Exercise Sheets

Lecture 16: Concurrency Control in Main-Memory DBMSs

Problem sets

Recap

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.

Videos

Text Documents

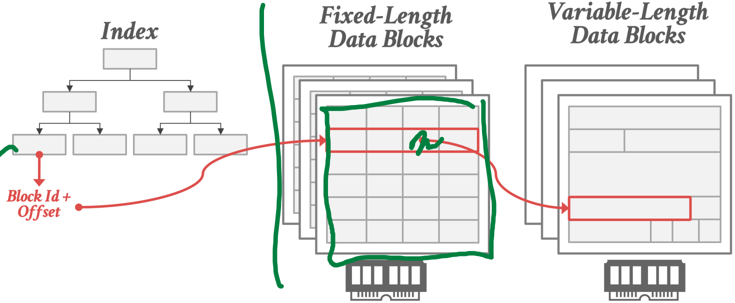
Today's Agenda

- Motivation
- Concurrency Control Schemes
- Concurrency Control Evaluation

In-memory Data Organization

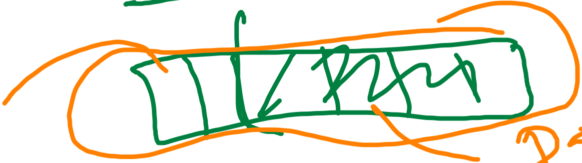
- An in-memory DBMS does not need to store the database in slotted pages but it will still organize tuples in blocks/pages:
  - Direct memory pointers vs. record ids
  - Fixed-length vs. variable-length data pools
  - Use checksums to detect software errors from trashing the database.

In-memory Data Organization



Concurrency Control

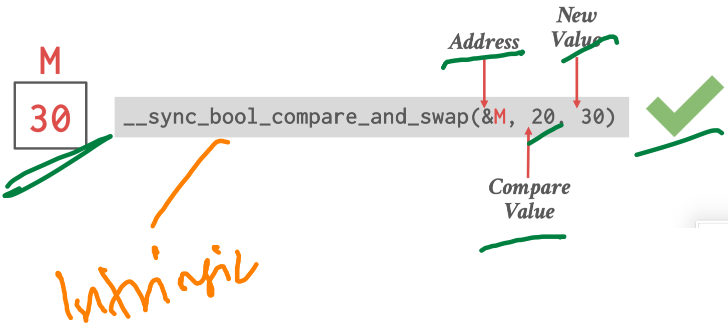
- For in-memory DBMSs, the cost of a txn acquiring a lock is the same as accessing data.
- New bottleneck is contention caused from txns trying access data at the same time.
- The DBMS can store locking information about each tuple together with its data.
  - This helps with CPU cache locality.
  - Mutexes are too slow. Need to use compare-and-swap (CAS) instructions.



