DEADLOCK AND ISOLATION LEVELS

Kathleen Durant PhD
Lesson 10
CS3200

Outline for the day

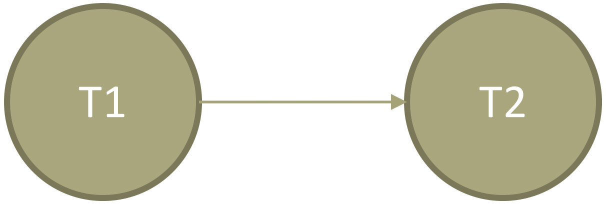
- Precedence graph
- Deadlock prevention and detection
  - Waits-for graph
- My SQL Granular Locking
- Concurrency without locking
  - Optimistic Concurrency Control
  - Timestamp based concurrency control

Precedence Graph

- To determine if a schedule is conflict serializable we use a precedence graph
- Transactions are vertices of the graph
- There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serial schedule
- Edge T1 -> T2 if in the schedule we have:
  - T1 Read(R) followed by T2 Write(R) for the same resource R
  - T1 Write(R) followed by T2 Read(R)
  - T1 Write(R) followed by T2 Write(R)
- The schedule is serializable if there are no cycles

Example 1: Precedence Graph

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	X(A)
R(B)	
W(B)	
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)



Fill in the edges

Example 2: Precedence graph

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	R(B)
	W(B)
X(C)	
R(C)	
W(C)	

Fill in the edges

Example 3: Precedence Graph

T1	T2
Read(A)	
Write(A)	
	Read(B)
	Write(B)
	Read(A)
	Write(A)
Read(B)	
Write(B)	

Fill in the edges

Concurrency Control Techniques

- **Two basic concurrency control techniques:**
  - **Locking**
  - **Timestamping**
- **Both are conservative approaches: delay transactions in case they conflict with other transactions.**
- **Optimistic methods assume conflict is rare and only check for conflicts at commit.**

7

Locking

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

- **Most widely used approach to ensure serializability.**
- **Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.**
- **Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.**

Locking - Basic Rules

- **If transaction has shared lock on item, can read but not update item.**
- **If transaction has exclusive lock on item, can both read and update item.**
- **Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.**
- **Exclusive lock gives transaction exclusive access to that item.**

Locking - Basic Rules

- **Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.**

Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Handling Deadlocks

- **Three general techniques for handling deadlock:**
  - **Timeouts.**
  - **Deadlock prevention.**
  - **Deadlock detection and recovery.**

Timeouts

- **Transaction that requests lock will only wait for a system-defined period of time.**
- **If lock has not been granted within this period, lock request times out.**
- **In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.**

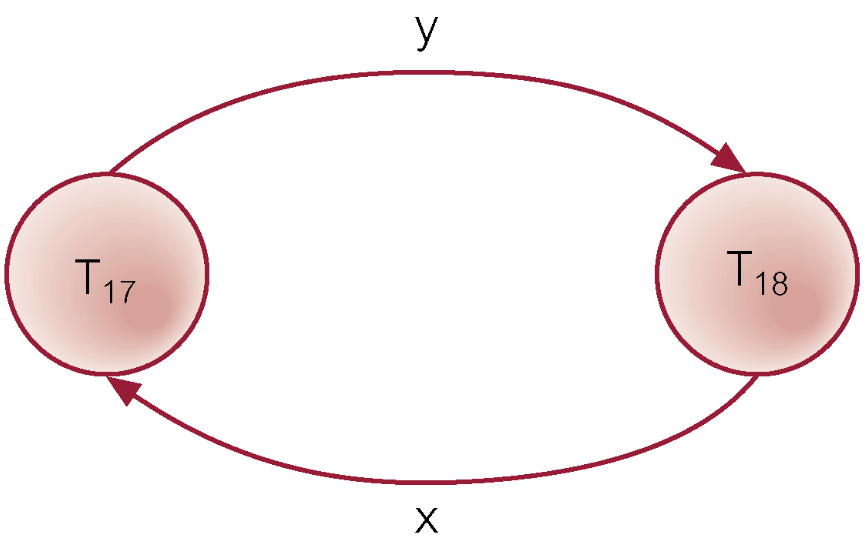
13

Deadlock Detection and Recovery

- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge T_i -> T_j, if T_i waiting to lock item locked by T_j.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.

