Transaction Management

COSC 404 – Database System Implementation

Transaction Management Overview

The database system must ensure that the data stored in the database is always ***consistent***.

There are several possible types of ***failures*** that may cause the data to become inconsistent.

A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.

- The input to a transaction is a consistent database, AND the output of the transaction must also be a consistent database.
- A transaction must execute completely or not at all.

Transaction Management
Motivating Example

Consider a person who wants to transfer $50 from a savings account with balance $1000 to a checking account with current balance $250.

- 1) At the ATM, the person starts the process by telling the bank to remove $50 from the savings account.
- 2) The $50 is removed from the savings account by the bank.
- 3) Before the customer can tell the ATM to deposit the $50 in the checking account, the ATM "crashes."

Where has the $50 gone?

It is lost if the ATM did not support transactions!
The customer wanted the withdraw and deposit to both happen in one step, or neither action to happen.

Transaction Definition

A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.

The basic assumption is that when a transaction starts executing the database is consistent, and when it finishes executing the database is still in a consistent state.

- Do not consider malicious or incorrect transactions.
- This assumption is called **The Correctness Principle**.

Note that the database may be inconsistent during transaction execution.

- For the bank example, the $50 is removed from the savings account and is not yet in the checking account at some point in time.

Consistency Definition

A database is ***consistent*** if the data satisfies all constraints specified in the database schema. A ***consistent database*** is said to be in a ***consistent state***.

A ***constraint*** is a predicate (rule) that the data must satisfy.

- Examples:
  - *StudentID* is a key of relation *Student*.
  - *StudentID* $\rightarrow$ *Name* holds in *Student*.
  - No student may have more than one major.
  - The field *Major* can only have one of the 4 values: {“BA”,”BS”,”CS”,”ME”}.
  - The field *Year* must be between 1 and 4.

Note that the database may be internally consistent but not reflect the real-world reality.

Consistency Issues

There are two major challenges in preserving consistency:

- 1) The database system must handle ***failures*** of various kinds such as hardware failures and system crashes.

- 2) The database system must support ***concurrent execution*** of multiple transactions and guarantee that this concurrency does not lead to inconsistency.

ACID Properties

To preserve integrity, transactions have the following properties:

- ***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 executing 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 Operations

Since a transaction is a general program, there are an enormous number of potential operations that a transaction can perform.

However, there are two really important operations:

- `read(A,t)` (or `read(A)` when *t* is not important)
  - Read database element *A* into local variable *t*.
- `write(A,t)` (or `write(A)` when *t* is not important)
  - Write the value of local variable *t* to the database element *A*.

For most of the discussion, we will assume that the buffer manager insures that database element is in memory. We could make the memory management more explicit by using:

- `input(A)`
  - Read database element *A* into local memory buffer.
- `output(A)`
  - Write the block containing *A* to disk.

Fund Transfer Transaction Example

Transaction to transfer $50 from account *A* to account *B*:

```
1.    read(A,t)
2.    t := t - 50
3.    write(A,t)
4.    read(B,t)
5.    t := t + 50
6.    write(B,t)
```

Fund Transfer Transaction Example (2)

Atomicity requirement – If the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, or inconsistency will result.

Consistency requirement – The sum of *A* and *B* is unchanged by the execution of the transaction.

Isolation requirement – If between steps 3 and 6, another transaction accesses the partially updated database, it will see an inconsistent database (*A + B* is less than it should be).

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

Durability requirement – Once the user has been notified that the transaction has completed (i.e., the $50 transfer occurred), the updates by the transaction must persist despite failures.

ACID Properties

Question: Two transactions running at the same time can see each other's updates. What ACID property is violated?

A) atomicity

B) consistency

C) isolation

D) durability

E) none of them

ACID Properties (2)

Question: A company stores a customer's address in the database. The customer moves and does not tell the company to update its database. What ACID property is violated?

A) atomicity

B) consistency

C) isolation

D) durability

E) none of them

Transaction Questions

Example database:

```
Student(Id,Name,Major,Year)
```

1) Write a transaction to change the name of a student to “Joe Smith.” Let *A* represent the database object currently storing the name.

2) Write a transaction to swap the names of two students with names *A* and *B*.

