Lecture #02

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

In-Memory Databases

@Andy_Pavlo // 15-721 // Spring 2020

BACKGROUND

Much of the development 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.
→ The database had to be stored on disk.
→ Disks were even slower than they are now.

BACKGROUND

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

→ Structured data sets are smaller.
→ Unstructured or semi-structured data sets are larger.

We need to understand why we can't always use a "traditional" disk-oriented DBMS with a large cache to get the best performance.

TODAY'S AGENDA

Disk-Oriented DBMSs

In-Memory DBMSs

Concurrency Control Bottlenecks

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 **pages** (aka blocks).

The system uses an in-memory **buffer pool** to cache pages fetched from disk.

→ Its job is to manage the movement of those pages 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:

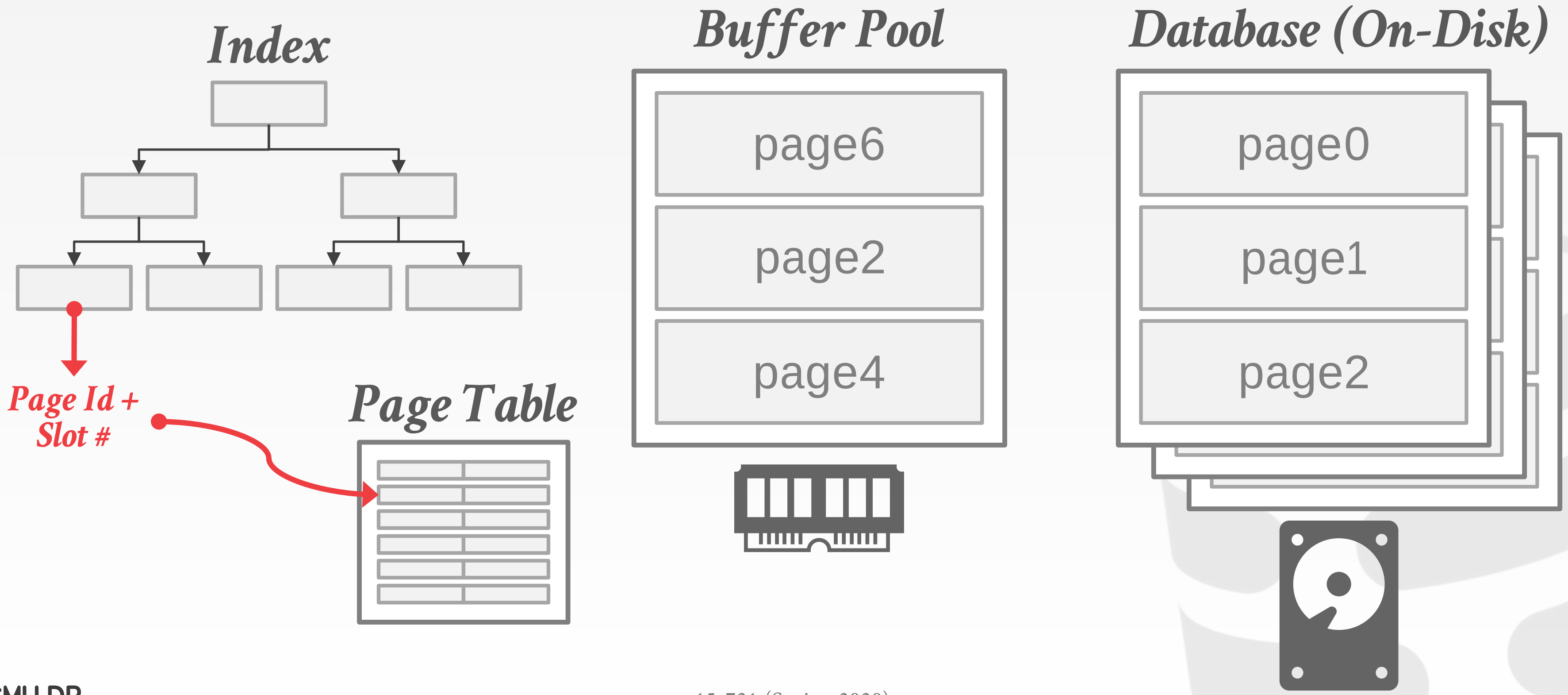
→ If it's not, then the DBMS must retrieve it from disk and copy it into a **frame** in its buffer pool.

→ If there are no free frames, then find a page to evict.

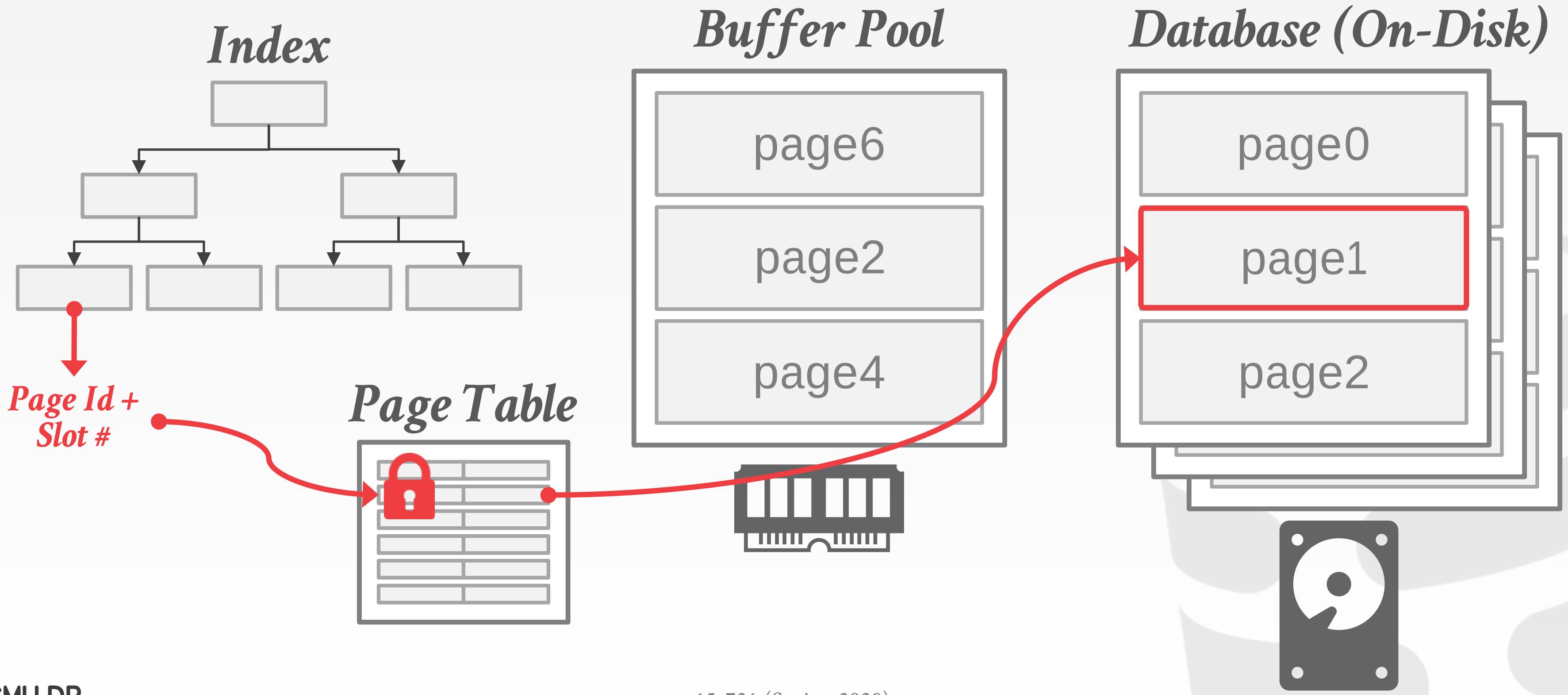
→ If the page being evicted is dirty, then the DBMS must write it back to disk.

