COMP9311: DATABASE SYSTEMS

Term 1 2024

Week 9 – Transaction Management

By Xiaoyang Wang, CSE UNSW

Disclaimer: the course materials are sourced from previous offerings of COMP9311 and COMP3311

Transaction

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:

- Concurrent execution of multiple transactions
- Failures of various kinds, such as hardware failures and system crashes

Issue (1)

Concurrent execution of **multiple** transactions is **needed**

Why?

- Multiple users/transactions may read/change the same data
- Allowing multiple transactions to update data concurrently can result in complications - data inconsistency.

Therefore, transaction processing systems...

- need to support multiple transactions at the same time.
- usually allow multiple transactions to run concurrently.

Issue (2)

Failures of various kinds

a. System failure
   - Disk failure - e.g., head crash, media fault
   - System crash - e.g., unexpected failure requiring a reboot

b. Program error
   - e.g., divide by zero

c. Exception conditions
   - e.g., no seats for your reservation

d. Concurrency control
   - e.g., deadlock, expired locks

Transaction Processing Systems need to be **robust** against failure

Example of Fund Transfer (1)

A **transaction** is a unit of program execution that accesses and possibly updates various data items.

Example: A possible 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)

Each transaction typically includes some **database access operations**

Operations relevant to transaction processing:

1. Read
2. Write
3. Computation

Example of Fund Transfer (2)

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 (**all-or-nothing**)

Example:
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)

Example of Fund Transfer (3)

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:
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)

Example of Fund Transfer (4)

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
- A transaction must see a consistent database

<u>During transaction execution</u> the database may be temporarily inconsistent.

- When the transaction completes successfully, the database must be consistent
- Erroneous transaction can lead to inconsistency

Example of Fund Transfer (5)

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.

ACID Summary

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 the ACID property.

- **Atomicity:** Either all operations of the transaction are properly reflected in the database, or none are.
- **Consistency:** Every transaction sees a consistent 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 Ti and Tj, it must appear to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction States

Active – the initial state; the transaction stays in this state while it is executing

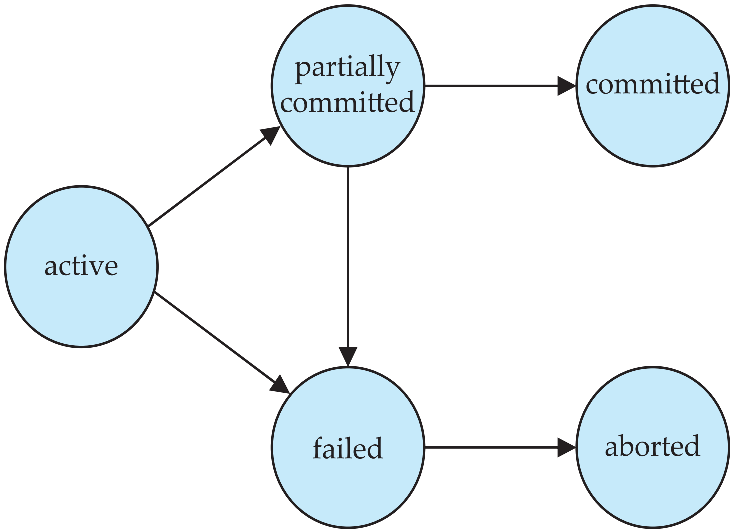
Partially committed – after the final statement has been executed.

Failed – after the discovery that normal execution can no longer proceed.

Aborted – after the transaction has been rolled back using log and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:

- Restart the transaction
- Kill the transaction

Committed – after successful completion.



Concurrent Executions

Multiple transactions are allowed to run concurrently in the system.

The 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 to 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

A Simple Transaction Model

We do not consider the full set of SQL language and ignore SQL insertion/delete operations.

Two operations:

- read(X) to transfer the data item X from database to a variable, also called X in a buffer in main memory.
- write(X) to transfer the value in the variable X in the buffer to the data item in the database.

Recall example: transaction transfers $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)

More Concepts

Concurrency/Isolation/Schedule/Control

- The concurrent execution in a database system is similar to the multiprogramming in an operating system (OS).
- When several transactions run concurrently, the isolation property may be violated.
- A schedule can help identify the executions that are guaranteed to ensure the isolation property and thus database consistency.
- A **concurrency-control scheme** is to control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database.

Schedules

Schedule – a sequence that specifies the 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 instruction as the last statement

- By default, transaction is 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

Two Transactions

Consider that the system receives two transactions

Let T_1 transfer $50 from A to B,

Let T_2 transfer 10% of the balance from A to B.

