Lecture #17

Timestamp
Ordering
Concurrency
Control
CONCUERENCY CONTROL APPROACHES

Two-Phase Locking (2PL)
→ Determine serializability order of conflicting operations at runtime while txns execute.

Timestamp Ordering
→ A serialization mechanism using timestamps.

Optimistic Concurrency Control
→ Run then check for serialization violations.

Pessimistic

Optimistic
Use timestamps to determine the serializability order of txns.

If $\text{TS}(T_i) < \text{TS}(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to the serial schedule where $T_i$ appears before $T_j$. 
Each txn $T_i$ is assigned a unique fixed timestamp that is monotonically increasing.

→ Let $TS(T_i)$ be the timestamp allocated to txn $T_i$.

→ Different schemes assign timestamps at different times during the txn.

Multiple implementation strategies:

→ System/Wall Clock.
→ Logical Counter.
→ Hybrid.
TODAY’S AGENDA

Basic Timestamp Ordering (T/O) Protocol
Optimistic Concurrency Control
Isolation Levels
BASIC T/O

Txns read and write objects without locks.

Every object $X$ is tagged with timestamp of the last txn that successfully did read/write:

$\rightarrow W^{-}\text{TS}(X)$ – Write timestamp on $X$

$\rightarrow R^{-}\text{TS}(X)$ – Read timestamp on $X$

Check timestamps for every operation:

$\rightarrow$ If txn tries to access an object “from the future”, it aborts and restarts.
BASIC T/O – READS

Don’t read stuff from the “future.”

Action: Transaction $T_i$ wants to read object $X$.

If $TS(T_i) < W-TS(X)$, this violates the timestamp order of $T_i$ with regard to the writer of $X$.
→ Abort $T_i$ and restart it with a new TS.

Else:
→ Allow $T_i$ to read $X$.
→ Update $R-TS(X)$ to $\max(R-TS(X), \ TS(T_i))$
→ Make a local copy of $X$ to ensure repeatable reads for $T_i$. 
BASIC T/O – WRITES

Can’t write if a future transaction has read or written to the object.

Action: Transaction $T_i$ wants to write object $X$.

If $TS(T_i) < R-TS(X)$ or $TS(T_i) < W-TS(X)$

→ Abort and restart $T_i$.

Else:

→ Allow $T_i$ to write $X$ and update $W-TS(X)$

→ Also, make a local copy of $X$ to ensure repeatable reads.
BASIC T/O – EXAMPLE #1

**Schedule**

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
</table>
BEGIN |     |     |
R(B)   |     |     |
R(A)   |     |     |
R(A)   |     |     |
COMMIT |     |     |
```

**Database**

```
<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
BASIC T/O – EXAMPLE #1

Schedule

TS(T₁) = 1

BEGIN
R(B)
R(A)
R(A)
COMMIT

BEGIN
R(B)
R(A)
R(A)
W(A)
COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TIME

T₁

T₂
BASIC T/O – EXAMPLE #1

Schedule

```
BEGIN
R(B)
R(A)
R(A)
COMMIT
```

```
BEGIN
R(B)
W(B)
R(A)
W(A)
COMMIT
```

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\(TS(T_1)=1\)
BASIC T/O – EXAMPLE #1

Schedule

BEGIN
R(B)
R(A)
R(A)
COMMIT

Database

Object | R-TS | W-TS
--- | --- | ---
A | 0 | 0
B | 1 | 0

TS(T₁)=1

TS(T₂)=2

TIME

TS(T₁)=1

T₂

BEGIN
R(B)
R(A)
R(A)
COMMIT

BEGIN
R(B)
W(B)
R(A)
W(A)
COMMIT
**BASIC T/O – EXAMPLE #1**

**Schedule**

```
BEGIN
R(B)
R(A)
R(A)
COMMIT

BEGIN
R(B)
W(B)
R(A)
W(A)
COMMIT
```

**Database**

```
<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Time**

$TS(T_1) = 1$

$TS(T_2) = 2$
BASIC T/O – EXAMPLE #1

Schedule

BEGIN
R(B)

R(A)

R(A)

COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$TS(T_1) = 1$

$TS(T_2) = 2$
**BASIC T/O – EXAMPLE #1**

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Schedule</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

**Example 1**

