

Transactions & Concurrency Control

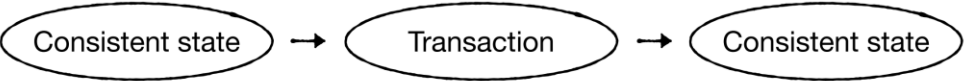
concurrency 7 / 193 56%

A DB has a notion of consistency, defined by the user

| | | |
|---|---|-------------------------------|
| Sum of money in a system should stay constant | An account balance cannot drop below zero | A user must have a valid SIN |
| $\text{sum}(\text{balance}) = X$ | $\text{balance} \geq 0$ | $\text{SIN} \neq \text{null}$ |

These can be set as integrity constraints

concurrency 10 / 193 56%



```
graph LR; A([Consistent state]) --> B([Transaction]); B --> C([Consistent state])
```

Example: Bank transfer

| | | |
|---------------|---|---|
| $A = A - 100$ | ← | Both operations must succeed or fail for the system to remain in a consistent state |
| $B = B + 100$ | ← | |

concurrency13 / 19356%

Example: Money withdrawn

Begin Transaction

b = Select balance from accounts where id = x

If $b \leq 0$

Abort

Else

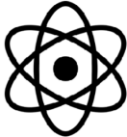
Update accounts set balance = $b - 100$ where id = x

Commit

concurrency18 / 19356%


ACID

Atomicity




All or nothing

Consistency




Transition across consistent states

Isolation



Not corrupted by concurrency

Durability

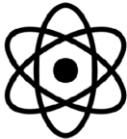


Recover from failure

concurrency19 / 19356%


Who is responsible for what?

Atomicity




DB

Consistency




User

Isolation



DB

Durability




DB

concurrency


20 / 193 | - 56% + | [icon] [icon]

What Endangers Consistency?

System Failure



Concurrency




concurrency


21 / 193 | - 56% + | [icon] [icon]

System Failure


Power failure



Hardware failure



Data center failure



concurrency


27 / 193 | - 56% + | [icon] [icon]

Concurrency

Concurrent transactions can result in an inconsistent state


Example 1

Interest & Payments



Example 2

Updates & statistics



Example 1

Initialization: $A=1000, B=1000$

| Time | T1 | T2 |
|------|--------------------|---------------|
| | $A = A \cdot 1.05$ | $A = A - 100$ |
| | | $B = B + 100$ |
| | $B = B \cdot 1.05$ | |
| | Commit | Commit |


Dirty read: T2 reads a modified but uncommitted data item from T1

Invalid Outcome: $A = 950, B = 1155$
 Valid Outcome 2: $A = 945, B = 1155$
 Valid Outcome 1: $A = 950, B = 1150$


Simplest solution: lock whole DB for any modification

Problems: terrible for performance

While one transaction waits for a storage I/O, another should be able to use the CPU



