Transactions and Concurrency Control

Web Applications and Services

Spring Term

Naercio Magaia

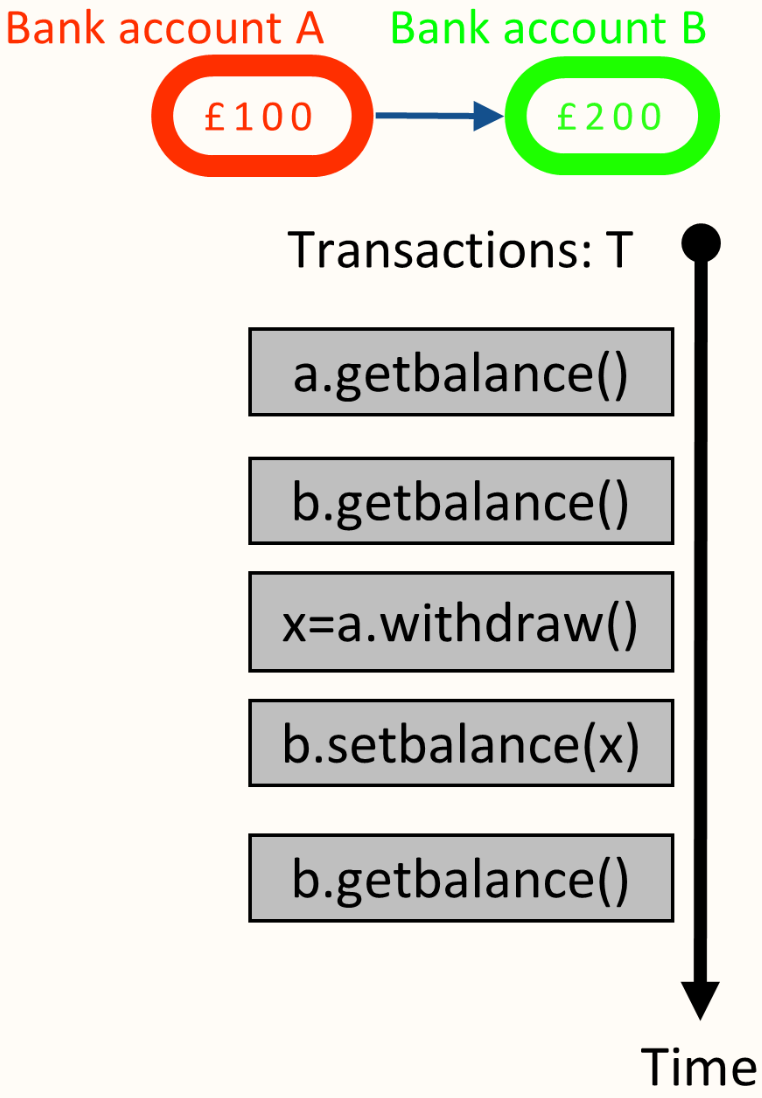
UNIVERSITY OF SUSSEX

Contents

UNIVERSITY OF SUSSEX

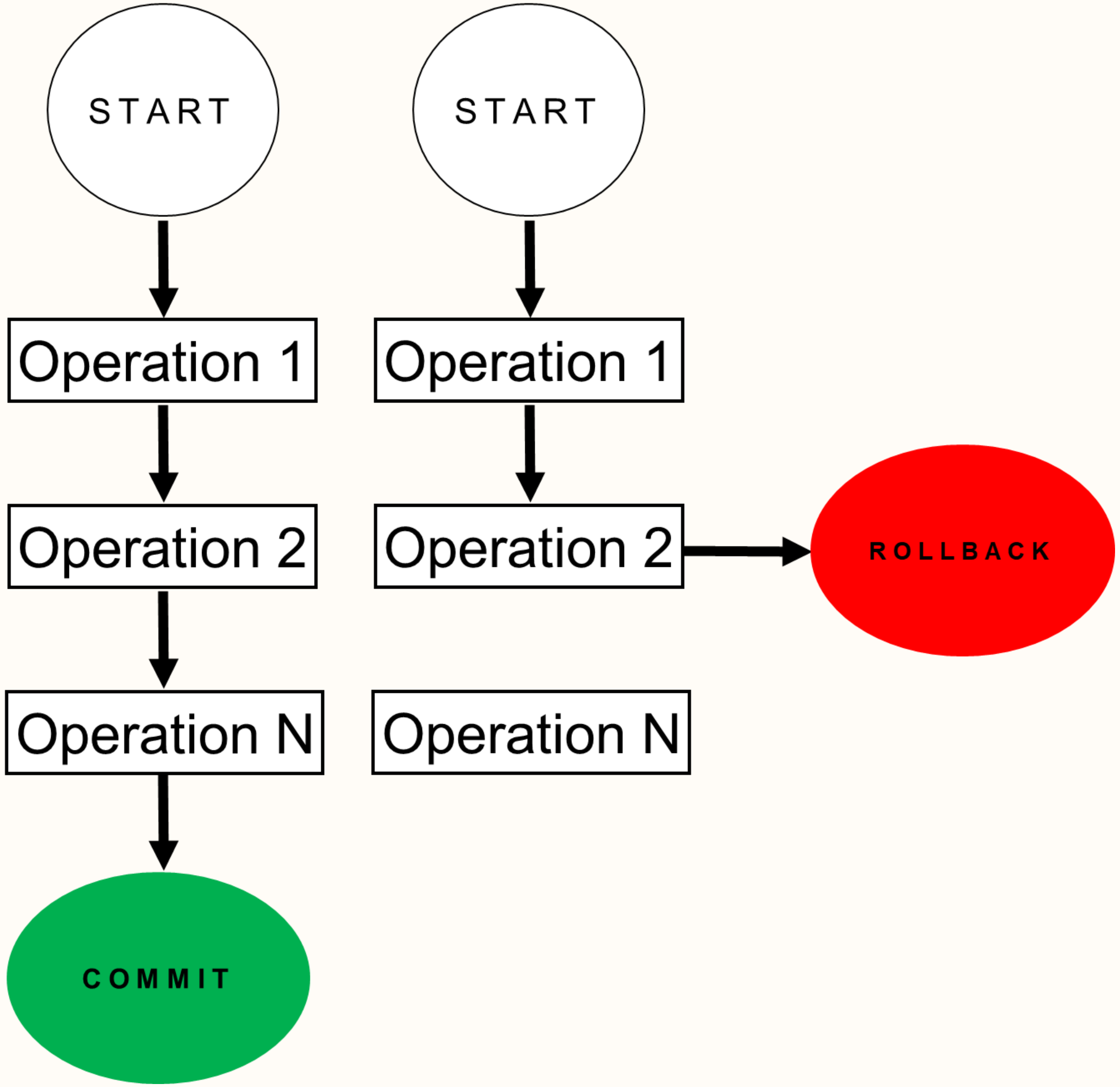
Transactions

- A set of operations on objects to be performed as an <u>indivisible unit</u> by the servers managing those objects
- Ensure that all objects managed remain in a consistent state
  - when accessed by multiple clients
  - in the presence of server crashes
- Recoverable objects: can be recovered after a server crash
  - Objects may be stored in volatile memory or persistent memory



Transactions

- A transaction is a collection of read/write operations succeeding only if all contained operations succeed.
- The data sources involved are all rolled back to their state as it was at the beginning of the operation.



