(Pessimistic) Timestamp Ordering

Another approach to concurrency control:

- Assign a *timestamp* $ts(T)$ to transaction $T$ at the moment it starts
- Using Lamport's timestamps: total order is given. In distributed systems, a timestamp consists of a pair $<\text{local timestamp}, \text{server-id}>$
- $ts(T)$ is assigned with each operation done by $T$. This allows a total order on all read and write events
- Each data item $x$ is assigned with two timestamps $ts_{RD}(x)$ and $ts_{WR}(x)$ for read/write operations. The read timestamp is set to $ts(T)$ of the transaction $T$ which most recently read $x$ (analogously: write timestamp)
- If two operations conflict, the data manager processes the one with the lower timestamp first
- A transaction's request to write an object is valid only if that object was last read and written by *earlier* transactions. A transaction's request to read an object is valid only if that object was last written by an *earlier* transaction.

(Tentative operations: work on own copy, make them permanent when committing.)
## Rules for read and write Operations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Operation 1</th>
<th>Operation 2</th>
<th>Condition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>write</td>
<td>read</td>
<td>( T_c ) must not write an object that has been read by any ( T_i ) where ( T_i &gt; T_c )</td>
<td>This requires that ( T_c &gt; ) the maximum read timestamp of the object.</td>
</tr>
<tr>
<td>2.</td>
<td>write</td>
<td>write</td>
<td>( T_c ) must not write an object that has been written by any ( T_i ) where ( T_i &gt; T_c )</td>
<td>This requires that ( T_c &gt; ) write timestamp of the committed object.</td>
</tr>
<tr>
<td>3.</td>
<td>read</td>
<td>write</td>
<td>( T_c ) must not read an object that has been written by any ( T_i ) where ( T_i &gt; T_c )</td>
<td>This requires that ( T_c &gt; ) write timestamp of the committed object.</td>
</tr>
</tbody>
</table>

### Example:

- \( T \) sends a \( \text{read}(T, x) \) with \( \text{ts}(T) < \text{ts}_{WR}(x) \) \( \Rightarrow \) \( T \) is aborted
- \( T \) sends a \( \text{read}(T, x) \) with \( \text{ts}(T) > \text{ts}_{WR}(x) \) \( \Rightarrow \) read operation is performed, \( \text{ts}_{RD}(x) = \max\{\text{ts}(T), \text{ts}_{RD}(x)\} \)
- \( T \) sends a \( \text{write}(T, x) \) with \( \text{ts}(T) < \text{ts}_{RD}(x) \) \( \Rightarrow \) \( T \) is aborted
- \( T \) sends a \( \text{write}(T, x) \) with \( \text{ts}(T) > \text{ts}_{RD}(x) \) \( \Rightarrow \) no younger operation has read the value; change it, set \( \text{ts}_{WR}(x) = \max\{\text{ts}(T), \text{ts}_{WR}(x)\} \)
Pessimistic Timestamp Ordering

- Let $ts_{RD}(x) = ts_{WR}(x) = ts(T_1)$, where $T_1$ has ended some time before.
- $T_2$ and $T_3$ start concurrently with $ts(T_2) < ts(T_3)$
- Look at $T_2$'s write (a - d) and read (e - h):

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{RD}(x) & \text{ts}_{WR}(x) & \text{ts}(T_2) \\

t_1 & t_1 & t_2 \\
\hline
(a) & \text{Time} \rightarrow & \\
\end{array} \\
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_1 & t_1 & t_2 \\
\hline
(b) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

- Do tentative write

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_1 & t_1 & t_2 \\
\hline
(c) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

- Abort

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_1 & t_3 & t_2 \\
\hline
(d) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_2 & t_3 & t_2 \\
\hline
(e) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

- OK

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_1 & t_1 & t_2 \\
\hline
(f) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

- OK

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_2 & t_3 & t_2 \\
\hline
(g) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

