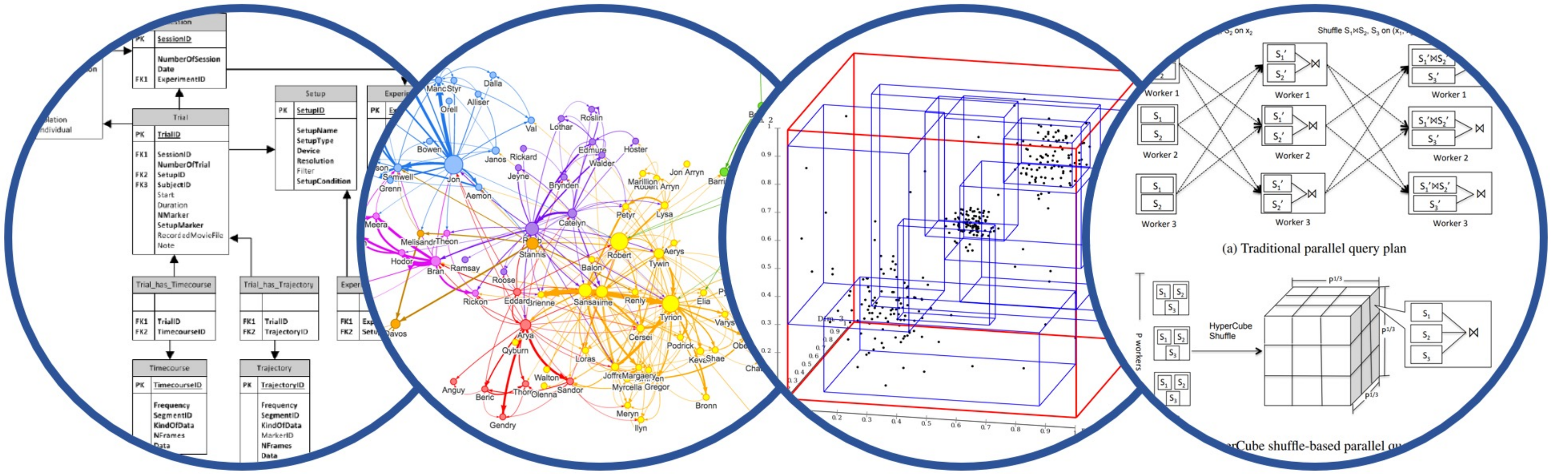


Introduction to Data Management

Transactions: Isolation Levels

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- HW5 is due on Friday

Recap

- TXN = sequence of Reads and Writes of elements
  - BEGIN TRANSACTION
  - COMMIT or ROLLBACK
- Schedule = interleaving of operations of TXNs
- Serial Schedule = one TXN after the other

A Schedule

time

T1	T2
READ(A, t)	
	READ(A, s)
	s := s*2
t := t+100	
WRITE(A, t)	
	WRITE(A,s)
	READ(B,s)
	s := s*2
READ(B, t)	
	WRITE(B,s)
t := t+100	
WRITE(B,t)	

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

time ↓

Recap

- Serializable Schedule = equivalent to a serial one
- Conflict Serializable Schedule = …

Serializable and Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

Non-Serializable, Non-Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable, Non-Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s+2
	WRITE(A,s)
	READ(B,s)
	s := s+2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable, Non-Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s+2
	WRITE(A,s)
	READ(B,s)
	s := s+2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

(x+100)+2=
(x+2)+100

Non-Serializable, Non-Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
	READ(A, s)
WRITE(A, t)	
	s := s+2
	WRITE(A,s)
	READ(B,s)
	s := s+2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Non-Serializable, Non-Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
	READ(A, s)
WRITE(A, t)	
	s := s+2
	WRITE(A,s)
	READ(B,s)
	s := s+2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Why not serializable?

Non-Serializable, Non-Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
	READ(A, s)
WRITE(A, t)	
	s := s+2
	WRITE(A,s)
	READ(B,s)
	s := s+2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Lost update

Why not serializable?

Discussion

- To check for conflict-serializability
  use the precedence graph

- To check for serializability:
  need to understand what TXNs are doing

Recap: Concurrency Control Manager

- Scheduler a.k.a. Concurrency Control Manager
- Pessimistic (Locks) or Optimistic (various…)

Locks:

- $L_i(A)$ = transaction T_i acquires lock for element A
- $U_i(A)$ = transaction T_i releases lock for element A

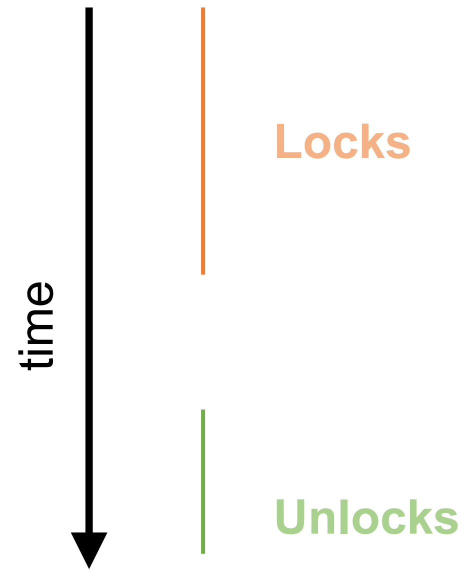
Locks Alone do not Enforce Serializability

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B), **READ**(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)
L1(B)	
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	

We used locks, but
this is a non-serializable schedule

Recap: Two-Phase Locking

In every TXN, all locks must come before any unlock



Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Recap: Non-recoverable Schedule

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	**L2**(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),**U2**(A)
.	**L2**(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),**U2**(B)
.	**COMMIT**
ROLLBACK	

Recap: Non-recoverable Schedule

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	**L2**(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),**U2**(A)
.	**L2**(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),**U2**(B)
.	**COMMIT**
ROLLBACK	

Dirty read

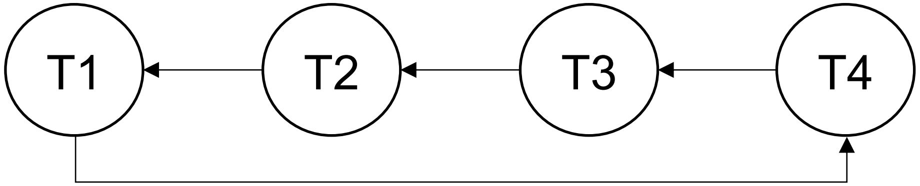
Recap: Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
…	…	…	…

Checking for deadlock:
- Construct the WAITS-FOR graph
- Check if it has a cycle

Checking for a cycle is fast (see CSE373), but it is very slow compared to the simple R/W operations

Abort a TXN



Summary

- Strict 2PL ensures conflict-serializable and recoverable schedules
- When the database is static (no insert/delete) then every conflict-serializable schedule is serializable
- When database is dynamic (has inserts/deletes) then it no longer holds because of fantoms (later)

Lock Types

Shared/Exclusive Locks

Reads don't conflict with each other.

