CSC 742
Database Management Systems

Topic #14: Concurrency Control
– 2 Phase Locking

Locks

Locks are objects that describe the usage status of a data item

- A data item may be
  - a record
  - a field
  - a page
  - an index
  - a table
  - the whole DB
- Granularity determines concurrency and overhead (hence a trade-off).

Kinds of Locks

- Binary locks:
  - Conceptually, each data item x needs a lock
  - Two operations:
    - lock(x)
    - unlock(x)
    - Must be atomic
  - Gives mutex but restrictive
  - Implementation
    - Lock table: Stores the active locks
    - Lock manager: maintain lock table

Use Binary Lock for Transactions

- A transaction T
  - Lock(x) before read(x) or write(x)
  - Unlock(x) after all read(x) and write(x) are completed
  - Will not issue lock(x) if it already has the lock on x
  - Will not unlock(x) unless it already has the lock on x.
- Question:
  - What if no transaction write(x)?

Kinds of Locks (Cont'd)

- Multimode
  - Intuition: distinguish locks for read(x) and write(x)
  - shared-lock(x) read(x): multiple transactions can read x concurrently.
  - exclusive-lock(x) write(x): only one transaction can write x at each time.

Use Multimode Locks

- A transaction T
  - Read_lock(x) or write_lock(x) before read(x)
  - Write_lock(x) before write(x)
  - Unlock(x) after all read(x) and write(x) are completed
  - Will not issue read_lock(x) if it already has a read lock on x
  - Will not issue write_lock(x) if it already has a write lock on x
  - Will not unlock(x) unless it already has a read or write lock on x.

Lock Conversion

- Lock conversion:
  - can be upgraded (read to write)
  - or downgraded (write to read)

Does locking guarantee serializability?

Time ↓

T1	T2
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);**	

Serialization Graph?

Two-Phase Locking

Moral: can't release locks too soon

- 2PL: All locking operations precede the first unlock operation.
  - growing phase
  - shrinking phase
- Guarantees serializability, but can lead to deadlock

Are these transactions using 2PL?

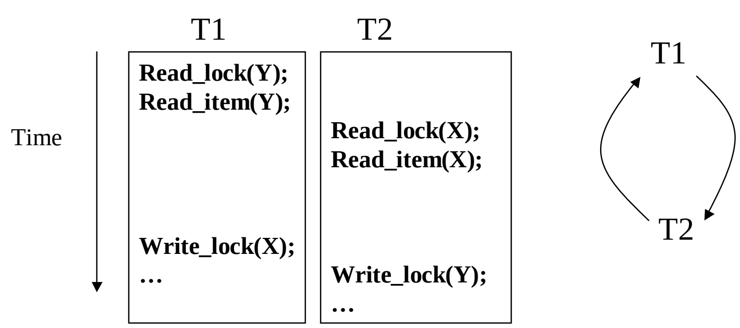
T1	T2	T3	T4
Read_lock(Y);	**Read_lock(X);**	**Read_lock(Y);**	**Read_lock(X);**
Read_item(Y);	**Read_item(X);**	**Read_item(Y);**	**Write_lock(Y);**
Unlock(Y);	**Unlock(X);**	**Write_lock(X);**	**Read_item(X);**
Write_lock(X);	**Write_lock(Y);**	**Unlock(Y);**	**Read_item(Y);**
Read_item(X);	**Read_item(Y);**	**Read_item(X);**	**Y:=X + Y;**
X:=X+Y:	**Y:=X + Y;**	**X:=X+Y:**	**Write_item(Y);**
Write_item(X);	**Write_item(Y);**	**Write_item(X);**	**Unlock(X);**
Unlock(X);	**Unlock(Y);**	**Unlock(X);**	**Unlock(Y);**

Basic 2PL

- Rules for basic 2PL scheduler
  - For any operation $p_i(x)$ (p is read or write), test if $p\_lock_i(x)$ conflicts with some $q\_lock_j(x)$ that is already set. If so, it delays $p_i(x)$ until it can set $p\_lock_i(x)$. If not, set $p\_lock_i(x)$.
    - No concurrent access to the same item.
  - Once the scheduler has set $p\_lock_i(x)$, it may not release it at least until $p_i(x)$ has been performed.
    - Further guarantee no concurrent access.
  - Once the scheduler has released a lock for Ti, it may not obtain any more locks for Ti.
    - Two phase rule

2PL

- 2PL guarantees serializability.
- Deadlock



Conservative 2PL

- Conservative or static 2PL
  - Obtain all locks before any operation
  - Make transaction wait (without any lock) if not all the locks can be obtained.
  - No deadlock: If T is waiting for a lock held by T', then T has no lock.
  - Disadvantage: you have to know what locks a transaction needs
    - How to get Read set and write set?

