Georgia Tech

DATABASE SYSTEM IMPLEMENTATION

GT 4420/6422 // SPRING 2019 // @JOY_ARULRAJ

LECTURE #2: IN-MEMORY DATABASES

CREATING THE NEXT®

TODAY'S AGENDA

Background
In-Memory DBMS Architectures
Early Notable In-Memory DBMSs

LAST CLASS

History of DBMSs

→ In a way though, it really was a history of data models

Data Models

→ Hierarchical data model (tree) (IMS)
→ Network data model (graph) (CODASYL)
→ Relational data model (tables) (System R, INGRES)

Overarching theme about all these systems

→ They were all disk-based DBMSs

BACKGROUND

Much of the history of DBMSs is about dealing with the limitations of hardware.

Hardware was much different when the original DBMSs were designed:

→ Uniprocessor (single-core CPU)
→ RAM was severely limited (few MB).
→ The database had to be stored on disk.
→ Disk is slow. No seriously, I mean really slow.

BACKGROUND

But now DRAM capacities are large enough that most databases can fit in memory.

→ Structured data sets are smaller (e.g., tables with schema).

→ Unstructured or semi-structured data sets are larger (e.g., videos, log files).

So why not just use a "traditional" disk-oriented DBMS with a really large cache?

DISK-ORIENTED DBMS

The primary storage location of the database is on non-volatile storage (e.g., HDD, SSD).

→ The database is stored in a **file** as a collection of fixed-length blocks called **slotted pages** on disk.

The system uses an in-memory (volatile) buffer pool to cache blocks fetched from disk.

→ Its job is to manage the movement of those blocks back and forth between disk and memory.

BUFFER POOL

When a query accesses a page, the DBMS checks to see if that page is already in memory:

→ If it's not, then the DBMS has to retrieve it from disk and copy it into a free frame in the buffer pool.

→ If there are no free frames, then find a page to evict guided by the **page replacement policy**.

→ If the page being evicted is dirty, then the DBMS has to write it back to disk to ensure the **durability** (ACI**D**) of data.

PAGE REPLACEMENT POLICY

Page replacement policy is a differentiating factor between open-source and commercial DBMSs.

→ What kind of data does it contain?
→ Is the page dirty?
→ How likely is the page to be accessed in the near future?
→ Examples: LRU, LFU, CLOCK, ARC (Adaptive Replacement Cache)

More Information on Page Replacement Policies: Wikipedia

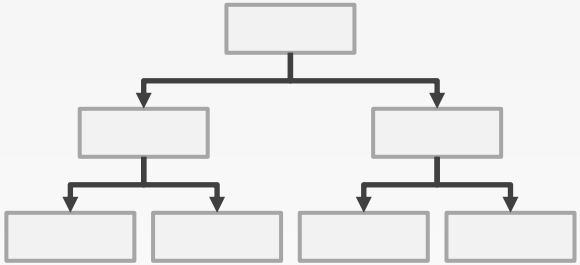
BUFFER POOL

Once the page is in memory, the DBMS translates any on-disk addresses to their in-memory addresses.

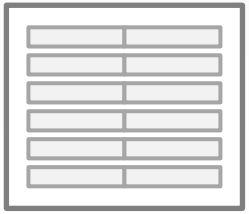
(Page Identifier)	(Page Pointer)
[#100]	[0x5050]

DATA ORGANIZATION

Index

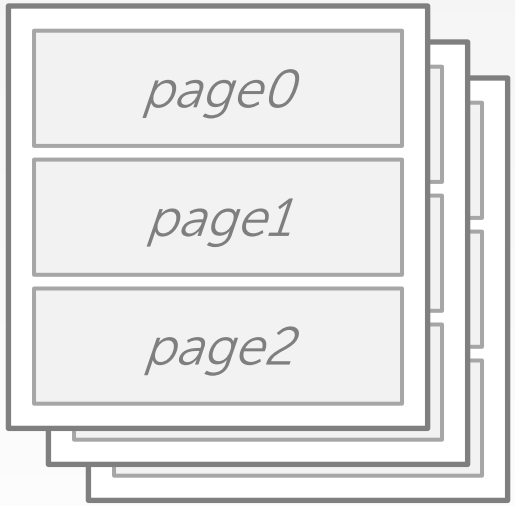


Page Table



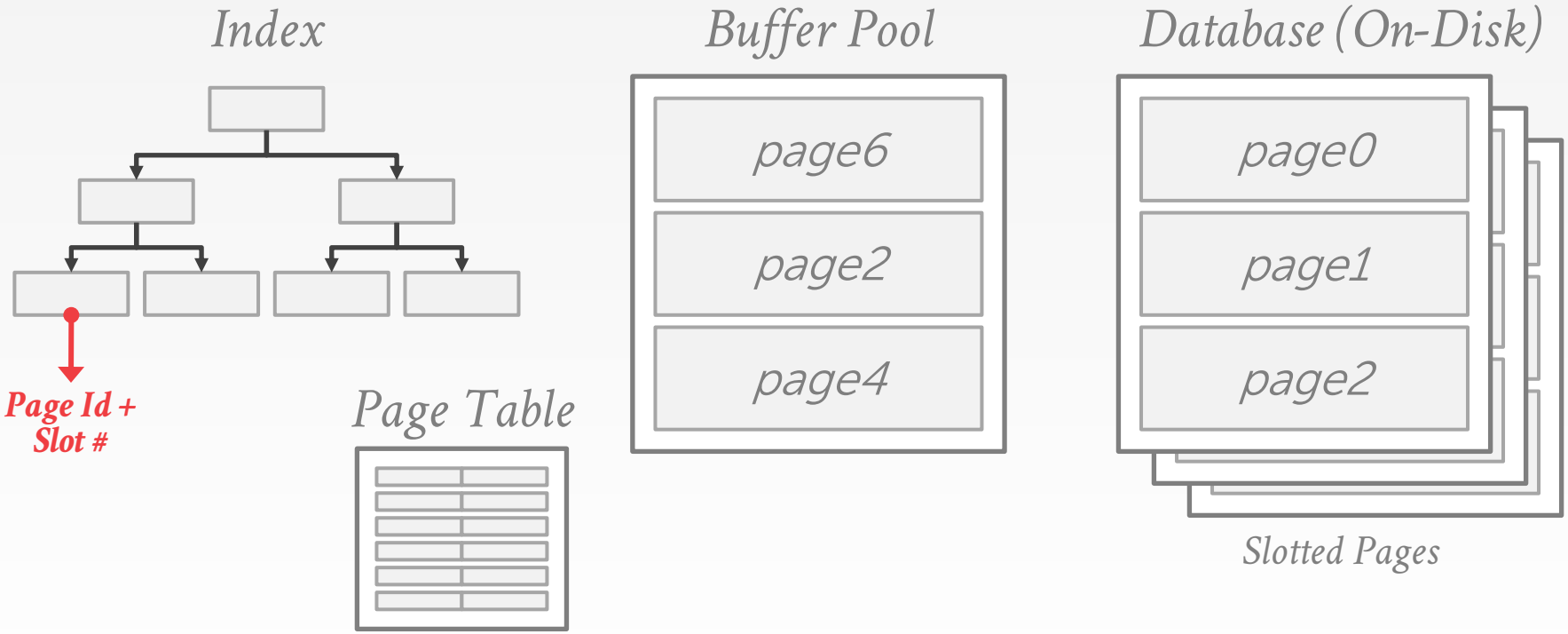
Buffer Pool

Database (On-Disk)

