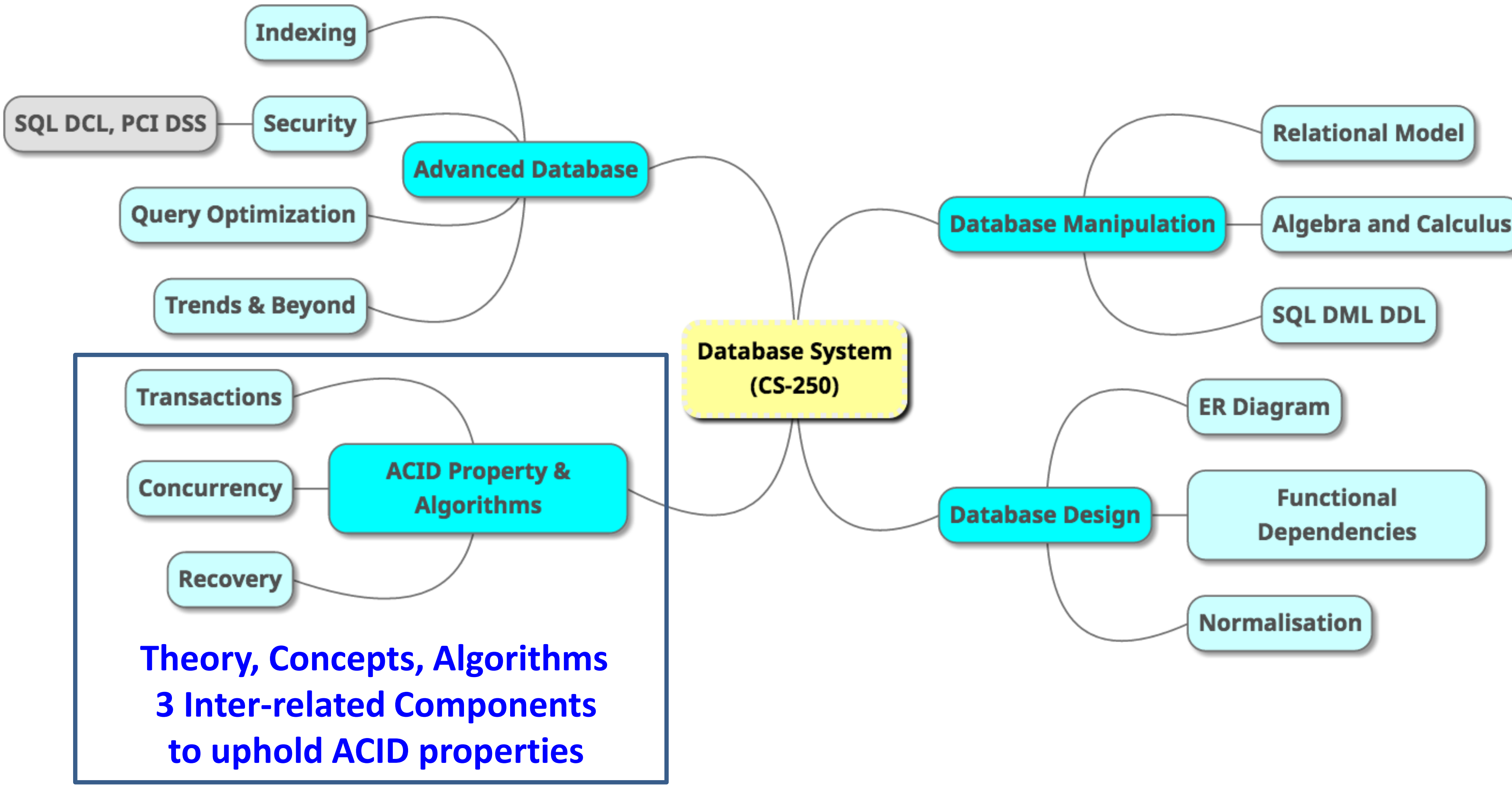
Transactions

Gary KL Tam

Department of Computer Science
Swansea University

Overview


Indexing
SQL DCL, PCI DSS
Security
Advanced Database
Query Optimization
Trends & Beyond
Relational Model
Database Manipulation
Algebra and Calculus
SQL DML DDL
Database System
(CS-250)
Transactions
Concurrency
ACID Property &
Algorithms
Recovery
ER Diagram
Database Design
Functional
Dependencies
Normalisation
Theory, Concepts, Algorithms
3 Inter-related Components
to uphold ACID properties

Motivation

The Telegraph

Home | News | World | Sport | Finance | Comment | Blogs | Culture | Travel | Life | Women | Fa

Companies | Comment | Personal Finance | Economics | Markets | Festival of Business | Yo

Investing | Saving Tips | Savings | Interest Rates | Funds | Gold | Mortgages | Credit Car

HOME » FINANCE » PERSONAL FINANCE » CONSUMER TIPS » BANKING

NatWest crisis could see knock-on effects drag on for weeks

People may suffer problems with NatWest bank accounts for weeks despite the company's claims to have fixed its computer problems, the Financial Ombudsman Service has warned.

19-22/06/2012

Drag on for 3 days!

At the end, take a week to fully recover.

Why take so long?

theguardian

News | Sport | Comment | Culture | Business | Money | Life & style

News › Technology › Software

How NatWest's IT meltdown developed

Guardian's investigations suggest bank's problems began on Tuesday night when it updated key piece of software called CA-7

- Richard Price, a Norwich-based systems developer who has worked on banking systems that linked into NatWest's, explains: "**Banking systems are like a huge game of Jenga [the tower game played with interlaced blocks of wood]**. ***Two unrelated transactions*** might not look related now, but 500,000 transactions from now they might have a huge relation. So everything needs to be processed ***in order***." Thus Tuesday's batch must run before Wednesday's or Thursday's to avoid, for example, penalising someone who has a large sum of money leave their account on Thursday that might put them in debt but which would be covered by money arriving on Wednesday.

Concurrency

Transaction

Recovery

All essential to uphold ACID properties!

Motivation - ACID Properties

- RDBMS supports ACID properties natively.
- That's, with a few configuration, ACID is ***fully*** supported.

- Some NoSQL DB (e.g. MongoDB) is not ***fully*** ACID-compliant.
- You need to understand ***when it is not/what not supported***.
- It is fine to study NoSQL, but study CS-250 properly <u>first</u>.

- Bassam Helal (BSc & MSc 2019)
  - Tell your students: this chapter is very useful.
  - When I studied MongoDB myself, these concepts all come back to me.

- Chuks Ajeh (BSc 2020)
  - My internship uses MongoDB (NoSQL)
  - My understanding wouldn't have been so thorough had it not been for your module!

This lecture

Overview

These cover the fundamental and prepare us for **Concurrency** lecture next week.

- Transaction Lecture
  - ACID
  - DBMS Basic and Hardware
- Concurrency Problem
  - Lost update, uncommitted update, inconsistent analysis
- Serialisability **Theory**
  - Example schedules
  - Conflict operations
  - Conflict serialisable schedules
  - Testing Conflict Serialisability

Note: theory is not easy.

Overview

- A **transaction** is a unit of execution
- Must either be completed (committed) or totally cancelled (aborted).
  - In reality a transaction may include numerous read, write and other operations

- **Concurrent** execution of multiple transactions.
  - Concurrency control.

- Failures types: hardware failures, system crashes, electricity outage...
  - **Recovery**.

ACID properties

Each transaction should achieve

- **Atomicity**: Either committed or aborted, ***no partial execution***

- **Consistency**: ***No constraint violation*** after the transaction finishes.
  - A transaction may be temporarily in an inconsistent state during execution.

- **Isolation**: Concurrent transactions are ***not aware*** of each other. Each thinks that it is the only running transaction.

- **Durability**. If a transaction is committed, its changes to the database are ***permanent***, even if there is a system failure.

Example of fund transfer

Transfer £50 from account A to B:

read(A)

$A = A - 50$

write(A)

read(B)

