Concurrency Control: Outline

- Two-Phase Locking
  - Basics and Motivation
  - Lock conversion
  - Strict two-phase protocol
- Graph-Based Locking
- Deadlock Handling
  - Deadlock Detection
  - Deadlock Prevention
- Timestamp-Based Protocol
- Lock Tuning

Concurrency Control

- Two-Phase Locking
  - Basics and Motivation
  - Lock conversion
  - Strict two-phase protocol
- Graph-Based Locking
  - Simple Tree Protocol
- Deadlock Handling
- Timestamp-Based Protocol
- Lock Tuning

A Locking Protocol

- This protocol ensures only serializable schedules by delaying transactions that could violate serializability.
- Two types of *locks* can be set on an item Q:
  - *Exclusive lock*: LX Q
  - *Shared lock*: LS Q
- Locks can be released.
  - *Unlock*: UN Q
- Privileges associated with locks
  - A transaction holding an X-lock may issue a write or read access request on the item.
  - A transaction holding as S-lock may issue a read access request on the item.

Compatibility Matrix

- This matrix is interpreted as follows: a transaction may set a lock on an item if this lock is compatible with locks already held on the item by other transactions.

		New X	*New* S
Old	X	False	False
Old	S	False	True

- It follows that
  - Any number of transactions can hold S-locks on an item.
  - If any transaction holds an X-lock on the item, no other transaction may hold any lock on the item.

Lock Granting

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- A transaction may be granted a lock on an item if the requested lock is *compatible* with locks already held on the item by other transactions
- If a lock cannot be granted, the requesting transaction is made to wait till all *incompatible* locks held by other transactions have been released. The lock is then granted.

Problems with Locking

- Transaction *T15* transfers \$50 from account B to account A.
- Transaction *T16* displays the total amount of money in accounts A and B.

T15:		*T16:*	
	LX B		LS A
	read B		read A
	$B \leftarrow B - 50$		UN A
	write B		LS B
	UN B		read B
	LX A		UN B
	read A		display A+B
	$A \leftarrow A + 50$		
	write A		
	UN A		

- Initially A = 100 and B = 200.

Problems with Early Unlocking

Schedule S7	
T15	*T16*
LX B	
read B	
B ← B-50	
write B	
UN B	
	LS A
	read A
	UN A
	LS B
	read B
	UN B
	display A +B
LX A	
read A	
A ← A +50	
write A	
UN A	

- Early unlocking can cause incorrect results (non-serializable schedules).

Problems with Late Unlocking

- So let's delay unlocking until the end of the transaction.

Schedule S8	
T17	*T18*
LX B	
read B	
B $\leftarrow$ B-50	
write B	
	LS A
	read A
	LS B
LX A	

- Late unlocking can cause *deadlocks*.

The Two-Phase Locking Protocol

- First phase (*growing phase*):
  - Transaction may request locks.
  - Transaction may not release locks.
- Second phase (*shrinking phase*):
  - Transaction may not request locks.
  - Transaction may release locks.

- When the first lock is released, the transaction moves from the first phase to the second phase.

Precedes Relation for Locks

- A locking protocol ensures serializability if and only if for all legal schedules under the protocol, the *precedes relation* is acyclic.
- *Ti precedes Tj* (*Ti* →*Tj*) for schedule *S* if
  - *i* does not equal *j*.
  - *Ti* read/write locks Q, *Tj* is the next transaction to write lock Q.
  - *Ti* write locks Q. *Tj* is the next transaction that read locks Q (after *Ti* unlocks Q but before any other transaction write locks Q).
- To discover that a (locking) schedule is serializable
  - Construct a conflict graph using the definition above.
  - Check acyclicity.
- Topological sort gives a serialization order.

Two-Phase Locking Works

- **Theorem** If *S* is any schedule of two-phase transactions, then *S* is serializable.
- **Proof**
  - Suppose not. Then conflict graph must have a cycle.

    $$Ti \rightarrow Tj \rightarrow \dots \rightarrow Tm \rightarrow Ti$$
  - A lock by *Tj* follows an unlock by *Ti*, and a lock by *Ti* follows an unlock by *Tm*. By transitivity, a lock by *Ti* follows an unlock by *Ti*.
  - Contradiction.
    - Not consistent with the two-phase locking protocol definition.

