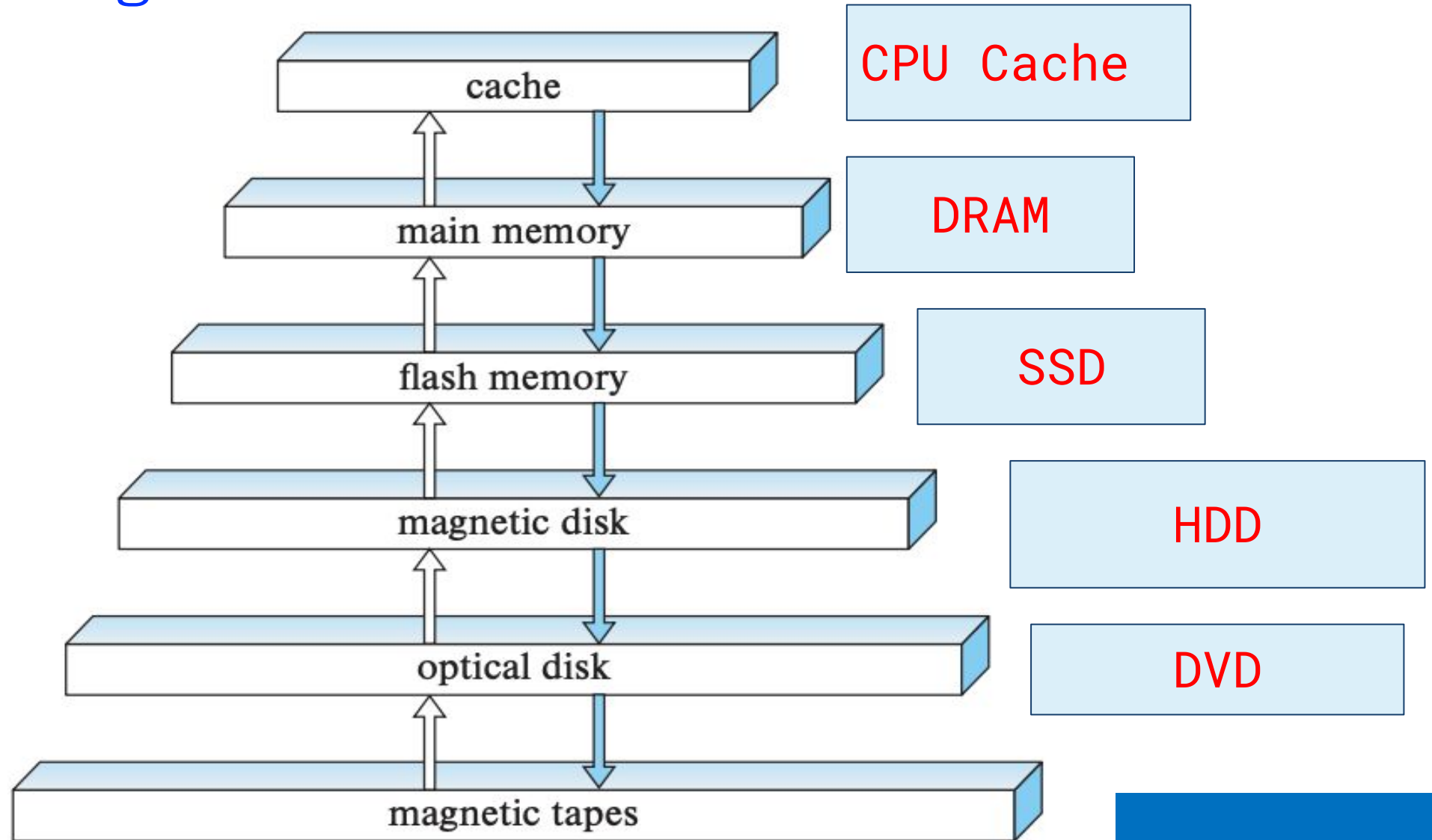


Physical Storage Media

Pg. 559 to Pg 569



Physical Storage Media



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.

Disk Based Architecture

Volatile Device:

- Data is lost when power is pulled.
- Volatile storage supports fast random access with byte-addressable locations. This means that the program can jump to any byte address and get the data that is there.

Disk Based Architecture

What is an example of volatile storage?

Disk Based Architecture

Non-Volatile Device:

- the storage device does not require continuous power in order for the device to retain the bits that it is storing.
- also block/page addressable. This means that in order to read a value at a particular offset, the program first has to load the 4 KB page into memory that holds the value the program wants to read.

Disk Based Architecture

Non-Volatile Device:

- Non-volatile storage is traditionally better at sequential access (reading multiple contiguous chunks of data at the same time).

Disk Based Architecture

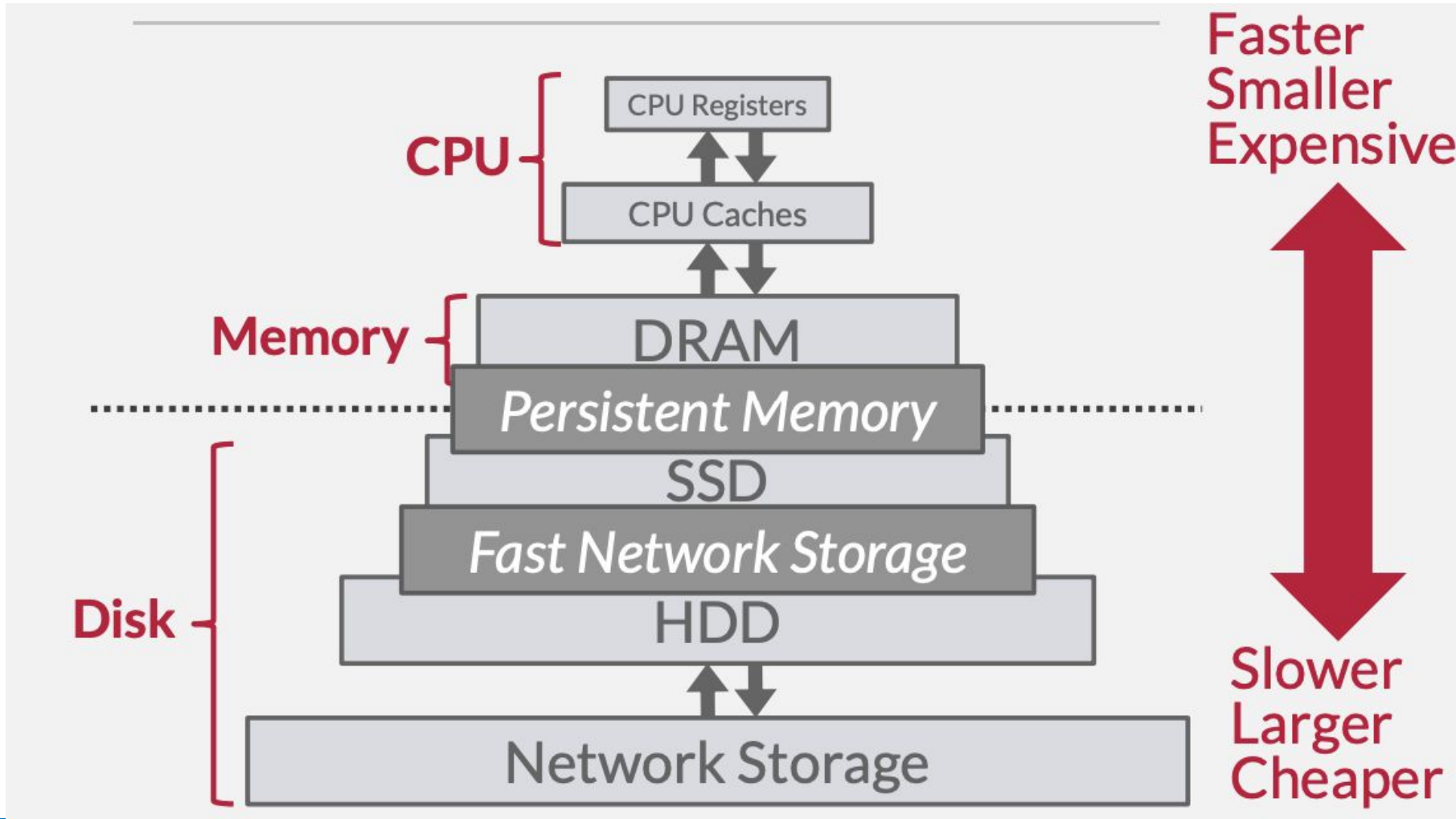
What is an example of non-volatile storage?

Disk Based Architecture

Persistent Memory(PHEM)

- Persistent memory (PMEM) is a solid-state high-performance byte-addressable memory device that resides on the memory bus.
- Being on the memory bus allows PMEM to have DRAM-like access to data, which means that it has nearly the same speed and latency of DRAM and the nonvolatility of NAND flash.

Storage Hierarchy



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 |

Sequential Vs Random Access

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

Applications like websites, databases, and multimedia are generating massive amounts of data, requiring numerous disk drives for storage despite rapid increases in disk capacity.

Having many disks in a system creates opportunities for faster data reading and writing by operating them in parallel. Multiple reads or writes can occur simultaneously, enhancing performance.

This setup also improves data storage reliability because redundant information can be stored across multiple disks, preventing data loss if one disk fails.

Raid

Redundant Arrays of Independent Disks (RAID) encompass various techniques aimed at enhancing performance and reliability by organizing disks effectively.

Initially, using multiple smaller, cheaper disks was considered cost-effective compared to larger, more expensive ones. However, as disk sizes have become smaller and larger-capacity disks are more cost-efficient per megabyte, RAID is now primarily used for reliability, performance, and easier management, rather than solely economic reasons.

Raid

Raid systems can help in

- Improvement of reliability via Redundancy
- Improvement in Performance via Parallelism

Raid Levels

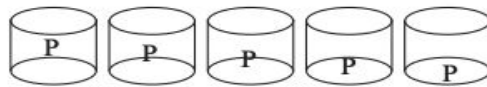
RAID levels define different ways of organizing and distributing data across the disks in the array.



(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity



(d) RAID 6: P + Q redundancy

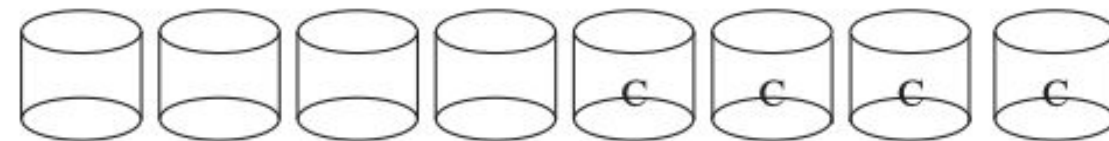
Figure 12.4 RAID levels.

Raid Levels

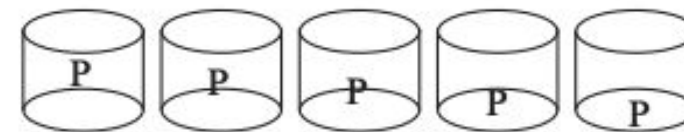
RAID 0 (Striping): In RAID 0, data is split into blocks and distributed across multiple disks. This improves performance by allowing parallel read and write operations, but it doesn't provide any redundancy or fault tolerance. If one disk fails, all data is lost.