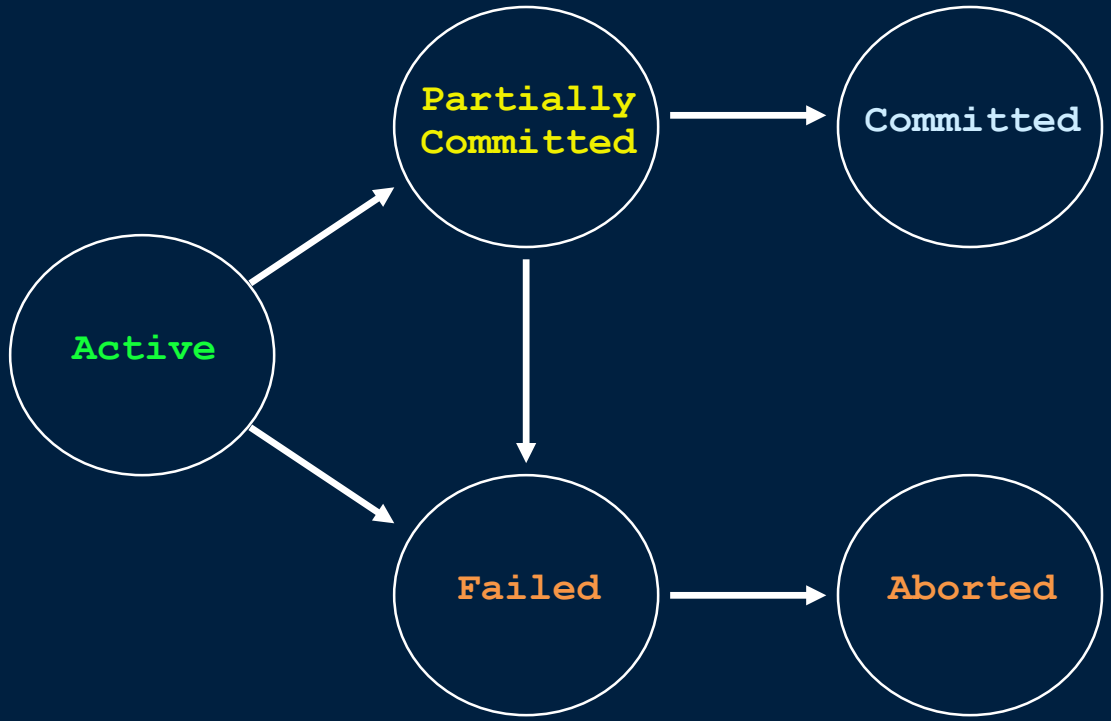
3) Write a transaction to increase the *Year* of all students by 1.

Transaction States

An executing transaction can be in one of several states:

- ***Active*** - is the initial state. The transaction stays in this state while it is executing.
- ***Partially committed*** **-** A transaction is partially committed after its final statement has been executed.
- ***Failed*** **-** A transaction enters the failed state after the discovery that normal execution can no longer proceed.
- ***Aborted*** **-** A transaction is aborted after it has been rolled back and the database restored to its prior state before the transaction. There are two options after abort:
  - restart the transaction – only if no internal logical error
  - kill the transaction - problem with transaction itself
- ***Committed*** **-** Commit state occurs after *successful completion*.
  - May also consider **terminated** as a transaction state.

Transaction State Diagram



Transaction States

Question: Is it possible for a transaction to be in the aborted and committed states at different times during its lifetime?

A) yes

B) no

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

- Increased processor and disk utilization, leading to better transaction ***throughput:*** one transaction can be using the CPU while another is reading from or writing to storage.
- Reduced ***average response time*** for transactions as short transactions need not wait behind long ones.

Concurrency control schemes are mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

- We will study concurrency control schemes after examining the notion of correctness of concurrent executions.

Schedules

A ***schedule*** is 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.
- We must preserve the order in which the instructions appear in each individual transaction.
- It is useful to think of a schedule as a journal of the database actions. It is a ***historical record*** that the database keeps as it is processing transactions.

A ***serial schedule*** is a schedule where the instructions belonging to each transaction appear together.

- i.e. There is no *interleaving* of transaction operations.
- For n transactions, there are $n!$ different serial schedules.