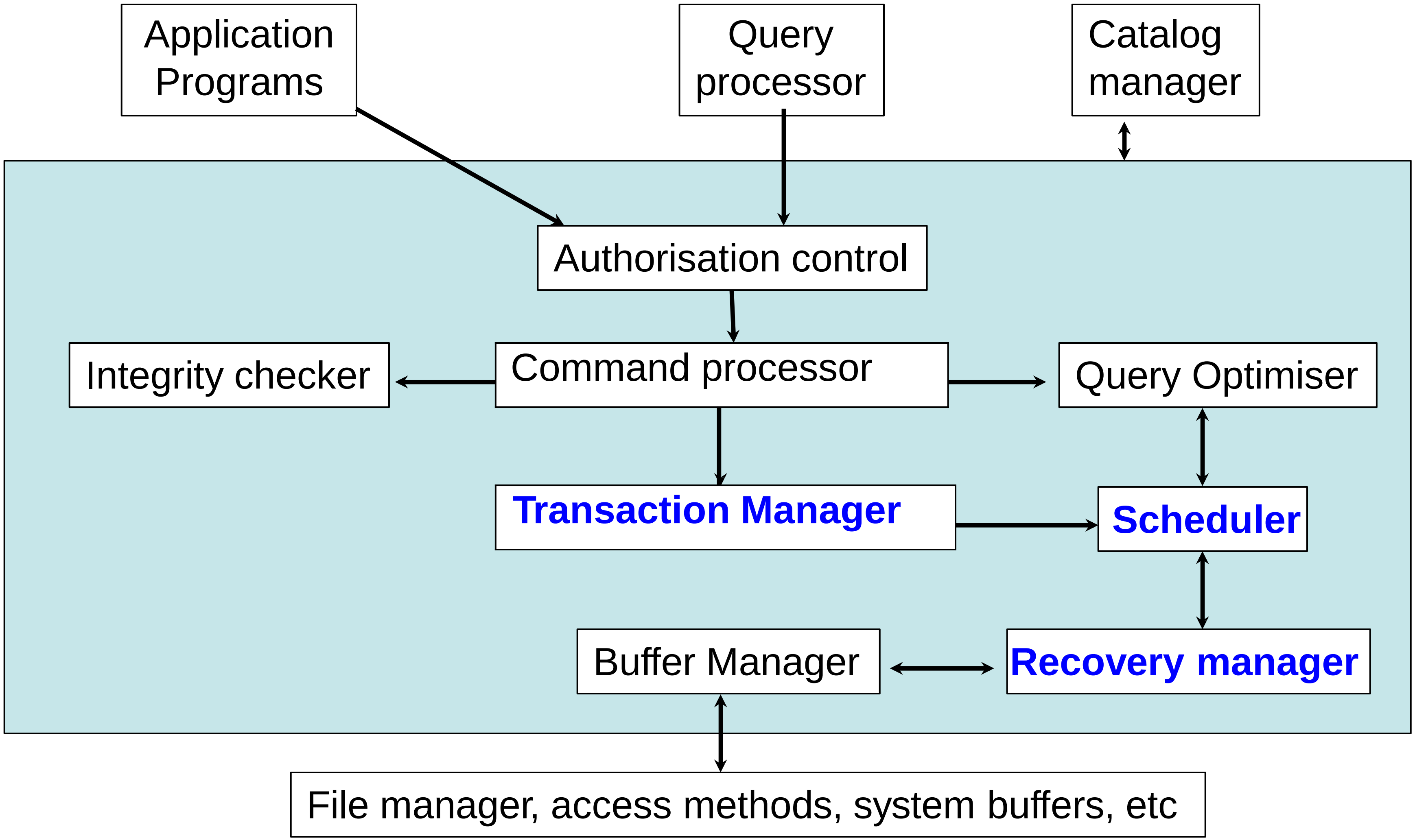
$B = B + 50$

write(B)

One transaction

- Atomicity – Shouldn't take money from A without giving it to B
- Consistency – Money isn't lost or gained overall
- Isolation – Other queries shouldn't see A or B change until completion
- Durability – The money does not return to A, even after a system crash

The Database Manager



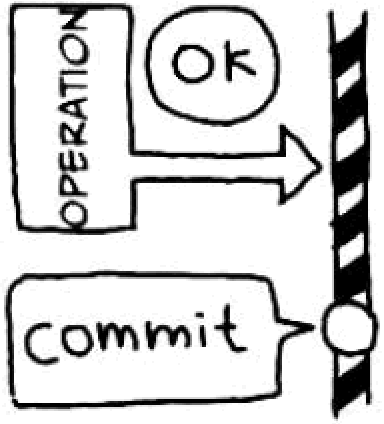
Connolly & Begg

Transaction Manager

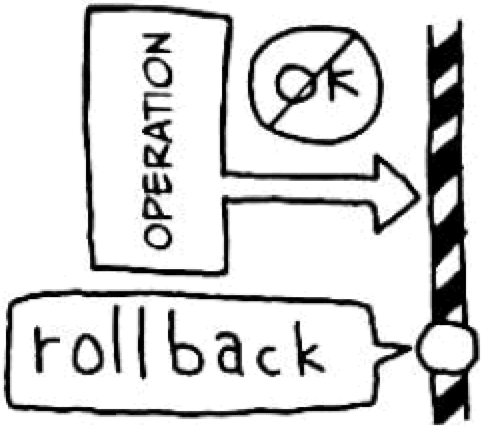
- The **transaction manager** enforces the ACID properties
  - **Schedules** the operations of all transactions
  - Uses COMMIT and ROLLBACK to ensure ***atomicity***
  - Locks are used to ensure ***consistency*** and ***isolation***
  - A log is kept to ensure ***durability***

Atomicity

- COMMIT - signal the ***successful*** end of a transaction
  - Any changes that have been made to the database should be made permanent
  - These changes are now available to other transactions



- ROLLBACK - signal the ***unsuccessful*** end of a transaction
  - Any changes that have been made to the database should be undone
  - It is now as if the transaction never happened, it can now be reattempted if necessary



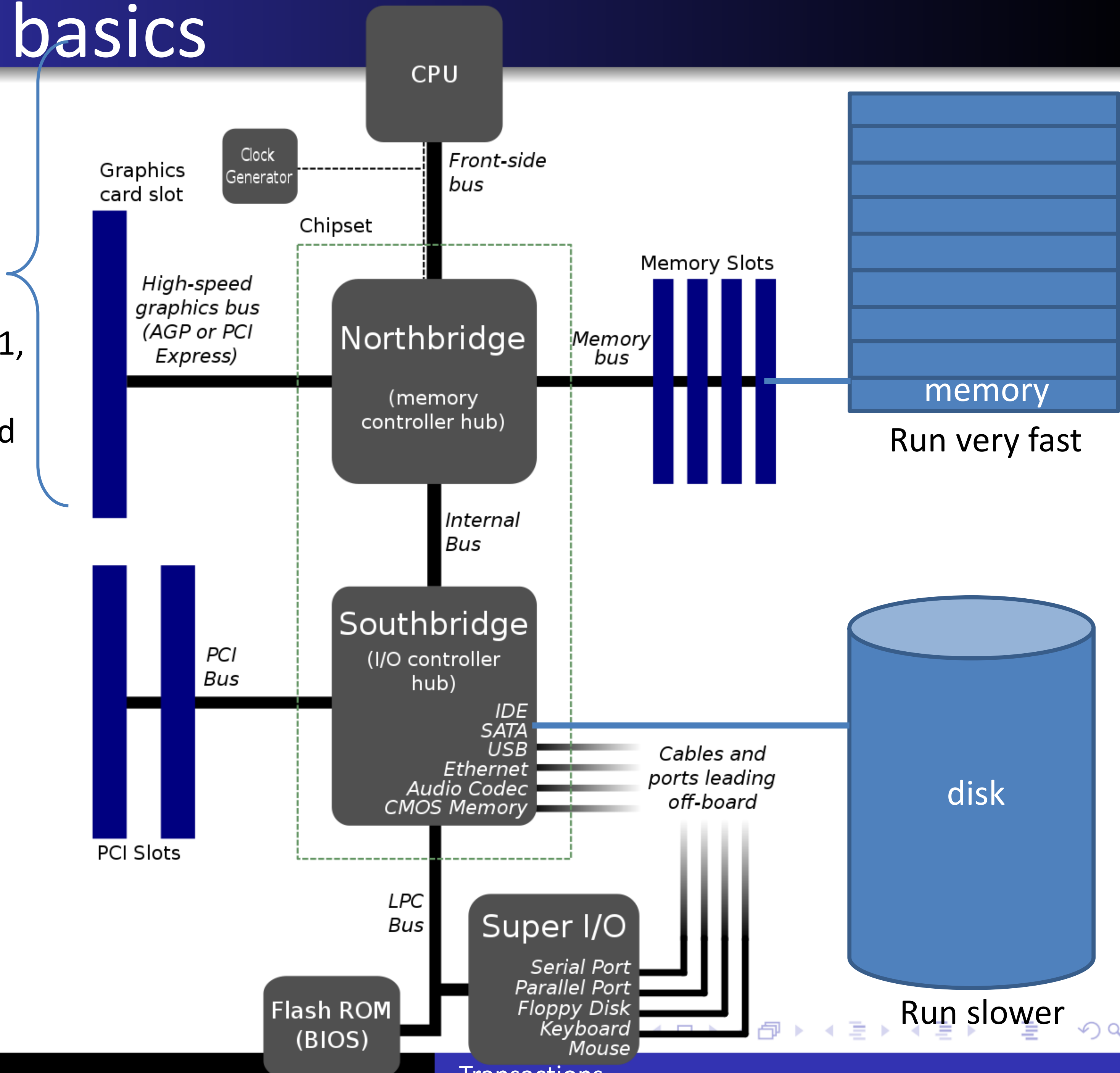
Concurrency

- Large databases are used by many people
  - e.g. **banks**!
  - Many transactions are to be run **simultaneously**
  - Faster than serial execution
  - Still need to preserve ACID properties

- Without concurrency
  - Easy to preserve atomicity and isolation
  - But have a long queue of transactions
  - Long transactions will delay others
    - e.g. do you want to queue at a counter on a busy day?

