CSIE30600/CSIEB0290 Database Systems

Lecture 13: Concurrency Control

Outline

- Databases Concurrency Control
  1. Purpose of Concurrency Control
  2. Two-Phase locking
  3. Limitations of CCMs
  4. Index Locking
  5. Lock Compatibility Matrix
  6. Lock Granularity

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.

Lock and Unlock

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock(X). Data item X is locked on 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.

Two-Phase Locking

- Two locks modes:
  - (a) shared (read)
  - (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.
- Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Two-Phase Locking (2)

- 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

Well-formed Transaction

- 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.

Lock Operation

- 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;
```

Unlock Operation

- 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;
```


read_Lock(X) 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;
```

write_Lock(X) 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;
```


unLock(X) 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 ≠ 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.
- **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: Example

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);	

Two-Phase Locking: Example

T1	T2	Result
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it.
unlock (Y);		violated two-phase policy.
	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);		

Two-Phase Locking: Example

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

Deadlock

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

Deadlock (T'1 and T'2)

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 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 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.

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

Timestamp Based Protocols

- **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.

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).

Multiversion Protocols

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effect:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Multiversion with Timestamp

- Assume X1, X2, …, Xn are the version of a data item X created by a write operation of transactions. With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.
- **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.
- **write_TS(Xi)**: The write timestamp of Xi that wrote the value of version Xi.
- A new version of Xi is created only by a write operation.

Multiversion with Timestamp

- To ensure serializability, the following two rules are used.

1. If transaction T issues write_item(X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).
2. If transaction T issues read_item(X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

- Rule 2 guarantees that a read will never be rejected.

Multiversion 2PL Using Certify Locks

- Concept
  - Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.
  - This is accomplished by maintaining **two** versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X.

Multiversion 2PL Using Certify Locks (2)

- **Steps**
  1. X is the committed version of a data item.
  2. T creates a second version X' after obtaining a write lock on X.
  3. Other transactions continue to read X.
  4. T is ready to commit so it obtains a certify lock on X'.
  5. The committed version X becomes X'.
  6. T releases its certify lock on X', which is X now.

	Read	Write
Read	yes	no
Write	no	no

Compatibility tables for read/write locking scheme

	Read	Write	Certify
Read	yes	no	no
Write	no	no	no
Certify	no	no	no

read/write/certify locking scheme

Multiversion 2PL Using Certify Locks (3)

- Note:
  - In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently.
  - This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes. It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

Validation(Optimistic) Protocols

- 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 Protocols (1)

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:
     1. Tj completes its write phase before Ti starts its read phase.
     2. 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 Protocols (2)

3. 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.

Validation Protocols (3)

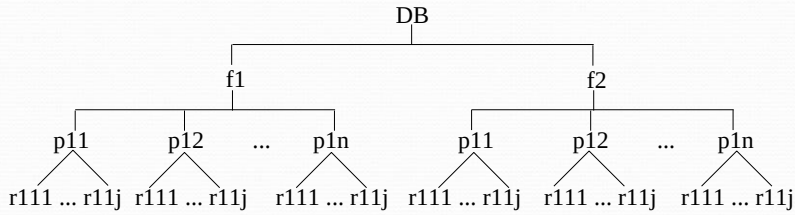
3. **Write phase**: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

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

Multiple Granularity Locking

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



Multiple Granularity Locking (2)

- 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).

Multiple Granularity Locking (3)

- 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)

Multiple Granularity Locking (4)

- 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.

Multiple Granularity Locking: Example

Granularity of data items and Multiple Granularity Locking: An example of a serializable execution:

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)

Multiple Granularity Locking: Example

- Granularity of data items and Multiple Granularity Locking: An example of a serializable execution (continued):

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)

Summary

- Databases Concurrency Control
  1. Purpose of Concurrency Control
  2. Two-Phase locking
  3. Limitations of CCMs
  4. Index Locking
  5. Lock Compatibility Matrix
  6. Lock Granularity