Database and Information Systems

Course Roadmap

Chapter 1	Introduction to Databases
Chapter 2	Integrity Constraints and ER Model
Chapter 3	Relational Databases and Schema Refinement
Chapter 4	Query Language
Chapter 5	Transaction and Concurrency Control
Chapter 6	Indexing

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
  - **Set of operations used to perform a logical unit of work**
  - Generally represent change in Database
  - Example: Ticket booking, bank money transfer
- E.g., transaction to transfer 1000 INR from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 1000
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 1000
  6. **write**(*B)*
  7. **commit**
- Two main issues to deal with:
  - Failures, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- Transaction to transfer 1000 INR from account A to account B: A=2000, B=2000
  1. **read**(*A*)
  2. *A* := *A* – 1000
  3. **write**(*A*) A= 1000
  4. **read**(*B*)
  5. *B* := *B* + 1000
  6. **write**(*B)*
  7. **Commit**
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 7, 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
    - Either All Operations or No Operations
- **Durability requirement** — Once the user has been notified that the transaction has completed (i.e., the transfer of the 1000 INR has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures. Permanent changes in database after a transaction is completed successfully

Example of Fund Transfer (Cont.)

- **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
    - 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

Before a transaction starts and after the transaction completed, sum of the money should be same.

Example of Fund Transfer (Cont.)

- **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 by running transactions **serially(**one after other**)**
  - Can we convert two parallel transactions in serial transaction conceptually. Why to convert: Serial Schedules are always **consistent**
- But, executing multiple transactions concurrently has significant benefits
  - Increased processor and disk utilization
  - Reduced average response time

ACID Properties

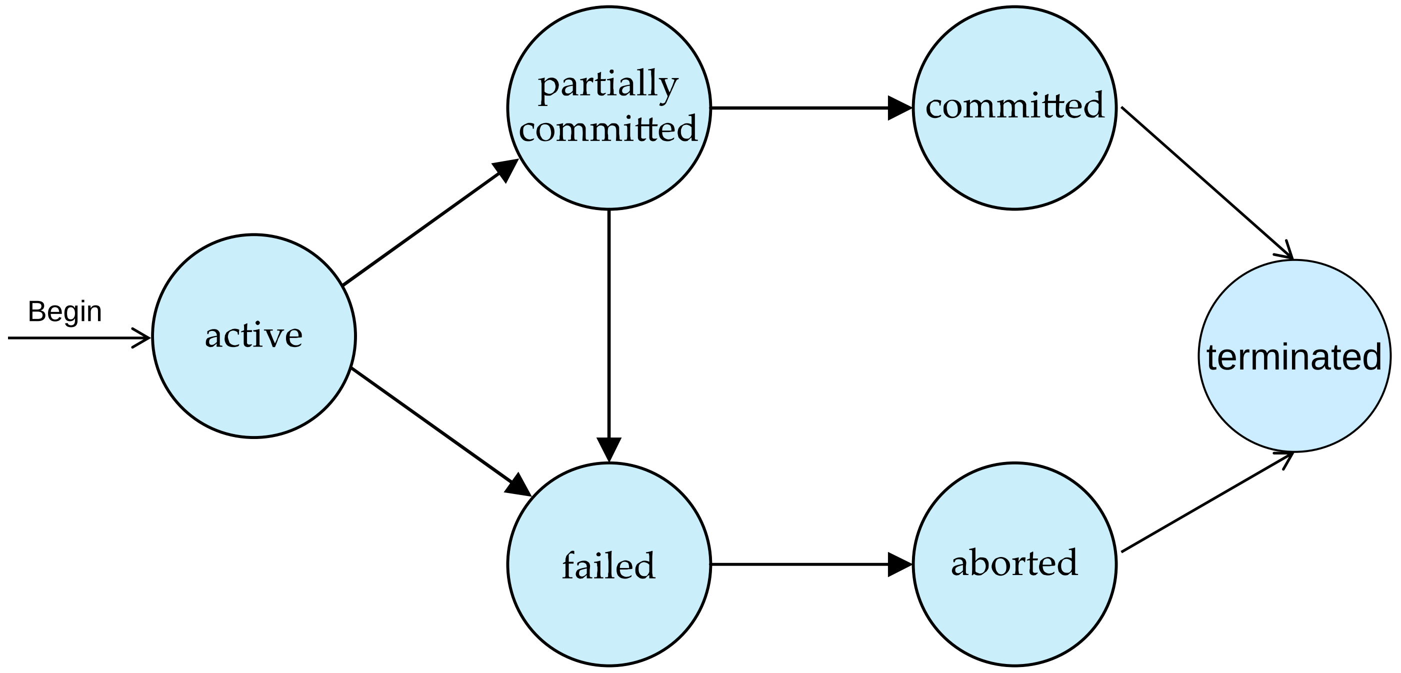
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

