Lecture #02 – In-Memory Databases

Andy Pavlo // Carnegie Mellon University // Spring 2016

TODAY'S AGENDA

Background
In-Memory DBMS Architectures
Historical Systems
Peloton Overview
Project #1

BACKGROUND

Much of the history of DBMSs is about avoiding the slowness of disks.

Hardware was much different when the original DBMSs were designed:
→ Uniprocessor (single-core CPU)
→ RAM was severely limited.
→ The database had to be stored on disk.

BACKGROUND

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

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 organized as a set of fixed-length blocks called **slotted pages**.

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.

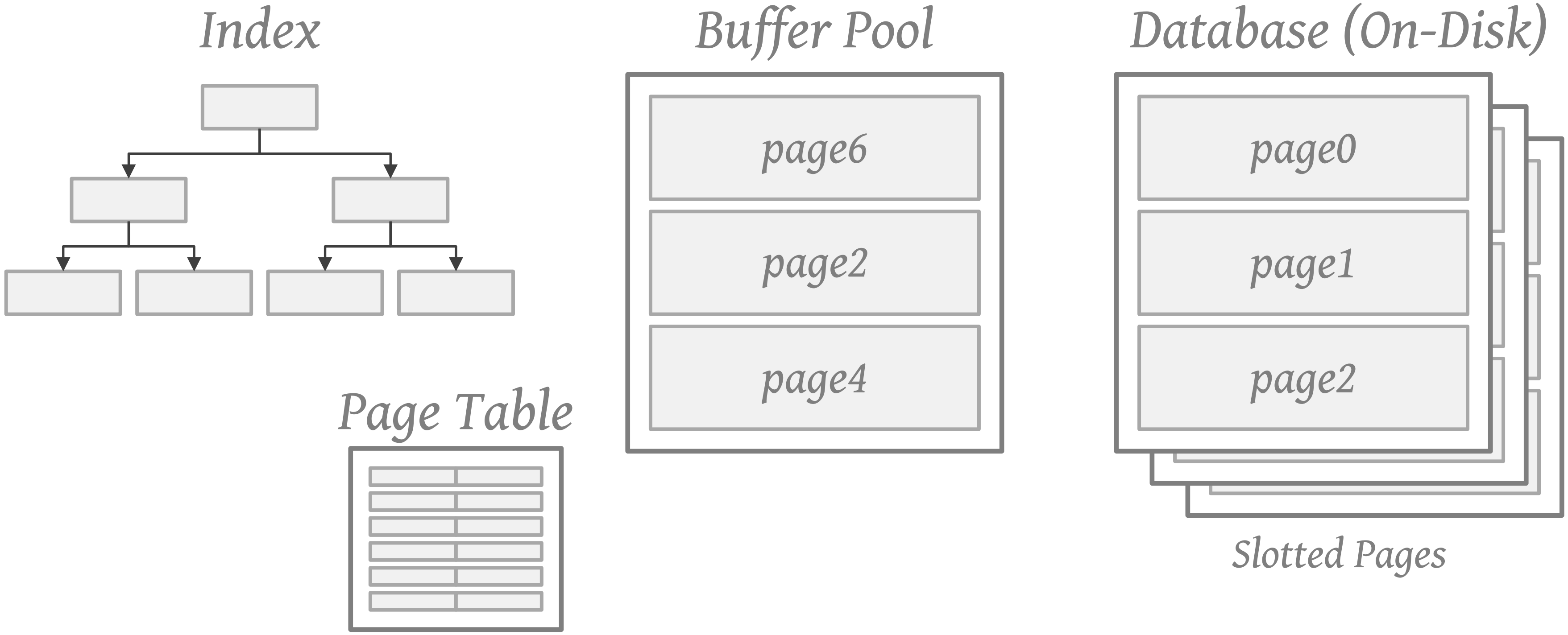
BUFFER POOL

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

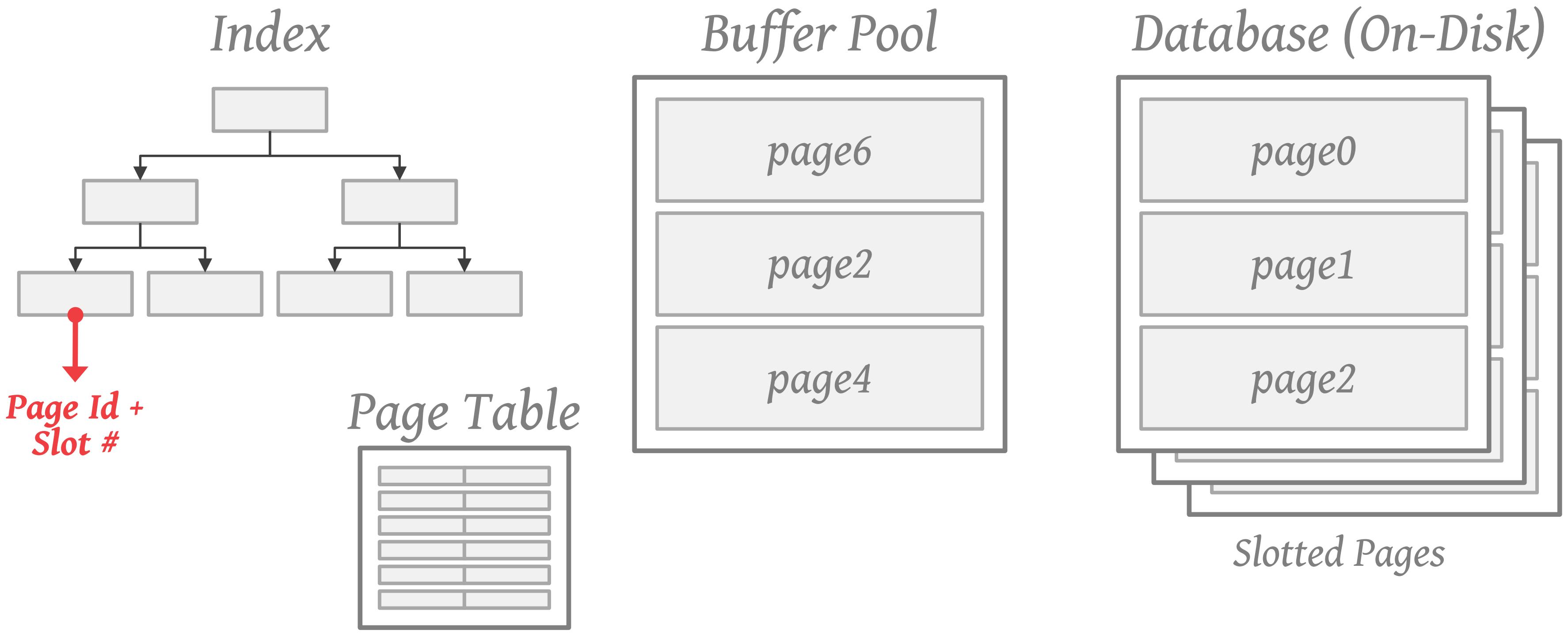
→ If it's not, then the DBMS has to retrieve it from disk and copy it into a frame in its buffer pool.
→ If there are no free frames, then find a page to evict.
→ If the page being evicted is dirty, then the DBMS has to write it back to disk.

