Lecture 20: Transaction

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**

Sections: 001, 002, 003

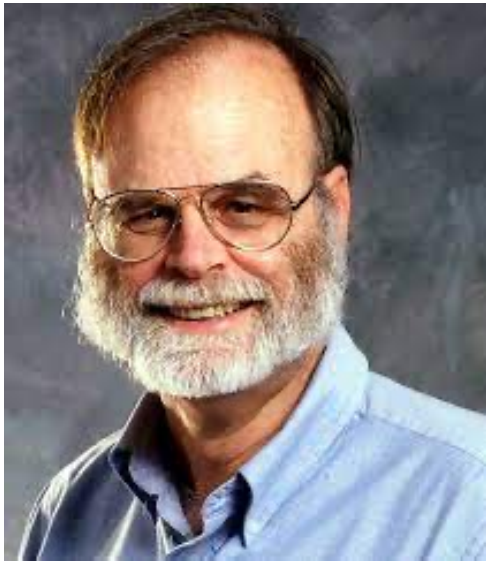
Announcements

- Assignment 3
  - Due next Tue, July 22
- Group Project
  - Schedule a demo time with TA by today
  - Final report and code due on July 29
- Student Course Perception Surveys
  - Please help submit your feedback about this course!
  - From today to July 30

(Recap) Transactions

```
-- Begins implicitly
SELECT … ;
UPDATE … ;
ROLLBACK | COMMIT
```

- A transaction is a sequence of database operations (read or write)
- ACID properties of transactions (TXs)
  - Atomicity: TXs are either completely done or not done at all
  - Consistency: TXs should leave the database in a consistent state
  - Isolation: TXs must behave as if they execute in isolation
  - Durability: Effects of committed TXs are resilient against failures



Jim Gray, Turing Award 1998, who coined this term (as well as data cube and many other things)

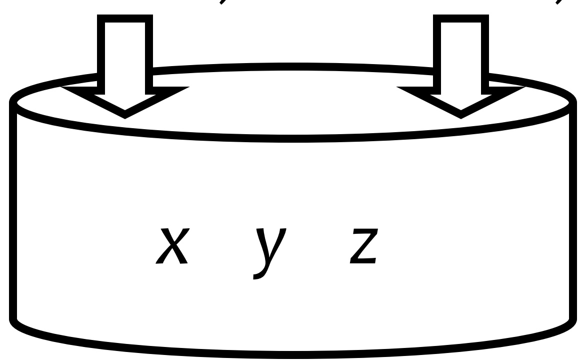
Outline for today

- Concurrency control -- Isolation
  - Locking-based control
- Recovery -- Atomicity and Durability
  - Logging for undo and redo

Concurrency control

- Goal: ensure the "I" (isolation) in ACID

T_1:	T_2:
r1(x);	r2(x);
w1(x);	w2(x);
r1(y);	r2(z);
w1(y);	w2(z);
commit;	commit;



(Recap) Serial execution histories

- $T_1 = \{w_1(x), w_1(y), c_1\}, T_2 = \{r_2(x), r_2(y), c_2\}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b ✓

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d ✓

(Recap) Serializable

Two execution histories are (conflict) equivalent if … and each pair of conflicting operations have the same ordering in each history

- A history H is said to be (conflict) serializable if there is some serial history H' (conflict) equivalent to H.

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

$H_a = H_b$ H_c ?

(Recap) Serializable

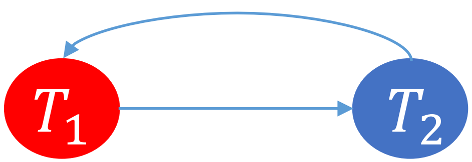
- Example: $H_c = w_1[x] r_2[x] r_2[y] w_1[y] c_1 c_2$

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

A history is serializable if and only if its serialization graph is acyclic (i.e., no cycles)

$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$;
$w_1[y]$ and $r_2[y]$ conflict, and $r_2[y] < w_1[y]$



Not serializable

How to help non-serializable history achieve serializability?

Locking

(Pessimistic) Assume that conflicts will happen and take preventive action

- If a transaction wants to read x , it must first request a shared lock (S mode) on x
- If a transaction wants to modify x, it must first request an exclusive lock (X mode) on x
- Allow one exclusive lock, or multiple shared locks

Mode of the lock requested

Mode of lock currently held by other transactions

	lockS	lockX
lockS	Yes	No
lockX	No	No

Grant the lock?

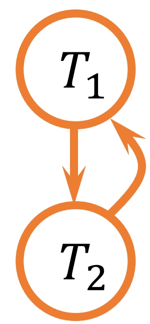
Compatibility matrix

Basic locking is not enough

T_1	T_2
lock-X(x)	
r1(x)	
w1(x)	
unlock(x)	
	lock-X(x)
	r2(x)
	w2(x)
	unlock(x)
	lock-X(y)
	r2(y)
	w2(y)
	unlock(y)
lock-X(y)	
r1(y)	
w1(y)	
unlock(y)	

Possible schedule under locking

But still not serializable!



