Concurrency Control

Dr. Jeevani Goonetillake

Database Concurrency Control

- 1 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.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Classification of Techniques:

1. **Locking** data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols have been proposed.
2. **Use of timestamps**. A timestamp is a unique identifier for each transaction, generated by the system.
3. **Multiversion** concurrency control protocols that use multiple versions of a data item.
4. **Optimistic Concurrency Control:** based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols.**

Locking

- A **lock**: a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
- Generally, there is one lock for each data item in the database.
- **Granularity of locking** varies : typically rows or sets of rows. An entire relation may be locked, or an entire database.

Types of Locks

- **Binary locks:** only two states of a lock; too simple and too restrictive; not used in practice.
- **Shared/exclusive locks:** which provide more general locking capabilities and are used in practical database locking schemes. (Read Lock as a shared lock, Write Lock as an exclusive lock).
- **Certify lock:** used to improve performance of locking protocols.

Binary Locks

A **binary lock** can have two **states** or **values**:
locked and unlocked (or 1 and 0, for simplicity).

A **binary** lock enforces **mutual exclusion** on the data item;
i.e., at a time only one transaction can hold a lock.

A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item.

If the value of the lock on X is 0, the item can be accessed when requested.

Binary Locks

If LOCK (X) = 1, the transaction is forced to wait.

If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X.

unlock_item(X) : sets LOCK(X) to 0 (**unlocks** the item) so that X may be accessed by other transactions.

Binary Locking Scheme

Every transaction must obey the following rules. Rules are enforced by the **LOCK MANAGER**

1. A transaction T must issue the operation lock_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 lock_item(X) operation if it already holds the lock on item X.
4. A transaction T will not issue an unlock_item(X) operation on X unless it already holds the lock on item X.

Binary Locks

The following code performs the lock operation:

```
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);
   goto B
   end;
```

Binary Locks

The following code performs the unlock operation:

LOCK (X) ← 0 (*unlock the item*)

if any transactions are waiting then

wake up one of the waiting the transactions;

Shared/Exclusive (or Read/Write) locks

- A lock associated with an item X,LOCK(X), now has three possible states:

  "read-locked," "write-locked," or "unlocked."

- A **read-locked item** is also called **share-locked**, because other transactions are allowed to read the item.
- A **write-locked item** is called **exclusive-locked**, because a single transaction exclusively holds the lock on the item.

Shared/Exclusive (or Read/Write) locks

– Two locks modes:
  - (a) shared (read) (b) exclusive (write).

– Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Shared/Exclusive (or Read/Write) locks

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 (or Read/Write) locks

The following code performs the write lock 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 (or Read/Write) locks