A short transaction should not have to wait until a long transaction completes



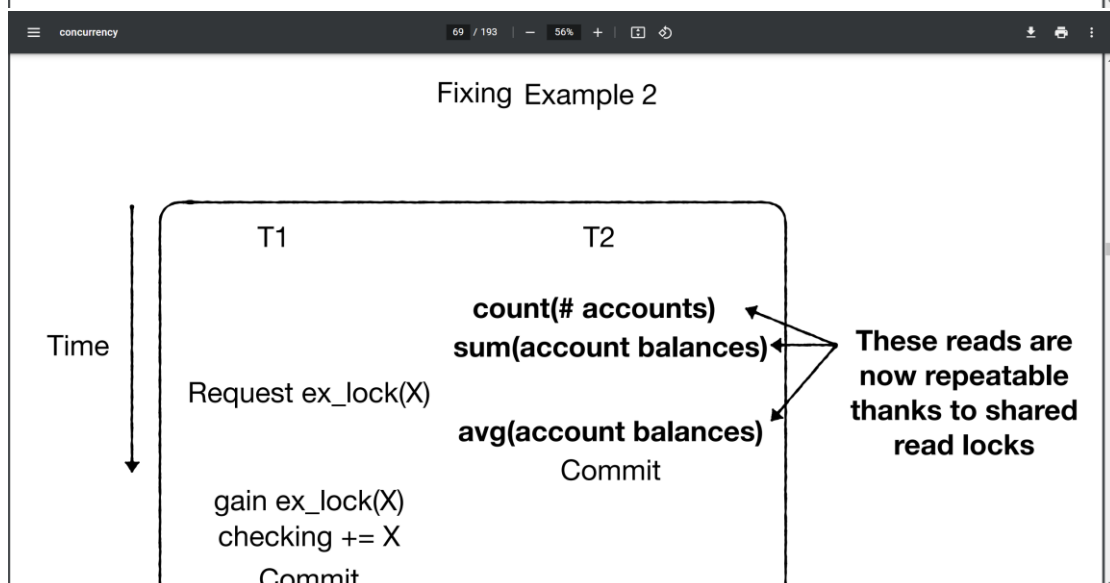
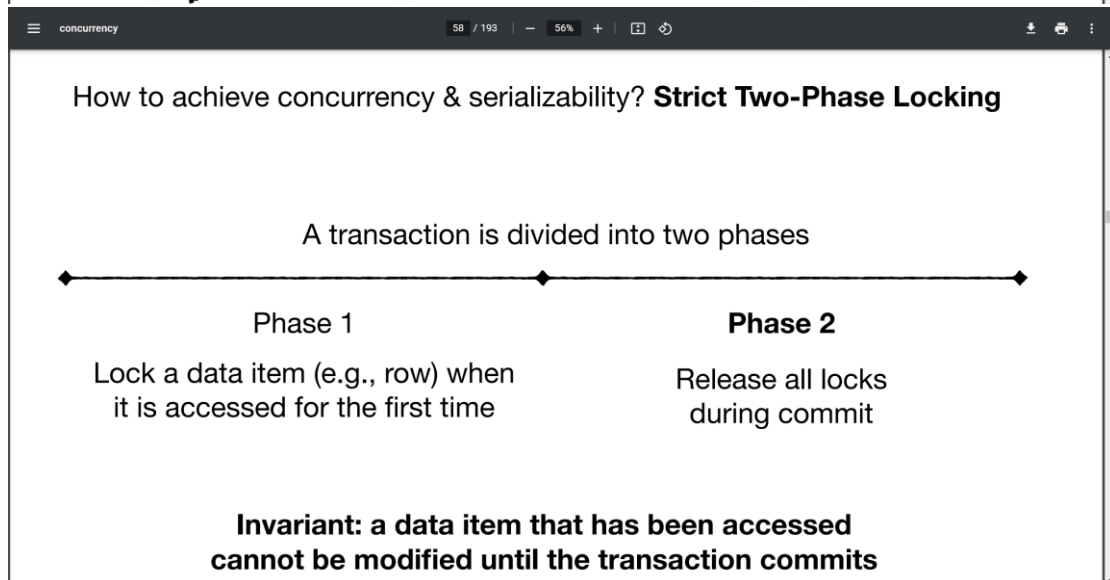
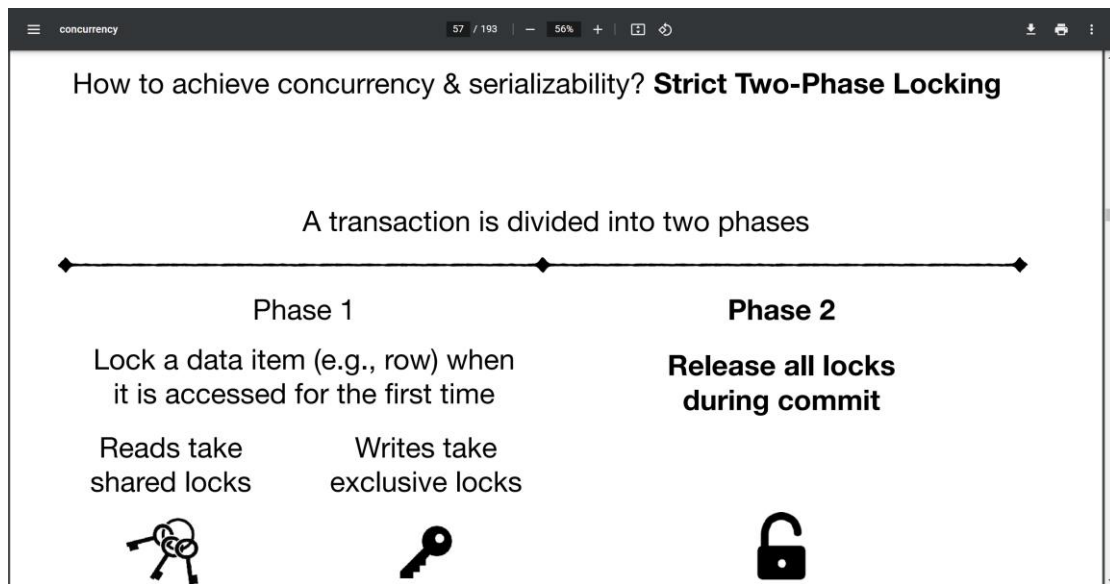
A good compromise: Serializability

