Introduction to Databases

(INFR10080)

(Transactions)

Instructor: Yang Cao

(Fall 2025)

THE UNIVERSITY *of* EDINBURGH

Changelog

v25.0 Initial version

Transactions

Transaction: a sequence of operations on database objects

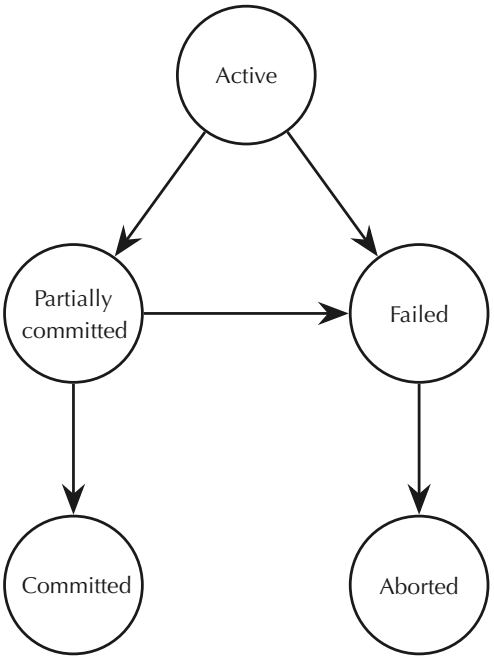
- All operations together form a **single logical unit**

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

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

AID: properties of systems
C: database (data) property, not about database systems

Why Isolation?

One of the most critical guarantees that a database system offers:

(a) Database always moves from one *consistent* state to another *consistent* state when a transaction commits

(b) Applications always see *consistent* database states.

Why challenging?

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

Without isolation, concurrency can cause inconsistent database states

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. $\text{write}(z, B)$

Consistency: $A + B$ should not change:

Money is not created and does not disappear

Motivating example: Serial execution 1

	T_1	T_2	Database	
1	$x := \text{read}(A)$		$A = 1000$	$B = 1000$
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		$\text{write}(z, B)$	$A = 810$	$B = 1190$

Motivating example: Serial execution 2

	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	$\text{write}(y, B)$		$A = 800$	$B = 1200$

Motivating example: Concurrent execution 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		$\text{write}(z, B)$	$A = 810$	$B = 1190$

Motivating example: Concurrent execution 2

	T_1	T_2	Database	
1	$x := \text{read}(A)$		$A = 1000$	$B = 1000$
2	$x := x - 100$		$A = 1000$	$B = 1000$
3		$x := \text{read}(A)$	$A = 1000$	$B = 1000$
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 !!!

Real-life case: AWS outage / worldwide chaos

News: https://www.bbc.co.uk/news/live/c5y8k7k6v1rt, https://www.theguardian.com/technology/2025/oct/24/amazon-reveals-cause-of-aws-outage

Date: 20 October 2025
Location: AWS's US-EAST-1 region, located in Northern Virginia.
Direct cause: empty DNS record for DynamoDB in US-EAST-1 region
Consequences: 1000+ services (e.g., Learn) down, cascading failures

Root cause: poor DNS routing design without sufficient isolation

How to avoid isolation issues?

Concurrency Control

Concurrency Control

Concurrency control protocols:

algorithmic rules to be followed by each *individual transaction*

permit only **concurrent executions** with **correct isolation**

We need formalisation to make these notions concrete

Transaction model

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

$\mathsf{r}(A)$ read data item A

$\mathsf{w}(A)$ 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 : $\mathsf{r}(A), \mathsf{w}(A), \mathsf{r}(B), \mathsf{w}(B)$

T_2 : $\mathsf{r}(A), \mathsf{w}(A), \mathsf{r}(B), \mathsf{w}(B)$

Transaction model: 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	

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
$r(A)$	
	$r(A)$
	$w(A)$
$w(A)$	
$r(B)$	
$w(B)$	
	$r(B)$
	$w(B)$