Example Schedules

Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*. Let *A*=100 and *B*=200. A serial schedule T_1 followed by T_2:

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
write(*A*,*t*)	
read(*B*,*o*)	
o := *o* + 50	
write(*B*,*o*)	
	read(*A*,*t*)
	temp := *t**0.1;
	t := *t* – *temp*
	write(*A*,*t*)
	read(*B*,*o*)
	o := *o* + *temp*
	write(*B*,*o*)

After schedule:
A=45, *B*=255

Is there another serial schedule?

Example Schedules (2)

Given T_1 and T_2 defined previously. The following schedule is not a serial schedule, but it is ***equivalent*** to the previous serial schedule:

T_1	T_2
read(A,t)	
$t := t - 50$	
write(A,t)	
	read(A,t)
	$temp := t*0.1$;
	$t := t - temp$
	write(A,t)
read(B,o)	
$o := o + 50$	
write(B,o)	
	read(B,o)
	$o := o + temp$
	write(B,o)

After schedule:
A=45, B=255

Example Schedules (3)

The following concurrent schedule does not preserve the value of the sum $A + B$: (***inconsistent state***)

T_1	T_2
read(A,t)	
$t := t - 50$	
	read(A,t)
	$temp := t*0.1$;
	$t := t - temp$
	write(A,t)
	read(B,o)
write(A,t)	
read(B,o)	
$o := o + 50$	
write(B,o)	
	$o := o + temp$
	write(B,o)

After schedule:
A=50, B=210

Is there another schedule with a different result?

Correct Schedules

Since the operating system can interleave the operations of concurrent transactions in any order, the database management system must ensure that only correct schedules are possible.

The database system guarantees only correct schedules are possible by implementing concurrency control protocols that guarantee that the schedule actually executed is **equivalent to some serial schedule**.

Schedules

Question: Is the schedule valid for the two transactions below?

Schedule:

T_1	T_2
read(*A,t*)	
read(*B,o*)	
write(*A,t*)	
write(*B,o*)	
	read(*A,t*)
	write(*A,t*)
	read(*B,o*)
	write(*B,o*)

Transaction T1:
read(A,t)
write(A,t)
read(B,o)
write(B,o)

Transaction T2:
read(A,t)
write(A,t)
read(B,o)
write(B,o)

A) yes

B) no

Why is Concurrency Control Needed?

Concurrency control is needed to ensure that the schedules executed leave the database in a consistent state.

Examples of concurrency control problems include:

- ***The Lost Update Problem*** - occurs when two transactions access the same data item, and one transaction reads the data item before the other transaction commits its written version. (The update from this transaction is *lost*.)
- ***Dirty Read Problem*** - occurs when a transaction reads a data value written by another transaction which later aborts.
- ***Incorrect Summary Problem*** - occurs when a transaction is calculating an aggregate function and some other transaction(s) is updating record values that may not all be reflected correctly in the summation calculation.

Lost Update Example

The ***lost update problem*** occurs when two transactions read the same value before either of them commits their write.

T_1	T_2
read(A,t)	
$t := t - 50$	
	read (A,t)
	temp := t *0.1
	$t = t$ – temp
	write(A,t)
read(B,o)	
write(A,t)	
write(B,o)	

A is written without T_1's changes!

Dirty Read Example

The ***dirty read*** (or temporary update) problem occurs when a transaction reads a value of a later aborted transaction.

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
write(*A*,*t*)	
	read (*A*,*t*)
	temp := *t* *0.1
	t = *t* – temp
	write(*A*,*t*)
read(*B*,*o*)	
abort	

If T_1 aborts, then T_2 has used its incorrect value of A, and should not be allowed to commit.

Incorrect Summary Example

The ***incorrect summary*** problem occurs when a transaction updates values when another transaction is calculating a sum.

T_1	T_2
	sum = 0 **read**(A) sum = sum + A ...
read(X) X = X -100 **write**(X)	X is updated before its value is used in summation.
	read (X) sum = sum + X **read** (Y) sum = sum + Y ...
read(Y) Y = Y +100 **write**(Y)	Y is updated after its value is used in summation. (not consistent with X)

Consistency Issues

Question: What consistency issue does this schedule have?

T_1	T_2
read(A,t)	
	read (A,t)
write(A,t)	
write(B, *10*)	
	read(B,u)
write(C,t)	
	write(C,t+u)

A) lost update **B)** dirty read **C)** incorrect summary **D)** none **E)** more than one

Serializability

A schedule is ***serializable*** if it is equivalent to a serial schedule.

