Lecture #04 – Optimistic Concurrency Control
ADMINISTRATIVE

Project #1 is due Tuesday Jan 31st @ 11:59pm

Project #2 will be released on Tuesday too.
→ You need a group of three people.
→ I will send out a sign-up sheet.
TODAY’S AGENDA

Isolation Levels
Stored Procedures
Optimistic Concurrency Control
Observation

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance.

We may want to use a weaker level of consistency to improve scalability.
ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.
Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
→ Dirty Read Anomaly
→ Unrepeatable Reads Anomaly
→ Phantom Reads Anomaly
ANSI ISOLATION LEVELS

SERIALIZABLE
→ No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS
→ Phantoms may happen.

READ COMMITTED
→ Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED
→ All of them may happen.
ISOLATION LEVEL HIERARCHY

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READS
- SERIALIZABLE
# ANSI ISOLATION LEVELS

<table>
<thead>
<tr>
<th></th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Greenplum</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>CURSOR STABILITY</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MySQL</td>
<td>REPEATABLE READS</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MemSQL</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Oracle</td>
<td>READ COMMITTED</td>
<td>SNAPSHOT ISOLATION</td>
</tr>
<tr>
<td>Postgres</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>VoltDB</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
</tbody>
</table>

Source: Peter Bailis
CRITICISM OF ISOLATION LEVELS

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS.

Two additional isolation levels:
→ CURSOR STABILITY
→ SNAPSHOT ISOLATION
The DBMS’s internal cursor maintains a lock on an item in the database until it moves on to the next item.

CS is a stronger isolation level in between **REPEATABLE READS** and **READ COMMITTED** that can (sometimes) prevent the **Lost Update Anomaly**.
LOST UPDATE ANOMALY

**Txn #1**

**BEGIN**

READ(A)

... ...

WRITE(A)

**COMMIT**

**Txn #2**

**BEGIN**

... ...

WRITE(A)

... ...

**COMMIT**
LOST UPDATE ANOMALY

*Txn #1*

```
BEGIN
READ(A) ...
WRITE(A)
COMMIT
```

*Txn #2*

```
BEGIN
WRITE(A) ...
COMMIT
```
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)

WRITE(A)

COMMIT

**Txn #2**

BEGIN

WRITE(A)

COMMIT
LOST UPDATE ANOMALY

Txn #1

BEGIN
READ(A)

• • •

WRITE(A)

COMMIT

Txn #2

BEGIN

• • • • • • • •

WRITE(A)

COMMIT
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)

... ...

WRITE(A)

COMMIT

**Txn #2**

BEGIN

... ...

WRITE(A)

... ...

COMMIT
LOST UPDATE ANOMALY

*Txn #1*

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

*Txn #2*

BEGIN | ... | WRITE(A) | ... | COMMIT
LOST UPDATE ANOMALY

\[ \text{Txn } #1 \]

\[ \begin{array}{c}
\text{BEGIN} \\
\text{READ}(A) \\
\text{• • •} \\
\text{WRITE}(A) \\
\text{COMMIT}
\end{array} \]

\[ \text{Txn } #2 \]

\[ \begin{array}{c}
\text{BEGIN} \\
\text{• • •} \\
\text{WRITE}(A) \\
\text{• • •} \\
\text{COMMIT}
\end{array} \]

Txn #2’s write to A will be lost even though it commits after Txn #1.

A **cursor lock** on A would prevent this problem (but not always).
SNAPSHOT ISOLATION (SI)

Guarantees that all reads made in a txn see a consistent snapshot of the database that existed at the time the txn started.

→ A txn will commit under SI only if its writes do not conflict with any concurrent updates made since that snapshot.

SI is susceptible to the Write Skew Anomaly
WRITE SKEW ANOMALY

\( \text{Txn } \#1 \)
Change white marbles to black.

\( \text{Txn } \#2 \)
Change black marbles to white.
**WRITE SKEW ANOMALY**

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
ISOLATION LEVEL HIERARCHY

- Serializable
  - Repeatable Reads
  - Snapshot Isolation
  - Cursor Stability
  - Read Committed
  - Read Uncommitted
Figure 4-1: A partial order to relate various isolation levels.

Source: Atul Adya
OBSERVATION

Disk stalls are (almost) gone when executing txns in an in-memory DBMS.

There are still other stalls when an app uses **conversational** API to execute queries on DBMS

→ ODBC/JDBC

→ DBMS-specific wire protocols
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL
Program Logic

SQL
Program Logic

;

