Transactions: Outline

- The Transaction Concept
  - ACID Properties
  - Transaction States
- The Concurrency Problem
  - System Model
  - Schedules
- Serializability of Schedules
  - View Serializable
  - Conflict Serializable
- Recoverability
  - Recoverable Schedules
  - Cascadeless Schedules

Transactions

- The Transaction Concept
  - ACID Properties
  - Transaction States
- The Concurrency Problem
- Serializability of Schedules
- Recoverability

Transaction Concept

- A **transaction** is a *unit* of program execution that *accesses* and possibly *updates* various data items.
- A transaction must see a *consistent* database.
- During transaction execution the database may be *temporarily inconsistent*.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

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

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. $A := A - 50$
  3. **write**(*A*)
  4. **read**(*B*)
  5. $B := B + 50$
  6. **write**(*B*)

- **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, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.

Example of Fund Transfer, cont.

- Transaction to transfer $50 from account A to account B:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- **Isolation requirement** — if between steps 3 and 6, another transaction 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).
  - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we will see later.
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

Transaction Implementation

- A transaction is a *unit* of program execution.
- Transaction boundaries are user-defined.
  - **`COMMIT`** work
  - **`ROLLBACK`** work
    - Abort the transaction.

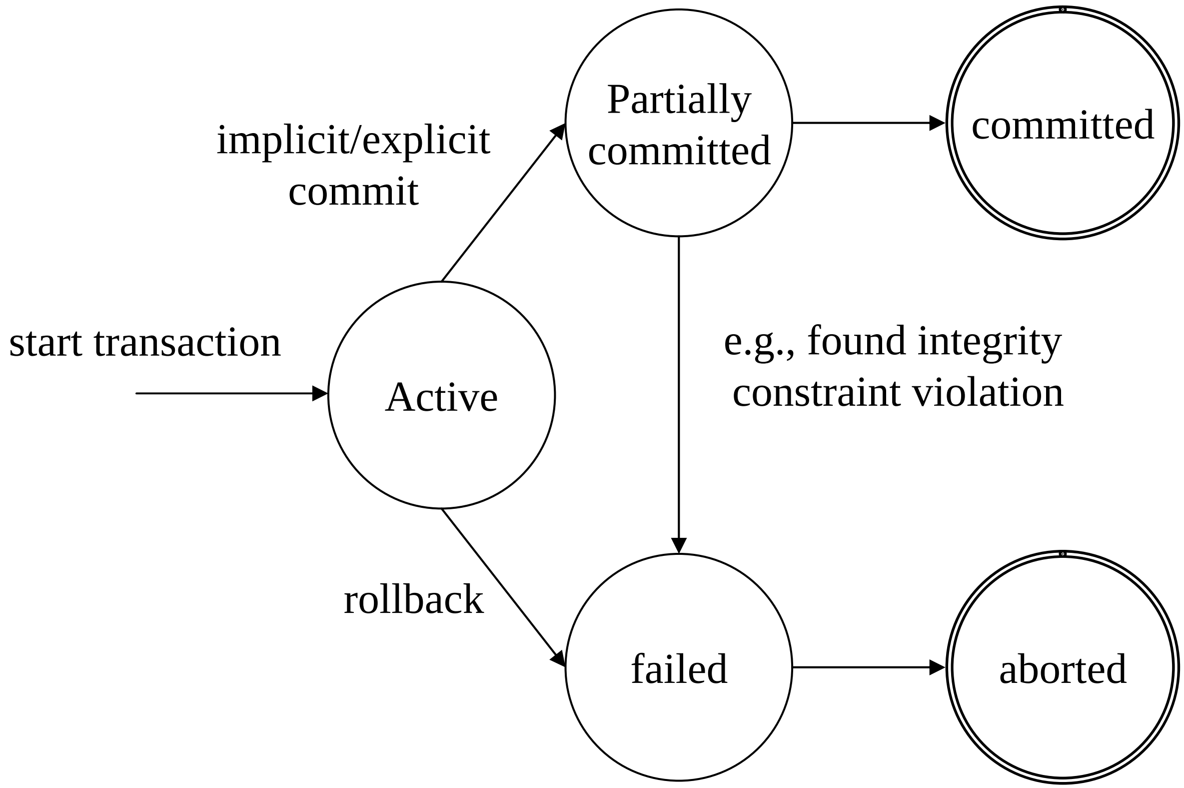
- Implementation of ACID properties
  - Atomicity: typically implemented via logs
  - Consistency: according to constraints/checks/assertions
  - Isolation: typically implemented via locks
  - Durability: typically implemented via logs

