CMSC 461, Database Management Systems
Spring 2018

Lecture 20 – Transactions

These slides are based on "Database System Concepts" 6th edition book (whereas some quotes and figures are used from the book) and are a modified version of the slides which accompany the book (http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html), in addition to the 2009/2012 CMSC 461 slides by Dr. Kalpakis

Dr. Jennifer Sleeman

Logistics

- Homework #4 due 4/9/2018
- Homework #5 due 4/18/2018
- Phase 4 due 4/23/2018

Lecture Outline

- ***Transaction Concepts***
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transactions as SQL Statements

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

Based on and image from "Database System Concepts" book and slides, 6th edition

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.

Based on and image from "Database System Concepts" book and slides, 6th edition

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*)
- **Consistency requirement** in above example:
  - 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

Based on and image from "Database System Concepts" book and slides, 6th edition

Example of Fund Transfer

- **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*	

Based on and image from "Database System Concepts" book and slides, 6th edition

Example of Fund Transfer

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits (WED)

Based on and image from "Database System Concepts" book and slides, 6th edition

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.

Based on and image from "Database System Concepts" book and slides, 6th edition

ACID Properties

- **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.

Based on and image from "Database System Concepts" book and slides, 6th edition

Storage Structure

- Volatile – does not usually survive crashes (main memory, cache memory)
- Nonvolatile – survives system crashes (magnetic disk and flash)
- Stable -
  - data loss highly unlikely
  - Approximated by techniques
    - Replicate in several nonvolatile storage media with independent failure modes
    - More in 16.2.1

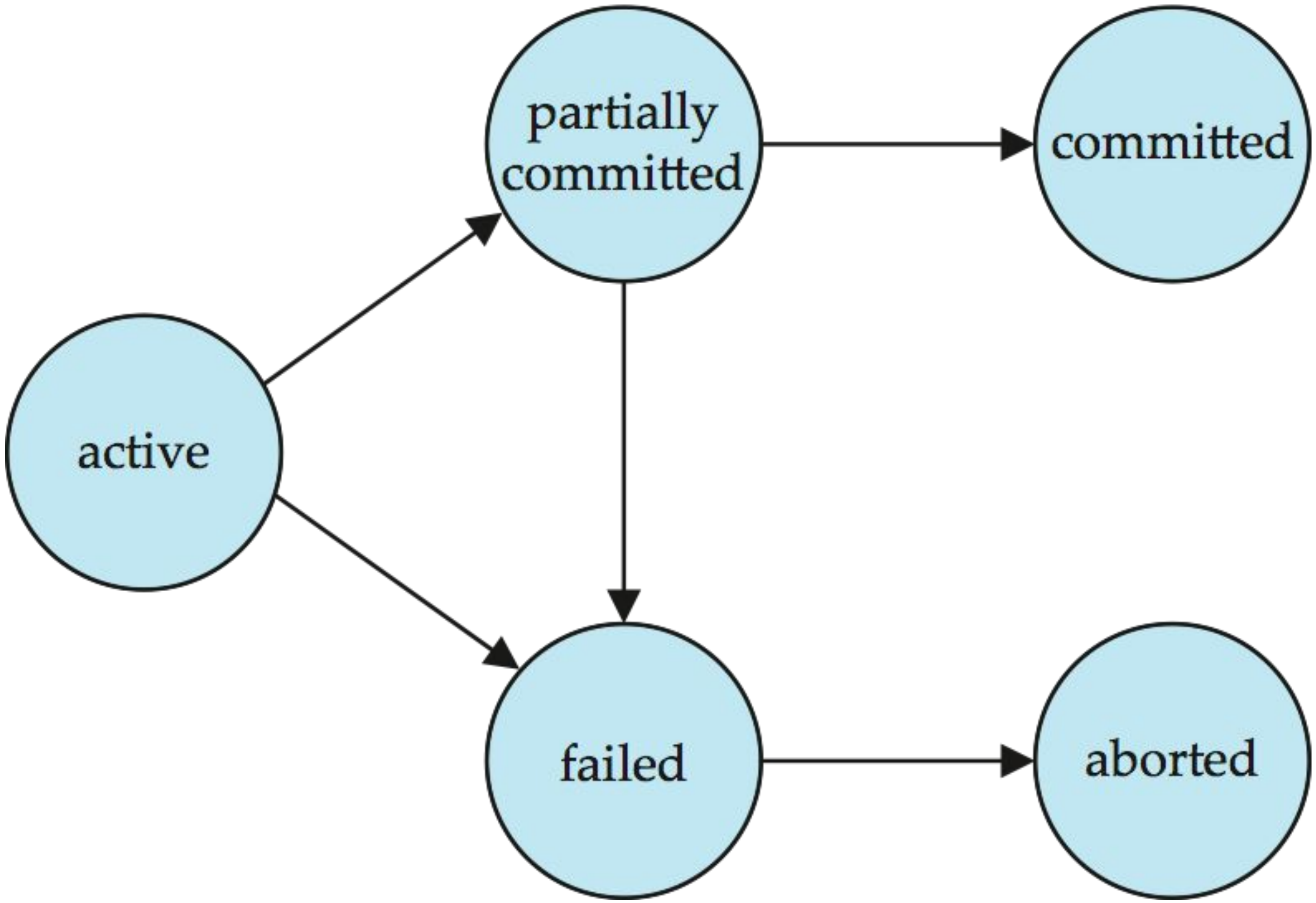
Based on and image from "Database System Concepts" book and slides, 6th edition

