Transactions, Concurrency Control, Recovery

Chapters 17, 18, 19

Ignore SQL

Forget about SQL **insert** and **delete** operations.

The operations are performed on **data items** (= single data value) typically named as A, B, C...

Transactions access data using two operations:

- **read(X)**
- **write(X)**

Transaction

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

Examples:

1. **read(A)**
2. **read(B)**

1. **A := 100**
2. **B := 20**
3. **write(A)**
4. **write(B)**

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

Potential problems:

1) Failures of various kinds: hardware failures and system crashes
2) Concurrent execution of multiple transactions

ACID properties

Who is responsible?

Atomicity - Either all operations of the transaction are properly reflected in the database or none are. → **Recovery System**

Consistency - Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. → **Concurrency-Control System**

Isolation - Each transaction must be unaware of other concurrently executing transactions. → **Concurrency-Control System**

Durability - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures. → **Recovery System**

Schedules

A sequence of instructions that specify the order in which instructions of concurrent transactions are executed.

1) Must include all instructions of those transactions.
2) Must preserve the order in which the instructions appear in each individual transaction.

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).

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

Serial Schedules

One transaction is executed completely before starting another transaction.

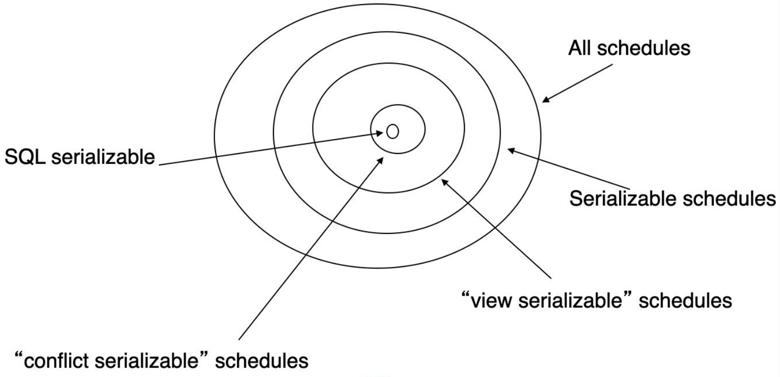
How to ensure consistency of the database under concurrent execution?
Any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution = serial schedule.

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

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	

Serializability

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



Serial schedule

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

Are these schedules serializable?

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

2

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

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.

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

Conflict Serializability (cont.)

If instructions refer to different data items, then we can swap them.

Otherwise...
Which instructions are ‘swappable’ (non-conflicting)?

1) READ/READ: no conflict
2) WRITE/WRITE: conflict because the value left in db depends on which write occurred last
3) READ/WRITE: conflict because the value read depends on whether write has occurred

Is this schedule conflict serializable?

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

Conflict Serializability (cont.)

How to test conflict serializability in a more efficient way?

Precedence Graph - a directed graph where the vertices are the transactions and the edges are the conflicts.

If it is **cyclic** -> NOT conflict serializable
It it is **acyclic** -> the serializability order can be obtained by topological sorting

Does a precedence graph for this schedule have a cycle?

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

View Serializability

S and S' are view equivalent if the following three conditions are met, for each data item Q:

1) For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S´, also read the initial value of Q.
2) For each data item Q, if transaction Ti reads the value of Q written by Tj in S, it also does in 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) operation in schedule S´.

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

View serializable but NOT conflict serializable →

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

Recoverable Schedules

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

Recoverable schedule — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti appears before the commit operation of Tj

This schedule is not recoverable.
How to make it recoverable?

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

Cascading Rollbacks

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks.

T_8	T_9	T_{10}
read(A) read(B) write(A)		
	read(A) write(A)	
		read(A)
abort		

Cascadeless Schedules

Cascadeless schedules — cascading rollbacks cannot occur if for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj.

How to make this schedule cascadeless?

NOTE: Every cascadeless schedule is also recoverable

Concurrency Control

Concurrency-control protocols impose a discipline that avoids non-serializable schedules.

Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.

Lock-Based Protocols

A **lock** is a mechanism to control concurrent access to a data item.

