Concurrency Control Techniques

Overview

- Introduction
- Locking Techniques
- Timestamp Ordering Technique

Introduction

- Need for Concurrency Control
- When operations of different transactions are executed concurrently by interleaving of operations, several problems are associated:
  - Lost Update problem
  - Dirty Read (Temporary Update) problem
  - Incorrect Summary problem

Introduction

- **Purpose of Concurrency Control**
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.

Introduction

- To overcome various problems associated with concurrency, several techniques were adopted to control it:
  - Locking Techniques for Concurrency Control
  - Concurrency Control Based on Timestamp Ordering
  - Multiversion Concurrency Control Techniques
  - Validation Techniques for Concurrency Control

Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read or
  - (b) permission to Write a data item for a transaction.
- Example: `Lock (X)`. Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example: `Unlock (X)`. Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Shared / Exclusive Lock

- Two locks modes (a) shared (read) and (b) exclusive (write).
- Shared mode: Shared lock (X).
    - More than one transaction can apply share lock on X for reading its value but *no write lock* can be applied on X by any other transaction.
- Exclusive mode: Write lock (X).
    - Only one write lock on X can exist at any time and *no shared lock* can be applied by any other transaction on X.

	Read	Write
Read	Y	N
Write	N	N

Lock Manager

- Lock Manager: Managing locks on data items.
- Lock table: Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked.
- One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Binary Lock

- Database requires that all transactions should be well-formed.
- A transaction is well-formed if:
  - It must lock the data item before it reads or writes to it.
  - It must not lock an already locked data items and it must not try to unlock a free data item.

Binary Lock

```
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;
```

Shared / Exclusive Lock

- The following code performs the *read* operation:

```
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;
```

Shared / Exclusive Lock

- The following code performs the *write* operation:

```
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;
```

Shared / Exclusive Lock

The following code performs the *unlock* operation:

```
if LOCK (X) = "write-locked" then
        begin LOCK (X) ← "unlocked";
                wakes up one of the transactions, if any
        end
else if 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";
                wake up one of the transactions, if any
            end
        end;
```

Lock Conversion

- **Lock upgrade: existing read lock to write lock**

  If Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then

  convert read-lock (X) to write-lock (X)

  else

  force Ti to wait until Tj unlocks X

- **Lock downgrade: existing write lock to read lock**

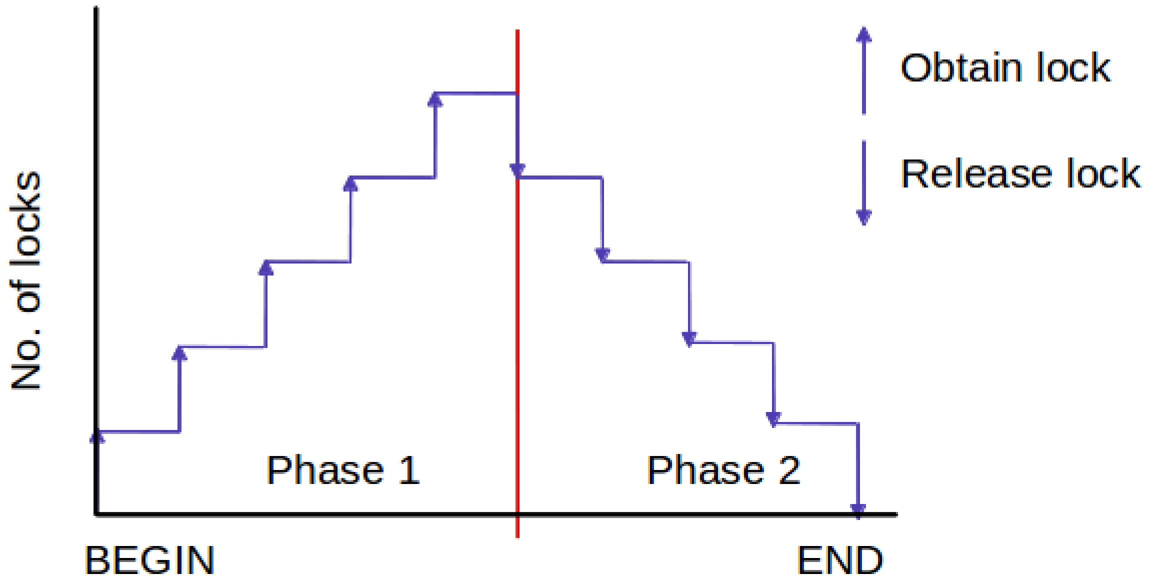
  Ti has a write-lock (X) (*no transaction can have any lock on X*)

  convert write-lock (X) to read-lock (X)

Two-Phase Locking : Algorithm

- **Two Phases**: (a) Locking (Growing) (b) Unlocking (Shrinking).
- **Locking (Growing) Phase**: A transaction applies locks (read or write) on desired data items one at a time.
  - Locks are *acquired*, not released
- **Unlocking (Shrinking) Phase**: A transaction unlocks its locked data items one at a time.
  - Locks are *released*, new locks can not be acquired
- **Requirement:** For a transaction these two phases must be mutually exclusive, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Two-Phase Locking : Algorithm



Two-Phase Locking : Algorithm

(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 of serial schedule T_1 followed by T_2 :
$X=50$, $Y=80$

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

Transactions T1 and T2 that do not obey 2PL

Two-Phase Locking : Algorithm

(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);	

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

A non-serializable schedule that uses locks

Two-Phase Locking : Algorithm

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

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);

Two-Phase Locking : Algorithm

- Two-phase policy generates two locking algorithms (a) Basic and (b) Conservative.
- **Basic 2PL**: Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Conservative 2PL**: Prevents deadlock by locking all desired data items before transaction begins execution. (deadlock-free protocol)
- **Strict 2PL**: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Dealing with Deadlock

Deadlock

T’1	**T’2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 did follow two-phase policy but they are deadlock

Deadlock (T’1 and T’2)

Dealing with Deadlock

- Deadlock Prevention Protocols
- Deadlock Detection
- Starvation

Dealing with Deadlock

- **Deadlock Prevention Protocols**
- Lock all data items before transaction begins → Conservative 2PL – is a deadlock prevention protocol but further limits concurrency.
- Not generally used because of unrealistic assumptions or overhead.
- What to do with a transaction involved in a deadlock?
  - Should it be blocked and made to wait ?
  - Should it be aborted ?
  - Should the transaction *preempt and abort* another transaction ?
- Transaction timestamp – TS(T) – a unique identifier assigned to each transaction.