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.

Based on and image from "Database System Concepts" book and slides, 6th edition

Transaction State



Based on and image from "Database System Concepts" book and slides, 6th edition

Lecture Outline

- Transaction Concepts
- ***Transaction Isolation***
- Serializability
- Transaction Isolation and Atomicity
- Transactions as SQL Statements

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.

Based on and image from "Database System Concepts" book and slides, 6th edition

Concurrent Executions

- **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

Based on and image from "Database System Concepts" book and slides, 6th edition

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.

Schedules

- 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*)	
$A := A - 50$	
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

Based on and image from “Database System Concepts” book and slides, 6th edition

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	

Based on and image from "Database System Concepts" book and slides, 6th edition

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)	
$A := A - 50$	
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.

Based on and image from "Database System Concepts" book and slides, 6th edition

Schedule 4

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

Based on and image from "Database System Concepts" book and slides, 6th edition

Lecture Outline

- Transaction Concepts
- Transaction Isolation
- ***Serializability***
- Transaction Isolation and Atomicity
- Transactions as SQL Statements

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:
  - conflict serializability
  - 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.

Based on and image from "Database System Concepts" book and slides, 6th edition

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

Based on and image from "Database System Concepts" book and slides, 6th edition

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

Based on and image from "Database System Concepts" book and slides, 6th edition

Conflict Serializability

- 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

Based on and image from "Database System Concepts" book and slides, 6th edition

Conflict Serializability

- 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 >.

Based on and image from "Database System Concepts" book and slides, 6th edition

Testing for Serializability

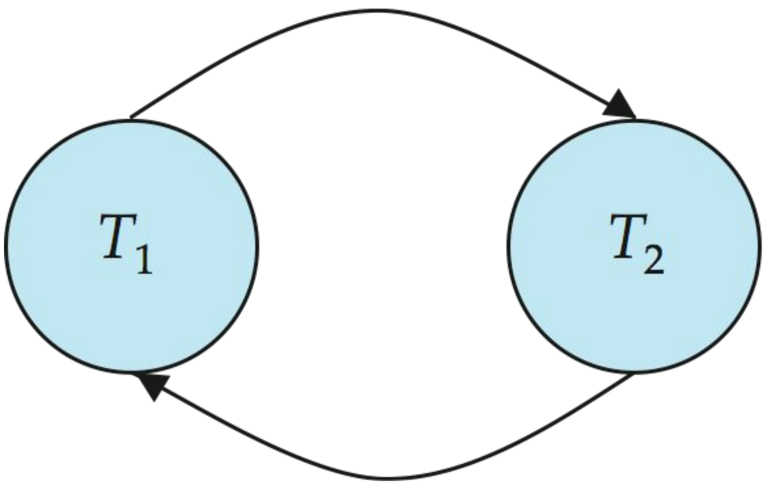
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a directed graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may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

Testing for Serializability

- Precedence graph for schedule 4.
- Contains edge T1 -> T2
  - T1 executes read(A) before T2 executes write(A)
- Contains edge T2 -> T1
  - T2 executes read(B) before T1 executes write(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

