Concurrency Control Techniques

Introduction

- Concurrency control protocols
  - Set of rules to guarantee serializability
- Two-phase locking protocols
  - Lock data items to prevent concurrent access
- Timestamp
  - Unique identifier for each transaction
- Multiversion currency control protocols
  - Use multiple versions of a data item
- Validation or certification of a transaction

21.1 Two-Phase Locking Techniques for Concurrency Control

- Lock
  - Variable associated with a data item describing status for operations that can be applied
  - One lock for each item in the database
- Binary locks
  - Two states (values)
    - Locked (1)
      - Item cannot be accessed
    - Unlocked (0)
      - Item can be accessed when requested

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Transaction requests access by issuing a lock_item(X) operation

```
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;
```

Figure 21.1 Lock and unlock operations for binary locks

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock table specifies items that have locks
- Lock manager subsystem
  - Keeps track of and controls access to locks
  - Rules enforced by lock manager module
- At most one transaction can hold the lock on an item at a given time
- Binary locking too restrictive for database items

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Shared/exclusive or read/write locks
  - Read operations on the same item are not conflicting
  - Must have exclusive lock to write
  - Three locking operations
    - read_lock(X)
    - write_lock(X)
    - unlock(X)

Figure 21.2 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;
```

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock conversion
  - Transaction that already holds a lock allowed to convert the lock from one state to another
- Upgrading
  - Issue a read_lock operation then a write_lock operation
- Downgrading
  - Issue a read_lock operation after a write_lock operation

Guaranteeing Serializability by Two-Phase Locking

- Two-phase locking protocol
  - All locking operations precede the first unlock operation in the transaction
  - Phases
    - Expanding (growing) phase
      - New locks can be acquired but none can be released
      - Lock conversion upgrades must be done during this phase
    - Shrinking phase
      - Existing locks can be released but none can be acquired
      - Downgrades must be done during this phase

Figure 21.3 Transactions that do not obey two-phase locking (a) Two transactions *T*1 and *T*2 (b) Results of possible serial schedules of *T*1 and *T*2 (c) A nonserializable schedule *S* that uses locks

(a)

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);

(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

(c)

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);	

Guaranteeing Serializability by Two-Phase Locking

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

Variations of Two-Phase Locking

- Basic 2PL
  - Technique described on previous slides
- Conservative (static) 2PL
  - Requires a transaction to lock all the items it accesses before the transaction begins
    - Predeclare read-set and write-set
  - Deadlock-free protocol
- Strict 2PL
  - Transaction does not release exclusive locks until after it commits or aborts

Variations of Two-Phase Locking (cont'd.)

- Rigorous 2PL
  - Transaction does not release any locks until after it commits or aborts
- Concurrency control subsystem responsible for generating read_lock and write_lock requests
- Locking generally considered to have high overhead

Dealing with Deadlock and Starvation

- Deadlock
  - Occurs when each transaction T in a set is waiting for some item locked by some other transaction *T'*
  - Both transactions stuck in a waiting queue

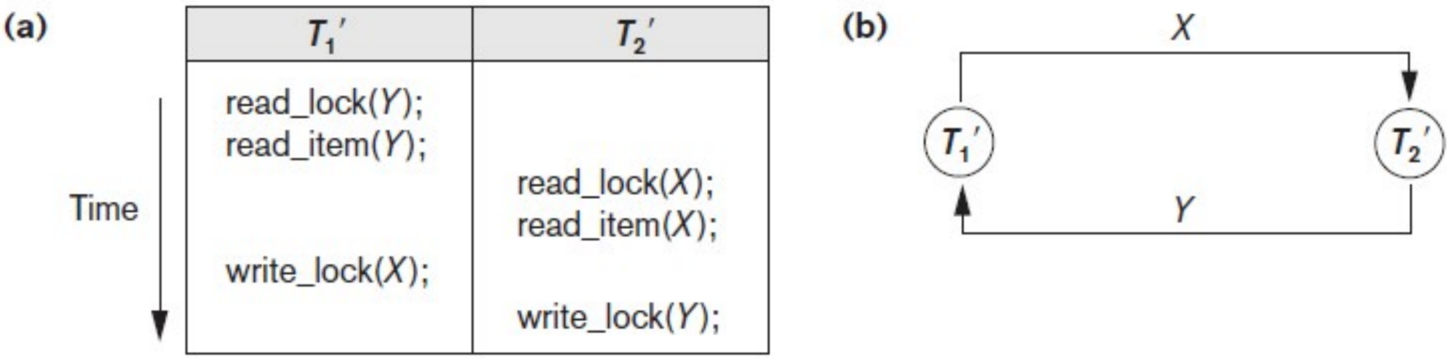

Figure 21.5 Illustrating the deadlock problem (a) A partial schedule of *T1′* and *T2′* that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

Dealing with Deadlock and Starvation (cont'd.)

- Deadlock prevention protocols
  - Every transaction locks all items it needs in advance
  - Ordering all items in the database
    - Transaction that needs several items will lock them in that order
  - Both approaches impractical
- Protocols based on a timestamp
  - Wait-die
  - Wound-wait

Dealing with Deadlock and Starvation (cont'd.)

- No waiting algorithm
  - If transaction unable to obtain a lock, immediately aborted and restarted later
- Cautious waiting algorithm
  - Deadlock-free
- Deadlock detection
  - System checks to see if a state of deadlock exists
  - Wait-for graph

Dealing with Deadlock and Starvation (cont'd.)

- Victim selection
  - Deciding which transaction to abort in case of deadlock
- Timeouts
  - If system waits longer than a predefined time, it aborts the transaction
- Starvation
  - Occurs if a transaction cannot proceed for an indefinite period of time while other transactions continue normally
  - Solution: first-come-first-served queue

21.2 Concurrency Control Based on Timestamp Ordering

- Timestamp
  - Unique identifier assigned by the DBMS to identify a transaction
  - Assigned in the order submitted
  - Transaction start time
- Concurrency control techniques based on timestamps do not use locks
  - Deadlocks cannot occur

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Generating timestamps
  - Counter incremented each time its value is assigned to a transaction
  - Current date/time value of the system clock
    - Ensure no two timestamps are generated during the same tick of the clock
- General approach
  - Enforce equivalent serial order on the transactions based on their timestamps

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Timestamp ordering (TO)
  - Allows interleaving of transaction operations
  - Must ensure timestamp order is followed for each pair of conflicting operations
- Each database item assigned two timestamp values
  - read_TS(X)
  - write_TS(X)

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Basic TO algorithm
  - If conflicting operations detected, later operation rejected by aborting transaction that issued it
  - Schedules produced guaranteed to be conflict serializable
  - Starvation may occur

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Basic TO algorithm
  1. Transaction T issues a write_item(X) operation:
     - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
     - If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).
  2. Transaction T issues a read_item(X) operation:
     - If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
     - If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Strict TO algorithm
  - Ensures schedules are both strict and conflict serializable
  - Transaction T issues a read_item(X) or write_item(X) operation:
    - If TS(T) > write_TS(X), then delay T until the transaction T' that wrote X has terminated (committed or aborted).

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Thomas's write rule
  - Modification of basic TO algorithm
  - Does not enforce conflict serializability
  - Rejects fewer write operations by modifying checks for write_item(X) operation

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Thomas's write rule
  - If read_TS(X) > TS(T) then abort and roll-back T and reject the operation.
  - If write_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
  - If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).