CS 61:
Database Systems

Transactions/Concurrency

Adapted from Silberschatz, Korth, and Sundarshan unless otherwise noted
Augmented by Claude Opus

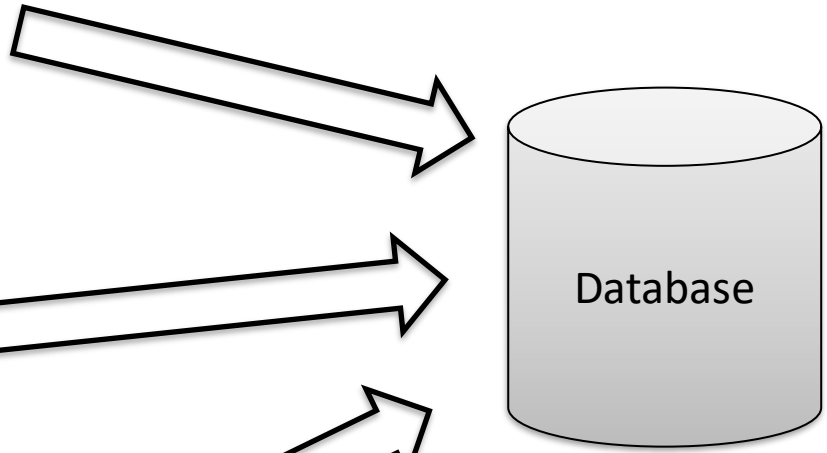
Agenda

1. Databases on ACID
2. Concurrency anomalies
3. Isolation
4. Locking
5. Deadlocks
6. Tips

Goal: quickly serve many users at the same time, but data must stay consistent!

Avoid handling user requests sequentially – too slow!

But, concurrent processing can lead to trouble!



Problem:
Must ensure data stays consistent with concurrent transactions

Assume database starts in consistent state

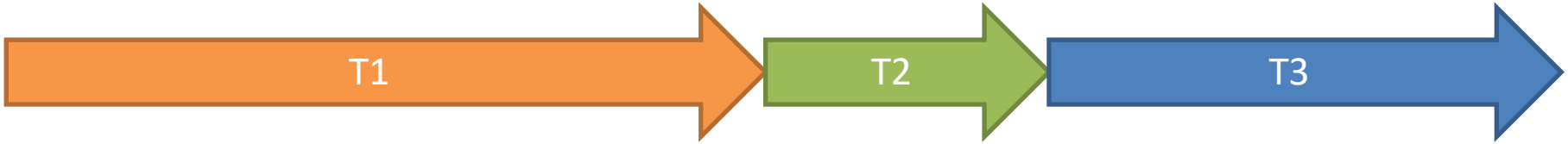
- All integrity constraints met
- All business rules followed

Multiple CPUs in database server could serve multiple requests at the same time

Result: increased throughput

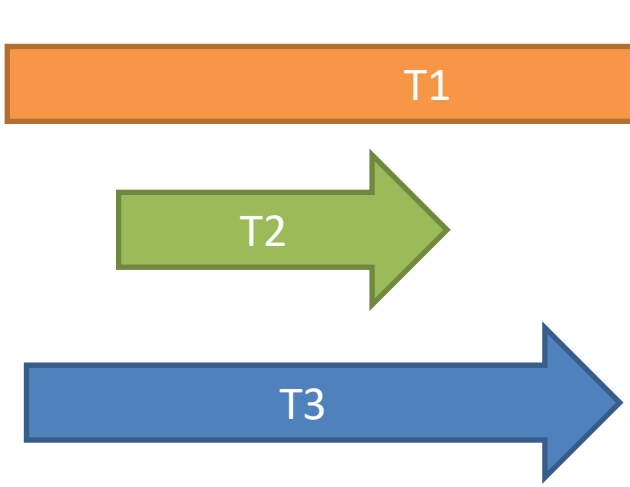
Goal: want transactions to run fast but not allow inconsistencies

Serial schedule (run consecutively; first come, first served)



Serialized schedule **interleaves execution and gives same result as if transaction ran serially**

Equivalent interleaved schedule (serialized)



Schedule is clearly serializable if:

- **Transactions operate on different data**
- **Only read operations**

- Consistency assumptions
  1. Database starts in consistent state
  2. Each transaction leaves database in consistent state when complete
  3. Serial execution of transactions preserves consistency
- Problems can arise if we allow simultaneous (concurrent) transaction execution
- But performance is low if transactions must run serially
- Some transactions do not interfere with each other (they can safely be serialized)

ACID properties safely run concurrent operations without corrupting data

Property	Description	InnoDB
Atomicity	• Transaction treated as indivisible unit of work • All commands complete successful or none do • Locks commonly used to ensure only one transaction accesses data at a time • Undo log allows rollback if transaction aborts or crashes	Transactions Locks Undo log

ACID properties safely run concurrent operations without corrupting data

Property	Description	InnoDB
Atomicity	• Transaction treated as indivisible unit of work • All commands complete successful or none do • Locks commonly used to ensure only one transaction accesses data at a time • Undo log allows rollback if transaction aborts or crashes	Transactions Locks Undo log
Consistency	• All data integrity constraints satisfied (PK, FK, CHECK, NOT NULL, UNIQUE) • Transaction must take database from one consistent state to another • If any integrity constraint is violated, transaction is aborted • Apps are responsible for higher level consistency! Bank app must ensure transfer don't loose money	Constraints + other three properties

ACID properties safely run concurrent operations without corrupting data

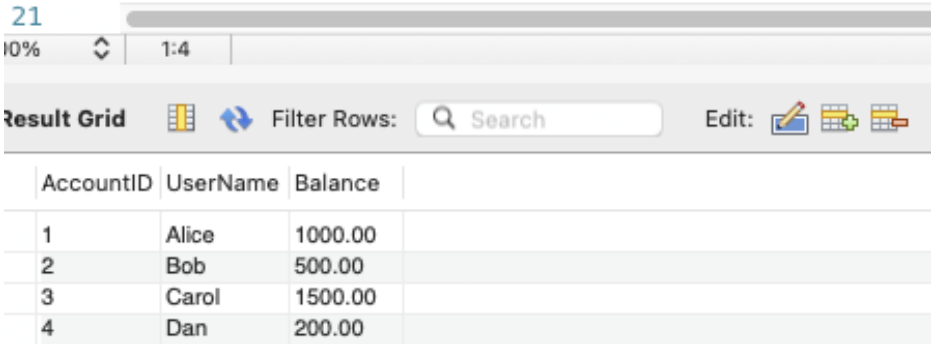
Property	Description	InnoDB
Atomicity	• Transaction treated as indivisible unit of work • All commands complete successful or none do • Locks commonly used to ensure only one transaction accesses data at a time • Undo log allows rollback if transaction aborts or crashes	Transactions Locks Undo log
Consistency	• All data integrity constraints satisfied (PK, FK, CHECK, NOT NULL, UNIQUE) • Transaction must take database from one consistent state to another • If any integrity constraint is violated, transaction is aborted • Apps are responsible for higher level consistency! Bank app must ensure transfer don't loose money	Constraints + other three properties
Isolation	• Data used during a transaction cannot be accessed by another transaction until the first transaction completes • As if each transaction runs by itself, gives same result as serial execution	Locks MVCC

ACID properties safely run concurrent operations without corrupting data

Property	Description	InnoDB
Atomicity	• Transaction treated as indivisible unit of work • All commands complete successful or none do • Locks commonly used to ensure only one transaction accesses data at a time • Undo log allows rollback if transaction aborts or crashes	Transactions Locks Undo log
Consistency	• All data integrity constraints satisfied (PK, FK, CHECK, NOT NULL, UNIQUE) • Transaction must take database from one consistent state to another • If any integrity constraint is violated, transaction is aborted • Apps are responsible for higher level consistency! Bank app must ensure transfer don't loose money	Constraints + other three properties
Isolation	• Data used during a transaction cannot be accessed by another transaction until the first transaction completes • As if each transaction runs by itself, gives same result as serial execution	Locks MVCC
Durability	• Once changes are committed, they cannot be undone • Undo and Redo log written before database table updated	Undo and Redo logs, Write-Ahead Logging (WAL)

We will use bank accounts as a motivating example

```sql
CREATE SCHEMA IF NOT EXISTS test_schema;
USE test_schema;
DROP TABLE IF EXISTS Accounts;
CREATE TABLE Accounts (
    AccountID INT AUTO_INCREMENT PRIMARY KEY,
    UserName VARCHAR(50),
    Balance DECIMAL(10,2)
);

INSERT INTO Accounts VALUES
    (1, 'Alice', 1000.00),
    (2, 'Bob',    500.00),
    (3, 'Carol', 1500.00),
    (4, 'Dan',    200.00);

SELECT * FROM Accounts;
```



AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

We will use bank accounts as a motivating example

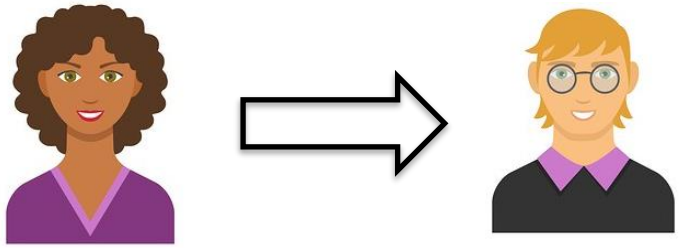
```sql
CREATE SCHEMA IF NOT EXISTS test_schema;
USE test_schema;
DROP TABLE IF EXISTS Accounts;
CREATE TABLE Accounts (
    AccountID INT AUTO_INCREMENT PRIMARY KEY,
    UserName VARCHAR(50),
    Balance DECIMAL(10,2)
);

INSERT INTO Accounts VALUES
    (1, 'Alice', 1000.00),
    (2, 'Bob',    500.00),
    (3, 'Carol', 1500.00),
    (4, 'Dan',    200.00);

SELECT * FROM Accounts;
```

