Database Management Systems

Transaction, Concurrency and Recovery

Adapted from Lecture notes by Goldberg @ Berkeley

Introduction

What is Concurrent Process (CP)?

- Multiple users access databases and use computer systems simultaneously.
- Example: Airline reservation system.
  - An airline reservation system is used by hundreds of travel agents and reservation clerks concurrently.

Why Concurrent Process?

- Better transaction throughput and response time
- Better utilization of resource

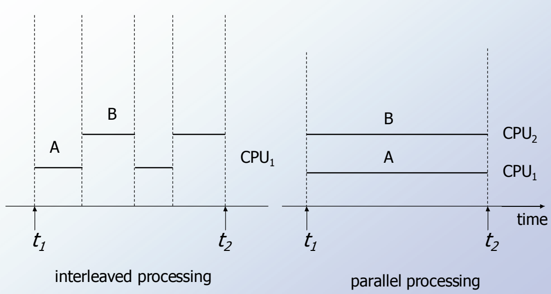
Transaction

- What is Transaction?
- A sequence of many actions which are considered to be one atomic unit of work.
- Basic operations a transaction can include "actions":
  - Reads, writes
  - Special actions: commit, abort

ACID Properties of transaction

- Atomicity: Transaction is either performed in its entirety or not performed at all, this should be DBMS' responsibility
- Consistency: Transaction must take the database from one consistent state to another if it is executed in isolation. It is user's responsibility to insure consistency
- Isolation: Transaction should appear as though it is being executed in isolation from other transactions
- Durability: Changes applied to the database by a committed transaction must persist, even if the system fail before all changes reflected on disk

Concurrent Transactions



Schedules

- What is Schedules
  - A schedule S of n transactions T1,T2,...Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear in the same order in which they occur in Ti.
  - Example: S_a: r1(A),r2(A),w1(A),w2(A), a1,c2;

T1	T2
Read(A)	
	Read(A)
Write(A)	
	Write(A)
Abort T1	
	Commit T2

Oops, something's wrong

- Reserving a seat for a flight
- If concurrent access to data in DBMS, two users may try to book the same seat simultaneously

time

Agent 1 finds seat 35G empty

Agent 2 finds seat 35G empty

Agent 1 sets seat 35G occupied

Agent 2 sets seat 35G occupied

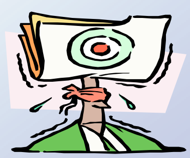
Another example

- Problems can occur when concurrent transactions execute in an uncontrolled manner.
- Examples of one problem.
  - A original equals to 100, after execute T1 and T2, A is supposed to be 100+10-8=102

Add 10 To A T1	Minus 8 from A T2	Value of A on the disk
Read(A)		100
A=A+10		100
	Read(A)	100
	A=A-8	100
Write(A)		110
	Write(A)	92

What Can Go Wrong?

- Concurrent process may end up violating Isolation property of transaction if not carefully scheduled
- Transaction may be aborted before committed
  - undo the uncommitted transactions
  - undo transactions that sees the uncommitted change before the crash



Conflict operations

- Two operations in a schedule are said to be *conflict* if they satisfy all three of the following conditions:

(1) They belong to different transactions

(2) They access the same item A;

(3) at least one of the operations is a write(A)

Example in Sa: r1(A),r2(A),w1(A),w2(A), a1,c2;

- r1(A),w2(A) conflict, so do r2(A),w1(A),
- r1(A), w1(A) do not conflict because they belong to the same transaction,
- r1(A),r2(A) do not conflict because they are both read operations.

Serializability of schedules

- Serial
  - A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.(No interleaving occurs in a serial schedule)
- Serializable
  - A schedule S of n transactions is **serializable** if it is *equivalent* to some *serial* schedule of the same n transactions.
- schedules are conflict equivalent if:
  - they have the same sets of actions, and
  - each pair of conflicting actions is ordered in the same way
- Conflict Serializable
  - A schedule is said to be conflict serializable if it is conflict equivalent to a serial schedule

Characterizing Schedules

1. Avoid cascading abort(ACA)
- Aborting T1 requires aborting T2!
  - Cascading Abort
- An ACA (avoids cascading abort)
  - A X act only reads data from committed X acts.