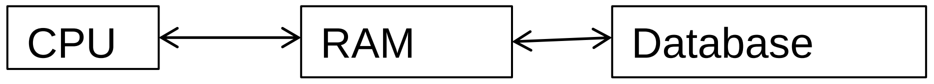
- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Before a transaction starts and after the transaction completed, sum of the money (value) should be same.
  - Execution of a transaction in isolation preserves the consistency 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 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. That is, running transactions **serially.**
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; a transaction stays in this state while it is executing
  - transaction is main memory and it is getting executed
- **Partially committed** – after the final statement has been executed
  - transaction has completed all the operations except commit
- **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
- **Terminated** – free the resources that were are used by transaction

Transaction State (Cont.)





What would be the problem if CPU directly interacts with HD?

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/operations 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 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
- Schedules are two types: Serial Schedule and Parallel Schedule

Serial Schedule vs. Parallel Schedule

Serial Schedule	**Parallel Schedule**
Transactions execute one after other	Transactions execute concurrently
Consistent	Can be Inconsistent
High waiting time	Less waiting time
Low throughput	High throughput
Low performance	High performance

Users always want high performance and the same time data, result or transactions should be consistent

Schedule 1

- Let T_1 transfer 50 INR 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

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	

Schedule 3

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

Buffer	Datbase

A=45
B=155
	A=100
	B=100

- 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).

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

T1	T2
1. R(A)	
2. A = A-50	
	3. R(A)
	4. Temp = A*0.1
	5. A = A - temp
	6. W(A)
	7. R(B)
8. W(A)	
9. R(B)	
10. B = B +50	
11. W(B)	
12. commit	
	13. B = B + temp
	14. W(B)
	15. commit

Buffer	Datbase
A=50	A=100
B= 110	B=100

Conflicts

- Conflicts may occur when read or write operations are performed on **same data** by **different transactions**

T1	T2	Conflict
R(A)	R(A)	No
R(A)	W(A)	Yes
W(A)	R(A)	Yes
W(A)	W(A)	Yes

Conflicts

- Example of conflict schedule

T1	T2
R(A)	
A=A-1	
	R(A)
	A=A-1
	W(A)
	commit
W(A)	
commit	

Table: Conflict Schedule

Irrecoverable Schedules

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

- Let us consider the following example:

T1	**T2**
R(A)	
A=A-1	
W(A)	
	R(A)
	A=A-1
	W(A)
	commit
*Fail	

Table: Irrecoverable Schedule

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.
  - **Transactions in a Schedule have ability to recover if any failure happens**
- The following schedule is not recoverable

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

- 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.

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		

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
- The above schedule is cascading schedule
- Problem due to cascading rollback
  - Poor CPU utilization
  - Less performance

