



Chapter 9: Transactions

modified from:

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Transaction Concept

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



Example of Fund Transfer

- n 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)
- n **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
- n **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.)

- n 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)
- n **Consistency requirement** in above example:
 - | the sum of A and B is unchanged by the execution of the transaction
- n 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.)

- n **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)

- n Isolation can be ensured trivially by running transactions **serially**
 - | that is, one after the other.
- n 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:

- n **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- n **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- n **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.
- n **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.



Transaction State

- n **Active** – the initial state; the transaction stays in this state while it is executing
- n **Partially committed** – after the final statement has been executed.
- n **Failed** -- after the discovery that normal execution can no longer proceed.
- n **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
- n **Committed** – after successful completion.



Transaction Model

n Operations

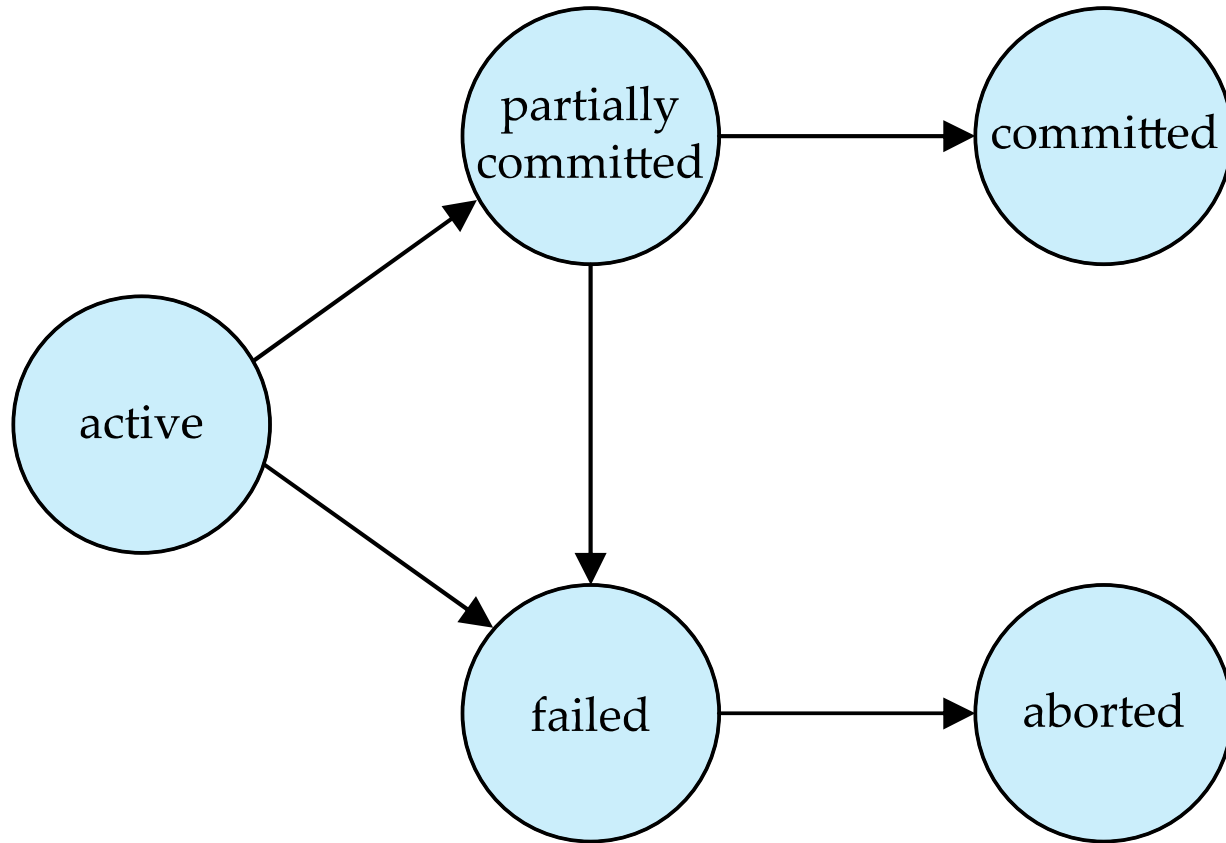
- | Read(A) – read value of data item A
- | Write(A) – write a new value of data item A
- | Commit – commit changes of the transaction
- | Abort – Revert changes made by the transaction

n Data Items

- | Objects in the data base
- | Usually we consider tuples (rows) or disk pages



Transaction State (Cont.)





Concurrent Executions

- n 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
 - ▶ In multi-processor systems each statement can use one or more CPUs
 - | **reduced average response time** for transactions: short transactions need not wait behind long ones.
- n **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



Schedules

- n **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.
- n 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
- n A transaction that fails to successfully complete its execution will have an abort instruction as the last statement



Schedule 1

- n Let T_1 transfer \$50 from A to B , and T_2 transfer 10% of the balance from A to B .
- n A **serial** schedule 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) commit	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit



Schedule 2

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



Schedule 3

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

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



Schedule 4

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

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



Serializability

- n **Basic Assumption** – Each transaction preserves database consistency.
- n Thus serial execution of a set of transactions preserves database consistency.
- n 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

- n 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
- n 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

- n 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**.
 - | That is the order of each pair of conflicting operations in S and S' is the same
- n We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule



Conflict Serializability (Cont.)

