CS 764: Topics in Database Management Systems

Lecture 7: Optimistic Concurrency Control

Xiangyao Yu

9/28/2020

Announcement

Guest lecture on Wednesday (Sep. 30) by Shasank Chavan from Oracle on "*Hardware Acceleration with Oracle Database In-Memory*"

Student round-table discussion after the talk (2:30—3:30)

Today’s Paper: Optimistic Concurrency Control

On Optimistic Methods for Concurrency Control

H.T. KUNG and JOHN T. ROBINSON
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are “optimistic” in the sense that they rely mainly on transaction backup as a control mechanism, “hoping” that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

Key Words and Phrases: databases, concurrency controls, transaction processing
CR Categories: 4.32, 4.33

1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

ACM Trans. Database Syst. 1981

Agenda

Pessimistic concurrency control

Optimistic concurrency control

Concurrency Control

Concurrency control ensures the correctness for concurrent operations

Assume **serializable** isolation level for this lecture

Concurrency Control

Concurrency control ensures the correctness for concurrent operations

Assume **serializable** isolation level for this lecture

Pessimistic: Resolve conflicts eagerly

Optimistic: Ignore conflicts during a transaction's execution and resolve conflicts lazily only when at a transaction's completion time

Concurrency Control

Concurrency control ensures the <u>correctness</u> for concurrent operations

Assume **serializable** isolation level for this lecture

Pessimistic: Resolve conflicts eagerly

Optimistic: Ignore conflicts during a transaction's execution and resolve conflicts lazily only when at a transaction's completion time

Other common concurrency control protocols

- Timestamp ordering (T/O)
- Multi-version concurrency control (MVCC)

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

T1

Begin

acquire S lock on A

Read(A) 

Time

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

T1

Begin

acquire S lock on A

Read(A) S

…

acquire X lock on B

Write(B) X

Time

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

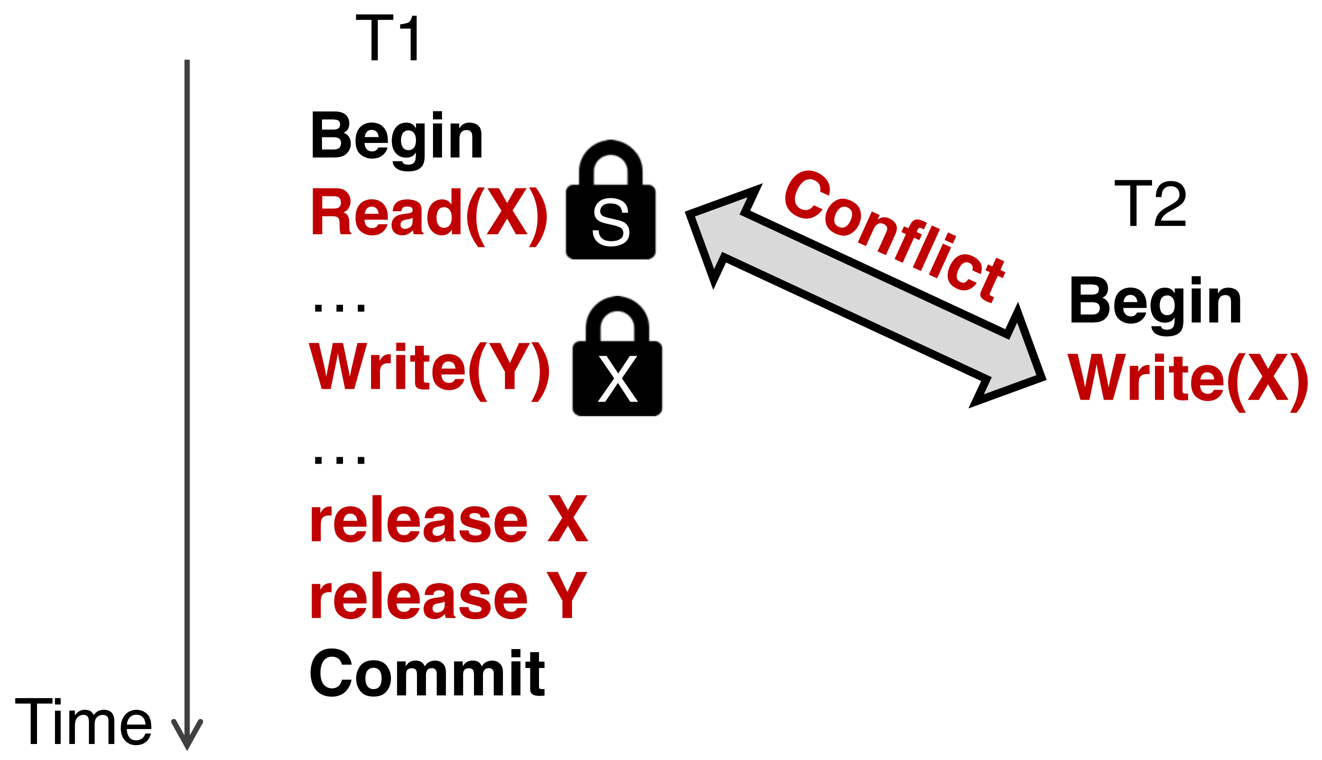
- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