Cascadeless Schedules

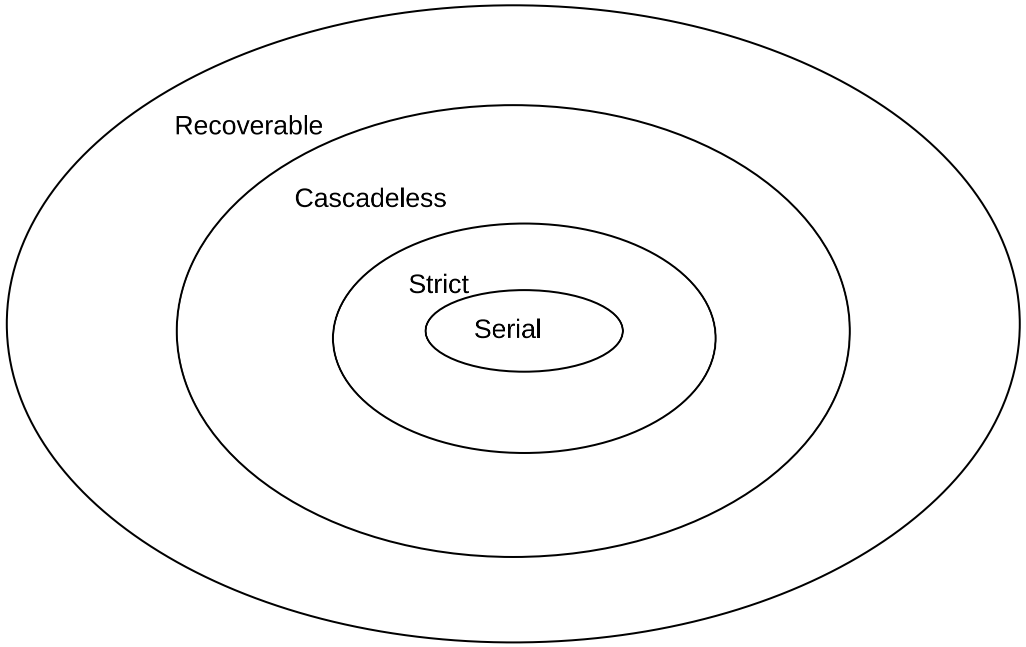
- **Cascadeless schedules** — cascading rollbacks cannot occur:
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
  - No Write-Read problem
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

T1	**T2**
R(A)	
A=A-1	
W(A)	
commit	
	R(A)
	commit

Table: Cascadeless Schedule

Strict Recoverable Schedule

- T_i writes before T_j writes or reads, then T_j must read or write after T_i commits or aborts then only the schedule will be strict recoverable.
  - In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.
  - Strict schedule is strict in nature



Note: In exam, you may get a question to check if a schedule is recoverable, irrecoverable, cascading, cascadeless, and strict recoverable

Example

Question: Name the type of this partial schedule?

T1	T2
R(A)	
A=A-1	
W(A)	
commit	
	R(A)
	W(A)

Table: ……….. Schedule

strict recoverable?

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.

T1	**T2**
R(A)	
	R(A)
	W(A)
W(B)	

Parallel Schedule S

T1	**T2**
R(A)	
W(B)	
	R(A)
	W(A)

Serial Schedule S1

or

T1	**T2**
	R(A)
	W(A)
R(A)	
W(B)	

Serial Schedule S2

- Serializability: For a parallel schedule, find a schedule that should be serial
  - **Can we convert a parallel schedule to a serial schedule. Can we convert S to S1 or S2. If yes, then S is a serializable schedule.**
- 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:
  1. **Conflict serializable**
  2. **View serializable**

Serializability

- For a given schedule of three transactions, how many possible serializable schedules?

 6?

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.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- If $\boldsymbol{l_i}$ **and** $\boldsymbol{l_j}$ **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

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

Can you check if the schedule is a conflict serializable?

conflict kaha hai?

Conflict Serializability

- Can you check if the schedule is a conflict serializable?

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

Conflict Serializability

- First, we will understand conflict equivalent.
- 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**.

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

- Can you check if two schedules are conflict equivalent?

Note: It would be useful to **check adjacent non-conflict pair of two transactions** and can swap their position to make serial schedule

Conflict Serializability (Cont.)

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
	R(B)
	W(B)

T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
	R(B)
	W(B)

T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
	R(B)
	W(B)

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Note: It would be useful to check adjacent non-conflict pair of two transactions and can swap their position to make serial schedule

Conflict Serializability (Cont.)

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule
  - For a schedule S, there can be many conflict equivalent schedules. If any conflict equivalent is a serial schedule, then given schedule is conflict serializable
- 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

Conflict Serializability (Cont.)

- Can you tell if given schedule (Se) is conflict serializable:

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

Schedule Se

Conflict Serializability (Cont.)

