Database Management System

UNIT II: Chapter 6: Concurrency ControlTechniques

Lock and unlock operations for binary locks

```
lock_item(X):
B:  if LOCK(X) = 0                        (* item is unlocked *)
        then LOCK(X) ←1                   (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                          (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

In binary locks every transaction must obey the following rules:

1. A transaction T must issue the operation 1ock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a 1ock_item(X) operation if it already holds the lock on item X.
4. A transaction T will not issue an unlock_item*(X) operation unless it already* holds the lock on item X.

Locking and unlocking operations for two mode locks (Read/Write Locks)

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                no_of_reads(X) ← 1
                end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
```

```
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
```

```
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Shared/Exclusive locking scheme

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any wri te_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.

Shared/Exclusive locking scheme

4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be **relaxed**.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be **relaxed**.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Does not guarantee serializability

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

A nonserializable schedule using locks

(c)

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Time

Result of schedule S:
X=50, Y=50
(nonserializable)

2PL [schedule is guaranteed to be serializable]

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Conversion of Locks: Relax 4 and 5

- **upgrade the lock**
- **downgrade the lock**

- **If lock conversion is allowed in 2PL, then**

  –upgrading of locks (from read-locked to write-locked) must be done during the **expanding phase**.

  –downgrading of locks (from write-locked to read-locked) must be done in the **shrinking phase**.

2PL and Serializability

- Two-phase locking protocol guarantees serializability.

- The use of locks can cause two additional problems:

  1. Deadlock and

  2. Starvation

Variations of 2PL

Basic,

Conservative,

Strict,

Rigorous

Conservative 2PL

A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to **lock** all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set.

If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it **waits** until all the items are available for locking.

Conservative 2PL is a **deadlock-free** protocol.

Strict 2PL

A transaction T does not release any of its **exclusive** (write) **locks** until after it commits or aborts.

Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a **strict schedule** for recoverability.

Strict 2PL is **not deadlock-free**.

Rigorous 2PL

In **Rigorous 2PL** a transaction T does not release any of its **locks (exclusive or shared)** until after it commits or aborts.

Rigorous 2PL also guarantees **strict schedules**.

Conservative v.s. Rigorous 2PL

Conservative 2PL must lock all its items before it starts so once the transaction starts it is in its **shrinking phase**.

Rigorous 2PL does not unlock any of its items until after it terminates (by committing or aborting) so the transaction is in its **expanding phase** until it ends.

2PL Example

Consider the following two transactions:

T1: read(A);
read(B);
if then B := B + 1;
write(B).
	T2: read(B);
	read(A);
	if then A := A + 1;
	write(A).

Add lock and unlock instructions to transactions T1 and T2, so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

2PL Example

Consider the following two transactions:

T1= w1(C) r1(A) w1(A) r1(B) w1(B)

T2= r2(B) w2(B) r2(A) w2(A)

Assume that the scheduler uses exclusive locks only. For each of the following instances involving T1 and T2, annotated with lock and unlock actions.

2PL Example cont…

Instance A

T1= L1(C) w1(C) L1(A) r1(A) w1(A) U1(A) L1(B) r1(B) w1(B) U1(C) U1(B)

T2= L2(B) r2(B) w2(B) U2(B) L2(A) r2(A) w2(A) U2(A)

Instance B

T1= L1(C) w1(C) L1(A) r1(A) w1(A) L1(B) r1(B) w1(B) COMMIT U1(A) U1(C) U1(B)

T2= L2(B) r2(B) w2(B) L2(A) r2(A) w2(A) COMMIT U2(A) U2(B)

Is it Serializable, 2PL, Strict 2PL?

2PL Example cont…

Instance C

T1= L1(C) L1(A) w1(C) r1(A) w1(A) L1(B) r1(B) w1(B) COMMIT U1(A) U1(C) U1(B)

T2= L2(B) r2(B) w2(B) L2(A) r2(A) w2(A) COMMIT U2(A) U2(B)

Instance D

T1= L1(B) L1(C) w1(C) L1(A) r1(A) w1(A) r1(B) w1(B) COMMIT U1(A) U1(C) U1(B)

T2= L2(B) r2(B) w2(B) L2(A) r2(A) w2(A) COMMIT U2(A) U2(B)

Is it Serializable, 2PL, Strict 2PL?

Timestamping

Timestamp: a unique identifier created by DBMS that indicates relative starting time of a transaction.

Can be generated by:

–using system clock at time transaction started, or

–incrementing a logical counter every time a new transaction starts.

Timestamp based concurrency control

Does not use any locks for the serializability of schedules. Hence, **no deadlocks**

Each transaction will be assigned a timestamp value, **TS(Ti)** and based upon this the transactions are executed

The timestamp ordering protocol ensures that any **conflicting read and write operations** are executed in timestamp order.

This order decides proper **serializability** of schedules

Notations

W-TS(X): is the largest timestamp of any transaction that executed W(X) successfully

R-TS(X): is the largest timestamp of any transaction that executed R(X) successfully

TS(Ti): Timestamp value of Transaction Ti

Basic Time stamp protocol/alg

```
if Ti requests read(X) { if
      TS(Ti) < W-TS(X) {
    rollback;
    }
    else {
    read(X);
    R-TS(X) = max(R-TS(X), TS(Ti));
    }
 }
```

Basic Time stamp protocol/alg

- **if Ti requests write(X)** {

```
        if (TS(Ti) <R-TS(X))||(TS(Ti) <  W-TS(X)) {
        rollback;
        }
        else {
        write(X);
        W-TS(X) =  TS(Ti);
        }
}
```

Strict Timestamp Ordering

1. **Transaction T issues a write_item(X) operation:**

   If TS(T) > R-TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

2. **Transaction T issues a read_item(X) operation:**

   If TS(T) > W-TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Thomas' Write Rule

Observation: if TS(Ti) < TS(Tj), and Ti writes X before TS(Tj), then we can just ignore that write, since it will be overwritten by Tj anyway

Thomas' write rule is a protocol tweak on the write(X) case based on the observation:

```
...
if TS(Ti) < W-TS(X) {
ignore write(X);
}
```

Resources

Chapter 22 of Fundamentals of Database Systems (FODS), 6th Edition.

Internet Surf

(Most of the slides adapted from Ramez Elmasri and Shamkant Navathe , FODS)