Compare-and-Swap

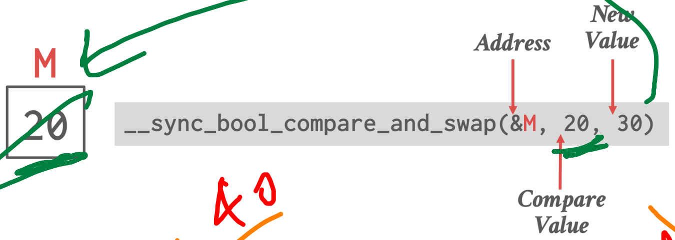
std::atomic

- 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



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



Concurrency Control Schemes

Concurrency Control Schemes

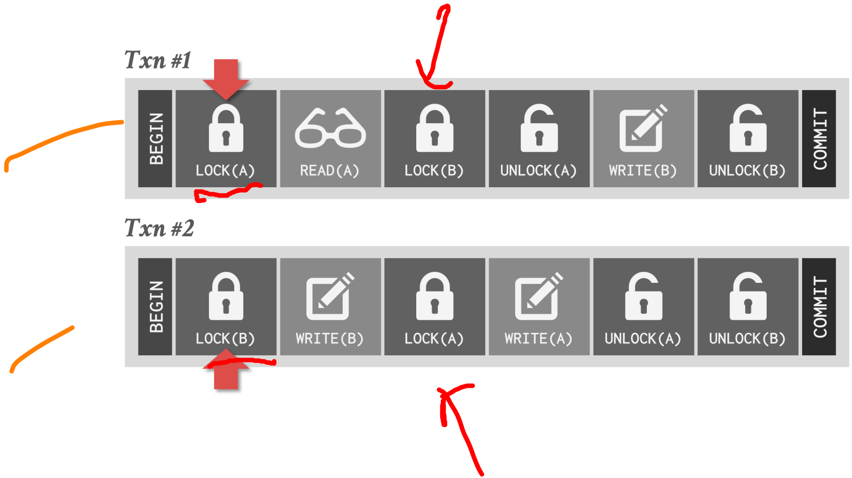
Pessimistic

- 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.

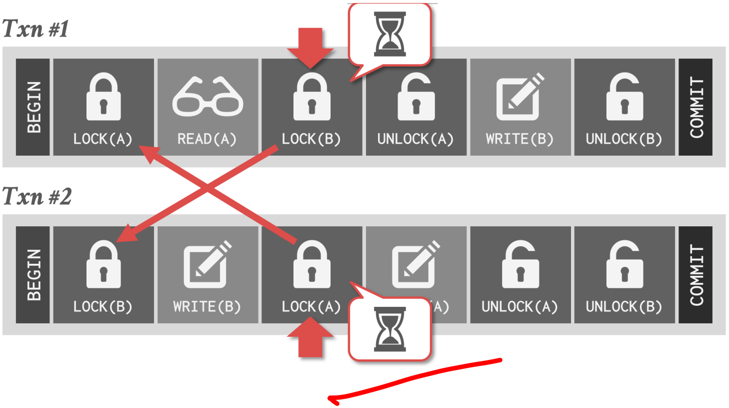
Optimistic

Two-Phase Locking

Two-Phase Locking



Two-Phase Locking



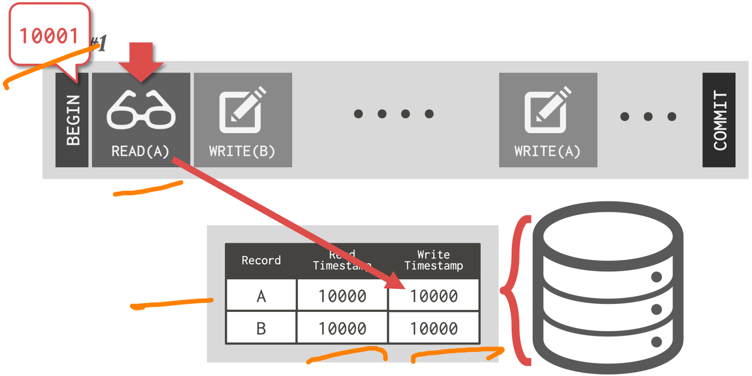
Two-Phase Locking

- **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.

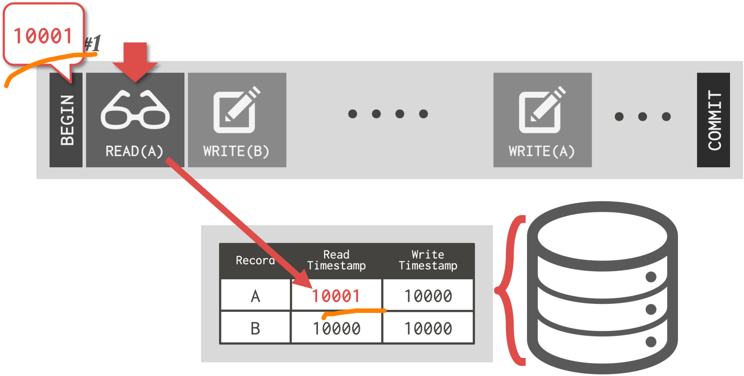
Timestamp Ordering

- **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.