- Exclusive/Write Lock → $\mathbf{X_i(A)}$
  - May read or write
  - No other locks may exist
- Shared/Read Lock → $\mathbf{S_i(A)}$
  - May only read
  - May exist with other shared locks
- Unlocked
  - No access

Shared/Exclusive Locks

…but another TXN holds this…

If a TXN requests this…

	unlocked	S	X
S	Yes	Yes	No
X	Yes	No	No

…then we do or don't grant permission

Discussion

- When TXN wants to read A, it requests S(A)
- If later it wants to write A, then it requests X(A)
- This is called lock escalation

More Discussion

Starvation:

- When a TXN waits for a lock, and never gets it
- Usually prevented by placing TXN in a queue
- Need to pay more attention to S/X locks
  - Some TXNs hold an S lock
  - One TXN requests X lock and waits
  - But more TXNs arrive and requests S locks, granted
  - Solution: stop granting S locks when X requests exists

More chances of deadlocks (next)

S/X Locks May Lead Easier to Deadlocks

T1	T2
S1(A), **READ**(A, t)	
t := t+100	
	S2(A), **READ**(A, s)
	s := s*2
X1(A)	
WRITE(A, t)	**X2**(A)
	WRITE(A,s)

Denied (X1(A))

Denied (X2(A))

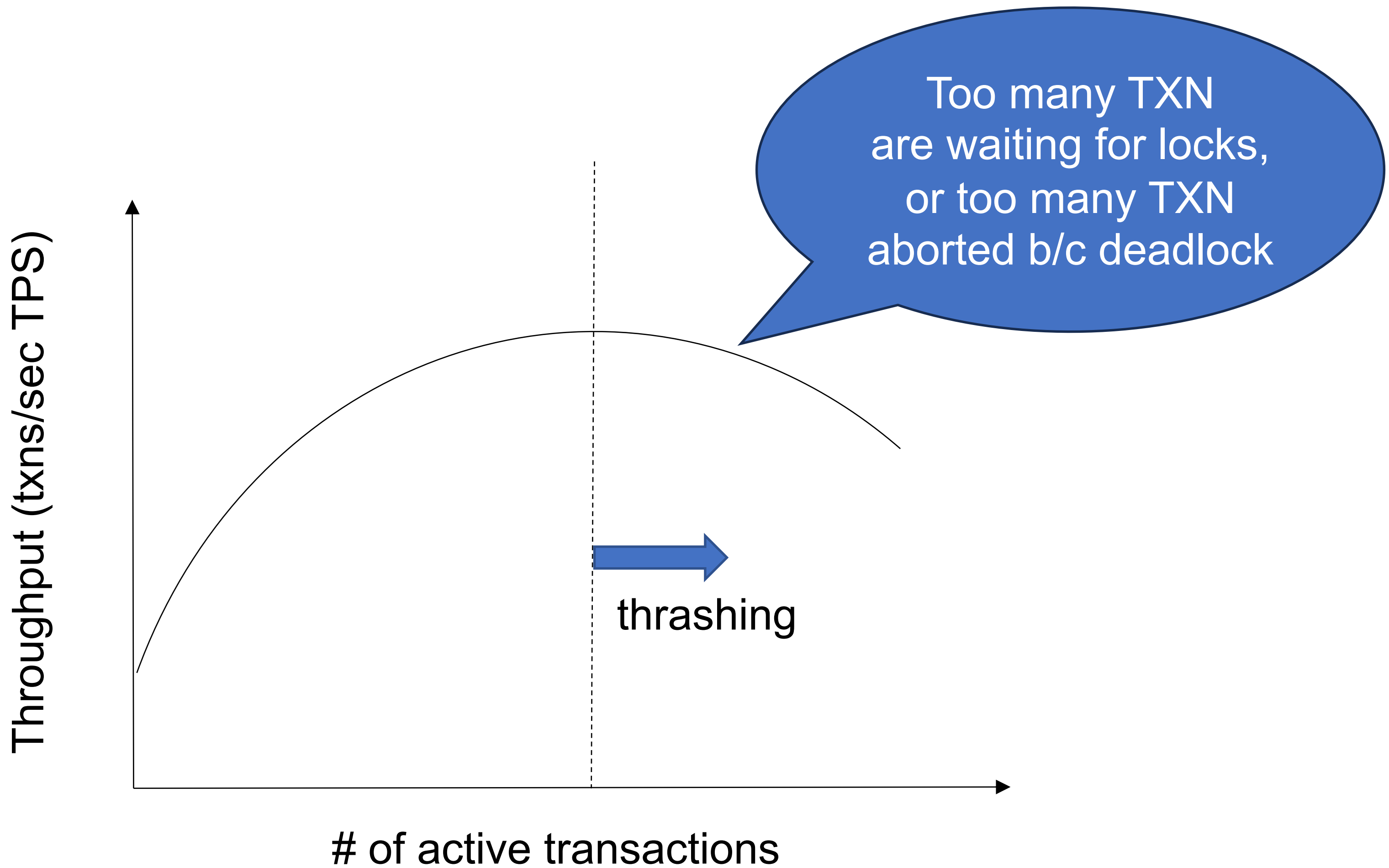
Dealock

Discussion

Enforcing ACID properties slows down the RDBMs

- Concurrency (I): need to wait, need to abort
- Recovery (A): need to double write to the log

Thrashing


Too many TXN
are waiting for locks,
or too many TXN
aborted b/c deadlock
Throughput (txns/sec TPS)
thrashing
# of active transactions

Discussion

- Isolated, atomic TXN usually incurs a high cost
- Performance is measured in TXN/sec (TPS) https://www.tpc.org/default5.asp
  - 1,000-10,000 is OK
  - 10,000-100,000 is AMAZING
  - 100,000-1,000,000 research papers only…
- For higher TPS use weaker isolation levels, which allow for some conflicts