No:

T1	T2
Read(A) Write(A)	
	Read(A) Write(A)
Abort	

Yes:

T1	T2
Read(A) Write(A) commit	
	Read(A) Write(A)

2. recoverable
- Aborting T1 requires aborting T2!
  - But T2 has already committed!
- A recoverable schedule is one in which this cannot happen.
  - i.e. a X act commits only after all the X acts it "depends on" (i.e. it reads from) commit.
  - ACA implies recoverable (but not vice-versa!).

No:

T1	T2
Read(A) Write(A)	
	Read(A) Write(A) Commit
Abort	

Yes:

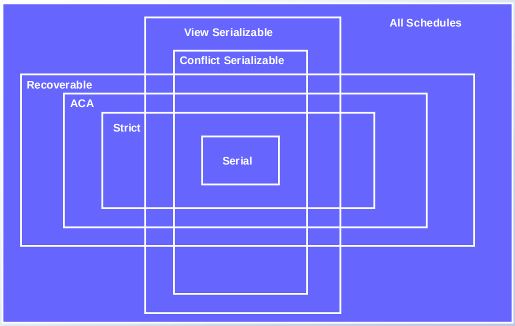
T1	T2
Read(A) Write(A)	
	Read(A) Write(A)
Commit	
	Commit

3. strict schedule

Yes:

T1	T2
Read(A) Write(A) Commit	
	Read(A) Write(A) Commit

Venn Diagram for Schedules



Example

T1:W(X), T2:R(Y), T1:R(Y), T2:R(X), C2, C1

- serializable: Yes, equivalent to T1,T2
- conflict-serializable: Yes, conflict-equivalent to T1,T2
- recoverable: No. Yes, if C1 and C2 are switched.
- ACA: No. Yes, if T1 commits before T2 reads X.

Sample Transaction (informal)

- Example: Move $40 from checking to savings account
- To user, appears as one activity
- To database:
  - Read balance of checking account: read(X)
  - Read balance of savings account: read (Y)
  - Subtract $40 from X
  - Add $40 to Y
  - Write new value of X back to disk
  - Write new value of Y back to disk

Sample Transaction (Formal)

T1

t_0 read_item(X);
read_item(Y);
X:=X-40;
Y:=Y+40;
write _item(X);
t_k write_item(Y);

Focus on concurrency control

- Real DBMS does not test for serializability
  - Very inefficient since transactions are continuously arriving
  - Would require a lot of undoing
- Solution: concurrency protocols
- If followed by every transaction, and enforced by transaction processing system, guarantee serializability of schedules

Concurrency Control Through Locks

- **Lock:** variable associated with each data item
  - Describes status of item wrt operations that can be performed on it
- Binary locks: Locked/unlocked
- Multiple-mode locks: Read/write
- Three operations
  - read_lock(X)
  - write_lock(X)
  - unlock(X)
- Each data item can be in one of three lock states

Two Transactions

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Let's assume serial schedule S1: T1;T2

Initial values: X=20, Y=30 → Result: X=50, Y=80

Locks Alone Don't Do the Trick!

Let's run T1 and T2 in interleafed fashion

Schedule S

T1	T2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

unlocked too early!

Non-serializable!

Result: X=50, Y=50

Two-Phase Locking (2PL)

- Def.: Transaction is said to follow the *two-phase-locking protocol* if all locking operations precede the *first* unlock operation

Example

T1'	T2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

- Both T1' and T2' follow the 2PL protocol
- Any schedule including T1' and T2' is guaranteed to be serializable
- Limits the amount of concurrency

Deadlock in 2PL

- Deadlock
  - T1 waits for T2 to unlock B
  - T2 waits for T1 to unlock A
  - Neither can proceed! A deadlock!