Transaction States

- **Active** – the initial state; the transaction stays in this state while it is executing.
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back 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.

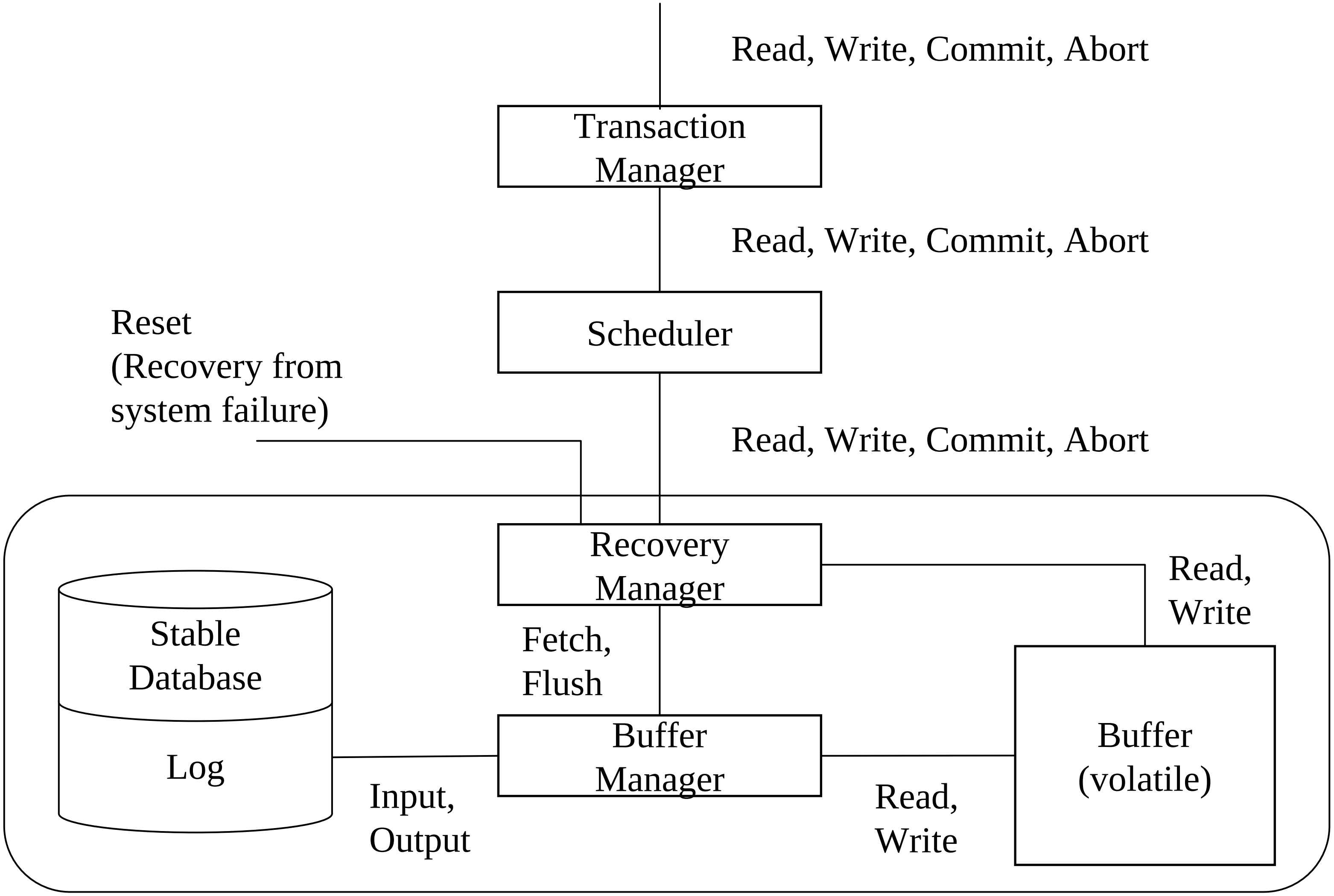
Transaction State Diagram



Transactions

- The Transaction Concept
- The Concurrency Problem
  - System Model
  - Schedules
- Serializability of Schedules
- Recoverability

System Model



The Concurrency Problem

- Model: Centralized system with concurrent access by several users.

- Example
  - Database consisting of two items, X and Y
  - Only criterion for correctness: X = Y
  - The following transactions (i.e., correct serial programs)

T1:	$X \leftarrow X + 1$	*T2*:	$X \leftarrow 2X$
	$Y \leftarrow Y + 1$		$Y \leftarrow 2Y$

- Initially, X=10 and Y=10.

Schedules

- When transactions execute concurrently, their operations are *interleaved*.
- A *schedule* is a sequence of operations from one or more transactions.
- Operations
  - *read(Q, q)*
    - Read the value of the database item *Q* and store in the local variable *q*.
  - *write(Q, q)*
    - Store in the database item *Q* the value of the local variable *q*.
  - Other operations such as arithmetic
  - *commit*
  - *rollback*

Example

- One possible schedule:
  - Initially,
    X=10 and Y=10.

Schedule S1	
T1	*T2*
read(X,x)	
$x \leftarrow x+1$	
write(X,x)	
	read(X,x)
	$x \leftarrow 2x$
	write(X,x)
	read(Y,y)
	$y \leftarrow 2y$
	write(Y,y)
read(Y,y)	
$y \leftarrow y+1$	
write(Y,y)	

