Optimistic Concurrency Control
Concurrency Control: Two approaches

- **Pessimistic**: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- **Optimistic**: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time
Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon’s Dynamo
- Preferable than pessimistic when conflicts are expected to be rare
  - But still need to ensure conflicts are caught!
First cut approach

• Most basic approach
  • Write and read objects at will
  • Check for serial equivalence at commit time
  • If abort, roll back updates made
  • An abort may result in other transactions that read dirty data, also being aborted
    • Any transactions that read from *those* transactions also now need to be aborted

 Почасти Abort
Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction $T$, both are true:
  1. $T$’s write to object $O$ allowed only if transactions that have read or written $O$ had lower ids than $T$.
  2. $T$’s read to object $O$ is allowed only if $O$ was last written by a transaction with a lower id than $T$.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
- Never results in a deadlock! Older transaction never waits on newer ones.
Timestamped ordering: per-object state

- Committed value.
- Transaction id (timestamp) that wrote the committed value.
- Read timestamps (RTS): List of transaction ids (timestamps) that have read the committed value.
- Tentative writes (TW): List of tentative writes sorted by the corresponding transaction ids (timestamps).
  - Timestamped versions of the object.
# Timestamped ordering rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
</tr>
</thead>
</table>
| 1. **write** | **read** | $T_c$ must not **write** an object that has been **read** by any $T_i$ where $T_i > T_c$  
This requires that $T_c \geq$ the maximum read timestamp of the object. |
| 2. **write** | **write** | $T_c$ must not **write** an object that has been **written** by any $T_i$ where $T_i > T_c$  
This requires that $T_c >$ write timestamp of the committed object. |
| 3. **read** | **write** | $T_c$ must not **read** an object that has been **written** by any $T_i$ where $T_i > T_c$  
This requires that $T_c >$ write timestamp of the committed object. |
Timestamped ordering: write rule

Transaction $T_c$ requests a write operation on object $D$
  if ($T_c \geq$ max. read timestamp on $D$
      && $T_c >$ write timestamp on committed version of $D$)
      
      Perform a tentative write on $D$:
      
      If $T_c$ already has an entry in the TW list for $D$, update it.
      Else, add $T_c$ and its write value to the TW list.

else

  abort transaction $T_c$

  //too late; a transaction with later timestamp has already read or written the object.
Timestamped ordering: write rule

(a) $T_3$ write

Before

After

(b) $T_3$ write

Before

After

(c) $T_3$ write

Before

After

(d) $T_3$ write

Before

After

Key:

Committed

Tentative

$T_1 < T_2 < T_3 < T_4$

Read timestamps not shown in this example. (assume zero reads)
Timestamped ordering: read rule

Transaction $T_c$ requests a read operation on object $D$

    if ($T_c > \text{write timestamp on committed version of } D$) {
        $D_s = \text{version of } D \text{ with the maximum write timestamp that is } \leq T_c$
        \text{//search across the committed timestamp and the TW list for object } D.
        if ($D_s$ is committed)
            read $D_s$ and add $T_c$ to RTS list (if not already added)
        else
            if $D_s$ was written by $T_c$, simply read $D_s$
            else
                wait until the transaction that wrote $D_s$ is committed or aborted, and
                reapply the read rule.
                \text{// if the transaction is committed, } T_c \text{ will read its value after the wait.}
                \text{// if the transaction is aborted, } T_c \text{ will read the value from an older transaction.}
    } else
        abort transaction $T_c$
        \text{//too late; a transaction with later timestamp has already written the object.}
Timestamped ordering: read rule

(a) $T_3$ read

(b) $T_3$ read

(c) $T_3$ read

(d) $T_3$ read

Key:

- Committed
- Tentative

$T_1 < T_2 < T_3 < T_4$
Timestamped ordering: committing

- Suppose $T_4$ is ready to commit.
- Must wait until $T_3$ commits or aborts.

- When a transaction is committed, the committed value of the object and associated timestamp are updated, and the corresponding write is removed from TW list.
Lost Update Example with Timestamped Ordering

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
<th>ABC123: state committed value = 10 committed timestamp = 0 RTS: TV:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{getSeats}(ABC123)$;</td>
<td>$x = \text{getSeats}(ABC123)$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if($x &gt; 1$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x = x - 1$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write($x$, ABC123);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x = x - 1$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write($x$, ABC123);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
Next Example with Timestamped Ordering

**Transaction T1**

- \( x = \text{getSeats}(\text{ABC123}) \);
- \( y = \text{getSeats}(\text{ABC789}) \);
- write(x-5, ABC123);
- write(y+5, ABC789);
- commit

**Transaction T2**

- \( x = \text{getSeats}(\text{ABC123}) \);
- \( y = \text{getSeats}(\text{ABC789}) \);
- print("Total:" x+y);
- commit

**ABC123**: state
- committed value = 10
- committed timestamp = 0
- RTS:
- TW:

**ABC789**: state
- committed value = 5
- committed timestamp = 0
- RTS:
- TW:
Concurrency Control: Summary

• How to prevent transactions from affecting one another?
• Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
• Target serial equivalence.
• Two approaches:
  • Pessimistic concurrency control: locking based.
    • read-write locks with two-phase locking and deadlock detection.
  • Optimistic concurrency control: abort if too late.
    • timestamped ordering.