

Introduction to Databases

《数据库引论》



Lecture 11: Transaction Processing

第11讲：事务处理

周水庚 / Shuigeng Zhou

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

复旦大学计算机科学技术学院

Outline of the Course

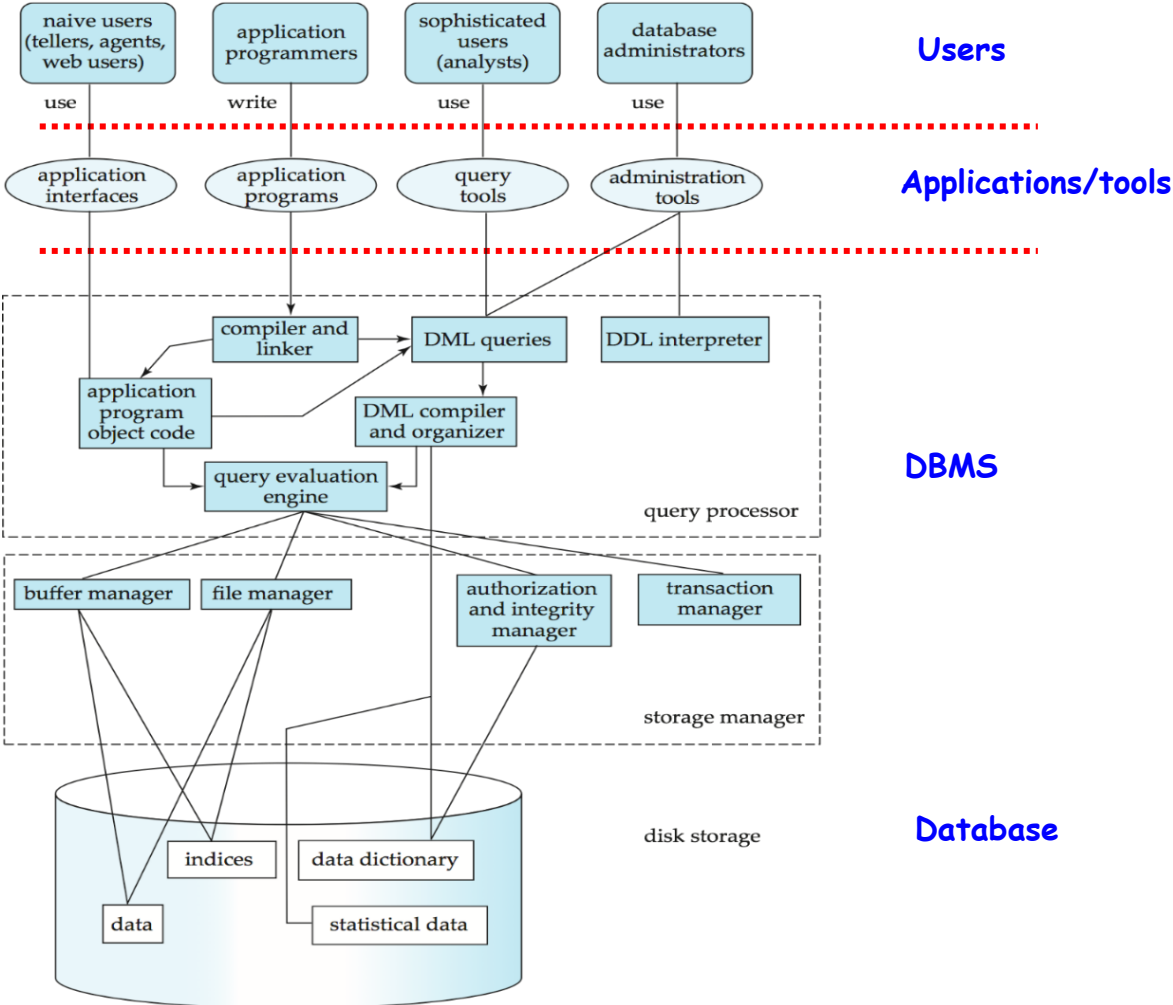
- **Part 0: Overview**
 - Lect. 1 (Feb. 29) - Ch1: Introduction
- **Part 1 Relational Databases**
 - Lect. 2 (Mar. 7) - Ch2: Relational model (data model, relational algebra)
 - Lect. 3 (Mar. 14) - Ch3: SQL (Introduction)
 - Lect. 4 (Mar. 21) - Ch4/5: Intermediate and Advanced SQL
- **Part 2 Database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

