TONGJI UNIVERSITY

ADMIS Lab
先进数据与机器智能系统实验室

并发控制
Concurrency Control

李文根/Wengen Li
Email: lwengen@tongji.edu.cn

先进数据与机器智能系统实验室（ADMIS）
https://admis.tongji.edu.cn/main.htm

TONGJI UNIVERSITY
电子与信息工程学院 计算机科学与技术系
College of Electronics and Information Engineering Department of Computer Science and Technology

课程概要

- **Part 0: Overview**
  - Ch1: Introduction
- **Part 1 Relational Databases**
  - Ch2: Relational model
  - Ch3: Introduction to SQL
  - Ch4: Intermediate SQL
  - Ch5: Advanced SQL
- **Part 2 Database Design**
  - Ch6: Database design based on E-R model
  - Ch7: Relational database design
- **Part 3 Application Design & Development**
  - Ch8: Complex data types
  - Ch9: Application development
- **Part 4 Big data analytics**
  - Ch10: Big data
  - Ch11: Data analytics
- **Part 5 Data Storage & Indexing**
  - Ch12: Physical storage system
  - Ch13: Data storage structure
  - Ch14: Indexing
- **Part 6 Query Processing & Optimization**
  - Ch15: Query processing
  - Ch16: Query optimization
- **Part 7 Transaction Management**
  - Ch17: Transactions
  - **Ch18: Concurrency control**
  - Ch19: Recovery system
- **Part 8 Parallel & Distributed Database**
  - Ch20: Database system architecture
  - Ch21-23: Parallel & distributed storage, query processing & transaction processing
- **Part 9**
  - DB Platform: **OceanBase**, MongoDB, Neo4J

目录

- 并发控制中的问题
- 基于锁的协议
- 基于图的协议
- 死锁处理
- 多粒度

并发控制中的问题

- **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

丢失修改（Lost Update）

- Transactions T_1 and T_2 read the same data item A 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	$A \leftarrow A-1$ W(A)=15	
④		$A \leftarrow A-1$ W(A)=15

不可重复读（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 B back
- 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

脏读（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	

目录

- 并发控制中的问题
- 基于锁的协议
- 基于图的协议
- 死锁处理
- 多粒度

基于锁的协议

- **Lock-based protocols**
  - 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**
  - Made to the concurrency control manager (并发控制管理器)
  - Transaction can proceed only after the lock request is granted

基于锁的协议（续）

- **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 the locks already held on the data item by other transactions
- If a lock cannot be granted, the requesting transaction **waits** till all the incompatible locks have been released

解决丢失修改

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

解决不可重复读

	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

解决脏读

	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

基于锁的协议（续）

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

- This locking 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

两阶段锁协议（Two-Phase Locking Protocol）

TONGJI UNIVERSITY

- 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

两阶段锁协议（续）

- **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；

两阶段锁协议（续）

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

两阶段锁协议（续）

- **Strict two-phase locking(严格两阶段封锁)**
  - Cascading roll-back is possible under two-phase locking
  - In strict two-phase locking, a transaction must hold all its **exclusive locks** till it commits
- **Rigorous two-phase locking (强两阶段封锁)**
  - All locks are held till the transaction commits
  - Transactions can be serialized in the order in which they commit

锁转换（Lock Conversions）

- 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)

锁的自动获取

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

if T_i *has a lock on D, then*

    read(D);

else

    wait until no other transactions have a lock-X on D;

    grant T_i *a lock-S on D;*

    read(D)

锁的自动获取（续）

- write(D) is processed as:

if T_i *has a lock-X on D, then*
  write(D);
else
  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);

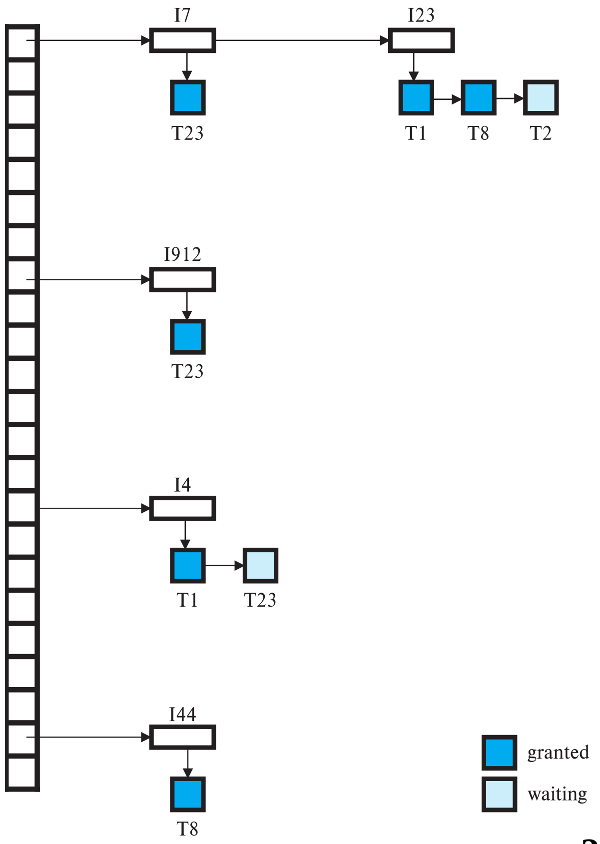
- All locks are released after the transaction commits