Generality of Two-Phase Locking

- In general, a locking protocol does not allow all serializable schedules. Some serializable schedules are illegal.
- *S9* is an illegal schedule under the two-phase protocol.

Schedule S9		
T19	*T20*	*T21*
	LX A	
	UN A	
		LX A
		UN A
LX B		
UN B		
	LX B	
	UN B	

- Serialization order: $T19 \rightarrow T20 \rightarrow T21$

Two-Phase With Lock Conversion

- First Phase:
  - Can acquire an S-lock on item.
  - Can acquire an X-lock on item.
  - Can *convert* (*upgrade*) an S-lock to an X-lock.
- Second Phase:
  - Can release an S-lock.
  - Can release an X-lock.
  - Can *convert* (*downgrade*) an X-lock to an S-lock.

- This protocol assures serializability.
- It relies on the application programmer to insert the appropriate locks.

Extended Conflict Graph

- *Ti* precedes *Tj* (*Ti* →*Tj*) if there exists a data-item Q such that
  - *Ti* has held lock-mode A on Q.
  - *Tj* has held lock-mode B on Q.
  - *Tj*'s lock is later than *Ti*'s lock.
  - COMP (A,B) = false. (COMP is the compatibility matrix.)
- *Ti* must occur before *Tj* in a serial schedule.

Practical Protocol

- The programmer (transaction) issues the standard read/write instructions.
- The system manages all aspects of the protocol, including the lock operations.
-

```
read D:
    if T has a lock on D
            perform read
    else
            wait until no other transaction has a X-lock on D
            grant T an S-lock on D
            perform read
```

Practical Protocol, cont.

write D:

    if T has a X-lock on D

        perform write

    else

        wait until no other transaction has a lock on D

        if T has an S-lock on D

            convert it to X-lock

        else

            grant it X-lock on D

        perform write

- All locks are released after commit or abort (Rollback)

Cascading Rollbacks

- One transaction aborting can cause other transactions to abort.

Schedule S11

T22	*T23*	*T24*
LX A		
LX B		
UN A		
	LX A	
	UN A	
		LX A

- *T22* aborts ⇒ we have to rollback *T23* and *T24*.
- How to eliminate these *cascading rollbacks*?
  - Don't let transactions read “dirty” uncommitted data.

Strict Two-Phase Locking

- In **strict two-phase locking**, *x-locks* are not released until after commit time.
- In **rigorous two-phase locking**, *all locks* are release after commit time.

- Hence, no cascading rollbacks.
- Loss of potential concurrency.

Concurrency Control

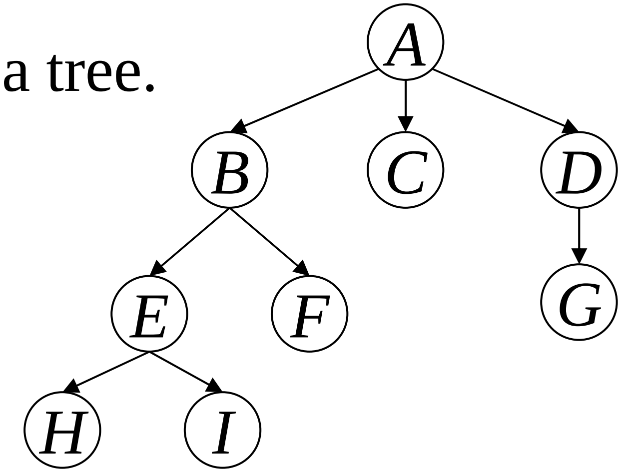
- Two-Phase Locking
- Graph-Based Locking
  - Simple Tree Protocol
- Deadlock Handling
- Timestamp-Based Protocol
- Lock Tuning

