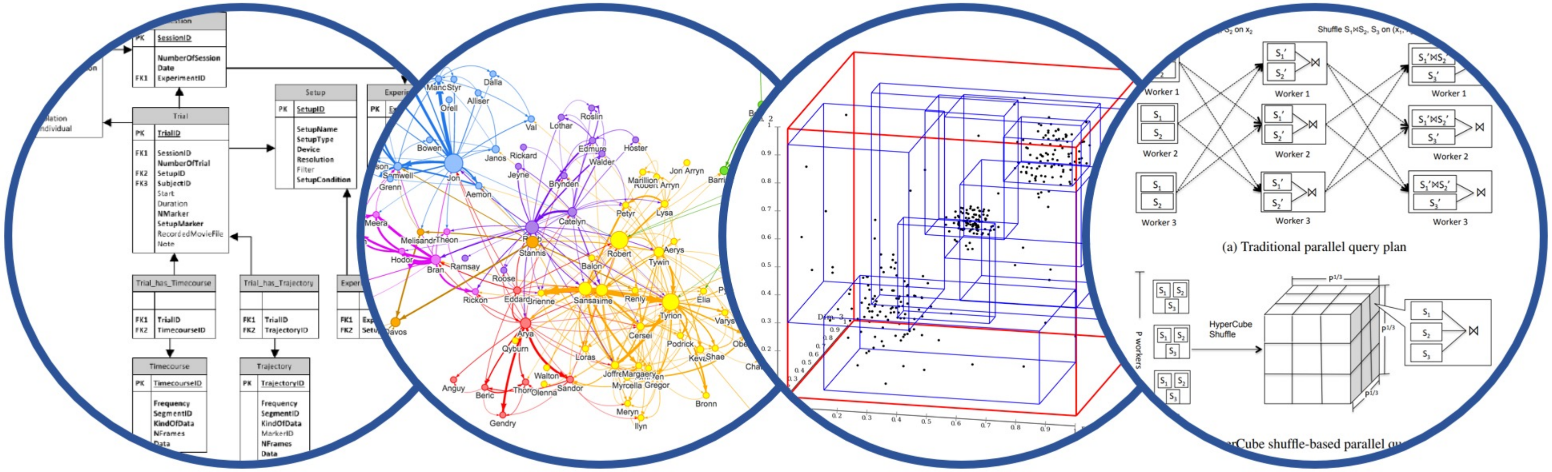


Introduction to Data Management

Transactions: Isolation Levels

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

Announcements

- HW5 is due on Friday

- HW6 has two parts:
  - Part 1 due 5/17. No late days (for quick feedback)
  - Part 2 due 5/24. Much more work than part 1

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

Discussion

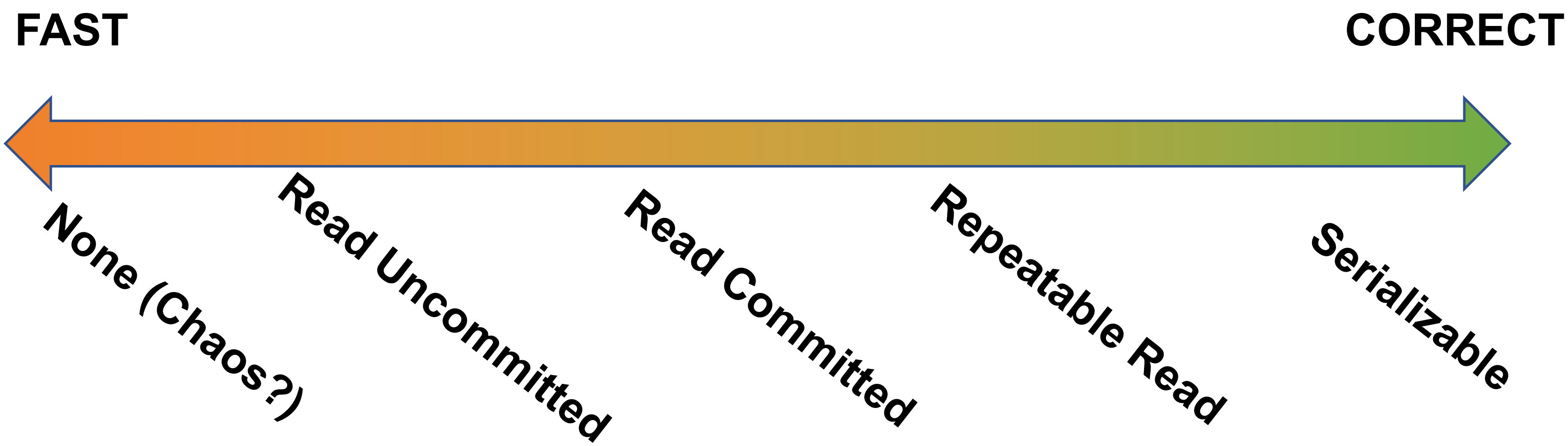
- TXNs slow down the DBMS significantly
- Performance is measured in TXN/sec (TPS) https://www.tpc.org/default5.asp
  - 1,000-10,000 is OK
  - 10,000-100,000 is AMAZING
  - 100,000-1,000,000 research papers only…
- For higher TPS use weaker isolation levels, which allow for some conflicts