- Resulting database: $X = 22$, $Y = 21$, $X \neq Y$.

The Concurrency Problem, cont.

- What do we mean by correctness?
- **Definition D1**: Concurrent execution of transactions must leave the database in a consistent state.
  - Assumes that each transaction, when started on a consistent state of the database leaves it in a consistent state.
- **Definition D2**: Concurrent execution of transactions must be (result) equivalent to some serial execution of the transactions.
  - *Result equivalent* means final database states must be identical.
- Which is better, **D1** or **D2**?

Example

- Example, with an initial database satisfying X = Y

- *S2* is not result equivalent to a serial execution of *T3*, *T4* even though the final database state is consistent.
  - Here only T4 takes effect

Schedule S2	
T3	*T4*
read(X,x)	
$x \leftarrow x+1$	
	read(X,x)
write(X,x)	
	$x \leftarrow 2x$
	write(X,x)
	read(Y,y)
	$y \leftarrow 2y$
read(Y,y)	
$y \leftarrow y+1$	
write(Y,y)	
	write(Y,y)

The Concurrency Problem, cont.

- Our choice is **D2**.
  - An execution sequence is *correct* if it is *result equivalent* to a serial execution.
- Suppose we are given a set of n transaction programs to be run concurrently *or* we can see a set of n transaction running concurrently. How do we check for correctness? This is not easy!
- A simplifying assumption:
  - We will look only at reads and writes on the databases to determine correctness.
  - This assumption is stronger than **D2**, as even fewer schedules are considered correct.