- Abort

\[\begin{array}{c}
\begin{array}{c|c|c|c}
\text{ts}_{WR}(x) & \text{ts}_{RD}(x) & \text{ts}(T_2) \\

t_2 & t_1 & t_2 \\
\hline
(h) & \text{Time} \rightarrow & \\
\end{array}
\end{array}\]

- OK
Write Operations and Timestamps

if \((T_c \geq \text{maximum read timestamp on object } D \&\& \ T_c > \text{write timestamp on committed version of } D)\)

- perform write operation on tentative version of \(D\) with write timestamp \(T_c\)

else /* write is too late */

- Abort transaction \(T_c\)

(a) \(T_3\) write

\[
\begin{array}{c|c}
\text{Before} & \text{After} \\
\hline
T_2 & T_2 \ T_3 \\
\end{array}
\]

Time

(b) \(T_3\) write

\[
\begin{array}{c|c}
\text{Before} & \text{After} \\
\hline
T_1 \ T_2 & T_1 \ T_2 \ T_3 \\
\end{array}
\]

Time

Key:

- \(T_1\) Committed
- \(T_i\) Tentative

(c) \(T_3\) write

\[
\begin{array}{c|c}
\text{Before} & \text{After} \\
\hline
T_1 \ T_4 & T_1 \ T_3 \ T_4 \\
\end{array}
\]

Time

Transaction aborts

(d) \(T_3\) write

\[
\begin{array}{c|c}
\text{Before} & \text{After} \\
\hline
T_4 & T_4 \\
\end{array}
\]

Time

Object produced by transaction \(T_i\)

(with write timestamp \(T_i\))

\(T_1 < T_2 < T_3 < T_4\)
Read Operations and Timestamps

if \( T_c > \text{write timestamp on committed version of } D \) { 
    let \( D_{\text{selected}} \) be the version of \( D \) with the maximum write timestamp \( \leq T_c \)
    if \( \) (\( D_{\text{selected}} \) is committed)
        perform read operation on the version \( D_{\text{selected}} \)
    else
        Wait until the transaction that made version \( D_{\text{selected}} \) commits or aborts
        then reapply the read rule
} else
    Abort transaction \( T_c \)
Transaction Commits with Timestamp Ordering

- When a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - Committed versions of an object must be created in timestamp order
  - The server may sometimes need to wait, but the client need not wait
  - To ensure recoverability, the server will save the ‘waiting to be committed versions’ in permanent storage

- The timestamp ordering algorithm is strict because
  - The read rule delays each read operation until previous transactions that had written the object have committed or aborted
  - Writing the committed versions in order ensures that the write operation is delayed until previous transactions that have written the object have committed or aborted
Remarks on Timestamp Ordering

Concurrency Control

The method avoids deadlocks, but e.g. a transaction seeing a larger timestamp aborts, whereas in locking it could wait or even proceed immediately (Pessimistic ordering)

– Modification known as ‘ignore obsolete write’ rule is an improvement

• If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.

• However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item

– Multiversion timestamp ordering

• Allows more concurrency by keeping multiple committed versions - late read operations need not be aborted
Optimistic Concurrency Control

Third approach to concurrency control:

- Just do your operations without taking care to any operations of other processes. Problems can be thought about later. (*Optimistic* control)
- This should work, because in practice, conflicts are seldom
- Control algorithm:
  - keep track on all data items used by a transaction
  - When committing, all other transactions are checked, if they had modified one of the used data items since the transaction had started.
  - If yes, the transaction is aborted.
- Deadlock free, no process has to wait for locks
- *Disadvantage*: a transaction can fail unnecessarily and forced to run again. In cases of heavy load, this decreases performance heavily
- No real comparison with other approaches are possible: has primarily focused on non-distributed systems, has seldom been implemented in commercial systems
Optimistic Concurrency Control

Each transaction has three phases:

1. **Working phase**
   - The transaction uses a tentative version of the objects it accesses (dirty reads can’t occur as we read from a committed version or a copy of it)
   - The coordinator records the *readset* and *write set* of each transaction

2. **Validation phase**
   - At `END TRANSACTION` the coordinator validates the transaction (looks for conflicts)
   - If the validation is successful the transaction can commit
   - If it fails, either the current transaction, or one it conflicts with is aborted

3. **Update phase**
   - If validated, the changes in its tentative versions are made permanent
   - Read-only transactions can commit immediately after passing validation
Validation of Transactions

- Use the read-write conflict rules to ensure that a particular transaction is serially equivalent with respect to all other overlapping transactions.
- Each transaction is given a transaction number (defines its position in time) when it starts validation (the number is kept if it commits).
- The rules ensure serializability of transaction $T_v$ (transaction being validated) with respect to transaction $T_i$.

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_i$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_i$ must not read objects written by $T_v$</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_v$ must not read objects written by $T_i$</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$</td>
</tr>
</tbody>
</table>

Rule 3 is valid if only one process at once can be in the update phase.

Rules 1 and 2 are more complicated: test for overlaps between $T_v$ and $T_i$. 

Chapter 6: Transaction Processing
Validation of Transactions

Two forms of validation:

- **Backward** validation
- **Forward** validation
Backward Validation of Transactions

- Check $T_v$ with preceding overlapping transactions

Backward validation of transaction $T_v$:

```java
boolean valid = true;
for (int $T_i = startTn+1; T_i <= finishTn; T_i++){
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;
}
```

- $startTn$ is the biggest transaction number assigned to some other committed transaction when $T_v$ started its working phase
- $finishTn$ is biggest transaction number assigned to some other committed transaction when $T_v$ started its validation phase
- On slide 63, $startTn + 1 = T_2$ and $finishTn = T_3$. In backward validation, the read set of $T_v$ must be compared with the write sets of $T_2$ and $T_3$
- The only way to resolve a conflict is to abort $T_v$
Forward Validation

- Rule 1: the write set of $T_v$ is compared with the read sets of all overlapping active transactions ($active1$ and $active2$)
- Rule 2: ($read \ T_v \ vs \ write \ T_i$) is automatically fulfilled because the active transactions do not write until $T_v$ has completed

Forward validation of transaction $T_v$:

```java
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
    if (write set of $T_v$ intersects read set of Tid)   valid = false;
}
```
Comparison of Forward and Backward Validation

- Which transaction should be aborted?
  - Forward validation allows flexibility, whereas backward validation allows only one choice (the one being validated)
- In general, read sets are larger than write sets
  - Backward validation
    - Compares a possibly large read set against the old write sets
    - Overhead of storing old write sets
  - Forward validation
    - Checks a small write set against the read sets of active transactions
    - Need to allow for new transactions starting during validation
Problem: different servers can order transactions in different ways

→ Parallel validation of transactions

Solution:

• Global validation: the orderings of all servers are checked to be serializable
• Globally unique transaction numbers for validation
Comparison of Methods for Concurrency Control

- Pessimistic approaches (detect conflicts as they arise)
  - *Timestamp ordering*: serialisation order decided statically
  - *Locking*: serialisation order decided dynamically
  - Timestamp ordering is better for transactions where reads >> writes,
  - Locking is better for transactions where writes >> reads
  - Strategy for aborts
    - Timestamp ordering – immediate
    - Locking – waits but can get deadlock

- Optimistic method
  - All transactions proceed, but may need to abort at the end
  - Efficient operations when there are few conflicts, but aborts lead to repeating work
Chapter 6: Transaction Processing

Comitting Transactions

Transaction Manager (Coordinator)
- Allocation of transaction IDs (TIDs)
- Assigning TIDs with operations
- Coordination of commitments, aborts, and recovery
- BEGIN_TRANSACTION, END_TRANSACTION