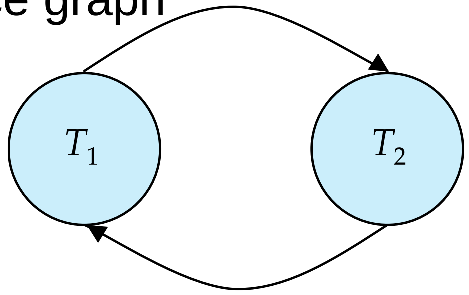
- Example of a schedule that is not conflict serializable:

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 >.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- We will make precedence graph
- **Precedence graph** — a direct 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.
  - Check the conflict pair with other transaction in schedule and draw edge
- We may label the arc by the item that was accessed
- Example of a precedence graph

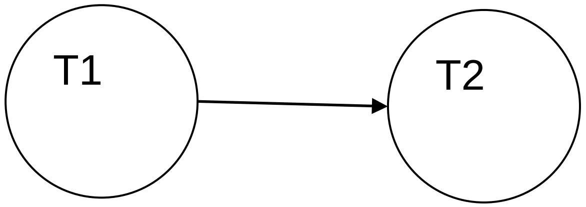


Check conflict pairs in other transactions and draw edges

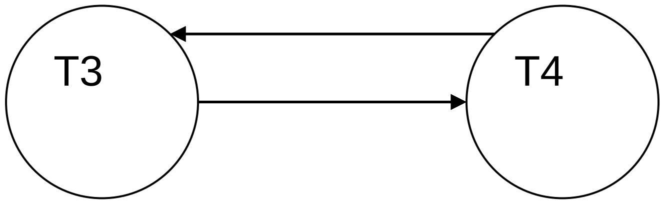
Testing for Serializability

- Check if the following schedules are conflict serializable

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

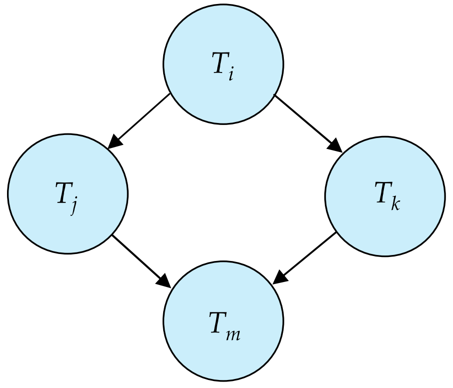


T_3	T_4
read (Q)	
	write (Q)
write (Q)	

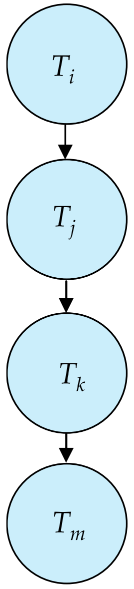


Test for Conflict Serializability

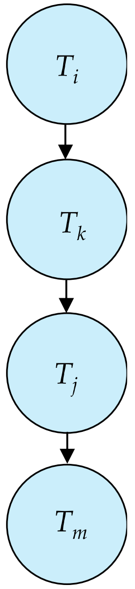
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
  - If no loop or cycle in precedence graph then schedule is conflict serializable
- Serializability order can be obtained by a *topological sorting* of the graph.
  - Sort based on indegree
    - Lowest indegree node will be first in sequence



(a)



(b)



(c)

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q* (**Initial Read**).
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j (**WR Sequence**).
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'* **(Final Write)**.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

