CSAI 302

Advanced Database Systems

Lec 02

Storage Management

Storage hierarchy

Volatile

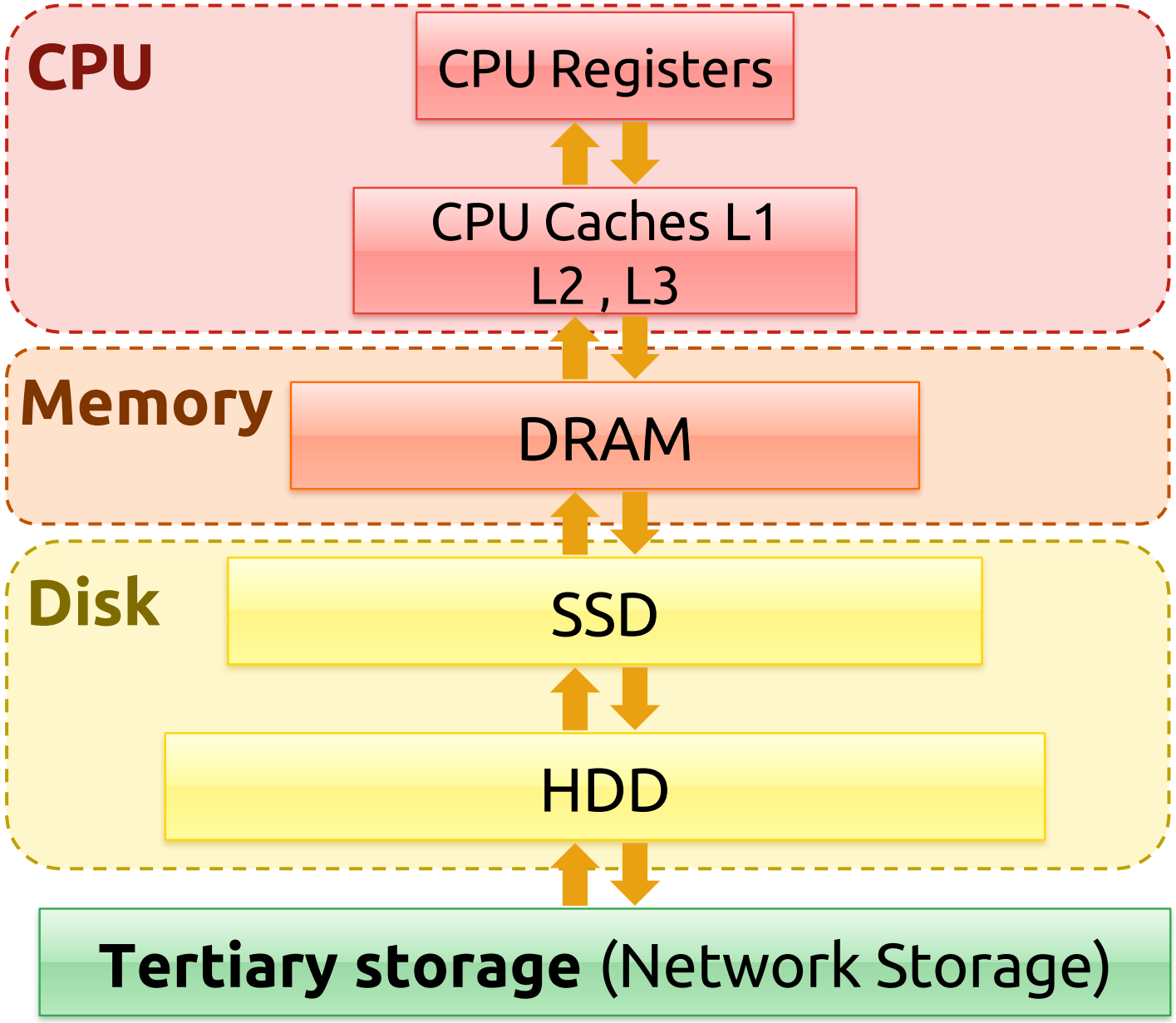
Random Access

Byte-Addressable

Non-Volatile

Sequential Access

Block-Addressable



Faster
Smaller
Expensive

Slower
Larger
Cheaper

Disk-based Architecture

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.
- Lowest layer of DBMS
  - Physical details hidden from higher levels of system
- Purpose
  - Map pages to locations on disk
  - Load pages from disk to memory
  - Save pages back to disk & ensuring write

Terminologies

- Block
  - ★unit of transfer for disk read/write
- Page
  - ★a block-sized chunk of RAM

Files of Pages of Records

- Each table is stored in one or more OS files
- Each file contains many pages
- Each page contains many records
- Pages are understood by multiple layers
  - Managed on disk by the disk space manager:
    - pages read from/written to physical disk/files
  - Managed in memory by the buffer manager:
    - higher levels of DBMS only operate in memory

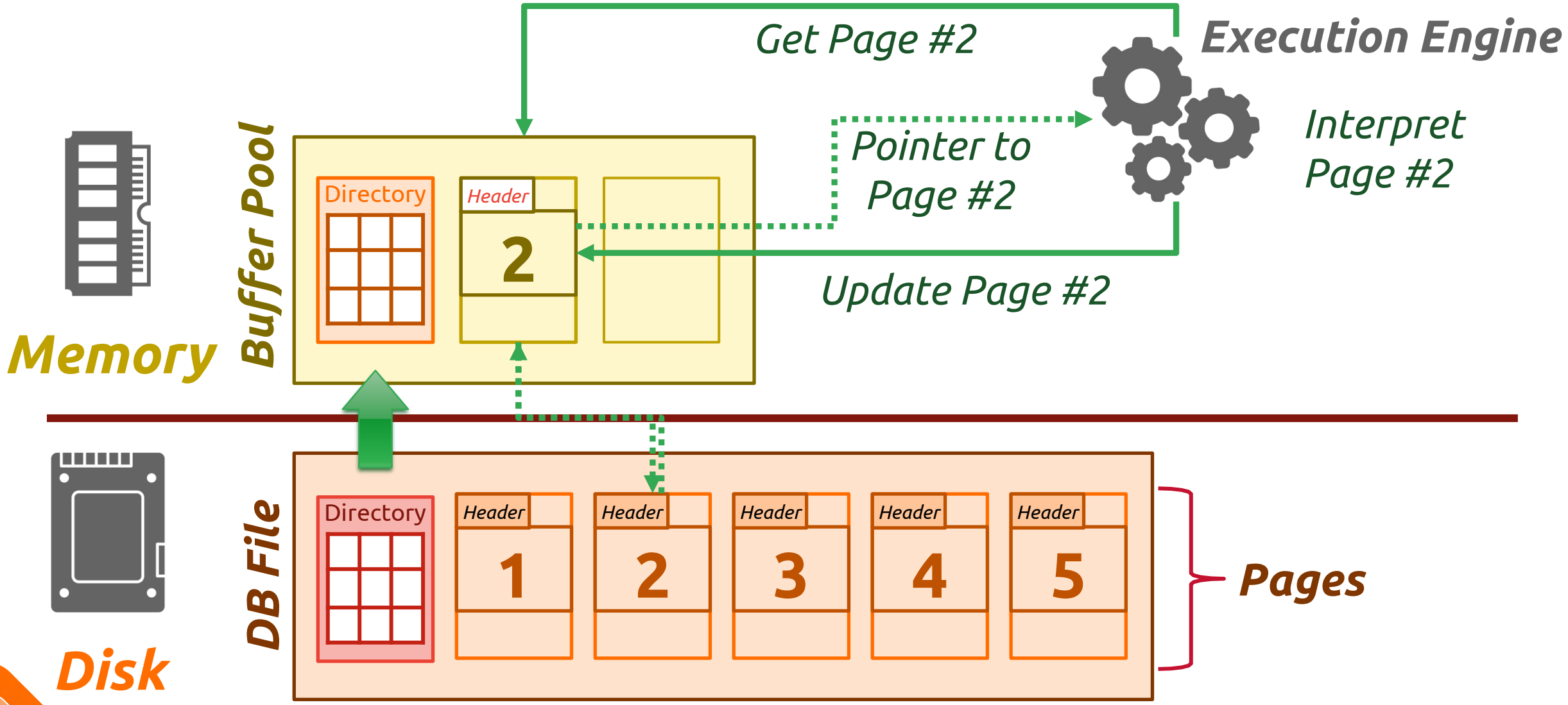
Representing Data Elements

	Data Element	Record	Collection
DBMS	Attribute	Tuple	Relation
File System	Field	Record	File

System design goals

- Allow the DBMS to manage databases that exceed the amount of memory available.
- Reading/writing to disk is expensive
  - It must be managed carefully to avoid large stalls and performance degradation.
- Random access on disk is usually much slower than sequential access
  - the DBMS needs to maximize sequential access.

Disk-oriented DBMS



Database Storage Questions

- How the DBMS represents the database in files on disk?

- 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.
- 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

- Responsible for maintaining a database's files.
- 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…
  - 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.
- DBMSs that specialize in read-only workloads have larger page sizes.

- Default DB Page Sizes
  - **4KB**
    - SQLite
    - ORACLE
    - IBM DB2
    - RocksDB
    - WIREDTIGER
  - **8KB**
    - Microsoft SQL Server
    - PostgreSQL
  - **16KB**
    - MySQL

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 ***not*** need to know anything about what is inside of the pages.