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

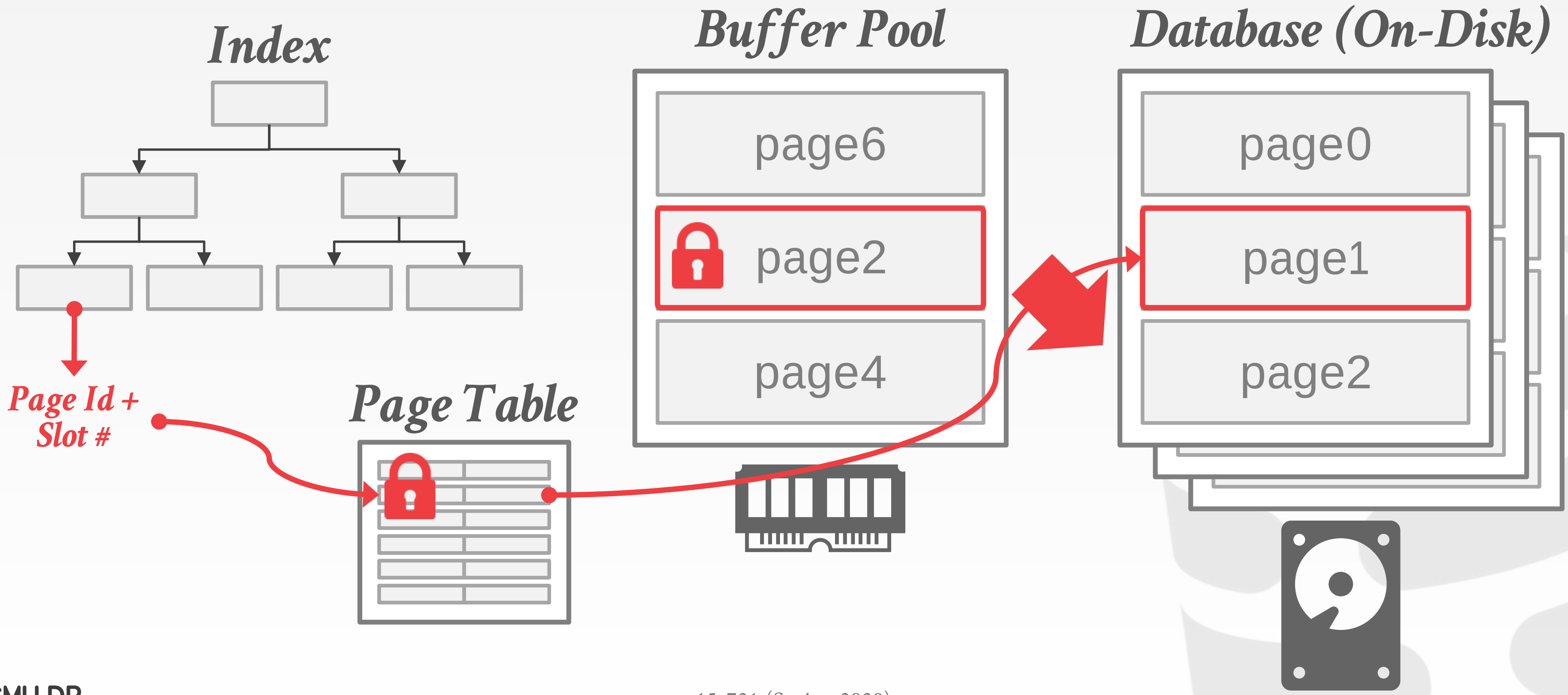
DISK-ORIENTED DATA ORGANIZATION



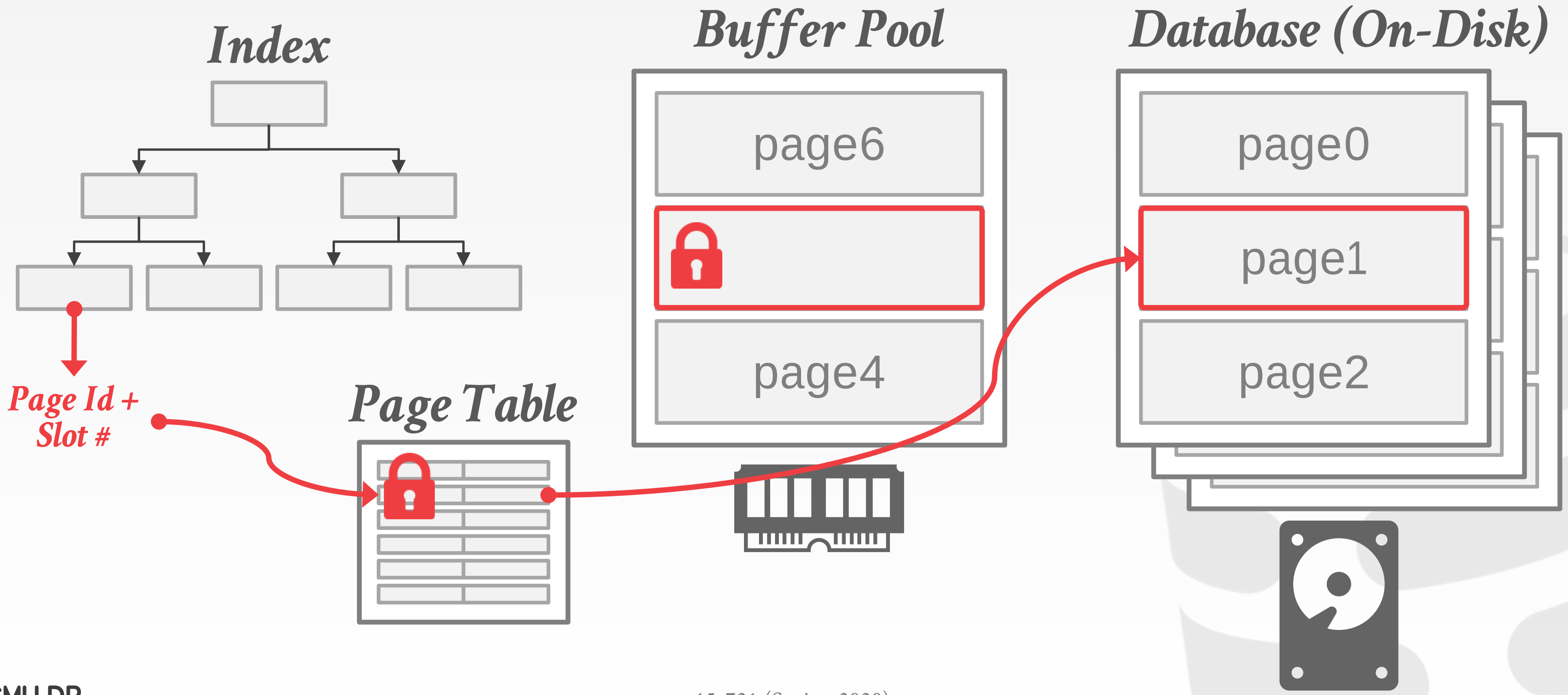
DISK-ORIENTED DATA ORGANIZATION



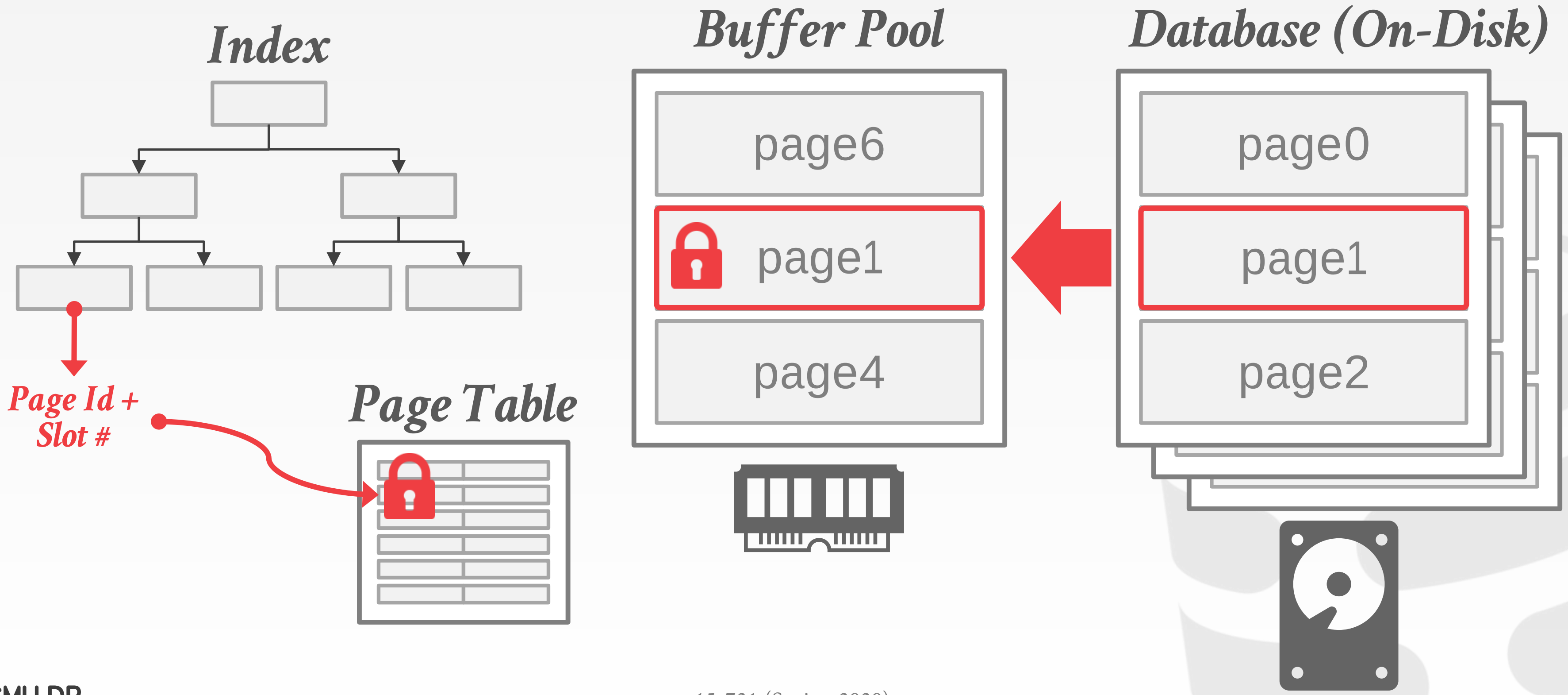
DISK-ORIENTED DATA ORGANIZATION



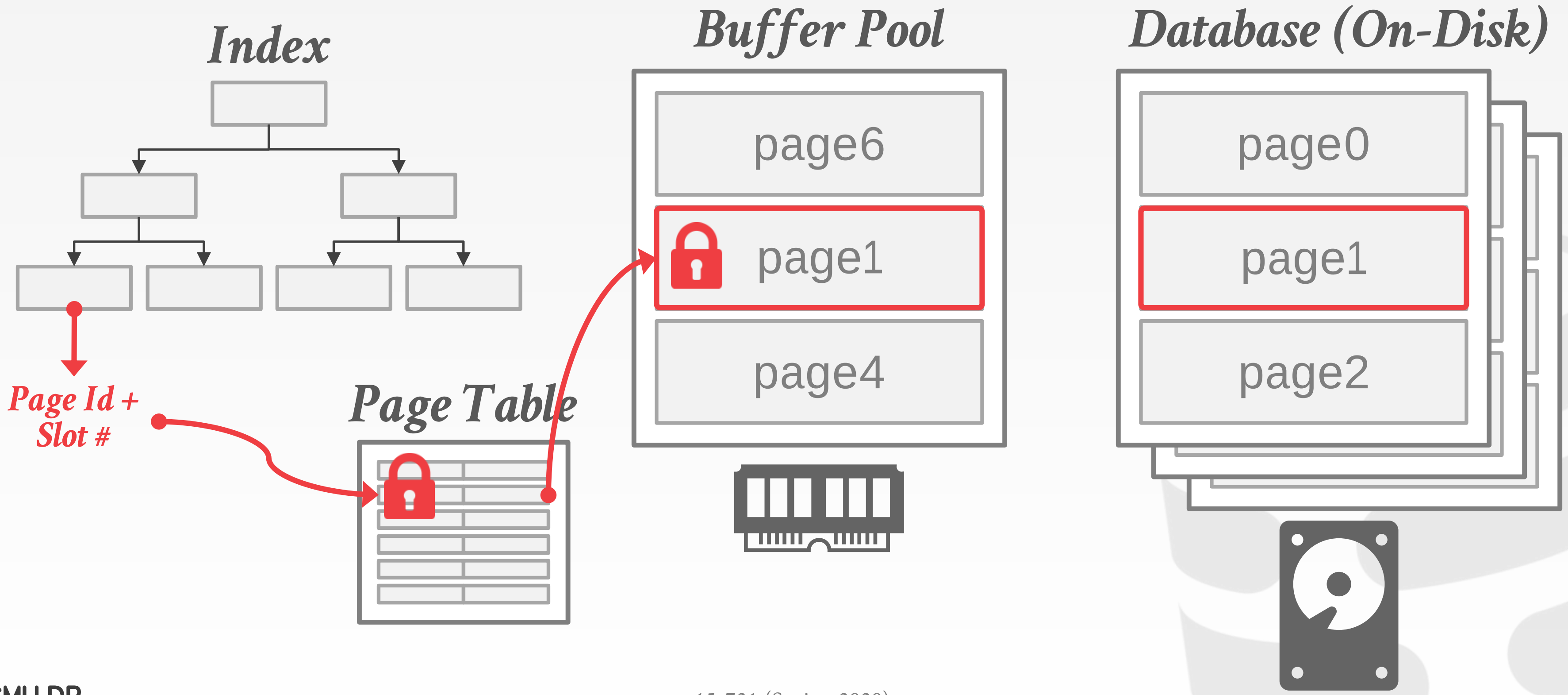
DISK-ORIENTED DATA ORGANIZATION



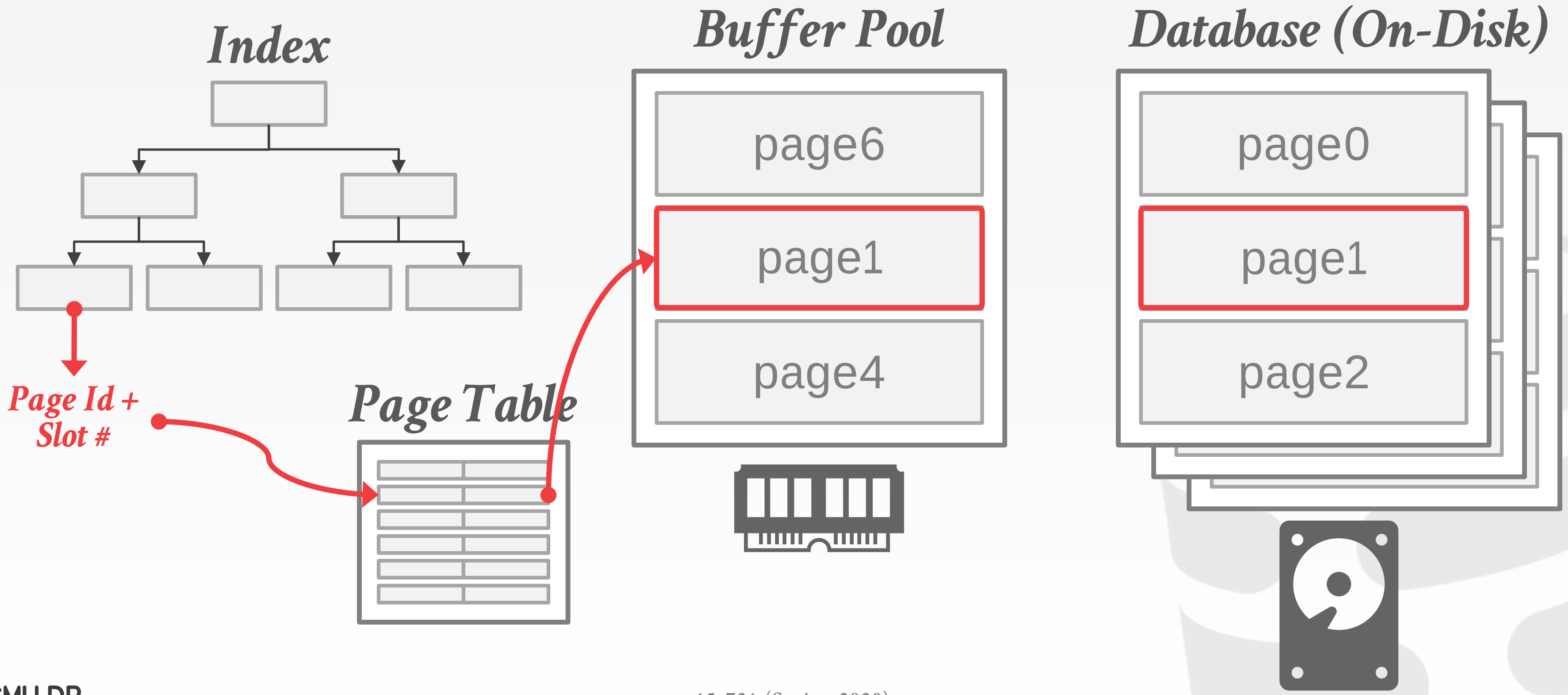
DISK-ORIENTED DATA ORGANIZATION



DISK-ORIENTED DATA ORGANIZATION



DISK-ORIENTED DATA ORGANIZATION



BUFFER POOL

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

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

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

CONCURRENCY CONTROL

The systems assumes that a txn could stall at any time whenever 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.
→ Set locks to provide ACID guarantees for txns.
→ Locks are stored in a separate data structure to avoid being swapped to disk.

LOGGING & RECOVERY

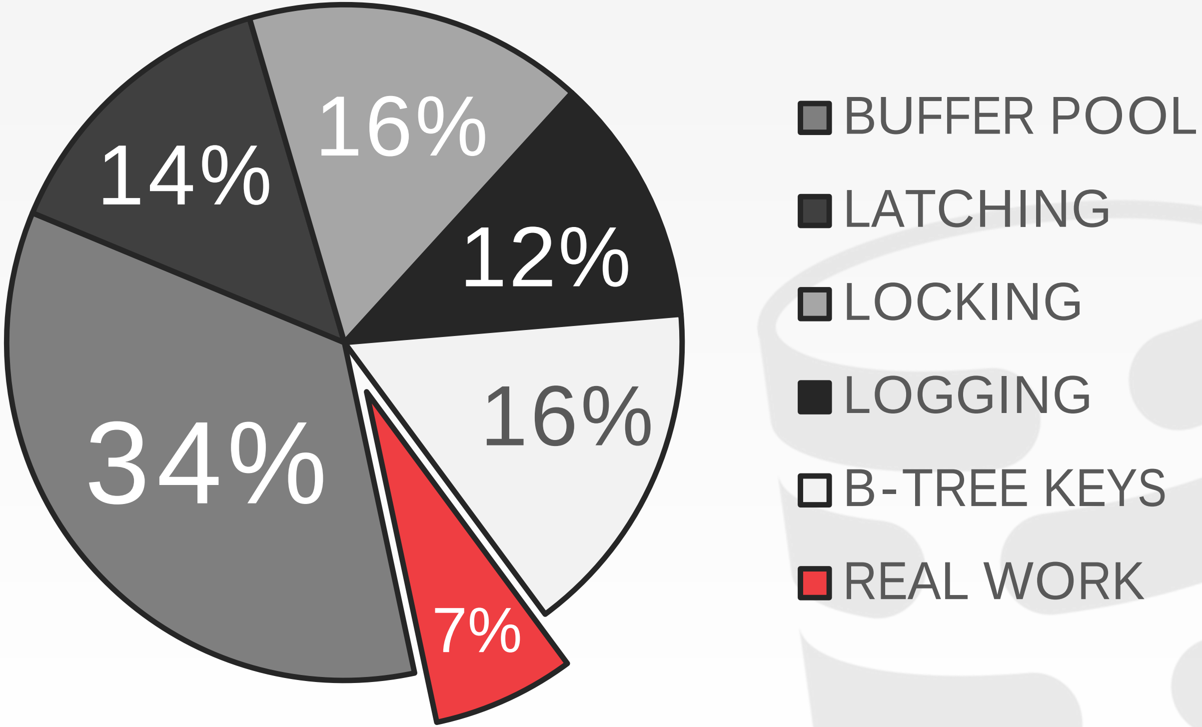
Most DBMSs use **STEAL** + **NO-FORCE** buffer pool policies, so all modifications have to be flushed to the WAL before a txn can commit.

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

Lots of work to keep track of LSNs all throughout the DBMS.

DISK-ORIENTED DBMS OVERHEAD

Measured CPU Instructions



OLTP THROUGH THE LOOKING GLASS, AND WHAT WE FOUND THERE
SIGMOD 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.

First commercial in-memory DBMSs were released in the 1990s.
→ Examples: TimesTen, DataBlitz, Altibase

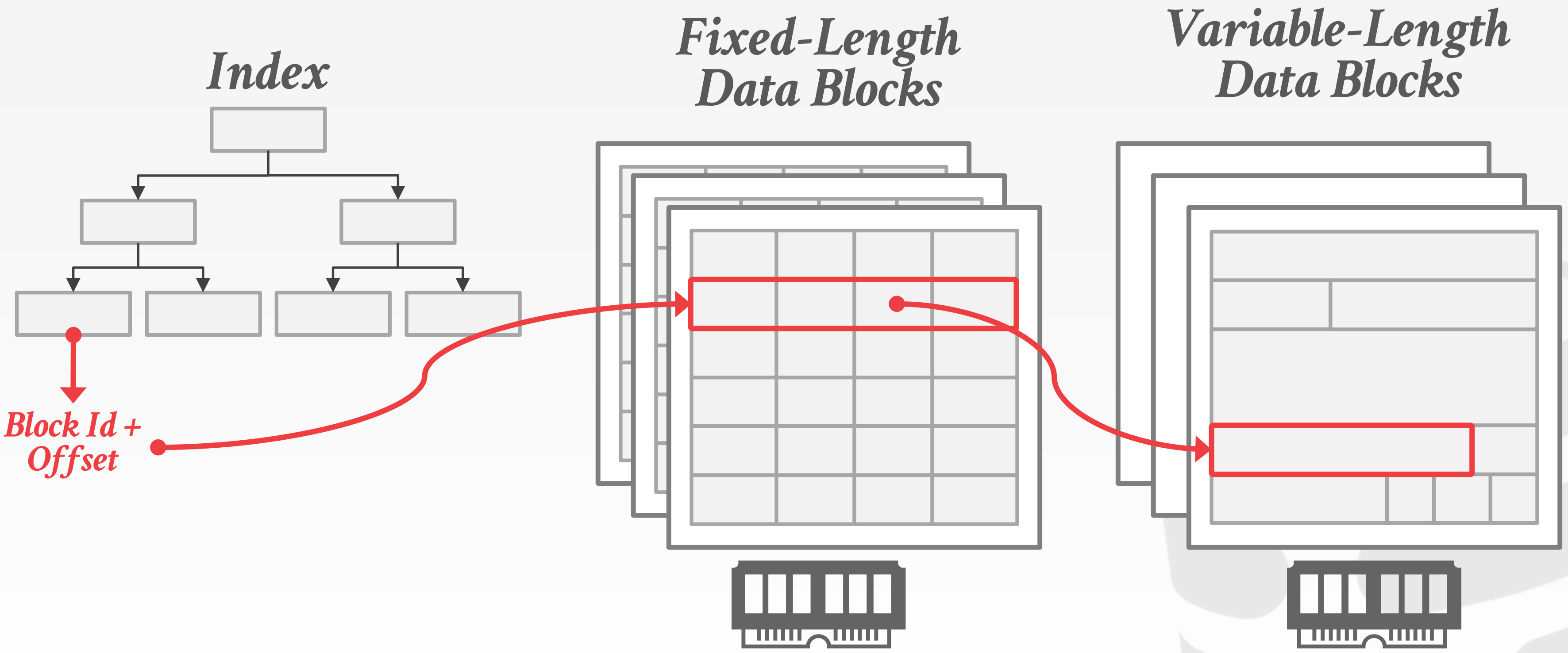
DATA ORGANIZATION

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

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

IN-MEMORY DATA ORGANIZATION



INDEXES

Specialized 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.