Basic locking is not enough

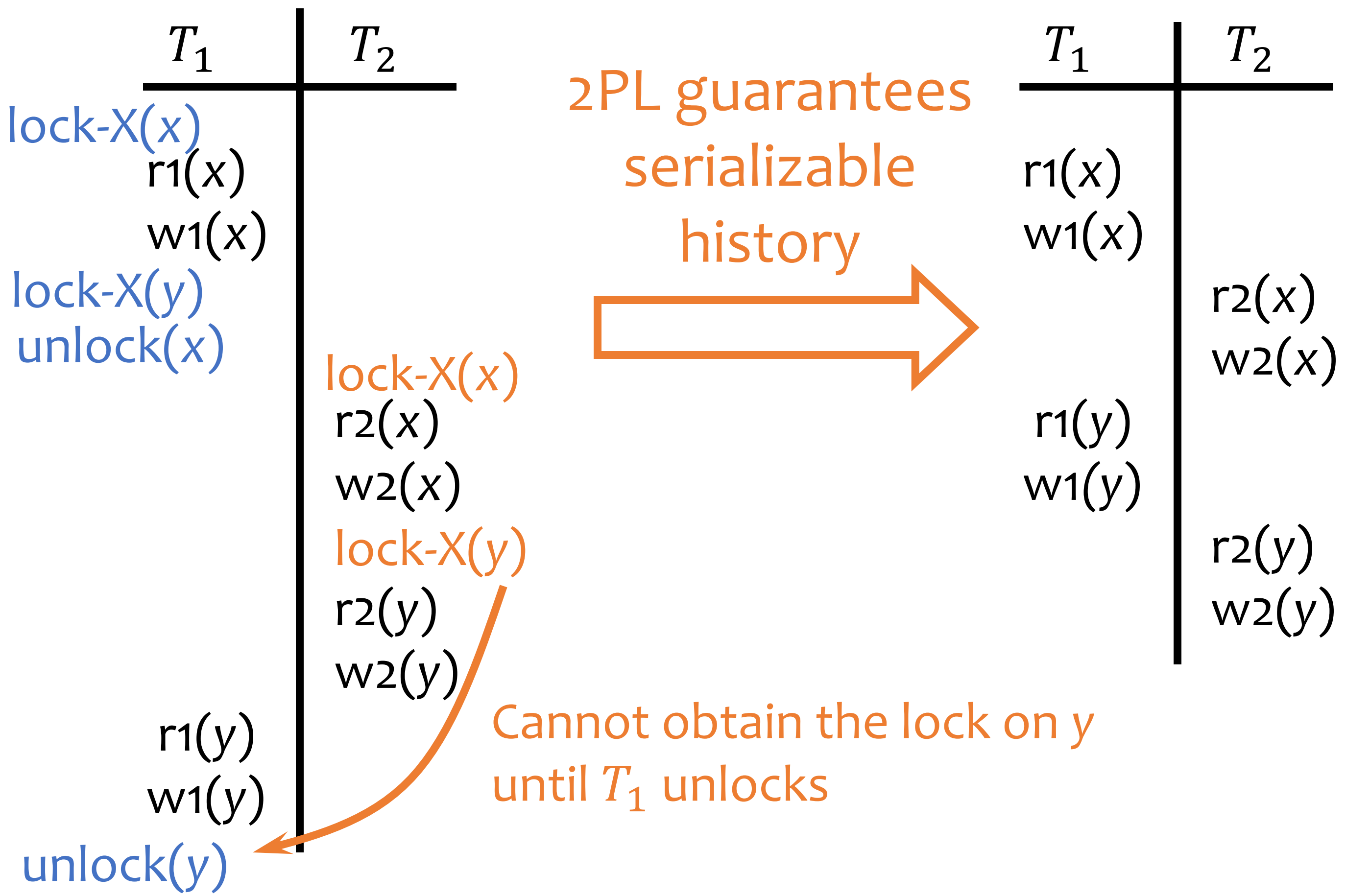
Suppose x=y=100

Add 1 to both *x* and *y* (preserve *x*=*y*)	T_1	T_2	Multiply both *x* and *y* by 2 (preserves *x*=*y*)
	lock-X(*x*)		
Read 100	r1(*x*)		
Write 100+1	w1(*x*)		
	unlock(*x*)		
		lock-X(*x*)	
		r2(*x*)	Read 101
		w2(*x*)	Write 101*2
		unlock(*x*)	
		lock-X(*y*)	
		r2(*y*)	Read 100
		w2(*y*)	Write 100*2
		unlock(*y*)	
	lock-X(*y*)		
Read 200	r1(*y*)		
Write 200+1	w1(*y*)		
	unlock(*y*)		

