Transactions

Transactions

- ACID Properties
- Concurrency Control
  - Schedules
    - Serial Schedules
  - Serializability
    - Invalid Schedules
  - Conflict Serializability

Introduction

- A transaction is a single execution of a user program in a DBMS
  - e.g. Transferring money between two bank accounts
    - If money is transferred between two bank accounts, *both* accounts' records must be updated
- A single *transaction* may result in *multiple actions*
  - For performance reasons, these actions may be interleaved with actions of another transaction
- Transactions should have the **ACID** properties

ACID

ACID Properties

- Transactions should be **atomic**
  - Either all or none of a transaction's actions are carried out
- A transaction must preserve DB **consistency**
  - The responsibility of the DB designers and the users
- A transaction should make sense in **isolation**
  - Transactions should stand on their own and
  - Should be protected from the effects of other concurrently executing transactions
- Transactions should be **durable**
  - Once a transaction is successfully completed, its effects should persist, even in the event of a system crash

Atomicity

- A transaction that is interrupted may leave the DB in an inconsistent state
  - Transactions consist of multiple actions, and are consistent only when considered in their entirety
  - Transactions must therefore be atomic
    - **All or Nothing!**
- A DB will remain consistent if transactions are
  - Consistent
  - Processed as if they occur in some serial order
  - Atomic, ensuring that no partial transactions occur

Consistency

- A DB is in a consistent state if it satisfies all of the constraints of the DB schema
  - For example key constraints, foreign key constraints
- A DBMS typically cannot detect all inconsistencies
  - Inconsistencies often relate to domain specific information
  - And may be represented by triggers, or policies
- Users are responsible for transaction consistency
  - This includes programs that access the DB
    - Casual users should only interact with a DB via programs that are consistency preserving
    - Expert users may interact directly with a DB

Examples

- Entering a new Patient record with a duplicate MSP
  - Allowing this transaction would leave the DB in an inconsistent state so the DB rejects the transaction
- Customers are only allowed if they have an account, accounts are only allowed if they have a customer
  - Entering a new customer who owns a new account is impossible if the DB attempts to maintain this constraint
  - The constraint can be modeled by an application program
- Employees are not allowed to be customers, but SIN is not recorded for customers
  - The constraint must be maintained by policy

Atomicity and Consistency

- Consider transferring money between two accounts
  - The database will only remain consistent if both sides of the transfer are processed
  - If the transfer is interrupted there is a risk that one half is processed and the other is not
- Note that this principle applies to almost any accounting entry
  - Most accounting systems use *double-entry bookkeeping*
- Many other DB transactions are composed of multiple actions

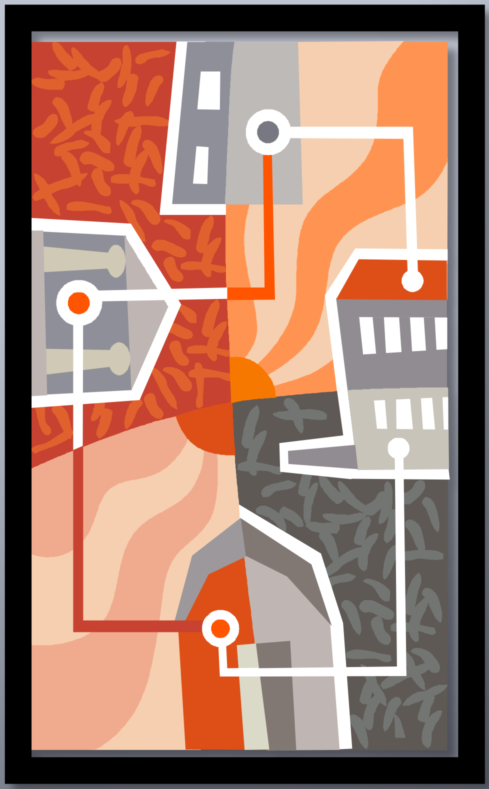
Isolation

- Multiple transactions may be interleaved
  - That is, processed concurrently
  - The net effect of processing the transactions should be the same as executing them in *some* serial order
    - There is no guarantee *which* serial order is chosen
- Consistency and isolation
  - If transactions leave the DB in a consistent state, and
  - If transactions are executed serially then,
  - The resulting DB should be in a consistent state

Durability

- Transactions are first processed in main memory
  - And later written to disk
    - For example when the affected memory pages are replaced
  - If the system crashes in between these two processes there is a possibility that the transaction is lost
- A DBMS maintains a *log* that records information about all writes to the database
  - In the event of a system crash the log is used to restore transactions that were not written to disk
- To achieve durability it is also necessary to maintain backups of the database

Concurrency



Scheduler

- The *scheduler* is responsible for executing reads and writes
  - Reads and writes take place in main-memory
  - Not the disk
- The scheduler may need to call on the buffer manager to read a page into main memory
- A DB element in main memory may be acted on by more than one transaction
  - These actions may interact

Transactions

- A single transaction consists of a series of actions
  - A list of reads and writes of objects in the DB
    - Denote reads of object *A* as *R*(*A*), and
    - Writes of object *A* as *W*(*A*)
- A transaction's last action is either to *commit* or *abort*
  - If a transaction is aborted all of its actions must be undone
- Assumptions
  - Transactions only interact with each other via reads and writes, and do not exchange messages
  - A DB is a *fixed* collection of *independent* objects
    - This assumption is not realistic, and will be discussed further

Transaction Schedules

- A schedule is a list of the actions contained in a set of transactions
  - An action can be a *read*, *write*, *commit*, or *abort*
  - A schedule lists actions in the order in which they occur
    - i.e. the order in which they are processed by the DB
- A *complete schedule* is one that contains either an abort or commit action for each of its transactions
- A *serial schedule* is one where actions from different transactions are not interleaved
  - All serial schedules leave the DB in a consistent state

Serial Schedule

- This schedule contains two transactions
- The transactions do not interact
  - T1 completes before T2 begins
- Key
  - R = Read
  - W = Write
  - A and B are data objects

T1	T2
R(A)	
R(B)	
A = A + 100	
B = B - 100	
W(A)	
W(B)	
Commit	
	R(A)
	A = A*2
	W(A)
	Commit

Concurrent Execution

- Transaction isolation could be guaranteed by never interleaving transactions
  - However this would negatively impact performance
  - Interleaving two transactions allows the CPU to process one while the other's data is being read from disk
- Interleaving transactions therefore increases system throughput
  - i.e. the average number of transactions completed
  - It allows short transactions to be interleaved with long transactions rather than waiting for their completion

Serializability

- A *serializable* schedule is guaranteed to be the same as *some* serial schedule
  - i.e. where one transaction is processed in its entirety before processing the next
  - Different serial orders of transactions may result in different results
- If a schedule is not serializable the DB may not be in a consistent state after processing the transactions
  - A schedule containing two consistency preserving transactions may therefore result in an inconsistent DB

Serializable Schedule

- The schedule contains two transactions
- The transactions do not access the same data objects
- Is it serializable?
  - Yes!
  - Though interleaved, the actions are unrelated

T1	T2
R(A)	
A = A + 100	
W(A)	
	R(C)
	C = C * 2
	W(C)
R(B)	
B = B - 100	
W(B)	
Commit	
	R(D)
	D = D * 2
	W(D)
	Commit

Is This Schedule Serializable?

- This schedule also has two transactions
- These transactions *do* access the *same* data objects
- Is it serializable?
  - **Yes!**
  - Note the order in which the objects are read and written

T1	T2	A	B
R(A)		25	200
A = A + 100		125	
W(A)			
	R(A)		
	A = A * 2	250	
	W(A)		
R(B)			
B = B - 100			100
W(B)			
Commit			
	R(B)		
	B = B * 2		200
	W(B)		
	Commit	250	200

… And This One?

