Introduction to Databases
《数据库引论》

Lecture 12: Concurrency Control
第12讲：并发控制

周水庚 / Shuigeng Zhou

邮件: sgzhou@fudan.edu.cn 网址：admis.fudan.edu.cn/sgzhou

复旦大学计算机科学技术学院

Outline of the Course

- **Part 0: Overview**
  - Lect. 1 (Feb. 29) - Ch1: Introduction
- **Part 1 Relational Databases**
  - Lect. 2 (Mar. 7) - Ch2: Relational model (data model, relational algebra)
  - Lect. 3 (Mar. 14) - Ch3: SQL (Introduction)
  - Lect. 4 (Mar. 21) – Ch4/5: Intermediate and Advanced SQL
- **Part 2 Database Design**
  - Lect. 5 (Mar. 28) - Ch6: Database design based on E-R model
  - **Apr. 4 (Tomb-Sweeping Day): no course**
  - Lect. 6 (Apr. 11/18) - Ch7: Relational database design
- **Midterm exam: Apr. 25**
  - 13: 00-15: 00, H3109
- **Part 3 Data Storage & Indexing**
  - Lect. 7 (May 2 -> Apr. 28) - Ch12/13: Storage systems & structures
  - Lect. 8 (May 10) - Ch14: Indexing and Hashing
- **Part 4 Query Processing & Optimization**
  - Lect. 9 (May 17) - Ch15: Query processing
  - Lect. 10 (May 24) - Ch16: Query optimization
- ☞ **Part 5 Transaction Management**
  - Lect. 11 (May 30) - Ch17: Transaction processing
  - Lect. 12 (May 30/Jun. 6) - Ch18: Concurrency control
  - Lect. 13 (Jun. 13) - Ch19: System recovery

Final exam: 13:00-15:00, Jun. 26

Outline

☞ Concurrent Control

- Lock-based Protocols
- Graph-based Protocols
- Multiple Granularity
- Deadlock Handling

Concurrent Control Problems

- Problems caused by concurrent transactions
  - Lost Update (丢失修改)
  - Non-repeatable Read (不可重复读)
  - Dirty Read (读“脏”数据)
- Symbols
  - R(x): read x
  - W(x): write x

Lost Update

- T_1 and T_2 read the same data item and modify it
- The committed result of T_2 eliminates the update of T_1

	T_1	T_2
①	R(A)=16	
②		R(A)=16
③	A←A-1 W(A)=15	
④		A←A-1 W(A)=15

Non-repeatable Read

	T_1	T_2
①	R(A)=50	
	R(B)=100	
	sum=150	
②		R(B)=100
		B←B*2
		W(B)=200
③	R(A)=50	
	R(B)=200	
	sum=250	
	(sum is not correct)	

- T_1 reads B=100
- T_2 reads B, then updates B=200, and writes back B
- T_1 reads B again, and B=200, not the same as the first read
- **Phantom Phenomenon** (幻影现象)
  - records disappear or new records appear for the same query

Dirty Read

- T_1 modifies C to 200, T_2 reads C as 200
- T_1 rolls back for some reason and its modification also rolls back. Then C recovers to 100
- T_2 reads C as 200, which is not consistent with the database

	T_1	T_2
①	R(C)=100	
	C←C*2	
	W(C)=200	
②		R(C)=200
③	ROLLBACK	
	C recover to 100	

Outline

- Concurrent Control
- ☞ **Lock-based Protocols**
- Graph-based Protocols
- Multiple Granularity
- Deadlock Handling

Lock-based Protocols

- A **lock** is a **mechanism to control concurrent access** to a data item
- Data items can be locked in **two modes**
  - **exclusive (X) mode (排他型)**. Data item can be read and written. **X-lock** is requested using **lock-X instruction**
  - **shared (S) mode (共享型)**. Data item can only be read. **S-lock** is requested using **lock-S instruction**
- Lock requests are made to **concurrency control manager (并发控制管理器)**. Transaction can proceed only after the request is granted.

