CS 564: Database Management Systems

Lecture 32: OCC and MVCC

Xiangyao Yu

4/12/2024

Module B4 Transactions

Concurrency control

OCC and MVCC

Logging

ARIES recovery

Outline

Optimistic concurrency control (OCC)

Multi-version concurrency control (MVCC)

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release all locks together only after the transaction commits or aborts
- Adopted in practical 2PL implementations

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release all locks together only after the transaction commits or aborts
- Adopted in practical 2PL implementations

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 all locks together only after the transaction commits or aborts
- Adopted in practical 2PL implementations

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 needed only if contention exists

Optimistic Concurrency Control (OCC) [1]

Key idea: Ignore conflicts during a transaction's execution and **resolve conflicts lazily** only at a transaction's completion time

[1] Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control." *ACM Transactions on Database Systems (TODS)* 6.2 (1981): 213-226.

Optimistic Concurrency Control (OCC) [1]

Key idea: Ignore conflicts during a transaction's execution and **resolve conflicts lazily** only at a transaction's completion time

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Read(B)	
commit	
	commit

This execution is serializable but does not happen in 2PL

[1] Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control." *ACM Transactions on Database Systems (TODS)* 6.2 (1981): 213-226.

Optimistic Concurrency Control (OCC) [1]

Key idea: Ignore conflicts during a transaction's execution and **resolve conflicts lazily** only at a transaction's completion time

Txn T1		Txn T2
Read(A)	conflict	
		Write(A)
		Read(B)
Read(B)		
commit		
		commit

This execution is serializable but does not happen in 2PL

T2 conflicts with T1

- T2.Write(A) cannot be executed until T1 releases lock

[1] Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control." *ACM Transactions on Database Systems (TODS)* 6.2 (1981): 213-226.

Optimistic Concurrency Control (OCC) [1]

Key idea: Ignore conflicts during a transaction's execution and **resolve conflicts lazily** only at a transaction's completion time

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Read(B)	
commit	
	commit

This execution is serializable but does not happen in 2PL

This execution can happen in OCC!

[1] Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control." *ACM Transactions on Database Systems (TODS)* 6.2 (1981): 213-226.

Optimistic Concurrency Control (OCC) [1]

Key idea: Ignore conflicts during a transaction's execution and **resolve conflicts lazily** only at a transaction's completion time

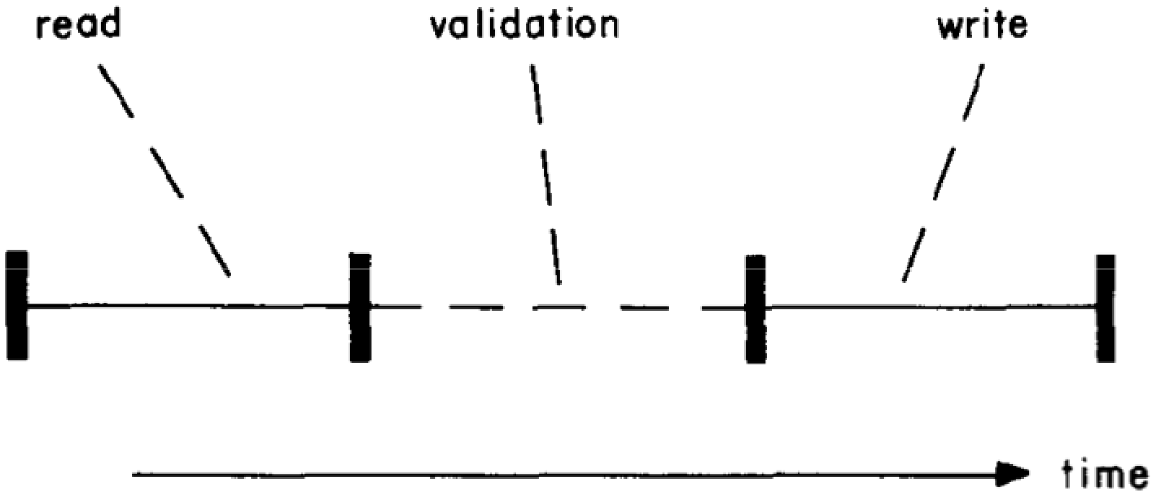


Fig. 1. The three phases of a transaction.

Read Phase: read data from DB; writes stay in transaction-local write buffer

Validation Phase: check whether the transaction violates serializability

Write Phase: copy writes from local buffer to the DB

[1] Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control." *ACM Transactions on Database Systems (TODS)* 6.2 (1981): 213-226.