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

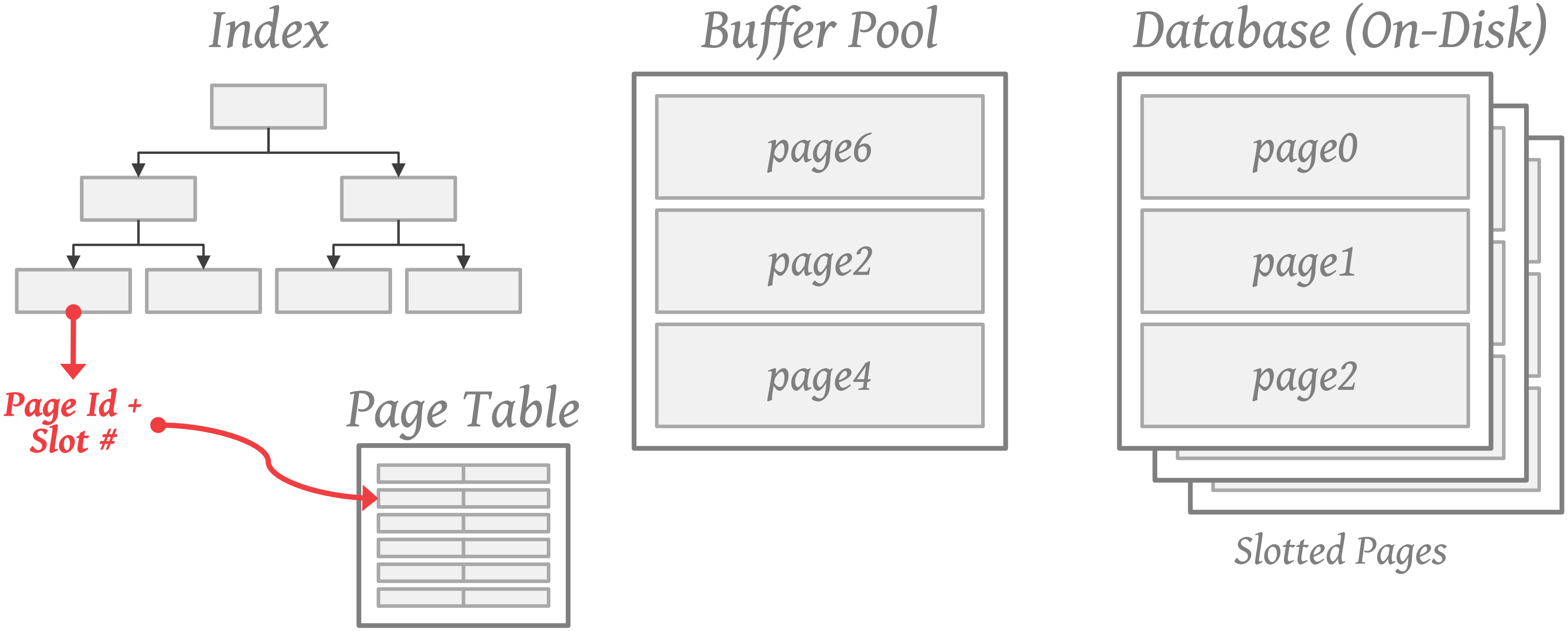
DATA ORGANIZATION



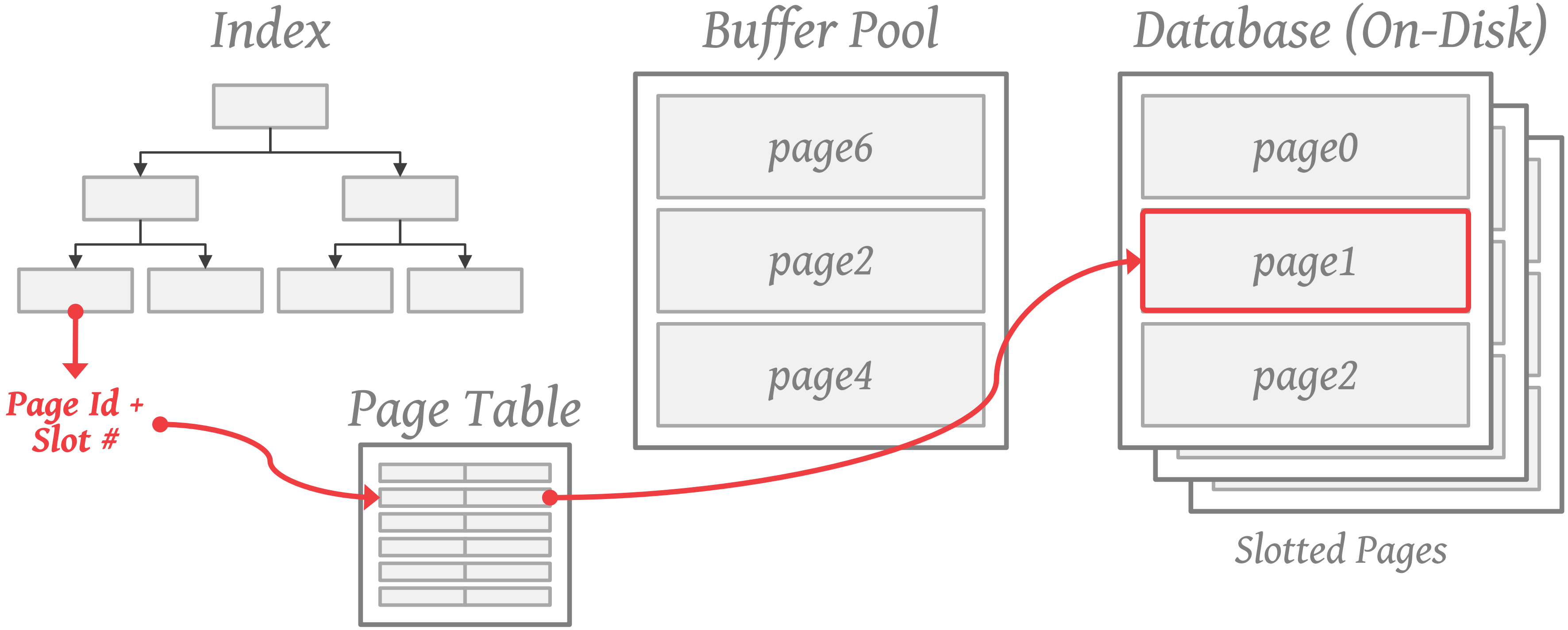
DATA ORGANIZATION



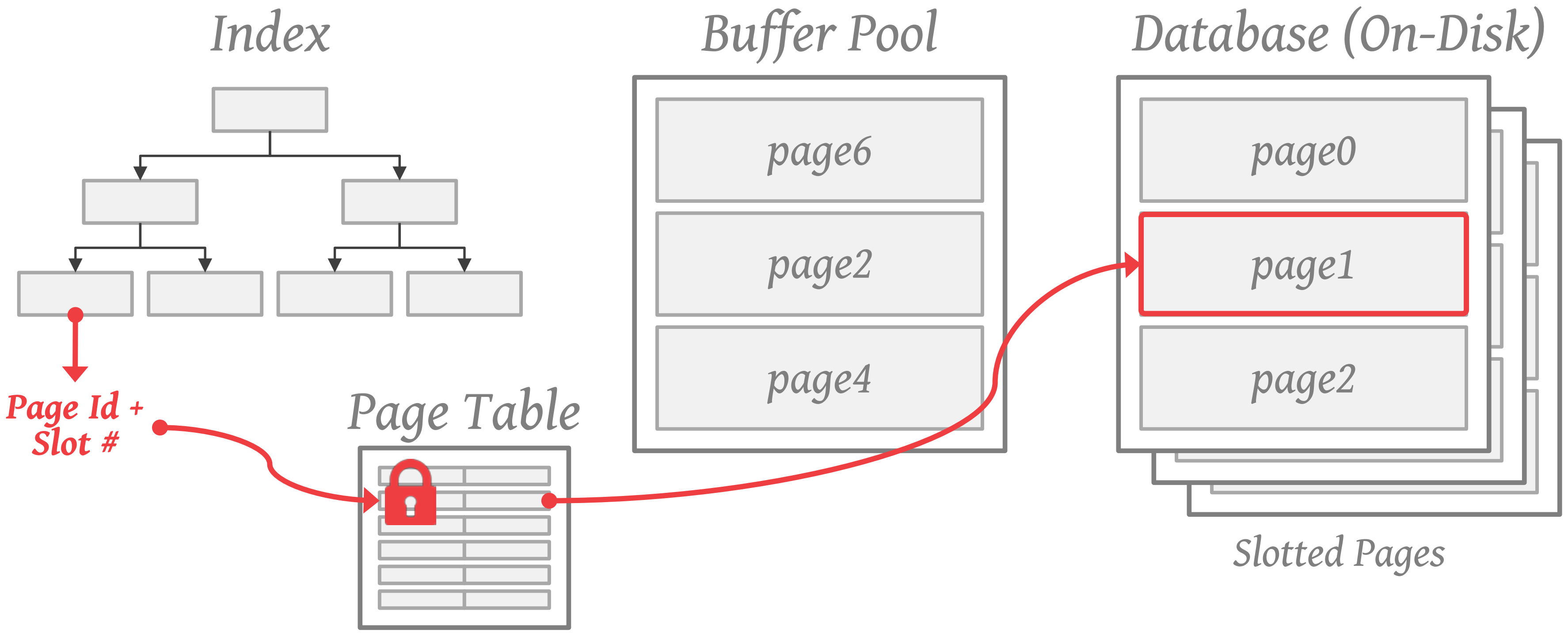
DATA ORGANIZATION