Conflicts Between Concurrent Operations

Common Concurrency Conflicts

- Dirty/Inconsistent Read
- Lost Update
- Unrepeatable Read
- Phantom Read

These never happen in serializable schedules, but may happen in weaker levels of isolation

Dirty/Inconsistent Read

- **Dirty/Inconsistent Read**
- Lost Update
- Unrepeatable Read
- Phantom Read

Dirty read reading data of uncommitted TXN
a.k.a. inconsistent read

time

Manager wants to balance project budgets

CEO wants to check company balance

Dirty/Inconsistent Read

- **Dirty/Inconsistent Read**
- Lost Update
- Unrepeatable Read
- Phantom Read

Dirty read reading data of uncommitted TXN
a.k.a. inconsistent read

time

Manager wants to balance project budgets	*CEO wants to check company balance*
	SELECT SUM(money) …
-$10mil from project A	
+$7mil to project B	
+$3mil to project C	

Dirty/Inconsistent Read

- **Dirty/Inconsistent Read**
- Lost Update
- Unrepeatable Read
- Phantom Read

Dirty read reading data of uncommitted TXN
a.k.a. inconsistent read

time

Manager wants to balance project budgets

-$10mil from project A

+$7mil to project B

+$3mil to project C

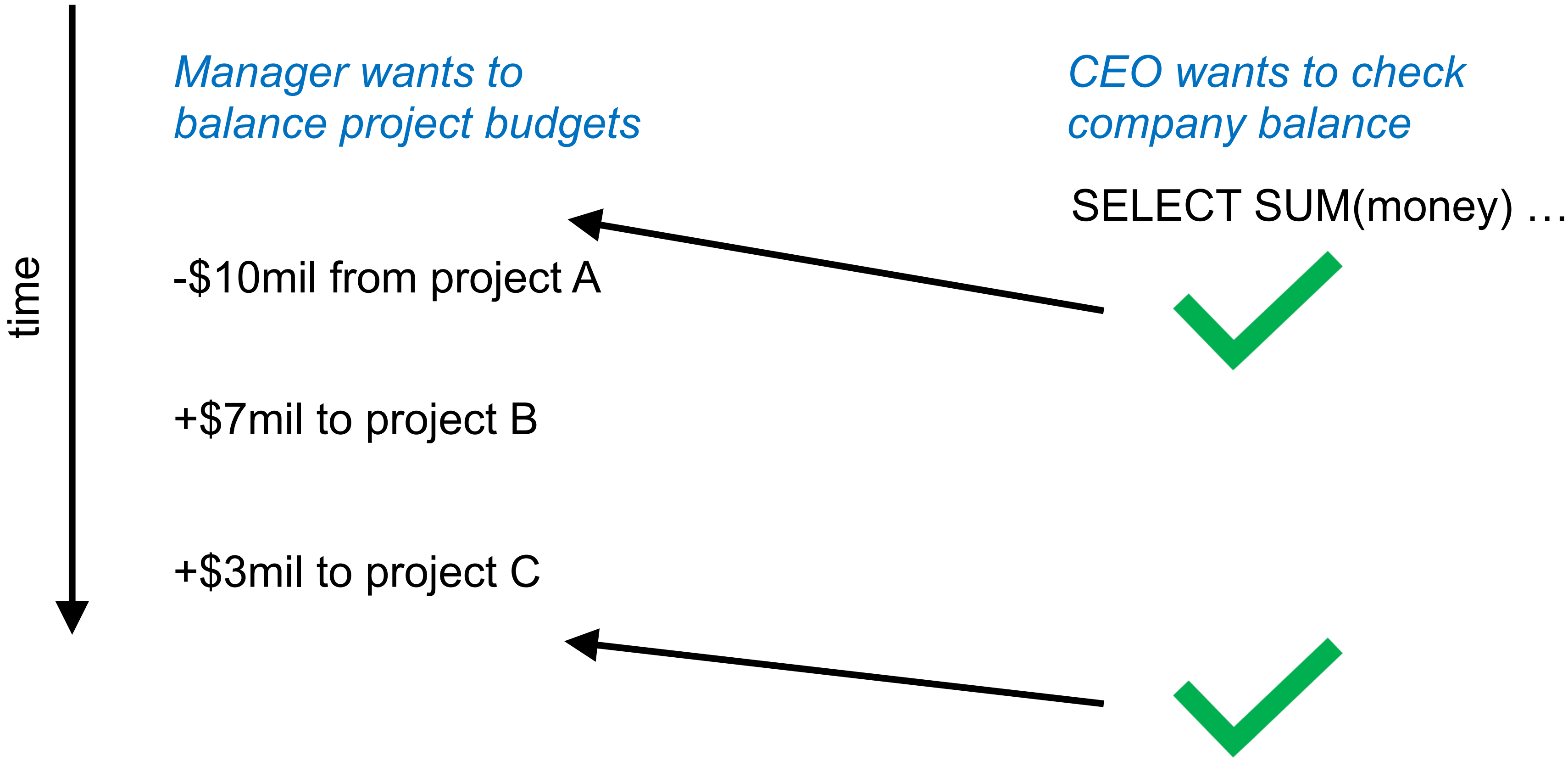
CEO wants to check company balance

SELECT SUM(money) …

Dirty/Inconsistent Read

- **Dirty/Inconsistent Read**
- Lost Update
- Unrepeatable Read
- Phantom Read

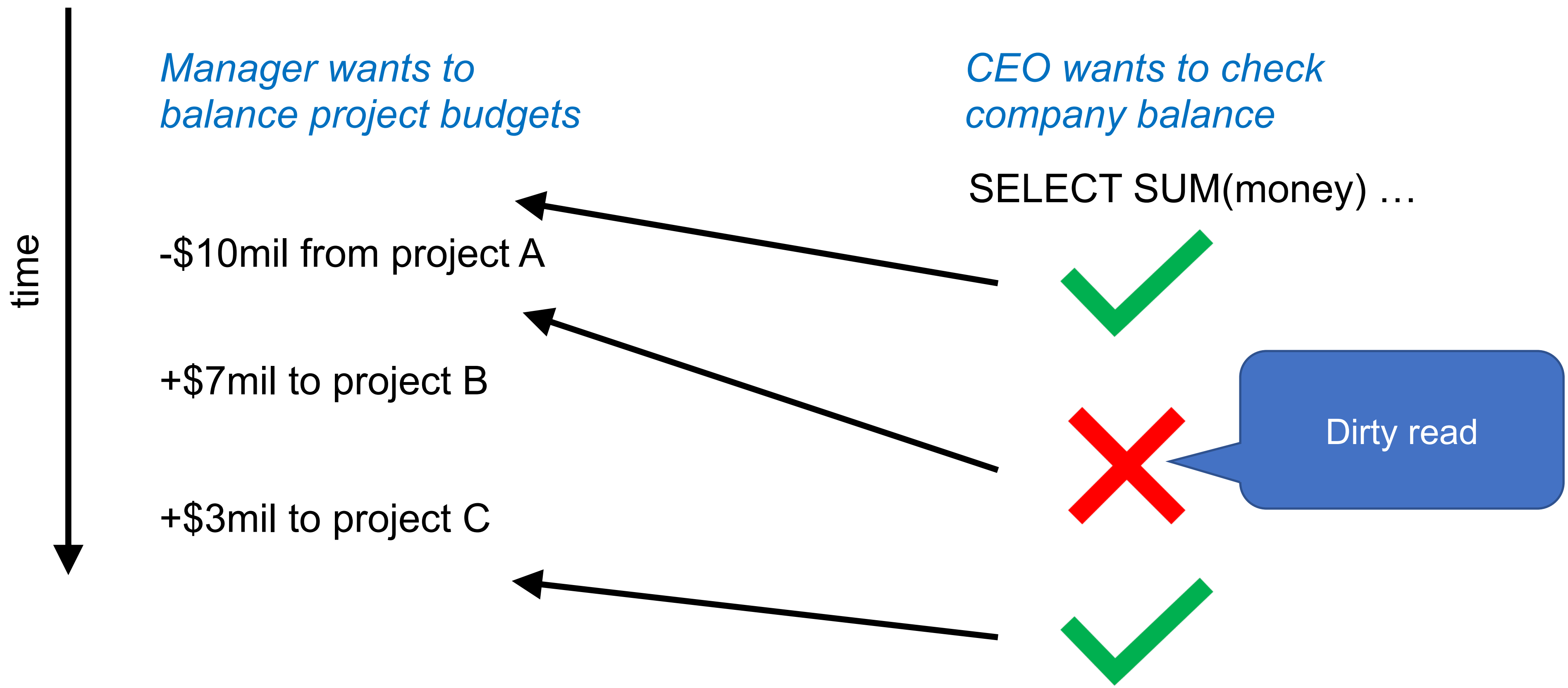
Dirty read reading data of uncommitted TXN
a.k.a. inconsistent read



Dirty/Inconsistent Read

- **Dirty/Inconsistent Read**
- Lost Update
- Unrepeatable Read
- Phantom Read

Dirty read reading data of uncommitted TXN
a.k.a. inconsistent read



Lost Update

A **lost update** happens when a write is overwritten by another TXN

- Dirty/Inconsistent Read
- **Lost Update**
- Unrepeatable Read
- Phantom Read

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Lost Update

- Dirty/Inconsistent Read
- **Lost Update**
- Unrepeatable Read
- Phantom Read

A **lost update** happens when a write is overwritten by another TXN

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
Set account 2 = 0	

Lost Update

- Dirty/Inconsistent Read
- **Lost Update**
- Unrepeatable Read
- Phantom Read

