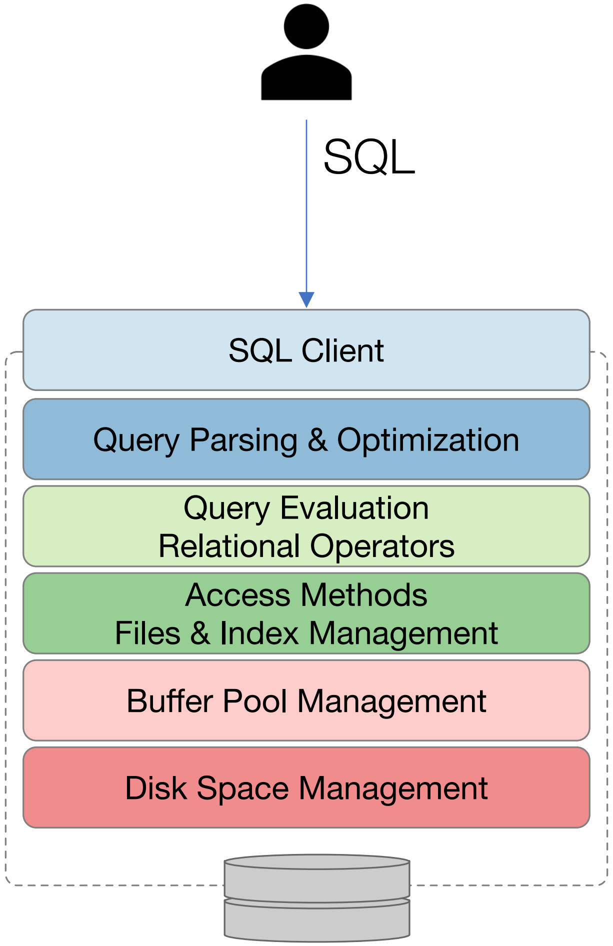
Transactions



SQL

SQL Client

Query Parsing & Optimization

Query Evaluation
Relational Operators

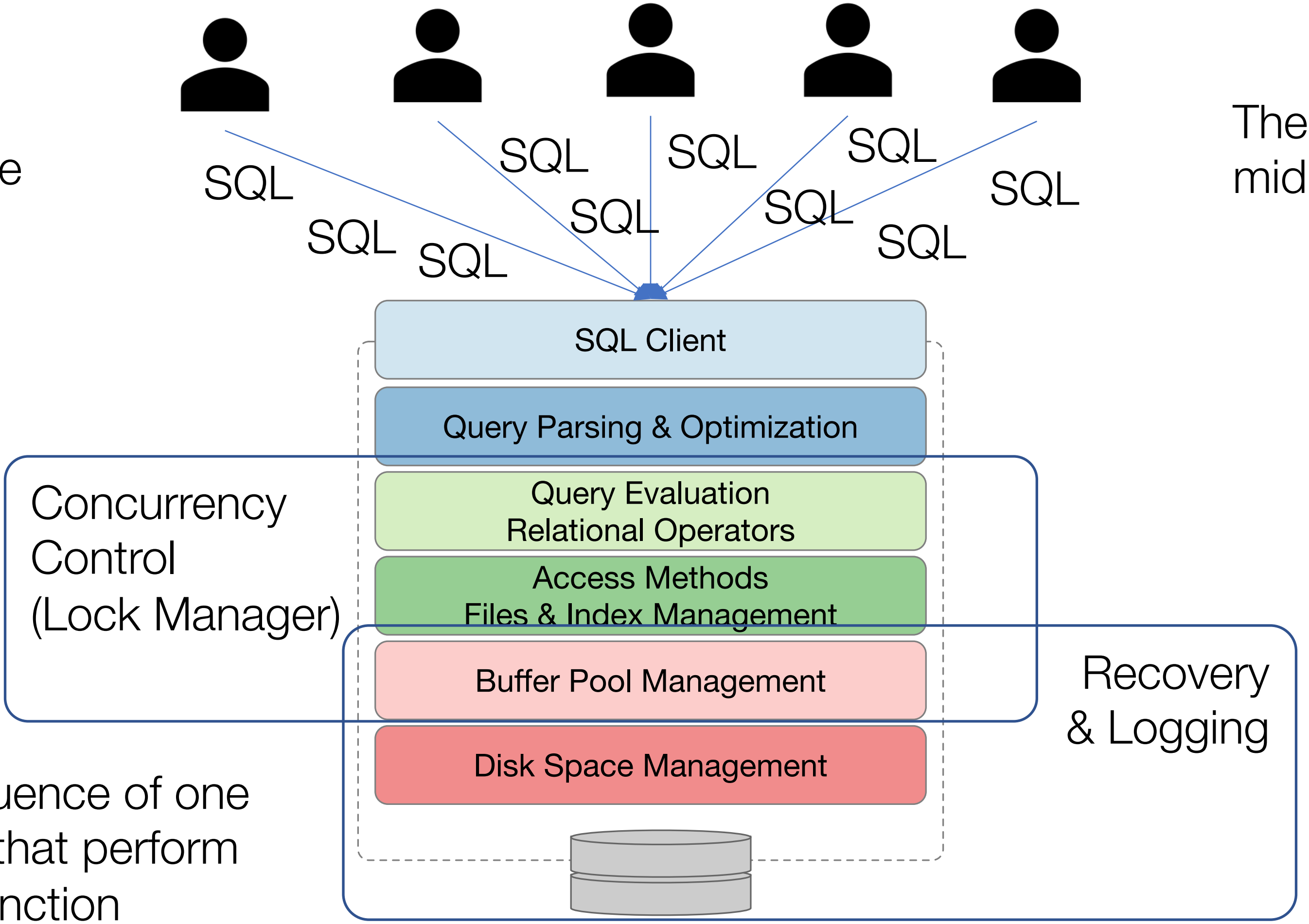
Access Methods
Files & Index Management

Buffer Pool Management

Disk Space Management

Two users change the same record at the same time.

The power fails in the middle of your update



SQL SQL SQL SQL SQL SQL SQL SQL SQL SQL

SQL Client

Query Parsing & Optimization

Query Evaluation
Relational Operators

Access Methods
Files & Index Management

Buffer Pool Management

Disk Space Management

Concurrency Control (Lock Manager)

Recovery & Logging

Transaction Manager

Transaction – a sequence of one or more operations that perform some higher-level function

DBMS provide certain transaction guarantees (e.g. ACID) that make the lives of programmers easy 😎

Transactions

A *sequence* of multiple actions to be executed as an *atomic* unit

A transaction ends in one of 2 ways:

- *Commit* after completing all its actions. If committed, the DBMS guarantees the update occurred!
- *Abort* (or be aborted by the DBMS) after executing some actions; A transaction that didn't complete due to a system crash is treated as an Abort

DBMS only sees a *sequence of reads and writes* devoid of application logic

ACID Transactions

Atomicity: All actions in a transaction happen, or none happen.