UNIVERSITY OF SUSSEX

ACID properties

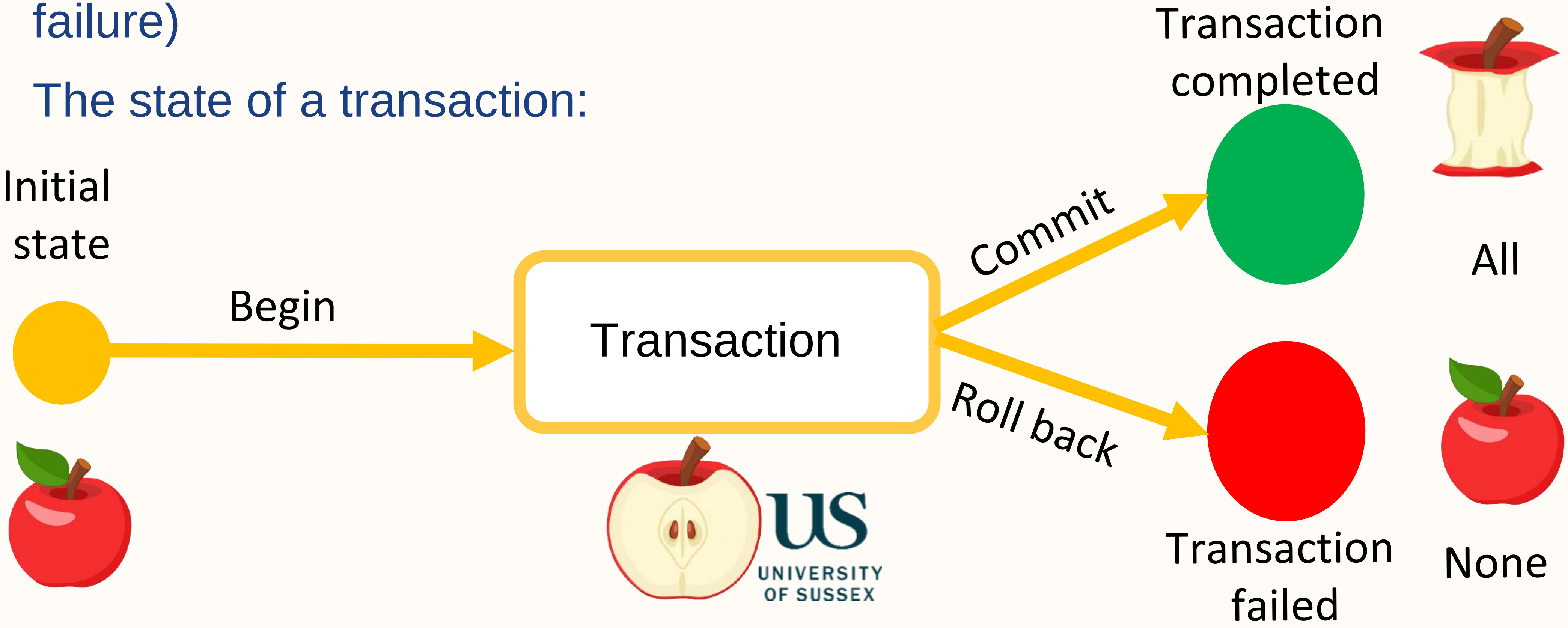
- ☑ Atomicity
- ☑ Consistency
- ☑ Isolation
- ☑ Durability

To ensure accuracy and data integrity

UNIVERSITY OF SUSSEX

ACID properties

- ✓ Atomicity: a transaction either completes successfully (the effects of all operations are recorded in the objects) or has no effect at all (aborted or failure)
- The state of a transaction:



ACID properties

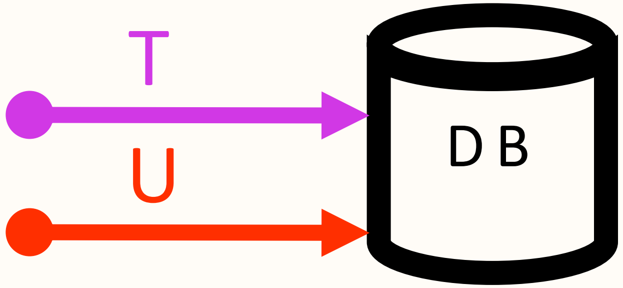
- ✓ Consistency : a transaction takes the system from one consistent state to another consistent state

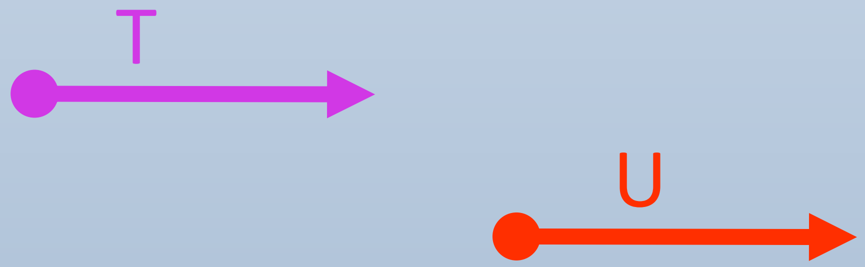
non-consistent?

UNIVERSITY OF SUSSEX

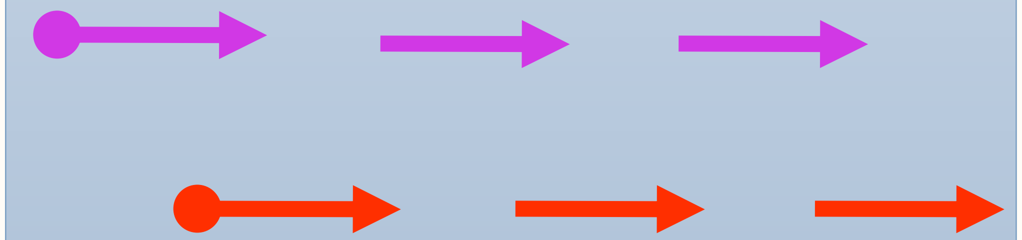
ACID properties

✓ Isolation: each transaction must be performed without interference from other transactions.





Easiest approach: perform transactions serially – one at a time, in some arbitrary order.



But the goal is to maximise concurrency

ACID properties

- ✓ Durability : after a transaction has completed successfully, all its effects are saved in permanent storage.