DB File Structures

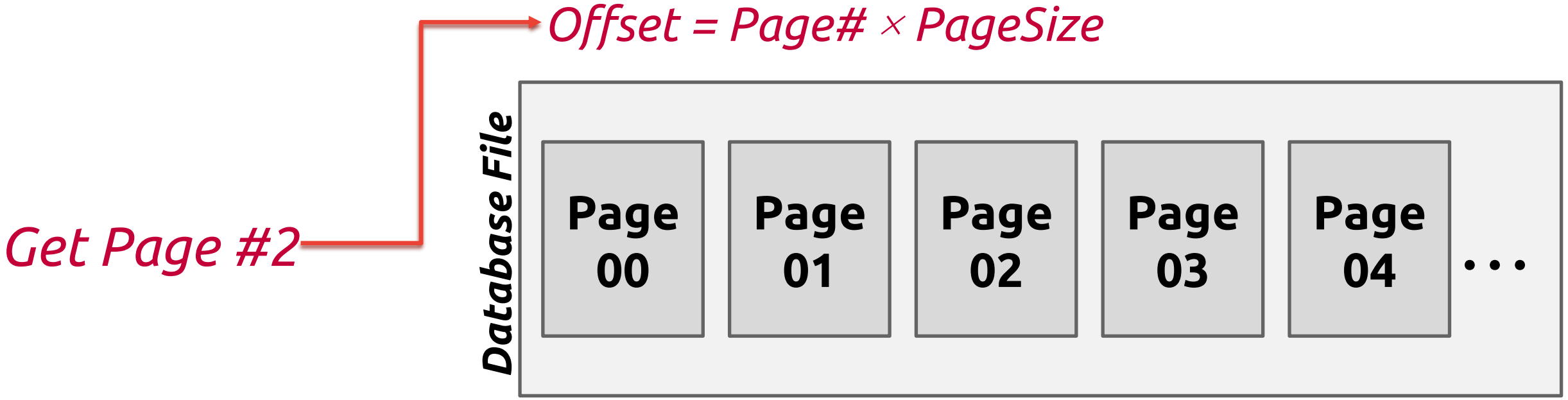
- Unordered heap files
  - Records placed arbitrarily across pages
  - As file shrinks/grows, pages (de)allocated
  - Suitable when typical access is a full scan of all records
- Clustered heap files
  - Group data into blocks to enable fast lookup and efficient modifications
- Sorted files
  - Pages and records are in strict sorted order
  - Best for retrieval in order, or when a range of records is needed
- Index files
  - B+ trees, linear hashing, extensible hashing, ……
  - May contain records or point to records in other files

Heap File Organization

Heap file

- A heap file is an unordered collection of pages with tuples that are stored in random order.
- Need ***additional meta-data*** to track location of files and free space availability.
- Heap files has one special ***header page***
  - ★Two pointers for free space and data
- Location of the heap file and the header page are saved in ***DB catalog***

Heap file

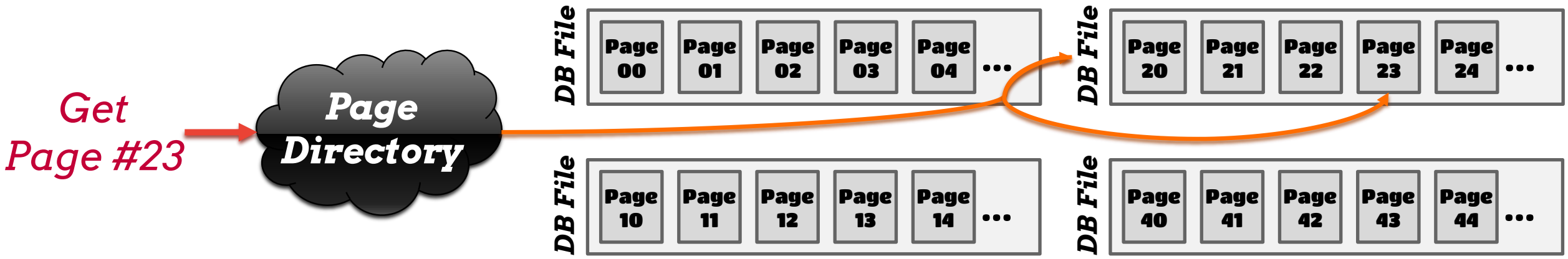

Offset = Page# × PageSize
Get Page #2
Database File
Page 00
Page 01
Page 02
Page 03
Page 04
• • •

Heap file: Page directory

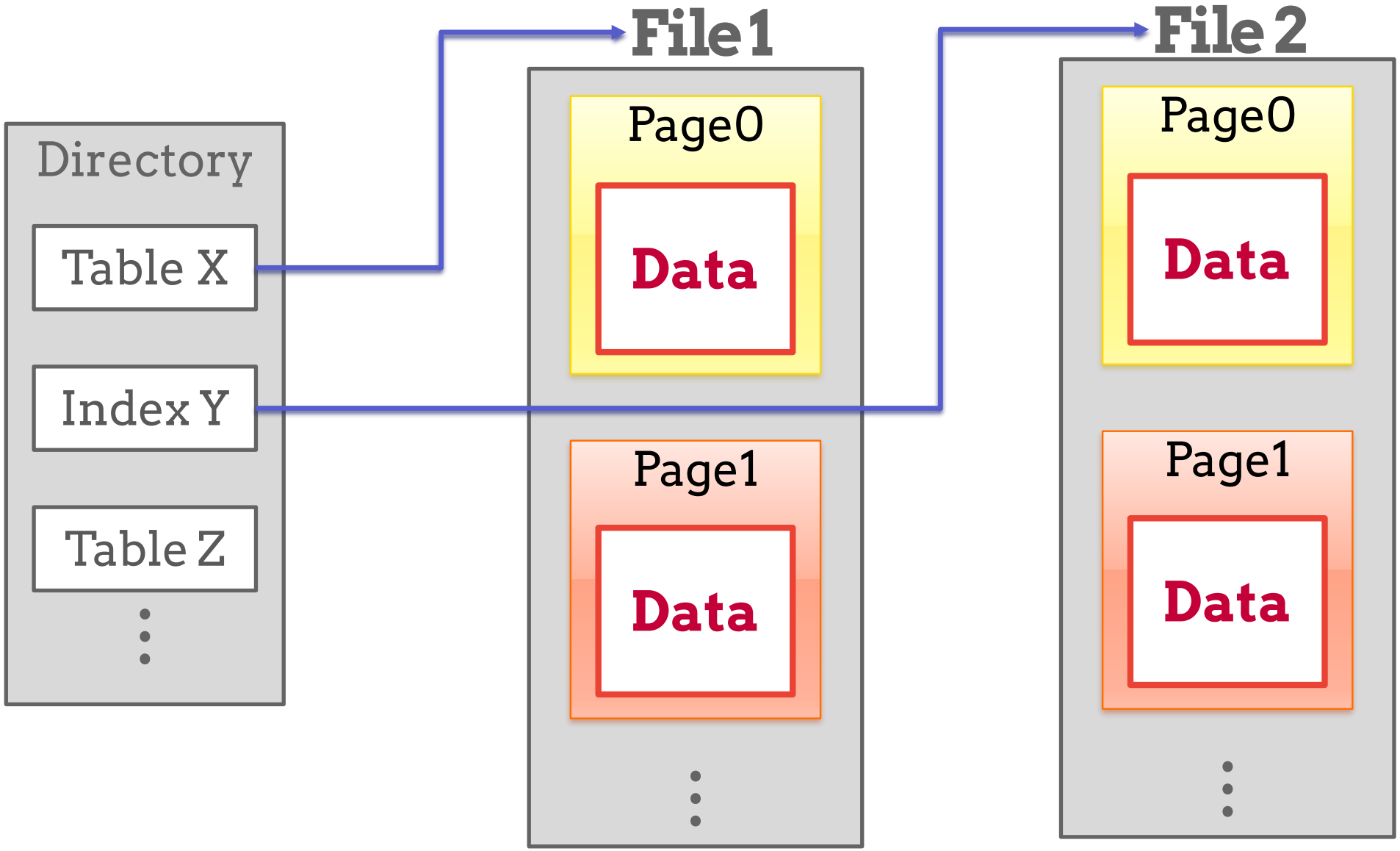
- The DBMS maintains special pages that tracks the ***location of*** data ***pages in*** the database ***files***.
  - One entry per database object.
  - Must make sure that the directory pages are in sync with the data pages.
- DBMS also keeps meta-data about pages' contents:
  - Amount of **free space** per page.
  - List of **free / empty** pages.
  - Page **type** (data vs. meta-data).

Page directory

File Location ⇒ Page# × PageSize

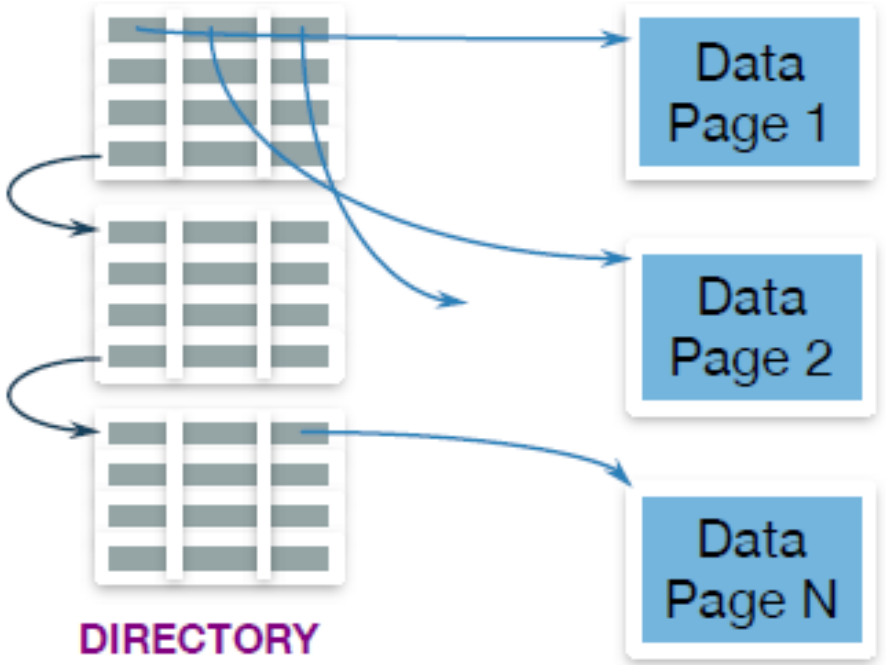


Heap file: Page directory



Unordered Heap Files With Page Directories

- Page directories, with multiple header pages, each encoding
  - A pointer to page
  - # free bytes on the page
- Header pages are accessed often, so likely deployed in cache

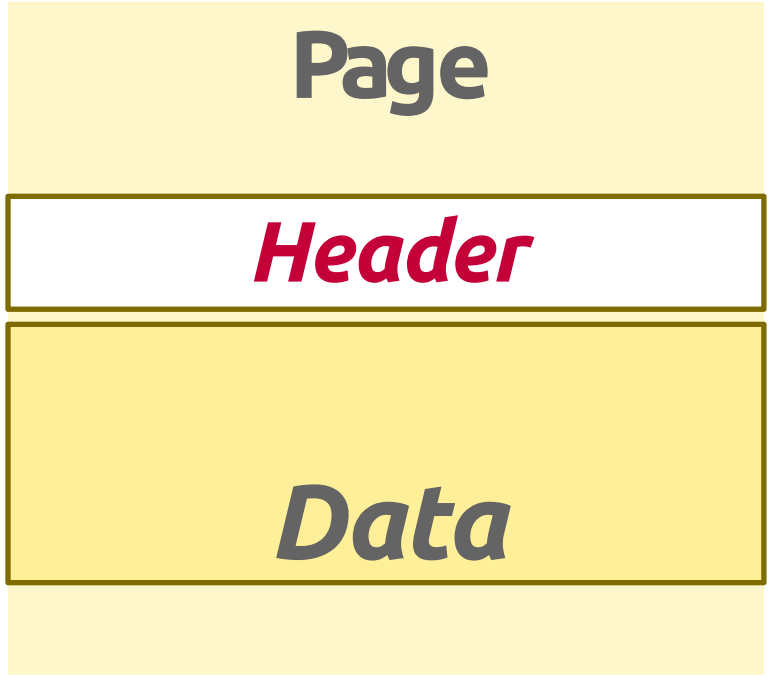


- Finding a page to fit a record can be done efficiently
  - One header page load reveals free space of MANY pages

Page 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).
  - Data and its associated Metadata should reside together on the same physical storage unit.

