Module 13

Transaction Processing Concepts

Objectives

- Learn about issues transaction processing, concurrency and crash recovery

95.305

Topics

- **Read and Write operations**
- **Concurrency**
- **Transactions**
- **Schedules and Recovery**

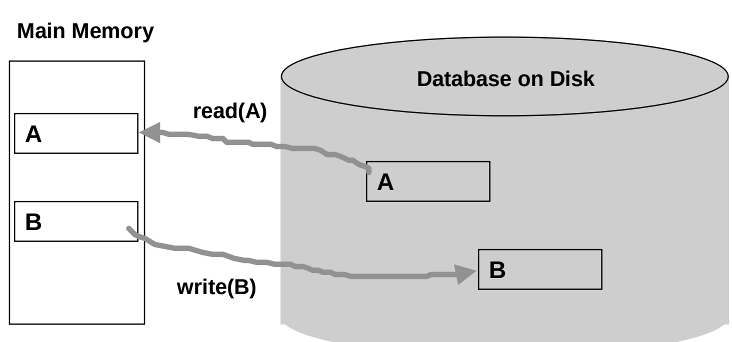
References

- **Elmasri & Navathe Chapters 17**

Observations

- Computers, sometimes fail
- Databases are designed to survive computer crashes
- Databases must move from consistent state to consistent state
- Databases are typically multi-user and it's desirable to interleave transactions from different users

Storage Model



read(A)
  if block(A) not in main memory
    read A from disk to main memory
  x := A

write(B)
  if block(B) not in main memory
    read B from disk to main memory
  B in Block(B) := x

Observations

- **Both** read() **and** write() **primitives may require disk block reads**
- **Neither operation specifically requires a disk block write**
- **Modified data block will eventually be written out to disk -but perhaps long after the transaction**
- **But, if system crashes between** write() **primitive and writing block back to disk, the modification is lost**

- **From the database point of view, transactions are user submissions which consist of**

  read(X)
  ...
  computation stuff
  ...
  write(X)
- **Transactions submitted by various users may execute concurrently and may access the same data**
- **Different, concurrent transactions can interact in a bad way**

Simple Transactions

T1: transfer $100 from savings to chequing account

read(X)
X := X-N
write(X)
read(Y)
Y := Y+N
write(Y)

T2: deposit $50 into savings account

read(X)
X:=X+M
write(X)

Serial Execution

		SavAcct	CheqAcct	
		$1000	$2000	{$1000 + $2000 =$3000}
T1: transfer $50 from savings to chequing account	**read(X)**			
	X := X-N			
	write(X)	$950		
	read(Y)			
	Y := Y+N			
	write(Y)		$2050	{$950 + $2050 =$3000}
T2: deposit $150 into savings account	**read(X)**			
	X:=X+M			
	write(X)	$1100		
		$1100	$2050	

Concurrent Execution: Lost update problem

T1: transfer $50 from savings to cheq. acct	T2: deposit $150 into savings acct	SavAcct $1000	CheqAcct $2000
read(X)			
X := X-N			
	read(X)		
	X:=X+M		
write(X)		$950	
read(Y)			
	write(X)	$1150	
Y := Y+N			$2050
write(Y)			
		$1150	$2050

Concurrent Execution: Dirty Read Problem

T1: transfer $50 from savings to cheq. acct	T2: deposit $150 into savings acct	SavAcct $1000	CheqAcct $2000
read(X)			
X := X-N			
write(X)		$950	
	read(X)		
	X:=X+M		
	write(X)	$1100	
read(Y)			
t1 crash		$1000	
		$1000	$2000

T1: crashes and the database restores values written by T1 to their starting values

Concurrent Execution: Incorrect Summary Problem

T1: transfer $50 from savings to cheq. acct	T3: add balances of sav. & cheq. acct	SavAcct $1000	CheqAcct $2000	
read(X)				
X := X-N				
write(X)		$950		
	read(X)			
	sum:= sum + X			
	read(Y)			
	sum := sum + Y			Sum
	write(sum)			$2950
read(Y)				
Y := Y+N				
write(Y)		$950	$2050	($3000)