COMMIT
CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ...
  COMMIT
```

Parser
Planner
Optimizer
Query Execution
CONVERSATIONAL DATABASE API

Application

BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ...
  COMMIT

Parser
Planner
Optimizer
Query Execution

CMU 15-721 (Spring 2017)
CONVERSATIONAL DATABASE API

Application

BEGIN
SQL
Program Logic
SQL
Program Logic
;
COMMIT
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL
Program Logic

SQL
Program Logic

;

COMMIT

Parser
Planner
Optimizer
Query Execution
SOLUTIONS

Prepared Statements
→ Removes query preparation overhead.

Query Batches
→ Reduces the number of network roundtrips.

Stored Procedures
→ Removes both preparation and network stalls.
A **stored procedure** is a group of queries that form a logical unit and perform a particular task on behalf of an application directly inside of the DBMS.

Programming languages:

→ **SQL/PSM** (standard)
→ **PL/SQL** (Oracle / IBM / MySQL)
→ **PL/pgSQL** (Postgres)
→ **Transact-SQL** (Microsoft / Sybase)
STORED PROCEDURES

Application

BEGIN

SQL
Program Logic

SQL
Program Logic

;

COMMIT
APPLICATION

CALL PROC(x=99)

PROC(x)

BEGIN
  SQL
  Program Logic
  SQL
  Program Logic

  COMMIT
CREATE PROCEDURE testProc
    (num INT, name VARCHAR) RETURNS INT
BEGIN
    DECLARE cnt INT DEFAULT 0;
    LOOP
        INSERT INTO student VALUES (cnt, name);
        SET cnt := cnt + 1;
        IF (cnt > 15) THEN
            RETURN cnt;
        END IF;
    END LOOP;
END;
ADVANTAGES

Reduce the number of round trips between application and database servers.

Increased performance because queries are pre-compiled and stored in DBMS.

Procedure reuse across applications.

Server-side txn restarts on conflicts.
DISADVANTAGES

Not as many developers know how to write SQL/PSM code.
→ Safe Languages vs. Sandbox Languages

Outside the scope of the application so it is difficult to manage versions and hard to debug.

Probably not be portable to other DBMSs.

DBAs usually don’t give permissions out freely…
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

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

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*

**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>
**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>888</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 CONCURREN CY 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>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

### Txn #1

<table>
<thead>
<tr>
<th>Action</th>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>WRITE</td>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>WRITE</td>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

### 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>
<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>
**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>

### Workspace Records

<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
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>888</td>
<td>10001</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>10001</td>
</tr>
</tbody>
</table>
READ PHASE

Track the read/write sets of txns and store their writes in a private workspace. The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.
VALIDATION PHASE

When the txn invokes **COMMIT**, the DBMS checks if it conflicts with other txns.

Two methods for this phase:
→ Backward Validation
→ Forward Validation
BACKWARD VALIDATION

Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.
BACKWARD VALIDATION

Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.
FORWARD VALIDATION

Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.
Check whether the committing txn intersects its read/write sets with any active txns that have **not** yet committed.
VALIDATION PHASE

Original OCC uses serial validation.
Parallel validation means that each txn must check the read/write sets of other txns that are trying to validate at the same time.
→ Each txn has to acquire locks for its write set records in some \textit{global order}.
→ The txn does not need locks for read set records.
WRITE PHASE

The DBMS propagates the changes in the txn’s write set to the database and makes them visible to other txns.

As each record is updated, the txn releases the lock acquired during the Validation Phase.
MODERN OCC

Harvard/MIT Silo
MIT/CMU TicToc
SILO

Single-node, in-memory OLTP DBMS.
→ Serializable OCC with parallel backward validation.
→ Stored procedure-only API.
No writes to shared-memory for read txns.
Batched timestamp allocation using epochs.

Pure awesomeness from Eddie Kohler.
SILO: EPOCHS

Time is sliced into fixed-length epochs (40ms).
All txns that start in the same epoch will be committed together at the end of the epoch.
→ Txns that span an epoch have to refresh themselves to be carried over into the next epoch.

Worker threads only need to synchronize at the beginning of each epoch.
SILO: TRANSACTION IDS

Each worker thread generates a unique txn id based on the current epoch number and the next value in its assigned batch.
### SILO: COMMIT PROTOCOL

#### Table

<table>
<thead>
<tr>
<th>TID Word</th>
<th>POINTER</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>Ø</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Ø</td>
<td>Tupac</td>
<td>$999</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Ø</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Ø</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
</tbody>
</table>

#### EPOCH | BATCH | TIMESTAMP | EXTRA

**Write Lock Bit**

**Latest Version Bit**

**Absent Bit**
SILO: COMMIT PROTOCOL