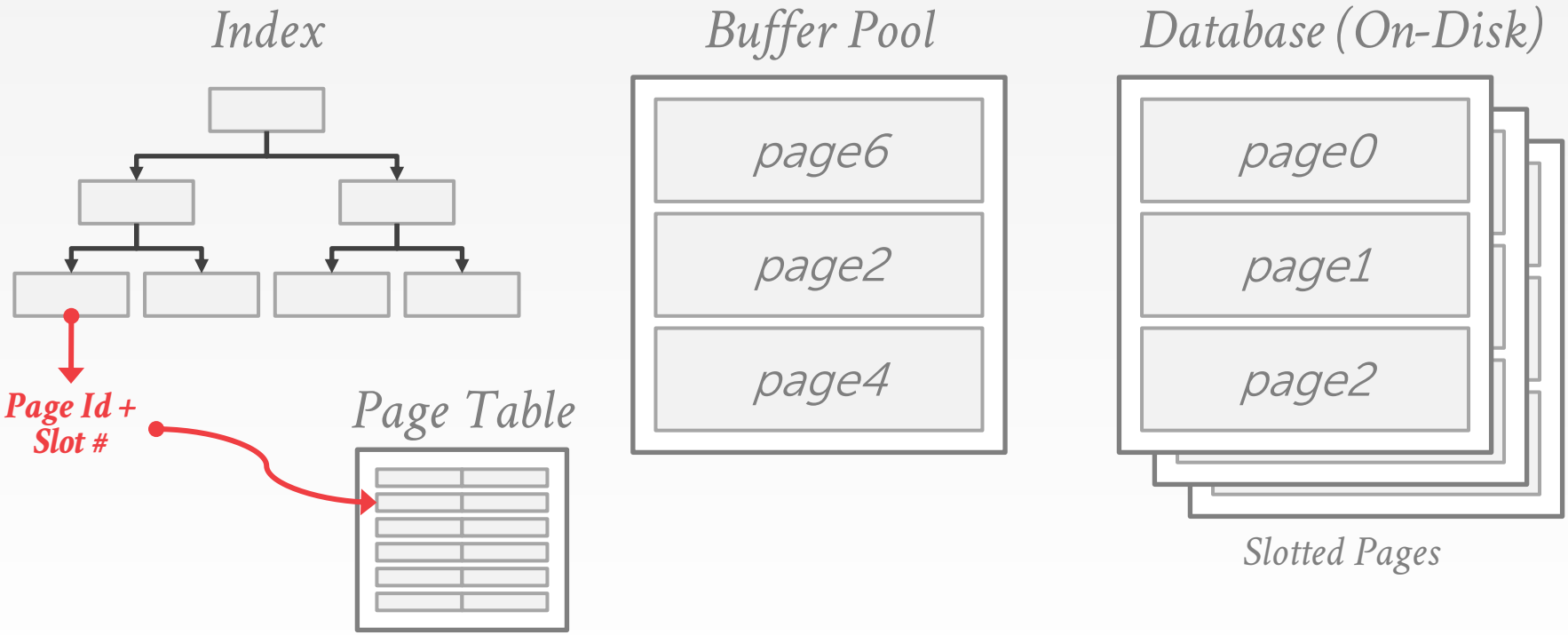


Slotted Pages

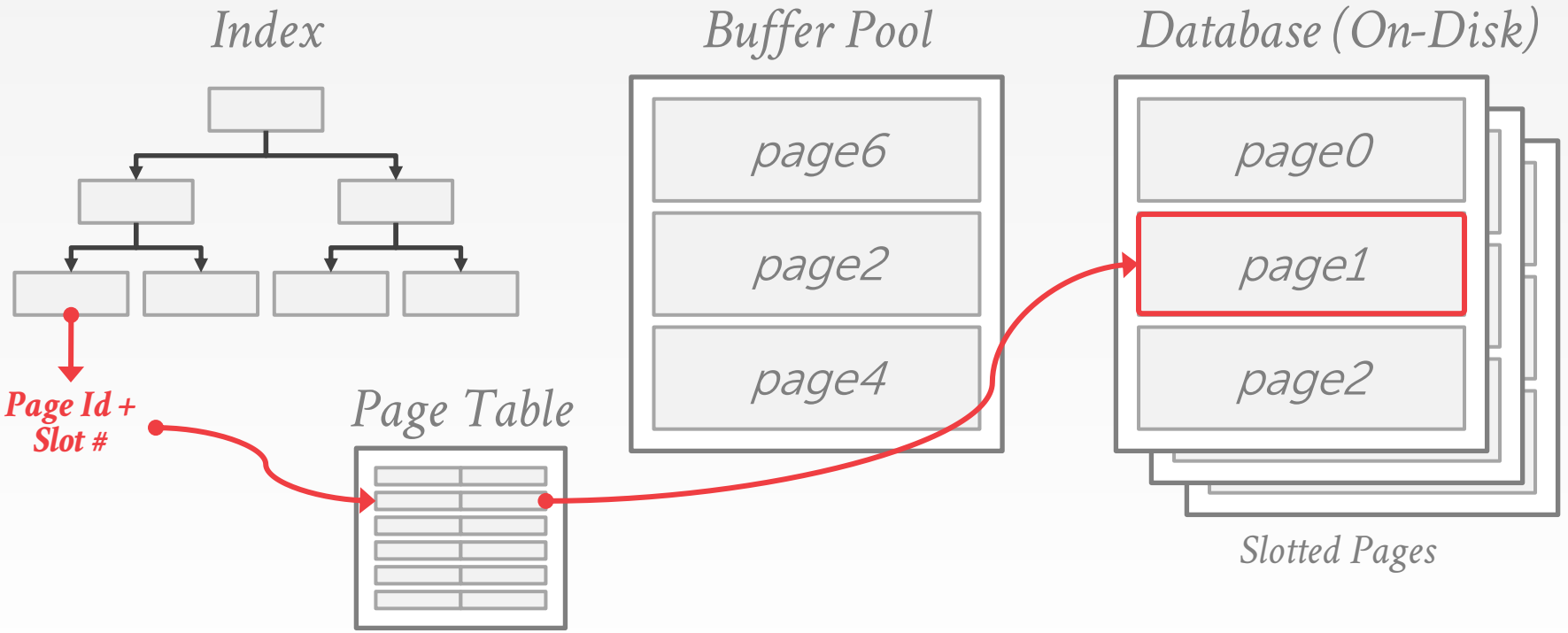
DATA ORGANIZATION



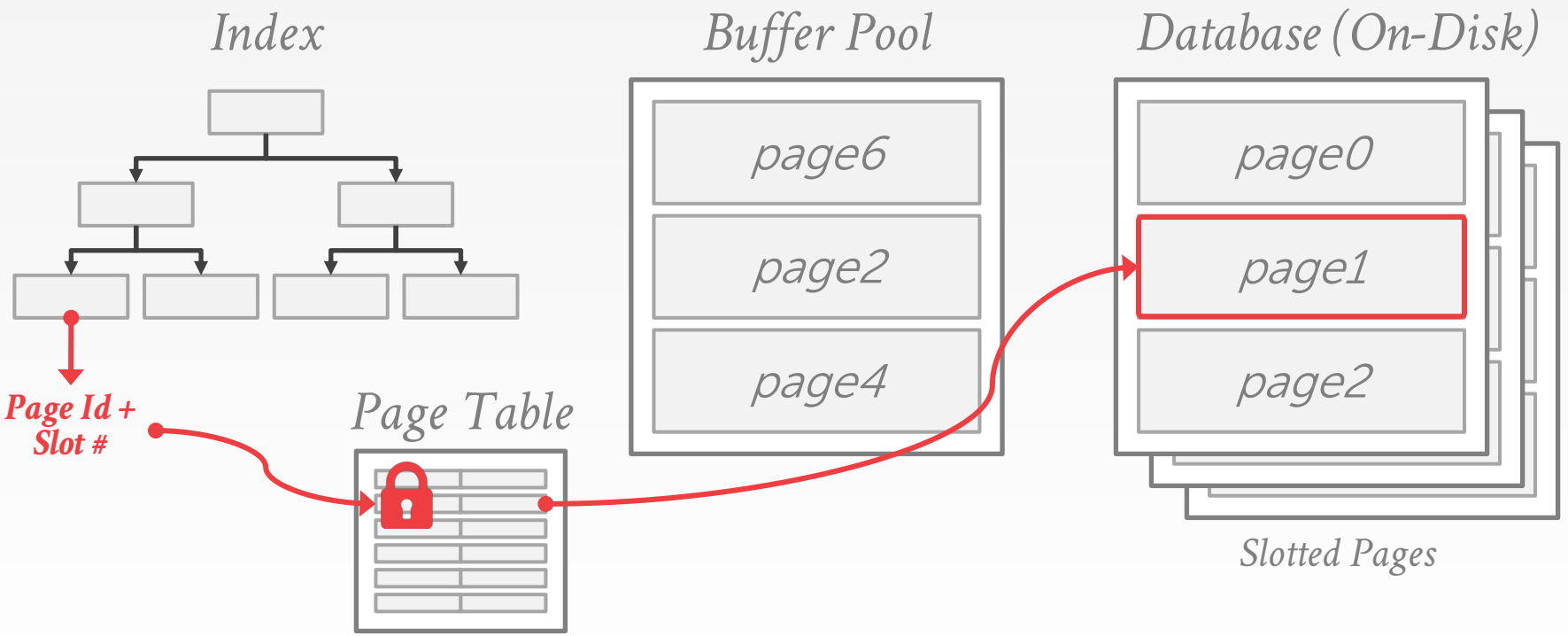
DATA ORGANIZATION



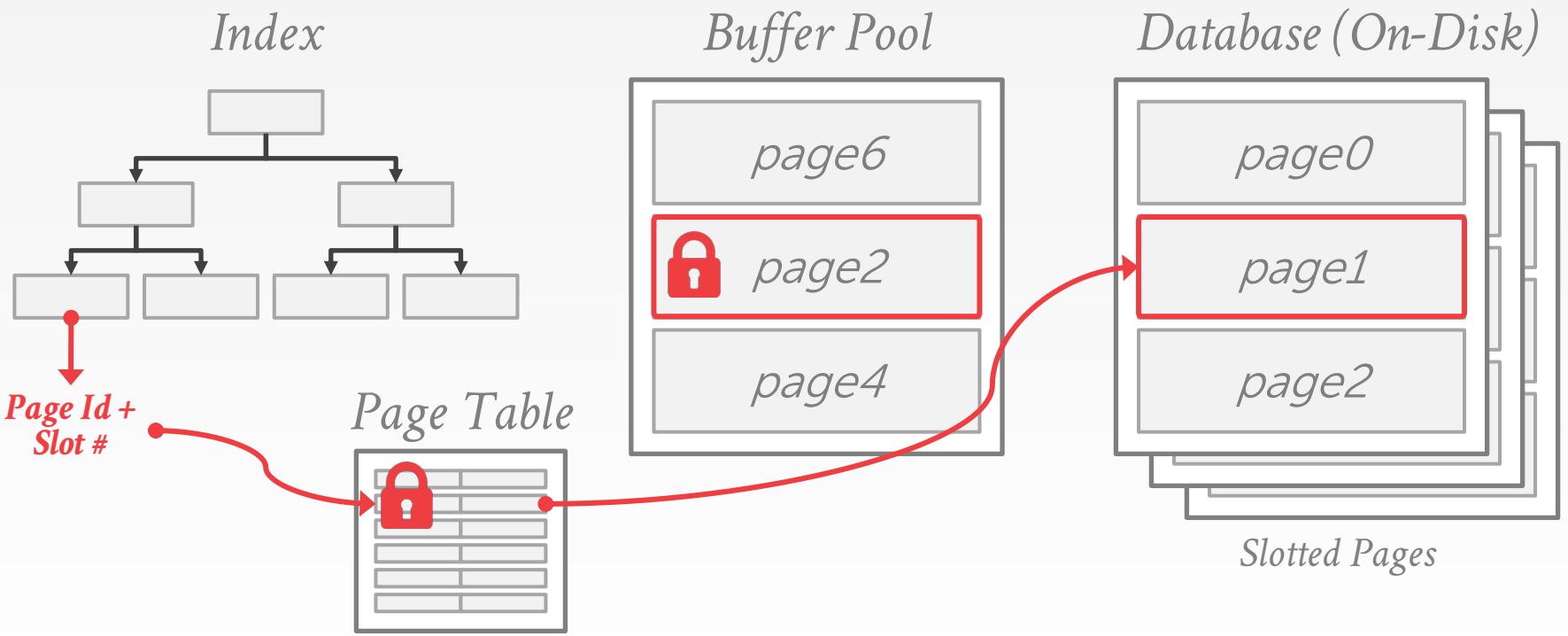
DATA ORGANIZATION



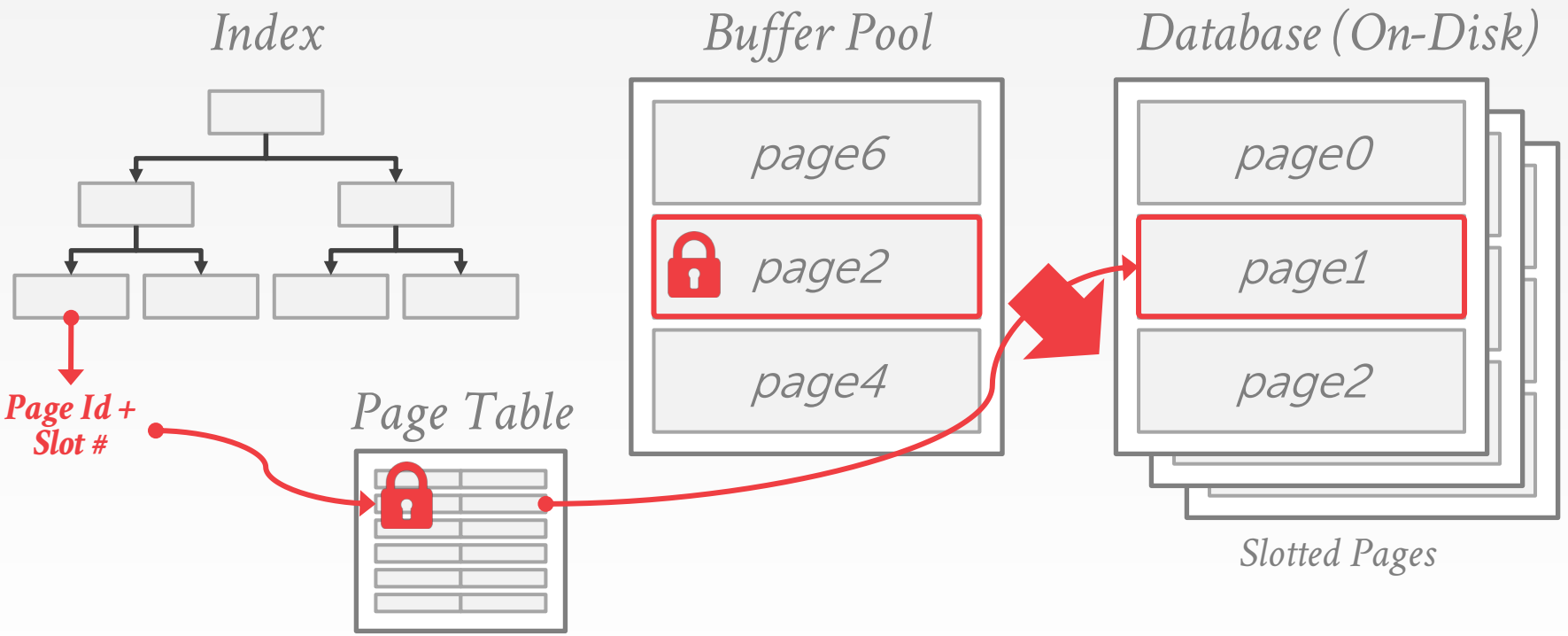
DATA ORGANIZATION



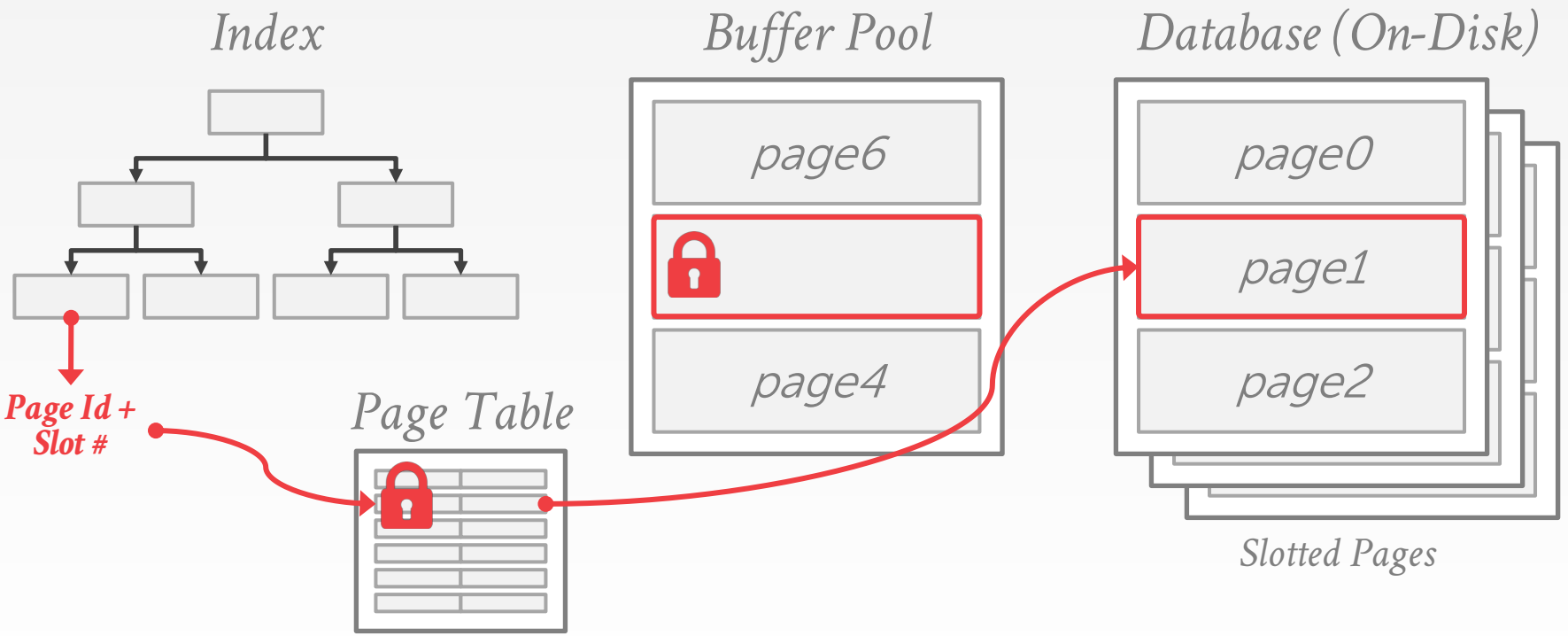
DATA ORGANIZATION



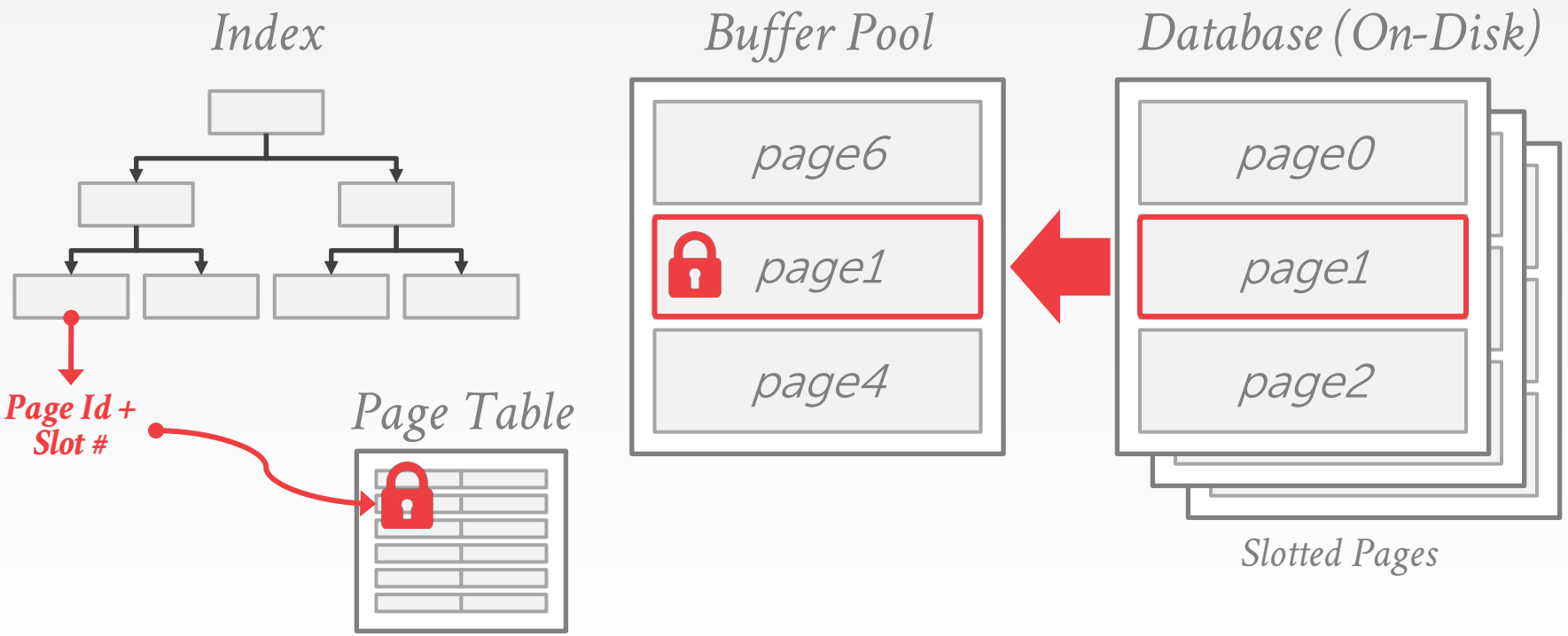
DATA ORGANIZATION



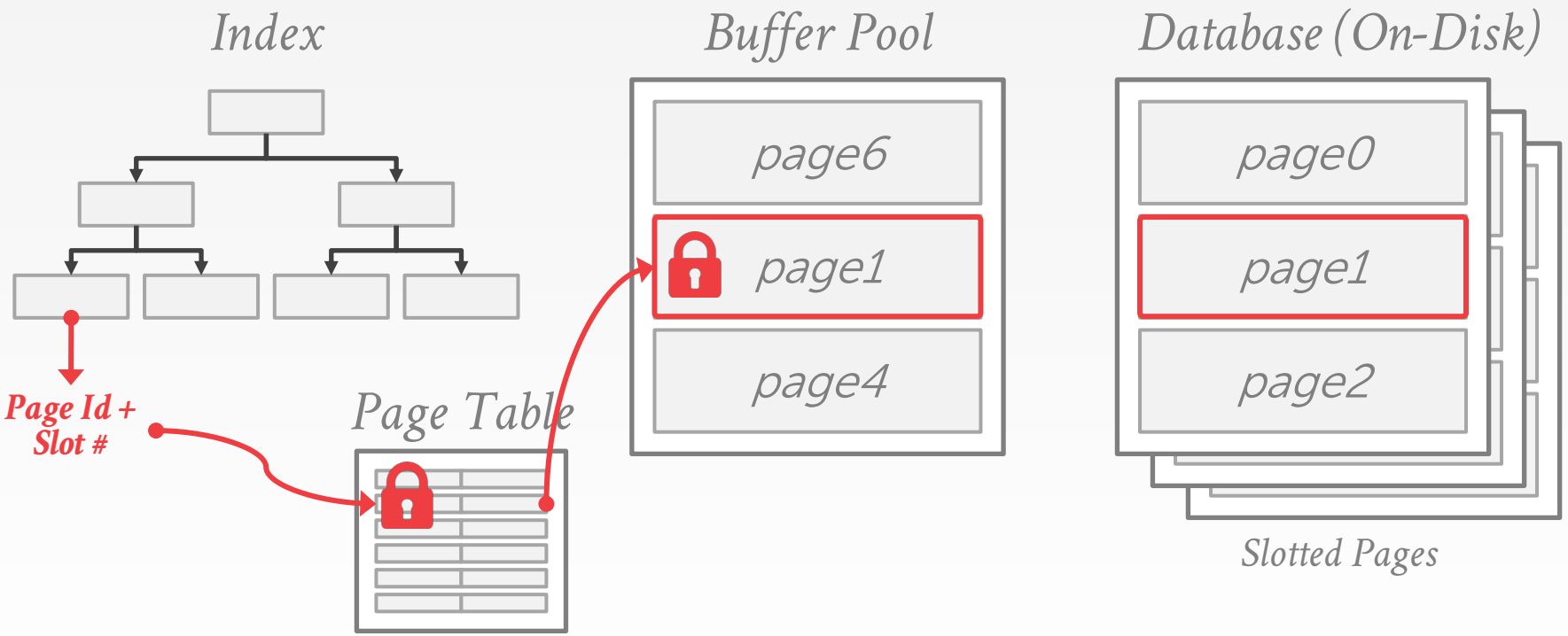
DATA ORGANIZATION



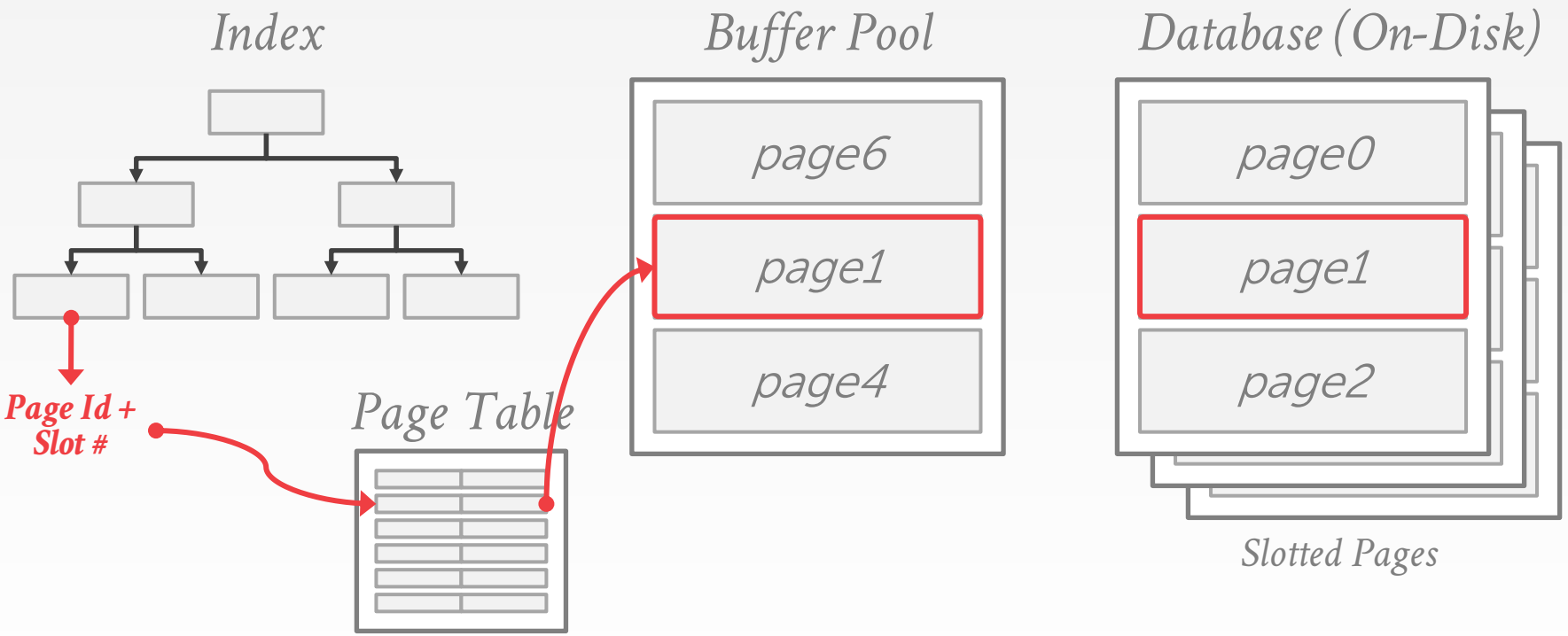
DATA ORGANIZATION



DATA ORGANIZATION



DATA ORGANIZATION



BUFFER POOL

Every tuple access has to go through the buffer pool manager regardless of whether that data will always be in memory.

→ Always have to translate a tuple's record id to its memory location.

→ Worker thread has to **pin** pages that it needs to make sure that they are not swapped to disk.

CONCURRENCY CONTROL

In a disk-oriented DBMS, the systems assumes that a txn could stall at any time when it tries to access data that is not in memory.

Execute other txns at the same time so that if one txn stalls then others can keep running.