14

Example - Wait-For-Graph (WFG)



15

Timestamping – preventing deadlocks

- **Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.**
- **Conflict is resolved by rolling back and restarting transaction.**
- **No locks so no deadlock.**

Deadlock Prevention

- **DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.**
- **Could order transactions using transaction timestamps:**
  - **<u>Wait-Die</u> - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp.**
  - **<u>Wound-Wait</u> - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).**

Timestamping

Timestamp

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- **Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.**

Timestamping (No locks)

- **Read/write proceeds only if *last update on that data item* was carried out by an older transaction.**
- **Otherwise, transaction requesting read/write is restarted and given a new timestamp.**
- **Also timestamps for data items (stored in the DB):**
  - **read-timestamp - timestamp of last transaction to read item;**
  - **write-timestamp - timestamp of last transaction to write item.**

Timestamping - Read(x)

- **Consider a transaction T with timestamp ts(T):**

ts(T) < write_timestamp(x)

- **x already updated by younger (later) transaction.**
- **Transaction must be aborted and restarted with a new timestamp.**

ts(T) < read_timestamp(x)

- **x already read by younger transaction.**
- **Roll back transaction and restart it using a later timestamp.**

Timestamping - Write(x)

ts(T) < write_timestamp(x)

- **x already written by younger transaction.**
- **Write can safely be ignored - *ignore obsolete write* rule.**

- **Otherwise, operation is accepted and executed.**

Example–Basic Timestamp Ordering

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)$^+$	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

$^+$ At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.

‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Optimistic Techniques

- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

26

Performance of Locking

- Locks force transactions to wait
  - Abort and restart due to deadlock wastes the work done by the aborted transaction
  - In practice, deadlocks are rare, e.g., due to lock downgrades approach
- Waiting for locks becomes bigger problem as more transactions execute concurrently
  - Allowing more concurrent transactions initially increases throughput, but at some point leads to thrashing
  - Need to limit maximum number of concurrent transactions to prevent thrashing
  - Minimize lock contention by reducing the time a transaction holds locks and by avoiding hotspots (objects frequently accessed)

Controlling Locking Overhead

- Declaring transaction as "READ ONLY" increases concurrency
- Isolation level: trade off concurrency against exposure of transaction to other transaction's uncommitted changes
  - Degrees of serializability

Isolation level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Isolation levels

- SERIALIZABLE: obtains locks on (sets of) accessed objects and holds them until the end
- REPEATABLE READ: same locks as for serializable transaction, but does not lock sets of objects at higher level
- READ COMMITTED: obtains X-locks before writing and holds them until the end; obtains S-locks before reading, but releases them immediately after reading
- READ UNCOMMITTED: does not obtain S-locks for reading; not allowed to perform any writes
  - Does not request any locks ever

Hierarchy of Granularity

- **Could represent granularity of locks in a hierarchical structure.**
- **Root node represents entire database, level 1s represent files, etc.**
- **When node is locked, all its descendants are also locked.**
- **DBMS should check hierarchical path before granting lock.**

Lock Modes: State Intent

	IS	IX	S	X
IS	✓	✓	✓	
IX	✓	✓		
S	✓		✓	
X				

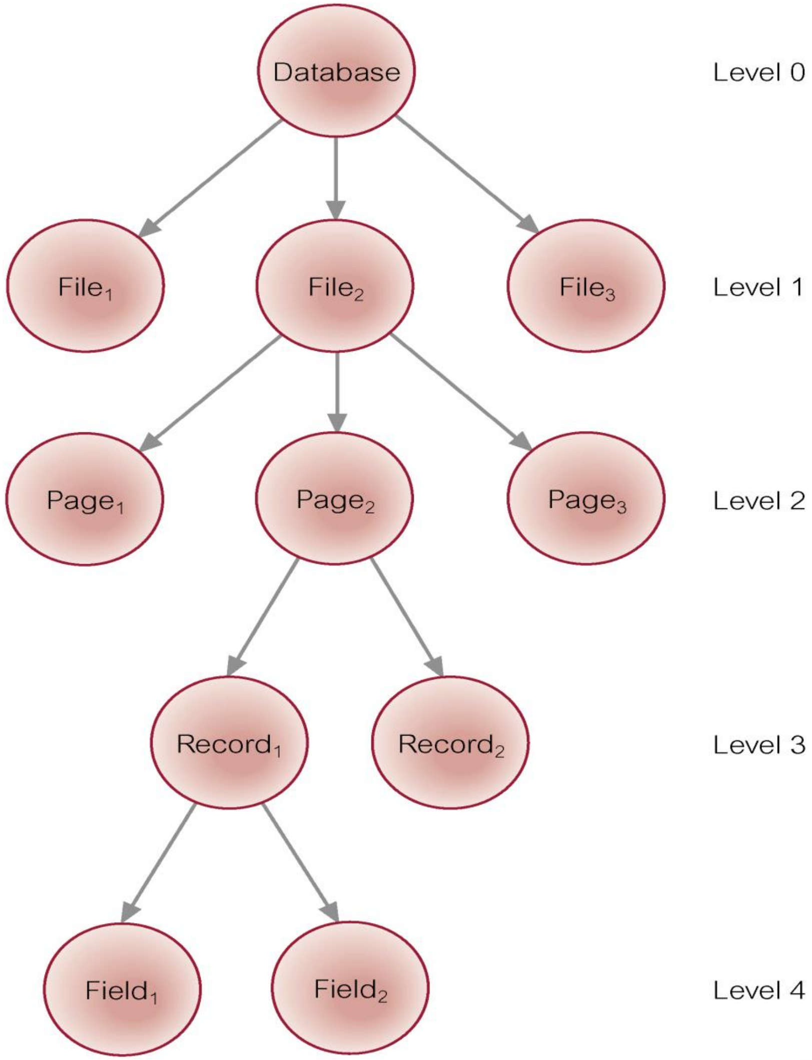
- Allows transactions to lock at each level but with a special protocol using new ‘intentions’ locks.
  - Can be read intent (intent share) or write intent (intent exclusive)
- Before viewing an item, transaction must set intention locks on all its ancestors (higher level containers)
- Locks are applied top-down, released bottom-up