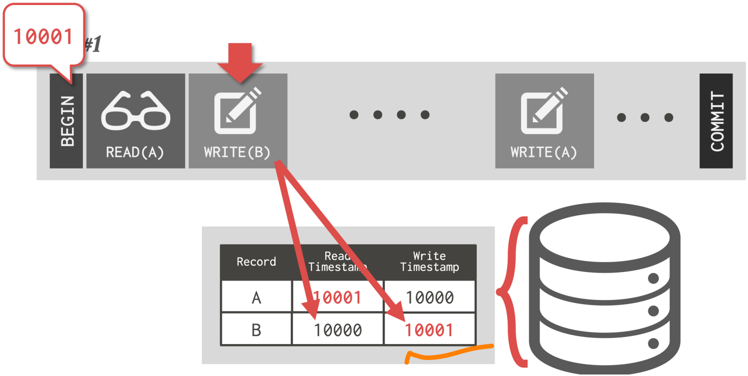
Basic T/O


10001
#1
BEGIN
READ(A)
WRITE(B)
WRITE(A)
COMMIT
Record
Read Timestamp
Write Timestamp
A
10000
10000
B
10000
10000

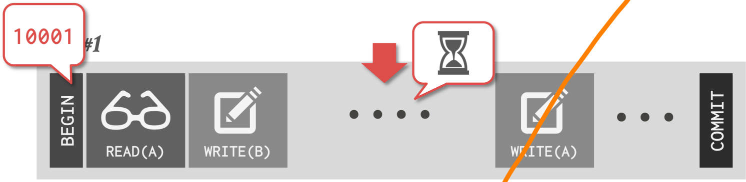
Basic T/O


10001
#1
BEGIN
READ(A)
WRITE(B)
WRITE(A)
COMMIT
Record
Read Timestamp
Write Timestamp
A
10001
10000
B
10000
10000

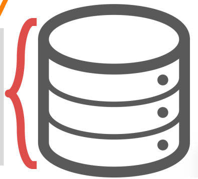
Basic T/O


10001
#1
BEGIN
READ(A)
WRITE(B)
WRITE(A)
COMMIT
Record
Read Timestamp
Write Timestamp
A
10001
10000
B
10000
10001

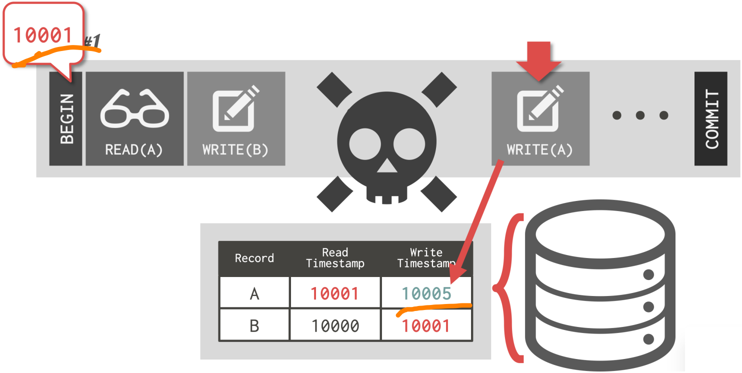
Basic T/O



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

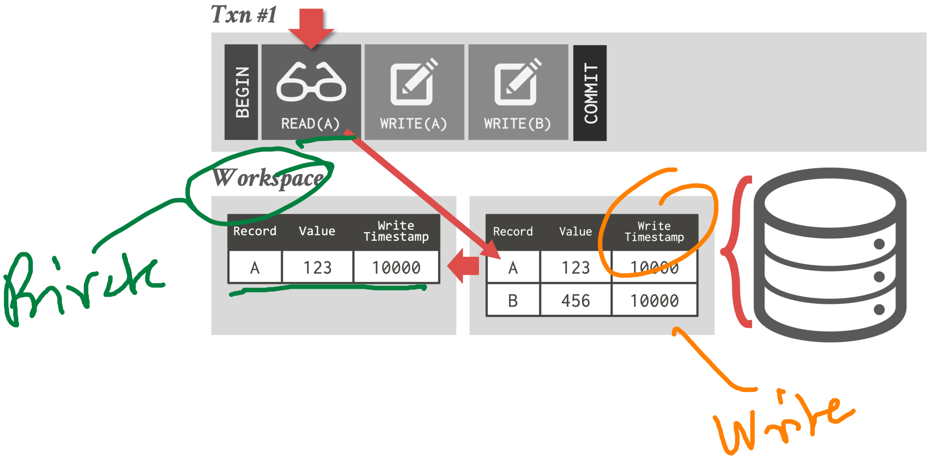
Optimistic Concurrency Control

- 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.