Focus of Logging & Recovery

Consistency: If the DB starts out consistent, it ends up consistent at the end of the Xact! (The DBMS aborts transactions that violate any Integrity Constraints)

Isolation: Execution of each Xact is isolated from that of others

Focus of Concurrency Control

Durability: If a Xact commits, its effects persist.

Do we need concurrency?

More Throughput (transactions per second)

Increase processor/disk utilization

- Single core: one transaction uses the CPU while another does IO
- Multicore: scale throughput in the number of processors

Latency (response time per transaction)

- A transaction does not need to wait for another unrelated transaction

What is the worst that could happen with concurrency?

The case of too many bank accounts!

User 1	User 2
BEGIN	BEGIN

`INSERT INTO StudentAccounts`
`SELECT * FROM Accounts`
`WHERE occupation ==`
`‘student’;`	

	`SELECT count(*)` `FROM StudentAccounts;`
	`SELECT count(*)` `FROM Accounts;`

`DELETE Accounts`
`WHERE occupation ==`
`‘student’;`	

COMMIT	COMMIT

Inconsistent Reads

What is the worst that could happen with concurrency?

The case of where did my money go!

User 1	User 2
BEGIN	BEGIN
DECLARE _bal numeric;	DECLARE _bal numeric;
SELECT balance FROM Accounts INTO _bal WHERE account_id =111;	SELECT balance FROM Accounts INTO _bal WHERE account_id =111;
UPDATE Accounts SET balance = _bal +100; WHERE account_id=111;	UPDATE Accounts SET balance = _bal + 300; WHERE account_id=111;
COMMIT	COMMIT

Lost Updates

What is the worst that could happen with concurrency?

The case of "Money" you never had!

User 1	User 2
BEGIN	BEGIN

`UPDATE Accounts`
`SET balance = 1000000`
`WHERE account_id=111;`	

	`SELECT balance`
	`FROM Accounts`
	`WHERE account_id=111;`

	COMMIT
ABORT	

Dirty Reads

Serializability

What makes an interleaving of concurrent executions correct?

A transaction schedule shows the sequence of reads and writes of each transaction.

T1	T2
begin	
read(A)	
write(A)	
read(B)	
write(B)	
commit	
	begin
	read(A)
	write(A)
	read(B)
	write(B)
	commit

A *serial schedule* (i.e. no interleaving of operations) is the yardstick of "correct concurrent executions!"
There can be multiple serial executions!

T1	T2
	begin
	read(A)
	write(A)
	read(B)
	write(B)
	commit
begin	
read(A)	
write(A)	
read(B)	
write(B)	
commit	

What makes two schedules equivalent?

- The schedules have the same transactions
- For each transaction, the sequence of actions has the same order
- The before and after state of the DB is the same across the schedules after their execution

T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

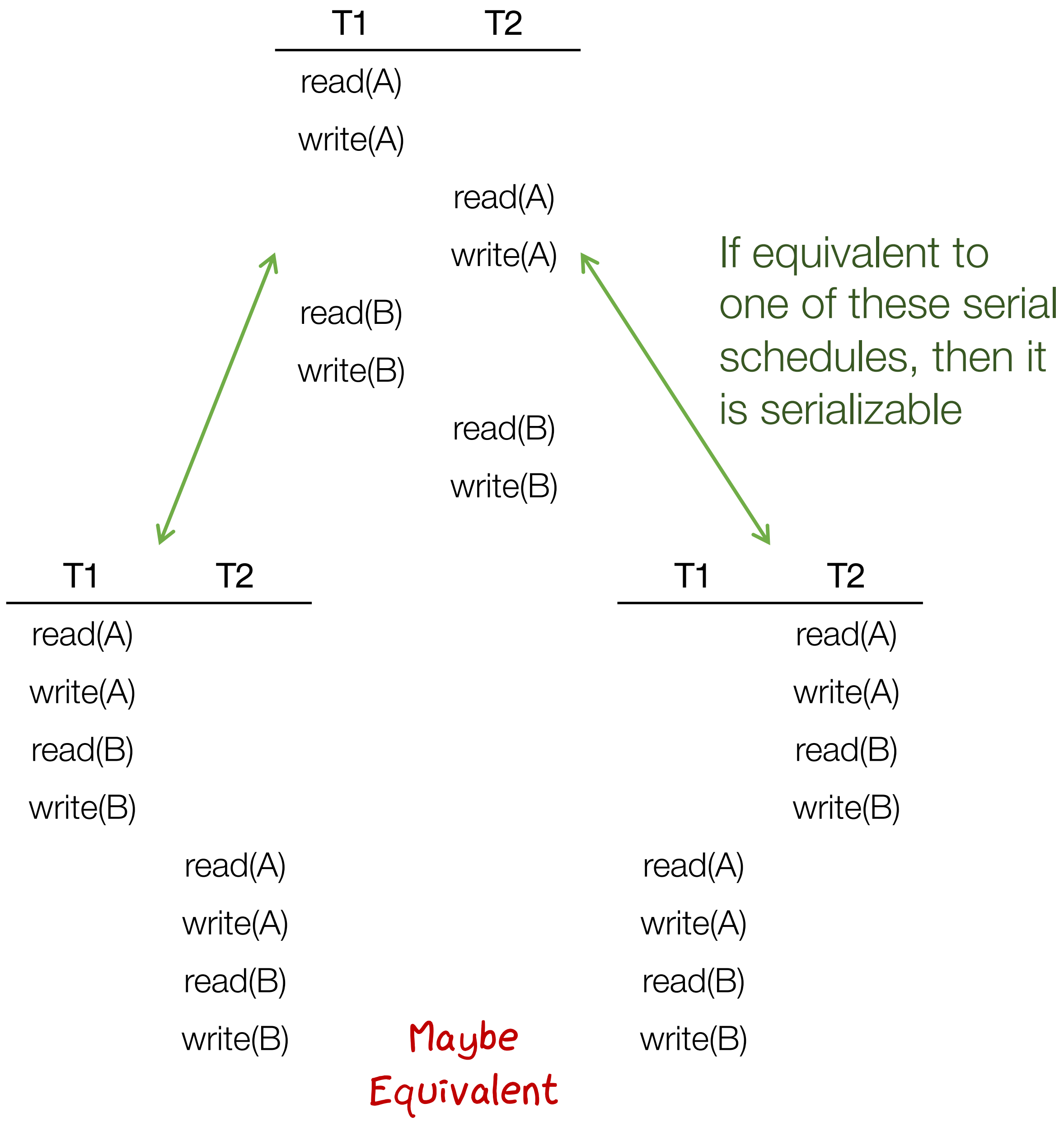
T1	T2
	read(A)
	write(A)
	read(B)
	write(B)
read(A)	
write(A)	
read(B)	
write(B)	

Both are serial

Maybe Equivalent

What makes a schedules serializable?

- The schedule is equivalent to a serial schedule.