x != y

Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks; Phase 2: release locks



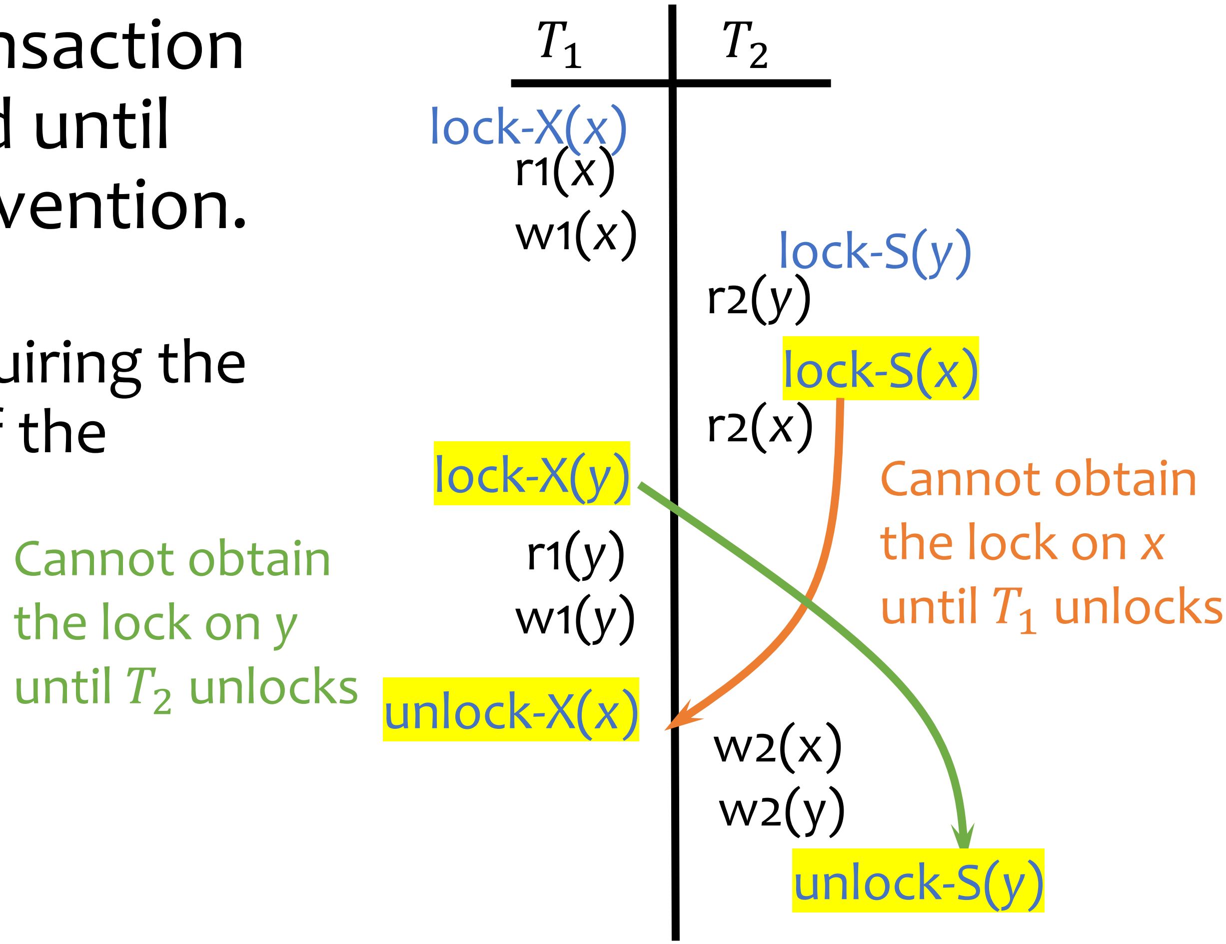
Remaining problems of 2PL

T_1	T_2
lock-X(x)	
r1(x)	
w1(x)	
lock-X(y)	
unlock(x)	
	lock-X(x)
	r2(x)
	w2(x)
r1(y)	
w1(y)	
unlock(y)	
	lock-X(y)
	r2(y)
	w2(y)
	unlock(x)
	unlock(y)
	commit
abort	

- T_2 has read uncommitted data written by T_1
- If T_1 aborts, then T_2 must abort as well
- Cascading aborts possible if other transactions have read data written by T_2
- Even worse, schedule is not recoverable if T_2 commits before T_1