Write to A, B	**Read from A, B**
T1	**T2**
write_lock(A)	
	read_lock(B)
write_lock(B)	
	read_lock(A)
write(A)	
write(B)	
	write(A)
	write(B)
unlock(A)	
unlock(B)	
	unlock(A)
	unlock(B)

Variations to the Basic Protocol

- Previous technique knows as *basic 2PL*
- *Conservative* 2PL (static) 2PL: Lock all items needed BEFORE execution begins by predeclaring its read and write set
  - If any of the items in read or write set is already locked (by other transactions), transaction waits (does not acquire any locks)
  - Deadlock free but not very realistic

Variations to the Basic Protocol

- *Strict* 2PL: Transaction does not release its write locks until AFTER it aborts/commits
  - Not deadlock free but guarantees recoverable schedules (strict schedule: transaction can neither read/write X until last transaction that wrote X has committed/aborted)
  - Most popular variation of 2PL

The Phantom Problem

- The concurrency control problem for insertion and deletion in database
- Example: A local bank

Accounts

acc_num	branch	amount
99	Easton	$100
120	Allentown	$500
190	Easton	$200

Assets

branch	assets
Easton	$300
Allentown	$500

Two Transactions

- T1 wants to verify that the accounts at the Easton branch add up to be equal to the total assets of the Easton branch
- T2 wants to add a new account (150, ‘Easton’, $50) to the accounts table
- Write schedules for both transaction following the 2-phase locking protocol

Schedule following 2PL

T1	T2

read_lock(Accounts[99]);
read_lock(Accounts[190]);
read_item(Accounts[99]);
read_item(Accounts[190]);
 	
read_lock(Assets[Easton]);
read_item(Assets[Easton]);
unlock(Accounts[99]);
unlock(Accounts[190]);
unlock(Assets[Easton]);
	write_lock(Accounts[150]);
	write_item(Accounts[150, ‘Easton’, $50]);
	write_lock(Assets[Easton]);
	write_item(Assets[Easton, $350);
	unlock(Accounts[150]);
	unlock(Assets[Easton]);

When Will the Phantom Problem Occur?

- When T1 and T2 are interleaved, the Phantom Problem may occur
  - See previous slide
- Does it mean 2PL is not suitable for insertion and deletion in database?
  - No. The phantom problem occurs because the control information of the table account was not locked
  - Solution: Lock the control information (e.g., the index) when insertion/deletion happens

Index Locking

- Suppose access to a table is controlled by a B-tree
- Should we use 2PL on B-tree?
  - 2PL says that a transaction must acquire all the locks before it can release any
  - Thus, the root of the B-tree must stay locked until all the locks of a transaction are acquired
  - Locking the root of a B-tree has the same effect as locking the whole tree

Tree Locking Protocol

- A transaction's first lock is at the root of the B-tree
- Subsequent locks may only be acquired if the transaction currently has a lock on the parent node
- Node may be unlocked at any time
- A transaction may not relock a node on which it has released a lock, even if it still holds a lock on the node's parent

B-Tree Locking

- Search: start with a readlock at the root, request a readlock on a child node while holding a lock on the parent, after the lock on the child node is received, release the lock on the parent
- Insertion (or Deletion): start with a writelock at the root, request a writelock on a child node while holding a lock on the parent, after the lock on the child node is received, release the lock on the parent only if the child node still has room for insertion (or deletion)

Tree Locking Protocol

- Why the tree locking protocol work?
  - Let's redo the bank accounts example

Concluding Remarks

- Concurrency control subsystem is responsible for inserting locks at right places into your transaction
  - Strict 2PL is widely used
  - Requires use of waiting queue
- All 2PL locking protocols guarantee serializability
- Does not permit all possible serial schedules

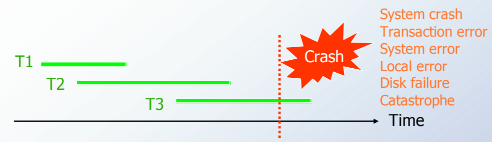
Why Recovery Is Needed

- Any system will fail
- Type of failures:
  - A computer failure
    - System crash, …
  - Transaction or system error
    - Integer overflow, divide by zero, logical error, …
  - Local error or exception
    - Data not found, Insufficient balance,

Types of Failures

- Concurrency control enforcement
  - Request for lock denied, deadlock …
- Disk failure
  - Disk malfunction, head crash, …
- Physical problems and catastrophes
  - Power/air-conditioning failure, fire, flood, …

Why “Database Recovery Techniques”?