Lock-based Protocols (Cont.)

- **Lock-compatibility matrix (锁相容性矩阵)**

	S	X
S	true	false
X	false	false

- **A transaction may be granted a lock on a data item** if the requested lock is compatible with locks already held on the data item by other transactions.
- If a lock cannot be granted, the requesting transaction **waits** till all incompatible locks have been released. The lock is then granted.

No Lost Update

T_1	T_2
① Xlock A	
② R(A)=16	
	Xlock A
③ A←A-1	wait
W(A)=15	wait
Commit	wait
Unlock A	wait
④	Get Xlock A
	R(A)=15
	A←A-1
⑤	W(A)=14
	Commit
	Unlock A

Repeatable Read

	T_1	T_2
①	Slock A	
	Slock B	
	R(A)=50	
	R(B)=100	
	sum=150	
②		Xlock B
		wait
		wait
③	R(A)=50	wait
	R(B)=100	wait
	sum=150	wait
	Commit	wait
	Unlock A	wait
	Unlock B	wait
④		get XlockB
		R(B)=100
		B←B*2
⑤		W(B)=200
		Commit
		Unlock B

No Dirty Read

	T_1	T_2
①	Xlock C	
	R(C)=100	
	C←C*2	
	W(C)=200	
②		Slock C
		wait
③	ROLLBACK	wait
	(C rec. 100)	wait
	Unlock C	wait
④		Get Slock C
		R(C)=100
⑤		Commit C
		Unlock C

Lock-based Protocols

```
T: lock-S(A);
   read (A);
   unlock(A);
   lock-S(B);
   read (B);
   unlock(B);
   display(A+B)
```

- Locking as above is not sufficient to guarantee serializability. If A and B get updated in-between the read of A and B, the displayed sum would be wrong
- **A locking protocol is a set of rules**
  - followed by all transactions while requesting and releasing locks
  - locking protocols restrict the set of possible schedules

Deadlock (死锁)

- Consider the following partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Such a situation is called a **deadlock**
  - To handle the deadlock, T_3 or T_4 must be rolled back and release its locks
  - Deadlock exists in most locking protocols

Starvation (饥饿)

- **Starvation**
  - E.g., a transaction may be waiting for an X-lock on a data item, while a sequence of other transactions request and are granted an S-lock on the same data item
  - **The same transaction is repeatedly rolled back due to deadlocks**
- Concurrency control manager can be designed to prevent starvation

Two-Phase Locking Protocol（两阶段加锁协议）

- 2PL is a protocol which ensures conflict-serializable schedules
  - Phase 1: Growing Phase (增长阶段)
    - transaction can obtain locks but cannot release locks
  - Phase 2: Shrinking Phase (缩减阶段)
    - transaction can release locks but cannot obtain locks
- **The protocol assures serializability**. It can be proved that the transactions can be **serialized** in the order of their **lock points** (封锁点)
  - Lock point: 事务获得最后加锁的位置

J. D. Ullman. Principles of Databse and Knowledge-base Systems. 1988

The Two-Phase Locking Protocol

- Satisfy 2PL

Slock A Slock B Xlock C Unlock B Unlock A Unlock C;

|← Growing →| |← Shrinking →|

- Not satisfy 2PL

Slock A Unlock A Slock B Xlock C Unlock C Unlock B;

The Two-Phase Locking Protocol

T_1	T_2
Slock(A)	
R(A=260)	
	Slock(C)
	R(C=300)
Xlock(A)	
W(A=160)	
	Xlock(C)
	W(C=250)
	Slock(A)
Slock(B)	wait
R(B=1000)	wait
Xlock(B)	wait
W(B=1100)	Wait
Unlock(A)	wait
	R(A=160)
	Xlock(A)
Unlock(B)	
	W(A=210)
	Unlock(C)
	Unlock(A)

