



# 并发控制

# Concurrency Control

李文根/Wengen Li

Email: [lwengen@tongji.edu.cn](mailto:lwengen@tongji.edu.cn)

先进数据与机器智能系统实验室

Advanced Data and Machine Intelligence Systems (ADMIS) Lab

<https://admis-tongji.github.io>

同济大学 计算机科学与技术学院

2026年05月

- **Part 0: Overview**
  - Ch1: Introduction
- **Part 1 Relational Languages**
  - Ch2: Relational model
  - Ch3: Introduction to SQL
  - Ch4: Intermediate SQL
  - Ch5: Advanced SQL
- **Part 2 Database Design**
  - Ch6: Database design via 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 Storage Management & Indexing**
  - Ch12: Physical storage systems
  - Ch13: Data storage structures
  - 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
- **Advanced topics**
  - DB Platform: **OceanBase**, MongoDB, Neo4J
  - RAG, Multimodal retrieval, ...

- **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

- **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 $A \leftarrow A - 1$<br>$W(A)=15$ |                                   |
| ④                                   | $A \leftarrow A - 1$<br>$W(A)=15$ |

## ► 不可重复读 (Non-repeatable Read)



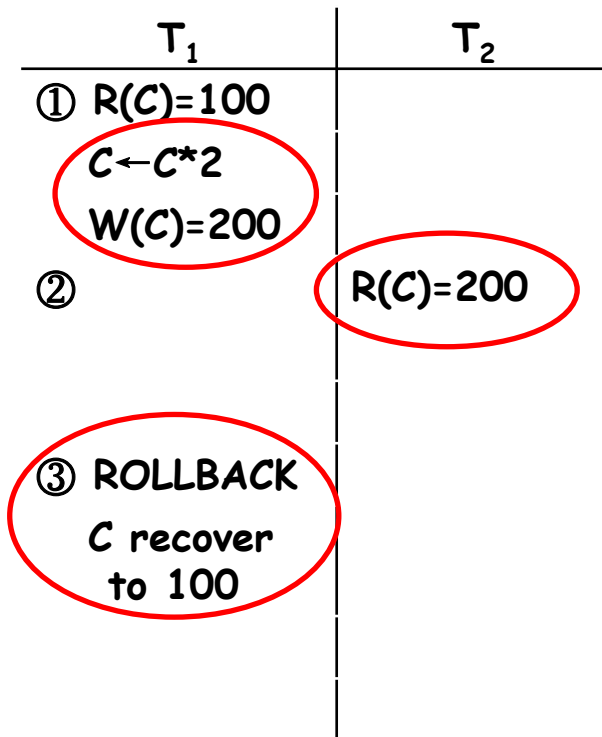
| $T_1$                                                        | $T_2$                                            |
|--------------------------------------------------------------|--------------------------------------------------|
| ① $R(A)=50$<br>$R(B)=100$<br>sum=150                         |                                                  |
| ②                                                            | $R(B)=100$<br>$B \leftarrow B * 2$<br>$W(B)=200$ |
| ③ $R(A)=50$<br>$R(B)=200$<br>sum=250<br>(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



- 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

- **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 $A \leftarrow A-1$ | wait               |
| $W(A)=15$            | wait               |
| Commit               | wait               |
| Unlock A             | wait               |
| ④                    | Get Xlock A        |
|                      | $R(A)=15$          |
|                      | $A \leftarrow A-1$ |
| ⑤                    | $W(A)=14$          |
|                      | Commit             |
|                      | Unlock A           |

## ► 解决不可重复读



| T <sub>1</sub>                                                     | T <sub>2</sub>                                                  |
|--------------------------------------------------------------------|-----------------------------------------------------------------|
| ① Slock A<br>Slock B<br>R(A)=50<br>R(B)=100<br>sum=150             |                                                                 |
| ②                                                                  | Xlock B<br>wait<br>wait<br>wait<br>wait<br>wait<br>wait<br>wait |
| ③ R(A)=50<br>R(B)=100<br>sum=150<br>Commit<br>Unlock A<br>Unlock B |                                                                 |
| ④                                                                  | get XlockB<br>R(B)=100<br>B ← B*2                               |
| ⑤                                                                  | W(B)=200<br>Commit<br>Unlock B                                  |