T1

Time

Begin
acquire S lock on A
Read(A) S
…
acquire X lock on B
Write(B) X
…
release lock on A
release lock on B
Commit

Conflicts in 2PL



Solution 1: T2 waits for T1 to release lock (e.g., **wait-die**, **deadlock-detection**)

Solution 2: T2 self aborts (e.g., **wait-die, no-wait**)

Solution 3: T2 forces T1 to abort (e.g., **wound-wait**)

Deadlock

T1

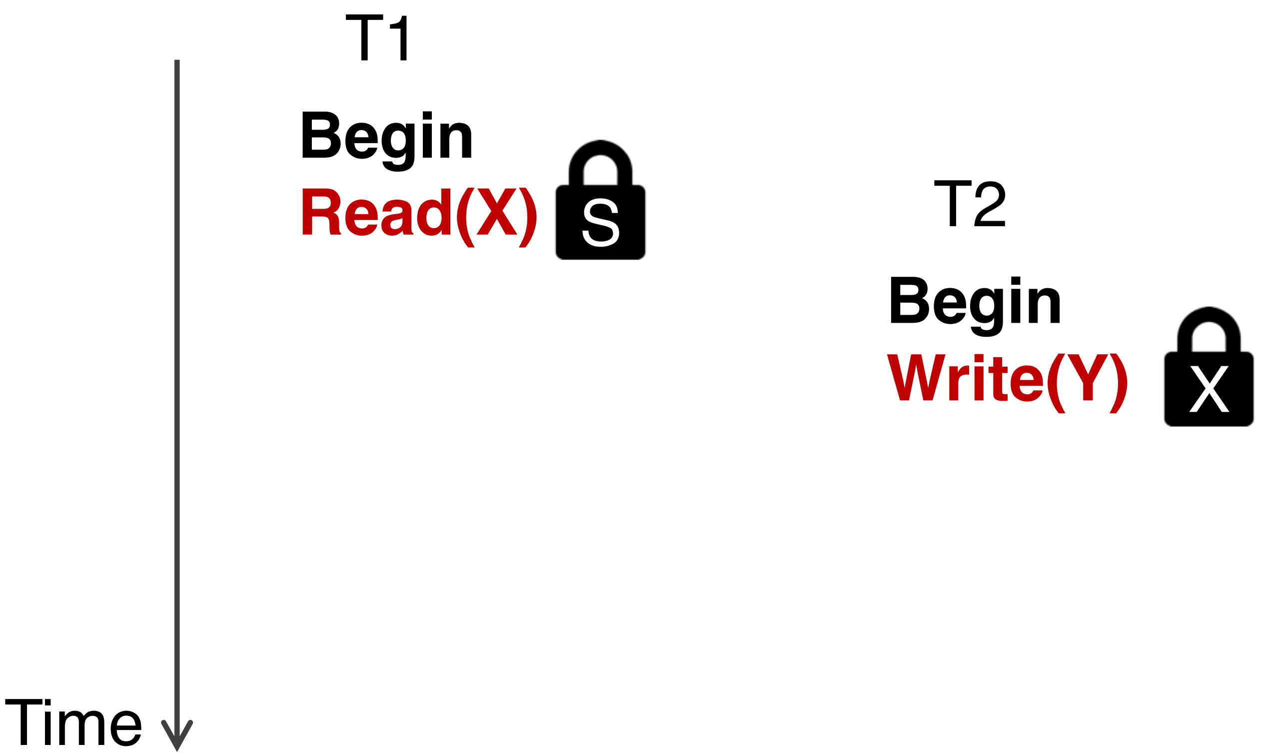
Begin

Read(X) S

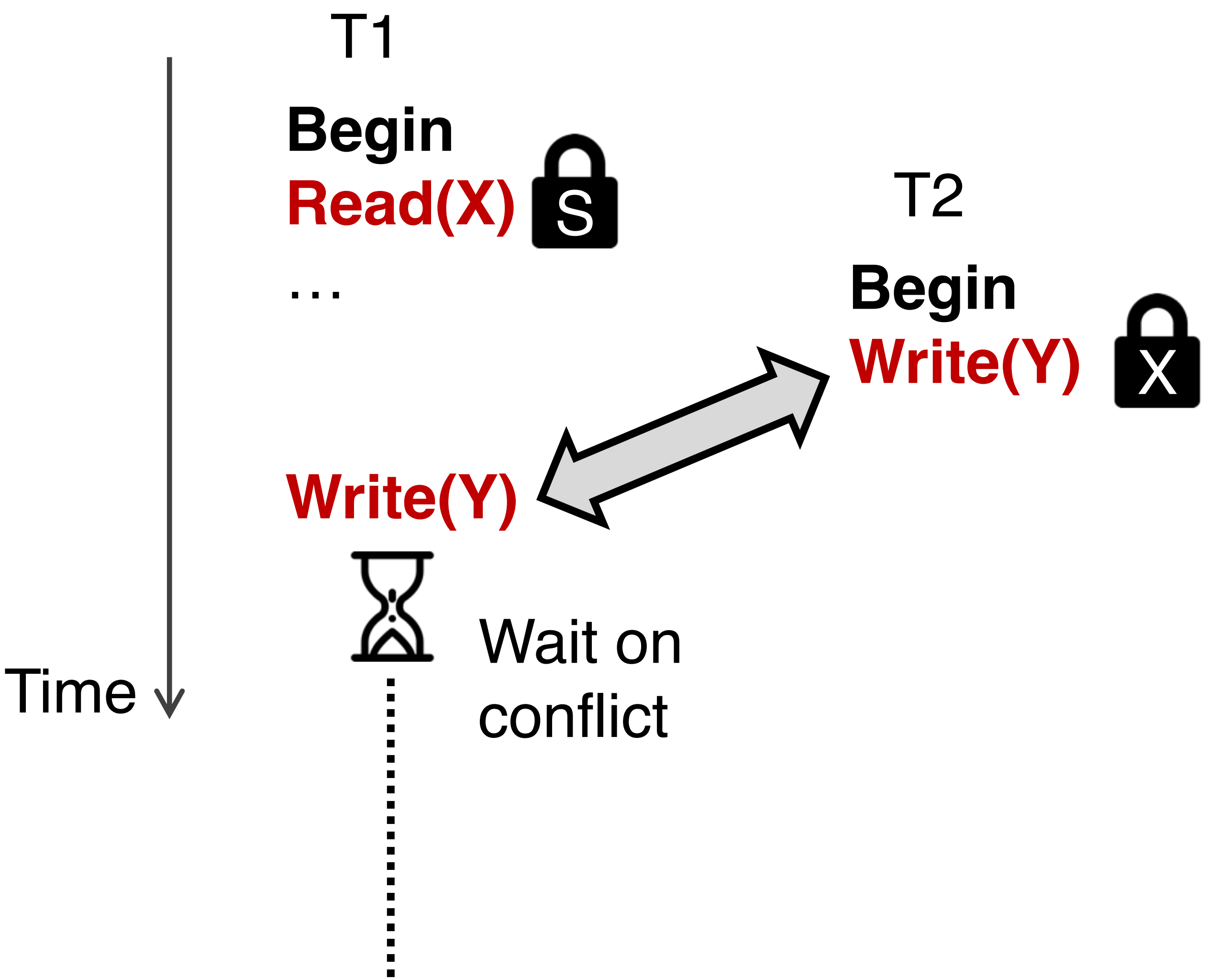
T2

Time

Deadlock

