CSE 707: Modern database system seminar

Week 12 - Main-memory DB on many cores

Zhuoyue Zhao
11/17/2021
Today’s reading

Scalability challenges in many-core systems

- DBMS in the past
  - Disk-based
  - One or tens of cores

- DBMS now and in the future
  - In-memory
  - Many core (hundreds or even thousands)

- Can the DBMS design still scale to hundreds of cores?
  - This paper studies concurrency control in particular.
Concurrency control

- Two-phase locking (2-PL)
  - w/ Deadlock Detection (DL_DETECTION)
  - w/ Non-waiting Deadlock Prevention (NO_WAIT)
  - w/ Waiting Deadlock Prevention (WAIT_DIE)

- Timestamp ordering (T/O)
  - basic T/O (TIMESTAMP)
  - Multi-Version Concurrency Control (MVCC)
  - Optimistic Concurrency Control (OCC)
  - T/O with Partition-Level Locking (H-STORE)
Two-Phase Locking (2-PL)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y  
T2: transfer $20 from y to x

Read/write locks on tuples or on higher-granularity items

Rule 1: no conflicting locks (e.g., no two threads simultaneously holding a write lock on the same tuple)
Rule 2: no new lock once a transaction starts to releasing locks

T1: R(x) R(y) W(x) W(y) Commit
Two-Phase Locking (2-PL)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
<td>T1: transfer $10 from x to y</td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
<td>T2: transfer $20 from y to x</td>
</tr>
</tbody>
</table>

• Shared/exclusive locks on tuples or on higher-granularity items
  • R(x): requires shared/exclusive lock on x
  • W(x): requires exclusive lock on x
• Rule 1: no conflicting locks
  • e.g., no two threads simultaneously holding a write lock on the same tuple
• Rule 2: no new lock once a transaction starts to releasing locks

T1: R(x) R(y) W(x) W(y) commit
Two-Phase Locking (2-PL)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks

T1: R(x) R(y) W(x) W(y) commit
### Two-Phase Locking (2-PL)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- **Rule 1**: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- **Rule 2**: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: R(x) R(y) W(x) W(y) commit
Two-Phase Locking (2-PL)

<table>
<thead>
<tr>
<th></th>
<th>$90</th>
<th>$200</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y  
T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: R(x) R(y) W(x) W(y) commit
Two-Phase Locking (2-PL)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90</td>
</tr>
<tr>
<td>y</td>
<td>$210</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: R(x) R(y) W(x) W(y) commit
Two-Phase Locking (2-PL)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90</td>
</tr>
<tr>
<td>y</td>
<td>$210</td>
</tr>
</tbody>
</table>

- T1: transfer $10 from x to y
- T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: R(x) R(y) W(x) W(y) commit
Deadlocks in 2-PL

T1: R(x)  R(y)  W(x)
T2:       R(y)  R(x)  W(y)

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: transfer $10 from x to y
T2: transfer $20 from y to x
Deadlocks in 2-PL

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$100</td>
<td>$200</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: R(x) R(y) → W(x)
T2: R(y) R(x) → W(y)

Deadlock!
Deadlocks Detection

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

\[
\begin{align*}
T1: & \quad R(x) \quad R(y) \quad W(x) \\
T2: & \quad R(y) \quad R(x) \quad W(y)
\end{align*}
\]

Deadlock! ⇔ a cycle in the wait-for graph.

Abort one of T1 and T2 will resolve the deadlock.
Deadlocks Detection

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>$100</th>
<th>T1: transfer $10 from x to y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td>$200</td>
<td>T2: transfer $20 from y to x</td>
</tr>
</tbody>
</table>

- Shared/exclusive locks on tuples or on higher-granularity items
  - R(x): requires shared/exclusive lock on x
  - W(x): requires exclusive lock on x
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: R(x) R(y) W(x)
T2: R(y) R(x) W(y)-abort

Deadlock! ⇔ a cycle in the wait-for graph.