→ This is not because the DBMS is trying to use all cores in the CPU. We are still focusing on single-core CPUs.

→ We do this to let the system make **forward progress** by executing another txn while the current txn is waiting for data to be fetched from disk

CONCURRENCY CONTROL

Concurrency control policy

→ Responsible for deciding how to interleave the operations of concurrency transactions in such a way that it appears as if they are running one after each other

→ This property is referred to as **serializability** of transactions

→ Has to set locks and latches to ensure the highest level of **isolation** (ACID) between transactions

→ Locks are stored in a separate data structure (**lock table**) to avoid being swapped to disk.

LOCKS VS. LATCHES

Locks

→ Protects the database's logical contents (e.g., tuple, table) from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

Latches

→ Protects the DBMS's internal physical data structures (e.g., page table) from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.

A SURVEY OF B-TREE LOCKING TECHNIQUES
TODS 2010

LOCKS VS. LATCHES

	Locks	*Latches*
Separate…	User transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

LOGGING & RECOVERY

This protocol is adopted by the DBMS to ensure the atomicity and durability properties (**A**CI**D**)

→ Durability: Changes made by **committed** transactions must be present in the database after recovering from a power failure.

→ Atomicity: Changes made by **uncommitted** (in-progress/aborted) transactions must **not** be present in the database after recovering from a power failure.

LOGGING & RECOVERY

Most DBMSs use **STEAL** + **NO-FORCE** buffer pool policies.

→ STEAL: DBMS can flush pages dirtied by uncommitted transactions to disk.

→ NO-FORCE: DBMS is not required to flush all pages dirtied by committed transactions to disk.

→ So all page modifications have to be flushed to the write-ahead log (**WAL**) before a txn can commit

LOGGING & RECOVERY

Each log entry contains the before and after images of modified tuples.

→ STEAL: Modifications made by uncommitted transactions that are flushed to disk have to rolled back.
→ NO-FORCE: Modifications made by committed transactions might not have been flushed to disk.
→ Recording the before and after images in the log is critical to ensuring the atomicity and durability properties
→ Lots of work to keep track of log sequence numbers (LSNs) all throughout the DBMS.