Page layout

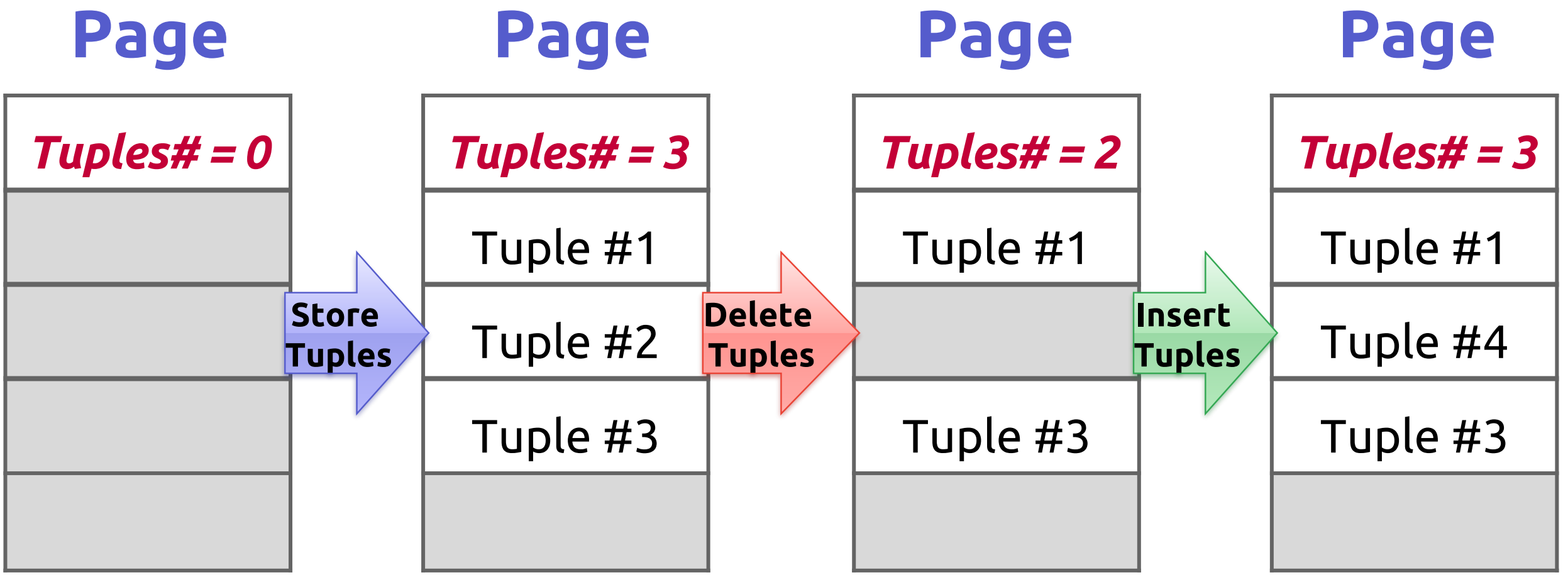
- For any page storage architecture, we now need to decide how to organize the data inside of the page.
  - ★ assuming that we are only storing ***tuples*** in a ***row-oriented storage*** model.

Tuple-oriented Storage	Log-structured Storage	Index-organized Storage

Tuple-oriented storage

- How to store tuples in a page?
  - 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?