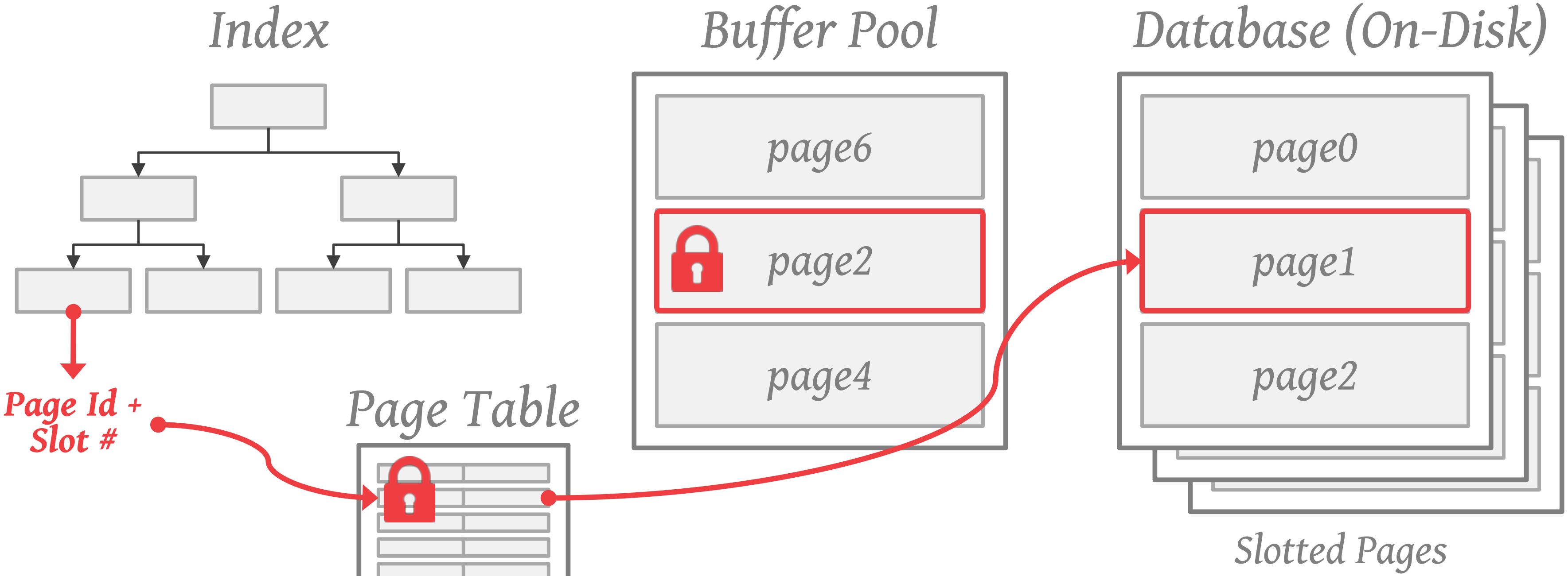
DATA ORGANIZATION



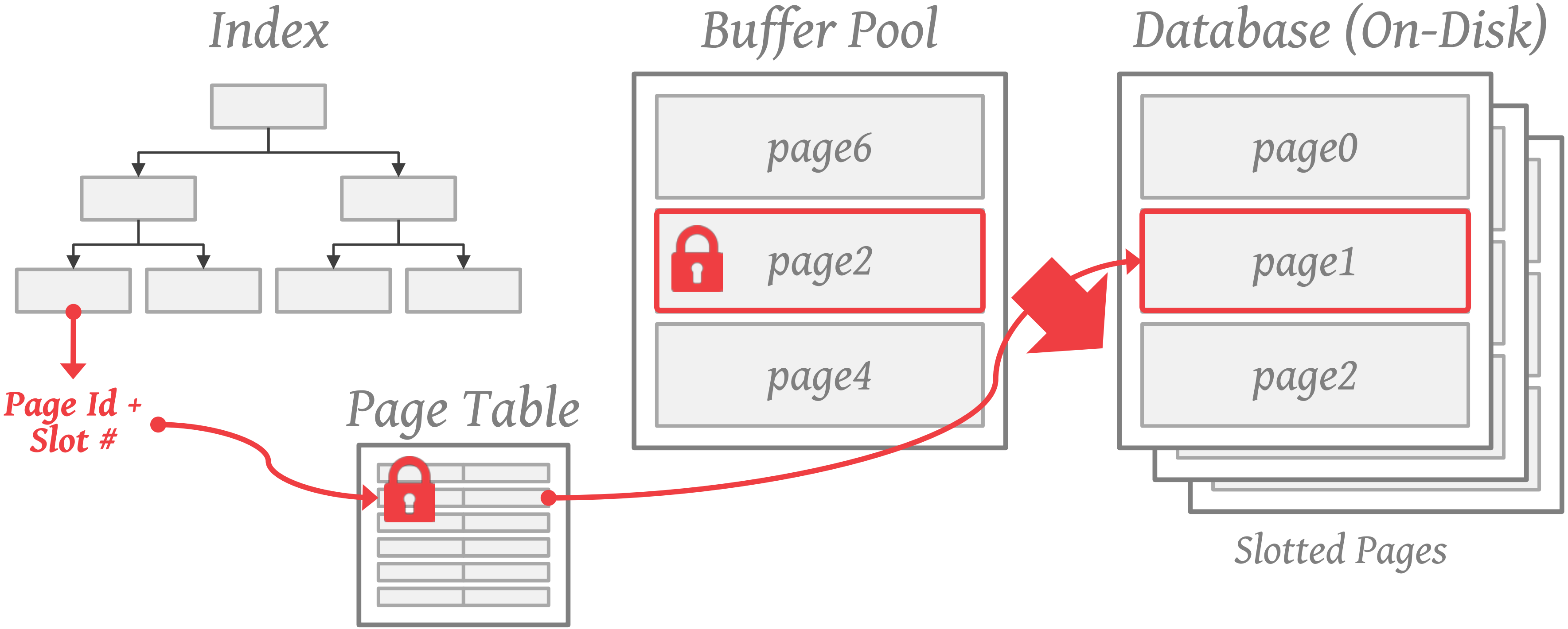
DATA ORGANIZATION



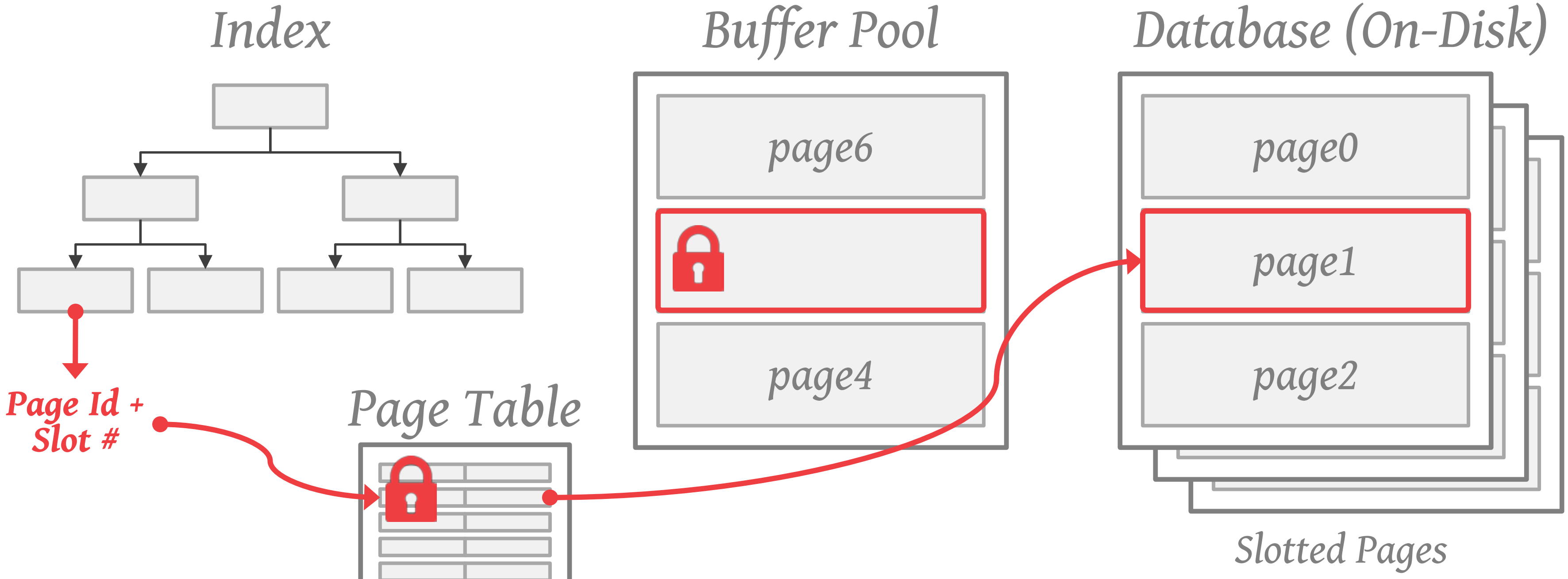
DATA ORGANIZATION



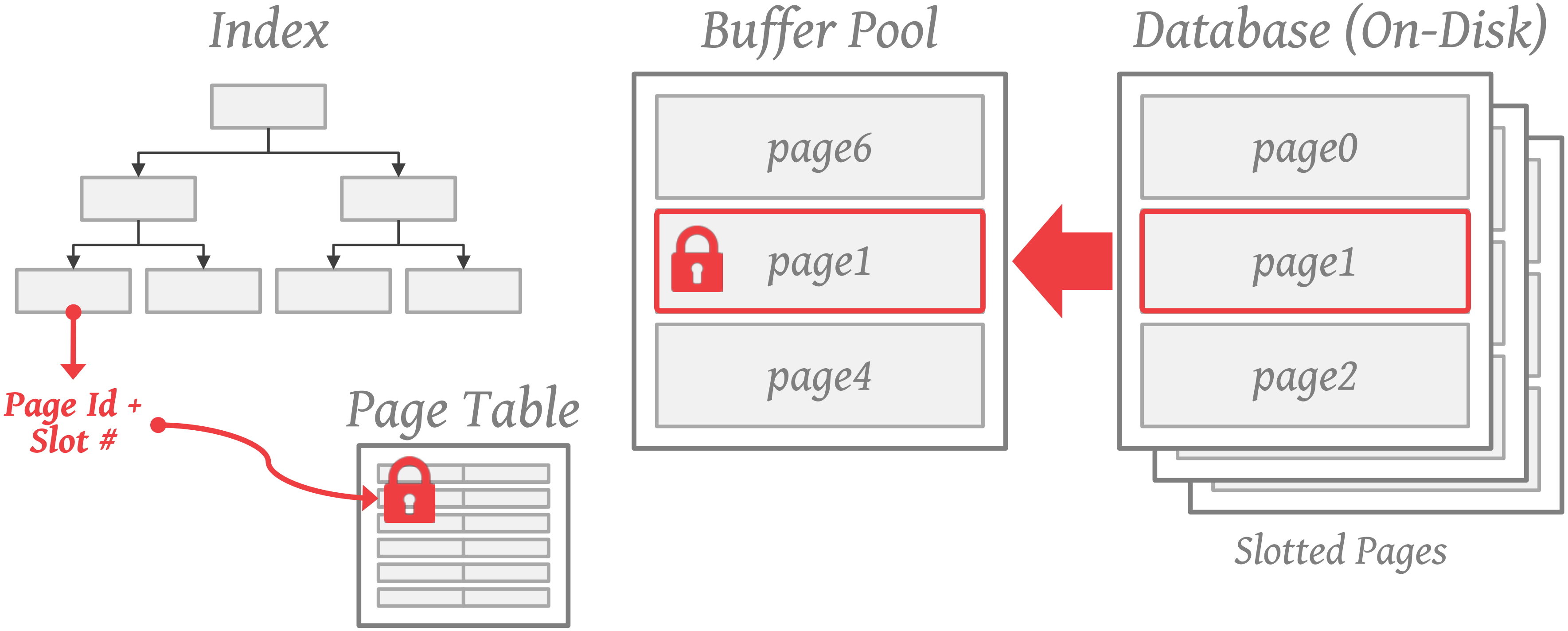
DATA ORGANIZATION



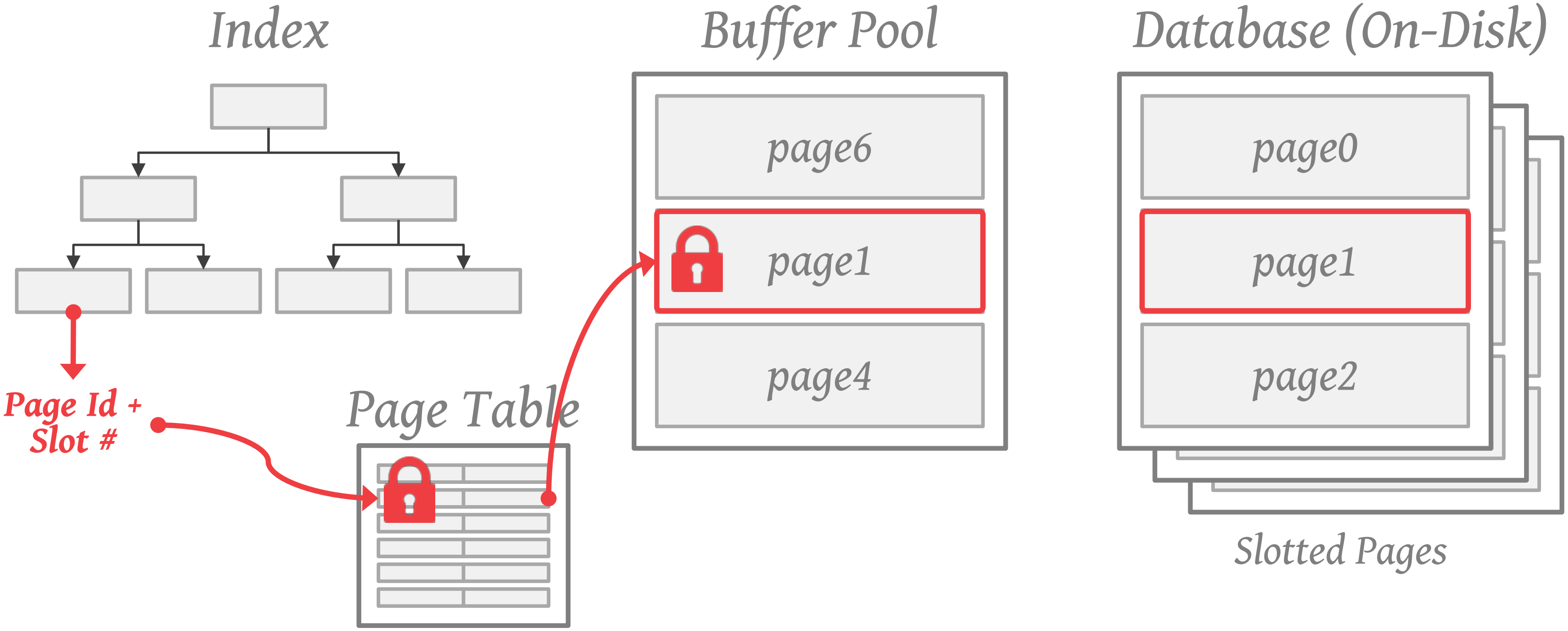