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).

Since there are no "dirty" pages, there is no need to track LSNs throughout the system.

BOTTLENECKS

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

→ $\boxed{\text{Locking/latching}}$
→ Cache-line misses
→ Pointer chasing
→ Predicate evaluations
→ Data movement & copying
→ Networking (between application & DBMS)

CONCURRENCY CONTROL

The protocol to allow txns to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system.

→ The goal is to have the effect of a group of txns on the database's state is equivalent to any serial execution of all txns.

Provides **A**tomicity + **I**solation in ACID

CONCURRENCY CONTROL

For in-memory DBMSs, the cost of a txn acquiring a lock is the same as accessing data.

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

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 **compare-and-swap** (CAS) instructions.

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V’** in **M**

→ Otherwise operation fails

Address *New Value*

M

20

```
__sync_bool_compare_and_swap(&M, 20, 30)
```

Compare Value

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails

New Value

Address

M

30

```
__sync_bool_compare_and_swap(&M, 20, 30)
```

Compare Value

CONCURRENCY CONTROL SCHEMES

Two-Phase Locking (2PL)

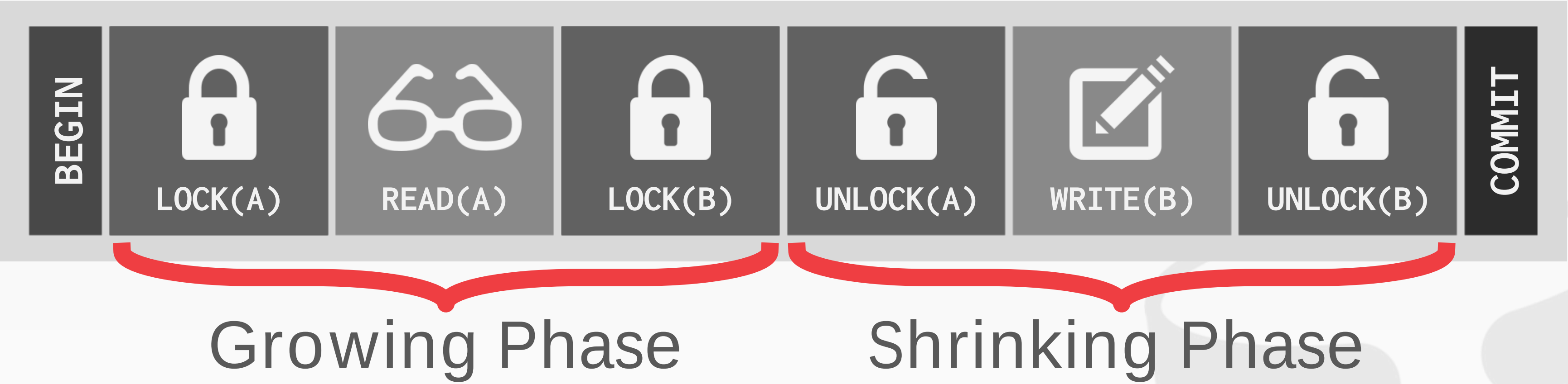
→ Assume txns will conflict so they must acquire locks on database objects before they are allowed to access them.

Timestamp Ordering (T/O)

→ Assume that conflicts are rare so txns do not need to first acquire locks on database objects and instead check for conflicts at commit time.