Transactions

- The Transaction Concept
- The Concurrency Problem
- Serializability of Schedules
  - View Serializable
  - Conflict Serializable
- Recoverability

View Equivalent Schedules

Definition: Two schedules *S1* and *S2* are *view equivalent* ($S1 \equiv S2$) if

- The set of transactions participating in *S1* and *S2* are the same.
- For each data item Q in *S1*, if T_i reads Q and the value of Q read by T_i was written by T_j, then the same holds in *S2*.
  - This requirement ensures that the same values are read by all transactions in both *S1* and *S2*.
  - Therefore, the same computation occurs.
- For each data item Q in *S1*, if transaction T_i executes the last write of Q, then the same holds in *S2*.
  - This requirement ensures the same final system state by both schedules.

View Serializability

- **Definition D3**: Let $\{T_1, T_2, \ldots, T_n\}$ be a set of transactions participating in schedule *S*. *S* is a *view serializable* schedule if there exists a serial schedule *Ss* such that $S \equiv Ss$.
- This provides a third notion of correctness (**D3**).
- A serializable (and serial) schedule:

Schedule S3		
T5	*T6*	*T7*
	read(X,x)	
	$x \leftarrow 2x$	
	write(X,x)	
		read(Y,y)
		$y \leftarrow y + 1$
		write(Y,y)
read(X,x)		
$x \leftarrow x + 1$		
write(X,x)		

Example

- Is it a result serializable schedule (by **D2**)?
- Is it a view serializable schedule (by **D3**)?
- Recall that in D3, we only consider read and write operations, and can make no assumptions about the transactions' semantics.

Schedule S4	
T8	*T9*
read(X,x)	
x ← x +1	
write(X,x)	
	read(Y,y)
	y ← y - 10
	write(Y,y)
read(Y,y)	
y ← y + 1	
write(Y,y)	
	read(X,x)
	x←x - 10
	write(X,x)

Example, cont.

- This schedule results in a correct database, but is not (view) serializable because of “lost information” in the definition.
- It is result serializable if we consider the values being stored by the final write operations (under **D2**). It yields the same results as *T8*, *T9* or *T9*, *T8*.
- But **D3** doesn't consider the values being stored.
- The operations we consider in a schedule w.r.t. **D3** are restricted to
  - read X
  - write X
  - commit
  - abort

Possible Transaction Conflicts

- Assume that there are only two transactions, *T1* and *T2*, in the system.

T1	*T2*
write (X,x)	
	read (X,x)

T1	*T2*
read (X,x)	
	write (X,x)

T1	*T2*
write (X,x)	
	write (X,x)

T1	*T2*
read (X,x)	
	read (X,x)

Conflict Equivalent Schedules

- Let I and J be consecutive instructions of a schedule S of different transactions.
- If I and J do not conflict, we can swap their order to produce a new schedule S'.
- The instructions appear in the same order in S and S', except for I and J, whose order does not matter.
- S and S' are termed *conflict equivalent schedules*.

- **Definition D4**: A schedule is *conflict serializable* if it is conflict equivalent to a serial schedule.

Example

- *S5* is view serializable, but is not conflict serializable, because every pair of consecutive instructions conflict.

view serializable?

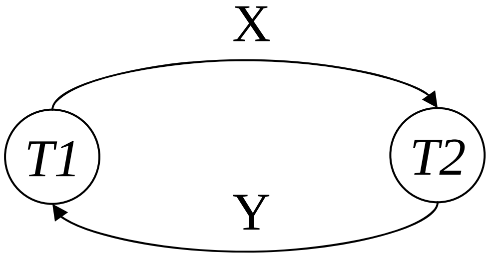
conflict serializable?

Schedule S5		
T1	*T2*	*T3*
read(X,x)		
	write(X,x)	
write(X,x)		
		write(X,x)

- *T2* and *T3* have write instructions without read instructions, termed *blind writes*.
- Blind writes appear in any view serializable schedule that is not conflict serializable.

