

Transaction Management

Chapter 16,17

in the Course Textbook

Transaction Processing, Concurrency Control

ramesh@cs.ubc.ca

<http://www.cs.ubc.ca/~ramesh/cpsc304>

TRANSACTIONS

Transaction

- Foundation for concurrent execution and recovery from system failure in a DBMS
- Defined as *any one* execution of a user program in a DBMS.
- Not all user programs are transactions.

Requirements are placed on them on the way they are executed that are not present for nontransactional programs.

For performance reasons, actions of several transactions are **interleaved** and executed simultaneously.

Interleaving - cause of some problems.

DB has to guarantee that effect of executing an interleaved set of transactions is *equivalent in terms of effect on DB* to some *serial* (one transaction at a time) execution of these transactions.

Concurrency Control - Deals with how the DB handles concurrent execution of transactions.

ACID Properties

ACID Properties - Distinguish transactions from ordinary programs

- **A**tomic: Each transaction is executed completely or not at all.
- **C**onsistency: The execution of each transaction in isolation maintains database consistency and results in a database whose new state correctly models the new state of the enterprise.
- **I**solation: The concurrent execution of a set of transactions in the database has the same effect as that of *some* serial execution of the set of transactions. i.e., transactions are isolated or protected from the effects of concurrently scheduling other transactions,
- **D**urability: The results of committed transactions are permanent - the committed transactions should be reflected in the database even if the system crashes before all its changes are reflected on disk.

Consistency - Transaction consistency Vs. Database Consistency

- Users are responsible for ensuring transaction consistency
- Every transaction sees a consistent database instance - Database Consistency - Follows from Transaction Atomicity, Isolation and Transaction Consistency

ACID Properties

Isolation Property:

- Actions of several transactions might be interleaved
- However, the net effect is the same as *some* serial order of execution of these transactions

Atomicity: Transactions that terminate halfway may leave the DB in an inconsistent state

All or Nothing property - Either the entire transaction is executed or none at all

- DB must remove effect of *partial* transactions from the database
- UNDO actions of partial transactions
- **LOG** - A record of all writes to the database

Durability: The LOG is also used to ensure durability

- If the system crashes before the changes made by a completed transaction are written to disk, the LOG is used to remember and restore the changes.

RECOVERY MANAGER: Component ensuring atomicity and durability.

Transactions

Transaction - A series or list of actions

- Actions - Reads and Writes of Database Objects
- O - Variable that refers to Object O
- $R_T(O)$, $W_T(O)$ - Read or write object O by transaction T ($R(O)$, $W(O)$ When T is clear from the context)
- Each transaction MUST specify as its final action commit or abort

Assumptions:

- Transactions interact only via read or writes - NO Message Exchanges
- Database - a fixed collection of objects

Schedules

Schedule - List of actions (reading, writing, aborting or committing) from a set of transactions.

- Order of actions in a transaction that appear in the schedule - Same as the order in which they appear in a transaction.
- Represents an actual or potential sequence of execution
- Move forward in time as we proceed from up to down
- Schedule - represents the actions of transactions *as seen by the DBMS*
- Effect of one transaction on another can be solely understood in terms of the common DB objects they read and write.

Complete Schedule: A schedule contains either an abort or commit for each transaction whose actions are listed.

Serial Schedule: If actions of different transactions are not interleaved - transactions are executed one by one from start to end

Example

- A schedule involving two transactions

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- Not a complete schedule
- Not a serial schedule - T_1 and T_2 are *interleaved*

Concurrent Execution - Motivation

- Transaction may wait for I/O (Page reads from disk)
- CPU is idle - Disk I/O and CPU can function in parallel
- CPU can be used to process another transaction
- Concurrent execution - Reduces IDLE time and hence increases *system throughput*
- Long transaction and Short transaction - A short transaction may end up scheduled behind a long transaction.
 - ▷ Without concurrent execution, the short transaction takes a long time to complete
 - ▷ Unpredictable *response time*
- **Throughput:** Average number of transactions completed in a given time
- **Response Time:** Average time taken to complete a transaction

Serializability

- **Serializable Schedule:** A serializable schedule S over a set of committed transactions is a schedule whose effect on any consistent database is guaranteed to be identical to that of some complete serial schedule over S .
- Database instance after executing a serializable schedule S - equivalent to executing the transactions of S in *some* serial order. (Does not consider aborted transactions)
- Example: A serializable schedule and its equivalent serial schedule