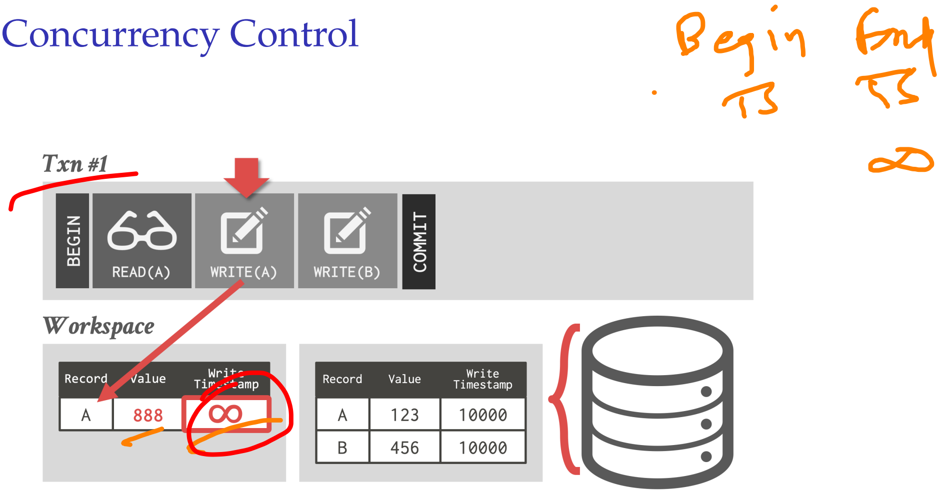
Optimistic Concurrency Control

Record	Value	Write Timestamp
A	123	10000
B	456	10000

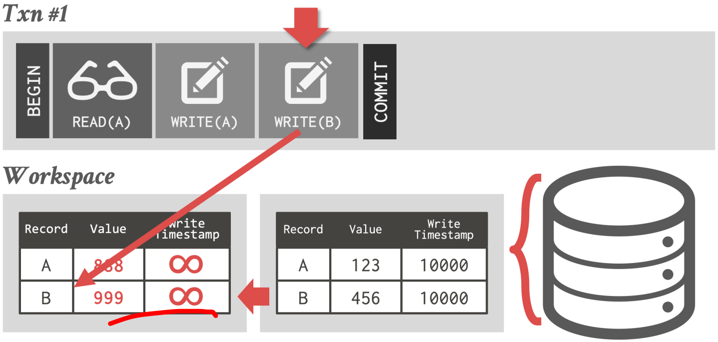
Optimistic Concurrency Control



Optimistic Concurrency Control

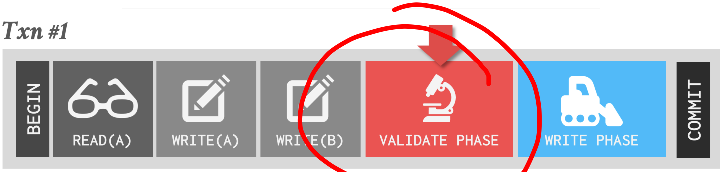


Optimistic Concurrency Control



Optimistic Concurrency Control

Txn #1

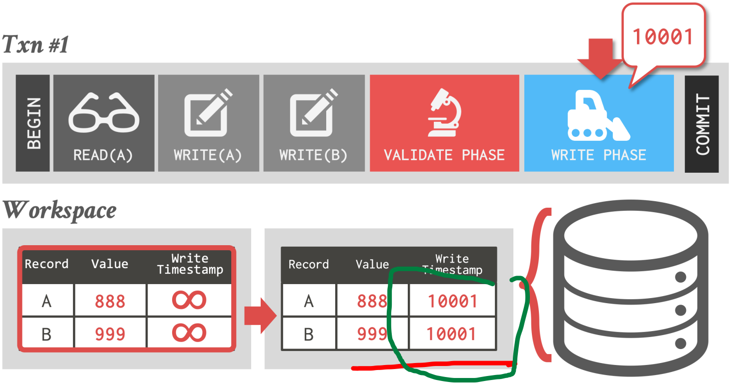


Workspace

Record	Value	Write Timestamp
A	888	∞
B	999	∞

Record	Value	Write Timestamp
A	123	10000
B	456	10000

Optimistic Concurrency Control



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

Concurrency Control Evaluation

1000 CPU cores

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

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.
