DATABASE SYSTEM PRINCIPLE - TRANSACTION OVERVIEW

李旭东 **Li-Xudong**
LEEXUDONG@NANKAI.EDU.CN
NANKAI UNIVERSITY

OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

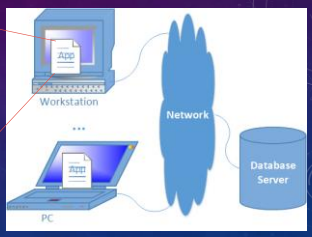
OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

APP: PROGRAM

APP

read (A)
$A := A - 10$
write (A)
read (B)
$B := B + 10$
write (B)



A SIMPLE BUSINESS (1.1)

T_0

read (A)
$A := A - 10$
write (A)
read (B)
$B := B + 10$
write (B)

- A business(collections of operations): transfer $10 from account A to account B
  - Initial value: A=1000, B=2000
  - ?result
  - If a failure happens in read(B), …?
    - In fact, We hope that None occur!!!

A SIMPLE BUSINESS (1.2)

T_0

read (A)
$A := A - 10$
write (A)
read (B)
$B := B + 10$
write (B)

- A business(collections of operations): transfer $10 from account A to account B
  - (1) Atomicity: as a single unit, all or none
  - (2) Consistency:
    - consistent state{A+B unchanged}
  - (3) Durability:
    - if execution completes successfully, all the updates persist, even if a system failure

A SIMPLE BUSINESS (2.1)

T_0	T_1
read (A)	
$A := A - 10$	
write (A)	
read (B)	
	read (A)
	$A := A - 50$
	write (A)
	read (B)
	$B := B + 50$
	write (B)
$B := B + 10$	
write (B)	

- There are two businesses:
  - T0: transfer $10 from account A to account B
  - T1: transfer $50 from account A to account B
  - Initial value: A=1000, B=2000
  - Interleaved交叉(Concurrency): ?result
  - Sequential(T0,T1): ?result
  - Sequential(T1,T0): ?result
  - ? Which is correct result ?

Sequential Execution: OK

A SIMPLE BUSINESS (2.1)

T_0	T_1
read (A)	
$A := A - 10$	
write (A)	
read (B)	
	read (A)
	$A := A - 50$
	write (A)
	read (B)
	$B := B + 50$
	write (B)
$B := B + 10$	
write (B)	

T_0	T_1
read (A)	
$A := A - 10$	
write (A)	
read (B)	
$B := B + 10$	
write (B)	
	read (A)
	$A := A - 50$
	write (A)
	read (B)
	$B := B + 50$
	write (B)

- A business: transfer $10 from account A to account B
  - (4)Isolation:
    - even though multiple businesses may execute concurrently, the system guarantees that every business executes in turn (Sequentially, serially)

A SIMPLE BUSINESS (3)

T_0
Begin transaction
read (A)
$A := A - 10$
write (A)
read (B)
$B := B + 10$
write (B)
End transaction

- We need to invent **Transaction** concept
  - *collections of operations* that form a single logical unit of work which do all or none.
  - Begin transaction, ..., end transaction

ACID Properties of Transaction

- (1) Atomicity: as a single unit, all or none
- (2) Consistency: consistent state{A+B unchanged}
- (3) Durability: if execution completes successfully, all the updates persist, even if a system failure
- (4)Isolation: even though multiple businesses may execute concurrently, the system guarantees that every business executes in turn (Sequentially, serially)

OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

TRANSACTION: ACID PROPERTIES

A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

TRANSACTION: HOW TO ENSURE THE ACID

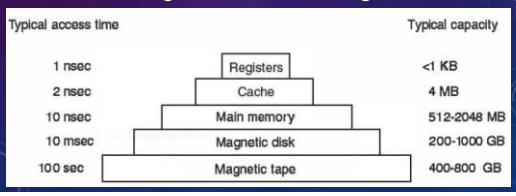
- (1)How to ensure the consistency?
  - (1.1)for a transaction, the logical operations must be correct
    - Erroneous transaction logic can lead to inconsistency
  - (1.2)If we can ensure the atomicity, durability and isolation
  - *(1.3)Don't worry that during transaction execution the database may be temporarily inconsistent*