DATA ORGANIZATION



DATA ORGANIZATION



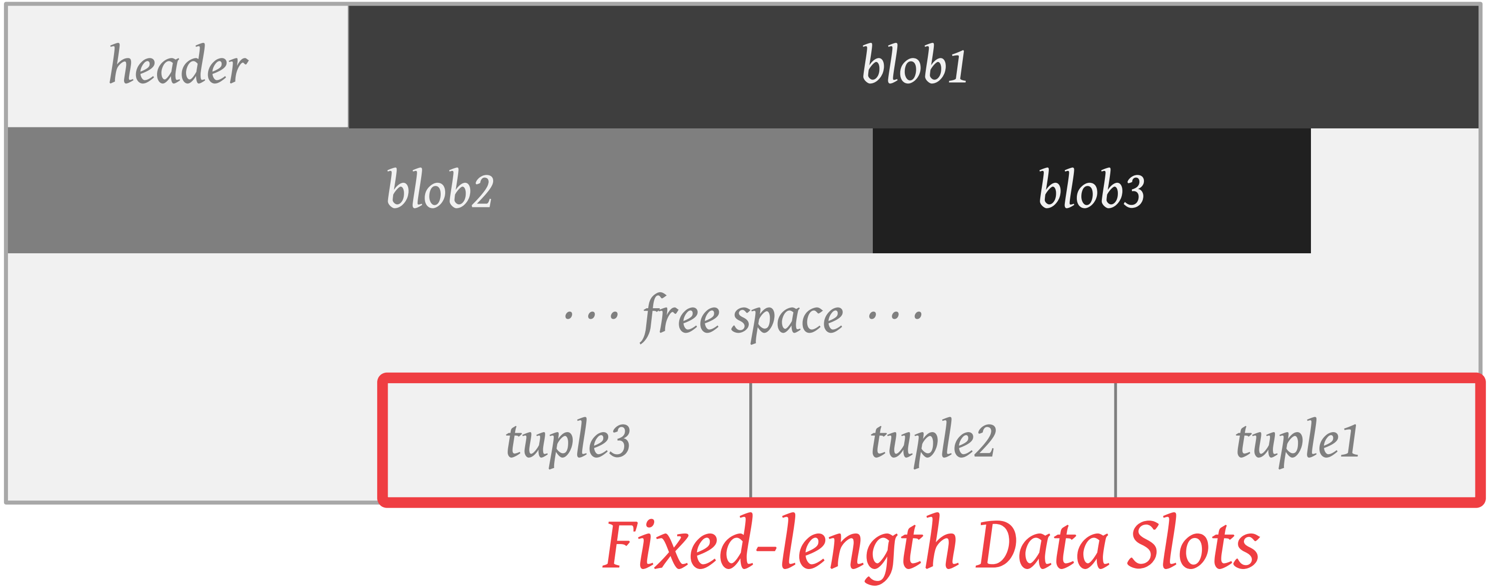
DATA ORGANIZATION



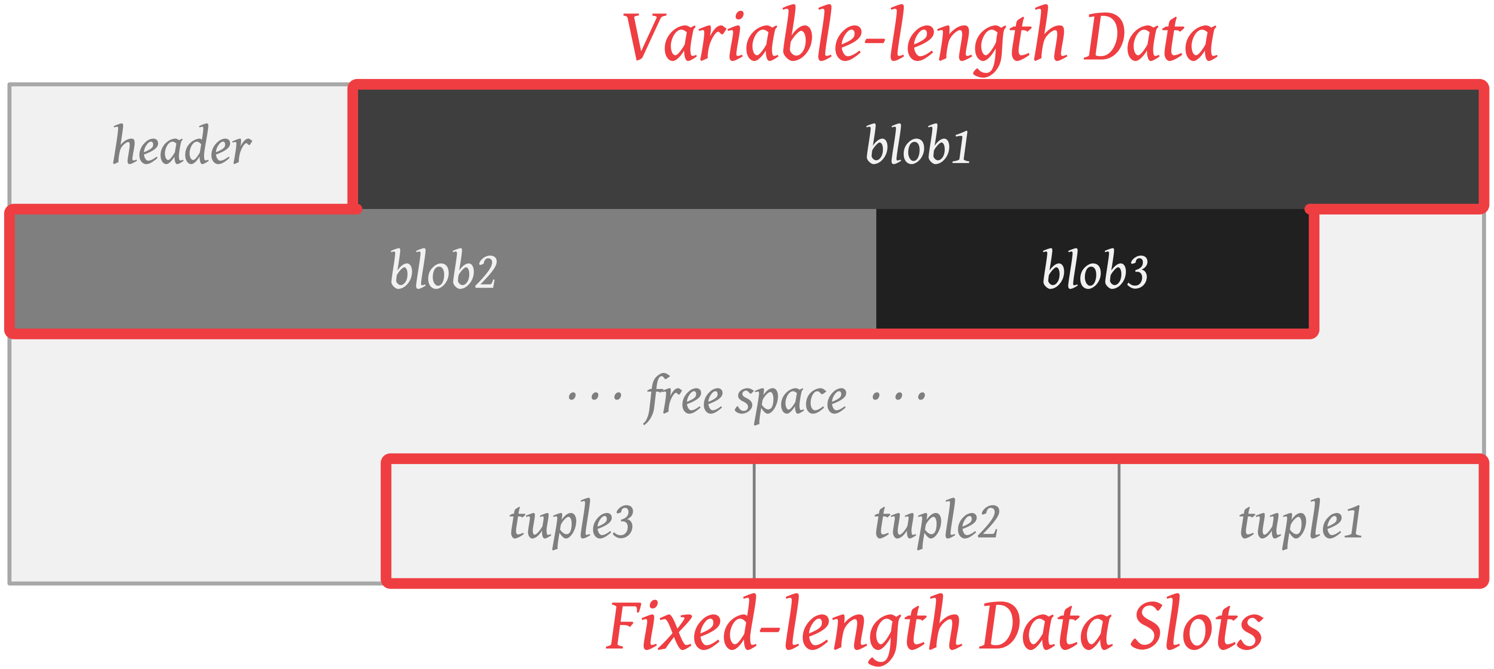
SLOTTED PAGES

SLOTTED PAGES

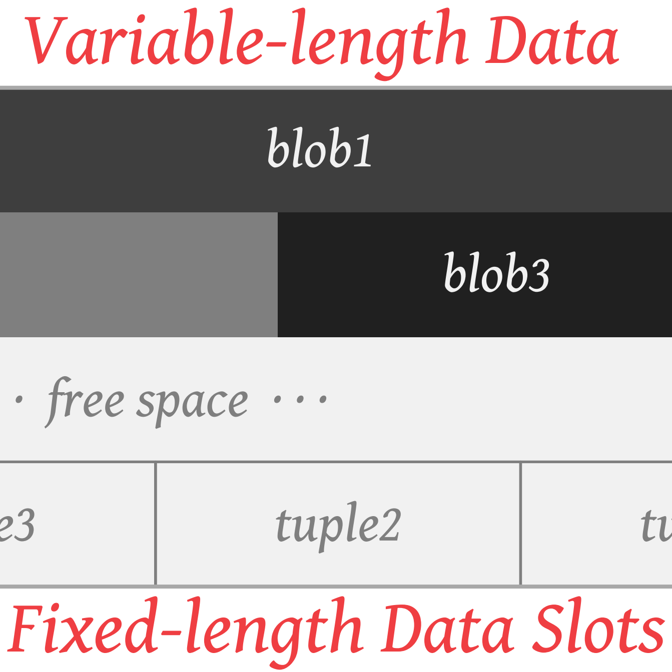
SLOTTED PAGES



