Databases: transaction processing

P.A.Rounce
Room 6.18
p.rounce@cs.ucl.ac.uk

ACID

Database operation is processing of a set of "transactions"

Required features of a database transaction should be

- **Atomicity – all or no changes should occur**
- **Consistency – should leave database in consistent state**
- **Isolation – should not affect other transactions**
- **Durability – results should persist and not be undone**

see: http://www.absoluteastronomy.com/encyclopedia/a/ac/acid.htm

Transaction example

Banking example: transfer £80 from Mary to Joe

Transaction, **T**	Let: $\mathbf{Balance_{Mary}}$ **= 2000** $\mathbf{Balance_{Joe}}$ **= -10** **Total = 1990**
Read(Mary)	
Mary := Mary – 80	
Write(Mary)	Bal_{Mary} = 1920, Bal_{Joe} = -10 **Total = 1910**
Read(Joe)	
Joe := Joe + 80	
Write(Joe)	Bal_{Mary} = 1920, Bal_{Joe} = 70 **Total = 1990**

Notation: Read(X) – read data item X from database into local variable X
Write(X) – write value from local variable X to database data item X

Transaction Failure:

Banking example: transfer £80 from Mary to Joe

Transaction fails after Write(Mary)!

Transaction, **T**	Let: $\mathbf{Balance_{Mary}}$ **= 2000** $\mathbf{Balance_{Joe}}$ **= -10** **Total = 1990**
Read(Mary)	
Mary := Mary – 80	
Write(Mary)	Bal_{Mary} = 1920, Bal_{Joe} = -10 Inconsistent state **Total ≠ 1910**
Read(Joe)	
Joe := Joe + 80	
Write(Joe)	

Must "undo" Write(Mary) during recovery from failure!

Atomicity: both Writes must succeed or none!

Consistency: database must not be left in inconsistent state

Transaction isolation

Banking example: transfer £80 from Mary to Joe

Transaction, **T**	Let: $\mathbf{Balance_{Mary} = 2000}$ $\mathbf{Balance_{Joe} = -10}$	**Total = 1990**
Read(Mary)		
Mary := Mary – 80		
Write(Mary)	$Bal_{Mary} = 1920, Bal_{Joe} = -10$	**Total = 1910**
Read(Joe)		
Joe := Joe + 80		
Write(Joe)	$Bal_{Mary} = 1920, Bal_{Joe} = 70$	**Total = 1990**

Isolation: No other transaction must see ($Bal_{Mary} = 1920, Bal_{Joe} = -10$)!

Durability: stable storage and logs

Storage types:

- volatile storage (main memory) – doesn't survive system crash
- non-volatile storage (disks & tapes) – usually survives system crash
- stable storage (theoretical) – *never lost!*

Transaction Writes (e.g. Write(Mary)):

1st : go to a buffer in main memory

2nd : go to disk

3rd: go to archival stable storage

Failure Recovery: *need log file of transaction operations in stable storage*

Logging

Transaction logging: **updates must be logged before update**

Why?

Suppose update then log, and a system crash occurs after update, but before log is made:

If update is only in volatile storage, update is lost!

Requirement:

log records must be written to stable storage before updates occur!

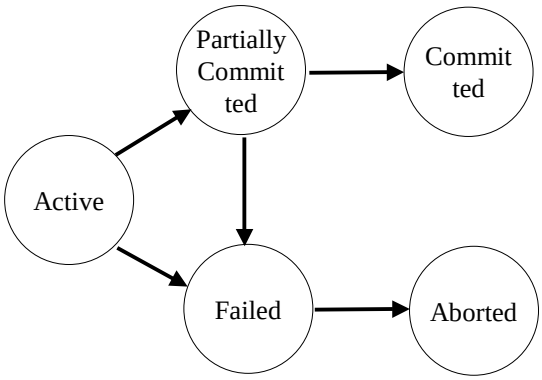
How is stable storage implemented on disks!

Have 2 or more copies of everything on separate disks if possible (disk mirroring)!

Failure

- There are several types of failures:
  - Logical errors: the internal state of a transaction is inconsistent, e.g., bad input, run out of memory, etc.
  - system errors or undesirable system states: e.g., deadlocks.
  - system crash: hardware malfunctions,
  - disk failures: disk looses a block of data due to head crashes.

Transaction Processing: transaction states



Transaction states:

Active: initial state

Partially committed: after last statement processed

Failed: after discovery that processing can no longer proceed

Aborted: after transaction rollback to state prior to transaction

Committed: after successful completion (after updates to database!)

