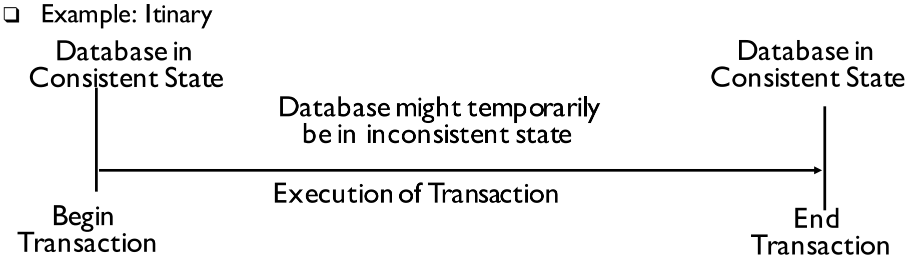
Transaction Basics

Database Transaction

- ❑ A transaction is a collection of actions that belong logically together
- ❑ Data centric:
  - ☆ a transaction is a sequence of read and write operations on data items
- ❑ OO centric
  - ☆ a transaction is a sequence of operations involving various objects
- ❑ Example: Money transfer
  - ☆ **Withdraw amount X from account A**
  - ☆ **Deposit amount X on account B**
- ❑ Example: Itinary



Transaction Processing

- ❑ Why
  - ☆ Originally, transaction processing was only used in large companies on their tightly coupled systems
  - ☆ But now also small and medium sized companies
    - maintain their own database
    - offer online access, and thus need electronic transactions.
    - The market for transaction processing is many tens of billions of dollars per year
  - ☆ Now transaction processing has become a standard in distributed systems
    - core component/service in J2EE, persistent storages, caches, etc.
    - There are may standards (XA-interface, Java Transaction API (JTA), Java Transaction Service (JTS), Web Services Transaction (WS-Transaction)
    - Many simple cloud storage systems start offering a transactional interface
      - Hbase
      - Google internally implemented several transactional APIs to control execution on their data
  - ☆ It is an accepted, proven and tested programming model and computing paradigm for complex applications.

Example

- ❑ Itinary:
  - ☆ book flight
  - ☆ book car
  - ☆ book hotel

Properties of Transactions

- Atomicity
  - All or nothing
  - Return Commit to user:
    - all updates have been successfully executed
  - Return Abort to user:
    - none of the updates is reflected on the data
    - Abort might be user-induced or system-induced
    - **Local Recovery**: eliminating partial results
- Example itinary:
  - if atomicity responsibility of programmer
    - check whether all flights, hotel and car available
    - if one is not available: return error
    - if all available: reserve one at a time
    - problem?
  - transaction based: indicate that all operations of itinary belong to one transaction
    - openTransaction
      - book flight1,
      - book flight…
      - book flight n
      - reserve car
      - reserve hotel
    - closeTransaction

Schedule

- Schedule: sequence of operations from a set of transactions which obeys the sequence of operations within a transaction
  - Reflects the order in which the DBMS/server executes the read and write operations on the data items operations;

	Transaction T:	**Transaction *U*:**
	balance = a.getBalance(); *a.setBalance(balance+10);* *b.insertRec("a,+10")*	*balance = a.getBalance();* *a.setBalance(balance+30);* *b.insertRec("a,+30");*
Schedule 1	*balance = a.getBalance();* $200	
	a.setBalance(balance+10); $210	
		balance = a.getBalance(); $210
		a.setBalance(balance+30); $240
		b.insertRec("a,+30");
	b.insertRec("a,+10");	
Schedule 2	*balance = a.getBalance();* $200	
		balance = a.getBalance(); $200
		a.setBalance(balance+30); $230
	a.setBalance(balance+10); $210	
		b.insertRec("a,+30");
	b.insertRec("a,+10");	

Property of Transactions

- Isolation
  - ☆ Ensuring correct results even when there are many transactions being executed concurrently on the same data
  - ☆ Execution of concurrent transactions controlled such that result the same as if executed serially
  - ☆ Enforced by a **concurrency control** protocol
  - ☆ Why is concurrent execution useful?
- Durability
  - ☆ Committed updates persistent despite failures
  - ☆ flush before commit or log before commit