Weaker Isolation Levels

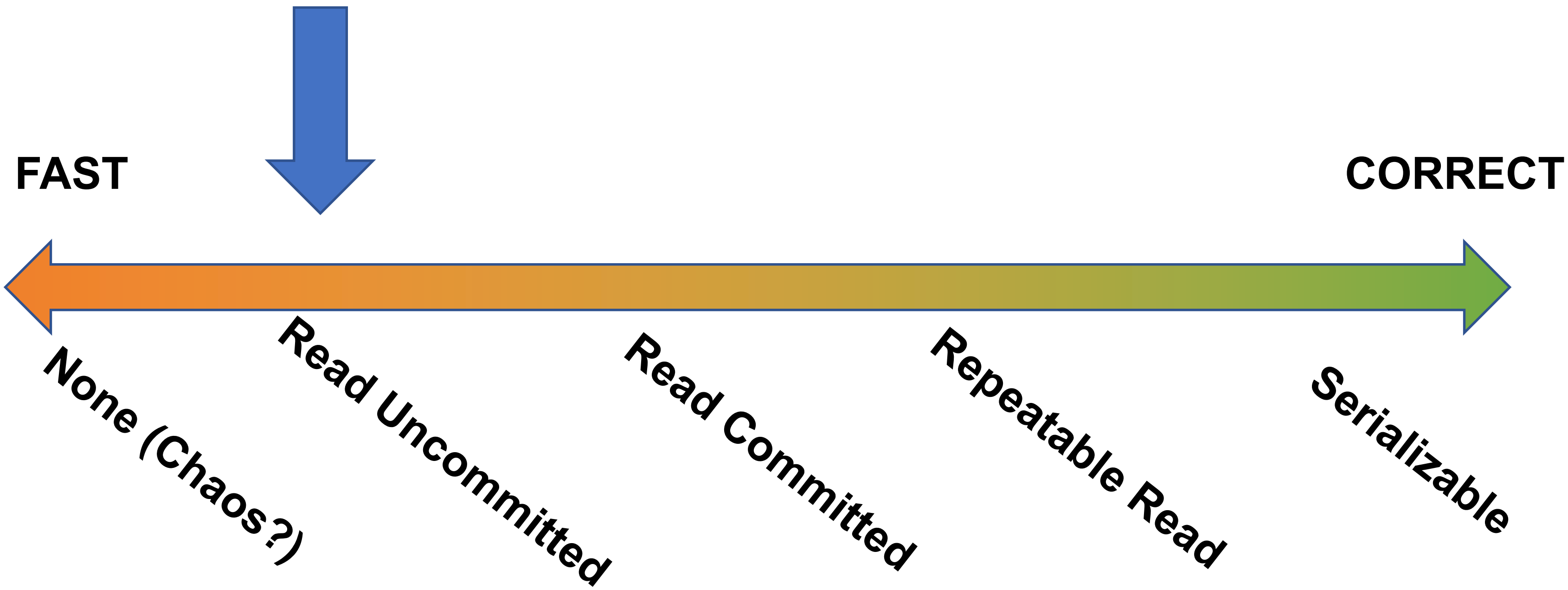
Isolation Levels

- **`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

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

Serial

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

Serial

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

Serializable (lucky!)