Read Phase

write(key, value)

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

```
write(key, value):
  if key in write_set
    write_set[key] = value
  else
    write_set.add(key, value)
```

Read Phase

write(key, value)

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

read(key)

- Read from local write_set if exists
- Otherwise read from database

```
write(key, value):
  if key in write_set
    write_set[key] = value
  else
    write_set.add(key, value)
```

```
read(key):
  if key in write_set
    return write_set[key]
  else
    return DB[key]
```

Read Phase

write(key, value)

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

read(key)

- Read from local write_set if exists
- Otherwise read from database

```
write(key, value):
  if key in write_set
    write_set[key] = value
  else
    write_set.add(key, value)
```

```
read(key):
  if key in write_set
    return write_set[key]
  else
    return DB[key]
```

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

Write Phase

All written values become visible in the database

```
foreach key in write_set:
  DB[key] = write_set[key]
```

All inserts and deletes are also reflected in the database

Validation Phase

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Validation Phase

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Read(B)	
validate	

Validation Phase

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Read(B)	
validate	
commit	

Validation Phase

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Read(B)	
validate	
commit	
	validate
	commit

does not violate serializability; equivalent to T1 -> T2

Validation Phase — Abort Example

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Write(B)	
validate	
commit	

Validation Phase — Abort Example

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Write(B)	
validate	
commit	
	validate
	abort

committing T2 would violate serializability!
must abort T2

Validation Phase — Abort Example

Does the current transaction T violate serializability with respect to transactions that have committed?

- If serializable: Commit T
- Otherwise: Abort T

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Write(B)	
validate	
commit	
	validate
	abort

Key Question: How to validate serializability of a transaction?

committing T2 would violate serializability!
must abort T2

Validation Phase

```
start_tn = tnc
execute read phase
```

tnc: transaction number counter; a global monotonically increasing counter

Validation Phase

current transaction: *c*

All transactions that committed after *c* starts

```
start_tn = tnc
execute read phase
finish_tn = tnc
for txn t from start_tn + 1 to finish_tn:
  if current txn’s read_set intersects t’s write_set:
    abort()
```

Validation Phase

```
start_tn = tnc
execute read phase
finish_tn = tnc
for txn t from start_tn + 1 to finish_tn:
  if current txn's read_set intersects t's write_set:
    abort()
```

current transaction: *c*

All transactions that committed after *c* starts

c reads a record that *t* writes to

But the *c* may not see *t*'s write because *c* started before *t* commits

Validation Phase

```
start_tn = tnc
execute read phase
finish_tn = tnc
for txn t from start_tn + 1 to finish_tn:
  if current txn's read_set intersects t's write_set:
    abort()
```

current transaction: *c*

All transactions that committed after *c* starts

c reads a record that *t* writes to

But the *c* may not see *t*'s write because *c* started before *t* commits

abort *c* because it may violate serializability

Validation Phase

```
start_tn = tnc
execute read phase
finish_tn = tnc
for txn t from start_tn + 1 to finish_tn:
  if current txn's read_set intersects t's write_set:
    abort()
if validation passes:
  execute write phase
  tnc = tnc + 1
  tn = tnc
```

If validation phase passes, apply all writes and commit c with transaction number tn = tnc and increment tnc

Validation Phase

```
start_tn = tnc
execute read phase
finish_tn = tnc
for txn t from start_tn + 1 to finish_tn:
  if current txn's read_set intersects t's write_set:
    abort()
if validation passes:
  execute write phase
  tnc = tnc + 1
  tn = tnc
```

Critical section: this code can be executed by at most one transaction at a time

- Optimization techniques exist to parallelize this code

Validation Example

Txn T1	Txn T2
Read(A)	
	Write(A)
	Read(B)
Write(B)	
validate	
commit	
	validate
	abort

T2 aborts because T2's read_set intersects with write_set of T1 (which has already committed); execution cannot be equivalent to T1->T2

Validation Example

Txn T1	Txn T2	Txn T3
write(A)		
commit		
	read(B)	
	write(C)	
		write(A)
		read(B)
	commit	
		validate

Can T3 commit?

Validation Example

Txn T1	Txn T2	Txn T3
write(A)		
commit		
	read(B)	
	write(C)	
		write(A)
		read(B)
	commit	
		commit

Can T3 commit?

T3 starts after T1 commits, so T3 can see all writes by T1; T3 is validated against only T2 and the validation passes

Outline

Optimistic concurrency control (OCC)

Multi-version concurrency control (MVCC)

MVCC – Motivation

Txn T1	Txn T2	Txn T3
read(A)		
	write(A)	
read(B)		
		write(B)
read(C)		
read(D)		

In 2PL?

MVCC – Motivation

Txn T1	Txn T2	Txn T3
read(A)		
	write(A)	
read(B)		
		write(B)
read(C)		
read(D)		

In 2PL? T2 and T3 waits for T1 to commit

MVCC – Motivation

Txn T1	Txn T2	Txn T3
read(A)		
	write(A)	
read(B)		
		write(B)
read(C)		
read(D)		

In 2PL? T2 and T3 waits for T1 to commit

In OCC?