1:4

Result Grid | Filter Rows: | Search | Edit:

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

Transfer $100 from Alice to Bob



```sql
-- Transfer $100 from Alice to Bob
-- first debit Alice
SET @amount = 100;
UPDATE Accounts
    SET Balance = Balance - @amount
    WHERE AccountID = 1;

-- then credit Bob
UPDATE Accounts
    SET Balance = Balance + @amount
    WHERE AccountID = 2;

SELECT * FROM Accounts;
```

11:24

Result Grid | Filter Rows: | Search | Edi

AccountID	UserName	Balance
1	Alice	900.00
2	Bob	600.00
3	Carol	1500.00
4	Dan	200.00

Works as expected
Alice now $900
Bob now $600

Things can (and will) go wrong!

```
5  CREATE SCHEMA IF NOT EXISTS test_schema;
6  USE test_schema;
7  DROP TABLE IF EXISTS Accounts;
8  CREATE TABLE Accounts (
9      AccountID INT AUTO_INCREMENT PRIMARY KEY,
10     UserName VARCHAR(50),
11     Balance DECIMAL(10,2)
12 );
13
14 INSERT INTO Accounts VALUES
15     (1, 'Alice', 1000.00),
16     (2, 'Bob',    500.00),
17     (3, 'Carol', 1500.00),
18     (4, 'Dan',    200.00);
19
20 SELECT * FROM Accounts;
21
```

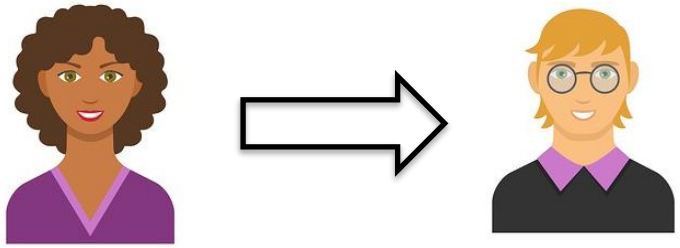
100% 1:4

Result Grid | Filter Rows: Search | Edit:

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

What if server crashed here?

Transfer $100 from Alice to Bob



```
25 -- Transfer $100 from Alice to Bob
26 -- first debit Alice
27 SET @amount = 100;
28 UPDATE Accounts
29     SET Balance = Balance - @amount
30     WHERE AccountID = 1;
31
32 -- then credit Bob
33 UPDATE Accounts
34     SET Balance = Balance + @amount
35     WHERE AccountID = 2;
36
37 SELECT * FROM Accounts;
38
```

100% 1:23

Result Grid | Filter Rows: Search | Ed

AccountID	UserName	Balance
1	Alice	900.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

When things go wrong, it can lead to problems

```
5  CREATE SCHEMA IF NOT EXISTS test_schema;
6  USE test_schema;
7  DROP TABLE IF EXISTS Accounts;
8  CREATE TABLE Accounts (
9      AccountID INT AUTO_INCREMENT PRIMARY KEY,
10     UserName VARCHAR(50),
11     Balance DECIMAL(10,2)
12 );
13
14 INSERT INTO Accounts VALUES
15     (1, 'Alice', 1000.00),
16     (2, 'Bob',    500.00),
17     (3, 'Carol', 1500.00),
18     (4, 'Dan',    200.00);
19
20 SELECT * FROM Accounts;
21
```

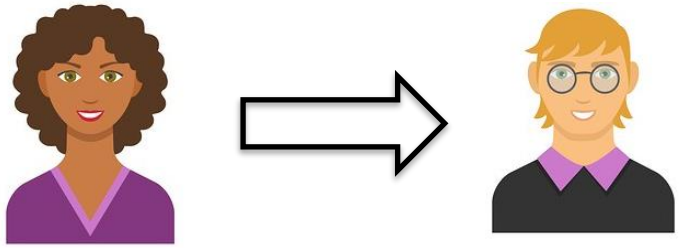
100% 1:4

Result Grid Filter Rows: Search Edit:

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

What if server crashed here?

Transfer $100 from Alice to Bob



```
25 -- Transfer $100 from Alice to Bob
26 -- first debit Alice
27 SET @amount = 100;
28 UPDATE Accounts
29     SET Balance = Balance - @amount
30     WHERE AccountID = 1;
31
32 -- then credit Bob
33 UPDATE Accounts
34     SET Balance = Balance + @amount
35     WHERE AccountID = 2;
36
37 SELECT * FROM Accounts;
38
```

100% 1:23

Result Grid Filter Rows: Search Ed

AccountID	UserName	Balance
1	Alice	900.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

Lost $100!
Alice now $900
Bob still at $500

Transactions can help some problems

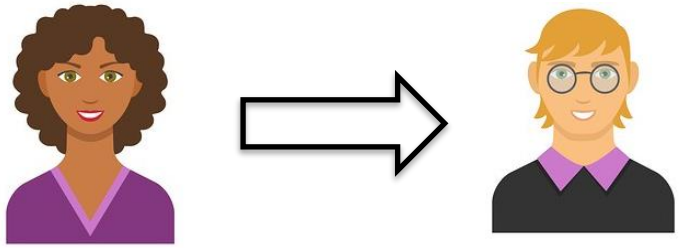
```sql
CREATE SCHEMA IF NOT EXISTS test_schema;
USE test_schema;
DROP TABLE IF EXISTS Accounts;
CREATE TABLE Accounts (
    AccountID INT AUTO_INCREMENT PRIMARY KEY,
    UserName VARCHAR(50),
    Balance DECIMAL(10,2)
);

INSERT INTO Accounts VALUES
    (1, 'Alice', 1000.00),
    (2, 'Bob',    500.00),
    (3, 'Carol', 1500.00),
    (4, 'Dan',    200.00);

SELECT * FROM Accounts;
```

100% 1:4

Result Grid | Filter Rows: Search | Edit:

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

Transfer $100 from Alice to Bob



Solve problem using TRANSACTION

```sql
START TRANSACTION;
SET @amount = 100;
UPDATE Accounts
    SET Balance = Balance - @amount
    WHERE AccountID = 1;

-- then credit Bob
UPDATE Accounts
    SET Balance = Balance + @amount
    WHERE AccountID = 2;

-- Write to disk
COMMIT;

SELECT * FROM Accounts;
```

COMMIT saves to disk

Rollback on crash before COMMIT

100% 1:48

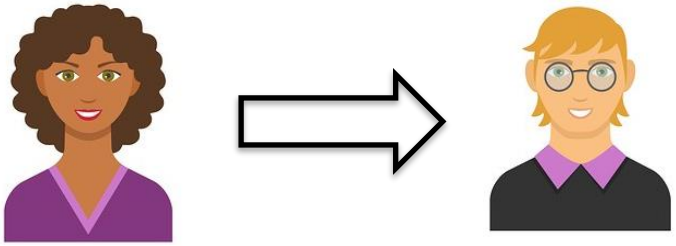
Result Grid | Filter Rows: Search | Edi

AccountID	UserName	Balance
1	Alice	900.00
2	Bob	600.00
3	Carol	1500.00
4	Dan	200.00

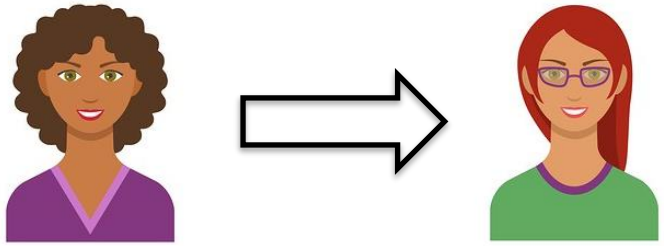
Works as expected
Alice now $900
Bob now $600

Concurrency raises other problems: lost update

T1: Transfer $100 from Alice to Bob



T2: Transfer $200 from Alice to Carol



Two transactions happen at the same time
What if both transactions read Alice's balance before either writes?

This condition is called the *lost update problem*

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

Time	Session 1 (T1)	Session 2 (T2)
t1	`Read Alice Balance $1000`	
t2		`Read Alice Balance $1000`
t3	`UPDATE Alice Balance to $900`	
t4	UPDATE Bob Balance to $600	
t5		`UPDATE Alice Balance to $800`
t6		`UPDATE Carol Balance to $1700`

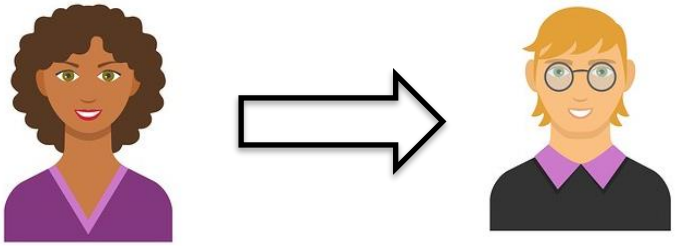
Both read Alice's Balance before update completed

As expected:
- **Bob is up $100**
- **Carol is up $200**

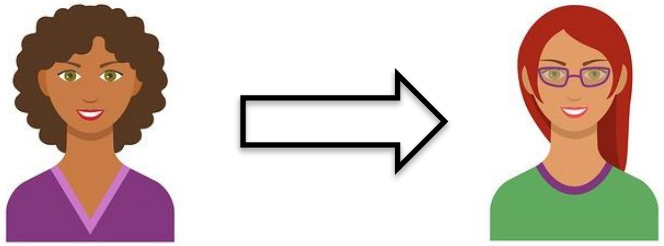
But:
- **Alice is only down $200**
- **Should be down $300**

