Chapter 14: Transactions

Database System Concepts, 29th Ed.

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
  - conflict serializability
  - view serializability
- Testing for Serializability
- Recoverability
- Levels of Consistency
- Transaction Definition in SQL

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $1000 from account A to account B:
  1. **read**(*A*)
  2. $A := A - 1000$
  3. **write**(*A*)
  4. **read**(*B*)
  5. $B := B + 1000$
  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

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $1000 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 1000
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 1000
6. **write**(*B*)

1. **read**(*A*)
2. *A* := *A* – 1000
3. **write**(*A*)

4. **read**(*B*)
5. *B* := *B* + 1000
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

ACID Properties

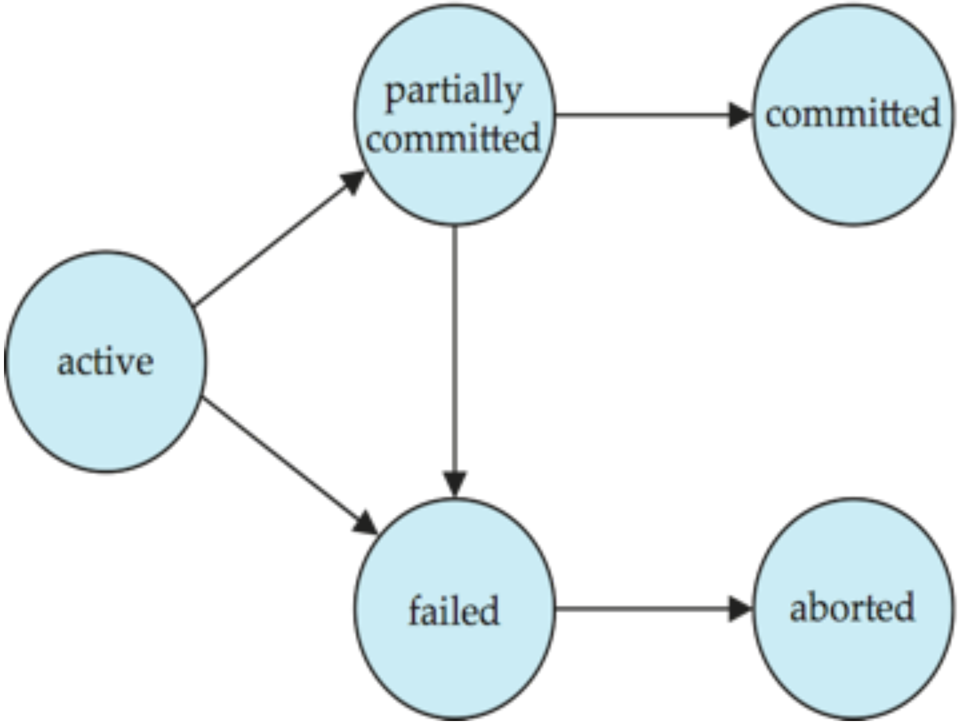
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **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.

Chapter 14: Transactions

- Transaction Concept
- **Transaction State**
- Concurrent Executions
- Serializability
  - conflict serializability
  - view serializability
- Testing for Serializability
- Recoverability
- Levels of Consistency
- Transaction Definition in SQL

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** – 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.

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- **Concurrent Executions**
- Serializability
  - conflict serializability
  - view serializability
- Testing for Serializability
- Recoverability
- Levels of Consistency
- Transaction Definition in SQL

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
  - control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

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
- 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*)	
$A := A - 50$	
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

A: 100, B: 0

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 :

A: 100, B: 0

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

A: 50, B: 50

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	
		A: 100, B: 0
read (*A*)		
$A := A - 50$		
write (*A*)		
read (*B*)		
$B := B + 50$		
write (*B*)		
commit		
		A: 50, B: 50
	read (*A*)	
	$temp := A * 0.1$	
	$A := A - temp$	
	write (*A*)	
	read (*B*)	
	$B := B + temp$	
	write (*B*)	
	commit	A: 45, B: 55

Schedule 2

- A serial schedule where T_2 is followed by T_1

A: 100, B: 0

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

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

A: 100, B: 0

A: 90, B: 10

Schedule 2

- A serial schedule where T_2 is followed by T_1

A: 100, B: 0

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

A: 90, B: 10

A: 40, B: 60

Schedule 3

- Is the following OK?

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

A: 100, B: 0

Schedule 3

- Is the following OK?

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

A: 100, B: 0

A: 45, B: 55

Schedule 3

- Is the following OK?
- 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.

A: 100, B: 0

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

A: 45, B: 55

Schedule 3

- Is the following OK?
- 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

A: 100, B: 0

A: 45, B: 55

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

Schedule 4

- Is the following OK?

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

A: 100, B: 0

Schedule 4

- Is the following OK?