T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

If equivalent to one of these serial schedules, then it is serializable

T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T1	T2
	read(A)
	write(A)
	read(B)
	write(B)
read(A)	
write(A)	
read(B)	
write(B)	

Maybe Equivalent

Serializability Example

A, B = 1000
T1 transfers 100$ from A to B
T2 increases amounts in A and B by 10%

T1	T2
read(A)	
A:=A-100	
write(A)	
read(B)	
B:=B+100	
write(B)	
	read(A)
	A:=A*1.1
	write(A)
	read(B)
	B:=B*1.1
	write(B)

A = 990; B=1210

equivalent

T1	T2
read(A)	
A:=A-100	
write(A)	
	read(A)
	A:=A*1.1
	A)
read(B)	
B:=B+100	
write(B)	
	read(B)
	B:=B*1.1
	write(B)

serializable

A = 990; B=1210

T1	T2
	read(A)
	A:=A*1.1
	write(A)
	read(B)
	B:=B*1.1
	write(B)
read(A)	
A:=A-100	
write(A)	
read(B)	
B:=B+100	
write(B)	

A = 1000; B=1200

Conflict Serializability

A, B = 1000
T1 transfers 100$ from A to B
T2 increases amounts in A and B by 10%

T1	T2
read(A)	
A:=A-100	
write(A)	
read(B)	
B:=B+100	
write(B)	
	read(A
	A:=A*1.
	write(A)
	read(B)
	B:=B*1.1
	write(B)

T1	T2
read(A)	
A:=A-100	
write(B)	
	read(B)
	B:=B*1.1
	write(B)

T1	T2
	read(A)
	A:=A*1.1
	te(A)
	d(B)
	B*1.1
	te(B)
write(A)	
read(B)	
B:=B+100	
write(B)	

We need a formal notion of equivalence that can be implemented without checking that the schedules left the database in the same state

A = 990; B=1210

A = 990; B=1210

A = 1000; B=1200

Serializability

Conflicts

Two operations *conflict* if they:

- Are by different transactions,
- Are on the same object,
- At least one of them is a write.

The order of non-conflicting operations has no effect on the final state of the database!

& Interleaved Execution Anomalies

The case of the vanishing 1000$

T1	T2
read(A)	
	read(A)
	A:=A-1000
	write(A)
read(A)	

Read-Write Conflict

Non-repeatable Reads

The case of "Money" you never had!

T1	T2
read(A)	
A:=A+1000,000	
write(A)	
	read(A)
	A:=A-1,000,000
	write(A)
ABORT	

Write-Read Conflict

Dirty Reads

The case of "why did A not get the student account open bonus!"

T1	T2
A:=100	A:=0
B:=100	B:=0
write(A)	
	write(A)
	write(B)
write(B)	

Write-Write Conflict

Overwriting Uncommitted Data / Lost Updates

Conflict Serializability

Conflict Equivalent

Schedules $S_1 \equiv_c S_2$ if:

- They involve the same actions of the same transactions, and
- Every pair of conflicting actions is ordered the same way

Conflict Serializable

S_1 is conflict serializable if $S_1 \equiv_c S_2$ and S_2 is a serial schedule
conflict serializable ⇒ serializable

S_1 is conflict serializable if
You can transform S_1 into a serial schedule S_2 by swapping consecutive non-conflicting operations of different transactions

T1	R (A)	W(A)			R(B)	W(B)		
T2			R(A)	W(A)			R(B)	W(B)

Conflict Serializability

Conflict Equivalent

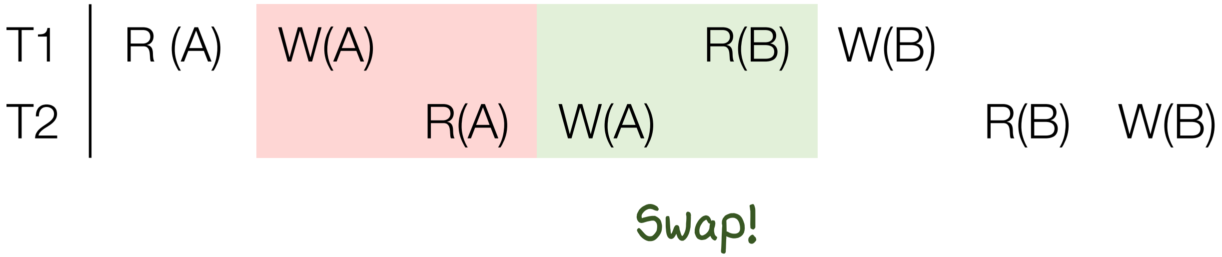
Schedules $S_1 \equiv_c S_2$ if:

- They involve the same actions of the same transactions, and
- Every pair of conflicting actions is ordered the same way

Conflict Serializable

S_1 is conflict serializable if $S_1 \equiv_c S_2$ and S_2 is a serial schedule
conflict serializable ⇒ serializable

S_1 is conflict serializable if
You can transform S_1 into a serial schedule S_2 by swapping consecutive non-conflicting operations of different transactions



T1	R (A)	W(A)			R(B)	W(B)		
T2			R(A)	W(A)			R(B)	W(B)

Swap!

Conflict Serializability

Conflict Equivalent

Schedules $S_1 \equiv_c S_2$ if:

- They involve the same actions of the same transactions, and
- Every pair of conflicting actions is ordered the same way

Conflict Serializable

S_1 is conflict serializable if $S_1 \equiv_c S_2$ and S_2 is a serial schedule
conflict serializable ⇒ serializable

S_1 is conflict serializable if
You can transform S_1 into a serial schedule S_2 by swapping consecutive non-conflicting operations of different transactions

T1	R (A)	W(A)		R(B)		W(B)		
T2			R(A)		W(A)		R(B)	W(B)

Swap!

Conflict Serializability

Conflict Equivalent

Schedules $S_1 \equiv_c S_2$ if:

- They involve the same actions of the same transactions, and
- Every pair of conflicting actions is ordered the same way

Conflict Serializable

S_1 is conflict serializable if $S_1 \equiv_c S_2$ and S_2 is a serial schedule
conflict serializable ⇒ serializable

S_1 is conflict serializable if
You can transform S_1 into a serial schedule S_2 by swapping consecutive non-conflicting operations of different transactions

T1	R (A)	W(A)	R(B)			W(B)		
T2				R(A)	W(A)		R(B)	W(B)

Swap!

Conflict Serializability

Conflict Equivalent

Schedules $S_1 \equiv_c S_2$ if:

- They involve the same actions of the same transactions, and
- Every pair of conflicting actions is ordered the same way

Conflict Serializable

S_1 is conflict serializable if $S_1 \equiv_c S_2$ and S_2 is a serial schedule
conflict serializable ⇒ serializable

S_1 is conflict serializable if
You can transform S_1 into a serial schedule S_2 by swapping consecutive non-conflicting operations of different transactions

T1	R (A)	W(A)	R(B)	W(B)				
T2					R(A)	W(A)	R(B)	W(B)

SERIAL!

Conflict Serializability

T1	R (A)			W(A)
T2		R(A)	W(A)	

S_1 is conflict serializable if
You can transform $\mathbf{S_1}$ into a serial schedule $\mathbf{S_2}$ by swapping consecutive non-conflicting operations of different transactions

NOT CONFLICT SERIALIZABLE!

This definition is operational but does not give us the most efficient test of conflict serializability. We need a faster algorithm!