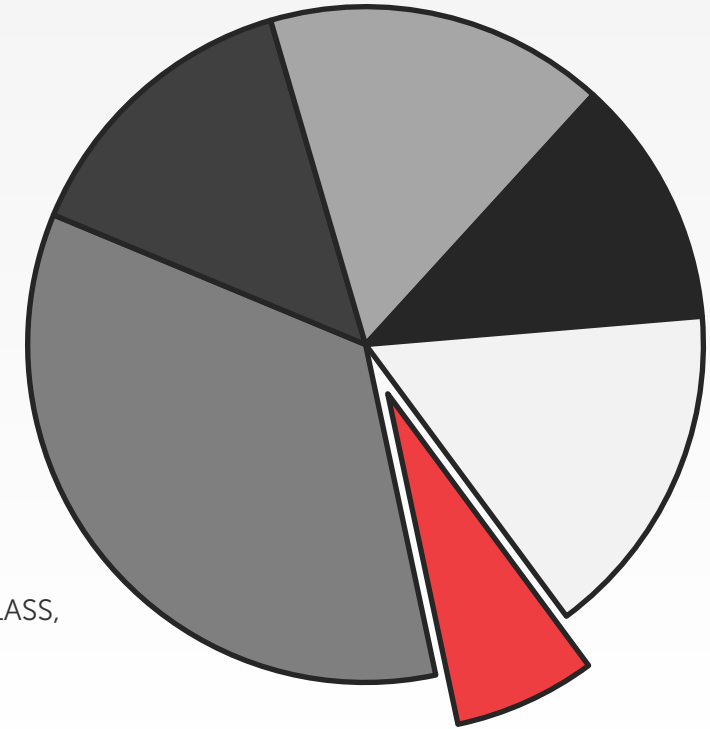
DISK-ORIENTED DBMS OVERHEAD

Measured CPU Instructions

OLTP THROUGH THE LOOKING GLASS,
AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