MVCC – Motivation

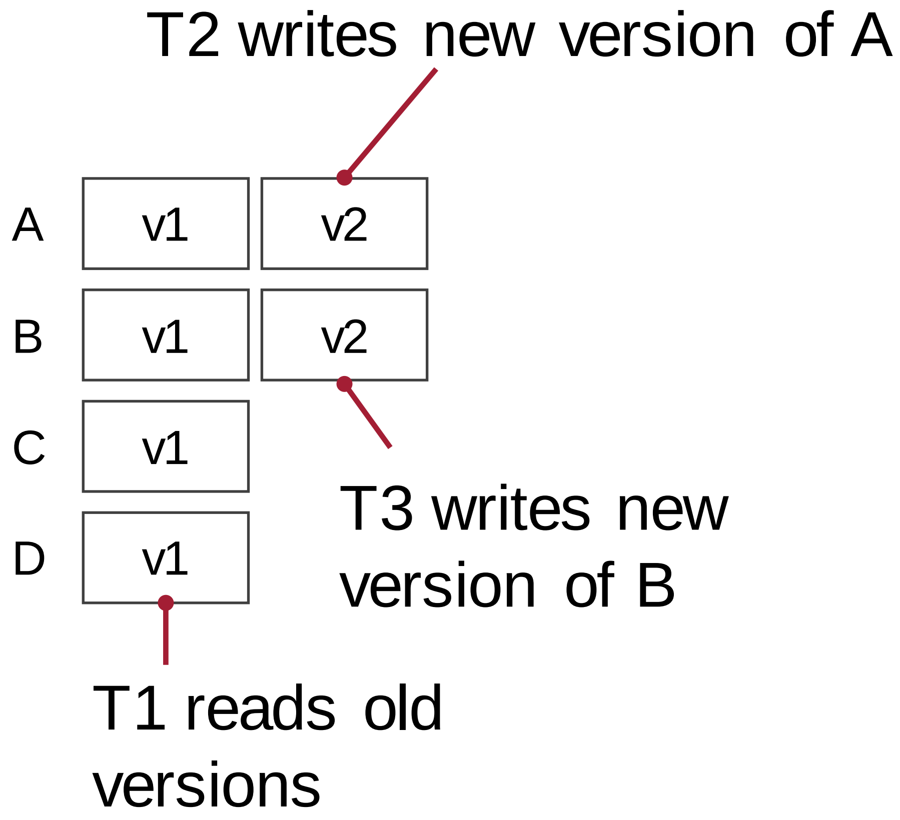
Txn T1	Txn T2	Txn T3
read(A)		
	write(A)	
read(B)		
		write(B)
read(C)		
read(D)		

In 2PL? T2 and T3 waits for T1 to commit

In OCC? T2 and T3 commits but T1 aborts

MVCC – Motivation

Txn T1	Txn T2	Txn T3
read(A)		
	write(A)	
read(B)		
		write(B)
read(C)		
read(D)		



If each write creates a new version of the record, then T1 can read old versions. All transactions can commit without waiting.

- Equivalent to T1->T2->T3

MVCC – Implementation

Each transaction is assigned a timestamp (*ts*) when the transaction starts

- ts increases monotonically and each transaction has a unique ts
- Timestamp order determines the serialization order

MVCC – Implementation

Each transaction is assigned a timestamp (*ts*) when the transaction starts

- ts increases monotonically and each transaction has a unique ts
- Timestamp order determines the serialization order

Each record version has a write timestamp (wts) and a read timestamp (rts)

- Different versions of the same record have non-overlapping (wts, rts) ranges

A	v1 (wts=0, rts=10)	v2 (wts=11, rts=20)
B	v1 (wts=0, rts=15)	v2 (wts=16, rts=25)

MVCC – Implementation

Each transaction is assigned a timestamp (*ts*) when the transaction starts

- ts increases monotonically and each transaction has a unique ts
- Timestamp order determines the serialization order

Each record version has a write timestamp (wts) and a read timestamp (rts)

- Different versions of the same record have non-overlapping (wts, rts) ranges

A transaction can read a version if wts ≤ ts ≤ rts

A	v1 (wts=0, rts=10)	v2 (wts=11, rts=20)
B	v1 (wts=0, rts=15)	v2 (wts=16, rts=25)
C	v1 (wts=0, rts=5)	
D	v1 (wts=4, rts=12)	

If T1.ts = 4, T1 can read the first version of all records

MVCC – Read

Record A	v1 (0, 10)	v2 (11, 20)	v3 (21, 30)

Transaction T (timestamp = ts) reads A

If exists a version where wts ≤ ts ≤ rts, return this version

MVCC – Read

Record A	v1 (0, 10)	v2 (11, 20)	v3 (21, 30)

Transaction T (timestamp = ts) reads A

If exists a version where wts ≤ ts ≤ rts, return this version

otherwise ts should be greater than rts of the last_version

    return last version

    last_version.rts = ts

A read does not cause abort !!

MVCC – Write

Record A | v1 (0, 10) | v2 (11, 20) | v3 (21, 30)

Transaction T (timestamp = ts) writes A

If exists a version where wts ≤ ts ≤ rts

    abort

MVCC – Write

Record A	v1 (0, 10)	v2 (11, 20)	v3 (21, 30)

Transaction T (timestamp = ts) writes A

If exists a version where wts ≤ ts ≤ rts

    abort

otherwise ts should be greater than rts of the cur_last_version

    cur_last_version.rts = ts-1

    create a new version where wts=rts=ts

A write and a read do not block each other !!

Summary

Optimistic concurrency control (OCC)

- Read phase
- Validation phase
- Write phase

Multi-version concurrency control (MVCC)

- Record versioning
- Read and write operations