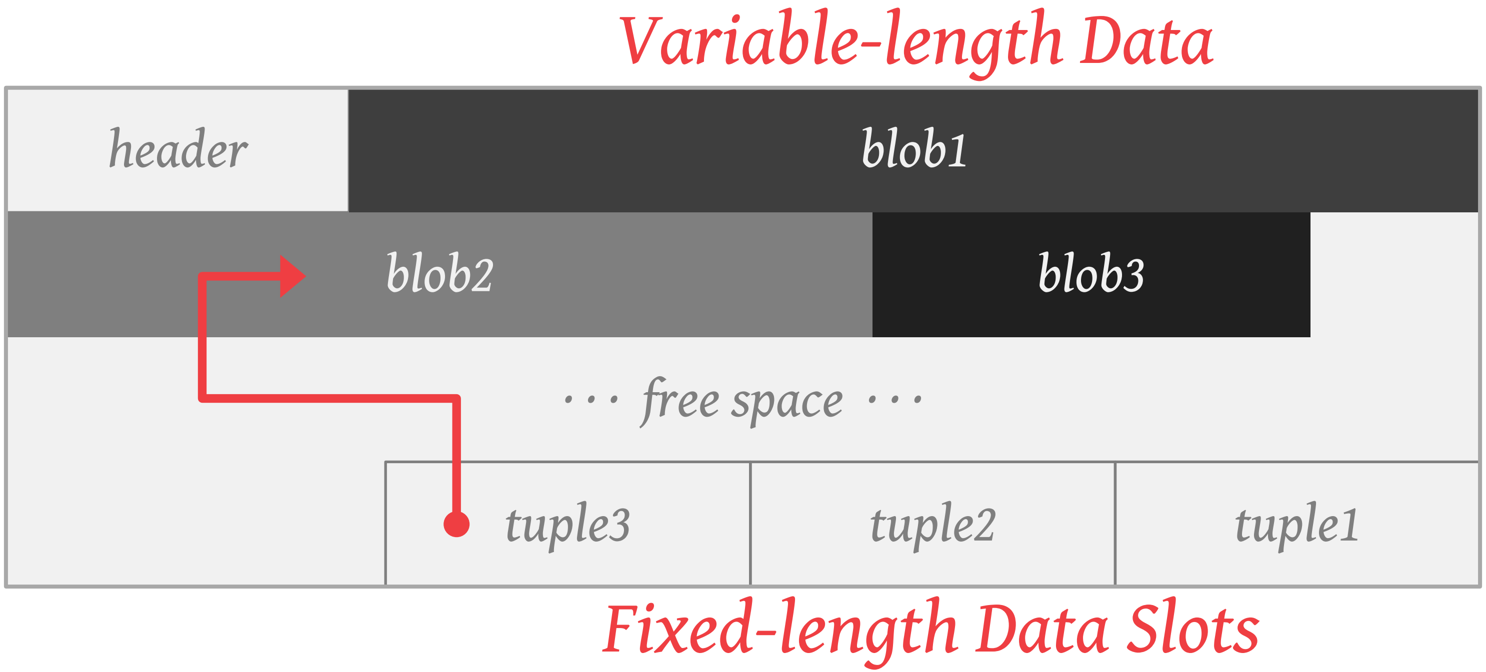
SLOTTED PAGES



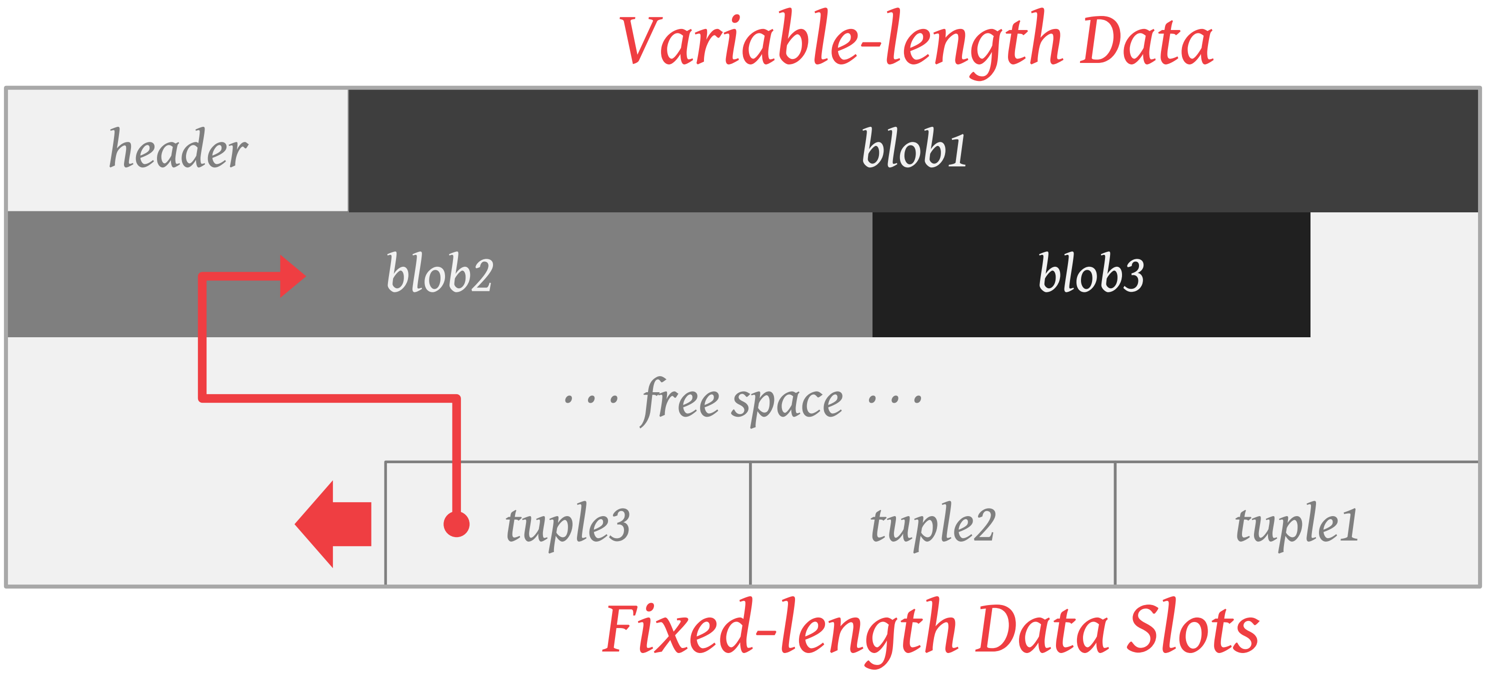
SLOTTED PAGES



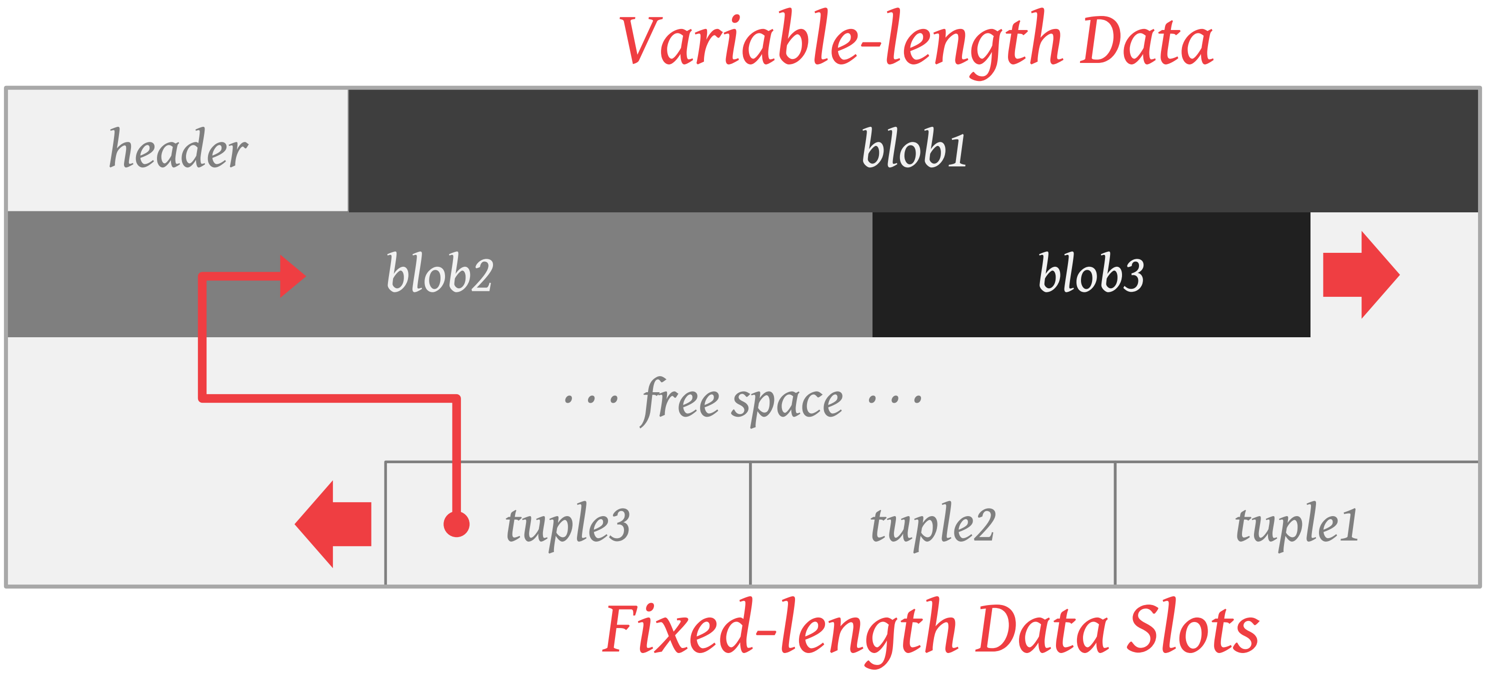
SLOTTED PAGES



SLOTTED PAGES



SLOTTED PAGES



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.