```
BEGIN
R(B)
R(A)
R(A)
COMMIT
```

```
BEGIN
R(B)
W(B)
R(A)
W(A)
COMMIT
```

**Time**

- \(TS(T_1) = 1\)
- \(TS(T_2) = 2\)
**BASIC T/O – EXAMPLE #1**

**Schedule**

```
BEGIN
R(B)
R(A)
R(A)
COMMIT
```

```
BEGIN
R(B)
W(B)
R(A)
W(A)
COMMIT
```

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
BASIC T/O – EXAMPLE #1

Schedule

TS(T₁) = 1

BEGIN
R(B)
R(A)
R(A)
COMMIT

TS(T₂) = 2

BEGIN
R(B)
R(A)
W(A)
W(B)
COMMIT

Database

TS(T₁) < TS(T₂)

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
BASIC T/O – EXAMPLE #1

Schedule

BEGIN
R(B)

T1

R(A)

TS(T1)=1

T2

R(A)

BEGIN
R(B)
W(B)

R(A)

W(A)

COMMIT

Database

Object  R-TS  W-TS
A      2      2
B      2      2

TS(T2)=2

TIME
**BASIC T/O – EXAMPLE #1**

**Schedule**

- **TS(T₁)=1**
  - BEGIN
  - R(B)
  - R(A)
  - COMMIT

- **TS(T₂)=2**
  - BEGIN
  - R(B)
  - W(B)
  - R(A)
  - W(A)
  - COMMIT

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

No violations so both txns are safe to commit.
### BASIC T/O – EXAMPLE #2

#### Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>BEGIN R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td>T_2</td>
<td>BEGIN W(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Time**: 20
BASIC T/O – EXAMPLE #2

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Schedule</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN R(A) W(A)</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>BEGIN W(A) COMMIT</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>R(A) COMMIT</td>
<td>B</td>
</tr>
</tbody>
</table>

Example 
BEGIN R(A) 
W(A) 
R(A) 
COMMIT

BEGIN W(A) COMMIT
**BASIC T/O – EXAMPLE #2**

**Schedule**

