KNU CSE
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DATA KNOWLEDGE ENGINEERING LABORATORY

TRANSACTION PROCESSING II

Chapter 21
- Concurrency control

Prof. Young-Kyoon Suh

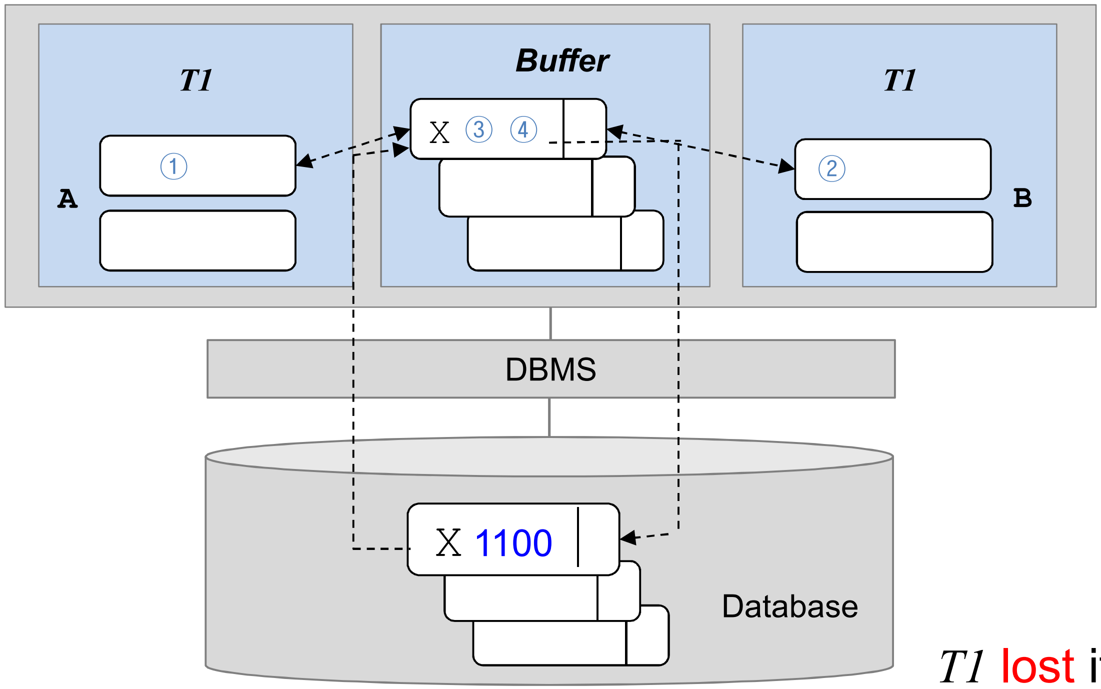
Introduction

- Concept of *Concurrency Control*
  - If concurrent transactions have access to the same data item, concurrency control techniques effectively orchestrates the access control of the item, not to harm consistency of the item.
  - Defines a set of rules to guarantee serializability
- Scenarios of read/write from two concurrent transactions: *T1* and *T2*

	T1	*T2*	Problem	Concurrency
Scenario 1	Read	Read	No	Allowed
Scenario 2	Read	**Write**	*Dirty read*, *Non-repeatable read*, *Phantom read*	Allowed or Disallowed along with isolation level
Scenario 3	**Write**	**Write**	*Lost Update* (never allowed)	Disallowed; but possible with ***locks***

Introduction – Lost Update (Cont'd)

T1	*T2*	Buffer Data
`A=read_item(X);` ① `A=A-100;`		X=1000
	`B=read_item(X);` ② `B=B+100;`	X=1000;
`write_item(A->X);` ③		X=**900**
	`write_item(B->X);` ④	X=**1100**



T1 lost its update by *T2*.

