COL362-632: Transaction Management

April 5, 2023

Transaction

- A logical unit of a program that reads and possibly updates data items.
- Data item at any granularity: token, tuple, relation

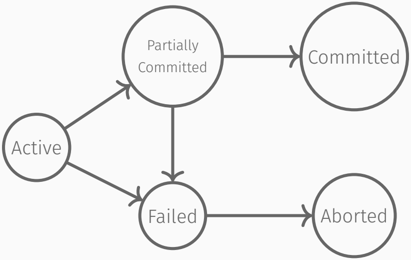
Transaction

- A logical unit of a program that reads and possibly updates data items.
- Data item at any granularity: token, tuple, relation

$A = 1000$ and $B = 950$

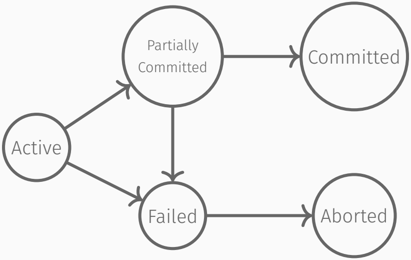
```
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);
```

Life cycle of a transaction



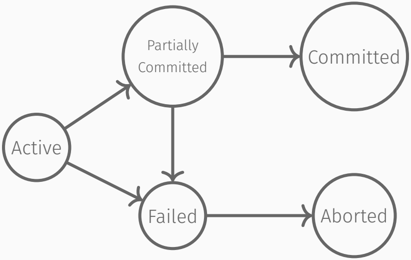
- Active: Executing instructions.

Life cycle of a transaction



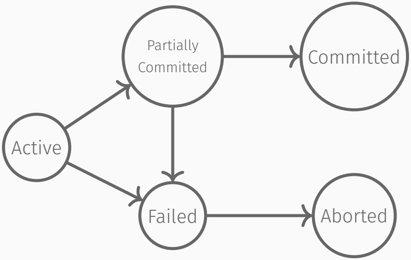
- Active: Executing instructions.
- Partially Committed: All instructions executed, waiting for extra information and changes to be written on disk (DB).

Life cycle of a transaction



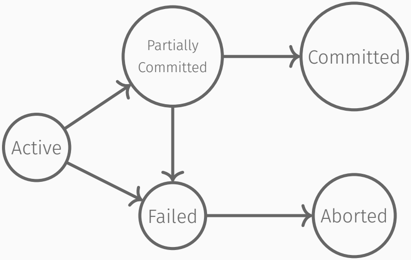
- Active: Executing instructions.
- Partially Committed: All instructions executed, waiting for extra information and changes to be written on disk (DB).
- Committed: After the changes reflected in DB.

Life cycle of a transaction



- Active: Executing instructions.
- Partially Committed: All instructions executed, waiting for extra information and changes to be written on disk (DB).
- Committed: After the changes reflected in DB.
- Failed: Transaction failed due to a failure/error.

Life cycle of a transaction



- Active: Executing instructions.
- Partially Committed: All instructions executed, waiting for extra information and changes to be written on disk (DB).
- Committed: After the changes reflected in DB.
- Failed: Transaction failed due to a failure/error.
- Aborted: Transaction has failed and is rolled backed.

Potential Issues

```
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);
<commit>
```

- Transaction fails after debiting A – money lost!

Potential Issues

```
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);
<commit>
```

- Transaction fails after debiting A – money lost!
- Logical error: $A := A + 50$ – application specified consistency violated!

Potential Issues

```
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);
<commit>
```

- Transaction fails after debiting A – money lost!
- Logical error: $A := A + 50$ – application specified consistency violated!
- Another transaction is computing the average monthly balance of A – can generate the wrong value!

Potential Issues

```
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);
<commit>
```

- Transaction fails after debiting A – money lost!
- Logical error: $A := A + 50$ – application specified consistency violated!
- Another transaction is computing the average monthly balance of A – can generate the wrong value!
- DB crashes after the transaction has commit operation: changes should not lost!

Desirable characteristics of transaction

- Atomicity: Either all the operations in a transaction are executed or none at all.

Desirable characteristics of transaction

- Atomicity: Either all the operations in a transaction are executed or none at all.
- Consistency: Execution of a transaction must preserve the consistency of the DB.

Desirable characteristics of transaction

- Atomicity: Either all the operations in a transaction are executed or none at all.
- Consistency: Execution of a transaction must preserve the consistency of the DB.
- Isolation: Each transaction in the system is unaware of the other concurrent transactions.

Desirable characteristics of transaction

- Atomicity: Either all the operations in a transaction are executed or none at all.
- Consistency: Execution of a transaction must preserve the consistency of the DB.
- Isolation: Each transaction in the system is unaware of the other concurrent transactions.
- Durability: The changes made by a successful transaction should persist against any system failure.

Popularly known as **ACID** properties

Components responsible for ensuring these properties

- Atomicity: Recovery System
- Consistency: Responsibility of a programmer; Beyond data-integrity constraints;
- Isolation: Concurrency control scheme
- Durability: Recovery System

Schedules

- Schedule: A chronological sequence of instructions of concurrent transactions.