2PL ensures serializable schedules

The Two-Phase Locking Protocol

- **Two-phase locking cannot avoid deadlocks**
- **Example:**

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

The Two-Phase Locking Protocol

- **Cascading roll-back** is possible under two-phase locking

T_5	T_6	T_7
lock-x (A)		
read (A)		
lock-s (B)		
read (B)		
write (A)		
unlock (A)		
	lock-x (A)	
	read (A)	
	write (A)	
	unlock (A)	
		lock-s (A)
		read (A)

  - To avoid this, follow a modified protocol called **strict two-phase locking (严格两阶段封锁)**
  - A transaction must **hold all its exclusive locks till it commits**

The Two-Phase Locking Protocol

- **Rigorous two-phase locking (强两阶段封锁)** is even stricter
  - **all locks are held till commit/abort**
  - transactions can be **serialized** in the order in which they commit

Lock Conversions (锁转换)

- Two-phase locking with lock conversions
  - Upgrade (升级)
    - lock-S -> lock-X
  - Downgrade (降级)
    - lock-X -> lock-S
- This protocol **assures serializability**

T8: read(a_1)
read(a_2)
…
read(a_n)
write(a_1)

T9: read(a_1)
read(a_2)
display(a_1+a_2)

T_8	T_9
lock-s (a_1)	
	lock-s (a_1)
lock-s (a_2)	
	lock-s (a_2)
lock-s (a_3)	
lock-s (a_4)	
	unlock-s (a_3)
	unlock-s (a_4)
lock-s (a_n)	
upgrade (a_1)	

Lock Conversions (锁转换)

- Two-phase locking with lock conversions
  - **First Phase:**
    - can acquire a **lock-S** on item
    - can acquire a **lock-X** on item
    - can convert a **lock-S** to a **lock-X** (**upgrade**)
  - **Second Phase:**
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol **assures serializability**

Automatic Acquisition of Locks

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

```
if Ti has a lock on D
then
   read(D)
else
   begin
      if necessary wait until no other transactions have a lock-X on D
         grant Ti a lock-S on D;
      read(D)
   end
```

Automatic Acquisition of Locks (Cont.)

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

```
if T_i has a lock-X on D
  then
     write(D)
  else
     begin
        if necessary wait until no other transactions have 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
           write(D)
     end;
```

- *All locks are released after commit*

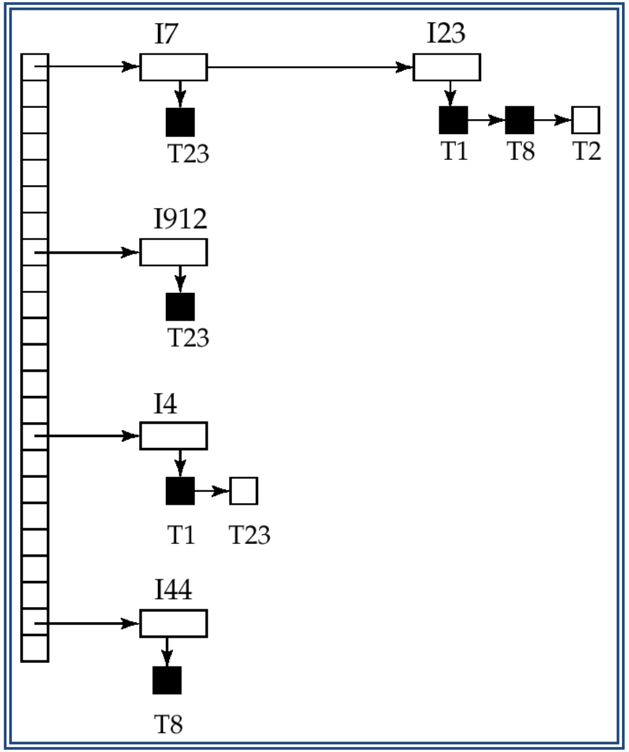
Implementation of Locking

- **Lock manager (锁管理器)**
  - 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 messages** (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 (锁表)** to record granted locks and pending requests