Scheduler
- Concurrency control
- Lock control (LOCK/RELEASE)
- Timestamp operations
- Deadlock detection

Data (Recovery) Manager
- Logging operations
- Make transactions permanent
- Undo transactions
- Recover after a crash
- Execute read/write operations

Operation sequences

Transaction Manager

Transaction Manager (Coordinator)

Scheduler

Data (Recovery) Manager

Clients

Operation sequences with TIDs

Operaction sequences

with TIDs

Serialization operations

read/write operations

read/write operations

Log

DB
Committing Transactions

• Commit: the operations should have been performed by all processes in a group, or by none.

• Often realized by using a coordinator

• Simple algorithm: One-phase commit
  - The coordinator asks all processes in the group to perform an operation.
  - Problem: if one process cannot perform its operation, it cannot notify the coordinator.

• Thus in practice better schemes are needed. Most common:

  Two-phase commit (2PC)
Two-phase Commit

- Assume: a distributed transaction involves several processes, each running on a different machine and no failures occur
- Algorithm: 2 phases with 2 steps each
  a. Voting phase
  b. Decision phase

1. The coordinator sends a VOTE_REQUEST message to all participants
2. Each receiving participant sends back a VOTE_COMMIT message to notify the coordinator that it is locally prepared to commit its part of the transaction
3. If the coordinator receives the VOTE_COMMIT from all participants, it commits the whole transaction by sending a GLOBAL_COMMIT to all participants. If one or more participants had voted to abort, the coordinator aborts as well and sends a GLOBAL_ABORT to all participants
4. A participant receiving the GLOBAL_COMMIT, commits locally. Same for abort.
Two Phase Commit (2PC) Protocol

Coordinator

Execute
- Precommit

Uncertain
- Send request to each participant
- Wait for replies (time out possible)

Abort
- Send ABORT to each participant

Commit
- Send COMMIT to each participant

Timeout or a NO

Participant

Execute

Abort
- Send NO to coordinator

Commit
- Make transaction visible

Precommit
- Send YES to coordinator

Commit decision

Abort decision

ABORT decision

YES

NO

not ready

ready

All YES
Two-Phase Commit

Finite state machine for the coordinator in 2PC

Finite state machine for a participant

- But... several problems arise if failures can occur: process crashes, message loss
- Timeouts are needed to avoid indefinite waiting for crashed processes in the INIT, WAIT resp. READY state. This is no problem with INIT and WAIT, because no commit or abort was decided till then. But when waiting in state READY, the participant cannot abort, because the coordinator may have given a commit to other processes
Two-Phase Commit

A participant $P$ blocked in the READY state can contact another participant $Q$ trying to decide from $Q$'s state what to do:

<table>
<thead>
<tr>
<th>State of $Q$</th>
<th>Action by $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

When all participants are in state READY, no decision can be made. In that case, the only option is blocking until the coordinator recovers.

*Recovery* means to save current states.
Two-Phase Commit: Steps taken by the Coordinator

**Actions by coordinator:**

- write START_2PC to local log;
- multicast VOTE_REQUEST to all participants;
- while not all votes have been collected {
  - wait for any incoming vote;
  - if timeout {
    - write GLOBAL_ABORT to local log;
    - multicast GLOBAL_ABORT to all participants;
    - exit;
  }
  - record vote;
}
- if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
  - write GLOBAL_COMMIT to local log;
  - multicast GLOBAL_COMMIT to all participants;
} else {
  - write GLOBAL_ABORT to local log;
  - multicast GLOBAL_ABORT to all participants;
}
Two-Phase Commit: Steps taken by a Participant

**Actions by participant:**

write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}

if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
    } else {
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
    }

