

Introduction to Data Management

Transaction: Isolation Levels

Alyssa Pittman

Based on slides by Jonathan Leang, Dan Suciu, et al

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

Recap

- Schedules under 2PL are conflict serializable
 - Locking phase □ unlocking phase
- Conflict serializable schedules follow the isolation principle of ACID
 - No dirty read (WR)
 - No unrepeatable read (RW)
 - No lost update (WW)
- Schedules under strict 2PL additionally provide recoverability
 - Locking phase □ unlock with commit or rollback
- Deadlocks can still occur

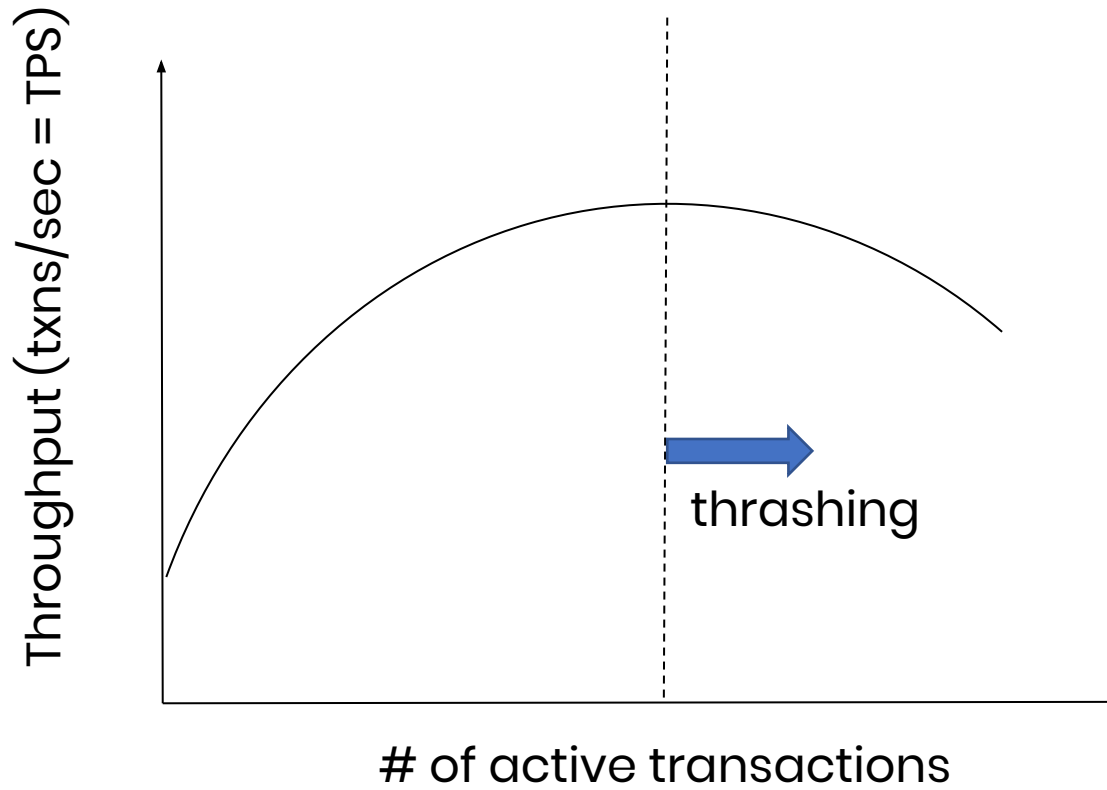
Outline

- Shared/Exclusive locks
- Isolation levels
- Phantom reads
- If time: SQLite Transactions demo

Practicality of Binary Locks

- Binary Locks ☐ full control or no control
- Leads to excessive deadlocking

Thrashing



Shared/Exclusive Locks

- Observation: Reads don't conflict with each other
- Simple 3-tier lock hierarchy:
 - Exclusive/Write Lock $\square \mathbf{x}_i(\mathbf{A})$
 - Full control
 - No other locks on A may exist concurrently
 - Shared/Read Lock $\square \mathbf{s}_i(\mathbf{A})$
 - Shared control
 - May exist concurrently with other shared locks on A
 - Unlocked

Lock Compatibility Matrix

Is the requested lock granted...