TRANSACTION: HOW TO ENSURE THE ACID

- (2)How to ensure the durability?
  - Stable Storage
  - (transaction) Log

TRANSACTION: HOW TO ENSURE THE ACID

- Storage Hierarchy
  - Volatile storage、Nonvolatile Storage

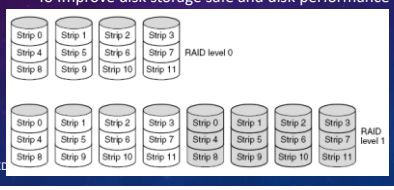


TRANSACTION: HOW TO ENSURE THE ACID

- Stable Storage
  - Information residing in stable storage is never lost!
    - Theoretically never cannot be guaranteed!
  - But, for any a single nonvolatile storage (magnetic disk or flash storage) , it is not absolutely reliable or safe!
  - How to implement stable storage?

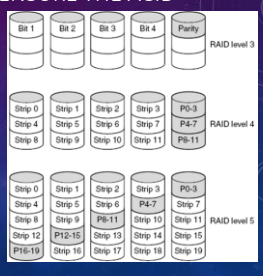
TRANSACTION: HOW TO ENSURE THE ACID

- Stable Storage candidate
  - RAID: Redundant Array of Independent Disks
    - To Improve disk storage safe and disk performance



TRANSACTION: HOW TO ENSURE THE ACID

- Stable Storage candidate
  - RAID



TRANSACTION: HOW TO ENSURE THE ACID

- Stable Storage candidates
  - RAID
  - ?LVM
  - ?Cloud
  - …

TRANSACTION: HOW TO ENSURE THE ACID

- (2)How to ensure the durability?
  - Stable Storage
  - (transaction) Log
    - for update log record
      - Transaction identifier、Data-item identifier
      - Old value, New value

TRANSACTION: LOG

- A log is kept on stable storage: a sequence of log records
- When transaction T_i starts, it registers itself by writing a record
  $<T_i$ **start**>
  to the log
- *Before* T_i executes **write**(X), a log record
  $<T_i, X, V_1, V_2>$
  is written, where V_1 is the value of X before the write (the old value), and V_2 is the value to be written to X (the new value).
- When T_i finishes it last statement, the log record $<T_i$ **commit**> is written.
- Two approaches using logs
  - (1) Immediate database modification
  - (2)Deferred database modification

TRANSACTION: HOW TO ENSURE THE ACID

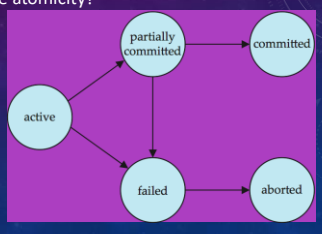
- (3)How to ensure the atomicity?
  - Aborted transaction
    - The transaction doesn't complete its execution successfully
    - Rolled back by log
  - Committed transaction
    - The transaction completes its execution successfully

TRANSACTION: HOW TO ENSURE THE ACID

- (3)How to ensure the atomicity?
  - Compensating transaction补偿事务
    - Once a transaction has committed, we cannot undo its effects by aborting it
    - The only way to undo the effects of a committed transaction is execute another transaction which do the opposite operations
  - example
    - A transaction : transfer $10 from account A to account B
    - A compensating transaction : transfer $10 from account B to account A

TRANSACTION: HOW TO ENSURE THE ACID

- (3)How to ensure the atomicity?
  - Transaction model
    - Transaction states



TRANSACTION STATES

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement(code) has been executed.
- **Failed** - after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - Restart the transaction : can be done only if no internal logical error
    - Kill the transaction
- **Committed** – after successful completion.