Midterm exam: Apr. 25

- 13: 00-15: 00, H3109

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

Database System Structure



Users

Applications/tools

DBMS

Database

Outline

☞ Transaction Concept

- Schedules
- Serializable Schedule
- Recoverable Schedule
- Testing for Serializability

Transaction Concept

- A transaction (事务) is a unit of program execution consisting of multiple operations
 - During transaction execution, the database may be **inconsistent**
 - After the **transaction is committed**, the database **must be consistent**
- **Two main issues**
 - **Concurrent execution of multiple transactions**
 - **Hardware failures and system crashes**

ACID Properties

- **Atomicity (原子性)**
 - Either all operations of the transaction are properly reflected in the database or none are
- **Consistency (一致性)**
 - Execution of a transaction in isolation preserves the consistency of the database
- **Isolation (隔离性)**
 - Although multiple transactions may execute concurrently, each transaction must be unaware of other transactions
- **Durability (持久性)**
 - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Example of Fund Transfer

- A transaction to transfer \$50 from account A to account B:
 1. read(A)
 2. $A := A - 50$
 3. write(A)
 4. read(B)
 5. $B := B + 50$
 6. write(B)
- **Consistency requirement**
 - The sum of A and B is unchanged by the execution of the transaction
- **Atomicity requirement**
 - If the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database. Otherwise, an inconsistency will occur

Example of Fund Transfer (Cont.)

- **Durability requirement**

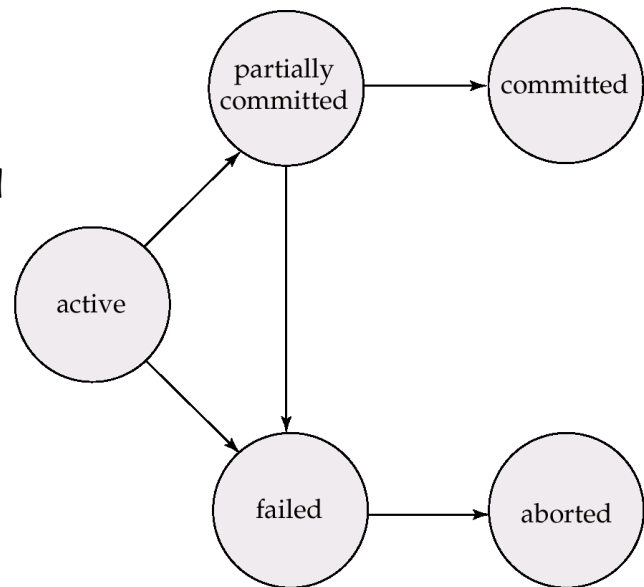
- Once the user was notified that the transaction has completed, the **updates** to the database by the transaction **must persist** despite failures

- **Isolation requirement**

- If between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database
- **Can be ensured trivially by running transactions serially**, i.e., one after the other. However, **executing multiple transactions concurrently** has significant benefits

Transaction State

- **Active(活跃)**
 - The initial state. The transaction stays in this state while it is executing
- **Partially committed(部分提交)**
 - After the final statement has been executed
- **Failed(失败)**
 - After discovering that normal execution can no longer proceed
- **Aborted(夭折、中止)**
 - After the transaction has been **rolled back (回滚)** and the database restored to its state prior to the start of the transaction
 - **Restart the transaction** - only if no internal logical error happens in the transaction
 - **Kill the transaction** - problems arising with the transaction, input data, no desirable data found in the database
- **Committed(提交)**
 - After successful completion



Outline

- Transaction Concept
- ☞ **Schedules**
- Serializable Schedule
- Recoverable Schedule
- Testing for Serializability

Concurrent Executions

- **Concurrent execution**
 - Multiple transactions are allowed to run concurrently in the system
 - **Advantages**
 - Increase processor and disk utilization
 - Reduce average response time
- **Concurrency control**
 - **Mechanisms to achieve isolation**, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule (调度)**

- sequences that indicate 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.