TWO-PHASE LOCKING

Txn #1



TWO-PHASE LOCKING

Txn #1

BEGIN | LOCK(A) | READ(A) | LOCK(B) | UNLOCK(A) | WRITE(B) | UNLOCK(B) | COMMIT

Txn #2

BEGIN | LOCK(B) | WRITE(B) | LOCK(A) | WRITE(A) | UNLOCK(A) | UNLOCK(B) | COMMIT

TWO-PHASE LOCKING

Txn #1

BEGIN	LOCK(A)	READ(A)	LOCK(B)	UNLOCK(A)	WRITE(B)	UNLOCK(B)	COMMIT

Txn #2

BEGIN	LOCK(B)	WRITE(B)	LOCK(A)	WRITE(A)	UNLOCK(A)	UNLOCK(B)	COMMIT

TWO-PHASE LOCKING

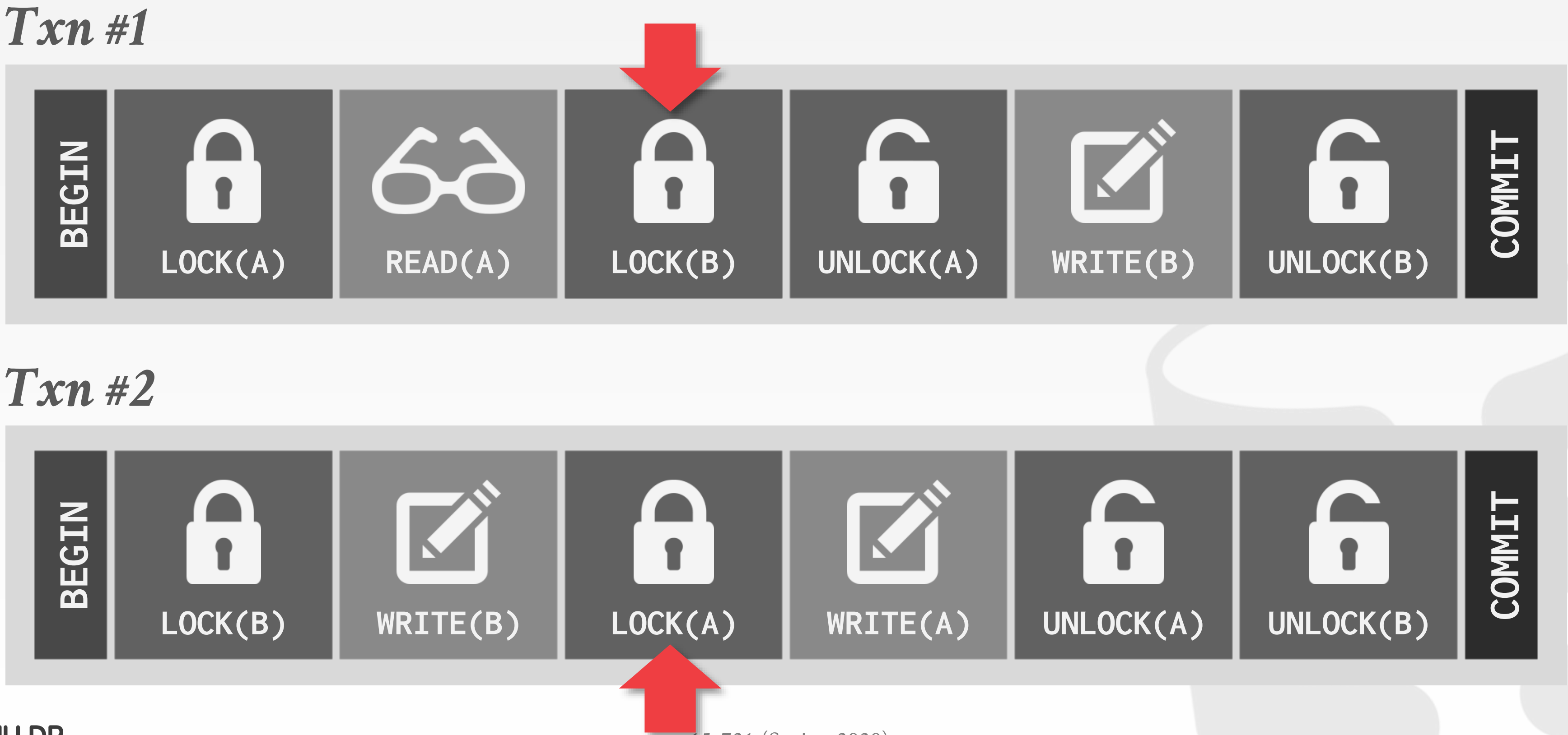
Txn #1

BEGIN	LOCK(A)	READ(A)	LOCK(B)	UNLOCK(A)	WRITE(B)	UNLOCK(B)	COMMIT

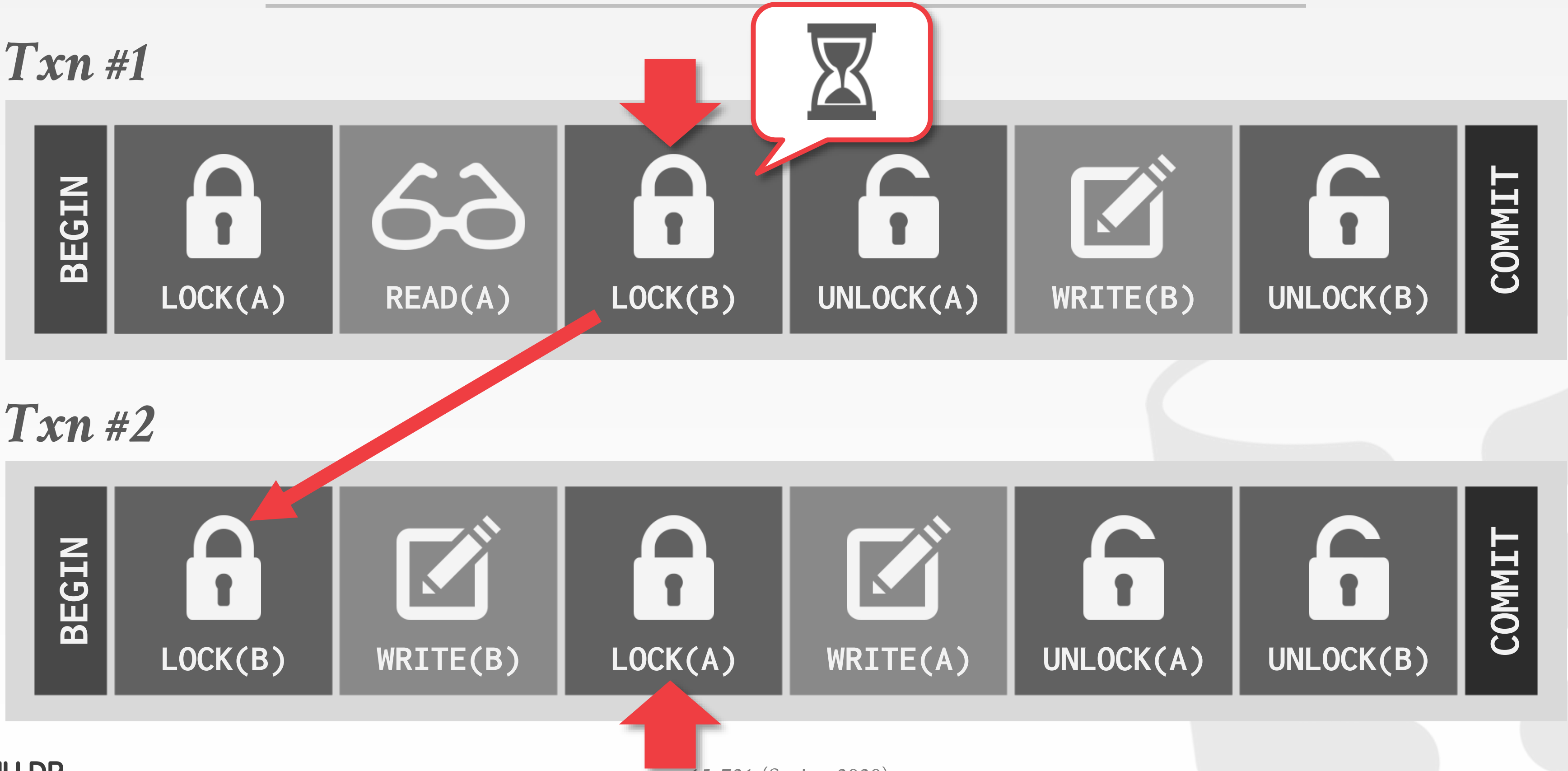
Txn #2

BEGIN	LOCK(B)	WRITE(B)	LOCK(A)	WRITE(A)	UNLOCK(A)	UNLOCK(B)	COMMIT

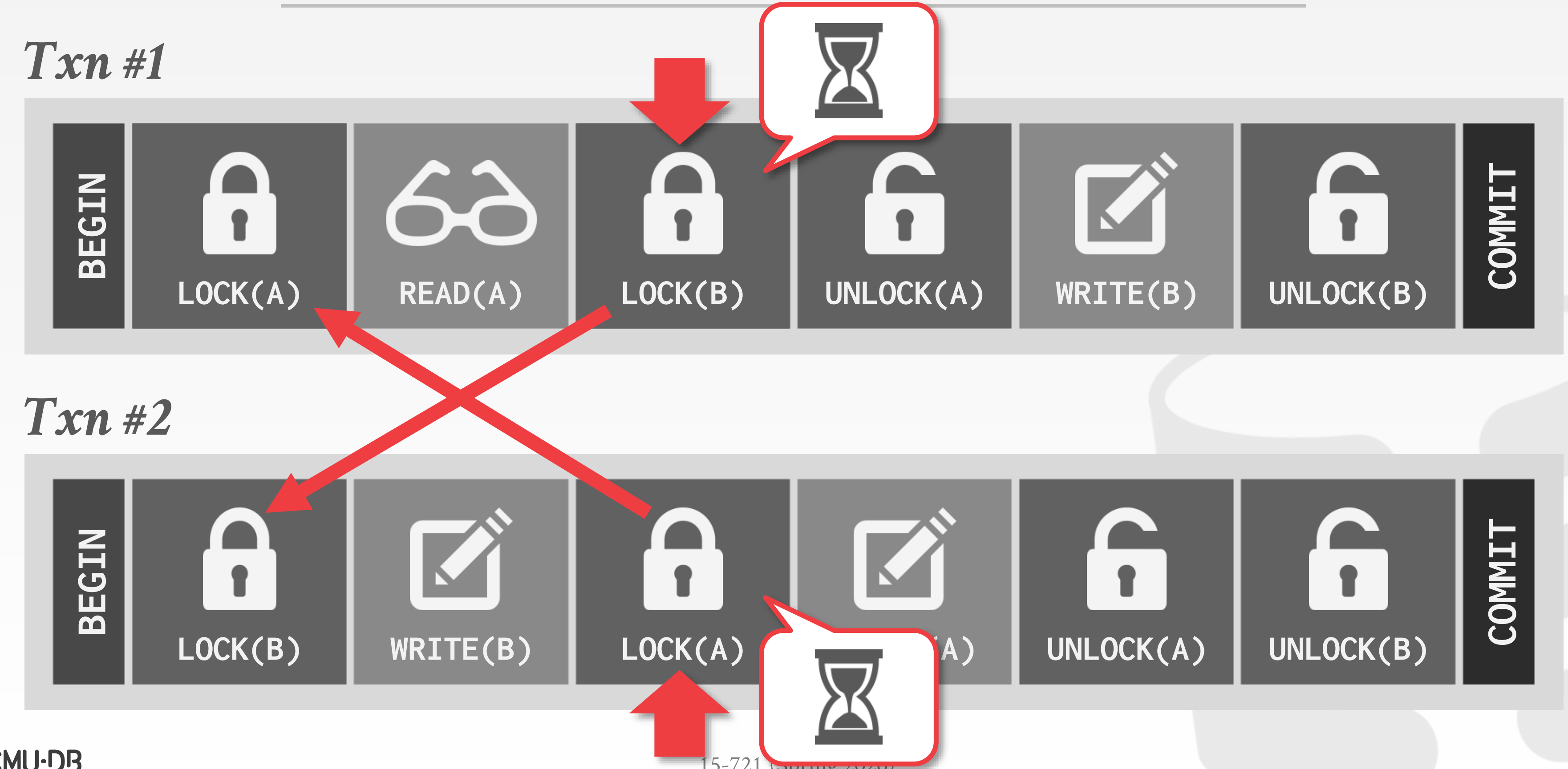
TWO-PHASE LOCKING



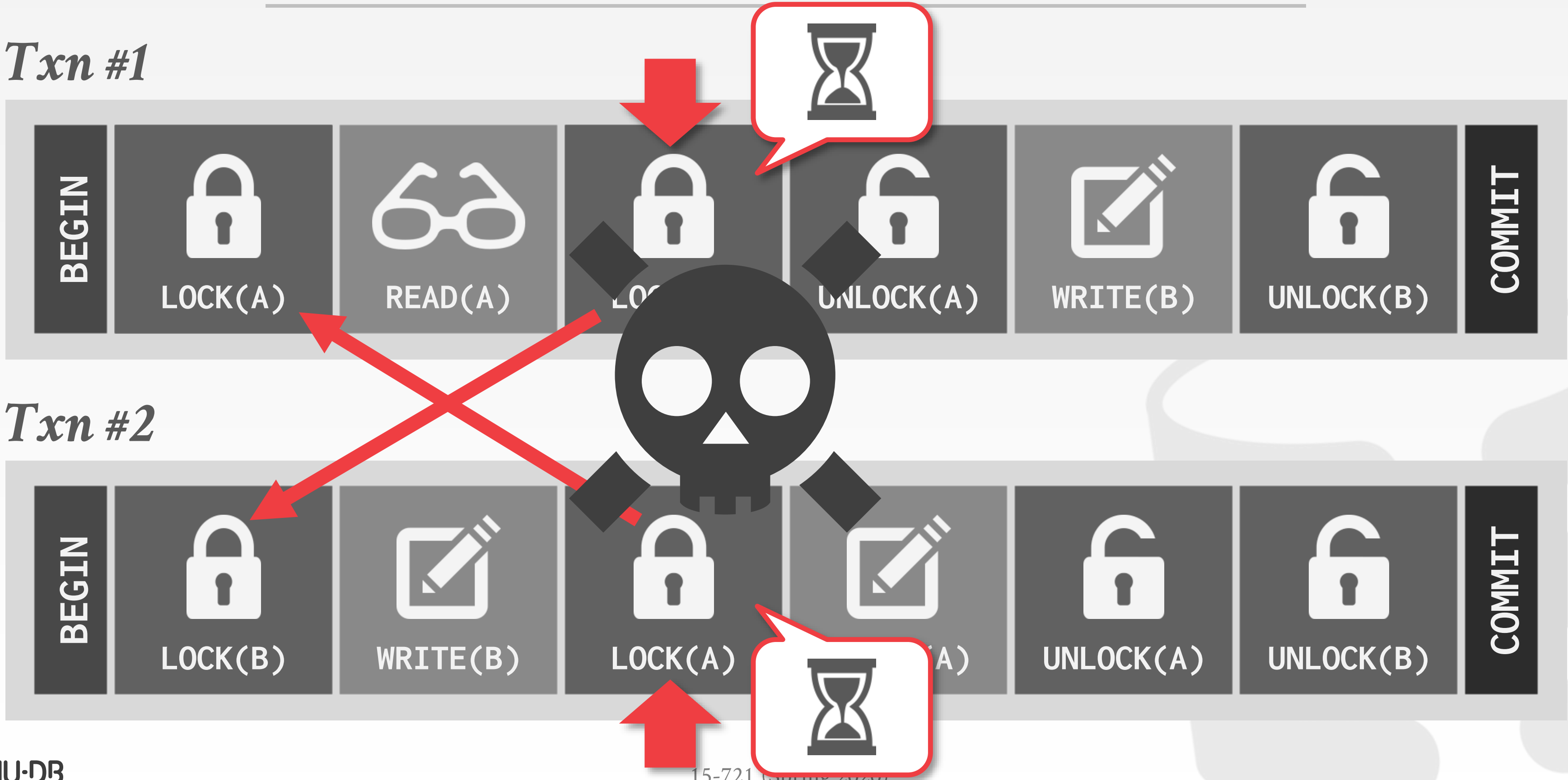
TWO-PHASE LOCKING



TWO-PHASE LOCKING



TWO-PHASE LOCKING



TWO-PHASE LOCKING

Deadlock Detection

→ Each txn maintains a queue of the txns that hold the locks that it waiting for.
→ A separate thread checks these queues for deadlocks.
→ If deadlock found, use a heuristic to decide what txn to kill in order to break deadlock.

Deadlock Prevention

→ Check whether another txn already holds a lock when another txn requests it.
→ If lock is not available, the txn will either (1) wait, (2) commit suicide, or (3) kill the other txn.

TIMESTAMP ORDERING

Basic T/O

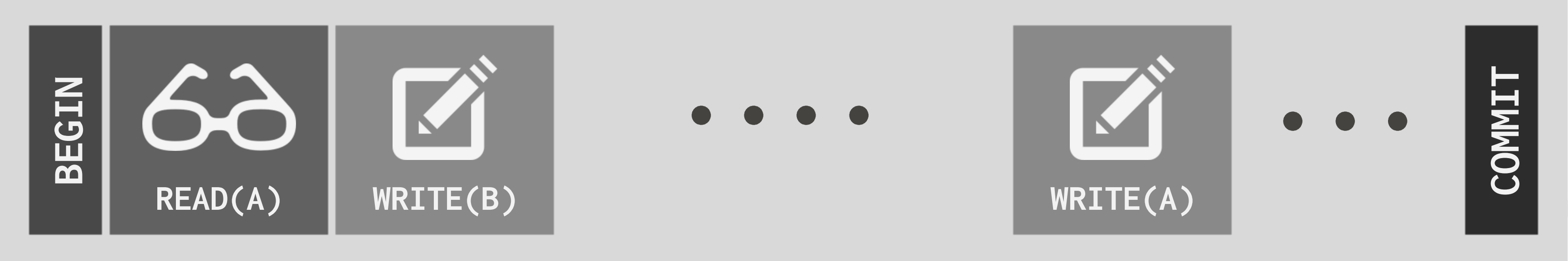
→ Check for conflicts on each read/write.
→ Copy tuples on each access to ensure repeatable reads.

Optimistic Currency Control (OCC)

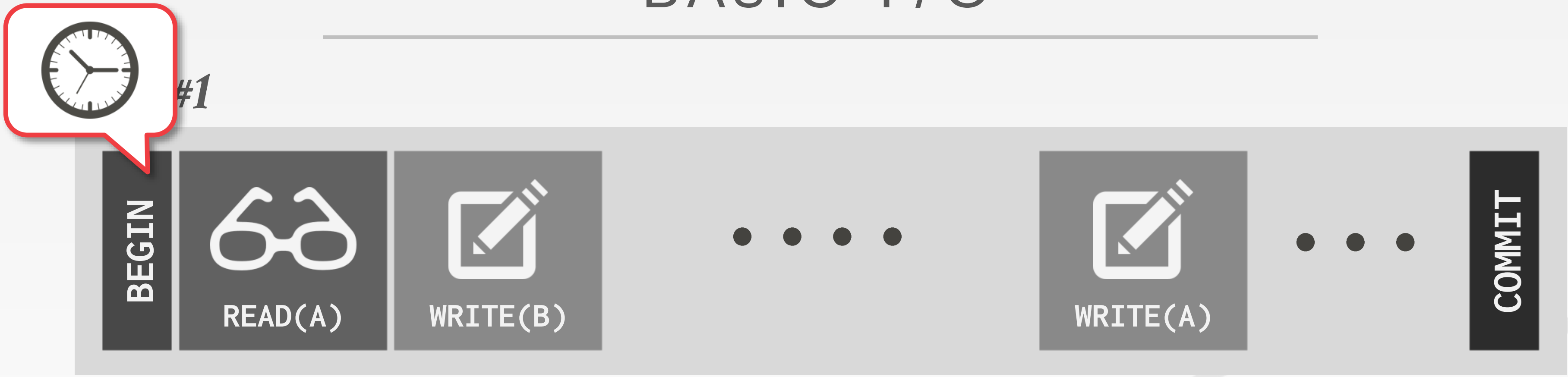
→ Store all changes in private workspace.
→ Check for conflicts at commit time and then merge.

BASIC T/O

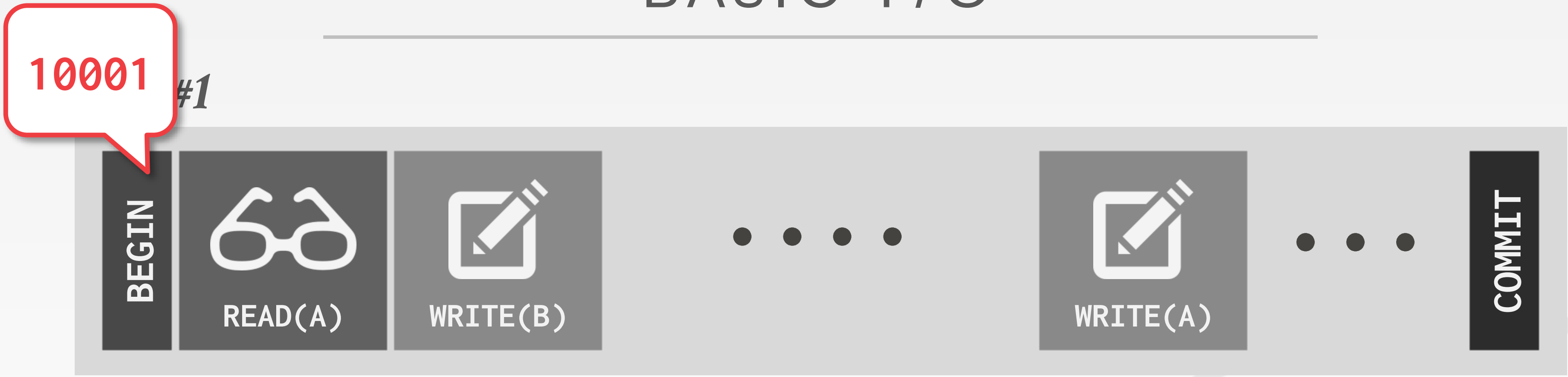
Txn #1



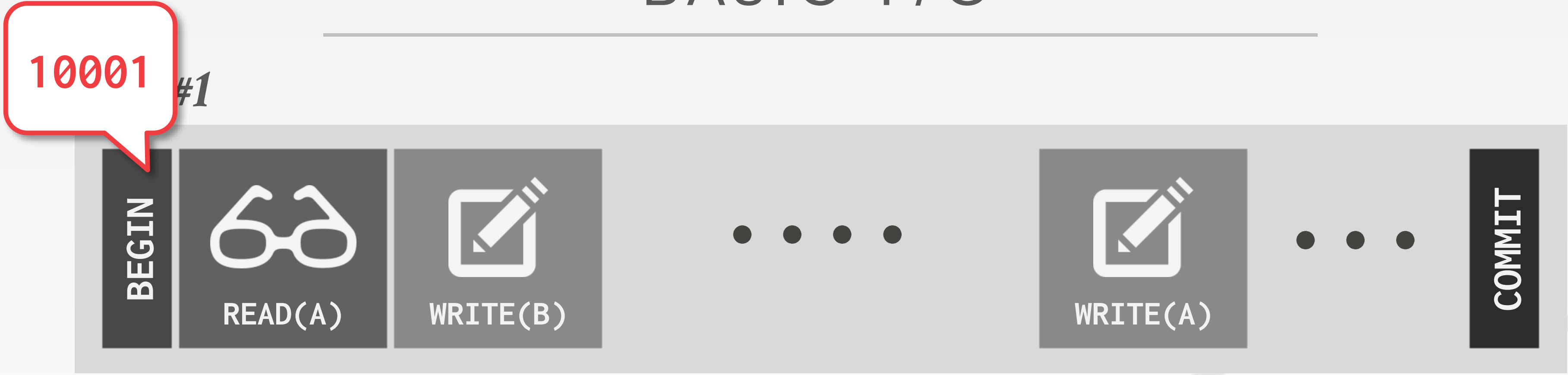
BASIC T/O



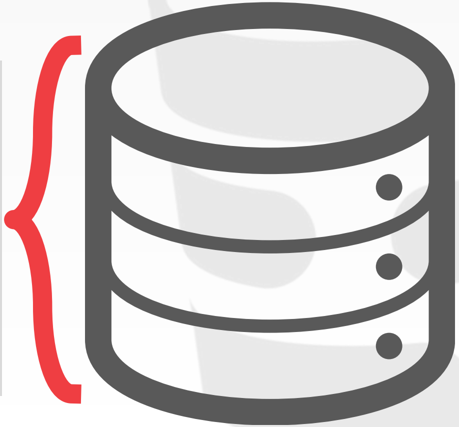
BASIC T/O



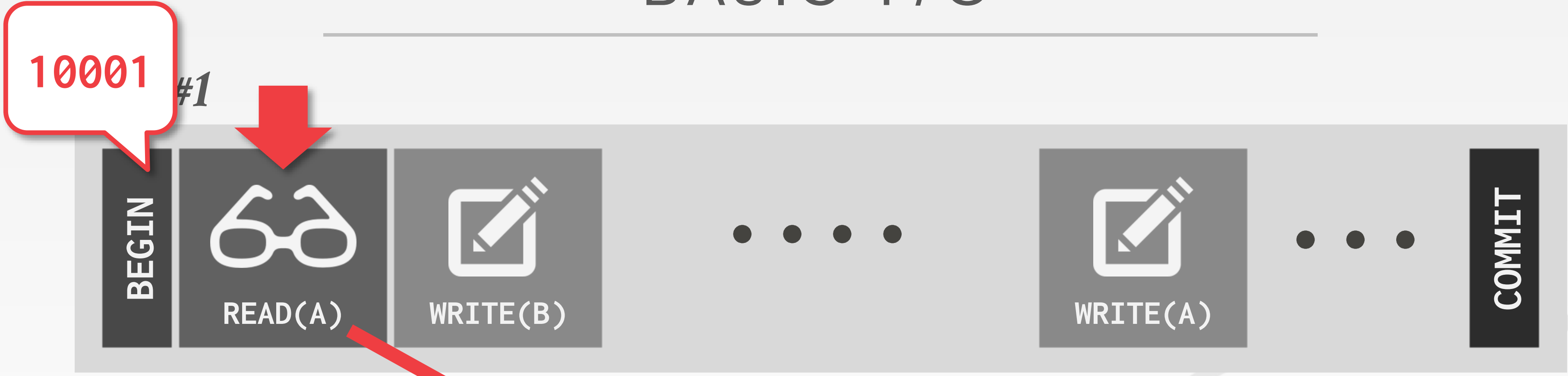
BASIC T/O



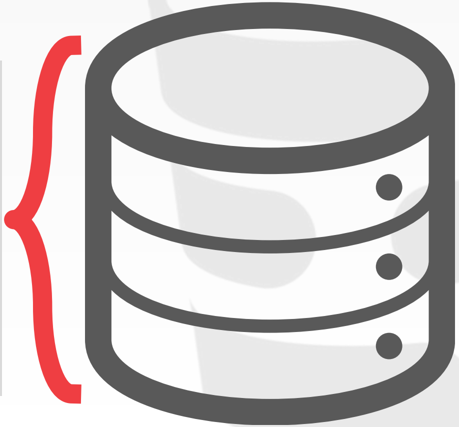
Record	Read Timestamp	Write Timestamp
A	10000	10000
B	10000	10000