©LXD

TRANSACTION: HOW TO ENSURE THE ACID

- (3)How to ensure the atomicity?
  - Use transaction

T_0	T_0	T_0	T_0
Begin transaction	*Begin transaction*	*Begin transaction*	*Begin transaction*
read (A)	read (A)	read (A)	read (A)
$A := A - 10$	$A := A - 10$	$A := A - 10$	$A := A - 10$
write (A)	write (A)	write (A)	write (A)
rollback	read (B)	read (B)	read (B)
read (B)	***rollback***	$B := B + 10$	$B := B + 10$
$B := B + 10$	$B := B + 10$	write (B)	write (B)
write (B)	write (B)	***rollback***	***commit***
End transaction	*End transaction*	*End transaction*	*End transaction*

©LXD

TRANSACTION: HOW TO ENSURE THE ACID

- (4)How to ensure the isolation?
  - Isolation in only one transaction in whole system
    - No problem
  - Isolation involved multi transactions
    - even though multiple businesses may execute concurrently, the system guarantees that every business executes in turn (Sequentially, serially)

©LXD

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

T_0	T_1
read (A)	read (A)
$A := A - 10$	$A := A - 50$
write (A)	write (A)
read (B)	read (B)
$B := B + 10$	$B := B + 50$
write (B)	write (B)

? What is the chronological order in which instructions of concurrent transactions are executed?

©LXD

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

Case 1

T_0	T_1
read (A)	
	read (A)
	$A := A - 50$
	write (A)
	read (B)
	$B := B + 50$
	write (B)
$A := A - 10$	
write (A)	
read (B)	
$B := B + 10$	
write (B)	

Case 2

T_0	T_1
read (A)	
$A := A - 10$	
	read (A)
	$A := A - 50$
	write (A)
	read (B)
	$B := B + 50$
	write (B)
write (A)	
read (B)	
$B := B + 10$	
write (B)	

...

Case N

T_0	T_1
read (A)	
$A := A - 10$	
write (A)	
read (B)	
$B := B + 10$	
write (B)	
	read (A)
	$A := A - 50$
	write (A)
	read (B)
	$B := B + 50$
	write (B)

©LXD **In different cases, time sequence of The CPU Instructors is different**

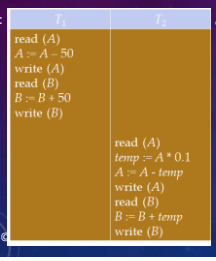
TRANSACTION SCHEDULER: SCHEDULE

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

©LXD

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

Sa1:

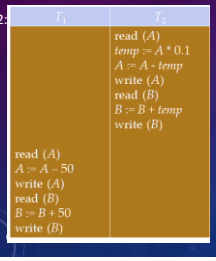


- There are two truncations:
  - T1: transfer $50 from account A to account B
  - T2: transfer A*0.1 from account A to account B
- Initial value: A=1000, B=2000
- ?result
- ? Does it satisfy the consistency?

OK

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

Sa2:

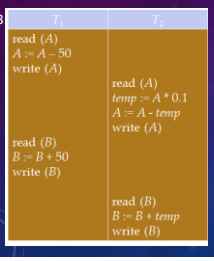


- There are two truncations:
  - T1: transfer $50 from account A to account B
  - T2: transfer A*0.1 from account A to account B
- Initial value: A=1000, B=2000
- ?result
- ? Does it satisfy the consistency?

OK

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

Sa3

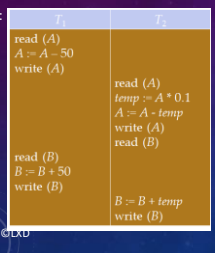


- There are two truncations:
  - T1: transfer $50 from account A to account B
  - T2: transfer A*0.1 from account A to account B
- Initial value: A=1000, B=2000
- ?result
- ? Does it satisfy the consistency?

OK

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

Sa4:



- There are two truncations:
  - T1: transfer $50 from account A to account B
  - T2: transfer A*0.1 from account A to account B
- Initial value: A=1000, B=2000
- ?result
- ? Does it satisfy the consistency?

error

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

- How to ensure the isolation with multi-transaction
  - The first solution: Insist all the related transactions run serially
- Some Drawbacks of serial executions
  - Long waiting time
    - Run these transactions slowly
  - Low throughput
    - In given amount of time, the system just do little transactions
  - Low resource utilization
    - CPU, I/O, Disk, Memory

TRANSACTION: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS?