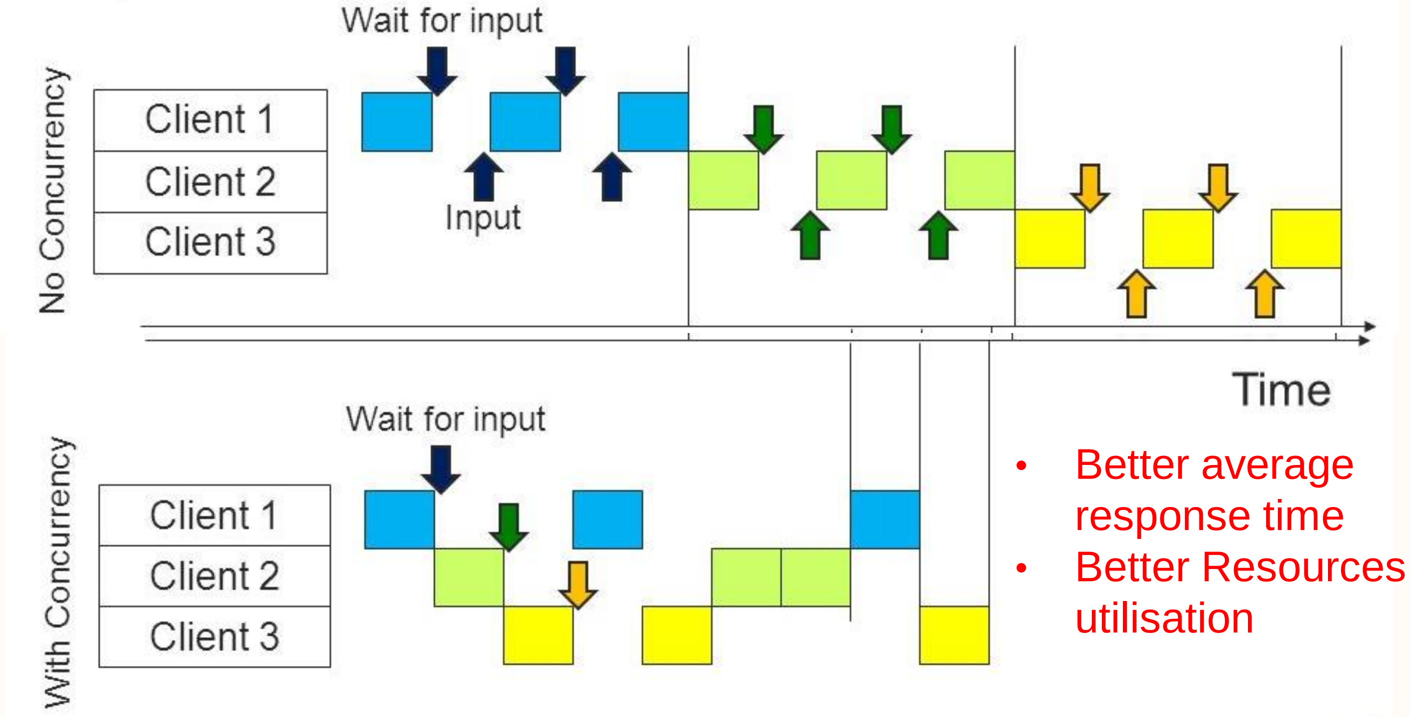
UNIVERSITY OF SUSSEX

Concurrency Control

- To ensure consistency and isolation of a transaction, transactions should be conducted serially one after another.
- However, there are disadvantages to this approach such as reduced resource utilisation and overall inefficiency.
- Interleaving (concurrency) of operations should produce the same effect as a non-interleaved execution.
- The challenge is how do you increase the level of concurrency (transactions/sec) while still not violating the ACID properties of transactions

Concurrency Control

- Concurrency is about dealing with a lot of things at once
- It is when the execution of multiple tasks is interleaved, instead of each task being executed sequentially one after another.



- Better average response time
- Better Resources utilisation

The appearance that multiple actions are occurring at the same time.

Problems of Concurrent Transactions

- Lost update problem
- Inconsistent Retrievals problem

The lost update problem

Init:

Bank account A: £100

Bank account B: £200

Bank account C: £300

A —T, amount X→ B ←U, amount Y— C

- Transaction T: transfers amount X from account A to account B
- Transaction U: transfers amount Y from account C to account B
- B's final balance should be increased by 10% twice to £242

Transaction T:

```
balance = b.getbalance()       ;£200


b.setbalance(balance*1.1)      ;£220
a.withdraw(balance/10)         ;£80

COMMIT
```

Transaction U:

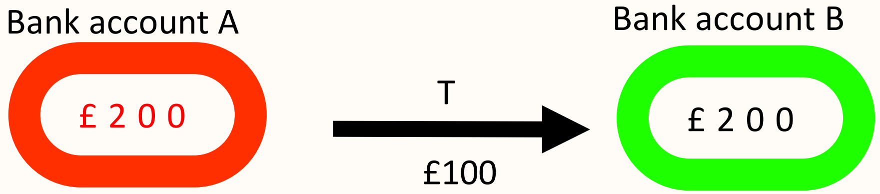
```

balance = b.getbalance()       ;£200
b.setbalance(balance*1.1)      ;£220


c.withdraw(balance/10)         ;£280
COMMIT
```

Time

£80 £240? £280

The inconsistent retrievals problem

Init:



- Transaction T: transfers £100 from account A to account B
- Transaction U: invokes the `branchTotal()` method

```
Transaction T:
a.withdraw(100)              ;£100


b.setbalance(100)            ;£300
```

Time

```
Transaction U:

total = a.getbalance()          ;£100
total = total + b.getbalance()  ;£300
total = total + c.getbalance()
```

UNIVERSITY OF SUSSEX

Serial Equivalence

- If each of several transactions is known to have the correct effect when it is done on its own
  - We can infer that if these transactions are done one at a time in some order the combined effect will also be correct
- An interleaving of the operations of transactions in which the combined effects is the same as if the transactions had been performed one at a time in some order is a <u>serial equivalent interleaving</u>
- The use of serial equivalence as a criterion for correct concurrent execution <u>prevents the occurrence</u> of lost update and inconsistent retrievals

UNIVERSITY OF SUSSEX

Serial Equivalence

Init:

Bank account A: £100 — T, amount X → Bank account B: £200 ← U, amount Y — Bank account C: £300

- Transaction T: transfers amount X from account A to account B
- Transaction U: transfers amount Y from account C to account B
- B’s final balance should be increased by 10% twice to £242

```
Transaction T:
balance = b.getbalance()       ;£200
b.setbalance(balance*1.1)      ;£220


a.withdraw(balance/10)         ;£80
COMMIT
```

```
Transaction U:


balance = b.getbalance()       ;£220
b.setbalance(balance*1.1)      ;£242


c.withdraw(balance/10)         ;£278
COMMIT
```

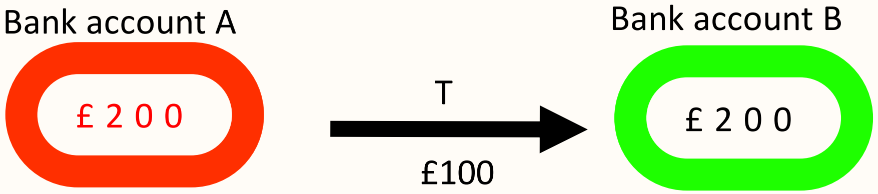
Time

£80 £242 £280

UNIVERSITY OF SUSSEX

Serial Equivalence

Init:



- Transaction T: transfers £100 from account A to account B
- Transaction U: invokes the `branchTotal()` method

```
Transaction T:
a.withdraw(100)                ;£100
b.setbalance(100)              ;£300
```

```
Transaction U:


total = a.getbalance()             ;£100
total = total + b.getbalance()  ;£400
total = total + c.getbalance()
```

Time

Conflicting Operations

- Operations whose combined effect depends on the order in which they are executed
- Operations are conflict if
  - they belong to different transactions
  - they access the same data item
  - at least one is a write operation

	Transactions	
	T	U
Operations	Read	Read
	Read	Write
	Write	Write

No Conflict

Conflicting Operations

UNIVERSITY OF SUSSEX

Conflicting Operations

T	U
T1: R(A)	
	U1: R(A)

Swapped

T	U
	U1: R(A)
T1: R(A)	

T1 and U1 $\Rightarrow$ non conflict pair

T	U
T1: R(A)	
	U1: W(A)

Swapped

T	U
T1: W(A)	
	U1: R(A)

T1 and U1 $\Rightarrow$ conflict pair

UNIVERSITY OF SUSSEX

Conflicting Operations

- For two transactions to be serially equivalent
  - it is necessary and sufficient that all pairs of conflicting operations of the two transactions <u>to be executed in the same order</u> at all of the objects they both access

