TODAY’S AGENDA

Transaction Models
Concurrency Control Overview
Many-Core Evaluation
TRANSACTION DEFINITION

A txn is a sequence of actions that are executed on a shared database to perform some higher-level function.

Txns are the basic unit of change in the DBMS. No partial txns are allowed.
ACTION CLASSIFICATION

Unprotected Actions
→ These lack all of the ACID properties except for consistency. Their effects cannot be depended upon.

Protected Actions
→ These do not externalize their results before they are completely done. Fully ACID.

Real Actions
→ These affect the physical world in a way that is hard or impossible to reverse.
 TRANSACTION MODELS

Flat Txns
Flat Txns + Savepoints
Chained Txns
Nested Txns
Saga Txns
Compensating Txns
Standard txn model that starts with \texttt{BEGIN}, followed by one or more actions, and then completed with either \texttt{COMMIT} or \texttt{ROLLBACK}.

\begin{itemize}
  \item \textit{Txn #1} \begin{itemize}
    \item \texttt{BEGIN}
    \item \texttt{READ(A)}
    \item \texttt{WRITE(B)}
    \item \texttt{COMMIT}
  \end{itemize}
  \item \textit{Txn #2} \begin{itemize}
    \item \texttt{BEGIN}
    \item \texttt{READ(A)}
    \item \texttt{WRITE(B)}
    \item \texttt{ROLLBACK}
  \end{itemize}
\end{itemize}
LIMITATIONS OF FLAT TRANSACTIONS

The application can only rollback the entire txn (i.e., no partial rollbacks).

All of a txn’s work is lost if the DBMS fails before that txn finishes.

Each txn takes place at a single point in time.
LIMITATIONS OF FLAT TRANSACTIONS

Multi-Stage Planning
→ An application needs to make multiple reservations.
→ All the reservations need to occur or none of them.

Bulk Updates
→ An application needs to update one billion records.
→ This txn could take hours to complete and therefore the DBMS is exposed to losing all of its work for any failure or conflict.
TRANSACTION SAVEPOINT

Save the current state of processing for the txn and provide a handle for the application to refer to that savepoint.

The application can control the state of the txn through these checkpoints:

→ **ROLLBACK** – Revert all changes back to the state of the DB at the savepoint.

→ **RELEASE** – Destroys a savepoint previously defined in the txn.
TRANSACTION SAVEPOINTS

**Txn #1**

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **ROLLBACK TO 1**
- **WRITE(C)**
- **COMMIT**

**New Savepoint**
**TRANSACTION SAVEPOINTS**

**Txn #1**

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **ROLLBACK TO 1**
- **WRITE(C)**
- **COMMIT**

**New Savepoint**

- A

---

**CMU 15-721 (Spring 2017)**
**Transaction Savepoints**

**Txn #1**
- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- ROLLBACK TO 1
- WRITE(C)
- COMMIT

**Savepoint #1**
- A

**New Savepoint**
**TRANSACTION SAVEPOINTS**

**Txn #1**

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **ROLLBACK TO 1**
- **WRITE(C)**
- **COMMIT**

New Savepoint

- **A**
- **B**
**TRANSACTION SAVEPOINTS**

*Txn #1*

**BEGIN**

**WRITE(A)**

**SAVEPOINT 1**

**WRITE(B)**

**ROLLBACK TO 1**

**WRITE(C)**

**COMMIT**

**Savepoint #1**

**A**

**New Savepoint**

**New Savepoint**
**TRANSACTION SAVEPOINTS**

*Txn #1*

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **ROLLBACK TO 1**
- **WRITE(C)**
- **COMMIT**

**Savepoint#1**

- A

**New Savepoint**

- X

**New Savepoint**

- C
**TRANSACTION SAVEPOINTS**

**Txn #1**

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- ROLLBACK TO 1
- WRITE(C)
- COMMIT