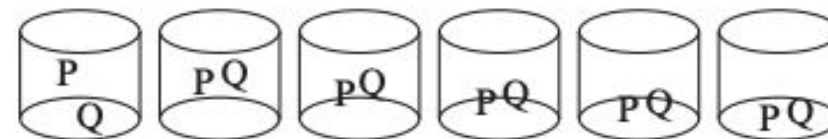
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity



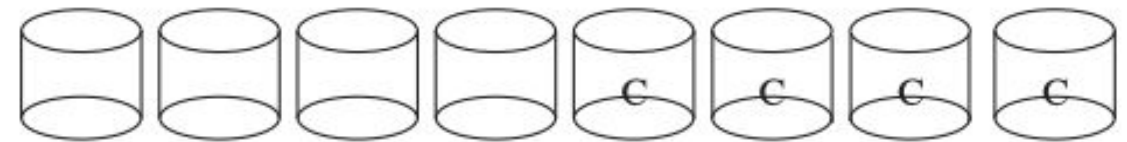
(d) RAID 6: P + Q redundancy

Raid Levels

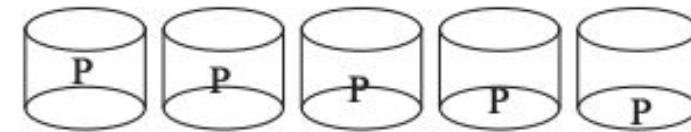
RAID 1 (Mirroring): In RAID 1, data is written identically to two or more disks, creating an exact copy or "mirror" of the data. This provides fault tolerance since if one disk fails, the data can be read from the other disk(s). However, it requires double the storage capacity and doesn't improve performance for read operations.



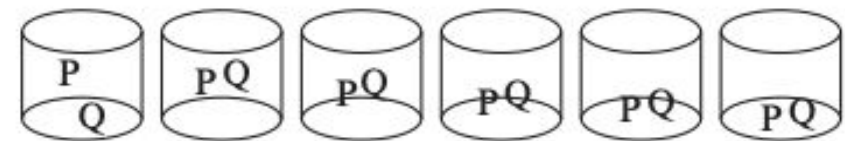
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity



(d) RAID 6: P + Q redundancy

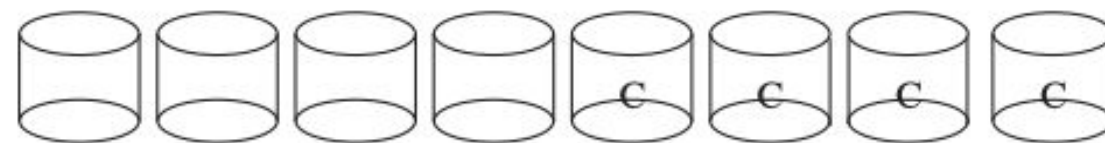
Raid Levels

RAID 5 (Distributed Parity):

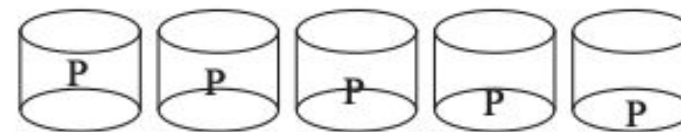
In RAID 5, data is striped across multiple disks, and parity information is distributed across all the disks. If one disk fails, the missing data can be reconstructed from the parity information and the remaining data on the other disks.



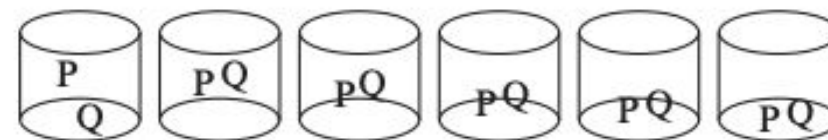
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity



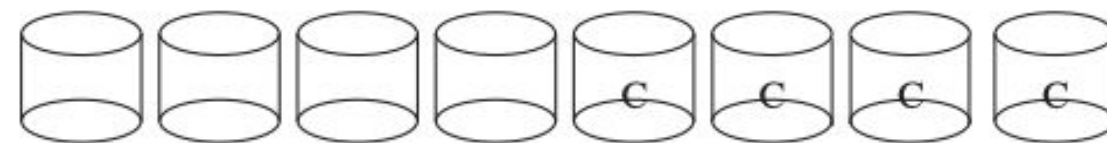
(d) RAID 6: P + Q redundancy

Raid Levels

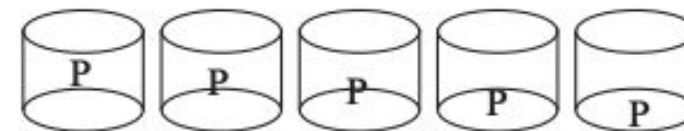
RAID 5 provides fault tolerance and improved performance for small read operations, but write operations can be slower due to the need to calculate and write parity data.



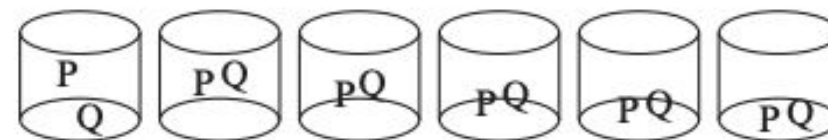
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



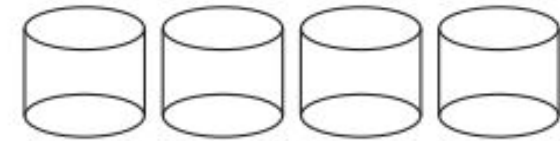
(c) RAID 5: block-interleaved distributed parity



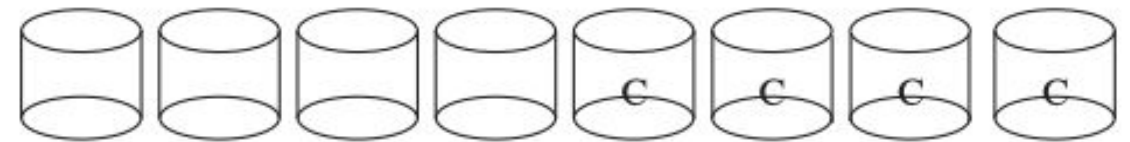
(d) RAID 6: P + Q redundancy

Raid Levels

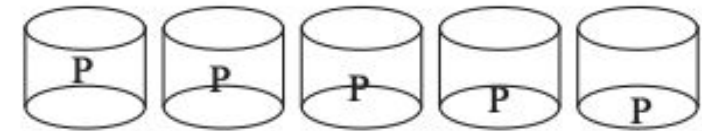
RAID 6 (Dual Parity): RAID 6 is similar to RAID 5, but it uses two sets of parity information instead of one. This allows the array to continue operating even if two disks fail simultaneously, providing even greater fault tolerance than RAID 5.



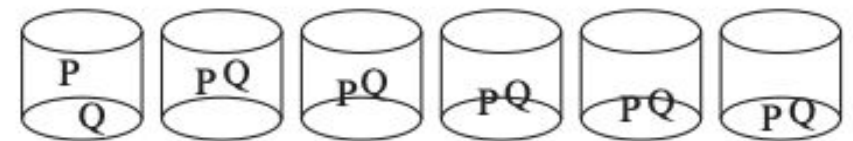
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity



(d) RAID 6: P + Q redundancy

Hardware Raid

Hardware vs Software RAID:

- Software RAID uses only software changes, no special hardware.
- Hardware RAID uses specialized hardware components for better performance and reliability.

Hardware Raid

Benefits of Hardware RAID:

- Non-volatile RAM to record writes before completing them, protecting against power failures.
- Automatic resynchronization (rebuilding data) after disk failure, without manual intervention.
- Scrubbing (reading all sectors periodically) to detect and fix latent failures or bit rot.
- Hot swapping of faulty disks without shutting down the system.
- Redundant components (power supplies, disk interfaces) to avoid single points of failure.

Hardware Raid

Choosing RAID Levels:

- RAID 0 (striping): High performance but no redundancy, used in specialized applications.
- RAID 1 (mirroring): Good write performance, used for log files, moderate storage requirements.
- RAID 5 (distributed parity): Lower storage overhead than RAID 1, good for read-intensive workloads.
- RAID 6 (double parity): Better reliability than RAID 1 or 5, can tolerate two disk failures.

Other Techniques For Improving Disk Access

- Buffering: Caching recently accessed blocks in memory.
- Read-ahead: Reading consecutive blocks into memory in anticipation of sequential access.
- Scheduling: Reordering requests to minimize disk arm movement (elevator algorithm).
- File organization: Storing related blocks together on disk for sequential access.
- Non-volatile write buffers: Using NVRAM to quickly acknowledge writes and reorder them on disk.

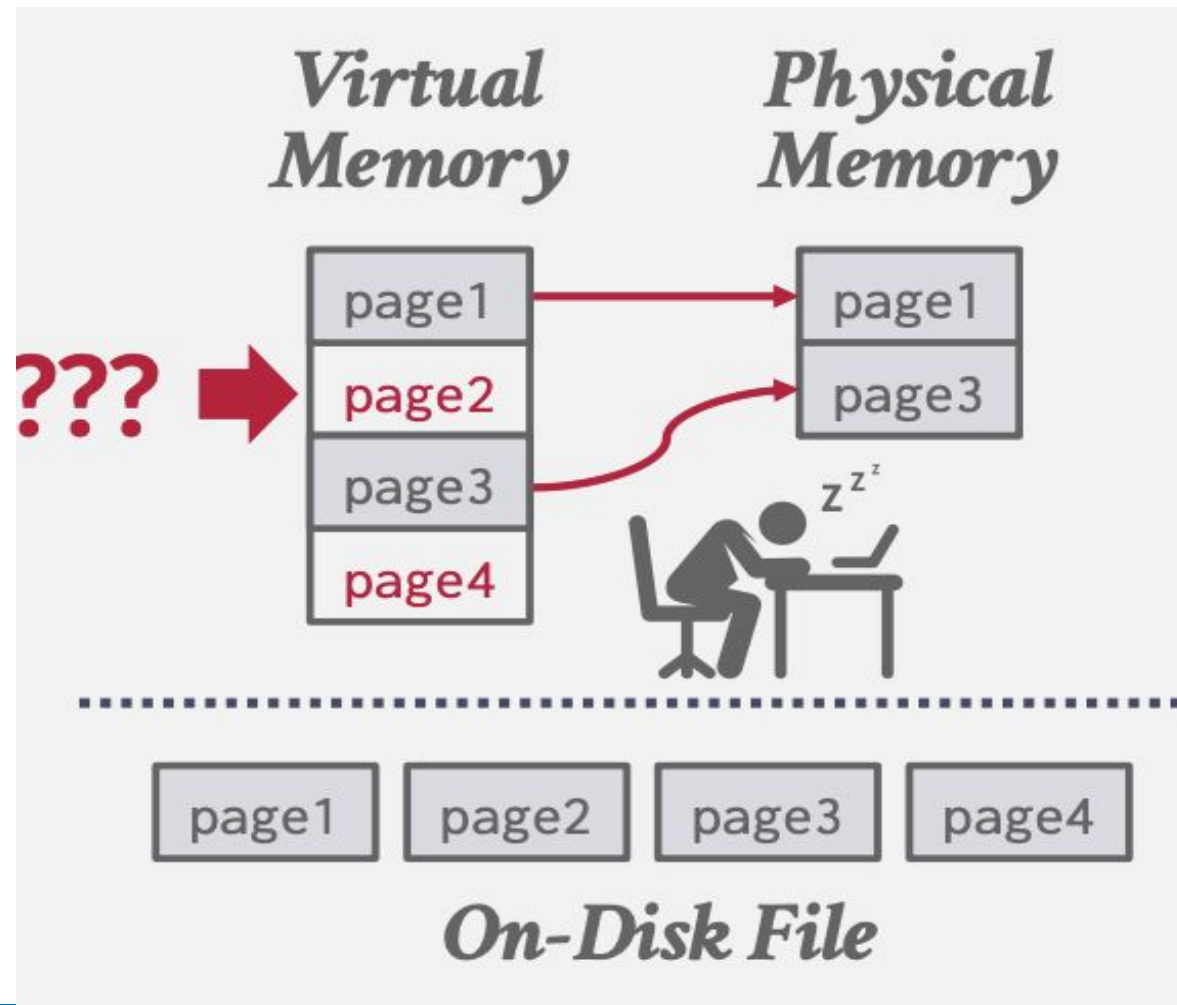
Break Time: 5 mins

File Storage in DBMS

The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program.

OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.

File Storage in DBMS



Memory Mapped I/O Problems

1. Transaction Safety: OS can flush dirty pages at any time.
2. I/O Stalls: DBMS doesn't know which pages are in memory. The OS will stall a thread on page fault.
3. Error Handling: Difficult to validate pages. Any access can cause a SIGBUS that the DBMS must handle.
4. Performance Issues: OS data structure contention. TLB shootdowns.

Memory Mapped I/O Problems

There are some solutions to some of these problems:

1. `madvise`: Tell the OS how you expect to read certain pages.
2. `mlock`: Tell the OS that memory ranges cannot be paged out.
3. `msync`: Tell the OS to flush memory ranges out to disk. Using these syscalls to get the OS to behave correctly is just as onerous as managing memory yourself.

Memory Mapped I/O Problems

There are some solutions to some of these problems:

1. `madvise`: Tell the OS how you expect to read certain pages.
2. `mlock`: Tell the OS that memory ranges cannot be paged out.
3. `msync`: Tell the OS to flush memory ranges out to disk. Using these syscalls to get the OS to behave correctly is just as onerous as managing memory yourself.

Why not use the OS?

DBMS (almost) always wants to control things itself and can do a better job than the OS.

- Flushing dirty pages to disk in the correct order.
- Specialized prefetching.
- Buffer replacement policy.
- Thread/process scheduling.
-

The OS is not your friend.

How do Databases Represent Database Files on Disk?

File Storage

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

- The OS doesn't know anything about the contents of these files.
- Early systems in the 1980s used custom filesystems on raw block storage.
 - Some "enterprise" DBMSs still support this.
 - Most newer DBMSs do not do this.

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

Database Pages

A page is a fixed-size block of data.

- It can contain tuples, meta-data, indexes, log records etc.
- Most systems do not mix page types.
- Some systems require a page to be self-contained.

Each page is given a unique identifier.

- 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.

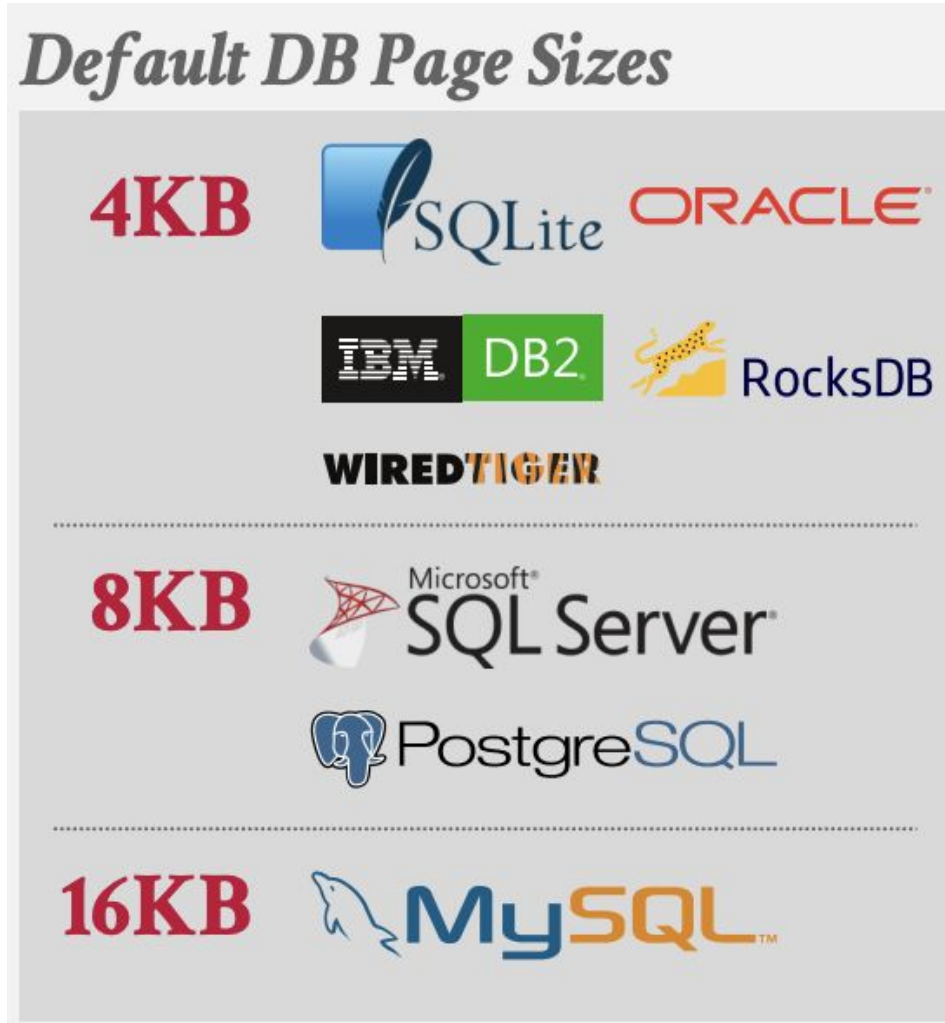
Database Pages

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.

Database Pages



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 don't need to know anything about what is inside of the pages.

Yesterday We learned about DB file storage:

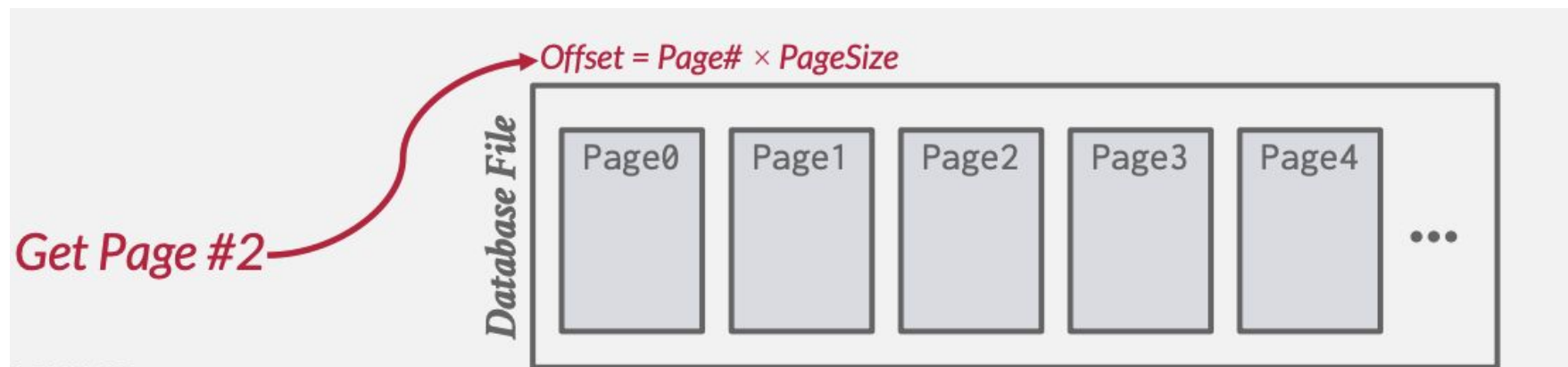
A common location for PGDATA is `/var/lib/pgsql/data`.

Heap File Organization

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.

It is easy to find pages if there is only a single file.



Heap File Organization



Heap File Organization

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.

It is easy to find pages if there is only a single file.

Need meta-data to keep track of what pages exist in multiple files and which ones have free space.

Heap File Page Directory

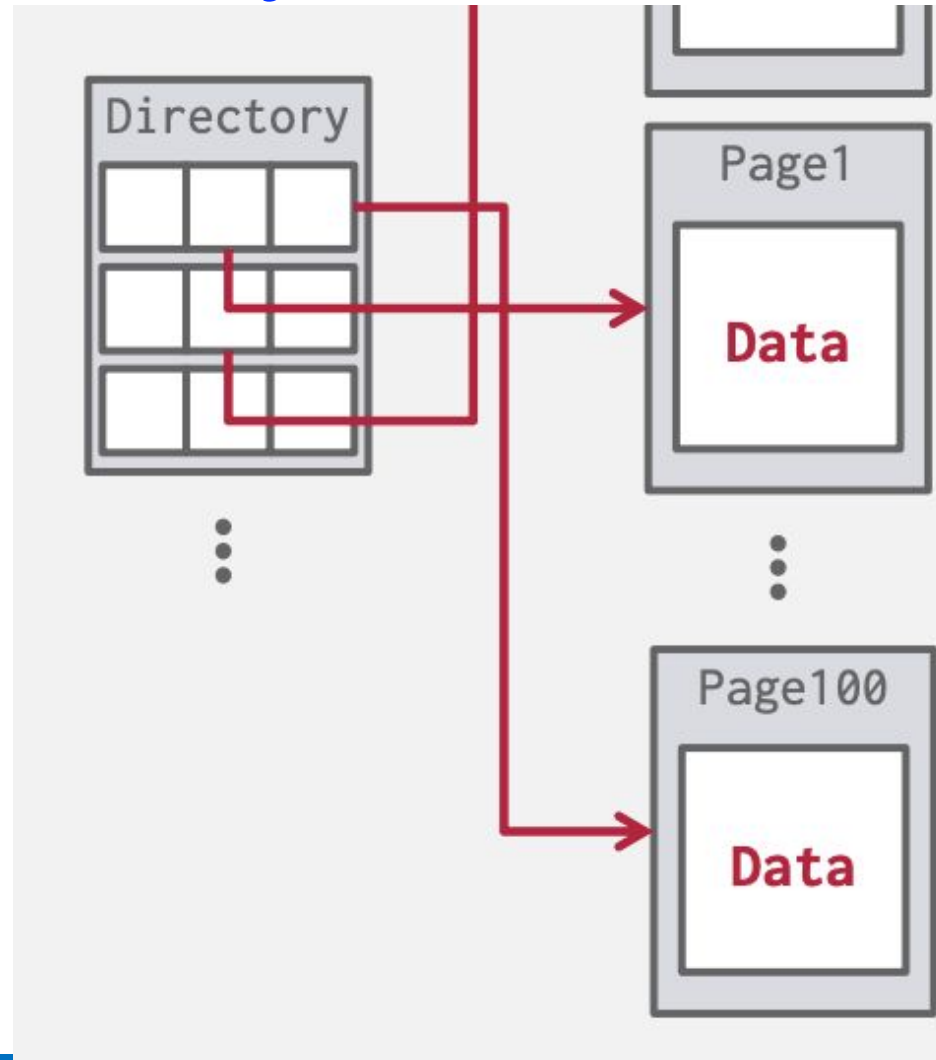
The DBMS maintains special pages that tracks the location of data pages in the database files.

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

The directory also records meta-data about available space:

- The number of free slots per page.
- List of free / empty pages.

Heap File Page Directory



In terms of Records/Tuples and Relations

Heap File Organization

- Records can be stored anywhere in the file.
- Free-space map tracks blocks with available free space.
 - Main free-space map has one entry per block.
 - Secondary free-space map summarizes main map for faster scans.
- Periodic scans recompute and update the free-space map on disk.

Sequential File Organization

- Records are stored in sorted order of a search key.
- Each record points to the next record in search-key order.
- Insertions use overflow blocks if no space in the correct block.
- Periodic reorganization is required to maintain search-key order.