## ► 解决脏读



| $T_1$                                                     | $T_2$                                   |
|-----------------------------------------------------------|-----------------------------------------|
| ① Xlock C<br>R(C)=100<br>$C \leftarrow C * 2$<br>W(C)=200 |                                         |
| ②                                                         | Slock C<br>wait<br>wait<br>wait<br>wait |
| ③ ROLLBACK<br>(C rec. 100)<br>Unlock C                    |                                         |
| ④                                                         | get Slock C<br>R(C)=100                 |
| ⑤                                                         | Commit C<br>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)



- 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

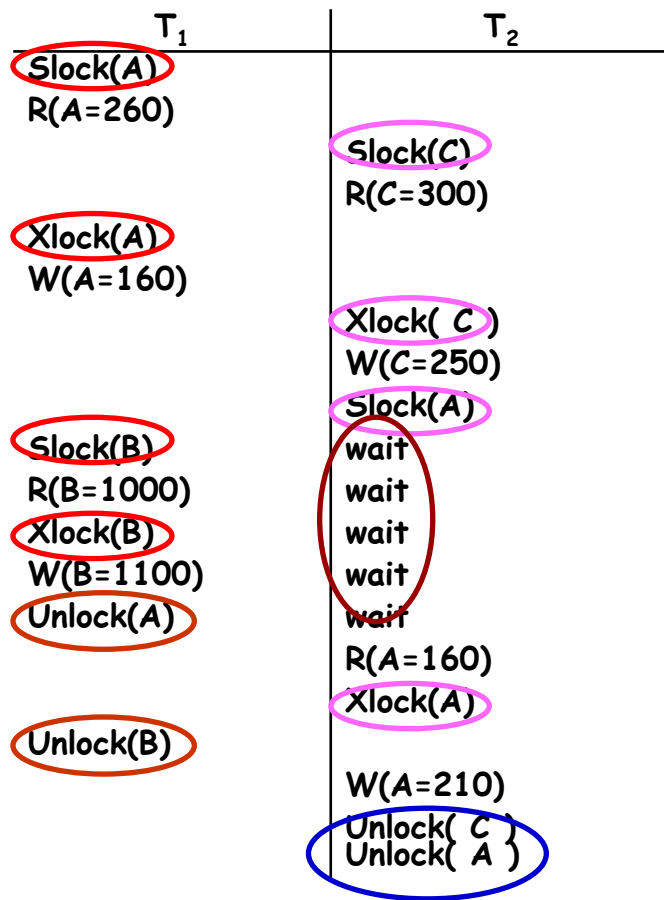
Slock A Slock B Xlock C Unlock B Unlock A Unlock C

|← Growing →| |← Shrinking →|

- Not satisfy 2PL

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

## ► 两阶段锁协议 (续)



2PL ensures serializable schedules

- **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  $\rightarrow$  lock-X
  - Downgrade (降级)
    - lock-X  $\rightarrow$  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( $a_1+a_2$ )

- 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

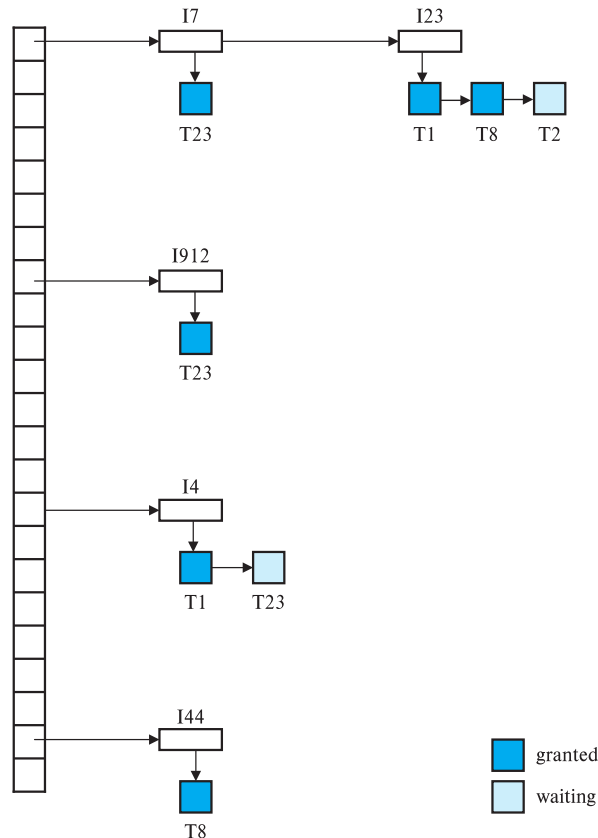
- **Lock manager (锁管理器)**

- 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 and 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)