- **The lock table** is usually implemented as an **in-memory hash table indexed on** the name of the **data item being locked**

Lock Table

- **Black rectangles** indicate **granted locks**, and **white ones** indicate **waiting requests**
- Lock table also records the type of lock granted or requested
- 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 related locks being deleted, and waiting requests are checked to see if they can now be granted
- If transaction **aborts**, **all waiting or granted** requests of the transaction are deleted
  - **lock manager** may keep **a list of locks held by each transaction**, to implement this efficiently



Outline

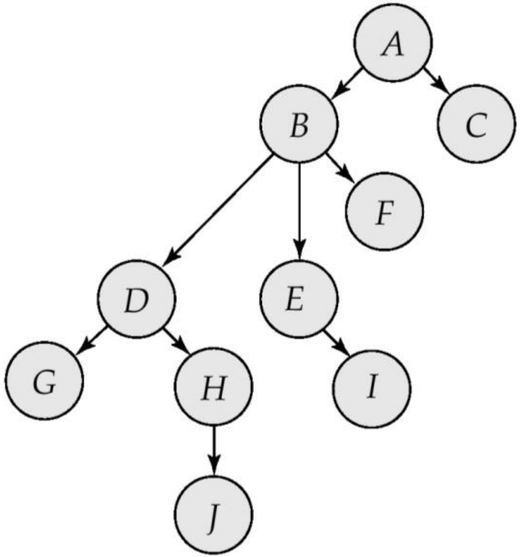
- Concurrent Control
- Lock-based Protocols
- ☞ **Graph-based Protocols**
- Multiple Granularity
- Deadlock Handling

Graph-based Protocols

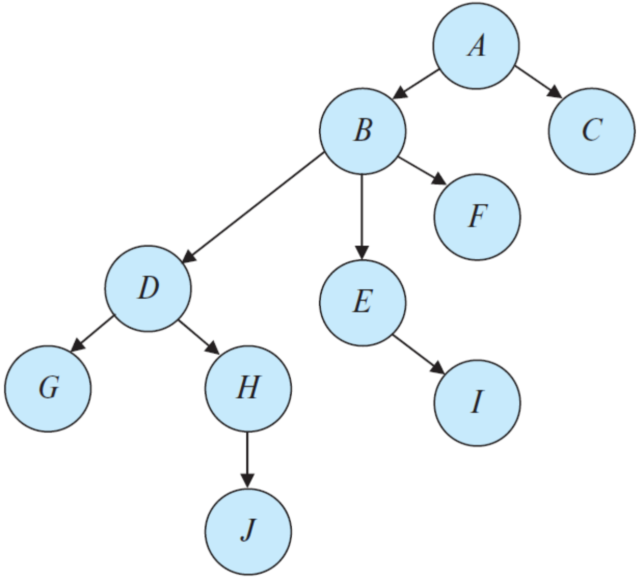
- **Graph-based protocols** are an alternative to two-phase locking
  - **Impose a partial ordering →(偏序)** on the set $D = \{d_1, d_2, \ldots, d_h\}$ of all data items
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j
  - Implies that the 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.

Tree Protocol

- **Only exclusive locks are allowed**
  - The **first lock** by T_i may be on **any data item**
  - Subsequently, a data 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 **cannot be relocked** by T_i



Graph-based Protocols



T_{10}	T_{11}	T_{12}	T_{13}
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)			

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

T_{11} -> T_{10} -> T_{12} -> T_{13}
T_{11} -> T_{10} -> T_{13} -> T_{12}

Graph-based Protocols

- **The tree protocol ensures conflict serializability as well as freedom from deadlock**
- Unlocking may occur **earlier** than in the two-phase locking protocol-2PL
  - shorter waiting times, and increase in **concurrency**
  - protocol is **deadlock-free**, no rollbacks are required
  - the abort of a transaction can still lead to **cascading rollbacks**
- However, may have to lock data items that it does not access
  - increased locking overhead, and additional waiting time
  - potential decrease in concurrency

Timestamp-based Protocols

- Each transaction is issued 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**
- In order to assure such behavior, the protocol maintains for each data **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 (Cont.)