Schedules

- Schedule: A chronological sequence of instructions of concurrent transactions.

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

Schedules

- Schedule: A chronological sequence of instructions of concurrent transactions.

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

	$\mathbf{T_1}$	$\mathbf{T_2}$
i_0	read(A)	
i_1		read(A)
i_2		write(A)
i_3	write(A)	
i_4	read(B)	
i_5	write(B)	
i_6		read(B)
i_7		write(B)

Schedules

- Schedule: A chronological sequence of instructions of concurrent transactions.

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

	$\mathbf{T_1}$	$\mathbf{T_2}$
i_0	read(A)	
i_1		read(A)
i_2		write(A)
i_3	write(A)	
i_4	read(B)	
i_5	write(B)	
i_6		read(B)
i_7		write(B)

- Are concurrent transactions the same as concurrent processes? – ordering of Read and Write operations is crucial.

Concurrent transactions

- $A = 1000$ and $B = 950$

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

$\mathbf{T_1}$	$\mathbf{T_2}$
read(A)	
	read(A)
	write(A)
write(A)	
read(B)	
write(B)	
	read(B)
	write(B)

Concurrent transactions

- $A = 1000$ and $B = 950$

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

$\mathbf{T_1}$	$\mathbf{T_2}$
read(A)	
	read(A)
	write(A)
write(A)	
read(B)	
write(B)	
	read(B)
	write(B)

- Lost updates: Changes made by a transaction are overwritten by another.

Concurrent transactions

- $A = 1000$ and $B = 950$

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

$\mathbf{T_1}$	$\mathbf{T_2}$
read(A)	
	read(A)
	write(A)
write(A)	
read(B)	
write(B)	
	read(B)
	write(B)

- Lost updates: Changes made by a transaction are overwritten by another.
- Dirty reads: A transaction reads the changes made by a failed transaction.

Serial Schedules

- Serial schedules: Execute all instructions of a transaction followed by the instructions of the next transaction.

Serial Schedules

- Serial schedules: Execute all instructions of a transaction followed by the instructions of the next transaction.
  - S1: T1 followed by T2
  - S2: T2 followed by T1

T_1
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

T_2
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

Are serial schedules the real solution?

- Ensures transaction isolation perfectly.
- If execution is serial then the concurrency is not actually supported.

Are serial schedules the real solution?

- Ensures transaction isolation perfectly.
- If execution is serial then the concurrency is not actually supported.
- Low throughput and under-utilized resources.
- Interleaving of instructions is desired.

Requirement of “serial non-serial” schedules

Requirement of “serial non-serial” schedules

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

$\mathbf{T_1}$	$\mathbf{T_2}$
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Requirement of “serial non-serial” schedules

$\mathbf{T_1}$
read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);

$\mathbf{T_2}$
read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B);

$\mathbf{T_1}$	$\mathbf{T_2}$
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

- Not all non-serial schedules lead to inconsistency!
- How to identify which schedules are ”allowed”?

Serializability

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

Serializability

- A schedule is called *serializable* if it is equivalent to a serial schedule.
- Two types of equivalence:
    - Conflict equivalence
    - View equivalence

Serializability

- A schedule is called *serializable* if it is equivalent to a serial schedule.
- Two types of equivalence:
  - Conflict equivalence
  - View equivalence
- A schedule S is conflict (view) serializable if it is conflict (view) equivalent to a serial schedule.

Conflicting instructions

- Let I and J be two instructions from different transactions T_i and T_j :

Conflicting instructions

- Let I and J be two instructions from different transactions T_i and T_j :
    1. $I = R_i(X)$ and $J = R_j(X)$: No conflict.

Conflicting instructions

- Let I and J be two instructions from different transactions T_i and T_j :
  1. $I = R_i(X)$ and $J = R_j(X)$: No conflict.
  2. $I = R_i(X)$ and $J = W_j(X)$: Value read by T_i is dependent on the order of I and J.

Conflicting instructions

- Let I and J be two instructions from different transactions T_i and T_j :
  1. $I = R_i(X)$ and $J = R_j(X)$: No conflict.
  2. $I = R_i(X)$ and $J = W_j(X)$: Value read by T_i is dependent on the order of I and J.
  3. $I = W_i(X)$ and $J = R_j(X)$: Similar as previous case.

Conflicting instructions

- Let I and J be two instructions from different transactions T_i and T_j :
  1. $I = R_i(X)$ and $J = R_j(X)$: No conflict.
  2. $I = R_i(X)$ and $J = W_j(X)$: Value read by T_i is dependent on the order of I and J.
  3. $I = W_i(X)$ and $J = R_j(X)$: Similar as previous case.
  4. $I = W_i(X)$ and $J = W_j(X)$: Order of I and J decides the final value of X

Conflicting Instructions

- Two instructions I and J are said to be conflicting if
  - I and J are from two different transactions
  - Both operate on the same data item
  - At least one of them is a *write* operation

Conflicting Instructions

- Two instructions I and J are said to be conflicting if
  - I and J are from two different transactions
  - Both operate on the same data item
  - At least one of them is a *write* operation

$\mathbf{T_1}$	$\mathbf{T_2}$
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Conflict Serializability

- Conflicting instructions impose a logical temporal ordering.
- Ordering of conflicting institutions should be preserved in an equivalent schedule.
- Consequently, non-conflicting instructions can be swapped.