Transactions can be concurrent but must result in a consistent state

```



graph LR
    CS1([Consistent state]) --> T1([Transaction 1])
    T1 --> T2([Transaction 2])
    T2 --> T3([Transaction 3])
    T3 --> CS2([Consistent state])
  
```

Any given state once a transaction commits should have been achievable through a serial execution of all committed transactions thus far



concurrency 70 / 193 56%



Both examples now work correctly

| | Example 1 | Example 2 |
|-----------|---|--|
| | Interest & Payments | Updates & Statistics |
| |  |  |
| Problems: | Dirty reads | Unrepeatable reads |
| Solution: | Exclusive write locks | Shared read locks |

concurrency 79 / 193 56%



Lock Manager - a hash table

Object ID → (type, lock count, queue of waiting requests)

| | |
|---|--|
| We lock an object the first time it is accessed by a transaction | Locking table implemented via OS locking primitives (mutexes, semaphores) |
|  |  |

concurrency 83 / 193 56%

An important distinction

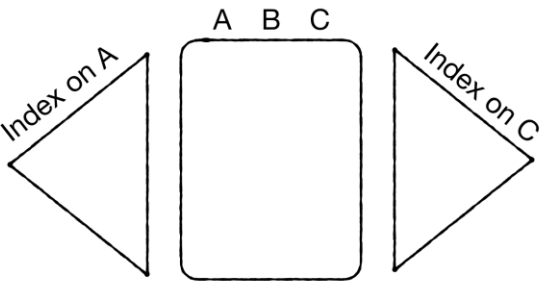
| | Locks | Latches |
|-----------|---|--|
| |  |  |
| Separate: | Transactions | Threads |
| Protect: | DB content | In-memory data structures |
| Duration: | Transaction | Critical section |

concurrency 86 / 193 56%

The DB modifies relevant indexes as a part of a transaction

e.g., update table set C = '...' where A '...'

Index on A




Schedule

- (1) Begin transaction
- (2) Search index A
- (3) Access row and update C
- (4) Update entry in C's index
- (5) Commit

concurrency 94 / 193 56%

What should do when we detect a deadlock?