- **When a transaction is submitted the DBMS must make sure that either**

  a) All operations of the transaction are completed successfully

  b) The transaction has no effect whatsoever

- **Interleaved transactions may not appear to fail but may lead to inconsistencies**

Common Failures

1) Computer Crash

2) Transaction Crash -operation in a transaction crashes (e.g. dividing account balance by zero)

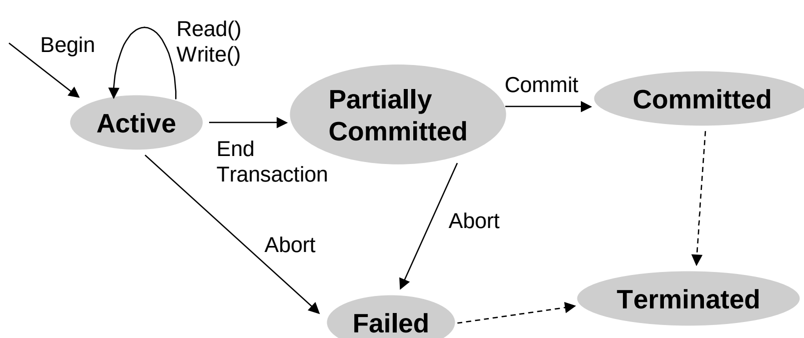
3)Local error: data not found for transaction to proceed

4) Concurrency control enforcement: a transaction is terminated because it conflicts with a currently executing one

The database must maintain sufficient information to recover from these errors

Transaction States

- Recovery Mechanism keeps track of a transaction's state



- What happens if a crash occurs in the Partially Committed state

Transaction Journal

- To be able to recover from failures the DBMS maintains a log, or journal
- Log is kept in database and written to disk before transaction is deemed committed
- Log entries:
  -start(TransID)
  -write(TransID DataItem oldValue, newValue)
  -read(TransID, DataItem)
  -commit(TransID)
  -abort(TransID)

- Assumptions: transactions don't nest, all permanent changes to database are done through transactions

- Transaction reaches its committed state when

  -all transaction operations have completed successfully

  AND

  -commit(TransID) has been written to the log (on disk)

Serial Execution

		SavAcct	CheqAcct	Log
		$1000	$2000	start(T1)
				read(T1, X)
T1: transfer $50 from savings to chequing account	**read(X)**			write(T1, X, 1000, 950)
	X := X-N			read(T1, Y)
	write(X)	$950		write(T1, Y, 2000, 2050)
	read(Y)			commit(T1)
	Y := Y+N			
	write(Y)		$2050	start(T2)
				read(T1, X)
T2: deposit $150 into savings account	**read(X)**			write(T1, X, 950, 1100)
	X:=X+M			commit(T2)
	write(X)	$1100		
		$1100	$2050	

Crash Recovery

- **can use a** redo(TranID) **primitive**
- redo(TransID)
  “set the value of all data items updated by transaction TransID to their newValue”
- **redo() primitive must be idempotent: executing it several times must be equivalent to executing it once**
- **Transaction TransID needs to be redone if a crash occurs and both** start(TransID) **and** commit(TransID) **appears in the log**
- **Log could be periodically checkpoint to ensure rollback need not go to far back**

Desirable Transaction Properties

- <u>Atomic:</u> transactions are either performed entirely or not at all
- <u>Consistency Preservation:</u> correct execution must take database from consistent state to consistent state
- <u>Isolation:</u> a transaction should not make its updates visible to other transactions until it has been committed (precludes concurrency)
- <u>Durability:</u> once a transaction has been committed its effects must not be lost due to subsequent failure

Buffer Management

- Log entries are written to disk before operations re executed
- Wasteful to do block write for every log entry
- Protocol: before commit(T) is written, all log records pertaining to T must be written to disk
- Before a data block is written all log records must have been written to disk

- Notice this puts a strain on virtual memory -the host OS might not want to swap pages this way

Concurrency

- **Concurrency: allowing several transactions to execute in an interleaved way**

Interleaving OK : does not produce inconsistency

T1: transfer $50 from savings to cheq. acct	T2: deposit $150 into savings acct	SavAcct	CheqAcct
		$1000	$2000
read(X)			
X := X-N			
write(X)		$950	
	read(X)		
	X:=X+M		
	write(X)	$1100	
read(Y)			
Y := Y+N			
write(Y)			$2050
		$1100	$2050