- **The timestamp ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order
- Suppose **a 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) ≥ W-timestamp(Q)**, then the read operation is executed, and **R-timestamp(Q)** is set to **max(R-timestamp(Q), TS(T_i))**

Timestamp-based Protocols (Cont.)

- Suppose that **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.
    - Hence, 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**.
    - Hence, this write operation is **rejected**, and T_i is **rolled back**.
  - Otherwise, the write operation is executed, and **W-timestamp(Q)** is set to **TS(T_i).**

Timestamp-based Protocols (Cont.)

T_{25}: read(B);
read(A);
display($A + B$).

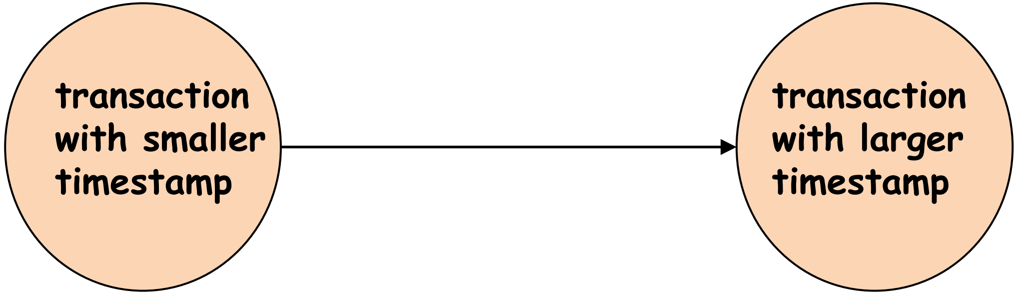
T_{26}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

T_{25} -> T_{26}

Timestamp-based Protocols (Cont.)

- **The timestamp-ordering protocol guarantees serializability** since all the arcs in the **precedence graph** are of the form:



  - Thus, there will be **no cycles in the precedence graph**
- Timestamp protocol ensures **freedom from deadlock** as no transaction ever waits
- But the schedule may **not** be **cascade-free**, and may **not** even be **recoverable**

Outline

- Concurrent Control
- Lock-based Protocols
- Graph-based Protocols
- ☞ **Multiple Granularity**
- Deadlock Handling

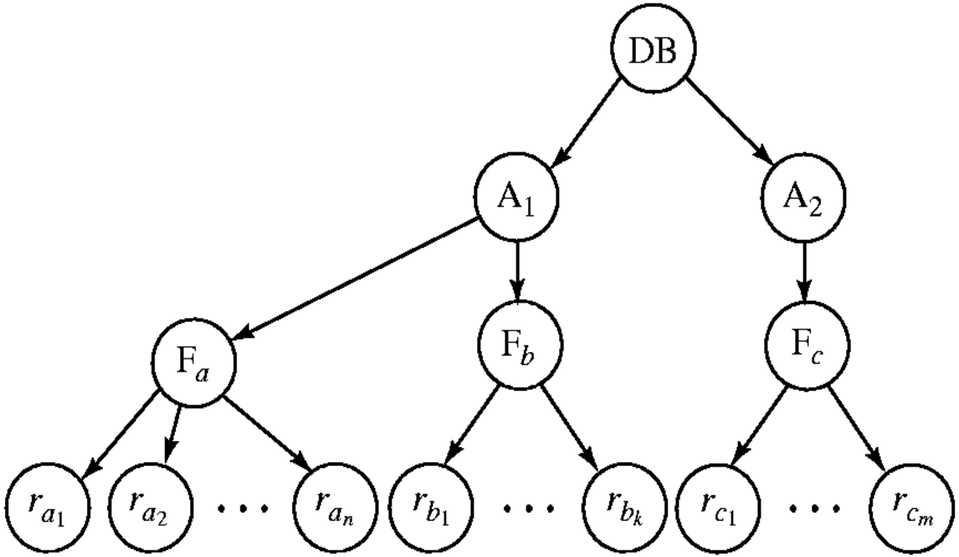
Multiple Granularity

- Allow **data items** to be of various sizes and define a hierarchy of data granularities
  - Database -> tables -> tuples -> attributes