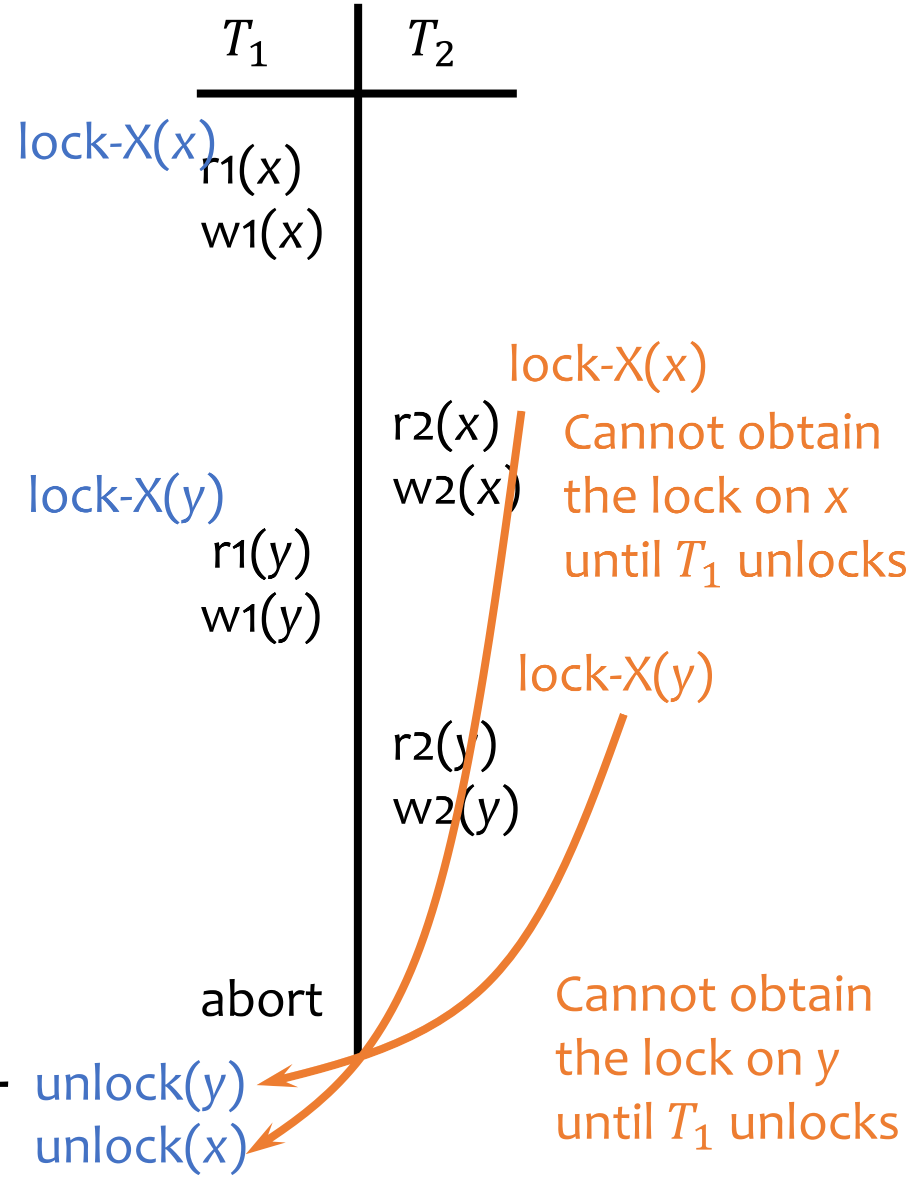
Remaining problems of 2PL

- Deadlock: A transaction remains blocked until there is an intervention.
  - 2PL may cause deadlocks, requiring the abort of one of the transactions



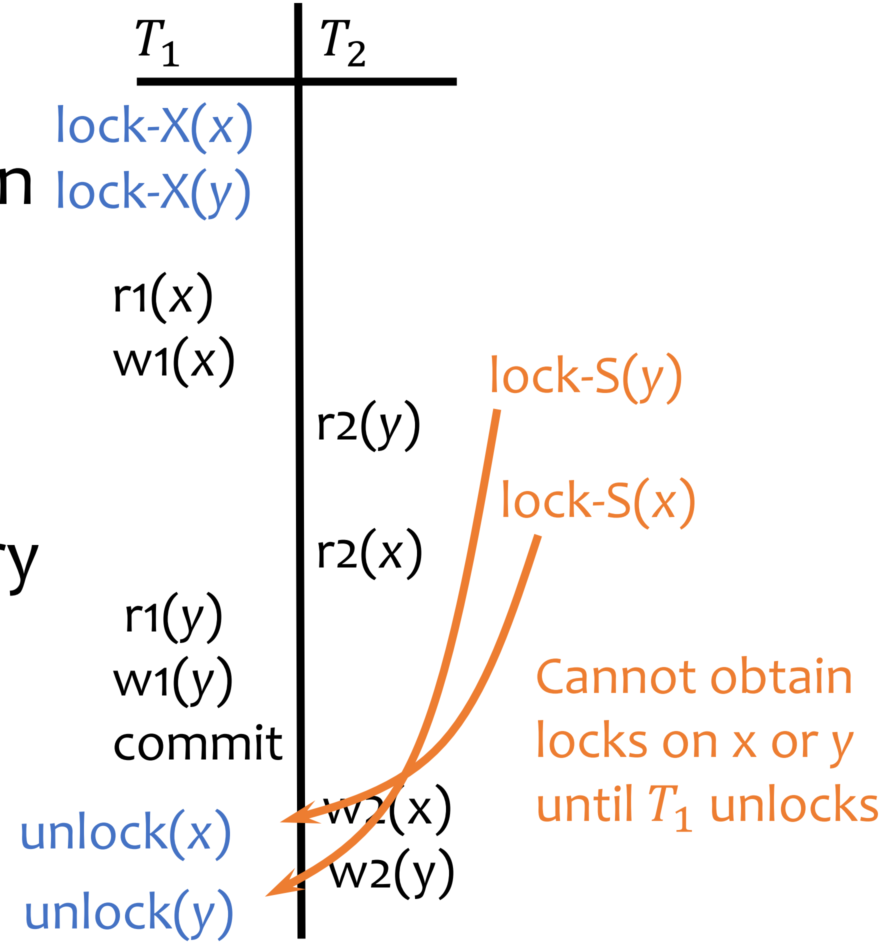
Strict 2PL

- Only release X-locks when commit/abort
  - A write will block all other reads until the write commits or aborts
- Used in many practical DBMSs
  - No cascading aborts
  - But it can still lead to deadlocks! (see slide 14)
- Less concurrency than 2PL