1st phase

2nd phase
Two-Phase Commit: Second Thread of Participant

**Actions for handling decision requests:** /* executed by separate thread */

```java
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

Stay blocked. If all processes are in the READY state, no more operations can be taken to recover. Thus: **blocking commit protocol**

Solution: **three-phase commit protocol (3PC)**
Three-Phase Commit

- Avoid blocking in the presence of process failures. (Not applied often in practice, because the blocking process failures occur very seldom)
- Coordinator: if all VOTE_COMMITs are received, a PRECOMMIT is sent. If one process crashes and do not respond to this request, the coordinator commits, because the crashed participant has already written a VOTE_COMMIT to its log.
- Participant: if all participants wait in state PRECOMMIT, then commit. If all are in state READY, then abort (No problem, because no crashed process can be in state COMMIT).
Two-Phase Commit Protocol for Nested Transactions

- Recall top-level transaction $T$ and sub-transactions $T_1, T_2, T_{11}, T_{12}, T_{21}, T_{22}$
- A sub-transaction starts after its parent and finishes before it
- When a sub-transaction completes, it makes an independent decision either to
  commit provisionally or to abort
  - A provisional commit is not the same as being prepared: it is a local decision and is not backed up on permanent storage
  - If the server crashes subsequently, its replacement will not be able to carry out a provisional commit
- A two-phase commit protocol is needed for nested transactions
  - It allows servers of provisionally committed transactions that have crashed to abort them when they recover
Example: 2PC in Nested Transactions

- Recall that
  1. A parent can commit even if a sub-transaction aborts
  2. If a parent aborts, then its sub-transactions must abort
- In the figure, each sub-transaction has either provisionally committed or aborted

Nested Distributed Transaction

Bottom up decision in 2PC
Information held by Coordinators of Nested Transactions

- When a top-level transaction commits it carries out a 2PC
- Each coordinator has a list of its sub-transactions
- At provisional commit, a sub-transaction reports its status and the status of its descendents to its parent
- If a sub-transaction aborts, it tells its parent

<table>
<thead>
<tr>
<th>Coordinator of transaction</th>
<th>Child transactions</th>
<th>Participant</th>
<th>Provisional commit list</th>
<th>Abort list</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T_1, T_2$</td>
<td>yes</td>
<td>$T_1, T_{12}$</td>
<td>$T_{11}, T_2$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>$T_{11}, T_{12}$</td>
<td>yes</td>
<td>$T_1, T_{12}$</td>
<td>$T_{11}$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_{21}, T_{22}$</td>
<td>no (aborted)</td>
<td>$T_{12}$ but not $T_{21}$</td>
<td>$T_{11}$</td>
</tr>
<tr>
<td>$T_{11}$</td>
<td></td>
<td>no (aborted)</td>
<td>$T_{21}$</td>
<td>$T_{11}$</td>
</tr>
<tr>
<td>$T_{12}, T_{21}$</td>
<td></td>
<td></td>
<td>$T_{22}$</td>
<td>$T_{22}$</td>
</tr>
<tr>
<td>$T_{22}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Access

- **Transaction Manager (Coordinator)**
  - Allocation of transaction IDs (TIDs)
  - Assigning TIDs with operations
  - Coordination of commitments, aborts, and recovery
  - `BEGIN_TRANSACTION, END_TRANSACTION`

- **Scheduler**
  - Concurrency control
  - Lock control (LOCK/RELEASE)
  - Timestamp operations
  - Deadlock detection

- **Data (Recovery) Manager**
  - Logging operations
  - Make transactions permanent
  - Undo transactions
  - Recover after a crash
  - Execute read/write operations

![Diagram showing the relationships between Transaction Manager, Scheduler, and Data (Recovery) Manager with operation sequences and TIDs.](image)
Problems with Aborts

- If a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects.

- Two problems:
  - *Dirty reads*
    - An interaction between a *read* operation in transaction $A$ and an earlier *write* operation on the same object by a transaction $B$ that then aborts.
    - $A$ has committed before $B$ aborts, so it has seen a value that has never existed.