  - NUCA (non-uniform cache access) architecture.

Target Workload

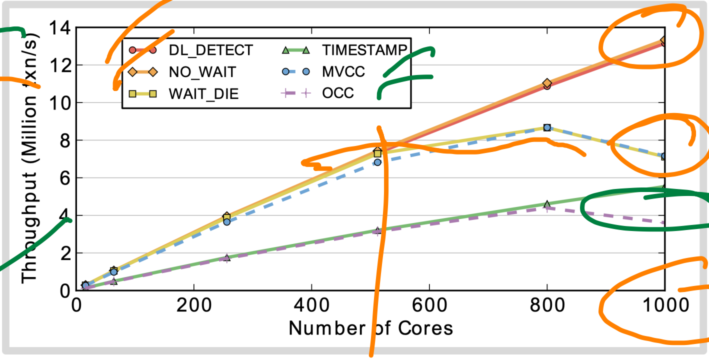
- Yahoo! Cloud Serving Benchmark (YCSB)
  - 20 million tuples
  - Each tuple is 1KB (total database is 20GB)
- Each transactions reads/modifies 16 tuples.
- Varying skew in transaction access patterns.
- Serializable isolation level.

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

Read-Only Workload

12 M

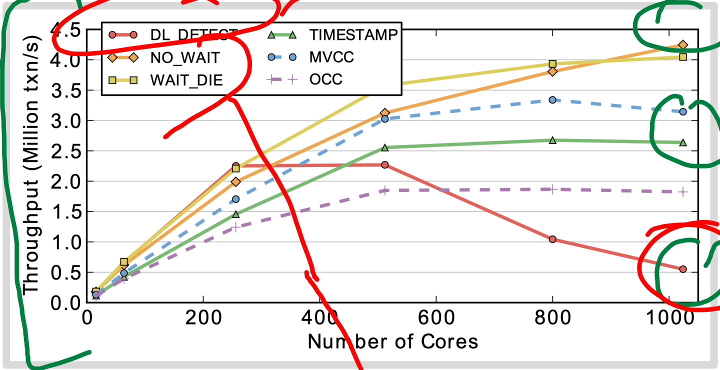


512
work

Read-Only Workload

- DL – DETECT / NO – WAIT – No overhead. No extra work. Everybody can acquire the shared locks on tuples.
- WAIT – DIE / MVCC – Timestamp allocation bottleneck.
- OCC / TIMESTAMP – Overhead of copying read tuples for repeatable reads.