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Simple Tree Protocol

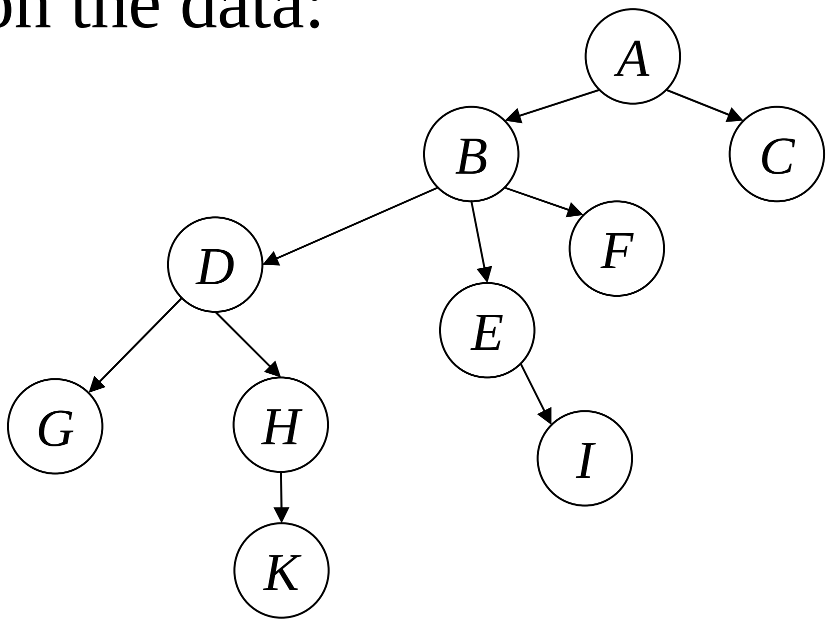
- Simple Tree Protocol
  - Partial order on DB items determines a tree.
  - Only exclusive locks are allowed.



- Rules
  - Each *Ti* can lock a data item at most once.
  - First lock on any data item.
  - Subsequently, *Ti* can lock item Q only if it currently holds a lock on the parent of Q.
  - Data items can be unlocked any time.

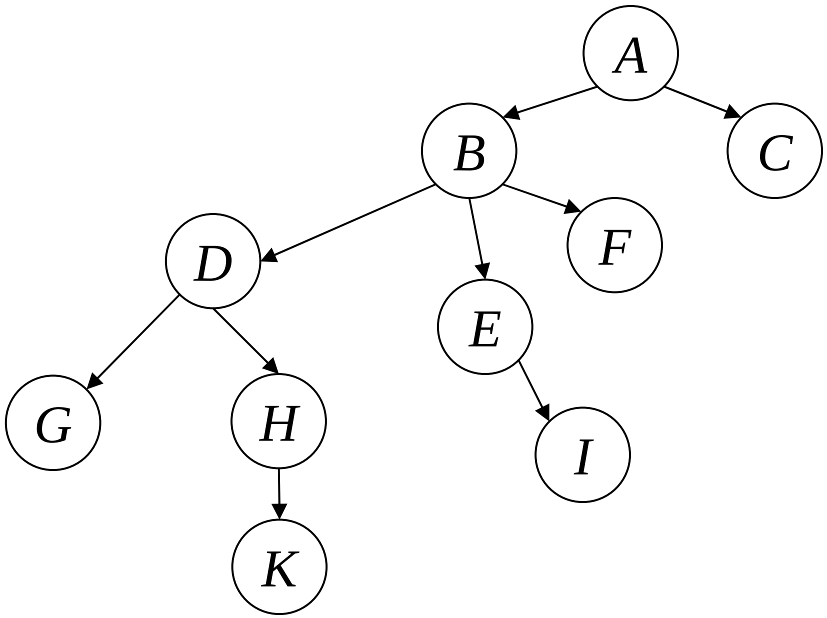
Simple Tree Protocol Example

- Partial order on the data:



- **LX** B; **LX** E; **UN** E; **LX** D; **UN** B; **LX** G; **UN** D; **UN** G.
- **LX** D; **LX** H; **UN** D; **LX** K; **UN** K; **UN** H.
- **LX** B; **LX** E; **UN** E; **UN** B.
- **LX** D; **LX** H; **UN** D; **UN** H.

Simple Tree Protocol Example, cont.



Schedule S12			
T25	T26	T27	T28
LX B			
	LX D		
	LX H		
	UN D		
LX E			
UN E			
LX D			
UN B			
		LX B	
		LX E	
	LX K		
	UN K		
	UN H		
LX G			
UN D			
			LX D
			LX H
			UN D
			UN H
		UN E	
		UN B	
UN G			

Concurrency Control

- Two-Phase Locking
- Graph-Based Locking
- Deadlock Handling
  - Deadlock Detection
  - Deadlock Prevention
- Timestamp-Based Protocol
- Lock Tuning

Deadlocks

- "Easy" solutions:
  - Newer wait and do (partial) rollback, may lead to *livelock*.
    - The states of the transactions involved constantly change with regard to one another, none progressing.
  - Linearly order all resources (preventions technique).