锁的实现

- **Lock manager (锁管理器)**
  - Usually 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

锁表（Lock Table）

- Dark blue rectangles indicate granted locks, and light blue 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 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
  - lock manager may keep a list of locks held by each transaction, to implement this operation efficiently



死锁（Deadlock）

TONGJI UNIVERSITY

- 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
  - **Two-phase locking, including the strict and rigorous versions, cannot avoid deadlocks**

饥饿（Starvation）

- **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

目录

同济大学 TONGJI UNIVERSITY

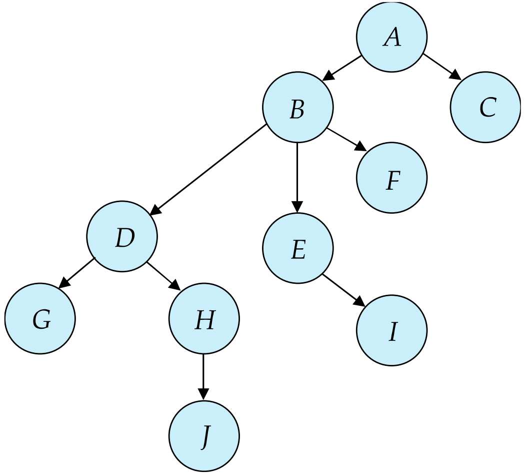
- 并发控制中的问题
- 基于锁的协议
- 基于图的协议
- 死锁处理
- 多粒度

基于图的协议

- **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
    - **Partial ordering**: according to the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control
  - If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j
  - The set D can be viewed as a directed acyclic graph, called **database graph**
- The tree-protocol is a simple kind of graph protocol

树协议（Tree Protocol）

- **Only exclusive locks are allowed in tree protocol**
  - 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
  - An unlocked data item cannot be relocked by T_i



基于图的协议（续）

- **Advantages**
  - The tree protocol ensures conflict serializability as well as freedom from deadlock
  - Unlocking may occur earlier than two-phase locking protocol, hence shorter waiting time and higher concurrency

- **Disadvantages**
  - The abort of a transaction can still lead to cascading rollbacks
  - May have to lock data items that it does not access, thus increasing locking overhead, and incurring additional waiting time

基于时间戳的协议

- **Timestamp of a transaction**
  - 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).

- **Timestamp-based protocol**
  - The protocol manages concurrent execution such that the timestamps determine the serializability order
  - To assure such behavior, the protocol maintains two timestamp values for each data Q:
    - **W-timestamp(Q)**: the largest time-stamp of any transaction that executed write(Q) successfully
    - **R-timestamp(Q)**: the largest time-stamp of any transaction that executed read(Q) successfully

基于时间戳的协议（续）

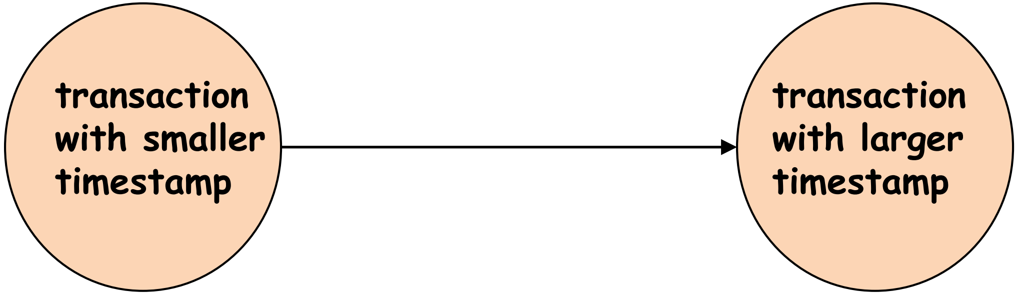
- The timestamp-based protocol ensures that any conflicting read and write operations are executed in timestamp order
- Suppose that transaction T_i issues a read(Q)
  - If TS(T_i) $\leq$ W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten
    - the read operation is rejected, and T_i is rolled back
  - If TS(T_i) $\geq$ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to max(R-timestamp(Q), TS(T_i))

基于时间戳的协议（续）

- 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 the 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
    - 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)

基于时间戳的协议（续）

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



  - 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

目录

- 并发控制中的问题
- 基于锁的协议
- 基于图的协议
- 死锁处理
- 多粒度

死锁的处理

- 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	

死锁的处理（续）

- 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 starts 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)