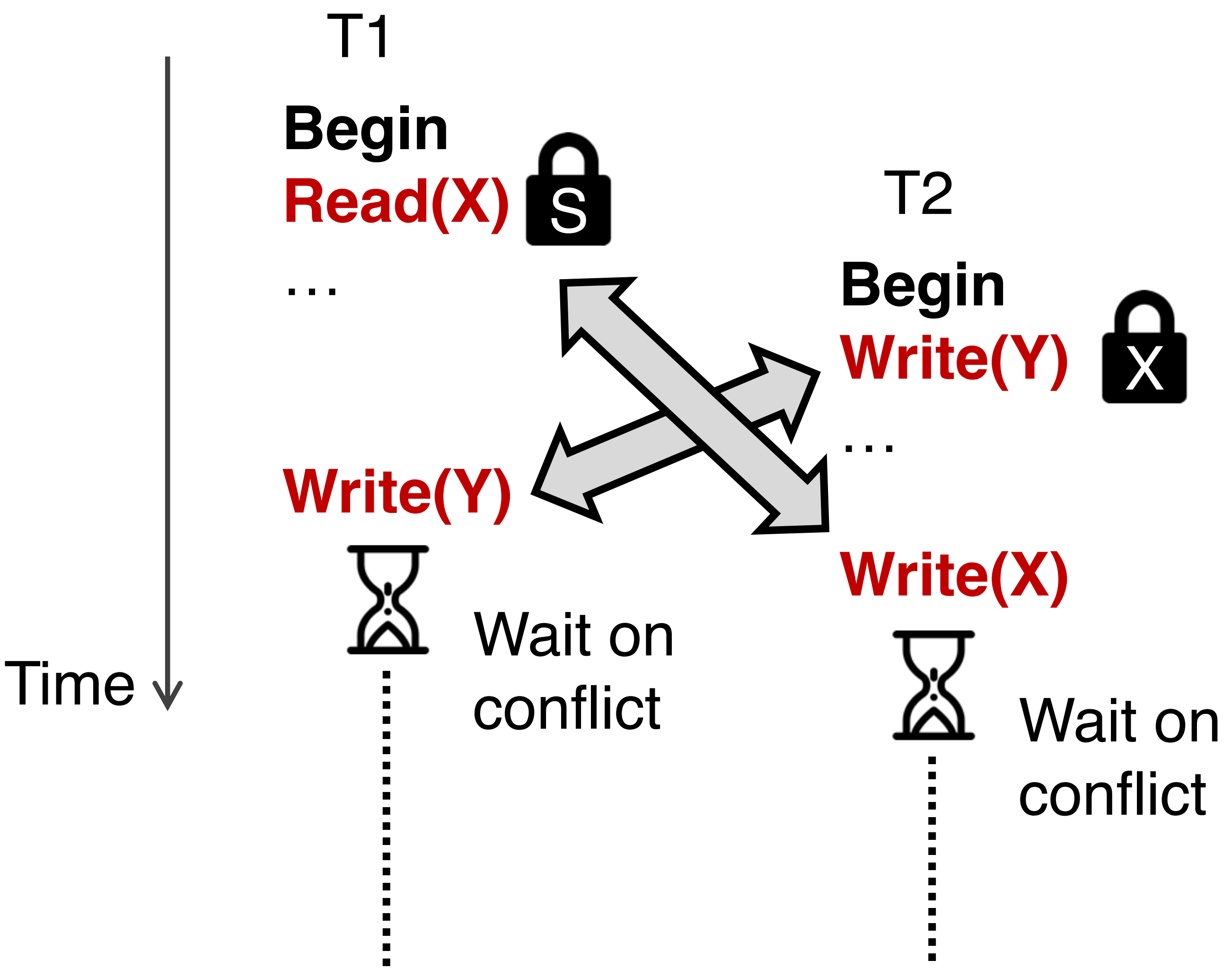


Deadlock



T1
Begin
Read(X)
S
…
T2
Begin
Write(Y)
X
Write(Y)
Wait on conflict
Time

Deadlock



Both transactions wait for each other

=> **Deadlock**

Deadlock Resolution

Deadlock detection (DL_DETECT)

