Database Management Systems 2010/11

– Chapter 7: Concurrency Control –

J. Gamper

- Lock-Based Protocols
- Graph-Based Protocols
- Timestamp-Based Protocols
- Multiple Granularity
- Multiversion Protocols
- Deadlock Handling

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Lock-Based Protocols

- One way to **ensure serializability** is to require that data items be accessed in a **mutually exclusive** manner
  - More precisely, while one transaction is accessing a data item, no other transaction can modify it.
- Lock is the most common mechanism to implement this requirement to control concurrent access to a data item.
- Data items can be locked in two modes:
  - **exclusive mode (X)**: Data item can be both read as well as written. X-lock is requested using **lock-X(A)** instruction.
  - **shared mode (S)**: Data item can only be read. S-lock is requested using **lock-S(A)** instruction.
- Locks can be released: **U-lock(A)**
- Lock requests are made to concurrency-control manager.
  - Transaction can proceed only after request is granted.

Lock-Based Protocols . . .

- **Locking protocol**: A set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols restrict the set of possible schedules.
  - Ensure serializable schedules by delaying transactions that might violate serializability.

Lock-Based Protocols . . .

- **Lock-compatibility matrix** tells whether two locks are compatible or not.
  - Any number of transactions can hold shared locks on a data item
  - If any transaction holds an exclusive lock on a data item no other transaction may hold any lock on that item.

		Lock 2	
		S	*X*
Lock 1	*S*	true	false
	X	false	false

- **Locking Rules/Protocol**
  - 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.

Pitfalls of Lock-Based Protocols

- Too early unlocking can lead to non-serializable schedules.
- Too late unlocking can lead to deadlocks.
- **Example:**
  - Transaction $T1$ transfers \$50 from account B to account A.
  - Transaction $T2$ displays the total amount of money in accounts A and B, that is, the sum of $A + B$.

Pitfalls of Lock-Based Protocols . . .

- **Example** (contd.): Early unlocking can cause **non-serializable schedules**, and therefore potentially incorrect results.
  - e.g., A = \$100, B = \$200
  - ⇒ display A + B shows \$250
  - $<T1, T2>$ and $<T2, T1>$ display \$300

T1	T2
1. X-lock(B)	
2. read B	
3. B := B-50	
4. write B	
5. U-lock(B)	
6.	S-lock(A)
7.	read A
8.	U-lock(A)
9.	S-lock(B)
10.	read B
11.	U-lock(B)
12.	display A + B
13. X-lock(A)	
14. read A	
15. A := A+50	
16. write A	
17. U-lock(A)	

Pitfalls of Lock-Based Protocols . . .

- **Example** (contd.): Late unlocking can lead to **deadlocks**
  - Neither T_1 nor T_2 can make progress:
    - executing *lock-S*(B) causes T_2 to wait for T_1 to release its lock on B.
    - executing *lock-X*(A) causes T_1 to wait for T_2 to release its lock on A.
- To handle a deadlock one of T_1 or T_2 must be rolled back and its locks released.

T1	T2
1. X-lock(B)	
2. read B	
3. B := B-50	
4. write B	
5.	S-lock(A)
6.	read (A)
7.	S-lock(B)
8. X-lock(A)	

Two-Phase Locking Protocol

- **Two-Phase Locking Protocol**: A locking protocol that ensures conflict-serializable schedules. It works in two phases:
    - **Phase 1: Growing Phase**
        - transaction may obtain locks
        - transaction may not release locks
    - **Phase 2: Shrinking Phase**
        - transaction may release locks
        - transaction may not obtain locks
- **Lock point**: Transition point form phase 1 into phase 2, i.e., when the first lock is released.

Two-Phase Locking Protocol ...