- **Example**

- Let T_1 transfer **\$50** from A to B , and T_2 transfer **10%** of the balance from A to B
- **Schedule 1** is a **serial schedule (串行调度)**, in which T_1 is followed by T_2

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)

Schedule 1

Example Schedule (Cont.)

- Another **serial schedule** where T_2 is followed by T_1

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)

Schedule 1



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)

Schedule 2

Example Schedule (Cont.)

- **Non-serial schedule**

- Let T_1 and T_2 be the transactions defined previously
- **Schedule 3** is not a serial schedule, but it is **equivalent to Schedule 1**
 - $A' = (A - 50) - (A - 50) * 0.1 = (A - 50) * 0.9$
 - $B' = B + 50 + (A - 50) * 0.1$
 - **$A' + B' = A + B$**

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)

Schedule 3

Example Schedule (Cont.)

- The following concurrent schedule does not preserve the value of the sum $A + B$.

- $A' = A - 50$
- $B' = B + A * 0.1$
- $A' + B' = A + B - 50 + A * 0.1 \neq A + B$

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

Schedule 4

Outline

- Transaction Concept
- Schedules
- ➡ **Serializable Schedule**
- Recoverable Schedule
- Testing for Serializability

Serializability (可串行化)

- **Assumption**
 - Each transaction preserves database consistency, thus **serial execution** of a set of transactions preserves database **consistency**
- **Serializability**
 - **A schedule is serializable if it is equivalent to a serial schedule**
 - **Conflict serializability (冲突可串行性)**
 - **View serializability (视图可串行性)**
- **Note**
 - We ignore operations other than read and write instructions. Our simplified schedules consist of **only read and write instructions**

Conflict Serializability

- **Conflict**

- Given instructions I_i and I_j of transactions T_i and T_j respectively, conflict occurs **iff** there exists some item Q accessed by both I_i and I_j , and **at least one of these instructions write Q**

- **Four cases**

- $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. I_i and I_j **(no conflict)**

- $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. **(conflict)**

- $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. **(conflict)**

- $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. **(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**

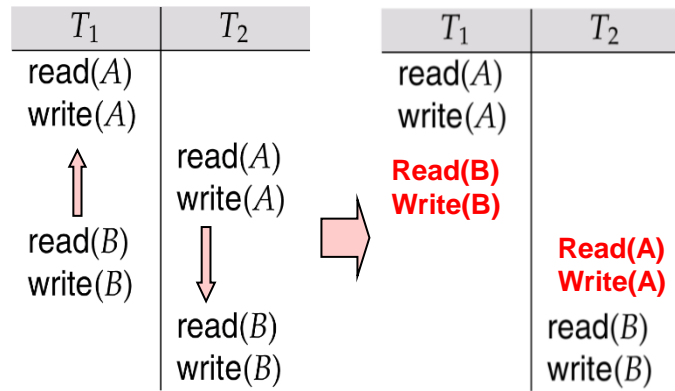
Conflict Serializability (Cont.)

- **Conflict equivalent**
 - If a schedule S can be transformed into a schedule S' by a series of **swaps of non-conflicting instructions**, we say that S and S' are **conflict equivalent**
 - We say that a schedule S is **conflict serializable** if it is **conflict equivalent to a serial schedule**
- **Example** of a schedule that is not conflict serializable
 - We are **unable to swap instructions** in the following schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.

T_3	T_4
read(Q)	
write(Q)	write(Q)

Conflict Serializability (Cont.)

- **Schedule 1** can be transformed into **Schedule 2**, a serial schedule where T_2 follows T_1 , by a series of swaps of non-conflicting instructions
- Therefore, **Schedule 1** is **conflict serializable**



Schedule 1

Schedule 2

Conflict Serializability (Cont.)

- Example

$Sc_1 = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

- Swap $w_2(A)$ and $r_1(B)w_1(B)$, then we have

$r_1(A)w_1(A)r_2(A)r_1(B)w_1(B)w_2(A)r_2(B)w_2(B)$

- Swap $r_2(A)$ and $r_1(B)w_1(B)$, then:

$Sc_2 = r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

- Sc_2 is equivalent to a serializable schedule T_1, T_2
- Then Sc_1 is conflict serializable

Conflict Serializability (Cont.)