There are two different forms of serializability:

1. **conflict serializability**
2. **view serializability**

We ignore operations other than **read** and **write** instructions, and 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.

Conflict Serializability
Conflicting Operations

To understand conflict serializability, we must understand what it means for two operations to conflict.

Operations O_i and O_j of transactions T_i and T_j respectively, ***conflict*** if and only if there exists some item Q accessed by both O_i and O_j, and at least one of these operations wrote Q.

Possibilities:

1. O_i = **read**(Q), O_j = **read**(Q). O_i and O_j do not conflict.
2. O_i = **read**(Q), O_j = **write**(Q). Conflict - order is important
3. O_i = **write**(Q), O_j = **read**(Q). Conflict - reverse of #2
4. O_i = **write**(Q), O_j = **write**(Q). Conflict - who writes last?

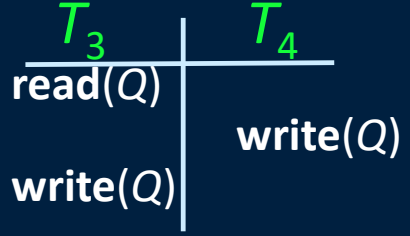
Intuitively, a conflict between O_i and O_j forces a (logical) temporal order between them. If O_i and O_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, *S* and *S´* are ***conflict equivalent***.

A schedule *S* is ***conflict serializable*** if it is conflict equivalent to a serial schedule.

Example of a schedule that is not conflict serializable:



- 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 >.

Conflict Serializability (2)

This schedule can be transformed into a serial schedule by a series of swaps of non-conflicting instructions. It is conflict serializable.

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

What is the serial schedule?

Conflict Serializability Question

Question: Is this schedule conflict serializable?

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

A) yes

B) no

Serializability Questions

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

Note shorthand notation!
E.g. $r_1(A) = T_1$ does read(A)

Questions:

- 1) How many possible serial schedules are there?
- 2) How many schedules are conflict equivalent to the serial order (T_1 , T_2)?
- 3) Write one non-serial schedule that is conflict equivalent to the serial execution (T_2 , T_1), if possible.

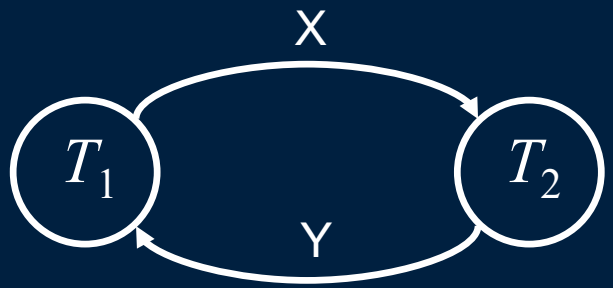
Testing for Serializability

It is possible to determine if some schedule of transactions T_1, T_2, ..., T_n is serializable using a precedence graph.

A ***precedence graph*** is a directed graph where the vertices are the transactions, and there is an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which they conflict earlier.

- We may label the arc using the item that was accessed.

