# 21121350
# Database System

## Lecture12: Transactions

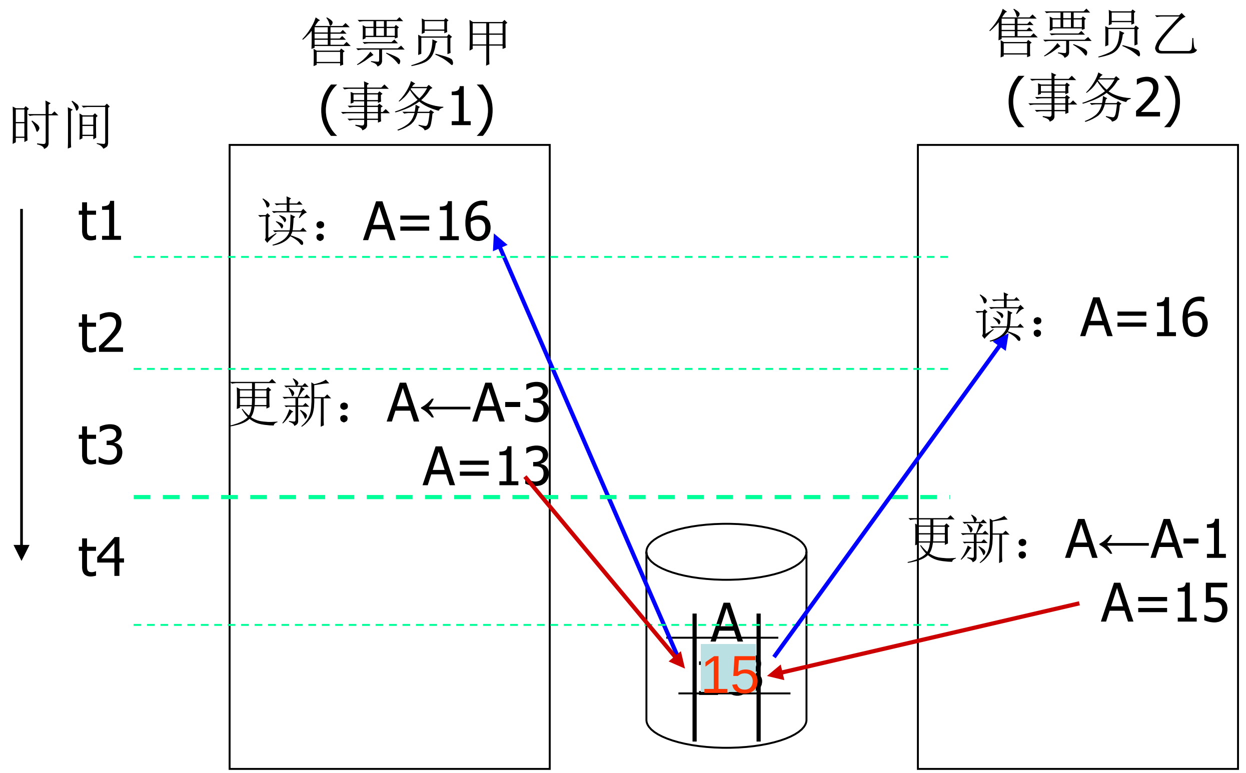
Lu Chen (陈璐)
College of Computer Science
Zhejiang University
Spring & Summer 2023
lucchen@zju.edu.cn/18868818726

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability

# Transaction Concept

- For a DBMS, two main issues must be dealt with:
  - Concurrent executions of multiple users or multi-programs.
  - Failures of various kinds, such as hardware failures and system crashes.


售票员甲
(事务1)
售票员乙
(事务2)
时间
t1
t2
t3
t4
读：A=16
读：A=16
更新：A←A-3
A=13
更新：A←A-1
A=15
A
15

并发操作丢失修改数据

# Transaction Concept (cont.)

- How to keep the database correctness, consistency, and integrity when it concurrently executes?