- Maintain a **wait-for** graph among transactions; abort a transaction if a cycle is formed

Deadlock Resolution

Deadlock detection (DL_DETECT)

- Maintain a **wait-for** graph among transactions; abort a transaction if a cycle is formed

NO_WAIT

- The requesting transaction self aborts when a conflict occurs

Deadlock Resolution

Deadlock detection (DL_DETECT)

- Maintain a **wait-for** graph among transactions; abort a transaction if a cycle is formed

NO_WAIT

- The requesting transaction self aborts when a conflict occurs

WAIT_DIE

- The requesting transaction waits if its priority is higher than the lock owner (**wait**), otherwise the requesting transaction self aborts (**die**)

Deadlock Resolution

Deadlock detection (DL_DETECT)

- Maintain a **wait-for** graph among transactions; abort a transaction if a cycle is formed

NO_WAIT

- The requesting transaction self aborts when a conflict occurs

WAIT_DIE

- The requesting transaction waits if its priority is higher than the lock owner (**wait**), otherwise the requesting transaction self aborts (**die**)

WOUND_WAIT

- The requesting transaction forces the lock owner to abort (**wound**) if its priority is higher than the lock owner, otherwise the requesting transaction waits (**wait**)

Issues with Pessimistic CC

Overhead

- Overhead of acquiring/releasing locks and maintaining lock metadata
- Even read-only transactions acquire locks

Deadlocks

Limited concurrency

Locks are held till the end of a transaction

Real workloads have low contention

- Locking is unnecessary if no contention exists

Optimistic Concurrency Control (OCC)

Goal: eliminating pessimistic locking

Three executing phases:

- Read
- Validation
- Write

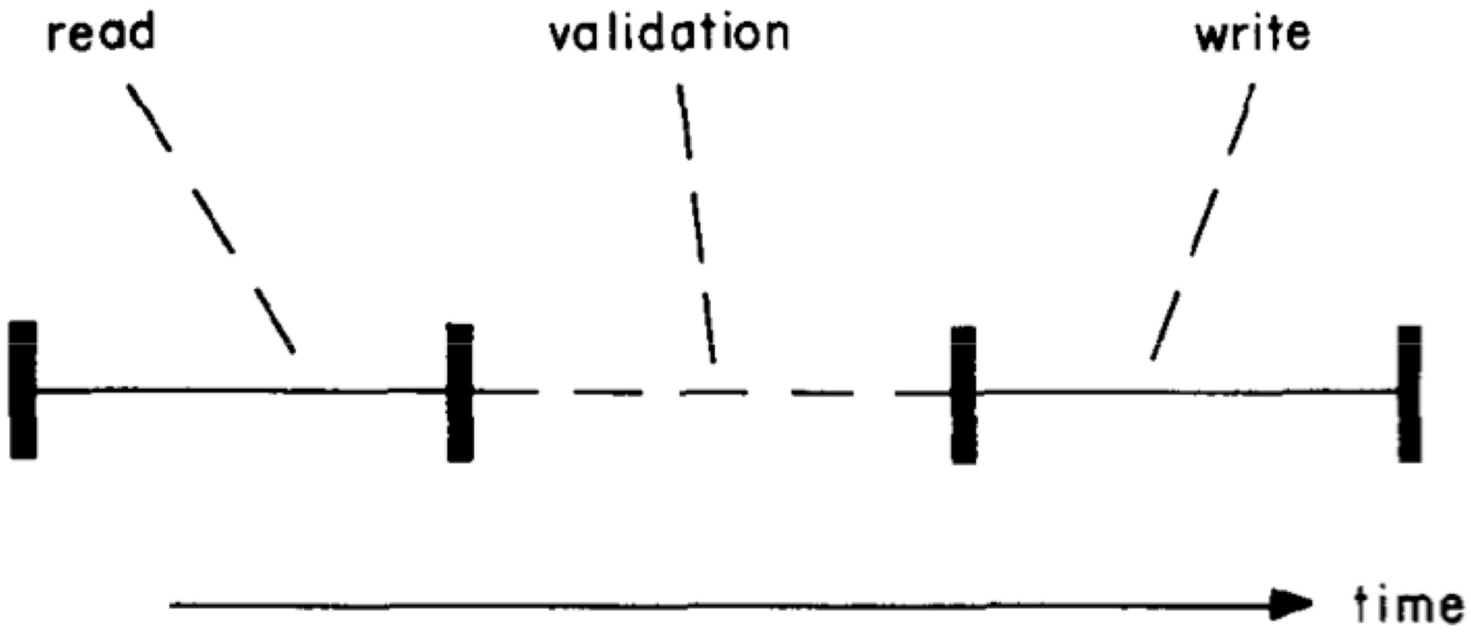


Fig. 1. The three phases of a transaction.

Read Phase

n = tcreate

$$
\begin{aligned}
&tcreate = (\\
&\quad n := create;\\
&\quad create\ set := create\ set \cup \{n\};\\
&\quad \mathbf{return}\ n)
\end{aligned}
$$

Read Phase

n = tcreate

twrite(n, i, v)

$$
\begin{array}{l}
\mathit{twrite}(n, i, v) = (\\
\quad \textbf{if } n \in \mathit{create\ set} \\
\qquad\quad \textbf{then } \mathit{write}(n, i, v) \\
\quad \textbf{else if } n \in \mathit{write\ set} \\
\qquad\quad \textbf{then } \mathit{write}(\mathit{copies}[n], i, v) \\
\quad \textbf{else } (\\
\qquad\quad m := \mathit{copy}(n); \\
\qquad\quad \mathit{copies}[n] := m; \\
\qquad\quad \mathit{write\ set} := \mathit{write\ set} \cup \{n\}; \\
\qquad\quad \mathit{write}(\mathit{copies}[n], i, v)))
\end{array}
$$

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

```
tread(n, i) = (
    read set := read set ∪ {n};
    if n ∈ write set
        then return read(copies[n], i)
    else
        return read(n, i))
```

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

tdelete(n)

$$\begin{aligned}&tdelete(n) = (\\&\quad delete\ set := delete\ set \cup \{n\}).\end{aligned}$$

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

tdelete(n)

All changes (i.e., inserts, updates, deletes) are kept local to the transaction without updating the database

Write Phase

All written values become "global"

$$\textbf{for } n \in \textit{write set } \textbf{do } \textit{exchange}(n, \textit{copies}[n]).$$

All created nodes become accessible

All deleted nodes become inaccessible

Validation Phase

A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- $t(i) < t(j)$ => exists a serial schedule where T_i is before T_j

Validation Phase

A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- $t(i) < t(j)$ => exists a serial schedule where T_i is before T_j

For $t(i) < t(j)$, one of the following must be true

1. T_i completes its write phase before T_j starts its read phase.
2. The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.
3. The write set of T_i does not intersect the read set or the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

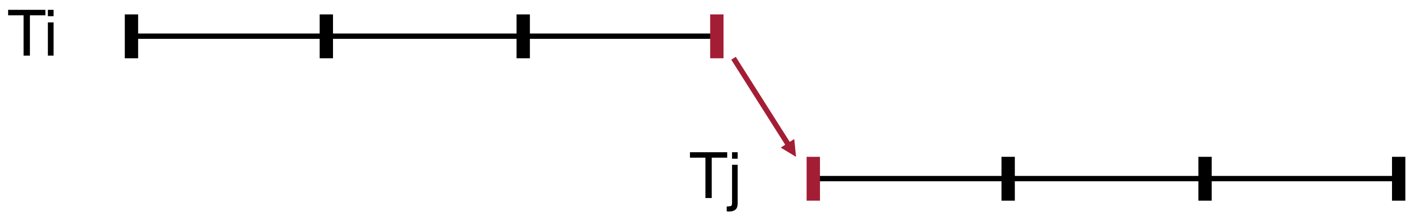
Validation Phase

A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- $t(i) < t(j)$ => exists a serial schedule where T_i is before T_j

For $t(i) < t(j)$, one of the following must be true

1. T_i completes its write phase before T_j starts its read phase.
2. The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.
3. The write set of T_i does not intersect the read set or the write set of T_j, and T_i completes its read phase before T_j completes its read phase.



