TRANSACTION MANAGEMENT

TRANSACTION

→ is a sequence of operations on database objects

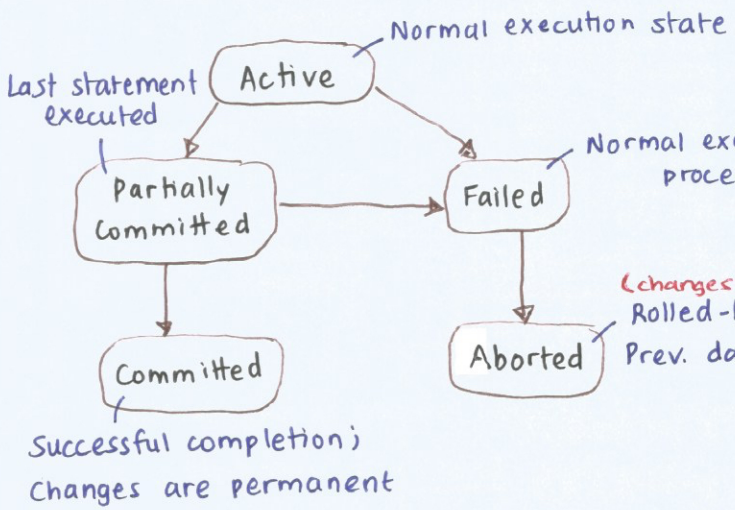
↳ All operations together form a single logical unit

i.e. Transfer £100 from account A to account B

form a single logical unit:

1. Read bal. from A into local buffer x
2. x := x − 100
3. Write new bal. x to A
4. Read bal. from B into local buffer y
5. y := y + 100
6. Write new bal. y to B

LIFE-CYCLE OF A TRANSACTION



we don't want some of the operations to be executed while others are not; they represent a single logical unit!

SCHEDULE

→ is a sequence S of operations frm. a set of transactions such tht. the order of operations in each transaction is the same as in S

e.g.

Transaction T_1 : op1, op2, op3

T_2 : op1', op2'

Concurrent schedule

	T_1	T_2
1		op1'
2	op1	
3	op2	
4		op2'
5	op3	

Serial schedule

	T_1	T_2
1		op1'
2		op2'
3	op1	
4	op2	
5	op3	

Note that order of operations is the same as in T_1 & T_2

CONCURRENT execution

→ The operations of different transaction are interleaved

↳ ↑ throughput

↳ ↓ response time

CONCURRENCY

→ Typically >1 transaction runs on a system

→ Each transaction consists of many I/O and CPU operations

→ Hence, we want concurrency bc. we don't want to wait for a transaction to completely finish before executing another

SERIAL execution

→ A schedule is serial if all operations of each transaction are executed before/after all operations of another transaction

MOTIVATING EXAMPLE

T_1 : transfer £100 from account A to account B

T_2 : transfer 10% of account A to account B

T_1
1. $x := \text{read}(A)$
2. $x := x - 100$
3. $\text{write}(x, A)$
4. $y := \text{read}(B)$
5. $y := y + 100$
6. $\text{write}(y, B)$

T_2
1. $x := \text{read}(A)$
2. $y := 0.1 * x$
3. $x := x - y$
4. $\text{write}(x, A)$
5. $z := \text{read}(B)$
6. $z := z + y$
7. $\text{write}(z, B)$

$A + B$ should not change:
Money is not created and does not disappear

SERIAL EXECUTION

Do T_1 first, then T_2

	T_1	T_2	Database	
1	$x := \text{read}(A)$		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3	$\text{write}(x, A)$		$A = 900$	$B = 1000$
4	$y := \text{read}(B)$		$A = 900$	$B = 1000$
5	$y := y + 100$		$A = 900$	$B = 1000$
6	$\text{write}(y, B)$		$A = 900$	$B = 1100$
7		$x := \text{read}(A)$	$A = 900$	$B = 1100$
8		$y := 0.1 * x$	$A = 900$	$B = 1100$
9		$x := x - y$	$A = 900$	$B = 1100$
10		$\text{write}(x, A)$	$A = 810$	$B = 1100$
11		$z := \text{read}(B)$	$A = 810$	$B = 1100$
12		$z := z + y$	$A = 810$	$B = 1100$
13		$\text{write}(z, B)$	$A = 810$	$B = 1190$

A+B = 2000 (✓)