```
BEGIN
R(A)
W(A)
R(A)
COMMIT
```

```
BEGIN
W(A)
COMMIT
```

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Time**

T1  T2
**BASIC T/O – EXAMPLE #2**

**Schedule**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN R(A)</td>
<td>BEGIN W(A) COMMIT</td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Violation: \( TS(T₁) < W-TS(A) \)
**BASIC T/O – EXAMPLE #2**

---

**Schedule**

T1

BEGIN
R(A)

BEGIN
W(A)

COMMIT

T2

BEGIN
R(A)

BEGIN
W(A)

COMMIT

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Violation: TS(T1) < W-TS(A)*

T1 cannot overwrite update by T2, so the DBMS must abort it!
THOMAS WRITE RULE

If $\text{TS}(T_i) < R - \text{TS}(X)$:
→ Abort and restart $T_i$.

If $\text{TS}(T_i) < W - \text{TS}(X)$:
→ **Thomas Write Rule**: Ignore the write to allow the txn to continue executing without aborting.
→ This violates timestamp order of $T_i$.

Else:
→ Allow $T_i$ to write $X$ and update $W - \text{TS}(X)$
If $\text{TS}(T_i) < R - \text{TS}(X)$:
→ Abort and restart $T_i$.

If $\text{TS}(T_i) < W - \text{TS}(X)$:
→ Thomas Write Rule: Ignore the write to allow the txn to continue executing without aborting.
→ This violates timestamp order of $T_i$.

Else:
→ Allow $T_i$ to write $X$ and update $W - \text{TS}(X)$.
BASIC T/O – EXAMPLE #2

Schedule

T₁  
BEGIN R(A)
W(A)
R(A)
COMMIT

T₂  
BEGIN W(A)
COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TIME
BASIC T/O – EXAMPLE #2

Schedule

T₁

BEGIN
R(A)

W(A)

R(A)

COMMIT

T₂

BEGIN
W(A)

COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TIME
**BASIC T/O – EXAMPLE #2**

**Schedule**

**T1**
- BEGIN
- R(A)
- W(A)
- R(A)
- COMMIT

**T2**
- BEGIN
- W(A)
- COMMIT

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We do not update W-TS(A)

Skip doing the actual write and allow T1 to commit. (Do write to the local copy if repeatable read is required.)
BASIC T/O

Generates a schedule that is conflict serializable if you do **not** use the **Thomas Write Rule**.

→ No deadlocks because no txn ever waits.

→ Possibility of starvation for long txns if short txns keep causing conflicts.

Not aware of any DBMS that uses the basic T/O protocol described here.

→ It provides the building blocks for OCC / MVCC.
BASIC T/O – PERFORMANCE ISSUES

High overhead from copying data to txn’s workspace and from updating timestamps.

→ Every read requires the txn to write to the database.

Long running txns can get starved.

→ The likelihood that a txn will read something from a newer txn increases.
If you assume that conflicts between txns are rare and that most txns are short-lived, then forcing txns to acquire locks or update timestamps adds unnecessary overhead.

A better approach is to optimize for the no-conflict case.
The DBMS creates a private workspace for each txn.

→ Any object read is copied into workspace.

→ Modifications are applied to workspace.

When a txn commits, the DBMS compares workspace write set to see whether it conflicts with other txns.

If there are no conflicts, the write set is installed into the “global” database.
OCC PHASES

#1 – Read Phase:
→ Track the read/write sets of txns and store their writes in a private workspace.

#2 – Validation Phase:
→ When a txn commits, check whether it conflicts with other txns.

#3 – Write Phase:
→ If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.
**OCC - EXAMPLE**

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>BEGIN READ R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Example**

BEGIN READ R(A)
READ R(A)
VALIDATE
WRITE
COMMIT

BEGIN READ R(A)
VALIDATE
WRITE
COMMIT

TIME 37
OCC - EXAMPLE

**Schedule**

- $T_i$
  - BEGIN
  - READ
  - R(A)
  - W(A)
  - VALIDATE
  - WRITE
  - COMMIT

- $T_j$
  - BEGIN
  - READ
  - R(A)
  - VALIDATE
  - WRITE
  - COMMIT

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Time**

- $T_i$ to $T_j$
OCC - EXAMPLE

Schedule

\begin{array}{|c|c|}
\hline
T_i & T_j \\
\hline
\text{BEGIN} & \text{BEGIN} \\
\text{READ} & \text{READ} \\
R(A) & R(A) \\
\hline
\text{VALIDATE} & \text{VALIDATE} \\
\text{WRITE} & \text{WRITE} \\
\text{COMMIT} & \text{COMMIT} \\
\hline
\end{array}

Database

\begin{array}{|c|c|c|}
\hline
\text{Object} & \text{Value} & \text{W-TS} \\
\hline
A & 123 & 0 \\
- & - & - \\
\hline
\end{array}

T_1 Workspace

\begin{array}{|c|c|c|}
\hline
\text{Object} & \text{Value} & \text{W-TS} \\
\hline
A & 123 & 0 \\
- & - & - \\
\hline
\end{array}
OCC - EXAMPLE

Schedule

\[ T_i \quad T_j \]

BEGIN
\[ \text{READ} \]
R(A)
\[ \text{W(A)} \]
\[ \text{VALIDATE} \]
\[ \text{WRITE} \]
COMMIT

BEGIN
\[ \text{READ} \]
R(A)
\[ \text{VALIDATE} \]
\[ \text{WRITE} \]
COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T_1 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T_2 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
OCC - EXAMPLE

Schedule

BEGIN
READ R(A)
VALIDATE
WRITE W(A)
COMMIT

T_i

BEGIN
READ R(A)
VALIDATE
WRITE W(A)
COMMIT

T_j

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T_1 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T_2 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

TIME

EXAMPLE

BEGIN
READ R(A)
VALIDATE
WRITE W(A)
COMMIT

BEGIN
READ R(A)
VALIDATE
WRITE W(A)
COMMIT
OCC - EXAMPLE

Schedule

\[
\begin{align*}
T_i & : & \text{BEGIN} & \text{READ} & R(A) & \text{VALIDATE} & \text{WRITE} & \text{COMMIT} \\
T_j & : & \text{BEGIN} & \text{READ} & R(A) & \text{VALIDATE} & \text{WRITE} & \text{COMMIT}
\end{align*}
\]

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T2 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\[TS(T_j) = 1\]
OCC - EXAMPLE

Schedule

BEGIN
READ R(A)
W(A)
VALIDATE
WRITE
COMMIT

TS(T_j) = 1

T_i

T_j

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T_1 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

T_2 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>
**OCC – EXAMPLE**

**Schedule**

\[ T_i \quad T_j \]

- **BEGIN**: 
  - **READ**: \( R(A) \)
  - **W(A)**
  - **VALIDATE**: 
  - **WRITE**: 
  - **COMMIT**

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**T_1 Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**TIME**

- \( TS(T_j)=1 \)
OCC - EXAMPLE

Schedule

\[ T_i \quad T_j \]

BEGIN
\begin{align*}
& \textit{READ} R(A) \\
& \textit{W(A)} \\