- How to ensure the isolation with multi-transaction
  - The second solution: concurrency并发 , but conflict serializability冲突可串行化
    - Concurrent executions
    - Serializability Theory
      - ……

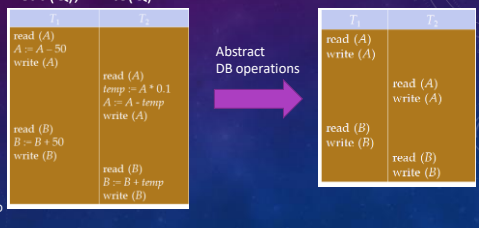
CONCURRENT EXECUTIONS并发执行

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

©LXD

TRANSACTION: SERIALIZABILITY THEORY

- Abstract database operations
  - Read(Q), Write(Q)



©LXD

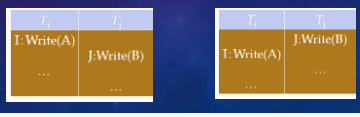
TRANSACTION: SERIALIZABILITY THEORY

- For a schedule (s) of two transactions
  - Example
    - there two consecutive instructions, I and J
    - $I \in T_i$, $J \in T_j$

©LXD

TRANSACTION: SERIALIZABILITY THEORY

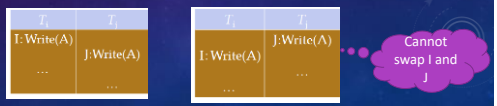
- For a schedule (s) of two transactions (cont.,)
  - Example
    - Case 1: If I and J refer to different data, then we can swap I and J without affecting the results of any instruction in the schedule s



©LXD

TRANSACTION: SERIALIZABILITY THEORY

- For a schedule (s) of two transactions (cont.,)
  - Example
    - Case 2: If I and J refer to the same data, then the order of the two steps may matter …



©LXD

TRANSACTION: SERIALIZABILITY THEORY

- For a schedule (s) of two transactions (cont.,)
  - Case 2: If I and J refer to the same data, then the order of the two steps may matter …

Only the first case, We can swap I and J , regardless of the order

(1) I=Read(A), J=Read(A)	(2) I=Read(A), J=Write(A)
(3) I=Write(A), J=Write(A)	(4) I=Write(A), J=Read(A)

- **Conflict**
  - We say that I and J conflict
  - If they are operations by different transactions on the same data item,
  - And at least one of these instructions is a write operation.

©LXD

CONFLICTING INSTRUCTIONS

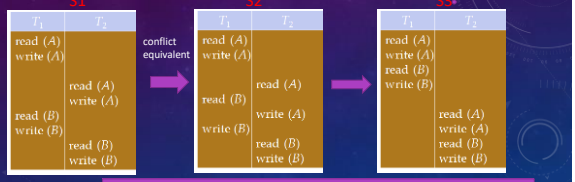
- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively. Instructions I_i and I_j conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

©LXD

TRANSACTION: SERIALIZABILITY THEORY

- **conflict serializable**

This is a **serial** schedule



- So, S1 and S2 and S3 are **conflict equivalent**
- So, S1 and S2 are both **conflict serializable**

©LXD

TRANSACTION: SERIALIZABILITY THEORY

- **conflict serializable**

- S4 and S5 are not **conflict equivalent**

©LXD

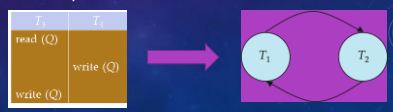
TRANSACTION: SERIALIZABILITY THEORY

- How to judge whether or not a set of transaction are conflict serializable?
  - Precedence graph(前驱图、优先图)
    - G=(V,E)
      - V: a set of vertices (transaction names)
      - E: a set of edges (conflict operation)
    - T_i -> T_j: three conditions holds
      - (1) T_i executes write(Q) before T_j executes read(Q)
      - (2) T_i executes read(Q) before T_j executes write(Q)
      - (1) T_i executes write(Q) before T_j executes write(Q)

©LXD

PRECEDENCE GRAPH

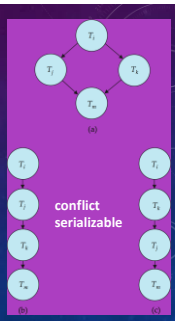
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j (T_i -> T_j) if the two transaction conflict, and T_j accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example**



©LXD

TESTING FOR CONFLICT SERIALIZABILITY

- A schedule is conflict serializable if and only if its precedence graph is acyclic非循环.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - That is, a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