Alternative notation

Schedule 1 : $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

Schedule 2 : $r_1(A), r_2(A), w_2(A), w_1(A), r_1(B), w_1(B), r_2(B), w_2(B)$

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

In a schedule, two **consecutive** non-conflicting operations (from **different** transactions) 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

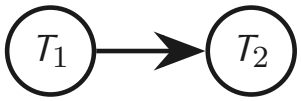
Schedule 1

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

Schedule 2

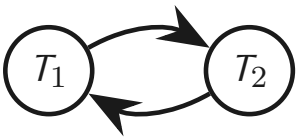
T_1	T_2
r(A)	
	r(A)
	w(A)
w(A)	
r(B)	
w(B)	
	r(B)
	w(B)

Precedence graph



(serializable schedule)

Precedence graph



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

Additional *lock* operations (injected by lock-based protocols at runtime):

$s(A)$	**shared lock** on A is acquired
$x(A)$	**exclusive lock** on A is acquired
$u(A)$	lock on A is released

In a schedule:

- 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

Only lock operations are shown (also referred to as *lock schedules*):

Schedule 1

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

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)
u(A)	

Two-Phase Locking (2PL)

A lock-based concurrency control protocol:

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

The protocol does **not** need to know all the reads/writes that a transaction will execute ahead of time.

Two-Phase Locking (2PL)

Isolation guarantee:

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

Not every completed serializable schedule is permitted by 2PL (okay)

However, under 2PL, a schedule may not complete ... (not okay)

Deadlocks

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

We may get a cycle of "waits" (during Growing phase of 2PL)

	T_1	T_2	T_3
1	$\mathsf{s}(A)$		
2		$\mathsf{x}(B)$	
3	req $\mathsf{s}(B)$		
4			$\mathsf{s}(C)$
5		req $\mathsf{x}(C)$	
6			req $\mathsf{x}(A)$

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

Schedules with aborted transations

To account for unsuccessful execution of transactions, we need to explicitly take into account abort and commit in transaction schedules.

Changes (writes) of T are persisted in the database only after T commits.

Writes before commit can be seen by concurrent transactions, though.

Aborted transaction will roll back: all its effect on database will be reverted.

Under 2PL, to resolve deadlock, a transaction in the shrinking phase will never be picked as the "victim" (i.e., be forced to abort).

However, transaction aborts can be caused by many other factors other than deadlock, e.g., constraint violation, triggers, or hardware failure.

Example: Transaction abort and roll back

Example: abort caused by **constraint violation** (application logic)

T: transfer money between accounts A ($9,000) and B ($9,950):

1. x(A) (Growing)
2. x(B) (Growing)
3. write($A - 100, A$)
4. u(A) (Shrinking phase begins)
5. write($B + 100, B$)
6. u(B)

Constraint (or application logic):

"No account can ever hold more than $10,000."

Database system (resp. application code) will abort T when it tries to update B with $B + 100$, which would violate the constraint (resp. application logic) if committed. A ROLLBACK command will be issued.

Even though T was in its shrinking phase, the entire transaction is aborted. The changes to both accounts A and B are undone (rollback).

Cascading abort

Aborting one transaction may force another transaction to abort, and so on

	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**)

Cascading abort

	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

2PL and aborted transactions

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

- T_1 and T_2 follow 2PL
- T_2 reads uncommited changes made by T_1
- But T_2 cannot be undone
- The schedule is **unrecoverable**

To summarize, 2PL permits

(some) conflict serializable schedules,

schedules with deadlocks (**resolvable**), and

unrecoverable schedules (**???**)

Strict 2PL

1. Growing phase: same as 2PL, before reading/writing a data item a transaction must acquire a shared/exclusive lock on it
2. No Shrinking phase: **The transaction holds all its locks until it is completed (aborts or commits)**
3. Lock Release: All locks are released **at once** in the end.

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

Concurrency Control, revisted

Concurrency control protocols:

algorithmic rules to be followed by each *individual transaction*

permit only **concurrent executions** with **correct isolation**

Questions: (not examinable, but **very** important)

protocols that are not lock-based?

permit all and only conflict-serializable schedules?

correct isolation = conflict serializability? can we challenge this?

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

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)