A **lost update** happens when a write is overwritten by another TXN

Account 1 = 100, Account 2 = 100

time

User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
Set account 2 = 0	
	Set account 2 = 200
	Set account 1 = 0

Lost Update

- Dirty/Inconsistent Read
- **Lost Update**
- Unrepeatable Read
- Phantom Read

A **lost update** happens when a write is overwritten by another TXN

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
Set account 2 = 0	
	Set account 2 = 200
	Set account 1 = 0

At end: Account 1 = 0, Account 2 = 200

Lost Update

A **lost update** happens when a write is overwritten by another TXN

- Dirty/Inconsistent Read
- **Lost Update**
- Unrepeatable Read
- Phantom Read

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Lost update

Set account 1 = 200

Set account 2 = 200

Set account 2 = 0

Set account 1 = 0

At end: Account 1 = 0, Account 2 = 0

Unrepeatable Read

- Dirty/Inconsistent Read
- Lost Update
- **Unrepeatable Read**
- Phantom Read

An **unrepeatable read** happens when data read twice differs

time

Accountant wants to check company assets	*Warehouse updates inventory levels*
SELECT inventory FROM Products WHERE pid = 1	
	UPDATE Products SET inventory = 0 WHERE pid = 1
SELECT inventory*price FROM Products WHERE pid = 1	

Unrepeatable Read

- Dirty/Inconsistent Read
- Lost Update
- **Unrepeatable Read**
- Phantom Read

An **unrepeatable read** happens when data read twice differs

time

Accountant wants to check company assets

Warehouse updates inventory levels

```
SELECT inventory
FROM Products
WHERE pid = 1
```

```
UPDATE Products
SET inventory = 0
WHERE pid = 1
```

```
SELECT inventory*price
FROM Products
WHERE pid = 1
```

Second read of Products.inventory is different

Phantom Read

- Dirty/Inconsistent Read
- Lost Update
- Unrepeatable Read
- **Phantom Read**

A **phantom read** happens when
a record is inserted/delete during reads

time

Accountant wants to check company assets

```
SELECT *
FROM products
WHERE price < 10.00
```

Warehouse receives new products

```
INSERT INTO Products
VALUES ('nuts', 10, 8.99)
```

```
SELECT *
FROM products
WHERE price < 20.00
```