DBMS basics

For Intel Sandy Bridge and AMD Accelerated Processing Unit processors introduced in 2011, these are merged/integrated

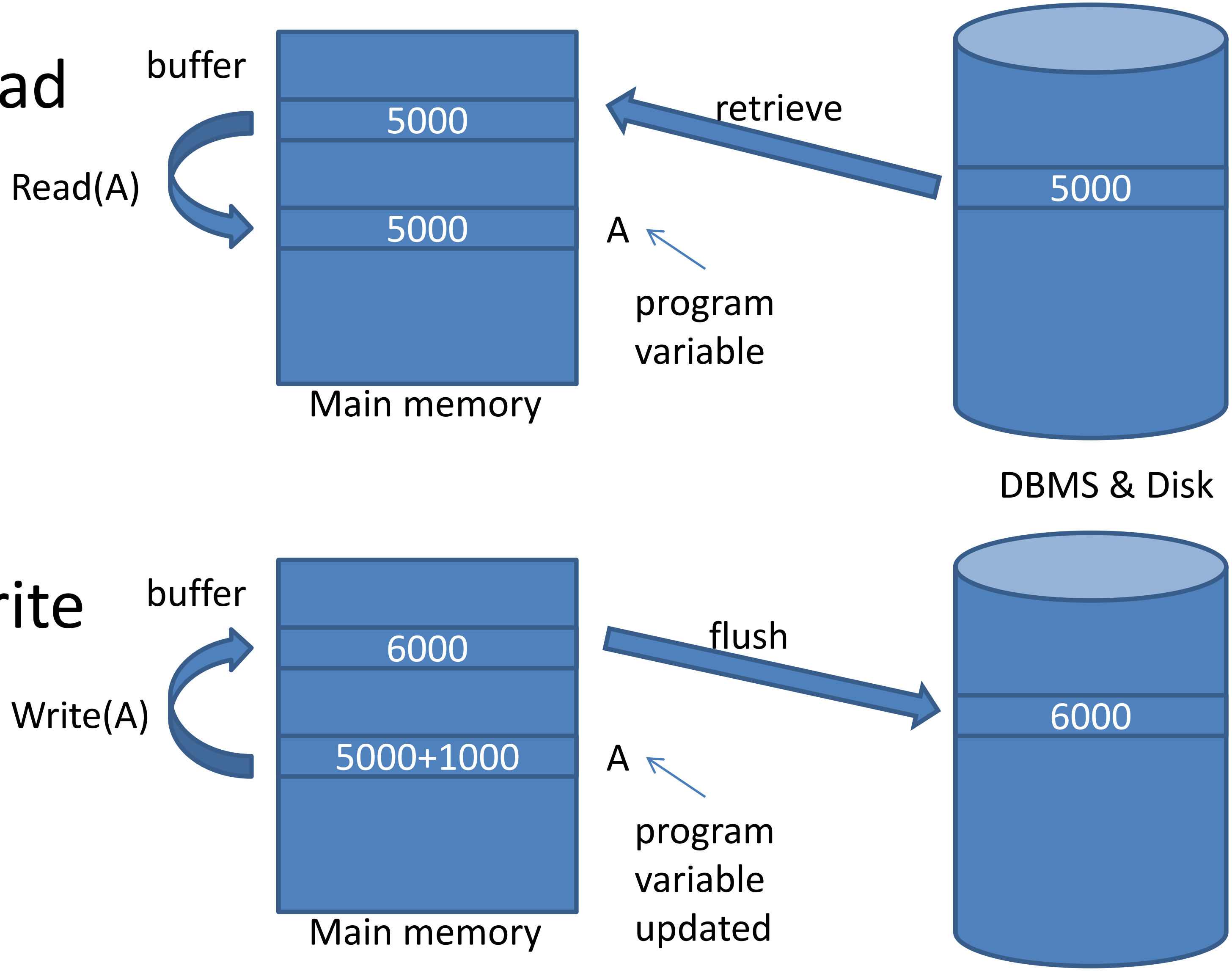


DBMS concerns disk access!

DBMS basics

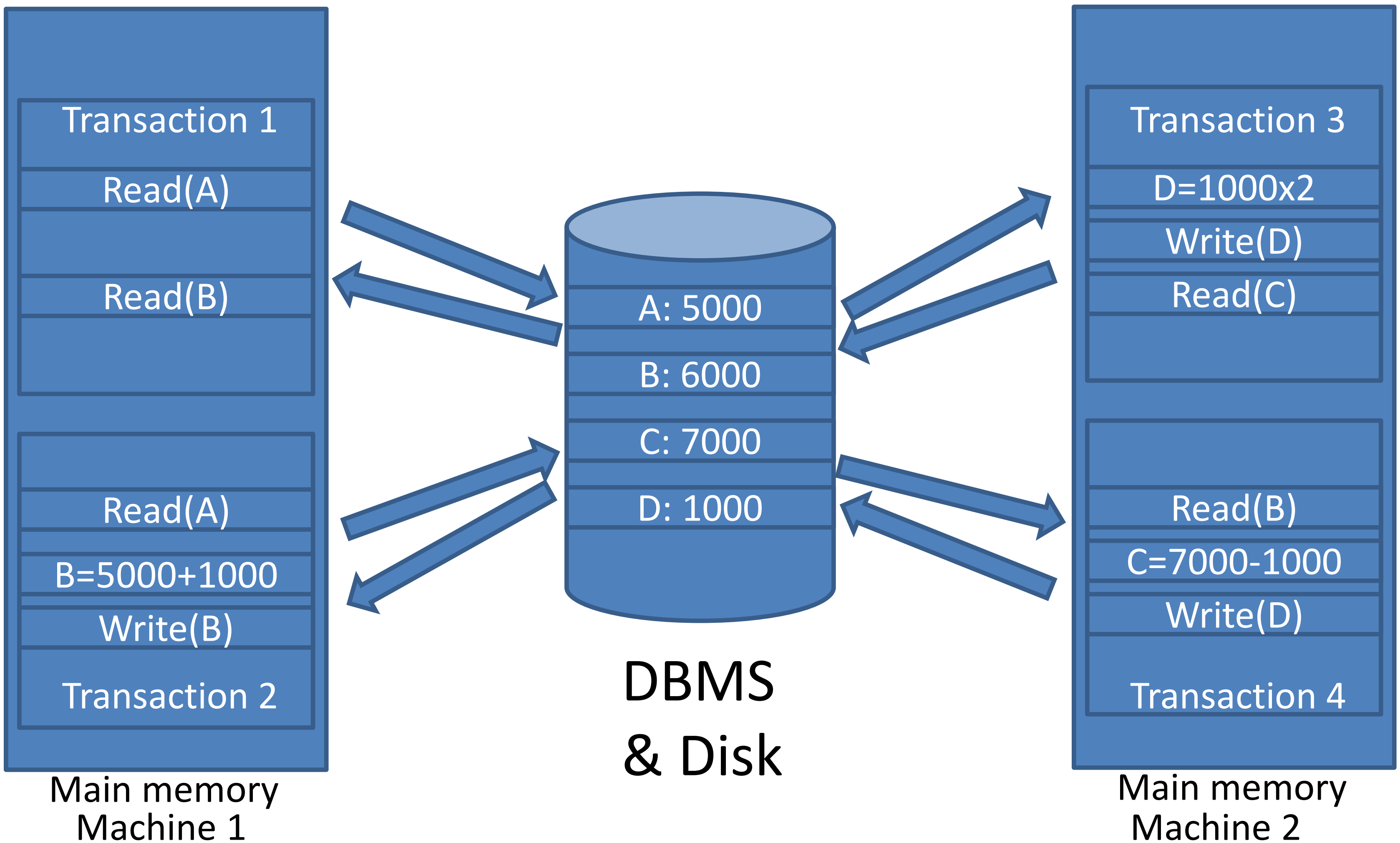
- Read

- Write

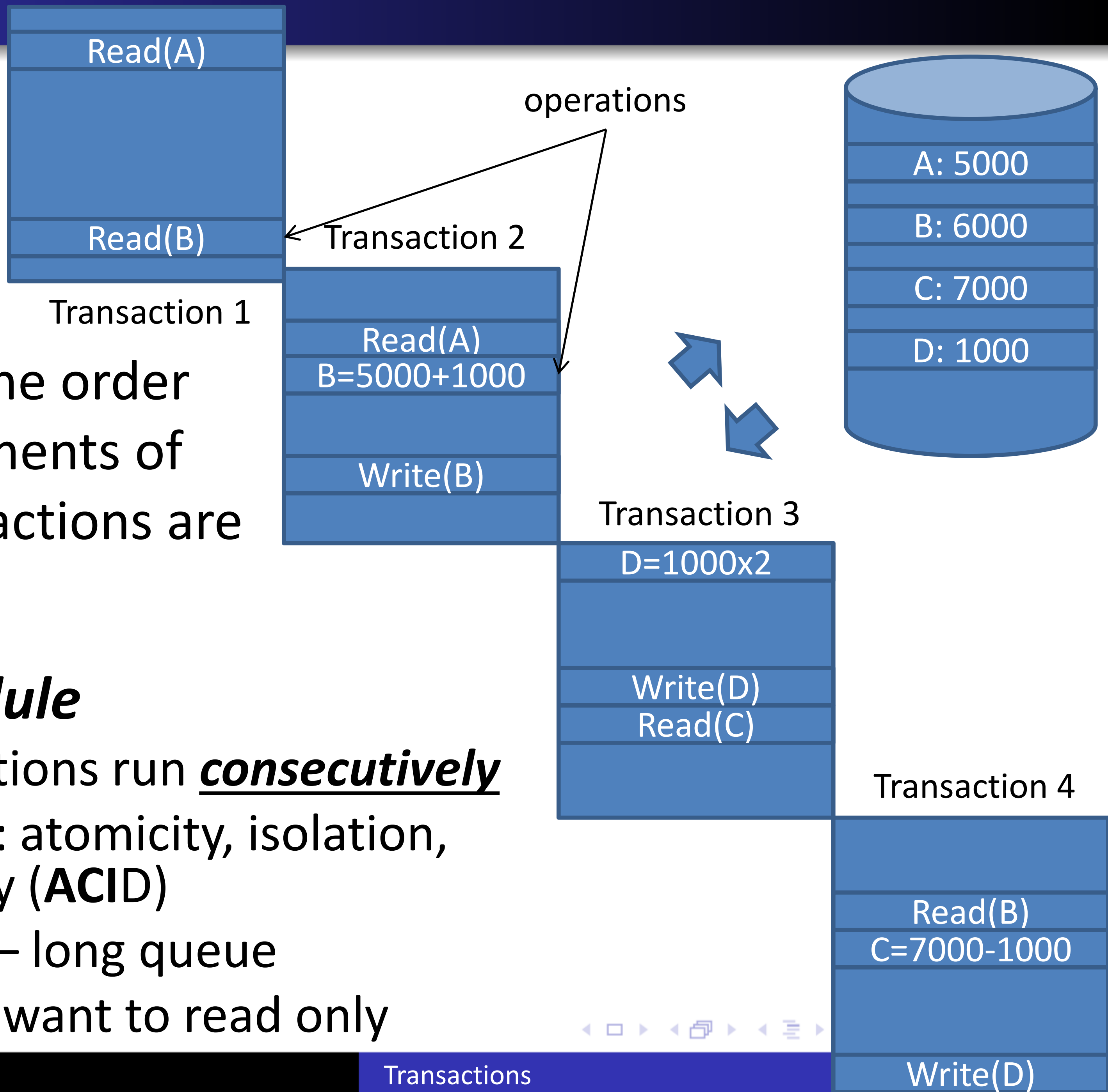


DBMS basics

- Multiple transactions, sharing ONE DBMS



Simplest



A ***schedule*** – the order that the statements of multiple transactions are executed.