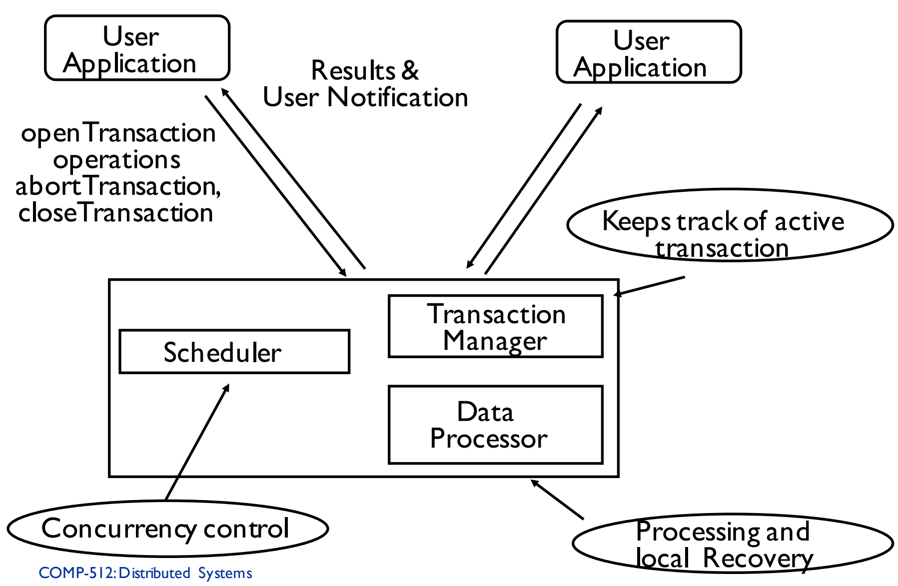
Server Interface

- *openTransaction() -> trans;*
  - starts a new transaction and delivers a unique transaction TID *trans*. This identifier will be used in the other operations in the transaction.
- *operation(trans, operationDetails);*
  - Each operation indicates the transaction it belongs to
- *closeTransaction(trans) -> (commit, abort);*
  - ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.
- *abortTransaction(trans);*
  - aborts the transaction.
- Some interfaces hide the TID
  - Each connection client / transaction system has always at most one open transaction

Transaction life

Successful	*Aborted by client*	*Aborted by server*	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*	server aborts transaction →	*operation*
⁝	⁝		⁝
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

Centralized Transaction Execution



Isolation

- Serial Schedules/History
  - By assumption *serial* schedules are good
  - No interleaving of transactions. That is, transactions are executed one at a time

Transaction T:	Transaction *U*:
balance = a.getBalance(); *a.setBalance(balance+10);* *date = b.getDate();*	*balance = a.getBalance();* *a.setBalance(balance+30);* *date = b.getDate();*

	Transaction T	Transaction U
Serial Schedule	*balance = a.getBalance();* \$200	
	a.setBalance(balance+10); \$210	
	date = b.getDate();	
		balance = a.getBalance(); \$210
		a.setBalance(balance+30); \$240
		date = b.getDate();

Serializable Schedules

- Serializable Schedules/Histories
  - allow operations of different transactions to interleave
  - But the "effect" of the interleaved schedule is "equivalent" to a serial schedule

Transaction T:	Transaction *U*:
balance = a.getBalance(); *a.setBalance(balance+10);* *date = b.getDate();*	*balance = a.getBalance();* *a.setBalance(balance+30);* *date = b.getDate();*

	Transaction T	Transaction U
Serializable Schedule	*balance = a.getBalance();* \$200	
	a.setBalance(balance+10); \$210	
		balance = a.getBalance(); \$210
		a.setBalance(balance+30); \$240
		date = b.getDate()
	date = b.getDate();	
Unserializable Schedule	*balance = a.getBalance();* \$200	
		balance = a.getBalance(); \$200
		a.setBalance(balance+30); \$230
	a.setBalance(balance+10); \$210	
		date = b.getDate();
	date = b.getDate();	

another unserializable schedule

Transaction *V*:	Transaction *W*:
a.withdraw(100) *b.deposit(100)*	*aBranch.branchTotal()*

a.balance = 200; b.balance = 400;

a.withdraw(100);	\$100		
		total = a.getBalance()	a=\$100
		total = total+b.getBalance()	b=\$400
		total = total+c.getBalance()	t=\$500
b.deposit(100)	\$500	:	

Conflicts

- Conflicting operations:
  - two operations O_{ij} and O_{kl} conflict
    - if they are from two different transactions T_i and T_k,
    - both access the same data item X and
    - at least one of them is a write operation
  - In this case, we also say that T_i and T_k conflict.