T ₁		T ₂		T ₁		T ₂
R(A)				R(A)		
W(A)				W(A)		
		R(A)		R(B)		
		W(A)		W(B)		
R(B)				commit		R(A)
W(B)						W(A)
		R(B)				R(B)
		W(B)				W(B)
		commit				commit
commit						

Remarks

- DBMS - sometimes executes transactions in a way that is not equivalent to any serial execution
 - ▷ Use of a concurrency control method that ensures the executed schedule, though not serializable, is equivalent to some serializable schedule
 - ▷ SQL gives application programmers the ability to choose non-serializable schedules

Interleaved Execution - Anomalies

- Two actions on the same data object **conflict** if at least one of them is a write.
- Three main ways in which schedule with two consistency preserving, committed transactions could execute on a consistent database and leave it inconsistent.
- **Reading Uncommitted Data - WR Conflicts**
- **Unrepeatable Reads - RW Conflicts**
- **Overwriting Uncommitted Data - WW Conflicts**

Reading Uncommitted Data - WR Conflicts

- Transaction T_2 could read a database object A that has been modified by T_1 which has not committed.
- These reads are called **dirty reads**.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit
R(B)	
W(B)	
commit	

T_1 writes some value into A
 T_2 reads and modifies A before T_1 commits

Unrepeatable Reads - RW Conflicts

- T_2 changes the value of an object A that has been read by T_1 and T_1 is still in progress
- Result of such a case is T_1 gets a different value of A when it reads it again.
- In a serial execution, this case does not happen - Called **unrepeatable read**

T_1	T_2	
$R(A)$		Suppose $A = 1$
	$R(A)$	T_1 reads A and finds it is > 0
	$W(A)$	T_2 reads A and finds it is > 0
	commit	T_2 decrements A and commits
$W(A)$		T_2 tries to decrement A - but cannot
commit		

- In serial execution - the second transaction will see A to be 0 and hence will not decrement it.

Overwriting Uncommitted Data - WW Conflicts

- T_2 overwrites the value of A modified by T_1 while T_1 is still in progress.
- **Blind write:** A write where the value of the object is NOT READ before being written.
- Suppose A and B MUST have the same value. T_1 sets both their values to 1 while T_2 sets both values to 2

T_1	T_2	
	$W(A)$	T_2 sets $A = 2$
$W(B)$		T_1 sets $B = 1$
	$W(B)$	T_2 sets $B = 2$
	commit	T_1 sets $A = 1$
$W(A)$		Final values are $A = 1, B = 2$
commit		

- **LOST UPDATE**

Schedules with Aborted Transactions

- A **serializable schedule** over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to some complete serial schedule over the set of committed transactions in S .
- Actions of ABORTED TRANSACTIONS - Undone completely - May be impossible in certain situations
- An unrecoverable schedule

T_1	T_2
R(A)	T ₁ reads and sets A = 1
W(A)	T ₂ reads and sets A = 5
	T ₂ reads and sets B = 2
	T ₂ commits
	T ₁ aborts
	T ₂ has committed
	A's value is not recoverable to undo
abort	

Schedules with Aborted Transactions

- **Recoverable Schedule:** Transactions commit only after all transactions whose changes they read commit
- Transactions read only the changes of committed transactions
- Aborting a transaction can be accomplished without cascading the abort to other transactions
- Potential problem in undoing transactions:

T_1	T_2	A has value 5 initially
$W(A)$		T_1 changes A to value 6
	$W(A)$	T_2 changes A to 7
abort		A becomes 5 again
	commit	T_2 commits and its change to A is lost

- Strict 2PL can prevent such problems

Lock Based Concurrency Control

- DBMS has to ensure that
 - ▷ Only serializable, recoverable schedules are allowed
 - ▷ No actions of committed transactions are lost when undoing aborted transactions
- **Locking Protocol** is used to achieve these properties
- **Lock** - A small bookkeeping object associated with a DB Object
- **Locking Protocol** - A set of rules to be followed by each transaction to ensure that
Even though actions of several transactions may be interleaved, the net effect is identical to some serial execution of these transactions

Strict Two-Phase Locking (Strict 2PL)