  Transaction --- Proposed by Jim Gray

- A transaction is a unit of program execution that accesses and possibly updates various data items.
  - Usually a transaction is of a number of SQL statements, ended with commit（提交） or rollback（回滚）statement.
  - A transaction must see a consistent database.

- During transaction execution the database may be inconsistent, but when the transaction is committed, the database must be consistent.

# 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(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 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(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

- 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*)

- Atomicity requirement
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- Durability requirement — Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

# Example of Fund Transfer (Cont.)

- 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 in the above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - E.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

# Example of Fund Transfer (Cont.)

- Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

| T1 | T2 |
|---|---|
| 1. read($A$) | |
| 2. $A := A - 50$ | |
| 3. write($A$) | |
| | read(A), read(B), print(A+B) |
| 4. read($B$) | |
| 5. $B := B + 50$ | |
| 6. write($B$) | |

- Isolation can be ensured trivially by running transactions serially
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

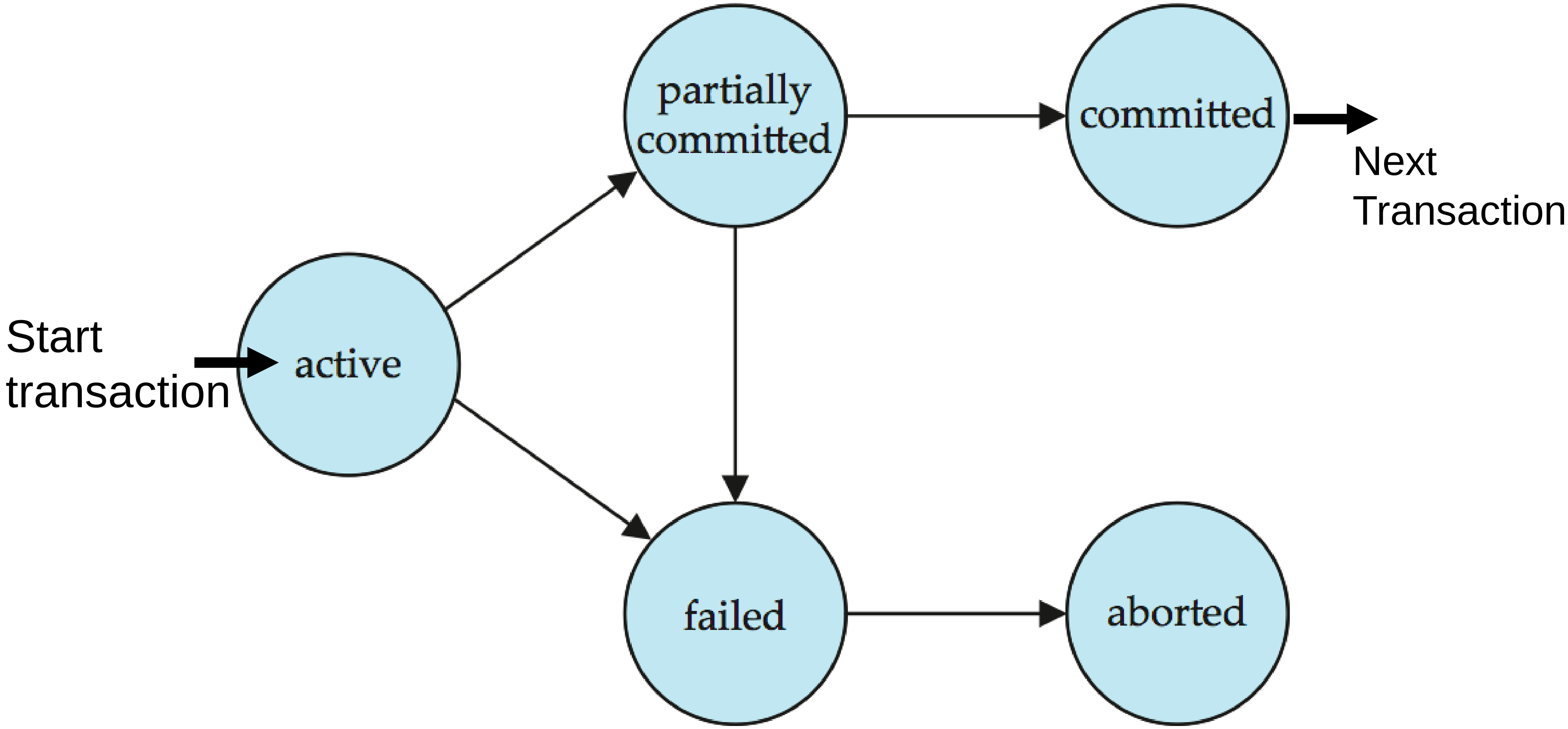
# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# Transaction State