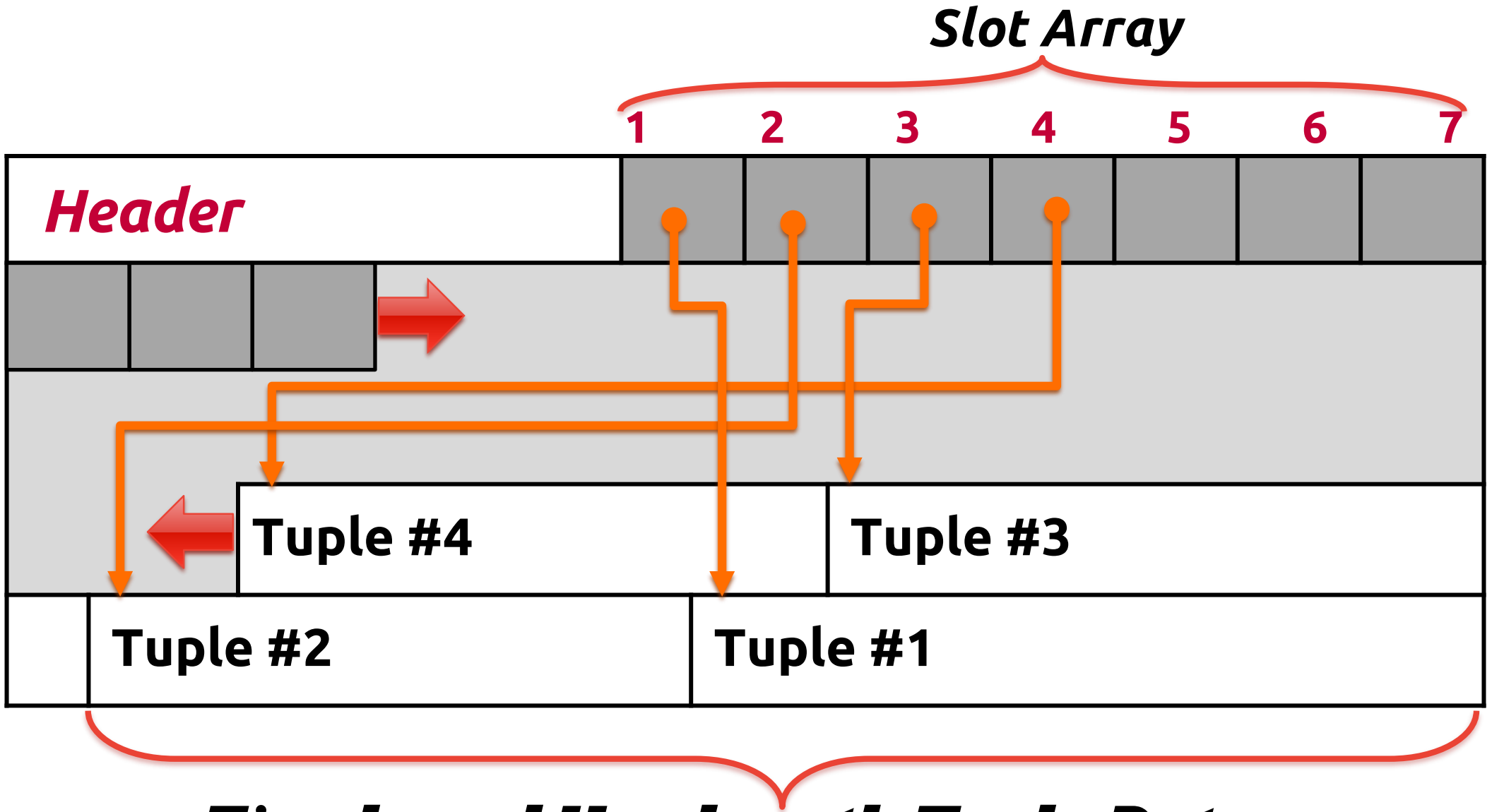
Tuples in a page



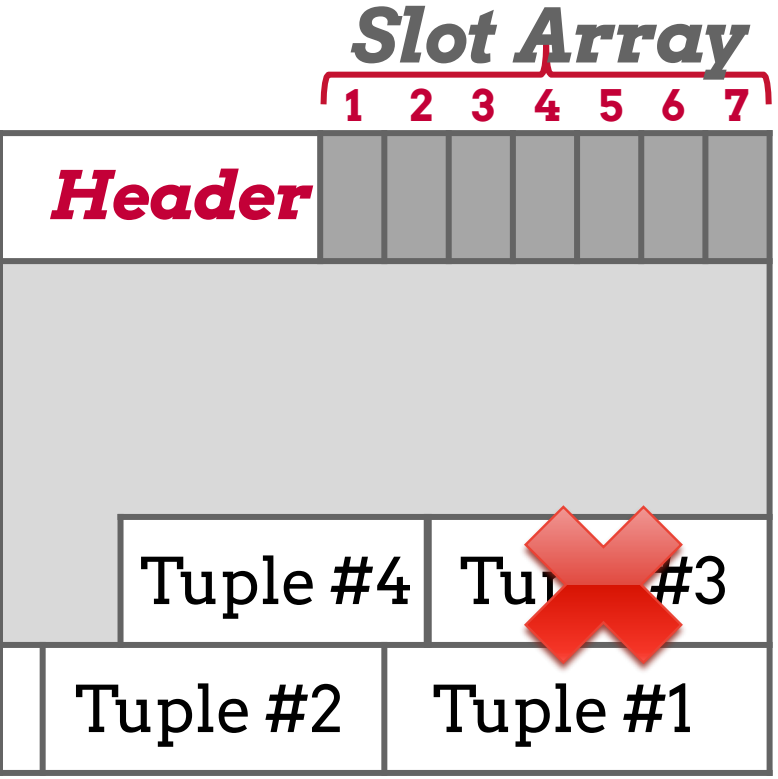
Variable-length attribute

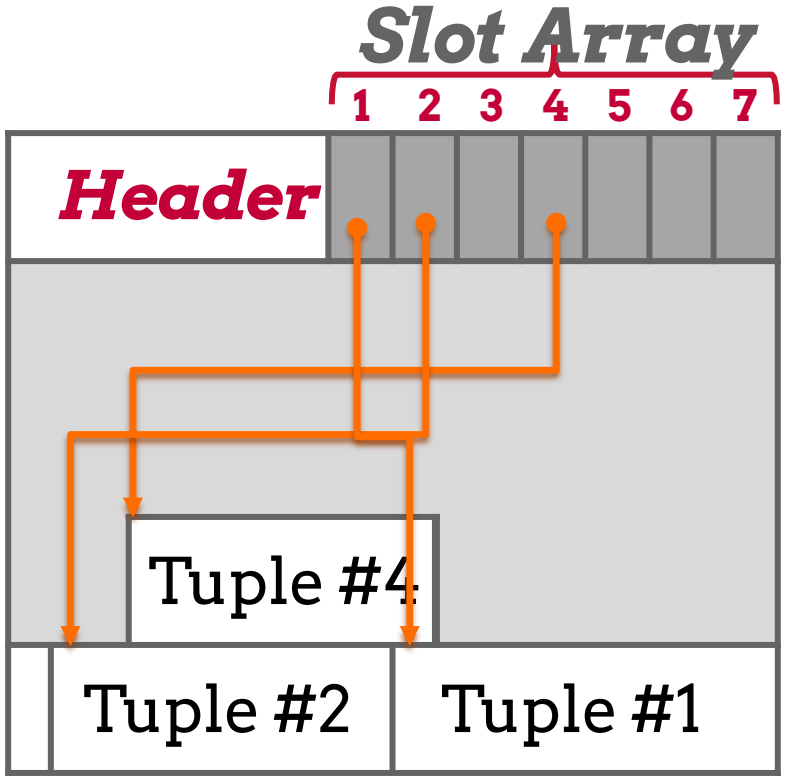
- The most common layout scheme is called ***slotted pages***.
- The slot array maps "slots" to the tuples' starting position offsets.
- The header keeps track of:
  - ★ The # of used slots
  - ★ The offset of the starting location of the last slot used.

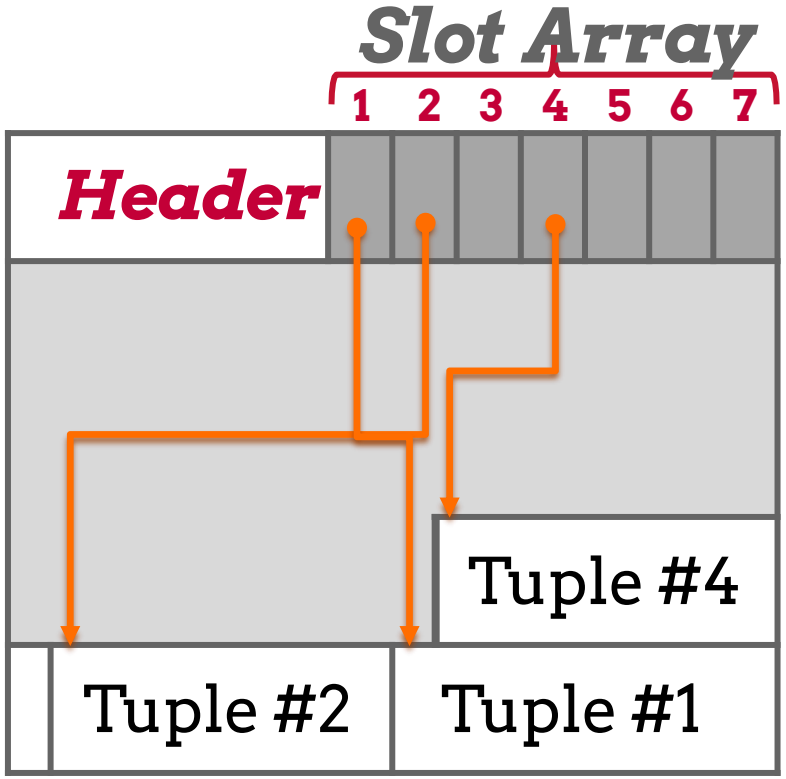
Slotted pages



Delete tuple







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.

Record IDs

CTID
(6-bytes)

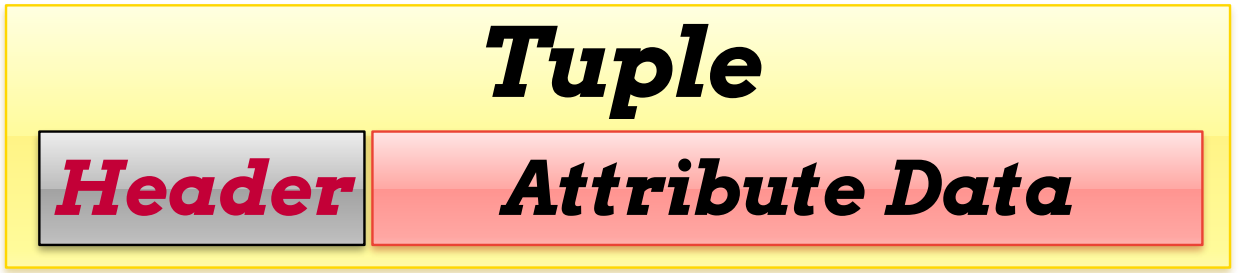
ROWID
(8-bytes)

%%physloc%%
(8-bytes)

ROWID
(10-bytes)

Tuple Layout

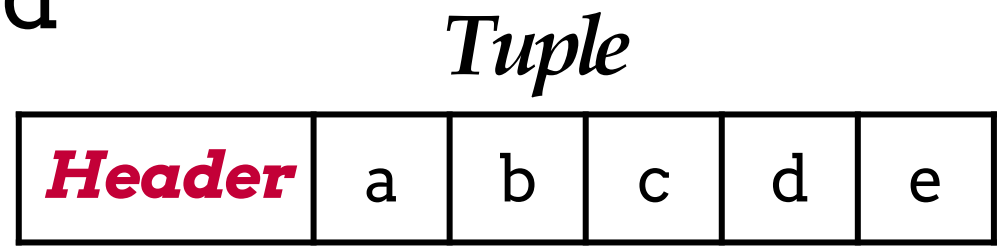
Tuple layout



- A tuple is essentially a sequence of bytes.
  - These bytes do not have to be contiguous.
- It is the job of the DBMS to interpret those bytes into attribute types and values.
- Each tuple is prefixed with a ***header*** that contains meta-data about it.
  - Visibility info (concurrency control)
  - Bit Map for **NULL** 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.




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

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***.

```
CREATE TABLE foo (
a INT PRIMARY KEY,
b INT NOT NULL,
);
```

```
CREATE TABLE bar (
c INT PRIMARY KEY,
a INT REFERENCES foo (a),
);
```

Denormalized tuple data

foo

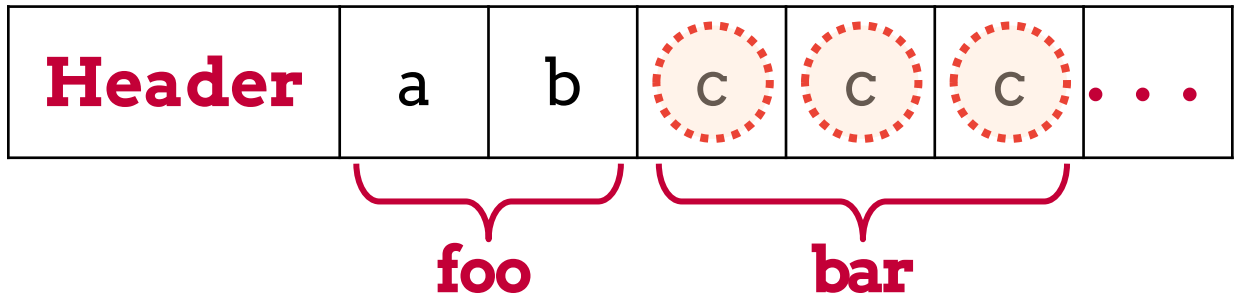
Header	a	b

bar

Header	c	a
Header	c	a
Header	c	a

```
SELECT * FROM foo JOIN bar
ON foo.a = bar.a;
```

foo



Denormalized tuple data

- Not a new idea.
  - IBM System R did this in the 1970s.
  - Several NoSQL DBMSs do this without calling it physical denormalization.