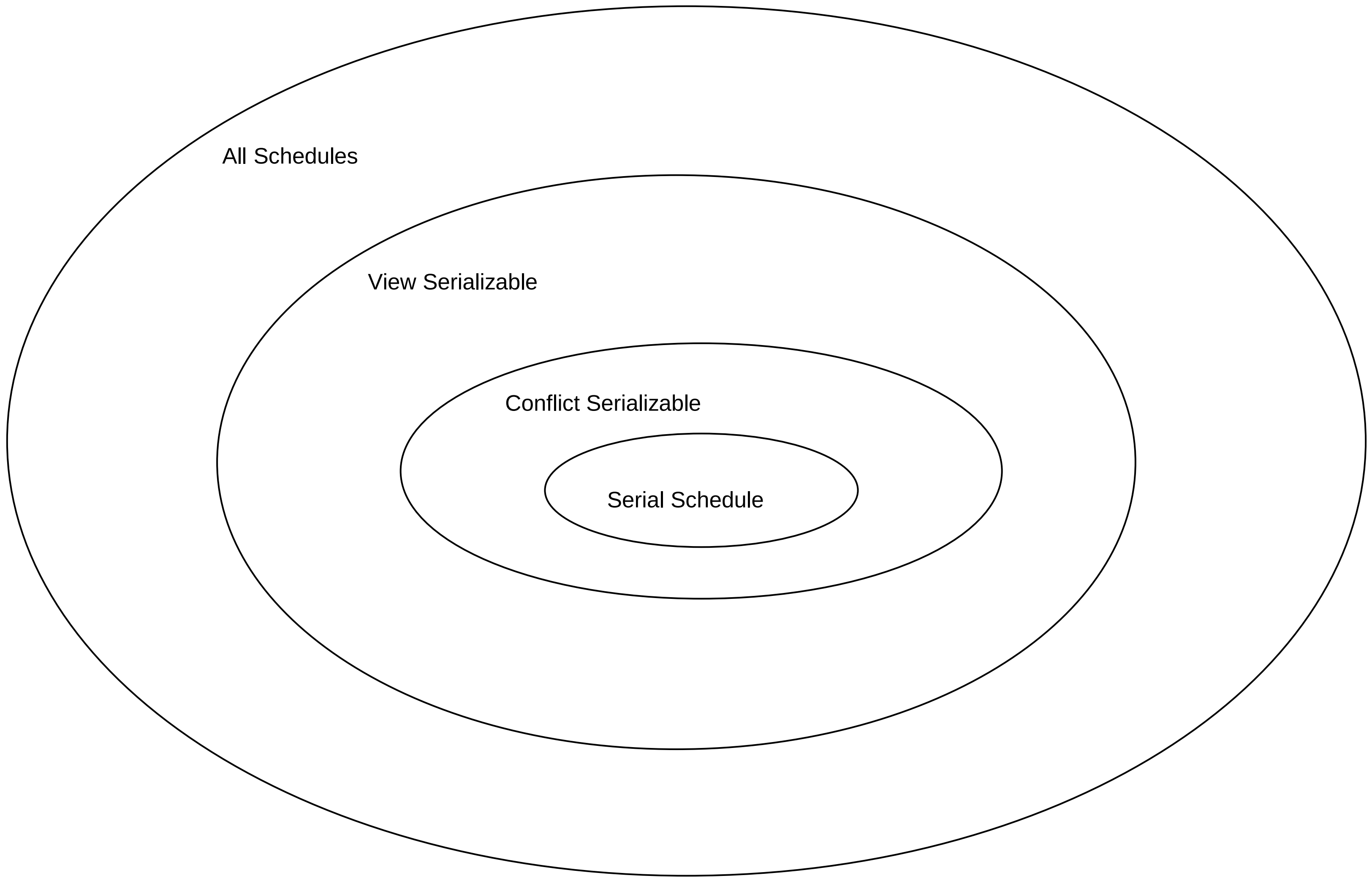
View Serializability (Cont.)

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

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

- What serial schedule is above equivalent to?

Expressive Power of Schedules



Questions on Serializability

- Check if the given schedule is view serializable or not?

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

Questions on Serializability

- Check if the given schedule is view serializable or not?

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

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

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.

Concurrency Control (Cont.)

- 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.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control Protocols

- Shared-Exclusive Locking
- Two Phase Locking (2PL)
- Basic Timestamp Ordering Protocol

Shared-Exclusive Locking

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
     If a transaction locked the data item in exclusive mode, then that transaction is allowed to perform both read and write
  2. **shared** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
     If a transaction locked the data item in shared mode, then that transaction is allowed to perform only read operation
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Shared-Exclusive Locking(Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking is *not sufficient* to guarantee serializability.

T_1	T_2	concurrency-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)		

Deadlock

- Consider the partial schedule

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)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. Example: A only common room in family
- Concurrency control manager can be designed to prevent starvation.

T1	**T2**	**T3**	**T4**
	S(A)		
X(A)			
..			
..		S(A)	
..	U(A)		
..			
..			S(A)
		U(A)	