- Active – the initial state; the transaction stays in this state while it is executing.
- Partially committed – after the final statement has been executed.
（此时要输出的结果数据可能还在内存buffer中）
- 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.

# Transaction State (Cont.)



# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

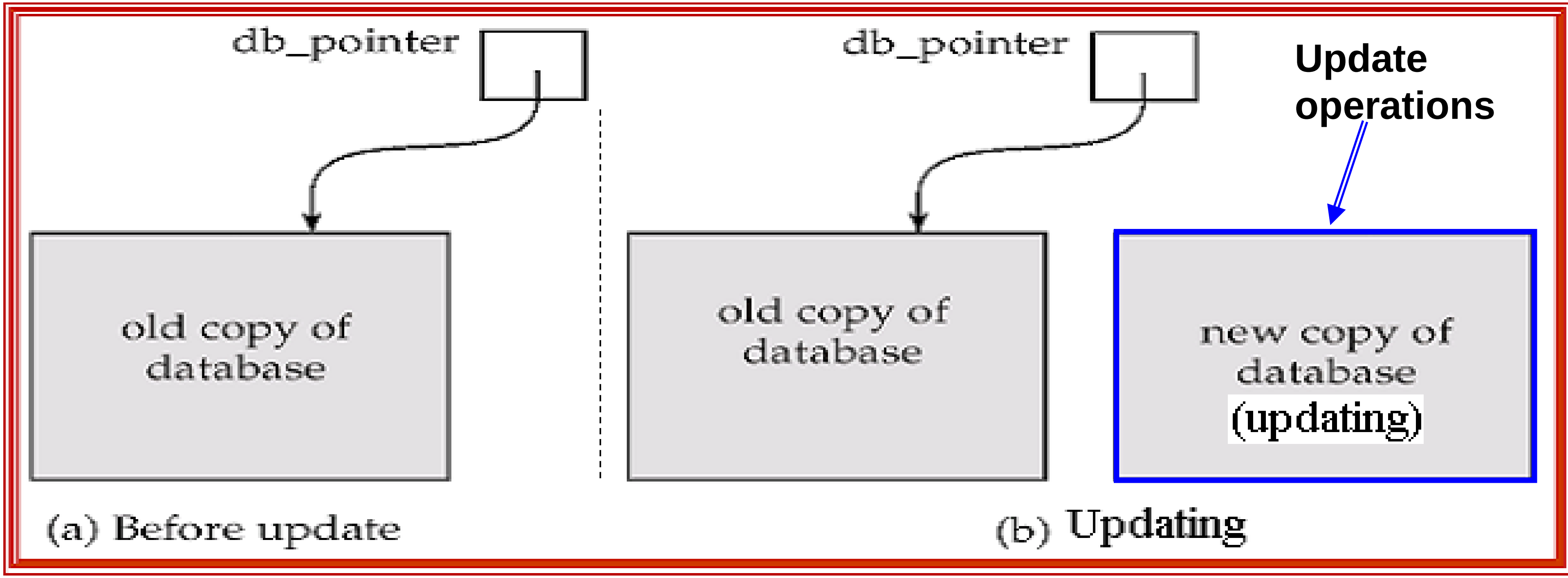
# Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.