T	U
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)

✓ Serially equivalent

T	U
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	

✓ Serially equivalent

UNIVERSITY OF SUSSEX

Serial Equivalence

T	U
› R(A)	
› W(A)	
	› R(B)
	› W(B)
› W(B)	
	› R(A)

× Not serially equivalent

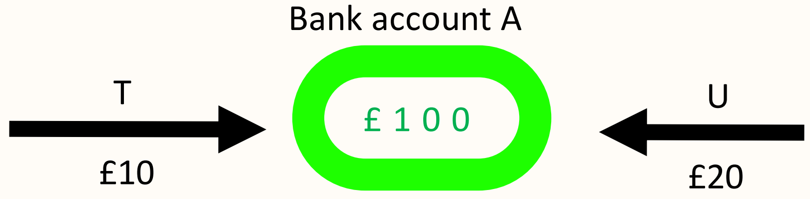
- Each transaction access to objects A and B is serialized with respect to one another
  - T makes all its accesses to A before U
  - U makes all its accesses to B before T
- It is not serially equivalent
  - the pairs of conflicting operations are not done in the same order at both objects
- Serially equivalent requires
  - T accesses A before U and T accesses B before U
  - U accesses A before T and U accesses B before T

Recoverability from aborts

- Dirty Reads
- Premature writes

The dirty read problem

Init:



- Transaction T: adds £10 to account A (this will be rolled back)
- Transaction U: adds £20 to account A

```
Transaction T:
balance = a.getBalance()   ;£100
a.setBalance(balance+10)   ;£110
        (Not committed yet)
    For some reason we want to

              Abort
```

```
Transaction U:


balance = a.getBalance(); £110
a.setBalance(balance+20); £130


              Commit
```

Time

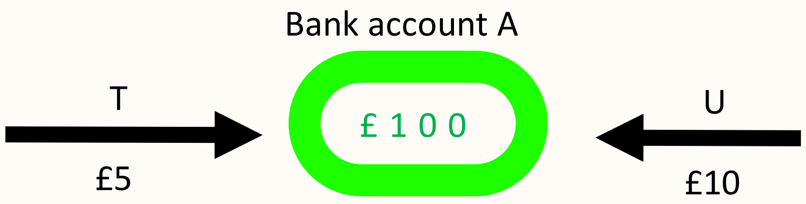
?

Should be £120 as T was aborted

OF SUSSEX

The premature writes

- The Before Image (BI) file stores information about every transaction which changes the database.
- If the transaction is aborted, this information will be read back from the BI file to restore the original values into the database if necessary



UNIVERSITY OF SUSSEX

The premature writes

BI: 100

```
Transaction T:

a.getbalance()              ;£100
a.setbalance(balance+5)     ;£105



         Abort
  Balance (from BI) = 100
     (should be 110?)
```

Time

BI: 100

```
Transaction U:
a.getbalance()              ;£100


a.setbalance(balance+10)    ;£110
            Commit
```

- T's Before Image is £100 - we get the wrong balance of £100
- Write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted

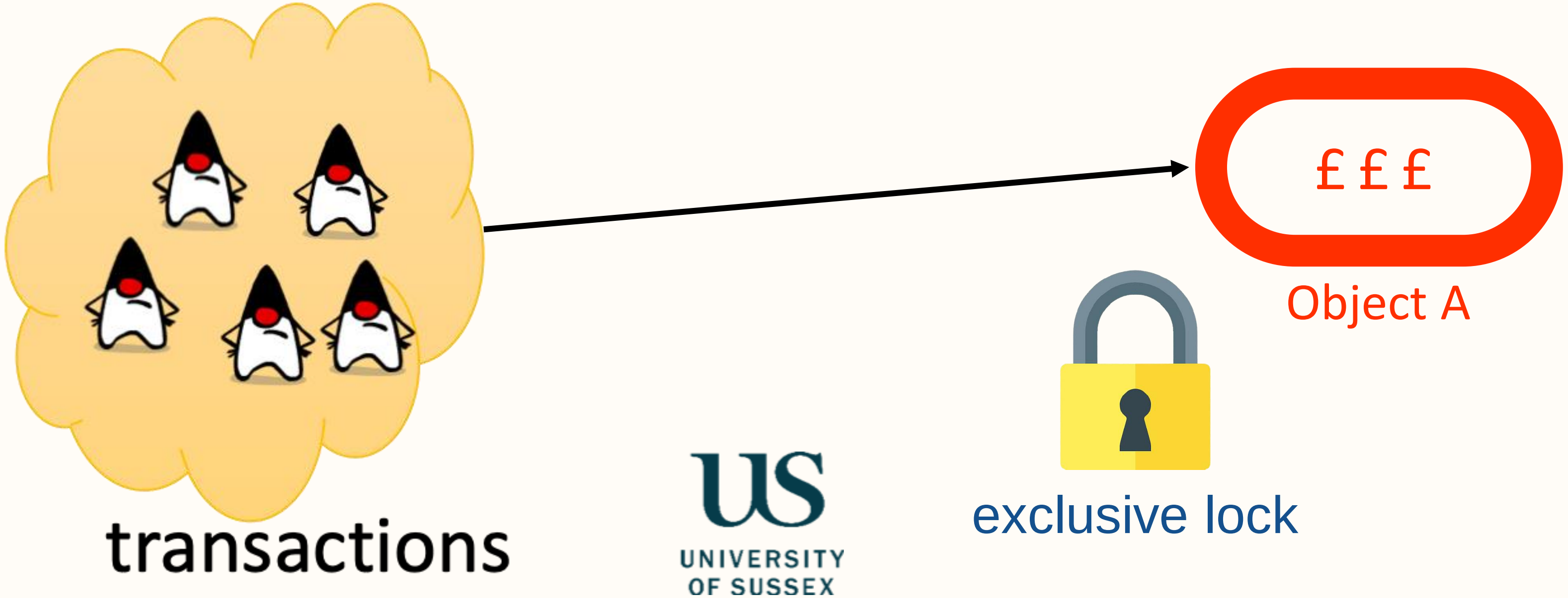
Concurrency control algorithms

- Locking
  - A transaction is granted the exclusive access by setting locks
  - Most used practice for concurrency control
- Optimistic concurrency control
  - No control while executing operations
  - Synchronisation takes place at the end of a transaction
  - If conflicts have occurred, some transactions are forced to abort
- Timestamp ordering
  - Operations are ordered by using timestamps before they are carried out
  - No conflicts can happen