Conflict Serializability

- Conflicting instructions impose a logical temporal ordering.
- Ordering of conflicting institutions should be preserved in an equivalent schedule.
- Consequently, non-conflicting instructions can be swapped.
- If a schedule S can be transformed into a schedule S’ by a series of swaps of non-conflicting instructions, then S and S’ are conflict equivalent.

Conflict Serializability

- Conflicting instructions impose a logical temporal ordering.
- Ordering of conflicting institutions should be preserved in an equivalent schedule.
- Consequently, non-conflicting instructions can be swapped.
- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, then S and S' are conflict equivalent.
- If S is *conflict* equivalent of a serial schedule, the S is said to be conflict serializable.

View Serializability

- Two schedules are said to be view equivalent if they produce the same output (view).
- Two schedules S and S' are view equivalent if (for each data item X),

View Serializability

- Two schedules are said to be view equivalent if they produce the same output (view).
- Two schedules S and S' are view equivalent if (for each data item X),
    1. Same transaction reads the initial value of X in both S and S'.

View Serializability

- Two schedules are said to be view equivalent if they produce the same output (view).
- Two schedules S and S' are view equivalent if (for each data item X),
    1. Same transaction reads the initial value of X in both S and S'.
    2. Same transaction performs the final write(X) operation in both schedules.

View Serializability

- Two schedules are said to be view equivalent if they produce the same output (view).
- Two schedules S and S’ are view equivalent if (for each data item X),
    1. Same transaction reads the initial value of X in both S and S’.
    2. Same transaction performs the final write(X) operation in both schedules.
    3. If a transaction T_i reads a value of X produced by write(X) operation of transaction T_j in schedule S, then the same should hold in schedule S’.

View and Conflict serializability relation

T_1	T_2	T_3
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- View equivalent to serial schedule $S = \{T1, T2, T3\}$

View and Conflict serializability relation

T_1	T_2	T_3
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- View equivalent to serial schedule $S = \{T1, T2, T3\}$
- Is it conflict serializable?

View and Conflict serializability relation

T_1	T_2	T_3
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- View equivalent to serial schedule $S = \{T1, T2, T3\}$
- Is it conflict serializable?
- Not conflict serializable because no pair of non-conflicting instructions!

Not all view serial schedules are conflict serializable, but all conflict serializable schedules are view serializable.

Serializability Test

- Precedence graph: A directed graphs $G(V, E)$ with $V = \{T_i, \ldots, T_j\}$ and and edge from transaction T_i to T_j exists if

Serializability Test

- Precedence graph: A directed graphs $G(V, E)$ with $V = \{T_i, \ldots, T_j\}$ and and edge from transaction T_i to T_j exists if
  - $R_i(X)$ precedes $W_j(X)$

Serializability Test

- Precedence graph: A directed graphs $G(V, E)$ with $V = \{T_i, \ldots, T_j\}$ and and edge from transaction T_i to T_j exists if
  - $R_i(X)$ precedes $W_j(X)$
  - $W_i(X)$ precedes $R_j(X)$

Serializability Test

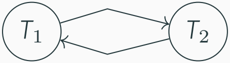
- Precedence graph: A directed graphs $G(V, E)$ with $V = \{T_i, \ldots, T_j\}$ and and edge from transaction T_i to T_j exists if
  - $R_i(X)$ precedes $W_j(X)$
  - $W_i(X)$ precedes $R_j(X)$
  - $W_i(X)$ precedes $W_j(X)$

Serializability Test

- Precedence graph: A directed graphs $G(V, E)$ with $V = \{T_i, \dots, T_j\}$ and and edge from transaction T_i to T_j exists if
  - $R_i(X)$ precedes $W_j(X)$
  - $W_i(X)$ precedes $R_j(X)$
  - $W_i(X)$ precedes $W_j(X)$

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

- $read_1(A) < write_2(A)$
- $write_2(A) < write_1(A)$



Serializability Testing

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

Serializability Testing

- A schedule is conflict serializable *if and only if* the precedence graph is acyclic.
- Detecting a cycle in a graph: O(n+m) time.
- Topological sorting over the precedence graph provides the equivalent serial schedule.

Serializability Testing

- A schedule is conflict serializable *if and only if* the precedence graph is acyclic.
- Detecting a cycle in a graph: O(n+m) time.
- Topological sorting over the precedence graph provides the equivalent serial schedule.
- Testing view serializability is NP-complete – no efficient algorithm exists.