WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems

Lecture 12: Optimistic Concurrency Control

Xiangyao Yu

10/17/2022

Announcement

Project proposal deadline: **Oct. 24**

Make sure to cover the following aspects (in 1 or 2 pages)

- Project name
- Author list
- Background and motivation (why is the problem important? what are the challenges)
- Task plan (what will you do in the project? what are your key contributions?)
- Timeline

Submission website: https://wisc-cs764-f22.hotcrp.com

Recommend ACM format

- https://www.acm.org/publications/proceedings-template

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

Downsides of pessimistic concurrency control

Optimistic concurrency control

- Read phase
- Write phase
- Validation phase

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 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

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

Downsides of pessimistic concurrency control

- Locking overhead, even for read-only transactions
- Deadlocks
- Limited concurrency due to (1) congestion and (2) holding locks till the end of a transaction

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

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

Downsides of pessimistic concurrency control

- Locking overhead, even for read-only transactions
- Deadlocks
- Limited concurrency due to (1) congestion and (2) holding locks till the end of a transaction

Observation: Locking is needed only if contention exists; real workloads have low contention

Optimistic Concurrency Control (OCC)

Goal: eliminating pessimistic locking

Three executing phases:

- Read
- Validation
- Write

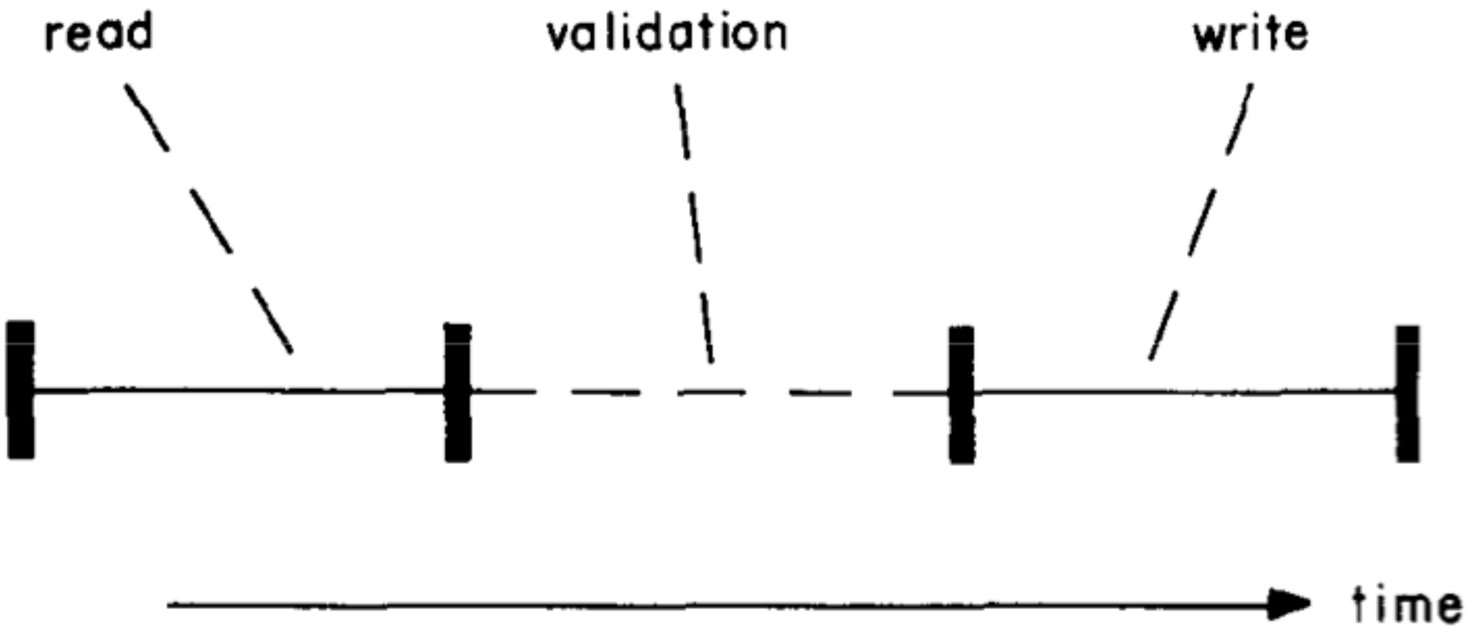


Fig. 1. The three phases of a transaction.