Conflict Graph

- We now proceed to construct a conflict (directed) graph for a schedule of a set of transactions.
  - We assume that a transaction will always read an item before it writes that item.
- Consider therefore some schedule of a set of transactions *T1*, *T2*, ... , *Tn*.
  - The vertices of the conflict graph are the transaction identifiers.
  - An edge from *Ti* to *Tj* denotes the two transactions conflicting, with *Ti* making the relevant access earlier.
  - Sometimes the edge is labelled with the item involved in the conflict.
- Example for schedule *S1*



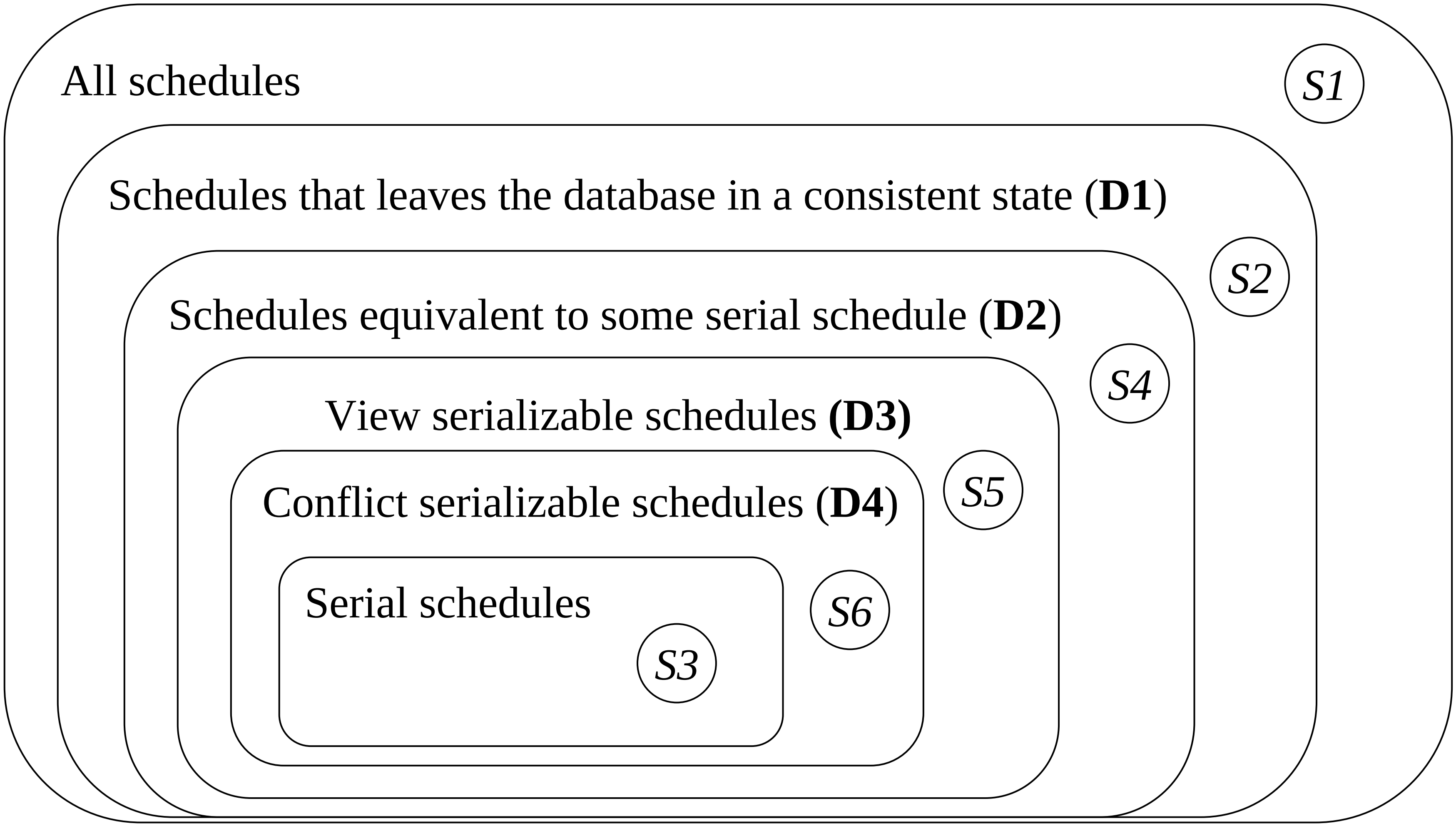
Conflict Graph Example

Schedule S6				
T10	*T11*	*T12*	*T13*	*T14*
	read X			
read Y				
read Z				
				read V
				read W
				write W
	read Y			
	write Y			
		read Z		
		write Z		
read U				
			read Y	
			write Y	
			read Z	
			write Z	
read U				
write U				

Exercises:

1. Draw out the conflict graph for this schedule.
2. Is it conflict serializable?

Relationship Among Schedules



Determining Serializability

- Given a schedule *S*, how do we determine that it is serializable?
- Use a slightly restricted definition: *conflict serializability* (**D4**).
- Two transactions *Ti* and *Tj* conflict if and only if there exists some item X, accessed by both *Ti* and *Tj*, and at least one of these transactions wrote X.
- Intuitively, a conflict between two transactions forces an execution order between them.
- We use conflict serializability, because it has a practical implementation.

Testing Serializability

- A schedule is *conflict serializable* if its conflict graph is acyclic.
- Examples
  - Schedule *S1* had a graph containing a cycle, and is therefore not equivalent to any serial schedule.
  - Schedule *S5* had a graph which is acyclic, and is therefore equivalent to some serial schedule.
- Our goal: to develop protocols that will assure serializability (we will do this in the next lecture).
  - The protocols will generally not examine the conflict graph as it is being created.
  - Instead a protocol will impose the discipline that avoids non-serializable schedules.