**Savepoint#1**

- A

**New Savepoint**

- X

**New Savepoint**

- C
**Transaction Savepoints**

*Txn #1*

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3
```
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3

New Savepoint

A
**Transaction Savepoints**

*Txn #1*

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- SAVEPOINT 2
- WRITE(C)
- SAVEPOINT 3
- RELEASE 2
- WRITE(D)
- ROLLBACK TO 3

**Savepoint #1**

- A

**New Savepoint**
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3

Savepoint#1

A

New Savepoint

B
**TRANSACTION SAVEPOINTS**

*Txn #1*

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **SAVEPOINT 2**
- **WRITE(C)**
- **SAVEPOINT 3**
- **RELEASE 2**
- **WRITE(D)**
- **ROLLBACK TO 3**

**Savepoint #1**
- A

**Savepoint #2**
- B

**New Savepoint**
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3

1. Transaction #1 starts with a **BEGIN** statement.
2. **WRITE(A)** modifies data A.
3. A **SAVEPOINT 1** is created.
4. **WRITE(B)** modifies data B.
5. A **SAVEPOINT 2** is created.
6. **WRITE(C)** modifies data C.
7. A **SAVEPOINT 3** is created.
8. **RELEASE 2** rolls back to the last savepoint (SAVEPOINT 2).
9. **WRITE(D)** modifies data D.
10. A **ROLLBACK TO 3** rolls back to the third savepoint (SAVEPOINT 3).
TRANSACTION SAVEPOINTS

**Txn #1**

BEGIN  
WRITE(A)  
SAVEPOINT 1  
WRITE(B)  
SAVEPOINT 2  
WRITE(C)  
SAVEPOINT 3  
RELEASE 2  
WRITE(D)  
ROLLBACK TO 3

**Savepoint#1**

A

**Savepoint#2**

B

**Savepoint#3**

C

**New Savepoint**
TRANSACTION SAVEPOINTS

**Txn #1**

<table>
<thead>
<tr>
<th>BEGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE(A)</td>
</tr>
<tr>
<td>SAVEPOINT 1</td>
</tr>
<tr>
<td>WRITE(B)</td>
</tr>
<tr>
<td>SAVEPOINT 2</td>
</tr>
<tr>
<td>WRITE(C)</td>
</tr>
<tr>
<td>SAVEPOINT 3</td>
</tr>
<tr>
<td>RELEASE 2</td>
</tr>
<tr>
<td>WRITE(D)</td>
</tr>
<tr>
<td>ROLLBACK TO 3</td>
</tr>
</tbody>
</table>

**Savepoint#1**

- **A**

**Savepoint#2**

- **B**

**Savepoint#3**

- **C**

**New Savepoint**
TRANSACTION SAVEPOINTS

**Txn #1**

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- SAVEPOINT 2
- WRITE(C)
- SAVEPOINT 3
- RELEASE 2
- WRITE(D)
- ROLLBACK TO 3

**Savepoint#1**

- A

**Savepoint#2**

- B

**Savepoint#3**

- C

**New Savepoint**

- D
**TRANSACTION SAVEPOINTS**

```
Txn #1
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3

Savepoint#1
A

Savepoint#2
B

Savepoint#3
C

New Savepoint
D

```
**TRANSACTION SAVEPOINTS**

**Txn #1**

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3
```

**Savepoint #1**

```
A
```

**Savepoint #2**

```
B
```

**Savepoint #3**

```
C
```

**New Savepoint**

```
D
```
NESTED TRANSACTIONS

Savepoints organize a transaction as a sequence of actions that can be rolled back individually.

Nested txns form a hierarchy of work.

→ The outcome of a child txn depends on the outcome of its parent txn.
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
WRITE(B)
BEGIN
WRITE(B)
COMMIT
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
BEGIN
WRITE(B)
COMMIT
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

**Txn #1**

```
BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

**Txn #1**

```
BEGIN
WRITE(A)
BEGIN
  BEGIN
  WRITE(C)
  COMMIT
END
WRITE(B)
ROLLBACK
WRITE(D)
BEGIN
```
**NESTED TRANSACTIONS**

**Txn #1**

```
BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
WRITE(B)
BEGIN
WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

**Txn #1**
- BEGIN
- WRITE(A)
- BEGIN
- COMMIT

**Sub-Txn #1.1**
- BEGIN
- WRITE(B)
- BEGIN
- WRITE(D)
- ROLLBACK

**Sub-Txn #1.1.1**
- BEGIN
- WRITE(C)
- COMMIT
NESTED TRANSACTIONS

**Txn #1**

```
BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
ROLLBACK
WRITE(D)
BEGIN
```
NESTED TRANSACTIONS

Txn #1

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>WRITE(A)</th>
<th>BEGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Sub-Txn #1.1

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>WRITE(B)</th>
<th>BEGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WRITE(D)</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLLBACK</td>
<td></td>
</tr>
</tbody>
</table>

Sub-Txn #1.1.1

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>WRITE(C)</th>
<th>BEGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WRITE(D)</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
ROLLBACK
WRITE(D)

Sub-Txn #1.1

SUB-Txn #1.1

BEGIN
WRITE(B)
BEGIN
WRITE(D)
COMMIT

BEGIN
WRITE(C)

BEGIN
ROLLBACK

✓
TRANSACTION CHAINS

Multiple txns executed one after another.
Combined **COMMIT / BEGIN** operation is atomic.
→ No other txn can change the state of the database as seen by the second txn from the time that the first txn commits and the second txn begins.