Conflict Dependency Graphs

Conflict Serializability

Dependency Graph $G(S)$

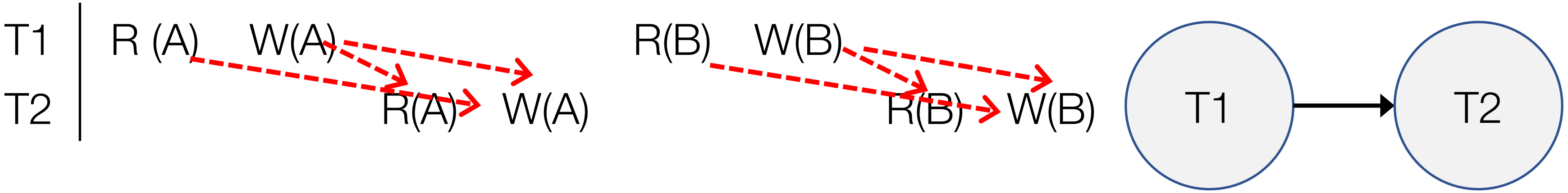
Each transaction T_i is a node

An edge from T_i to T_i exists if:

- An operation O_i of T_i conflicts with an operation O_j of T_j and
- O_i appears earlier in the schedule than O_j

Conflict Serializable

S is conflict serializable iff $G(S)$ is *acyclic*



No Cycles

CONFLICT SERIALIZABLE!

Cycle!

NOT CONFLICT SERIALIZABLE!

Two-Phase Locking (2PL)

2PL → Conflict Serializability

Pessimistic
Concurrency
Control Protocol

Assumes conflicts will occur, requires transactions to lock the items they will access before access!

Rules:

- Xact gets S (shared) lock before reading, and an X (exclusive) lock before writing.
- Xact cannot get new locks after releasing any lock

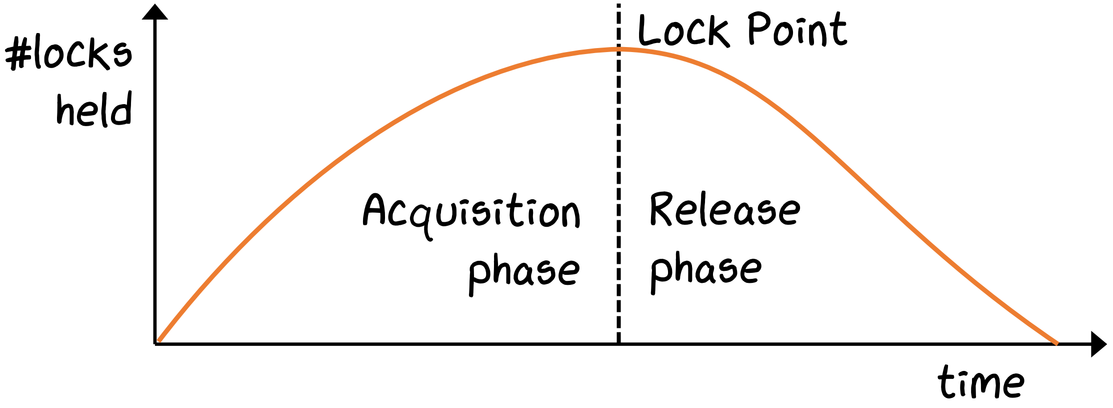
Lock
Compatibility
Matrix

	S	X
S	✓	✗
X	✗	✗

Multiple transactions can get a shared lock on one object but only one can get an "exclusive" lock

2PL → Conflict Serializability

Why?



At lock point, transaction has everything it needs.

Conflicting concurrent transactions either:

- Started release before lock point
- Blocked and waiting for release of some locks

What is the equivalent serial schedule?

- Two conflicting transactions are ordered by the lock point
- The order of the lock points is the equivalent serial schedule

2PL Issues

Cascading Aborts

T1	R(A)	W(A)			ABORT
T2			R(A)	W(A)	

Rolling back Xact T1 rolls back T2!

Strict 2PL

Lock Management

- Who issues and manages locks on items in the database?
- What items do we lock?

The Lock Manager

Multi-Lock Granularity

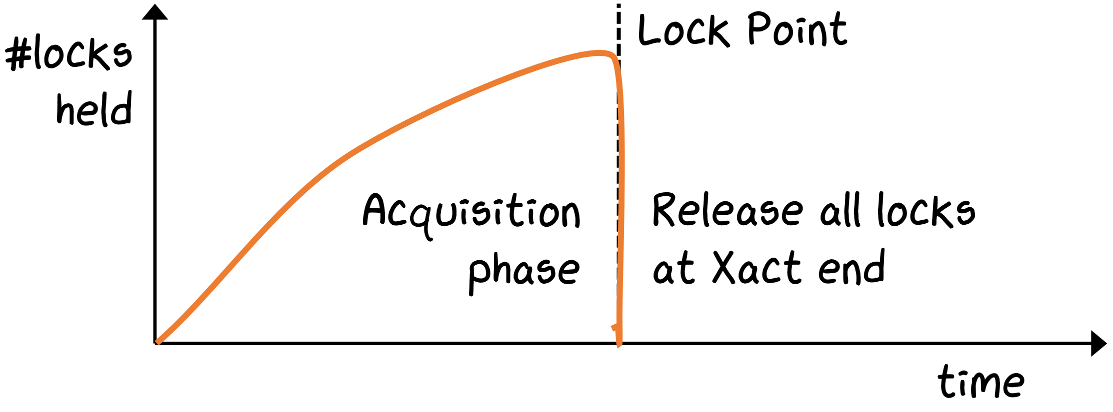
Deadlocks

T1 has a lock on A; T2 has a lock on B
T1 wants a lock on B; T2 wants a lock on A

Deadlock Avoidance, Prevention, Detection + Resolution

Strict 2PL

Strict 2PL → Conflict Serializability + No Cascading Aborts



Strict 2PL = 2PL + release all locks when:

- Transaction committed (all writes are now durable)
- Transaction aborted (all writes undone)

→ *No cascading aborts*

Conflicting transactions blocked and waiting for locks release

→ *Conflict Serializability*

2PL & Strict 2PL in Action

Lock, Access, & Release

A has 100$, B has 50$

T1 transfers 10$ from account A to B.

T2 sums the amounts in A and B.

What does T2 output?

T1	T2	
Lock-X(A)		
Read(A)		A: 100
	Lock-S(A)	
A := A-10		
Write(A)		
Unlock(A)		
	Read(A)	A: 90
	Unlock(A)	
	Lock-S(B)	
Lock-X(B)		
	Read(B)	B: 50
	Unlock(B)	
	Print (A+B)	140
Read(B)		B: 50
B := B+10		
Write(B)		
Unlock(B)		

2PL

A has 100$, B has 50$

T1 transfers 10$ from account A to B.

T2 sums the amounts in A and B.

What does T2 output?