Concurrency raises other problems: uncommitted data

T1: Transfer $100 from Alice to Bob



T2: Transfer $200 from Alice to Carol



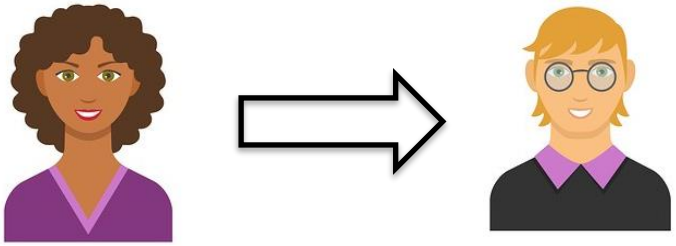
Now two transactions happen in sequence

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

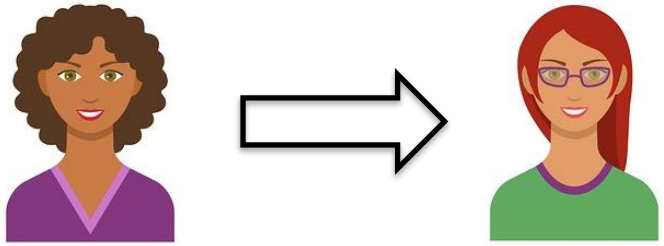
Time	Session 1 (T1)	Session 2 (T2)
t1	`Read Alice Balance $1000`	
t2	`UPDATE Alice Balance to $900`	
t3	UPDATE Bob Balance to $600	
t4		`Read Alice Balance $900`
t5		`UPDATE Alice Balance to $700`
t6		`UPDATE Carol Balance to $1700`
t7	COMMIT	`COMMIT`

Concurrency raises other problems: uncommitted data

T1: Transfer $100 from Alice to Bob



T2: Transfer $200 from Alice to Carol



Now two transactions happen in sequence

This concurrency situation an isolation problem

This case is called the *uncommitted data problem*

AccountID	UserName	Balance
1	Alice	1000.00
2	Bob	500.00
3	Carol	1500.00
4	Dan	200.00

Time	Session 1 (T1)	Session 2 (T2)
t1	`Read Alice Balance $1000`	
t2	`UPDATE Alice Balance to $900`	
t3	UPDATE Bob Balance to $600	
t4		`Read Alice Balance $900`
t5		`UPDATE Alice Balance to $700`
t6		`UPDATE Carol Balance to $1700`
t7	ROLLBACK	

T2 reads uncommitted data

But T1 rolls back

Alice should have $800, but has $700

Relation-level inconsistencies can occur when results depend on transaction order

Relation-level inconsistency

T1

```
UPDATE Apply
SET Decision = 'Y'
Where StudentID IN (SELECT StudentID from Student WHERE GPA > 3.9);
```

T2

```
UPDATE Student
SET GPA = 1.1*GPA
WHERE HighSchoolSize > 2500;
```

Apply holds student applications for college

- **Simple admission criteria based only on grade**
- **But maybe large school students get a GPA bump**

Some rows in the Apply table are affected by order in which these transactions are run

- **If T1 runs before T2, some students won't be accepted that would have been accepted if T2 ran first**
- **Here updates are applied to different relations, but could give different results**
- **T1 operates on two tables, T2 operates on one of those two**

Based on https://lagunita.stanford.edu/assets/courseware/v1/b91aa86921e55e62d426677a4a36e85e/c4x/DB/Indexes/asset/TransactionsProperties.pdf

Multi-statement inconsistencies can occur when results depend on transaction order

Multi-statement inconsistency

Results from SELECT statements depend on whether they run before, after, or between INSERT/DELETE statements

```
INSERT INTO Archive
    SELECT * FROM Apply WHERE Decision = 'N';
DELETE FROM Apply WHERE Decision = 'N';

SELECT COUNT(*) from Apply;
SELECT COUNT(*) from Archive;
```

If SELECT runs here

- **DELETE has not yet run**
- **Total count will be incorrect because 'N' decision not yet deleted from Apply**

Based on https://lagunita.stanford.edu/assets/courseware/v1/b91aa86921e55e62d426677a4a36e85e/c4x/DB/Indexes/asset/TransactionsProperties.pdf

Multi-statement inconsistencies can occur when results depend on transaction order

Multi-statement inconsistency

```
INSERT INTO Archive
    SELECT * FROM Apply WHERE Decision = 'N';
DELETE FROM Apply WHERE Decision = 'N';

SELECT COUNT(*) from Apply;
SELECT COUNT(*) from Archive;
```

Must we force all transactions to run serially (one after the other)?

- **Defeats the purpose of large databases serving many simultaneous users**
- **Want concurrency so we have highest possible performance**

What about system failures?

- **Power goes out during transaction**
- **Disgruntled employee types: rm –rf /**

Transactions to the rescue!

Based on Prof Palmer lecture notes

Transactions are a logical unit of work that must be entirely completed or aborted

```
-- extended example from MySQL Docs section 15.7.2.2
DROP TABLE IF EXISTS Customers;
CREATE TABLE Customers (a INT, b CHAR (20), INDEX (a));

-- Do two transactions and commit
START TRANSACTION;
INSERT INTO Customers VALUES (10, 'Heikki');
INSERT INTO Customers VALUES (11, 'Elvis');
COMMIT; -- transactions saved
SELECT * FROM Customers;

START TRANSACTION;
INSERT INTO Customers VALUES (15, 'John');
INSERT INTO Customers VALUES (20, 'Paul');
DELETE FROM Customers WHERE b = 'Heikki';

-- Now we undo those last 2 inserts and the delete.
ROLLBACK;
SELECT * FROM Customers;
```

Transaction starts and data inserted

Power failure here would rollback changes at restart (not committed)

Data committed

Second transaction inserts two rows and deletes one

Inserts and delete rolled back, no change to Customers table

Result Grid | Filter

a	b
10	Heikki
11	Elvis

Transactions can make multiple commands Atomic, Isolated, and Durable

Consistency follows when the application also respects constraints and invariants

Based on Prof Palmer lecture notes

When a row is modified a set of operations ensure Atomicity and Durability

When you modify a row, InnoDB performs these actions in order:

1 Write Undo Log

- Old data written to the undo log buffer (in memory)
- e.g., “old Balance was $150”

2 Write Redo Log

- New change, including the undo log modification, written to the redo log buffer (in memory)
- e.g., “new Balance is $100”

3 Flush Logs to Disk

- Logs flushed from memory to disk at COMMIT
- Write-Ahead Logging (WAL): logs hit disk before database table pages

4 Update Database Table

- Database table pages on disk updated later in the background
- Update undo and redo logs after data written to tables

Transaction ROLLBACK

Uses the UNDO LOG

If ROLLBACK is called (or the transaction fails partway), InnoDB reads the undo records and reverses every change

Example: ***Balance restored to $150***

Crash recovery (after restart)

Uses BOTH logs

Redo log: rolls *forward* — replays committed changes whose data pages weren't yet written to disk

Undo log: rolls *back* — reverses any in-flight transactions that hadn't committed

Rollback uses undo

Crash recovery uses both: redo rolls forward, undo rolls back

Claude AI

Agenda

1. Databases on ACID
2. Concurrency anomalies
3. Isolation
4. Locking
5. Deadlocks
6. Tips

There are four common concurrency anomalies that can occur in a transaction

1. Dirty read: read uncommitted data
2. Non-repeatable read: same row but read different values
3. Phantom read: new rows appear or rows disappear
4. Lost update: data silently overwritten

We will discuss database isolation levels to deal with these anomalies

1) Dirty Read: read uncommitted data

1) T1 updates Alice's Balance

2) T2 reads uncommitted update
Makes decision based on value read

Time	T1	T2
t1	START TRANSACTION;	
t2		START TRANSACTION;
t3	UPDATE Accounts SET Balance=5000 WHERE ID=1;	
t4		SELECT Balance FROM Accounts WHERE ID=1; -- returns 5000 ← DIRTY
t5		-- approves loan, COMMIT;
t6	ROLLBACK; -- Alice still has 1000	

3) Update transaction is rolled back

T2 made a decision based on data that never officially existed

2) Non-repeatable read: same row different values

1) T1 reads Alice's Balance

Time	T1 (READ COMMITTED)	T2
t1	START TRANSACTION;	
t2	SELECT Balance FROM Accounts WHERE ID=1; -- 1000	
t3		START TRANSACTION; UPDATE Accounts SET Balance=1500 WHERE ID=1; COMMIT;
t4	SELECT Balance FROM Accounts WHERE ID=1; -- 1500 ← CHANGED	
t5	COMMIT;	

3) T1 reads Alice's Balance a second time
Sees a different value from the first read

2) T2 updates Alice's Balance and commits

T1 sees two different values for the *same row* in one transaction

3) Phantom read: new rows appear or rows disappear

1) T1 counts balances over $100

Time	T1 (REPEATABLE READ)	T2
t1	`START TRANSACTION;`	

t2	`SELECT COUNT(*) FROM Accounts`
	`  WHERE Balance > 1000;`
	`-- returns 1 (Carol)`	

t3		`START TRANSACTION;`
		`INSERT INTO Accounts VALUES`
		`  (5, 'Eve', 2000);`
		`COMMIT;`

t4	`SELECT COUNT(*) FROM Accounts`
	`  WHERE Balance > 1000;`
	`-- returns 2  ← PHANTOM`	

3) T1 counts balances over $1000 again, but now there are new rows that did not exist previously

2) T2 adds new row that did not exist when T1 counted rows

Non-repeatable read: a row already read changed

Phantom read: rows appear (or disappear) that match query

4) Lost update: data silently overwritten

1) T1 reads Alice's Balance

Time	T1	T2
t1	START TRANSACTION;	
t2	SELECT Balance FROM Accounts WHERE ID=1; -- app reads 1000	**2) T2 also reads Alice's Balance**
t3		START TRANSACTION; SELECT Balance FROM Accounts WHERE ID=1; -- app reads 1000
t4	-- app computes 1000-100=900 UPDATE Accounts SET Balance=900 WHERE ID=1;	
t5	**3) T1 updates Balance - 100**	-- app computes 1000-200=800 UPDATE Accounts SET Balance=800 WHERE ID=1;
t6	COMMIT;	**4) T2 also updates Balance but -200**
t7		COMMIT;