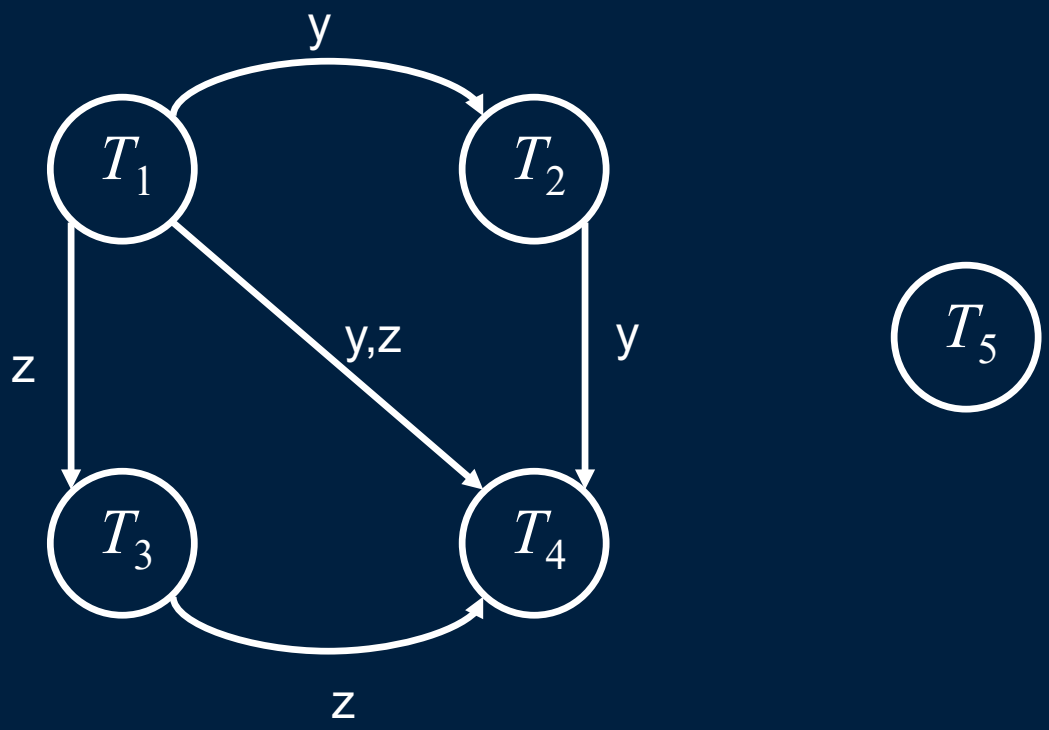
Example: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$;



Precedence Graph Example Schedule

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Precedence Graph for Schedule



Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is ***acyclic***.

Cycle-detection algorithms exist which take $O(n^2)$ time, where n is the number of vertices in the graph.

- Better algorithms take $O(n + e)$ where e is the # of edges.

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 example, one possible serializability order for the previous example would be:

$$T_5 \Rightarrow T_1 \Rightarrow T_3 \Rightarrow T_2 \Rightarrow T_4$$

Precedence Graph Questions

Give the precedence graph for the following schedules:

1) $r_2(B)$; $w_2(B)$; $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$;

2) $w_1(A)$; $w_2(B)$; $w_3(C)$; $w_4(D)$; $w_5(E)$; $w_5(A)$;

3) Construct a non-serial schedule with 3 transactions and 3 data items that has a precedence graph containing 6 arcs, but is still conflict serializable.

- Assume you put a separate edge for each data item that causes a conflict.

Other Schedule Properties

There are other desirable schedule properties:

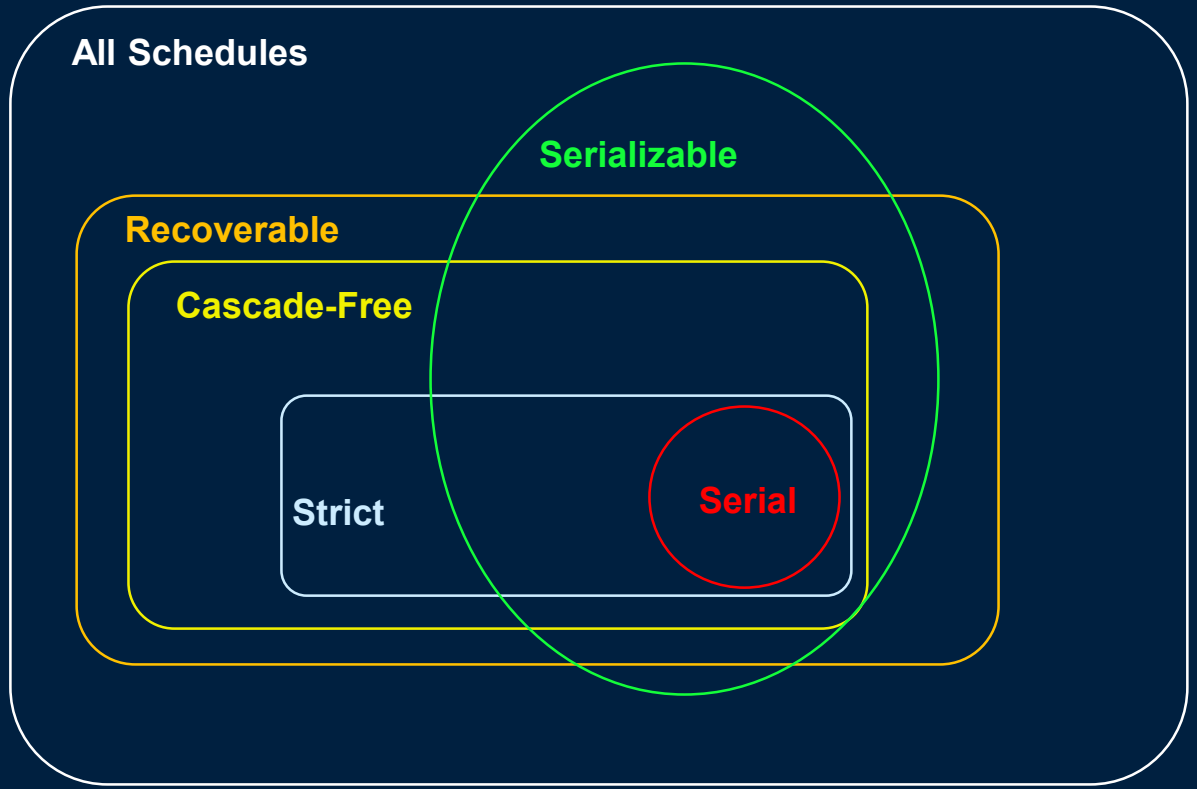
Recoverability - A ***recoverable*** schedule insures that a database can recover from failure even when concurrent transactions have been executing.