- 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**

- **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

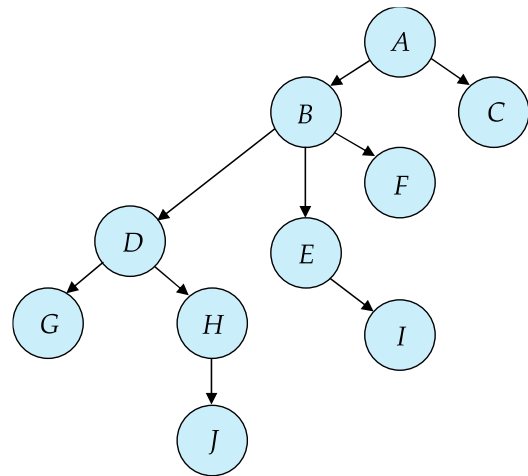
- 并发控制中的问题
- 基于锁的协议
- **基于图的协议**
- 死锁处理
- 多粒度

- **Graph-based protocols** are an alternative to two-phase locking
  - Impose a partial ordering  $\rightarrow$ (偏序) on the set  $D = \{d_1, d_2, \dots, d_n\}$  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 type 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  $T_i$



- **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

- **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)



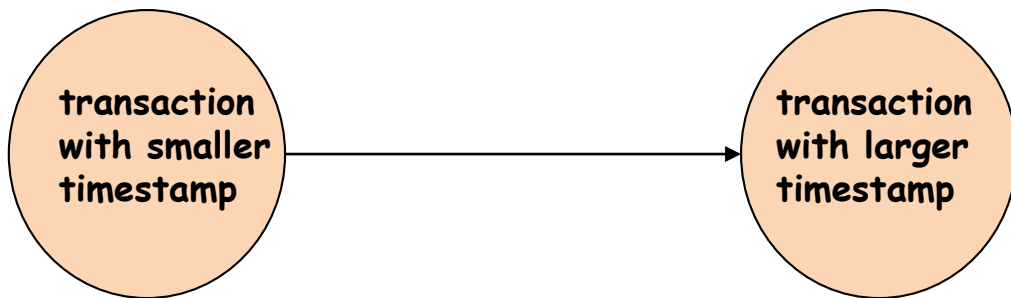
- The timestamp-based protocol ensures that any conflicting read and write operations are executed in timestamp order
- Suppose that transaction  $T_i$  issues a read(Q)
  - If  $TS(T_i) \leq W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of Q that was already overwritten
    - the read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

- Suppose that transaction  $T_i$  issues write(Q)
  - If  $TS(T_i) < R\text{-timestamp}(Q)$ , 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  $TS(T_i) < W\text{-timestamp}(Q)$ , 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  $W\text{-timestamp}(Q)$  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

- 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<br>write (X) | lock-X on Y<br>write (Y)<br>wait for lock-X on X |
| wait for lock-X on Y     |                                                  |



## 死锁的处理（续）



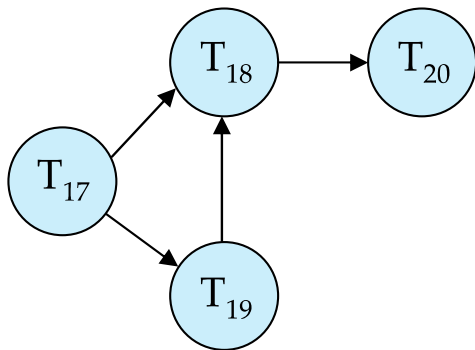
- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set
- Deadlock prevention protocols ensure that the system will never enter into a deadlock state.
  - Require that each transaction locks all its data items before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

- 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

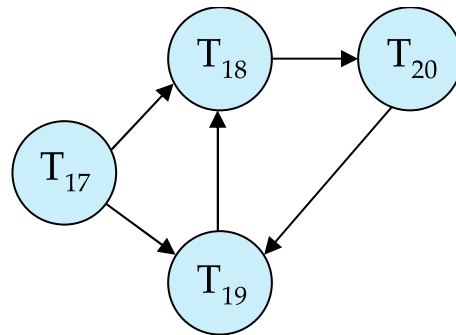
- Both in wait-die and in wound-wait schemes
  - a rolled back transactions is restarted with its original timestamp
  - older transactions thus have precedence over newer ones, and starvation is hence avoided
- **Timeout-based schemes (基于超时的机制)**
  - a transaction waits for a lock for a specified amount 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.