©LXD

TRANSACTION: SERIALIZABILITY THEORY

- An exception to between Precedence graph and conflict serializable

S6

T_1	T_2
read (A)	
A:=A-50	
write (A)	
	read (B)
	B:=B-10
	write (B)
read (B)	
B:=B+50	
write (B)	
	read (A)
	A:=A+10
	write (A)

- S6 precedence graph is acyclic?
- Is S6 conflict equivalent to S7?
- The outcomes of S6 and S7?

view equivalence
视图等价

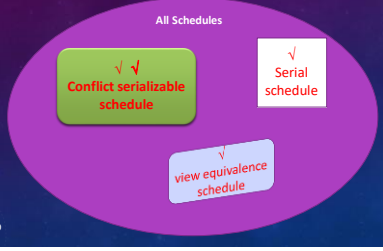
S7

T_1	T_2
read (A)	
A:=A-50	
write (A)	
read (B)	
B:=B+50	
write (B)	
	read (B)
	B:=B-10
	write (B)
	read (A)
	A:=A+10
	write (A)

SUMMARY: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS? (1/2)

- How to ensure the isolation with multi-transactions
  - The first solution:
    - Insist all the related transactions run serially
  - The second solution:
    - concurrency并发 , but conflict serializability冲突可串行化

SUMMARY: HOW TO ENSURE THE ISOLATION INVOLVED MULTI TRANSACTIONS? (2/2)



OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

MULTI-TRANSACTIONS FAILURES: - ISOLATION AND ATOMICITY

- For Multi-Transactions
  - Case 1: these transactions are mutually independent
    - If one transaction failed, just rollback this transaction
  - Case 2: these transactions are mutually dependent
    - T_j is dependent on T_i : T_j has read data written by T_i
    - If T_i transaction failed, how to rollback ?

RECOVERABLE SCHEDULES可恢复调度

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i **must** appear before the commit operation of T_j.
- The following schedule is not recoverable if T_7 commits immediately after the read(A) operation.

S8

T_6	T_7
read (A)	
write (A)	
	read (A)
	commit
read (B)	

S8 is a nonrecoverable schedule

- If T_6 should abort, T_7 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

CASCADING ROLLBACKS级联回滚

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
  - Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

S9

T_8	T_9	T_{10}
read (A)		
read (B)		
write(A)		
	read (A)	
	write(A)	
		read(A)
abort		

If T_8 fails, T_9 and T_{10} must also be rolled back.

- Can lead to the undoing of a significant amount of work

CASCADELESS SCHEDULES无级联调度

- **Cascadeless schedules** — for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

S10

T_8	T_9	T_{10}
read (A)		
read (B)		
write(A)		
abort		
	read (A)	
	write(A)	
		read(A)

- When T8 is rolled back
  - T9 and T10 don't need to be rolled back

OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

CONSISTENCY (1/2)

- 1. Strong Consistency: serializable
  - All accesses are seen by all parallel processes (or nodes, processors etc.) in the same order (sequentially)
  - Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions

CONSISTENCY (2/2)

- 2.Weak Consistency: not serializable
  - (1)All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  - (2) All other accesses may be seen in different order on different processes (or nodes, processors).
  - (3) The set of both read and write operations in between different synchronization operations is the same in each process.

WEAK LEVELS OF CONSISTENCY

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

LEVELS OF CONSISTENCY IN SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
- **SQL cmd:** set transaction isolation level serializable;

LEVELS OF CONSISTENCY

- Dirty writes脏写
  - All the isolation levels above additionally disallow dirty writes
    - That is, the disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted

OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

HOW TO PROVIDE CONCURRENCY SCHEDULES

- A database must provide a mechanism that will ensure that all possible schedules are:
  - Conflict serializable (or View serializable)
  - Recoverable
  - preferably cascadeless无级联
- (1) A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- ...

GOAL OF CONCURRENCY CONTROL

- (2) Concurrency-control schemes tradeoff
  - between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure **serializability** and provide a high degree of concurrency.

CONCURRENCY CONTROL PROTOCOLS

- Locking (*in this ppt*)
- Graph-based protocols
- Timestamps
- Validation-based protocols
- Multiple versions and snapshot isolation
- ...