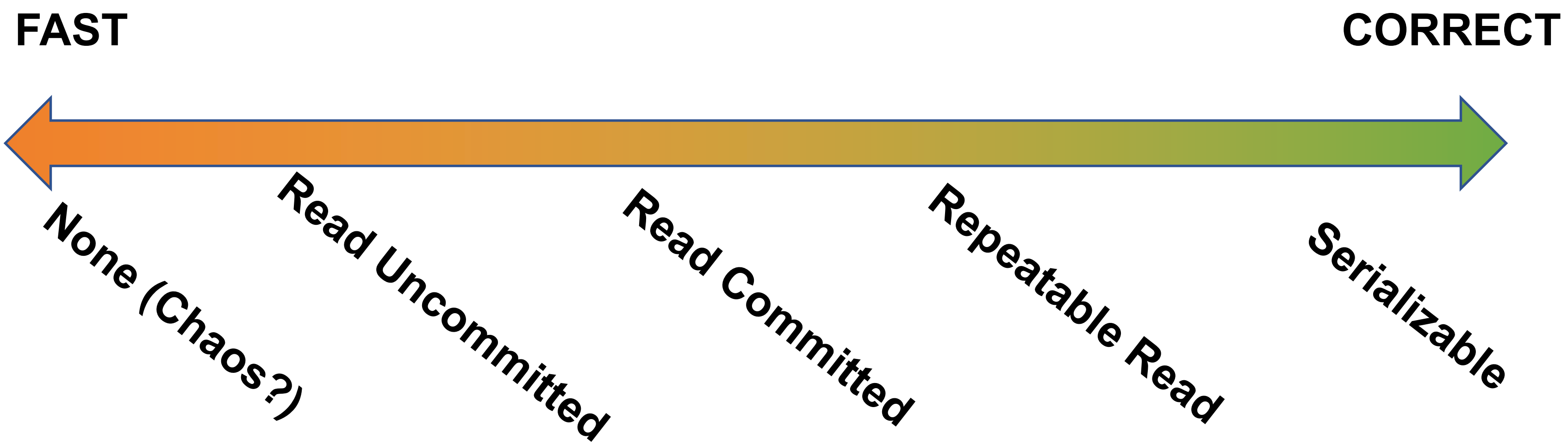
Returns a product
that should have been
in the first query

Weaker Isolation Levels

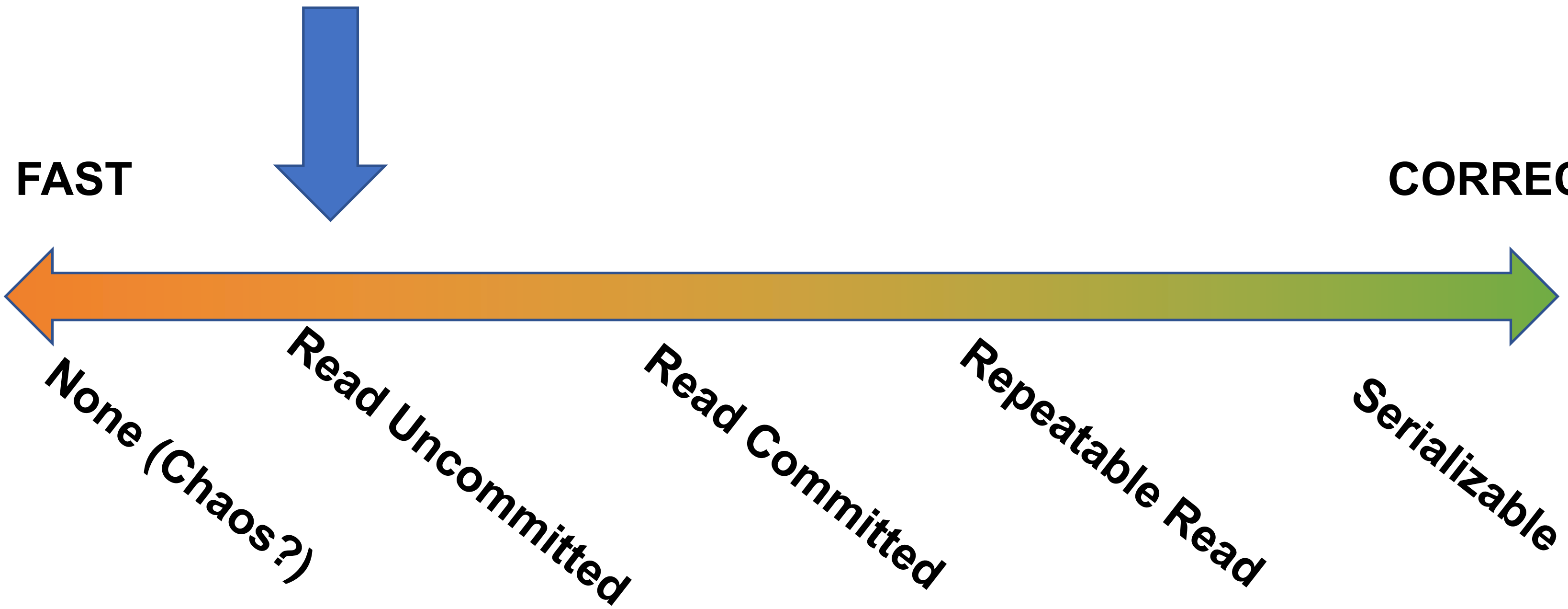
Isolation Levels

- **`SET TRANSACTION ISOLATION LEVEL …`**
  - **`READ UNCOMMITED`**
  - **`READ COMMITED`**
  - **`REPEATABLE READ`**
  - **`SERIALIZABLE`**
  - `SNAPSHOT ISOLATION` (MVCC)
- Default isolation level and configurability depends on the DBMS (read the docs)
- Serializable is often not the default

Isolation Level Design Spectrum



Isolation Level Design Spectrum



READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed
- Reads never wait! But dirty reads are possible

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed
- Reads never wait! But dirty reads are possible

Write lock obeys Strict 2PL

Read executes whenever

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed
- Reads never wait! But dirty reads are possible

Still possible to get isolated results, but you have to be "lucky" when a write operation is done

T1	T2
X(A) W(A)	
ABORT U(A)	
	R(A)
	COMMIT

T1	T2
	R(A)
	COMMIT
X(A) W(A)	
ABORT U(A)	

T1	T2
	R(A)
X(A) W(A)	
ABORT U(A)	
	COMMIT

READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed
- Reads never wait! But dirty reads are possible

Still possible to get isolated results, but you have to be "lucky" when a write operation is done

T1	T2
X(A) W(A)	
ABORT U(A)	
	R(A)
	COMMIT

Serial

T1	T2
	R(A)
	COMMIT
X(A) W(A)	
ABORT U(A)	

Serial

T1	T2
	R(A)
X(A) W(A)	
ABORT U(A)	
	COMMIT

Serializable (lucky!)