- This schedule is similar to the previous one
- The two transactions also access the same data objects
- Is it serializable?
  - Yes!
  - Again, note the order

T1	T2	A	B
	R(A)	25	200
	A = A * 2	50	
	W(A)		
R(A)			
	R(B)		
	B = B * 2		400
	W(B)		
A = A + 100		150	
W(A)			
R(B)			
B = B - 100			300
W(B)			
	Commit		
Commit		150	300

Interleaved Execution Anomalies

- There are three ways in which transactions in a schedule can *conflict*
  - So that even if the individual transactions are consistent the DB can be in an inconsistent state after the execution
- Two actions conflict if they act on the same data object and if one of them is a write
  - *write-read* conflicts (*WR*)
  - *read-write* conflicts (*RW*)
  - *write-write* conflicts (*WW*)

Write-Read Conflicts

- One transaction writes data, and a second transaction reads that data *before it commits*
  - Referred to as a *dirty read*
  - The first transaction may not be complete before the second transaction begins processing
- If the first transaction is not complete, the DB may be in an inconsistent state at that point
  - Note – recall that *during* the processing of a transaction, the database may be temporarily inconsistent

Serial Schedule 1 …

- T1 – Transfer $100 from account B to account A
- T2 – Double amounts in both accounts A and B
- The diagram shows a serial schedule T1, T2
- A – 250
- B – 200

T1	T2	A	B
R(A)		25	200
A = A + 100		125	
W(A)			
R(B)			
B = B - 100			100
W(B)			
Commit			
	R(A)		
	A = A * 2	250	
	W(A)		
	R(B)		
	B = B * 2		200
	W(B)		
	Commit	250	200

… and Serial Schedule 2

- T1 – Transfer $100 from account B to account A
- T2 – Double amounts in both accounts A and B
- The diagram shows a serial schedule T2, T1
- A – 150
- B – 300

T1	T2	A	B
	R(A)	25	200
	A = A * 2	50	
	W(A)		
	R(B)		
	B = B * 2		400
	W(B)		
	Commit		
R(A)			
A = A + 100		150	
W(A)			
R(B)			
B = B - 100			300
W(B)			
Commit		150	300

WR Conflict Schedule

- T1 – Transfer $100 from account B to account A
- T2 – Double amounts in both accounts A and B
- The diagram shows an interleaved schedule with a *dirty read* of A
- Result is not the same as either serial schedule
- A – 250
- B – 300

T1	T2	A	B
R(A)		25	200
A = A + 100		125	
W(A)			
	R(A)		
	A = A * 2	250	
	W(A)		
	R(B)		
	B = B * 2		400
	W(B)		
	Commit		
R(B)			
B = B - 100			300
W(B)			
Commit		250	300

Compare Schedules

T1	T2	A	B
R(A)		25	200
A = A + 100		125	
W(A)			
	R(A)		
	A = A * 2	250	
	W(A)		
	R(B)		
	B = B * 2		400
	W(B)		
	Commit		
R(B)			
B = B - 100			300
W(B)			
Commit		250	300

WR Conflict

T1	T2	A	B
R(A)		25	200
A = A + 100		125	
W(A)			
	R(A)		
	A = A * 2	250	
	W(A)		
R(B)			
B = B - 100			100
W(B)			
Commit			
	R(B)		
	B = B * 2		200
	W(B)		
	Commit	250	200

Serializable

Read-Write Conflicts

- One transaction reads data which is then written by a second transaction
  - Referred to as an *unrepeatable read*
  - Although the first transaction did not modify the data, if it tries to read it again it would obtain a different result
- What if the first transaction modifies the data based on the value it obtained from its initial read?
  - This value is no longer correct,
  - Therefore an error or an inappropriate modification may result

Serial Schedule

- T1 – $User_1$ considers purchasing 21 items
  - Reads A
  - Waits (contemplating)
  - Reads A again
  - Writes A
- T2 – $User_2$ wants to purchase 13 items
- The diagram shows a serial schedule T1, T2

T1	T2
R(A) items = 25	
R(A) items = 25	
A = A - 21	
W(A) items = 04	
Commit	
	R(A) items = 04
	no write*
	Commit
*4 – 13 is negative so *user program* will not remove the items	

RW Conflict Example - 1

- T1 – $User_1$ considers purchasing 21 items
  - Reads A
  - Waits (contemplating)
- T2 – $User_2$ purchases 13 items
- T1 – When $user_1$ reads the data again the amount has changed
  - making the purchase impossible
  - $user_1$ is upset!

T1	T2
R(A) items = 25	
	R(A) items = 25
	A = A - 13
	W(A) items = 12
	Commit
R(A) items = 12	
no write*	
Commit	
*12 – 21 is negative so user program will remove the widgets	

Write-Write Conflicts

- One transaction writes data that has already been read or written by a second, incomplete, transaction
  - Referred to as a *lost update*, a special case of an unrepeatable read
  - If the initial action of the first transaction was a read, a subsequent write is replaced by the second transaction
- Other WW conflicts exist that do not involve unrepeatable reads

A blind write is a write with no prior read

  - Blind writes to objects whose values should be related
  - Lost updates caused by aborted transactions

Serial Schedule

- T1 – Increase account A by $10,000
- T2 – Decrease account A by $7,000

T1	T2
R(A) 21,000	
W(A) 31,000	
Commit	
	R(A) 31,000
	W(A) 24,000
	Commit

WW Lost Update

- T1 – Increase account A by $10,000
- T2 – Decrease account A by $7,000
- The diagram shows an interleaved schedule with a lost update (T1)
  - Caused by an unrepeatable read

T1	T2
R(A) 21,000	
	R(A) 21,000
W(A) 31,000	
Commit	
	W(A) 14,000
	Commit

Serial Schedule, Blind Writes

- A and B should always have the same value
- T1 and T2 both change the values of A and B
  - In both cases the existing values are not read - *blind writes*
- In this serial schedule the relationship between A and B is maintained

T1	T2
W(A) 10,000	
W(B) 10,000	
Commit	
	W(A) 7,000
	W(B) 7,000
	Commit

Lost Update with Blind Writes

- A and B should always have the same value
- A and B do not have the same values after both the transactions have committed
- Remember that transactions may be inconsistent *during* processing

T1	T2
W(A) 10,000	
	W(B) 7,000
W(B) 10,000	
Commit	
	W(A) 7,000
	Commit

Equivalent Schedules

- The scheduler does not consider the details of calculations
  - i.e. it does not know what a transaction is doing
  - It assumes that if a transaction *could* result in the DB being inconsistent it will

Is there a serial schedule equivalent to this schedule?

T1	T2	A	B
R(A)		25	200
A = A + 100		125	
W(A)			
	R(A)		
	A = A + 200	325	
	W(A)		
	R(B)		
	B = B + 200		400
	W(B)		
	Commit		
R(B)			
B = B - 100			300
W(B)			
Commit		325	300

Serializable Schedules

- When is a non-serial schedule *guaranteed* to leave a DB in a consistent state?
  - If it is equivalent to some serial schedule
  - That is, if the schedule is *serializable*
- We will look at two tests of serializability
  - View equivalent
  - Conflict equivalent

View Equivalence

- Two schedules are *view-equivalent* if
  - They contain the same transactions
  - Each transaction reads the same value for each data object in each schedule
    - Before modification
    - And after modification by one of the transactions
  - The same transaction must perform the final write of each data object
- A schedule is *view-serializable* if it is view-equivalent to some serial schedule

View-Equivalent Schedules

T1	T2
R(A) initial read - T1	
W(A)	
R(B) initial read - T1	
W(B)	
Commit	
	R(A) written by T1
	W(A) final write - T2
	R(B) written by T1
	W(B) final write - T2
	Commit

Serial Schedule

T1	T2
R(A) initial read - T1	
W(A)	
	R(A) written by T1
	W(A) final write - T2
R(B) initial read - T1	
W(B)	
Commit	
	R(B) written by T1
	W(B) final write - T2
	Commit

View Equivalent

Not View-Equivalent

T1	T2
R(A) initial read - T1	
W(A)	
R(B) initial read - T1	
W(B)	
Commit	
	R(A) written by T1
	W(A) final write - T2
	R(B) written by T1
	W(B) final write - T2
	Commit

Serial Schedule

T1	T2
R(A) initial read - T1	
W(A)	
	R(A) written by T1
	W(A) final write - T2
	R(B) initial read - T2
	W(B)
R(B) written by T2	
W(B) final write - T1	
Commit	
	Commit

Not Equivalent

Conflict Serializability

- View-serializability is hard to prove and implement
  - As it is necessary to find an equivalent serial schedule
    - Which is an NP-hard problem
- *Conflict-serializability* is a practical alternative
  - Two schedules that are *conflict* equivalent have the same effect on a DB
  - A conflict-serializable schedule is always view-serializable
  - In some (rare) cases a view-serializable schedule is not conflict-serializable
    - This only occurs when the schedule contains blind writes

Conflicts

- Two actions *conflict* if they operate on the same DB object and one of them is a write
  - Note that conflicts are often unavoidable and do not necessarily result in inconsistency
- The outcome of a schedule depends on the order of the conflicting operations
  - Non-conflicting operations can be reordered with no impact on the final result
- If the conflicting actions of two schedules are in the same order the schedules are *conflict equivalent*