Schedule 4

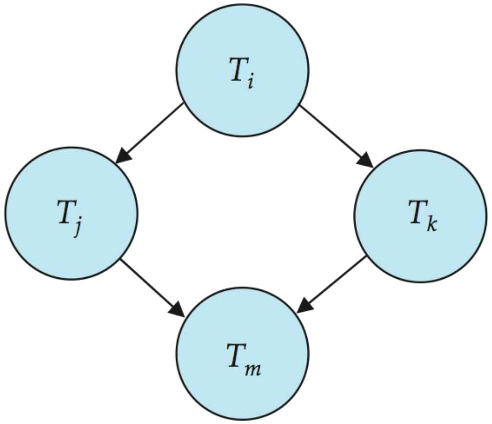


Precedence Graph for Schedule 4

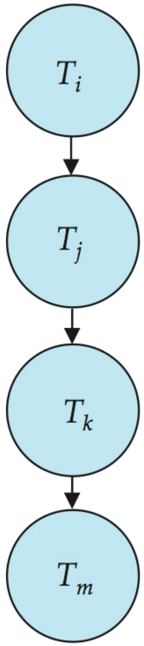
Based on and image from "Database System Concepts" book and slides, 6th edition

Test for Conflict Serializability

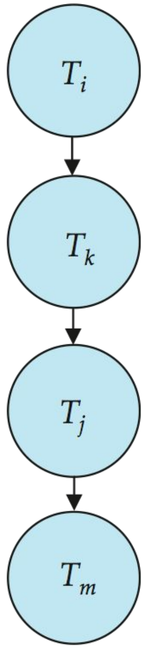
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)



(a)



(b)

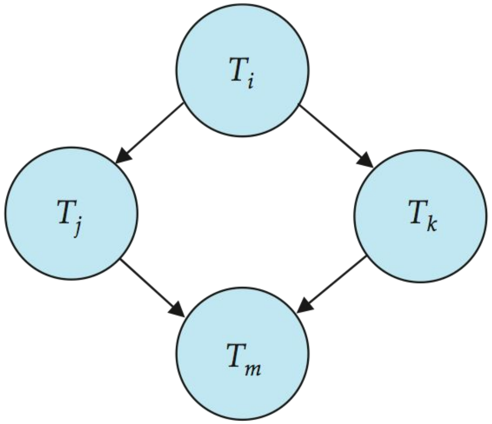


(c)

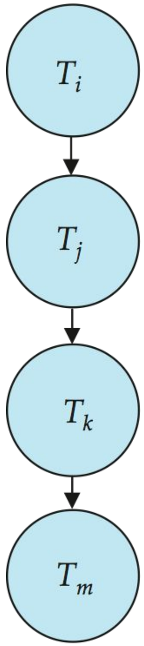
Based on and image from "Database System Concepts" book and slides, 6th edition

Test for Conflict Serializability

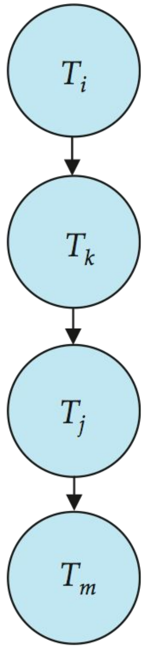
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



(a)



(b)



(c)

Based on and image from "Database System Concepts" book and slides, 6th edition

Lecture Outline

- Transaction Concepts
- Transaction Isolation
- Serializability
- ***Transaction Isolation and Atomicity***
- Transactions as SQL Statements

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.

Recoverable Schedules

- The following schedule is not recoverable if T_9 commits immediately after the read

- 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.

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

Based on and image from "Database System Concepts" book and slides, 6th edition

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		

Based on and image from "Database System Concepts" book and slides, 6th edition

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

Based on and image from "Database System Concepts" book and slides, 6th edition

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.

Based on and image from "Database System Concepts" book and slides, 6th edition

Concurrency Control

- 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.

Lecture Outline

- Transaction Concepts
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- ***Transactions as SQL Statements***

Levels of Consistency

- ***Serializable*** — default
- ***Repeatable read*** — only committed records to be read, repeated reads of same record must return same value.
  - No other transaction allowed to update it
  - transaction may not be serializable w/r to other transactions
- ***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.

Based on and image from "Database System Concepts" book and slides, 6th edition

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.

Based on and image from "Database System Concepts" book and slides, 6th edition

Transaction Definition in SQL

- 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

Transaction Definition in SQL

- 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

Based on and image from "Database System Concepts" book and slides, 6th edition