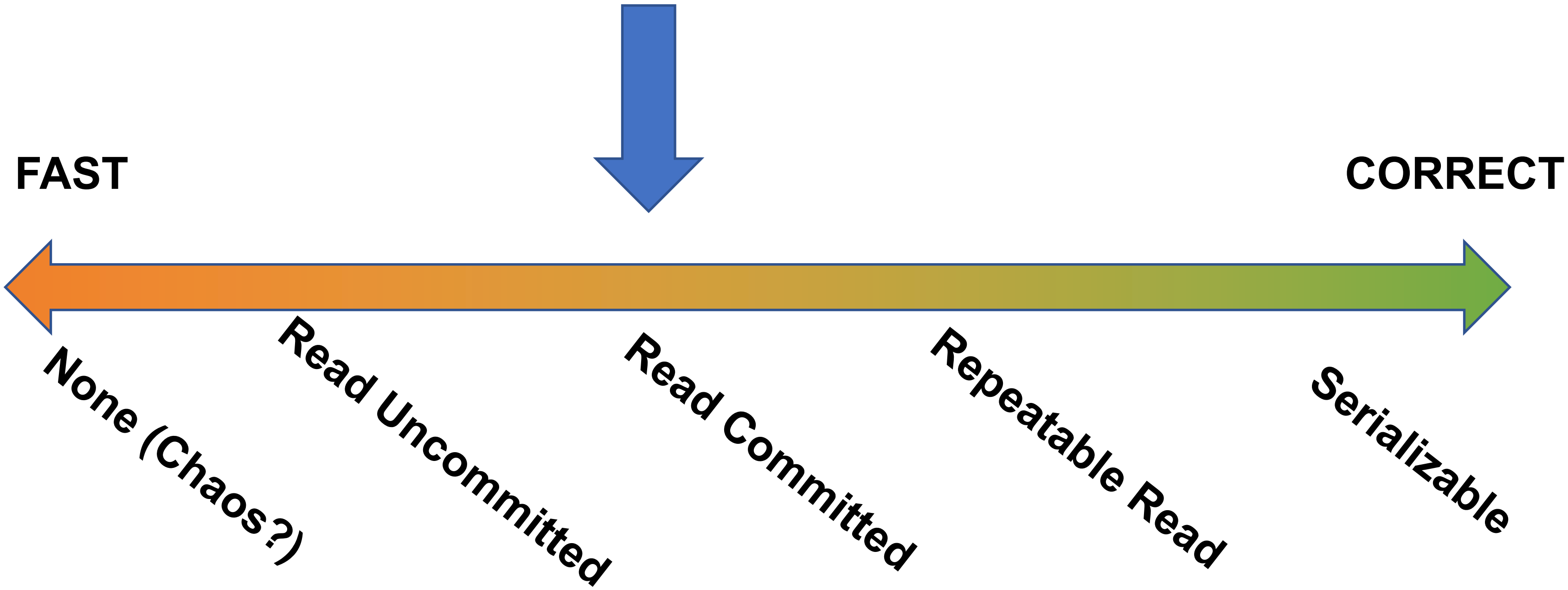
READ UNCOMMITTED

Extremely fast READ due to zero lock management overhead

Use cases:

- Static data (few or no writes after data initialization)
- Read coverage/accuracy is not mission critical

Isolation Level Design Spectrum



READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads. But non-repeatable reads possible.

READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads. But non-repeatable reads possible.

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads. But non-repeatable reads possible.

A dirty read could only happen if a read occurs after a write and before a COMMIT/ROLLBACK

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads. But non-repeatable reads possible.

A dirty read could only happen if a read occurs <u>after</u> a write and <u>before</u> a COMMIT/ROLLBACK

T1	T2
X(A) W(A)	
	~~R(A)~~
	~~COMMIT~~
ABORT U(A)	

→

T1	T2
X(A) W(A)	
	S(A) blocked…
ABORT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads.
  But non-repeatable reads possible.

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	X(A)
	W(A)
	COMMIT U(A)

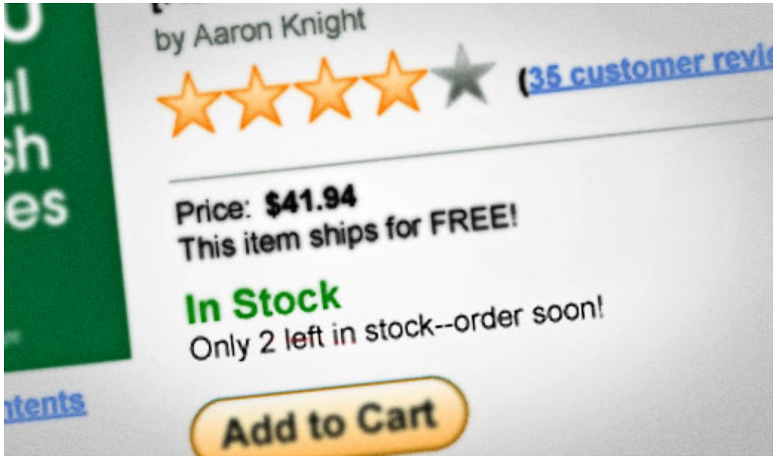
READ COMMITTED

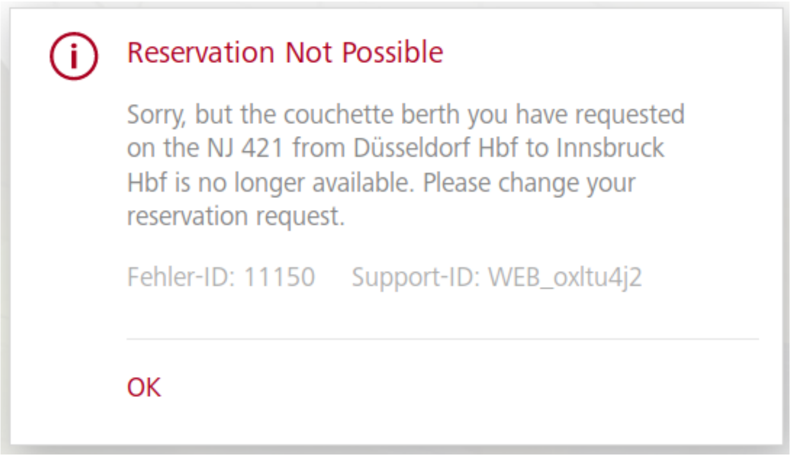
- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads.
  But non-repeatable reads possible.