Measured CPU Instructions

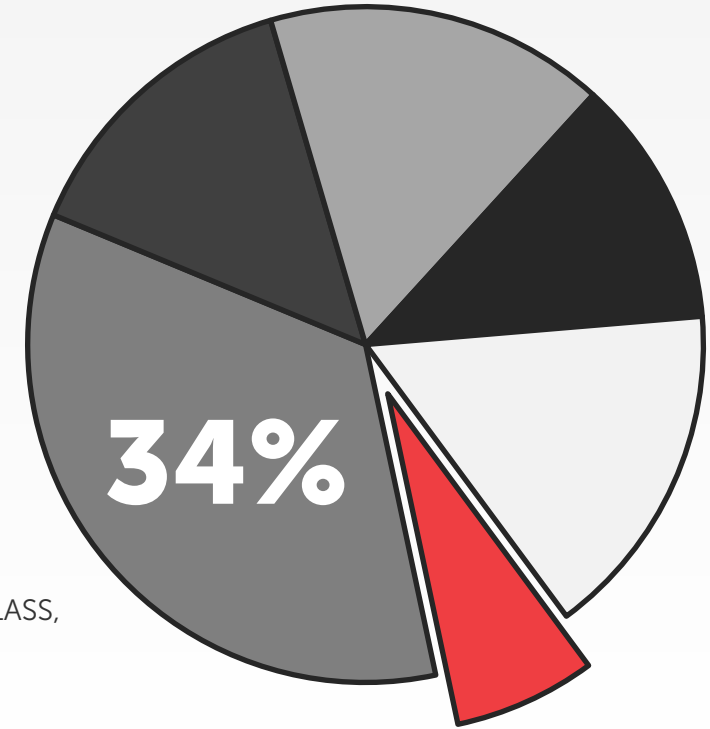


- BUFFER POOL
- LATCHING
- LOCKING
- LOGGING
- B-TREE KEYS
- REAL WORK

OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

Measured CPU Instructions

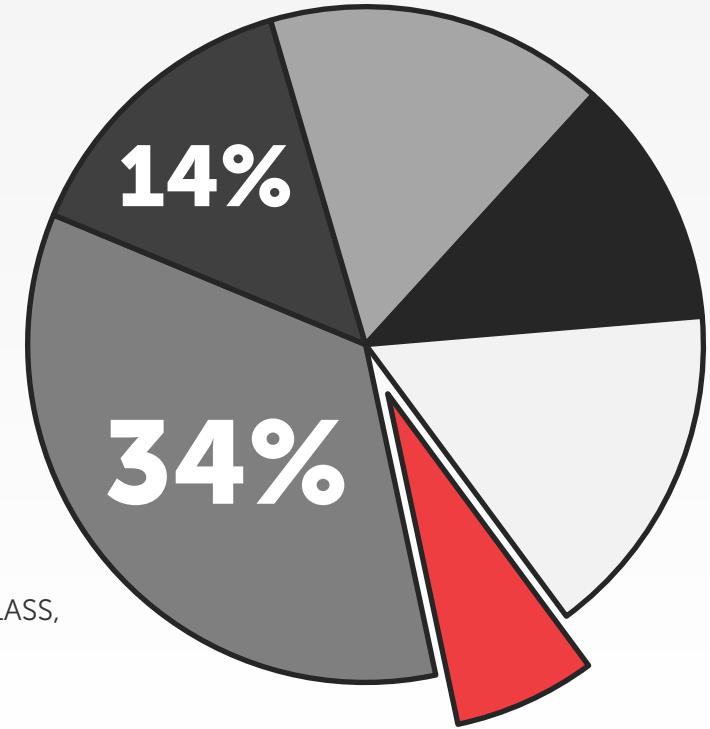


BUFFER POOL
LATCHING
LOCKING
LOGGING
B-TREE KEYS
REAL WORK

OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

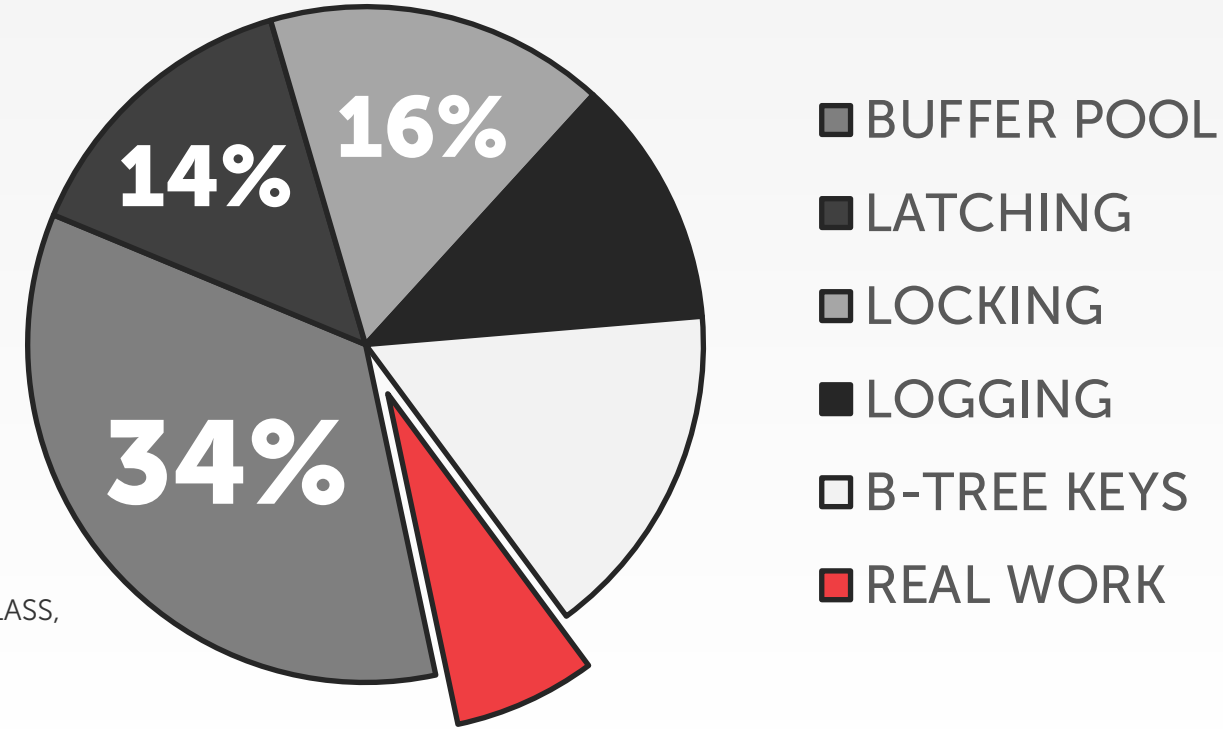
Measured CPU Instructions



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

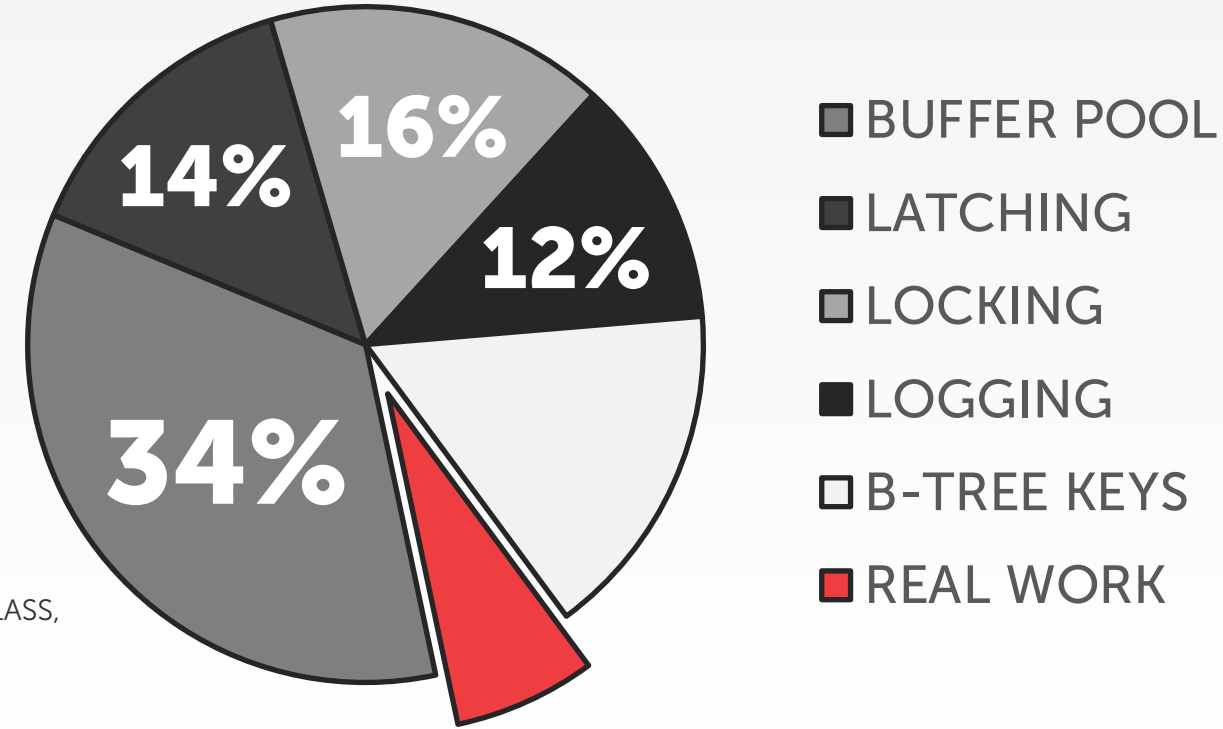
Measured CPU Instructions



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

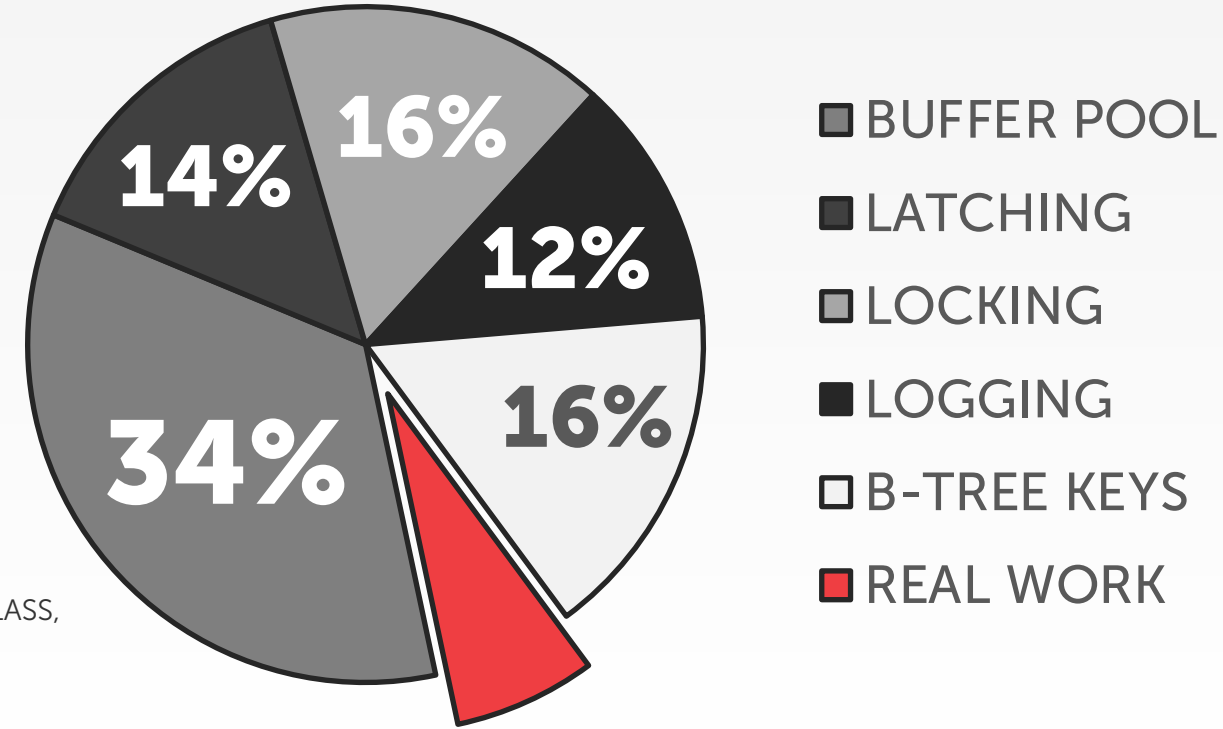
Measured CPU Instructions



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

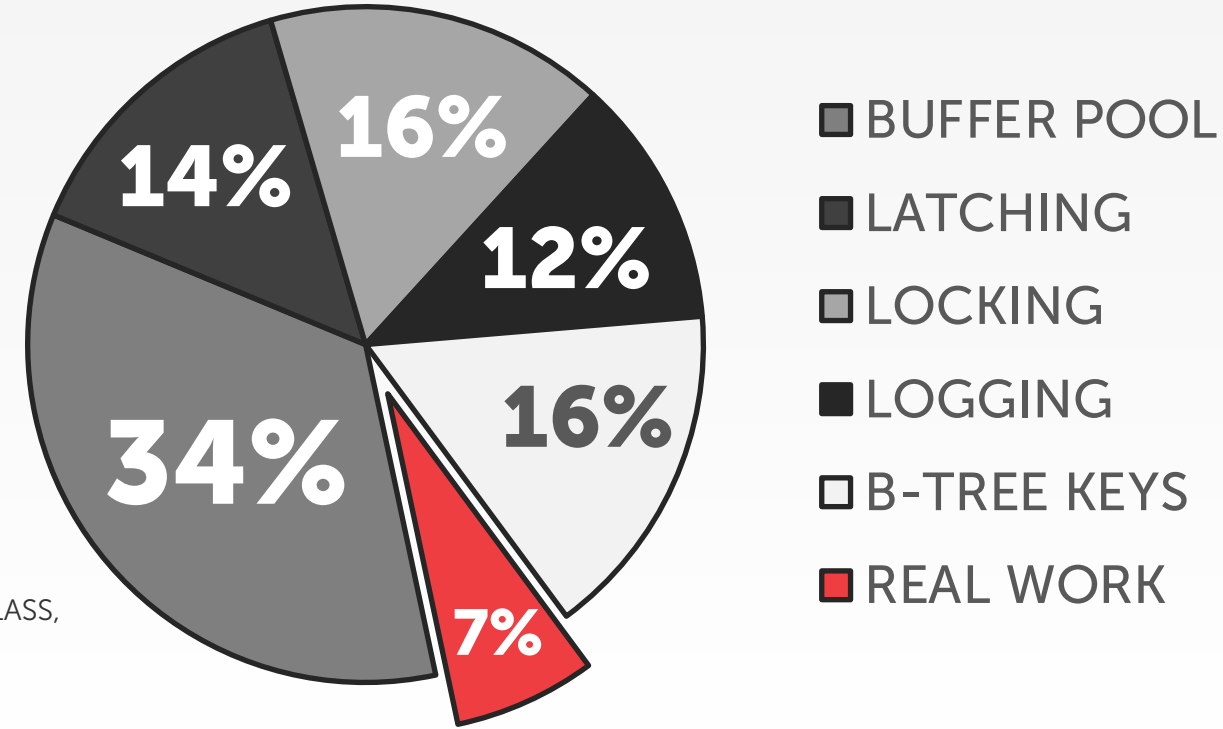
Measured CPU Instructions



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

DISK-ORIENTED DBMS OVERHEAD

Measured CPU Instructions



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD, pp. 981-992, 2008.

TAKEAWAYS

Disk-oriented DBMSs do a lot of extra stuff because they are predicated on the assumption that data has to reside on disk

In-memory DBMSs maximize performance by optimizing these protocols and algorithms

IN-MEMORY DBMSS

Assume that the primary storage location of the database is **permanently** in memory.

Early ideas proposed in the 1980s but it is now feasible because DRAM prices are low and capacities are high.

BOTTLENECKS

If I/O is no longer the slowest resource, much of the DBMS's architecture will have to change account for other bottlenecks:

→ Locking/latching
→ Cache misses
→ Pointer chasing (e.g., virtual function lookup tables)
→ Predicate evaluations
→ Data movement & copying (e.g., multi-socket machine)
→ Networking (between application & DBMS)

STORAGE ACCESS LATENCIES

	L3	DRAM	SSD	HDD
Read Latency	~20 ns	60 ns	25,000 ns	10,000,000 ns
Write Latency	~20 ns	60 ns	300,000 ns	10,000,000 ns

LET'S TALK ABOUT STORAGE & RECOVERY METHODS FOR NON-VOLATILE MEMORY DATABASE SYSTEMS
SIGMOD, pp. 707-722, 2015.

IN-MEMORY DATABASES

Jim Gray's analogy:

→ Reading from L3 cache: Reading a book on a table

→ Reading from HDD: Flying to Pluto to read that book

Because everything fits in DRAM, we can do more sophisticated things in software.

DATA ORGANIZATION

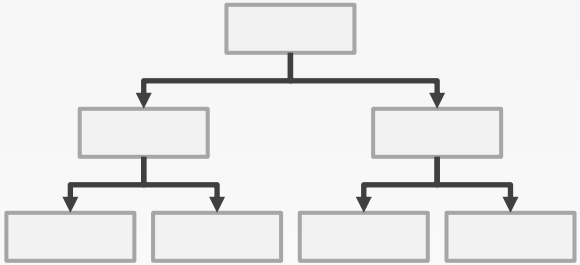
An in-memory DBMS does not need to store the database in slotted pages but it will still organize tuples in blocks/pages:

→ Direct memory pointers vs. tuple identifiers
→ Separate pools for fixed-length (e.g., date of birth) and variable-length data (e.g., medical notes)
→ Use checksums to detect software errors from trashing the database.

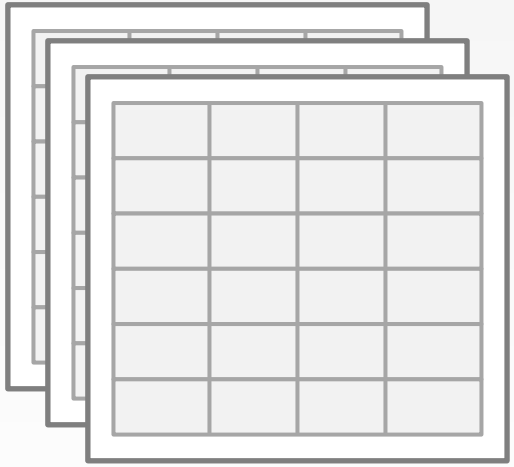
The OS organizes memory in pages too. We will cover this later.