T1
read (A)
A := A – 50
write (A)
read (B)
B := B + 50
write (B)
commit

T2
read (A)
temp := A * 0.1
A := A - temp
write (A)
read (B)
B := B + temp
write (B)
commit

Schedule 1

A schedule S is **serial** if for every transaction T in the schedule, all (the operations of) T are executed consecutively in the schedule.

Example: a **serial** schedule in which T_1 is followed by T_2 :

T1	T2
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

Example: another **valid** serial schedule where T_2 is followed by T_1

T1	T2
	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 3

Let T_1 and T_2 be the transactions given previously.

- The following schedule is **not** a serial schedule
- but it is equivalent to Schedule 1

T1	T2
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

The following concurrent schedule:

- does **not** preserve the value of (A + B).

T1	T2
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

Not all concurrent schedules are desirable

How to Avoid These Problems?

If operations are interleaved arbitrarily, incorrect results may occur.

- Isolation can be **ensured trivially** by running transactions **serially**.

Question: why not run only serial schedules?

Answer: Because of very poor throughput due to disk latency

- If a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time.
- Additionally, if some transaction T is quite long, the other transactions must wait for T to complete all its operations before starting.

How to Avoid These Problems?

Question: why not run only serial schedules?

- It is desirable to interleave the operations of transactions in an **appropriate way**.

  We can fully utilize **resources.**

  For example, if one transaction is waiting for I/O to complete, another transaction can use the CPU.

Point:

- serial schedules are considered unacceptable in practice
- executing multiple transactions concurrently has significant benefits.

Motivation (1)

We need to study the notion of correctness of concurrent executions.

- Every transaction is executed from beginning to end in **isolation** from the operations of other transactions, we get a correct end result on the database.

We first need to define types of schedules that are always considered to be correct when concurrent transactions are executing.

Motivation (2)

Question: How do we determine if non-serial schedules are correct?

Intuition: If we can determine which non-serial schedules are equivalent to a serial schedule, we can allow these schedules to occur.

Summary

- Every serial schedule is considered correct
- We can assume this because every transaction is assumed to be correct if executed on its own (according to the consistency preservation property).
- For serial schedules, it does not matter which transaction execute first. They are all correct.

Basic Assumption – Each transaction preserves database consistency

- Therefore, a **serial** execution of a set of transactions preserves database consistency.
- Serial executions are correct

Serializability

A (possibly concurrent) schedule S of n transactions is **serializable** if

- it is **equivalent** to some serial schedule of the same n transactions.

There are many notions forms of schedule equivalence give rise to the notion of **conflict serializability** (Will Discuss Later)

A 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.

An example: For a transaction that transfers $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)

We simply considers the read/write only

read(A)
write(A)
read(B)
write(B)

Conflicting Instructions

Instructions I_i and I_j of different 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

Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them. i.e., changing their order can result in a different combined outcome.

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.

For example, read–read operations are are not conflicting.

Summary: Conflicting Instructions

Summary: Two operations O_1 and O_2 are conflicting if

- They are in different transactions
- They access the same data item,
- At least one of them must be a write.

Language: The transaction of the second operation in the pair is said to be in conflict with the transaction of the first operation.

Conflict Equivalence

Two schedules are said to be **conflict equivalent** if:

- the order of <u>any</u> two conflicting operations is the same in both schedules.

Two schedules are conflict equivalent if:

- Involve the same actions of the same transactions
- Every pair of conflicting actions is ordered the same way

We define equivalence of schedules by conflict equivalence, which is the more commonly used definition

A better definition of equivalence compared to result equivalence .

Conflict Serializability

Using the notion of conflict equivalence, we define a schedule S to be **conflict serializable** if it is (conflict) equivalent to some serial schedule S.

For conflict serializable schedules:

- we can reorder the nonconflicting operations in S until we form the equivalent serial schedule S.

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 (2)

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.

T1	T2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T1	T2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Note: any conflict serializable schedule is also a serializable schedule

Conflict Serializability (3)

Example of a schedule that is not conflict serializable:

T3	T4
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>.

Note: Not all schedules are conflict serializable.

Allow Some Concurrent Schedules

Now we characterized the types of schedules that are always considered to be **correct** when concurrent transactions are executing.

The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

Since serial schedules are not practical, we can allow conflict serializable schedules since they are correct!

Allow Some Concurrent Schedules

A nonserial schedule S is serializable is equivalent to a serial schedule saying that it is correct

- because it is equivalent to a serial schedule, which is always considered correct.

Practice:

1. Is a serializable schedule correct?
2. Is being serializable the same as being serial.
3. Is a non-serializable schedule correct?
4. Is a nonserial schedule correct?

Testing Conflict Serializability

There is a simple algorithm for determining whether a particular schedule is conflict serializable or not.