Introduction (Cont'd)

- To solve the *lost update* problem, use **locking** techniques.
  - ***Lock***: a *variable* associated with a data item
    - Describes the status of the item, with respect to possible operations that can be applied to it
      - E.g., "My item is locked. You cannot use mine right now."
    - Used as a means of ***synchronizing*** the access by concurrent transactions to the database items.
    - One lock for each item in the database

T1

T2

Time

```
SQL> SET TRANSACTION NAME 'T1';

Transaction set.

SQL>
SQL> SELECT *
  2  FROM BOOK
  3  WHERE bookid = 1;

    BOOKID BOOKNAME   PUBLISHER                       PRICE
---------- ---------- -------------------- ----------
         1 Database   Pearson                         20000

SQL> UPDATE BOOK
  2  SET price = 21000
  3  WHERE bookid = 1;

1 row updated.

SQL>
```

Now the Book tuple is locked by T1.

```
SQL> SET TRANSACTION NAME 'T2';

Transaction set.

SQL>
SQL> SELECT *
  2  FROM BOOK
  3  WHERE bookid = 1;

    BOOKID BOOKNAME   PUBLISHER                       PRICE
---------- ---------- -------------------- ----------
         1 Database   Pearson                         20000

SQL> UPDATE BOOK
  2  SET price = 21000
  3  WHERE bookid = 1;
```

```
SQL> SELECT *
  2  FROM BOOK
  3  WHERE bookid = 1;

    BOOKID BOOKNAME   PUBLISHER                       PRICE
---------- ---------- -------------------- ----------
         1 Database   Pearson                         21000

SQL>
SQL> commit;

Commit complete.

SQL>
```

T2 waits until the lock on the Book tuple is released.

```
1 row updated.

SQL>
```

TWO-PHASE LOCKING TECHNIQUES FOR CONCURRENCY CONTROL

Chapter 20.1

Types of Locks and System Lock Tables

1) *Binary locks* (used by the binary locking scheme)

- Two states (values)
  - *Locked* (1): Item cannot be accessed
  - *Unlocked* (0): Item can be accessed when requested
- Locking notation: **`lock(X)`**
  - Indicates the current status (value) of the lock associated with item X
- Two operations are used with binary locks:
  - **`lock_item`**
    - Must be invoked *before* use
  - **`unlock_item`**
    - Must be invoked *after* use

```
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;
```

Types of Locks and System Lock Tables (Cont'd)

1) Binary locks (Cont'd)

- Easy to implement via a binary-valued variable, say **`LOCK`**
- Each lock can be a record with three fields:
  <*Data_item_name*, `LOCK`, *Locking_Transaction*>.
- ***Lock table***: implemented as a hash file on the item name
  - Used to maintain these records only for the items currently locked
  - Specifies data items that have locks
- ***Lock manager subsystem*** of the DBMS
  - Keeps track of and controls access to locks
  - Enforces locking rules ("call `lock_item` before use").
- At most one transaction can hold the lock on an item at a given time
- Binary locking too restrictive for database items

Types of Locks and System Lock Tables (Cont'd)

2) ***Shared*/*Exclusive*** (or ***Read*/*Write***) Locks (used by the shared/exclusive locking scheme)

- Called a *multiple-mode* lock
- Three states (values) of `lock(X)`
  - *read-locked* (or shared-locked): Item can be read by other transactions.
  - *write-locked* (or exclusive-locked): A single transaction exclusively holds the lock on the item; Item can **NOT** be accessed for a shared-lock request.
  - *unlocked*: Item can be accessed
- Has three locking operations:
  - `read_lock(X)`, `write_lock(X)`: Must be invoked *before* use
  - `unlock(X)`: Must be invoked *after* use
- Each record in the lock table has four fields:
  - <*Data_item_name*, (read|write) **LOCK**, *# of reads, Locking_Transaction(s)*>.
- System maintains lock records *only for locked items* in the table.
  - If an item not locked, it would not be in the table.

Conversion of Locks

- Transaction that already holds a lock is allowed to “convert” the lock from one state to another.
  - Called *lock conversion*
- *Upgrading*
  - Issues `read_lock(X)` followed by `write_lock(X)`
- *Downgrading*
  - Issues `write_lock(X)` followed by `read_lock(X)`

Problems of the Previous Locking Schemes

- Using binary locks or read/write locks in transactions *doesn't guarantee serializability of schedules* on its own.