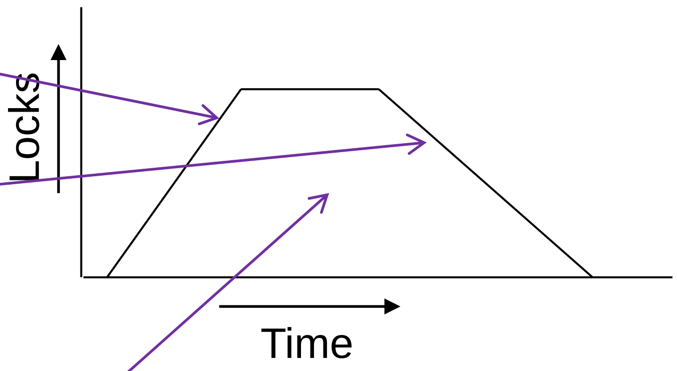
Irrecoverability

- Shared-Exclusive Locking may suffer from irrecoverability

T1	**T2**
X(A)	
R(A)	
W(A)	
U(A)	
..	
..	S(A)
..	R(A)
..	U(A)
	Commit
Fail/Abort	

The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction obtain locks
  - Transaction do not release locks
- Phase 2: **Shrinking Phase**
  - Transaction release locks
  - Transaction do not obtain locks
- The protocol **assures serializability**. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its first unlock).



The Two-Phase Locking Protocol

T1	**T2**
X(A)	
R(A)	
W(A)	
	S(A)
	R(A)
S(B)	
R(B)	
U(A)	
Unlock started, now no locks	
	U(A)

T1 -> T2

Achieve serializability and maintain consistency

Drawbacks in Two Phase Locking

- May not free from irrecoverabilty

T1	**T2**
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
	Commit
*Fail	

- May not free from cascading rollback
- May not free from deadlock
- May not free from starvation

Strict 2PL, Rigorous 2PL, Conservative 2PL

- Strict 2PL
  - It should satisfy the basic 2PL and all exclusive locks should hold until commit/abort- Cascadeless, recoverable (strict)

T1	**T2**
X(A)	
R(A)	
W(A)	
	S(A)
	R(A)
	U(A)
	commit
commit	
U(A)	

- Rigorous 2PL
  - It should satisfy the basic 2PL and all shared, exclusive locks should hold until commit/abort
- Deadlock, starvation still can occur
- Conservative 2PL is free from deadlock
  - Lock all the items it access before the Transaction begins

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
    - Tells the order when a Transaction enters into system
  - Timestamp could be based on a logical counter
    - Real time may not be unique
- Timestamp-based protocols manage concurrent execution such that
  time-stamp order = serializability order
  - The transaction that comes earlier will execute earlier

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- Maintains for each data *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*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
- **If T_1 comes before T_2 then serializabilty order will be T_1 -> T_2**
  - The transaction that comes earlier will execute earlier

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_i issues a **read**(*Q*)

1. If TS(T_i) < **W**-timestamp(*Q*), then T_i needs to read a value of *Q* that was already overwritten.
    - Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i) $\geq$ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Timestamp-Based Protocols (Cont.)

T1 (100)	T2 (200)	T3 (300)
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

	A	B	C
RTS	0	0	0
WTS	0	0	0

Timestamp-Based Protocols (Cont.)

T1 (100)	T2 (200)	T3 (300)
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

Transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
   Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i) $\geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

	A	B	C
RTS	100	0	0
WTS	0	0	0

Timestamp-Based Protocols (Cont.)

T1 (100)	T2 (200)	T3 (300)
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

Transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.

   Hence, the **read** operation is rejected, and T_i is rolled back.

2. If TS(T_i) $\geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

	A	B	C
RTS	100	200	0
WTS	0	0	0

Timestamp-Based Protocols (Cont.)

T1 (100)	T2 (200)	T3 (300)
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

Transaction T_i issues **write**(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
   - Hence, the **write** operation is rejected, and T_i is rolled back.
2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
   - Hence, this **write** operation is rejected, and T_i is rolled back.
3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

	A	B	C
RTS	100	200	0
WTS	0	0	100

Timestamp-Based Protocols (Cont.)

- Basic Timestamp Ordering Protocol
  - Always ensure serializability
  - Free from deadlock
  - Do not ensure recoverable or cascadeless schedules
- Strict Timestamp Ordering Protocol
  - A Transaction T that issues a R_item(X) or W_item(X) **such that TS(T) > W_TS(X)** has its read or write operation delayed until the Transaction **T**' that wrote the values of X has committed or aborted
  - Ensures recoverable and cascadeless schedules
  - Free from deadlock
  - Ensures serializability

References

- Silberschatz, Abraham, Henry F. Korth, and Shashank Sudarshan. *Database system concepts*. Vol. 6. New York: McGraw-Hill, 1997.
- Ramez Elmasri, Shamkant B. Navathe. *Fundamentals of Database Systems.* Edition 6. Pearson, 2010.