- A conflict serializable schedule is a serializable schedule, but a serializable schedule is not always conflict serializable.

- E.g., three transactions

$T1=W1(Y)W1(X)$, $T2=W2(Y)W2(X)$, $T3=W3(X)$

■ $L1=W1(Y)W1(X)W2(Y)W2(X)W3(X)$ is serializable

■ $L2=W1(Y)W2(Y)W2(X)W1(X)W3(X)$ is not equivalent to $L1$, and not conflict serializable.


■ $L2$ is serializable, the result of the schedule is equivalent to $L1$ (final write X is from $T3$, final write Y is from $T2$)

View Serializability

- **S and S' are view equivalent** if the following three conditions are met:
 - For each data item Q , if transaction T_i **reads the initial value** of Q in schedule S , then transaction T_i must, in schedule S' , **also read the initial value of Q** .
 - For each data item Q , if transaction T_i executes **read(Q)** in schedule S , and that value **was produced by transaction T_j** (if any), then transaction T_i must in schedule S' **also read the value of Q that was produced by transaction T_j** .
 - For each data item Q , **the transaction** (if any) that **performs the final write(Q)** operation in schedule S must **perform the final write(Q)** operation in schedule S'
- As can be seen, **view equivalence** is also based purely on **reads and writes alone**.

View Serializability (Cont.)

- If a schedule **S** is **view serializable**, it is **view equivalent** to a serial schedule.
- **Every conflict serializable schedule is also view serializable.**
- **A schedule which is view-serializable but not conflict serializable.**
Equivalent to T_3, T_4, T_6

T_3	T_4	T_6		T_3	T_4	T_6
read(Q)				read(Q)		
write(Q)	write(Q)			Write(Q)	Write(Q)	
		write(Q)				write(Q)

- **Every view serializable schedule that is not conflict serializable has blind writes - write without read**


Other Notions of Serializability

- The following schedule produces **the same outcome** as the **serial schedule $\langle T_1, T_5 \rangle$** , yet it is **not conflict equivalent or view equivalent**

T_1	T_5
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)

- Determining such equivalence requires analysis of operations other than read and write.

Outline

- Transaction Concept
- Schedules
- Serializable Schedule
-  **Recoverable Schedule**
- Testing for Serializability

Recoverability (可恢复性)

- Recoverable schedule (可恢复调度)

- If a transaction T_j reads a data items previously written by a transaction T_i , the commit operation of T_i appears before the commit operation of T_j
- The following schedule is **not recoverable** if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Recoverability (Cont.)

- **Cascading rollback(级联回滚)**

- A single transaction failure leads to a series of transaction rollbacks
- Consider the following schedule where none of the transactions has yet committed

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back
- Can lead to the **undoing of a significant amount of work**

Recoverability (Cont.)

- **Cascadeless schedules (无级联回滚调度)**
 - 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
 - **Cascading rollbacks cannot occur** and **every cascadeless schedule is also recoverable**
 - It is **desirable to restrict** the schedules to those that are **cascadeless**

Transaction Definition in SQL

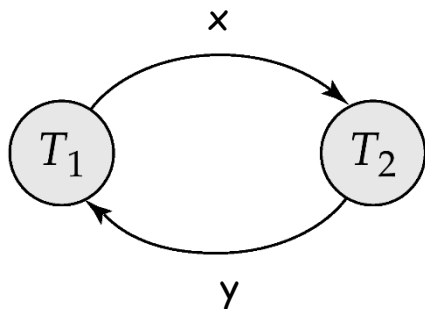
- DML must include a construct for specifying the set of actions that comprise a transaction
- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
 - **Commit** work: commits current transaction and begins a new one.
 - **Rollback** work: causes current transaction to abort.
- Levels of **isolation** specified by SQL-92
 - **Serializable - default**: 保证可串行化调度
 - **Repeatable read**: 只允许读取已提交数据, 两次读取之间数据不能更新
 - **Read committed**: 只允许读取已提交数据, 不要求可重复读
 - **Read uncommitted**: 允许读取未提交数据