- ***Serial schedule***
  - All transactions run ***consecutively***
  - Guarantee: atomicity, isolation, consistency (**ACI**D)
  - But SLOW – long queue
  - Some may want to read only

This lecture

These cover the fundamental and prepare us for **Concurrency** lecture next week.

Overview

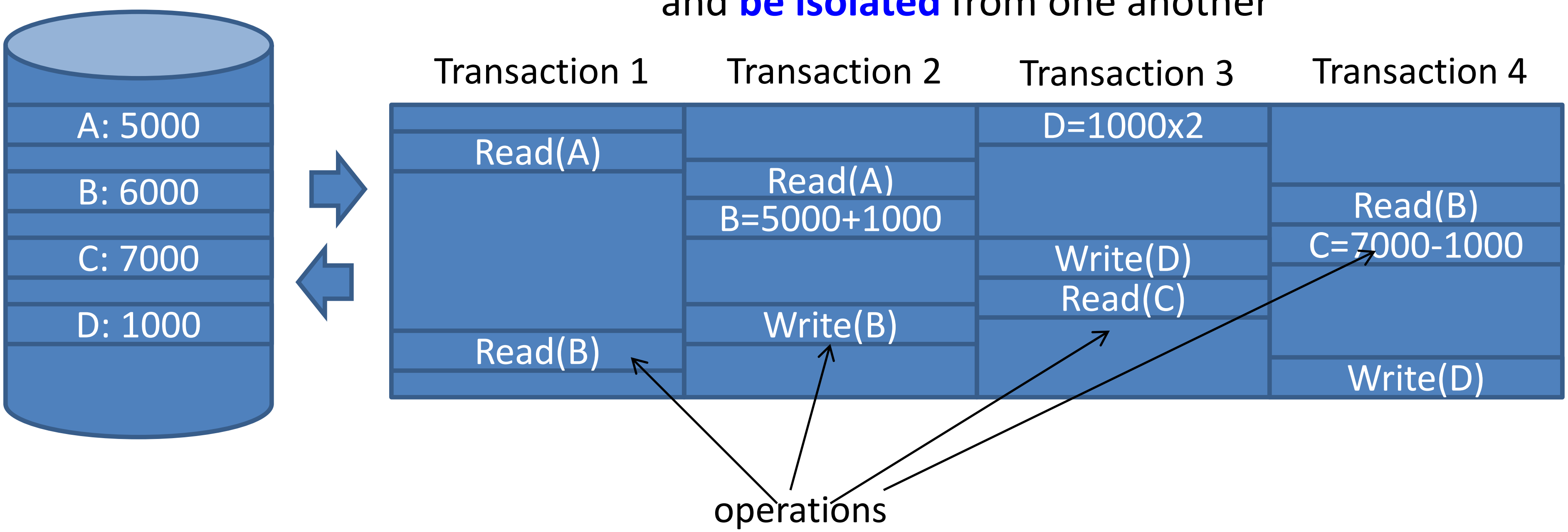
- Transaction Lecture
  - ACID
  - DBMS Basic and Hardware
- Concurrency Problem
  - Lost update, uncommitted update, inconsistent analysis
- Serialisability **Theory**
  - Example schedules
  - Conflict operations
  - Conflict serialisable schedules
  - Testing Conflict Serialisability

Note: theory is not easy.

Concurrency Problem

- Concurrent transactions
  - Quicker, but operations must be interleaved
  - Each gets a share of computing time

They all run **simultaneously,** and **must not aware others' existence** and **be isolated** from one another



Concurrency Problem

- If isolation is broken... (or not taken care of)

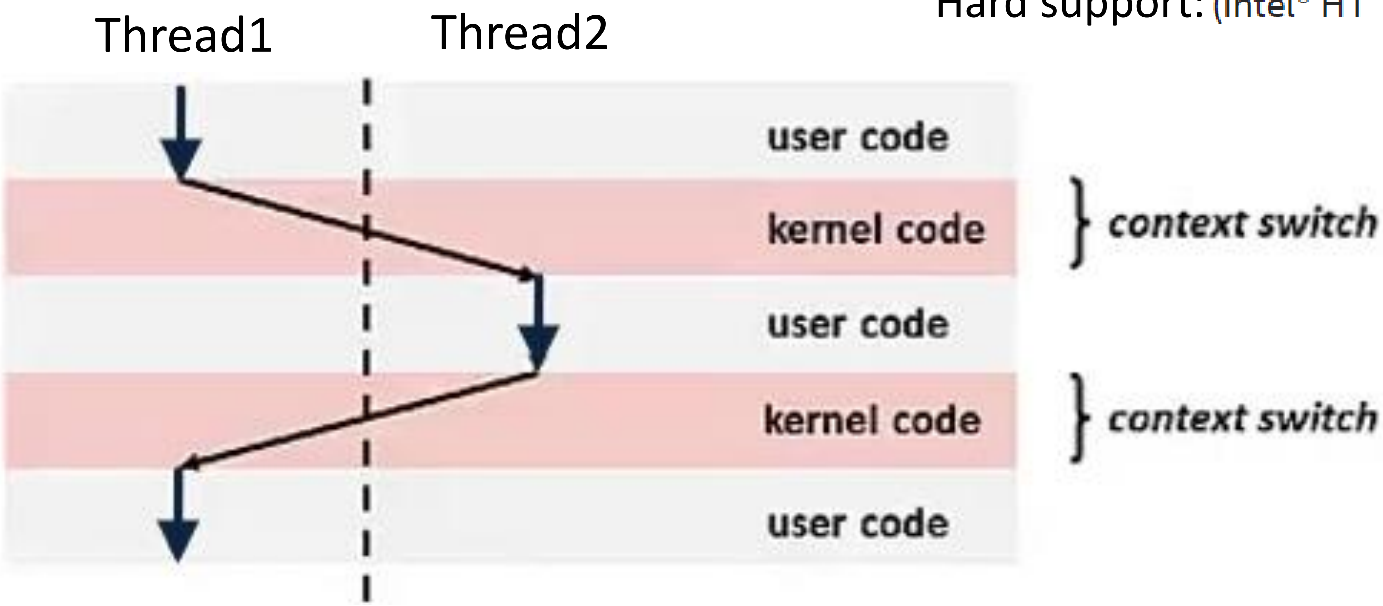
- Three problems of Concurrent transactions:
  - Lost Updates
  - Uncommitted Updates
  - Inconsistent Analysis

- Concurrent transactions
  - Multiple transactions, each run on a thread
  - Multiple threads running on CPU core(s)

Muddiest Point: Background

- Single core CPU, two threads:

Soft support: OS
Hard support: (Intel® HT Technology)



- In CS-250, our concern is data **on disk**.

- For multiple core/thread processing:
  CS-210 – Concurrency (TB2)
- CSCM98 - Operating Systems and Architectures