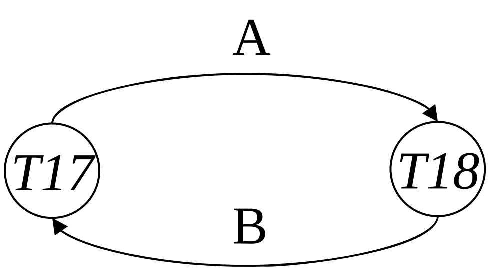
- Alternatives for deadlock handling
  - Detect deadlock and resolve
  - Prevent
  - Timeout
    - All systems must ultimately depend on timeout to detect some deadlocks.

Deadlock Detection

- Create a “wait for” graph and check for cycles.
  - An edge from *T1* to *T2* if *T1* waits for *T2*
- Invoke detection algorithm repeatedly.
- If deadlock is detected,
  - Select appropriate victim.
  - Abort victim and release its locks.

Schedule S8

T17	*T18*
LX B	
read B	
B ← B-50	
write B	
	LS A
	read A
	LS B
LX A	



Deadlock Detection Example

T1	*T2*	*T3*	*T4*
	LX A		
LX B			
LX C			
		LX B	
LX D			
			LX E
	LX D		
			LX A
LX E			

Exercise:

Draw out the “wait for” graph, and detect if deadlock exits.

Hint:

List in a table all resources, locks on each, and requests on each.

Deadlock Prevention Protocols

- Give each transaction a timestamp (separate from any timestamp used for concurrency control) that is retained even across rollbacks.
- Assume *Ti* requests an item held by *Tj*.

- *Wait-Die*
  - Wait: If *Ti* is older than *Tj*, then *Ti* waits.
  - Die: If *Ti* is younger than *Tj*, then abort *Ti*.

- *Wound-Wait*
  - Wound: If *Ti* is older than *Tj*, then abort *Tj*.
  - Wait: If *Ti* is younger than *Tj*, then *Ti* waits.

- No starvation in either scheme, because transactions advance.

Wait-Die Examples

T10 is older than T20.

T10	T20
LX A	
	LX A
	ROLLBACK
	LX A
	ROLLBACK
	LX A
	ROLLBACK
UN A	
	LX A
	...
	UN A

T10 starts first.

T10	T20
	LX A
LX A	
WAIT	
WAIT	
WAIT	
WAIT	
WAIT	
	UN A
LX A	
...	
UN A	

T20 starts first.

Wound-Wait Examples

T10 is older than T20.

T10	*T20*
LX A	
	LX A
	WAIT
	WAIT
	WAIT
	WAIT
	WAIT
UN A	
	LX A
	...
	UN A

T10 starts first.

T10	*T20*
	LX A
LX A	
	ROLLBACK
	LX A
	WAIT
	WAIT
	WAIT
UN A	
	LX A
	...
	UN A

T20 starts first.

Concurrency Control

- Two-Phase Locking
- Graph-Based Locking
- Deadlock Handling
- Timestamp-Based Protocol
- Lock Tuning

Timestamp Protocol

- Each transaction is issued a timestamp when it enters the system.
- Timestamps are drawn from an increasing sequence of integers.
- The protocol will manage the concurrent execution so that it will be equivalent to a serial execution in timestamp order.
- If two transactions conflict in the schedule, then the protocol will ensure that the one with the lower timestamp accesses the item first.

The Timestamp Protocol, cont.

- In order to assure such behavior the protocol maintains for each item Q two integers, used for synchronization.
  - *r-max(Q)*: the maximal timestamp of a transaction that read that item.
  - *w-max(Q)*: the maximal timestamp of a transaction that wrote the item.
- If a transaction wants to access an item in a way that will not create a prohibited conflict, the access is allowed and a synchronization value is updated.
- If the conflict is prohibited, the transaction is aborted, and restarted with a *new (later) timestamp*.