Conflict Equivalence

- Two schedules are *conflict equivalent* if
  - They involve the same actions of the same transactions
  - They order each pair of conflicting actions in the same way
- A schedule is *conflict serializable* if it is conflict equivalent to some serial schedule
  - Some serializable schedules are not conflict serializable
    - Such a schedule has conflicting actions that cannot be ordered in the same way as a serial schedule, but that
    - Does not result in a different state from a serial schedule

Identifying Conflicts

- Two actions of the same transaction always conflict
  - e.g. $R_{T1}(X)$, $W_{T1}(Y)$
  - Since the *order* of actions *within* a transaction cannot be changed
- Two writes of the same database object by different transactions conflict
  - e.g. $W_{T1}(X)$, $W_{T2}(X)$
- A read and a write of the same database object by different transactions conflict
  - e.g. $R_{T1}(X)$, $W_{T2}(X)$ or $W_{T1}(X)$, $R_{T2}(X)$

Shorthand Transaction Schedules

- Transaction schedules can be written in shorthand, denote
  - $r_t(O)$ – where r is a read, t is the transaction and O is the data object
    - $r_1(A)$ – read of object A by transaction 1
  - $w_t(O)$ – where w is a write, t is the transaction and O is the data object
    - $w_2(B)$ – write of object B by transaction 2
  - The order from left to right shows the order in which the actions take place

Shorthand Transaction Schedules

- Example schedule
  - $r_1(A)$, $w_1(A)$, $r_2(A)$, $w_2(A)$, $r_1(B)$, $w_1(B)$, $r_2(B)$, $w_2(B)$
- We can demonstrate that a schedule is or is not conflict equivalent by swapping actions
  - Except that actions that conflict are not allowed to be swapped
  - If actions can be swapped such that the schedule becomes a serial schedule it is conflict serializable

Example 1

- $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$
- Goal – try to swap actions to create a serial schedule
- $r_1(A), w_1(A), r_2(A), r_1(B), w_2(A), w_1(B), r_2(B), w_2(B)$
- $r_1(A), w_1(A), r_1(B), r_2(A), w_2(A), w_1(B), r_2(B), w_2(B)$
- $r_1(A), w_1(A), r_1(B), r_2(A), w_1(B), w_2(A), r_2(B), w_2(B)$
- $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$
- This technique is not used by the scheduler to determine if a schedule is (conflict) serializable
  - But it allows us to reason about schedules

Example 2

- $r_1(A)$, $w_1(A)$, $r_2(A)$, $w_2(A)$, $r_2(B)$, $w_2(B)$, $r_1(B)$, $w_1(B)$
- Goal – move T1's read and write of B up to the front, just after T1's read and write of A
  - First swap $r_1(B)$ and $w_2(B)$
  - But they conflict, because they act on the same object
- $r_1(A)$, $w_1(A)$, $r_2(A)$, $w_2(A)$, $r_2(B)$, $w_2(B)$, $r_1(B)$, $w_1(B)$
- The schedule cannot be rearranged into a serial schedule
  - And is therefore not conflict serializable

Conflict-Equivalent Schedules

T1	T2
$R(A)^{1}$	
$W(A)^{1}$	
$R(B)^{1}$	
$W(B)^{1}$	
Commit	
	$R(A)^{2}$
	$W(A)^{2}$
	$R(B)^{2}$
	$W(B)^{2}$
	Commit

Serial Schedule

T1	T2
$R(A)^{1}$	
$W(A)^{1}$	
	$R(A)^{2}$
	$W(A)^{2}$
$R(B)^{1}$	
$W(B)^{1}$	
Commit	
	$R(B)^{2}$
	$W(B)^{2}$
	Commit

Conflict Equivalent

Not Conflict-Equivalent

T1	T2
$R(A)^{1}$	
$W(A)^{1}$	
$R(B)^{1}$	
$W(B)^{1}$	
Commit	
	$R(A)^{2}$
	$W(A)^{2}$
	$R(B)^{2}$
	$W(B)^{2}$
	Commit

Serial Schedule

T1	T2
$R(A)^{1}$	
$W(A)^{1}$	
	$R(A)^{2}$
	$W(A)^{2}$
	$R(B)^{1}$
	$W(B)^{1}$
$R(B)^{2}$	
$W(B)^{2}$	
Commit	
	Commit

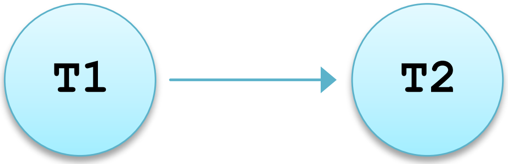
Not Conflict Equivalent

Precedence Graphs

- Conflicts between transactions can be shown in a *precedence graph*
  - Also known as a serializability graph
- A precedence graph for a schedule contains
  - Nodes for each committed transaction
  - An arc from transaction T_i, to T_j if an action of T_i *precedes* and *conflicts* with one of T_j's of actions
- A schedule is only conflict serializable if and only if its precedence graph is *acyclic*

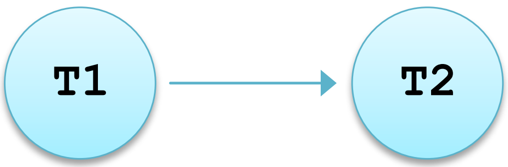
Precedence Graph 1

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



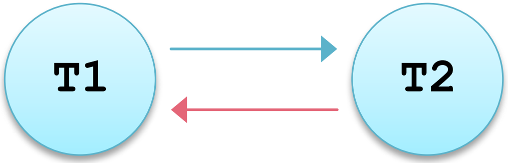
Precedence Graph 2

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



Precedence Graph 3

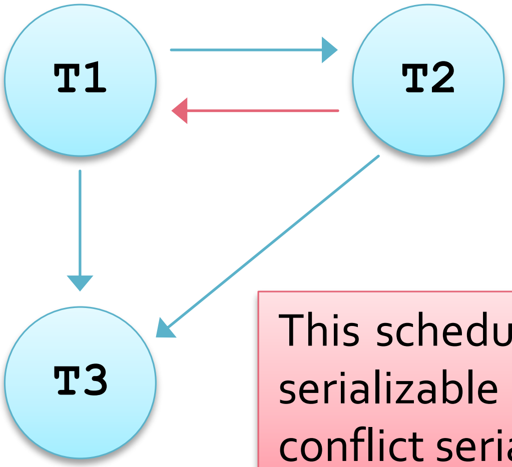
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	
Commit	
	Commit



The cycle indicates that the schedule is not conflict serializable

Precedence Graph 4

T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit



This schedule is view serializable but not conflict serializable

Aborted Transactions

- If a transaction is aborted all its actions have to be reversed as if the actions had never occurred
  - To achieve this other transactions may also have to be aborted in a *cascading abort*
    - This may be required when transactions have acted on the same objects as the transaction to be aborted
- A transaction that has already been committed cannot be aborted as part of a cascading abort
  - If an aborted transaction is interleaved with a committed transaction the schedule may be *unrecoverable*
  - In a *recoverable* schedule transactions only commit after all transactions *that they read* have committed

Serial Schedule

- T1 adds $2,000 to A
- T2 adds $3,000 to A
- In this serial schedule A is aborted
  - Any changes made by A are reversed
- The value of A would be the same regardless of the order
  - T1, T2 or T2, T1

T1	T2
R(A) 10,000	
W(A) 12,000	
Abort	
	R(A) 10,000
	W(A) 13,000
	Commit

Lost Update Due to Abort

- T1 adds $2,000 to A
- T2 adds $3,000 to A
- T2 commits after A has aborted so the results of both transactions are lost
  - Since A's value is reset to 10,000
- This schedule is another example of an unrepeatable read

T1	T2
R(A) 10,000	
W(A) 12,000	
	R(A) 12,000
	W(A) 15,000
Abort	
	Commit

Unrecoverable Schedule

- T1 – Deduct $3,000 from account
- T2 – Add interest of 10% to account
- To reverse T1 it would also be necessary to reverse T2
- But T2 has already committed

T1	T2
R(A) 10,000	
W(A) 7,000	
	R(A) 7,000
	W(A) 7,700
	Commit
Abort	

Locking Protocols

Concurrency Control

- A DBMS must ensure that schedules are
  - Equivalent to some serial schedule (serializable) and
  - Recoverable
- Often achieved by using a *locking protocol*
  - A *lock* is associated with a particular DB object and
  - Restricts access to that object
- The most widely used locking protocol is *Strict Two-Phase Locking* (*Strict 2PL*)
  - A variant of the *Two-Phase Locking* (*2PL*) protocol

Locking Basics

- There are two kinds of lock
  - If a transaction wants to read an object it first has to request a *shared* lock on that object
  - If a transaction wants to modify an object it first has to request an *exclusive* lock on that object
    - Which also allows the transaction to read the object
- When a transaction requests a lock either
  - The lock is granted, the transaction becomes the owner of that lock, and the transaction continues, or
  - The transaction is suspended until it is able to be granted the requested lock

Shared and Exclusive Locks

- Shared locks allow transactions to read objects
  - Multiple shared locks can be granted to different transactions on the same database object
  - Allowing all of the transactions with shared locks to read the object
- Exclusive locks allow transactions to write objects
  - Exclusive locks are only granted on objects with no other locks
    - Shared or exclusive
  - No other locks are granted on objects that are already exclusively locked

Locking Issues

- When should a transaction issue a lock?
  - It must ensure that a schedule is both serializable and recoverable
- When should a transaction release a lock?
- What are the side effects of locking, and how are they dealt with?
  - Deadlock prevention and detection
- How is locking implemented?