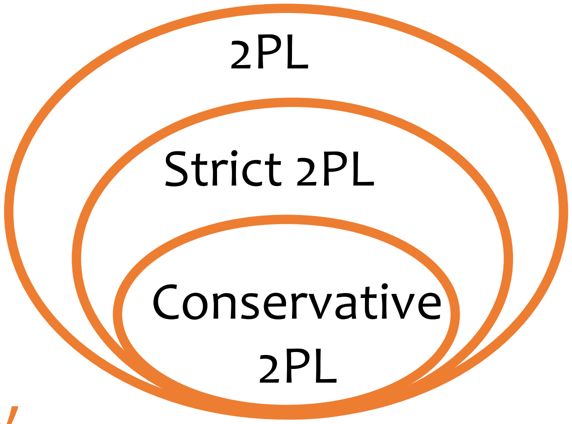
Conservative 2PL

- Only acquire locks at the beginning of the transaction and release X-locks when commit/abort
- Not practical due to the very limited concurrency
  - No cascading aborts
  - No deadlocks



Outline for today

- Concurrency control -- isolation
  - Serializability: all
  - Concurrency: conservative 2PL < strict 2PL < 2PL
  - No cascading aborts: conservative 2PL, strict 2PL
  - No deadlocks: conservative 2PL

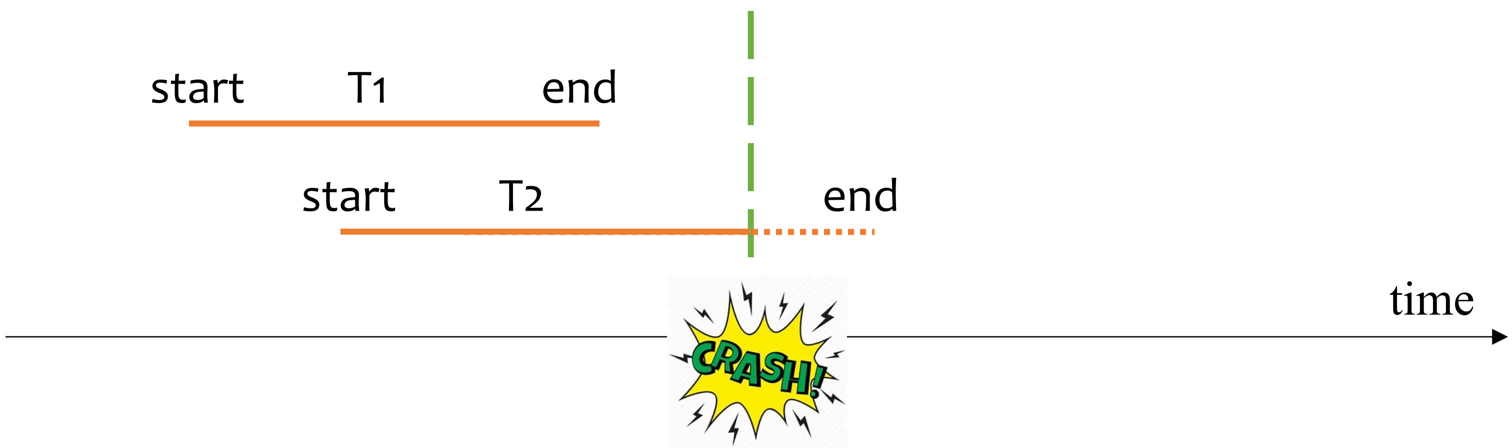


- Recovery – atomicity and durability
  - Logging for undo and redo

Failures

- System crashes right after a transaction *T1* commits; but not all effects of *T1* were written to disk
  - How do we complete/redo *T1* (durability)?

- System crashes in the middle of a transaction *T2*; partial effects of *T2* were written to disk
  - How do we undo *T2* (atomicity)?