LOCK-BASED PROTOCOLS

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode 排他锁.* Data item can be both read as well as written. X-lock is requested using lock-X instruction.
  2. *shared (S) mode 共享锁.* Data item can only be read. S-lock is requested using lock-S instruction.
- Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted授权.

©LXD

LOCK-BASED PROTOCOLS (CONT.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- *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
- Any number of transactions can hold shared locks on an item,
  - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- 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.

©LXD

LOCK-BASED PROTOCOLS (CONT.)

- When a transaction Ti requests a lock on a data item Q in a particular mode M
  - The concurrency-control manager grants the lock provided that:
    - (1) There is no other transaction holding a lock on Q in a mode that conflicts with M
    - (2) There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti

©LXD

EXAMPLE(A): LOCK-BASED PROTOCOLS - SOLUTION 1

T_{11}	T_{12}
read (B) $B := B - 50$ write (B) read (A) $A := A + 50$ write (A)	read (B) read (A) Display(A+B)

T_{11}	T_{12}
Lock-X(B) read (B) $B := B - 50$ write (B) Unlock(B) Lock-X(A) read (A) $A := A + 50$ write (A) Unlock(A)	Lock-S(B) read (B) Unlock(B) Lock-S(A) read (A) Unlock(A) Display(A+B)

©LXD

EXAMPLE(A): LOCK-BASED PROTOCOLS - SOLUTION 1

T_{11}	T_{12}
read (B) $B := B - 50$ write (B) read (A) $A := A + 50$ write (A)	read (B) read (A) Display(A+B)

Initial values:
A=100
B=200

error

T_{11}	T_{12}	Concurrency-control manager
Lock-x(B)		Grant-x(B,T_{11})
read (B) $B := B - 50$ write (B) Unlock(B)		
	Lock-S(B)	Grant-S(B,T_{12})
	read (B) Unlock(B) Lock-S(A)	Grant-S(A,T_{12})
	read (A) Unlock(A) Display(A+B)	
Lock-X(A)		Grant-x(A,T_{11})
read (A) $A := A + 50$ write (A) Unlock(A)		

©LXD

EXAMPLE(A): LOCK-BASED PROTOCOLS - SOLUTION 2

T_{11}	T_{12}
read (B) $B := B - 50$ write (B) read (A) $A := A + 50$ write (A)	read (B) read (A) Display(A+B)

OK!

T_{11}	T_{12}
Lock-X(B) read (B) $B := B - 50$ write (B) Lock-X(A) read (A) $A := A + 50$ write (A) Unlock(A) Unlock(B)	Lock-S(B) read (B) Lock-S(A) read (A) Display(A+B) Unlock(A) Unlock(B)

©LXD

EXAMPLE(A): LOCK-BASED PROTOCOLS - SOLUTION 2

The Two-Phase Locking Protocol

T_{11}	T_{12}
read (B)	
$B := B - 50$	
write (B)	
	read (B)
	read (A)
	Display(A+B)
read (A)	
$A := A + 50$	
write (A)	

Initial values:
A=100
B=200

OK!

T_{11}	T_{12}	Concurrency-control manager
Lock-x(B)		
		Grant-x(B,T_{11})
read (B)		
$B := B - 50$		
write (B)		
	Lock-S(B)	
		Grant-S(B,T_{12})
	read (B)	
	Lock-S(A)	
		Grant-S(A,T_{12})
	read (A)	
	Display(A+B)	
	Unlock(A)	
	Unlock(B)	
Lock-X(A)		
		Grant-x(A,T_{11})
read (A)		
$A := A + 50$		
write (A)		
Unlock(A)		
Unlock(B)		

©LXD

THE TWO-PHASE LOCKING PROTOCOL

- This protocol ensures conflict-serializable schedules.
- Phase 1: Growing Phase增长阶段
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase缩减阶段
  - Transaction may release locks
  - Transaction may not 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 (i.e., the point where a transaction acquired its final lock).