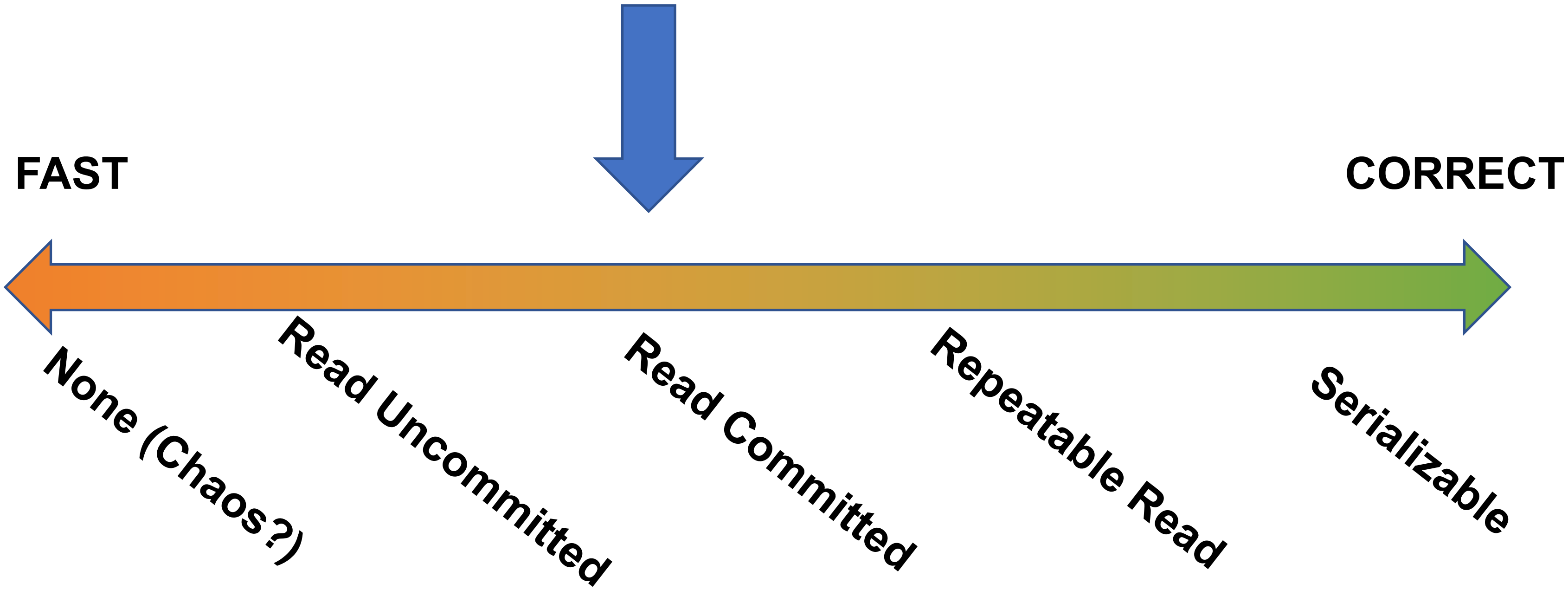
READ UNCOMMITTED

Reads never wait

Use cases:

- Static data (few or no writes after data initialization)
- Read 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 <u>after</u> a write and <u>before</u> 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 after
a write and before 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)

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…	
…	

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)**
	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)

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…

Wants to read again

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)	

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)

READ COMMITTED

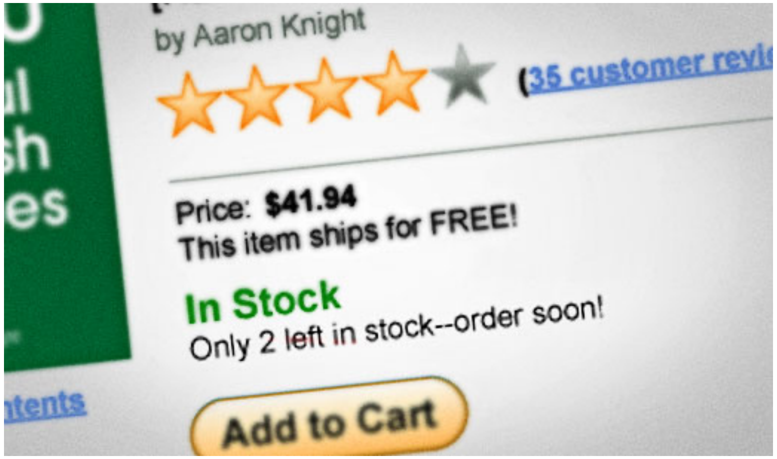
- Writes → Strict 2PL write locks
- Reads → Short-duration read locks
  - Acquire lock right before, release right after (not 2PL)
- No dirty reads.
  But non-repeatable reads possible.