Multitable Clustering File Organization

- Records from multiple relations are stored in the same blocks.
- Records are clustered based on a common cluster key attribute.
- Useful for queries involving a join on the cluster key.
- May slow down other queries that don't use the cluster key.

Partitioning

- Large relations are partitioned into smaller relations.
- Partitioning is typically done based on an attribute value (e.g., year).
- Queries can be optimized to access only relevant partitions.
- Reduces overhead and allows storing partitions on different devices.

Partitioning

- Query optimizers can rewrite such a query to only access the smaller relation corresponding to the requested year, and they can avoid reading records corresponding to other years. For example, a query

```
select *  
from transaction  
where year=2019
```

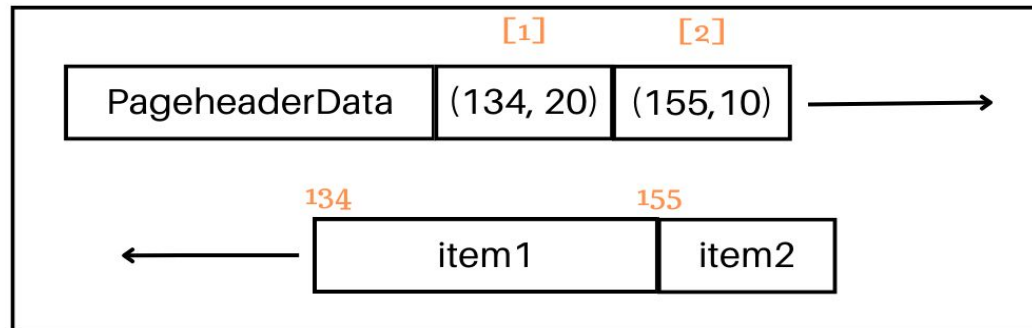
would only access the relation transaction 2019, ignoring the other relations.

Till Now:

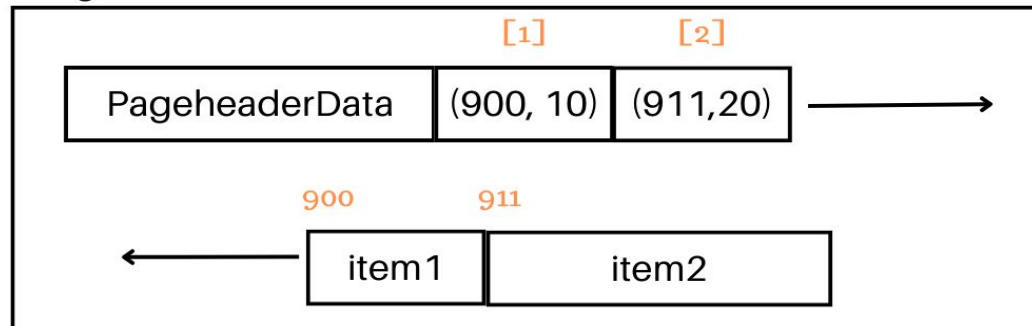
- **DBMS File Storage**
- Page Layout
- Tuple Layout

Postgres Page Layout

Page 0



Page1



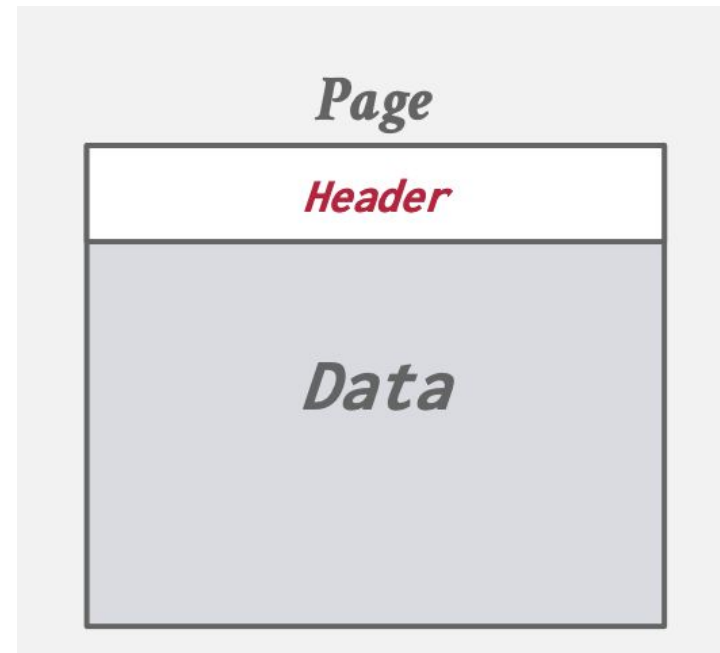
For example Item1 in Page 0 can be referenced with ctid (0,1) where 0 is the page and 1 is the index of the item pointer which has the value of (134, 20).

This way we can change the tuple location in page and update the item pointer while the ctid still remain (0,1)

Page Header

Every page contains a header of metadata about the page's contents.

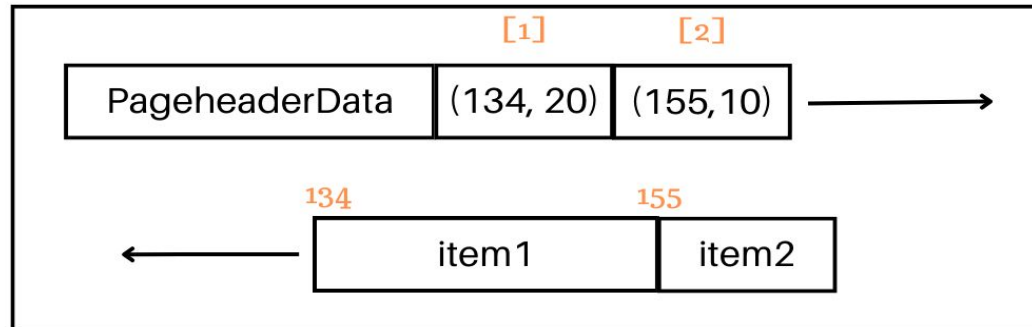
- Page Size
- Checksum
- DBMS Version
- Transaction Visibility
- Compression / Encoding Meta-data
- Schema Information
- Data Summary / Sketches



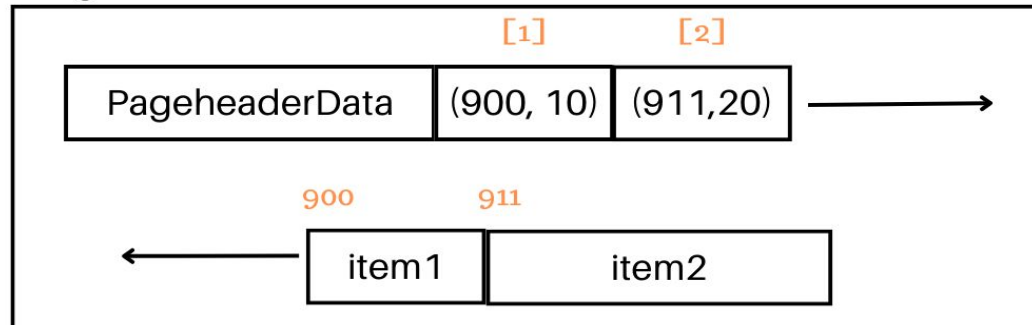
Some systems require pages to be self contained (e.g., Oracle).

Postgres Page Layout

Page 0



Page1



For example Item1 in Page 0 can be referenced with ctid (0,1) where 0 is the page and 1 is the index of the item pointer which has the value of (134, 20).

This way we can change the tuple location in page and update the item pointer while the ctid still remain (0,1)

Page Layout

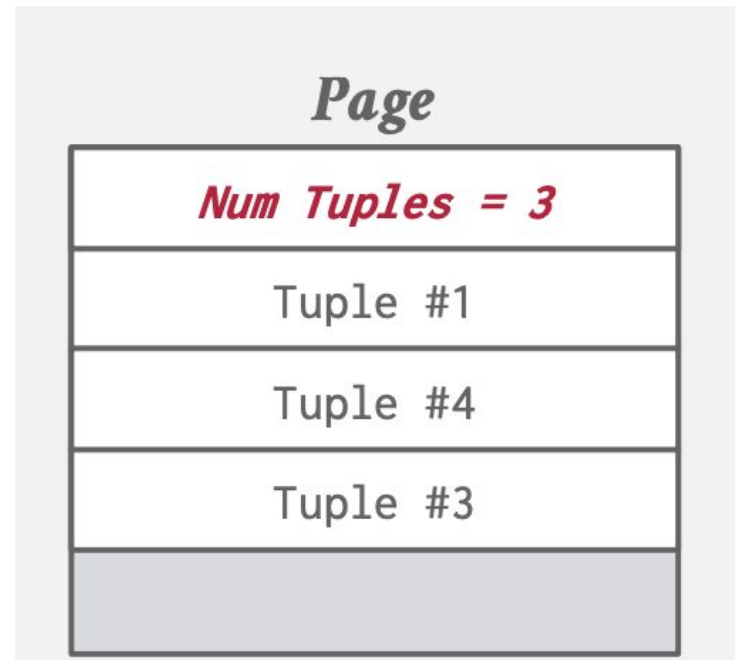
For any page storage architecture, we now need to decide how to organize the data inside of the page.

- We are still assuming that we are only storing tuples in a row-oriented storage model.
- 1. Tuple/Records-oriented Storage
- 2. Log-structured Storage
- 3. Index-organized Storage

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?

What happens if we have a variable length attribute?

Tuple Oriented Storage

Slotted Pages

The most common layout scheme is called slotted pages.

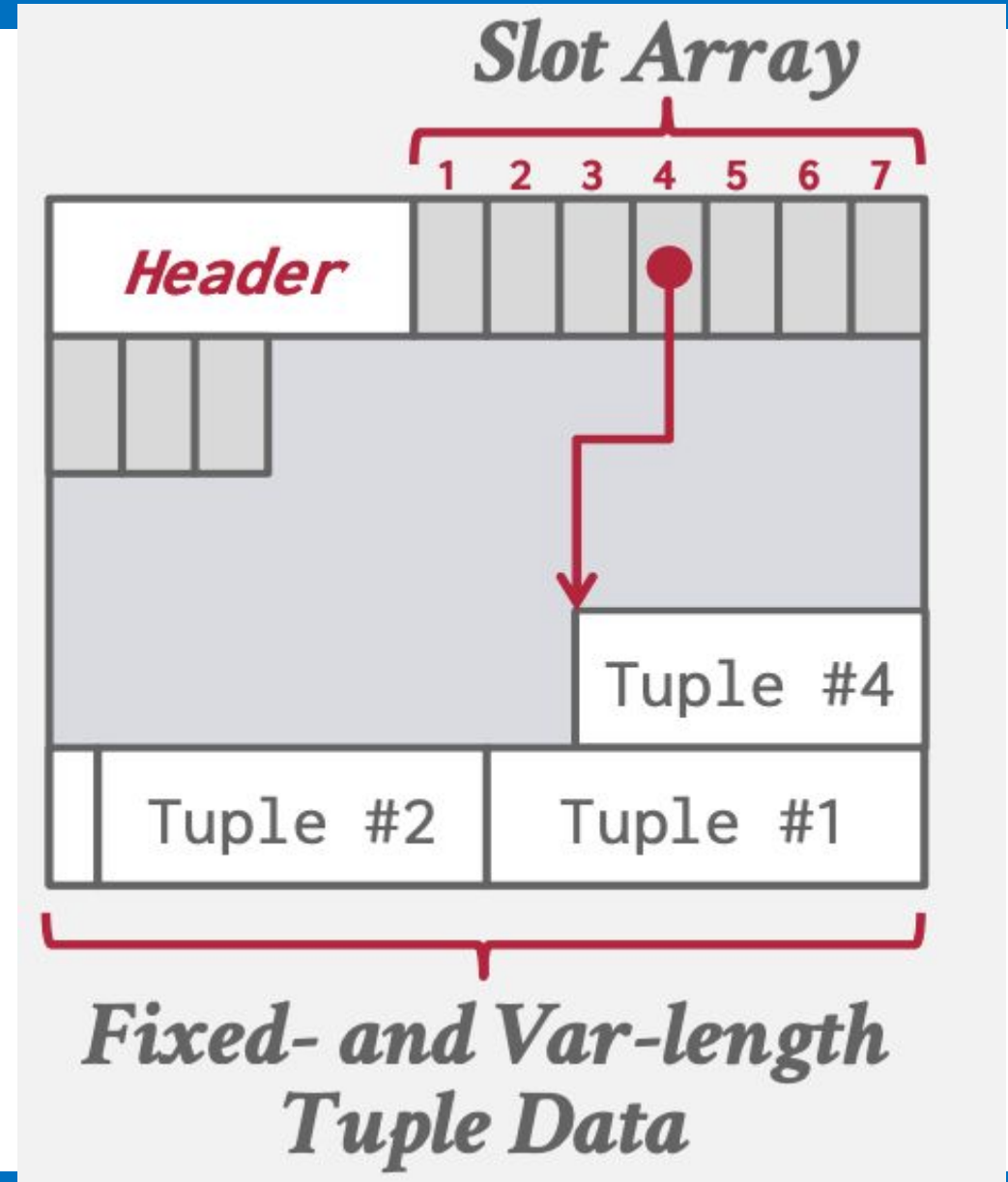
The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The number of used slots
- The offset of the starting location of the last slot used.