5) T1 commit first

6) T2 commit second, overwrites T1

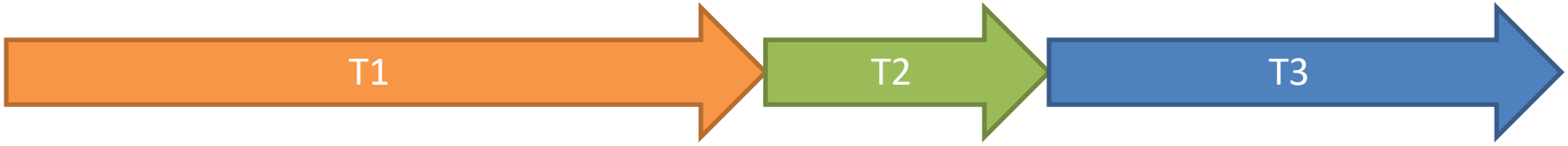
Balance *should be* $700
Balance *is* $800
T1 withdrawal silently disappeared

Agenda

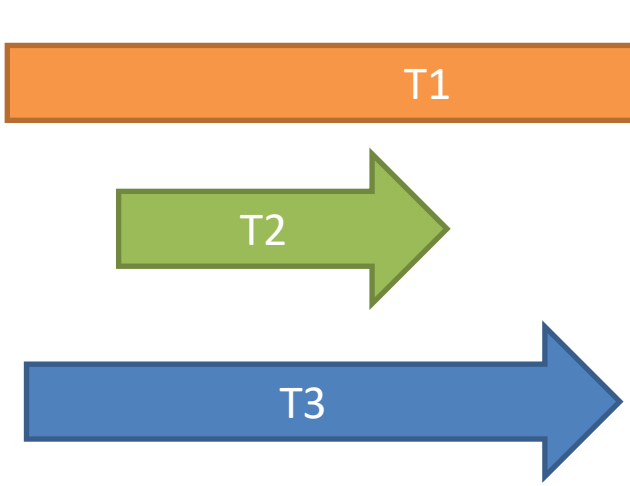
1. Databases on ACID
2. Concurrency anomalies
3. Isolation
4. Locking
5. Deadlocks
6. Tips

Recall: we want Isolated transactions do not interfere with each other

Serial schedule (run consecutively; first come, first served)



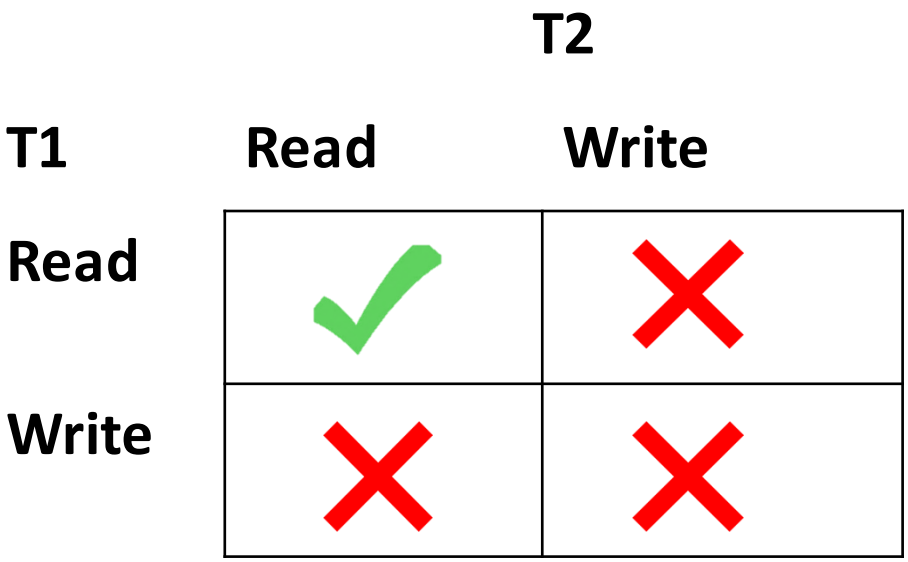
Equivalent interleaved schedule (serialized)



- Consistency assumptions
  - Database starts in consistent state
  - Each transaction leaves database in consistent state when complete
  - Serial execution of transactions preserves consistency
- Serial schedule has poor performance; transactions must wait for preceding transactions to finish
- A schedule is ***serializable*** if it is interleaved, but equivalent to a serial schedule (not all schedules are serializable)
- Serialized schedule results in increased performance and ***Isolation***

Most combinations of reads and writes of related data can cause potential problems

Inconsistent retrieval and uncommitted data problems



Note:
Read often written as S
Write often written as X

If T1 and T2 operate on different data (e.g., T1 updates Employees, T2 updates Products)

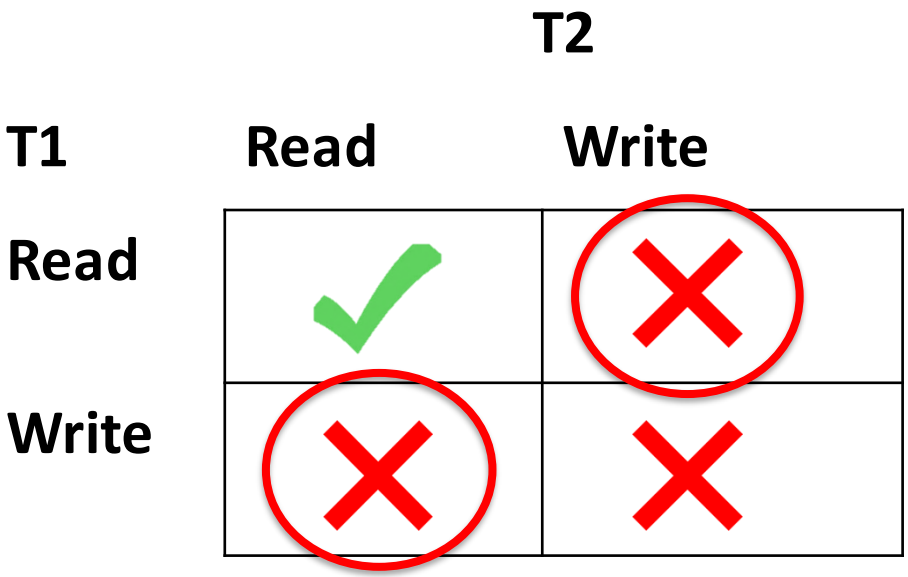
- No problems running concurrently
- Each can run concurrently

If T1 and T2 operate on the *same* data

- Could have problems if one or both write data
- No problem if both only read data

Most combinations of reads and writes of related data can cause potential problems

Inconsistent retrieval and uncommitted data problems



If T1 and T2 operate on different data (e.g., T1 updates Employees, T2 updates Products)

- No problems running concurrently
- Each can run concurrently

If T1 and T2 operate on the same data

- Could have problems if one or both write data
- No problem if both only read data

Problems if one transaction reads and another writes

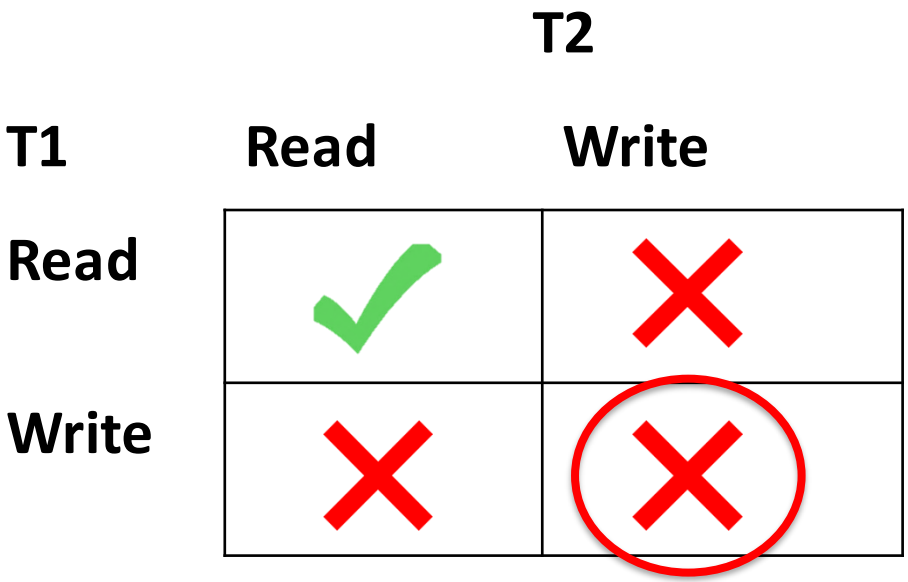
*Inconsistent retrieval problem: r*ead operation may read data that is no longer current

- Example: T1 calculates summary info over set of data while T2 updates portion of same data

Uncommitted data problem: if T1 reads after T2 writes, but T2 rolls back, T1's data is incorrect

Most combinations of reads and writes of related data can cause potential problems

Inconsistent retrieval and uncommitted data problems



Problems if two transactions write the same data

Lost update problem:

- Each transaction reads the same data, changes it, then writes it back
- Last update wins

If T1 and T2 operate on different data (e.g., T1 updates Employees, T2 updates Products)

- No problems running concurrently
- Each can run concurrently

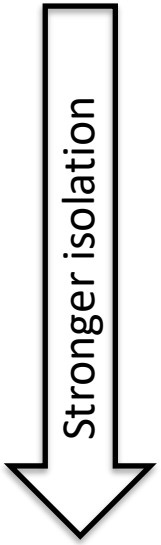
If T1 and T2 operate on the same data

- Could have problems if one or both write data
- No problem to if both only read data