Naïve approach: Force -- durability

T1 (balance transfer of $100 from *A* to *B*)

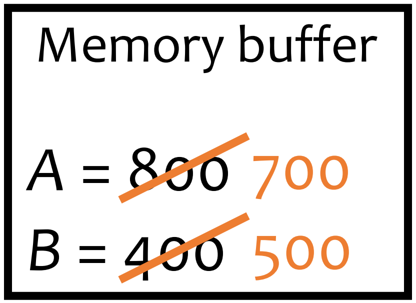
read(*A*); *A* = *A* – 100;

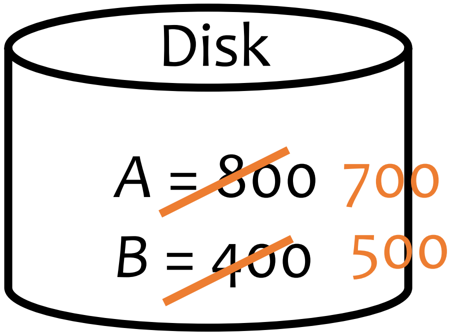
write(*A*);

read(*B*); *B* = B+ 100;

write(*B*);

commit;





Force: All updates are immediately written to the disk, so when a transaction commits all changes are reflected on disk

But lots of random writes hurt performance!

Naïve approach: No steal -- atomicity

T1 (balance transfer of $100 from *A* to *B*)

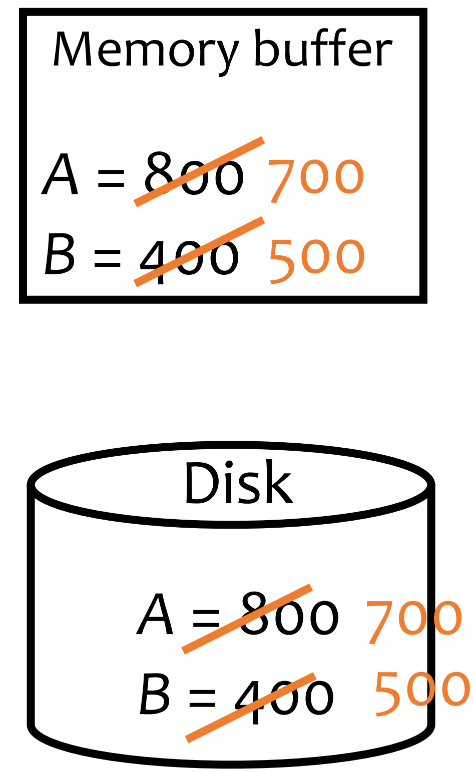
read(*A*); *A* = *A* – 100;

write(*A*);

read(*B*); *B* = *B* + 100;

write(*B*);

commit;

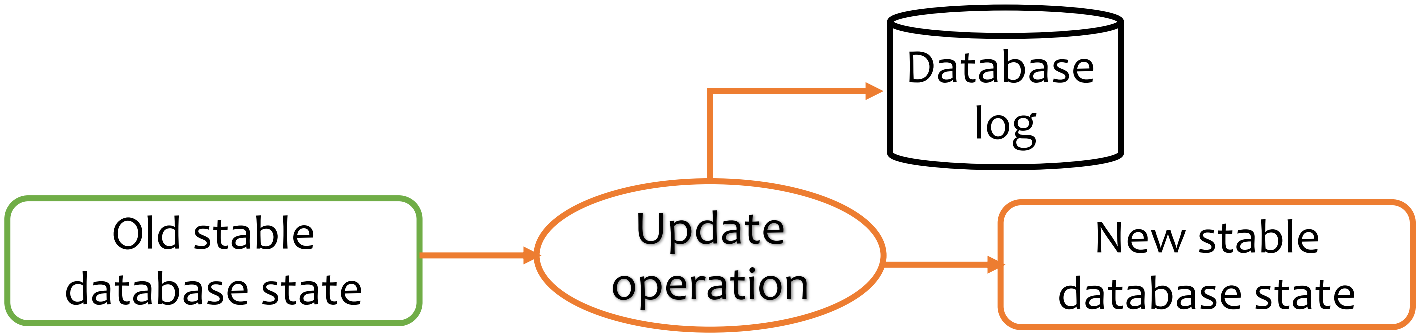


No steal: all writes are held in memory until the transaction commits, so it is always possible to revert to a consistent state, as uncommitted changes are never lost.

But lots of dirty data requires large memory

Logging

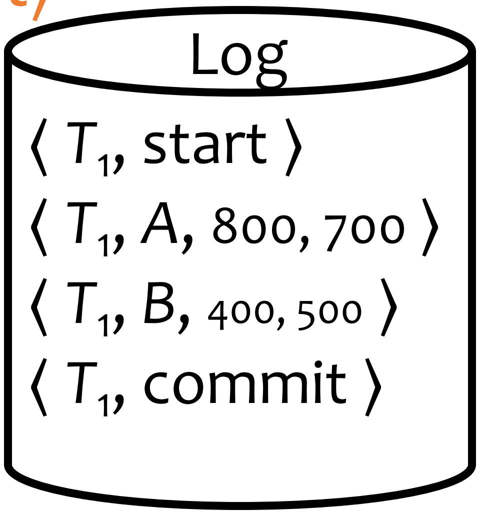
- Database log: sequence of log records, recording all changes made to the database, written to stable storage (e.g., disk) during normal operation



- One change turns into two -- bad for performance?
  - But writes to log are sequential (append to the end of log)

Log

- When a transaction T starts: $\langle T$, start$\rangle$
- Record values before and after each modification of data item X: $\langle T, X, old\_value\_of\_X, new\_value\_of\_X\rangle$
- When a transaction T commits: $\langle T$, commit$\rangle$
- When a transaction T aborts: $\langle T$, abort$\rangle$



When to write log records?