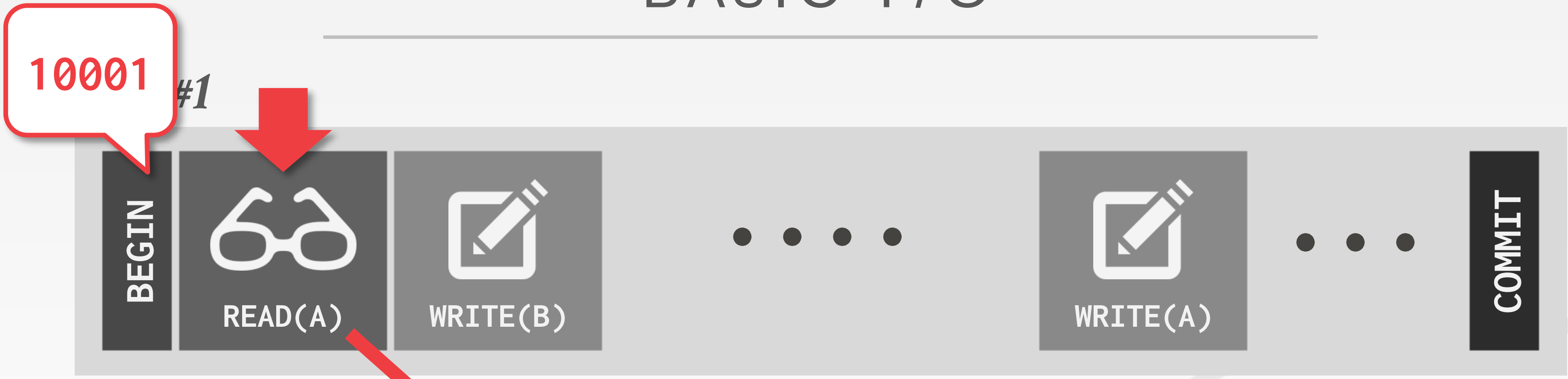
BASIC T/O



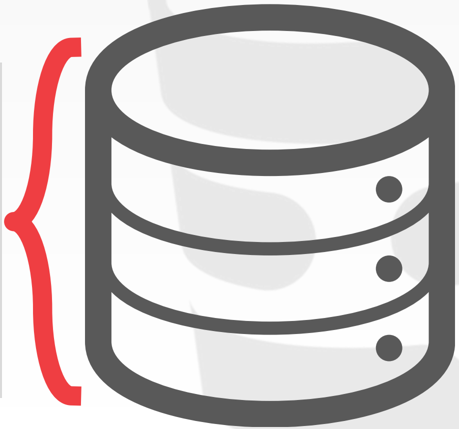
Record	Read Timestamp	Write Timestamp
A	10000	10000
B	10000	10000



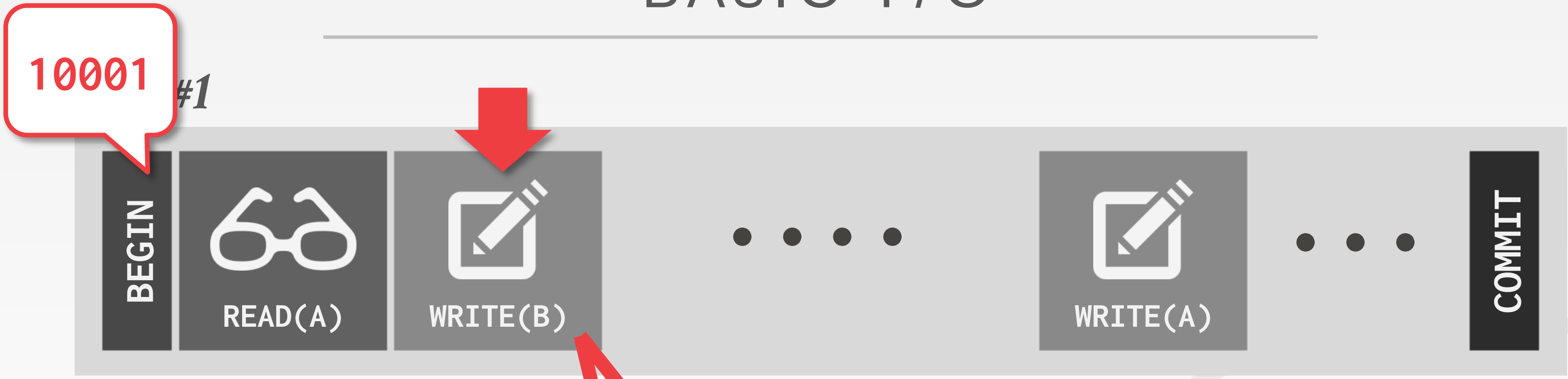
BASIC T/O



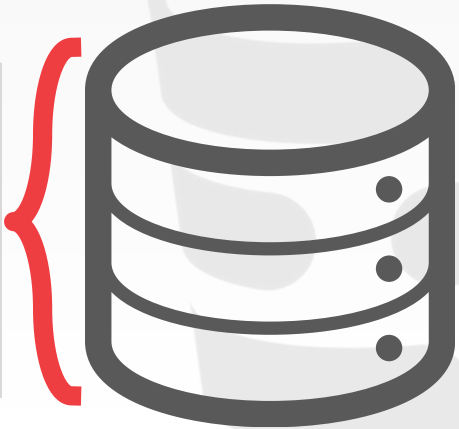
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10000



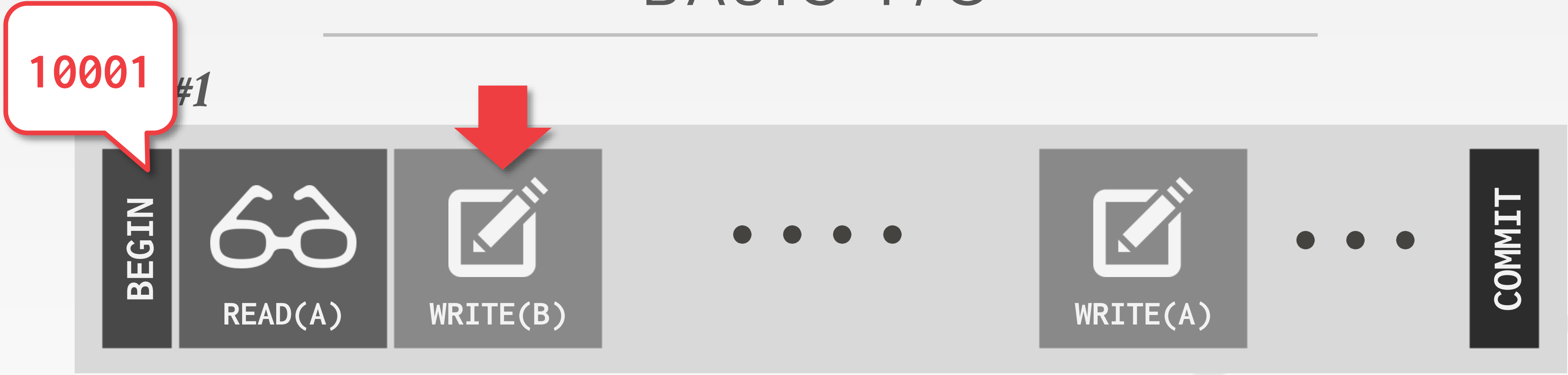
BASIC T/O



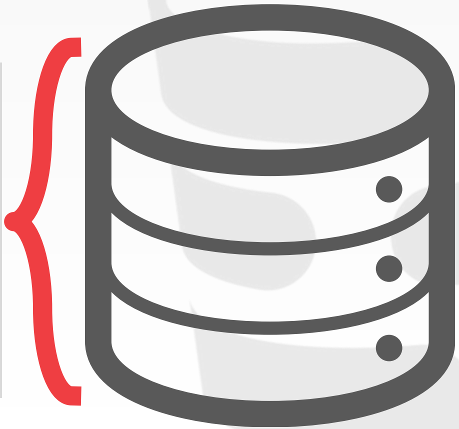
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10001



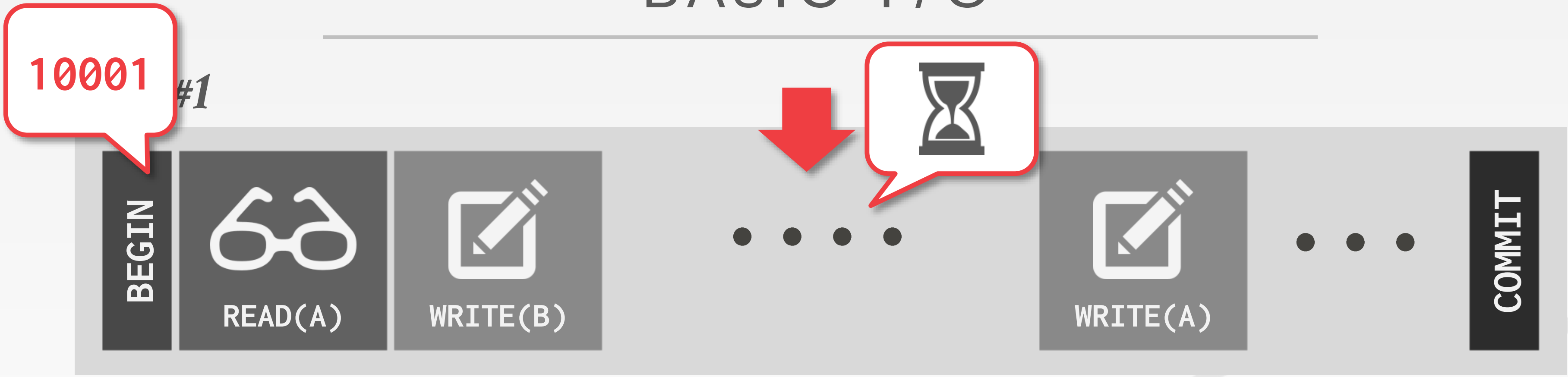
BASIC T/O



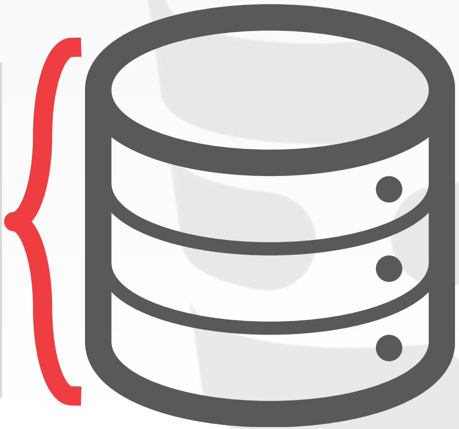
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10001



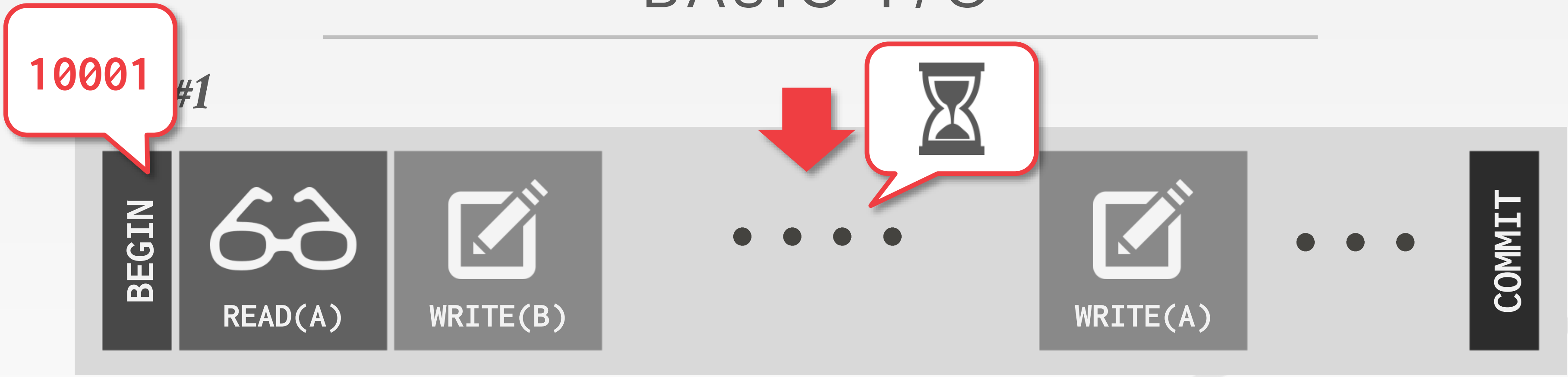
BASIC T/O



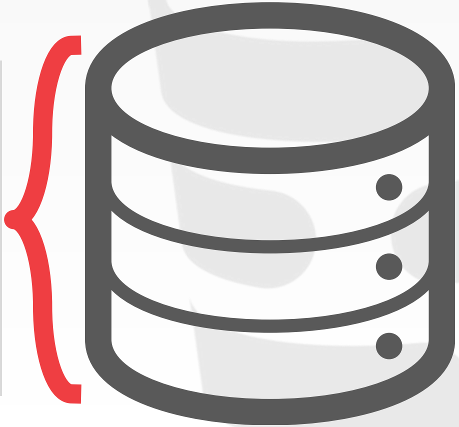
Record	Read Timestamp	Write Timestamp
A	10001	10000
B	10000	10001



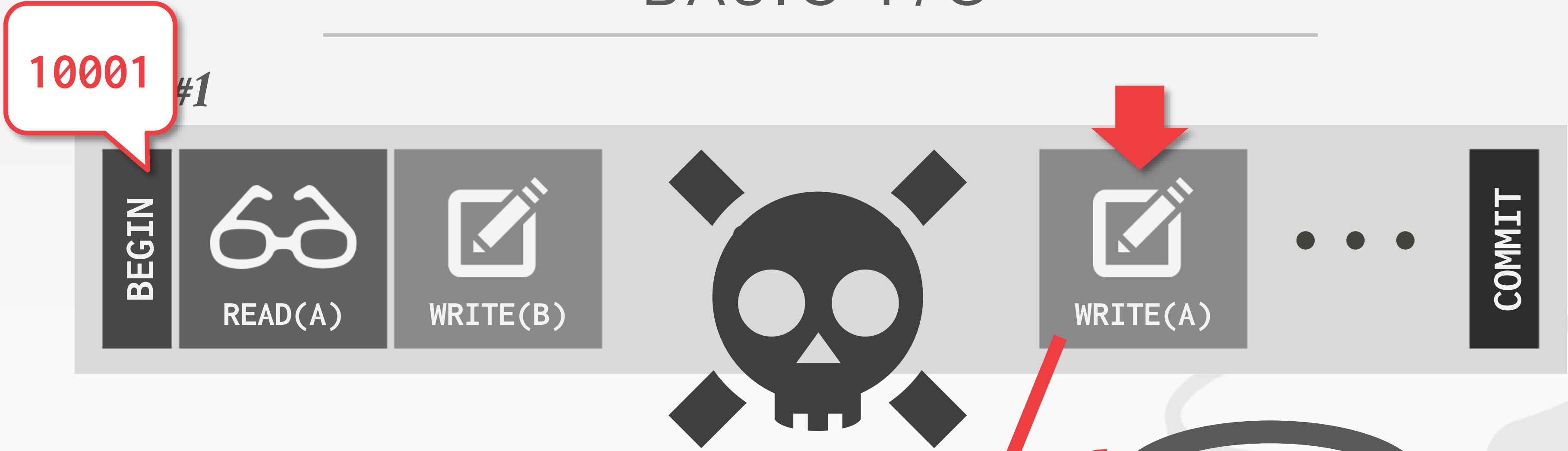
BASIC T/O



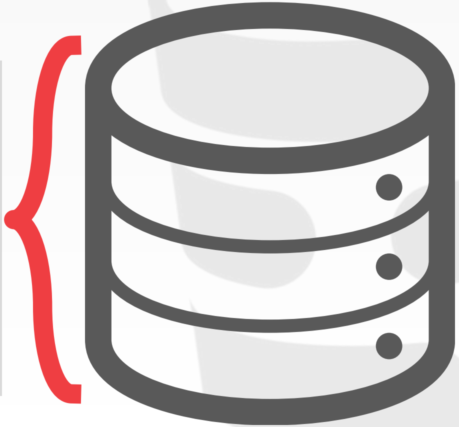
Record	Read Timestamp	Write Timestamp
A	10001	10005
B	10000	10001



BASIC T/O



Record	Read Timestamp	Write Timestamp
A	10001	10005
B	10000	10001



OPTIMISTIC CONCURRENCY CONTROL

Timestamp-ordering scheme where txns copy data read/write into a private workspace that is not visible to other active txns.