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;
```

Shared/Exclusive (or Read/Write) locks

RULES FOR Read/Write LOCKS

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 write_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 (or Read/Write) locks

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.
   (EXCEPTIONS: DOWNGRADING OF LOCK from WRITE TO READ)

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.
   (EXCEPTIONS: UPGRADING OF LOCK FROM READ TO WRITE)

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.

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 ≠ 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

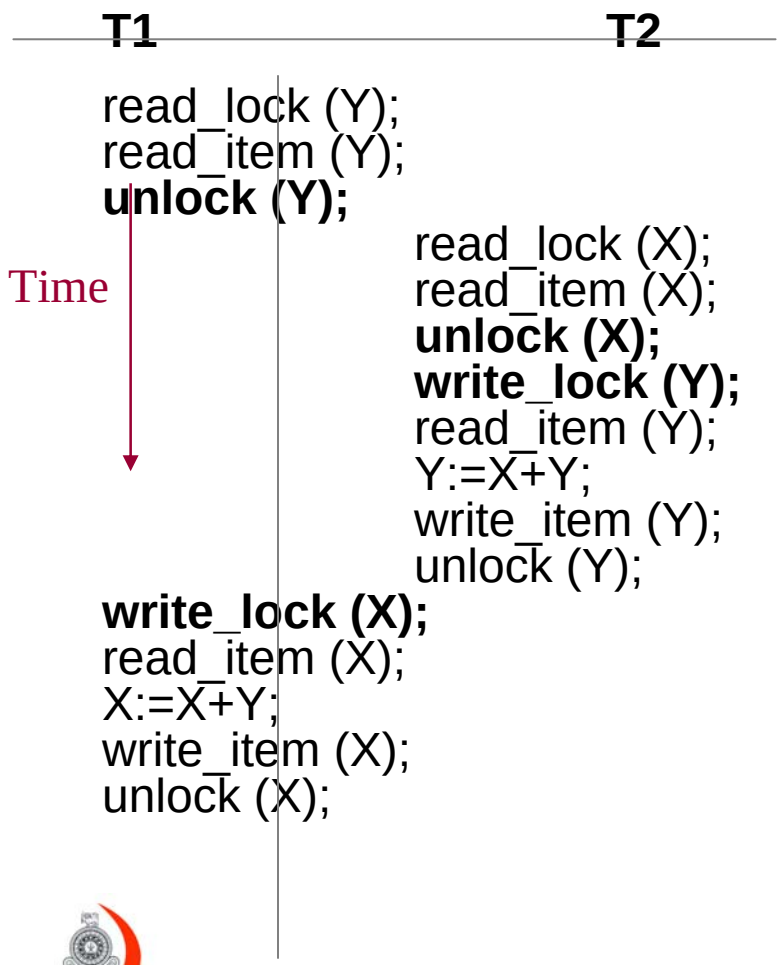
Ti has a write-lock (X) (*no transaction can have any lock on X*)

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

Database Concurrency Control

T1	T2	Result
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Database Concurrency Control



Result

X=50; Y=50

Two-Phase Locking Techniques

- 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.
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time.
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Two-Phase Locking Techniques

T'1	**T'2**	
read_lock (Y);	read_lock (X);	T1 and T2 follow two-phase
read_item (Y);	read_item (X);	policy but they are subject to
write_lock (X);	Write_lock (Y);	deadlock, which must be
unlock (Y);	unlock (X);	dealt with.
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 Techniques

- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A stricter version of Basic, where X-unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.
- **Rigorous:**
  - Like s2PL, but **all** unlocking is performed upon termination.

Limitations Of 2 PL

1. The two-phase locking protocol guarantees serializability but it does not permit *all possible* serializable schedules.

2. Use of **locks** can cause two additional problems: deadlock and starvation.

Deadlock

– **Deadlock**

T'1	**T'2**	
read_lock (Y);		T1 and T2 did follow
two-phase		
read_item (Y);		policy but they are
deadlock		
	read_lock (X);	
	read_item (Y);	
write_lock (X);		
(waits for X)	write_lock (Y);	
	(waits for Y)	

– Deadlock (T'1 and T'2)

Deadlock

Deadlock prevention

- A transaction locks all data items it refers to before it begins execution.
- This way of locking prevents deadlock since a transaction never waits for a data item.
- The conservative two-phase locking uses this approach.

Deadlock

- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. 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. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.

Deadlock

- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Stravation

- 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.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Timestamp

– A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.

– Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

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 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).

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).

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).

Validation (Optimistic) Concurrency Control Schemes

In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.

