Transaction Management

Dr Paolo Guagliardo

THE UNIVERSITY of EDINBURGH
informatics

Fall 2020 (v20.1.0)

Transactions

Transaction: a sequence of operations on database objects

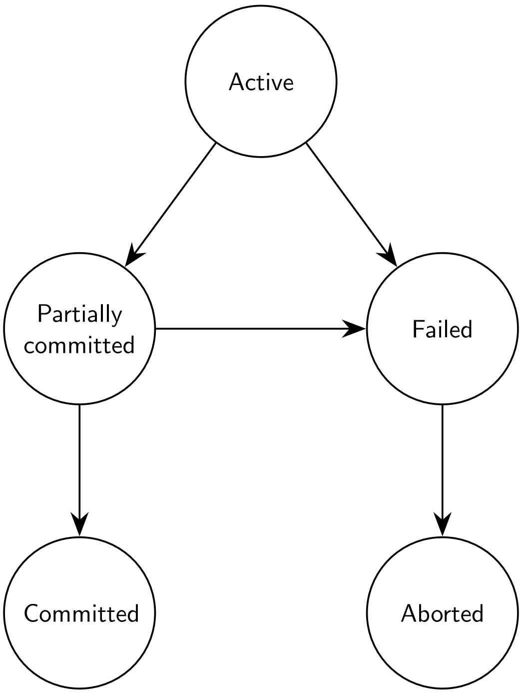
- All operations together form a **single logical unit**

Example

Transfer £100 from account A to account B

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

Life-cycle of a transaction



Active
Normal execution state

Partially Committed
Last statement executed

Failed
Normal execution cannot proceed

Aborted
Rolled-back
Previous database state restored

Committed
Successful completion
Changes are permanent

Schedules

Schedule: a sequence S of operations from a set of transactions s.t. the order of operations in each transaction is **the same** as in S

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

Example

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	

Concurrency

- Typically more than one transaction runs on a system
- Each transation consists of many I/O and CPU operations
- We don't want to wait for a transaction to completely finish before executing another

Concurrent execution

The operations of different transaction are **interleaved**

- increases throughput
- reduces response time

The ACID properties

Atomicity
: Either all operations are carried out or none are

Consistency
: Successful execution of a transaction leaves the database in a coherent state

Isolation
: Each transaction is protected from the effects of other transactions executed concurrently

Durability
: On successful completion, changes persist

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. $x :=$ read(A)
2. $x := x - 100$
3. write(x, A)
4. $y :=$ read(B)
5. $y := y + 100$
6. write(y, B)

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

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

Motivating example: Serial execution 1

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

Motivating example: Serial execution 2

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

Motivating example: Concurrent execution 1

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

Motivating example: Concurrent execution 2

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

We created £100 !!!

Transaction model

The only important operations in scheduling are **read** and **write**

r(A) read data item A

w(A) write data item A

Other operations do not affect the schedule

We represent transactions by a sequence of read/write operations

The transactions in the motivating example are represented as:

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

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

Transaction model: Schedules

The schedules in the motivating example are represented as:

Schedule 1

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

Schedule 2

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

Schedule 1 is **equivalent to a serial execution**, Schedule 2 is not

Serializability

Two operations (from different 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 in a schedule can be swapped

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

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 (for $T_i \neq T_j$) if an action of T_i **precedes** and **conflicts** with one of T_j's actions

A schedule is conflict serializable

if and only if

its precedence graph is acyclic

An equivalent serial schedule is given by any **topological sort** over the precedence graph

Precedence graph: Example

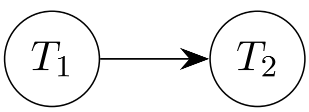
Schedule 1

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

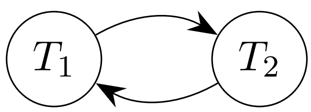
Schedule 2

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

Precedence graph



Precedence graph



Schedules with aborted transations (1)

We assumed transactions commit successfully after the last operation

But abort and commit must be taken excplicitly into account

	T_1	T_2
1	r(A)	
2	w(A)	
3		r(A)
4		w(A)
5		r(B)
6		w(B)
7	Abort	

- T_2 read uncommited changes made by T_1
- But T_2 has not yet committed
- We can recover by aborting also T_2 (**cascading abort**)