...if the element is in this state?

	unlocked	S	X
S	Yes	Yes	No
X	Yes	No	No

Practicality of Serializability

- **Easy to reason about**

- Application programming is easier under serializability assumptions

- **Expensive to use**

- Slow
- Resource intensive

- **Applications often don't need serializability**

- Application functionality may not depend on serializability
- Financial/User experience cost is low enough for tradeoff considerations

Isolation Levels

▪ SET TRANSACTION ISOLATION LEVEL ...

- **READ UNCOMMITTED**
- **READ COMMITTED**
- **REPEATABLE READ**
- **SERIALIZABLE**
- SNAPSHOT ISOLATION
- ...



Default for
SQL Server

ACID

- Default isolation level and configurability depends on the DBMS (read the docs)

READ UNCOMMITTED


- Writes ☐ Strict 2PL write locks
- Reads ☐ No locks needed
- Dirty reads are 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 before reading and release lock after (not 2PL)
- **Dirty reads are prevented**

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

- **Unrepeatable reads are possible**

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

T1	T2
X(A) blocked...	S(A)
	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
- **Phantom reads are possible...**

Phantom Reads

- Conflict serializability implying serializability assumes a **static database**
 - Conflicts only matter for the same element
 - Inserting a new element (tuple-level granularity) means that the conflict model no longer is able to encapsulate it

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

Phantom Reads

- **Dynamic database** serializability needs either:
 - Table locking (prevent insertions) or
 - Predicate locking (lock based on query filters)

SERIALIZABLE

- Write Lock ☐ Strict 2PL
- Read Lock ☐ Strict 2PL
- Plus predicate locks and/or table locks

Isolation Level Summary

	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	X	X	X
READ COMMITTED		X	X
REPEATABLE READ			X
SERIALIZABLE			



No anomalies!

Applying Transaction Logic

- Applications generally need to
 - **Check/Set isolation levels**
 - **Specify operations as transactions**
- Common mistakes/misconceptions:
 - You do not need to implement locking. The DBMS takes care of it.
 - You must **close all explicit transactions** with COMMIT or ROLLBACK. Not doing so will cause the application to hang (wait due to unfinished locking).

Isolation Levels in Practice

“In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions.”

- [ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications](#)

Isolation Levels in Practice

- Level-based isolation anomalies
 - Using the wrong isolation level
- Scoping isolation anomalies
 - Encapsulating transactions incorrectly

Isolation Levels in Practice

- Level-based isolation anomalies
 - Using the wrong isolation level
- Scoping isolation anomalies
 - Encapsulating transactions incorrectly

```
1 def withdraw(amt, user_id): (a)
2   bal = readBalance(user_id)
3   if (bal >= amt):
4     writeBalance(bal - amt, user_id)
```

```
1 def withdraw(amt, user_id): (b)
2   beginTxn()
3   bal = readBalance(user_id)
4   if (bal >= amt):
5     writeBalance(bal - amt, user_id)
6   commit()
```

Scope problem
- someone
could
withdraw
money before
we write

Isolation Levels in Practice

- Level-based isolation anomalies
 - Using the wrong isolation level
- Scoping isolation anomalies
 - Encapsulating transactions incorrectly

```
1 def withdraw(amt, user_id): (a)
2   bal = readBalance(user_id)
3   if (bal >= amt):
4     writeBalance(bal - amt, user_id)
```

```
1 def withdraw(amt, user_id): (b)
2   beginTxn()
3   bal = readBalance(user_id)
4   if (bal >= amt):
5     writeBalance(bal - amt, user_id)
6   commit()
```

Potential level problem - at levels \leq READ_COMMITTED, the balance can still change

Isolation Levels in Practice

The researchers used the web UIs of various sites to find exploits:

- inventory invariant –
 - a product's stock must be non-negative
 - an item's final stock should reflect orders placed.
- voucher invariant –
 - cannot spend more than their specified limit.
- the cart invariant –
 - total amount charged = total value of goods

Isolation Levels in Practice

The researchers used the web UIs of various applications to find exploits:

“While this invariant may seem obvious, we found that in several of the applications it was possible to add an item to the cart concurrent with checkout, resulting in the user paying for the original total of items in the cart, but placing a valid order including the new item as well. This allows users to obtain items for free. For example, a user might buy a pen and add a laptop to their cart during checkout, paying for the pen but placing an order for the pen and laptop.”

Takeaways

- Isolation levels allow you to balance ACID principles with performance based on your application's requirements.
- Make sure you understand the default behavior of your DBMS.