- Three phases:
  1. **Read phase**
  2. **Validation phase**
  3. **Write phase**

1. **Read phase**:
   - A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).

Validation (Optimistic) Concurrency Control Schemes

2. **Validation phase**: Serializability is checked before transactions write their updates to the database.
   - This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
     - Tj completes its write phase before Ti starts its read phase.
     - Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj

Validation (Optimistic) Concurrency Control Schemes

- Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase.

- When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and Ti is aborted.

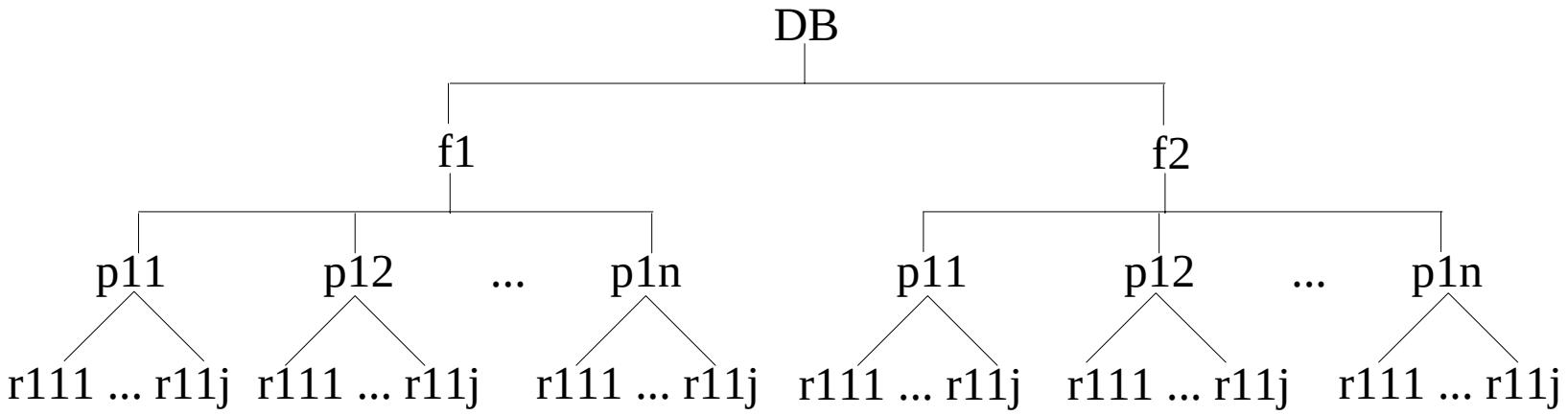
3. **Write phase**: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

Granularity of data items and Multiple Granularity Locking

- A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation).
- Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity.
- Example of data item granularity:
  1. A field of a database record (an attribute of a tuple)
  2. A database record (a tuple or a relation)
  3. A disk block
  4. An entire file
  5. The entire database

Granularity of data items and Multiple Granularity Locking

- The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



Granularity of data items and Multiple Granularity Locking

To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:

- **Intention-shared (IS)**: indicates that a shared lock(s) will be requested on some descendent nodes(s).
- **Intention-exclusive (IX)**: indicates that an exclusive lock(s) will be requested on some descendent node(s).
- **Shared-intention-exclusive (SIX)**: indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

Granularity of data items and Multiple Granularity Locking

Granularity of data items and Multiple Granularity Locking

- These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Intention-shared (IS
Intention-exclusive (IX)
Shared-intention-exclusive (SIX)

Granularity of data items and Multiple Granularity Locking

- The set of rules which must be followed for producing serializable schedule are
  1. The lock compatibility must adhered to.
  2. The root of the tree must be locked first, in any mode.
  3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
  4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
  5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
  6. T can unlock a node, N, only if none of the children of N are currently locked by T.

Granularity of data items and Multiple Granularity Locking

- T1 wants to update record r_{111} and record r_{211}.
- T2 wants to update all records on page p_{12}.
- T3 wants to read record r_{11j} and the entire f_2 file.

Granularity of data items and Multiple Granularity Locking

T1	T2	T3
IX(db)		
IX(f1)		
	IX(db)	
		IS(db)
		IS(f1)
		IS(p11)
IX(p11)		
X(r111)		
	IX(f1)	
	X(p12)	
		S(r11j)
IX(f2)		
IX(p21)		
IX(r211)		
Unlock (r211)		
Unlock (p21)		
Unlock (f2)		
		S(f2)

Granularity of data items and Multiple Granularity Locking

T1	T2	T3
	unlock(p12)	
	unlock(f1)	
	unlock(db)	
unlock(r111)		
unlock(p11)		
unlock(f1)		
unlock(db)		
		unlock (r111j)
		unlock (p11)
		unlock (f1)
		unlock(f2)
		unlock(db)