- **Example**: Schedule with locking instructions following the Two-Phase Locking Protocol

T1	T2
1. X-lock(B)	
2. read B	
3. B := B-50	
4. write B	
5. X-lock(A)	
6. U-lock(B)	
7.	S-lock(B)
8.	read(B)
9. read(A)	
10. A := A+50	
11. write(A)	
12. unlock(A)	
13.	S-lock(A)
14.	read(A)
15.	display A + B
16.	unlock(B)
17.	unlock(A)

Two-Phase Locking Protocol . . .

- **Properties** of the Two-Phase Locking Protocol
  - Ensures serializability
    - It can be shown that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).
  - Does not ensure freedom from deadlocks
  - Cascading roll-back is possible

- Modifications of the two-phase locking protocol
  - **Strict two-phase locking**
    - A transaction must hold **all its exclusive locks** till it commits/aborts
    - Avoids cascading roll-back
  - **Rigorous two-phase locking**
    - **All locks** are held till commit/abort.
    - Transactions can be serialized in the order in which they commit.

Two-Phase Locking Protocol . . .

- Refinement the two-phase locking protocol with **lock conversions**
  - Phase 1:
    - can acquire a **lock-S** on item
    - can acquire a **lock-X** on item
    - can convert a **lock-S to a lock-X (upgrade)**
  - Phase 2:
    - can release a **lock-S**
    - can release a **lock-X**
    - can convert a **lock-X to a lock-S (downgrade)**
- Ensures serializability
- Strict and rigorous two-phase locking (with lock conversions) are used extensively in DBMS.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction without explicit locking calls (by the programmer).
- The operation *read*(D) is processed as follows:

```
if $T_i$ has a lock on $D$ then
    read($D$);
else
    If necessary wait until no other transaction has a lock-X on $D$;
    Grant $T_i$ a lock-S on $D$;
    read($D$);
end
```

Automatic Acquisition of Locks . . .

- The operation *write*(D) is processed as:

```
if $T_i$ has a lock-X on $D$ then
    write($D$);
else
    If necessary wait until no other transaction has any lock on $D$;
    if $T_i$ has a lock-S on $D$ then
        Upgrade lock on $D$ to lock-X;
    else
        Grant $T_i$ a lock-X on $D$;
    end
    write(D);
end
```

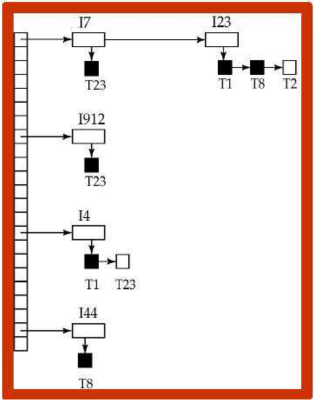
- All locks are released after commit or abort

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests.
- The lock manager replies to a lock request by sending a lock grant message (or a message asking the transaction to roll back, in case of a deadlock).
- The requesting transaction waits until its request is answered.
- The lock manager maintains a data structure called a **lock table** to record granted locks and pending requests.

Lock-Based Protocols . . .

- **Lock table**
  - Implemented as in-memory hash table indexed on the data item being locked
    - Black rectangles indicate granted locks.
    - White rectangles indicate waiting requests.
    - Records also the type of lock granted/requested.
  - Processing of requests:
    - New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
    - Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
    - If transaction aborts, all waiting or granted requests of the transaction are deleted.
      (Index on transaction to implement this efficiently.)



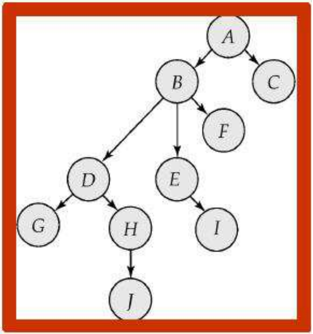
Graph-Based Protocols

- **Graph-based protocols**
  - Impose a partial order ($\rightarrow$) on the set $\mathbf{D} = \{d_1, d_2, ..., d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both di and dj must access d_i before accessing dj.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a database graph.

- Graph-based protocols are an alternative to two-phase locking and ensure conflict serializability.