- 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

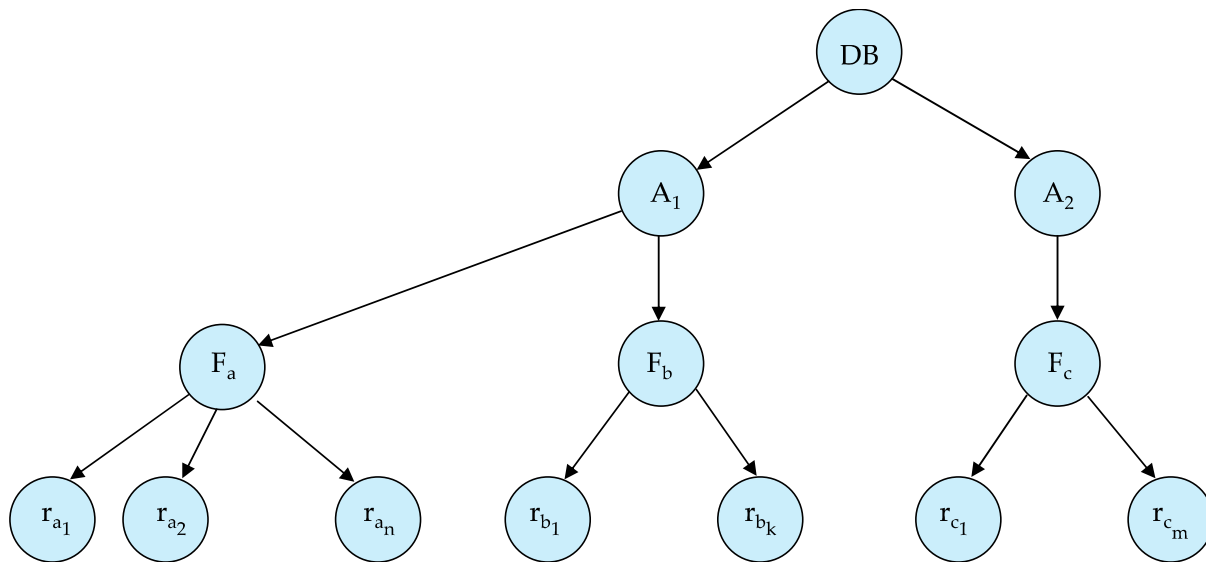
- **When deadlock is detected**

- 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

- 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

- 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

- 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)



- 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 |

- 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



# 恢复系统 Recovery System

李文根/Wengen Li

Email: [lwengen@tongji.edu.cn](mailto:lwengen@tongji.edu.cn)

先进数据与机器智能系统实验室

Advanced Data and Machine Intelligence Systems (ADMIS) Lab

<https://admis-tongji.github.io>

同济大学 计算机科学与技术学院

2026年05月

- **Part 0: Overview**
  - Ch1: Introduction
- **Part 1 Relational Languages**
  - Ch2: Relational model
  - Ch3: Introduction to SQL
  - Ch4: Intermediate SQL
  - Ch5: Advanced SQL
- **Part 2 Database Design**
  - Ch6: Database design via 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 Storage Management & Indexing**
  - Ch12: Physical storage systems
  - Ch13: Data storage structures
  - 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
- **Advanced topics**
  - DB Platform: **OceanBase**, MongoDB, Neo4J
  - RAG, Multimodal retrieval, ...