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Serializability

- *Conflict equivalence:*
  - ☆ Two histories are conflict equivalent, if the relative order of execution of conflicting operations belonging to committed transactions is the same.
- *Serializable schedule*: conflict-equivalent to a serial schedule.
  - ☆ .

Serializable Schedules	$r(a)$		$w(a)$	
	$w(a)$			$r(a)$
		$r(a)$		
			$w(b)$	
		$w(a)$		
				$r(b)$
		$r(b)$		
	$r(b)$			
Unserializable Schedules	$r(a)$		$w(a)$	
		$r(a)$		$r(a)$
		$w(a)$		$r(b)$
	$w(a)$		$w(b)$	
		$r(b)$		
	$r(b)$			

Further Examples

T1	T2
r1(x)	
w1(x)	
	r2(z)
	r2(y)
	w2(x)
	c2
w1(y)	
c1	

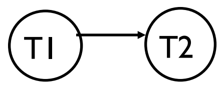
T1	T2
w1(x)	
	r2(x)
	r2(y)
w1(x)	
w1(y)	
c1	
	w2(x)
	c2

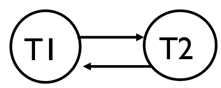
T1	T2
w1(x)	
	r2(x)
w1(z)	
W1(y)	
c1	
	r2(y)
	w2(x)
	c2

Serializability and Dependency Graphs

- Dependency graph / Serialization graph / precedence graph / Serializability graph for a schedule:
  - ☆ Let S be a schedule over a set of transactions T
  - ☆ Each transaction $T_i \in T$ is represented by a node
    - There is an edge from T_i to T_j if an operation of T_i precedes and conflicts with on of T_j' s operations in the schedule.

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)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Concurrency Control

- The database system uses a concurrency control mechanism to enforce that only serializable schedules exist
- implemented within the scheduler
  - ☆ schedules when operations may execute

Concurrency Control: Locking

- No conflict: transactions can execute at the same time
- Upon first conflict: the second transaction has to wait until the first transaction releases the lock
- Locks: Two types, because two read operations do not conflict
- Basics of locking:
  - ☆ Each transaction T must obtain a S (*shared*) lock on object *before* reading, and an X (*exclusive*) lock on object *before* writing.
  - ☆ If an X lock is granted on object O, no other lock (X or S) might be granted on O at the same time.
  - ☆ If an S lock is granted on object O, no X lock might be granted on O at the same time.
  - ☆ Conflicting locks are expressed by the compatibility matrix:

For one object		*Lock requested*	
		shared	*exclusive*
Lock already set	*none*	OK	OK
	shared	OK	wait
	exclusive	wait	wait

Two-phase locking (2PL)

- Each transaction T must request a S (*shared*) lock on object *before* reading, and an X (*exclusive*) lock on object *before* writing.
  - ☆ A transaction does not need to request a S lock on an object for which it already holds an X lock.
  - ☆ If a transaction has an S lock and needs an X lock it must wait until all other S locks (except its own) are released
- After a transaction has released one of its lock (unlock) it may not request any further locks (2PL: growing phase / shrinking phase)
- Using strict two-phase locking (strict 2PL) a transactions releases all its lock at the end of its execution -> WHY?

2PL allows only serializable schedules

strict 2PL forbids dirty reads and premature writes

Example

T1:	T2:	T2:
S(A)		S(A)
R(A)		R(A)
X(B)		
W(B)	X(A)	S(B)
Commit		
U(A) U(B)		
		R(B)
		Commit
		U(A),U(B)
	W(A)	
	Commit	
	U(A)	

Lock Table:

A: T1-S,T3-S

A: T1-S,T3-S, B: T1-X

A: T1-S,T3-S, T2-X B: T1-X, T3-S

A: T3-S, T2-X, B: T3-S

A: T2-X

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Waits-for graph:
  - ☆ Nodes are transactions
  - ☆ There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Deadlock detection: look for cycles in the wait-for graph

T1:	T2:
S(A)	S(B)
R(A)	R(B)
X(B)	X(A)