Dealing with Deadlock

- **Deadlock Prevention Protocols:**
- If Ti tries to lock an item X; but X is locked by Tj, then
- Wait-die

  If $TS(T_i) < TS(T_j)$, then (T_i older than T_j)
  T_i is allowed to wait;

  Else

  abort T_i and restart it later *with the same timestamp;*

- Wound-wait

  If $TS(T_i) < TS(T_j)$, then (T_i older than T_j)
  abort T_j and restart it later *with the same timestamp;*

  Else

  T_i is allowed to wait;

- Issue: some transactions to be aborted and restarted needlessly

Dealing with Deadlock

- **Deadlock Detection**
- In this approach, deadlocks are allowed to happen.
- Suitable for transactions that are short and locks only a few items.
- The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph.
- One of the transaction of the cycle is selected and rolled back – select transactions that have not made many changes.

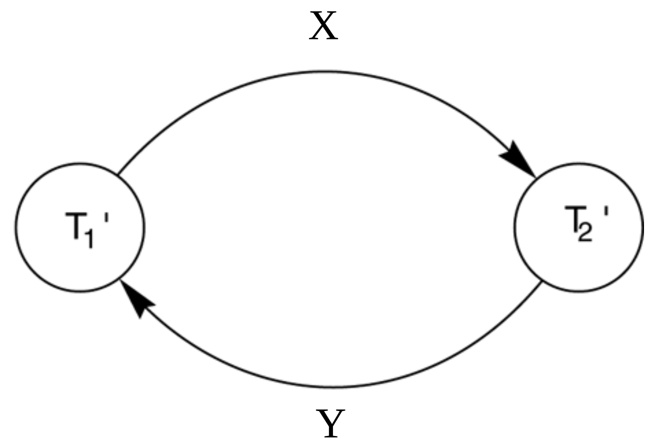
Dealing with Deadlock

(a)

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

Time ↓

(b)



Wait-for graph to detect deadlock

Dealing with Deadlock

- **Starvation**
- Starvation occurs when a particular *transaction consistently waits* or *restarted* and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- Solution: using a *first-come-first-served* queue

Concurrency Control Based on TO

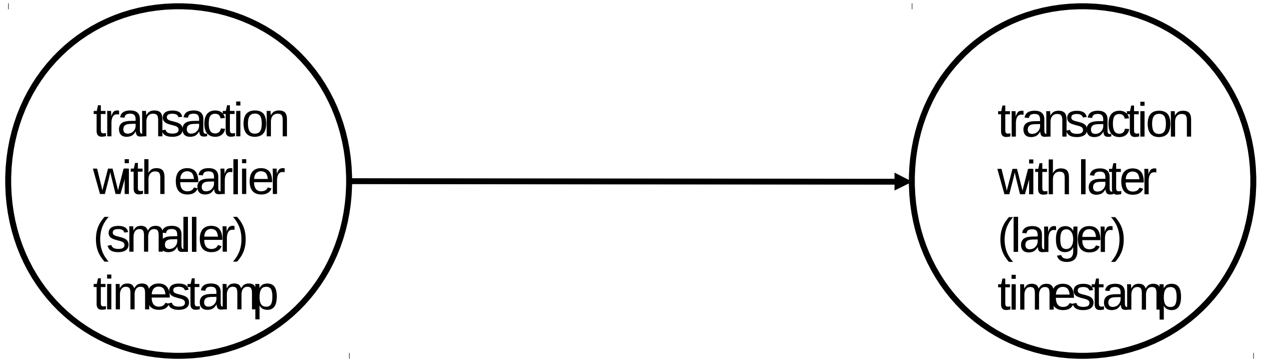
- A timestamp is a unique identifier created by the DBMS to identify a transaction.
- TS(T) refers to timestamp of transaction T.
- Concurrency control techniques based on timestamps do not use locks; hence deadlock *cannot* occur.
- Generation of timestamps by counter, system clock
- A larger timestamp value indicates a more recent event or operation.

Concurrency Control Based on TO

- Ordering of transactions are based on their timestamp value – Timestamp Ordering (TO)
- It uses two timestamp TS values:

read_TS(X): the TS of last transaction which reads data item X successfully.

write_TS(X): the TS of last transaction which writes data item X successfully.



Concurrency Control Based on TO

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

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/written 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 TO

- <u>Issues : Basic Timestamp Ordering</u>
- The schedules produced by Basic TO are *conflict serializable*. It leads to cascading rollback problem.
- Basic TO doesn't ensure recoverable schedules; and hence it doesn't ensure cascadeless schedules or *strict schedules*.
- A variation of basic TO called **strict TO** ensures that the schedules are both strict and serializable

Concurrency Control Based on TO

Timestamp based concurrency control algorithm

Strict Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   - If TS(T) > read_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) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Concurrency Control Based on TO

Timestamp based concurrency control algorithm

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).

References

- Fundamentals of Database Systems, Elmasri and Navathe, 5th Edition


THANK YOU!