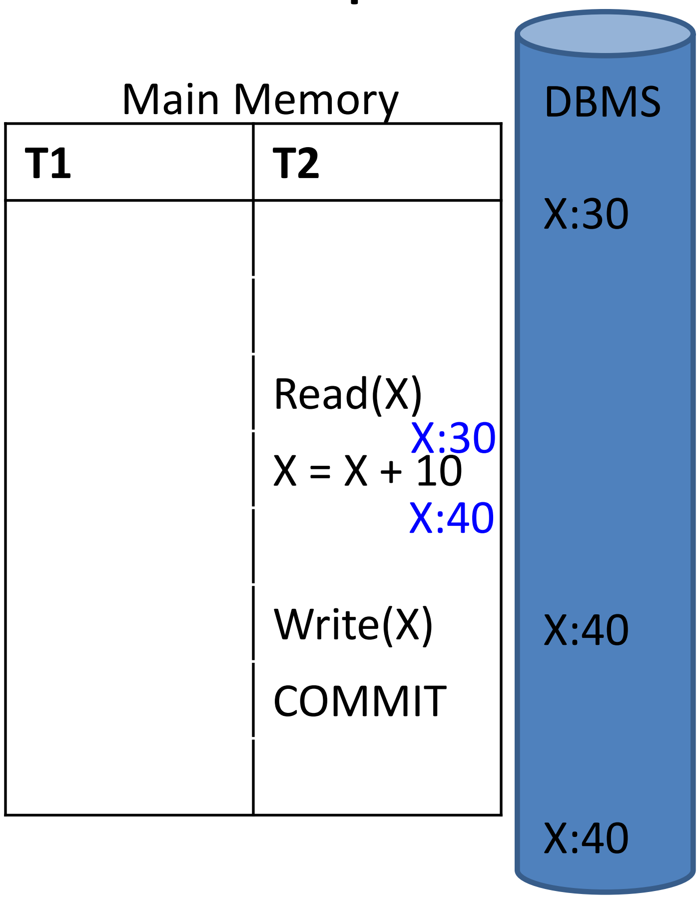
- Lost Update

Main Memory

T1	**T2**
Read(X) X:30	
X = X + 10 X:40	
Write(X)	
COMMIT	

Concurrency Problems

- Lost Update

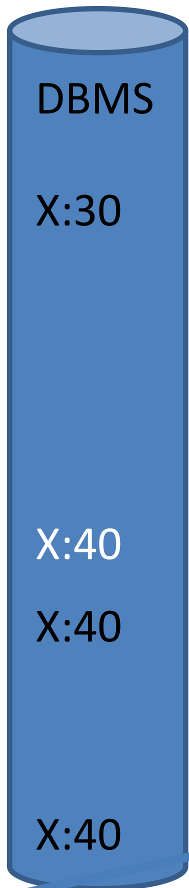


Concurrency Problems

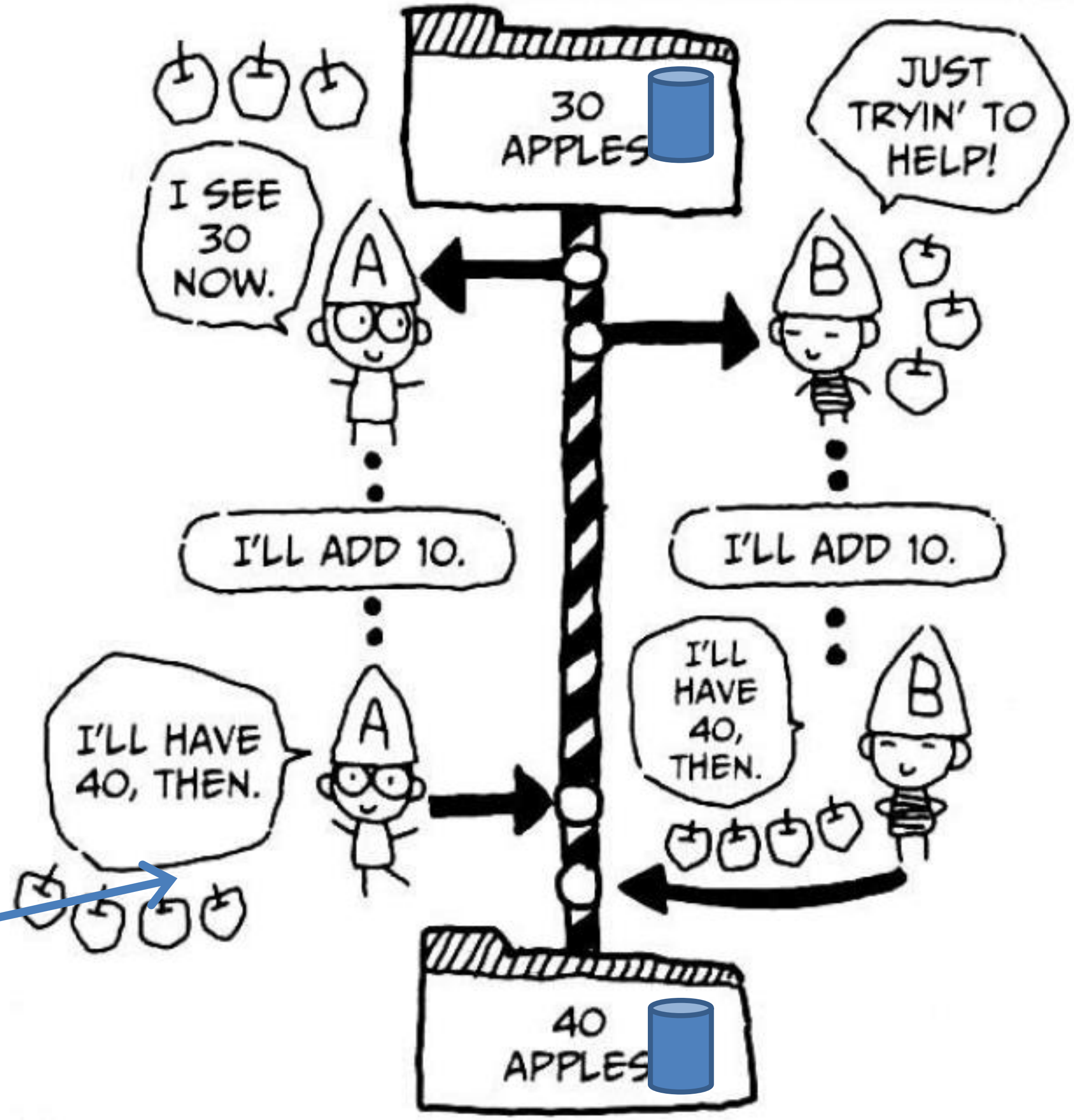
- Lost Update

Main Memory

T1	T2
Read(X) X:30	
X = X + 10 X:40	
	Read(X) X:30
	X = X + 10 X:40
Write(X)	
	Write(X)
	COMMIT
COMMIT	



This update is lost!



there should be 50 APPLES afterwards!