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

A: 100, B: 0

A: 50, B: 10

Schedule 4

- Is the following OK?
- 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

A: 100, B: 0

A: 50, B: 10

Chapter 14: Transactions

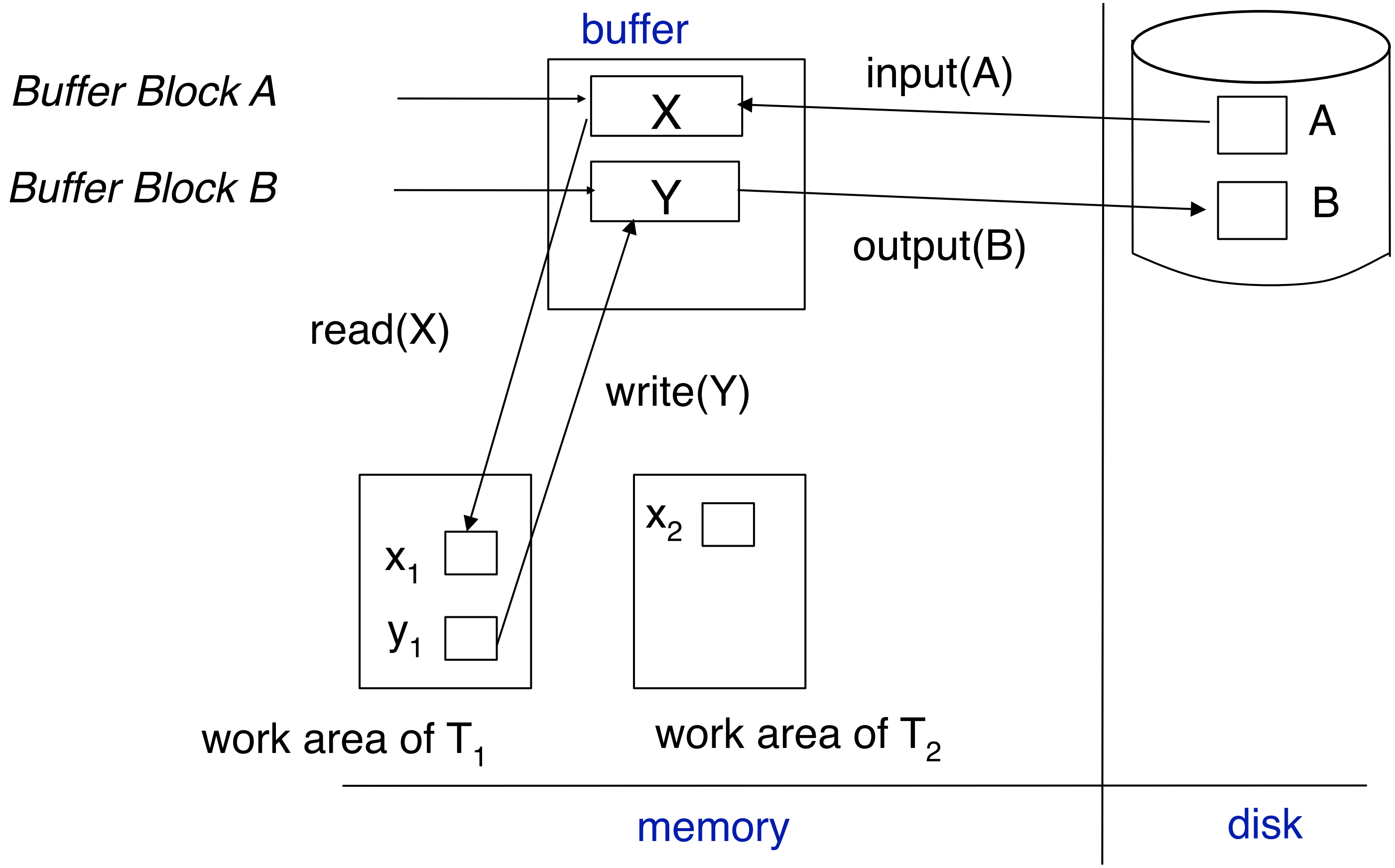
- Transaction Concept
- Transaction State
- Concurrent Executions
- **Serializability**
  - conflict serializability
  - view serializability
- Testing for Serializability
- Recoverability
- Levels of Consistency
- Transaction Definition in SQL

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

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

A: 100, B: 0

read(A)
A-50

write(A)

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 = **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

A: 100, B: 0

read(A)	
A-50	
	read(A) // A: 100
write(A)	

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

A: 100, B: 0

read(A)
A-50

write(A)
read(A) // A: 50

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

A: 100, B: 0

read(A)	
A-50	
write(A)	
	read(A) // A: 50

A: 100, B: 0

	read(A) // A: 100
read(A)	
A-50	
write(A)	

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 = **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

A: 100, B: 0

read(A)	
A-50	
write(A)	
	read(A) // A: 50

A: 100, B: 0

	read(A) // A: 100
read(A)	
A-50	
	write(A)
write(A)	

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 = **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

A: 100, B: 0

read(A)

A-50

write(A)

read(A) // A: 50

A: 100, B: 0

read(A) // A: 100

read(A)

A-50

write(A)

write(A)

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 4 can be transformed into Schedule 5, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 4 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 4

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 5

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,*
  1. ***(Initial reads)*** 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. ***(Read-from relationships)*** If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by **write**(Q) operation of transaction T_j , then in schedule *S'* transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. ***(Final writes)*** 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'.*