*Abort one of T1 and T2 will resolve the deadlock.*
**Deadlocks Detection**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>$90</th>
<th>T1: transfer $10 from x to y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td>$210</td>
<td>T2: transfer $20 from y to x</td>
</tr>
</tbody>
</table>

• Shared/exclusive locks on tuples or on higher-granularity items
  • R(x): requires shared/exclusive lock on x
  • W(x): requires exclusive lock on x

• Rule 1: no conflicting locks
  • e.g., no two threads simultaneously holding a write lock on the same tuple

• Rule 2: no new lock once a transaction starts to releasing locks
  • Growing phase: obtains as many locks as needed
  • Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

T1: \( R(x) \ R(y) \ W(x) \ W(y) \) commit
T2: \( R(y) \ R(x) \ W(y) \)-abort

Deadlock! \( \Leftrightarrow \) a cycle in the wait-for graph.

*Abort one of T1 and T2 will resolve the deadlock.*
Deadlocks Detection

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$90$</td>
<td>T1: transfer $10$ from $x$ to $y$</td>
</tr>
<tr>
<td>$y$</td>
<td>$210$</td>
<td>T2: transfer $20$ from $y$ to $x$</td>
</tr>
</tbody>
</table>

- Shared/exclusive locks on tuples or on higher-granularity items
  - $R(x)$: requires shared/exclusive lock on $x$
  - $W(x)$: requires exclusive lock on $x$
- Rule 1: no conflicting locks
  - e.g., no two threads simultaneously holding a write lock on the same tuple
- Rule 2: no new lock once a transaction starts to releasing locks
  - Growing phase: obtains as many locks as needed
  - Shrinking phase: releasing locks only when we no longer need new locks (e.g., at commit/abort)

$T_1$: $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit
$T_2$: $R(y)$ $R(x)$ $W(y)$-abort

Deadlock! ⇔ a cycle in the wait-for graph.

Abort one of $T_1$ and $T_2$ will resolve the deadlock.
Deadlock prevention

- Alternative: prevent any cycle in the wait-for graph

- No wait
  - T aborts immediately when T tries to obtain a conflicting lock

- Wait die
  - When T tries to obtain a conflicting lock and doing so would make it wait for T'
    - T aborts if T starts after T'
    - T waits if T starts before T'
Bottlenecks for 2-PL

- Deadlock detection
  - Contention on the central wait-for-graph structure
  - partitioning wait-for graph across cores mitigates the issue
Bottlenecks for 2-PL

- Deadlock detection
  - Contention on the central wait-for-graph structure
  - Partitioning wait-for graph across cores mitigates the issue

- Lock thrashing
  - Threads block other conflicting threads until it commits/aborts

![Figure 4: Lock Thrashing](image) – Results for a write-intensive YCSB workload using the DL_DETECT scheme without deadlock detection. Each transaction acquires locks in their primary key order.
Bottlenecks for 2-PL

- **Deadlock detection**
  - Contention on the central wait-for-graph structure
  - Partitioning wait-for graph across cores mitigates the issue

- **Lock thrashing**
  - Threads block other conflicting threads until it commits/aborts
  - Abort after timeout

![Graph showing waiting vs aborting](image)

**Figure 5: Waiting vs. Aborting** – Results for DL_DETECT with varying timeout threshold running high contention YCSB ($\theta=0.8$) at 64 cores.
Concurrent control

- Two-phase locking (2-PL)
  - w/ Deadlock Detection (DL_DETECTION)
  - w/ Non-waiting Deadlock Prevention (NO_WAIT)
  - w/ Waiting Deadlock Prevention (WAIT_DIE)

- Timestamp ordering (T/O)
  - basic T/O (TIMESTAMP)
  - Multi-Version Concurrency Control (MVCC)
  - Optimistic Concurrency Control (OCC)
  - T/O with Partition-Level Locking (H-STORE)
**Basic Timestamp Ordering (T/O)**

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$10$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

T1: transfer $10$ from x to y  
T2: transfer $20$ from y to x

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
    - Cache $x$ for repeatable read
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>200</td>
<td>10</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

($t_{T1} = 20$) $T1 : R(x) \ R(y) \ W(x) \ W(y)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

- T1: transfer $10$ from x to y
- T2: transfer $20$ from y to x

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, T must abort if $t_T < t_w(x)$
  - For each $W(x)$, T must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20)$ T1 : $R(x)$ R(y) W(x) W(y) commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$20$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20)$ T1: $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$20$</td>
</tr>
</tbody>
</table>