Tuple Oriented Storage

Slotted Pages



Tuple Oriented Storage

Record IDS

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

- 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.

Tuple Oriented Storage

Record IDS



CTID (6-bytes)



ROWID (8-bytes)



%%physloc%% (8-bytes)



ROWID (10-bytes)

Till Now:

- **DBMS File Storage**
- **Page Layout**
- Tuple Layout

Tuple Layout

A tuple is essentially a sequence of bytes.

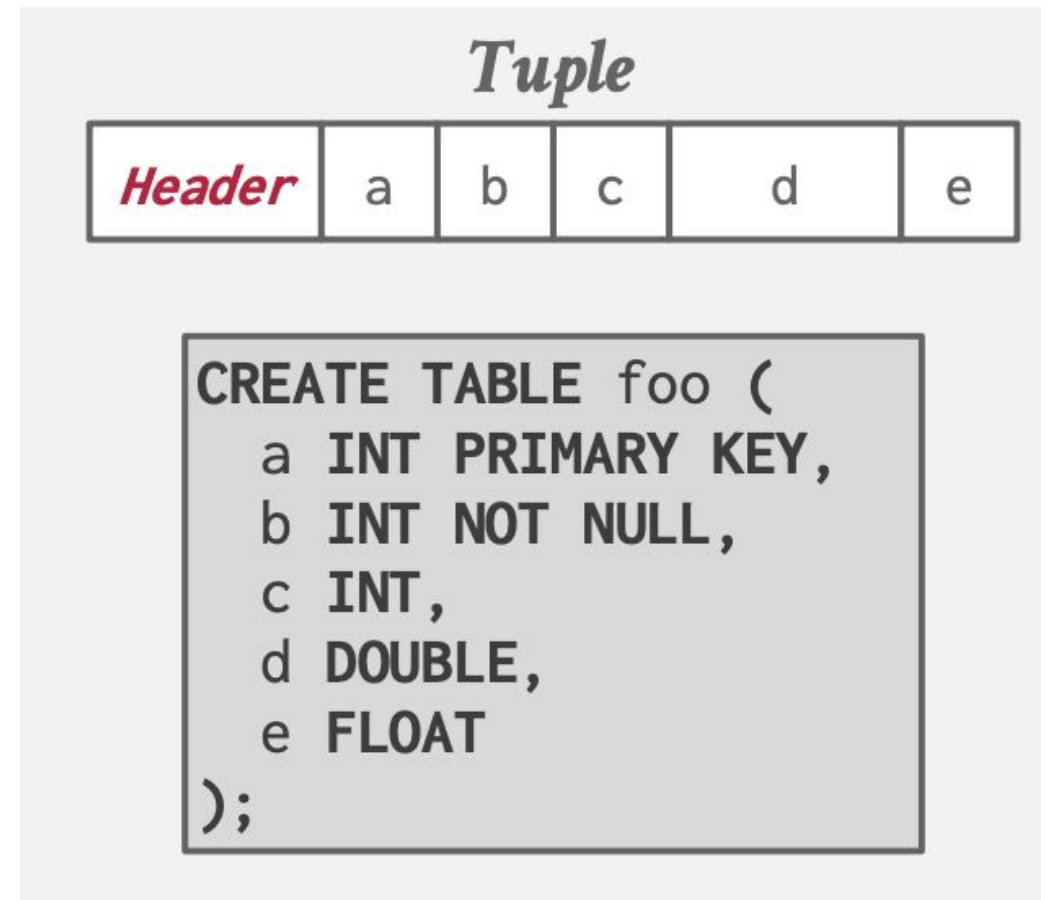
It's the job of the DBMS to interpret those bytes into attribute types and values

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.

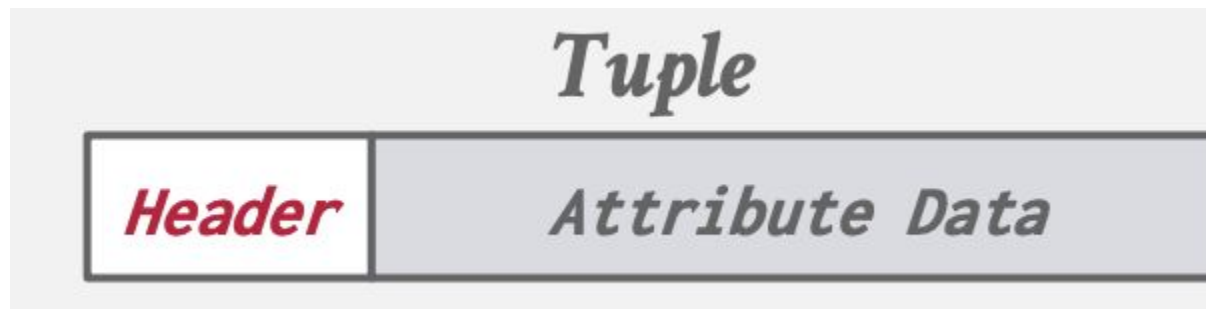


Tuple Layout

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.



Denormalized Tuple data

DBMS can physically denormalize (e.g., "pre-join") related tuples and store them together in the same page.

- Potentially reduces the amount of I/O for common workload patterns.
- Can make updates more expensive.

Not a new idea.

- IBM System R did this in the 1970s.
- Several NoSQL DBMSs do this without calling it physical denormalization.

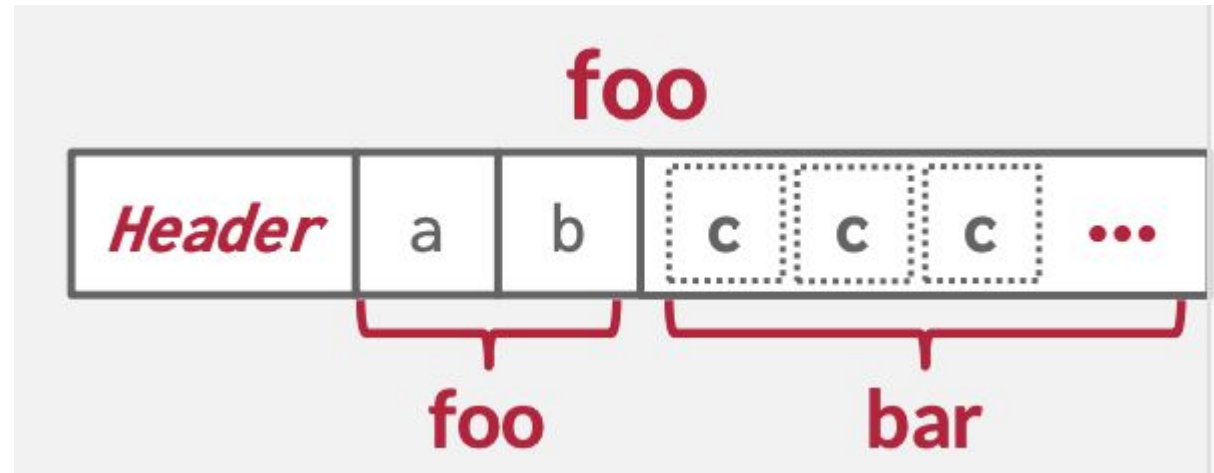
Denormalized Tuple data

foo

| | | |
|---------------|---|---|
| <i>Header</i> | a | b |
|---------------|---|---|

bar

| | | |
|---------------|---|---|
| <i>Header</i> | c | a |
| <i>Header</i> | c | a |
| <i>Header</i> | c | a |



Tuple Oriented Storage

Insert a new tuple:

- Check page directory to find a page with a free slot.
- Retrieve the page from disk (if not in memory).
- Check slot array to find empty space in page that will fit.

Tuple Oriented Storage

Update an existing tuple using its record id:

- Check page directory to find location of page.
- Retrieve the page from disk (if not in memory).
- Find offset in page using slot array.
- If new data fits, overwrite existing data. Otherwise, mark existing tuple as deleted and insert new version in a different page.

Tuple Oriented Storage

Problem #1: Fragmentation

- Pages are not fully utilized (unusable space, empty slots).

Problem #2: Useless Disk I/O

- DBMS must fetch entire page to update one tuple.

Problem #3: Random Disk I/O

- Worse case scenario when updating multiple tuples is that each tuple is on a separate page.

What if the DBMS cannot overwrite data in pages and could only create new pages? Examples: Some object stores, HDFS

Log- Structured Storage

Instead of storing tuples in pages, the DBMS maintains a log that records changes to tuples.

- Each log entry represents a tuple PUT/DELETE operation.
- Originally proposed as log-structure merge trees (LSM Trees) in 1996.

The DBMS appends new log entries to an in memory buffer and then writes out the changes sequentially to disk.

Log- Structured Storage

DBMS stores log records that contain changes to tuples (PUT, DELETE).

- Each log record must contain the tuple's unique identifier.
- Put records contain the tuple contents.
- Deletes marks the tuple as deleted.

As the application makes changes to the database, the DBMS appends log records to the end of the file without checking previous log records.

Log- Structured Storage

When the page gets full, the DBMS writes it out disk and starts filling up the next page with records.

- All disk writes are sequential.
- On-disk pages are immutable.

The DBMS may also flush partially full pages for transactions but we will ignore that for now...