& \textit{VALIDATE} \\
& \textit{WRITE} \\
& \textit{COMMIT}
\end{align*}

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

T1 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
OCC - EXAMPLE

Schedule

\begin{align*}
&\text{BEGIN} \\
&\text{READ} \ R(A) \\
&W(A) \\
&\text{VALIDATE} \\
&W(A) \\
&\text{WRITE} \\
&\text{COMMIT} \\
&\text{BEGIN} \\
&\text{READ} \ R(A) \\
&\text{VALIDATE} \\
&W(A) \\
&\text{WRITE} \\
&\text{COMMIT} \\
\end{align*}

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\(\text{TS}(T_j) = 1\)

T₁ Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>\infty</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
OCC - EXAMPLE

**Schedule**

- \(T_i\)
  - BEGIN
  - READ \(R(A)\)
  - \(W(A)\)
  - VALIDATE
  - WRITE
  - COMMIT
- \(T_j\)
  - BEGIN
  - READ \(R(A)\)
  - \(\text{VALIDATE}\)
  - WRITE

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>0</td>
</tr>
</tbody>
</table>

**T_1 Workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Time**

- \(TS(T_i) = 2\)
- \(TS(T_j) = 1\)
OCC - EXAMPLE

Schedule

BEGIN
READ R(A)
W(A)
VALIDATE
WRITE
COMMIT

BEGIN
READ R(A)
VALIDATE
WRITE

Database

Object | Value | W-TS
---|---|---
A | 456 | 2
-
-

T₁ Workspace

Object | Value | W-TS
---|---|---
A | 456 | ∞
-
-

TS(T₁)=2

TS(T₂)=1
OCC – 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.

→ We can ignore for now what happens if a txn reads/writes tuples via indexes.
OCC: THREE PHASES

When to assign the transaction number? At the end of the read phase.

1. **READ** phase: Read and write objects, making local copies.
2. **VALIDATION** Phase: Check for serializable schedule-related anomalies.
3. **WRITE** Phase: It is safe. Write the local objects, making them permanent.
OCC: VALIDATION \((T_i < T_j)\)

Case 1: \(T_i\) completes its write phase before \(T_j\) starts its read phase.

No conflict as all of \(T_i\)'s actions happen before \(T_j\)'s.
OCC: VALIDATION (T_i < T_j)

Case 2: T_i completes its write phase before T_j starts its write phase.

\[
\text{Check that the write set of } T_i \text{ does not intersect the read set of } T_j, \text{ namely: } \text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset
\]
OCC: VALIDATION \((T_i < T_j)\)

Case 3: \(T_i\) completes its \textbf{read} phase before \(T_j\) completes its \textbf{read} phase.

Check that the write set of \(T_i\) does not intersect the read or write sets of \(T_j\), namely:

\[
\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset
\]

\[
\text{AND } \text{WriteSet}(T_i) \cap \text{WriteSet}(T_j) = \emptyset
\]
### OCC: VALIDATION ($T_i < T_j$)

<table>
<thead>
<tr>
<th>Case</th>
<th>Event Sequence</th>
<th>$R \rightarrow W$</th>
<th>$W \rightarrow R$</th>
<th>$W \rightarrow W$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case 1</strong></td>
<td>$T_i$ → $V$ → $W$ → $T_j$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Case 2</strong></td>
<td>$T_i$ → $V$ → $W$ → $T_j$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **WriteSet($T_i$) \ ∩ \ ReadSet($T_j$) = Ø**
- **WriteSet($T_i$) \ ∩ \ ReadSet($T_j$) = Ø**

<table>
<thead>
<tr>
<th>Case 3</th>
<th>Event Sequence</th>
<th>$R \rightarrow W$</th>
<th>$W \rightarrow R$</th>
<th>$W \rightarrow W$</th>
</tr>
</thead>
</table>

- **WriteSet($T_i$) \ ∩ \ ReadSet($T_j$) = Ø**
- **WriteSet($T_i$) \ ∩ \ WriteSet($T_j$) = Ø**
OCC – WRITE PHASE

Propagate changes in the txn’s write set to database to make them visible to other txns.

Serial Commits:
→ Use a global latch to limit a single txn to be in the Validation/Write phases at a time.

Parallel Commits:
→ Use fine-grained write latches to support parallel Validation/Write phases.
→ Txns acquire latches in primary key order to avoid deadlocks.
OCC – OBSERVATIONS

OCC works well when the # of conflicts is low:
→ All txns are read-only (ideal).
→ Txns access disjoint subsets of data.

If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.
OCC – PERFORMANCE ISSUES

High overhead for copying data locally.

Validation/Write phase bottlenecks.

Aborts are more wasteful than in 2PL because they only occur after a txn has already executed.
DYNAMIC DATABASES

Recall that so far, we have only dealt with transactions that read and update existing objects in the database.

But now if txns perform insertions, updates, and deletions, we have new problems...
THE PHANTOM PROBLEM

Schedule

BEGIN
SELECT COUNT(age) FROM people
WHERE status='lit'

BEGIN
SELECT COUNT(age) FROM people
WHERE status='lit'

INSERT INTO people (age=30, status='lit')

COMMIT

99

100

CREATE TABLE people (
id SERIAL,
name VARCHAR,
age INT,
status VARCHAR
);

TIME
OOPS?

How did this happen?

→ Because $T_1$ locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.
THE PHANTOM PROBLEM

Approach #1: Re-Execute Scans
→ Run queries again at commit to see whether they produce a different result to identify missed changes.

Approach #2: Predicate Locking
→ Logically determine the overlap of predicates before queries start running.

Approach #3: Index Locking
→ Use keys in indexes to protect ranges.
RE-EXECUTE SCANS

The DBMS tracks the **WHERE** clause for all queries that the txn executes.

→ Retain the scan set for every range query in a txn.

Upon commit, re-execute just the scan portion of each query and check whether it generates the same result.

→ Example: Run the scan for an **UPDATE** query but do not modify matching tuples.
PREDICATE LOCKING

Proposed locking scheme from System R.

→ Shared lock on the predicate in a WHERE clause of a SELECT query.

→ Exclusive lock on the predicate in a WHERE clause of any UPDATE, INSERT, or DELETE query.

It is rarely implemented in systems; an example of a system that uses it is HyPer (precision locking).
PREDICATE LOCKING

SELECT COUNT(age) FROM people WHERE status='lit'

INSERT INTO people VALUES (age=30, status='lit')

Records in Table "people"

status='lit'
age=30 ∧ status='lit'
INDEX LOCKING SCHEMES

Key-Value Locks
Gap Locks
Key-Range Locks
Hierarchical Locking
KEY-VALUE LOCKS

Locks that cover a single key-value in an index.

Need “virtual keys” for non-existent values.

*B+Tree Leaf Node*

10  12  14  16

Key [14, 14]
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node

