

# Distributed Systems (ICE 601)

## Concurrency Control - Part2

Dongman Lee  
ICU

## Optimistic Concurrency Control

- Principle
  - transaction proceeds without checking conflict with others and prior to commit, *validates* its change by checking to see if data items have changed by *committed transactions*
  - each transaction has three phases
    - ◆ Read phase
      - committed version of data items for read - *read set*
      - tentative version of data items for write - *write set*
    - ◆ Validation phase
      - starts with EndTransaction request
      - validate its change by checking to see if data items have changed by other transactions
      - if no conflicts, commit; otherwise, abort
    - ◆ Write phase
      - make changes permanent

Distributed Systems - Transactions & Concurrency Control (2/2)

## Class Overview

- Transactions
- Why Concurrency Control
- Concurrency Control Protocols
  - pessimistic
  - optimistic
  - time-based

## Optimistic Concurrency Control (cont.)

- Validation test rule
    - $T_j$  is *serializable* with respect to overlapping  $T_i$  if their operations conform to the following rules
- | $T_i$        | $T_j$        | Rule   |
|--------------|--------------|--|
| <i>Read</i>  | <i>Write</i> | 1. $T_i$ must not read data items written by $T_j$   |
| <i>Write</i> | <i>Read</i>  | 2. $T_j$ must not read data items written by $T_i$   |
| <i>Write</i> | <i>Write</i> | 3. $T_i$ must not write data items written by $T_j$ and $T_j$ must not write data items written by $T_i$ |
- (Assumption:  $T_i$  always preceeds  $T_j$  if  $i < j$  and  $T_i$  overlaps with  $T_j$ )
- *transaction # is sequentially assigned when validation phase starts*
- Validation mechanisms
  - backward validation
  - forward validation

## Optimistic Concurrency Control (OCC)

- Backward validation

- algorithm

- checks transaction in validation phase with other preceding overlapping transactions that have entered validation phase

- Write* operations are ok since Read operations of earlier transactions are done already (Rule 1)

- check if *Read* operations have any conflict with *Write* operations of earlier overlapping transactions (Rule 2) => if yes, abort transaction

```
Valid := True;
```

```
for Ti := startTn + 1 to finishTn
```

```
do
```

```
  if read set of Tj intersects write set of Ti
```

```
    Valid := False;
```

```
end
```

- no check is needed for transaction with only *Write* operations

Distributed Systems - Transactions & Concurrency Control (2/2)

## Optimistic Concurrency Control (OCC)

- Forward validation

- algorithm

- checks transaction in validation phase with other overlapping active transactions

- Read* operations are ok since later transactions do not write until the Tj is done (Rule 2)

- check if *Write* operations have any conflict with *Read* operations of overlapping active transactions (Rule 1) => if yes, abort transaction

```
Valid := True;
```

```
for Ti := active1 to activeN
```

```
do
```

```
  if write set of Tj intersects read set of Ti
```

```
    Valid := False;
```

```
end
```

- no check is needed for transaction with only *Read* operations

- other options than aborting the current transaction

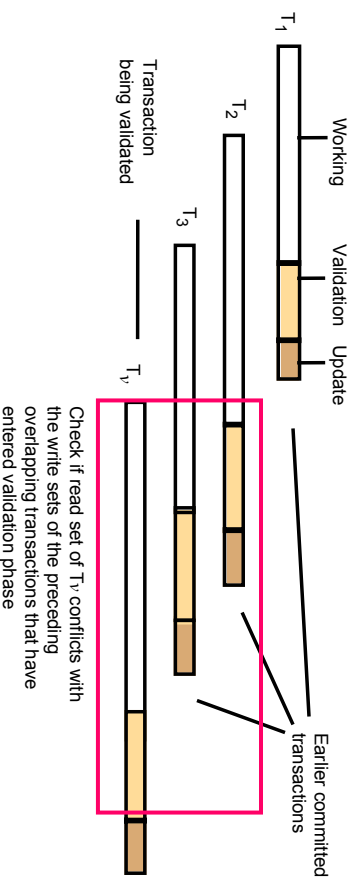
- defer validation until conflicting transaction is done