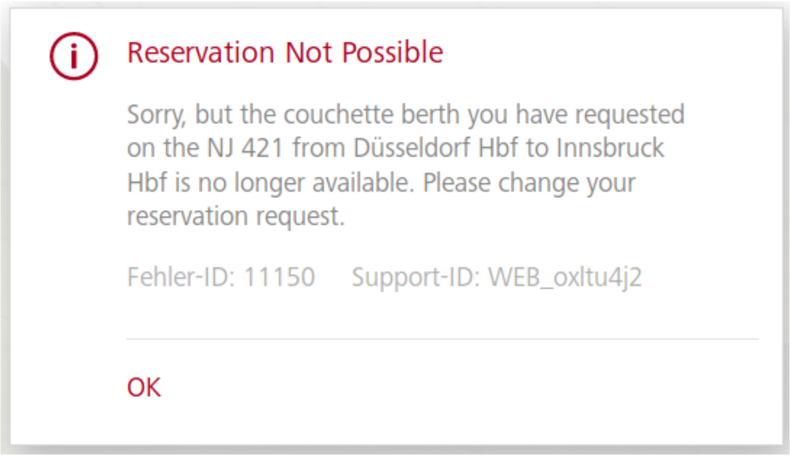
T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	…
	COMMIT

Second Read different value

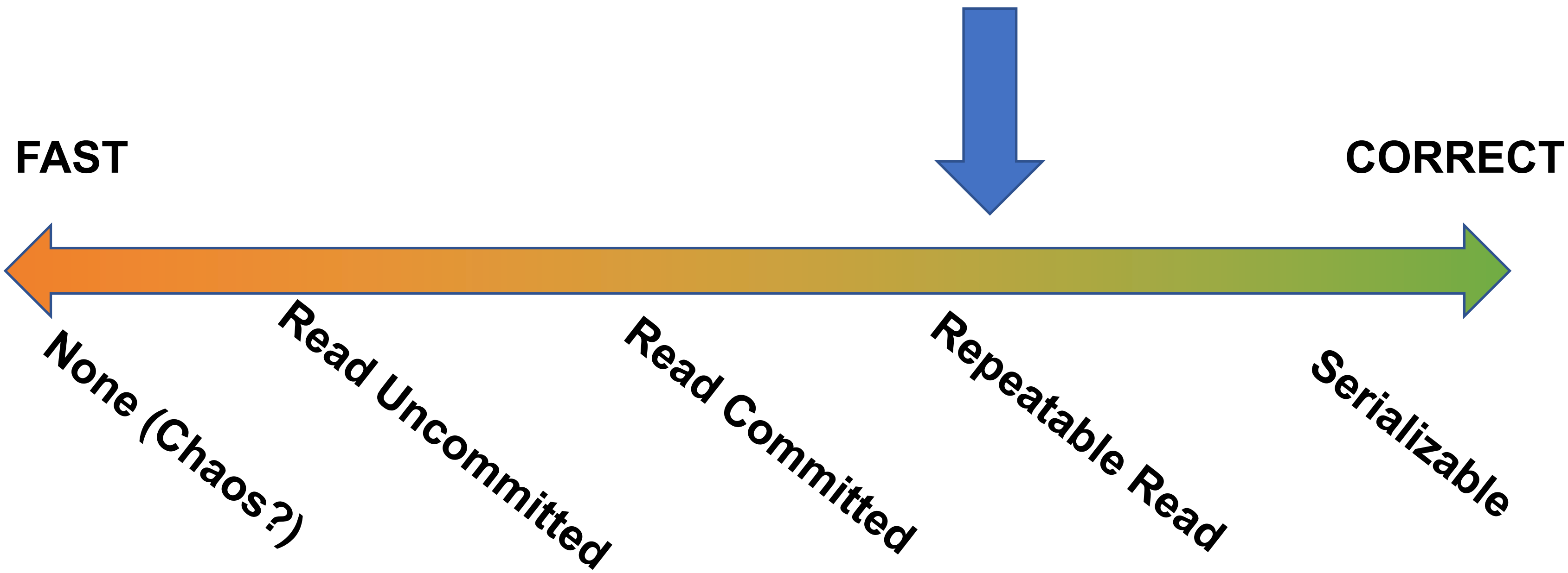
READ COMMITTED

- 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!