RethinkDB


CouchDB
MarkLogic
RAVENDB
MongoDB

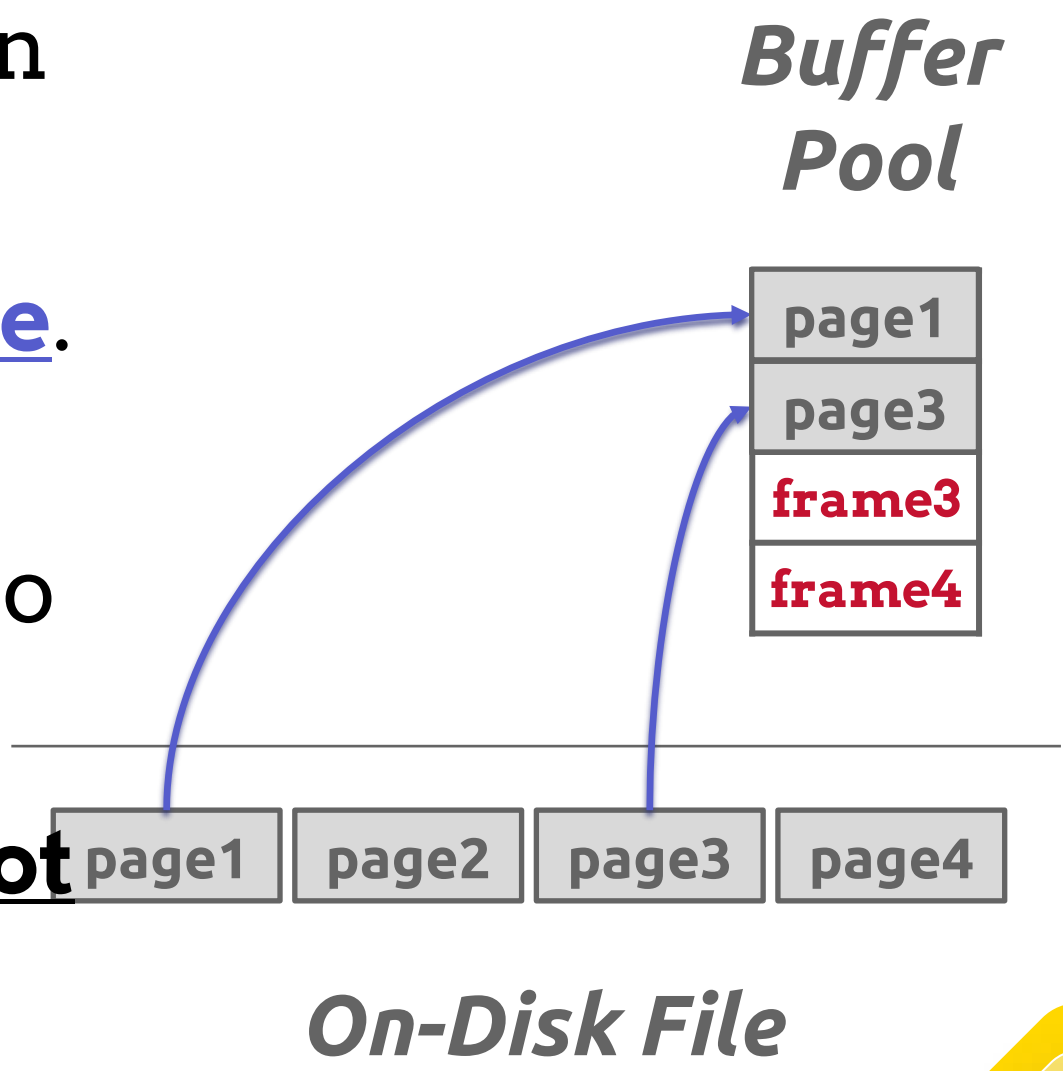
Other Memory Pools

Other memory pools

- The DBMS needs memory for things other than just tuples and indexes.
- These other memory pools may not always backed by disk, depends on implementation.
  - Sorting + Join Buffers
  - Query Caches
  - Maintenance Buffers
  - Log Buffers
  - Dictionary Caches

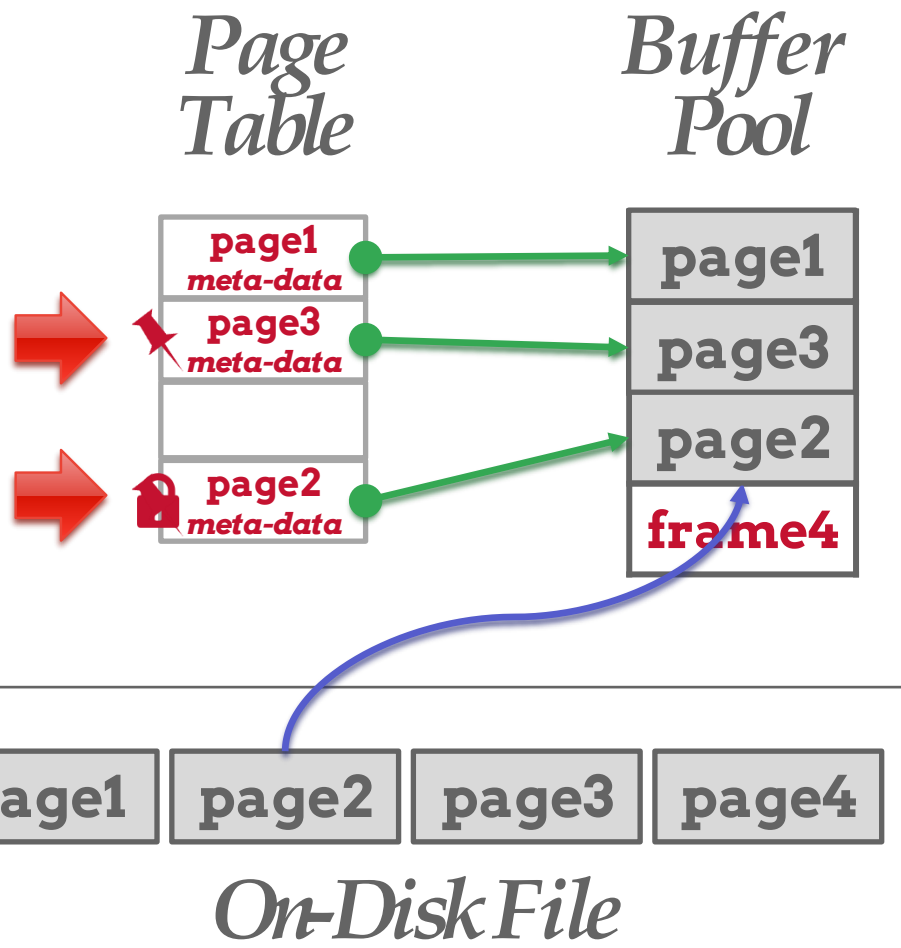
Buffer pool organization

- Memory region organized as an array of ***fixed-size pages***.
- An array entry is called a **frame**.
- When the DBMS requests a page, an exact copy is placed into one of these frames.
- Dirty pages are buffered and **not** written to disk immediately
  - Write-Back Cache



Buffer pool meta-data

- The ***page table*** keeps track of pages
- that are currently in memory.
  - Usually a fixed-size hash table protected with latches to ensure thread-safe access.
- Additional meta-data per page:
  - Dirty Flag
  - Pin/Reference Counter
  - Access Tracking Information



Page table vs. Page directory

Page directory

- The mapping from page ids to page locations in the database files.
  - All changes must be recorded on disk to allow the DBMS to find on restart.

Page table

- The mapping from page ids to a copy of the page in buffer pool frames.
  - This is an in-memory data structure that does not need to be stored on disk.

When a Page is Requested ...

1. If requested page is NOT in the buffer pool:
   1. Choose an **un-pinned** (pin count = 0) frame for replacement
   2. If frame is **dirty**, write current page to disk, mark as "**clean**"
   3. Read requested page into frame
2. Pin the page and return its address

- If requests can be predicted (e.g., sequential scans), pages can be prefetched
  - ★ Several pages at a time
- When the buffer pool is full ...
  - ★ Need to evict an existing page
  - ★ Need a **page replacement policy**

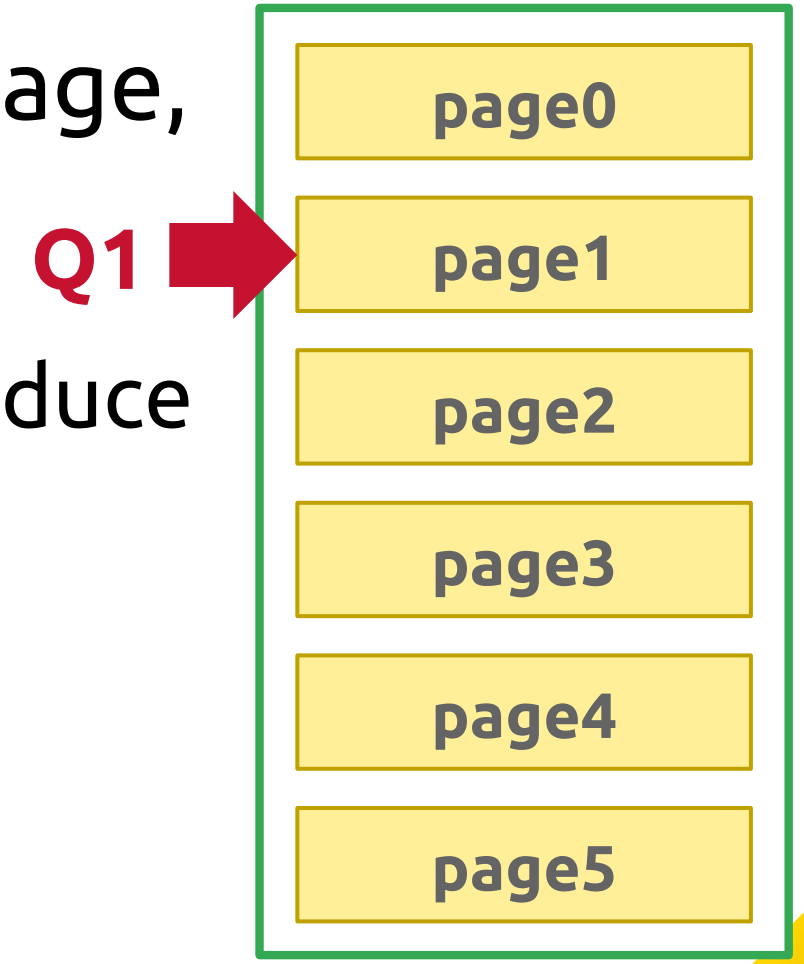
Buffer Replacement

Buffer Replacement Policies

- When the DBMS needs to free up a frame to make room for a new page, it must decide which page to ***evict*** from the buffer pool.
- Goals:
  - Correctness
  - Accuracy
  - Speed
  - Meta-data overhead

LEAST-RECENTLY USED

- Maintain a single timestamp of when each page was last accessed.
- When the DBMS needs to evict a page, select the one with the oldest timestamp.
  - Keep the pages in sorted order to reduce the search time on eviction.



LRU List

LRU Replacement Policy

- Least Recently Used (LRU)
  - ***Pinned frame:*** not available to replace
  - Track time for each frame last unpinned (end of use)
  - Replace the frame which was least recently used
    - Need to "find min" on the last used attributed (priority queue)
  - Very common policy: intuitive and simple
    - Good for repeated access to popular pages (temporal locality)
- Sequential Scan + LRU
  - Lead to **sequential flooding**
    - 0% hit rate in cache
  - Repeated sequential scan is very common in DB workload
    - Nested-loop join

Better Policies: LRU-K

- Track the history of last K references to each page as timestamps and compute the interval between subsequent accesses.
  - Can distinguish between reference types
- Use this history to estimate the next time that page is going to be accessed.
  - Replace the page with the oldest "K-th" access.
  - Balances recency vs. frequency of access.
  - Maintain an ephemeral in-memory cache for recently evicted pages to prevent them from always being evicted.