- Can be represented graphically as a tree
- When a transaction **locks a node** in the tree explicitly, it implicitly **locks all the node's descendants** in the same mode
- **Granularity of locking:**
  - **fine** granularity (lower in tree): high concurrency, high locking overhead
  - **coarse** granularity (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy

- The highest level in the example hierarchy is the entire database.
- The levels below are of type area, file (table) and record (tuple) in that order.



How to efficiently determine whether a lock can be imposed on a node?

Intention Lock (意向锁) Modes

- Three additional lock modes with multiple granularity:
  - **intention-shared (IS)**
    - indicates explicit locking at a lower level of the tree but only with **shared** locks
  - **intention-exclusive (IX)**
    - indicates explicit locking at a lower level with **exclusive or shared** locks
  - **shared and intention-exclusive (SIX)**
    - the subtree rooted by that node is locked explicitly in **shared** mode and explicit locking is being done at a lower level with **exclusive**-mode locks

- Intention locks allow a higher level node to be locked in **S** or **X** mode without having to check **all descendent nodes**.

Compatibility Matrix with Intention Lock Modes

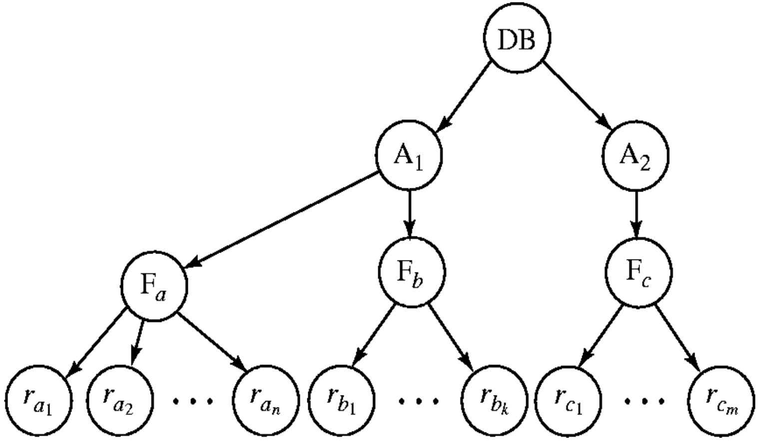
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node **Q**, using the following rules:
  - **The lock compatibility matrix must be observed**.
  - The **root of the tree** must be **locked first**, and may be locked in any mode.
  - A node **Q** can be locked by T_i in **S** or **IS** mode only if the parent of **Q** is currently locked by T_i in either **IX** or **IS** mode.
  - A node **Q** can be locked by T_i in **X**, **SIX**, or **IX** mode only if the parent of **Q** is currently locked by T_i in either **IX** or **SIX** mode.
  - T_i can lock a node only if it has not previously unlocked any node (that is, T_i **is two-phase**).
  - T_i can unlock a node Q only if none of the children of **Q** are currently locked by T_i.
- **Locks** are **acquired** in **root-to-leaf order**, whereas they are **released** in **leaf-to-root order**
- **The multiple-granularity locking protocol can ensure serializability**
- **Deadlock is possible in the multiple-granularity protocol, as it is in the two-phase locking protocol**