T1	T2	
Lock-X(A)		
Read(A)	Lock-S(A)	A: 100
A: = A-10		
Write(A)		
Lock-X(B)		
Unlock(A)		
	Read(A)	A: 90
	Lock-S(B)	B: 50
Read(B)		
B := B +10		
Write(B)		
Unlock(B)		
	Unlock(A)	
	Read(B)	B: 60
	Unlock(B)	
	Print (A+B)	150

Strict 2PL

A has 100$, B has 50$

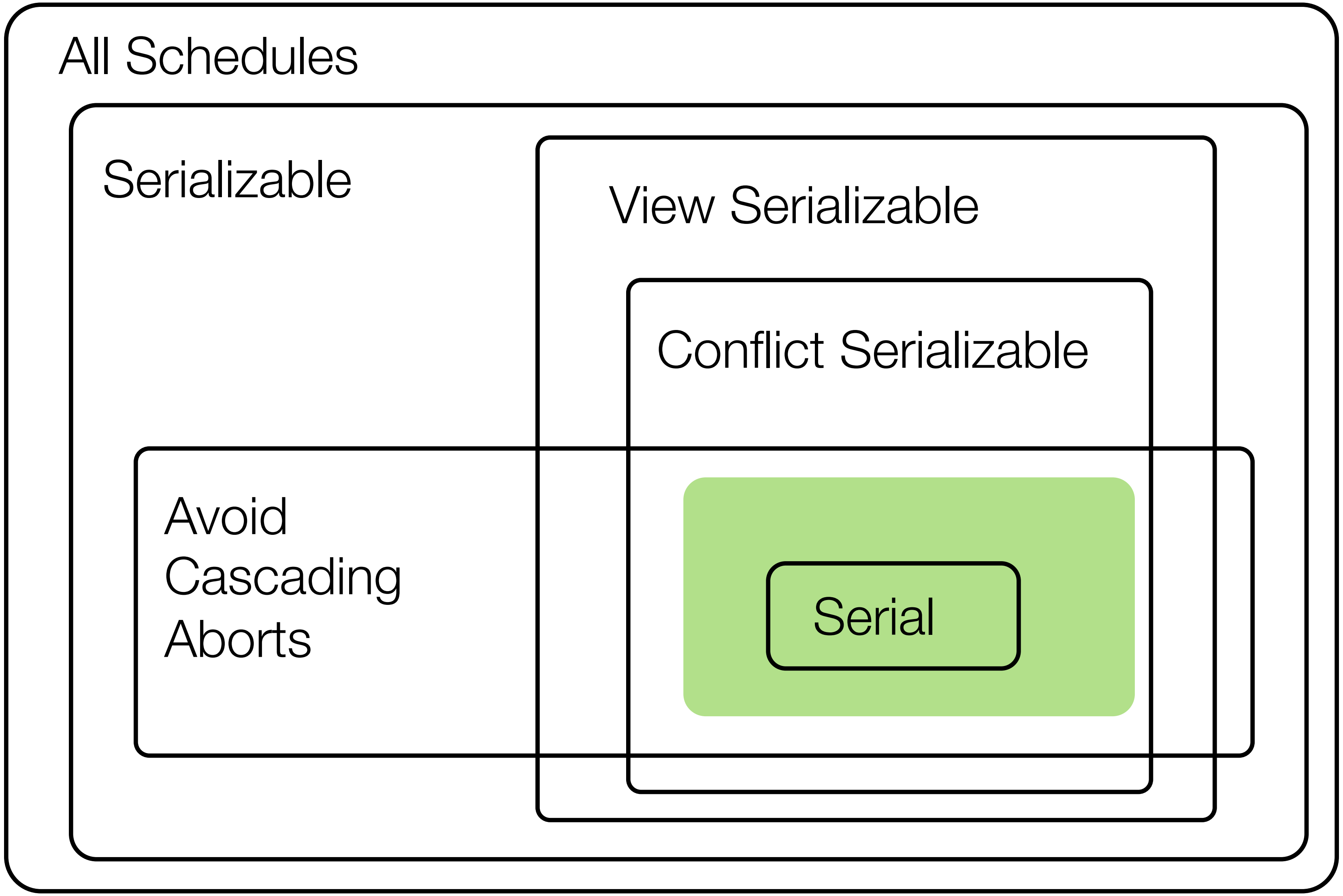
T1 transfers 10$ from account A to B.

T2 sums the amounts in A and B.

What does T2 output?

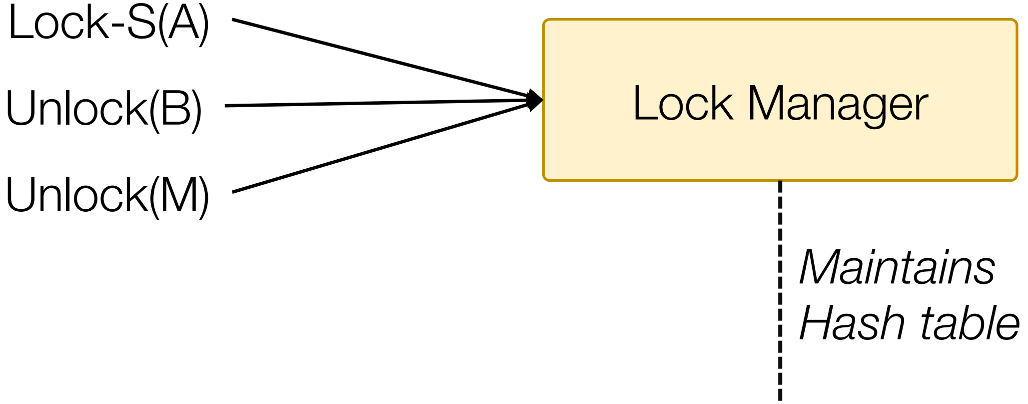
T1	T2	
Lock-X(A)		
Read(A)		A: 100
	Lock-S(A)	
A: = A-10		
Write(A)		
Lock-X(B)		
Read(B)		B: 50
B := B +10		
Write(B)		
Unlock(A)		
Unlock(B)		
	Read(A)	A: 90
	Lock-S(B)	
	Read(B)	B: 60
	Print (A+B)	
	Unlock(A)	
	Unlock(B)	150

2PL Schedules



Lock Manager

How Do We Lock Data?



Item	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3 ← T7 ← T6
B	{T4}	X	T8 ← T5
		…	

(FIFO/priority given to upgraders)

Lock Request/Upgrade

Does requesting Xact conflict with Xacts in granted set?