→ Has to set locks and latches to provide ACID guarantees for txns.

→ Locks are stored in a separate data structure to avoid being swapped to disk.

LOGGING & RECOVERY

Most DBMSs use **STEAL + NO-FORCE** buffer pool policies, so all modifications have to be flushed to the WAL before a txn can commit.

Each log entry contains the before and after image of record modified.

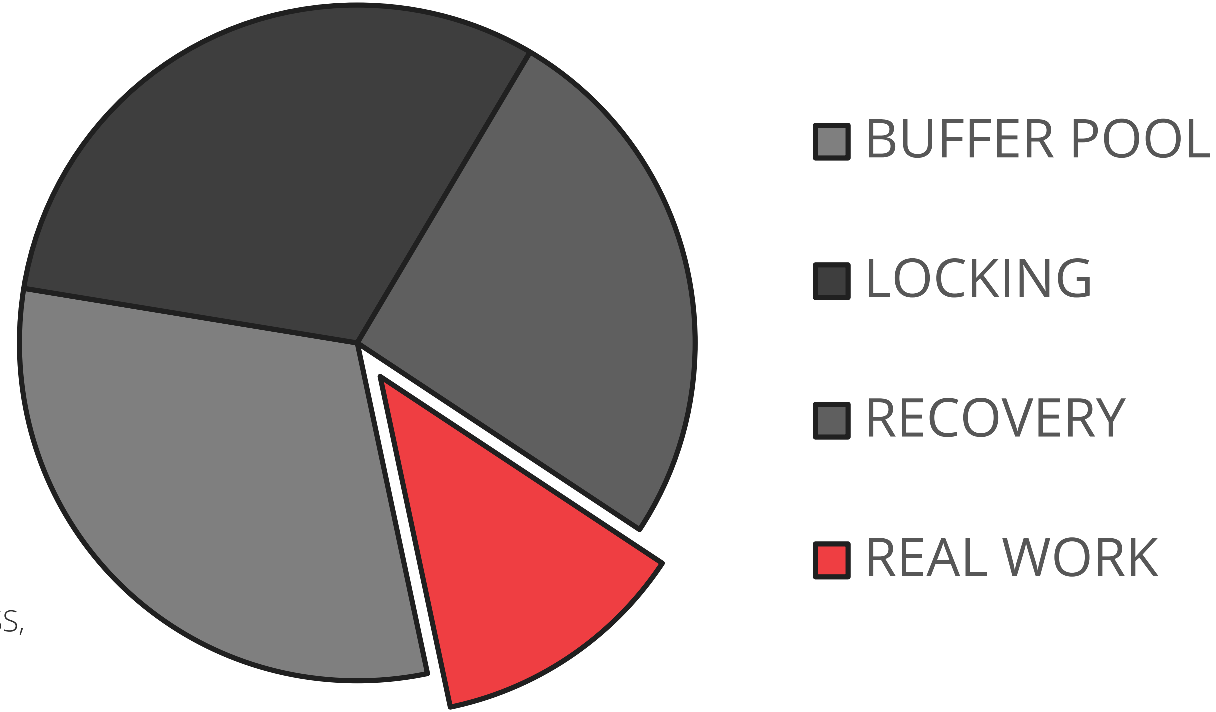
DISK-ORIENTED DBMS OVERHEAD

Measured CPU Cycles

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

DISK-ORIENTED DBMS OVERHEAD

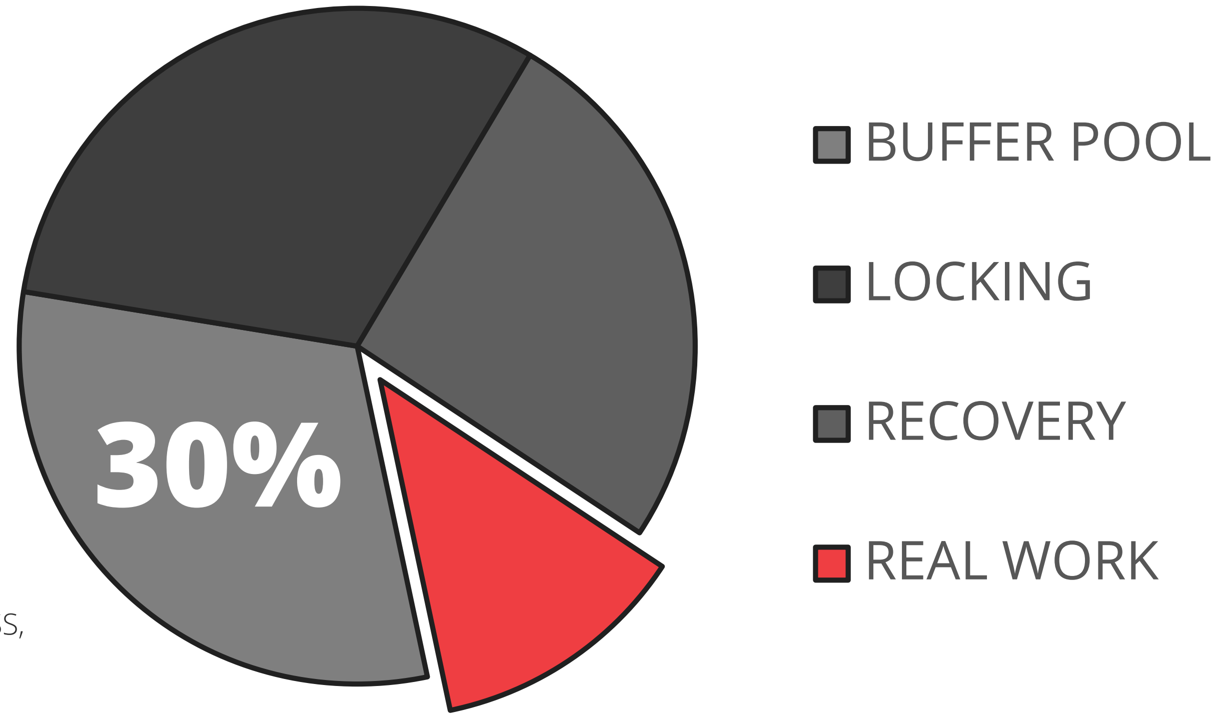
Measured CPU Cycles



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

DISK-ORIENTED DBMS OVERHEAD

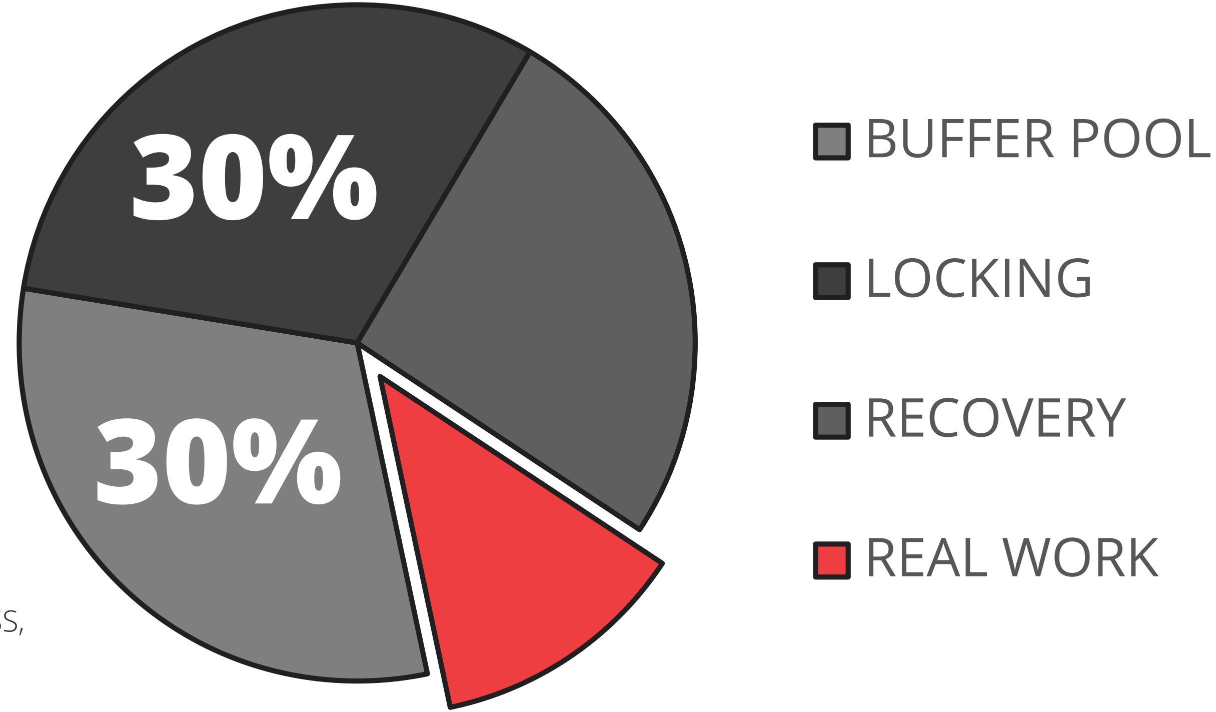
Measured CPU Cycles



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

DISK-ORIENTED DBMS OVERHEAD

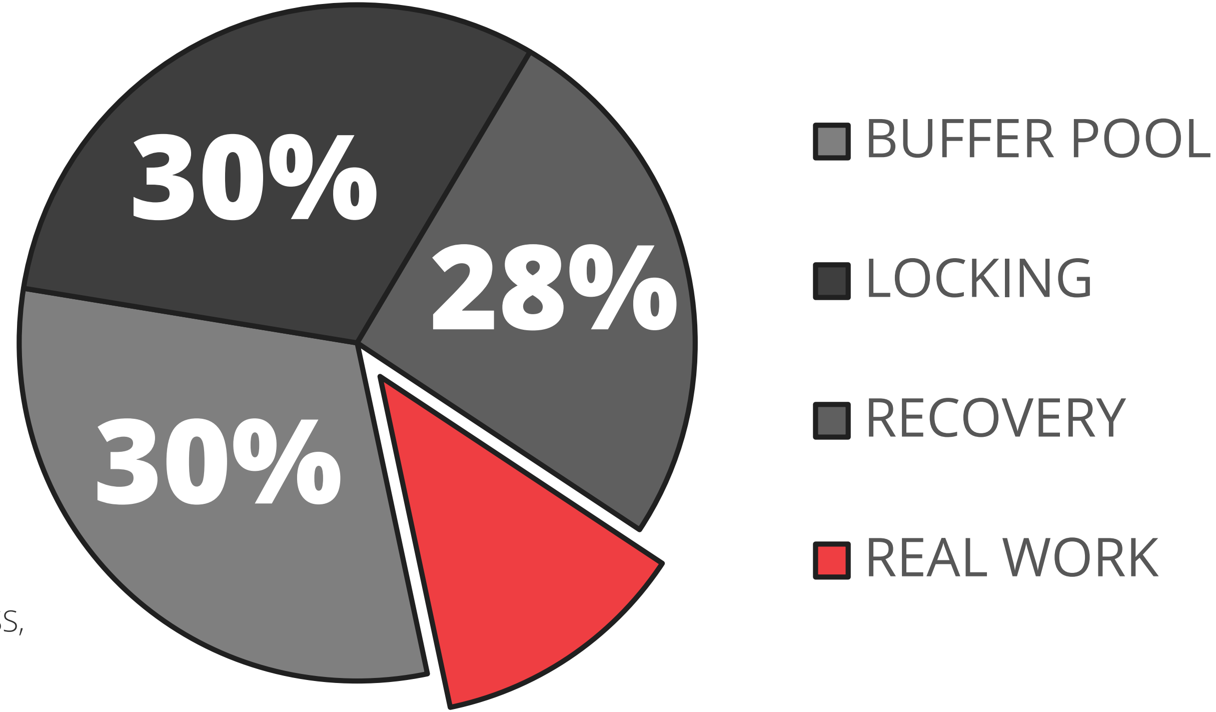
Measured CPU Cycles



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

DISK-ORIENTED DBMS OVERHEAD

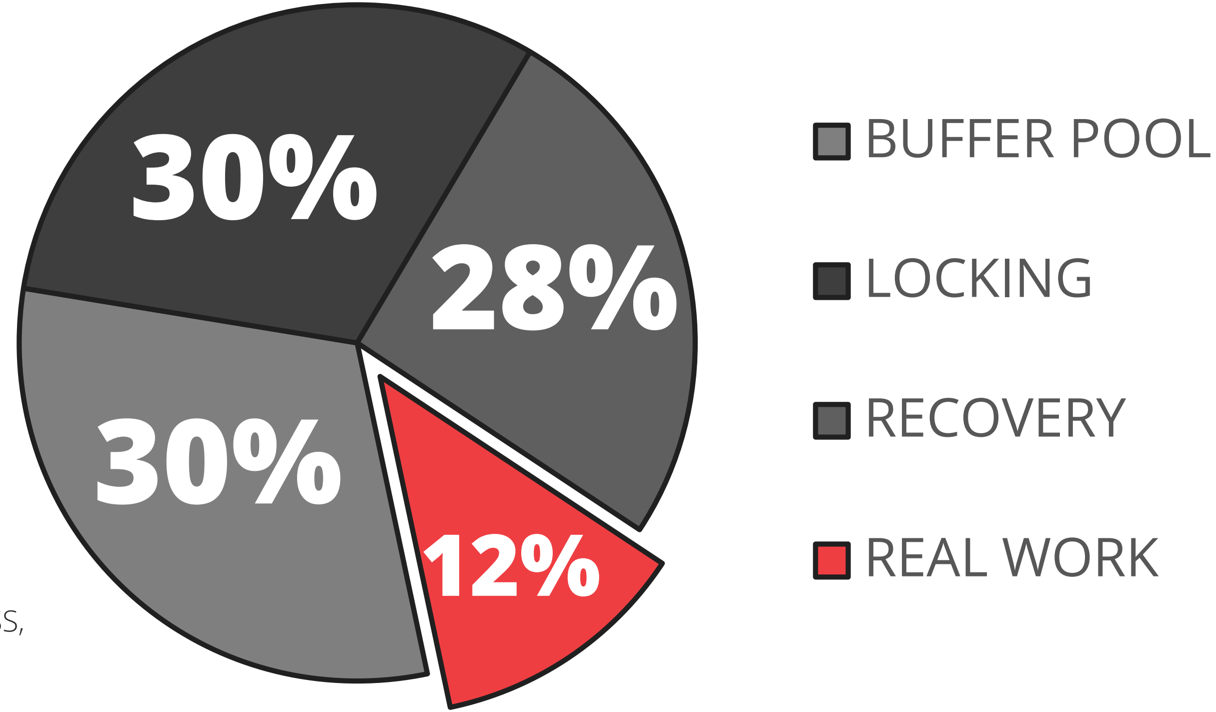
Measured CPU Cycles



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

DISK-ORIENTED DBMS OVERHEAD

Measured CPU Cycles



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

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.

WHY NOT MMAP?

Memory-map 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.

→ 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.

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

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-line misses
→ Pointer chasing
→ Predicate evaluations
→ Data movement & copying
→ 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.

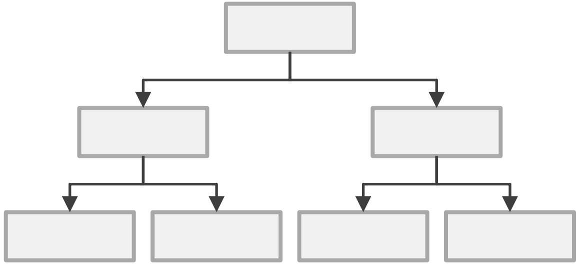
DATA ORGANIZATION

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

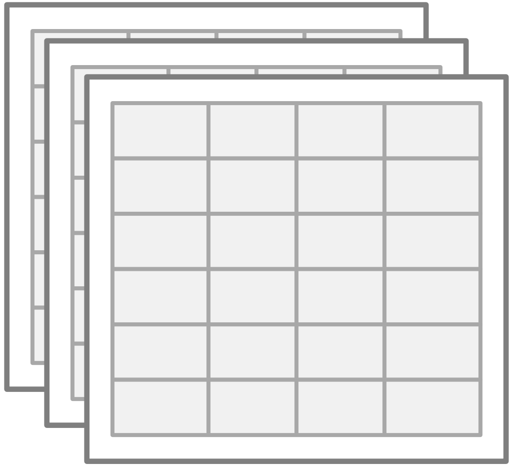
→ Direct memory pointers vs. record ids
→ Fixed-length vs. variable-length data pools
→ Use block checksums to detect software errors from trashing the database.