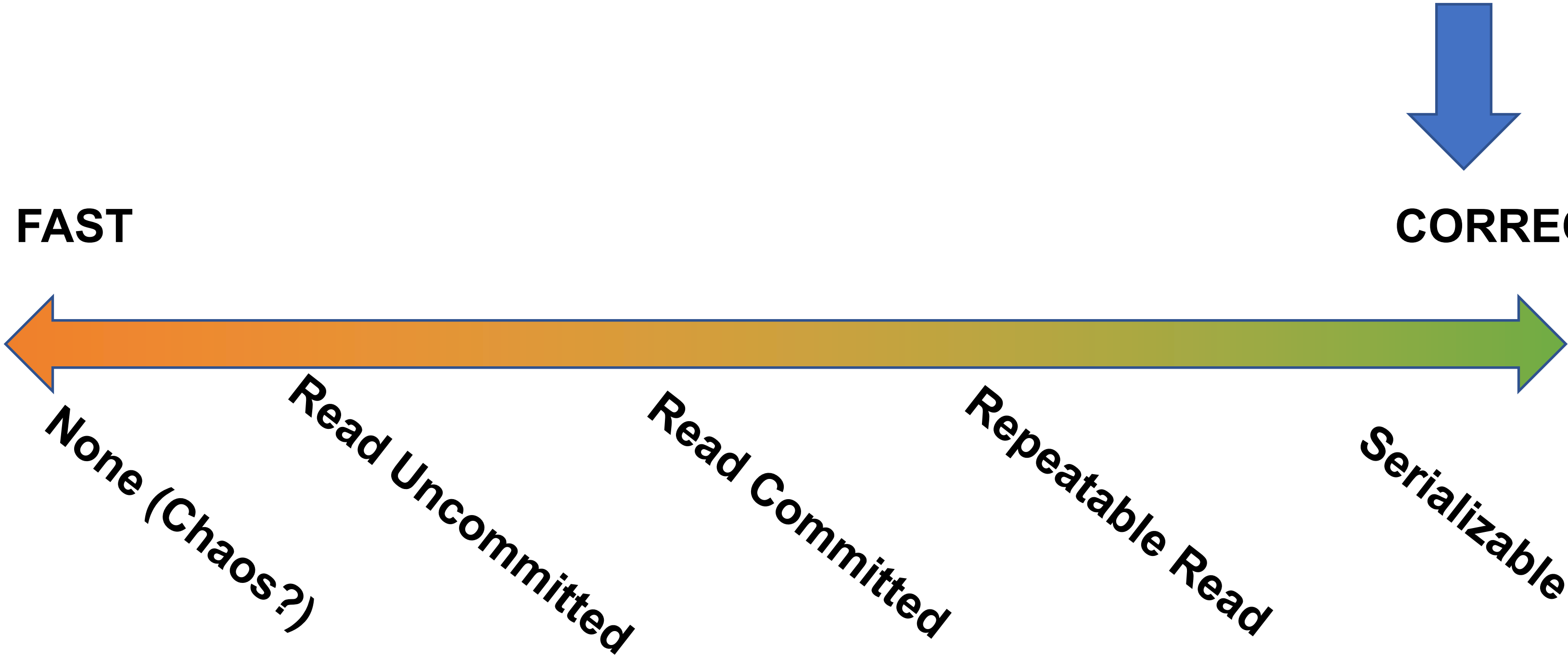
T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

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: in a static database (no insert/delete) conflict serializability implies serializability
- Use cases: consistency is mission critical

Isolation Level Design Spectrum



The Phantom Menace

- Same read has more rows
- Asset checking scenario:

time

Accountant wants to check company assets

SELECT *
FROM products
WHERE price < 10.00

SELECT *
FROM products
WHERE price < 20.00

Warehouse catalogs new products

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

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)		
		W(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)		
		W(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)		
		W(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)		
		W(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)		
		W(C)	INSERT INTO Table VALUES (C…);
SELECT * FROM Table;	R(A)		
	R(B)		
	R(C)		

Answer: T2, T1
(make sure you know why)

Phantom Reads

- Conflict serializability does not prevent phantoms.

”All models are wrong, some are useful*”

* George Box

A conflict-serializable schedule!

Modeling the DB as a set of elements is only useful for static databases.