```
10 {Gap} 12 {Gap} 14 {Gap} 16
```

Gap (14, 16)
KEY-RANGE LOCKS

A txn takes locks on ranges in the key space.

→ Each range is from one key that appears in the relation, to the next that appears.

→ Define lock modes so conflict table will capture commutativity of the operations available.
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

→ Need “virtual keys” for artificial values (infinity)

**B+Tree Leaf Node**

- **Prior Key** (12, 14)
- **Next Key** (14, 16)
HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

→ Reduces the number of visits to lock manager.
LOCKING WITHOUT AN INDEX

If there is no suitable index, then to avoid phantoms the txn must obtain:

→ A lock on every page in the table to prevent a record’s `status='lit'` from being changed to `lit`.

→ The lock for the table itself to prevent records with `status='lit'` from being added or deleted.
WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency 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 Reads
→ Unrepeatable Reads
→ Phantom Reads
ISOLATION LEVELS

**SERIALIZABLE**: No phantoms, all readsrepeatable, nodirty reads.

**REPEATABLE READS**: Phantoms may happen.

**READ COMMITTED**: Phantoms and unrepeatabled reads may happen.

**READ UNCOMMITTED**: All of them may happen.
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 (Low → High)
# ISOLATION LEVELS

<table>
<thead>
<tr>
<th></th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERIALIZABLE</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>REPEATABLE READ</strong></td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>READ COMMITTED</strong></td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>READ UNCOMMITTED</strong></td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
</tbody>
</table>
ISOLATION LEVELS

**SERIALIZABLE**: Obtain all locks first; plus index locks, plus strong strict 2PL.

**REPEATABLE READS**: Same as above, but no index locks.

**READ COMMITTED**: Same as above, but S locks are released immediately.

**READ UNCOMMITTED**: Same as above but allows dirty reads (no S locks).
SQL-92 ISOLATION LEVELS

You set a txn’s isolation level **before** you execute any queries in that txn.

Not all DBMS support all isolation levels in all execution scenarios → Replicated Environments

The default depends on implementation…