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	X(A)
	W(A)
	COMMIT U(A)

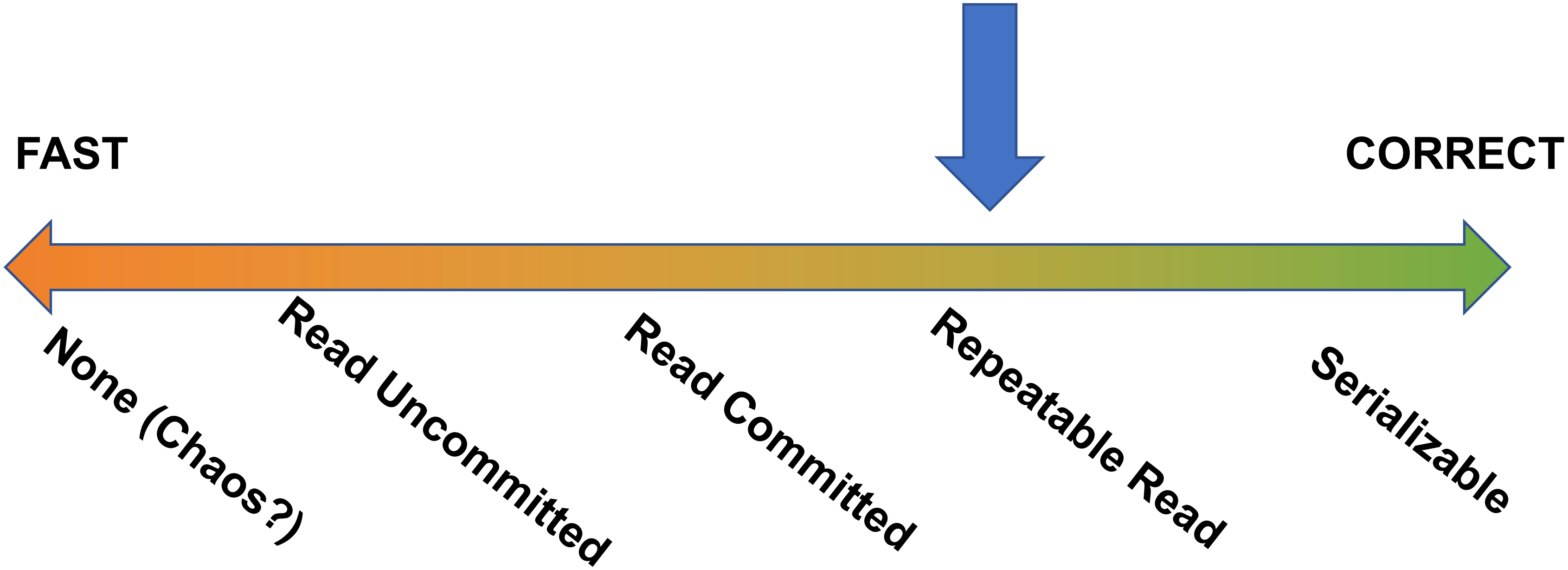
READ COMMITTED

- Fast READ since operation happens as soon as write txns are done
- Use cases:
  - Guarantee that read result is valid at some point
  - Often useful for e-commerce situations
    - Guarantee customer has good info to start with but doesn't block other customers from purchasing





Isolation Level Design Spectrum



REPEATABLE READ

- Writes → Strict 2PL write locks
- Reads → Strict 2PL read locks
- Unrepeatable reads are prevented

REPEATABLE READ

- Writes → Strict 2PL write locks
- Reads → Strict 2PL read locks
- Unrepeatable reads are prevented

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

REPEATABLE READ

- Writes → Strict 2PL write locks
- Reads → Strict 2PL read locks
- Unrepeatable reads are prevented

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

→

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…	**R(A)**
…granted X(A)	COMMIT U(A)
W(A)	
COMMIT U(A)	

REPEATABLE READ

- Writes → Strict 2PL write locks
- Reads → Strict 2PL read locks
- Unrepeatable reads are prevented

Conflict serializable!