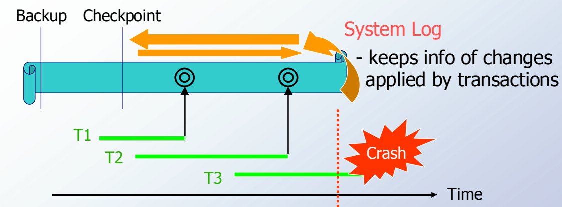


- ACID properties of Transaction

Database system should guarantee

- Durability : Applied changes by transactions must not be lost. ~ T3
- Atomicity : Transactions can be aborted. ~ T1, T2

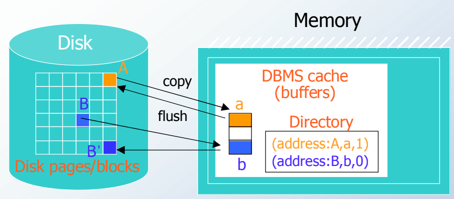
Basic Idea : “Logging”



- Undo/Redo by the Log
  → recover Non-catastrophic failure
- Full DB Backup
  > Differential Backup
  > (Transaction) Log

  } Catastrophic failure

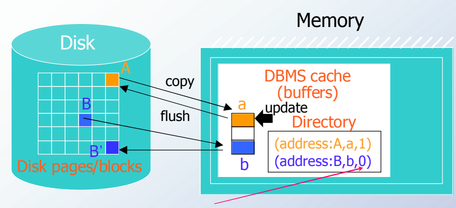
Physical View - How they work - (1)



Action :

1) Check the directory whether in the cache
2) If none, copy from disk pages to the cache
3) For the copy, old buffers needs to be flushed from the cache to the disk pages

Physical View - How they work - (2)



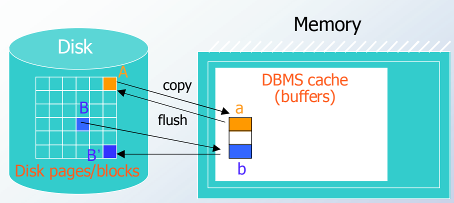
4) Flush only if a dirty bit is 1

Dirty bit : (in the directory) whether there is a change after copy to the cache

1 – updated in the cache

0 – not updated in the cache (no need to flush)

Physical View - How they work - (3)

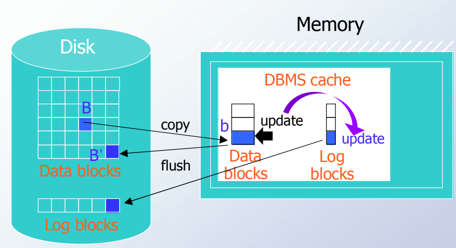


A-a : “in-place updating”

- when flushing, overwrite at the same location
- logging is required

B-b : “shadowing”

Physical View - How they work - (4)



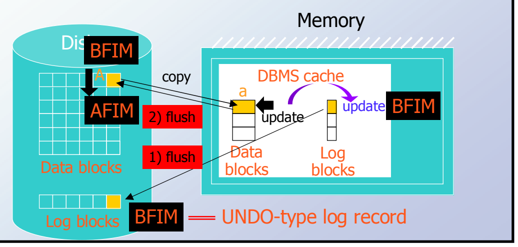
(1) copy (from the disk to the cache)

(2) update the cached data, record it in the log

(3) flush the log and the data (from the cache to the disk)

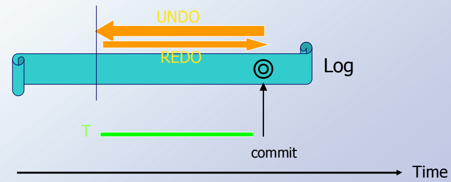
WAL : Write-Ahead Logging (1)

- in-place updating → A log is necessary
  BFIM (BeFore IMage) – overwrite – AFIM (AFter)
- WAL (Write-Ahead Logging)
  Log entries flushed before overwriting main data



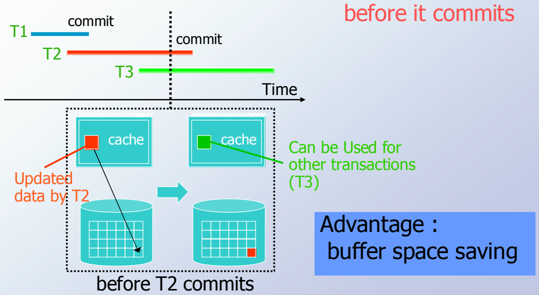
WAL : Write-Ahead Logging (2)

- WAL protocol requires UNDO and REDO
  - BFIM cannot be overwritten by AFIM on disk until all UNDO-type log have force-written to disk.
  - The commit operation cannot be completed until all UNDO/REDO-type log have force-written.