Log- Structured Storage

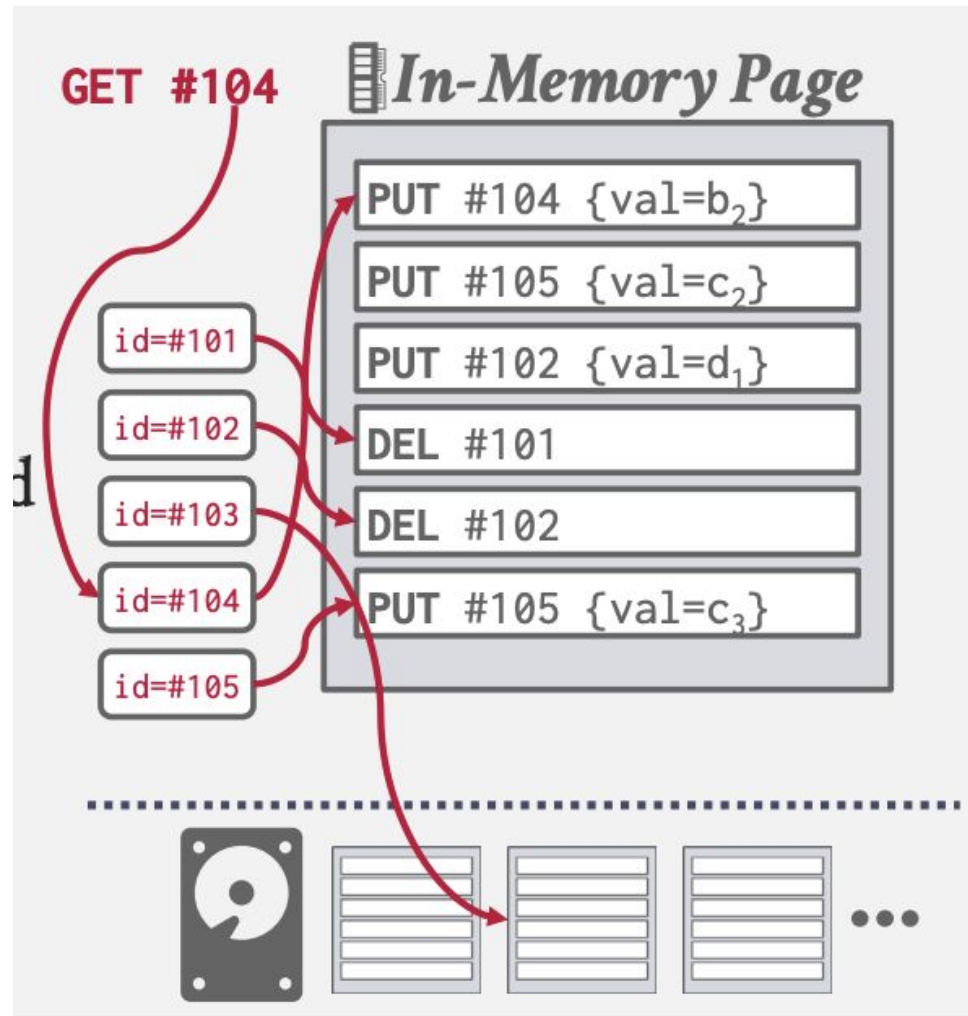
To read a tuple with a given id, the DBMS finds the newest log record corresponding to that id.

- Scan log from newest to oldest.

Maintain an index that maps a tuple id to the newest log record.

- If log record is in-memory, just read it.
- If log record is on a disk page, retrieve it.

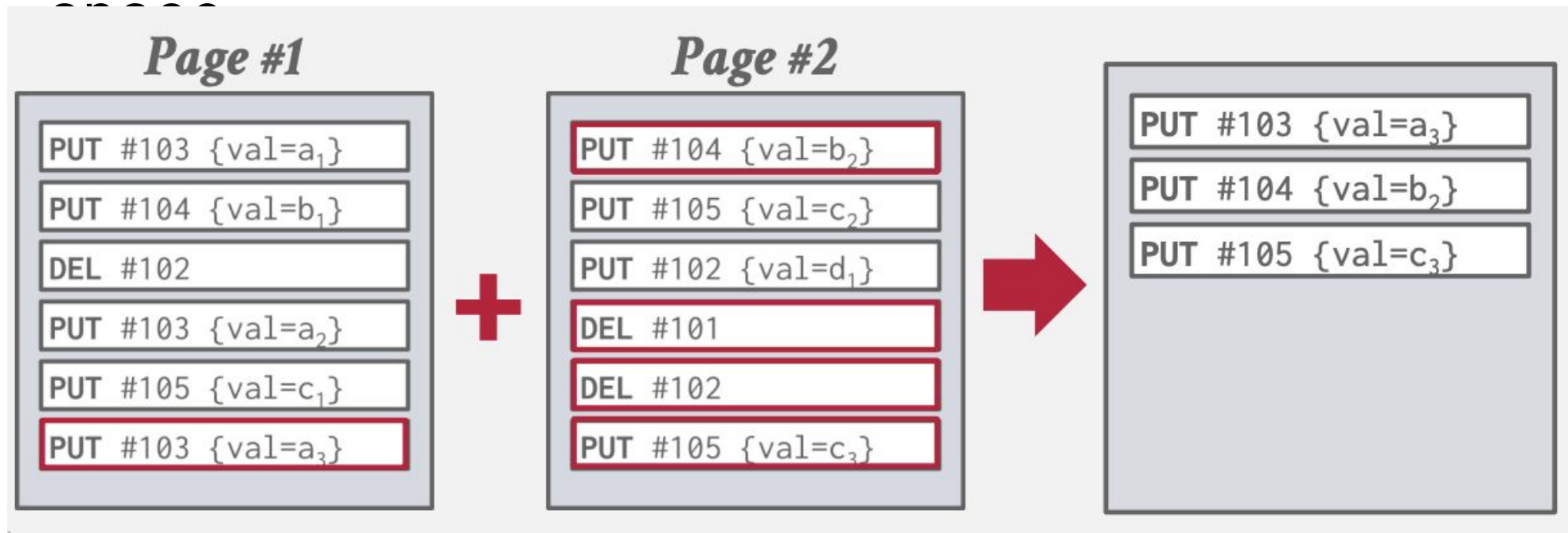
Log- Structured Storage



Log- Structured Compaction

DBMS (usually) does not need to maintain all older log entries for a tuple indefinitely.

- Periodically compact pages to reduce wasted



Log- Structured Compaction

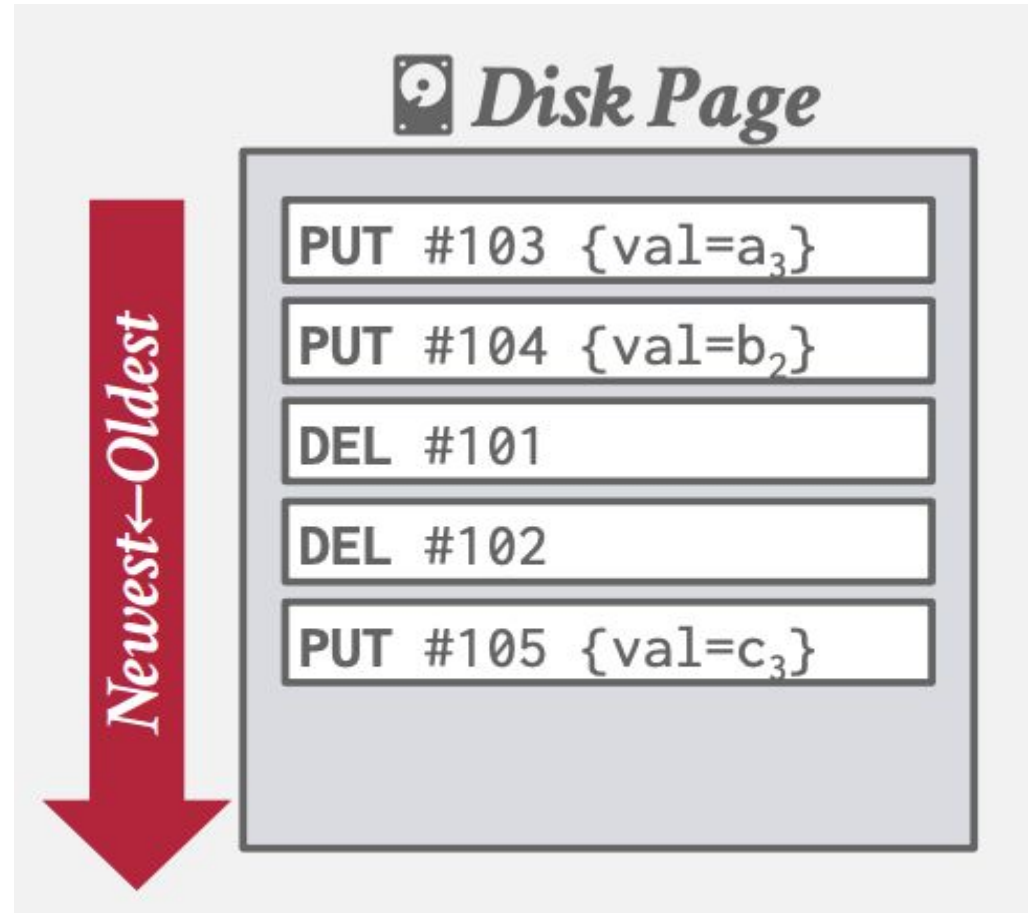
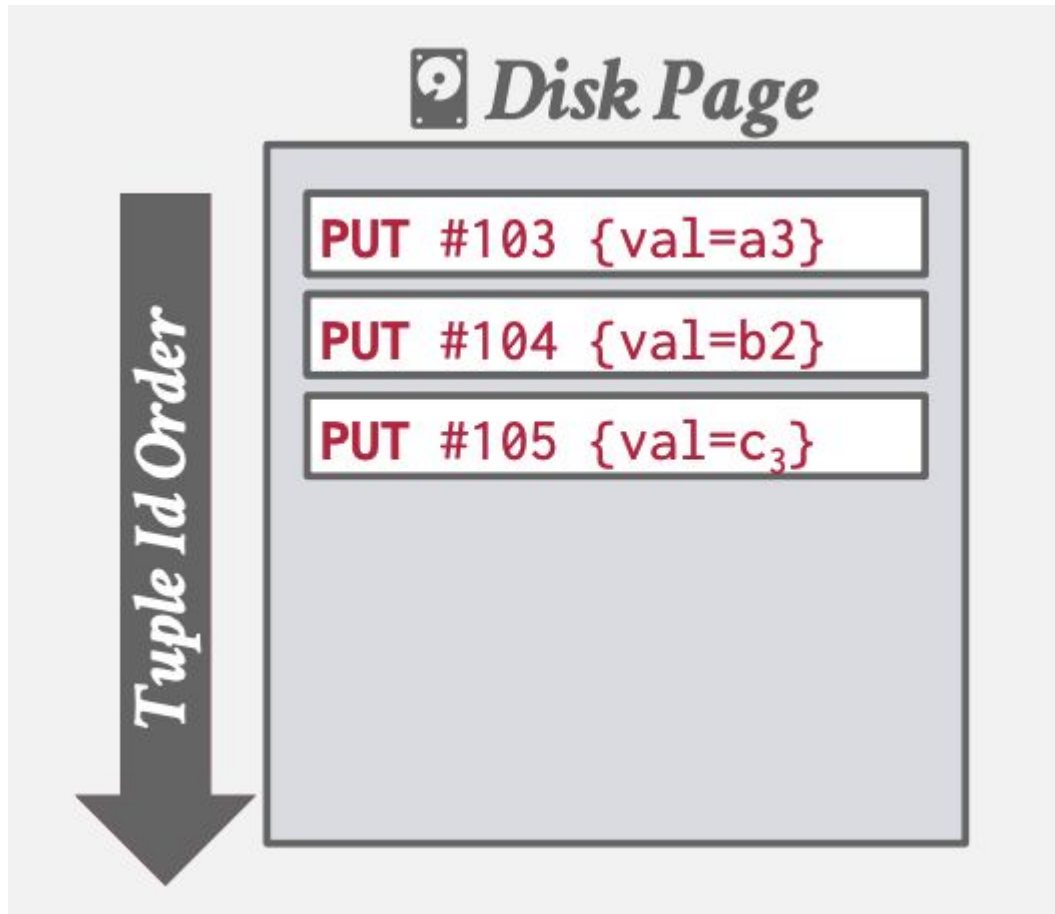
After a page is compacted, the DBMS does not need to maintain temporal ordering of records within the page.