- The *shadow-database* scheme---a simple but inefficient scheme: a pointer called db_pointer always points to the current consistent copy of the database.
  - All updates are made on a newly created copy of the database. The original copy---*shadow copy* (影子拷贝) is untouched---pointed by db_pointer
    - If aborted: merely delete the new copy.
    - If commit:
      - 1) Put all pages in memory of the new copy to disk(in unix, *flush* command).
      - 2) db_pointer is changed to point to the new copy---become the current copy and at the same time the old copy is then deleted.

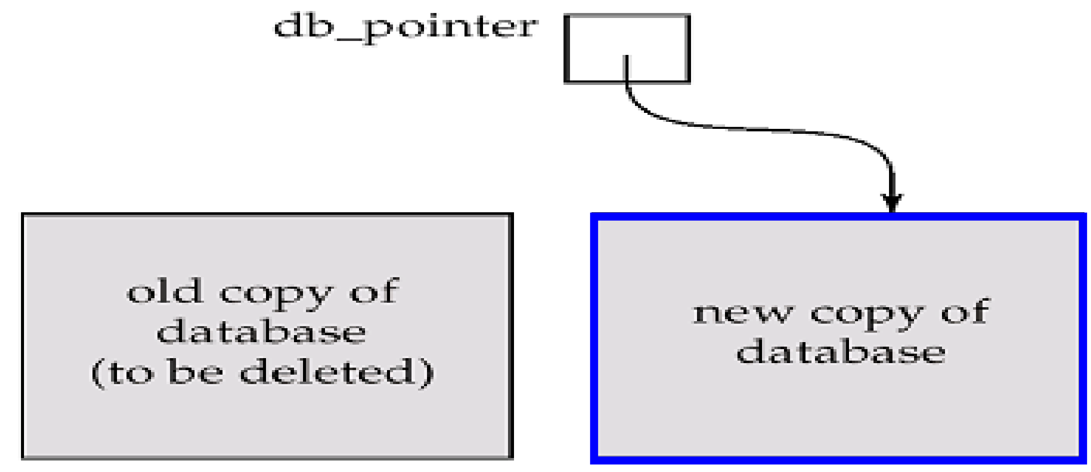
# Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



# Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



(c) After update

- Required: update db-pointer atomically (which is guaranteed by disk system, store it in a single sector); no concurrent transactions; assumes disks not to fail.
- The idea is useful for text editors.
- But extremely inefficient for large databases: executing a single transaction requires copying the *entire* database.
  - Will see better schemes in Chapter 20.

    And using Rollback Segment in Oracle

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the database system.
- Advantages of concurrent executions are:
  - Increased processor and disk utilization, leading to better transaction *throughput*(吞吐量)*:* 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.
- Problems: concurrency may destroy consistency, despite the correctness of each individual transaction. (如并发售票问题)
- Concurrency control schemes – 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.---- The important job of DBMS
  - Here we study the notion of correctness of concurrent executions, serializability, recoverability, etc.
  - Will study concurrency control in Chapter 18

# Schedules

Example: two transactions T1 and T2 are running concurrently, which transfer funds between account A and B.

Because CPU can only execute instructions serially, we should……

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

- **Schedules** – 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 the transactions
  - Must preserve the order in which the instructions appear in each individual transaction.

# Example Schedules ---- Serial schedule(串行调度)

Before transaction:
A=1000，B=2000，
A+B=3000

**Schedule 1**

| | $T_1$ | $T_2$ | |
|---|---|---|---|
| **950** | read ($A$) | | |
| | $A := A - 50$ | | |
| | write ($A$) | | |
| **2050** | read ($B$) | | |
| | $B := B + 50$ | | |
| | write ($B$) | | |
| | commit | | |
| | | read ($A$) | **950** |
| | | $temp := A * 0.1$ | |
| | | $A := A - temp$ | |
| | | write ($A$) | **855** |
| | | read ($B$) | |
| | | $B := B + temp$ | |
| | | write ($B$) | **2145** |
| | | commit | |