- NO: Put into "granted set" and let proceed
- YES: Put to sleep in wait queue

Unlock

Move first Xact (and all non-conflicting Xacts) from wait queue if any to granted set and wake them up

Lock Manager State

T1	T2
Lock-X(A)	
	Lock-S(B)
	Read(B)
	Lock-S(A)
Read(A)	
A: = A-50	
Write(A)	
Lock-X(B)	

Item	Granted Set	Mode	Wait Queue
A	{T1}	X	

Item	Granted Set	Mode	Wait Queue
A	{T1}	X	
B	{T2}	S	

Item	Granted Set	Mode	Wait Queue
A	{T1}	X	T2(S)
B	{T2}	S	

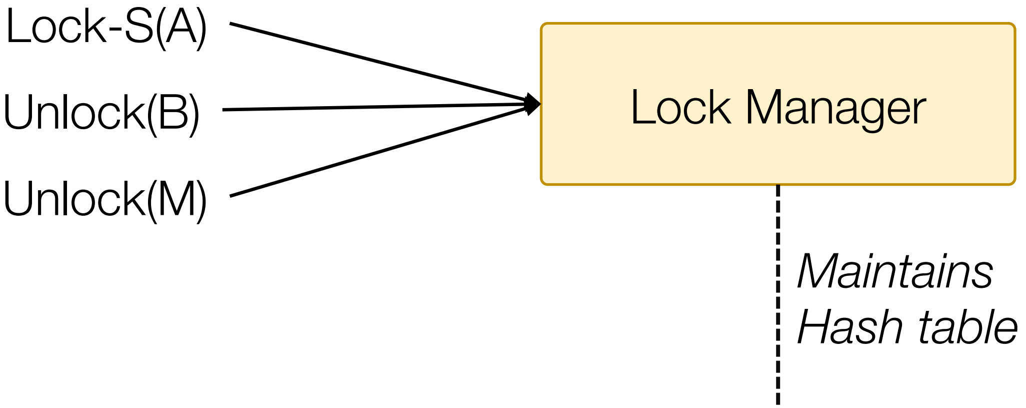
Item	Granted Set	Mode	Wait Queue
A	{T1}	X	T2(S)
B	{T2}	S	T1(X)

What if T1, T2 touch millions of records in a table?

How do we handle deadlocks?

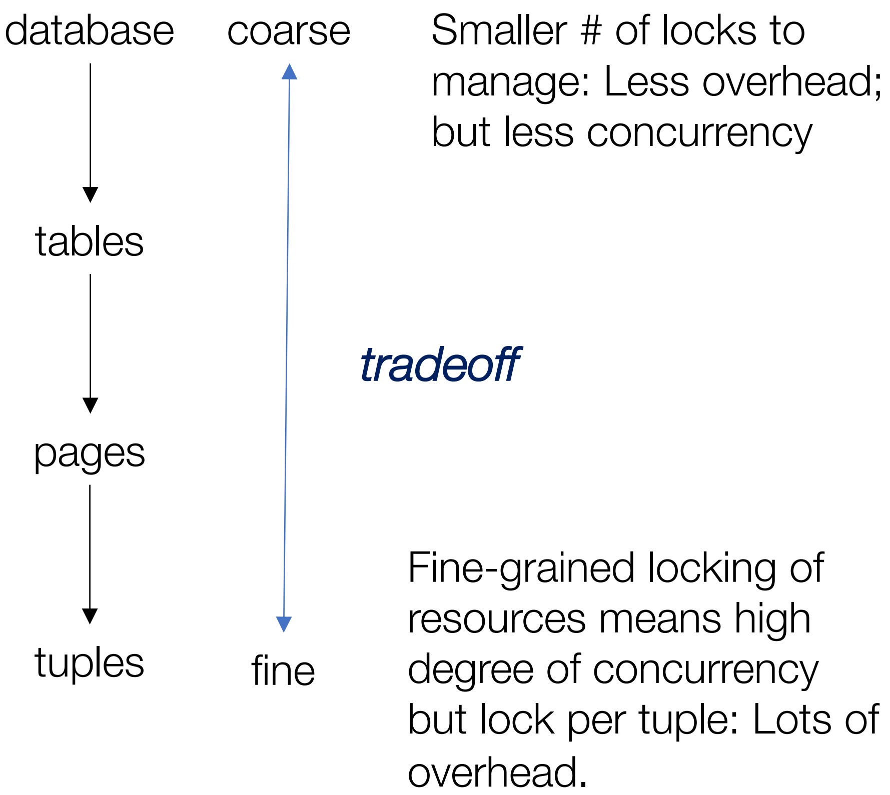
Lock Granularity

Multiple Lock Granularity



Item	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3 ← T7 ← T6
B	{T4}	X	T8 ← T5

Lock manager maintains a lock for each item locked by a transaction...

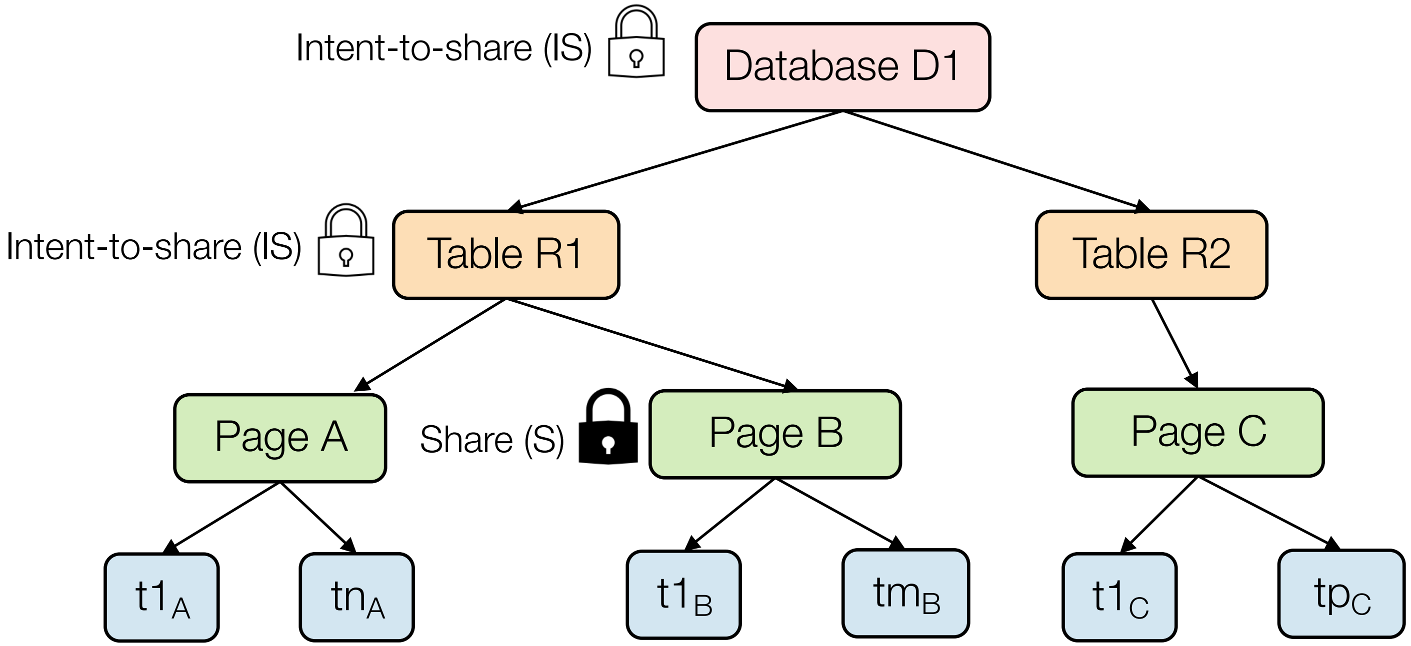


Smaller # of locks to manage: Less overhead; but less concurrency

Fine-grained locking of resources means high degree of concurrency but lock per tuple: Lots of overhead.