Two-Phase Locking (2PL)

- Shared or exclusive locks are requested before each read or write respectively, and
  - A transaction's lock requests must precede its unlocks
    - Once it has released any locks it cannot request additional locks
    - 2PL transactions therefore have *growing* and *shrinking* phases
- 2PL ensures that precedence graphs are acyclic
  - Resulting in conflict-serializable schedules
  - When a conflict occurs, the transaction causing the conflict waits until the other transaction finishes

2PL – Read Write Conflict

- This schedule includes an unrepeatable read
- Can this schedule occur with 2PL?
  - No!

T1	T2
R(A) items = 25	
	R(A) items = 25
	W(A) items = 12
	Commit
R(A) items = 12	
no write*	
Commit	
*intending to purchase 21 items, since 12 – 21 is negative *user program* will not remove them	

2PL – Write Read Conflict

- This schedule includes a dirty read
- Can this schedule occur with 2PL?
  - No!

T1	T2
R(A) 10,000	
W(A) 7,000	
	R(A) 7,000
	W(A) 7,700
	R(B) 12,000
	W(B) 13,200
	Commit
R(B) 13,200	
W(B) 16,200	
Commit	

2PL – Write Write Conflict

- This schedule includes a lost update
- Can this schedule occur with 2PL?
  - No!

T1	T2
R(A) 21,000	
	R(A) 21,000
W(A) 31,000	
Commit	
	W(A) 14,000
	Commit

2PL Schedule

Write Write Conflict

T1	T2
R(A) 21,000	
	R(A) 21,000
W(A) 31,000	
Commit	
	W(A) 14,000
	Commit

Schedule with 2PL

T1	T2
X(A)	
R(A) 21,000	
	T2 suspended
W(A) 31,000	…
Commit	…
	X(A)
	R(A) 31,000
	W(A) 24,000
	Commit

2PL and Aborted Transactions

- A transaction in this schedule is aborted
- The schedule is unrecoverable
- Can this schedule occur with 2PL?
  - **Yes!**

T1	T2
R(A) 10,000	
W(A) 7,000	
	R(A) 7,000
	W(A) 7,700
	Commit
Abort	

Unrecoverable 2PL Schedule

Unrecoverable Schedule

T1	T2
R(A) 10,000	
W(A) 7,000	
	R(A) 7,000
	W(A) 7,700
	Commit
Abort	

Schedule with 2PL

T1	T2
X(A)	
R(A) 10,000	
W(A) 7,000	
release lock	
	X(A)
	R(A) 7,000
	W(A) 7,700
	Commit
Abort	

The 2PL protocol can be modified to prevent unrecoverable schedules

Strict 2PL

- Strict 2PL is similar to 2PL
  - Write and read operations request shared and exclusive locks respectively
- Strict 2PL differs on when locks are *released*
  - All locks held by a transaction are released only when the transaction is *completed* (committed or aborted)
- This prevents transactions from reading DB objects which were modified by uncommitted transactions

Strict 2PL Notes

- Only safe interleaving of transactions is allowed
  - If two transactions access different DB objects they are allowed concurrent access, so interleaving is possible
  - If two transactions require access to the same object, and one wants to modify it, their actions are ordered serially
- Strict 2PL prevents unrecoverable schedules from occurring
  - The protocol only releases locks when a transaction ends
  - Which prevents a transaction from accessing a DB object that was modified by a prior transaction that aborts

Strict 2PL and Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic
- The precedence graph for any Strict 2PL schedule is acyclic
  - If T2 writes an object written by T1, then T1 must have released its lock on that object before T2 obtained its lock
  - Under Strict 2PL, transactions only unlock data objects when they commit (or abort)
  - Therefore, two transactions cannot precede each other, forming a cycle in the precedence graph

Lock Management

- The *lock manager* keeps track of which locks have been issued to transactions
  - It maintains a *lock table*, a hash table with the data object ID as the key, each lock table entry contains
    - The number of transactions holding a lock on the object
    - Type of lock (shared or exclusive)
    - A pointer to a queue of lock requests
- The DBMS also maintains an entry for each transaction in a *transaction table*
  - Including a pointer to a list of locks held by the transaction

Lock Requests

- A transaction that needs a lock issues a lock request
- Shared lock requests are only granted if
  - The request queue is empty, and
  - The object is not locked in exclusive mode
- Exclusive lock requests are only granted if
  - There is no lock on the object and the request queue is empty
- In any other case the lock is not granted
  - The request is added to the request queue, and
  - The transaction is suspended

Unlocking and Starvation

- When a transaction aborts or commits it releases its locks
  - The lock table is updated for the object
  - The request at the head of the queue is considered and if it can be granted it is unsuspended and given the lock
    - If several requests for a shared lock are at the head of the queue they can all be granted
- If T_1 has a shared lock on an object and T_2 requests an exclusive lock T_2 is suspended
  - If T_3 then requests a shared lock on the same object, T_3 is also suspended even though it is compatible with T_1
  - This rule ensures that T_2 does not starve

Additional Lock Types

- Update locks
  - An update lock allows a transaction to read a record
    - But can be later upgraded to an exclusive lock
  - Update locks can be granted when another transaction has a shared lock
  - And prevent any other locks being taken on the object
- Increment locks allow objects to be incremented or decremented
  - Multiple increment locks are allowed on the same object
    - Other locks are not granted on objects with increment locks

Strict 2PL and Deadlock

T1	T2
X(A)	
R(A)	
W(A)	
	X(B)
	R(B)
	W(B)
X(B) - denied	
… suspended	
	X(A) - denied
	… suspended

T1: R(A), W(A), R(B), W(B)

T2: R(B), W(B), R(A), W(A)

- Deadlock – when two or more transactions are suspended
- Waiting for each other to complete and unlock an object
- Deadlock is not prevented by Strict 2PL
- Deadlock can be detected and dealt with
- Or avoided

Other Locking Issues

Phantom Problem

- Initially we assumed that a database is a *fixed* collection of *independent* objects
  - In practice, database transactions may include insertions, violating the first part of this assumption
- Insertions may result in unrepeatable reads
  - Locks only apply to DB objects that exist
  - Using the 2PL protocol, all records that meet some criteria can be locked
  - This does not prevent additional records (that meet the criteria) being inserted during the lock
  - Such records are referred to as *phantoms*

Phantom Problem Example

- T1 reads the Patient table to find the ages of the oldest patients suffering from scurvy and leprosy
  - T1 locks all pages for patients with scurvy, and finds the oldest such patient (who is 77)
- T2 inserts a new patient, aged 93, with scurvy
  - The page that the patient is inserted on is not locked by T1
  - T2 now locks the page containing the oldest patient with leprosy (who is 89) and deletes the record
  - T2 commits and releases its locks
- T1 finds the oldest patient with leprosy, who is 88
  - T1's result would not be possible from any serial schedule

Phantom Diseases

- Query T1 returns the two ages as
  - Scurvy – 77
  - Leprosy – 88
- This is not equivalent to any serial schedule
- Serial schedules would either return
  - 77, and 89, or
  - 93, and 88
- This occurred because T1 locked specific pages
  - Rather than the set of patients

T1	T2
S(scurvy pages)	
R(scurvy)$^{\text{age } 77}$	
	X(new scurvy)
	W(new scurvy)$^{\text{add } 93}$
	X(leprosy)
	W(leprosy)$^{\text{del } 89}$
	Commit
S(leprosy)	
R(leprosy)$^{\text{age } 88}$	
Commit	