SQL defines four isolation levels, choose the lowest one that meets your needs

Four standard SQL Isolation Levels

Less locking
More anomalies
Higher throughput



More locking
Fewer anomalies
Lower throughput

1. READ UNCOMMITTED
2. READ COMMITTED
3. REPEATABLE READ
4. SERIALIZABLE

You can set the database to these four isolation levels

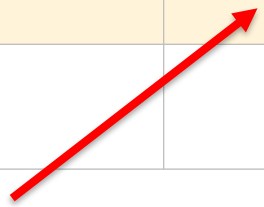
Isolation levels

Level	Dirty Read	Non-Repeatable Read	Phantom Read
`READ UNCOMMITTED`	✗ allowed	✗ allowed	✗ allowed
`READ COMMITTED`	✓ prevented	✗ allowed	✗ allowed
`REPEATABLE READ`	✓ prevented	✓ prevented	✗ allowed
`SERIALIZABLE`	✓ prevented	✓ prevented	✓ prevented

You can set the database to these four isolation levels

Isolation levels

Level	Dirty Read	Non-Repeatable Read	Phantom Read
`READ UNCOMMITTED`	✗ allowed	✗ allowed	✗ allowed
`READ COMMITTED`	✓ prevented	✗ allowed	✗ allowed
`REPEATABLE READ`	✓ prevented	✓ prevented	✓ InnoDB
`SERIALIZABLE`	✓ prevented	✓ prevented	✓ prevented



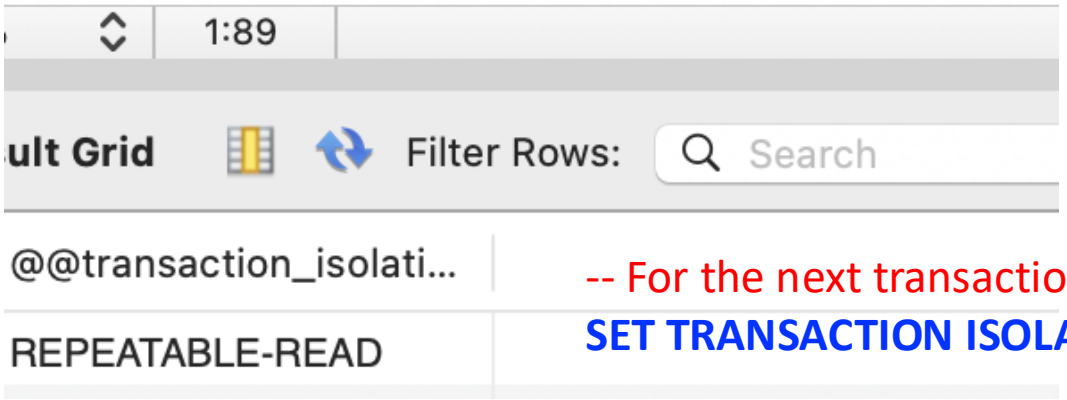
InnoDB REPEATABLE READ isolation level uses next-key locking
Stronger than the SQL standard requires

InnoDB uses REPEATABLE READ by default

Isolation levels

1. READ UNCOMMITTED
2. READ COMMITTED
3. REPEATABLE READ
4. SERIALIZABLE

```
SELECT @@transaction_isolation;
```



Most other databases use READ COMMITTED as default

- **Oracle**
- **SQL Server**
- **Postgres**

Result: SQL code can break across databases!

```
-- For the next transaction only
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- For all transactions in this session
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Globally (for new connections)
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

DEMO: Dirty Read by setting isolation level to READ UNCOMMITTED

Start two terminals
Login to database with: mysql -u root –p test_schema
USE test_schema;

Set READ UNCOMMITTED, then start a transaction (you might think it should be atomic but is not at this isolation level!)

Time	T1	T2
t1	SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;	SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
t2	START TRANSACTION;	START TRANSACTION;
t3	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1000	
t4	**Read Balance ($1000)**	UPDATE Accounts SET Balance = 1500 WHERE AccountID = 1;
t5	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1500	
t6	ROLLBACK;	ROLLBACK;
t7		

T2 updates balance to $1500, but does not commit

With READ UNCOMMITTED, T1 sees a Balance of $1500, even through it is still in the same transaction and T2 has not yet committed!
Why?
Isolation level set to READ UNCOMMITTED!

DEMO: Prevent Dirty Read by setting isolation level to READ COMMITTED

Start two terminals
Login to database with: mysql -u root –p test_schema
USE test_schema;

Try again with READ COMMITTED (not UNCOMMITTED)

Time	T1	T2
t1	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
t2	START TRANSACTION;	START TRANSACTION;
t3	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1000	
t4		UPDATE Accounts SET Balance = 1500 WHERE AccountID = 1;
t5		SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1500
t6	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1000	
t7	ROLLBACK;	ROLLBACK;

With READ COMMITTED, T1 sees a Balance of $1000
Dirty read is prevented

DEMO: Prevent non-repeatable read by setting isolation level to REPEATABLE READ

Start two terminals
Login to database with: mysql -u root –p test_schema
USE test_schema;

Try again with REPEATABLE READ

Time	T1	T2
t1	SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;	SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
t2	START TRANSACTION;	START TRANSACTION;
t3	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1000	
t4		UPDATE Accounts SET Balance = 1500 WHERE AccountID = 1;
t5		COMMIT;
t6		SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1500
t7	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1000	
t8	COMMIT;	
t9	SELECT Balance FROM Accounts WHERE AccountID = 1; -- 1500	

T1 sees updated value only after it commits

T2 sees new value after commit

Balance value is preserved during transaction

REPEATABLE READ pins a snapshot at the first read
Subsequent reads see the same row versions throughout the transaction, served from the undo log if newer versions exist

SERIALIZABLE silently makes SELECT into SELECT FOR SHARE

SERIALIZABLE in InnoDB

At SERIALIZABLE, InnoDB silently converts plain SELECT into:

```
SELECT ... FOR SHARE
```

FOR SHARE make query a locking read

Any reader blocks any writer

Expensive: all reads lock operations

SQL defines four isolation levels, choose the lowest one that meets your needs

Isolation levels

Level	Dirty Read	Non-Repeatable Read	Phantom Read	Notes
`READ UNCOMMITTED`	✗ allowed	✗ allowed	✗ allowed	Almost never used Approximate reporting only
`READ COMMITTED`	✓ prevented	✗ allowed	✗ allowed	Most OLTP workloads PostgreSQL/Oracle default for a reason
`REPEATABLE READ`	✓ prevented	✓ prevented	✗ allowed	Consistent snapshots batch jobs, multi-statement reads (InnoDB's default, REPEATABLE READ prevented)
`SERIALIZABLE`	✓ prevented	✓ prevented	✓ prevented	When correctness must be airtight Pays heavily in throughput

InnoDB's REPEATABLE READ is stronger than the SQL standard requires

- **MVCC handles phantoms for plain reads by serving a consistent snapshot to transaction**
- **Next-key locking handles phantoms by locking the gaps between records**

SQL defines four isolation levels, choose the lowest one that meets your needs

Isolation levels

Level	Dirty Read	Non-Repeatable Read	Phantom Read	Notes
`READ UNCOMMITTED`	✗ allowed	✗ allowed	✗ allowed	Almost never used Approximate reporting only
`READ COMMITTED`	✓ prevented	✗ allowed	✗ allowed	Most OLTP workloads PostgreSQL/Oracle default for a reason
`REPEATABLE READ`	✓ prevented	✓ prevented	✓ InnoDB	Consistent snapshots batch jobs, multi-statement reads (InnoDB's default, REPEATABLE READ prevented)
`SERIALIZABLE`	✓ prevented	✓ prevented	✓ prevented	When correctness must be airtight Pays heavily in throughput

InnoDB's REPEATABLE READ is stronger than the SQL standard requires

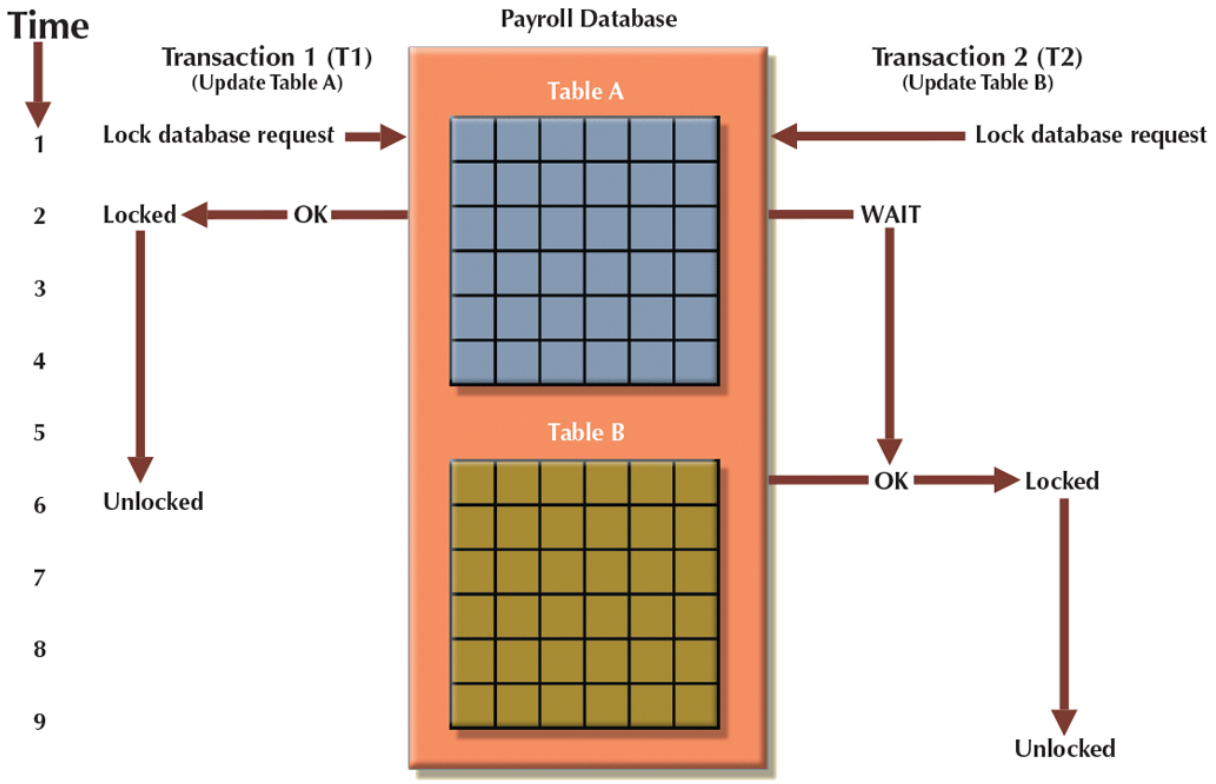
- **MVCC handles phantoms for plain reads by serving a consistent snapshot to transaction**
- **Next-key locking handles phantoms by locking the gaps between records**

Agenda

1. Databases on ACID
2. Concurrency anomalies
3. Isolation
4. Locking
5. Deadlocks
6. Tips

Database-level locks can implement Atomic and Isolation properties

Database-level lock



Database-level locks tie up the entire database while a transaction executes

- **Good for batch processes**
- **Normally not used otherwise**
- **Implements serialization!**
- **Prevents all four anomalies**

Based on Coronel and Morris

Locks implemented at the table-level allow unrelated transactions to run concurrently

Table-level lock

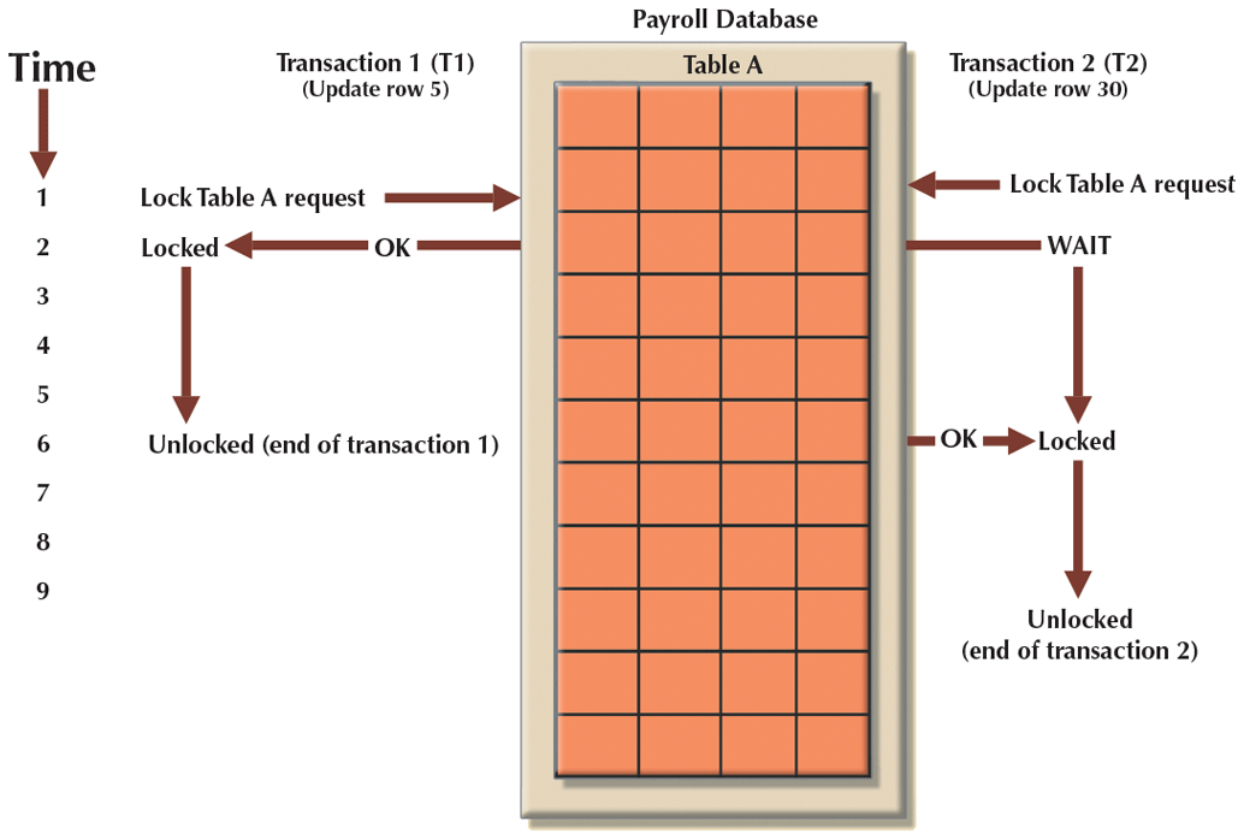


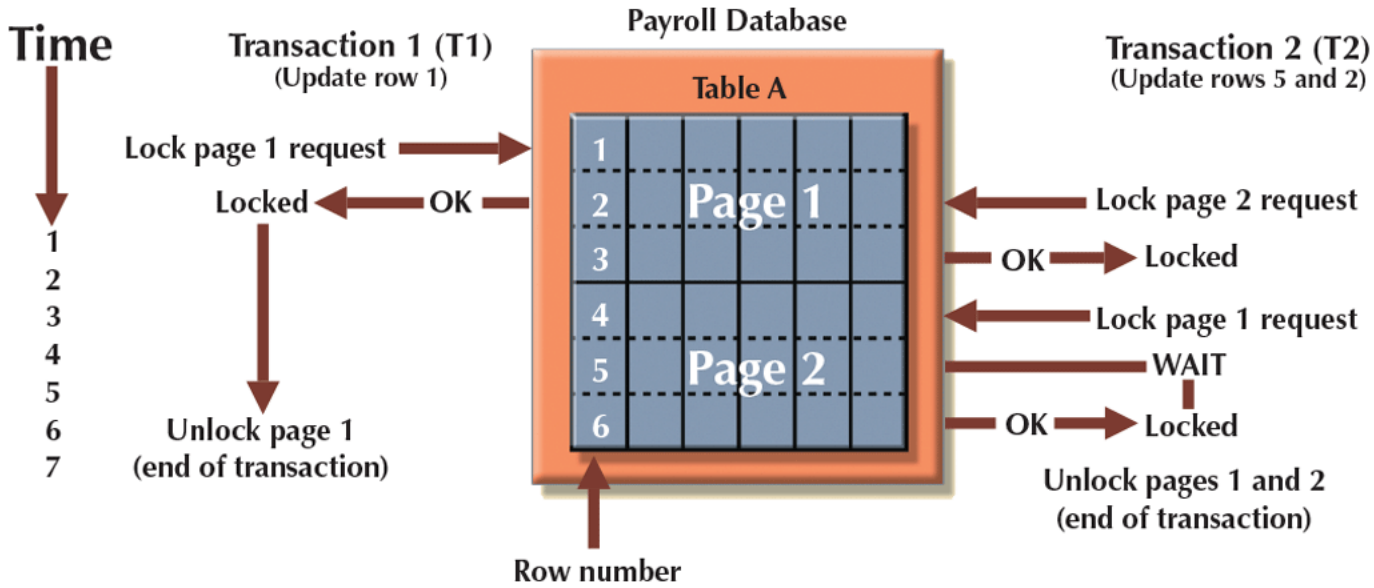
Table is locked during transaction

- **Other tables can be accessed by different transactions**
- **Transactions attempting to access locked table must wait**
- **Lock manager notifies waiting a transaction it can proceed**
- **Prevents all four anomalies**
- **Still too coarse grained for many multi-users systems**

Based on Coronel and Morris

Page-level locks allow concurrent access to different areas of one table

Page-level lock



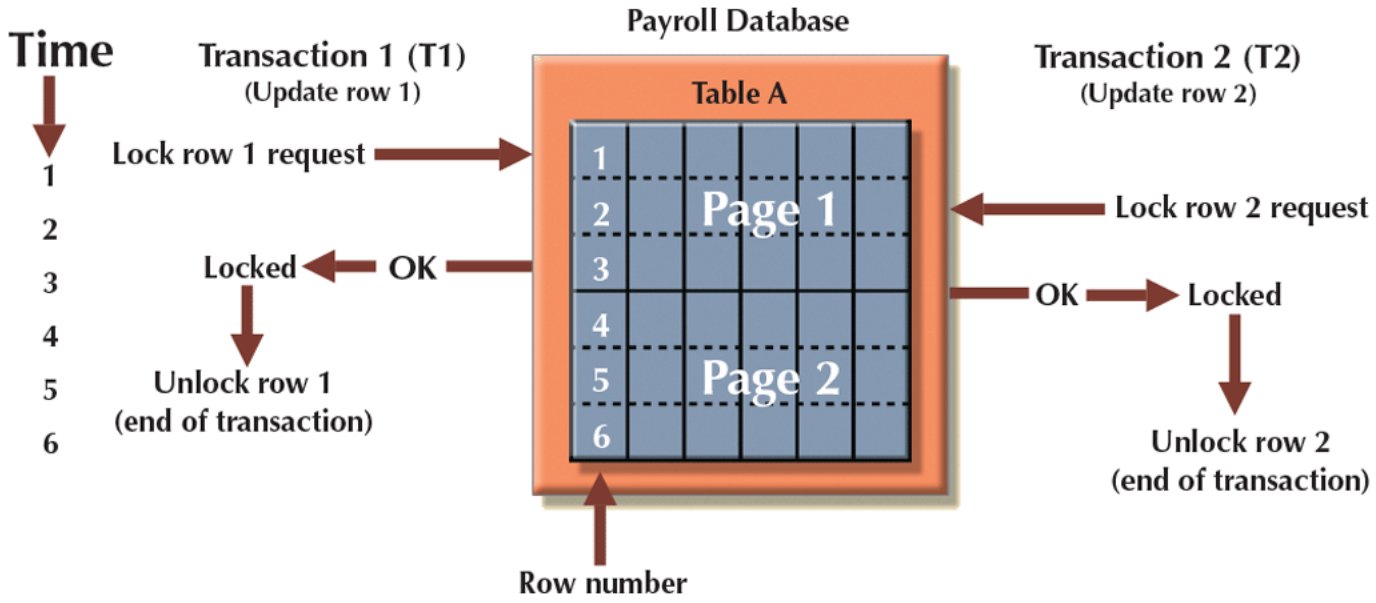
Database locks a disk page (disk block)

- **Page normally fixed size (4K, 8K, or 16K)**
- **To write 73 bytes to a 4K page, must read all 4K bytes, make update, then write all 4K bytes back to disk**
- **Table may be several pages long**
- **Multiple processes can access same table simultaneously**
- **May not prevent phantom reads (insert into another block)**

Based on Coronel and Morris

Row-level locks allow concurrent access to different rows of a table

Row-level lock



Database locks a single row in a table

- **Improves availability of data**
- **Requires high overhead to track**

Phantom reads still possible

Field-level locks are conceptually possible, but not often used (too much overhead)

Based on Coronel and Morris

Row-level locking lets multiple writers progress simultaneously

Row-level locking

InnoDB locks rows, not tables
There is no conflict if two transactions updates different rows of the same table