	T1	T2	
SELECT * FROM Table;	R(A)		
	R(B)		
		W(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

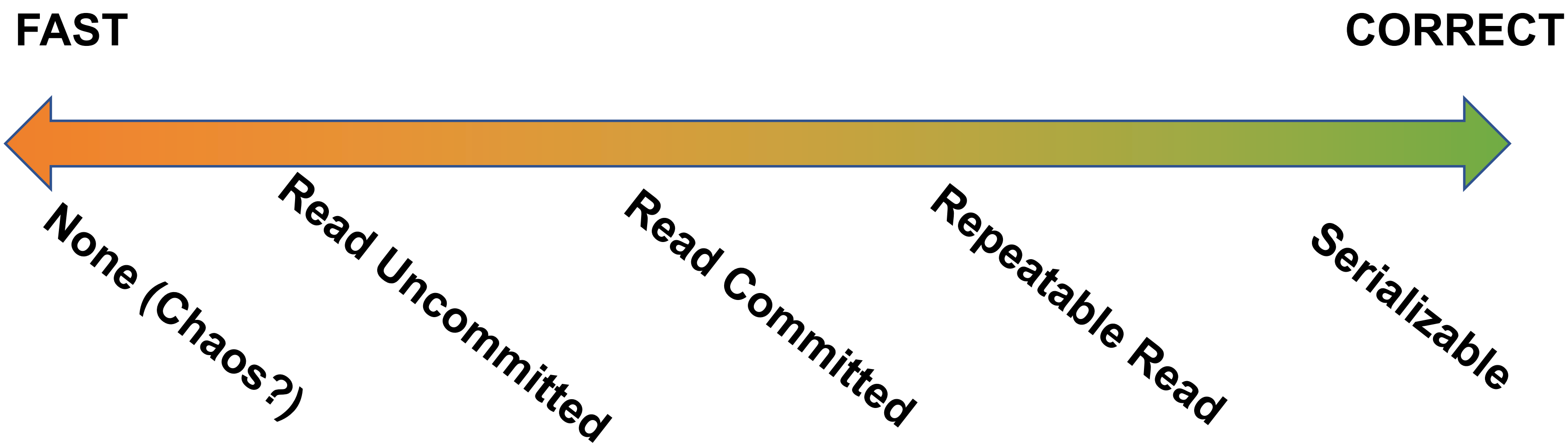
- Write Lock → Strict 2PL
- Read Lock → Strict 2PL
- Locks on tables to handle phantom problem

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

Practical Aspects of TXN

Rule of Thumb

Write the TXN as short as possible, but not shorter

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
…
Prompt user
  for input
…

COMMIT
```

NO

Rule of Thumb

Write the TXN as short as possible, but not shorter

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
…
Prompt user
  for input
…

COMMIT
```

Never!!

NO

Rule of Thumb

Write the TXN as short as possible, but not shorter

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
…
Prompt user
   for input
…

COMMIT
```

NO

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
COMMIT
…
Prompt user
   for input
BEGIN TRANSACTION
…

COMMIT
```

Rule of Thumb

Write the TXN as short as possible, but not shorter

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
…
Prompt user
   for input
…

COMMIT
```

NO

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
COMMIT
…
Prompt user
   for input
BEGIN TRANSACTION
…

COMMIT
```

YES

Rule of Thumb

Write the TXN as short as possible, but not shorter

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
…
Prompt user
   for input
…

COMMIT
```

NO

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
COMMIT
…
Prompt user
   for input
BEGIN TRANSACTION
…

COMMIT
```

YES

```
BEGIN TRANSACTION
…
Read(A)Write(A)
COMMIT
BEGIN TRANSACTION
Read(B)Write(B)
COMMIT
…
Prompt user
   for input
BEGIN TRANSACTION
…

COMMIT
```

Rule of Thumb

A,B to be updated in same TXN

Write the TXN as short as possible, but not shorter

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
…
Prompt user
  for input
…

COMMIT
```

NO

```
BEGIN TRANSACTION
…
Read(A)Write(A)
Read(B)Write(B)
COMMIT
…
Prompt user
  for input
BEGIN TRANSACTION
…

COMMIT
```

YES

```
BEGIN TRANSACTION
…
Read(A)Write(A)
COMMIT
BEGIN TRANSACTION
Read(B)Write(B)
COMMIT
…
Prompt user
  for input
BEGIN TRANSACTION
…

COMMIT
```

NO

Autocommit

```
BEGIN TRANSACTION;
  INSERT …
  SELECT …
  …
COMMIT
```

v.s.

```
INSERT …

SELECT …

  …
```

Autocommit

```
BEGIN TRANSACTION;
  INSERT …
  SELECT …
  …
COMMIT
```

v.s.

```
INSERT …

SELECT …

 …
```

By default,
each statement
is one TXN

Autocommit

```
BEGIN TRANSACTION;
  INSERT …
  SELECT …
  …
COMMIT
```

v.s.