Multiple Locking Granularity

- Establish a ***hierarchy*** of DB objects.
- Allow Xact to lock a node in the tree ***explicitly*** (e.g. a page)
- This ***implicitly*** locks all the node's descendants in the same mode (e.g. tuples in the page).



Problem

Can I immediately lock a table if it has no locks?

No, must check lower levels for locks!

Solution: Intention Locks

To get S or X lock on an object (e.g. a tuple), Xact must have proper ***intent*** locks on all its ancestors in the granularity hierarchy (e.g. page, table and database).

3 new lock modes:

- *IS:* Intent to get S lock(s) at finer granularity.
- *IX:* Intent to get X lock(s) at finer granularity.
- *SIX:* Like S & IX at the same time.

2PL + Multiple Locking Granularity

Request

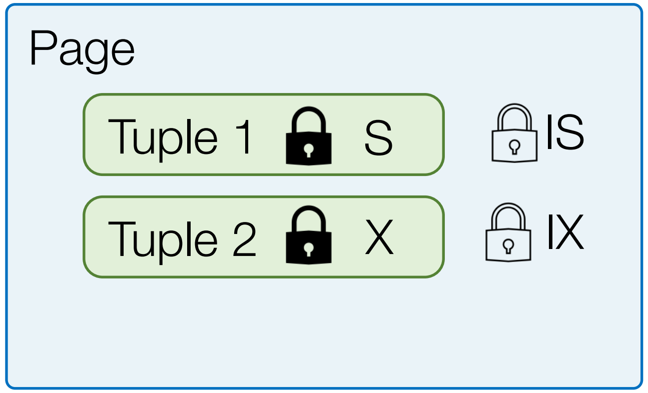
- Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.

Release

- Release locks in bottom-up order.

2PL and lock compatibility matrix rules enforced

How to know if two locks are compatible?


Page

Tuple 1 S

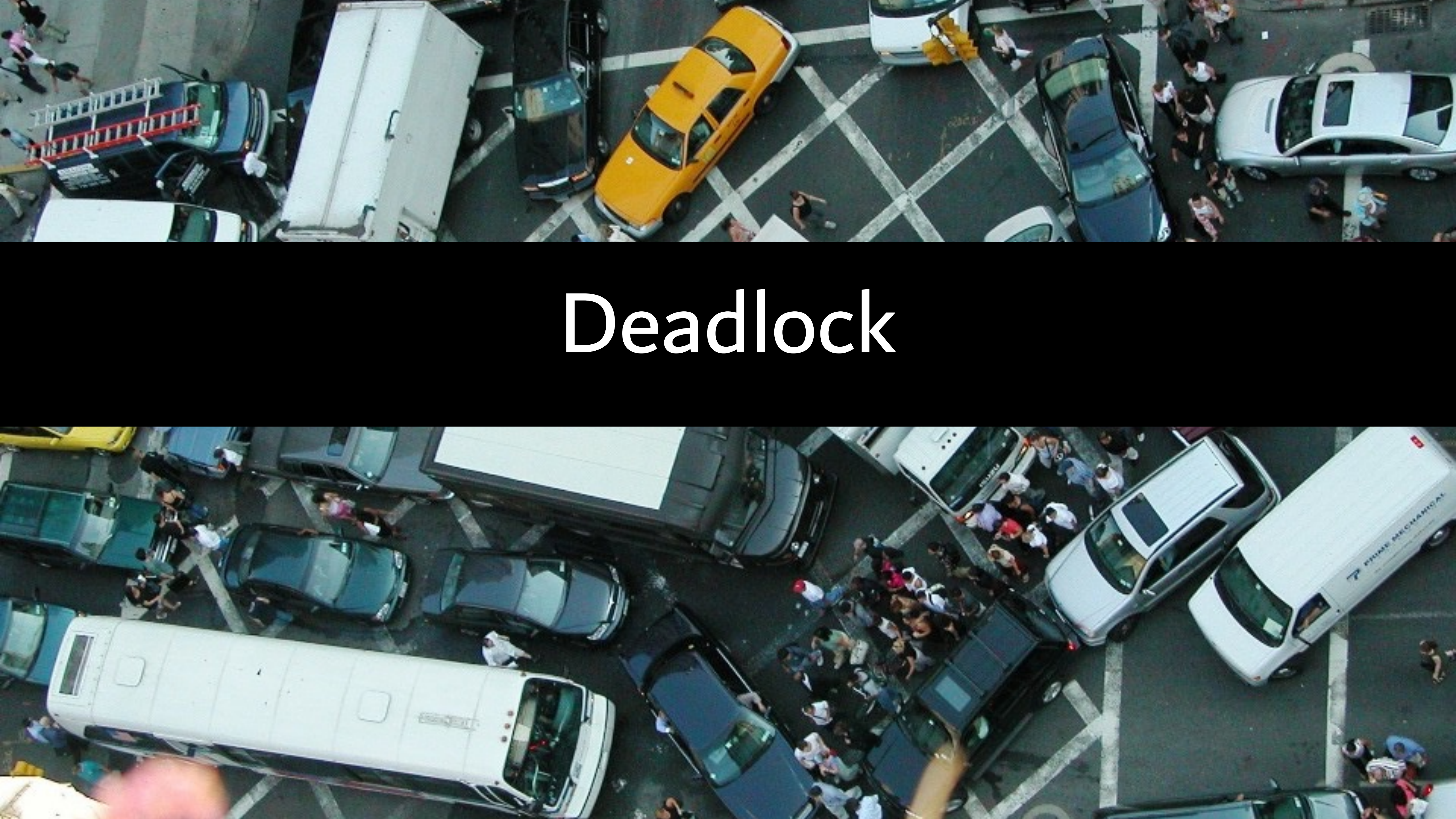
IS

Tuple 2 X

IX

Lock Compatibility Matrix

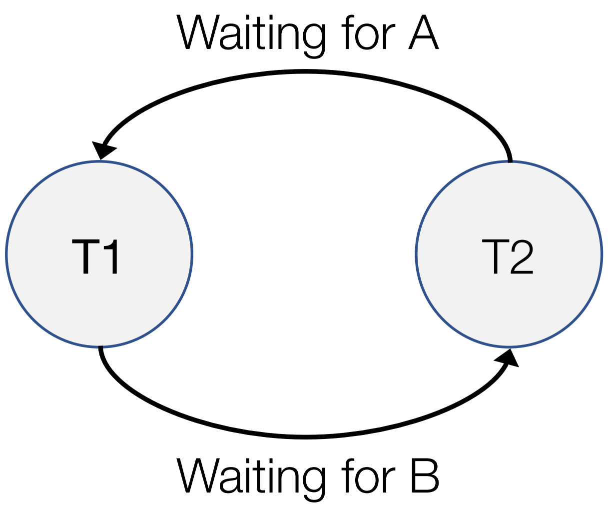
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗



Deadlock

How do deadlocks arise?