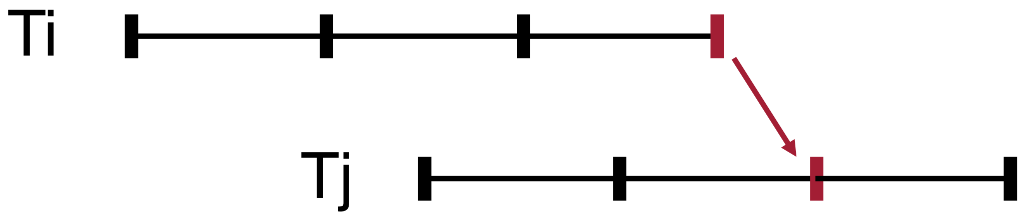
Validation Phase

A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- $t(i) < t(j)$ => exists a serial schedule where T_i is before T_j

For $t(i) < t(j)$, one of the following must be true

1. T_i completes its write phase before T_j starts its read phase.
2. The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.
3. The write set of T_i does not intersect the read set or the write set of T_j, and T_i completes its read phase before T_j completes its read phase.



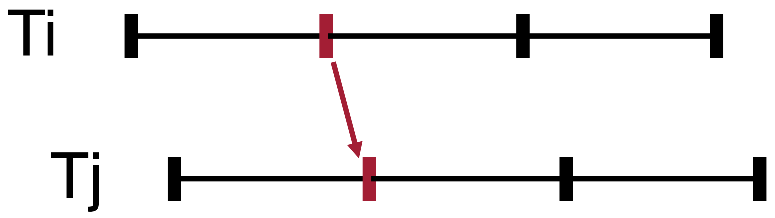
Validation Phase

A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- *t(i)* < *t(j)* => exists a serial schedule where T_i is before T_j

For *t(i)* < *t(j)*, one of the following must be true

1. T_i completes its write phase before T_j starts its read phase.
2. The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.
3. The write set of T_i does not intersect the read set or the write set of T_j, and T_i completes its read phase before T_j completes its read phase.



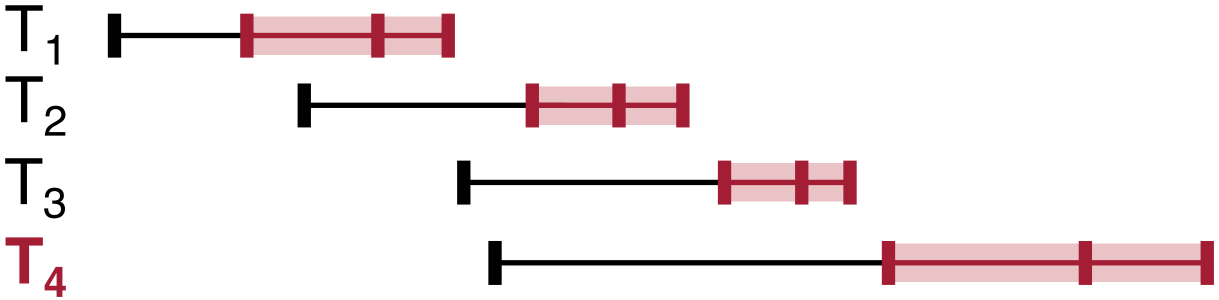
Serial Validation

```
tbegin = (
    start tn := tnc)
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
           then valid := false;
   if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
        then (cleanup)
        else (backup)).
```

Critical Section

Which transactions will T2, T3, and T4 be validated against?