When a txn commits, the DBMS verifies that there are no conflicts.

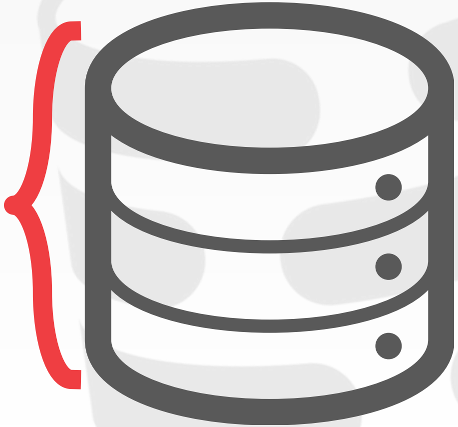
First proposed in 1981 at CMU by H.T. Kung.

ON OPTIMISTIC METHODS FOR CONCURRENCY CONTROL
ACM TRANSACTIONS ON DATABASE SYSTEMS 1981

OPTIMISTIC CONCURRENCY CONTROL

Txn #1

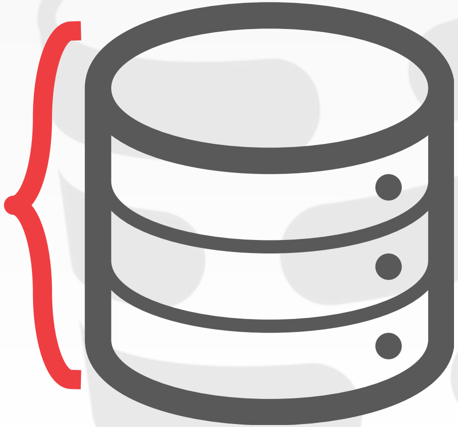
Record	Value	Write Timestamp
A	123	10000
B	456	10000



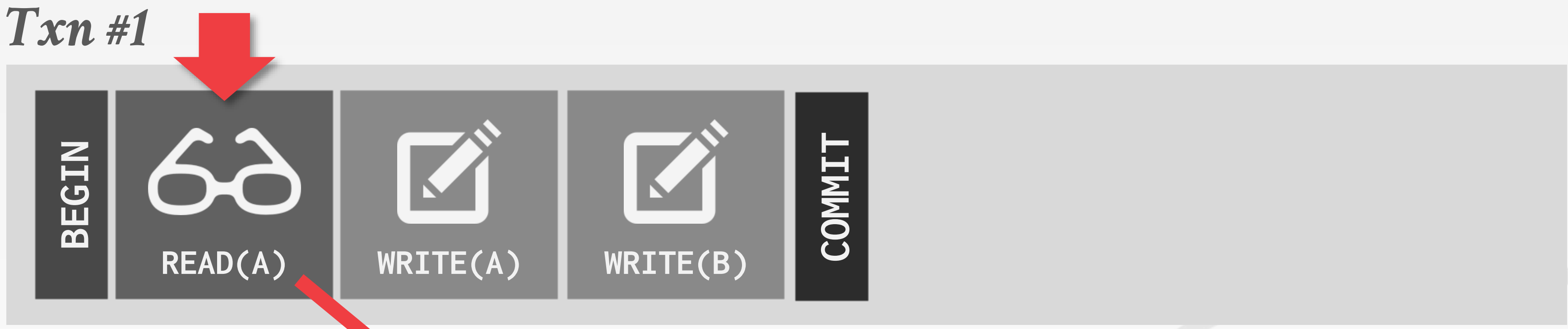
OPTIMISTIC CONCURRENCY CONTROL

Txn #1

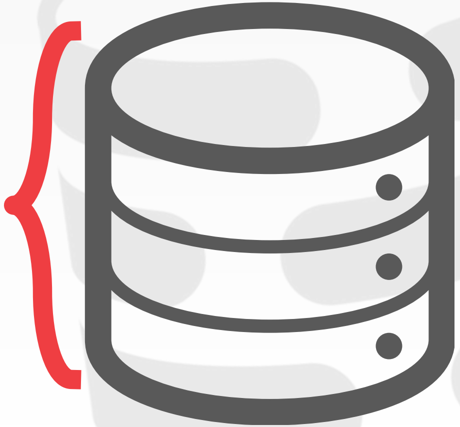
Record	Value	Write Timestamp
A	123	10000
B	456	10000



OPTIMISTIC CONCURRENCY CONTROL



Record	Value	Write Timestamp
A	123	10000
B	456	10000



OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN	READ(A)	WRITE(A)	WRITE(B)	COMMIT

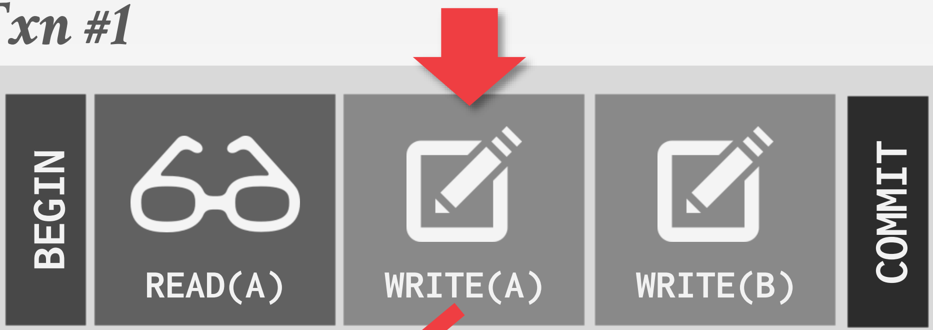
Workspace

Record	Value	Write Timestamp
A	123	10000

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

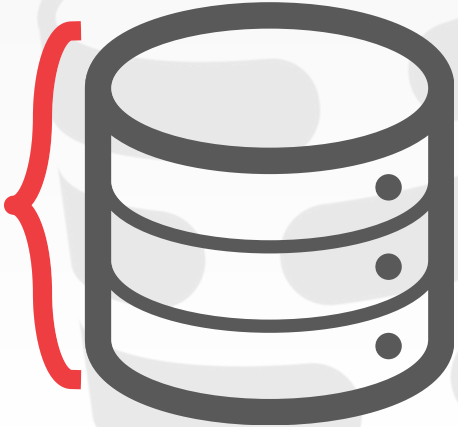
Txn #1



Workspace

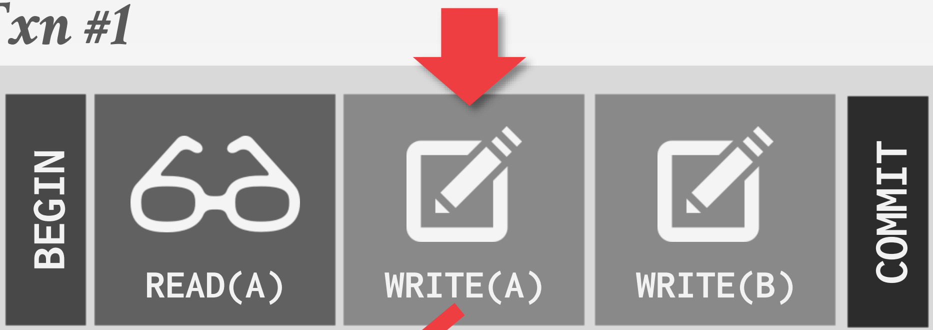
Record	Value	Write Timestamp
A	123	10000

Record	Value	Write Timestamp
A	123	10000
B	456	10000



OPTIMISTIC CONCURRENCY CONTROL

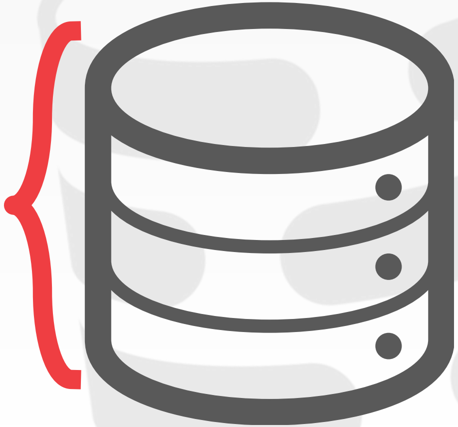
Txn #1



Workspace

Record	Value	Write Timestamp
A	888	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000



OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | COMMIT

Workspace

Record	Value	Write Timestamp
A	888	∞
B	456	10000

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | COMMIT

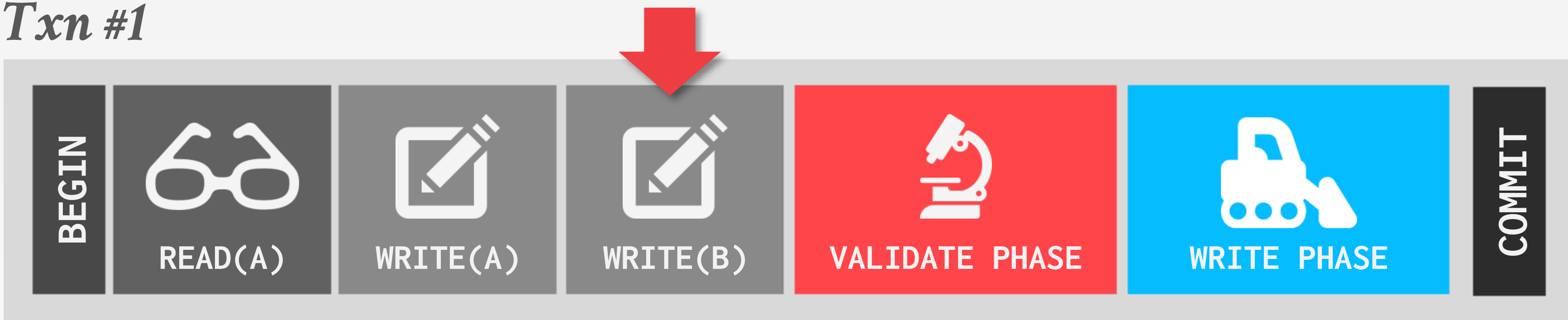
Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

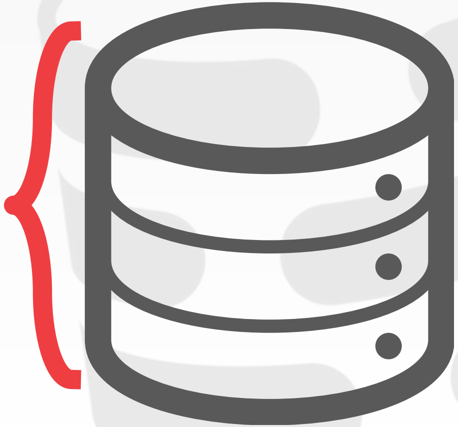
Txn #1



Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000



OPTIMISTIC CONCURRENCY CONTROL

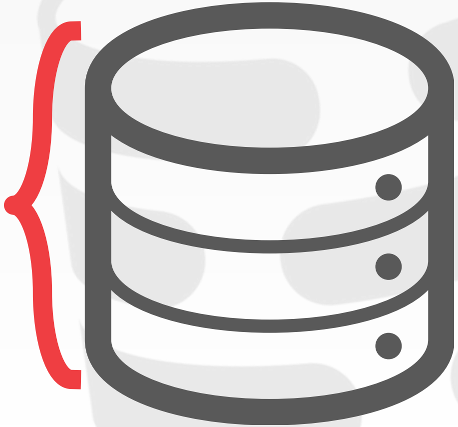
Txn #1

BEGIN | READ(A) | WRITE(A) | WRITE(B) | VALIDATE PHASE | WRITE PHASE | COMMIT

Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000



OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN	READ(A)	WRITE(A)	WRITE(B)	VALIDATE PHASE	WRITE PHASE	COMMIT

10001

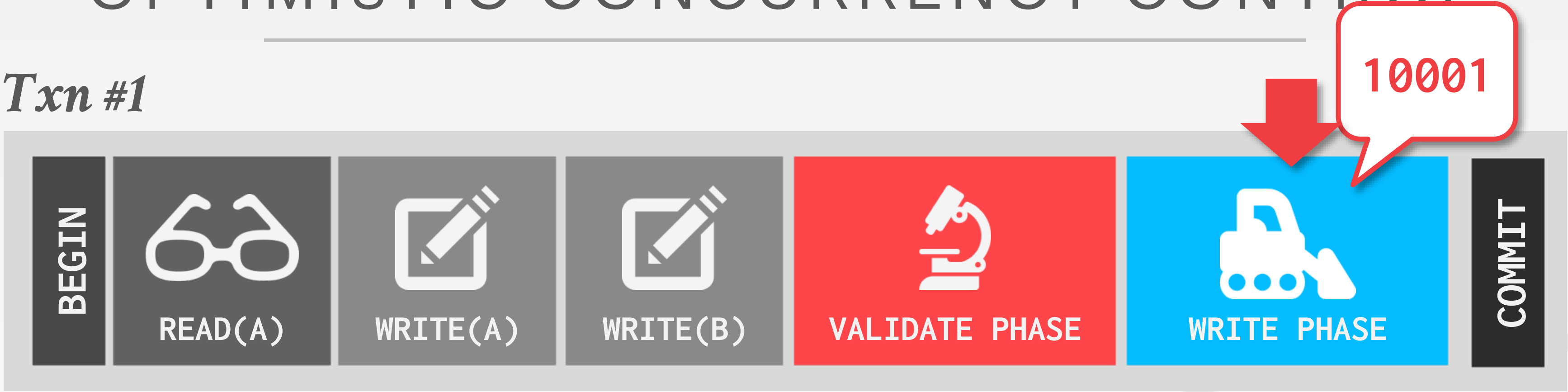
Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

OPTIMISTIC CONCURRENCY CONTROL

Txn #1



Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	888	10001
B	999	10001

OBSERVATION

When there is low contention, optimistic protocols perform better because the DBMS spends less time checking for conflicts.

At high contention, the both classes of protocols degenerate to essentially the same serial execution.

CONCURRENCY CONTROL EVALUATION

Compare in-memory concurrency control protocols at high levels of parallelism.

→ Single test-bed system.
→ Evaluate protocols using core counts beyond what is available on today's CPUs.

Running in extreme environments exposes what are the main bottlenecks in the DBMS.

STARING INTO THE ABYSS: AN EVALUATION OF CONCURRENCY CONTROL WITH ONE THOUSAND CORES
VLDB 2014

1000-CORE CPU SIMULATOR

DBx1000 Database System

→ In-memory DBMS with pluggable lock manager.
→ No network access, logging, or concurrent indexes.
→ All txns execute using stored procedures.

MIT Graphite CPU Simulator

→ Single-socket, tile-based CPU.
→ Shared L2 cache for groups of cores.
→ Tiles communicate over 2D-mesh network.

TARGET WORKLOAD

Yahoo! Cloud Serving Benchmark (YCSB)
→ 20 million tuples
→ Each tuple is 1KB (total database is ~20GB)

Each transactions reads/modifies 16 tuples.

Varying skew in transaction access patterns.

Serializable isolation level.

CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

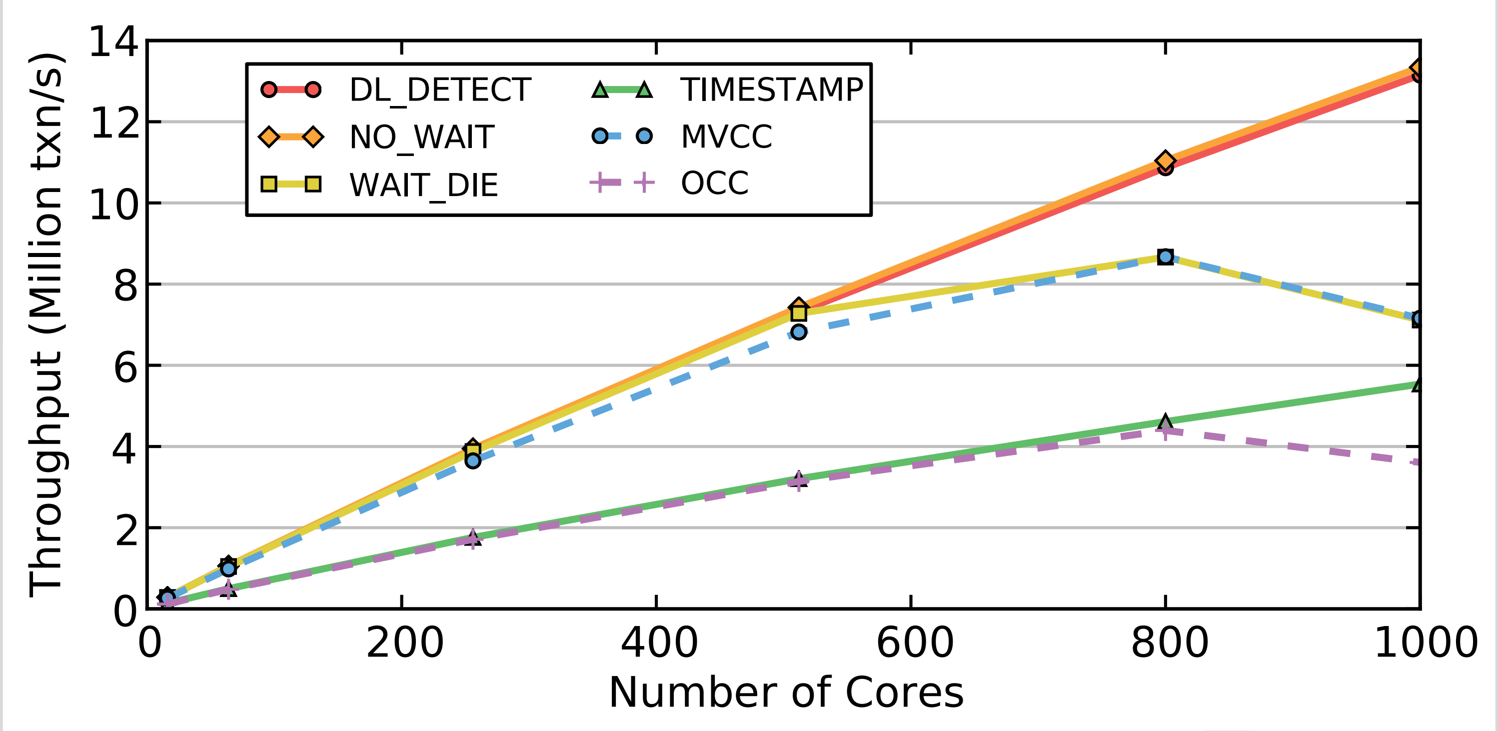
CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

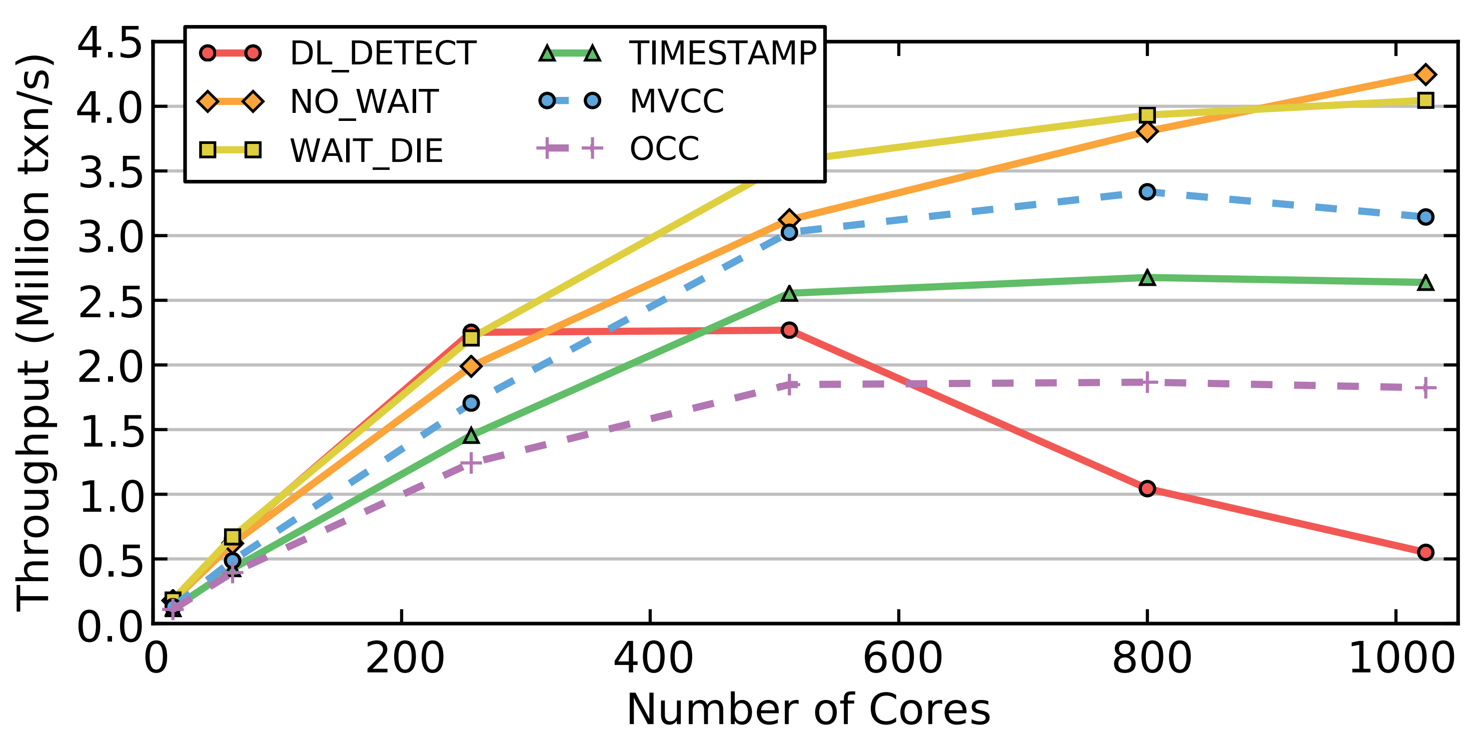
CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