The algorithm looks at only the read_item and write_item operations in a schedule. (A Simplified View of Transactions)

Algorithm

Step 1: Construct a precedence graph.

Step 2: Check if the graph is cyclic:

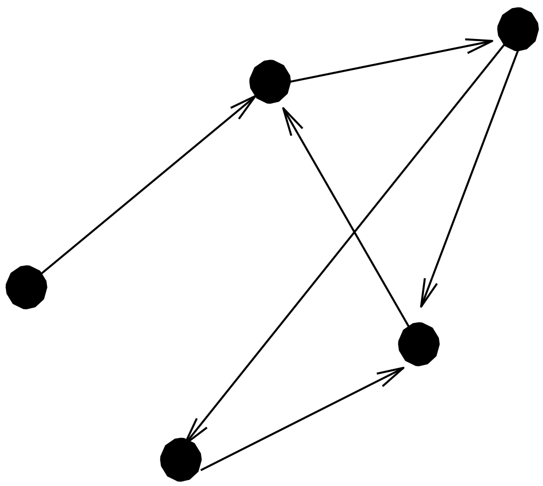
- Cyclic: non-serializable.
- Acyclic: serializable.

Preliminary

A directed graph G = (V, A) consists of

- a vertex set V
- an arc set A such that each arc connects two vertices.

Cyclic: G is cyclic if G contains a directed cycle.



Cyclic Graph

Serializability Testing: Precedence Graph

Consider some schedule of a set of transactions $\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n$

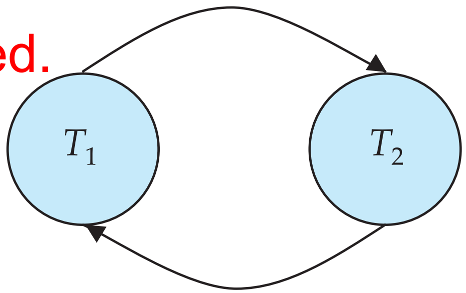
Precedence graph — a directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ where the vertices (V) are the transactions.

We draw an arc from T_i to T_j, $\mathrm{T}_i \rightarrow \mathrm{T}_j$, if the two transactions are conflict, and T_i accessed the data item earlier.

- T_i executes write(Q) before T_j executes read(Q)
- T_i executes read(Q) before T_j executes write(Q)
- T_i executes write(Q) before T_j executes write(Q)

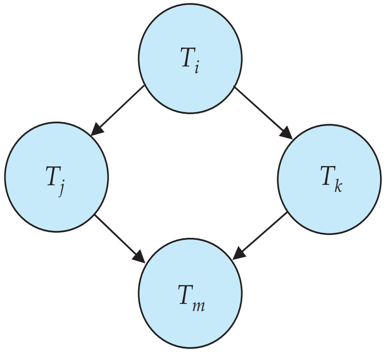
We may <u>label</u> the arc by the item that was accessed.

Example (of a precedence graph):



Conflict Serializability Testing

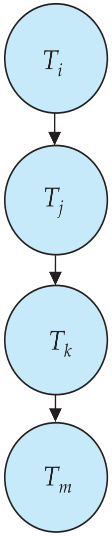
A schedule is conflict serializable if and only if its precedence graph is acyclic (cycle free).



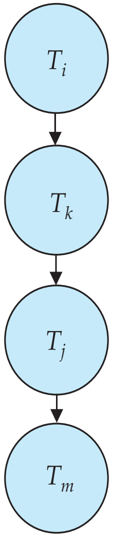
(a)

If the 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 (a), there are two linear orders (b) and (c).



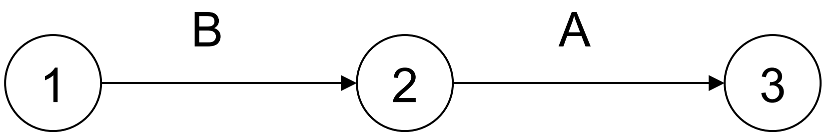
(b)



(c)

Example (1)

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



If there is **no** cycle in the precedence graph, this schedule is conflict-serializable

Note: Here we label the arc by the item that was accessed.

Example (2)

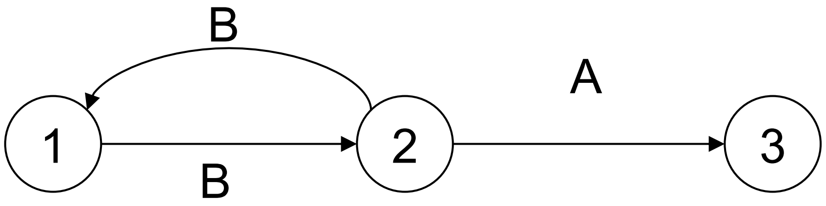
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

If there is a cycle in the precedence graph, this schedule is NOT conflict-serializable

Note: Here we label the arc by the item that was accessed.