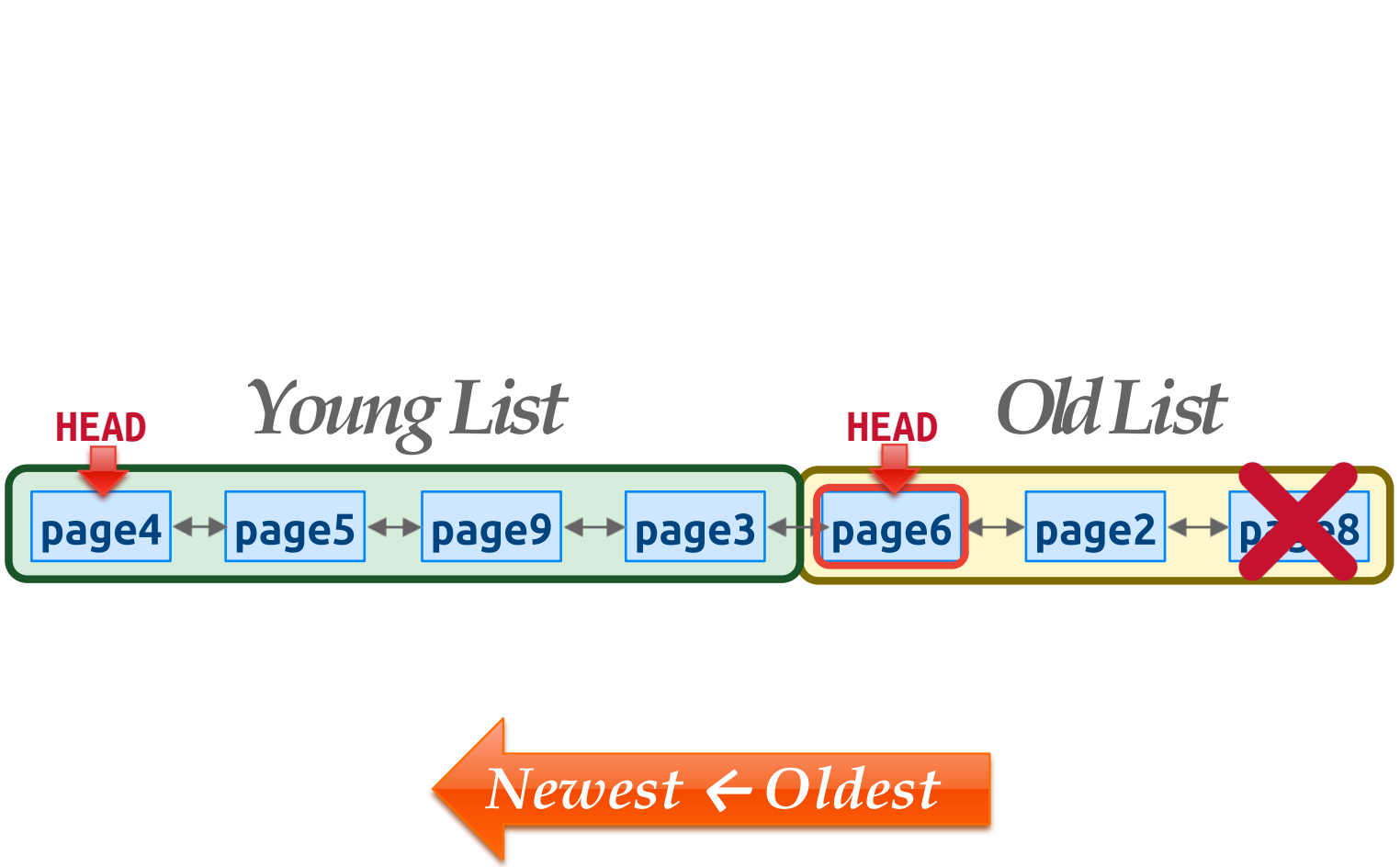
Example: LRU-3

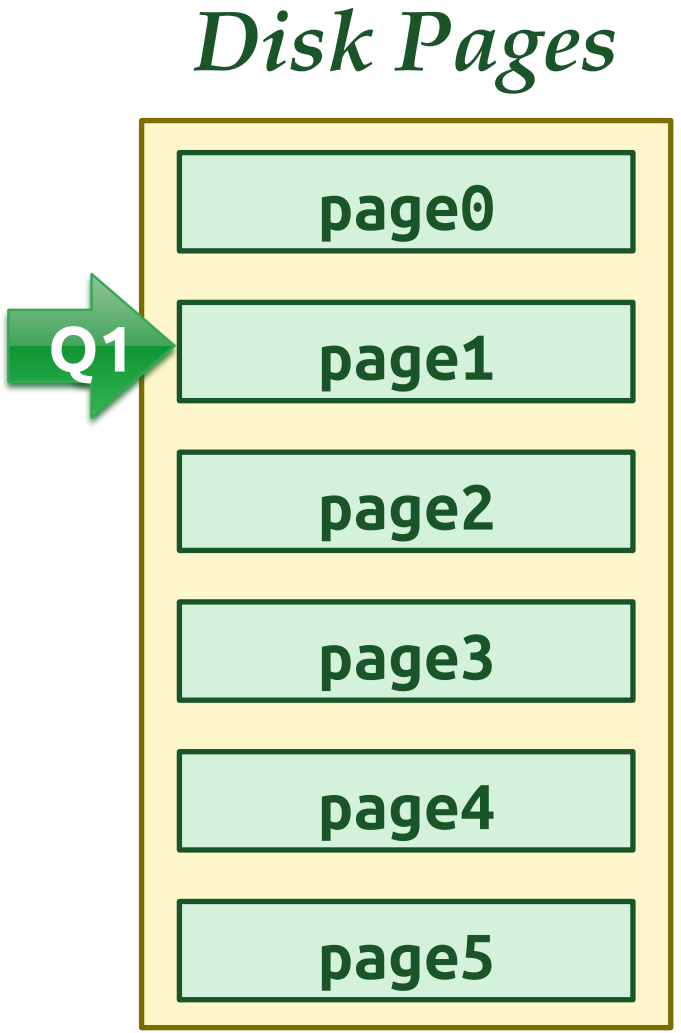
- Page A
  - Access at T=10 , T=40 , T=80
- Page B
  - Access at T=5 , T=30 , T=90

MySQL approximate LRU-K

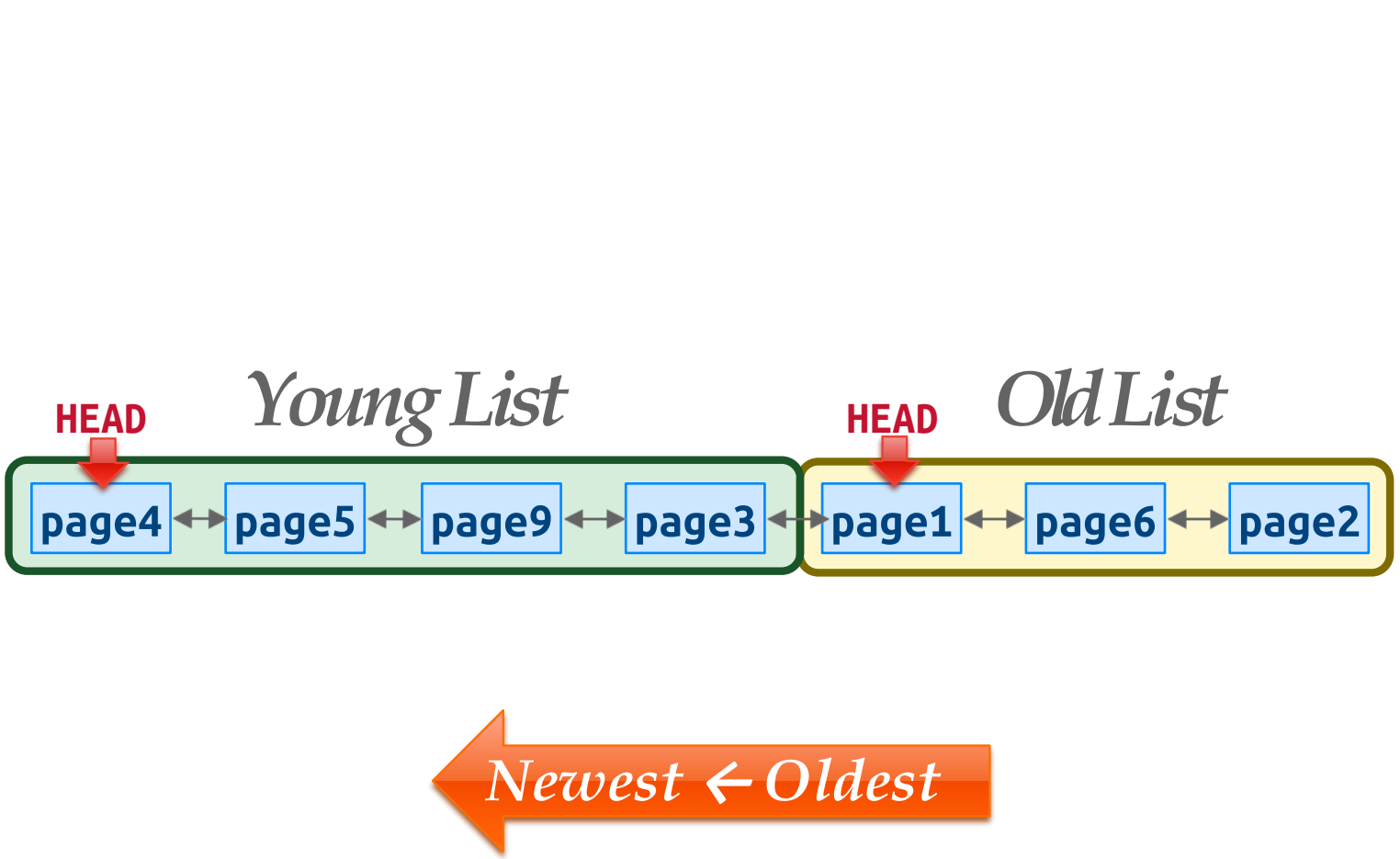
- Single LRU linked list but with two entry points ("old" vs "young").
  - New pages are always inserted to the head of the old list.
  - If pages in the old list is accessed again,
    - then insert into the head of the young list.