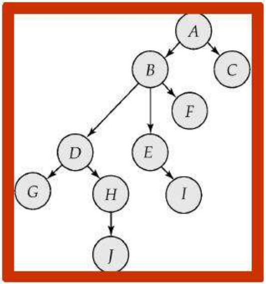
Graph-Based Protocols . . .

- **Tree-protocol**: A simple kind of graph-based protocol that works as follows:
    - Only exclusive locks **lock-X** are allowed.
    - The first lock by a transaction T_i may be on any data item.
    - Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
    - Data items may be unlocked at any time.
    - A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.



Graph-Based Protocols . . .

- **Example**: The following 4 transactions follow the treeprotocol on the database graph below.
  - *T10: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock- X(G); unlock(D); unlock(G);*
  - *T11: lock-X(D); lock-X(H); unlock(D); unlock(H);*
  - *T12: lock-X(B); lock-X(E); unlock(E); unlock(B);*
  - *T13: lock-X(D); lock-X(H); unlock(D); unlock(H);*



Graph-Based Protocols . . .

- **Example**: (contd.) One possible schedule

T10	T11	T12	T13
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Graph-Based Protocols . . .

- The tree protocol:
  - ensures conflict serializability;
  - ensures freedom from deadlock;
  - the abort of a transaction might lead to cascading rollbacks;
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times and increase in concurrency
- However, in the tree-protocol a transaction may have to lock data items that it does not access.
  - increased locking overhead and additional waiting time
  - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol and vice versa.

Timestamp-Based Protocols

- **Timestamp-based protocols**
  - Each transaction gets a timestamp when it enters the system.
  - If an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
  - The protocol manages concurrent execution such that the timestamps determine the serializability order as follows:
    - If $TS(T_i) < TS(T_j)$, the produced schedule is equivalent to the serial schedule $\langle Ti, Tj \rangle$

- The protocol maintains for each data item Q two timestamp values:
  - **W-timestamp(Q**) is the largest time-stamp of any transaction that executed **write(Q)** successfully
  - **R-timestamp(Q)** is the largest time-stamp of any transaction that executed **read(Q)** successfully.

Timestamp-Based Protocols ...

- **Timestamp ordering protocol**
  - Is a specific timestamp-based protocol that ensures that conflicting **read** and **write** operations are executed in timestamp order by imposing the following rules.
  - Transaction T_i issues a **read(Q)**:
    - If $TS(T_i) < W - timestamp(Q)$, then T_i needs to read a value of Q that was already overwritten. The **read** operation is rejected, and T_i is rolled back.
    - If $TS(T_i) \geq W - timestamp(Q)$, then the **read** operation is executed, and Rtimestamp(Q) is set to the maximum of R-timestamp(Q) and $TS(T_i)$.
  - Transaction T_i issues **write(Q)**:
    - If $TS(T_i) < R - timestamp(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. The **write** operation is rejected, and T_i is rolled back.
    - If $TS(T_i) < W - timestamp(Q)$, then T_i is attempting to write an obsolete value of Q. The **write** operation is rejected, and T_i is rolled back.
    - Otherwise, the **write** op. is executed, and W-timestamp(Q) is set to $TS(T_i)$.

Timestamp-Based Protocols . . .

- **Example**: The following schedule is possible under the timestamp ordering protocol.
  - Since we assume $TS(T_{14}) < TS(T_{15})$, the schedule must be conflict equivalent to the schedule $\langle T_{14}, T_{15} \rangle$

T_{14}	T_{15}
read(B)	
	read(B)
	B := B - 50
	write(B)
read(A)	
	read(A)
display(A+B)	
	A := A + 50
	write(A)
	display(A+B)

Timestamp-Based Protocols . . .

- **Properties** of the timestamp-ordering protocol
  - Guarantees conflict serializability, since conflicting operations are processed in timestamp order.
    - All arcs in the precedence graph are of the following form

    - Thus, there will be no cycles in the precedence graph
  - Ensures freedom from deadlock as no transaction ever waits.
  - The schedule may not be cascade-free and may not even be recoverable.