Do T_2 first, then T_1

	T_1	T_2	Database	
1		$x := \text{read}(A)$	$A = 1000$	$B = 1000$
2		$y := 0.1 * x$	$A = 1000$	$B = 1000$
3		$x := x - y$	$A = 1000$	$B = 1000$
4		$\text{write}(x, A)$	$A = 900$	$B = 1000$
5		$z := \text{read}(B)$	$A = 900$	$B = 1000$
6		$z := z + y$	$A = 900$	$B = 1000$
7		$\text{write}(z, B)$	$A = 900$	$B = 1100$
8	$x := \text{read}(A)$		$A = 900$	$B = 1100$
9	$x := x - 100$		$A = 900$	$B = 1100$
10	$\text{write}(x, A)$		$A = 800$	$B = 1100$
11	$y := \text{read}(B)$		$A = 800$	$B = 1100$
12	$y := y + 100$		$A = 800$	$B = 1100$
13	$\text{write}(y, B)$		$A = 800$	$B = 1200$

A+B = 2000 (✓)

Balances depend on the serial ordering

CONCURRENT EXECUTION

Schedule #1:

	T_1	T_2	Database	
1	$x := \text{read}(A)$		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3	$\text{write}(x, A)$		$A = 900$	$B = 1000$
4		$x := \text{read}(A)$	$A = 900$	$B = 1000$
5		$y := 0.1 * x$	$A = 900$	$B = 1000$
6		$x := x - y$	$A = 900$	$B = 1000$
7		$\text{write}(x, A)$	$A = 810$	$B = 1000$
8	$y := \text{read}(B)$		$A = 810$	$B = 1000$
9	$y := y + 100$		$A = 810$	$B = 1000$
10	$\text{write}(y, B)$		$A = 810$	$B = 1100$
11		$z := \text{read}(B)$	$A = 810$	$B = 1100$
12		$z := z + y$	$A = 810$	$B = 1100$
13		$\text{write}(z, B)$	$A = 810$	$B = 1190$

A+B = 2000 (✓)

Schedule #2:

	T_1	T_2	Database	
1	$x := \text{read}(A)$		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3	900	$x := \text{read}(A)$ (1000)	$A = 1000$	$B = 1000$
4		$y := 0.1 * x$	$A = 1000$	$B = 1000$
5		$x := x - y$	$A = 1000$	$B = 1000$
6		$\text{write}(x, A)$	$A = 900$	$B = 1000$
7	$\text{write}(x, A)$		$A = 900$	$B = 1000$
8	$y := \text{read}(B)$		$A = 900$	$B = 1000$
9	$y := y + 100$		$A = 900$	$B = 1000$
10	$\text{write}(y, B)$		$A = 900$	$B = 1100$
11		$z := \text{read}(B)$	$A = 900$	$B = 1100$
12		$z := z + y$	$A = 900$	$B = 1100$
13		$\text{write}(z, B)$	$A = 900$	$B = 1200$

We created £100 !!!

A+B = 2100 (X)

↳ From example above, it can be seen that the only important operations in scheduling are read and write.
↳ other operations don't affect the schedule

↳ We represent transactions by a sequence of read/write operations

↳ In the example above, transactions are represented as:

T_1 = r(A), w(A), r(B), w(B)

T_2 = r(A), w(A), r(B), w(B)

Schedule #1 is hence represented as:

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

→ is equivalent to a serial execution

→ Sched. #2 is not!

→ Interleave operations

SERIALIZABILITY

→ Two operations (from diff. transactions) are **conflicting** if:

- they refer to the same data item (AND...)
- at least one of them is a write

→ Two consecutive non-conflicting operations (from diff. transactions) in a schedule can be swopped

→ A schedule is **conflict serializable** if it can be transformed into a serial sched. by a sequence of swap operations

a bit cumbersome to work w/, so we use a...

PRECEDENCE GRAPH

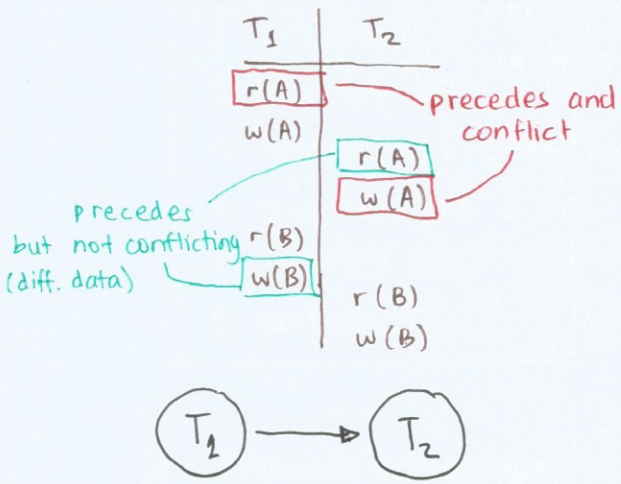
→ captures all potential conflicts between transactions in a schedule