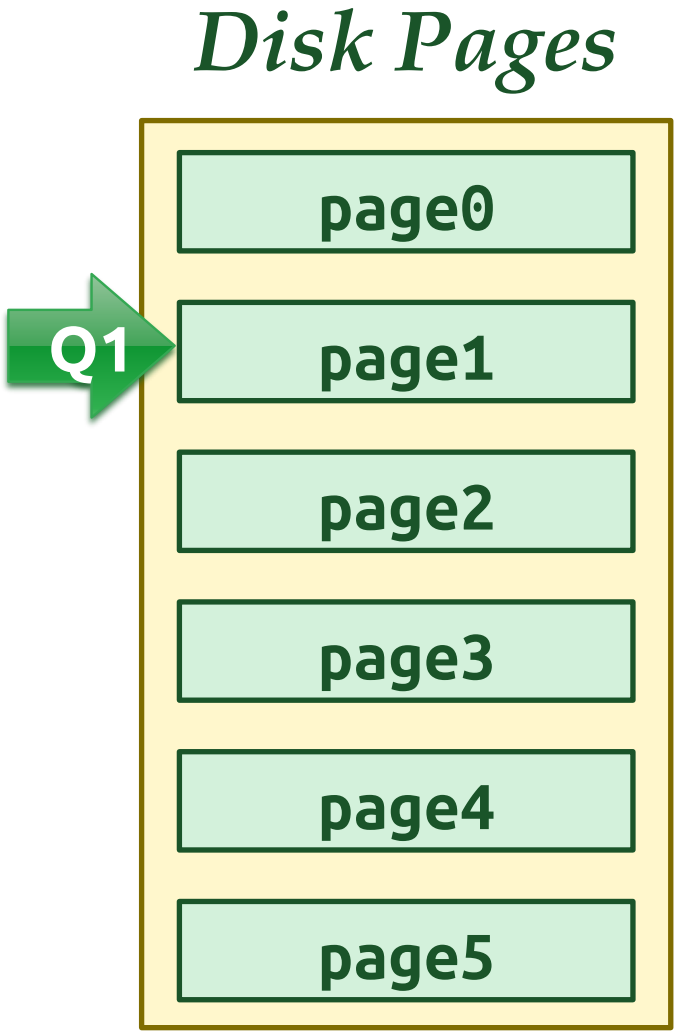
MySQL approximate LRU-K



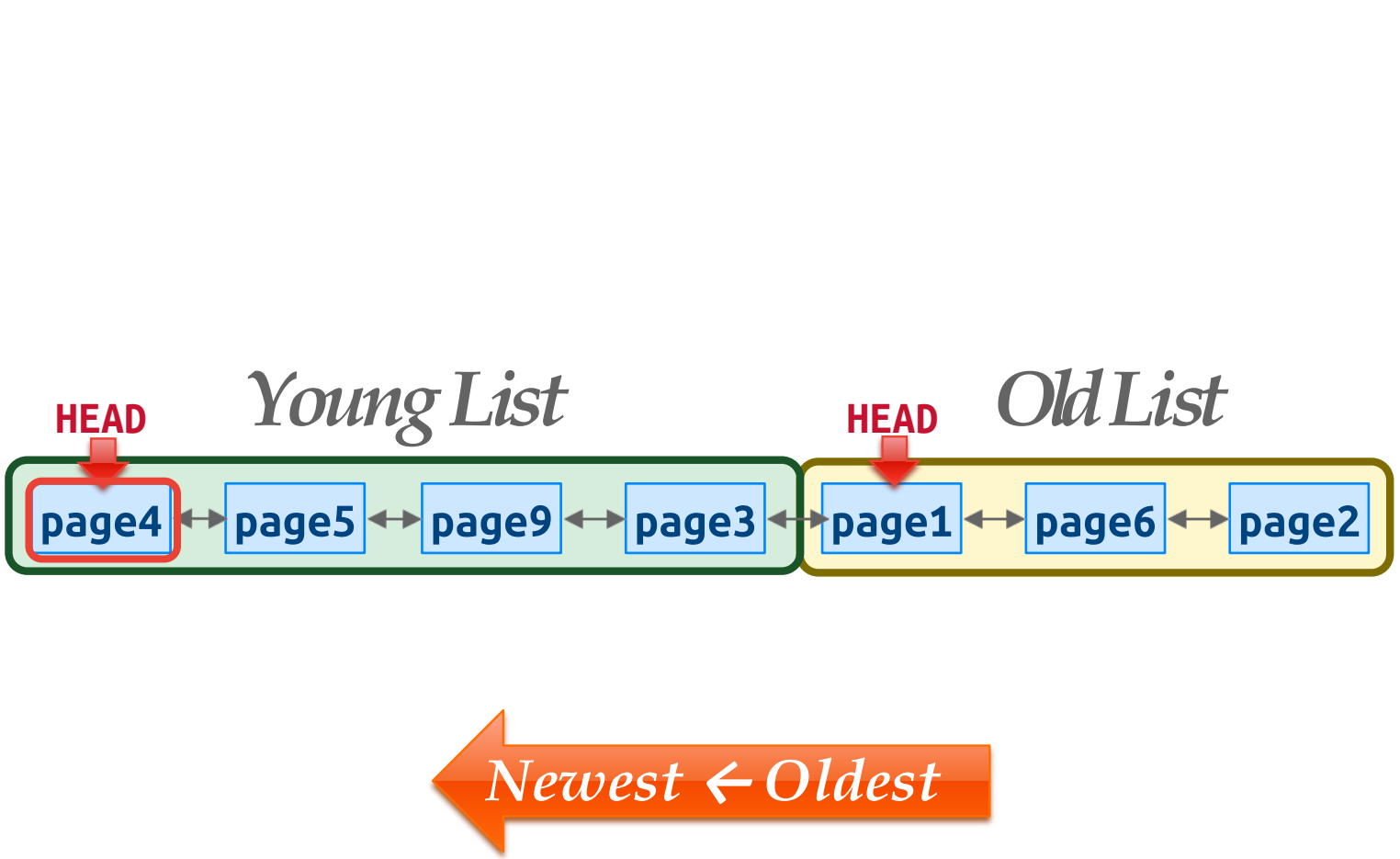


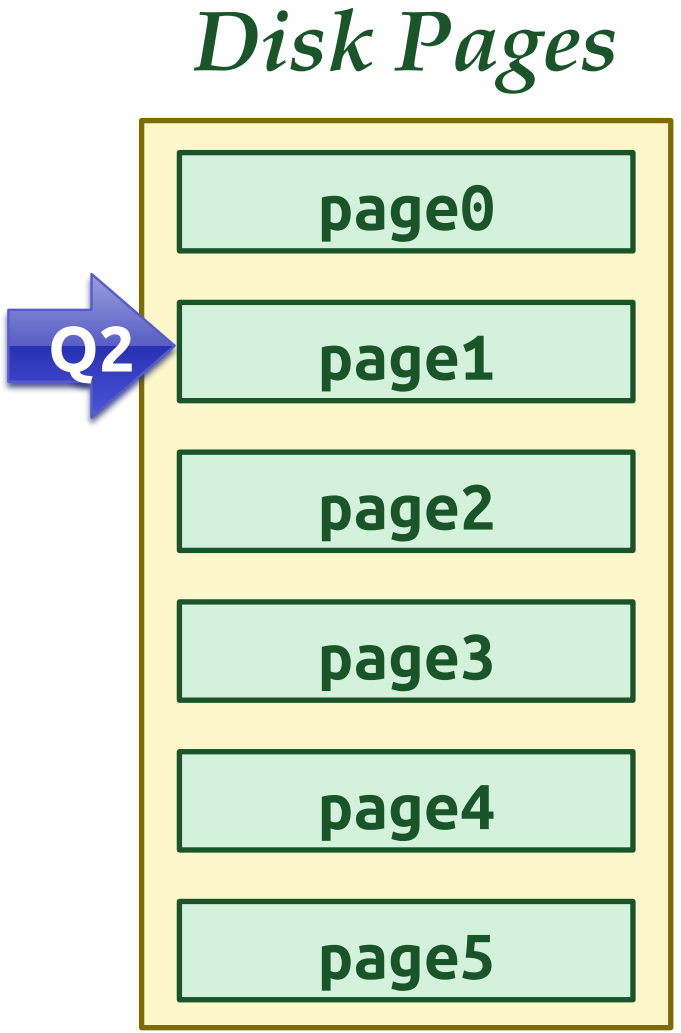
MySQL approximate LRU-K



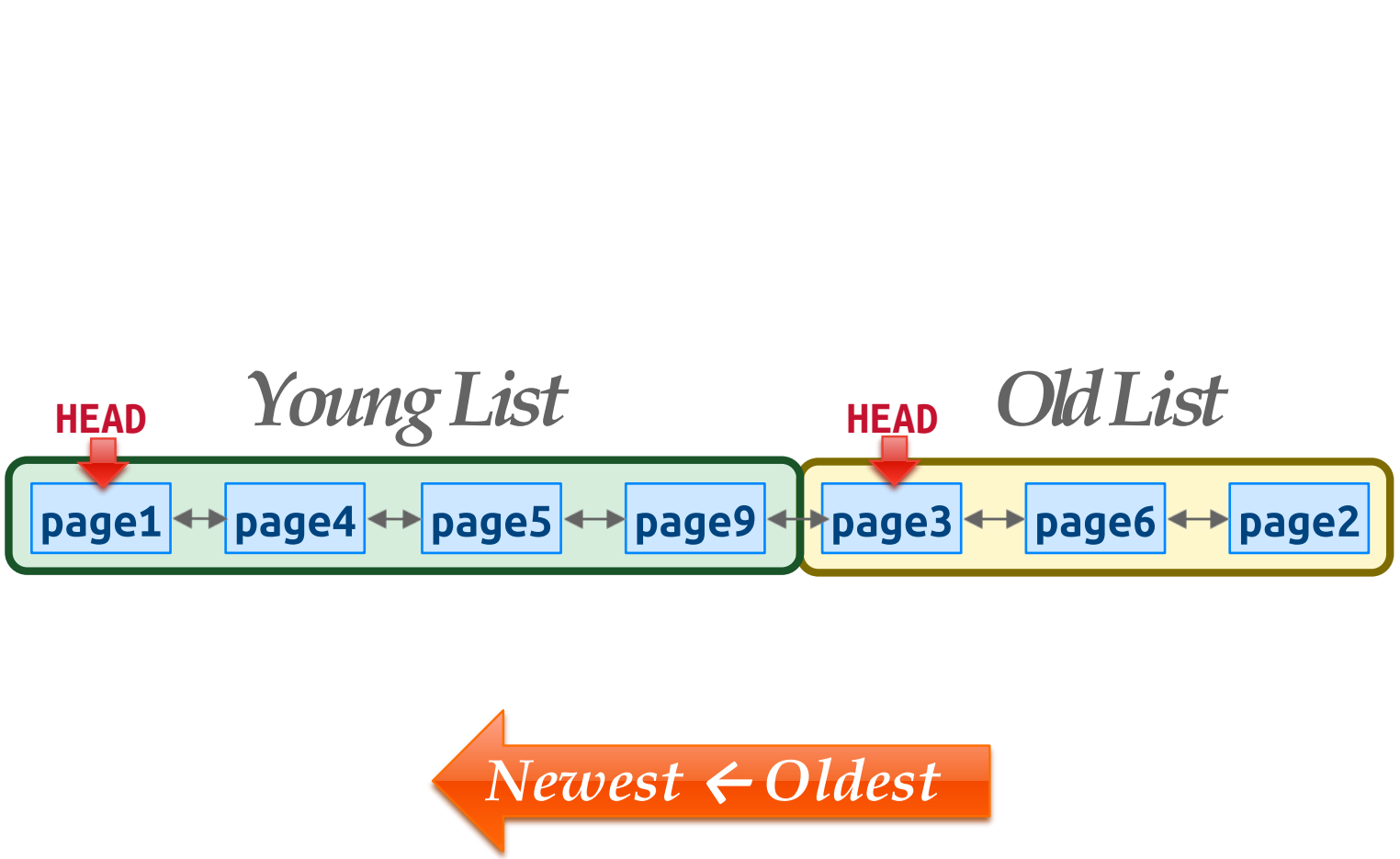


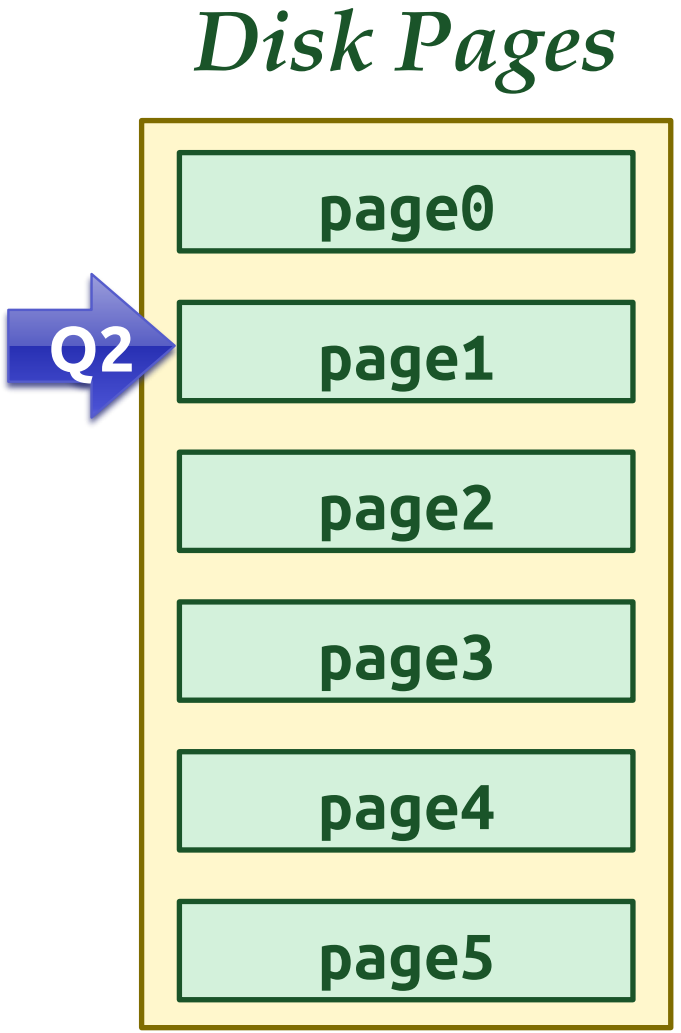
MySQL approximate LRU-K





MySQL approximate LRU-K





Better policies: Localization

- The DBMS chooses which pages to evict on a per query basis.
- This minimizes the pollution of the buffer pool from each query.
  - ★ Keep track of the pages that a query has accessed.
- Example
  - ★ Postgres assigns a limited number of buffer pool pages to a query and uses it as a ***circular ring buffer***.

Better policies: Priority hints

- The DBMS knows about the context of each page during query execution.
- It can provide hints to the buffer pool on whether a page is important or not.

Dirty Pages

- Fast Path
  - If a page in the buffer pool is ***not*** dirty, then the DBMS can simply "drop" it.
- Slow Path
  - If a page is **dirty**, then the DBMS must write back to disk to ensure that its changes are persisted.
- Trade-off between ***fast evictions*** versus dirty writing pages that will ***not be read again*** in the future.