T1: transfer $10$ from $x$ to $y$
T2: transfer $20$ from $y$ to $x$

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20)$ T1: $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$210$</td>
<td>$20$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20)$ T1: $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$210$</td>
<td>$20$</td>
</tr>
</tbody>
</table>

T1: transfer $10$ from x to y
T2: transfer $20$ from y to x

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20)$ T1 : $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th>x</th>
<th>$100</th>
<th>20</th>
<th>10</th>
<th>T1: transfer $10 from x to y</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$200</td>
<td>20</td>
<td>10</td>
<td>T2: transfer $20 from y to x</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20) \quad T1 : R(x) \quad R(y) \quad W(x)$

$(t_{T2} = 30) \quad T2 : \quad R(x)$
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>y</td>
<td>200</td>
<td>20</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, T must abort if $t_T < t_w(x)$
  - For each $W(x)$, T must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$t_{T1} = 20$ T1: $R(x)$ $R(y)$ $W(x)$
$t_{T2} = 30$ T2: $R(x)$
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T1} = 20) \quad T1 : R(x) \quad R(y) \quad W(x)$
$(t_{T2} = 30) \quad T2 : \quad R(x)$

T1: transfer $10 from x to y
T2: transfer $20 from y to x
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$30$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$20$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

$(t_{T_1} = 20)$  
$T_1: R(x)$  
$R(y)$  
$W(x)$ abort: $t_{T_1} < t_r(x)$

$(t_{T_2} = 30)$  
$T_2: R(x)$
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$10$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

T1: transfer $10$ from x to y
T2: transfer $20$ from y to x

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each R($x$), T must abort if $t_T < t_w(x)$
  - For each W($x$), T must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

- Recoverability: T must not commit until all its dependencies commit
## Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>( t_r(x) )</th>
<th>( t_w(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90</td>
<td>20</td>
</tr>
<tr>
<td>y</td>
<td>$210</td>
<td>20</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

- Each tuple \( x \) has a timestamp \( t_x \)
- \( T \) receives a timestamp \( t_T \) when it starts
  - For each \( R(x) \), \( T \) must abort if \( t_T < t_w(x) \)
  - For each \( W(x) \), \( T \) must abort if \( t_T < t_r(x) \lor t_T < t_w(x) \)

- Recoverability: \( T \) must not commit until all its dependencies commit
  
  \((t_{T1} = 20) \ T1 : R(x) \ R(y) \ W(x) \ W(y)\)

  \((t_{T2} = 30) \ T2 : \ R(x) \ R(y) \ W(x) \ W(y)\)
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$110$</td>
<td>$30$</td>
</tr>
<tr>
<td>y</td>
<td>$190$</td>
<td>$30$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

- Recoverability: $T$ must not commit until all its dependencies commit
  - $(t_{T1} = 20)$ $T1: R(x) \ R(y) \ W(x) \ W(y)$
  - $(t_{T2} = 30)$ $T2: R(x) \ R(y) \ W(x) \ W(y)$

T1: transfer $10$ from $x$ to $y$
T2: transfer $20$ from $y$ to $x$
Basic Timestamp Ordering (T/O)

$\begin{array}{c|cc}
  x & t_r(x) & t_w(x) \\
  \hline
  x & $110 & 30 & 30 \\
  y & $190 & 30 & 30 \\
\end{array}$

- T1: transfer $10 from x to y
- T2: transfer $20 from y to x

• Each tuple $x$ has a timestamp $t_x$
• $T$ receives a timestamp $t_T$ when it starts
  • For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  • For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

• Recoverability: $T$ must not commit until all its dependencies commit

$(t_{T1} = 20)$ T1 : $R(x)$ $R(y)$ $W(x)$ $W(y)$
$(t_{T2} = 30)$ T2 : $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$110$</td>
<td>$30$</td>
</tr>
<tr>
<td>y</td>
<td>$190$</td>
<td>$30$</td>
</tr>
</tbody>
</table>

- T1: transfer $10 from x to y
- T2: transfer $20 from y to x

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, T must abort if $t_T < t_w(x)$
  - For each $W(x)$, T must abort if $t_T < t_r(x) \lor t_T < t_w(x)$