Phantoms and Serializability

- The phantom problem can lead to schedules that are not equivalent to any serial schedule
  - Conflict serializability does not guarantee serializability if items are added to the DB
- This can be solved by using *predicate locking*
  - All records that fall within a range of values are locked
  - General predicate locking is expensive to implement
    - Key-range locking (locking a range of key values) is more common
  - *Index locking* can be used if the DB file has B+ tree index on the attribute used in a transaction's condition

Locking Units

- The size of DB object that can be locked varies
  - Largest lock unit – the entire DB
  - Smallest lock unit – single record (table row)
  - Or: a table, or page
- The size of the locking unit affects performance
  - Smaller lock units generally allow more concurrency, but
  - Complex transactions may need access to many such objects, leading to high overhead and large lock queues
- The solution is to allow multiple lock granularity
  - With a separate lock table for objects of each type

Intent Modes

- Multiple lock granularity results in a problem
  - If T2 holds a lock on a *record*, and T1 wants a lock on the same *page*, how is T1 prevented from overriding T2's lock?
- Introduce two new lock types to indicate that a lock is held at a finer granularity
  - Intention shared (IS) – conflicts only with X locks, and
  - Intention exclusive (IX) – conflicts with S and X locks
  - Intent locks are applied to all ancestors of a locked object
- IS and IX locks can co-exist with other IS and IX locks at the same lock table

Lock Hierarchy Model

- Consider the DB as a tree
- Each node represents a lock unit
- Each level represents a different granularity
  - The entire DB is the root
  - Individual records are leaf nodes
- To lock a target lock unit (a node)
  - Request a lock on every node on the path from the root to the target lock unit
  - All locks are IS (or IX), except the target, which is S (or X)

Lock Modes Summary

- Shared
  - Implies locks on all nodes below the current one
- eXclusive
  - Implies locks on all nodes below the current one
- Intention Shared
  - Intent to set an S lock at a finer granularity
- Intention eXclusive
  - Intent to set an X lock at a finer granularity
- SIX (S and IX)
  - Commonly used where a transaction needs read an entire file and modify some of the records

Granularity Locking Protocol

- Acquire locks from root to leaf
- Release locks from leaf to root
  - This is necessary to prevent another transaction acquiring a (higher level) conflicting lock
- To acquire an *S* or *IS* mode on a non-root node, all ancestors must be held in *IS* mode
- To acquire an *X*, *SIX* or *IX* mode on a non-root node, all ancestors must be held in *IX* mode
- Use Strict 2PL locking protocol

Browsing Records

- *SIX* locks are used to search a file to find the desired record to update

```
SIX lock the table
for each record in the table
        if (condition is true) //record is the target
                upgrade the S lock to X to lock the record
                update the record
                release the X lock
        end if
end for
release SIX lock
```

Lock Escalation

- What granularity of locking is appropriate for a given transaction?
  - First obtain fine granularity locks (at the record level)
  - When the number of locks granted reaches a threshold
  - Obtain locks at the next higher granularity
- An alternative approach is to start with coarser granularity locks
  - Break the locks into multiple finer granularity locks when contention occurs
  - i.e. Lock de-escalation

Locking and B+ Trees

Locking in B+ Trees

- Problem: How can a leaf node in a B+ tree be locked efficiently?
  - The naive solution is to ignore the tree structure and treat each page as a data object
  - This has very poor performance as the root (and other high level nodes) become bottlenecks
- Two useful observations
  - Higher levels of the tree only direct searches
    - All of the data is in the leaf levels
  - For inserts, a node must be (exclusively) locked only if a split can propagate to it from the leaf

Tree Locking

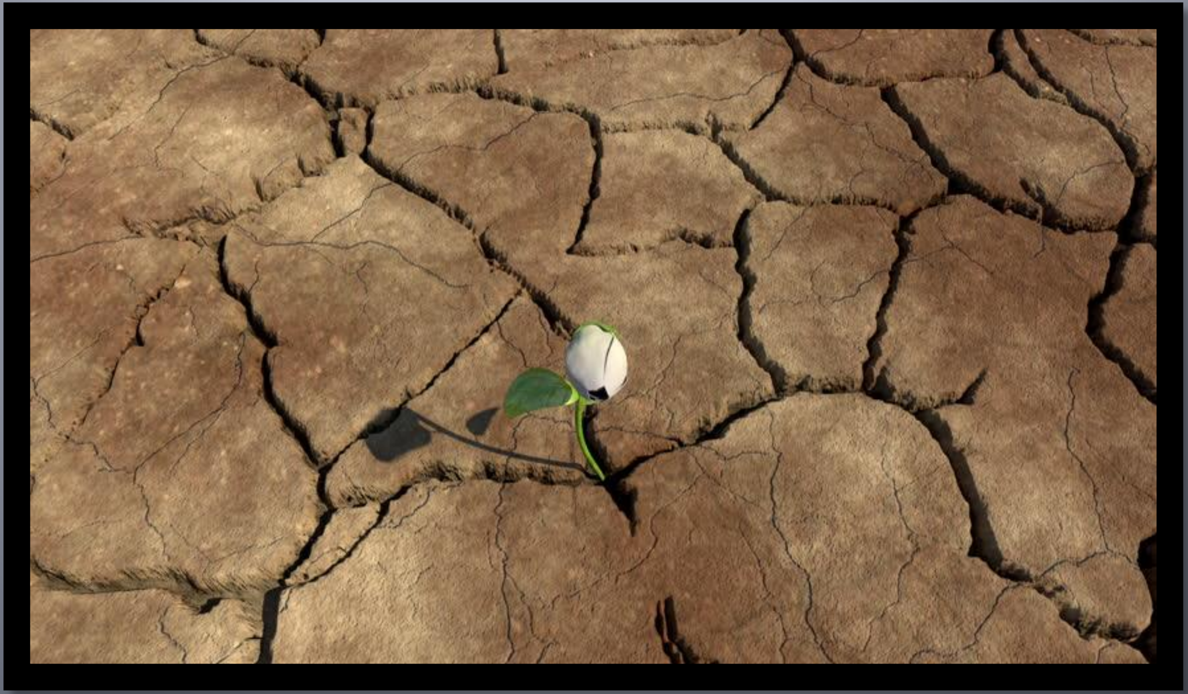
- Searches
  - Obtain shared locks on nodes on the path from root to the leaf
  - As each child is locked, unlock its parent
- Inserts and deletes
  - Start at the root and obtain exclusive locks on the nodes on the path to the desired leaf
  - Check each child to see if it is **safe**, a node is safe if changes will not propagate up the tree
    - For inserts, the node is not full
    - For deletes, the node is not half-empty
  - If a node is safe, release all the locks on its ancestors

Index Locking

- If there is an available index, a transaction can request a lock on the appropriate index page
  - i.e. the leaf page (or bucket) of the B+ tree
  - This prevents any records with key values on that index page being inserted
- An index bucket should be *S* locked to scan the rows pointed to by data entries in that bucket
- An index bucket should be *X* locked to modify any of the rows pointed to by the bucket
  - Or to insert a value in the bucket

Concurrency Control With No Locking

Optimistic Concurrency Control

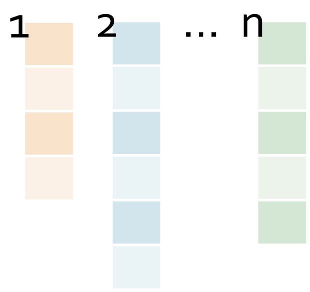


Optimistic Concurrency Control

- Locking protocols are *pessimistic* as they aim to abort or block conflicts
  - This requires overhead when there is little contention
  - In optimistic concurrency control assume that conflicts are rare
- There are two main versions of optimistic CC
  - *Timestamps* – maintain timestamps of transactions and reads and writes of database objects
  - *Validation* – similar to the timestamp system except that data is recorded about the actions of transactions
    - Rather than data about database objects
    - Not discussed (appendix)

Timestamps

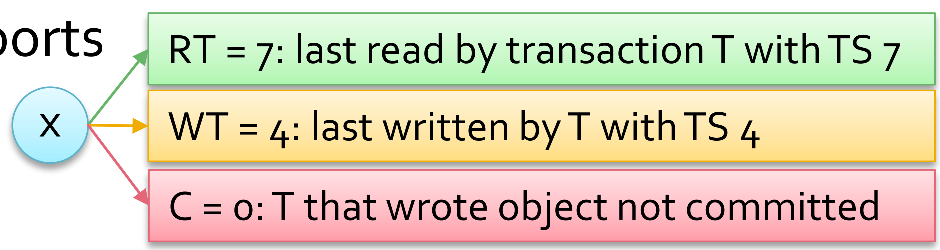
- Transactions are issued *timestamps*
  - Given in ascending order when transactions begin
    - Referred to as TS(T) in this presentation