Problem: The entire validate process happens in the critical section



Improved Serial Validation

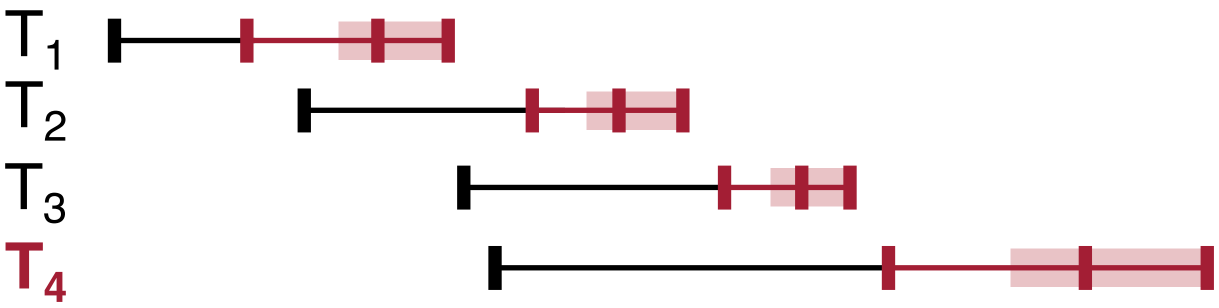
```
tend := (
    mid tn := tnc;
    valid := true;
    for t from start tn + 1 to mid tn do
    if (write set of transaction with transaction number t intersects read set)
        then valid := false;
⟨finish tn := tnc;
 for t from mid tn + 1 to finish tn do
     if (write set of transaction with transaction number t intersects read set)
        then valid := false;
 if valid
     then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
 if valid
     then (cleanup)
     else (backup)).
```

Critical Section

Part of the validation process happens outside the critical section

The optimization can be applied repeatedly

Readonly transactions do not enter the critical section



Parallel Validation

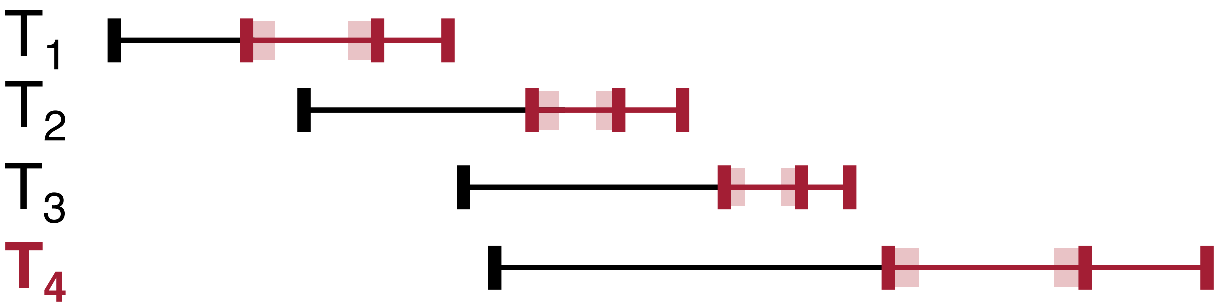
```
tend = (
  ⟨finish tn := tnc;
   finish active := (make a copy of active);
   active := active ∪ {id of this transaction}⟩;
  valid := true;
  for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
          then valid := false;
  for i ∈ finish active do
       if (write set of transaction T_i intersects read set or write set)
          then valid := false;
  if valid
       then (
          (write phase);
          ⟨tnc := tnc + 1;
           tn := tnc;
           active := active—{id of this transaction}⟩;
          (cleanup))
       else (
          ⟨active := active—{id of transaction}⟩;
          (backup))).
```

Critical Sections

Validation against other transactions and writes both happen outside the critical section

Length of the critical section is independent of the number of validating transactions

Leading to unnecessary aborts



Q/A – OCC

Why write and validation phases likely take place in RAM?

Hybrid CC that combines OCC and 2PL?

- Yes. Checkout MOCC and CormCC

Concurrent way to deal with unnecessary aborts in parallel validation?

tbegin vs. tcreate?

Why any serial order of transactions acceptable? Shouldn't it be the submission order?

- Strict serializability: If T1 finishes before T2 starts, T1 is before T2 in the global serial order

Practical systems using 2PL vs. OCC?

OCC vs. 2PL in performance?

Group Discussion

What are the downsides of OCC compared to 2PL?

Before Next Lecture

Submit discussion summary to https://wisc-cs764-f20.hotcrp.com

- Title: **Lecture 7 discussion. group ##**
- Authors: Names of students who joined the discussion
- Summary submission Deadline: **Tuesday 11:59pm**

Submit review for

- Philip L. Lehman, S. Bing Yao: Efficient Locking for Concurrent Operations on B-Trees. ACM Trans. Database Syst. 1981.
- **Before next Monday (Oct. 5)**