- Recoverability: T must not commit until all its dependencies commit
  (38.1) $t_{T1} = 20$ T1 : $R(x)$ $R(y)$ $W(x)$ $W(y)$ abort
  (38.2) $t_{T2} = 30$ T2 : $R(x)$ $R(y)$ $W(x)$ $W(y)$ commit

Cannot happen in any serial schedule where T1 aborts.
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$
    - Thomas write rule
- Recoverability: $T$ must not commit until all its dependencies commit

(t_{T1} = 20) T1 : R(x)                        W(x)
(t_{T2} = 30) T2 : W(x) commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- T receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x) \lor t_T < t_w(x)$
    - **Thomas Write rule**
- Recoverability: $T$ must not commit until all its dependencies commit

$(t_{T1} = 20)$ T1 : $R(x)$
$(t_{T2} = 30)$ T2 : $W(x)$ abort
$(t_{T2} = 30)$ T2 : $W(x)$ commit
Basic Timestamp Ordering (T/O)

<table>
<thead>
<tr>
<th></th>
<th>$t_r(x)$</th>
<th>$t_w(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100$</td>
<td>$20$</td>
</tr>
<tr>
<td>y</td>
<td>$200$</td>
<td>$10$</td>
</tr>
</tbody>
</table>

- Each tuple $x$ has a timestamp $t_x$
- $T$ receives a timestamp $t_T$ when it starts
  - For each $R(x)$, $T$ must abort if $t_T < t_w(x)$
  - For each $W(x)$, $T$ must abort if $t_T < t_r(x)$
    - **Thomas write rule**: ignore $W(x)$ if $t_T < t_w(x)$
- Recoverability: $T$ must not commit until all its dependencies commit

$t_{T1} = 20)$ $T1 : R(x)$  $W(x)$ commit
$t_{T2} = 30)$ $T2 : W(x)$ commit
Optimistic Concurrency Control

- Transactions read/write in a private buffer
- When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

T1 : R(x) R(y) W(x)        W(y)
T2:                         R(x)          R(y) W(x) W(y)

T1’s write buffer

T1’s read set = {}

T2’s write buffer

T2’s read set = {}
Optimistic Concurrency Control

• Transactions read/write in a private buffer
• When a transaction commits,
  • Checks whether there are any conflicts with concurrent transactions’ reads/writes

<table>
<thead>
<tr>
<th>x</th>
<th>$100</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$200</td>
<td>10</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

T1: \( R(x) \ R(y) \ W(x) \ W(y) \)
T2: \( R(x) \ R(y) \ W(x) \ W(y) \)

\( T1 \)’s write buffer
\( T1 \)’s read set = \( \{x,y\} \)

\( T2 \)’s write buffer
\( T2 \)’s read set = \( \{\} \)
Optimistic Concurrency Control

<table>
<thead>
<tr>
<th>t(x)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
</tr>
</tbody>
</table>

Transactions read/write in a private buffer

- When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

T1: transfer $10 from x to y

T2: transfer $20 from y to x

T1: \( R(x) \quad R(y) \quad W(x) \quad W(y) \)
T2: \( R(x) \quad R(y) \quad W(x) \quad W(y) \)

\( T1 \)’s write buffer \( T1 \)’s read set = \{x, y\} \( T2 \)’s write buffer \( T2 \)’s read set = {}
Optimistic Concurrency Control

Transactions read/write in a private buffer
When a transaction commits,
  * Checks whether there are any conflicts with concurrent transactions’ reads/writes

\[
\begin{array}{c|c|c}
\text{x} & $100 & 10 \\
\text{y} & $200 & 10 \\
\end{array}
\]

T1: transfer $10 from x to y
T2: transfer $20 from y to x

T1 : R(x) R(y) W(x) W(y)
T2: R(x) R(y) W(x) W(y)

T1’s write buffer
T1’s read set = \{x,y\}

T2’s write buffer
T2’s read set = \{x\}
### Optimistic Concurrency Control

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>$100</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$200</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