- Timestamps can be generated
  - By using the system clock
  - By maintaining a counter within the scheduler
- The scheduler maintains a table of active transactions and their timestamps

Timestamp Data

- For each database element record
  - RT(X) – the *read* time of the object X
    - The highest timestamp of a transaction that has read X
  - WT(X) – the *write* time of the object X
    - The highest timestamp of a transaction that has written X
  - C(X) – the *commit* bit
    - True iff the most recent transaction to write X has committed
    - Maintained to avoid one transaction reading data by another transaction that later aborts
    - i.e. a dirty read



Physically Unrealizable Behaviour

- Optimistic concurrency control supposes that transactions are instantaneous

So must be atomic

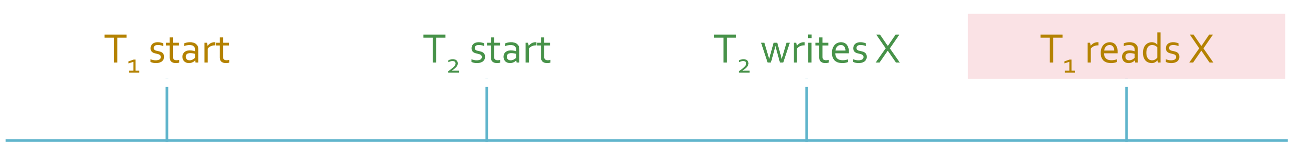
  - That is, all the actions take place at the same time
  - In reality this is, of course, not the case
    - As actions are performed one at a time
    - Possibly interleaved with actions of other transactions
- If the results of transactions could not have occurred if transactions were instantaneous
  - The behaviour is said to be *physically unrealizable*

Read Too Late

- There are two kinds of possible problems that can result in physically unrealizable behaviour
- Read too late

T started before X written

  - Transaction T tries to read X but $TS(T) < WT(X)$
    - Which means that X has been written to by another transaction after T began
  - Transactions are supposed to be instantaneous
    - If so, T would have read X before the later transaction wrote it



Write Too Late

- The second type of physically unrealizable schedule is referred to as write too late
  - T tries to write X but $WT(X) < TS(T) < RT(X)$
    - Or $RT(X) > TS(T)$ i.e. $TS(T) < RT(X)$

Another transaction read X before it was written by T

  - This means that X has been read by another transaction after T began

Commit Bit and Dirty Data

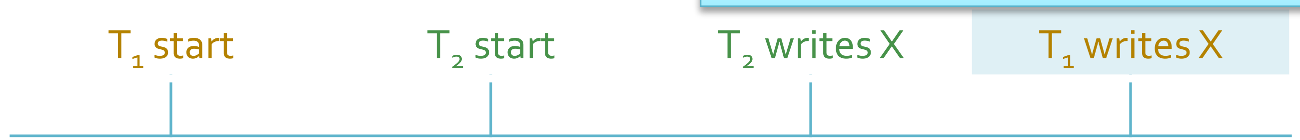
- The commit bit solves problems with *dirty reads*
  - When T_1 reads data after it is written by T_2, but before T_2 commits
    - If T_2 aborts the read by T_1 will be incorrect
    - In this case the Thomas Write Rule should *not* be applied
    - Since T_2's actions should not occur

Thomas Write Rule

If $TS(T_1) < TS(T_2)$ and T_1 and T_2 write to the same object then T1's write should be ignored

Since T_2's timestamp is later than T_1's meaning that T_1 would have been over-written

If T_2 commits T_1's write can be ignored



Scheduler Options

- The scheduler has three options when it receives a read or write request from T
  - Grant the request
  - Abort T and restart it with a new timestamp
    - Referred to as a *rollback*
  - Delay T
    - Decide later whether to grant T's request or abort T
    - Usually when T is waiting for some other transaction to commit

Scheduling Rules – Read Request

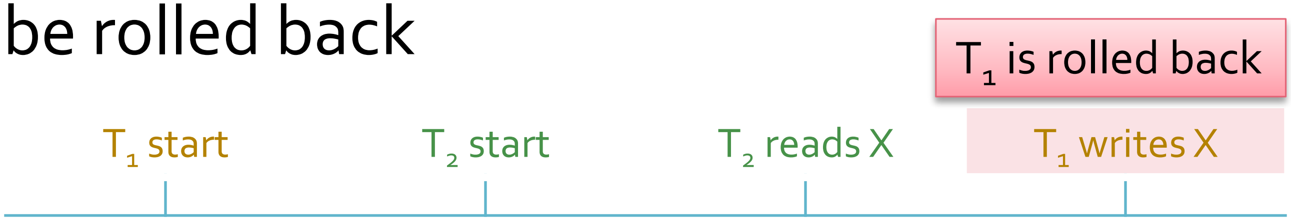
- The scheduler receives a read request $R_T(X)$
- If TS(T) ≥ WT(X) the read is physically realizable
  - If C(X) is true, grant the request and update RT(X)
    - If TS(T) > RT(X), set RT(X) to TS(T)
  - Otherwise delay T until C(X) becomes true, or the transaction that wrote X aborts

- If TS(T) < WT(X) rollback T
  - X has been written by another transaction after T started

Scheduling Rules – Write Request

- The scheduler receives a write request $W_T(X)$
  - There are three possible outcomes
- If TS(T) ≥ RT(X) and TS(T) ≥ WT(X) the write is physically realizable

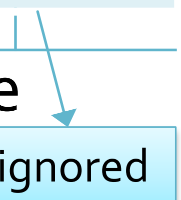
  - Write new value for X, set WT(X) to TS(T) and C(X) to false
    - Set C(X) to true when T commits
- If TS(T) < RT(X) the write is not physically realizable and T must be rolled back



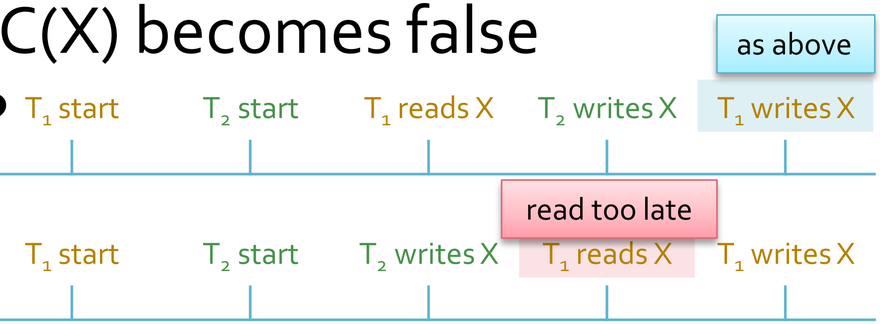
Scheduling Rules – Write Request

- The third possible outcome of a write request involves the Thomas Write Rule
- If $TS(T) \geq RT(X)$ but $TS(T) < WT(X)$ there is a later value in X

  - If C(X) is true ignore T's write – Thomas Write Rule



  - Otherwise delay T until C(X) is true
  - Or proceed with write if C(X) becomes false
- What if T_1 also reads X?
  - or