A=855，B=2145，A+B=3000

**Schedule 2**

| $T_1$ | $T_2$ | |
|---|---|---|
| | read ($A$) | |
| | $temp := A * 0.1$ | |
| | $A := A - temp$ | |
| | write ($A$) | **900** |
| | read ($B$) | |
| | $B := B + temp$ | |
| | write ($B$) | **2100** |
| | commit | |
| read ($A$) | | |
| $A := A - 50$ | | |
| write ($A$) | | |
| read ($B$) | | |
| $B := B + 50$ | | |
| write ($B$) | | |
| commit | | |

A=850，B=2150，A+B=3000

- Both schedule1 and schedule2 are serial（串行的）
  - N个并行事务有n!种可选择的串行调度 (如<t1,t2,t3,…>, <t2,t1,t3,…>,<t3,t1,t2,…>, …)
- 串行的调度必能保持一致性， 但低效。

# Example Schedule (Cont.) ---- concurrent schedule

**Schedule 3**

| | $T_1$ | $T_2$ | |
|---|---|---|---|
| | read ($A$) | | |
| | $A := A - 50$ | | |
| **950** | write ($A$) | | |
| | | read ($A$) | **950** |
| | | $temp := A * 0.1$ | |
| | | $A := A - temp$ | |
| | | write ($A$) | **855** |
| | read ($B$) | | |
| | $B := B + 50$ | | |
| **2050** | write ($B$) | | |
| | commit | | |
| | | read ($B$) | |
| | | $B := B + temp$ | |
| | | write ($B$) | **2050+95 =2145** |
| | | commit | |

A=855, B=2145, A+B=3000

**Schedule 4**

| | $T_1$ | $T_2$ | |
|---|---|---|---|
| | read ($A$) | | |
| | $A := A - 50$ | | |
| | | read ($A$) | **A=1000** |
| | | $temp := A * 0.1$ | |
| | | $A := A - temp$ | |
| | | write ($A$) | **A=900** |
| | | read ($B$) | |
| **A=950** | write ($A$) | | |
| | read ($B$) | | **X** |
| | $B := B + 50$ | | |
| **B=2050** | write ($B$) | | |
| | commit | | |
| | | $B := B + temp$ | |
| | | write ($B$) | **B=2000+100** |
| | | commit | |

A=950, B=2100, A+B=3050

- In both Schedule 1, 2, and 3, the sum A + B is preserved.--- consistency
- But Schedule 4 does not preserve the value of the the sum A + B. --- not consistent. --- a bad schedule!

**How to judge ?**
**How to schedule**

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. Conflict serializability
  2. View serializability

# Simplified view of transactions

- We ignore operations other than read and write instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only read and write instructions.

# Conflicting Instructions

- Instructions $l_i$ and $l_j$ of transactions $T_i$ and $T_j$ respectively, **conflict** if and only if there exists some item $Q$ accessed by both $l_i$ and $l_j$, and at least one of these instructions wrote $Q$.
  1. $l_i$ = **read**($Q$), $l_j$ = **read**($Q$). $l_i$ and $l_j$ don't conflict.
  2. $l_i$ = **read**($Q$), $l_j$ = **write**($Q$). They conflict.
  3. $l_i$ = **write**($Q$), $l_j$ = **read**($Q$). They conflict
  4. $l_i$ = **write**($Q$), $l_j$ = **write**($Q$). They conflict
- Intuitively, a conflict between $l_i$ and $l_j$ forces a (logical) temporal order between them.
  - If $l_i$ and $l_j$ 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.

★若2个操作是有冲突的，则二者执行次序不可交换。
若2个操作不冲突，则可以交换次序。

# Conflict Serializability

- 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.

# Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where $T_2$ follows $T_1$, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

| $T_1$ | $T_2$ |
|---|---|
| read ($A$) | |
| write ($A$) | |
| | read ($A$) |
| | write ($A$) |
| read ($B$) | |
| write ($B$) | |
| | read ($B$) |
| | write ($B$) |

Schedule 3

| $T_1$ | $T_2$ |
|---|---|
| read ($A$) | |
| write ($A$) | |
| read ($B$) | |
| write ($B$) | |
| | read ($A$) |
| | write ($A$) |
| | read ($B$) |
| | write ($B$) |

Schedule 6

# Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

| $T_3$ | $T_4$ |
|---|---|
| read ($Q$) | |
| | write ($Q$) |
| write ($Q$) | |

- We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

# *View Serializability

- Let $S$ and $S'$ be two schedules with the same set of transactions. $S$ and $S'$ are view equivalent if the following three conditions are met, for each data item $Q$,

首读 1. If in schedule S, transaction $T_i$ reads the initial value of $Q$, then in schedule $S'$ also transaction $T_i$ must read the initial value of $Q$.

写读 2. If in schedule S transaction $T_i$ executes read($Q$), and that value was produced by transaction $T_j$ (if any), then in schedule $S'$ also transaction $T_i$ must read the value of $Q$ that was produced by the same write(Q) operation of transaction $T_j$ .

末读 3. The transaction (if any) that performs the final write($Q$) operation in schedule $S$ must also 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.)

- A schedule $S$ is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

| $T_{27}$ | $T_{28}$ | $T_{29}$ |
|---|---|---|
| read ($Q$) | | |
| | write ($Q$) | |
| write ($Q$) | | |
| | | write ($Q$) |

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

# Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule < $T_1$, $T_5$ >, yet is not conflict equivalent or view equivalent to it.

| $T_1$ | $T_5$ |
|---|---|
| read ($A$)<br>$A := A - 50$<br>write ($A$) | |
| | read ($B$)<br>$B := B - 10$<br>write ($B$) |
| read ($B$)<br>$B := B + 50$<br>write ($B$) | |
| | read ($A$)<br>$A := A + 10$<br>write ($A$) |

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

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- Recoverable schedule — if a transaction $T_j$ reads a data item previously written by a transaction $T_i$ , then the commit operation of $T_i$ appears before the commit operation of $T_j$.
- The following schedule (Schedule 11) is not recoverable if $T_9$ commits immediately after the read.

| $T_8$ | $T_9$ |
|---|---|
| read ($A$) | |
| write ($A$) | |
| | read ($A$) |
| | commit |
| read ($B$) | |

- If $T_8$ should abort, $T_9$ would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

# Cascading Rollbacks

- 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)

| $T_{10}$ | $T_{11}$ | $T_{12}$ |
|---|---|---|
| read ($A$) | | |
| read ($B$) | | |
| write ($A$) | | |
| | read ($A$) | |
| | write ($A$) | |
| | | read ($A$) |
| abort | | |

  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

# Cascadeless Schedules

- Cascadeless schedules — cascading rollbacks cannot occur; 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.

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# Implementation of Isolation

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur. ---- Chpt18
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# Transaction Definition in SQL

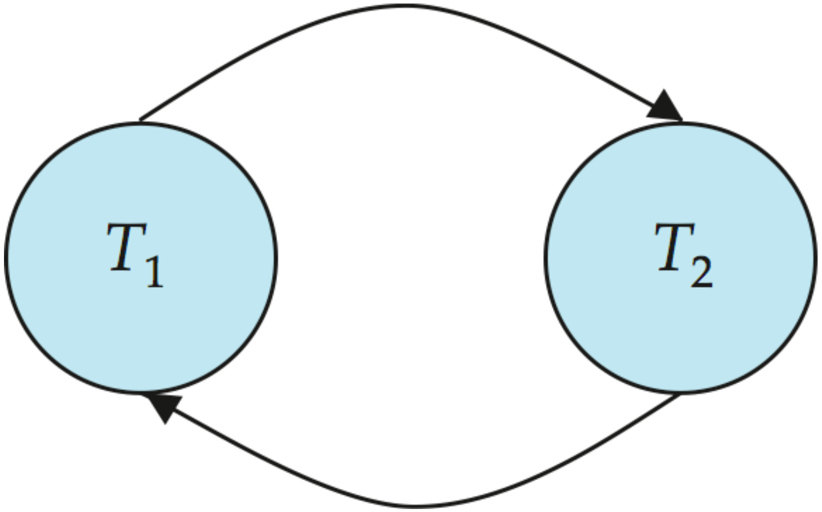
- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - Commit work commits current transaction and begins a new one.
  - Rollback work causes current transaction 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)