Two types of locks:

1) Exclusive (X) mode "**lock-X**": Data item can be both read as well as written.
2) Shared (S) mode "**lock-S**": Data item can only be read.

Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

Deadlock

Can T3 or T4 make progress? No!

Deadlock - a state where neither of these transactions can ever proceed with its normal execution.

To handle a deadlock one of T3 or T4 must be rolled back and its locks released. -> Costly

Another issue: **Starvation**

- T1 holds shared lock on Q
- T2 requests exclusive lock on Q: blocks
- T3, T4, ..., Tn request shared locks: granted
- T2 is starved!

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Schedule With Lock Grants

Not serializable!

T1	T2
read(B) B := B - 50 write(B)	
	read(A) read(B) display(A+B)
read(A) A := A - 50 write(A)	

Is is serializable now?

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B) $B := B - 50$ write(B) unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A) unlock(A) lock-S(B)	
		grant-S(B, T_2)
	read(B) unlock(B) display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A) $A := A - 50$ write(A) unlock(A)		

The Two-Phase Locking Protocol

Locks do not ensure serializability by themselves

-> Need a protocol!

Phase 1: **Growing Phase**

- Transaction may obtain locks
- Transaction may not release locks

Phase 2: **Shrinking Phase**

- Transaction may release locks
- Transaction may not obtain locks

The protocol ensures serializability. Transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock)

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

The Two-Phase Locking Protocol (cont.)

Problems:

- **Deadlock** is not prevented -> *Solution* - Deadlock prevention protocols
- **Cascadeless rollback** is not prevented -> *Solution* - **Strict Two-Phase Locking Protocol**

Strict Two-Phase Locking Protocol

Exclusive locks must be held until transaction commits.

- Ensures any data written by uncommitted transaction not read by another -> Recoverable

Deadlock Handling

Deadlock prevention protocols ensure that the system will never enter into a deadlock state.

Some prevention strategies:

- Require that each transaction locks all its data items before it begins execution (predeclaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol)

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock Handling (cont.)

Wait-die scheme — non-preemptive

- Older transaction may wait for younger one to release data item.
- Younger transactions never wait for older ones; they are rolled back instead.
- A transaction may die several times before acquiring a lock

Wound-wait scheme — preemptive

- Older transaction wounds (forces rollback) of younger transaction instead of waiting for it.
- Younger transactions may wait for older ones.

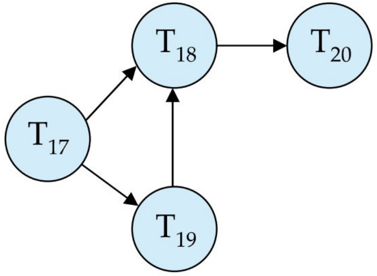
A rolled back transactions is restarted with its original timestamp.

Deadlock Detection

Wait-for graph

- Vertices: transactions
- Edge from Ti →Tj if Ti is waiting for a lock held in conflicting mode by Tj

The system is in a deadlock state if and only if the wait-for graph has a cycle.



T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Timestamp-Based Protocols

Each transaction Ti is issued a timestamp TS(Ti) when it enters the system.
Timestamp-based protocols manage concurrent execution such that

time-stamp order = serializability order

Maintains for each data Q two timestamp values:

- **W-TS(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.
- **R-TS(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully

Imposes rules on read and write operations to ensure that

- any conflicting operations are executed in timestamp order
- out of order operations cause transaction rollback

Timestamp-Based Protocols (cont.)

Assume that initially:
R-TS(A) = W-TS(A) = 0
R-TS(B) = W-TS(B) = 0
And TS(T25) = 25 and TS(T26) = 26

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

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

- When a transaction is rolled back the system assigns it a new timestamp and restarts it.
- The timestamp-ordering protocol guarantees serializability.
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recovery

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures.

Recovery algorithms have two parts:

1) Actions taken during normal transaction processing to ensure enough information exists to recover from failures.
2) Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

Log-Based Recovery

A **log** is a sequence of **log records**. The records keep information about update activities on the database.