- abort conflicting transaction instead

Distributed Systems - Transactions & Concurrency Control (2/2)

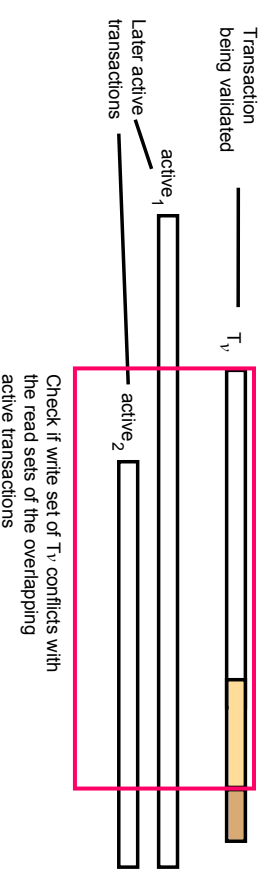
## Optimistic Concurrency Control (cont.)

- Backward validation example



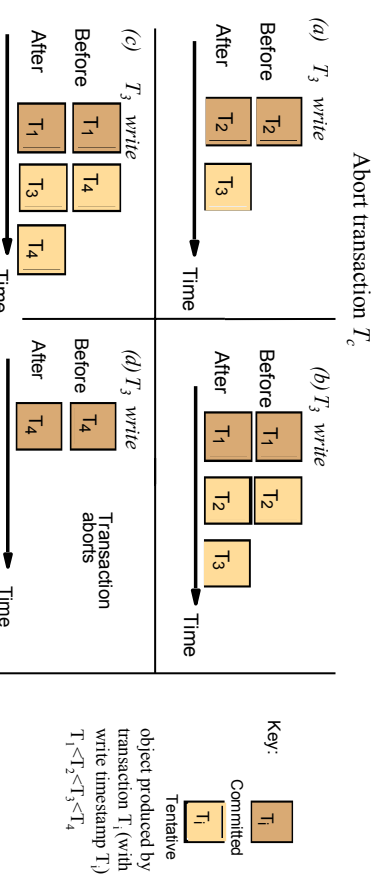
## Optimistic Concurrency Control (cont.)

- Forward validation example



- Issues in optimistic concurrency control
  - Overhead
    - ♦ Backward validation
      - if there exists long transaction, retention of old write sets of data item may be a problem
    - ♦ Forward validation
      - a new transaction can start during the validation process -> increase chances by which the current transaction is forced to abort or delay
  - Starvation
    - ♦ prevention of a transaction ever being able to commit

- Write operations and time stamp
  - if ( $T_c \geq$  maximum read timestamp on  $D$  &&
  - $T_c >$  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  $T_c$