Concurrency Problems

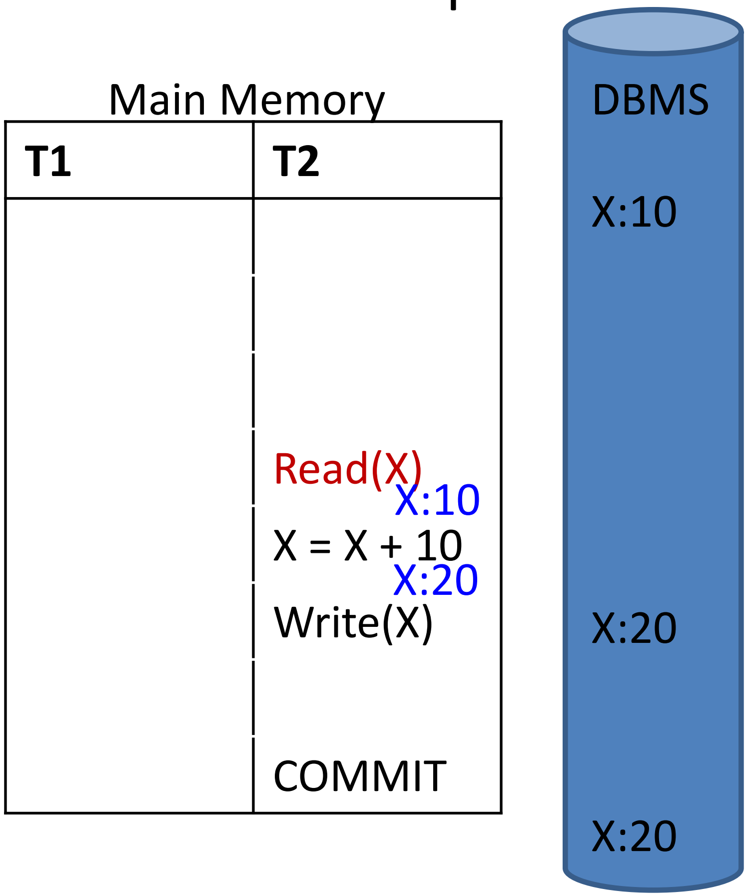
- Uncommitted Update

Main Memory

T1	T2
Read(X)	
X:10	
X = X + 10	
X:20	
Write(X)	
ROLLBACK	

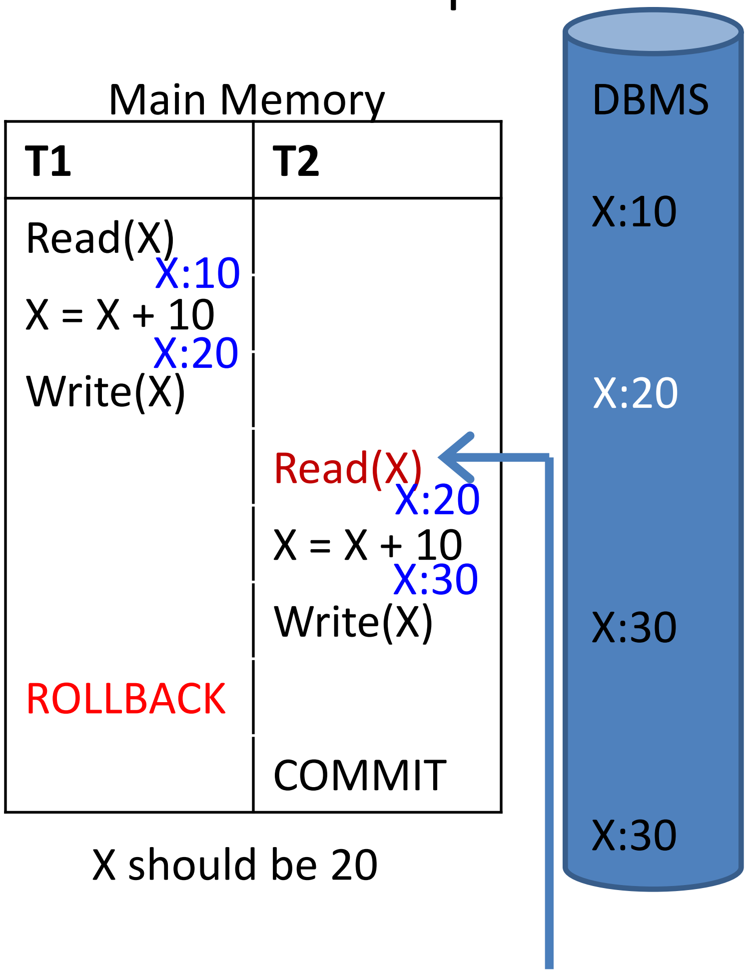
Concurrency Problems

- Uncommitted Update



Concurrency Problems

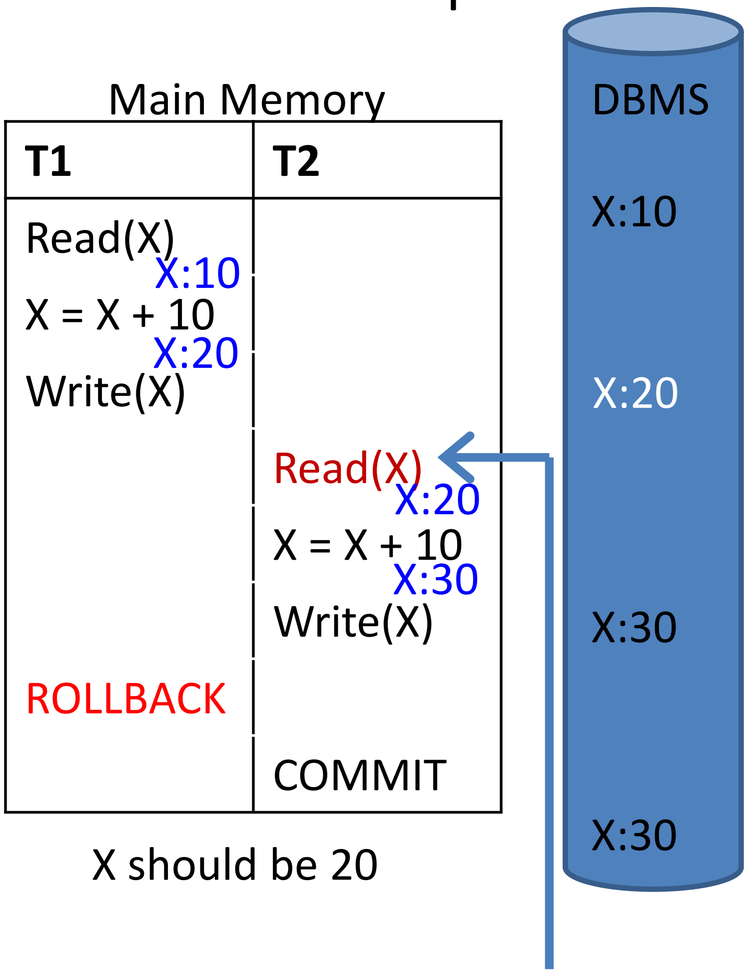
- Uncommitted Update



A read(X:20) that T2 should not have seen

Concurrency Problems

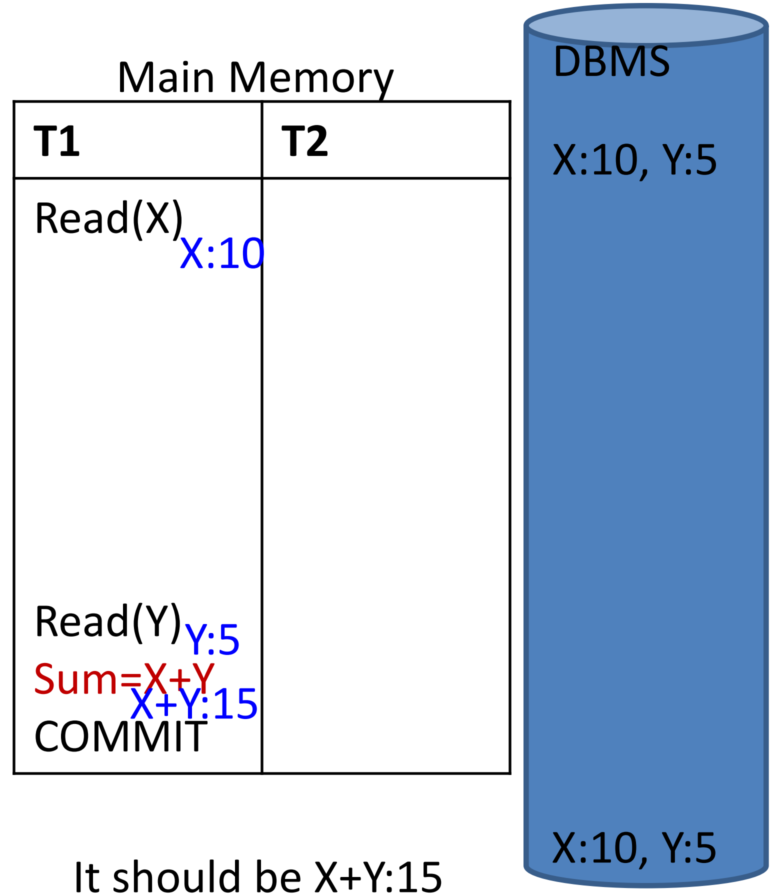
- Uncommitted Update



X should be 20

A read(X:20) that T2 should not have seen

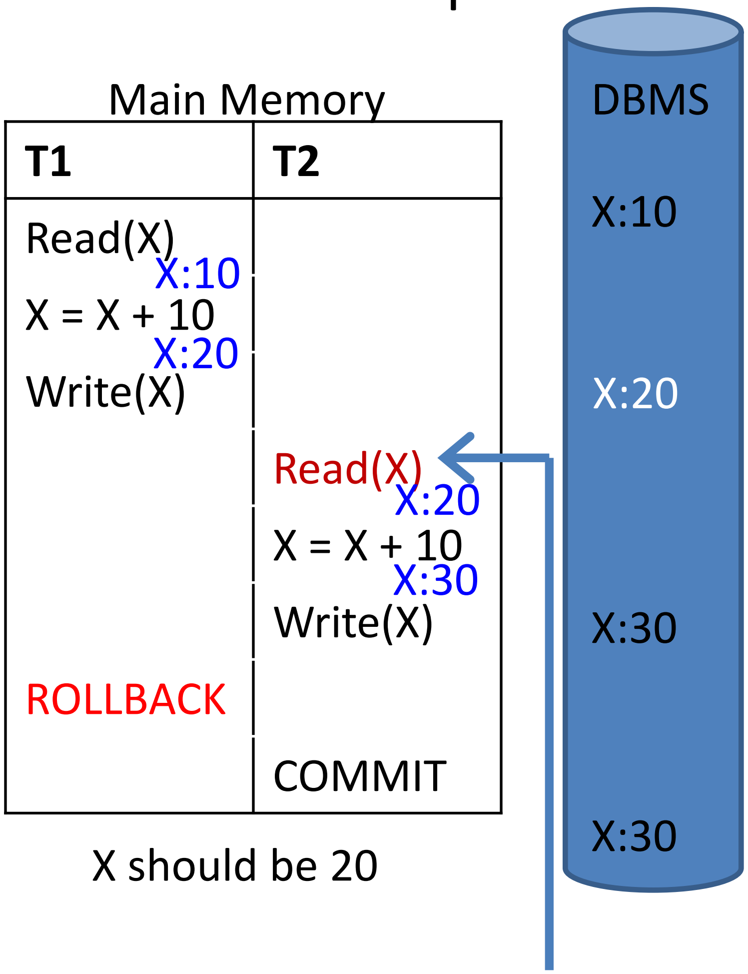
- Inconsistent Analysis



It should be X+Y:15

Concurrency Problems

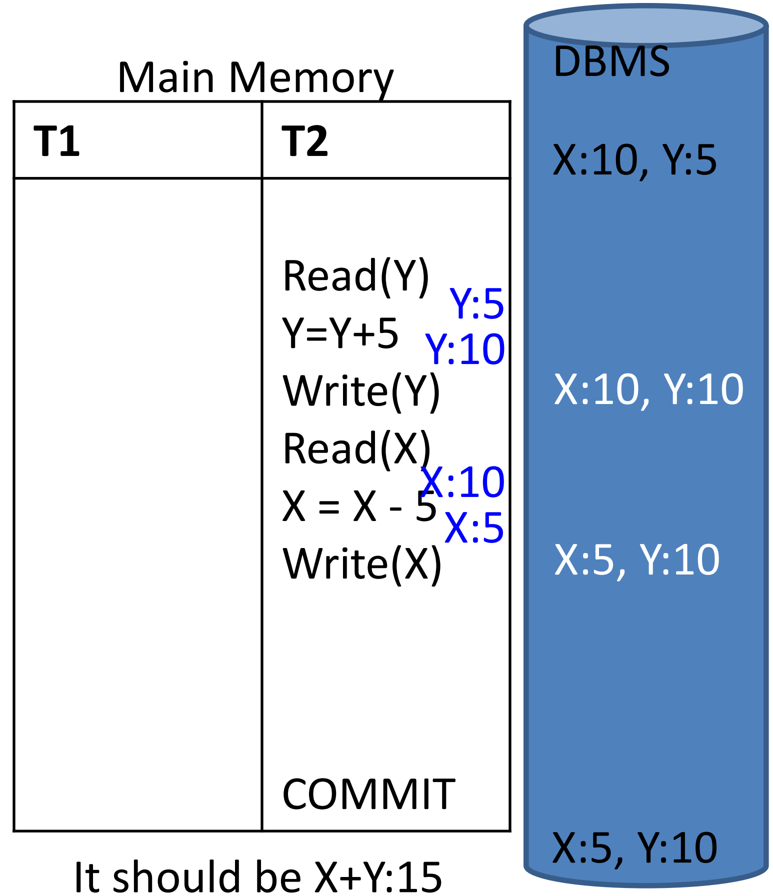
- Uncommitted Update



X should be 20

A read(X:20) that T2 should not have seen

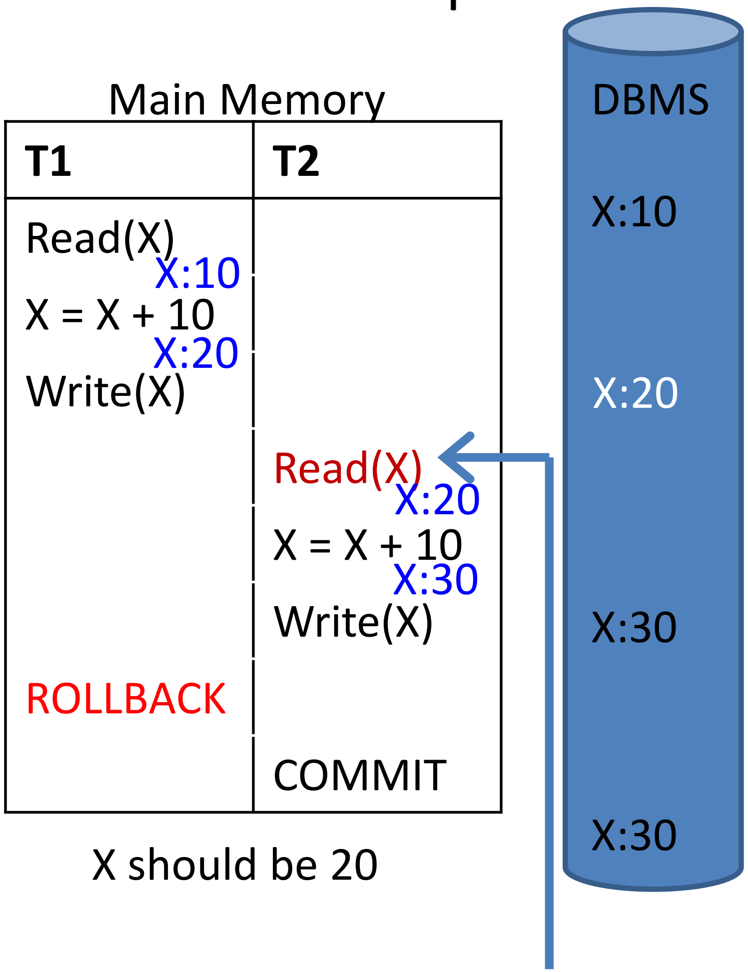
- Inconsistent Analysis



It should be X+Y:15

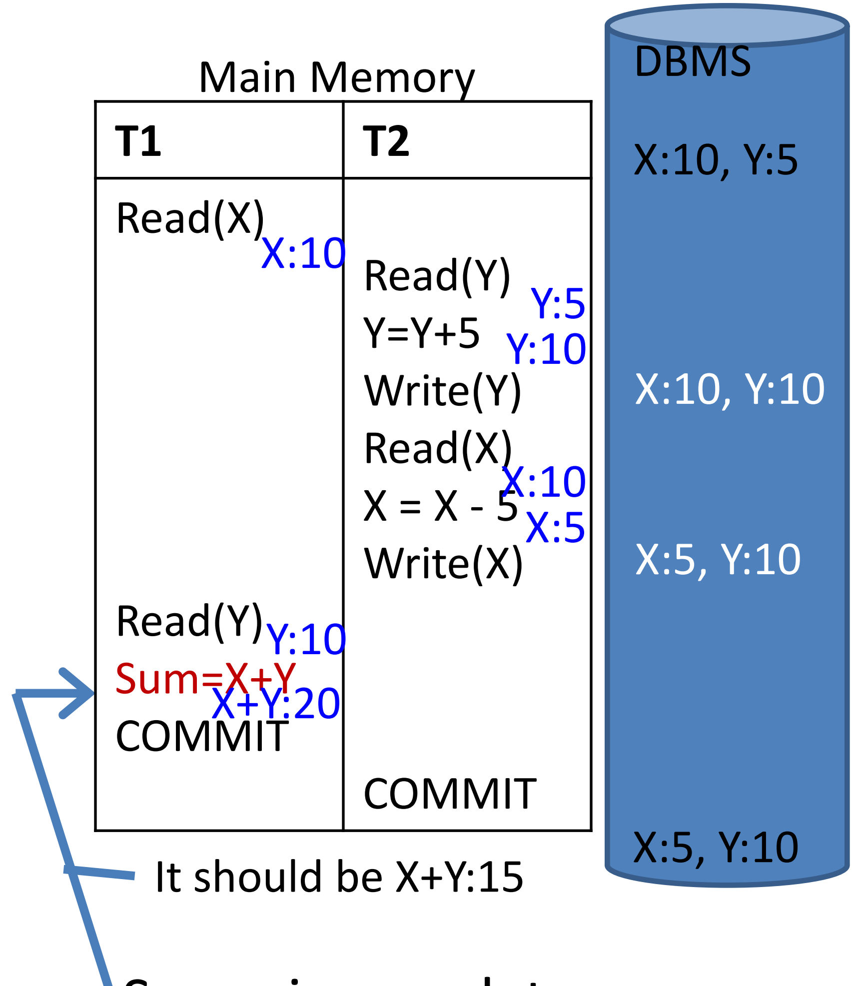
Concurrency Problems

- Uncommitted Update



A read(X:20) that T2 should not have seen