- Each tuple id is guaranteed to appear at most once in the page.

The DBMS can instead sort the page based on id order to improve efficiency of future lookups. -
Called Sorted String Tables (SSTables)

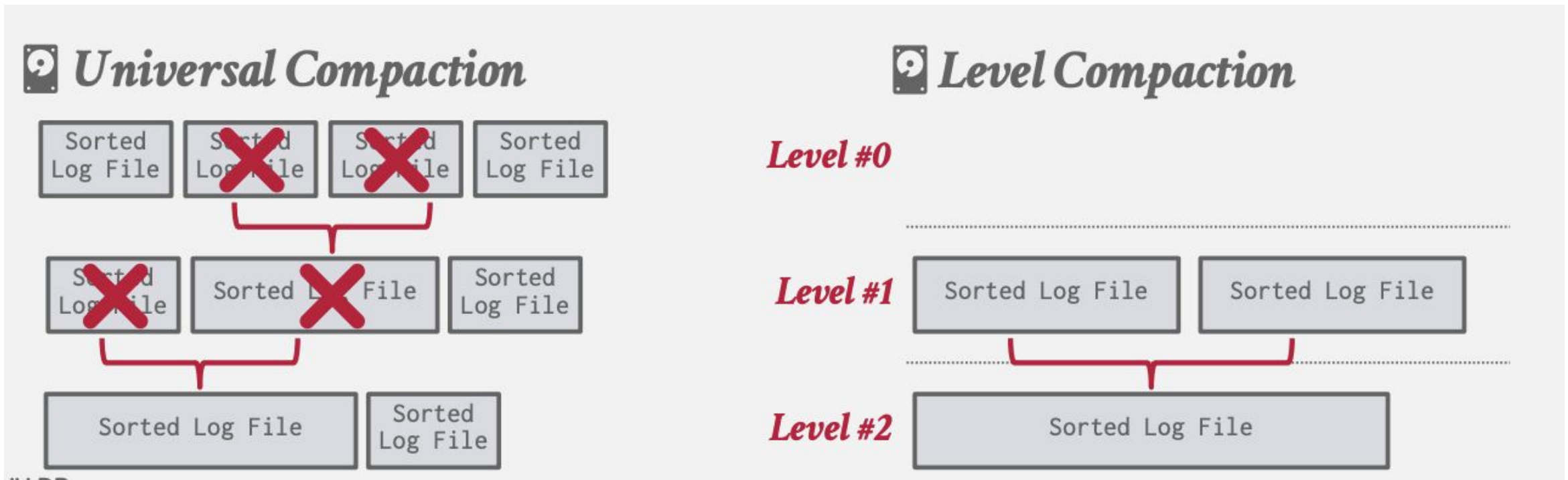
- Embed indexes / filters in the header for reducing search times.

Log- Structured Compaction



Log- Structured Compaction

Coalesce larger disk-resident log files into smaller files by removing unnecessary records.



Log- Structured Storage Managers

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

What are some downsides of this approach?

- Write-Amplification
- Compaction is Expensive

The two table storage approaches we've discussed so far rely on indexes to find individual tuples.

- Such indexes are necessary because the tables are inherently unsorted.

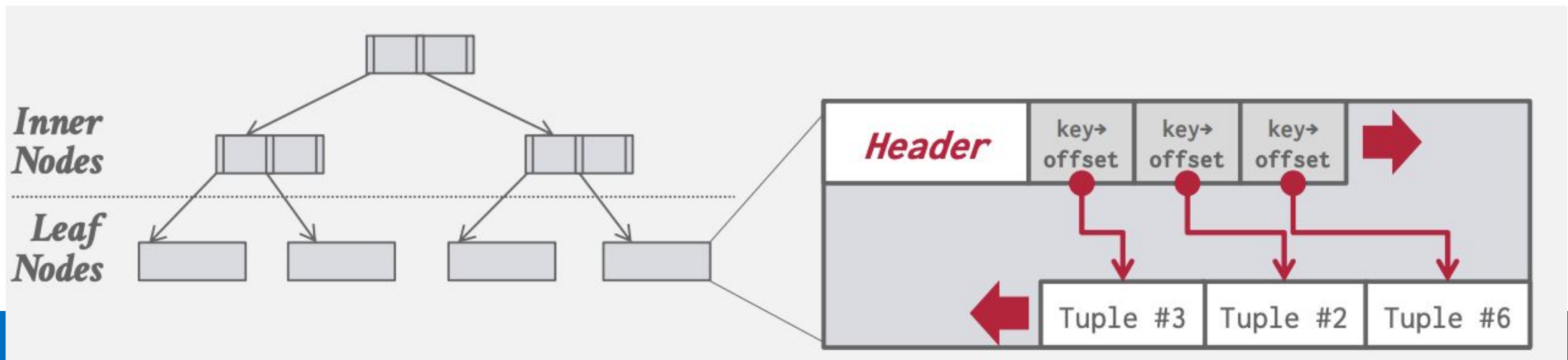
But what if the DBMS could keep tuples sorted automatically using an index?

Index-organised storage

DBMS stores a table's tuples as the value of an index data structure.

- Still use a page layout that looks like a slotted page.

Tuples are typically sorted in page based on key.




Tuple Storage

A tuple is essentially a sequence of bytes. It's 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.

Data Layout



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

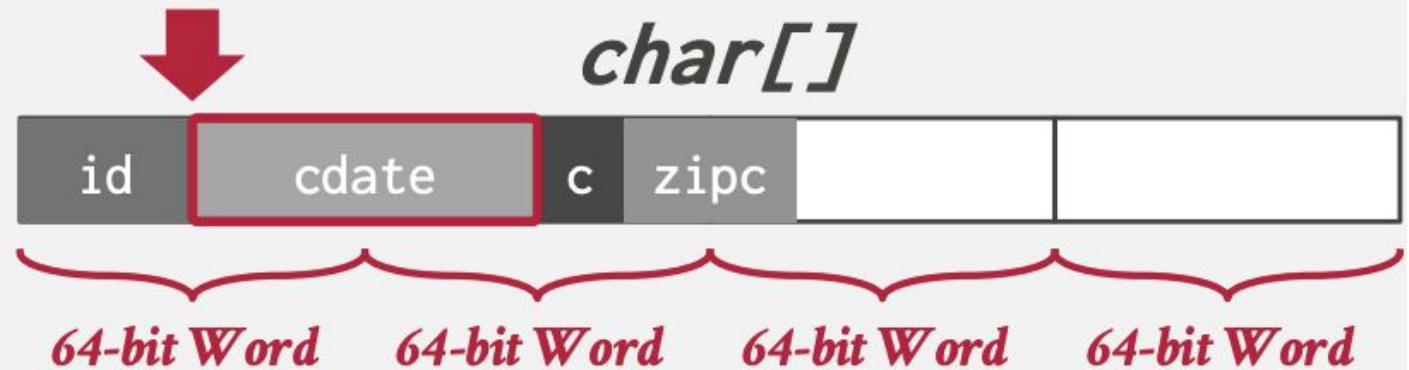


```
reinterpret_cast<int32_t*>(address)
```

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



Word-Aligned Tuples

Approach #1: Perform Extra Reads

- Execute two reads to load the appropriate parts of the data word and reassemble them.

Approach #2: Random Reads

- Read some unexpected combination of bytes assembled into a 64-bit word.

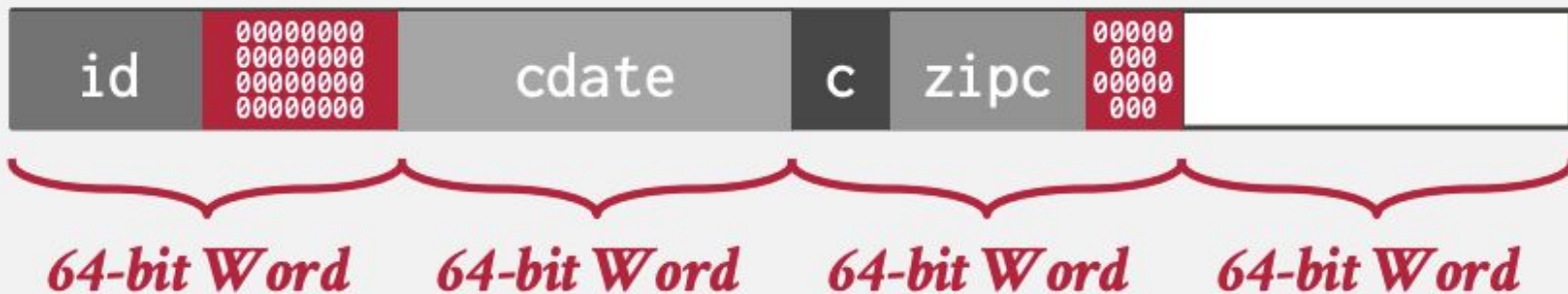
Approach #3: Reject

- Throw an exception and hope app handles it.

Word-Aligned: Padding

Add empty bits after attributes to ensure that tuple is word aligned.

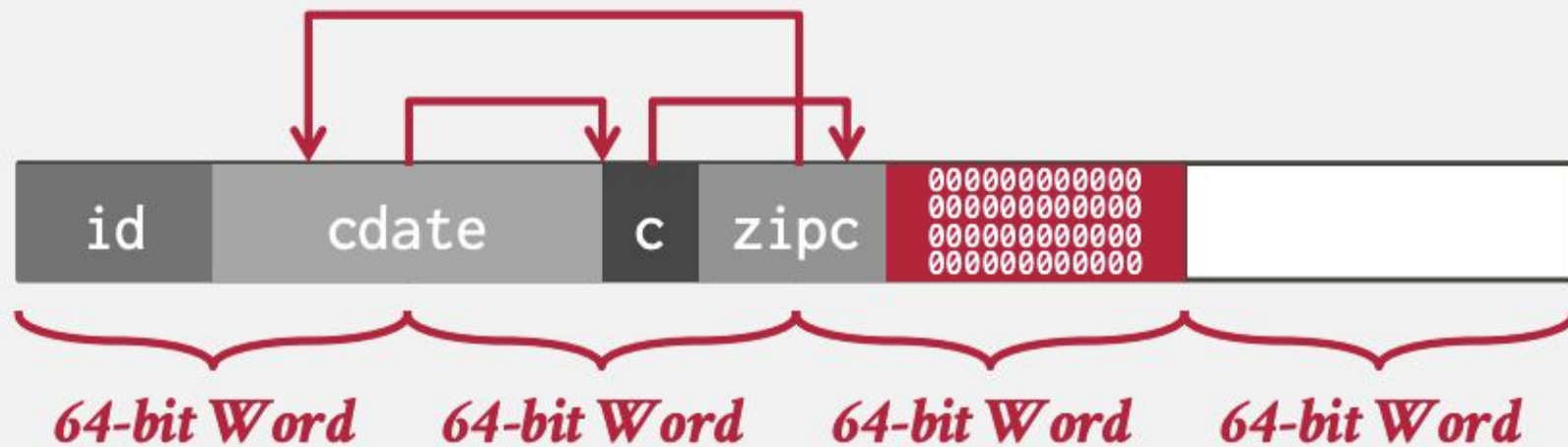
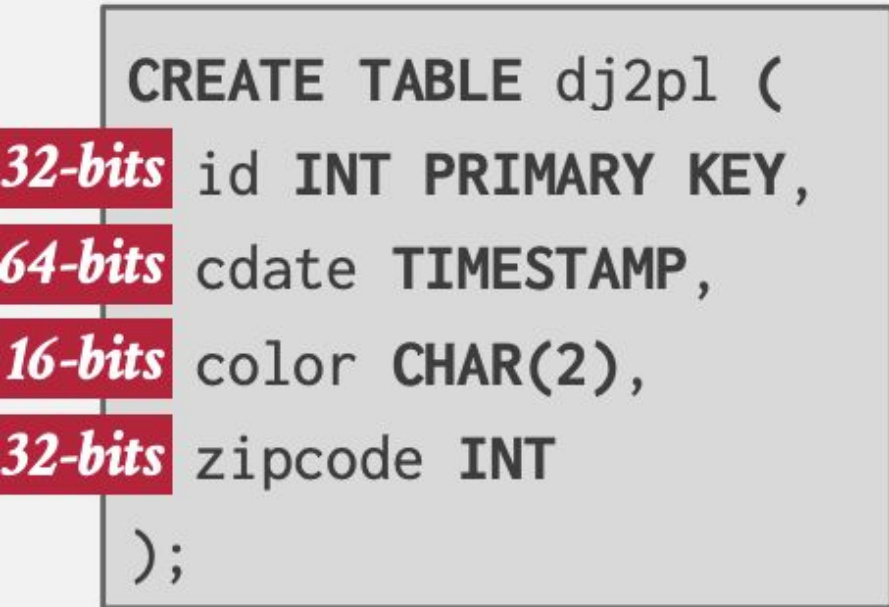
```
CREATE TABLE dj2p1 (  
  32-bits id INT PRIMARY KEY,  
  64-bits cdate TIMESTAMP,  
  16-bits color CHAR(2),  
  32-bits zipcode INT  
);
```



Word-Aligned: Reordering

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

- May still have to use padding.