- Each node is a transaction
- There is an edge from T_i to T_j ($T_i \neq T_j$) if an action of T_i (come before) precedes and conflicts w/ one of T_j's actions

→ A schedule is conflict serializable iff. its precedence graph is (acyclic)

→ An equivalent serial sched. is given by any topological sort over the precedence graph

e.g. Schedule #1



SCHEDULES W/ ABORTED TRANSACTIONS

→ Up until this pt., we assumed transactions commit successfully after last operation

→ But abort and commit must be taken explicitly into account

i.e.

CASE 1

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
abort	

→ A 'dirty read'; T_2 is reading uncommitted changes made by T_1 but T_2 has not yet committed.

(CASCADING ABORT)

→ Can recover frm. this situation by aborting also T_2

CASE 2

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
abort	

→ Similar situation: T_2 makes a dirty read but T_2 has already committed. Hence...

→ We cannot do cascading abort!

This sched. is unrecoverable

→ We actually want sched. that are recoverable w/o cascading aborts.

- Achieved by ensuring tht. transactions commit only after all transactions whose changes they read commit

e.g. The 'commit' in T_2 of CASE 2 would NOT be allowed bc. T_2 reads change made by T_1 and T_1 has not yet committed

LOCK-BASED CONCURRENCY CONTROL

→ **Lock** is a book-keeping object associated w/ a data item

- ↳ tells us whether the data item is available for read and/or write
- ↳ always has an **owner** which is the transaction currently operating on the data item

↳ 2 KINDS of lock

- **Shared Lock** → data item is available for read to owner
  → can be acquired by more than one transaction
- **Exclusive Lock** → available for read & write
  → cannot be acquired by other transactions

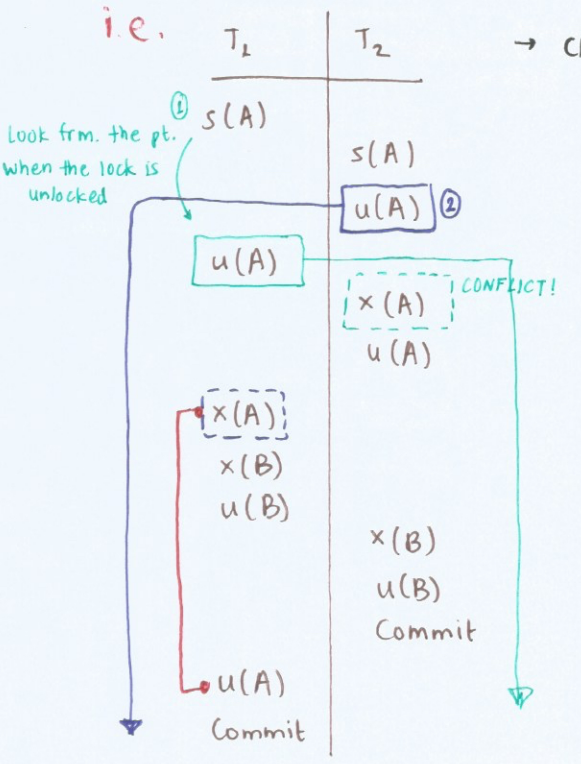
→ 2 locks on the same data item are conflicting if (at least) one of them is exclusive.

TRANSACTION MODEL w/ LOCKS

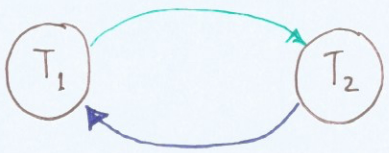
→ Operations:
- s(A) – shared lock on A is acquired
- x(A) – exclusive lock on A is acquired
- u(A) – lock on A is released

→ In a schedule:
1) A transaction cannot acquire a lock on A before all exclusive locks on A have been released
2) A transaction cannot acquire an exclusive lock on A before all locks on A have been released

i.e.



→ Check if it's conflict serializable – build precedence graph



→ NOT conflict serializable!

TWO-PHASE LOCKING (2PL)

1.) Before reading / writing a data item, a transaction must acquire a shared / exclusive lock on it.

2.) A transaction cannot request additional locks once it releases any locks.

