Transactions

Example of Fund Transfer

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items
- For example, 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*)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Required Properties of a Transaction

Atomicity requirement

- If the transaction fails after step 3 and before step 6, 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

- 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*)

Required Properties of a Transaction

Consistency requirement in above example:

- For example, the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
- Explicitly specified integrity constraints
  - Primary keys and foreign keys
- Implicit integrity constraints
  - Sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, 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

- 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*)

Required Properties of a Transaction

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 trivially by running transactions **serially**
  - That is, one after the other
- However, executing multiple transactions concurrently has significant benefits

- 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*)

Required Properties of a Transaction

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 even if there are software or hardware failures

- 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*)

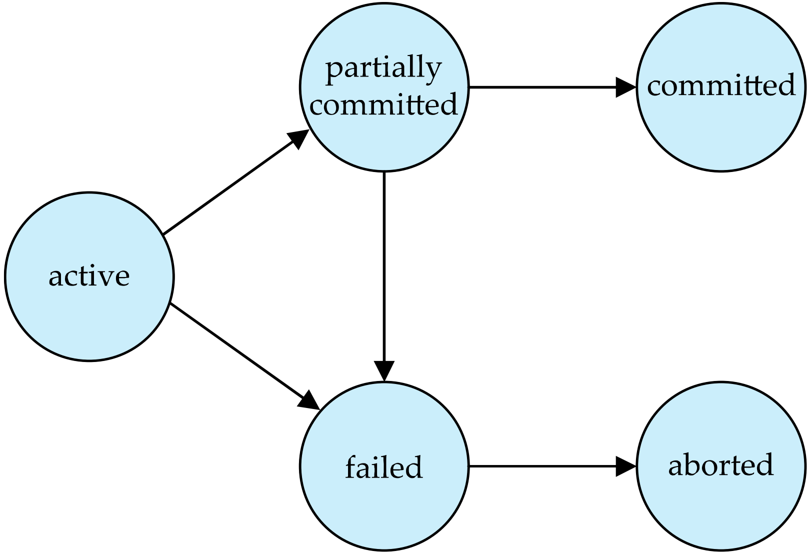
ACID Properties

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**
  - 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

Transaction State

- **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 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*)

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system
- Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - For example, one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: Short transactions need not wait behind long ones
- **Concurrency control schemes:** Mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

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 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

Schedule 1

- Let T_1 transfer $50 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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
45	250	295	T2, write A	
45	255	300	T2, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ 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	

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	210	300	T2, write B	@ Commit
40	210	250	T1, write A	
40	260	300	T1, write B	@Commit

Consistent @ Commit

Inconsistent @ Transit

Inconsistent @ Commit

Value of A and B are different from schedule 1, yet consistent

Schedule 3

- Let T_1 and T_2 be the transactions defined previously
- The following schedule is not a serial schedule, but it is **equivalent** to Schedule 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

Schedule 3

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 1

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
45	200	245	T2, write A	
45	250	295	T1, write B	@ Commit
45	255	300	T2, write B	@Commit

Consistent @ Commit

Inconsistent @ Transit

Inconsistent @ Commit

- 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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	200	290	T1, write A	
90	250	340	T1, write B	@ Commit
90	260	350	T2, write B	@Commit

Consistent @ Commit

Inconsistent @ Transit

Inconsistent @ Commit

Next Lecture

Transactions

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in

© Silberschatz, Korth and Sudarshan