```
-- T1
UPDATE Accounts SET Balance=900 WHERE ID=1;   -- write-lock on row 1

-- T2 — runs concurrently, no wait
UPDATE Accounts SET Balance=400 WHERE ID=2;   -- write-lock on row 2
```

There are three kinds of row locks: record, gaps, and next-key

Row lock varieties in InnoDB

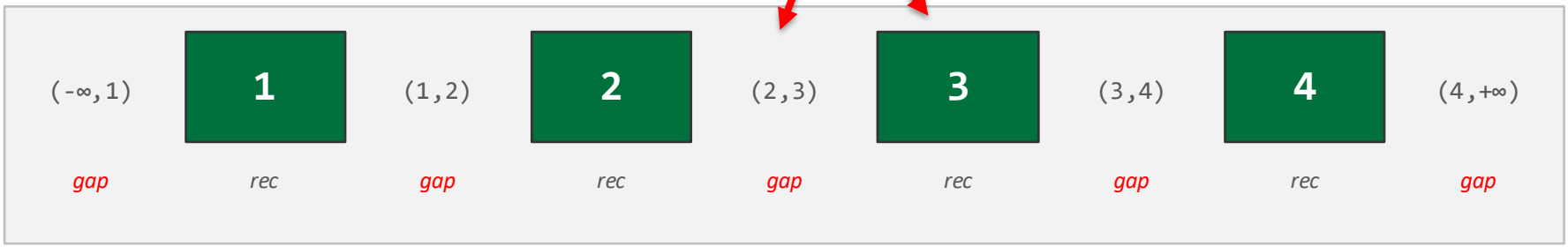
Lock type	Locks	Prevents
Record lock	Locks a single record	Updates/deletes to that exact row
Gap lock	Locks the space between records	Inserts into the gap between records Prevents phantoms at REPEATABLE READ
Next-key lock	Record + preceding gap Default of InnoDB REPEATABLE READ	Both of the above, combined

A next-key lock covers a record and its preceding gap

Gap locks visualized

Index on ID containing: 1, 2, 3, 4

Example: lock record 3 and gap (2 to 3)
Prevents inserting something between 2 and 3



Why only lock the gap before?

- Database reads B+ tree in order, lowest to highest
- Already passed prior record, but hasn't read the next record
- Each gap is "owned" by one record (InnoDB keeps a "supremum" record for +∞)

SELECT * FROM Accounts WHERE ID >= 2 **FOR UPDATE**

Locks rows for write

Range scan locks gap before last record

- Locks gap (1,2), (2,3), (3,4), and (4,+∞)
- INSERT INTO Accounts VALUES (5, …) blocks on supremum
- Other queries can access ID =1

Claude AI

MVCC: multiple row versions let readers skip locks

Multi-Version Concurrency Control (MVCC)

InnoDB keeps multiple versions of each row, so:

- Readers don't block writers
- Writers don't block readers

Uses the undo log, the same one used for atomicity
Undo log pulls double duty (undo + MVCC)

Claude AI

Read views select which row versions a transaction sees

MVCC — how a read sees a snapshot

Each transaction has a read view — a snapshot of which transactions were committed when:

- Transaction started — under REPEATABLE READ
- Statement started — under READ COMMITTED

Reading a row:

1. Check the row's current version
2. If committed by a transaction in our read view then read it
3. Otherwise, walk the undo chain to find an older version we *can* read

Four types of reads in InnoDB

Read type	MVCC or current?	Locks taken	Use
Plain SELECT	MVCC snapshot	None	Most of the time
SELECT … FOR SHARE	Current state	S locks	When you need a stable value without preventing other readers from also pinning it (uncommon)
SELECT … FOR UPDATE	Current state	X locks	Intend to modify the rows you're reading, and there's application logic between the read and the write that must be stable
Reads under SERIALIZABLE	Current state	S locks	Implicitly converts plain SELECTs into SELECT … FOR SHARE

Most of the time you should use plain SELECT with REPEATABLE READ
MVCC prevents dirty and non-repeatable reads, and hides phantoms for plain reads
InnoDB also adds gap locking to prevent phantoms for locking reads

Claude AI

Consistent reads use MVCC; locking reads acquire locks

Consistent read vs locking read

	Consistent (non-locking) read	Locking read
Syntax	`SELECT ...`	`SELECT ... FOR UPDATE` `SELECT ... FOR SHARE`
Uses	MVCC snapshot	Current data + acquires locks
Blocks?	**Never**	**Yes, if conflicting locks held**
Sees concurrent commits?	No	Yes

The distinction in InnoDB — internalize this and most behavior makes sense

Claude AI

Within one transaction, plain and locking SELECTs may differ

The MVCC / locking surprise

Time	T1 (REPEATABLE READ)	T2
t1	`START TRANSACTION;`	

t2	`SELECT Balance FROM Accounts`
	`  WHERE ID=1;`
	`-- 1000 (snapshot set)`	

t3		`START TRANSACTION;`
		`UPDATE Accounts SET Balance=9999`
		`  WHERE ID=1;`
		`COMMIT;`

t4	`SELECT Balance FROM Accounts`
	`  WHERE ID=1;`
	`-- STILL 1000  (MVCC)`	

t5	`SELECT Balance FROM Accounts`
	`  WHERE ID=1 FOR UPDATE;`
	`-- 9999  (current, locked)`	

Within one transaction, plain and locking SELECTs can return different values

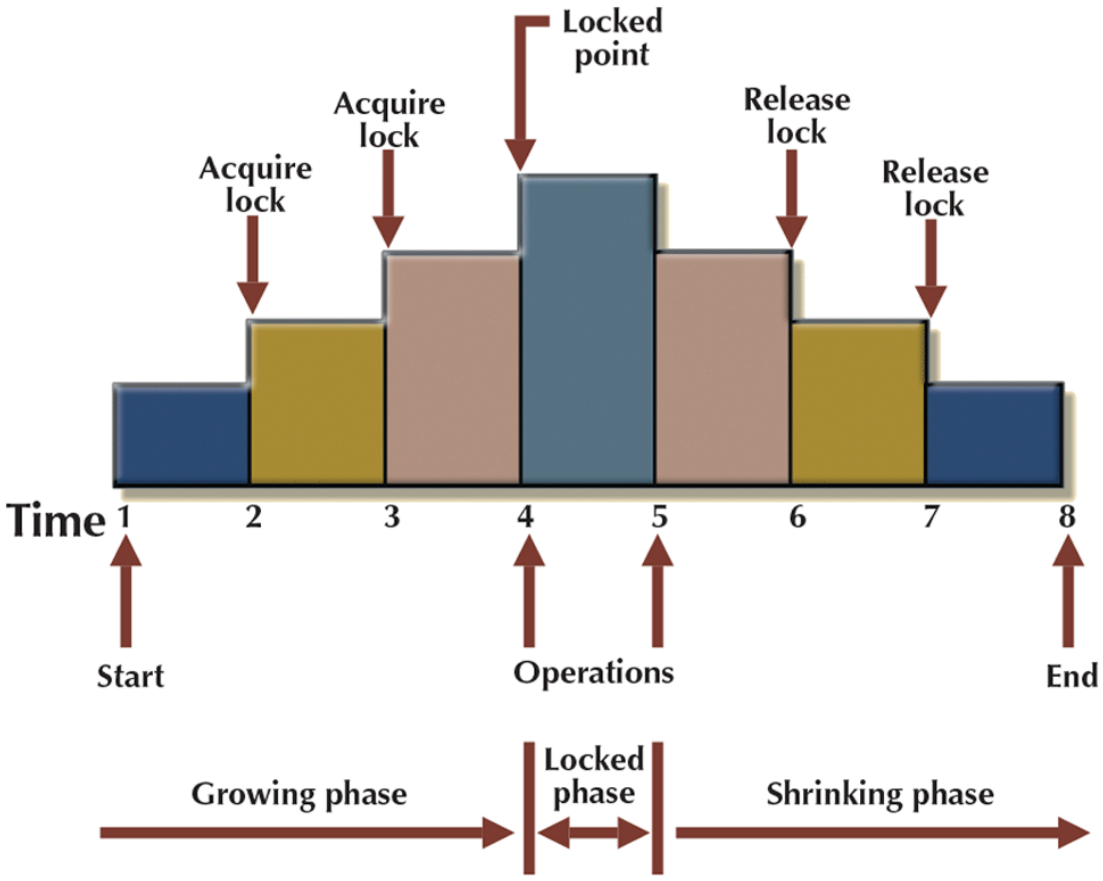
Claude AI

Agenda

1. Databases on ACID
2. Concurrency anomalies
3. Isolation
4. Locking
5. Deadlocks
6. Tips

Two-phase locking protocol guarantees serializability, but may deadlock

Two-phase locking to ensure serializability



Phase 1: growing phase

- Acquire all needed locks before conducting data operations
- Two transaction cannot both hold conflicting lock (two reads are not a conflict)
- No data is affected until all locks are obtained (atomic)

Phase 2: shrinking phase

- Release all locks and cannot obtain a new lock until all locks released
- No unlock operation can precede a lock operation in same transaction

Ensures serializability but might deadlock!

Based on Coronel and Morris

Transactions can deadlock, either prevent them or detect and recover from them

Deadlocks

T1	T2
Exclusive lock (A)	
Read (A)	
A=A-1	
Write (A)	Shared lock (B)
	Read (B)
	Exclusive lock (A)
Exclusive lock (B)	

Shared lock (S) – read only, many can hold
Exclusive lock (X) – for writes, only one holds
T1: exclusive locks A and tries to exclusive lock B
T2: shared locks B and tries to exclusive lock A
Result is deadlock (exclusive lock request does not override existing shared lock)
System must roll back (and unlock) one transaction

To deadlock, four conditions must each be met

1. Mutual exclusion – only one transaction can access data at a time
2. Hold and wait – one process holding a resource while waiting for another
3. No preemption – no transaction can be forced to give up a lock
4. Circular wait – must be a circular chain of locks waiting for access

Break any of these condition and you can overcome deadlock

Transactions can deadlock, either prevent them or detect and recover from them

Deadlocks

T1	T2
Exclusive lock (A)	
Read (A)	
A=A-1	
Write (A)	Shared lock (B)
	Read (B)
	Exclusive lock (A)
Exclusive lock (B)	

Shared lock (S) – read only, many can hold
Exclusive lock (X) – for writes, only one holds
T1: exclusive locks A and tries to exclusive lock B
T2: shared locks B and tries to exclusive lock A
Result is deadlock (exclusive lock request does not override existing shared lock)
System must roll back (and unlock) one transaction

Book covers graph-based methods that do not deadlock, but have high overhead

Deadlock options

- **Prevention** – never allow deadlock to occur
  - Make acquisition of all locks atomic operation (break hold and wait)
  - Use if probability of deadlocks is high
- **Recovery** – detect deadlock and roll back a victim transaction
  - Force one transaction to release locks and roll back (break no preemption)
  - Use if probability of deadlocks is low
  - Look for cycles in graph

Classic deadlock: opposite lock ordering creates a cycle

Time	T1	T2
t1	START TRANSACTION;	START TRANSACTION;

t2	UPDATE Accounts
	  SET Balance=Balance-50
	  WHERE ID=1;
	-- X-lock on ID=1	

t3		UPDATE Accounts
		  SET Balance=Balance-50
		  WHERE ID=2;
		-- X-lock on ID=2

t4	UPDATE Accounts
	  SET Balance=Balance+50
	  WHERE ID=2;
	-- WAITS for T2	

t5		UPDATE Accounts
		  SET Balance=Balance+50
		  WHERE ID=1;
		-- DEADLOCK

t6	-- InnoDB rolls one back -- ERROR 1213	

If both had locked the lower ID (1) first, no deadlock is possible
No hold and wait

Catch ERROR 1213 and retry with exponential backoff

Handling deadlocks

Deadlocks are not bugs — they are a normal cost of concurrency

Applications must catch and retry