- Before X is modified or after?

- Write-ahead logging (WAL): Before *X* is modified on disk, the log record pertaining to *X* must be flushed

- Without WAL, system might crash after *X* is modified on disk but before its log record is written to disk—no way to undo

Undo/redo logging example

T1 (balance transfer of $100 from *A* to *B*)

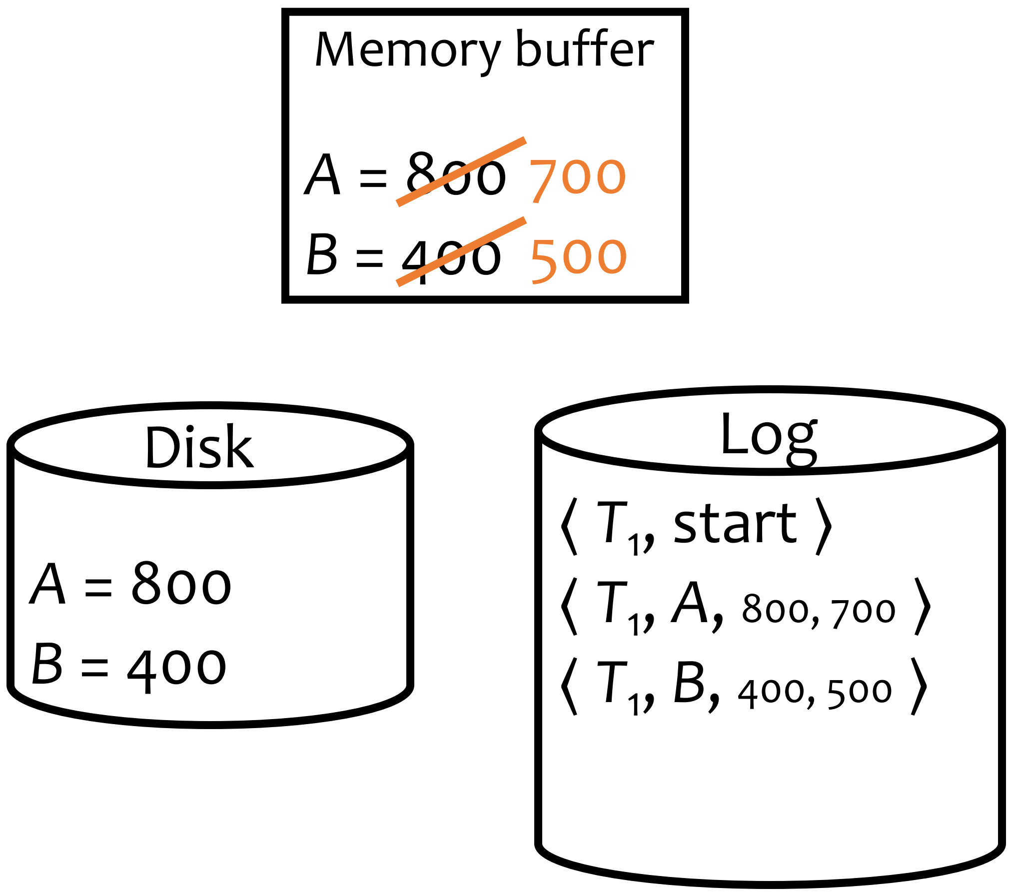
read(*A*); *A* = *A* – 100;

write(*A*);

read(*B*); *B* = B+ 100;

write(*B*);

commit;



WAL: Before A,B are modified on disk, their log info must be flushed

Undo/redo logging example

T1 (balance transfer of $100 from *A* to *B*)

read(*A*, *a*); *a* = *a* – 100;

write(*A*, *a*);

read(*B*, *b*); *b* = *b* + 100;

write(*B*, *b*);

Memory buffer

A = ~~800~~ 700

B = ~~400~~ 500

Disk

A = ~~800~~ 700

B = 400

Log

$\langle T_1, \text{start} \rangle$

$\boxed{\langle T_1, A, 800, 700 \rangle}$

$\langle T_1, B, 400, 500 \rangle$

Steal: can flush before commit

If system crashes before *T1* commits, we have the old value of A stored on the log to **undo** T1

Undo/redo logging example

T1 (balance transfer of $100 from *A* to *B*)

read(A, a); $a = a - 100$;

write(A, a);

read(B, b); $b = b + 100$;

write(B, b);

commit;

Memory buffer

$A = \cancel{800}$ 700

$B = \cancel{400}$ 500

Disk

$A = 800$

$B = 400$

Log

⟨ T_1, start ⟩

⟨ T_1, A, 800, 700 ⟩

⟨ T_1, B, 400, 500 ⟩

⟨ T_1, commit ⟩

No force: can flush after commit

If system crashes before we flush the changes of A, B to the disk, we have their new committed values on the log to **redo** T1

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 T2 T3

Log

Start of log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
	T_2, commit
	T_4, start
	T_3, abort
	T_4, y, 200, 50