Cascade-Free **-** A ***cascading rollback*** occurs when a single transaction failure leads to a series of transaction rollbacks. A cascade-free schedule avoids cascading rollbacks.

Strict - Strict schedules simplify recovery procedures in the advent of failure.

Each of these properties subsumes the next. That is, all strict schedules are also cascade-free and recoverable. All cascade-free schedules are recoverable.

Schedule Properties Diagram



Schedule Properties Questions

Question: How many of the following statements are true?

- i) Every serial schedule is a strict schedule.
- ii) A serializable schedule may not be recoverable.
- iii) Every cascade-free schedule is also a strict schedule.
- iv) There are more recoverable schedules than cascade-free schedules.

A) 0

B) 1

C) 2

D) 3

E) 4

Recoverability

We need to address the effect of transaction failures on concurrently running transactions.

- Let a transaction T_j read a data value written by another transaction T_i. If T_i aborts, then T_j should also abort because the data it read was inconsistent.

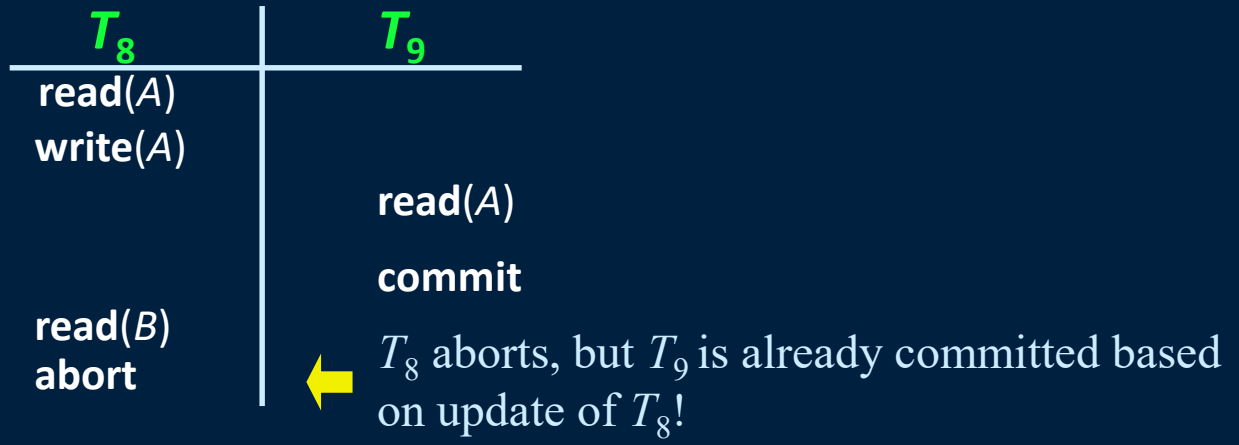
A ***recoverable*** schedule has the property that if a transaction T_j reads a data item previously written by a transaction T_i, the commit of T_i appears before the commit of T_j.