Read Phase

n = tcreate

- Insert a new record

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

Read Phase

n = tcreate

twrite(n, i, v)

- Write to local write set
- No modification to the database

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

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

- Read from either the local write set or the database

$$
\begin{aligned}
&tread(n, i) = (\\
&\quad read\ set := read\ set \cup \{n\}; \\
&\quad \textbf{if}\ n \in write\ set \\
&\qquad \textbf{then return}\ read(copies[n], i) \\
&\quad \textbf{else} \\
&\qquad \textbf{return}\ read(n, i))
\end{aligned}
$$

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

tdelete(n)

- Mark delete in local delete set
- No deletion from the database

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

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

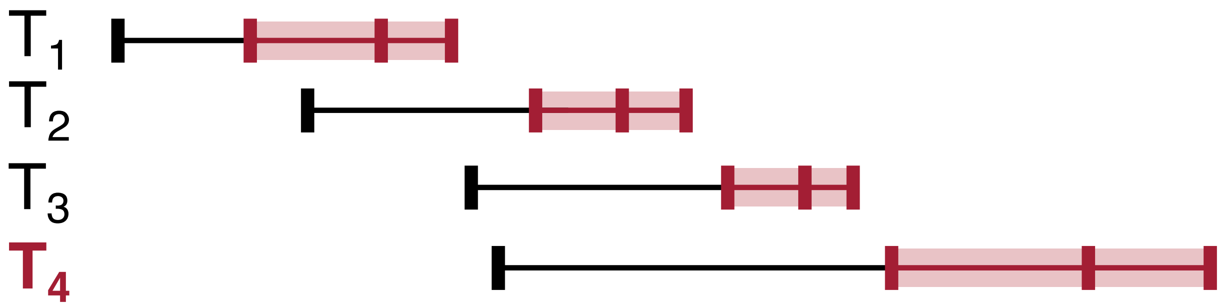
A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- Transaction number determines global serialization order
- $t(i) < t(j)$ => exists a serial schedule where T_i is before T_j
- If execution does not obey this order, the validating transaction aborts

Serial Validation

Critical Section

```
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)).
```