# Outline

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

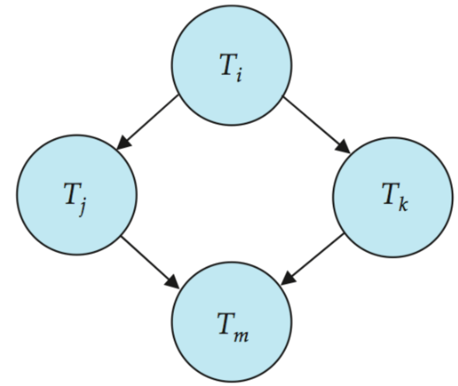
# Testing for Serializability

- 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 $T_j$ if the two transaction conflict, and $T_i$ accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example 1

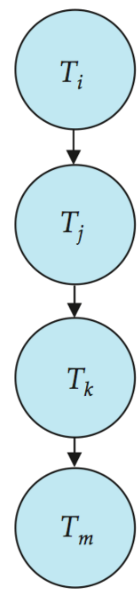


# Test for Conflict Serializability

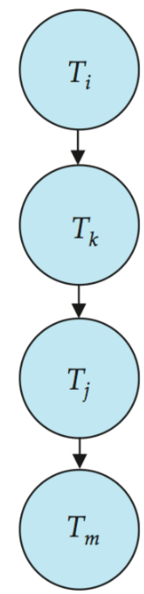
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order $n^2$ time, where $n$ is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where $e$ is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be
    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



(a)



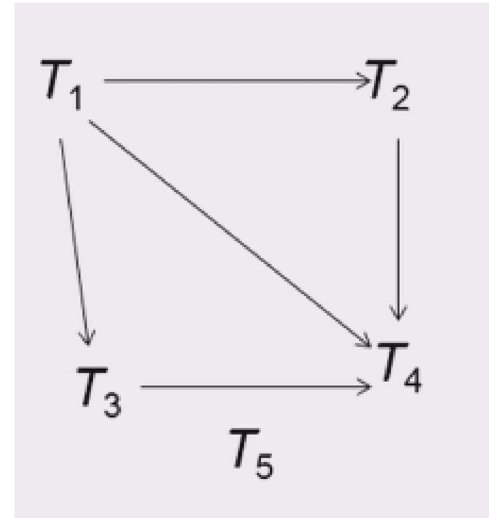
(b)

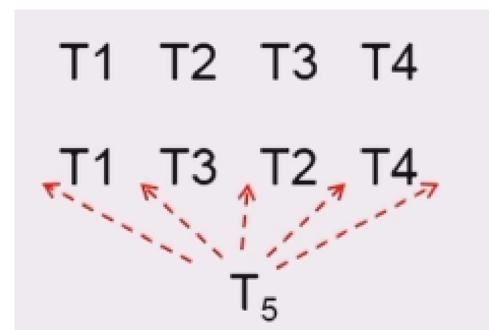


(c)

# Test for Conflict Serializability

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



# *Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- 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 *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

# Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 18.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

# 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.
  - Successive reads of record may return different (but committed) values.
- Read uncommitted — even uncommitted records may be read.
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

# Q & A

Your questions and suggestions are expected for me.

Thanks a lot!

Questions?

Exercises: 17.6, 17.7, 17.12 (see Pages 831-832)