- Most widely used locking protocol: Has two rules.
 1. If a transaction T wants to *read* (respectively *modify*) an object, it first requests a **shared** (respectively **exclusive**) lock on the object
 2. All locks held by a transaction are released when the transaction is completed
- An exclusive lock automatically allows reading objects too.
- A transaction that requests a lock is suspended until the DB is able to grant it.
- DBMS keeps track of granted locks and guarantees that **ONLY** one transaction holds an exclusive lock on any given object.
- Locking protocol - allows only "safe" interleavings of transactions
- $S_T(O), X_T(O)$ - Transaction T requesting a shared or exclusive lock on object O ($S(O), X(O)$ when the context of transaction T is clear)

Example

- Such interleaving is disallowed in Strict 2PL. A strict 2PL schedule is shown beside.
- A schedule with interleaved actions is also shown

T ₁	T ₂
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit
R(B)	
W(B)	
commit	

T ₁	T ₂
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	commit

T ₁	T ₂
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	R(B)
	W(B)
	commit
X(C)	
R(C)	
W(C)	
commit	

Strict 2PL - Some Remarks

- Actions of different transactions could be interleaved in Strict 2PL
- Strict 2PL algorithm allows ONLY serializable schedules
- None of the anomalies discussed would arise if DBMS implements strict 2PL

Deadlocks

- Consider the following schedule:

T_1	T_2
$X(A)$	T_1 sets exclusive lock on A
	T_2 sets exclusive lock on B
	T_1 requests exclusive lock on B - queued
	T_2 requests exclusive lock on A - queued

- Cycle of transactions waiting for locks to be released
- No further progress
- They may hold locks required by other transactions
- Identifying deadlocks: Timeout mechanism - If transaction has been waiting too long, then assume it might potentially be a deadlock

Performance of Locking

- Resolve conflicts between transactions : Blocking and Aborting
- Performance penalty:
 - ▷ Blocked transactions force other transactions to wait (may holds locks needed by them)
 - ▷ Aborting - wastes work done thus far by a transaaction
- Deadlocks - Extreme instance of blocking, account for fewer than 1% of transactions in practice
- Overhead of Locking - Delays due to blocking
- Throughput - Increases slowly by increasing number of active transactions
- Keeps increasing until adding more active transactions reduces the throughput.
- New transaction added blocks and competes with existing transactions - System **thrashes** at this point.

Increasing Throughput

- Throughput can be increased by
 - ▷ Locking the smallest sized objects possible - reducing the likelihood that two transactions need the same lock
 - ▷ Reducing the time that transactions hold locks - Other transactions are blocked for a shorter time
 - ▷ Reducing **hot spots** - Database objects that are frequently accessed, modified and hence cause a lot of blocking delays
- Granularity of locking - depends on system's implementation of locking

Concurrency Control

Chapter 17

in the Course Textbook

Concurrency Control

ramesh@cs.ubc.ca

<http://www.cs.ubc.ca/~ramesh/cpsc304>

2PL - Serializability - Recoverability

- **Conflict Equivalent:** Two schedules are *conflict equivalent* if they involve the actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the same way.
- Two actions conflict if they operate on the same data object and at least one of them is a write.
- Outcome of a schedule depends only on the order of conflicting operations
- Nonconflicting operations - order can be interchanged without altering the effect of the schedule on the database
- Conflict equivalent schedules - same effect on the database

Conflict Serializability

- **Conflict Serializable:** A schedule is *conflict serializable* if it is conflict equivalent to some serial schedule.
- Assumption: DB does not grow or shrink i.e., Objects can be modified but not added or removed from the DB
- Under this assumption, EVERY conflict serializable schedule is serializable
- Some serializable schedules are NOT conflict serializable.

T_1	T_2	T_3
R(A)		
	W(A) commit	
W(A) commit		
		W(A) commit

- This schedule is equivalent to T_1, T_2, T_3 but is not conflict equivalent to this serial schedule

Precedence Graph

- Capturing potential conflicts - **Precedence Graph** or **Serializability Graph**
- Precedence graph for a schedule S
 - ▷ A node for each committed transaction in S
 - ▷ An arc from T_i to T_j if an action of T_i precedes and conflicts with an action of T_j .
- Strict 2PL protocol - allows only conflict serializable schedules which follows from the following results.
 - ▷ A schedule S is conflict serializable if and only if its precedence graph is acyclic. (Topological sort of the precedence graph gives an equivalent serial schedule)
 - ▷ Strict 2PL ensures that the precedence graph for any schedule that it allows is acyclic.