Strict 2PL

- Strict 2PL
  - Release all locks at once when the transaction commits or aborts
  - ensures strict schedules
  - but can deadlock

Deadlock Prevention

- Pessimistic: prevent deadlock from even becoming possible by restricting access when Ti tries to get an element locked by Tj
- Deadlock prevention using timestamps (TS)
  - An older transaction has smaller TS.
  - Two variations:
    - Wait-die
    - Wound-wait

Deadlock Prevention (Cont'd)

- Suppose Ti tries to lock x but is not able to because x is locked by Tj with a conflicting lock.
  - *wait-die*:
    - If TS(Ti) < TS(Tj) then wait Ti
    - else abort Ti and restart with same time
  - Old transactions are allowed to wait.
  - How can wait-die prevent deadlock?

Deadlock Prevention (Cont'd)

- Suppose Ti tries to lock x but is not able to because x is locked by Tj with a conflicting lock.
  - *wound-wait*:
    - If TS(Ti) < TS(Tj) abort Tj and restart with some timestamp,
    - else Ti wait
  - Young transactions are allowed to wait.
  - How can wound-wait prevent deadlock?

Deadlock Prevention (Cont'd)

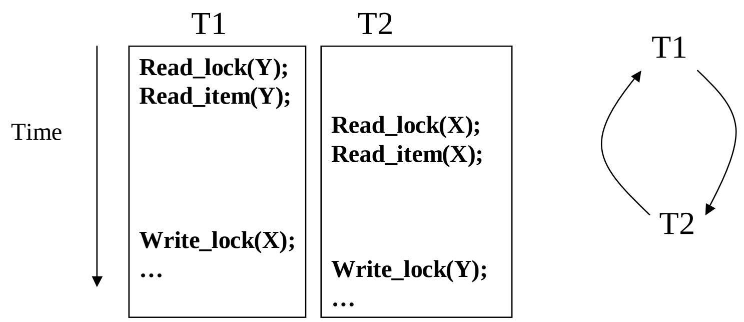
- Prevent deadlock by Limiting Waiting
  - *No waiting*: abort transaction immediately if lock not obtained
  - *Cautious waiting*: abort transaction only if current lock holder is itself blocked

Deadlock Detection

- Optimistic strategy
- Detect a cycle in *waits-for graph*
- Choose a *victim* transaction
- Abort it thereby removing the deadlock
- Potentially unfair: the same victim is repeatedly chosen

Deadlock Detection

- Wait-for Graph
  - One node for each transaction
  - An edge from Ti to Tj if Ti is waiting to lock x that is currently locked by Tj.
  - Cycle means deadlock.



Multiversion 2PL

- Basic idea:
  - Maintain up to two versions of each data item x.
  - Each x must have one committed version, supplied to transactions that read x.
  - Create a new version when T needs to write x
  - Once T that writes x is ready to commit, it must obtain a *certify lock* on all items that it currently holds write locks on before it can commit.
    - To install new versions.

Multi-version 2PL (Cont'd)

- Lock compatibility tables
  - 2PL

	Read	Write
Read	Yes	No
Write	No	No

What do we gain via multi-version 2PL?

  - Multi-version 2PL

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Is deadlock possible in multi-version 2PL?

T1	T2
Read_lock(Y); **Read_item(Y);**	
	Read_lock(X); **Read_item(X);**
Write_lock(X);	
	Write_lock(Y);
Certify_lock(X); **…**	
	Certify_lock(Y); **…**

Multi-granularity locking

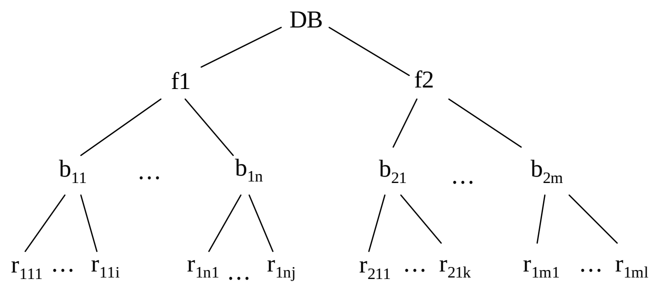
- Granularity: the size of a data item
  - Database
  - Database file
  - Disk block
  - Relation
  - Tuple