Schedules with aborted transations (2)

	T_1	T_2
1	r(A)	
2	w(A)	
3		r(A)
4		w(A)
5		r(B)
6		w(B)
7		Commit
8	Abort	

- T_2 read uncommited changes made by T_1
- But T_2 has already committed
- The schedule is **unrecoverable**

Recoverable schedules without cascading aborts

Transactions commit only after, and if,
all transactions whose changes they read commit

Lock-based concurrency control

Lock

- Bookkepeing object associated with a data item
- Tells whether the data item is available for read and/or write
- Owner: Transaction currently operating on the data item

Shared lock	Data item is available for read to owner Can be acquired by more than one transaction
Exclusive lock	Data item is available for read/write to owner Cannot be acquired by other transactions

Two locks on the same data item are **conflicting**
if one of them is exclusive

Transaction model with 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
Abort	transaction aborts
Commit	transaction commits

In a schedule:

- A transaction cannot acquire a lock on A
  before all exclusive locks on A have been released
- A transaction cannot acquire an exclusive lock on A
  before all locks on A have been released

Examples of schedules with locking

Schedule 1

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

Schedule 2

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

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** lock

Each transaction has

Growing phase when locks are acquired

Shrinking phase when locks are released

Every completed schedule of committed transactions
that follow the 2PL protocol is conflict serializable

2PL and aborted transactions

	T_1	T_2
1	x(A)	
2	u(A)	
3		x(A)
4		x(B)
5		u(A)
6		u(B)
7		Commit
8	Abort	

- T_1 and T_2 follow 2PL
- But T_1 cannot be undone
- The schedule is **unrecoverable**

Strict 2PL

1. Before reading/writing a data item
   a transaction must acquire a shared/exclusive lock on it
2. **All locks held by a transaction are released
   when the transaction is completed** (aborts or commits)

Ensures that

- The schedule is always recoverable
- All aborted transactions can be rolled back
  without cascading aborts
- The schedule consisting of the committed transactions
  is conflict serializable

Deadlocks

A transaction requesting a lock must wait
until all conflicting locks are released

We may get a cycle of "waits"

	T_1	T_2	T_3
1	s(A)		
2		x(B)	
3	req s(B)		
4			s(C)
5		req x(C)	
6			req x(A)

T_1 waits for T_2 , T_2 waits for T_3 , T_3 waits for T_1

Deadlock prevention

Each transaction is assigned a **priority** using a timestamp:
The older a transaction is, the higher priority it has

Suppose T_i requests a lock and T_j holds a conflicting lock

Two policies to prevent deadlocks:

Wait-die: T_i waits if it has higher priority, otherwise aborted

Wound-wait: T_j aborted if T_i has higher priority, otherwise T_i waits

In both schemes, the higher priority transaction is never aborted

Starvation: a transaction keeps being aborted
because it never has sufficiently high priority

Solution: restart aborted transactions with their initial timestamp

Deadlock detection

Waits-for graph

- Nodes are active transactions
- There is an edge from T_i to T_j (with $T_i \neq T_j$)
  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

Crash recovery

The log (a.k.a. **trail** or **journal**)

Records every action executed on the database

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

Fields in a log record:

LSN	ID of the record
prevLSN	LSN of previous log record
transID	ID of the transaction
type	of action recorded
before	value before the change
after	value after the change

The state of the database is periodically recorded as a **checkpoint**

ARIES

Recovery algorithm used in major DBMSs

Works in three phases

1. Analysis
   - identify changes that 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 database to the state at the time of crash
3. Undo
   - undo actions of transactions that did not commit
   - the database reflects only actions of committed transactions

Principles behind ARIES

Write-Ahead Logging

Before writing a change to disk, a corresponding log record must be inserted and the log forced to stable storable

Repeating history during Redo

Actions before the crash are retraced to bring the database to the state it was when the system crashed

Logging changes during Undo

Changes made while undoing transactions are also logged (protection from further crashes)