2PL and Strict 2PL

- A *variant* of strict 2PL - **Two-Phase Locking (2PL)**
- In 2PL The second rule of *Strict 2PL* is replaced by
 - ▷ A transaction cannot request additional locks once it releases *any* lock
- Every transaction - growing phase where it accumulates locks and shrinking phase where it relinquishes them
- Every *nonstrict 2PL* ensures acyclicity of the precedence graph - hence allows only conflict serializable schedules
- Equivalent serial schedules - Order in which transactions enter their shrinking phase
- **Strict Schedule:** If a value written by a transaction T is not overwritten or read by other transactions until T either aborts or commits
- Strict schedules - Recoverable, no cascading aborts, actions undone by restoring original values of modified objects
- Strict 2PL - improves on 2PL by guaranteeing that every allowed schedule is strict in addition to conflict serializable

View Serializability

- S_1 and S_2 - two schedules over the same set of transactions
- They are **view equivalent** under the following conditions
 - ▷ If T_i reads the initial value of object A in S_1 , it also does so in S_2
 - ▷ If T_i reads the value of object A written by T_j in S_1 , it also does so in S_2
 - ▷ For each data object A , the transaction (if any) that performs the final write on A in S_1 must also perform the final write on A in S_2
- A schedule is **view serializable** if it is view equivalent to some serial schedule.
- Every conflict serializable schedule is view serializable though the reverse is not true
- Testing view serializability - expensive

Lock Management

- **Lock Manager:** Part of the DBMS that keeps track of locks issued to transactions
- **Lock Table:** Hash table with the data object identifier as the key
- **Transaction Table:** Maintains descriptive entry for each transaction, pointer to a list of locks held by the transaction
- **Lock Table Entry for an object:**
 - ▷ The number of transactions holding a lock on the object
 - ▷ Type of lock (Shared or Exclusive)
 - ▷ Pointer to the queue of lock requests

Lock and Unlock Requests

- Strict 2PL - transaction T must obtain a shared or exclusive lock before it reads or writes a database object O
- To get a lock on the object - issues a lock request to the lock manager
 - ▷ Requests a shared lock, queue of requests is empty, The object is currently not locked in exclusive mode, lock manager grants the lock and updates lock table entry for the object
 - ▷ Requests an Exclusive lock and no transaction currently holds any lock on the object (queue of requests is empty), lock manager grants lock and updates lock table entry
 - ▷ Otherwise, lock cannot be granted and lock request is queued. Transaction is suspended
- Transaction aborts or commits - releases all locks - Lock table entry is updated
- Examines the lock request the the head of the queue - if it can be granted, it is.
- If T_1 has a shared lock on O and T_2 requests an exclusive lock on O , then request is queued. If T_3 requests shared lock on O - it is queued behind T_2 even though it can be granted (to prevent starvation)

Atomicity of Locking and Unlocking

- Lock and Unlock commands - Atomic operations. Guarded by Operating Systems synchronization mechanism like semaphores
- Entire sequence of actions in a lock or unlock request must be implemented as an atomic operation
- Locks are held for long duration.
- **Latches** - supported for short duration
- Latches ensure that the physical read/write operation is atomic
- Latches are unset as soon as the physical operations are completed
- **Convoys:** When most of the CPU time is spent on process switching (due to concurrent transactions)
- Transaction T holding a heavily used lock may get suspended by the OS
- All other transactions needing this lock get queued and these may get long - called **convoys**
- Stabilizes after a while

Lock Conversions

- Transaction requests exclusive lock on object on which it holds a shared lock - **Lock Upgrade**
- Must be granted immediately if no other transaction holds a shared lock on the object
- Queueing such requests may cause deadlocks
- However, granting them immediately does not prevent deadlocks either - between two lock upgrades
- Avoid lock upgrades - Grant exclusive lock first and then **downgrade** them to shared, once it is clear that shared locks are sufficient
- Downgrading - reduces concurrency by requesting write locks when not needed but improves throughput by reducing deadlocks
- New kind of lock - **Update** lock - Does not conflict with other shared locks and conflicts only with other update or exclusive locks
- Can be downgraded to a shared lock or upgraded to an exclusive lock