- Note that if T_i aborts ***before*** T_j commits then the schedule is recoverable. It is not recoverable if T_i aborts ***after*** T_j commits.

Obviously, the database system wants to only allow recoverable schedules in advent of failures.

Non-Recoverable Schedules

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



The schedule is **not recoverable** because the commit for T_9 cannot be undone, but it should be because T_8 was never committed!

Recoverable Schedule Question

Question: Is this schedule recoverable?

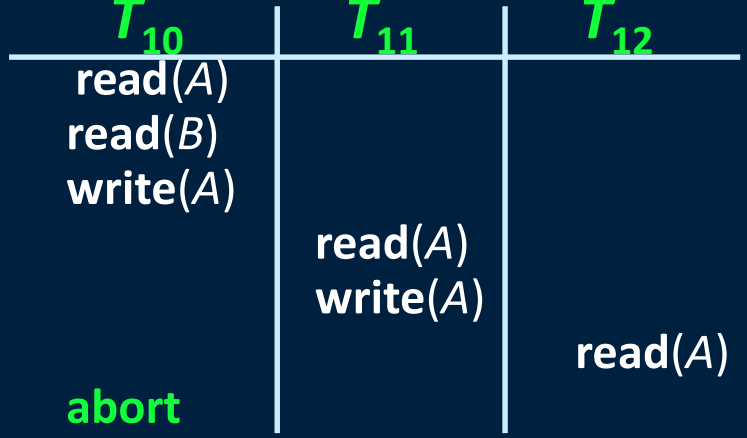
T_8	T_9
read(A)	
write(A)	
	read(A)
	commit
read(B)	
commit	

A) yes

B) no

Cascading Rollback

Cascading rollback occurs when a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where no transactions have yet committed (so the schedule is recoverable):



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!

- T10 does not have to abort for the schedule to have cascading rollback. T11 and T12 will be **FORCED** to abort if T10 aborts. Even if T10 commits, the schedule is not cascade-free because it has the *potential* for cascading aborts.

Cascadeless Schedules

In a ***cascadeless*** schedule, 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 of T_i appears before the read operation of T_j.
- That is, transactions only read committed values.

Every cascadeless schedule is also recoverable.

A recoverable schedule never rolls back committed transactions, but may cascade rollback *uncommitted* transactions.

Cascade-Free Schedule Question

Question: Is this schedule cascade-free?

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

A) yes

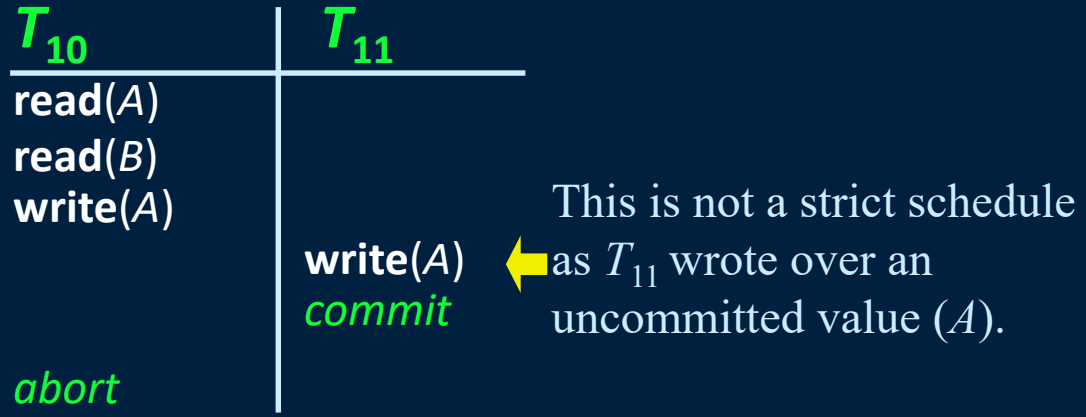
B) no

Strict Schedules

In a ***strict*** schedule, a transaction can neither read nor write a data item until the last transaction that wrote the data item commits (or aborts).

- Strict schedules simplify recovery because undoing an item write of an aborted transaction just involves restoring the before image (old value) of the item.
- A strict schedule is always recoverable and cascadeless, but not vice versa.