DATA ORGANIZATION

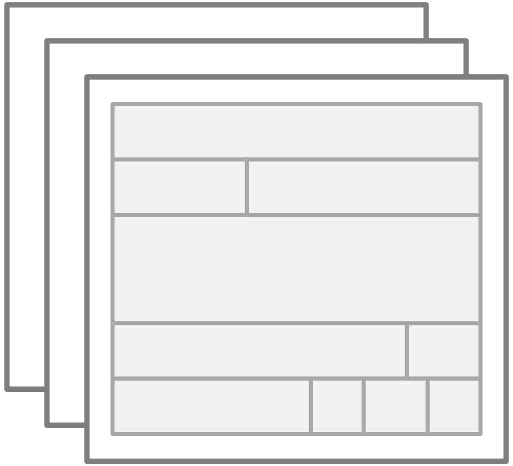
Index



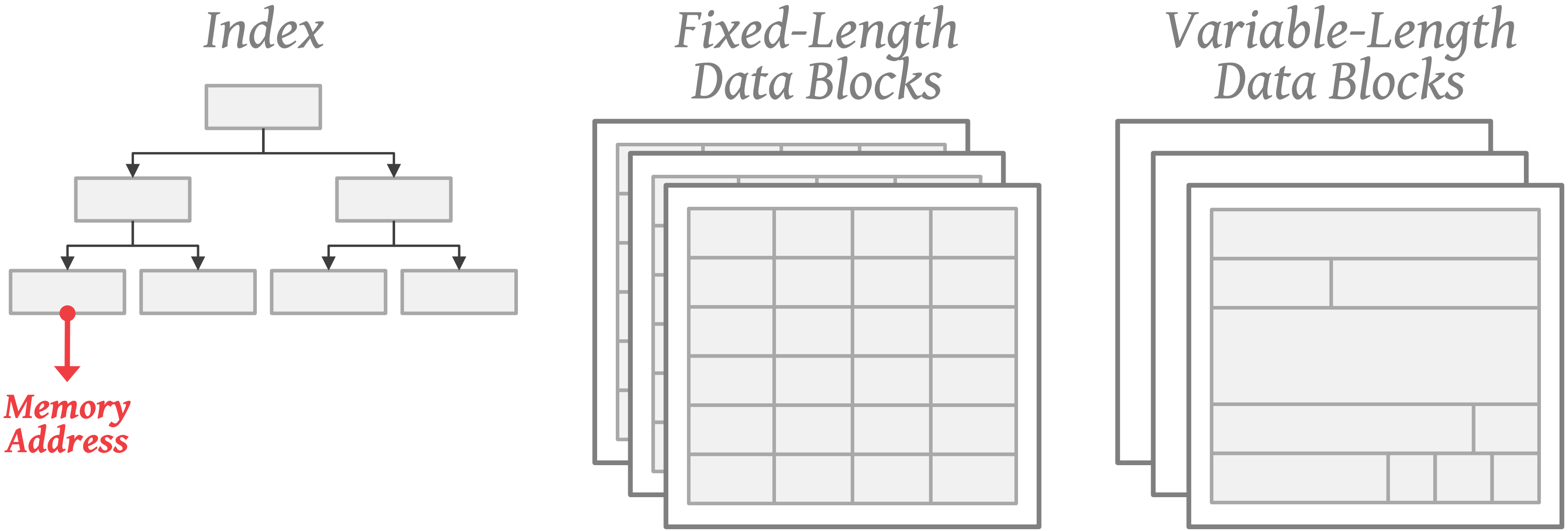
Fixed-Length Data Blocks



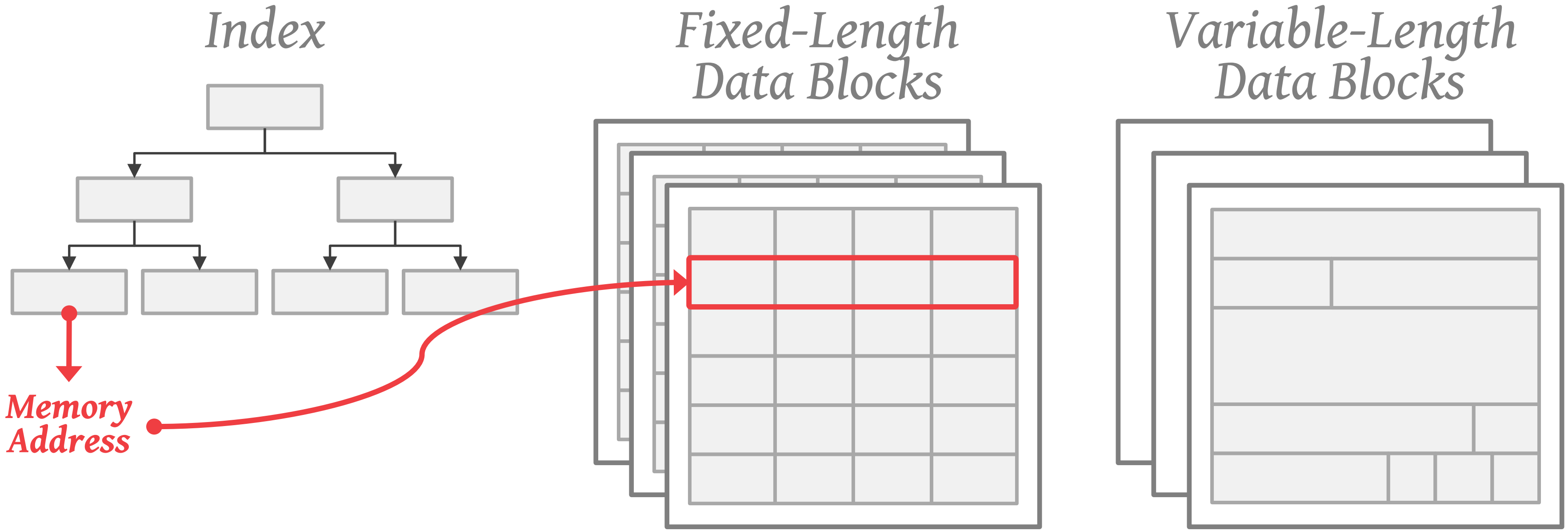
Variable-Length Data Blocks



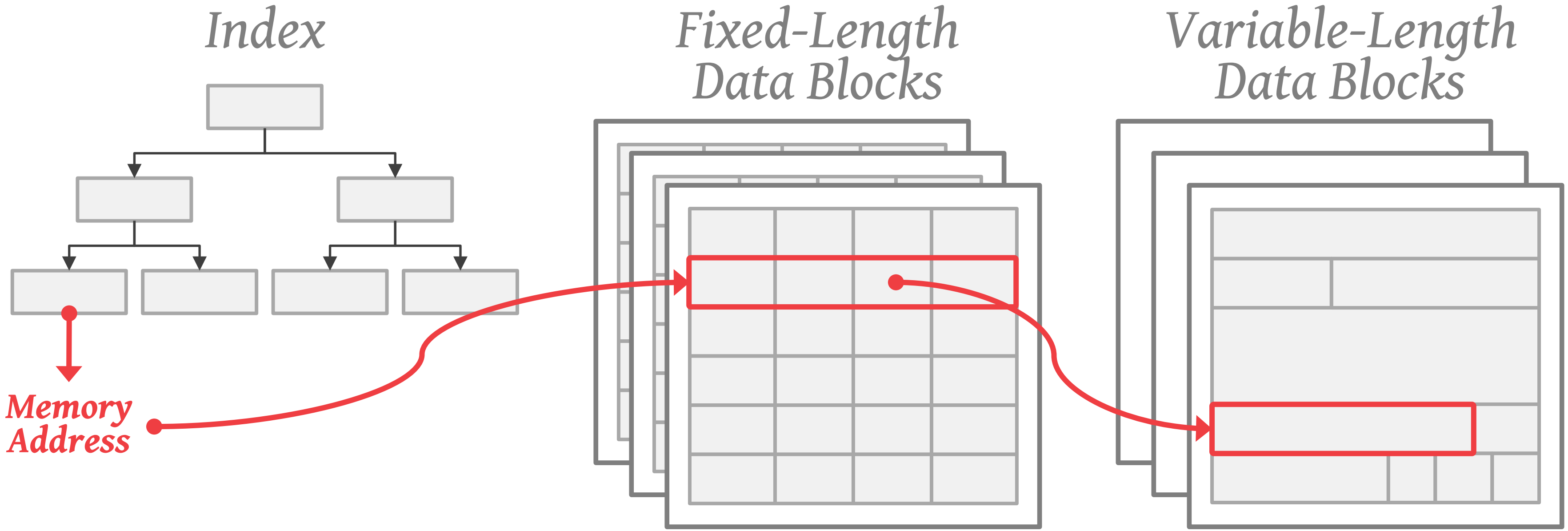
DATA ORGANIZATION



DATA ORGANIZATION



DATA ORGANIZATION



CONCURRENCY CONTROL

Observation: The cost of a txn acquiring a lock is the same as accessing data.

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.

New bottleneck is contention caused from txns trying access data at the same time.

INDEXES

Main-memory indexes 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

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

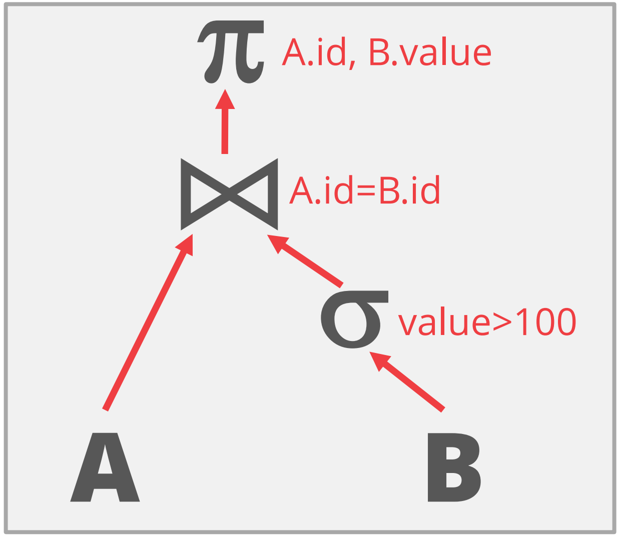
→ Sequential scans are no longer significantly faster than random access.

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

→ This problem is more significant in OLAP DBMSs.

QUERY PROCESSING

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

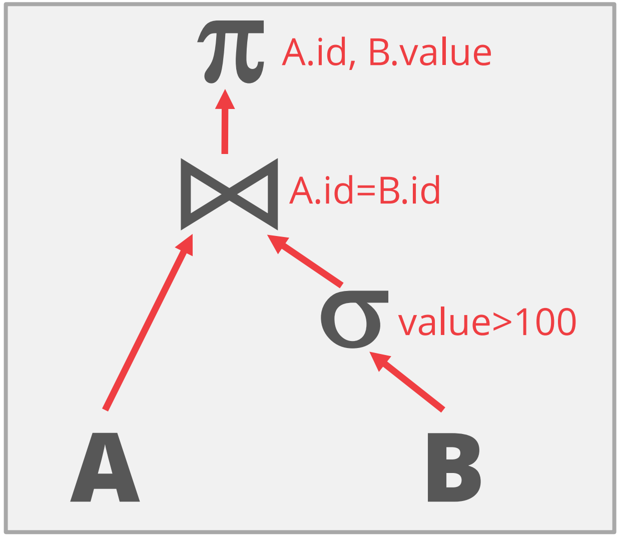


Tuple-at-a-time

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

QUERY PROCESSING

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



Tuple-at-a-time

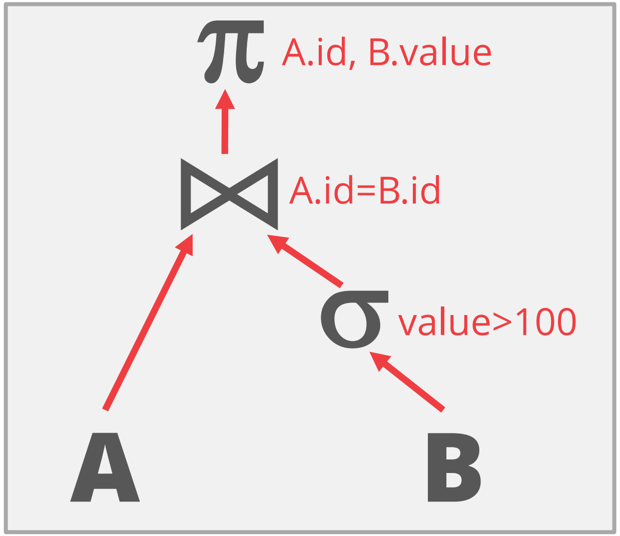
→ Each operator calls **next** on their child to get the next tuple to process.

Operator-at-a-time

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

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.

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 if using coarse-grained locking (redo only).

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.

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
→ Cold Data: OLAP Queries

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 memory.

Again, we'll cover this topic later.

NOTABLE IN-MEMORY DBMSs

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

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

NOTABLE IN-MEMORY DBMSs

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

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.

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.

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.

PELOTON DBMS

CMU's in-memory hybrid relational DBMS

→ Multi-version concurrency control.
→ Tile-based storage manager.
→ Multi-threaded architecture.
→ Based on PostgreSQL 9.3

Currently supports most of SQL-92.

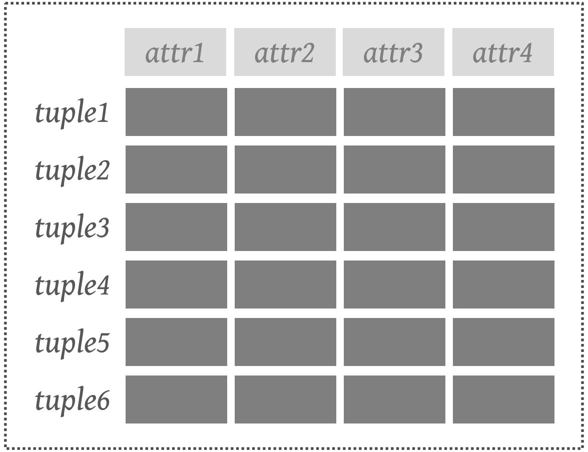
PELOTON DBMS

CMU's in-memory hybrid relational DBMS
→ Multi-version concurrency control.
→ Tile-based storage manager.
→ Multi-threaded architecture.
→ Based on PostgreSQL 9.3

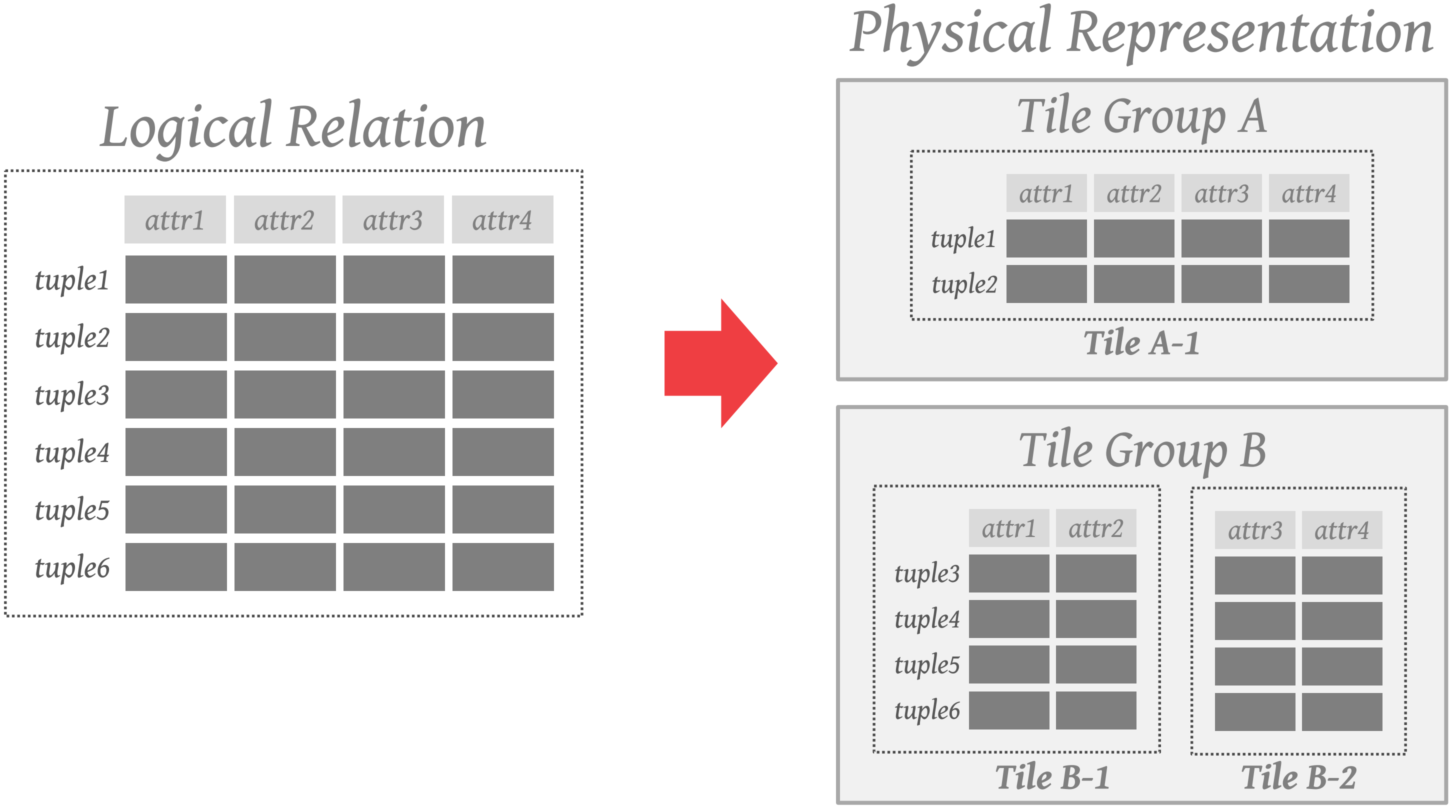
Currently supports most of SQL-92.