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

Too early

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

Allows this schedule, S

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)

Why?

TWO-PHASE LOCKING

- Guarantees Serializability (as well as solves the lost update problem)

Two-Phase Locking Protocol

- All locking operations precede the *first* unlock operation in the transaction: lock requests -> lock releases (never backward).

- Consists of two phases:
  - *Expanding* (growing) phase
    - New locks can be acquired but none can be released
      - Don't release acquired locks
    - Lock conversion *upgrades* must be done during this phase.
  - *Shrinking* phase
    - Existing locks can be released BUT none can be acquired.
      - Once you begin to unlock, then you can't acquire a lock until all the locks are released.
    - *Downgrades* must be done during this phase.

Review: Transactions That Don't Obey 2PL

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

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

Time

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);	

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

Transactions Following 2PL

But deadlock happens

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);

2PL guarantees serializability of these two transactions!

Variations of 2PL

- Basic 2PL: covered
- *Conservative* 2PL
  - Requires a transaction to lock "all" the items it accesses *before the transaction begins execution*
    - How? By predeclaring (i) **read-set** and (ii) **write-set**
  - Waits until needed items are available for locking, if *any* of the predeclared items cannot be locked.
  - Once the transaction starts, it's already in the **shrinking** phase.
    - *Different* from rigorous 2PL to be discussed soon shortly
  - ***Deadlock-free*** protocol; but not in practice; Why?

Variations of 2PL – Strict/Rigorous 2PL (Cont'd)

	Strict 2PL	***Rigorous*** 2PL
Common	Guarantees *strict* schedules (in terms of recoverability)	
	NOT deadlock-free	
Differences	T does *not* releases *write* locks *until after* T commits or aborts.	T does *not* releases *any* (read or write) locks *until after* T commits or aborts.
	No other transaction can read/write an item written by T *unless* T *has committed*	
	Harder to implement than rigorous 2PL	*Easier* to implement than rigorous 2PL
	-	In the **expanding** phase until transaction ends *- Different from conservative 2PL*

HOW TO RESOLVE DEADLOCK?

- Deadlock prevention
- Deadlock detection

Dealing with *Deadlock*

- Locking reveals <u>three</u> major problems:
  - 1) A high overhead (lock request followed by read or write), 2) Deadlock, and 3) Starvation
- **Deadlock**:
  - Occurs when each transaction T in a set is waiting for some item locked by some other transaction T' in the set.
  - Both transactions are *stuck* in a waiting queue

time

T_1'	T_2'
read_lock(Y); read_item(Y);	
	read_lock(X); read_item(X);
write_lock(X);	
	write_lock(Y);

[A partial schedule of T_1' and T_2']

Dealing with Deadlock: ***Deadlock Prevention*** Protocols

1) *Conservative 2PL*: deadlock-free
   - Impractical; Limits concurrency

2) *Another protocol*:
   - *Orders all the items* in the database, and makes sure that a transaction that needs several items will lock them according to that order.
     - Limits concurrency as well
     - The programmer (or the system) is aware of the chosen order of the items, which is also impractical

3) *Timestamp(TS)-based protocol*:
   - Suppose that for 2 transactions (T_1, T_2), T_1 tries to lock an item locked by T_2.
   - *Wait-die* (scheme): non-preemptive from the perspective of lock requestor
     - If TS(T_1) < TS(T_2), then T_1 waits; otherwise, abort and restart T_1 with same TS.
   - *Wound-wait* (scheme): preemptive from the perspective of lock requestor
     - If TS(T_1) < TS(T_2), then aborts and restart T_2 with same TS; otherwise, T_1 waits.