- **故障分类**
- **存储器**
- **恢复与原子性**
- **恢复算法**
- **缓冲区管理**

- **Transaction failure (事务故障)**
  - Logical errors, e.g., illegal inputs
  - System errors, e.g., dead locks
- **System crash (系统崩溃)**
  - A power failure, or other hardware and software failures cause the system to crash
- **Disk failure (磁盘故障)**
  - A head crash or similar disk failures destroy all or part of disk storage

- Techniques to ensure database consistency and transaction atomicity **despite failures**
- Recovery algorithms have two parts
  - Actions taken during normal transaction processing
    - 保证有足够的信息用于故障恢复
  - Actions taken after a failure
    - 恢复数据库到某个一致性状态

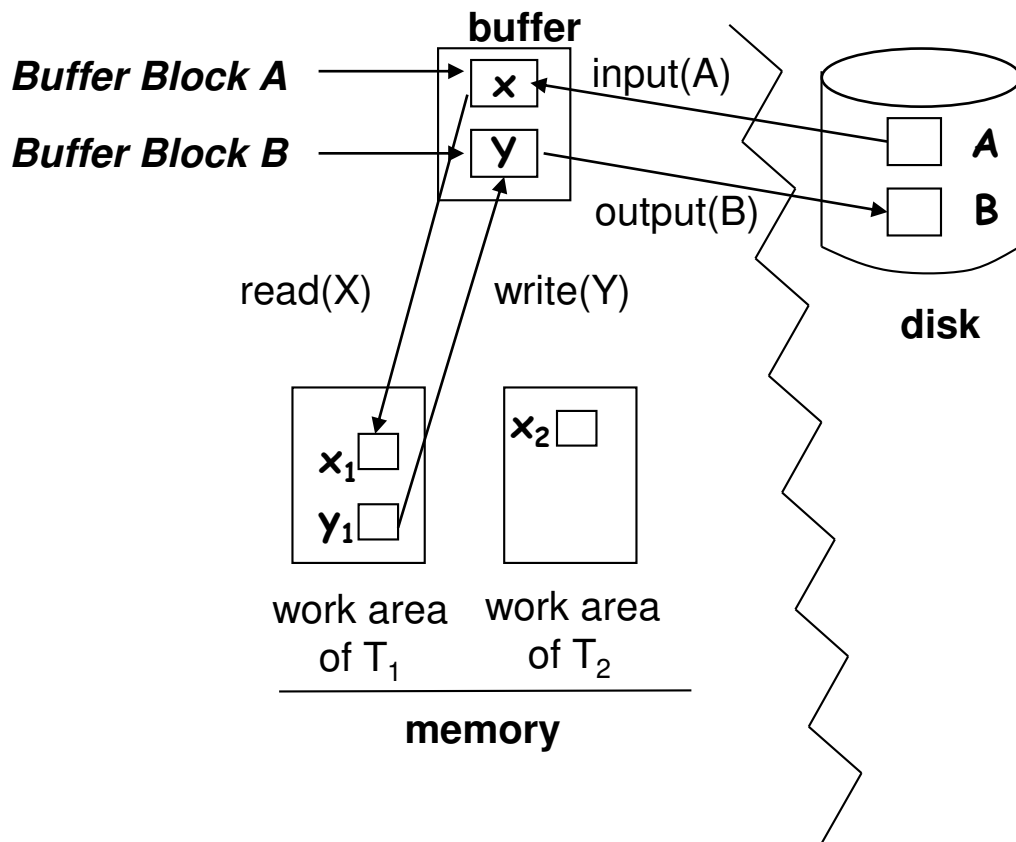
- 故障分类
- **存储器**
- 恢复与原子性
- 恢复算法
- 缓冲区管理

- **Volatile storage (易失性存储器)**
  - does not survive system crashes
  - e.g., main memory, cache memory
- **Non-volatile storage (非易失性存储器)**
  - survives system crashes
  - e.g., disk, tape, flash memory
- **Stable storage (稳定存储器)**
  - a mythical form of storage that survives all failures
  - achieved by maintaining multiple copies on distinct non-volatile media

- **Physical blocks (物理块)**
  - the blocks residing on the disk
- **Buffer blocks (缓冲块)**
  - the blocks residing temporarily in main memory
- Block movements between disk and main memory
  - **input(B)**: physical block  $\rightarrow$  memory (buffer)
  - **output(B)**: buffer block  $\rightarrow$  disk
- Each transaction  $T_i$  has its **private work-area** (私有工作区)
  - $T_i$ 's local copy of a data item  $X$  is denoted by  $x_i$