Step #1: Lock Write Set
## SILO: COMMIT PROTOCOL

### Workspace

<table>
<thead>
<tr>
<th>Read Set</th>
<th>Write Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-# O.D.B. $13</td>
<td>Tupac $777</td>
</tr>
<tr>
<td>#-####-# Tupac $999</td>
<td></td>
</tr>
</tbody>
</table>

### TID Word | POINTER | ATTR1 | ATTR2 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>X</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#-####-#</td>
<td>X</td>
<td>Tupac</td>
<td>$999</td>
</tr>
<tr>
<td>#-####-#</td>
<td>X</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#-####-#</td>
<td>X</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
</tbody>
</table>

**Step #1:** Lock Write Set

**Step #2:** Examine Read Set
SILO: COMMIT PROTOCOL

Workspace

<table>
<thead>
<tr>
<th>Read Set</th>
<th>TID Word</th>
<th>POINTER</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-#####-#</td>
<td>O.D.B.</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Tupac</td>
<td>$999</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Set</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tupac</td>
<td></td>
<td>$777</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step #1: Lock Write Set

Step #2: Examine Read Set
**SILO: COMMIT PROTOCOL**

**Workspace**

<table>
<thead>
<tr>
<th>Read Set</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tupac</td>
<td>$777</td>
</tr>
</tbody>
</table>

**Step #1:** Lock Write Set

**Step #2:** Examine Read Set
**SILO: COMMIT PROTOCOL**

**Workspace**

**Read Set**

<table>
<thead>
<tr>
<th>#-###-#</th>
<th>O.D.B.</th>
<th>$13</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-###-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
</tbody>
</table>

**Write Set**

| Tupac | $777 |

**Step #1:** Lock Write Set

**Step #2:** Examine Read Set

**Step #3:** Install Write Set

<table>
<thead>
<tr>
<th>TID Word</th>
<th>POINTER</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-#####-#</td>
<td>☐</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>☐</td>
<td>Tupac</td>
<td>$777</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>☐</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>☐</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
</tbody>
</table>
Cooperative threads GC.

Each worker thread marks a deleted object with a **reclamation epoch**.

→ This is the epoch after which no thread could access the object again, and thus can be safely removed.

→ Object references are maintained in thread-local storage to avoid unnecessary data movement.
SILO: RANGE QUERIES

DBMS handles phantoms by tracking the txn’s scan set (node set) on indexes.
→ Have to include “virtual” entries for keys that do not exist in the index.

We will discuss key-range and index gap locking next week...
SILO: PERFORMANCE

*Database: TPC-C with 28 Warehouses*

*Processor: 4 sockets, 8 cores per socket*
TICTOC

Serializable OCC implemented in DBx1000.
→ Parallel backward validation
→ Stored procedure-only API

No global timestamp allocation. Txn timestamps are derived from records.
**TICTOC: RECORD TIMESTAMPs**

**Write Timestamp (W-TS):**
→ The logical timestamp of the last committed txn that wrote to the record.

**Read Timestamp (R-TS):**
→ The logical timestamp of the last txn that read the record.

A record is considered valid from W-TS to R-TS
TICTOC: VALIDATION PHASE

$Txn$

$\text{WRITE}(A)$

$\text{READ}(B)$

$\text{READ}(C)$

$W-TS$

$R-TS$

LOGICAL TIME

1 2 3 4
TICTOC: VALIDATION PHASE

Step #1: Lock Write Set
TICTOC: VALIDATION PHASE

Step #1: Lock Write Set
Step #2: Compute CommitTS
TICTOC: VALIDATION PHASE

**Txn**

- **Step #1**: Lock Write Set
- **Step #2**: Compute CommitTS
- **Step #3**: Validate Read Set

**LOGICAL TIME**

- **WRITE(A)**: Lock at Step #1
- **READ(B)**: Read at Step #2
- **READ(C)**: Read at Step #2

**CommitTS**
**TICTOC: VALIDATION PHASE**

**Txn**

- **Step #1:** Lock Write Set
- **Step #2:** Compute CommitTS
- **Step #3:** Validate Read Set

**LOGICAL TIME**

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td>2</td>
<td>READ(B)</td>
</tr>
<tr>
<td>3</td>
<td>CommitTS</td>
</tr>
<tr>
<td>4</td>
<td>READ(C)</td>
</tr>
</tbody>
</table>
**TICTOC: VALIDATION PHASE**

**Txn**

1. **Step #1:** Lock Write Set
2. **Step #2:** Compute CommitTS
3. **Step #3:** Validate Read Set
   - **Case 1:** Latest Version
TICTOC: VALIDATION PHASE

Txn

Step #1: Lock Write Set
Step #2: Compute CommitTS
Step #3: Validate Read Set

Case 1: Latest Version
Case 2: Updated Before CommitTS
**TICTOC: VALIDATION PHASE**

*Txn*

**Logics Time**

- **Step #1:** Lock Write Set
- **Step #2:** Compute CommitTS
- **Step #3:** Validate Read Set

**Case 1:** Latest Version
**Case 2:** Updated Before CommitTS
**Case 3:** Updated After CommitTS
TICTOC: PERFORMANCE

Database: 10GB YCSB
Processor: 4 sockets, 10 cores per socket

Throughput (txn/sec) vs. Thread Count

- **Medium Contention**: 90% Reads / 10% Writes
- **High Contention**: 50% Reads / 50% Writes
PARTING THOUGHTS

Trade-off between aborting txns early or later.
→ **Early**: Avoid wasted work for txns that will eventually abort, but has checking overhead.
→ **Later**: No runtime overhead but lots of wasted work under high contention.

Silo is a very influential system.
PARTING THOUGHTS

Trade-off between aborting txns early or later.

→ Early: Avoid wasted work for txns that will eventually abort, but has checking overhead.

→ Later: No runtime overhead but lots of wasted work under high contention.

Silo is a very influential system.
NEXT CLASS

Multi-Version Concurrency Control