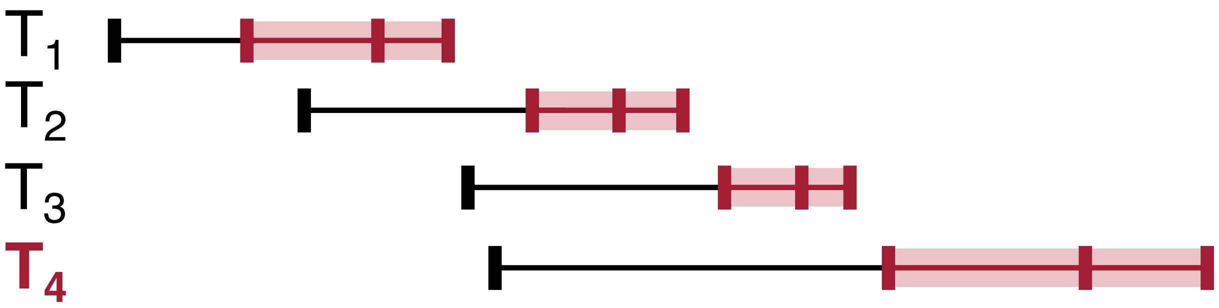


Serial Validation

```
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

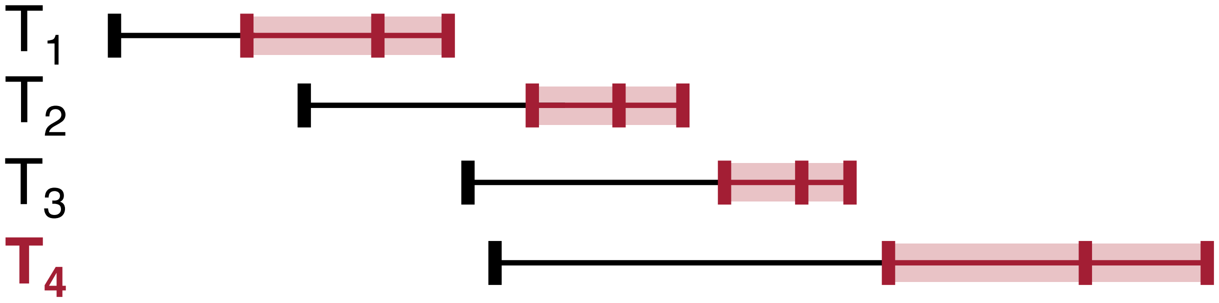
Which transactions will T2, T3, and T4 be validated against?



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: Both *validate* and *write* phases happen in the critical section

Improved Serial Validation

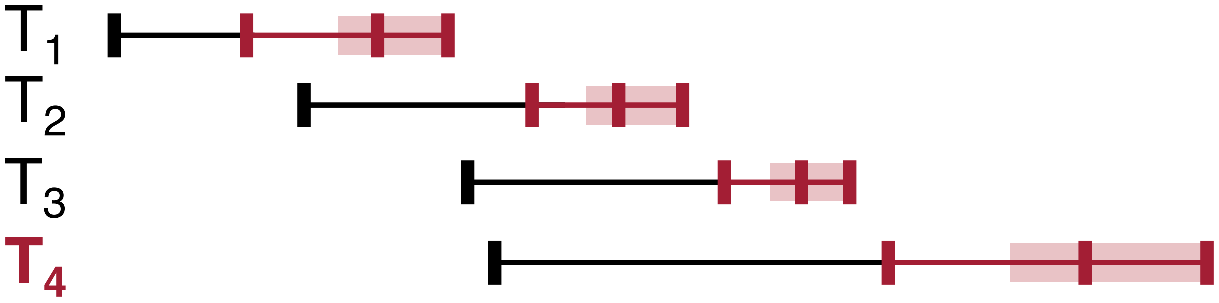
```
tend := (
    mid tn := tnc;
    valid := true;
    for t from start tn + 1 to mid tn do
    if (write set of transaction with transaction number t intersects read set)
        then valid := false;
  ⟨finish tn := tnc;
    for t from mid tn + 1 to finish tn do
        if (write set of transaction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

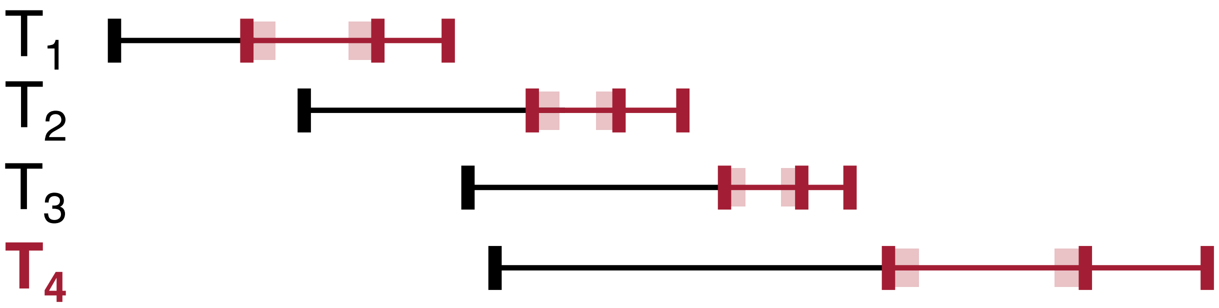
Readonly transactions do not enter the critical section



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 Ti 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
- Abort due to conflict with an aborted transaction

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))).
```

Question: Why need to consider both read set and write set when validating against transactions in *finish active*?

2PL vs. OCC

Revisit the motivation of OCC:

- **Locking overhead**, even for read-only transactions
- **Deadlocks**
- **Limited concurrency** due to (1) congestion and (2) holding locks till the end of a transaction

Comments:

- Optimized locks have low overhead, relative to disk and network cost
- When 2PL has limited concurrency, OCC may have high abort rate

Q/A – OCC

What existing systems use optimistic concurrency control?

Is conflict really rare in real workloads?

Automatically choose locking strategy based on workload?

Achieve optimistic locking with only an additional version attribute?

Granularity of locking for OCC?

Implement different isolation levels in OCC?

Undefined behavior because of reading inconsistent database?

- Opacity

Discussion

What are the downsides of OCC compared to 2PL?

Before Next Lecture

Submit review for

- Stephen Tu, et al., Speedy transactions in multicore in-memory databases. SOSP, 2013