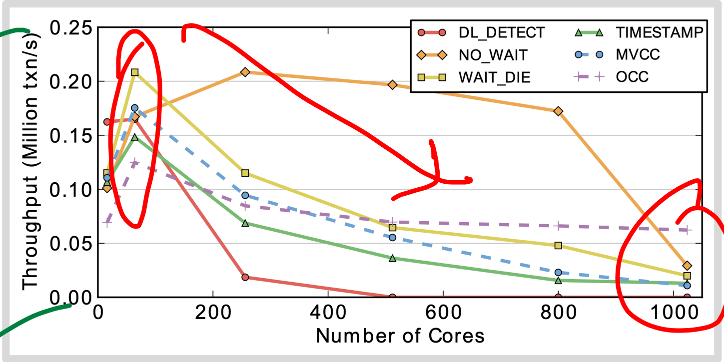
Write-Intensive / Medium-Contention



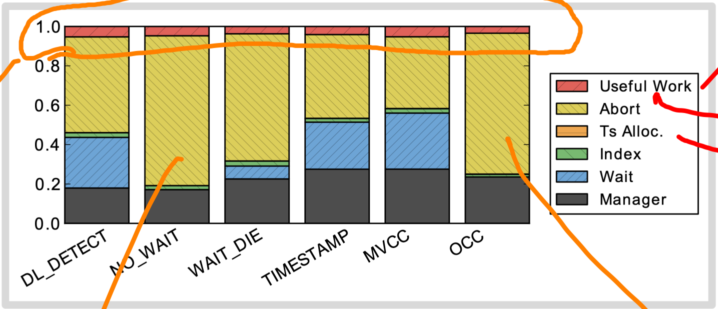
Write-Intensive / Medium-Contention

- 60% of txns are accessing 20% of the database.
- DL – DETECT – The worst because more conflicts. Spend more time trying to find deadlocks. Longer stalls.
- NO – *WAIT*/ *WAIT* – DIE – The best because they are simple. Cost of restarting txns in DBx1000 is cheap.
- OCC / TIMESTAMP – These protocols are roughly all the same because of copying.

Write-Intensive / High-Contention



Write-Intensive / High-Contention



Write-Intensive / High-Contention

- 90% of txns are accessing 10% of the database.
- All protocols flat-lined and converge to zero at 1000 cores. At high-contention, they all perform the same.
- NO – *WAIT* does the best. Only executing 200k txn/sec which is not a lot compared to the previous graphs. Lots of restarts.

Bottlenecks

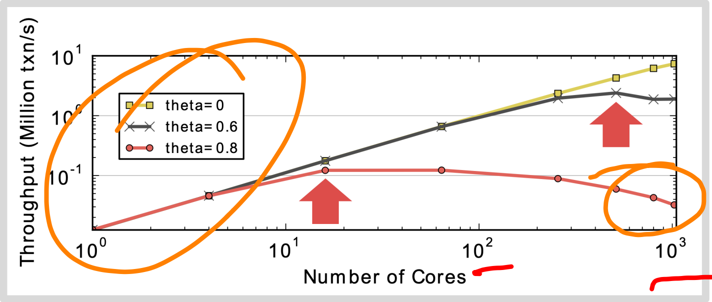
- **Lock Thrashing**
  - DL – DETECT, *WAIT* – DIE
- **Timestamp Allocation**
  - All T/O algorithms + *WAIT* – DIE
- **Memory Allocations**
  - OCC + MVCC

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.

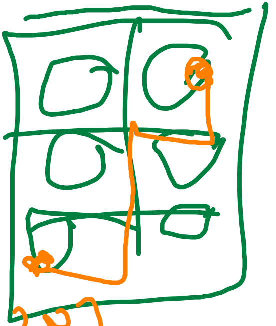
Lock Thrashing

distributed system

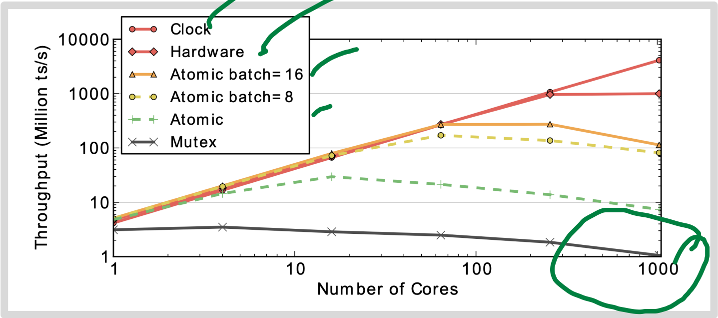


Timestamp Allocation

- **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



CAS

std::mutex

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.

Conclusion

Parting Thoughts

- The design of an 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