Timestamp-Based Protocols . . .

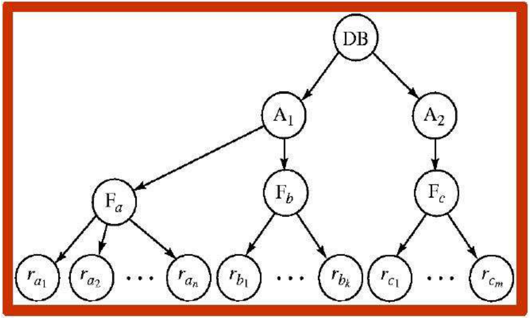
- **Recoverability and cascade freedom** in the timestamp-ordering protocol
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is **not recoverable**.
  - Further, any transaction that has read a data item written by T_j must abort, which might lead to a **cascading rollback**.
- **Solution:**
  - All writes are performed at the end of a transaction and they form an atomic action in the sense that while the writes are in progress no transaction may access any of the data items that have been written.
  - A transaction that aborts is restarted with a new timestamp.

Multiple Granularity

- Instead of locks on individual data items, sometimes it is advantageous to group several data items and to treat them as one individual synchronization unit (e.g. if a transaction accesses the entire DB).
- Define a **hierarchy of data granularities** of different size, where the small granularities are nested within larger ones.
  - Can be represented graphically as a tree
- When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendents in the same mode.

Multiple Granularity ...

- **Example**: Graphical representation of a hierarchy of granularities
  - The highest level is the entire database.
  - The levels below are of type area, file and record in that order.



- **Granularity of locking** (= level in tree where locking is done):
  - fine granularity (lower in tree): high concurrency, high locking overhead
  - coarse granularity (higher in tree): low locking overhead, low concurrency

Multiversion Protocols

- Concurrency control protocols studied thus far ensure serializability by either delaying an operation or aborting the transaction.
- **Multiversion protocols** keep old versions of data items to increase concurrency.
  - Each successful **write(Q)** creates a new version of Q.
    - Timestamps are used to label versions.
  - When a **read(Q)** operation is issued, select an appropriate version of Q based on the timestamp of the transaction.
  - **Read**s never have to wait as an appropriate version is available.
- Two types of multiversion protocols
  - Multiversion timestamp ordering
  - Multiversion two-phase locking

Multiversion Timestamp Ordering

- **Multiversion timestamp ordering protocol**
  - For each data item Q a sequence of versions $\langle Q_1, Q_2, ..., Q_m \rangle$ is maintained.
  - Each version Q_k contains 3 data fields:
    - **Content** – value of version Q_k.
    - **W-timestamp**(Q_k) – timestamp of the transaction that created (wrote) version Q_k
    - **R-timestamp**(Q_k) – largest timestamp of transaction that successfully read version Q_k
  - When a transaction T_i creates a new version Q_k of Q, the W-timestamp and R-timestamp of Q_k are initialized to $TS(T_i)$.
  - R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and $TS(T_j) > R\text{-}timestamp(Q_k)$.

Multiversion Timestamp Ordering . . .

- The following **multiversion timestamp-ordering protocol** ensures serializability.
  1. If transaction T_i issues a **read(Q)**, then the value returned is the content of version Q_k, which is the version of Q with the largest write timestamp less than or equal to $TS(T_i)$
  2. If transaction T_i issues a **write(Q)**:
     - If $TS(T_i) <$ R-timestamp(Q_k), then transaction T_i is rolled back
     - Otherwise, if $TS(T_i) =$ W-timestamp(Q_k), the contents of Q_k are overwritten.
     - Otherwise a new version of Q is created.

Multiversion Timestamp Ordering ...