Dealing with Deadlock: ***Deadlock Prevention*** Protocols (Cont'd)

4) *No waiting algorithm* (not requiring TS)

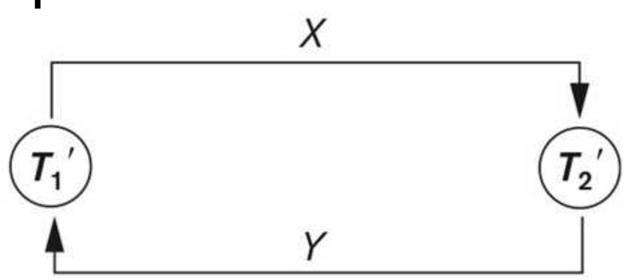
- If transaction is unable to obtain a lock, then it's immediately aborted and then restarted *after a delay*

5) *Cautious waiting algorithm*: **deadlock-free**

- Suppose that T_1 tries to lock an item locked by T_2.
- If T_2 is not blocked (or, not waiting for some other locked item), then T_1 is *blocked and allowed to **wait***; otherwise, T_1 is aborted.
- Designed to reduce # of needless restarts/aborts

Dealing with Deadlock: ***Deadlock Detection***

- The DBMS checks if a state of deadlocks actually exists.
- A simple way: drawing a ***wait-for*** graph
  - Lock wait -> add an edge,
  - Lock release -> drop that edge
  - If a cycle exists, then that's the state of deadlock.
  - Checking for a cycle everytime: *too much overhead*
    - One solution: doing the cycle check periodically, based on (i) *# executing transactions counts* or (ii) *period of time waiting for locking items*
  - *Victim selection*
    - Chooses which transaction to *abort*
    - Criteria: younger transactions that have not made many changes; Why so ?



[A wait-for graph for the partial schedule]

Dealing with Deadlock: ***Timeouts***

- Another simple scheme for deadlock detection
  - Low overhead and simplicity
  - If T times out beyond a system-defined threshold, then T aborted
    - Regardless of whether the deadlock actually exists

Dealing with Deadlock: ***Starvation***

- Occurs when a transaction cannot proceed for infinite time
  - Also occurs for victim selection on the same transaction
  - Solution 1: *first-come*, *first-served*
    - Called a *fair-waiting scheme*
  - Solution 2
    - Giving some priority to transactions so that they can eventually proceed
  - Other solutions:
    - Wait-die, or wound-wait

Summary

- Concept of Concurrency Control
- Lock Types and Lock Conversion
- Two-Phase Locking
- Variation of Two-Phase Locking Schemes
- Deadlock Detection, Prevention, Starvation

APPENDIX

Types of Locks and System Lock Tables (Cont'd)

1) Binary locks (Cont'd)

- Rules enforced by the binary locking scheme
  - A transaction (*T*) must issue `lock_item(X)` before any `read_item(X)` or `write_item(X)`.
  - *T* must issue `unlock_item(X)` after all `read_item(X)` and `write_item(X)` are completed.
  - *T* will not issue `lock_item(X)` if it already holds the lock on item `X`.
  - *T* will not issue `unlock(X)` unless it already holds the lock on item `X`.

Types of Locks and System Lock Tables (Cont'd)

2) ***Shared***/***Exclusive*** (or ***Read***/***Write***) Locks (Cont'd)

- Locking and unlocking operations for two-mode (read/write, or shared/exclusive) 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;
```

Types of Locks and System Lock Tables (Cont'd)

2) ***Shared***/***Exclusive*** (or ***Read***/***Write***) Locks (Cont'd)

- Rules enforced by the shared/exclusive locking scheme
  - A transaction (*T*) must issue `read_lock(X)` or `write_lock(X)` before any `read_item(X)`.
  - *T* must issue `write_lock(X)` before any `write_item(X)`.
  - *T* must issue `unlock(X)` after all `read_item(X)` and `write_item(X)`.
  - *T* will not issue `read_lock(X)` if it already holds a read or a write lock on item `X`.
    - This rule may be relaxed for *downgrading* of locks.
  - *T* will not issue `write_lock(X)` if it already holds a read or a write lock on item `X`.
    - This rule may also be relaxed for *upgrading* of locks.
  - *T* will not issue `unlock(X)` unless it already holds a read lock or a write lock on item `X`.