```
INSERT …

SELECT …

  …
```

By default, each statement is one TXN

In python:

We say here if we want autocommit

```
con = sqlite3.connect("bank.db", autocommit=True)
```

Case Study: SQLite

- Uses locks
- Element = entire database (!!!)
- Let’s see the details

http://www.sqlite.org/atomiccommit.html

Case Study: SQLite

- Sqlite reads data from the file on disk,…
- …updates it in main memory…
- …writes it back to disk at commit time

- Multiple users can access the same file…
- …and are coordinated via locks

Case Study: SQLite

Lock types

- READ LOCK (to read)
- RESERVED LOCK (to write)
- PENDING LOCK (wants to commit)
- EXCLUSIVE LOCK (to commit)

SQLite

Step 1: when a transaction begins

- Acquire a READ LOCK (aka "SHARED" lock)
- TXNs read data from file to local memory
- If the transaction commits without writing anything, then it simply releases the lock

T1	T2
S	
R(A)	
R(B)	S
	R(C)
R(C)	
	R(A)
…	…
	CO

SQLite

Step 2: when one transaction wants to write

- Acquire a RESERVED LOCK
- May coexists with READ LOCKs
- Writer TXN may write; in local memory!
- Reader TXN continue to read from the file
- New READ LOCKs accepted
- No other RESERVED LOCK allowed

T1	T2	T3	T4
R(A)		S	
R(B)	RES		
	W(B)		
	R(A)	R(B)	S
R(A)	W(A)		R(A)
		R(A)	

Step 2: when one transaction wants to write

- Acquire a RESERVED LOCK
- May coexists with READ LOCKs
- Writer TXN may write; in local memory!
- Reader TXN continue to read from the file
- New READ LOCKs accepted
- No other RESERVED LOCK allowed

T1	T2	T3	T4
R(A)		S	
R(B)	RES		
	W(B)		
	R(A)	R(B)	S
R(A)	W(A)		R(A)
		R(A)	

Update only
in local memory,
not on file

SQLite

Step 2: when one transaction wants to write

- Acquire a RESERVED LOCK
- May coexists with READ LOCKs
- Writer TXN may write; in local memory!
- Reader TXN continue to read from the file
- New READ LOCKs accepted
- No other RESERVED LOCK allowed

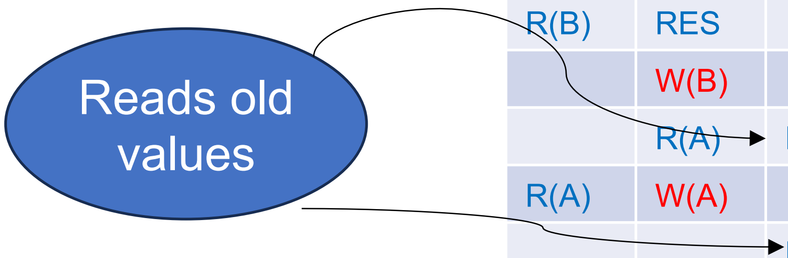
T1	T2	T3	T4
R(A)		S	
R(B)	RES		
	W(B)		
	R(A)	R(B)	S
R(A)	W(A)		R(A)
		R(A)	

SQLite

Step 2: when one transaction wants to write

- Acquire a RESERVED LOCK
- May coexists with READ LOCKs
- Writer TXN may write; in local memory!
- Reader TXN continue to read from the file