- Inconsistent Analysis



Summing up data while it is being updated

Jenga?

T1
Read(X)
X = X + 10
Write(X)
COMMIT

=5, X+Y=15

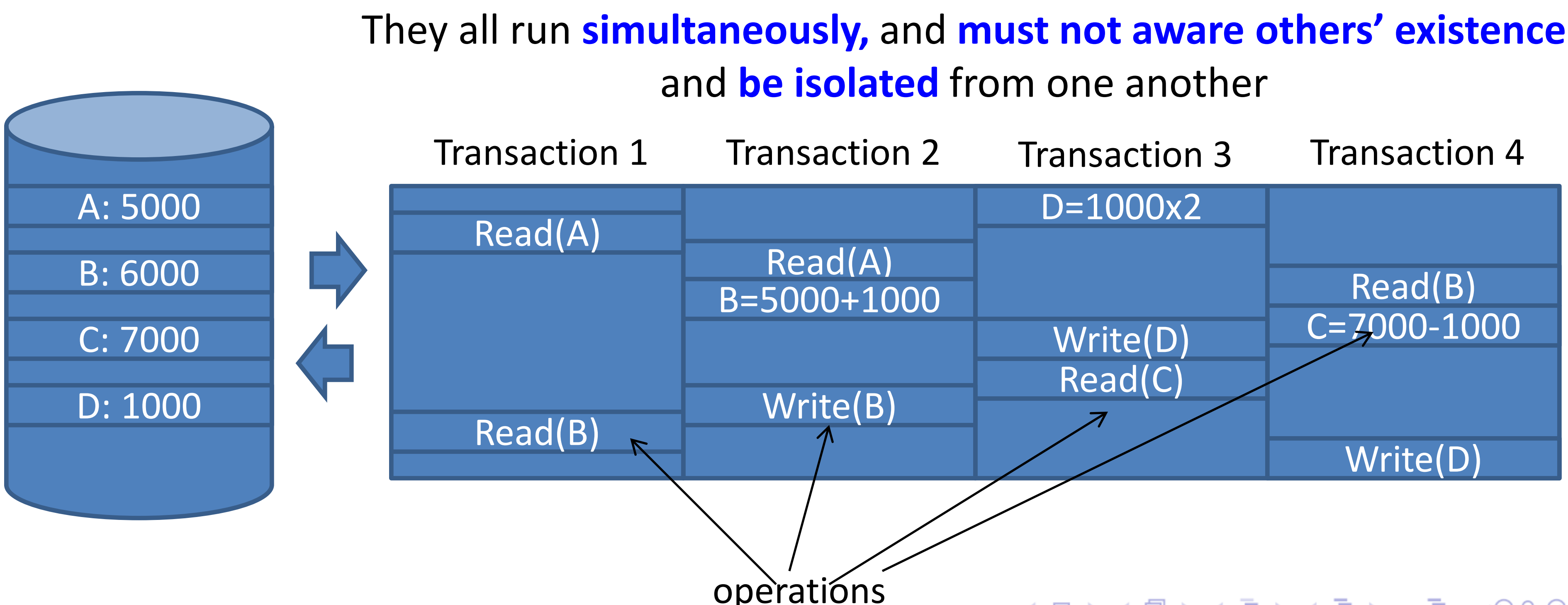
T2
Read(Y)
Y=Y+5
Write(Y)
Read(X)
X = X - 5
Write(X)
COMMIT

0, X+Y = 20

- Each statements are inter-related to one another
- If not careful, it breaks the system.
- If amount is £10 ***billion***, and there is an uncommitted update?
- Bank goes bankrupt!

Concurrency Problem

- How to solve?
  - Study **conflict serialisability**
  - Allow interleaved operations, but same result as serial schedule!



This lecture

These cover the fundamental and prepare us for **Concurrency** lecture next week.

Overview

- Transaction Lecture
  - ACID
  - DBMS Basic and Hardware
- Concurrency Problem
  - Lost update, uncommitted update, inconsistent analysis
- Serialisability **Theory**
  - Example schedules
  - Conflict operations
  - Conflict serialisable schedules
  - Testing Conflict Serialisability

Note: theory is not easy.