Abort transaction & undo all its changes

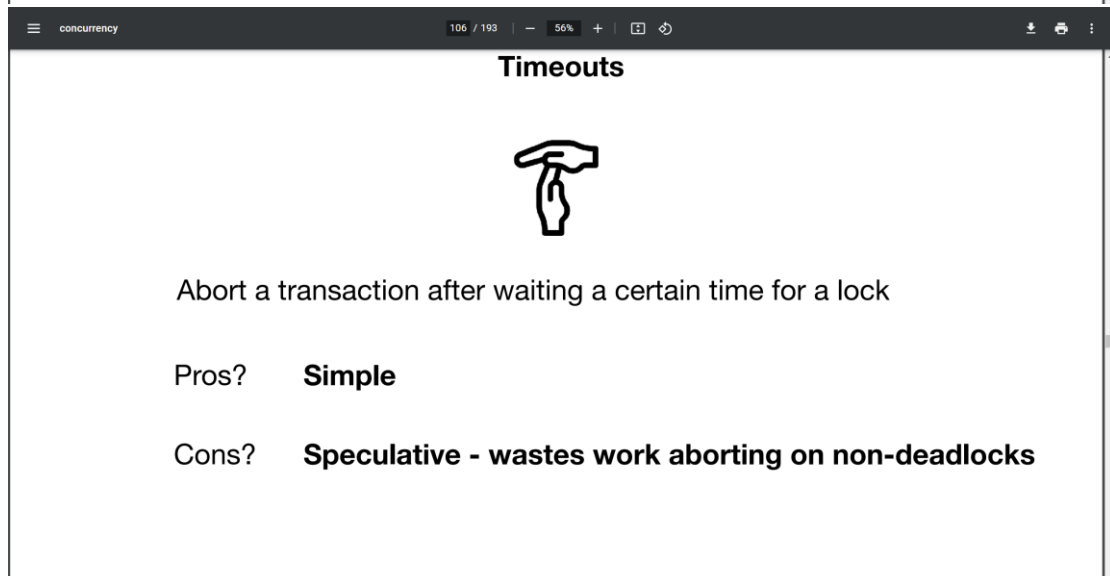
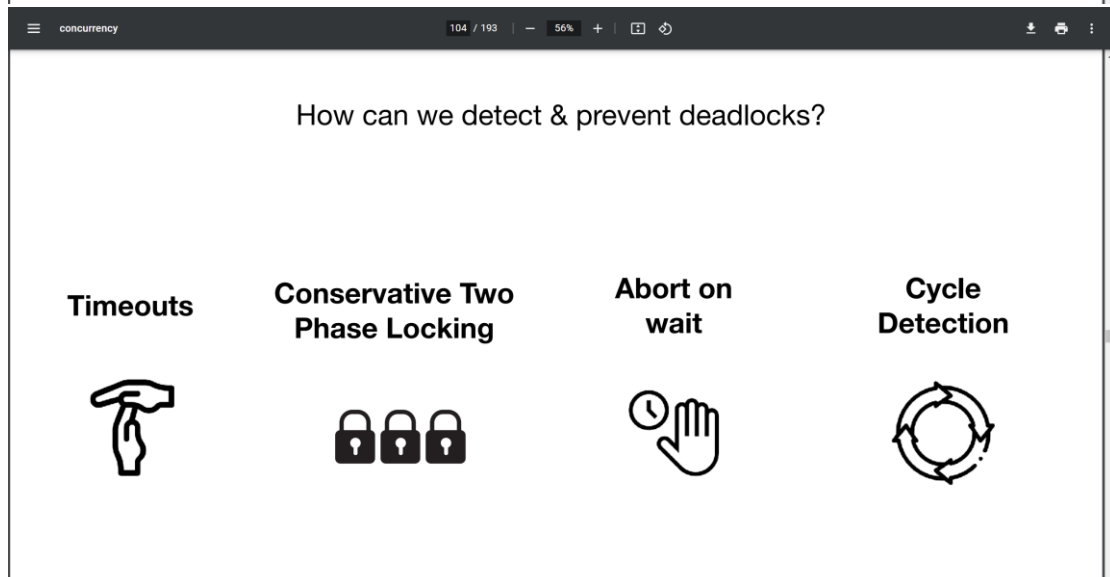
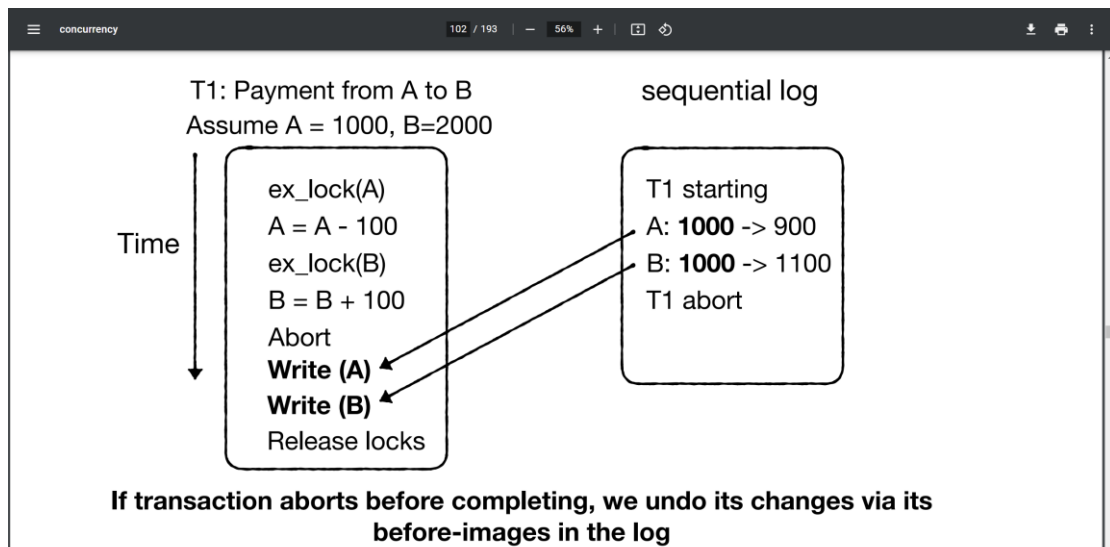


concurrency 98 / 193 56%

How to undo?

record before-image for all changes to the DB in a sequential log.





Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Schedule

| T1 | T2 | T3 | T4 |
|------|------|------|------|
| S(A) | | | |
| | X(B) | | |
| S(B) | | | |
| | X(C) | S(C) | |
| | | X(A) | |
| | | | X(B) |

Graph

```

graph TD
    T1((T1)) --> T2((T2))
    T2 --> T3((T3))
    T3 --> T4((T4))
    T4 --> T1
    T1 --> T3
    T4 --> T2
        
```

Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Abort the transaction that:

- (1) has done the least work
- (2) Farthest from completion
- (3) Been aborted the least times**

```

graph TD
    T1((T1)) --> T2((T2))
    T2 --> T3((T3))
    T3 --> T1
        
```

Conservative Two Phase Locking

A transaction is divided into two phases

Phase 1

Take all locks the transaction
could possibly need

Phase 2

Release all locks
during commit

Pros: no deadlocks

Con: takes more locks and holds them for longer

concurrency 113 / 193 56%

Abort on Wait

When a transaction is blocked, abort it or the transaction holding the lock.

T2 is blocked → [Lock Icon] ← T1 holds lock

↖ Abort one ↗

Pros: no deadlocks
Con: defeatist (wastes work, or aborts many transactions)

concurrency 138 / 193 56%

How can we prevent phantom reads?

Time ↓

| T1 | T2 |
|------------------------|---|
| Insert A = 0 Commit | count(# accounts) sum(account balances) avg(account balances) |

More broadly, this is a **phantom read**: a transaction accesses a set of rows twice, and qualifying rows are added in-between

concurrency 142 / 193 56%

How can we prevent phantom reads?

| | | |
|------------|---------------------|-----------------------------------|
| lock table | Predicate locking | Index locking |
| | | |
| Aggressive | Complex & expensive | Good but requires an index |

concurrency
145 / 193
56%

Index Locking

Lock B-tree leaf storing relevant range

Example: Select * from accounts
where ID="Cindy"

So we do not have to lock the whole table as long as we have an index :)

concurrency
153 / 193
56%

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

| | Dirty read | Unrepeatable read | Phantom reads |
|---------------------|---------------------|---------------------|---------------------|
| Read uncommitted | Possible | Possible | Possible |
| Read committed | Not Possible | Possible | Possible |
| Repeatable reads | Not Possible | Not Possible | Possible |
| Serializable | Not Possible | Not Possible | Not Possible |

concurrency
157 / 193
56%

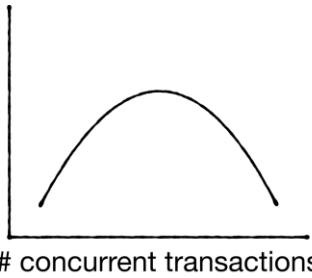
ANSI/ISO Transaction Isolation Levels - part of the SQL standard

| | |
|---------------------|---|
| Read uncommitted | Exclusive write locks not held until a transaction ends |
| Read committed | Shared read locks not held until a transaction ends |
| Repeatable reads | Range locks not employed (e.g., no tree/table locking) |
| Serializable | Everything is fully correct |

concurrency 163 / 193 56%

What do you think this curve should look like?

Throughput
(Transactions / sec)



concurrent transactions

How can we address this?

- (1) Design the application such that transactions are short
- (2) **Restrict the maximum number of concurrent transactions**

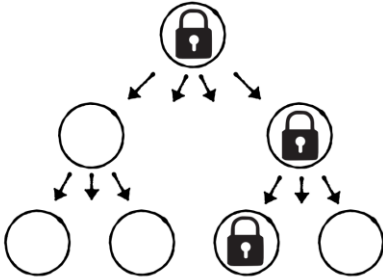
concurrency 166 / 193 56%

With Strict Two-Phase Locking, a transaction locks all objects on its path until it commits

Why is this a problem for B-tree update?

Lots of locking contention at root and upper levels

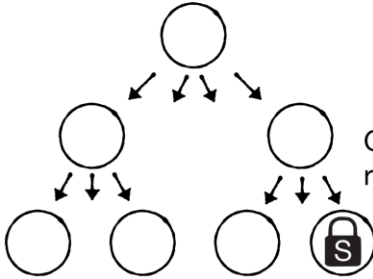
Any solutions?



concurrency 174 / 193 56%

What must we lock at minimum to ensure correctness?

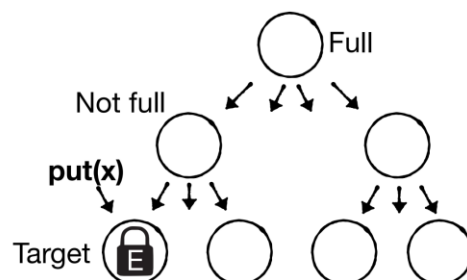
- (1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



Once we hold lock on child, release lock on parent

What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



Shows that while strict two-phase locking is correct, it can sometimes be relaxed while maintaining correctness