Background writing

- The DBMS can periodically walk through the page table and write dirty pages to disk.
- When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.
- Need to be careful that the system doesn't write dirty pages before their log records are written.

Buffer Pool Optimizations

Multiple Buffer Pools

Pre-Fetching

Scan Sharing

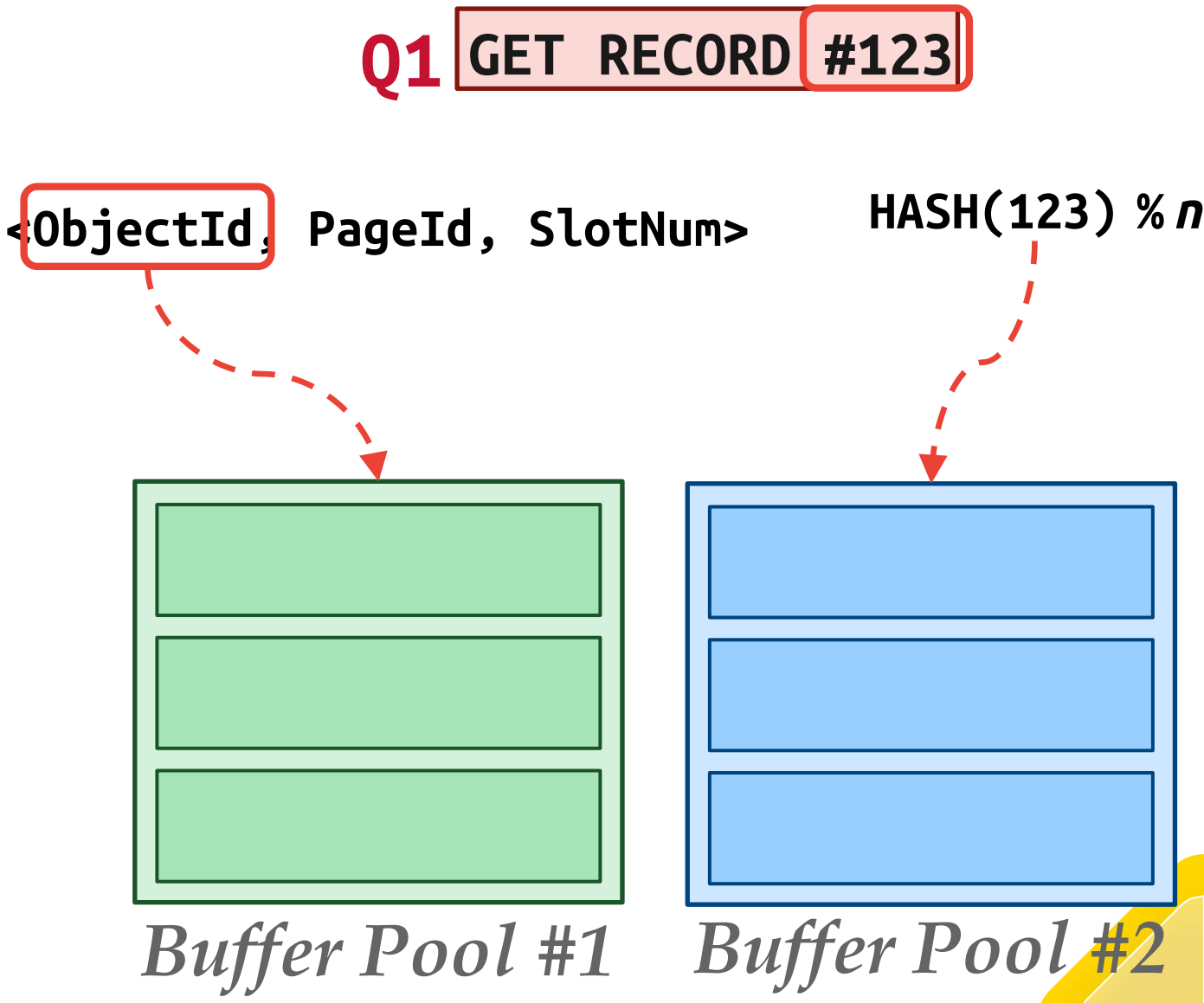
Buffer Pool Bypass

Multiple buffer pools

- The DBMS does not always have a single buffer pool for the entire system.
  - Multiple buffer pool instances
  - Per-database buffer pool
  - Per-page type buffer pool
- Partitioning memory across multiple pools helps reduce latch contention and improve locality.
  - Avoids contention on LRU tracking meta-data.

Multiple buffer pools

- Object Id
  - Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.
- Hashing
  - Hash the page id to select which buffer pool to access.



Pre-fetching

- The DBMS can also prefetch pages based on a query plan.
  - Examples: Sequential vs. Index Scans
- Some DBMS prefetch to fill in empty frames upon start-up.

THANK YOU