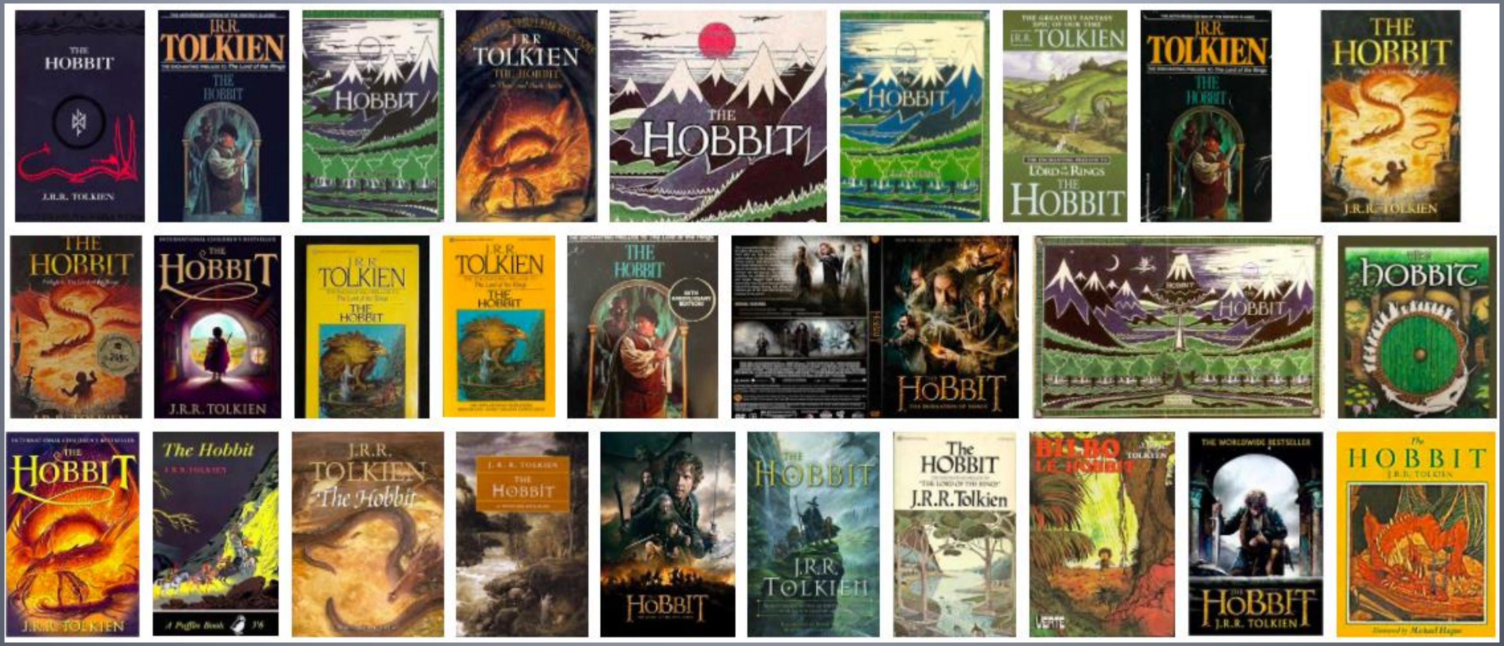
Other Scheduling Rules

- If there is request to commit T
  - Find all elements written by T and set each $C(X_i)$ to true
    - A list of such elements should be maintained by the scheduler
  - If any transactions are waiting for X_i to commit, those transactions can proceed
- If there is a request to abort T, or T is rolled back
  - Any transaction waiting for an element written by T repeats its attempt to read or write the element

Timestamps Versus Locks

- Timestamps are superior to locks where most transactions are read-only
  - Or when it is rare for concurrent transactions to read and write the same element
- Locking performs better when there are many conflicts
  - Locking delays transactions
  - But rollbacks will be more frequent, leading to even more delay

Multiversion Concurrency Control



Multiversion Concurrency Control

- Multiversion concurrency conctrol (MVCC) is another concurrency control technique
  - Where several versions of data items are maintained
- Allowing transactions to read the appropriate version of an item that has been modified
  - Where the read would be rejected in other concurrency control systems
- The obvious drawback with MVCC is that it requires additional storage

Timestamp MVCC

- For each version of a data item
  - Record the value and
  - The read timestamp (RT) – the largest timestamp of transactions that have read the item
  - The write timestamp (WT) – the timestamp of the transaction that wrote the version
- When a data item is written a new version is created
  - With RT and WT set to the timestamp of the transaction
  - If a transaction reads the item RT is set to the larger of its current value and the transactions timestamp

Timestamp MVCC Rules

- If transaction *T writes* data item *X*
  - If the highest WT(*X*) <= TS(*T*) and RT(*X*) > TS (*T*)
    - Abort and roll back *T*

      Another transaction read *X* after *T*
  - Otherwise create a new version of *X*
    - Where RT(*X*) = WT(*X*) = TS(*T*)
- If transaction *T reads* data item *X*
  - Find the version of *X* with highest WT(*X*) <= TS(*T*)
    - Return value of *X* to *T* and set the value of RT(*X*) to the greater of its current value and TS(*T*)
  - Note that reads are always successful

Multiversion 2PL

- Multiversion two-phase locking allows for increased concurrency
  - It allows reads of a data item to continue while a single transaction has a write lock on the item
    - By allowing two versions of data items, a committed version and a local version
- The technique adds a *certify* lock mode
  - Write locks must be upgraded to certify locks when a write is ready to commit

SQL Locking

Concurrency Control and SQL

- SQL allows programmers to specify three characteristics of transactions
  - **Access mode**
  - **Diagnostics size** – determines the number of error conditions that can be recorded
  - **Isolation level** – affects the level of concurrency
- The access mode can be either
  - READ ONLY – transaction is not allowed to modify the DB
    - Increases concurrency as only shared locks are required
  - READ WRITE – this mode is required for INSERT, DELETE, UPDATE, or CREATE commands

Isolation Levels

- SERIALIZABLE is the highest degree of isolation
  - Obtains locks on sets of objects (index locking), and
  - Obtains and holds locks according to Strict 2PL
- REPEATABLE READ is similar to SERIALIZABLE
  - Obtains and holds locks according to Strict 2PL, but
  - Does not lock sets of objects
- READ COMMITTED
  - Obtains *X* locks before writing and holds until committed
  - Obtains *S* locks before reading, but releases them immediately
- READ UNCOMMITTED does not obtain any locks
  - And is required to be READ ONLY

Isolation Level Summary

Level	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	Possible	Possible	Possible
Read Committed	No	Possible	Possible
Repeatable Read	No	No	Possible
Serializable	No	No	No

Deadlocks

Deadlocks

- A *deadlock* occurs when two transactions require access to data objects locked by each other
  - e.g. T1 has locked A and requires B, and T2 requires B, and has locked A
    - Both transactions must wait for the other to unlock so neither transaction can proceed, and to make matters worse
    - They may hold locks required by other transactions
- Deadlocks must be either *detected and avoided* or *resolved*
  - A simple method for identifying deadlocks is to use a timeout mechanism

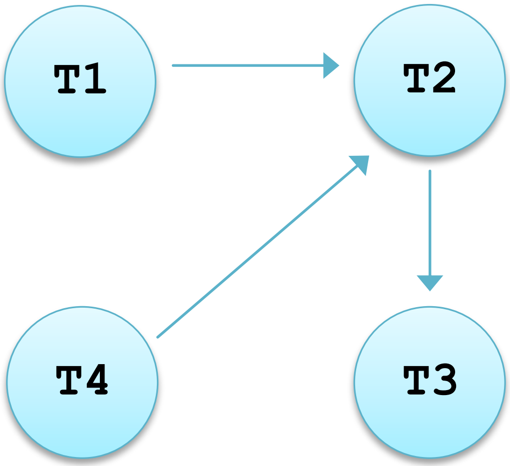
Deadlock Detection

- In practice deadlocks are rare and usually only involve a few transactions
- The lock manager maintains and periodically checks a *waits-for graph* to detect deadlocks
  - The nodes correspond to active transactions
  - An arc from T1 to T2 represents that T1 is waiting for T2 to release a lock
- A waits-for graph can be used to detect cycles, which indicate deadlocks
  - Deadlocks are resolved by aborting one of the transactions

Waits-For Graph

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)

waits-for graph for schedule

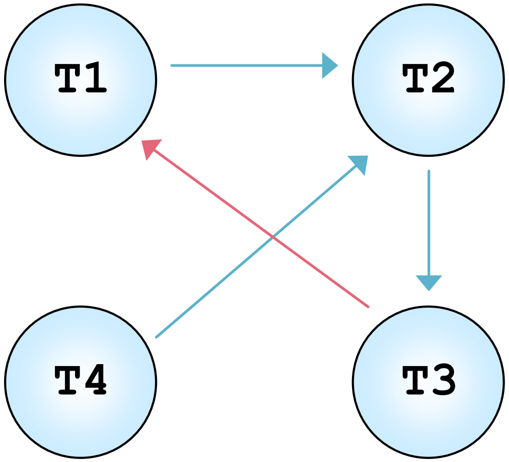


no deadlock

Waits-For Graph – Deadlock

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

waits-for graph for schedule



deadlock

Dealing with Deadlocks

- A deadlock is resolved by aborting one of the transactions
- Several criteria can be considered when choosing the transaction to be aborted
  - The transaction with the fewest locks
  - The transaction that has performed the least work to date
  - The one that is furthest from completion
  - ...
- Transactions may be repeatedly aborted
  - If so, at some point, they should be given precedence and allowed to complete

Deadlock Prevention

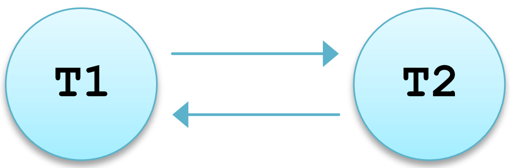
- Deadlocks can be prevented by not allowing transactions to wait
  - Each transaction is given a priority
    - The transactions timestamp can be used as the priority, the lower the timestamp, the higher the priority
  - Lower priority transactions are not allowed to wait for higher priority transactions
- When a transaction requests a lock which is already held one of two policies can be used
  - Wait-die
  - Wound-wait