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>

in #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 = 0.299999999999999998890
```

Fixed Precision Numbers

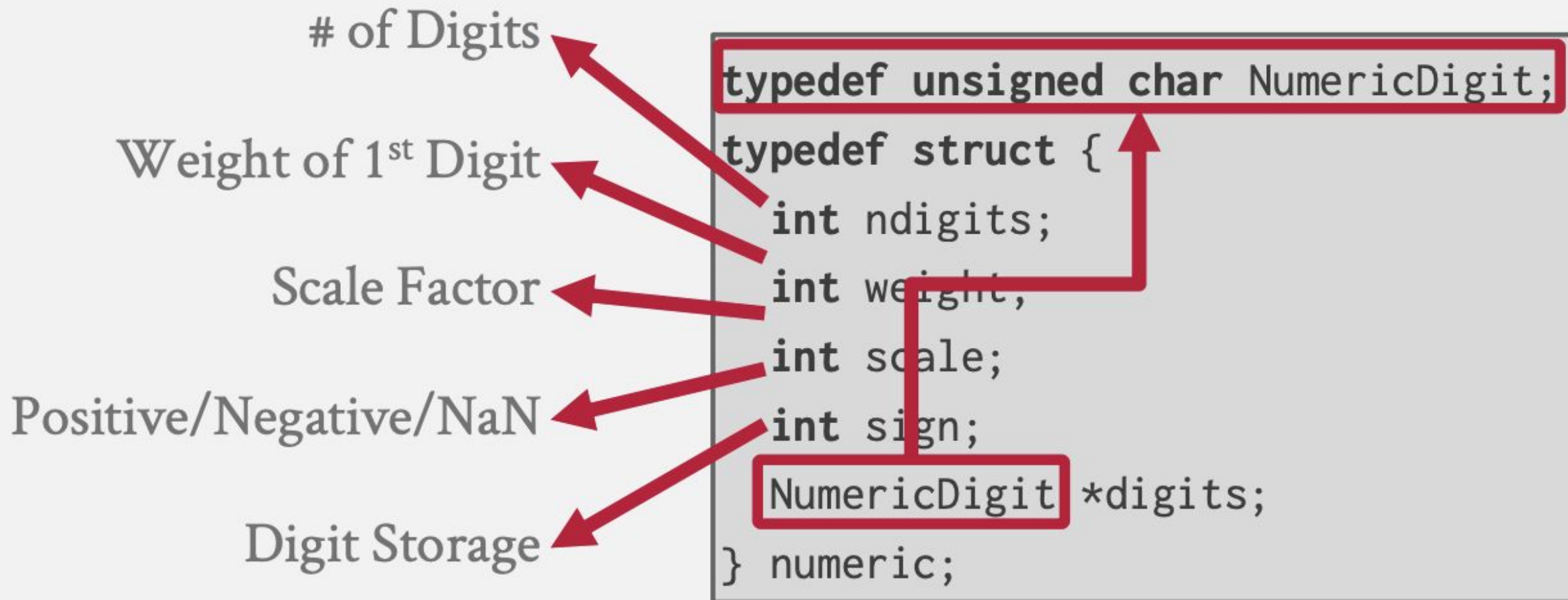
Numeric data types with (potentially) arbitrary precision and scale. Used when rounding errors are unacceptable.

- 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



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.

Choice #2: Special Values

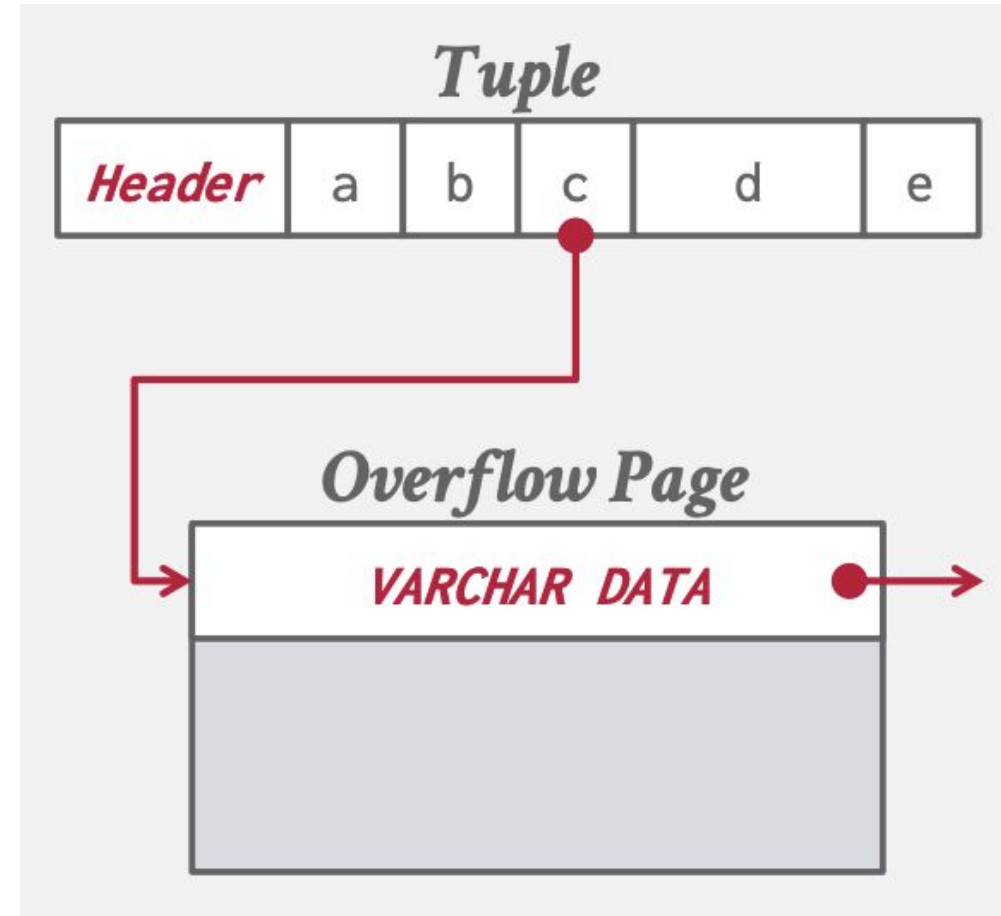
- Designate a value to represent NULL for a data type (e.g., INT32_MIN).

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.

Large Values

- Most DBMSs don't 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)



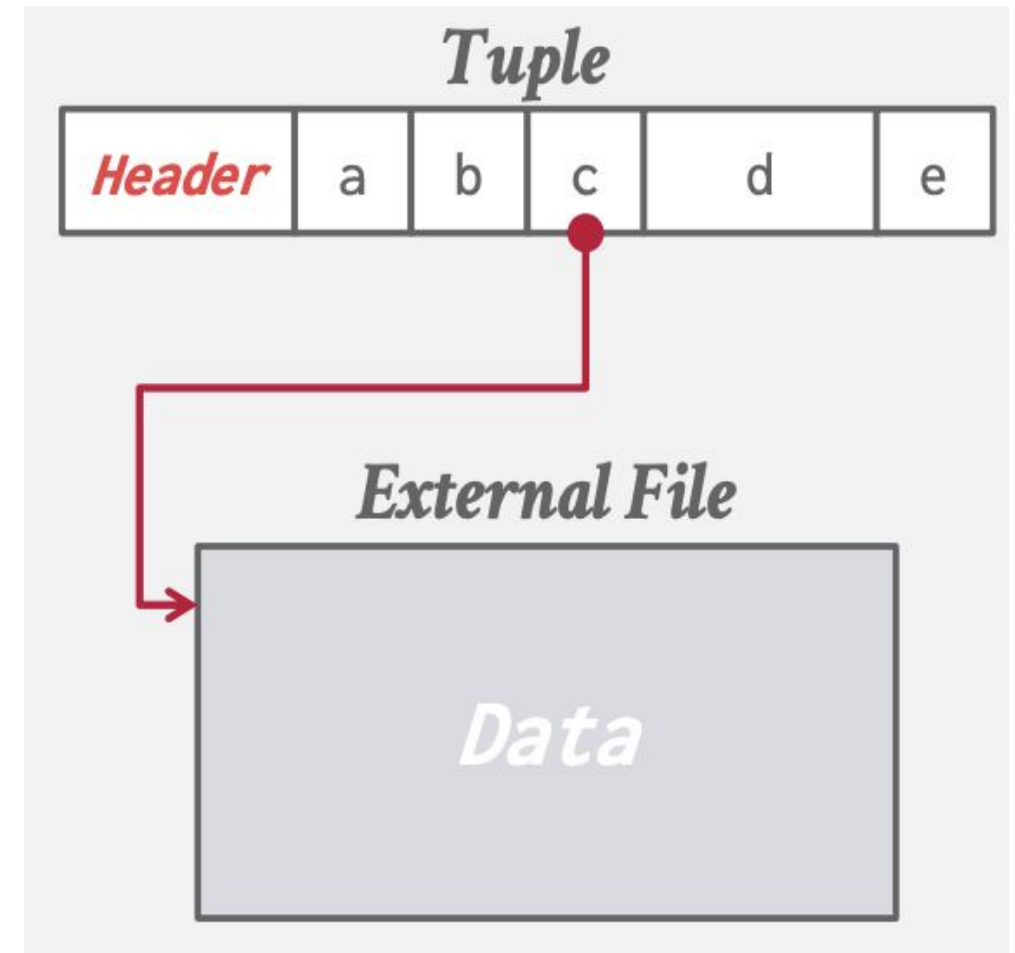
External Value Storage

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.



Conclusion

Log-structured storage is an alternative approach to the page-oriented architecture.

- Ideal for write-heavy workloads because it maximizes sequential disk I/O.

The storage manager is not entirely independent from the rest of the DBMS.

Next Lesson

Unit VII : Database Storage Structures

7.5 Data-Dictionary Storage

7.6 Database Buffer

7.7 Column-Oriented Storage

7.8 Storage Organization in Main-Memory Databases