- Transaction transfers data items between system buffer blocks and its private work-area using
  - read(X)
  - write(X)
- **Transactions**
  - Perform read(X) while accessing X for the first time
  - All subsequent accesses are to the local copy
  - After the last access, transaction executes write(X)
- **output( $B_x$ )** does not need to immediately follow write(X)
  - System performs the output operation when it deems fit

## ► 数据访问 (续)



- 故障分类
- 存储器
- 恢复与原子性
- 恢复算法
- 缓冲区管理

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Consider transaction  $T_i$  that transfers \$50 from account A to account B
  - Several output operations may be required for  $T_i$  to output A and B
  - A failure may occur after one of these modifications have been made but before all of them are made
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database
- **Two approaches**
  - Shadow-paging (影子页)
  - **Log-based recovery (基于日志的恢复)**



- A log is kept on stable storage
  - The log is a sequence of log records
- When transaction  $T_i$  starts, it registers itself by writing a  $\langle T_i \text{ start} \rangle$  log record
  - Before  $T_i$  executes  $\text{write}(X)$ , a log record  $\langle T_i, X, V_1, V_2 \rangle$  is written, where  $V_1$  is the old value and  $V_2$  is the new value
  - When  $T_i$  finishes the last statement, the log record  $\langle T_i, \text{commit} \rangle$  is written
- Two approaches to use logs
  - **Deferred database modification (延迟数据库修改)**
  - **Immediate database modification (即刻数据库修改)**



- Record all modifications to the log, but defer all the writes to after partial commit
  - Transaction starts by writing  $\langle T_i \text{ start} \rangle$  record to log
  - A write(X) operation results in a log record  $\langle T_i, X, V_1, V_2 \rangle$  being written, where  $V_2$  is the new value
  - The write is not performed on X at this time, but is deferred
  - When  $T_i$  partially commits,  $\langle T_i \text{ commit} \rangle$  is written to the log
  - Finally, **the log records are read** and used to actually execute the previously deferred writes

- **Recovery after a crash**

- A transaction needs to be redone iff both  $\langle T_i \text{ start} \rangle$  and  $\langle T_i \text{ commit} \rangle$  are in the log
- $\text{Redo}(T_i)$  sets the value of all data items updated by the transaction to the new values

- **Example:**

- $T_0$  executes before  $T_1$ , and initial:  $A=1000$ ,  $B=2000$ ,  $C=700$

$T_0$ : read (A)

$T_1$  : read (C)

$A := A - 50$

$C := C - 100$

write (A)

write (C)

read (B)

$B := B + 50$

write (B)

- Recovery actions in each failure case below:
  - (a) no redo actions need to be taken
  - (b) redo( $T_0$ ) must be performed
  - (c) redo( $T_0$ ) must be performed followed by redo( $T_1$ )

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

(a)

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  commit>

< $T_1$  start>

< $T_1$ , C, 700, 600>

(b)

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  commit>

< $T_1$  start>

< $T_1$ , C, 700, 600>

< $T_1$  commit>

(c)



- Allows database updates of an uncommitted transaction
  - Update log records must be written **before** database items are written
  - The output of updated blocks can take place at any time before or after transaction commit
  - Order in which blocks are output can be different from the order in which they are written

## ▶ 即刻数据库修改 (续)



| Log                                  | Write      | Output     |
|--------------------------------------|------------|------------|
| $\langle T_0 \text{ start} \rangle$  |            |            |
| $\langle T_0, A, 1000, 950 \rangle$  |            |            |
| $\langle T_0, B, 2000, 2050 \rangle$ |            |            |
|                                      | $A = 950$  |            |
|                                      | $B = 2050$ |            |
| $\langle T_0 \text{ commit} \rangle$ |            |            |
| $\langle T_1 \text{ start} \rangle$  |            |            |
| $\langle T_1, C, 700, 600 \rangle$   |            |            |
|                                      | $C = 600$  |            |
|                                      |            | $B_B, B_C$ |
| $\langle T_1 \text{ commit} \rangle$ |            | $B_A$      |

**Note:**  $B_X$  denotes the block containing  $X$

- Recovery procedure has two operations
  - $\text{undo}(T_i)$ : restore the values of all the data items updated by transaction  $T_i$  to the old values
  - $\text{redo}(T_i)$ : set the values of all data items updated by transaction  $T_i$  to the new values
- When recovering after failure
  - Transaction  $T_i$  needs to be undone if the log contains the record  $\langle T_i \text{ start} \rangle$ , but does not contain  $\langle T_i \text{ commit} \rangle$
  - Transaction  $T_i$  needs to be redone if the log contains both the record  $\langle T_i \text{ start} \rangle$  and  $\langle T_i \text{ commit} \rangle$
- Undo operations are performed first, then redo operations