- Testing for *view serializability* via conflict graphs is an NP-complete problem.

Transactions

- The Transaction Concept
- The Concurrency Problem
- Serializability of Schedules
- Recoverability
  - Recoverable Schedules
  - Cascadeless Schedules

Recoverable Schedules

- **Recoverable schedule**: For each pair of transactions *Ti* and *Tj*, where *Tj* reads data items written by *Ti*, *Ti* must commit before *Tj* commits.
- The following schedule is not recoverable, if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
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

- If *Ti* fails it must be rollback to retain the atomicity property of transactions. If another transaction *Tj* has read a data item written by *Ti*, *Tj* must also be rollbacked.
- This is called *cascading rollback*, which we should avoid as it can lead to the undoing of a significant amount of work.
- Example
  - *T22* aborts.
  - We have to also rollback *T23* and *T24*.
  - They read “dirty” data.

Schedule S11

T22	*T23*	*T24*
read (A,a)		
read (B,b)		
write (A,a)		
write (B,b)		
	read (A,a)	
		read (A,a)
		read (B,b)
rollback		

Cascadeless Schedules

- **Cascadeless schedule**: For each pair of transactions *Ti* and *Tj*, where *Tj* reads data items written by *Ti*, the commit operation of *Ti* must appear before the read operation of *Tj*.
  - Every cascadeless schedule is also recoverable
  - It is desirable to restrict the schedules to those that are cascadeless

Schedule S11		
T22	*T23*	*T24*
read (A,a)		
read (B,b)		
write (A,a)		
write (B,b)		
commit		
	read (A,a)	
		read (A,a)
		read (B,b)

Summary

- Each transaction preserves database consistency.
- The serial execution of a set of transactions preserves database consistency.
- In a concurrent execution, steps of a set of transactions may be interleaved.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- View serializability provides a convenient definition of correctness (NP-complete problem to check).
- Conflict serilizability (has a practical implementation).
- Schedules must be cascadeless.