©LXD

EXAMPLE(1): THE TWO-PHASE LOCKING PROTOCOL

T_1	T_2
read (A)	read (A)
$A := A - 50$	$temp := A * 0.1$
write (A)	$A := A - temp$
read (B)	write (A)
$B := B + 50$	read (B)
write (B)	$B := B + temp$
commit	write (B)
	commit

- There are two truncations:
  - T1: transfer $50 from account A to account B
  - T2: transfer A*0.1 from account A to account B
  - Initial value: A=1000, B=2000

©LXD

EXAMPLE(1): THE TWO-PHASE LOCKING PROTOCOL

T_1	T_2
read (A)	read (A)
$A := A - 50$	$temp := A * 0.1$
write (A)	$A := A - temp$
read (B)	write (A)
$B := B + 50$	read (B)
write (B)	$B := B + temp$
commit	write (B)
	commit

T_1	T_2
Lock-x(A)	Lock-x(A)
read (A)	read (A)
$A := A - 50$	$temp := A * 0.1$
write (A)	$A := A - temp$
Lock-x(B)	write (A)
read (B)	Lock-x(B)
$B := B + 50$	read (B)
write (B)	$B := B + temp$
Unlock(B)	write (B)
Unlock(A)	Unlock(B)
	Unlock(A)

©LXD

EXAMPLE(1): THE TWO-PHASE LOCKING PROTOCOL

Sa1:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)

T_1	T_2
Lock-x(A)	
read (A)	
$A := A - 50$	
write (A)	
Lock-x(B)	
read (B)	
$B := B + 50$	
write (B)	
Unlock(B)	
Unlock(A)	
	Lock-x(A)
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	Lock-x(B)
	read (B)
	$B := B + temp$
	write (B)
	Unlock(B)
	Unlock(A)

©LXD

EXAMPLE(1): THE TWO-PHASE LOCKING PROTOCOL

Sa2:

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	

T_1	T_2
	Lock-x(A)
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	Lock-x(B)
	read (B)
	$B := B + temp$
	write (B)
	Unlock(B)
	Unlock(A)
Lock-x(A)	
read (A)	
$A := A - 50$	
write (A)	
Lock-x(B)	
read (B)	
$B := B + 50$	
write (B)	
Unlock(B)	
Unlock(A)	

©LXD

EXAMPLE(1): THE TWO-PHASE LOCKING PROTOCOL

Sa3:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
	read (B)
	$B := B + temp$
	write (B)

T_1	T_2
Lock-x(A)	
read (A)	
$A := A - 50$	
write (A)	
	Lock-x(A)
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
Lock-x(B)	
read (B)	
$B := B + 50$	
write (B)	
Unlock(B)	
Unlock(A)	
	Lock-x(B)
	read (B)
	$B := B + temp$
	write (B)
	Unlock(B)
	Unlock(A)

©LXD

EXAMPLE(1): THE TWO-PHASE LOCKING PROTOCOL

Sa4:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
read (B)	
$B := B + 50$	
write (B)	
	$B := B + temp$
	write (B)

T_1	T_2
Lock-x(A)	
read (A)	
$A := A - 50$	
write (A)	
	Lock-x(A)
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	Lock-x(B)
	read (B)
Lock-x(B)	
read (B)	
$B := B + 50$	
write (B)	
Unlock(B)	
Unlock(A)	
	$B := B + temp$
	write (B)
	Unlock(B)
	Unlock(A)

©LXD

S4 is not conflict serializable

EXAMPLE(2): THE TWO-PHASE LOCKING PROTOCOL (1/2)

T_{11}	T_{12}
read (B)	read (A)
$B := B - 50$	read (B)
write (B)	Display(A+B)
read (A)	
$A := A + 50$	
write (A)	

T_{11}	T_{12}
Lock-X(B)	Lock-S(A)
read (B)	read (A)
$B := B - 50$	Lock-S(B)
write (B)	read (B)
Lock-X(A)	Display(A+B)
read (A)	Unlock(B)
$A := A + 50$	Unlock(A)
write (A)	
Unlock(A)	
Unlock(B)	

©LXD

EXAMPLE(2): THE TWO-PHASE LOCKING PROTOCOL (2/2)

T_{11}	T_{12}
read (B)	
$B := B - 50$	
write (B)	
	read (A)
	read (B)
	Display(A+B)
read (A)	
$A := A + 50$	
write (A)	

Deadlock!