死锁预防

- 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

死锁预防（续）

- 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

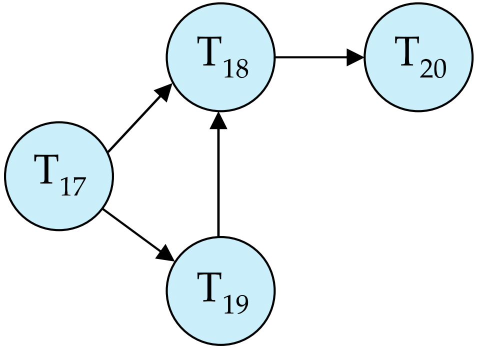
- **Timeout-based schemes (基于超时的机制)**
  - a transaction waits for a lock for a specified amount of time. After that, the transaction is rolled back, thus deadlocks are not possible
  - simple to implement but starvation is possible. Also difficult to determine the good value of the timeout interval.

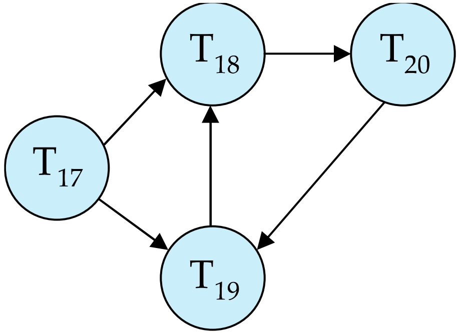
死锁检测

- Deadlocks can be described as a wait-for graph(等待图) G = (V,E)
  - V is a set of vertices corresponding to all the transactions in the system
  - E is a set of edges and each edge is an ordered pair $T_i \rightarrow T_j$ indicating 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

死锁检测（续）





Wait-for graph without a cycle　　Wait-for graph with a cycle

死锁恢复

- **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 the same transaction is always chosen as victim
  - Include the number of rollbacks in the cost factor to avoid starvation

目录

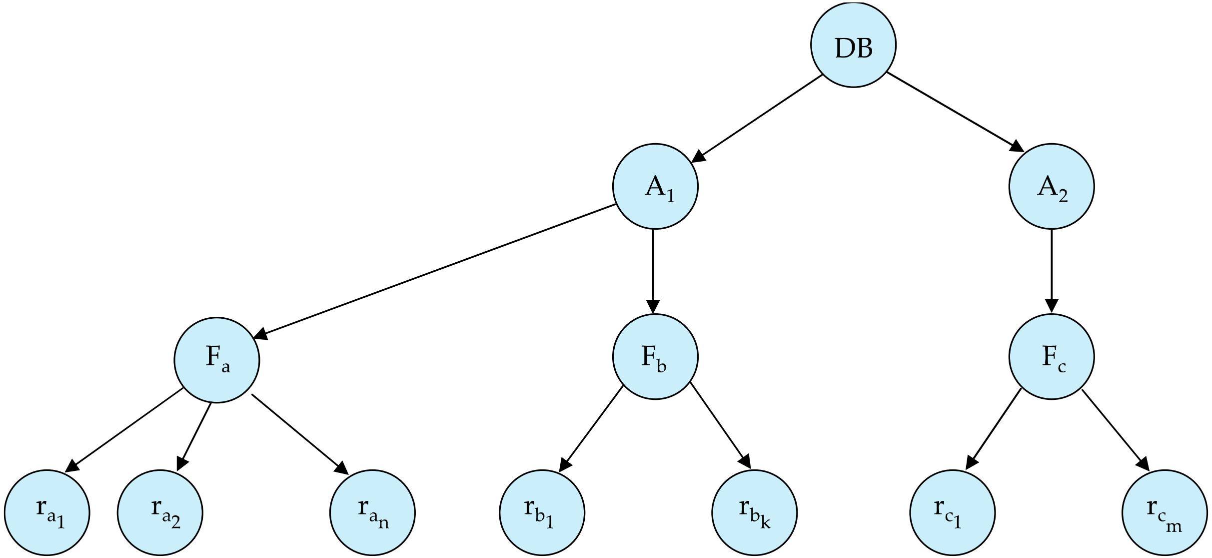
- 并发控制中的问题
- 基于锁的协议
- 基于图的协议
- 死锁处理
- 多粒度

多粒度

- Allow data items to be of various sizes and define a hierarchy of data granularities
- Can be represented 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 concurrency, low locking overhead

多粒度举例

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



意向锁（Intention Lock）

TONGJI UNIVERSITY 同济大学

- Three additional lock modes with multiple granularity
  - intention-shared (IS): 意向共享模式锁
    - indicates explicit locking at a lower level of the tree but only with shared locks
  - intention-exclusive (IX): 意向排他模式锁
    - indicates explicit locking at a lower level with exclusive or shared locks
  - shared intention-exclusive (SIX): 共享意向排他锁
    - 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

锁相容性矩阵

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

多粒度锁模式

- Transaction T_i can lock a node Q, using the following rules:
  - The lock compatibility matrix must be followed
  - 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