```
SET TRANSACTION ISOLATION LEVEL <isolation-level>;
```

```
BEGIN TRANSACTION ISOLATION LEVEL <isolation-level>;
```
## ISOLATION LEVELS

<table>
<thead>
<tr>
<th>Database</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>CURSOR STABILITY</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Google Spanner</td>
<td>SERIALIZABLE</td>
<td>STRICT SERIALIZABLE</td>
</tr>
<tr>
<td>MSFT SQL Server</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MySQL</td>
<td>REPEATABLE READS</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Oracle</td>
<td>READ COMMITTED</td>
<td>SNAPSHOT ISOLATION</td>
</tr>
<tr>
<td>PostgreSQL</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>
<tr>
<td>YugaByte</td>
<td>SNAPSHOT ISOLATION</td>
<td>SERIALIZABLE</td>
</tr>
</tbody>
</table>
DATABASE ADMIN SURVEY

What isolation level do transactions execute at on this DBMS?

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th># of Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>10, 8, 4</td>
</tr>
<tr>
<td>Read Committed</td>
<td>12, 6, 26, 22</td>
</tr>
<tr>
<td>Cursor Stability</td>
<td>12, 10</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>10, 12, 5, 2</td>
</tr>
<tr>
<td>Snapshot Isolation</td>
<td>11, 3, 3, 5</td>
</tr>
<tr>
<td>Serializable</td>
<td>8, 11, 2, 0</td>
</tr>
</tbody>
</table>
CONCLUSION

Every concurrency control can be broken down into the basic concepts that have been described in the last two lectures.

Every protocol has pros and cons.
"ANSI SQL-92 ... defines Isolation Levels in terms of phenomena: Dirty Reads, Non-Repeateable Reads, and Phantoms. ... these phenomena and the ANSI SQL definitions fail to characterize several popular isolation levels, including the standard locking implementations of the levels. Investigating the ambiguities of the phenomena leads to clearer definitions; in addition new phenomena that better characterize isolation types are introduced. An important multiversion isolation type, Snapshot Isolation, is defined."
NEXT CLASS

Multi-Version Concurrency Control