UNIVERSITY OF SUSSEX

Lock-based protocols

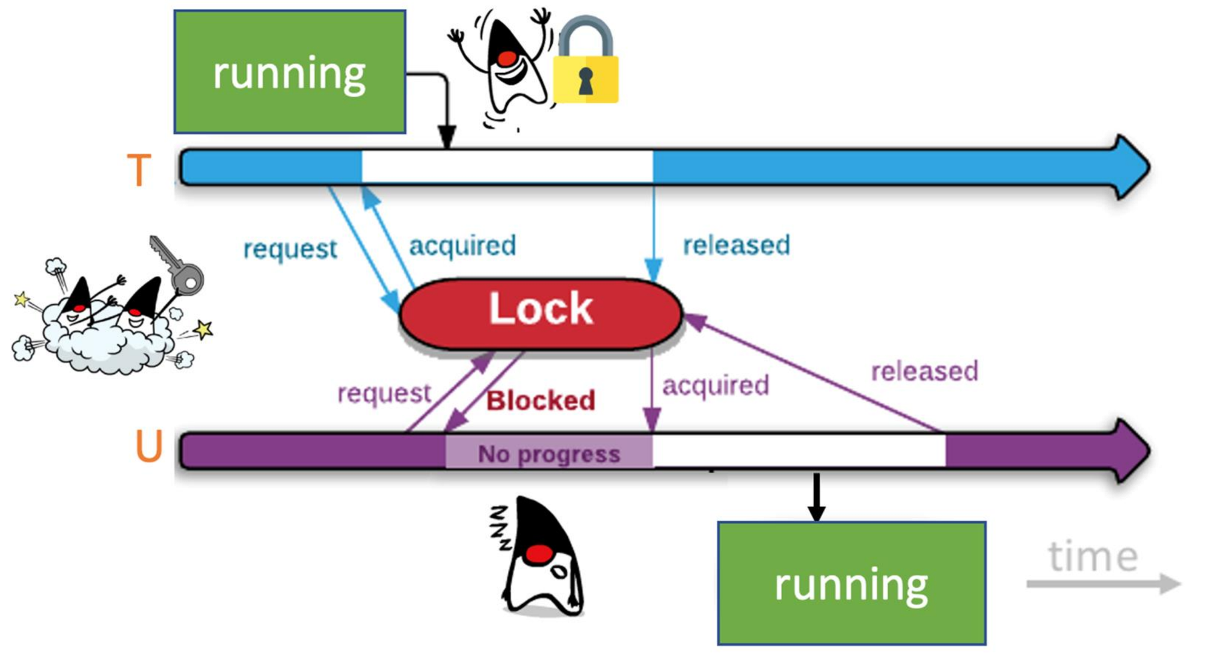
- Main goal of concurrency control is to ensure the isolation property for concurrently running transactions
- The most common way to achieve serial equivalence is using locks



UNIVERSITY OF SUSSEX

Lock-based protocols

- Each data item is locked by at most one transaction
- Before accessing an object, the transaction must acquire a lock on that object
- While it is locked, no other transaction has access to it



UNIVERSITY OF SUSSEX

Lock-based protocols

When T is about to use b, it gets a lock on it

When U is about to use b, it is still locked for T and U waits

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
BEGIN_TRANSACTION			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*BEGIN_TRANSACTION*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s
END_TRANSACTION		•••	
	unlock *A,B*		lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		END_TRANSACTION	unlock *B,C*

U can now continue

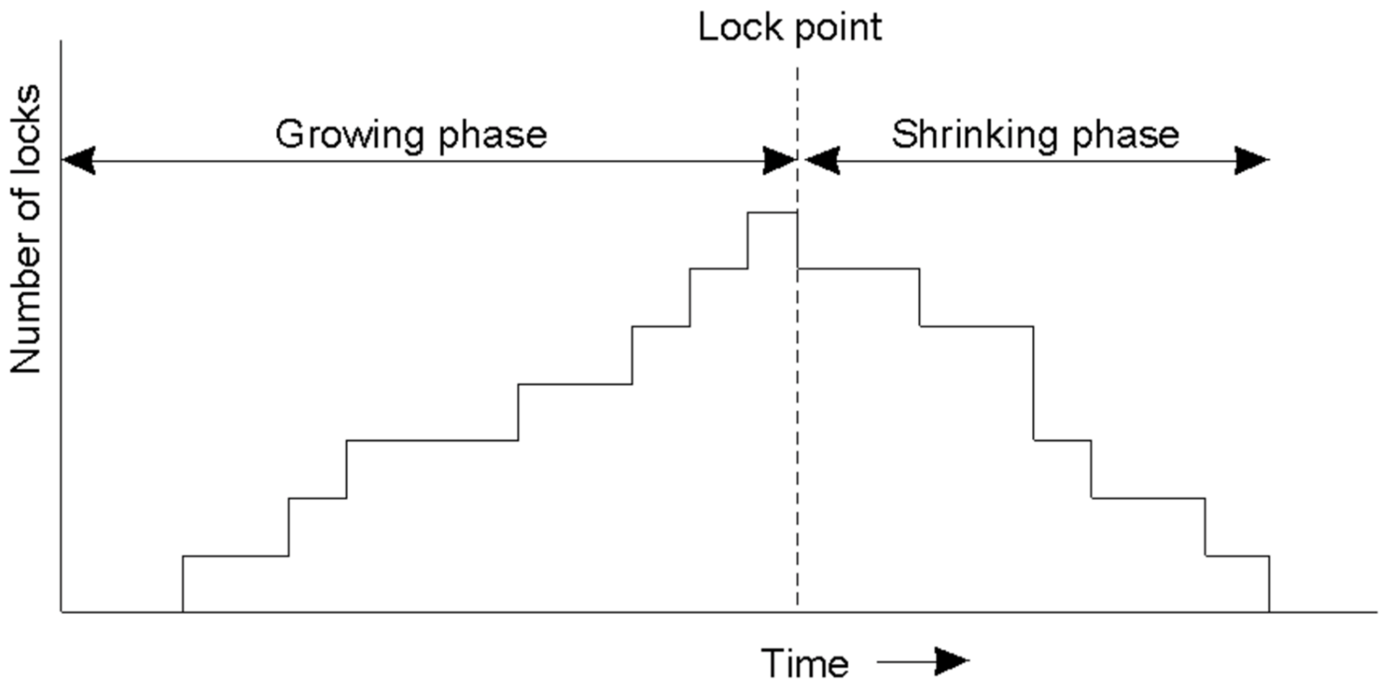
When T commits, it unlocks B

The use of the lock on B effectively serialises access to it

UNIVERSITY OF SUSSEX

Two-phase locking

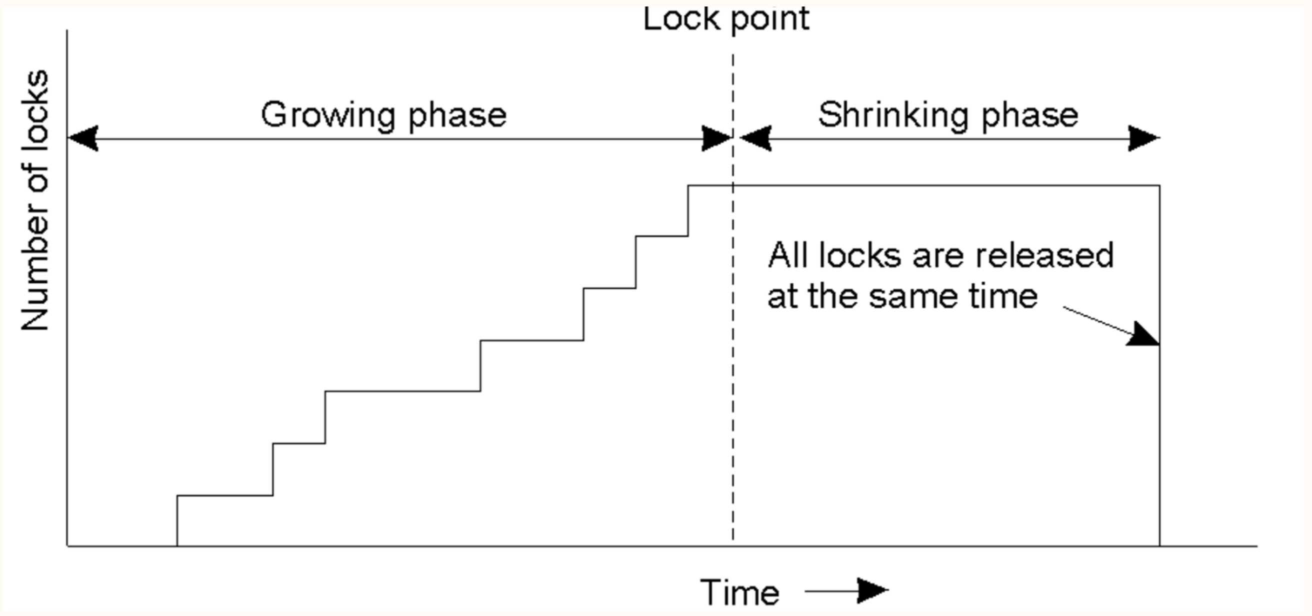
- Each transaction has two phases:
  - Growing phase: a transaction may request new locks, but may not release any locks
  - Shrinking phase: a transaction may release locks, but may not request any locks



Commit

Strict Two-Phase Locking

- In many systems, locks are released only if the transaction commits or aborts (delaying read and write operations)
- A transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted
- Remember dirty read and premature writes!