Wait-Die Policy

- A transaction with a higher priority than an existing and conflicting transaction is allowed to wait
- A transaction with a lower priority dies
- Assume that T_1 has requested a lock and that T_2 holds a conflicting lock
  - If T_1 has the higher priority, it waits, otherwise it is aborted
  - For a deadlock to occur T_1 must be waiting for a lock held by T_2 while T_2 is waiting for a (different) lock held by T_1
  - But T_2, waiting for T_1, must have a lower priority so T_2 dies and the deadlock is prevented
    - In general more transactions could be involved

Simple Wait-Die Example

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



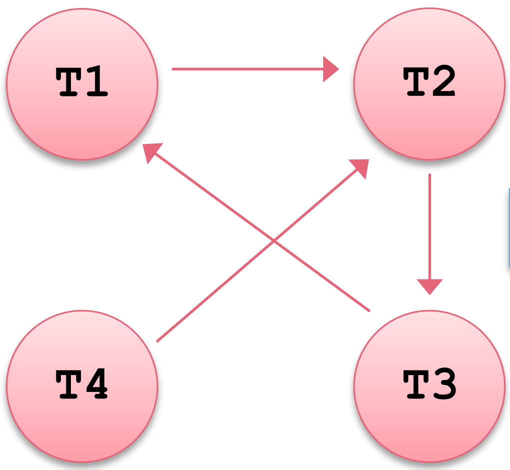
T1 requests a lock on B, if it has the higher priority, then T2 is aborted, allowing T1 to proceed

Otherwise T2 must have the higher priority so T1 is aborted, and T2 proceeds

Wound-Wait Policy

- Assume that T1 has requested a lock and that T2 holds a conflicting lock
- If T1 has the higher priority, abort T2, otherwise wait
  - *Wounding* refers to the process of aborting a transaction
  - If the wounded transaction is already releasing its locks when the wound takes effect it is allowed to complete
- How does this prevent deadlock?
  - If T1 has a lower priority, it waits, however, if T2 is waiting for T1 it must have a higher priority so T1 is aborted

Wait-Die Example



Priority is T1, T2, T3, T4

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)[1]			
		S(C)	
		R(C)	
	X(C)[2]		
			X(B)[3]
		X(A)[4]	

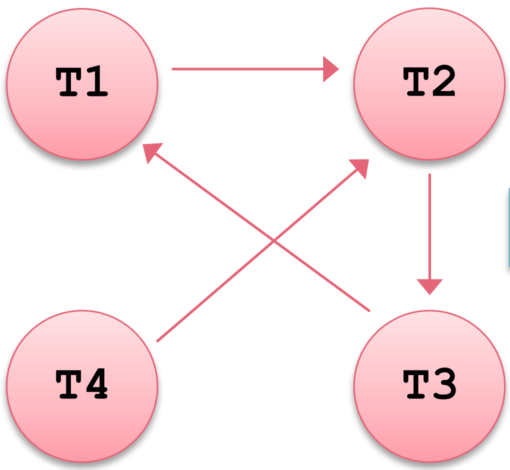
1 - T1 requests a lock that conflicts with T2 so **waits**

2 - T2 requests a lock that conflicts with T3 so **waits**

3 - T4 requests a lock that conflicts with T2 so **dies** but the conflict is not resolved

4 - T3 requests a lock that conflicts with T1 so **dies** resolving the conflict

Wound-Wait Example



Priority is T1, T2, T3, T4

1 - T1 requests a lock that conflicts with T2 so **wounds** T2, aborting it and resolving the conflict

2 - T1 commits, releasing its locks

3 - T3 proceeds without conflict

4 - T4 proceeds without conflict

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)[1]			
		S(C)	
		R(C)	
	X(C)[2]		
			X(B)[3]
		X(A)[4]	

Wound-Wait vs. Wait-Die

- Wait-die is non-preemptive
  - Only transactions that request locks are aborted
- In contrast, wound-wait is preemptive
  - A transaction may abort a second transaction that has all the locks that it needs

Locking Performance

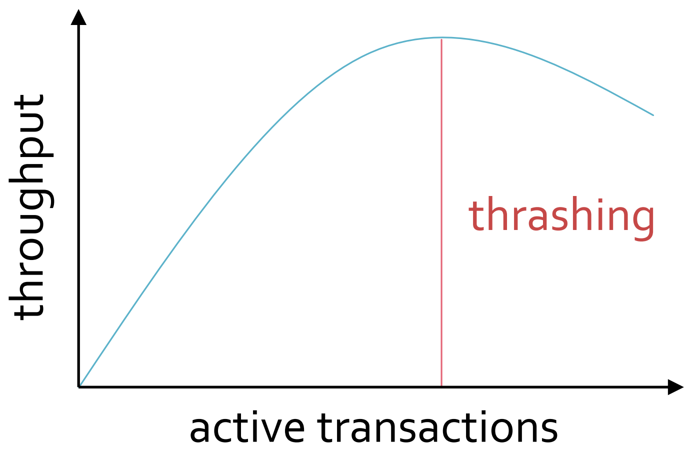
- Locking schemes use two basic mechanisms
  - Blocking, and
  - Aborting
- Blocked transactions may hold locks that force other transactions to wait
  - A deadlock is an extreme instance of blocking where a set of transactions is blocked forever
- Aborting a transaction wastes the work performed by the transaction before being aborted
- In practice, there are usually few deadlocks
  - The cost of locking comes primarily from blocking

Blocking and Throughput

- Delays due to blocking increase with the number of active transactions
  - As more transactions execute concurrently the probability that they block each other increases
- Throughput therefore increases more slowly than the increase in the number of transactions
  - At some point adding another transaction actually reduces throughput
    - The new transaction is blocked, and competes with existing transactions
  - This is referred to as *thrashing*

Thrashing

- When thrashing occurs the number of transactions allowed to run concurrently should be reduced
- Thrashing usually occurs when 30% of active transactions are blocked
- The percentage of blocked transactions should be monitored



Increasing Throughput

- Always lock the smallest sized database object
  - e.g. a set of rows, rather than an entire table
  - This reduces the chance that two transactions need the same lock
- Reduce the time that transactions hold locks
- Reduce *hot spots*
  - A hot spot is a DB object that requires frequent access (and modification)

The End

Summary

- Lock-based locking schemes adopt a pessimistic approach to concurrency control
  - However they are very effective and have low overhead
- Index locking for B+ trees can be much more efficient than predicate locking for data pages.
- In real-life DBMS systems
  - Transaction dependency is rare
  - Users are allowed to balance the demands of performance and serializability
  - Transactions with different isolation levels may run concurrently inside a DBMS

Appendix

Validation

- A transaction that validates is treated as if it executed at the moment of validation
  - Each transaction has a *read set* RS(T) and *write set* WS(T)
- Transactions have three phases
  - *Read* – the transaction reads all of its elements in its read set
  - *Validate* – the transaction is validated by comparing its read and write set to other transactions' sets
    - If validation fails the transaction is rolled back
  - *Write* – if there is no conflict ,changes are written

Scheduler Data

- The scheduler maintains three *sets* of data
  - **Start** – transactions that have started but not finished
    - Records the start time, START(T), for each transaction
  - **Val** – transactions that have been validated but not finished writing
    - Records START(T) and validation time, VAL(T), for each transaction
  - **Finish** – transactions that have completed writing
    - Records START(T), VAL(T) and FIN(T) for each transaction
    - Transactions with FIN(T_1) less than START(T_2) are removed

Problems

- There are two situations in which a write by a transaction T_1 could be physically unrealizable
  - T_2 writes to a data object after it was read by T_1
    - T_2 has validated (T_2 is in VAL or FIN)
    - $FIN(T_2) > START(T_1)$ – T_2 did not finish before T_1 started
    - $RS(T_1) \cap WS(T_2)$ is non empty
  - T_1 and T_2 write to a data object in the wrong order
    - T_2 is in VAL
    - $FIN(T_2) > VAL(T_1)$ – T_2 did not finish before T1 entered validation
    - $WS(T_1) \cap WS(T_2)$ is non empty

Validation Rules

- Check that $RS(T_1) \cap WS(T_2) = 0$
  - For any validated T_2 where $FIN(T_2) > START(T_1)$
  - If not, then rollback T_1
- Check that $WS(T_1) \cap WS(T_2) = 0$
  - For any validated T_2 where $FIN(T_2) > VAL(T_1)$
  - If not, then rollback T_1