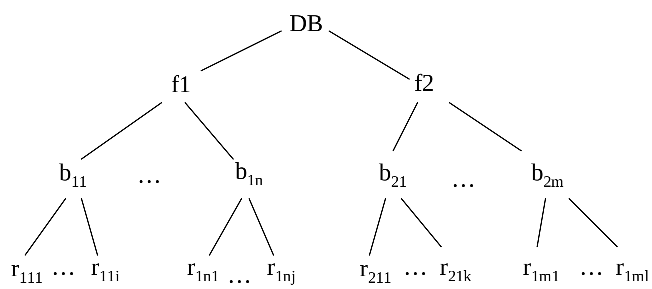
Multi-granularity locking (Cont'd)

- Transaction 1: update 75% of the tuples in relation Employee.
- Transaction 2: update 1 tuple in relation Employee.
- How should we set the granularity of data items?
  - Coarse: less concurrency
  - Fine: more locks

Multi-granularity locking (Cont'd)

- Basic idea:
  - Support multiple granularities.





T1: I want to read_lock(r111). Is there any conflicting lock?

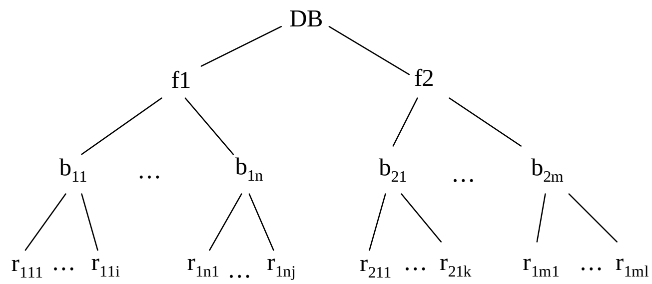
T2: I want to read_lock(f1). Is there any conflicting lock?

Multi-granularity locking (Cont'd)

- Solution to reducing search for conflicting locks
  - Intention lock:
    - For the nodes along the path from the root to the item of choice (excluding the final node)
    - Indicate what types of lock T wants to obtain for the current node's descendants

Multi-granularity locking (Cont'd)

- Intention locks:
  - Intention-shared (IS): a shared lock will be requested on some descendants
  - Intention-exclusive (IX): an exclusive lock will be requested on some descendants
  - Shared-intension-exclusive (SIX): the current node is locked in shared mode, but an exclusive lock will be requested on some descendants.



T1: I want to read (r111).

T2: I want to write (r111).

T3: I want to go through the Employee relation stored in f1 and update the tuples with Salary > 30000.

What locks will be requested?

Compatibility matrix for multi-granularity locking

	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multi-granularity locking protocol

1. The lock compatibility matrix must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node N can be locked by T in S or IS only if the parent node is already locked by T in IS or IX.
4. A node N can be locked by T in X, IX, or SIX mode only if the parent is already locked by T in IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (2-phase rule).
6. T can unlock a node N only if none of the children of N are locked by T (2-phase rule).

Phantom problem

- Phantom problem occurs when there are insertions.
  - When a new record being inserted by T satisfies a condition that a set of records accessed by T' must satisfy.

Accounts

Account#	Location	Balance
111	Raleigh	100
222	Apex	200
333	Apex	300

Assets

Location	Balance
Raleigh	100
Apex	500

What's the Result?

What's wrong?

T1	T2
Read(Accounts[111]); **Read(Accounts[222]);** **Read(Accounts[333]);**	
	Insert(Accounts[444, Raleigh, 100])
Compute assets[Raleigh]; **Compute assets[Apex];** **Write(Assets[Raleigh]);** **write(Assets[Apex]);**	

Phantom problem (Cont'd)

- Solutions
  - Index locking
  - Predicate locking

Optimistic Concurrency Control

- Three phases of a transaction T
  - *Read phase*: T reads data, updates local copies
  - *Validation phase*: check to ensure that serializability will not be violated if the updates are applied to the DB
  - *Write phase*: if valid, write to DB
- Basic idea: do all checks at once.
- write-set(T): items written by T
- read-set(T): items read by T

Optimistic Protocol

- Validate Ti w.r.t. any Tj that committed or is being validated
  - Tj completed its write phase before Ti began its read phase
    - Serial transactions
  - Ti starts its write phase after Tj completes its write phase, and read_set(Ti)$\cap$write_set(Tj) = $\emptyset$.
    - All possible conflicting pairs of operations are from Tj to Ti.
  - Tj completed its read phase before Ti completes its read phase, read_set(Ti)$\cap$write_set(Tj) = $\emptyset$, and write_set(Ti)$\cap$write_set(Tj) = $\emptyset$.
    - All possible conflicting pairs of operations are from Tj to Ti.