DATA ORGANIZATION

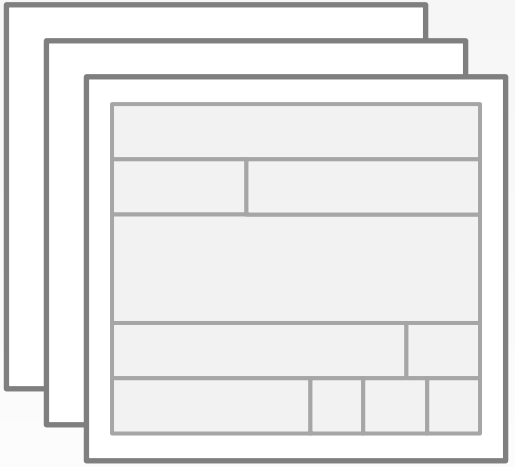
Index



Fixed-Length Data Blocks

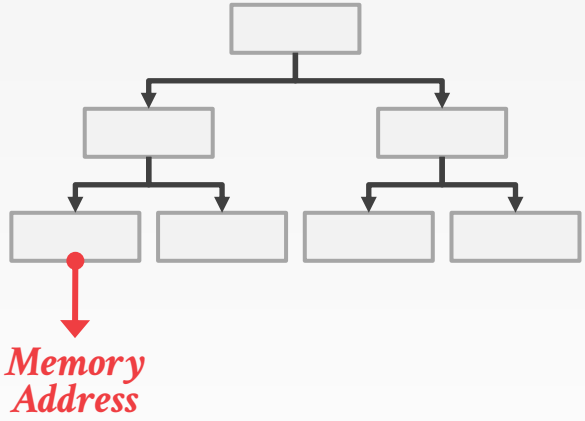


Variable-Length Data Blocks

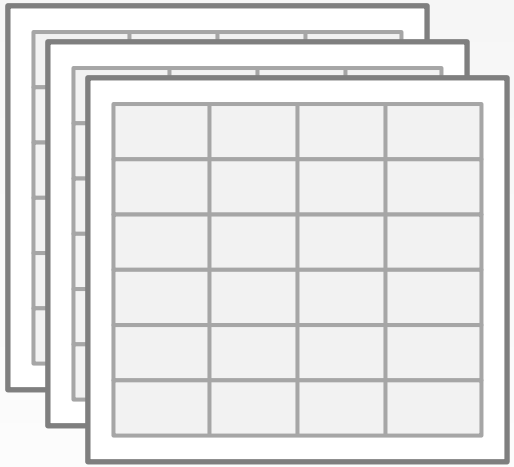


DATA ORGANIZATION

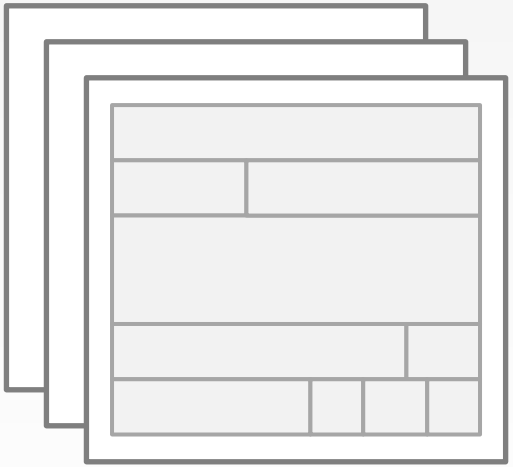
Index



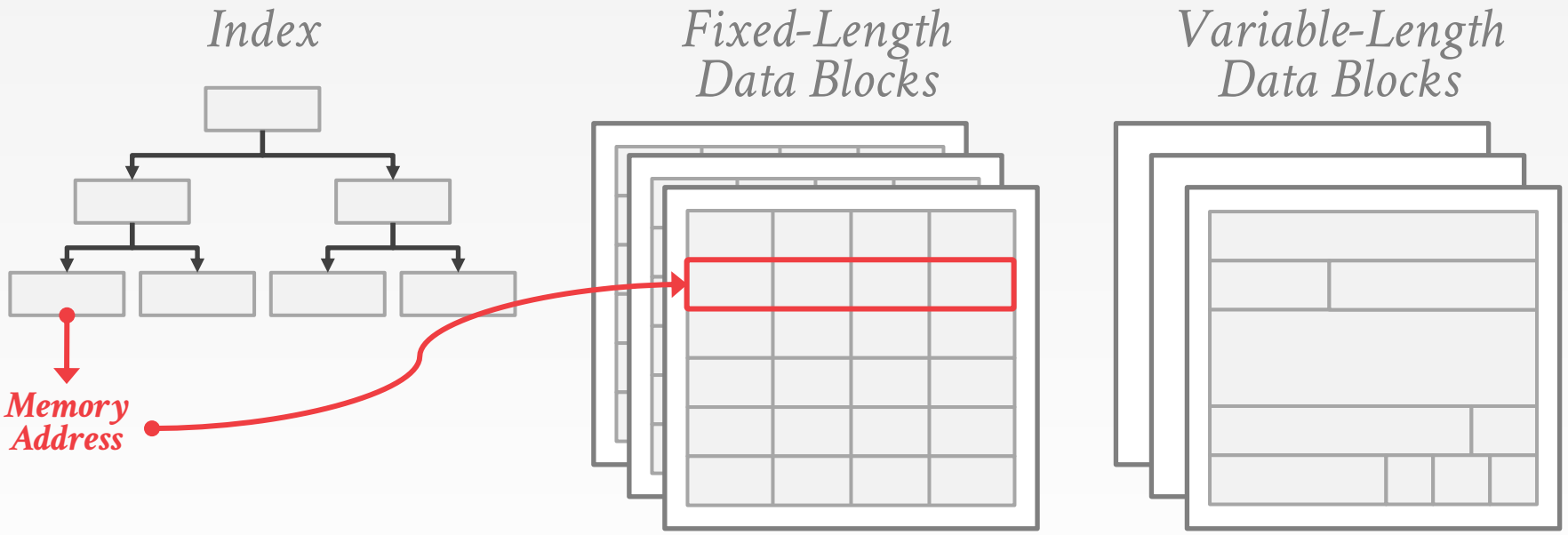
Fixed-Length Data Blocks



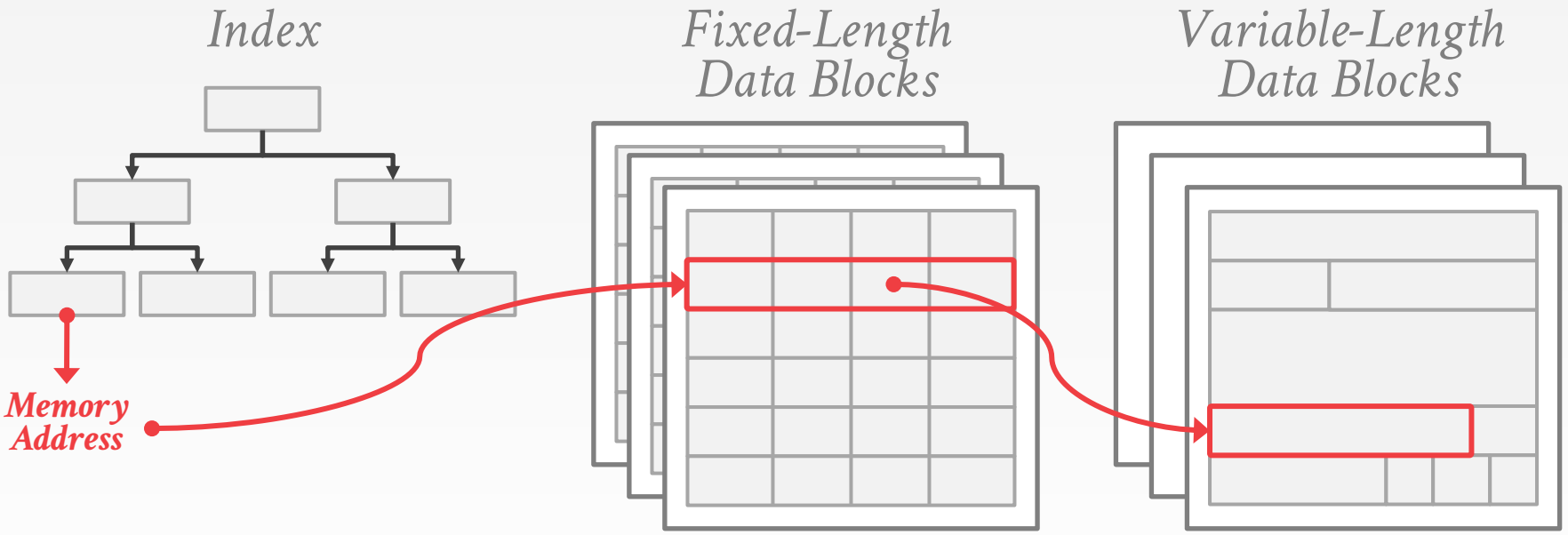
Variable-Length Data Blocks



DATA ORGANIZATION



DATA ORGANIZATION



WHY NOT MMAP?

Memory-map (`mmap`) a database file into DRAM and let the OS be in charge of swapping data in and out as needed.

Use `madvise` and `msync` to give hints to the OS about what data is safe to flush.

Notable `mmap` DBMSs:

→ MongoDB (pre WiredTiger)
→ MonetDB
→ LMDB

WHY NOT MMAP?

Using `mmap` gives up fine-grained control on the contents of memory to the OS.

→ Cannot perform non-blocking memory access.
→ The "on-disk" representation has to be the same as the "in-memory" representation.
→ The DBMS has no way of knowing what pages are in memory or not.
→ Various mmap-related syscalls are not portable.

A well-written DBMS **always** knows best.

CONCURRENCY CONTROL

Observation: The cost of a txn acquiring a lock is the same as accessing data (since the lock data is also in memory).