Differences with savepoints:
→ **COMMIT** allows the DBMS to free locks.
→ Cannot rollback previous txns in chain.
**TRANSACTION CHAINS**

*Txn #1*

```
BEGIN
WRITE(A)
COMMIT
```

*Txn #2*

```
BEGIN
WRITE(B)
COMMIT
```

*Txn #3*

```
BEGIN
WRITE(C)
ROLLBACK
```
BULK UPDATE PROBLEM

These other txn models are nice, but they still do not solve our bulk update problem.

Chained txns seems like the right idea but they require the application to handle failures and maintain it’s own state.
→ Has to be able to reverse changes when things fail.
COMPENSATING TRANSACTIONS

A special type of txn that is designed to semantically reverse the effects of another already committed txn.

Reversal has to be **logical** instead of physical.
→ Example: Decrement a counter by one instead of reverting to the original value.
SAGA TRANSACTIONS

A sequence of chained txns $T_1,...,T_n$ and compensating txns $C_1,...,C_{n-1}$ where one of the following is guaranteed:

$\longrightarrow$ The txns will commit in the order $T_1,...,T_n$

$\longrightarrow$ The txns will commit in the order $T_1,...,T_j,C_j,...,C_1$ (where $j < n$)

SAGAS
SAGA TRANSACTIONS

Txn #1
- BEGIN
- WRITE(A+1)
- COMMIT

Txn #2
- BEGIN
- WRITE(B+1)
- COMMIT

Txn #3
- BEGIN
- WRITE(C+1)

Comp Txn #1
- BEGIN
- WRITE(A-1)
- COMMIT

Comp Txn #2
- BEGIN
- WRITE(B-1)
- COMMIT
SAGA TRANSACTIONS

Txn #1

BEGIN
WRITE(A+1)
COMMIT

Txn #2

BEGIN
WRITE(B+1)
COMMIT

Txn #3

BEGIN
WRITE(C+1)

Comp Txn #1

BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2

BEGIN
WRITE(B-1)
COMMIT
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
TXN INTERNAL STATE

Undo Log Entries
→ Stored in an in-memory data structure.
→ Dropped on commit.

Redo Log Entries
→ Append to the in-memory tail of WAL.
→ Flushed to disk on commit.

Read/Write Set
→ Depends on the concurrency control scheme.
CONCURRENCY CONTROL SCHEMES

Two-Phase Locking (2PL)
→ Assume txns will conflict so they must acquire locks on elements before they are allowed to access them.

Timestamp Ordering (T/O)
→ Assume that conflicts are rare so txns do not need to acquire locks and instead check for conflicts at commit time.
TWO-PHASE LOCKING

Txn #1

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

Growing Phase

Shrinking Phase
TWO-PHASE LOCKING

**Txn #1**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>UNLOCK(A)</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>
TWO-PHASE LOCKING

Txn #1
BEGIN
LOCK(A) READ(A) LOCK(B) WRITE(B) UNLOCK(A) 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) WRITE(B) UNLOCK(A) 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)
- WRITE(B)
- UNLOCK(A)
- UNLOCK(B)
- WRITE(B)
- COMMIT

**Txn #2**

BEGIN

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

**Conflict Resolution**

- The transactions conflict because they both attempt to modify the same data (A) in different phases.
- The system must choose one transaction to commit and the other to abort, ensuring data consistency.

**Resolution Example**

- Assume Transaction #1 is chosen to commit.
- Transaction #2 is aborted.
- The database is left in a consistent state with Transaction #1's changes.
TWO-PHASE LOCKING

Txn #1

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

Txn #2

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

#1

BEGIN
READ(A)

WRITE(B)

WRITE(A)

COMMIT
TIMESTAMP ORDERING

<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>
### TIMESTAMP ORDERING

<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>
**Timestamp Ordering**

<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>

**Example Transaction:**

- **BEGIN**
- **READ(A)**
- **WRITE(B)**
- **WRITE(A)**
- **COMMIT**
TIMESTAMP ORDERING

<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>
TIMESTAMP ORDERING

<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>
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.