Outline

- Transaction Concept
- Schedules
- Serializable Schedule
- Recoverable Schedule
- ☞ **Testing for Serializability**

Testing for Serializability

- Given a set of transactions T_1, T_2, \dots, T_n
- **Precedence graph (优先图)**
 - A **direct graph** where the **vertices** are the **transactions**
 - We draw an **arc** from T_i to T_j if the two transactions **conflict**, and T_i accessed the data item on which the conflict arose earlier.
 - We label the arc by the data item that was accessed
- **Example**

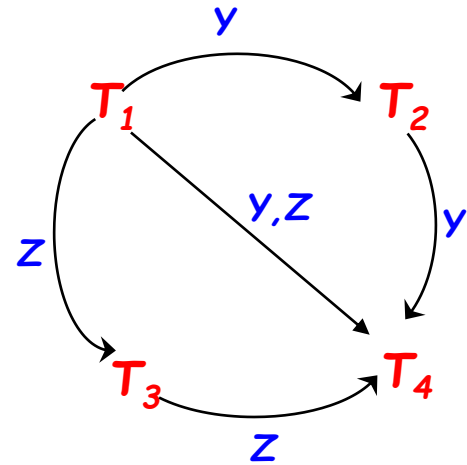


T1 write(x) **before** **T2** read(x)
T1 write(x) **before** **T2** write(x)
T1 read(x) **before** **T2** write(x)

T2 write(y) **before** **T1** read(y)
T2 write(y) **before** **T1** write(y)
T2 read(y) **before** **T1** write(y)

Example Schedule A

T_1	T_2	T_3	T_4	T_5
read(Y) read(Z)	read(X)			read(V) read(W) read(W)
read(U)	read(Y) write(Y)	write(Z)	read(Y) write(Y) read(Z) write(Z)	
read(U) write(U)				



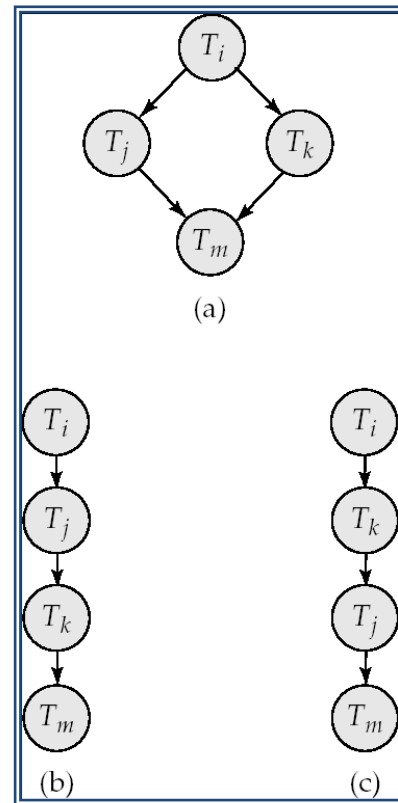
$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$

$T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4 \rightarrow T_5$

.....???

Test for Conflict Serializability

- A schedule is **conflict serializable** if and only if its **precedence graph** is **acyclic** (无环)
- If precedence graph is acyclic, the **serializability order** can be obtained by a **topological sorting of the graph**.
 - For example, a serializability order for **Schedule A** in last slide would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
 - Any others?



Test for View Serializability

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability
- The problem of checking if a schedule is view serializable falls in the **class of NP-complete problems**.
 - Thus existence of an efficient algorithm is unlikely
 - However practical algorithms that just check some *sufficient conditions* for view serializability can still be used

Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability after it has executed is **too late**
- **Goal** - to develop **concurrency control protocols** that will **assure serializability**.
 - They will generally **not examine the precedence graph** as it is being created
 - Instead **a protocol** will impose a discipline that **avoids non-serializable schedules**
- Tests for serializability help understand **why a concurrency control protocol is correct**

Assignments

- Practice Exercises: 17.6
- Exercises: 17.15
- Submission DDL: 12:59pm, June 5, 2024

End of Lecture 11