    - A transaction that committed with a ‘dirty read’ is not recoverable. Solution: delay the own commit of the transaction performing a read which could become a dirty read.
  
- *Premature writes*
  - Interactions between *write* operations on the same object by different transactions, one of which aborts.
  - Write operations must be delayed until the first transaction which has modified the data item has committed or aborted.
### A Dirty Read when Transaction T aborts

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.getBalance()</code></td>
<td><code>a.getBalance()</code></td>
</tr>
<tr>
<td><code>a.setBalance(balance + 10)</code></td>
<td><code>a.setBalance(balance + 20)</code></td>
</tr>
<tr>
<td><code>balance = a.getBalance()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>balance = a.getBalance()</code> $110`</td>
</tr>
<tr>
<td><code>balance = a.getBalance()</code></td>
<td><code>balance = a.getBalance()</code> $130`</td>
</tr>
<tr>
<td></td>
<td><code>a.setBalance(balance + 20)</code> $130`</td>
</tr>
<tr>
<td><code>a.setBalance(balance + 10)</code></td>
<td></td>
</tr>
</tbody>
</table>

- **U** reads **A**’s balance (which was set by **T**) and then commits

<table>
<thead>
<tr>
<th><strong>T</strong> subsequently aborts</th>
<th><strong>commit transaction</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abort transaction</code></td>
<td></td>
</tr>
</tbody>
</table>

**U** has committed, so it cannot be undone. It has performed a dirty read.
## Premature Writes - Overwriting uncommitted Values

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$100$</td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$105$</td>
</tr>
</tbody>
</table>

Interaction between write operations when a transaction aborts

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.setBalance(110)$</td>
<td>$110$</td>
</tr>
</tbody>
</table>

Some database systems keep ‘before images’ and restore them after aborts

- E.g. $100$ is before image of $T$’s write, $105$ is before image of $U$’s write
- If $U$ aborts we get the correct balance of $105$,
- But if $U$ commits and then $T$ aborts, we get $100$ instead of $110$
Implementation of Data Manager

There are two common methods for implementing the data manager:

**Private Workspace**

- A starting process is given copies of all the files which it has to access
- Until commit/abort, all operations are made on the private workspace
- Optimisations:
  - Do not copy files which only have to be read, only set a pointer to the parent's workspace.
  - For writing into files, only copy the file's index. When writing in a file block, only copy this block and update the address in the copied file index.
- In distributed transactions: one process is started on each involved machine, getting a workspace containing the files necessary on this machine (as above)

**Writeahead Log**

- Files are modified in place, but before changing a block, a record is written to a log, containing: which transaction is making the change, which file/block is changed, what are the old and new values.
- In case of an abort: make a simple rollback, read the log from the end and undo all recorded changes
Private Workspace

(a) The file index and disk blocks for a three-block file
(b) The situation after a transaction has modified block 0 and appended block 3
(c) After committing, the changes are stored
Writeahead Log

\[ x = 0; \]
\[ y = 0; \]
\[ \text{BEGIN\_TRANSACTION;} \]
\[ x = x + 1; \quad [x = 0/1] \]
\[ y = y + 2 \quad [y = 0/2] \]
\[ x = y * y; \quad [x = 1/4] \]
\[ \text{END\_TRANSACTION;} \]

(a) A transaction
(b) – d) The log before each statement is executed
Conclusion

Cooperation very often need *synchronization*:
- Based on absolute time (NTP)
- Based on logical ordering (Lamport timestamps, more often needed)

Also often needed for cooperation: a *coordinator*
- Election algorithms

For fault tolerance: making *decisions* on values/actions in a group of components:
- Consensus algorithms

When having data access, a *coordinated access* has to be implemented:
- Maekawa’s Voting Algorithm for mutual exclusion for single access operations
- Distributed transactions with a distributed transaction manager (2PC) and cooperating schedulers (usually 2PL)