```
for attempt in range(MAX_RETRIES):
    try:
        with connection.transaction():
            do_transfer(from_id, to_id, amount)
        break
    except DeadlockError:    # MySQL error 1213
        sleep(random.uniform(0, 0.1 * 2**attempt))  # backoff
```

Make sure the re-tried operation is idempotent (same every time)

Exponential backoff with jitter avoids “thundering-herd” retries (every process backs off the same amount and retries at the same time again)

SHOW ENGINE INNODB STATUS shows the latest deadlock

Inspecting deadlocks

```
SHOW ENGINE INNODB STATUS\G
```

Look for the LATEST DETECTED DEADLOCK section:

- Which transactions were involved
- What locks each held
- What locks each was waiting for
- Which one InnoDB rolled back (the “victim”)

Agenda

1. Databases on ACID
2. Concurrency anomalies
3. Isolation
4. Locking
5. Deadlocks
6. Tips

Practical guidance for using transactions

1. Use transactions explicitly. Don't trust autocommit for multi-statement work
2. Match isolation level to need. Default is fine for most workloads
3. Keep transactions short. Don't hold locks across user input or network calls
4. Lock in consistent order. Single best deadlock prevention
5. Handle deadlocks with retry in app code. ERROR 1213 is normal, not a bug
6. Test under concurrency. Single-threaded tests miss the bugs

Lost update fix 1: atomic UPDATE with arithmetic in SQL

Fix 1: atomic UPDATE

```
START TRANSACTION;
UPDATE Accounts SET Balance = Balance - 100 WHERE ID = 1;
COMMIT;
```

InnoDB takes an X lock during the UPDATE. Read + Write happen as one operation.

Use when: no application logic needed between read and write.

Many "lost update" bugs disappear if you just push the arithmetic into SQL.

Claude AI

Lost update fix 2: pessimistic locking with SELECT FOR UPDATE

Fix 2: pessimistic locking

FOR UPDATE locks row for write

```
START TRANSACTION;
SELECT Balance FROM Accounts WHERE ID=1 FOR UPDATE;
-- T2 doing the same blocks here
-- ... application logic, fraud checks, etc goes here ...
UPDATE Accounts SET Balance = Balance - 100 WHERE ID=1;
COMMIT;
```

Use when: application logic between read and write needs to see the locked value.

Cost: serializes access to hot rows. Hot-row contention is a common scaling bottleneck.

Lost update fix 3: optimistic concurrency control

Fix 3: version column

```
ALTER TABLE Accounts ADD COLUMN Version INT DEFAULT 0;

-- App reads balance and version
SELECT Balance, Version FROM Accounts WHERE ID=1;  -- 1000, v=7

-- App writes back with version check
UPDATE Accounts
  SET Balance=900, Version=8
  WHERE ID=1 AND Version=7;
-- If rows_affected = 0, someone updated; retry.
```

Use when: conflict is rare; hot rows would hurt throughput.

No locks held between read and write — better concurrency. Cost: retry logic.

Practice: find the lost-update bug in this coupon code

The coupon problem

```
CREATE TABLE Coupons (
  Code VARCHAR(20) PRIMARY KEY,
  Discount DECIMAL(5,2),
  MaxUses INT,
  TimesUsed INT DEFAULT 0
);
INSERT INTO Coupons VALUES
  ('LAUNCH50', 50, 100, 0);
```

Coupons are issues to Users

Each coupon can be used a limited number of times

Track how many times each coupon has been used so far

Do not allow a coupon to be used more than MaxUses

```
-- The naive code (find the bug)
START TRANSACTION;
SELECT TimesUsed, MaxUses
  FROM Coupons WHERE Code='LAUNCH50';

INSERT INTO Redemptions (Code, Used)
  VALUES ('LAUNCH50', NOW());

UPDATE Coupons
  SET TimesUsed = TimesUsed + 1
  WHERE Code='LAUNCH50';
COMMIT;
```

Fix issues with this code:

- **Recall you can lock a row with SELECT … FOR UPDATE**
- **Check how many times each coupon has been used**

Claude AI

Transaction log allows database to rollback if a transaction aborts

Transaction log Like our Audit table

If system failure or ROLLBACK, use log to return to prior consistent state

```
START TRANSACTION;
UPDATE Products SET Quantity = 23 WHERE ProductID = 1558;
UPDATE Customers SET Balance = 615.73 WHERE CustomerID = 1001;
COMMIT;
```

Log often kept on separate/ multiple disks (RAID)

Write changes to transaction log first , then update database (called a Write-Ahead-Log (WAL) protocol)

LogID	TransID	Prev	Next	Op	Table	RowID	Attribute	Before value	After value
341	101	Null	352	Start	** Start				
352	101	341	363	Update	Products	1558	Quantity	25	23
363	101	352	365	Update	Customer	1001	Balance	525.75	615.73
365	101	363	Null	Commit	** End				

Prev and Next LogID

Log and transaction IDs assigned by database

Operation, table, row, and attribute affected by change

Before and after values

Doesn't clean up variables that change or updates to other schemas

Based on Coronel and Morris

Transaction log allows database to rollback if a transaction aborts

Transaction log

```
START TRANSACTION;
UPDATE Products SET Quantity = 23 WHERE ProductID = 1558;
UPDATE Customers SET Balance = 615.73 WHERE CustomerID = 1001;
COMMIT;
```

LogID	TransID	Prev	Next	Op	Table	RowID	Attribute	Before value	After value
341	101	Null	352	Start	** Start				
352	101	341	363	Update	Products	1558	Quantity	25	23
363	101	352	365	Update	Customer	1001	Balance	525.75	615.73
365	101	363	Null	Commit	** End				

Two common approaches:

1. **Deferred-write – transaction log updated immediately, but database tables not updated until commit; if aborts, no changes made to tables; write "dirty buffers" at commit**
2. **Write-through – transaction log updated immediately, then database tables updated directly afterward; use transaction log to rollback if needed**

Based on Coronel and Morris