↳ This is so tht. each transaction has:

- **Growing Phase** when locks are acquired
- **Shrinking Phase** when locks are released

→ Every completed sched. of committed transactions tht. follow 2PL protocol is conflict serializable

2PL & ABORTED TRANSACTIONS

T_1	T_2
x(A)	
u(A)	
	x(A)
	x(B)
	u(A)
	u(B)
	Commit
Abort	

grows, then shrinks

→ T_1 & T_2 follow 2PL and still, the sched. is unrecoverable

∴ 2PL does not guarantee tht. sched. w/ aborted transactions are recoverable

STRICT 2PL

→ Rule #2 changes to:

All locks held by a transaction are released when the transaction is completed (aborts / commits)

→ Ensures tht:

1) The sched. is always recoverable
2) The sched. is conflict serializable
3) All aborted transactions can be rolled back without cascading aborts

DEADLOCKS

→ All transactions requesting a lock must wait until all conflicting locks are released

→ may get a cycle of 'waits'

T_1	T_2	T_3
s(A)		
	x(B)	
req s(B)		
		s(C)
	req x(C)	
		req x(A)

waits for T_2 to release the lock on B

→ T_1 waits for T_2

T_2 waits for T_3

T_3 waits for T_1

cycle of 'waits' [DEADLOCK!]

HOW TO PREVENT? [APPROACH #1]

→ Each transaction is assigned a priority using a timestamp

(The older the transaction is, the higher priority it has)

→ Suppose T_i requests a lock and T_j holds a conflicting lock

- 2 policies to prevent deadlocks
  - **Wait-die** — T_i waits if it has ↑priority, otherwise aborted
  - **Wound-wait** — T_j aborted if T_i has ↑priority, otherwise T_i waits

	$T_i > T_j$	$T_i \leq T_j$
Wait-die	T_i waits	T_i dies
Wound-wait	T_j dies	T_i waits

↑priority transaction is never aborted

may lead to → **Starvation** — a transaction keeps being aborted bc. it never has sufficiently high priority

SOLN → restart aborted transactions w/ their initial timestamp

DEADLOCK DETECTION [APPROACH #2]

→ We let deadlocks happen but we detect and solve them

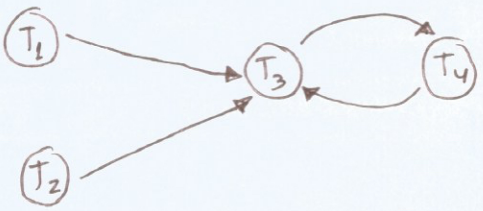
→ **Waits-for-graph**

- Nodes are transactions
- There is an edge from T_i to T_j ($T_i \neq T_j$) ← Transaction is never waiting for itself
  if T_i waits for T_j to release a (conflicting) lock

Each cycle represents a deadlock

→ **Recovering from deadlocks**

- Choose a minimal set of transactions such that rolling them back will make the waits-for-graph acyclic



→ Deleting T_3 is better than deleting T_4 bc. we freed resources not only for T_4, but also for T_1 and T_2

CRASH RECOVERY (Note that crash ≠ error)

→ The **log** (a.k.a. trail/journal) records every action executed on the dbase

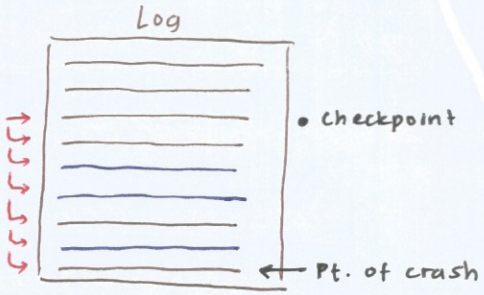
→ Each log record has a unique ID called log sequence number (LSN)

→ Fields in a log record:

1) LSN
2) prevLSN
3) trans ID
4) type
5) before – value before the change
6) after – value after the change

→ The state of the dbase is periodically recorded as a checkpoint

ARIES



→ recovery algo. used in major DBMSs

→ works in 3 phases:

1. ANALYSIS

- identify changes tht. have not been written to disk
- identify active transactions at the time of crash

2. REDO

- repeat all actions starting from latest checkpoint
- restore the dbase to the state at the time of crash

3. UNDO

- undo actions of transactions tht. did not commit
- the dbase reflects only actions of committed transactions

→ This is based on these three principles:

changes are in the log but they are not reflected in the dbase storage

1) Write-ahead logging

- Before writing a change to disk, a corresponding log record must be inserted and the log is put in a stable storage

2) Repeating history during Redo

- Actions before the crash are retracted to bring the dbase to the state it should have been when the sys. crashed

3) Logging changes during Undo

- changes made while undoing transactions are also logged (Protection frm. further crashes)