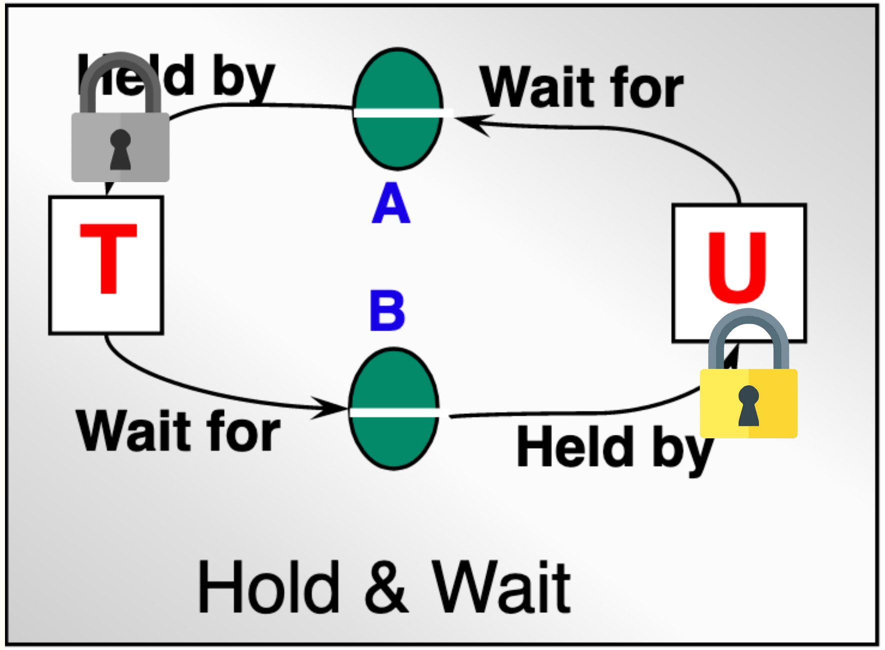


UNIVERSITY OF SUSSEX

Deadlocks

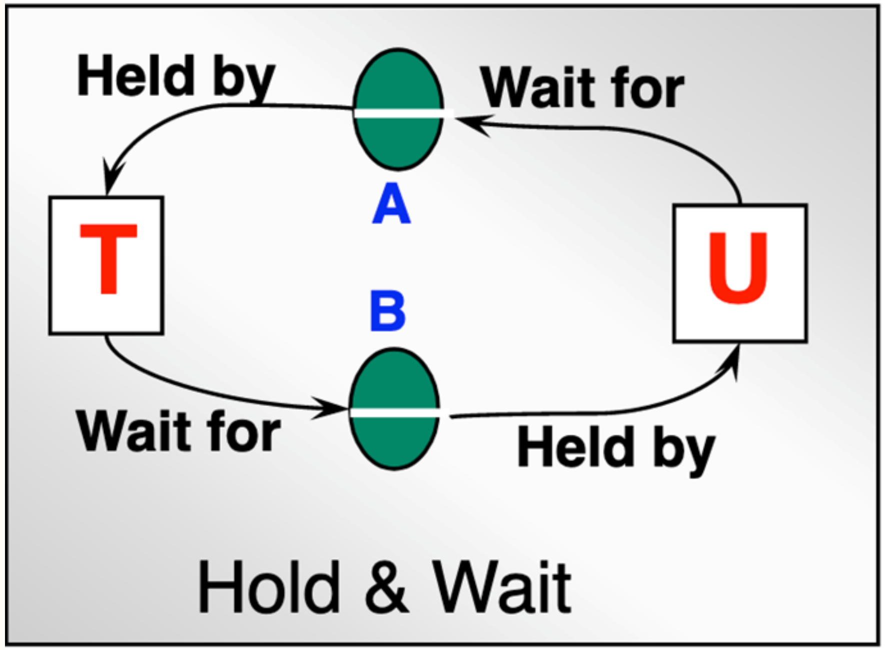
- A state in which each member of a group of transactions is waiting for some other member to release a lock
- <u>wait-for graph</u> used to represent the waiting relationships between current transactions

Transaction *T*		Transaction *U*	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	



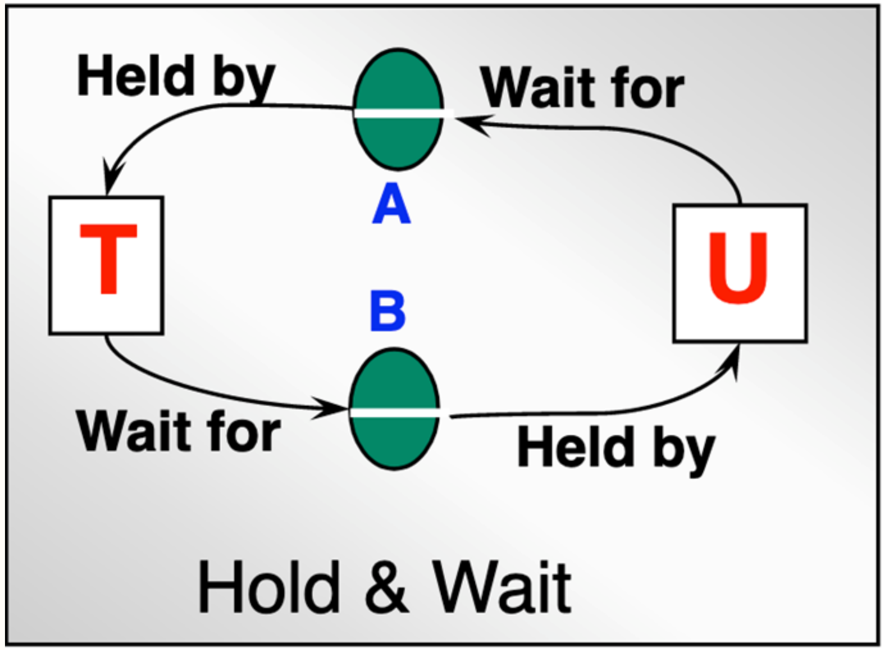
Deadlocks

- Deadlock prevention
  - lock all of the objects used by a transaction when it starts (not good – bad concurrency)
  - Get locks in a predefined order (prematurely acquire lock – bad)
  - impossible to predict which objects will be used



Deadlocks

- Deadlock detection
  - detected by finding cycles in the wait-for graph
  - presence of cycles (deadlocks) is checked each time an edge is added
  - When a deadlock is detected, one of the transactions in the cycle must be <u>aborted</u>
  - Which one? age of the transaction and number of CPU cycles