Example:

<Ti start >

<Ti , X, V1, V2>

<Ti commit >

Transaction T1	Log
Read(A) A =A-50 Write(A) Read(B) B = B+50 Write(B)	<T1, start> <T1, A, 1000, 950> <T1, B, 2000, 2050> <T1, commit>

Deferred Database Modification

- All logs are written on to the stable storage and the database is updated only when a transaction commits.
- <Ti, X, V> to log: V is the new value for X (the old value is not needed)
- DB updates by reading and executing the log:
  <Ti start> …… <Ti commit>

Recovery:

- Redo: if both <Ti start> and <Ti commit> are there in the log.

What is the correct recovery action?

<T0, start>
<T0, B, 20, 30>
<T0, A, 10, 5>
<T0, commit>
<T1, start>
<T1, C, 80, 100>
<T1, A, 5, 16>

Immediate Database Modification

- Database updates of an uncommitted transaction are allowed
- Log records must be of the form: <Ti, X, Vold, Vnew >
- Output of DB blocks can occur before or after commit in any order

Recovery:

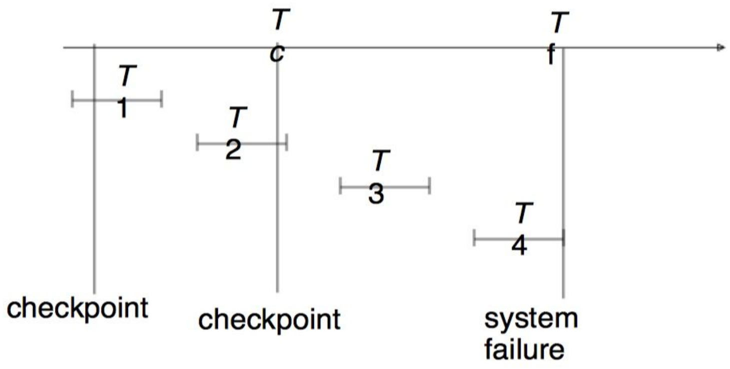
- Undo : if <Ti, start > is in the log but <Ti commit> is not.
- Redo: if <Ti start> and <Ti commit> are both in the log.

What is the correct recovery action?

<T0, start>
<T0, B, 20, 30>
<T0, A, 10, 5>
<T0, commit>
<T1, start>
<T1, C, 80, 100>
<T1, A, 5, 16>

Checkpoints

Objective: avoid redundant redo operations
How: Put marks in the log indicating that at that point **DB and log are consistent**.



Recovery With Concurrent Transactions

<checkpoint L> L: the list of transactions active at the time of the checkpoint

1. Initialize **undo-list** and **redo-list** to empty
2. Scan the log backwards from the end, stopping when the first <checkpoint L> record is found.
3. For each record found during the backward scan:
   - if the record is <Ti commit>, add Ti to redo-list
   - if the record is <Ti start>, then if Ti is not in redo-list, add Ti to undo-list
4. For every Ti in L, if Ti is not in redo-list, add Ti to undo-list

Recovery With Concurrent Transactions (cont.)

LOG:
<T0 **start**>
<T0, A, 0, 10>
<T0 **commit**>
<T1 **start**>
<T1, B, 0, 10>
<T2 **start**>
<T2, C, 0, 10>
<T2, C, 10, 20>
<checkpoint {T1, T2}>
<T3 **start**>
<T3, A, 10, 20>
<T3, D, 0, 10>
<T3 **commit**>

Recovery With Concurrent Transactions (cont.)

LOG:
<T0 **start**>
<T0, A, 0, 10>
<T0 **commit**>
<T1 **start**>
<T1, B, 0, 10>
<T2 **start**>
<T2, C, 0, 10>
<T2, C, 10, 20>
<checkpoint {T1, T2}>
<T3 **start**>
<T3, A, 10, 20>
<T3, D, 0, 10>
<T3 **commit**>

Undo-list: T1, T2
Redo-list: T3
Ignore: T0

Undo:
Set C to 10
Set C to 0
Set B to 0

Redo:
Set A to 20
Set D to 10

After recovery:
A: 20, B: 0, C:0, D: 10