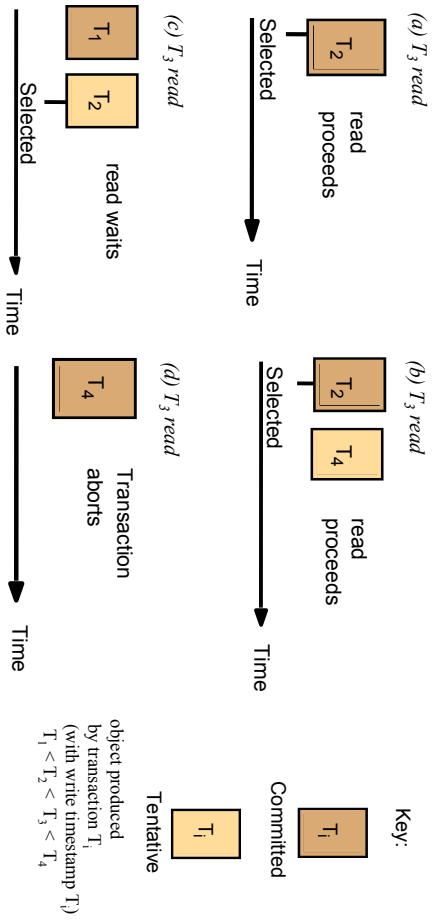
## Timestamp Ordering

- Assumption
  - each transaction is given a unique timestamp when it starts
  - there is only one version of data item and only one transaction can access it at a time => multiple tentative versions of data to increase concurrency
- Rule
  - Write operation is valid only if the data was last read and written by earlier transaction
    - ♦ Rule 1:  $T_i$  must not write data item read by any  $T_j$  where  $T_i > T_j$  (i.e.  $T_j >=$  max read time stamp of data item)
    - ♦ Rule 2:  $T_j$  must not write data item written by any  $T_i$  where  $T_i > T_j$  (i.e.  $T_j >$  max write time stamp of committed data item)
  - Read operation is valid only if the data was last written by earlier transaction
    - ♦ Rule 3:  $T_i$  must not read data item written by  $T_j$  where  $T_i > T_j$  (i.e.  $T_i >$  write

## Timestamp Ordering (cont.)

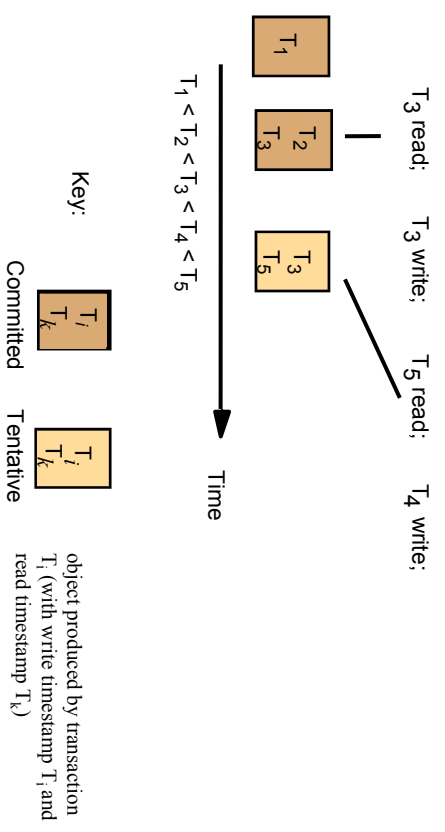
- Read operations and time stamp
  - if ( $T_c >$  write timestamp on committed version of  $D$ ) {
  - let  $D_{selected}$  be the version of  $D$  with the maximum write timestamp  $\leq T_c$
  - if ( $D_{selected}$  is committed)
  - perform *read* operation on the version  $D_{selected}$
  - else
  - Wait* until the transaction that made version  $D_{selected}$  commits or aborts
  - then reapply the *read* rule
  - } else
  - Abort transaction  $T_c$

• Read operations and time stamp - example



Distributed Systems - Transactions & Concurrency Control (2/2)

• Multi-version timestamp ordering - example



Distributed Systems - Transactions & Concurrency Control (2/2)

## Timestamp Ordering (cont.)

- Multi-version timestamp ordering
  - keep old versions of committed data as well as tentative versions
  - read operation is always allowed; may need to wait for earlier transactions to complete
  - no conflict between write operations since each transaction writes its own committed version (remove rule 2)
  - write rule
    - if read time stamp (most recent version)  $\leq T_i$  then perform write operation on a tentative version with write time stamp  $T_j$

## Comparison

- Locking vs. timestamp ordering
  - both are pessimistic
  - dynamic vs static ordering
  - write-dominated vs. read-dominated
- Optimistic
  - efficient when there are few conflicts
- New requirements to concurrency control
  - multi-user applications
    - immediate notification of change (relaxed isolation)
    - need to be able to access uncommitted data item
  - co-operative CAD/CAM
    - co-operations of users to resolve data conflicts