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

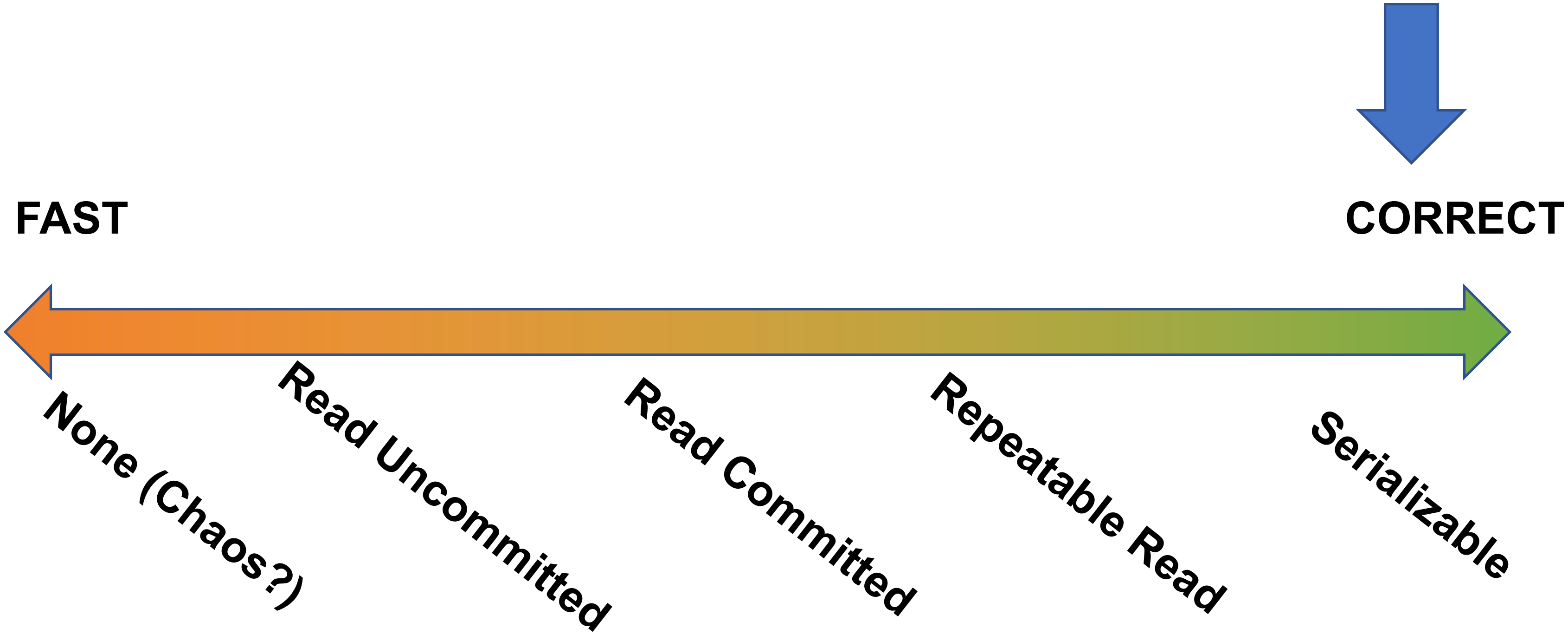
→

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…	**R(A)**
…granted X(A)	COMMIT U(A)
W(A)	
COMMIT U(A)	

REPEATABLE READ

- Ensures conflict serializability
- Recall: if the database is static (no insert/delete) then conflict serializability implies serializability
- Use cases: few insert/deletes

Isolation Level Design Spectrum



The Phantom Menace

- Same read has more rows
- Asset checking scenario:

time

Accountant wants to check company assets

Warehouse catalogs new products

SELECT *
FROM products
WHERE price < 10.00

INSERT INTO Products
VALUES (‘nuts’, 10, 8.99)

SELECT *
FROM products
WHERE price < 20.00

Phantom Reads

- Conflict serializability does not prevent phantoms.

These are the SQL queries

SELECT * FROM Table;

INSERT INTO Table VALUES (C…);

SELECT * FROM Table;

Phantom Reads

- Conflict serializability does not prevent phantoms.

These are the SQL queries

And this is how we modeled the TXNs using R/W to elements

	T1	T2	
SELECT * FROM Table;	R(A)		
	R(B)		
		I(C)	INSERT INTO Table VALUES (C…);
SELECT * FROM Table;	R(A)		
	R(B)		
	R(C)		

Phantom Reads

- Conflict serializability does not prevent phantoms.

	T1	T2	
SELECT * FROM Table;	R(A)		
	R(B)		
		I(C)	INSERT INTO Table VALUES (C…);
SELECT * FROM Table;	R(A)		
	R(B)		
	R(C)		

Phantom Reads

- Conflict serializability does not prevent phantoms.

A conflict-serializable schedule!

	T1	T2	
SELECT * FROM Table;	R(A)		
	R(B)		
		I(C)	INSERT INTO Table VALUES (C…);
SELECT * FROM Table;	R(A)		
	R(B)		
	R(C)		

Phantom Reads

- Conflict serializability does not prevent phantoms.

A conflict-serializable schedule!

What is the equivalent serial schedule?

	T1	T2	
SELECT * FROM Table;	R(A)		
	R(B)		
		I(C)	INSERT INTO Table VALUES (C…);
SELECT * FROM Table;	R(A)		
	R(B)		
	R(C)		

Recap

In a static database:

- Conflict serializability implies serializability

In a dynamic database:

- This no longer holds: we need to handle phatoms

SERIALIZABLE Level

- Write Lock → Strict 2PL
- Read Lock → Strict 2PL
- Locks on tables to handle phantom problem

SERIALIZABLE Level

- Write Lock → Strict 2PL
- Read Lock → Strict 2PL
- Locks on tables to handle phantom problem

T1	T2
R(A)	
R(B)	
	I(C)
R(A)	
R(B)	
R(C)	

SERIALIZABLE Level

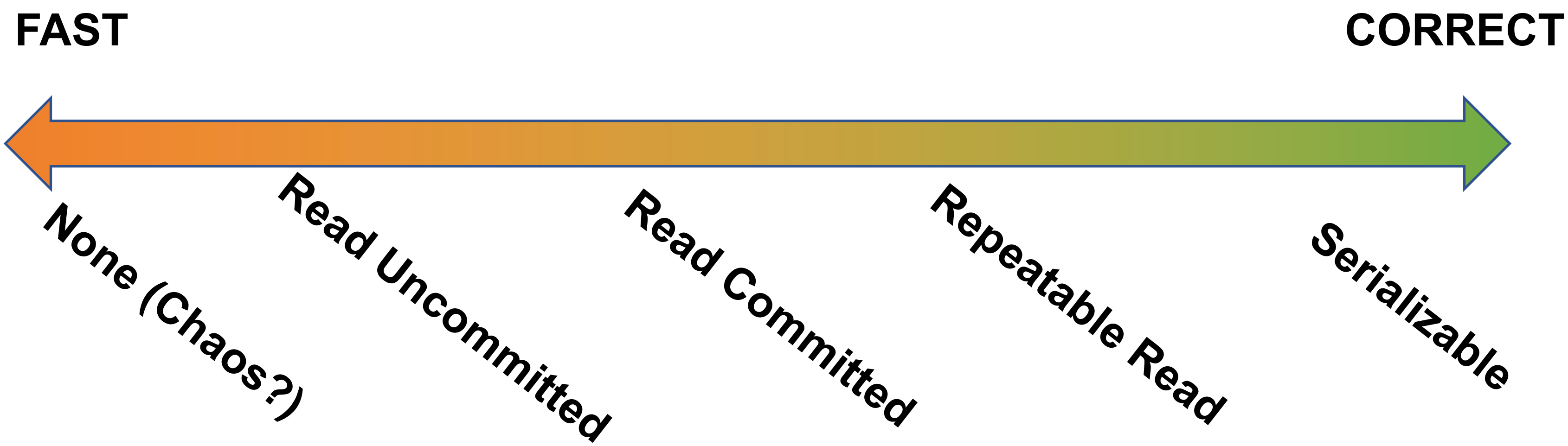
- Write Lock → Strict 2PL
- Read Lock → Strict 2PL
- Locks on tables to handle phantom problem

T1	T2
R(A)	
R(B)	
	I(C)
R(A)	
R(B)	
R(C)	

Change element granularity to Table →

T1	T2
S(T)	
R(T)	
	X(T) blocked…
R(T)	…
COMMIT U(T)	…granted X(T)
	W(T)
	COMMIT U(T)

Summary


FAST
CORRECT
None (Chaos?)
Read Uncommitted
Read Committed
Repeatable Read
Serializable