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](https://wisc-cs764-f22.hotcrp.com)

Recommend ACM format
- [https://www.acm.org/publications/proceedings-template](https://www.acm.org/publications/proceedings-template)
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: 4.3.2, 4.3.3

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.
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

\[
tcreate = ( \\
  n := create; \\
  create set := create set \cup \{ n \}; \\
  return n )
\]
Read Phase

\[ n = tcreate \]

\[ twrite(n, i, v) \]
- Write to local write set
- No modification to the database

\[
twrite(n, i, v) = ( \\
\text{if } n \in \text{create set} \\
\text{then write}(n, i, v) \\
\text{else if } n \in \text{write set} \\
\text{then write}(\text{copies}[n], i, v) \\
\text{else} ( \\
\quad m := \text{copy}(n) \\
\quad \text{copies}[n] := m; \\
\quad \text{write set} := \text{write set} \cup \{n\}; \\
\quad \text{write}(\text{copies}[n], i, v)))
\]
Read Phase

\[ n = \text{tcreate} \]
\[ \text{twrite}(n, i, v) \]
\[ \text{value} = \text{tread}(n, i) \]
  – Read from either the local write set or the database

\[
\text{tread}(n, i) = ( \\
\text{read set} := \text{read set} \cup \{n\}; \\
\text{if } n \in \text{write set} \\
\text{then return } \text{read}(\text{copies}[n], i) \\
\text{else} \\
\text{return } \text{read}(n, i))
\]
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

\[
\begin{align*}
tdelete(n) &= ( \\
& \text{delete set} := \text{delete set} \cup \{n\}).
\end{align*}
\]
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”

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

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
- $t(i) < t(j) \implies$ exists a serial schedule where $T_i$ is before $T_j$
- If execution does not obey this order, the validating transaction aborts
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
Serial Validation

\[
t_{\text{begin}} = (\\n\text{start } t_n := tnc)\\n\]
\[
t_{\text{end}} = (\\n\text{finish } t_n := tnc;\\n\text{valid} := \text{true};\\n\text{for } t \text{ from } \text{start } t_n + 1 \text{ to } \text{finish } t_n \text{ do}\\n\text{if (write set of transaction with transaction number } t \text{ intersects read set)}\\n\text{then valid := false;}\\n\text{if valid}\\n\text{then ((write phase); } tnc := tnc + 1; t_n := tnc));\\n\text{if valid}\\n\text{then (cleanup)}\\n\text{else (backup)).}
\]

Critical Section

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

\[
t\begin{align*}
\text{tbegin} &= ( \\
&\quad \text{start } t_n := t_{nc}) \\
\text{tend} &= ( \\
&\quad (\text{finish } t_n := t_{nc}; \\
&\quad \text{valid} := \text{true}; \\
&\quad \text{for } t \text{ from start } t_n + 1 \text{ to finish } t_n \text{ do} \\
&\quad \quad \text{if (write set of transaction with transaction number } t \text{ intersects read set) } \\
&\quad \quad \quad \text{then } \text{valid} := \text{false;} \\
&\quad \quad \text{if } \text{valid} \\
&\quad \quad \quad \text{then } ((\text{write phase}); t_{nc} := t_{nc} + 1; t_n := t_{nc}); \\
&\quad \quad \text{if } \text{valid} \\
&\quad \quad \quad \text{then } (\text{cleanup}) \\
&\quad \quad \text{else } (\text{backup})).
\end{align*}
\]

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

Part of the validation process happens outside the critical section

The optimization can be applied repeatedly

Readonly transactions do not enter the critical section

\[
t_{\text{end}} := (\text{\
mid t_n := t_{\text{nc}};\\
\text{valid} := \text{true};\\
\text{for } t \text{ from } \text{start } t_n + 1 \text{ to } \text{mid } t_n \text{ do}\\
\text{if (write set of transaction with transaction number } t \text{ intersects read set)}\\
\quad \text{then valid} := \text{false};\\
\text{finish } t_n := t_{\text{nc}};\\
\text{for } t \text{ from } \text{mid } t_n + 1 \text{ to } \text{finish } t_n \text{ do}\\
\quad \text{if (write set of transaction with transaction number } t \text{ intersects read set)}\\
\quad \quad \text{then valid} := \text{false};\\
\text{if valid}\\
\quad \text{then (write phase); } t_{\text{nc}} := t_{\text{nc}} + 1; t_n := t_{\text{nc}};\\
\text{else (backup)}).\]
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

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

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))).
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?
Submit review for