T_{11}	T_{12}	Concurrency-control manager
Lock-x(B)		
		Grant-x(B,T_{11})
read (B)		
$B := B - 50$		
write (B)		
	Lock-S(A)	
		Grant-S(A,T_{12})
	read (A)	
	Lock-S(B)	
		Grant-S(B,T_{12})
	read (B)	
	Display(A+B)	
	Unlock(B)	
	Unlock(A)	
Lock-X(A)		
		Grant-x(A,T_{11})
read (A)		
$A := A + 50$		
write (A)		
Unlock(A)		
Unlock(B)		

©LXD

EXAMPLE(3): THE TWO-PHASE LOCKING PROTOCOL

S6

T_1	T_2
read (A)	
A: A-50	
write (A)	
	read (B)
	B:=B-10
	write (B)
read (B)	
B:=B+50	
write (B)	
	read (A)
	A:=A+10
	write (A)

1. How to follow two-phase locking?
2. The result of schedule?

©LXD

THE TWO-PHASE LOCKING PROTOCOL

- Two-phase locking does not ensure freedom from deadlock

©LXD

THE TWO-PHASE LOCKING PROTOCOL

- Variant of Two-phase locking
  - Strict two-phase locking protocol
    - All exclusive-mode locks taken by a transaction be held until that transaction commits
  - Rigorous two-phase locking protocol
    - All locks be held by a transaction be held until that transaction commits

©LXD

THE TWO-PHASE LOCKING PROTOCOL

- Lock conversions转化 of Two-phase locking
  - upgrade
    - From shared mode lock to exclusive mode
  - downgrade
    - From exclusive mode lock to shared mode

©LXD

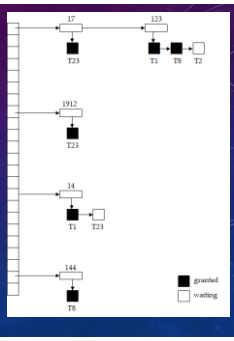
THE TWO-PHASE LOCKING PROTOCOL

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
  - Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

©LXD

IMPLEMENTATION OF LOCKING

- Lock table
  - Data items
    - I4, I7, I23, I44, I912
  - Granted locks
    - T23, T1, T8
  - Waiting locks
    - T23, T2



©LXD

OBJECTIVES

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

©LXD

TRANSACTIONS AS SQL STATEMENT

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, connection.setAutoCommit(false);

©LXD

```
#!/usr/bin/python3
import sys
import mysql.connector
config = {
    'host': 'localhost',
    'user': 'myuser',
    'password': xxx',
    'port': 3306,
    'database': 'dbsclab2018',
    'charset': 'utf8'
}
```

```
#db connect
try:
    cnn = mysql.connector.connect(**config)
except mysql.connector.Error as e:
    cnn = None
    print('connect fails!{}'.format(e))
#
if None==cnn:
  sys.exit()
#open cursor
cursor = cnn.cursor()
```

©LXD

```
try:
  cursor.execute('select name,salary from instructor01')
  for name,salary in cursor:
    print(name.decode('utf-8'),',',salary)
  #
  cursor.execute('update instructor01 set salary=salary-100 where name="Bondi"')
  cursor.execute('update instructor01 set salary=salary+100 where name="Dale"')
  cursor.execute('select name,salary from instructor01')
  for name,salary in cursor:
    print(name.decode('utf-8'),',',salary)
  #
  cnn.rollback()
```

©LXD

```
  #
    cursor.execute('select name,salary from instructor01')
    for name,salary in cursor:
        print(name.decode('utf-8'),',',salary)
except mysql.connector.Error as e:
    print('query error!{}'.format(e))
finally:
    cursor.close()
    cnn.close()
#
print('exit')
```

©LXD

SUMMARY

- Transaction Concept
- ACID Properties
- How to ensure ACID
- Multi-Transactions Failures: Isolation and Atomicity
- Multi-Transactions Isolation Levels
- How to provide concurrency schedules
- Transactions as SQL Statement

©LXD

How to get a high degree of concurrency?

Q&A?

THANKS!

leexudong@nankai.edu.cn