Granularity of Data Items

- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A table.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.

Levels of locking

- Each transaction starts from the root of the hierarchy
- To get S or IS lock on a node, must hold IS or IX on parent node
- To get X or IX on a node, must hold IX on parent node
- Must release locks in bottom-up order
- Equivalent to directly setting locks at the leaf levels



Granularity of Data Items

- **Tradeoff:**
  - **coarser, the lower the degree of concurrency;**
  - **finer, more locking information that is needed to be stored.**
- **Best item size depends on the types of transactions.**

ISOLATION LEVEL: MYSQL

- **SET TRANSACTION** ISOLATION LEVEL l*evels*;
  - SERIALIZABLE
  - REPEATABLE READ
  - READ COMMITTED
  - READ UNCOMMITTED
- Default is that the command affects the next transaction
- Can also set the ISOLATION LEVEL for the current session and globally
  - SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL l*evels*;
  - **GLOBAL** applies globally for all subsequent sessions. Existing sessions are unaffected.
  - **SESSION** applies to all subsequent transactions performed within the current session
- Can also define the access method for the query
  - **SET TRANSACTION READ ONLY**
  - **SET TRANSACTION READ WRITE**

INNODB and Transactions

- All user activity occurs inside a transaction
- If autocommit mode is enabled, each SQL statement forms a single transaction on its own.
- Perform a multiple-statement transaction by starting it with an explicit START TRANSACTION
- autocommit mode is disabled within a session with SET autocommit = 0,
  - The session will have a transaction open until it is explicitly closed
  - Issue commit or rollback to close the transaction
- Default InnoDB Isolation level is REPEATABLE READ
- InnoDB performs row level locking
  - Only if two transactions try to modify the same row does one of the transactions wait for the other to complete

InnoDB and locks

- InnoDB implements standard row-level locking where there are two types of locks
  - (S) shared locks
    - permits the transaction that holds the lock to read a row.
  - (X) exclusive locks
    - permits the transaction that holds the lock to update or delete a row.
- InnoDB supports *multiple granularity locking* which permits coexistence of record locks and locks on entire tables.
  - Intention locks are table locks in InnoDB that indicate which type of lock a transaction will require later for a row in that table.
  - Intention shared (IS) Transaction T intends to set *S* locks on individual rows in table t. (SELECT … LOCK IN SHARE MODE)
  - Intention exclusive(IX) Transaction T intends to set *X* locks on individual rows in table t (SELECT … LOCK FOR UPDATE)

Granting locks

- A lock is granted to a requesting transaction if it is compatible with existing locks
- A transaction waits until the conflicting existing lock is released
- If a lock request conflicts with an existing lock and cannot be granted because it would cause deadlock, an error occurs
- Main purpose of *IX* and *IS* locks is to show that someone is locking a row, or going to lock a row in the table.
- SHOW ENGINE INNODB STATUS;
  - To report on any transactions and deadlock conditions.

Summary

- Precedence graph allow us to represent transactions whose actions involve reading and writing the same data object
- Deadlocks can be assumed, prevented or detected.
  - Assumed if a transaction is waiting longer than the system time limit n – the system aborts and restarts the transaction
  - Detected via waits-for graph
  - Prevented via timestamps
- Optimistic concurrency control aims to minimize the cost of Concurrency Control
  - Best when reads are common and writes are rare