In-Memory Databases
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.
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

- **Index**
- **Buffer Pool**
  - page6
  - page2
  - page4
- **Database (On-Disk)**
  - page0
  - page1
  - page2

Page Id + Slot #
DISK-ORIENTED DATA ORGANIZATION

Index

Buffer Pool

Database (On-Disk)

Page Table

Page Id + Slot #
DISK-ORIENTED DATA ORGANIZATION

Index

Buffer Pool

Database (On-Disk)

Page Table

Page Id + Slot #
DISK-ORIENTED DATA ORGANIZATION

**Index**

**Buffer Pool**

**Database (On-Disk)**

**Page Table**

Page Id + Slot #
DISK-ORIENTED DATA ORGANIZATION

Index

Buffer Pool

Database (On-Disk)

Page Table

Page Id + Slot #
DISK-ORIENTED DATA ORGANIZATION

Index

Buffer Pool

Database (On-Disk)

Page Table

Page Id + Slot #
DISK-ORIENTED DATA ORGANIZATION

Index

Buffer Pool

Database (On-Disk)

Page Table

Page Id + Slot #

page6
page1
page4

page0
page1
page2
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.
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

- BUFFER POOL: 16%
- LATCHING: 14%
- LOCKING: 12%
- LOGGING: 34%
- B-TREE KEYS: 16%
- REAL WORK: 7%
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

Index

Fixed-Length Data Blocks

Variable-Length Data Blocks

Block Id + Offset
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.
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:

→ 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 Atomicity + Isolation in ACID
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

__sync_bool_compare_and_swap(&$M$, 20, 30)
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

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

Growing Phase

Shrinking Phase

Txn #1

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

Growing Phase

Shrinking Phase
**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

15-721 (Spring 2020)
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**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>UNLOCK(A)</th>
<th>WRITE(B)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

**Txn #2**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>LOCK(A)</th>
<th>WRITE(A)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

15-721 (Spring 2020)
### TWO-PHASE LOCKING

#### Txn #1

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>UNLOCK(A)</th>
<th>WRITE(B)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

#### Txn #2

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>LOCK(A)</th>
<th>UNLOCK(A)</th>
<th>WRITE(B)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

The transactions are executed as follows:

**Txn #1**
- **BEGIN**: Locks record A.
- **READ(A)**: Reads record A.
- **LOCK(B)**: Locks record B.
- **UNLOCK(A)**: Unlocks record A.
- **UNLOCK(B)**: Unlocks record B.
- **WRITE(B)**: Writes record B.

**Txn #2**
- **BEGIN**: Locks record B.
- **WRITE(B)**: Writes record B.
- **LOCK(A)**: Locks record A.
- **UNLOCK(A)**: Unlocks record A.
- **UNLOCK(B)**: Unlocks record B.
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)
  - UNLOCK(A)
  - WRITE(B)
  - UNLOCK(B)
  - COMMIT
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

BEGIN
READ(A)
WRITE(B)

WRITE(A)

COMMIT
BASIC T/O

#1

BEGIN
READ(A)
WRITE(B)

.......

WRITE(A)

COMMIT
BASIC T/O

10001

#1

BEGIN
READ(A)
WRITE(B)

WRITE(A)

COMMIT
#1

**BASIC T/O**

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>

**Timestamps:***

- **10000**
- **10001**
BASIC T/O

10001

BEGIN
READ(A)
WRITE(B)

Record | Read Timestamp | Write Timestamp
--- | --- | ---
A | 10000 | 10000
B | 10000 | 10000

COMMIT
### BASIC T/O

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Step 1:** Begin transaction
- **Step 2:** Read record A
- **Step 3:** Write record B
- **Step 4:** Write record A
- **Step 5:** Commit transaction
BASIC T/O

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>
# BASIC T/O

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>

**Transaction #1**

BEGIN

READ(A)

WRITE(B)

COMMIT

Txn #1

BEGIN

READ(A)

WRITE(B)

WRITE(A)

COMMIT
# Basic T/O

**Record** | **Read Timestamp** | **Write Timestamp**
--- | --- | ---
A | 10001 | 10000
B | 10000 | 10001
BASIC T/O

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10005</td>
<td>10001</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>
BASIC T/O

BEGIN
READ(A)
WRITE(B)

COMMIT

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10005</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>

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.
OPTIMISTIC CONCURRENCY CONTROL

**Txn #1**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>READ(A)</th>
<th>WRITE(A)</th>
<th>WRITE(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

Record | Value | Write Timestamp |
-------|-------|-----------------|
A      | 123   | 10000           |
B      | 456   | 10000           |
**Optimistic Concurrency Control**