Multiple Granularity Locking Scheme

- Suppose that transaction T21 reads record r_{a2} in file F_a. Then, T21 needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a2} in S mode.
- Suppose that transaction T22 modifies record r_{a9} in file F_a. Then, T22 needs to lock the database, area A_1, and file F_a (and in that order) in IX mode, and finally to lock r_{a9} in X mode.
- Suppose that transaction T23 reads all the records in file F_a. Then, T23 needs to lock the database and area A_1 (and in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T24 reads the entire database. It can do so after locking the database in S mode



Outline

- Concurrent Control
- Lock-based Protocols
- Graph-based Protocols
- Multiple Granularity

☞ **Deadlock Handling**

Deadlock Handling

- Consider the following two transactions:

  T_1: write(X) T_2: write(Y)
  write(Y) write(X)

- Schedule with 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

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set
- **Deadlock prevention protocols** ensure that the system will never enter into a deadlock state.
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - 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).

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention
  - **wait-die scheme — non-preemptive(非抢占)**
    - older transactions wait for younger ones to release data items, younger transactions never wait for older ones and roll back instead.
    - one transaction may die several times before acquiring the needed data item
  - **wound-wait scheme — preemptive(抢占)**
    - older transactions would force the rollback of younger transactions instead of waiting for them, younger transactions may wait for older ones.
    - may be fewer rollbacks than wait-die scheme

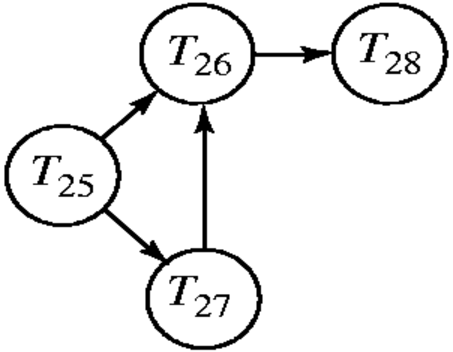
Deadlock Prevention (Cont.)

- Both in wait-die and in wound-wait schemes
  - a rolled back transactions is restarted with its original timestamp
  - older transactions thus have precedence over newer ones, and starvation is hence avoided
- **Timeout-based schemes (基于超时的机制)**
  - a transaction waits for a lock for a specified amount of time. After that, the transaction is rolled back
  - thus deadlocks are not possible
  - simple to implement but starvation is possible. Also difficult to determine the good value of the timeout interval.

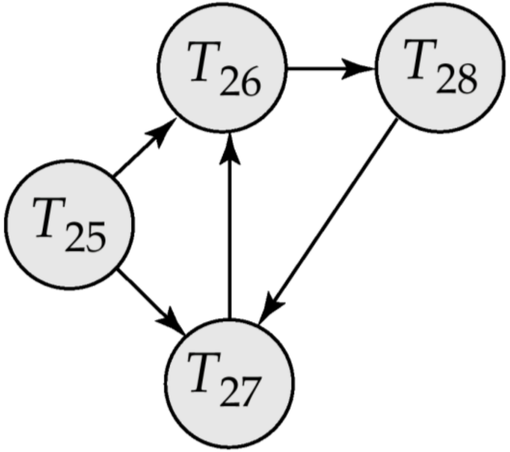
Deadlock Detection

- Deadlocks can be described as **a wait-for graph(等待图)** $G = (V,E)$
  - **V** is a set of vertices (all the transactions in the system)
  - **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**, then 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
- The system is in a deadlock state **iff the wait-for graph has a cycle**. Must invoke a **deadlock-detection algorithm** periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph **without a cycle**



Wait-for graph **with a cycle**

Deadlock Recovery

- **When deadlock is detected**
  - Some transaction needs to roll back
  - **Rollback** -- determine how far to roll back the transaction
    - **Total rollback:** abort the transaction and then restart it
    - **Partial rollback:** more effective to roll back transaction only as far as necessary to break the deadlock
  - **Starvation** happens if same transaction is always chosen as victim
  - Include the number of rollbacks in the cost factor to avoid starvation

Assignments

- Practice exercises: 18.2
- Submission: 12:59pm, June 12, 2024

End of Lecture 12