Example:



Schedule Questions

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; c_1

T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; c_2

T_3: $r_3(B)$; $r_3(A)$; $w_3(B)$; c_3

Given the three transactions T_1, T_2, T_3, come up with the following schedules:

- a) A serial schedule
- b) A conflict serializable schedule (non-serial)
- c) A non-conflict serializable schedule
- d) A non-recoverable, non-serial schedule
- e) A cascade-free, non-serial schedule
- f) A strict, non-serial schedule

View Serializability

Let S and S' be two schedules with the same transactions. S and S' are ***view equivalent*** if these three conditions are met:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must also read the initial value of Q in schedule S'.
2. For each data item Q, if transaction T_i executes **`read`**(Q) in schedule S, and that value was produced by transaction T_j, then transaction T_i must also read the value of Q that was produced by transaction T_j in schedule S'.
3. For each data item Q, the transaction (if any) that performs the final **`write`**(Q) operation in schedule S must perform the final **`write`**(Q) in schedule S'.

Conditions 1 and 2 ensure each transaction reads the same values, and condition 3 ensures the same final result.

View Serializability (2)

A schedule *S* is ***view serializable*** if it is view equivalent to a serial schedule.

- Every conflict serializable schedule is also view serializable.
  - Every view serializable schedule which is not conflict serializable has *blind writes*. (A write without a read.)

This schedule is view serializable but *not* conflict serializable:

T_3	T_4	T_8
read(Q)		
	write(Q)	
write(Q)		
		write (Q)

Schedule is equivalent to serial schedule: $T_3 \Rightarrow T_4 \Rightarrow T_8$

Test for View Serializability

The precedence graph test for conflict serializability can be modified to test for view serializability:

- Construct a *labeled precedence graph.*
- Look for an acyclic graph that is derived from the labeled precedence graph by choosing one edge from every pair of edges with the same non-zero label. (2^n such graphs)
- Schedule is view serializable if and only if such an acyclic graph can be found.

The problem of looking for such an acyclic graph falls in the class of *NP*-complete problems.

- Thus existence of an efficient algorithm is unlikely.
  However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

Other Notions of Serializability

The schedule below produces the same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict or view equivalent.

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)

Why DO these schedules result in the same answer?

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

Concurrency Control and Serializability Tests

Testing a schedule for serializability ***after*** it has executed is a little too late!

The goal is to develop concurrency control protocols that will ensure serializability.

- They do not use the precedence graph as it is being created.
- Instead a protocol will impose a discipline that avoids non-serializable schedules.

Tests for serializability help understand why a concurrency control protocol is correct.

Transaction Management Summary

A ***transaction*** is a unit of program execution that accesses and may update data values and must be executed atomically.

Transactions should demonstrate the ***ACID properties***:

- atomicity, consistency, isolation, and durability

A ***schedule*** is the sequence of operations (possibly interleaved) from multiple concurrent transactions. A schedule is serializable if it can be proven equivalent to a serial schedule.

- Two types: conflict serializability and view serializability
- Tests for conflict serializability involves defining a precedence graph and checking for cycles.
- A schedule may also be recoverable, cascade-free, or strict.

Serializability tests are re-active, concurrency control protocols are pro-active. (prevent non-serializability)

Major Objectives

The "One Things":

- List and explain the ACID properties of transactions.
- Test for conflict serializability using a precedence graph.

Major Theme:

- Transactions are used to guarantee a set of operations are performed in an atomic manner. The DBMS must ensure interleaving of concurrent transactions is (conflict) serializable using a concurrency control method.

Objectives

- Define: transaction, atomic, consistent, constraint
- Explain the two challenges in preserving consistency.
- List and explain the ACID properties of transactions.
- Write a transaction using read/write operations.
- List the transactions states and draw the state diagram.
- Define schedules and serial schedules.
- List three problems that motivate concurrency control.
- Define conflict serializability and conflicting operations.
- Test for conflict serializability using a precedence graph.
- Define, recognize, and create examples of recoverable, cascade-free, and strict schedules.
- Draw the Venn diagram for schedules.

Objectives (2)

- Define view serializability and the 3 rules for view equivalent schedules.
- Define and give an example of a blind write.
- Recognize and create view serializable schedules.


UBC
THE UNIVERSITY OF BRITISH COLUMBIA