UNIVERSITY OF SUSSEX

Deadlocks

- Timeouts are commonly used
- Each lock is given a <u>limited period </u>in which it is invulnerable
- A vulnerable lock remains locked if no other transaction is competing for the object
  - opposite case - a vulnerable lock is broken (object is unlocked) – Transaction aborts
- In overloaded systems locks become vulnerable without deadlocks!
- As always, difficult to set the right value for timeouts

UNIVERSITY OF SUSSEX

Disadvantages of locking

- Lock maintenance has overhead
  - Even for read-only transactions
  - Clashing is rare for some applications
- Deadlocks
- Locks cannot be released until the end of the transaction
- Why not be ‘optimistic’?
  - The likelihood of two transactions accessing the same object can be low

UNIVERSITY OF SUSSEX

Optimistic Concurrency Control

- Working phase
  - Each transaction has a tentative version of each object (i.e., a copy of the most recently committed version of the object)
  - With tentative versions, a transaction can abort (with no effect on the objects)
  - Read operations are performed immediately (from the tentative version or the source) from committed versions so no dirty reads can occur
  - Write operations record the new values of the objects as tentative values (i.e., invisible to other transactions)
  - Different versions of tentative objects may coexist

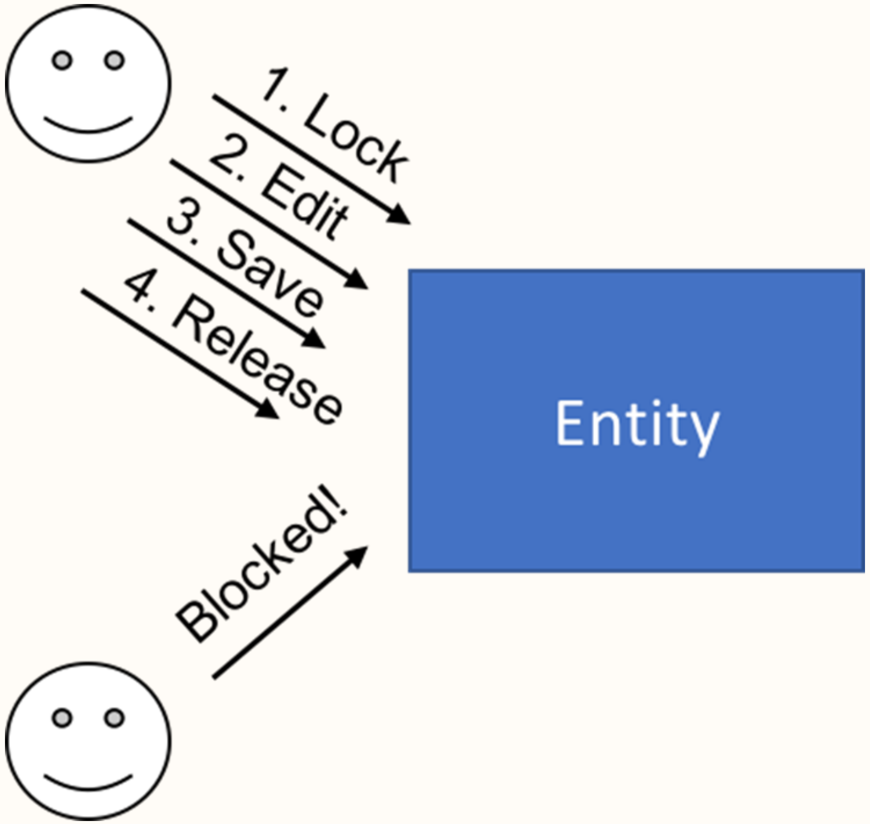
UNIVERSITY OF SUSSEX

Optimistic Concurrency Control

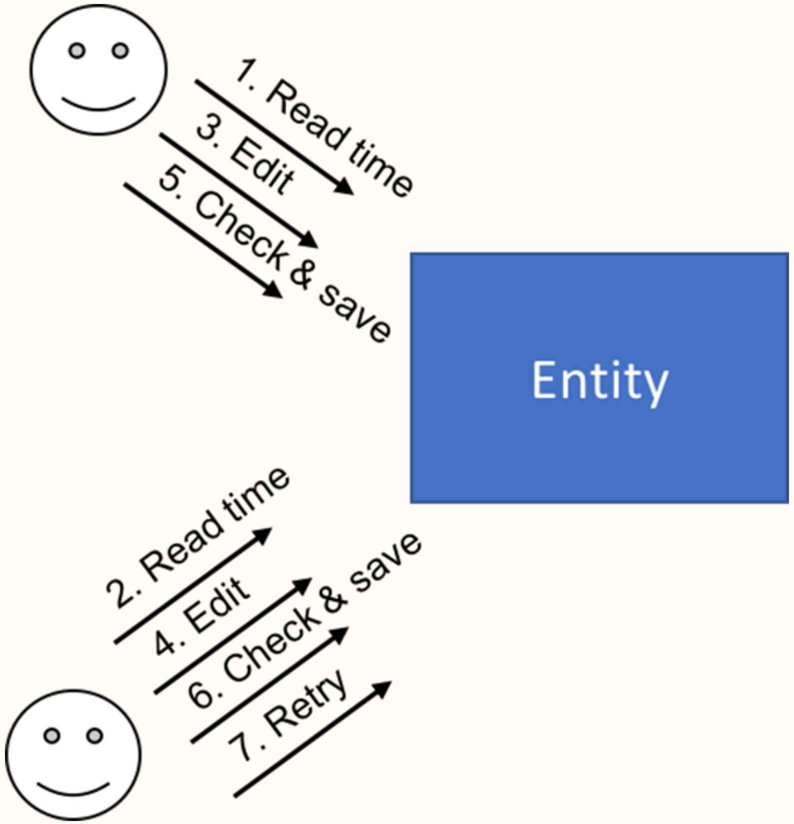
- Validation phase
  - Just before committing, the transaction is validated
  - Any operations on objects that conflict with other transactions?
    - No: transaction commits
    - Yes: conflict resolution
- Update phase
  - All of the changes recorded in its tentative versions are made permanent
  - Read-only transactions can commit immediately after passing validation
  - Write transactions commit once the tentative versions of the objects have been recorded in permanent storage