End of log

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:

T1 ~~T2~~ T3

Start of log

	Log
redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
	T_4, start
	T_3, abort
	T_4, y, 200, 50

End of log

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ T3 T4

Start of log

End of log

Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
	T_3, abort
	T_4, y, 200, 50

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

when T aborts, we reverse all operations before abort

List of active transactions at crash:
T1 ~~T2~~ T3 T4

Start of log

Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
redo	T_3, abort
	T_4, y, 200, 50

End of log

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:

T1 ~~T2~~ ~~T3~~ T4

Start of log

	Log
redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
redo	T_3, abort
redo	T_4, y, 200, 50

End of log

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ ~~200~~ 50
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ ~~T3~~ T4

Log

Start of log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
redo	T_3, abort
redo	T_4, y, 200, 50

End of log

Log example - undo

- Undo phase: T1, T4

x: ~~99~~ ~~100~~ 99
y: ~~199~~ ~~200~~ ~~50~~ 200
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ ~~T3~~ T4

Start of log

End of log

Log

*	T_1, start
undo	T_1, x, 99, 100
	T_2, start
	T_2, y, 199, 200
	T_3, start
	T_3, z, 51, 50
	T_2, w, 1000, 10
	T_2, commit
*	T_4, start
	T_3, abort
undo	T_4, y, 200, 50

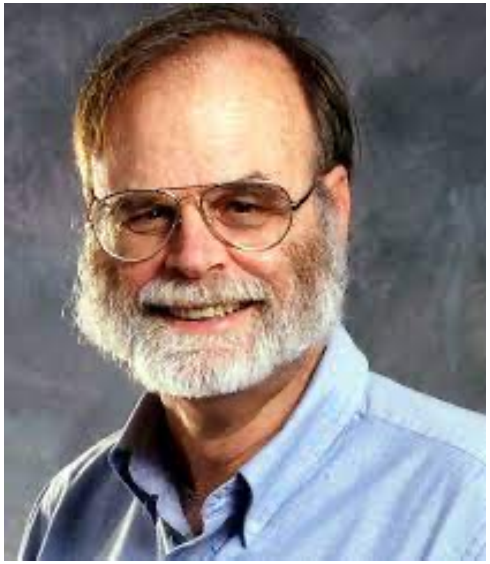
T_4, y, 200
T_4, abort
T_1, x, 99
T_1, abort

Undo/redo logging - repeat history!

- U: track the set of active transactions at crash
- Redo phase: scan forward to the end of the log
  - For a log record ⟨ *T*, start ⟩, add *T* to *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *new*)
  - For a log record ⟨ *T*, commit | abort ⟩, remove *T* from *U*
    - *If abort, undo changes of T* i.e., for a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *old*)
- Undo phase: scan backward to the start of the log
  - Undo the effects of transactions in *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩ where *T* is in *U*, issue write(*X*, *old*), and log this operation too, i.e., add ⟨ *T*, *X*, *old* ⟩
  - Log ⟨*T*, abort ⟩ when all effects of *T* have been undone

Summary of Transactions

- ACID properties of transactions (TXs)
  - Atomicity: TXs are either completely done or not done at all (logging)
  - Consistency: TXs should leave the database in a consistent state
  - Isolation: TXs must behave as if they execute in isolation (serializable; concurrency control)
  - Durability: Effects of committed TXs are resilient against failures (logging)



Jim Gray, Turing Award 1998, who coined this term (as well as data cube and many other things)

What's next?

- No lectures next week
- Final review on July 29
- Please help submit your feedback via SCP surveys!

Student Course Perceptions Surveys

Your chance to share your learning experience.
Your feedback is important!

- Login using your WatIAM credentials
- Select your course from the list
- Answer all questions in one sitting
- Check the instructor + course to make sure you had the **right learning experience in mind** while responding!
- Hit Submit!

Credentials are never linked to responses!

perceptions.uwaterloo.ca

Quick Anonymous Impactful

Difficulties?
Contact kabecker@uwaterloo.ca

Student Course Perceptions Surveys

Your chance to share your learning experience.
Your feedback is important!

Who has access to SCP results?

- Written comments: only the course instructor
- Numerical ratings: course instructor and academic leaders

How are SCP results used?

- Help instructors improve teaching and courses
- Inform pay and tenure decisions
- Contribute to decisions about program improvement and future teaching assignments

perceptions.uwaterloo.ca

Quick Anonymous Impactful

Difficulties?
Contact kabecker@uwaterloo.ca

Student Course Perceptions Surveys

Your chance to share your learning experience.
Your feedback is important!

Giving Effective Feedback

- Be honest: write about your learning experience
- Be specific: provide examples
- Be focused: restrict comments to your own experience
- Be constructive: offer suggestions for improvement

Please always use language that supports your instructors' well-being. Abusive comments (e.g., about aspects of instructor identity) may result in your entire survey response being removed.

perceptions.uwaterloo.ca

Quick Anonymous Impactful

Difficulties?
Contact kabecker@uwaterloo.ca