Multi-Version Concurrency Control (MVCC)
→ Create a new version of a tuple whenever a txn modifies it. Use timestamps as version id.
→ Check visibility on every read/write.
## 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**
- Basic T/O Algorithm

**MVCC**
- Multi-Version T/O

**OCC**
- 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>

(postgres, oracle, informix, memsql, sap_hana, nuodb, peloton, cmu_dmg)
1000-CORE CPU SIMULATOR

**DBx1000 Database System**
- In-memory DBMS with pluggable lock manager.
- No network access, logging, or concurrent indexes

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

STARING INTO THE ABYSS: AN EVALUATION OF CONCURRENCY CONTROL WITH ONE THOUSAND CORES
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.
READ-ONLY WORKLOAD

![Graph showing throughput vs. number of cores for different workloads](image)

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

Throughput (Million txn/s) vs. Number of Cores
READ-ONLY WORKLOAD

![Graph showing throughput (Million txn/s) vs. number of cores for various concurrency control mechanisms. The y-axis represents throughput in Milliontxn/s, ranging from 0 to 14. The x-axis represents the number of cores, ranging from 0 to 1000. Different markers correspond to different concurrency control mechanisms: DL_DETECT (red circle), TIMESTAMP (green triangle), NO_WAIT (orange diamond), MVCC (blue dotted line), WAIT_DIE (yellow square), and OCC (purple plus sign). The graph indicates that as the number of cores increases, the throughput also increases for all mechanisms, with DL_DETECT having the highest performance followed by NO_WAIT and MVCC.](image-url)
READ-ONLY WORKLOAD
WRITE-INTENSIVE / MEDIUM-CONTENTION

![Graph showing throughput vs. number of cores for different concurrency control mechanisms.]

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

Throughput (Million txn/s)

Number of Cores

0 200 400 600 800 1000

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5
WRITE-INTENSIVE / MEDIUM-CONTENTION

![Graph showing throughput vs. number of cores for different concurrency control strategies. The graph compares DL_DETECT, TIMESTAMP, NO_WAIT, MVCC, WAIT_DIE, and OCC strategies. The x-axis represents the number of cores ranging from 0 to 1000, and the y-axis represents throughput in million transactions per second (txn/s) ranging from 0.0 to 4.5. The graph demonstrates how different strategies affect the system's performance under varying core loads.]
WRITE-INTENSIVE / MEDIUM-CONTENTION

Throughput (Million txn/s)

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

Number of Cores
WRITE-INTENSIVE / HIGH-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
LOCK THRASHING

Lock Thrashing

By reducing the frequency of lock conversion deadlock, we have disproved deadlocks as a major performance bottleneck. In a recent study of transaction performance in a large, realistic database system, it was found that deadlocks occur at a rate of 1 in 100,000 transactions. The problem is that even with high levels of locking, deadlocks can still occur. In this case, it will not improve the transaction speed or wait time, since it doesn't know when it is safe to do so.

When the number of active transactions gets too high, many transactions suddenly become blocked, and transaction speed is reduced. This is called lock thrashing.

The key is to increase the number of active transactions to maximize throughput without increasing the waiting time for other transactions.

![Graph showing throughput vs. number of active transactions.](image)
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

![Graph showing throughput (Million ts/s) vs number of cores for different methods like Clock, Hardware, Atomic batch=16, Atomic batch=8, Atomic, Mutex. The graph illustrates the performance comparison between these methods as the number of cores increases.]
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.
PARTITION-LEVEL LOCKING

The database is split up into horizontal partitions:
→ Each partition is assigned a single-threaded execution engine that has exclusive access to its data.
→ In-place updates.

Only one txn can execute at a time per partition.
→ Order txns based on when they arrive at the DBMS.
→ A txn acquires the lock for a partition when it has the lowest timestamp.
→ It is not allowed to access any partition that it does not hold the lock for.
READ-ONLY WORKLOAD
MULTI-PARTITION WORKLOADS
PARTING THOUGHTS

Concurrency control is hard to get correct and perform well.

Evaluation did not consider HTAP workloads.
NEXT CLASS

Isolation Levels
Stored Procedures
High-Performance OCC
PELOTON TEAM MEETING

Wednesdays @ 12:00pm in GHC 9115
Come if you are interested in doing an independent study or capstone project. Free food.