TILE STORAGE ARCHITECTURE

Logical Relation

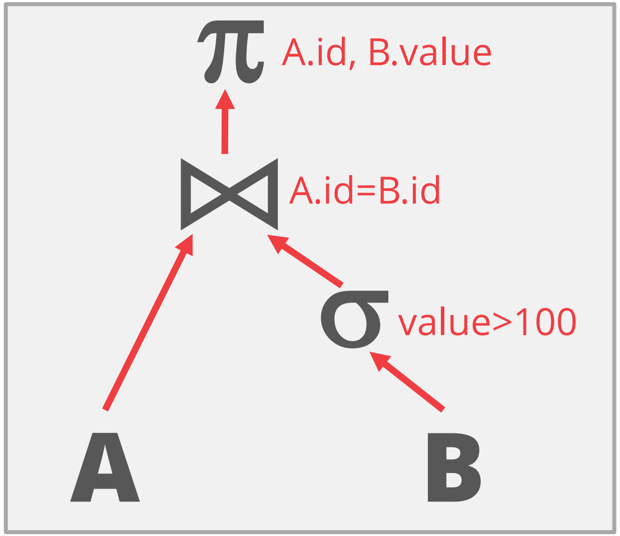


TILE STORAGE ARCHITECTURE



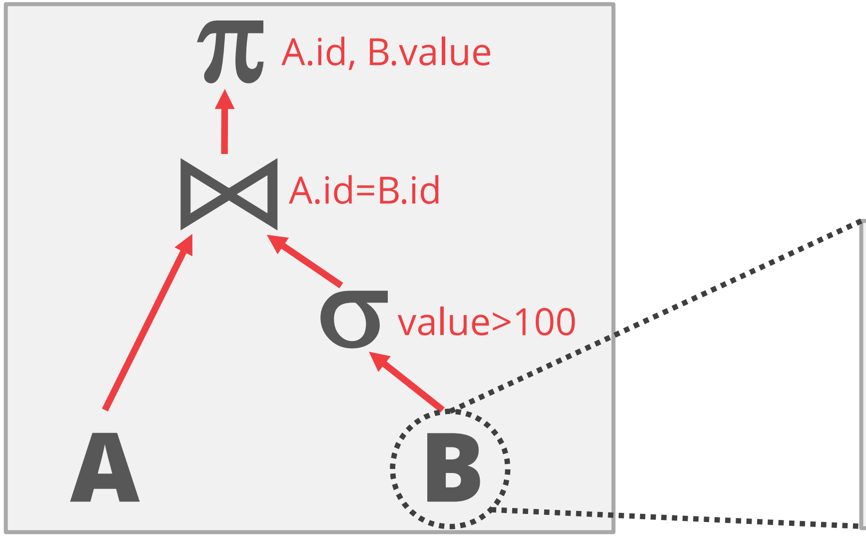
TILE STORAGE ARCHITECTURE

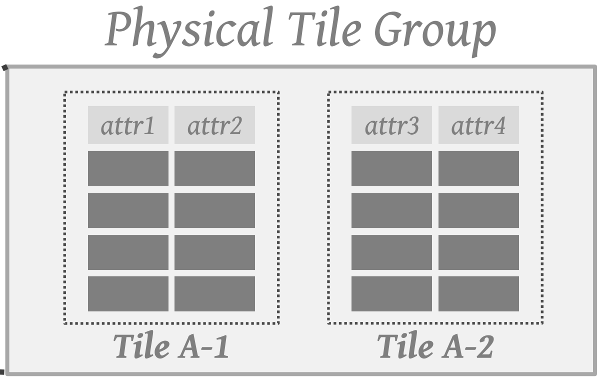
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



TILE STORAGE ARCHITECTURE

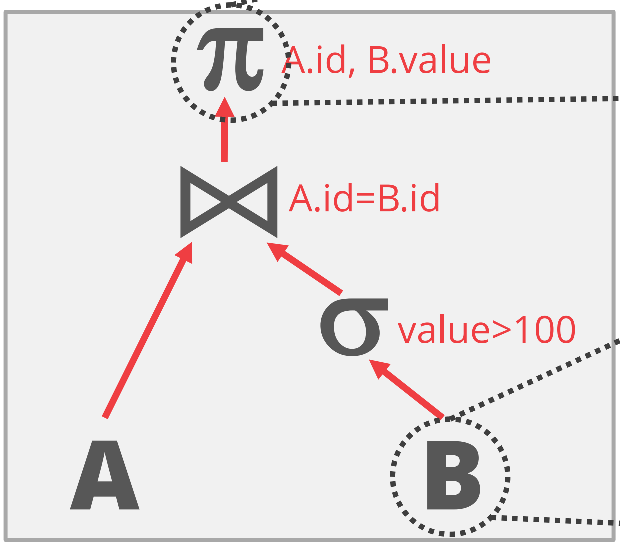
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

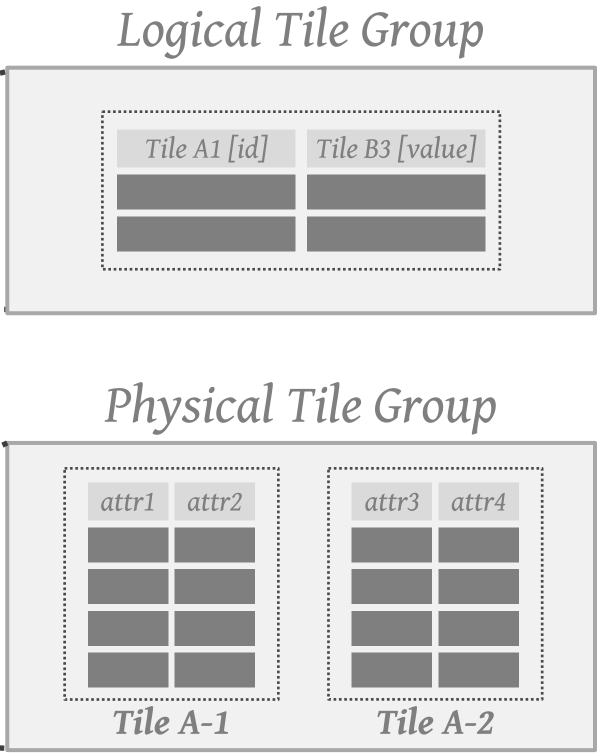




TILE STORAGE ARCHITECTURE

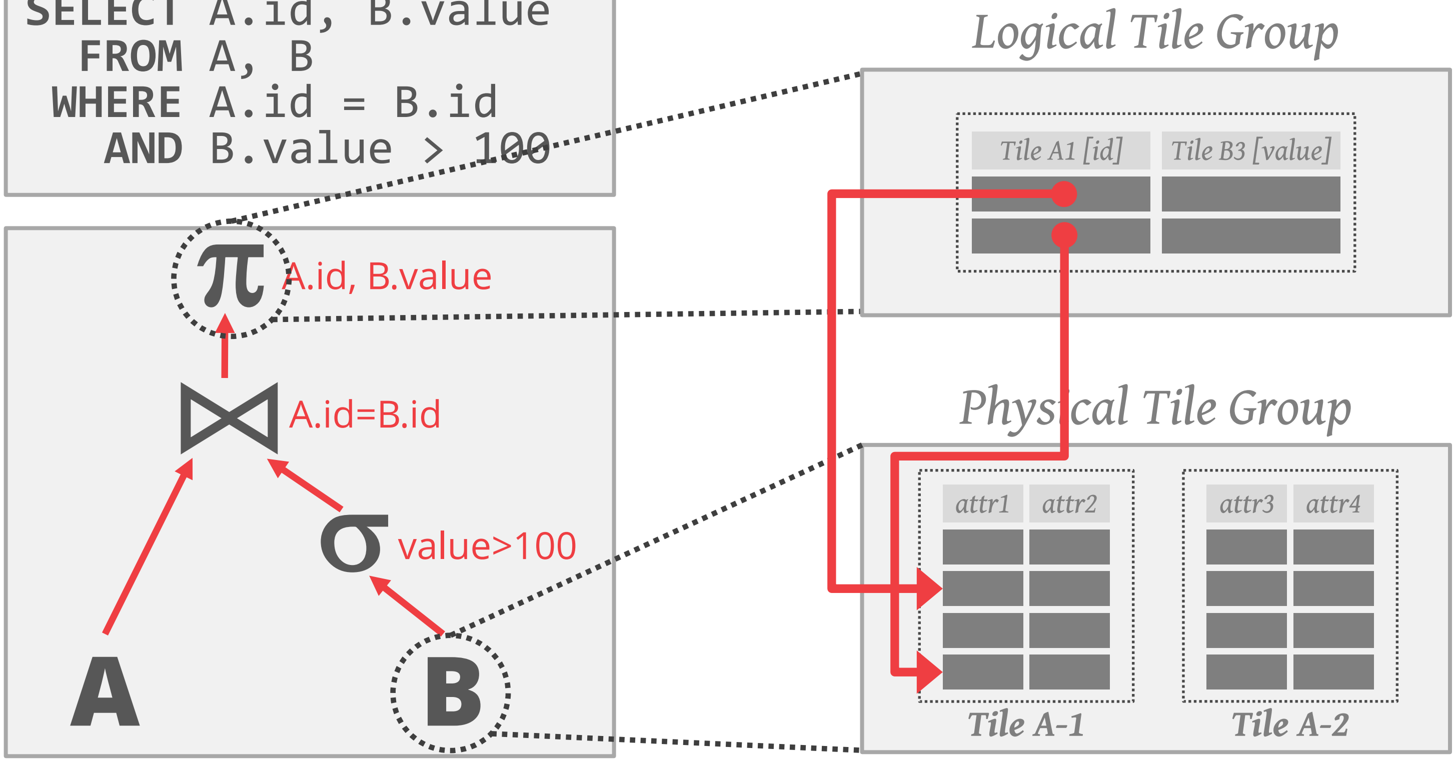
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```





TILE STORAGE ARCHITECTURE

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



PROJECT #1

Implement an in-memory hash join operator that supports four different join types:

→ INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN

You are free to implement either the "classic" algorithm or the GRACE hash join algorithm.

PROJECT #1 – TESTING

We are providing you with a C++ unit test for you check your implementation.

We also have a SQL batch script that will execute a couple different queries.

We strongly encourage you to do your own additional testing.

→ Make sure that you disable the other join types to force the optimizer to always pick hash join plans.

PROJECT #1 – GRADING

We will run additional tests beyond what we provided you for grading.

→ Bonus points will be given to the student with the fastest implementation.

→ We will use Valgrind when testing your code.

All source code must pass ClangFormat syntax formatting checker.

→ See Peloton documentation for formatting guidelines

DEVELOPMENT ENVIRONMENT

Peloton only builds on 64-bit Linux.

But you can do development on either Linux or OSX (through a VM).

→ We have a Vagrant config file to automatically create a development Ubuntu VM for you.

This is CMU so I'm going to assume that each of you are capable of getting access to a machine.

GITHUB PRIVATE REPO

If you want to use Github for your projects, you **must** use a private repo for Projects #1 and #2.

Sign up for a student account on Github to get five free private repositories:
https://education.github.com/pack

PROJECT #1

Due Date: February 8th, 2016 @ 11:59pm
Projects will be turned in using Autolab.

Full description and instructions:
http://15721.courses.cs.cmu.edu/spring2016/project1.html

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.

NEXT CLASS

Transactions & Concurrency Control