- Transactions read/write in a private buffer
- When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

\[
\begin{align*}
T_1 &: \text{transfer } $10 \text{ from } x \text{ to } y \\
T_2 &: \text{transfer } $20 \text{ from } y \text{ to } x
\end{align*}
\]

\[
\begin{align*}
T_1 &: R(x) \ R(y) \ W(x) \ W(y) \\
T_2 &: R(x) \ R(y) W(x) W(y)
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|}
\text{T1’s write buffer} & \text{T1’s read set} = \{x,y\} & \text{T2’s write buffer} & \text{T2’s read set} = \{x\} \\
\hline
x & $90 & \text{ } & \text{ } & \text{ } \\
\hline
y & $210 & \text{ } & \text{ } & \text{ }
\end{array}
\]
Optimistic Concurrency Control

Transactions read/write in a private buffer
When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

\[ T1: R(x) \ R(y) \ W(x) \ W(y) \]
\[ T2: R(x) \ R(y) \ W(x) \ W(y) \]

\[ T1’s \ write \ buffer \]
\[ T1’s \ read \ set = \{x, y\} \]

\[ T2’s \ write \ buffer \]
\[ T2’s \ read \ set = \{x\} \]

\[ \begin{array}{c|cc}
\text{t(x)} & x & y \\
\hline
x & $100 & 10 \\
y & $200 & 10 \\
\end{array} \]

T1: transfer $10 from x to y
T2: transfer $20 from y to x
**Optimistic Concurrency Control**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$100</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
<td>10</td>
</tr>
</tbody>
</table>

- Transactions read/write in a private buffer
- When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>R(x) R(y) W(x) W(y)</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>R(x) R(y) W(x) W(y)</td>
<td></td>
</tr>
</tbody>
</table>

**T1’s write buffer**

**T1’s read set** $= \{x,y\}$

**T2’s write buffer**

**T2’s read set** $= \{x\}$
Optimistic Concurrency Control

- Transactions read/write in a private buffer
- When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

<table>
<thead>
<tr>
<th>x</th>
<th>$100</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$200</td>
<td>10</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: transfer $20 from y to x

<table>
<thead>
<tr>
<th>t(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

T1: R(x) R(y) W(x) W(y)
T2: R(x) R(y) W(x) W(y)

T1’s write buffer: [x, y]
T1’s read set = {x, y}
T2’s write buffer: [x, y]
T2’s read set = {x, y}

<table>
<thead>
<tr>
<th>x</th>
<th>$90</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>$120</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$180</td>
</tr>
</tbody>
</table>
Optimistic Concurrency Control

• Transactions read/write in a private buffer
• When a transaction commits,
  • Checks whether there are any conflicts with concurrent transactions’ reads/writes

\[
\begin{array}{c|c|c}
  t(x) & \text{Validation & commit } & t_{T_1} = 20 \\
\hline
  x & \text{R}(x) & \text{W}(x) \\
  y & \text{R}(y) & \text{W}(y) \\
\end{array}
\]

\[
\begin{array}{c|c}
  \text{T1's write buffer} & \text{T1's read set } = \{x,y\} \\
\hline
  x & \$90 \\
  y & \$210 \\
\end{array}
\]

\[
\begin{array}{c|c}
  \text{T2's write buffer} & \text{T2's read set } = \{x,y\} \\
\hline
  x & \$120 \\
  y & \$180 \\
\end{array}
\]

T1: transfer $10 from x to y
T2: transfer $20 from y to x
Optimistic Concurrency Control

- Transactions read/write in a private buffer
- When a transaction commits,
  - Checks whether there are any conflicts with concurrent transactions’ reads/writes

T1: transfer $10 from x to y
T2: transfer $20 from y to x

<table>
<thead>
<tr>
<th>x</th>
<th>$90</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$210</td>
<td>20</td>
</tr>
</tbody>
</table>

\[ t(x) \]

_Validation & commit @ \( t_{T1} = 20 \)
_Validation & abort @ \( t_{T2} = 30 \)
Multi-Version Concurrency Control

- Writes create new versions of tuples
  - tagged with a creation timestamp
- Reads access a visible version – does not reject read operation that arrives late

\[
\begin{array}{c|c|c}
\text{x} & \$100 & 10 \\
\hline
\text{y} & \$200 & 10 \\
\end{array}
\]

\( t_{T1} = 20 \)  

T1: transfer $10 from x to y

T2: withdraw $300 from y

\((t_{T1} = 20)\)  

T1: \( R(x) \ R(y) \ W(x) \)  