Bad Interleaving: produces inconsistencies

T1: transfer $50 from savings to cheq. acct	T2: deposit $150 into savings acct	SavAcct	CheqAcct
		$1000	$2000
read(X)			
X := X-N			
	read(X)		
	X:=X+M		
write(X)		$950	
read(Y)			
	write(X)	$1150	
Y := Y+N			$2050
write(Y)			
		$1150	$2050

Serializable Schedules

- **An interleaving of transactions is OK if it is equivalent to some serial schedule**

T1: transfer $50 from savings to cheq. acct	T2: deposit $150 into savings acct
read(X)	
X := X-N	
write(X)	
	read(X)
	X:=X+M
	write(X)
read(Y)	
Y := Y+N	
write(Y)	

=

T1: transfer $50 from savings to cheq. acct	T2: deposit $150 into savings acct
read(X)	
X := X-N	
write(X)	
read(Y)	
Y := Y+N	
write(Y)	
	read(X)
	X:=X+M
	write(X)

Serializable Schedules

- **An interleaving of transactions is OK if it is equivalent to some serial schedule**

T1: transfer \$50 from savings to cheq. acct	T2: deposit \$150 into savings acct
read(X)	
X := X-N	
	read(X)
	X:=X+M
	write(X)
write(X)	
read(Y)	
Y := Y+N	
write(Y)	

$\neq$

T1: transfer \$50 from savings to cheq. acct	T2: deposit \$150 into savings acct
read(X)	
X := X-N	
write(X)	
read(Y)	
Y := Y+N	
write(Y)	
	read(X)
	X:=X+M
	write(X)

- **Idea: DBMS can allow interleaving as long as it remains “convinced” that a serial equivalent schedule exists**
- **i.e. that the interleaving is equivalent to some serial schedule**

Simple Transaction Model

read(X)
...
stuff
...

write(X)

=

read(X)
write(X)

Instruction Conflicts

T1:	T2:	
read(X)	read(X)	order doesn't matter
read(X)	write(X)	order matters
write(X)	read(X)	order matters
write(X)	write(X)	order matters
read(X)	read(Y)	order doesn't matter
read(X)	write(Y)	"
write(X)	read(Y)	"
write(X)	write(Y)	"

T1:	T2:
read(X)	
write(X)	
	read(X)
	write(X)
read(Y)	
write(Y)	

=

T1:	T2:
read(X)	
write(X)	
	read(X)
read(Y)	
	write(X)
write(Y)	

T1:	T2:
read(X)	
write(X)	
	read(X)
read(Y)	
	write(X)
write(Y)	

=

T1:	T2:
read(X)	
write(X)	
read(Y)	
	read(X)
write(Y)	
	write(X)

T1:	T2:
read(X)	
write(X)	
read(Y)	
	read(X)
write(Y)	
	write(X)

=

T1:	T2:
read(X)	
write(X)	
read(Y)	
write(Y)	
	read(X)
	write(X)

conflict equivalent serial schedule

T1:	T2:
read(X)	
	write(X)
write(X)	

not conflict serializable

not the same as:

T1; T2

or

T2; T1

Not serializable by equivalent

T1:	T2:
read(X)	
i := X-50	
write(X)	
	read(Y)
	j := Y-10
	write(Y)
read(Y)	
k := Y+50	
write(Y)	
	read(X)
	l := X+10
	write(X)

T1:	T2:
read(X)	
i := X-50	
write(X)	
read(Y)	
k := Y+50	
write(Y)	
	read(Y)
	j := Y-10
	write(Y)
	read(X)
	l := X+10
	write(X)

- Serializability is a sufficient, but not necessary, condition to prevent interleaving inconsistencies

- Serializability is too difficult to test for in practice
- Alternative is to determine what actions a transaction can take which will ensure that serializability results
- Transactions follow protocols which allows them to interleave with assurance of consistency

- Example protocol, or alternative, is to lock the data values with a mutual exclusion semaphore
- This ensures that only one transaction has access to data (prevent read and write by others)
- Problem: this can lead to deadlock and starve transactions which mutually need each others data