Implementing the Timestamp Protocol

- Let t be the timestamp of the executing transaction.

read X: **if** $t \geq$ w-max(X)
  perform read
  r-max(X) $\leftarrow$ max(t, r-max(X))
else abort

write X: **if** $t \geq$ r-max(X) **and** $t \geq$ w-max(X)
  perform write
  w-max(X) $\leftarrow t$
else
  if $t <$ r-max(X)
    abort
  else
    do nothing

Timestamp Protocol Example

- We present a partial history for several items for transactions with timestamps 1, 2, 3, 4, and 5.

Schedule S13

T29	*T30*	*T31*	*T32*	*T33*
				read X
	read Y			
read Y				
		write Y		
		write Z		
				read Z
	read Z			
	abort			
read X				
			write Z	
			abort	
				write X
				write Z

Exercise:

Figure out r-max(Q) and w-max(Q) values after each step, for Q=X, Y, Z

Timestamp vs. Locking

- Schedule allowed by locks but not timestamps.

Schedule S14	
T34	*T35*
read A	
	read B
	write B
read B	

- Schedule allowed by timestamps but not by locks.

Schedule S15		
T36	*T37*	*T38*
write A		
	write A	
		write A
write B		
	write B	

Strict Timestamp Based Concurrency Control

- How to avoid cascading rollbacks?
  - Transactions should read only committed values.
- *Strict timestamp concurrency control protocol*
- *read X*: **if** $t \geq$ w-max(X)
  - r-max(X) $\leftarrow$ max(t, r-max(X))
  - wait for a committed value of X
  - perform read
  - **else** abort
- *write X*: **if** $t \geq$ r-max(X) **and** $t \geq$ w-max(X)
  - w-max (X) $\leftarrow t$
  - wait until X value is a committed value and pending reads are done
  - perform write
  - **else**
    - **if** $t <$ r-max(X) abort
    - **else** do nothing

Concurrency Control

- Two-Phase Locking
- Graph-Based Locking
- Deadlock Handling
- Timestamp-Based Protocol
- Lock Tuning

Lock Tuning: Various Techniques

- Eliminate locking when it is unnecessary.
  - Switch off locking when only one transaction is running, e.g., when loading a database.
  - Eliminate locking when all transactions are read-only (e.g., in Data Warehousing)
- Apply transaction chopping/reduce the length of transactions.
  - A long transaction must wait for locks and makes other transactions wait for locks.
  - Interactive transactions that involve human interaction, e.g., airline reservations, may be broken into separate transactions.
  - Chopping may affect correctness if one is not careful.
- Avoid DDL statements.
  - DDL statements update the system catalog, which all transactions must access.
  - DDL statements thus tend to slow down transactions.

Use Different Isolation Levels

- Consider weakening isolation (correctness) when the application permits it.
- Isolation levels:
  - **Degree 0**: Reads may see uncommitted ("dirty") data (yielding "dirty reads"). Reads are *non-repeatable*. Overwriting of dirty data is possible.
  - **Degree 1**: Dirty and non-repeatable reads, but no overwriting of dirty data.
  - **Degree 2**: Non-repeatable reads allowed, but no dirty reads. Using standard locking, read locks are released immediately; write locks are managed according to the two-phase protocol.
  - **Degree 3**: Reads are now also repeatable. ACID transactions result.
- Guideline:
  - Degree 3 is the default. If transactions are long and there is a lot of deadlocks and blocking, consider using Degree 2 or 1.

Select the Right Lock Granularity

- Three levels of locks are common:
  - record-level, page-level, and table-level locks

- Rule of thumb
  - If every transaction accesses well below 1% of the records of a table, and the records are on different pages, record-level locking is best.
    - This is the case in environments with many short transactions.
  - Medium-length transactions that use clustering indices should use page-level locking.
  - Long transactions should use table-level locking.

Hotspots

- Avoid hotspots.
  - Monitoring tools are used for locating hotspots.
  - Use partitioning.
  - Access hotspots late in a transaction.

- Partitioning example:
  - Suppose sequential keys create concurrency control bottlenecks.
    - Use many insertion points and insert randomly.
    - Cluster on an uncorrelated attribute.

Summary

- Many concurrency control protocols have been developed.
- There are several entire books on the topic.
- Most relational DBMS's use rigorous two-phase locking.
- Many refinements, such as page and record locking, are used.
- Tuning is important for good performance.