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)

- To what serial schedule is the above equivalent?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

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)

T27	T28	T29
read(Q)		
write(Q)		
	write(Q)	
		write(Q)

- To what serial schedule is the above equivalent?
- 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)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

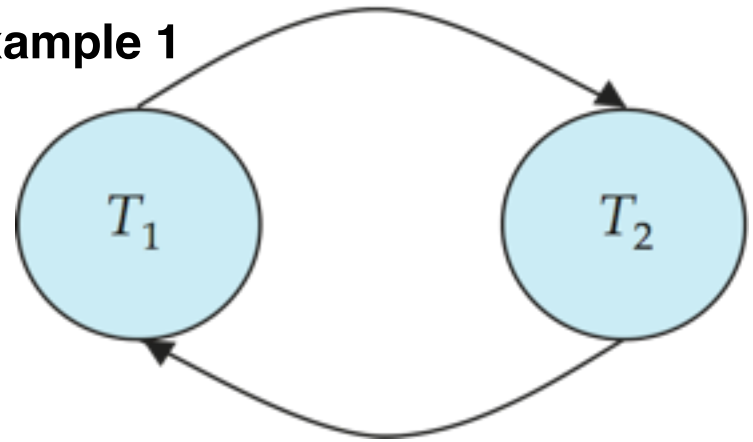
- Determining such equivalence requires analysis of operations other than read and write.

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
  - conflict serializability
  - view serializability
- **Testing for Serializability**
- Recoverability
- Levels of Consistency
- Transaction Definition in SQL

Testing for Serializability

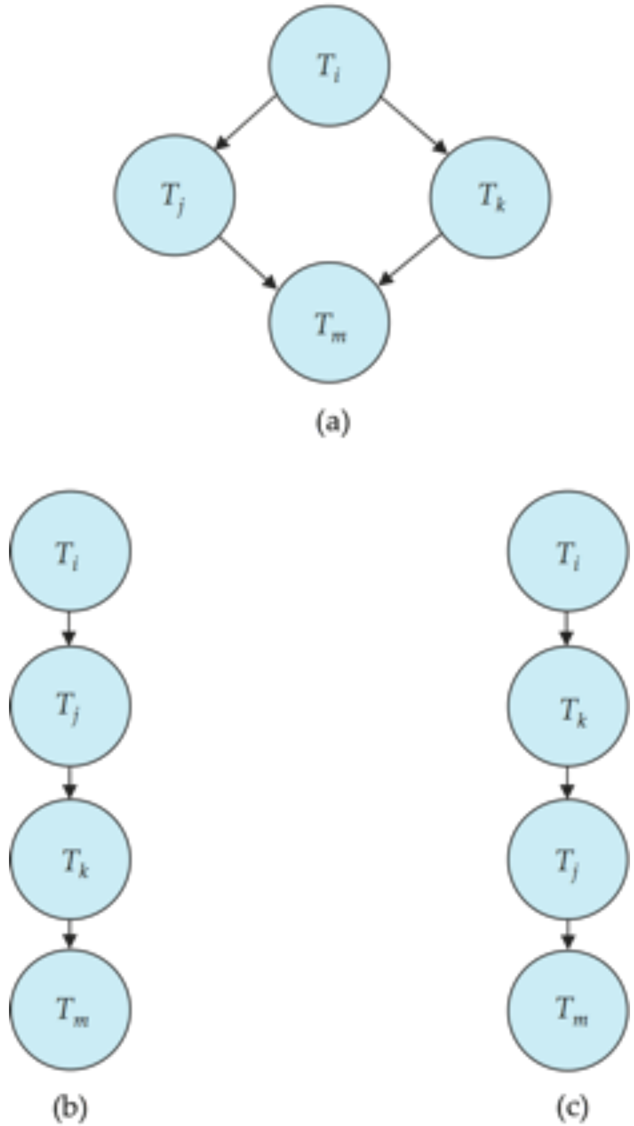
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may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



T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

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.
  - Consider (b), (c).
  - How can we do topological sorting?



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 *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
  - conflict serializability
  - view serializability
- Testing for Serializability
- **Recoverability**
- Levels of Consistency
- Transaction Definition in SQL

Recoverable Schedules

What is the problem of the following schedule?

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

Recoverable Schedules

What is the problem of the following schedule?

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

- The above schedule is **not recoverable** (what if T_8 *needs to abort after* T_9 commits?)
- **Early commit** of T9 causes the **consistency problem!**
- **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.

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
- **Early reads** (reading of uncommitted values) cause this **performance** problem (not a consistency problem).

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 (consistency)** and preferably cascadeless (performance)
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- **Goal** – to develop concurrency control protocols that will assure serializability and recoverability.

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
  - conflict serializability
  - view serializability
- Testing for Serializability
- Recoverability
- Levels of Consistency
- Transaction Definition in SQL

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 other records inserted by the transaction.
- **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.

- Lower degrees of consistency useful for gathering approximate information about the database

Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
  - conflict serializability
  - view serializability
- Testing for Serializability
- Recoverability
- Levels of Consistency
- **Transaction Definition in SQL**

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.
  - **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);