In-memory DBMS may want to detect conflicts between txns at a different granularity.

→ **Fine-grained locking** allows for better concurrency but requires more locks.

→ **Coarse-grained locking** requires fewer locks but limits the amount of concurrency.

CONCURRENCY CONTROL

The DBMS can store locking information about each tuple together with its data.

→ This helps with CPU cache locality.

→ Mutexes are too slow. Need to use CAS instructions.

Disk-oriented DBMSs: Stalling during disk I/O

Memory-oriented DBMSs: New bottleneck is contention caused from txns executing on multiple cores trying access data at the same time.

INDEXES

Specialized main-memory indexes (e.g., T-Tree) were proposed in 1980s when cache and memory access speeds were roughly equivalent.

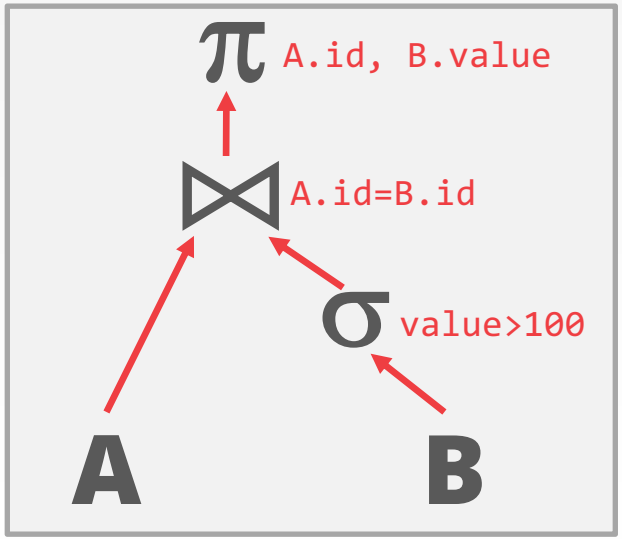
But then caches got faster than main memory:

→ Memory-optimized indexes performed worse than the B+trees because they were not cache aware.

Indexes are usually rebuilt in an in-memory DBMS after restart to avoid logging overhead.

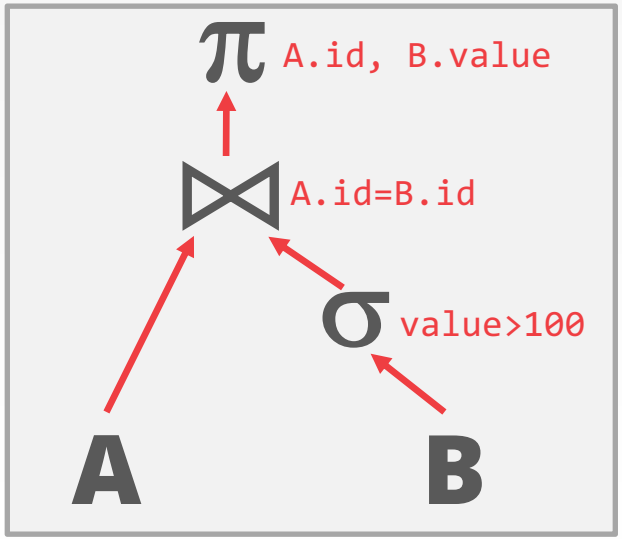
QUERY PROCESSING

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



QUERY PROCESSING

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



Tuple-at-a-time

→ Each operator calls **next** on their child to get the next tuple to process.

Operator-at-a-time

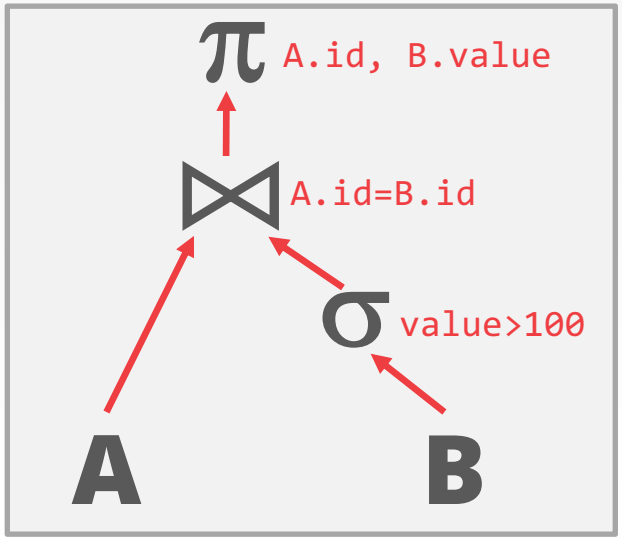
→ Each operator materializes their entire output for their parent operator.

Vector-at-a-time

→ Each operator calls **next** on their child to get the next chunk of data to process.

QUERY PROCESSING

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



Tuple-at-a-time

→ Each operator calls **next** on their child to get the next tuple to process.

Operator-at-a-time

→ Each operator materializes their entire output for their parent operator.

Vector-at-a-time

→ Each operator calls **next** on their child to get the next chunk of data to process.

QUERY PROCESSING

The best strategy for executing a query plan in a DBMS changes when all of the data is already in memory.

→ Sequential scans are no longer significantly faster than random access.

The traditional **tuple-at-a-time** iterator model is too slow because of function calls.

→ This problem is more significant in OLAP DBMSs.

LOGGING & RECOVERY

The DBMS still needs a WAL on non-volatile storage since the system could halt at anytime.

→ Use **group commit** to batch log entries and flush them together to amortize `fsync` cost.

→ May be possible to use more lightweight logging schemes (e.g., only store redo information, NO-STEAL).

But since there are no "dirty" pages, there is no need to maintain LSNs all throughout the system.

LOGGING & RECOVERY

The system also still takes checkpoints to speed up recovery time.

Different methods for checkpointing:

→ Old idea: Maintain a second copy of the database in memory that is updated by replaying the WAL.

→ Switch to a special "copy-on-write" mode and then write a dump of the database to disk.

→ Fork the DBMS process and then have the child process write its contents to disk (leveraging virtual memory).

LARGER-THAN-MEMORY DATABASES

DRAM is fast, but data is not accessed with the same frequency and in the same manner.

→ Hot Data: OLTP Operations (Tweets posted yesterday)
→ Cold Data: OLAP Queries (Tweets posted last year)

We will study techniques for how to bring back disk-resident data without slowing down the entire system.

NON-VOLATILE MEMORY

Emerging hardware that is able to get almost the same read/write speed as DRAM but with the persistence guarantees of an SSD.

→ Also called *storage class memory*

→ Examples: Phase-Change Memory, Memristors

It's not clear how to build a DBMS to operate on this kind of memory.

Again, we'll cover this topic later.

NOTABLE IN-MEMORY DBMSs

Oracle TimesTen
Dali / DataBlitz
Altibase
P*TIME
SAP HANA
VoltDB / H-Store

Microsoft Hekaton
Harvard Silo
TUM HyPer
MemSQL
IBM DB2 BLU
Apache Geode

NOTABLE IN-MEMORY DBMSs

- Oracle TimesTen
- Dali / DataBlitz
- Altibase
- P*TIME
- SAP HANA
- VoltDB / H-Store
- Microsoft Hekaton
- Harvard Silo
- TUM HyPer
- MemSQL
- IBM DB2 BLU
- Apache Geode

P*TIME

Korean in-memory DBMS from the 2000s.

Performance numbers are still impressive.

Lots of interesting features:

→ Uses differential encoding (XOR) for log records.
→ Hybrid storage layouts.
→ Support for larger-than-memory databases.

Sold to SAP in 2005. Now part of HANA.

P*TIME: HIGHLY SCALABLE OLTP DBMS FOR MANAGING UPDATE-INTENSIVE STREAM WORKLOAD
VLDB, pp. 1033-1044, 2004.

TIMESTEN

Originally SmallBase from HP Labs in 1995.

Multi-process, shared memory DBMS.

→ Single-version database using two-phase locking.
→ Dictionary-encoded columnar compression.

Bought by Oracle in 2005.

Can work as a cache in front of Oracle DBMS.

ORACLE TIMESTEN: AN IN-MEMORY DATABASE FOR ENTERPRISE APPLICATIONS
VLDB, pp. 1033-1044, 2004.

DALI / DATABLITZ

Developed at AT&T Labs in the early 1990s.

Multi-process, shared memory storage manager using memory-mapped files.

Employed additional safety measures to make sure that erroneous writes to memory do not corrupt the database.

→ Meta-data is stored in a non-shared location.

→ A page's checksum is always tested on a read; if the checksum is invalid, recover page from log.

DALI: A HIGH PERFORMANCE MAIN MEMORY STORAGE MANAGER
VLDB, pp. 48-59, 1994.

PELOTON DBMS

CMU's in-memory hybrid relational DBMS

→ Latch-free Multi-version concurrency control.
→ Latch-free Bw-Tree Index
→ LLVM-based Execution Engine
→ Tile-based storage manager.
→ Multi-threaded architecture.
→ ~~Write-Ahead Logging + Checkpoints~~
→ Cascades-style Query Optimizer
→ Zone Maps
→ PL/pgSQL UDFs (preliminary)

Currently supports **some** of SQL-92.

PARTING THOUGHTS

Disk-oriented DBMSs are a relic of the past.
→ Most databases fit entirely in DRAM on a single machine.

The world has finally become comfortable with in-memory data storage and processing.

Never use `mmap` for your DBMS.

COURSE LOAD REDUCTION

The frequency of reading reviews is reduced to one review due **every two weeks** (earlier it was one review due every week).

The **final exam** (15%) will be a take home assignment. The exam will be long-form questions based on the topics discussed during the entire semester and you will get a week to complete it.

DEVELOPMENT ENVIRONMENT

Install Ubuntu 18.04 LTS Linux OS on your laptop (either natively or in a virtual machine).

You will be using this environment for programming assignments and research project.

NEXT CLASS

Storage Models

Reminder: Homework 0 is due on Tuesday Jan 15th. Submit via Gradescope.