- n Example of a schedule that is not conflict serializable:

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

- n 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

- n 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.)

- n A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- n Every conflict serializable schedule is also view serializable.
- n 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)

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



Other Notions of Serializability

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

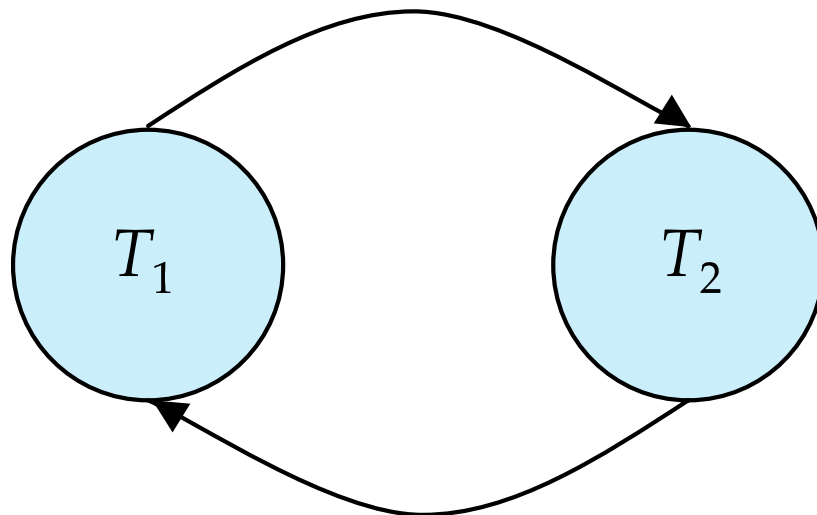
T_1	T_5
read (A) $A := A - 50$ write (A)	
	read (B) $B := B - 10$ write (B)
read (B) $B := B + 50$ write (B)	
	read (A) $A := A + 10$ write (A)

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



Testing for Serializability

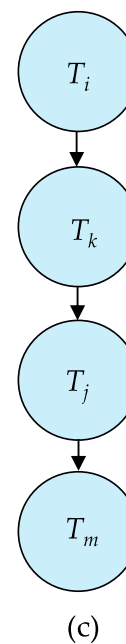
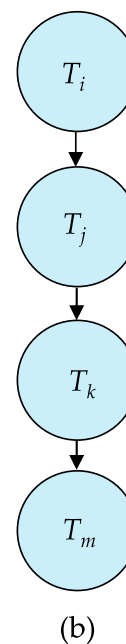
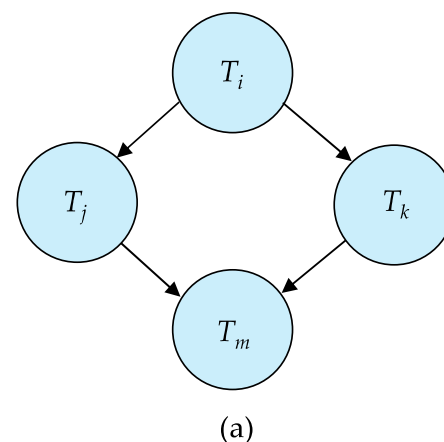
- n Consider some schedule of a set of transactions T_1, T_2, \dots, T_n
- n **Precedence graph** — a directed graph where the vertices are the transactions (names).
- n We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- n We may label the arc by the item that was accessed.
- n **Example 1**





Test for Conflict Serializability

- n A schedule is conflict serializable if and only if its precedence graph is acyclic.
- n 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.)
- n If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.





Test for View Serializability

- n 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.
- n 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.
- n 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.

- n **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 .
- n 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)	

- n 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

- n **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.

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



Cascadeless Schedules

- n **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 .
- n Every cascadeless schedule is also recoverable
- n It is desirable to restrict the schedules to those that are cascadeless



Concurrency Control

- n 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
- n 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?
- n Testing a schedule for serializability *after* it has executed is a little too late!
- n **Goal** – to develop concurrency control protocols that will assure serializability.



Concurrency Control (Cont.)

- n Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- n A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- n Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- n 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

- n Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- n Concurrency control protocols generally do not examine the precedence graph as it is being created
 - | Instead a protocol imposes a discipline that avoids nonserializable schedules.
 - | We study such protocols in Chapter 10.
- n Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- n Tests for serializability help us understand why a concurrency control protocol is correct.



Weak Levels of Consistency

- n 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
- n Tradeoff accuracy for performance



Levels of Consistency in SQL-92

- n **Serializable** — default
 - n **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.
 - n **Read committed** — only committed records can be read, but successive reads of a record may return different (but committed) values.
 - n **Read uncommitted** — even uncommitted records may be read.
-
- n Lower degrees of consistency useful for gathering approximate information about the database
 - n Warning: some database systems do not ensure serializable schedules by default
 - | E.g. Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)



Transaction Definition in SQL

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



End of Chapter 10

modified from:

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use