Roadmap –Serialisability Theory

- I) **Example schedules**
  - Serial schedules
  - Good non-serial schedule
  - Bad non-serial schedule
- II) Conflict operations
  - operations if swap, lead to incorrect results.
- III) Conflict serialisable schedules
- IV) Testing Conflict Serialisability

Serial Schedules

- Schedule1

A=£1000, B=£2000, A+B=**£3000**

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

A=£855, B=£2145: A+B=***£3000***

Schedule2

A=£1000, B=£2000, A+B=**£3000**

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

A=£850, B=£2150, A+B=***£3000***

Both in **consistent** state

Good non-serial schedule

- DBMS may run operations in transactions in any order.
- ***If lucky***, one get this schedule: the state is consistent after execution
- The right schedule is not a serial one, but it is ***equivalent*** to the serial schedule 1

A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B) commit	
	read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

Schedule3

Bad non-serial schedule

- Often, we are ***unlucky***.
- Schedule4 leads to inconsistent state!
- Not equivalent to any serial schedule – PROHIBITED!
- The job of database system must make sure the state is consistent

A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	$\boxed{\text{read}(B)}$
$\boxed{\text{write}(A)}$	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Only difference from Schedule 3

A=£950, B=£2100: **A+B=£3050**

Schedule4

Roadmap –Serialisability Theory

- I) Example schedules
  - Serial schedules
  - Good non-serial schedule
  - Bad non-serial schedule
- II) **Conflict operations**
  - operations if swap, lead to incorrect results.
- III) Conflict serialisable schedules
- IV) Testing Conflict Serialisability

Conflict operations

- Resources Q , K (e.g. tables, rows)
- Let *a* and *b* be two operations (read, write)
  - in ***two separate transactions*** respectively
- They are in conflict if switching their order would lead to different results

a	b	
Read/Write(**Q**)	Read/Write(**K**)	Ok, ***No*** conflict Q, K are different resources
Read(Q)	Read(Q)	OK, ***No*** conflict
Read(Q)	Write(Q)	**Conflict**
Write(Q)	Read(Q)	**Conflict**
Write(Q)	Write(Q)	**Conflict**

Conflict operations

Roadmap –Serialisability Theory

- I) Example schedules
  - Serial schedules
  - Good non-serial schedule
  - Bad non-serial schedule
- II) Conflict operations
  - operations if swap, lead to incorrect results.
- III) **Conflict serialisable schedules**
- IV) Testing Conflict Serialisability

Good non-serial schedule

A=£1000, B=£2000: **A+B=£3000**

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

✓

A=£855, B=£2145: **A+B=£3000**

Schedule3

Bad non-serial schedule

A=£1000, B=£2000: **A+B=£3000**

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

✗

A=£950, B=£2100: **A+B=£3050**

Schedule4

Conflict equivalent and Seriablisable

- Given a schedule S3, we create another one by **swapping non-conflicting instructions**

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule3 ---------------Showing only read & write------------------- Schedule1

- S3 and S1 are ***conflict equivalent***
- S (e.g. schedule 3) is ***conflict seriablisable*** if it is conflict equivalent to a serial schedule (e.g. schedule 1)

Conflict Serialisability

- Two schedules are equivalent if they have the same effect.
- A schedule is ***serialisable*** if it is equivalent to some serial schedule

Schedule3

A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B) commit	
	read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

equivalent

Schedule1

A=£1000, B=£2000, A+B=£3000

T_1	T_2
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B) commit	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B) commit

A=£855, B=£2145: A+B=£3000

Non-"Conflict Serialisable" Schedule

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- All operations are in conflict.
- ***cannot*** swap non-conflict instructions to arrive at a serial schedule
- => Non-"conflict serializable" schedule

Roadmap –Serialisability Theory

- I) Example schedules
  - Serial schedules
  - Good non-serial schedule
  - Bad non-serial schedule
- II) Conflict operations
  - operations if swap, lead to incorrect results.
- III) Conflict serialisable schedules
- IV) **Testing Conflict Serialisability**

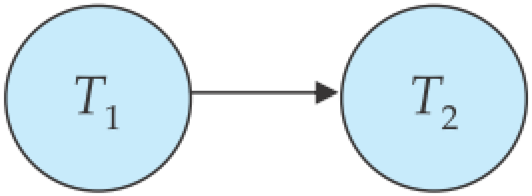
Conflict Serialisability

- Conflict serialisable schedules are the main focus of concurrency control in DBMS
  - Allow interleaving operations
  - Guaranteed to behave as a serial schedule

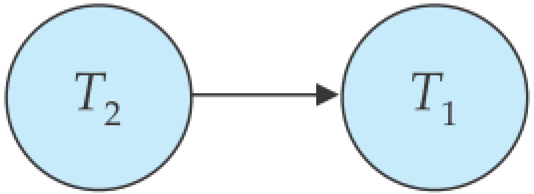
- Important questions
  - How do we test a schedule is conflict serialisable?
  - How do we construct conflict serialisable schedules? (next lecture)

Testing Conflict Serialisability?

- Precedence Graph – a directed graph
  - A vertex for each transaction (T1, T2) involved in the schedule being tested.
  - An arc from T1 to T2 if T1 should be executed before T2.



Precedence graph
for serial schedule 1

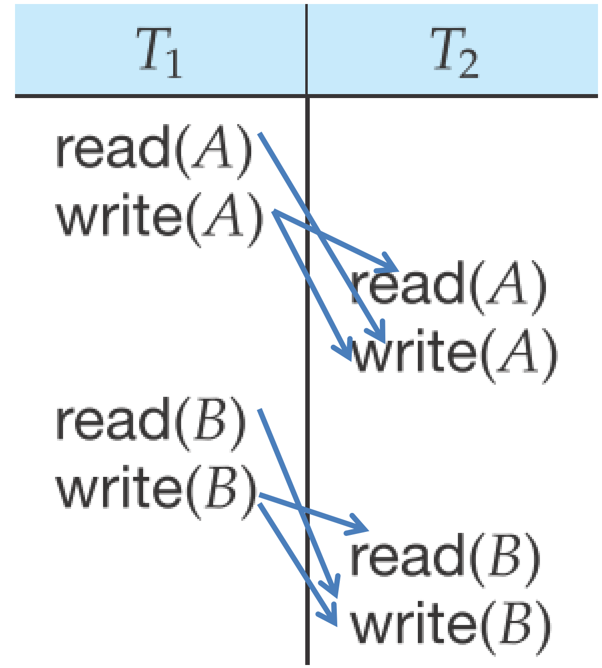


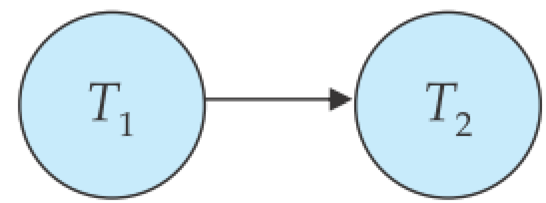
Precedence graph
for serial schedule 2

Precedence Graph

conflicting operations, no swap

- Precedence Graph
  - For each operation
    - From the top
    - Look downwards
    - Are the operations in conflict?
    - Add an arc
  - Summarise all arcs
  - Each case indicates that T1 should be executed before T2



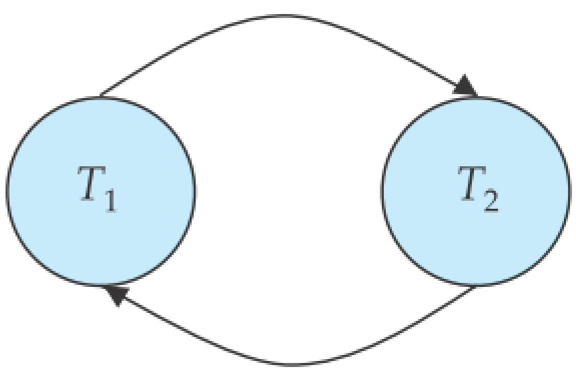


Precedence graph for serial schedule 3

Precedence Graph

- Schedule 4 has this precedence graph

- T1 reads A before T2 writes A
- T1 reads B before T2 writes B
- T2 reads A before T1 writes A
- T2 writes A before T1 writes A
- T2 reads B before T1 writes B



Schedule4

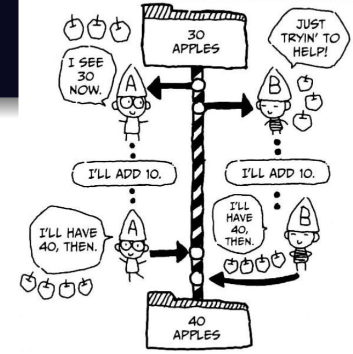
A=£1000, B=£2000: A+B=£3000

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

A=£950, B=£2100: **A+B=£3050**

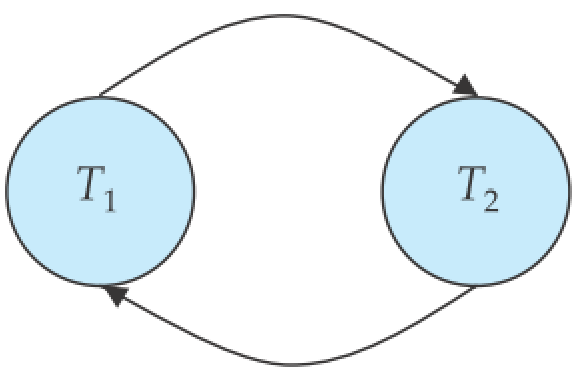
Cycle => Data Inconsistency

Precedence Graph – Lost Update



- The lost update problem has this precedence graph

- T1 reads X before T2 writes X
- T1 writes X before T2 writes X
- T2 reads X before T1 write X



T1	T2
Read(X)	
X = X + 10	
	Read(X)
	X = X + 10
Write(X)	
	Write(X)
	COMMIT
COMMIT	

Precedence Graph

- A schedule is ***conflict serialisable*** if and only if its precedence graph is ***acyclic (no cycle)***.

- To test for conflict serialisability
  - Construct the precedence graph
  - Invoke cycle-detection algorithm (e.g. Depth-First Search)

These cover the fundamental and prepare us for **Concurrency** lecture next week.

Overview

- Transaction Lecture
  - ACID
  - DBMS Basic and Hardware
- Concurrency Problem
  - Lost update, uncommitted update, inconsistent analysis
- Serialisability **Theory**
  - Example schedules
  - Conflict operations
  - Conflict serialisable schedules
  - Testing Conflict Serialisability

Note: theory is not easy.

Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			read(Z)	
read(U)				
write(U)				

Conflict serialisable?

Draw a precedence graph.