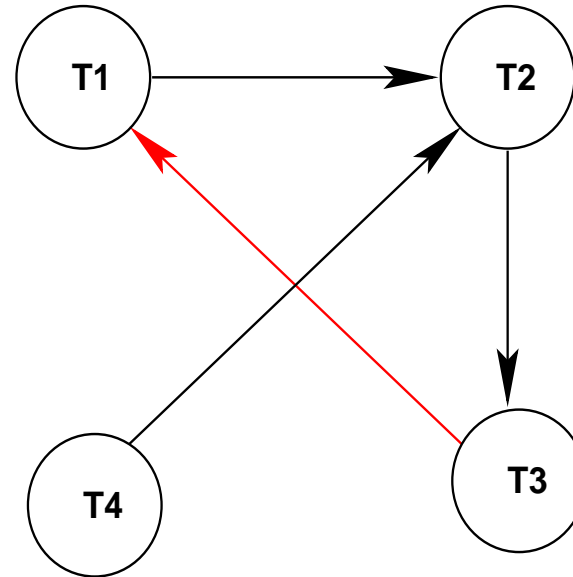
Deadlocks

- Deadlocks - tend to be rare and involve only very few transactions
- DBMS - periodically check for deadlocks
- T_i - suspended because a lock it needs cannot be granted
- Waits until all transactions holding conflicting locks release them
- **Waits-for Graph** - structure maintained by the lock manager
- Nodes - Active transactions, Edge from T_i to T_j iff T_i is waiting for T_j to release a lock
- Lock Manager: Adds edges when queuing lock requests, removes edges when granting lock requests
- Cycles imply deadlocks

Example

- An example schedule and deadlock

T ₁	T ₂	T ₃	T ₄
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	



Deadlocks

- Waits-for graph: Repeatedly checked for cycles - deadlocks
- Deadlocks are resolved by aborting a transaction that is on a cycle and releasing its locks
- Action allows some waiting transactions to proceed
- Choice of which transaction to abort - many criteria : one with fewest locks, one that has done least work, one that is farthest from completion etc.
- Transactions that are repeatedly started - favored and allowed to complete
- Alternative to maintaining a waits-for graph: Identify deadlocks through a timeout mechanism
- Abort transactions that have been waiting too long

Deadlock Prevention

- Empirical Studies: Indicate deadlocks are relatively infrequent and detection based schemes are good in practice
- However, high levels of contention for locks - increase likelihood of deadlocks.
- Hence we need prevention based schemes
- Prevention of Deadlocks - Give priority to transactions
- Ensures lower priority transactions do not wait for higher priority transactions
- Priority - **Timestamp** a transaction when it begins
- Lower the timestamp - higher the priority (Older the transaction, higher the priority)

Deadlock Prevention

- T_i requests a lock and T_j holds a conflicting lock
- Lock manager uses one of the following policies:
 - ▷ **Wait-Die:** If T_i has higher priority, it is allowed to wait; otherwise it is aborted.
 - ▷ **Wound-Wait:** If T_i has higher priority, abort T_j ; Otherwise, $T - i$ waits.
- Wait-Die scheme - Lower priority transactions can NEVER wait for higher priority transactions
- Wound-Wait scheme - converse is true
- Problem: Need to ensure that no transaction is perennially aborted because it never has sufficient high priority
- When transaction is restarted, give the SAME timestamp as it had originally.
- Ensures that a transaction gets a chance to age and get higher priority to obtain all locks

Remarks on Deadlock prevention schemes

- Wait-Die scheme: Nonpreemptive
- Only a transaction *requesting* a lock can be aborted
- As transactions age, their priority increases and they tend to wait for younger and younger transactions
- Younger transaction that conflicts with an older one may be repeatedly aborted - disadvantage when compared with wound-wait
- A transaction that has all the locks it needs is NEVER aborted for deadlock reasons - An advantage when compared with wound-wait

Conservative 2PL

- **Conservative 2PL:** A variant of 2PL that can also prevent deadlocks
- A transaction obtains all the locks it will **EVER** need when it begins or blocks waiting for these locks to become available
- Ensures that there will be **NO** deadlocks
- A transaction that already holds some locks will not block waiting for other locks
- Lock contention is heavy: Conservative 2PL can reduce the time that locks are held on average
- Transactions that hold locks are never blocked
- A transaction acquires locks earlier and if lock contention is low, then locks are held longer
- In practice - it is difficult to know apriori what are all the locks a transaction would ever need.
- Overhead for setting locks - high (transactions have to release all locks and get them again if it fails to obtain even a single lock it needs) - Not used in practice

Specialized Locking Techniques