- New READ LOCKs accepted
- No other RESERVED LOCK allowed

T1	T2	T3	T4
R(A)		S	
R(B)	RES		
	W(B)		
	R(A)	R(B)	S
R(A)	W(A)		R(A)
		R(A)	



SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

T1	T2	T3	T4	T5
	W(B)			
R(A)	W(A)		R(B)	
		R(A)		
	CO?			
				S?

SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

T1	T2	T3	T4	T5
	W(B)			
R(A)	W(A)		R(B)	
		R(A)		
	CO?			
				~~S?~~

No new READ LOCK

SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

Why not write to disk right now?

T1	T2	T3	T4	T5
	W(B)			
R(A)	W(A)		R(B)	
		R(A)		
	CO?			
			R(A)	

SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

Why not write to disk right now?

T1	T2	T3	T4	T5
	W(B)			
R(A)	W(A)		R(B)	
		R(A)		
	CO?			
			R(A)	

This must be the old value for serializability

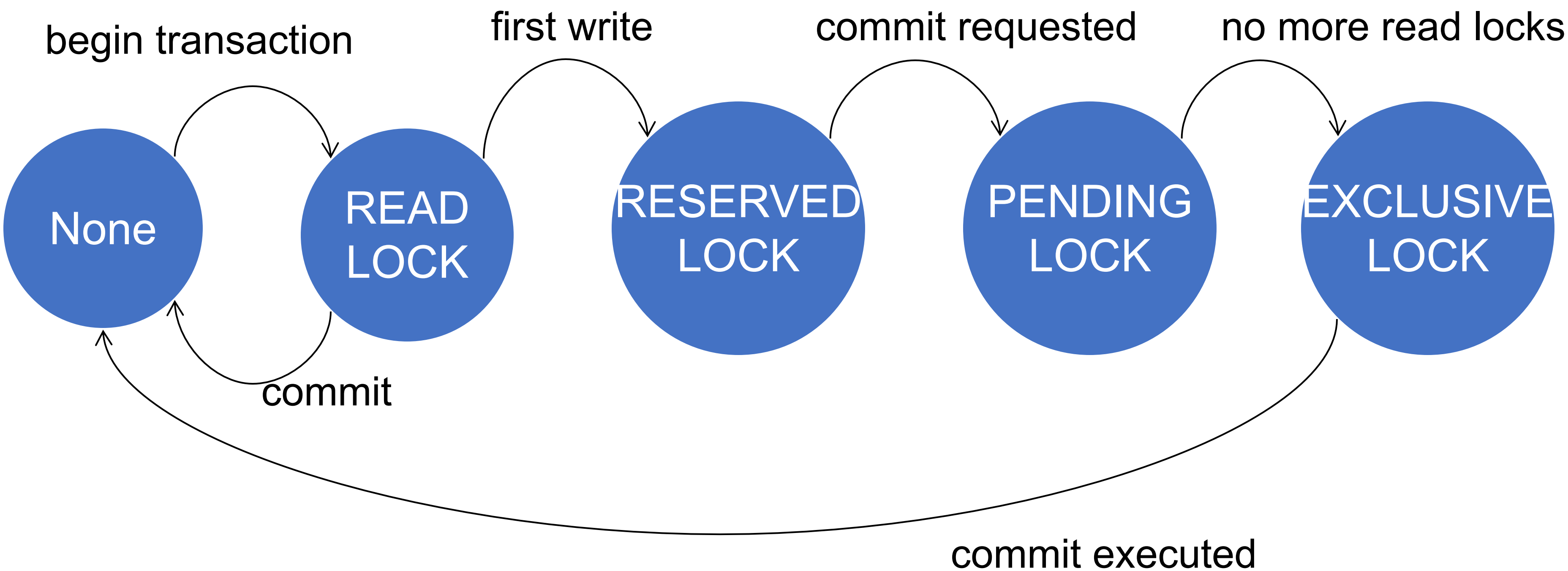
SQLite

Step 4: when all read locks have been released

- Acquire the EXCLUSIVE LOCK
- Nobody can touch the database now
- All updates are written permanently to file
- Release the lock and COMMIT

T1	T2	T3	T4	T5
	W(B)			
R(A)	W(A)		R(B)	
		R(A)		
	CO?			
			R(A)	
CO				
		CO	CO	
	CO			

SQLite


begin transaction
first write
commit requested
no more read locks
None
READ LOCK
RESERVED LOCK
PENDING LOCK
EXCLUSIVE LOCK
commit
commit executed

SQLite Demo

```
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);
```

Demonstrating Locking in SQLite

T1:

```
begin transaction;
select * from r;
-- T1 has a READ LOCK
```

T2:

```
begin transaction;
select * from r;
-- T2 has a READ LOCK
```

Demonstrating Locking in SQLite

T1:

```
update r set b=11 where a=1;
-- T1 has a RESERVED LOCK
```

T2:

```
update r set b=21 where a=2;
-- T2 asked for a RESERVED LOCK:  DENIED
```

Demonstrating Locking in SQLite

T3:

```
begin transaction;
select * from r;
commit;
-- everything works fine, could obtain READ LOCK
```

Demonstrating Locking in SQLite

T1:

commit;

-- SQL error: database is locked

-- T1 asked for PENDING LOCK -- GRANTED

-- T1 asked for EXCLUSIVE LOCK -- DENIED

Demonstrating Locking in SQLite

T3':

```
begin transaction;
select * from r;
-- T3 asked for READ LOCK-- DENIED (due to T1)
```

T2:

```
commit;
-- releases the last READ LOCK; T1 can commit
```