*Txn #1*

**Read Phase**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**Optimistic Concurrency Control**

**Txn #1**

- **BEGIN**: READ(A)
- **WRITE(A)**
- **WRITE(B)**
- **COMMIT**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

**Txn #1**

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

COMMIT

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

Txn #1

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

Workspace

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**Transaction #1**

**Workspace**

- **Record** | **Value** | **Write Timestamp**
  - A | 888 | ∞

**Record** | **Value** | **Write Timestamp**
- A | 123 | 10000
- B | 456 | 10000

**Optimistic Concurrency Control**
**OPTIMISTIC CONCURRENCY CONTROL**

*Txn #1*

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**OPTIMISTIC CONCURRENCY CONTROL**

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

**Txn #1**

BEGIN

READ(A)

WRITE(A)

WRITE(B)

COMMIT
OPTIMISTIC CONCURRENCY CONTROL

**Txn #1**

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

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**OPTIMISTIC CONCURRENcy CONTROL**

**Txn #1**

- **BEGIN**
  - READ(A)
  - WRITE(A)
  - WRITE(B)

- **VALIDATE PHASE**
  - WRITE PHASE

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

Txn #1

Workspace

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

10001
# Optimistic Concurrency Control

## Txn #1

### Workspace

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

### Logs

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

Commit timestamp: 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.
**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

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
### CONCURRENCY CONTROL SCHEMES

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
</tbody>
</table>

- TIMESTAMP
- MVCC
- OCC

*Basic T/O Algorithm*

*Multi-Version T/O*

*Optimistic Concurrency Control*
### CONCURRENcy CONTROL SCHEMES

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
READ-ONLY WORKLOAD

The graph shows the throughput (Million txn/s) as a function of the number of cores for different workloads:

- **DL_DETECT**
- **TIMESTAMP**
- **NO_WAIT**
- **MVCC**
- **WAIT_DIE**
- **OCC**

As the number of cores increases, the throughput for all workloads also increases. The **DL_DETECT** workload shows the highest throughput, followed by **TIMESTAMP** and **NO_WAIT**. The throughput for **MVCC**, **WAIT_DIE**, and **OCC** workloads is lower compared to the others, with **OCC** showing the lowest throughput.

The x-axis represents the number of cores ranging from 0 to 1000, and the y-axis represents the throughput ranging from 0 to 14 million transactions per second (txn/s).
WRITE-INTENSIVE / MEDIUM-CONTENTION

The diagram shows the throughput (in million txns/s) as a function of the number of cores. Different lines represent different operations and their performance under varying contention levels. The operations include DL_DETECT, NO_WAIT, WAIT_DIE, TIMESTAMP, MVCC, and OCC.
WRITE-INTENSIVE / HIGH-CONTENTION

Throughput (Million txn/s)

Number of Cores
WRITE-INTENSIVE / HIGH-CONTENTION

![Graph showing throughput (Milliontxn/s) vs. number of cores for various strategies: DL_DETECT, TIMESTAMP, NO_WAIT, MVCC, WAIT_DIE, and OCC. The graph highlights the performance differences across different numbers of cores.](image-url)
WRITE-INTENSIVE / HIGH-CONTENTION

![Chart showing performance metrics for different strategies: DL_DETECT, NO_WAIT, WAIT_DIE, TIMESTAMP, MVCC, OCC. The chart compares the percentage of Useful Work, Abort, Ts Alloc., Index, Wait, and Manager over various strategies.]
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

![Graph showing lock thrashing](image)

- Throughput (Million txn/s) vs Number of Cores
- Different lines represent different values of theta: theta=0, theta=0.6, theta=0.8
- Arrows indicate points of interest on the graph.
Lock Thrashing

By reducing the frequency of lock conversion deadlock we can improve performance in a major performance bottleneck. The solution is to introduce only minimal locking—a lock in the database. Until lock holding reaches a particular point, it introduces only moderate locking, but it is a serious problem. At some point, where many transactions request locks a large number of transactions can make progress. This results in lock thrashing. Similarly, if enough transactions are blocked, throughput severely decreases. This is called lock thrashing. Figure 1: lock thrashing. When the number of active transactions gets too high, many transactions may become blocked, and the throughput can decrease.
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

The diagram illustrates the throughput (in Million ts/s) for different methods as a function of the number of cores. The methods compared include:

- Clock
- Hardware
- Atomic batch=16
- Atomic batch=8
- Atomic
- Mutex

The graph shows a clear trend where the throughput increases as the number of cores increases, with the Clock method achieving the highest throughput across all core counts.
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