- **Properties** of the multiversion timestamp-ordering protocol
  - **reads** always succeed and never have to wait
    - A transaction reads the most recent version that comes before it in time.
    - In a typical DBMS reading is a more frequent operation than writing, hence this advantage might be significant.
  - **write**: A transaction is aborted if it is “too late”in doing a write
    - i.e., a write by T_i is rejected if another transaction T_j that should read $T_i's$ write has already read a version created by a transaction older than T_i.
- **Disadvantages**
  - Reading of a data item also requires the updating of the R-timestamp, resulting in two disk accesses rather than one.
  - The conflicts between transactions are resolved through rollbacks rather than through waits.

Deadlock Handling

- Consider the following two transactions:

T_1: write (X)
write(Y)

T_2: write(Y)
write(X)

- Schedule with a deadlock

T_1	T_2
lock-X on X	
write (X)	
	lock-X on Y
	write (Y)
	wait for **lock-X** on X
wait for **lock-X** on Y	

Deadlock Handling . . .

- **Deadlock**: A system is in a deadlock state if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- A deadlock has to be resolved by rolling back some of the transactions involved in the deadlock.
- Deadlocks are addressed in two ways:
  - Deadlock prevention protocols are used
  - Deadlocks are detected and resolved

Deadlock Prevention Protocols

- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state.
- Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
    - Difficult to know in advance
    - Data-item utilization may be very low
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
    - Tree protocol
    - Data items have to be known in advance

Deadlock Prevention Protocols . . .

- Deadlock prevention protocols using **transaction timestamps**.
  - Wait-die scheme
  - Wound-wait scheme

- **Wait-die scheme** - non-preemptive technique
  - Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead (dies)
- **Example**: Transactions T_{22}, T_{23}, T_{24} with timestamps 5, 10, 15
  - T_{22} requests data item held by T_{23} : T_{22} will wait
  - T_{24} requests data item held by T_{23} : T_{24} will be rolled back.
- A transaction may die several times before acquiring the needed data item

Deadlock Prevention Protocols . . .

- **Wound-wait scheme** - preemptive technique
  - Older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones
- **Example**: Transactions T_{22}, T_{23}, T_{24} with timestamps 5, 10, 15
  - T_{22} requests data item held by T_{23} : T_{23} will be rolled back
  - T_{24} requests data item held by T_{23} : T_{24} will wait.
- May be fewer rollbacks than wait-die scheme.

- Both in *wait-die* and in *wound-wait* protocols, a rolled-back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Deadlock Prevention Protocols . . .

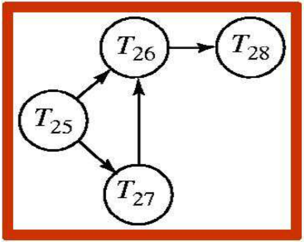
- **Timeout-based protocols**
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Thus, deadlocks are not possible.
  - Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection and Recovery

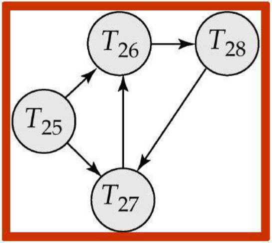
- Deadlocks can be described as a **wait-for graph**, which consists of a pair $G = (V, E)$
  - V is a set of vertices representing all the transactions
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, there is a directed edge from T_i to T_j, implying that T_i is **waiting** for T_j to release a data item.
- If T_i requests a data item being held by T_j, edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- A deadlock-detection algorithm must be invoked periodically to look for cycles.

Deadlock Detection and Recovery . . .

- Wait-for graph without a cycle



- Wait-for graph with a cycle



Deadlock Detection and Recovery . . .

- When a deadlock is detected, the system must **recover** from the deadlock.
- The most common solution is to roll back one or more transactions to break the deadlock. Three actions are required:
  1. **Selection of a victim**: Select that transaction(s) to roll back that will incur minimum cost.
  2. **Rollback**: Determine how far to roll back transaction
     - Total rollback: Abort the transaction and then restart it.
     - More effective to roll back transaction only as far as necessary to break deadlock.
  3. **Check Starvation**: happens if same transaction is always chosen as victim.
     - Include the number of rollbacks in the cost factor to avoid starvation