Optimistic Concurrency Control

- Locking



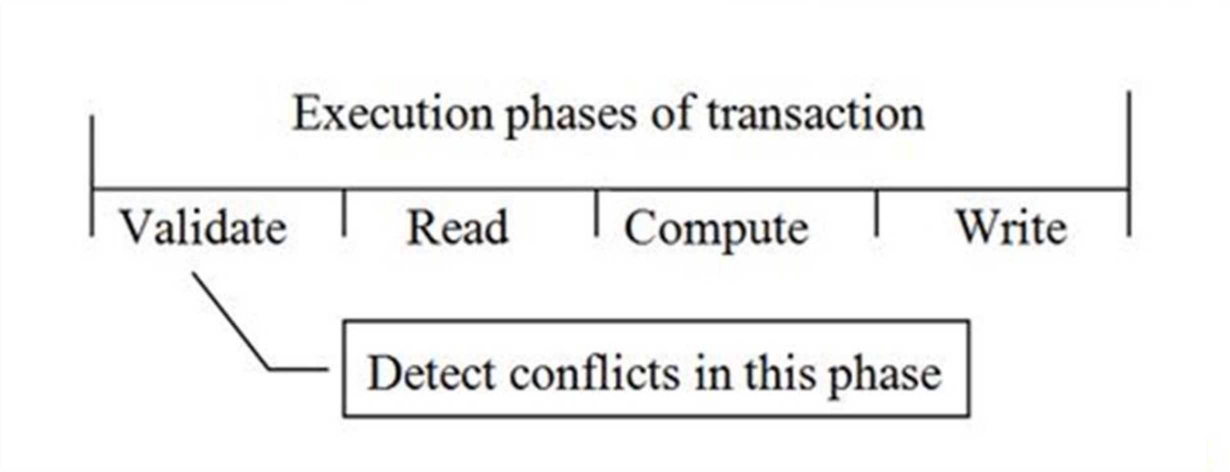
- Optimistic Locking



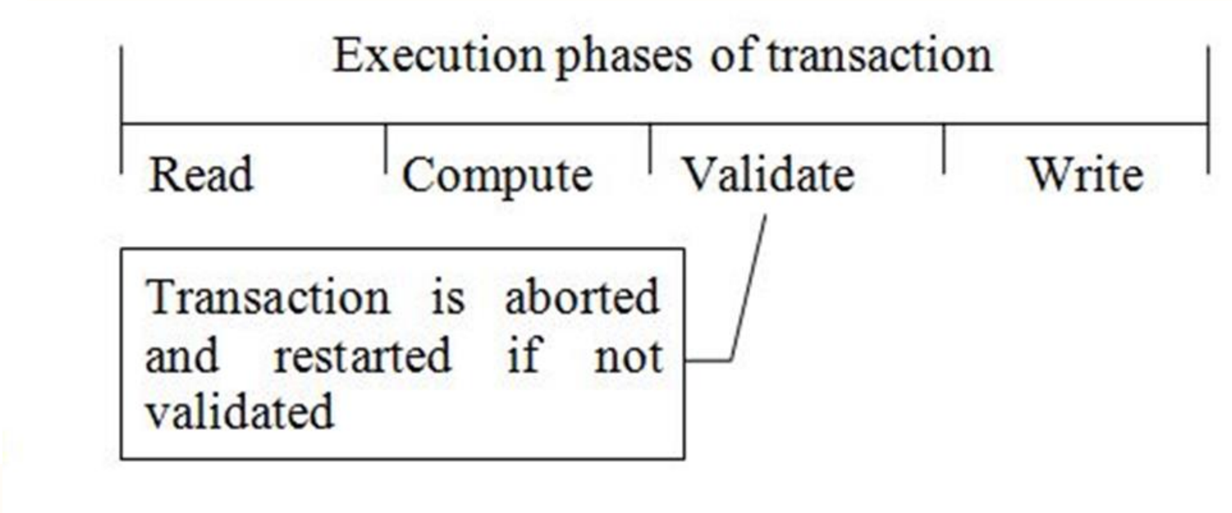
UNIVERSITY OF SUSSEX

Optimistic Concurrency Control

- Pessimistic algorithms assume conflict happens quite often



- Optimistic algorithms delay the validation phase until write phase



UNIVERSITY OF SUSSEX

Starvation

- Only problem is potential starvation, i.e., transactions being repeatedly aborted
- Occurrences of starvation should be rare
- Repeatedly aborted transactions can be just given exclusive access

Transactions in Django

- In the SQL standards, each SQL query starts a transaction, unless one is already active
  - Such transactions must then be explicitly committed or rolled back
  - But this isn't always convenient for application developers
- Therefore, most databases provide an autocommit mode
  - When autocommit is turned on and no transaction is active, each SQL query gets wrapped in its own transaction
- The Python Database API Specification v2.0, requires autocommit to be initially turned off
  - Django overrides this default and turns autocommit on.

Transactions in Django

- Django's default behavior is to run in autocommit mode
  - Each query is immediately committed to the database, unless a transaction is active
- Django uses transactions or savepoints automatically to guarantee the integrity of ORM operations that require multiple queries
  - especially `delete()` and `update()` queries
- It is possible to disable Django's transaction management for a given database by setting AUTOCOMMIT to False in its configuration
  - This requires the developer to commit explicitly every transaction
    - even those started by Django or by third-party libraries

UNIVERSITY OF SUSSEX

Tying transactions to HTTP requests

- A common way to handle transactions on the web is to wrap each request in a transaction
  - Set ATOMIC_REQUESTS to True in the configuration of each database
- It works as follows. Django
  - starts a transaction before calling a view function
  - commits the transaction if the response is produced without problems
  - rolls back the transaction if the view produces an exception
- It is also possible to perform subtransactions using savepoints in the view code with the `atomic()` context manager
  - either all or none of the changes will be committed at the end of the view

Tying transactions to HTTP requests

- Only the execution of the view is enclosed in the transactions
  - For example, rendering of template responses runs outside of the transaction
- To prevent views from running in a transaction, add the decorator `non_atomic_requests(using=None)`

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    do_stuff()

@transaction.non_atomic_requests(using='other')
def my_other_view(request):
    do_stuff_on_the_other_database()
```

UNIVERSITY OF SUSSEX

Controlling transactions explicitly

- Django provides a single API to control database transactions
- `atomic(using=None, savepoint=True, durable=False)`
  - allows creating a block of code within which the atomicity on the database is guaranteed
    - If the block of code is successfully completed, the changes are committed to the database
    - If there is an exception, the changes are rolled back
  - blocks can be nested
    - when an inner block completes successfully, its effects can still be rolled back if an exception is raised in the outer block at a later point
  - It can be used as a decorator and a context manager

UNIVERSITY OF SUSSEX

Controlling transactions explicitly

- `atomic` is usable both as a *decorator*:

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    do_stuff()
```

This code executes inside a transaction

- `atomic` is usable both as a *context manager*:

```
from django.db import transaction

def viewfunc(request):
    do_stuff()

    with transaction.atomic():
        do_more_stuff()
```

This code executes in autocommit mode (Django's default)

This code executes inside a transaction

UNIVERSITY OF SUSSEX

Controlling transactions explicitly

- Wrapping `atomic` in a try/except block allows for natural handling of integrity errors

```
from django.db import IntegrityError, transaction

@transaction.atomic
def viewfunc(request):
    create_parent()

    try:
        with transaction.atomic():
            generate_relationships()
    except IntegrityError:
        handle_exception()

    add_children()
```

What if `generate_relationships()` causes a database error?

UNIVERSITY OF SUSSEX

Controlling transactions explicitly

- Django's transaction management code works as follows:
  - opens a transaction when entering the outermost atomic block
  - creates a savepoint when entering an inner atomic block
  - releases or rolls back to the savepoint when exiting an inner block
  - commits or rolls back the transaction when exiting the outermost block

Performing actions after commit

- It may be necessary to perform an action related to the current database transaction, but only if the transaction successfully commits.
  - Examples include an email notification, or a cache invalidation.
- Django provides the `on_commit()` function
  - It registers callback functions that should be executed after a transaction is successfully committed
  - `on_commit(`*`func, using=None`*`)`
    - Pass any function (that takes no arguments) to `on_commit()`

```
from django.db import transaction

def do_something():
    pass  # send a mail, invalidate a cache, etc.

transaction.on_commit(do_something)
```

Next Lecture …

- ✓ Introduction
- ✓ HTTP, Caching, and CDNs
- ✓ Views
- ✓ Templates
- ✓ Forms
- ✓ Models
- ✓ Security
- ✓ Transactions
- ➢ **Remote Invocation**
- Web Services
- RESTful Services
- Time
- Elections/Group Communication
- Zookeeper

UNIVERSITY OF SUSSEX