Transaction logging Data

Log records

Log:

transaction start: **$<T_i$ start$>$**

transaction writes: **$<T_i, X_j, V_1, V_2>$**

transaction commit: **$<T_i$ commit$>$**

T_i – unique transaction identifier

X_j – data item identifier

V_1 – value of X_j before write

V_2 – new value of X_j (value after write)

Transaction example of logging

Let: $\mathbf{Balance_{Mary} = 2000}$
$\mathbf{Balance_{Joe} = -10}$

Transaction, **T**	Log records
Read(Mary)	<T start>
Mary := Mary – 80	
Write(Mary)	<T, Mary, 2000, 1920>
Read(Joe)	
Joe := Joe + 80	
Write(Joe)	<T, Joe, -10, 70>
	<T commit>

Larger logging example: 2 transactions

Let: $\mathbf{Balance_{Mary} = 2000, Balance_{Joe} = -10\ \ Balance_{Zack} = 230}$

Transactions	Log records	
T_1: Read(Mary)	$<T_1$ start>	
T_1: Mary := Mary – 80		
T_1: Write(Mary)	$<T_1$, Mary, 2000, 1920>	
T_1: Read(Joe)		
T_1: Joe := Joe + 80		
T_1: Write(Joe)	$<T_1$, Joe, -10, 70>	
	$<T_1$ commit>	
T_2: Read(Zack)	$<T_2$ start>	
T_2: Zack := Zack + 100	$<T_2$, Zack, 230, 330>	*Log contains sequence of transactions*
T_2: Write(Zack)	$<T_2$ commit>	

Deferred Database Modification

- This technique ensures **atomicity** by deferring all writes of a transaction until transaction partially commits.
- When transaction partially commits, the log records for the transaction are used in executing the deferred writes

(Remember the log records are written to stable storage before these writes)

System actions:

- Log record <T start> is written to log
- Each Write(X) creates a write record <T, X,V_1,V_2> in log
- At end of transaction (on partially commiting) <T commit> record written to log
- Log records written to stable storage
- Log consulted and writes made to database

Deferred Modification: failure recovery

On system failure, the log is used to recover the database to a consistent state.

Recovery process:

Go to start of log and read log records sequentially

For each transaction with a <T_i start> and a <T_i commit> record:

execute a *redo(T_i)* operation

redo: sets all values written to by T_i to their new values(V_2)

When all log records have been analysed, database will be in consistent state.

All transaction with a <T_i start> but no <T_i commit> will have to be re-executed: remember database writes are deferred until after <T commit> record written, so no other action is required on incomplete transactions.

Note: old value, V_1 is not used – so doesn't need to be logged for deferred modification.

Immediate Database Modification

- This technique updates the database as Write operations occur.
- Log records are written to stable storage before each Write.
- There is increased disk activity because of this.

System actions during transaction

- Log record <T start> is written to log
- Each Write(X) creates a write record <T, X,V_1,V_2> in log
- Log records written to stable storage
- New value of X is written into database
- When transaction partially commits, <T commit> record written to log.

This technique needs an *undo(X)* operation during failure recovery

Immediate Modification: failure recovery

On system failure, the log is used to recover the database to a consistent state.

Recovery process:

Go to start of log and read log records sequentially

For each transaction with a <T_i start> and a <T_i commit> record:

execute a *redo(T_i)* operation

For each transaction with a <T_i start> but no <T_i commit> record:

execute a undo*(T_i)* operation

redo: sets all values written to by T_i to their new values(V_2)

undo: sets all values written to by T_i to their values(V_1) at the start of T_i

Note: old value, V_1 is used for this technique

Recovery example: 2 transactions

Let: $\mathbf{Balance_{Mary} = 2000, Balance_{Joe} = -10\ \ Balance_{Zack} = 230}$

Log records

$<T_1$ start>

$<T_1$, Mary, 2000, 1920>

$<T_1$, Joe, -10, 70>
$<T_1$ commit>
$<T_2$ start>
$<T_2$, Zack, 230, 330>

Deferred Modification Recovery

Redo(T_1): // commit record found
Mary <-- 1920
Joe <-- 70

Immediate Modification Recovery

Redo(T_1): // commit record found
Mary <-- 1920
Joe <-- 70
Undo(T_2): // no commit record found
Zack <-- 230

Checkpoints: reducing recovery activity

In theory, log file needs to be read from beginning during recovery failure.

This process is time-consuming if log is large: theoretically log contains all activity since database created.

Most transactions in log will have had their updates written into stable storage.

Checkpoint actions:-
output all log records currently in main memory to stable storage
output all modified data buffers in main memory into stable storage
output <checkpoint> log record to stable storage.

On failure recovery:
search log backwards from end looking for latest <checkpoint> record – start recovery from this point in log.