$\langle T_0 \text{ start} \rangle$   
 $\langle T_0, A, 1000, 950 \rangle$   
 $\langle T_0, B, 2000, 2050 \rangle$

(a)

$\langle T_0 \text{ start} \rangle$   
 $\langle T_0, A, 1000, 950 \rangle$   
 $\langle T_0, B, 2000, 2050 \rangle$   
 $\langle T_0 \text{ commit} \rangle$   
 $\langle T_1 \text{ start} \rangle$   
 $\langle T_1, C, 700, 600 \rangle$

(b)

$\langle T_0 \text{ start} \rangle$   
 $\langle T_0, A, 1000, 950 \rangle$   
 $\langle T_0, B, 2000, 2050 \rangle$   
 $\langle T_0 \text{ commit} \rangle$   
 $\langle T_1 \text{ start} \rangle$   
 $\langle T_1, C, 700, 600 \rangle$   
 $\langle T_1 \text{ commit} \rangle$

(c)

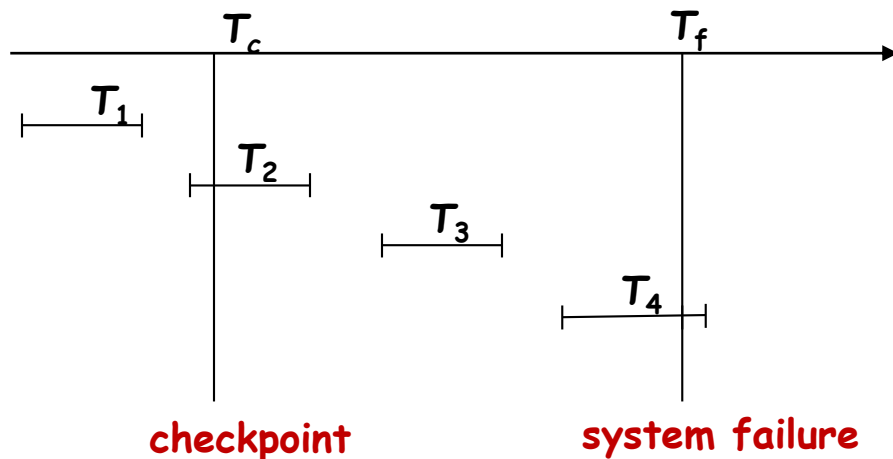
- Recovery actions in each case above are:
  - (a)  $\text{undo}(T_0)$
  - (b)  $\text{undo}(T_1)$  and  $\text{redo}(T_0)$
  - (c)  $\text{redo}(T_0)$  and  $\text{redo}(T_1)$

- **Problems in the recovery procedure**
  - Searching the entire log is time-consuming
  - Might unnecessarily redo transactions which have already output their updates to the database
- **Recovery procedure sets checkpoints periodically**
  - Output all the **log records** currently residing in main memory to stable storage
  - Output all the **modified buffer blocks** to the disk
  - Write a log record <checkpoint> to stable storage

- **During recovery**

- Scan backwards from the end of log to find the most recent <checkpoint> record
- Continue scanning backwards till a record  $\langle T_i \text{ start} \rangle$  is found. We assume that all transactions are executed serially
  - Need only consider the part of log following above start record
  - For all transactions (starting from  $T_i$  or later) with no  $\langle T_j \text{ commit} \rangle$ , execute  $\text{undo}(T_j)$ .
  - Scanning forward in the log, for all transactions starting from  $T_i$  or later with a  $\langle T_j \text{ commit} \rangle$ , execute  $\text{redo}(T_j)$ .