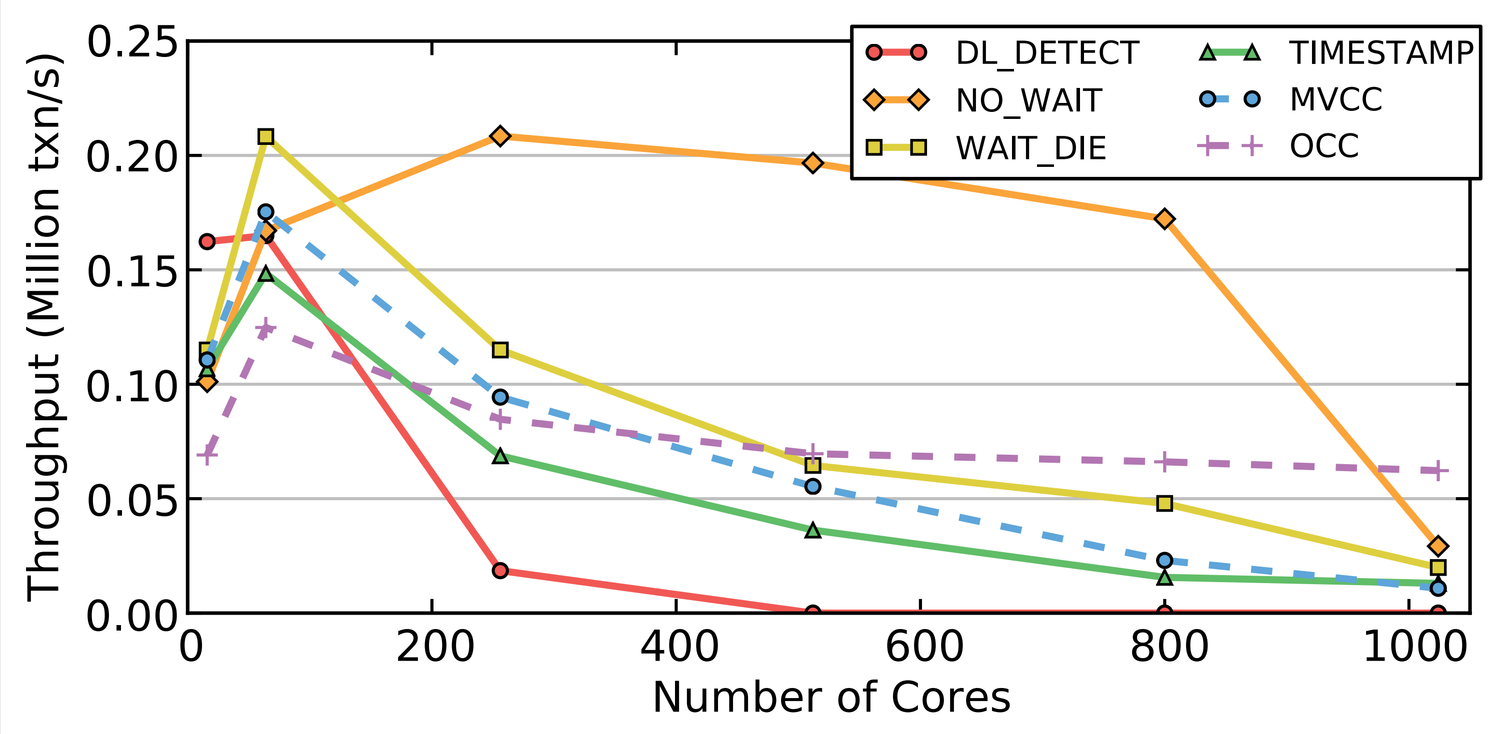
READ-ONLY WORKLOAD



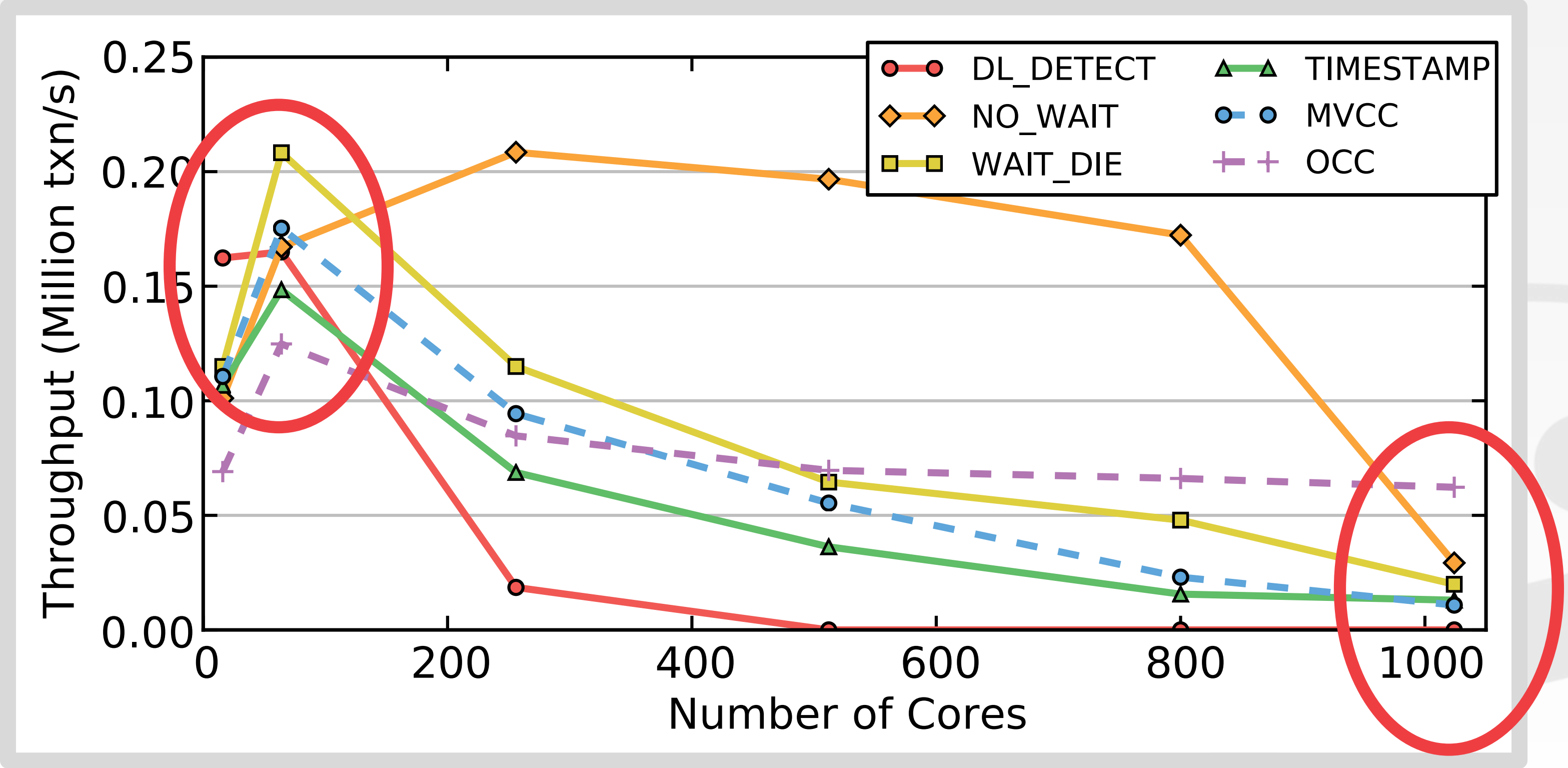
WRITE-INTENSIVE / MEDIUM-CONTENTION



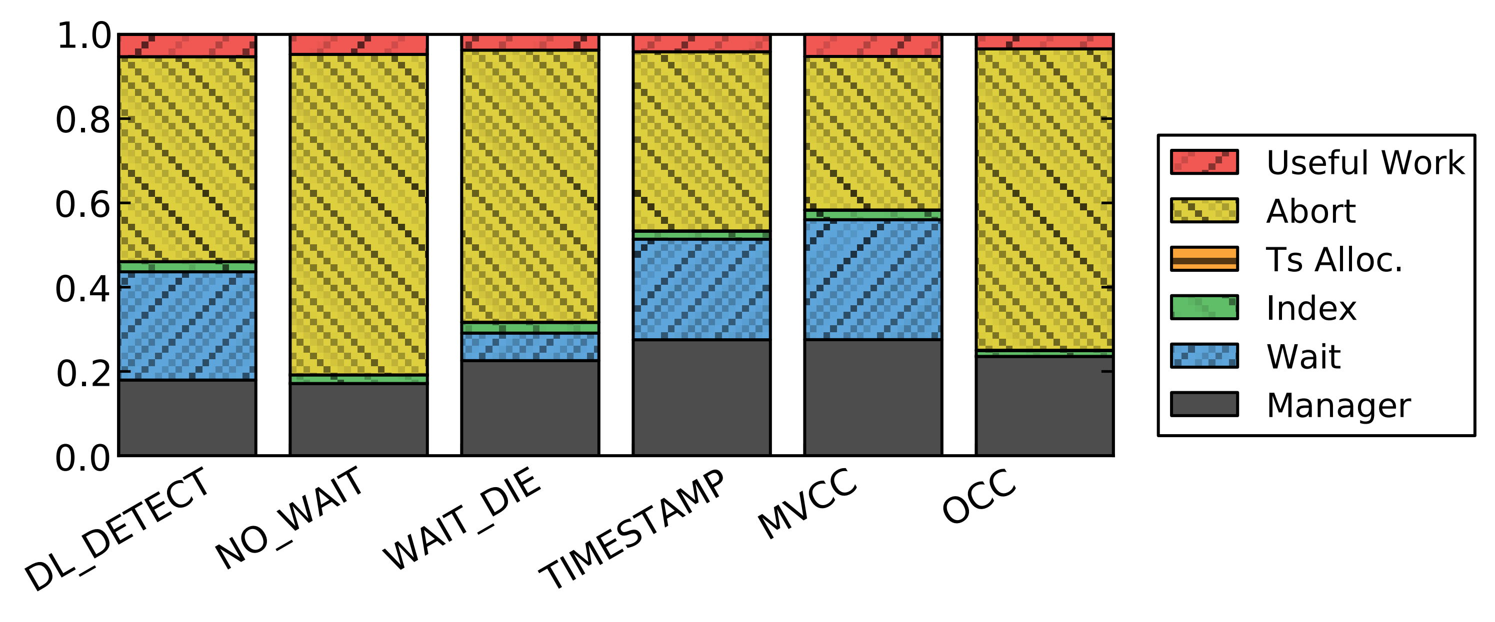
WRITE-INTENSIVE / HIGH-CONTENTION



WRITE-INTENSIVE / HIGH-CONTENTION



WRITE-INTENSIVE / HIGH-CONTENTION



BOTTLENECKS

Lock Thrashing

→ DL_DETECT, WAIT_DIE

Timestamp Allocation

→ All T/O algorithms + WAIT_DIE

Memory Allocations

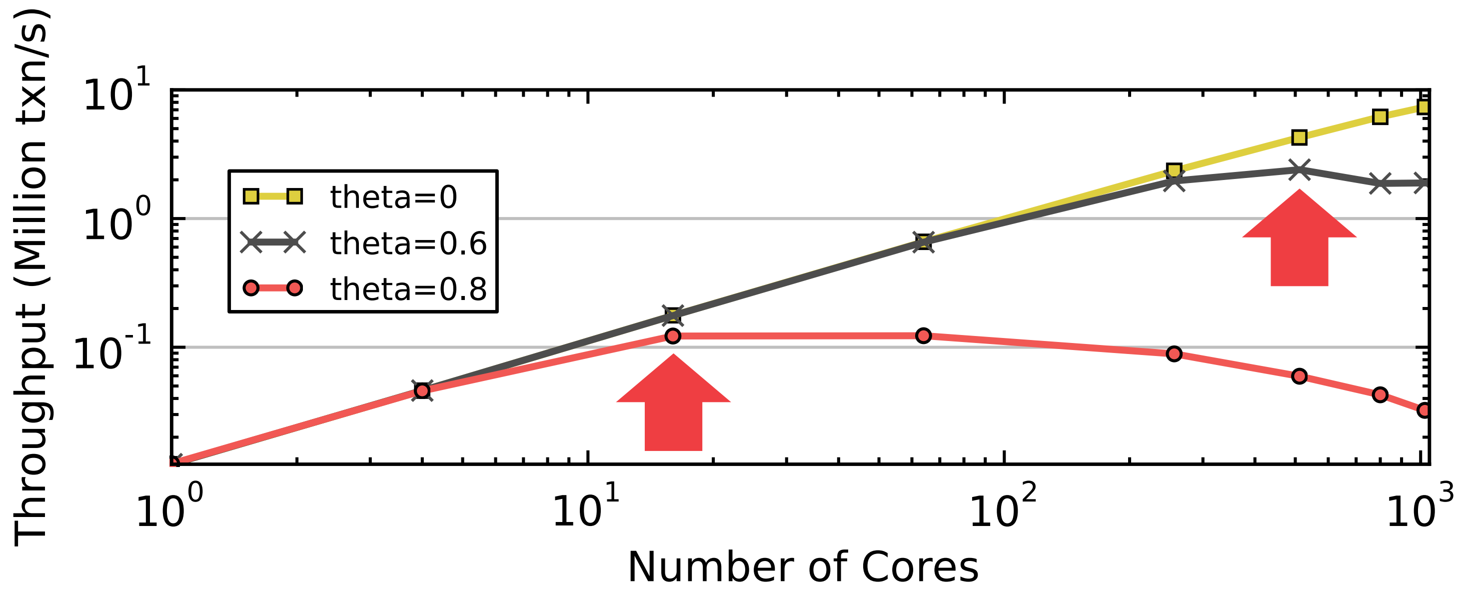
→ OCC + MVCC

LOCK THRASHING

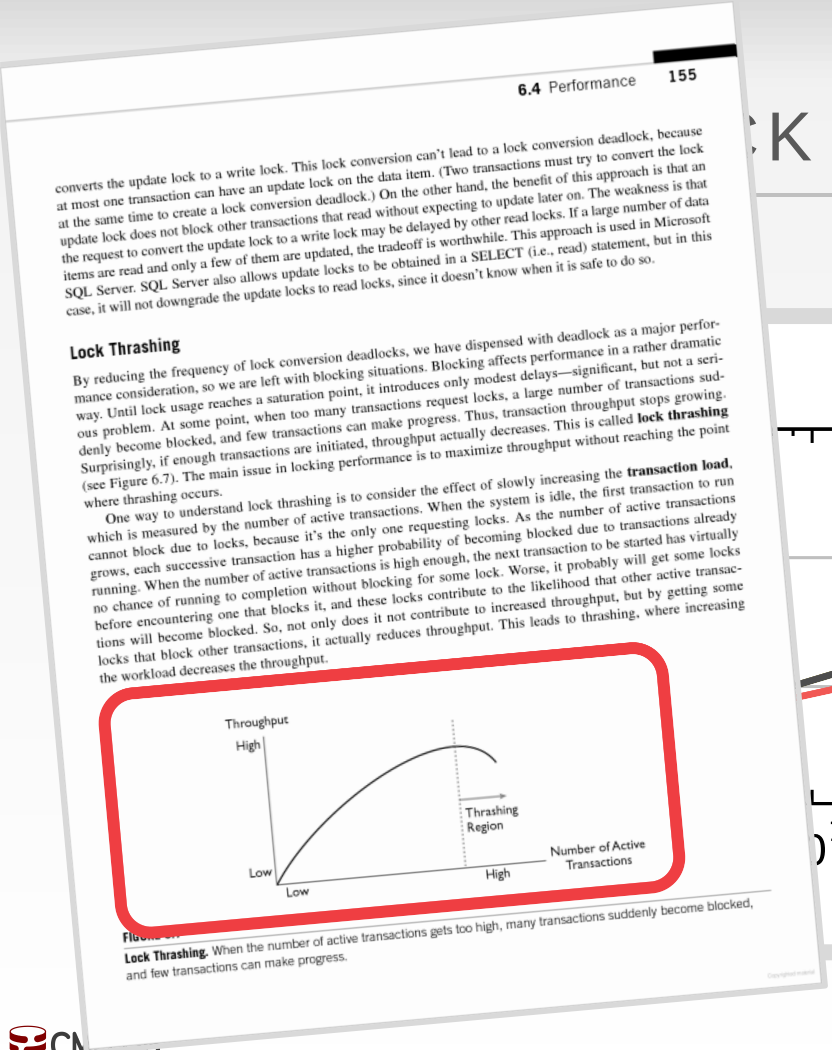
Each txn waits longer to acquire locks, causing other txn to wait longer to acquire locks.

Can measure this phenomenon by removing deadlock detection/prevention overhead.
→ Force txns to acquire locks in primary key order.
→ Deadlocks are not possible.

LOCK THRASHING



CK THRASHING



converts the update lock to a write lock. This lock conversion can't lead to a lock conversion deadlock, because at most one transaction can have an update lock on the data item. (Two transactions must try to convert the lock at the same time to create a lock conversion deadlock.) On the other hand, the benefit of this approach is that an update lock does not block other transactions that read without expecting to update later on. The weakness is that the request to convert the update lock to a write lock may be delayed by other read locks. If a large number of data items are read and only a few of them are updated, the tradeoff is worthwhile. This approach is used in Microsoft SQL Server. SQL Server also allows update locks to be obtained in a SELECT (i.e., read) statement, but in this case, it will not downgrade the update locks to read locks, since it doesn't know when it is safe to do so.

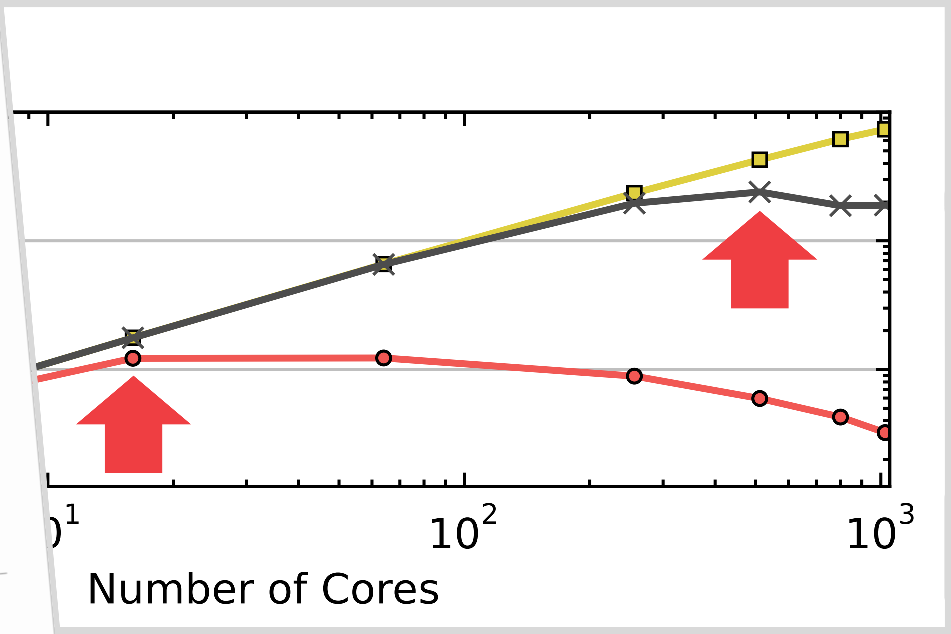
Lock Thrashing

By reducing the frequency of lock conversion deadlocks, we have dispensed with deadlock as a major performance consideration, so we are left with blocking situations. Blocking affects performance in a rather dramatic way. Until lock usage reaches a saturation point, it introduces only modest delays—significant, but not a serious problem. At some point, when too many transactions request locks, a large number of transactions suddenly become blocked, and few transactions can make progress. Thus, transaction throughput stops growing. Surprisingly, if enough transactions are initiated, throughput actually decreases. This is called **lock thrashing** (see Figure 6.7). The main issue in locking performance is to maximize throughput without reaching the point where thrashing occurs.

One way to understand lock thrashing is to consider the effect of slowly increasing the **transaction load**, which is measured by the number of active transactions. When the system is idle, the first transaction to run cannot block due to locks, because it's the only one requesting locks. As the number of active transactions grows, each successive transaction has a higher probability of becoming blocked due to transactions already running. When the number of active transactions is high enough, the next transaction to be started has virtually no chance of running to completion without blocking for some lock. Worse, it probably will get some locks before encountering one that blocks it, and these locks contribute to the likelihood that other active transactions will become blocked. So, not only does it not contribute to increased throughput, but by getting some locks that block other transactions, it actually reduces throughput. This leads to thrashing, where increasing the workload decreases the throughput.

Throughput
High
Low
Low
High
Thrashing Region
Number of Active Transactions

Lock Thrashing. When the number of active transactions gets too high, many transactions suddenly become blocked, and few transactions can make progress.



TIMESTAMP ALLOCATION

Mutex
→ Worst option.

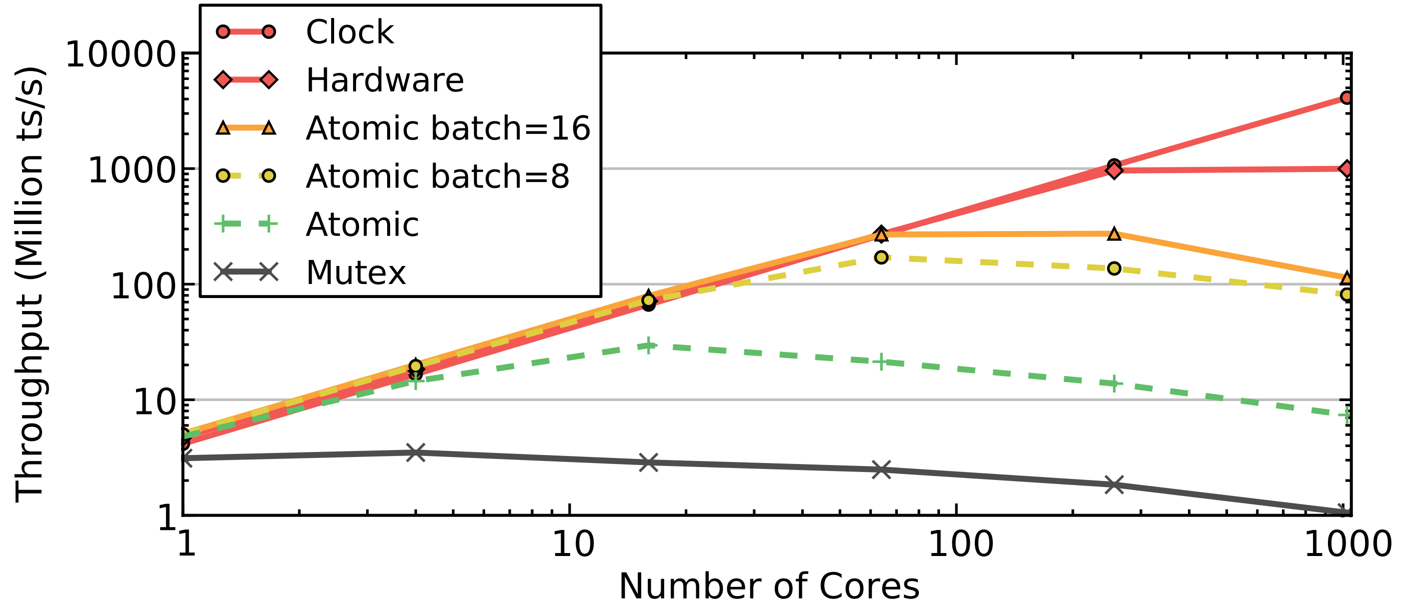
Atomic Addition
→ Requires cache invalidation on write.

Batched Atomic Addition
→ Needs a back-off mechanism to prevent fast burn.

Hardware Clock
→ Not sure if it will exist in future CPUs.

Hardware Counter
→ Not implemented in existing CPUs.

TIMESTAMP ALLOCATION



MEMORY ALLOCATIONS

Copying data on every read/write access slows down the DBMS because of contention on the memory controller.

→ In-place updates and non-copying reads are not affected as much.

Default libc `malloc` is slow. Never use it.

→ We will discuss this further later in the semester.

PARTING THOUGHTS

The design of a in-memory DBMS is significantly different than a disk-oriented system.

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

Increases in DRAM capacities have stalled in recent years compared to SSDs…

NEXT CLASS

Multi-Version Concurrency Control