Example (2)

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



If there is a cycle in the precedence graph, this schedule is NOT conflict-serializable

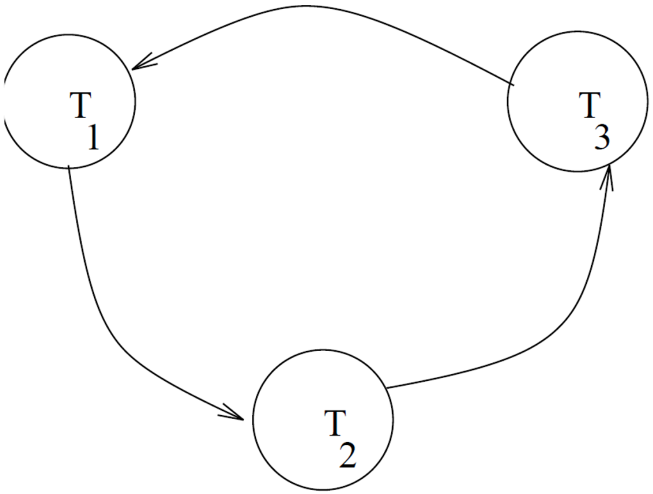
Note: Here we label the arc by the item that was accessed.

Example (3)

Schedule	T_1	T_2	T_3
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
write(B)		write(B)	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
write(A)	write(A)		
read(B)			read(B)
read(A)		read(A)	
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)	read(C)		
write(A)		write(A)	
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)	write(C)		
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)

Example (3)

Schedule	T_1	T_2	T_3
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
write(B)		write(B)	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
write(A)	write(A)		
read(B)			read(B)
read(A)		read(A)	
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)	read(C)		
write(A)		write(A)	
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)	write(C)		
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)

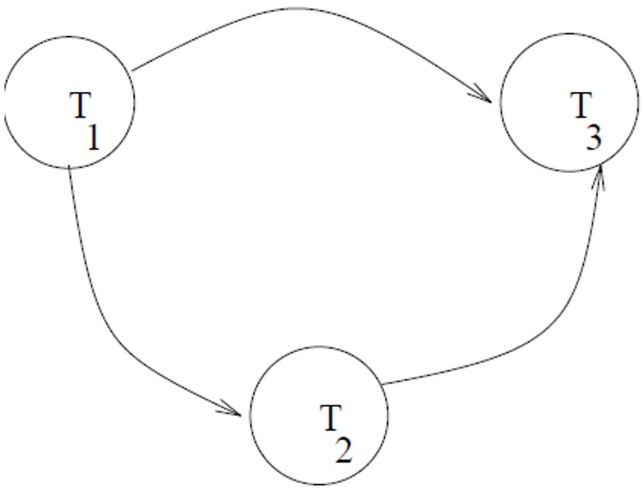


Example (4)

Schedule	T_1	T_2	T_3
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$		$B \leftarrow f_3(B)$	
write(B)		write(B)	
$C \leftarrow f_4(C)$			$C \leftarrow f_4(C)$
read(B)			read(B)
write(C)			write(C)
$A \leftarrow f_5(A)$		$A \leftarrow f_5(A)$	
write(A)		write(A)	
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)

Example (4)

Schedule	T_1	T_2	T_3
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$		$B \leftarrow f_3(B)$	
write(B)		write(B)	
$C \leftarrow f_4(C)$			$C \leftarrow f_4(C)$
read(B)			read(B)
write(C)			write(C)
$A \leftarrow f_5(A)$		$A \leftarrow f_5(A)$	
write(A)		write(A)	
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)



Concurrency Control

A database must provide a mechanism that will ensure that all possible schedules executed are serializable.

A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency

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 schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Concurrency Control vs. Serializability Tests

Concurrency-control protocols allow concurrent schedules, ensure that the schedules are serializable.

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 a protocol later)

There are many 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 if and why a concurrency control protocol is correct.

Discussion

Testing for serializability on the fly is not practical.

Instead, several protocols have been developed which ensure that if every transaction obeys the rules, then every schedule will be serializable, and thus correct.

We discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols, and we briefly mentioned less restrictive definitions of schedule equivalence.

Learning Outcomes

Transaction Concept

Transaction State

Concurrent Executions

Serializability

Not all schedules are serializable

A nonserial schedules can be one of the following case:

- those that are equivalent to one (or more) of the serial schedules
- those that are not equivalent to *any* serial schedule and hence are not serializable.