## ► 检查点 (续)



- $T_1$  can be ignored (updates have been already output to disk according to the checkpoint)
- redo  $T_2$  and  $T_3$
- undo  $T_4$

- 故障分类
- 存储器
- 恢复与原子性
- **恢复算法**
- 缓冲区管理



## 并发事务的恢复



- Modify the log-based recovery scheme to allow multiple transactions to execute concurrently
  - All transactions share **a single disk buffer** and **a single log**
  - A buffer block can have data items updated by multiple transactions
- Assume that concurrency control uses strict two-phase locking
- Logging is done as described earlier
  - Log records of different transactions may be interspersed (散布) in the log
- The checkpointing technique and actions taken on recovery have to be changed

- Checkpoints are performed as before, except that the checkpoint log record is the form  $\langle \text{checkpoint } L \rangle$ 
  - $L$  is a list of active transactions at the time of the checkpoint
  - No update is in progress while the checkpoint is carried out
- When the system recovers from a crash
  - Initialize undo-list and redo-list to empty
  - Scan the log backwards until a  $\langle \text{checkpoint } L \rangle$  record is found
    - if the record is  $\langle T_i \text{ commit} \rangle$ , add  $T_i$  to redo-list
    - if the record is  $\langle T_i \text{ start} \rangle$  and  $T_i$  is not in redo-list, add  $T_i$  to undo-list
    - for every  $T_i$  in  $L$ , if  $T_i$  is not in redo-list, add  $T_i$  to undo-list

## ► 并发事务的恢复（续）



- Go through the steps of the recovery algorithm on the following log

< $T_0$  **start**>  
< $T_0$ , A, 0, 10>  
< $T_0$  **commit**>  
< $T_1$  **start**>  
< $T_1$ , B, 0, 10>  
< $T_2$  **start**>  
< $T_2$ , C, 0, 10>  
< $T_2$ , C, 10, 20>  
<checkpoint { $T_1$ ,  $T_2$ }>  
< $T_3$  **start**>  
< $T_3$ , A, 10, 20>  
< $T_3$ , D, 0, 10>  
< $T_3$  **commit**>

- **Recovery**

- Scan log backwards from the end of the log
  - During the scan, perform undo for each log record that belongs to a transaction in undo-list
- Locate the most recent <checkpoint L> record, and scan log forwards from the <checkpoint L> record till the end of the log
  - During the scan, perform redo for each log record that belongs to a transaction on redo-list

- 故障分类
- 存储器
- 恢复与原子性
- 恢复算法
- 缓冲区管理



- **Log record buffering**

- Log records are **buffered in main memory**, instead of being output directly to stable storage
- Log records are output to stable storage when a block of log records in the buffer is full, or a log force operation is executed
- **Log force** is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage
- Multiple log records can be output using a single output operation, thus reducing the I/O cost

- **Write-ahead logging (WAL) rule for buffering log records**
  - Log records are output to stable storage in the order in which they are created
  - Transaction  $T_i$  enters the commit state only when the log record  $\langle T_i \text{ commit} \rangle$  has been output to stable storage
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, an existing block should be removed from buffer if the buffer is full
  - If the block chosen for removal has been updated, it must be output to disk
- No update should be in progress on a block when it is output to disk, which is ensured as follows
  - Before a block is output to disk, the system acquires an **exclusive latch (锁)** on the block
    - Ensures no update can be in progress on the block

- **Database buffer can be implemented either**
  - In an area of real **main-memory** reserved for the database, or in **virtual memory**
- Implementing buffer in reserved main-memory has drawbacks
  - Memory is partitioned before-hand between database buffer and applications, limiting the flexibility of main-memory
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory

- Database buffers are generally implemented in **virtual memory** in spite of some drawbacks
  - When OS needs to evict (逐出) a page that has been modified, the page is written to swap space on disk
  - When DB decides to write buffer page to disk, buffer page may be in swap space (交换区), and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/Os
    - Known as dual paging (双分页) problem
  - Ideally when swapping out a database buffer page, operating system should **pass control to database**, which in turn outputs the page to database instead of to swap space
    - Dual paging can thus be avoided, but common operating systems do not support such functionality

- **Technique similar to checkpointing is used to deal with loss of non-volatile storage**
  - Periodically dump the entire content of the database to stable storage
  - No transaction may be active during the dump procedure, and a procedure similar to checkpointing must take place
    - output all log records currently residing in main memory onto stable storage
    - output all buffer blocks onto the disk
    - copy the contents of the database to stable storage
    - output a record <dump> to log on stable storage
  - To recover from disk failure
    - restore database from the most recent dump
    - consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump, known as fuzzy dump or online dump