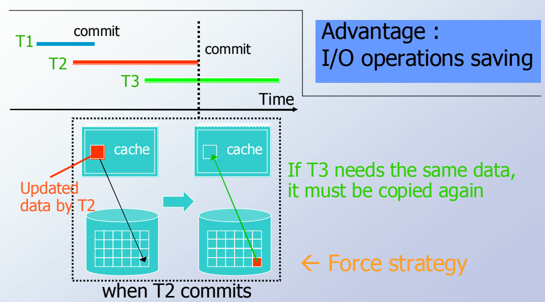
Steal & No-Force (1)

- Typical DB employs a steal/no-force strategy
- Steal strategy : a transaction can be written to disk before it commits



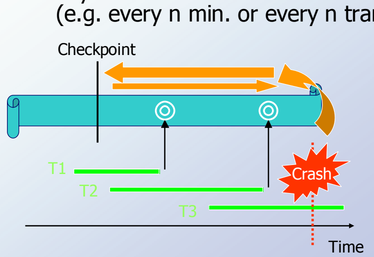
Steal & No-Force (2)

- No-Force strategy : a transaction need not to be written to disk immediately when it commits



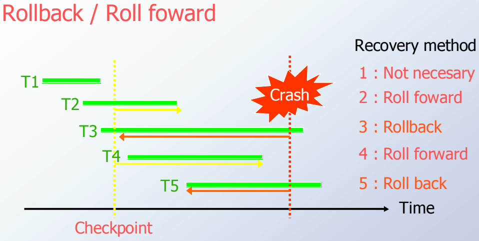
Checkpointing

- Checkpoint
  - All DMBS buffers modified are wrote out to disk.
  - A record is written into the log. ([checkpoint])
  - Periodically done
    (e.g. every n min. or every n transaction



Transaction Rollback (1)

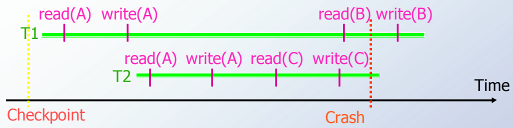
- Rollback / Roll foward



- Steal : transaction may be written on disk before it commits

Transaction Rollback (2)

- example :



Name	Account
Mr.A	$10
Mr.B	$2,000
Mr.C	$30,000

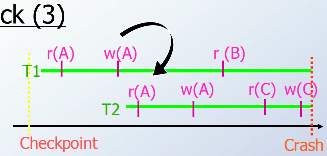
T1 : A company pays salary to employees
i) transfer $2,000 to Mr. A's account
ii) transfer $2,500 to Mr B's account …

T2 : Mr.A pays the monthly rent.
i) withdraw $1,500 from Mr.A's account
ii) transfer $1,500 to Mr.C's account

Transaction Rollback (3)

- Cascading Rollback

-T1 is interrupted
(needs rollback)



System Log	A	C
[checkpoint]	$10	$30,000
[start_transaction, T1]		
[read_item, T1, A]	$10	
[write_item, T1, A, 10, 2010]	$2,010	
[start_transaction, T2]		
[read_item, T2, A]	$2,010	
[write_item, T2, A, 2010, 510]	$510	
[read_item, T1, B]		
[read_item, T2, C]		$30,000
[write_item, T2, C, 1500, 31500]		$31,500
~~ CRASH ~~~~		

-T2 uses value modified by T1 (also needs rollback)

Categorization of Recovery Algorithm

- Deferred update
  - Do not physically update the database on disk until after a transaction reaches its commit point
  - Known as the No-UNDO/REDO algorithm

- Immediate update
  - The database may be updated before a transaction reaches its commit point
  - Both UNDO and REDO are required
  - Known as the UNDO/REDO algorithm