W(y) commit

\((t_{T2} = 15)\)  

T2: \( R(y) \ W(y) \) commit
Multi-Version Concurrency Control

- Writes create new versions of tuples
  - tagged with a creation timestamp
- Reads access a visible version – does not reject read operation that arrives late

\[
(t_{T_1} = 20) \quad T_1 : R(x) \ W(x) \quad R(y) \ W(y) \ \text{commit}
\]

\[
(t_{T_2} = 15) \quad T_2 : \quad R(y) \ W(y) \ \text{commit}
\]

<table>
<thead>
<tr>
<th></th>
<th>$100</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>$200</td>
<td>10</td>
</tr>
</tbody>
</table>

T1: transfer $10 from x to y
T2: withdraw $300 from y
Multi-Version Concurrency Control

- Writes create new versions of tuples
  - tagged with a creation timestamp
- Reads access a visible version – does not reject read operation that arrives late

\[ (t_{T1} = 20) \quad T1 : R(x) W(x) \quad R(y) \quad W(y) \quad \text{commit} \]
\[ (t_{T2} = 15) \quad T2 : \quad R(x) \quad R(y) \quad W(y) \quad \text{commit} \]

T1: transfer $10 from x to y
T2: withdraw $300 from y
Multi-Version Concurrency Control

- Writes create new versions of tuples
  - tagged with a creation timestamp
- Reads access a visible version – does not reject read operation that arrives late

\[
\begin{align*}
(t_{T1} = 20) & \quad T1 : R(x) \ W(x) \\
(t_{T2} = 15) & \quad T2 : R(x) \ R(y) \ W(y) \ commit
\end{align*}
\]

T1: transfer $10 from x to y
T2: withdraw $300 from y
Multi-Version Concurrency Control

- Writes create new versions of tuples
  - tagged with a creation timestamp
- Reads access a visible version – does not reject read operation that arrives late

\[
(t_{T1} = 20) \quad T1 : \text{R(x) W(x)} \quad \text{R(y) W(y) commit}
\]
\[
(t_{T2} = 15) \quad T2 : \text{R(x) R(y) W(y) commit}
\]

T1: transfer $10 from x to y
T2: withdraw $300 from y
## Multi-Version Concurrency Control

- **Writes** create new versions of tuples
  - tagged with a creation timestamp
- **Reads** access a visible version – does not reject read operation that arrives late

\[
\begin{align*}
(t_{T1} = 20) & \quad T1 : R(x) \ W(x) \quad R(y) \ W(y) \quad \text{commit} \\
(t_{T2} = 15) & \quad T2 : R(x) \ R(y) \ W(y) \quad \text{commit}
\end{align*}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$90</td>
<td>20</td>
<td>$100</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>-$90</td>
<td>20</td>
<td>-$100</td>
<td>15</td>
</tr>
</tbody>
</table>

$100 - $10 = $90$

$200 - 15 = 10$

T1: transfer $10 from x to y
T2: withdraw $300 from y
Bottlenecks for T/O methods

- Timestamp allocation
  - Mutex – very expensive, barely scales
  - Atomic add
    - Does not scale beyond 20 cores due to expensive cache coherence protocol
  - Batched atomic add
    - Scales to < 100 cores
    - High abort rate with high contention
  - Hardware/software synchronized clock
    - Special hardware needed
  - Hardware counter
    - Not available in existing CPU architecture

Figure 6: Timestamp Allocation Micro-benchmark – Throughput measurements for different timestamp allocation methods.

Figure 7: Timestamp Allocation – Throughput of the YCSB workload using TIMESTAMP with different timestamp allocation methods.
Performance evaluation

- YCSB/TPC-C
- Scalability and sensitivity to data skewness
Read-only workload

**Figure 8: Read-only Workload** – Results for a read-only YCSB workload.
Write-intensive workload

Figure 9: Write-Intensive Workload (Medium Contention) – Results for YCSB workload with medium contention ($\theta = 0.6$).
Level of contention

Figure 11: Write-Intensive Workload (Variable Contention) – Results for YCSB workload with varying level of contention on 64 cores.
Next time

- No class next week. Happy thanksgiving!

- We’ll meet again on 12/1.
  - Presenter will be Isha Narula.