



# Module 17: Transactions

**Database System Concepts, 7<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

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



# Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., 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)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions



# 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.)

- **Consistency requirement** in 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**

1. **read**(A)
2.  $A := A - 50$
3. **write**(A)
4. **read**(B)
5.  $B := B + 50$
6. **write**(B)

**T2**

read(A), read(B), print(A+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.



# 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.



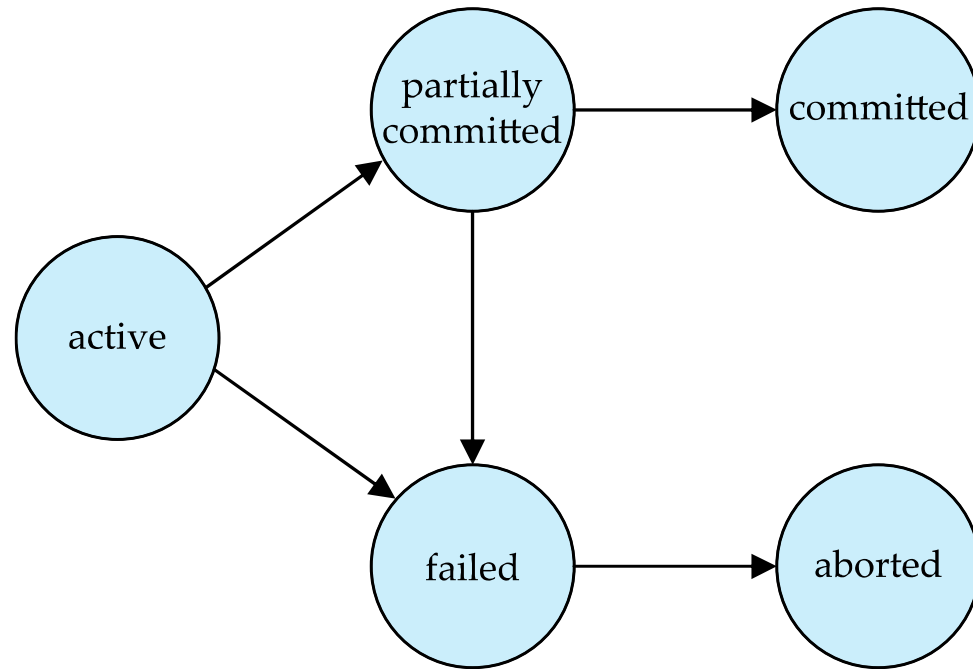
# Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **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.)





# Concurrent Executions

- 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
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.



# Schedules

- **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



# Schedule 1

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- A **serial** schedule in which  $T_1$  is followed by  $T_2$  :

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



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



## Schedule 3

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

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

- In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.



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



# 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  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **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.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . 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.



# 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)<br>write (A) |                       |
|                       | read (A)<br>write (A) |
| read (B)<br>write (B) |                       |
|                       | read (B)<br>write (B) |

Schedule 3

| $T_1$  | $T_2$  |
|--|--|
| read (A)<br>write (A)<br>read (B)<br>write (B) |  |
|  | read (A)<br>write (A)<br>read (B)<br>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  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .



# 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  $Q$ ,
  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  $Q$ .
  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  $T_j$ .
  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.

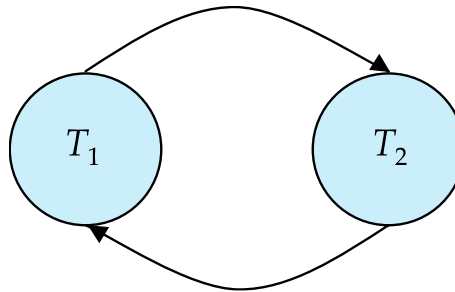
| $T_{27}$      | $T_{28}$      | $T_{29}$      |
|---------------|---------------|---------------|
| read ( $Q$ )  | write ( $Q$ ) |               |
| write ( $Q$ ) |               | write ( $Q$ ) |

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



# Testing for Serializability

- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **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 of a precedence graph







# 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.



# 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

| $T_8$                 | $T_9$              |
|-----------------------|--------------------|
| read (A)<br>write (A) |                    |
|                       | read (A)<br>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)<br>read (B)<br>write (A) | read (A)<br>write (A) |          |
| abort                             |                       | 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



# 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



# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.



# Concurrency Control (Cont.)

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



# 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 non-serializable schedules.
  - We study such protocols in Chapter 16.
- 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.



# End of Chapter 17