	Granted Set	Mode	Wait Queue
A	{T1}	X	T2(S)
B	{T2}	S	T1(X)



1. Mutual Exclusion
2. Hold and wait!
3. No Preemption
4. Circular wait

	Granted Set	Mode	Wait Queue
A	{T2}	S	T3(X) ←T4(X) ←T2(X)

Bad Implementation! Waiting on myself

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T2(X) ←T1(X) ←T3(X) ← T4(X)

Multiple lock upgrades

Dealing with Deadlocks

Do nothing!

Eventually the application will abort the long-running transaction and try again!

We could do better!

Timeout & Kill

Observe the current Xacts, kill ones that have been running for a while

What if we have a long-running one?

IBM DB2
Distributed DBMS

Prevention

1. ~~Mutual Exclusion~~
2. ~~Hold and wait!~~
3. No Preemption
4. Circular wait

Xact T_l holds a lock

- Wait-Die: $T_i > T_l$, T_i waits for T_l; else T_i dies (aborts)
- Wound-Wait: $T_i > T_l$, T_l is wounded (aborts); else T_i waits

Unnecessary termination to prevent a rare occurrence!

Resource Ordering

- Can only lock DB objects in a certain order

How can a DBMS force an order on how tuples are locked?

Advice found in many manuals

Detection & Resolution

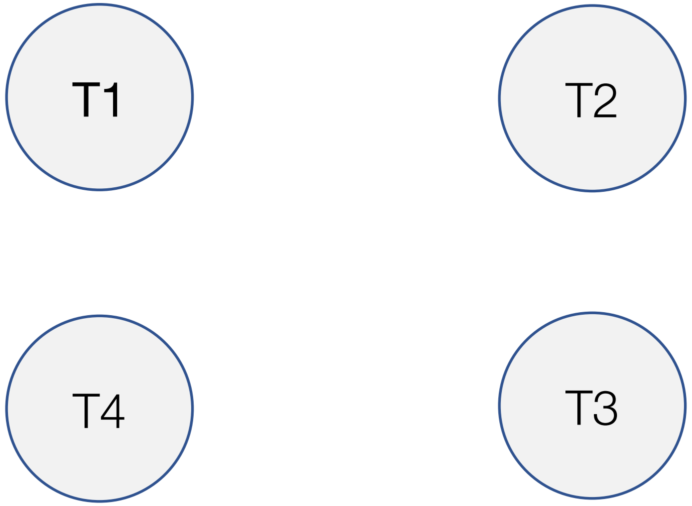
Maintain a ***waits-for-graph***, periodically look for cycles, abort a victim to break they cycle.

Used in MySQL, Postgres, Oracle, ...

Deadlock Detection in Action

T1	S (A)	S(D)	
T2			X(B)
T3			
T4			

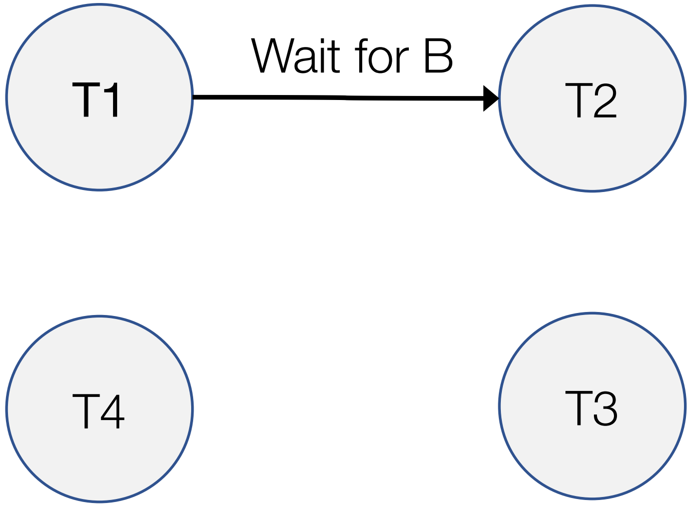
	Granted Set	Mode	Wait Queue
A	{T1}	S	
D	{T1}	S	
B	{T2}	X	



Deadlock Detection in Action

T1	S (A)	S(D)		S(B)
T2			X(B)	
T3				
T4				

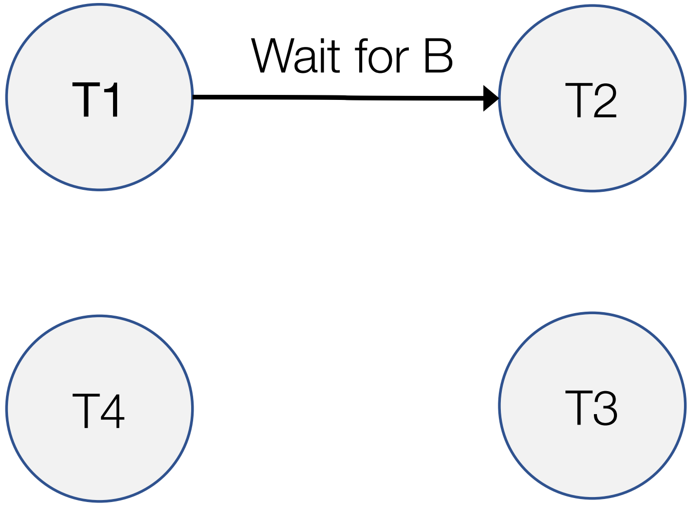
	Granted Set	Mode	Wait Queue
A	{T1}	S	
D	{T1}	S	
B	{T2}	X	T1(S)
C			



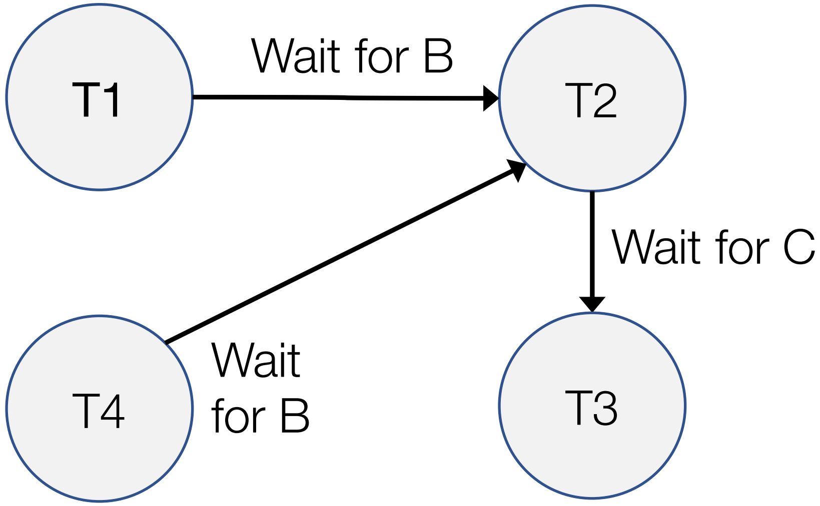
Deadlock Detection in Action

T1	S (A)	S(D)		S(B)		
T2			X(B)			
T3					S(D)	S(C)
T4						

	Granted Set	Mode	Wait Queue
A	{T1}	S	
D	{T1}	S	
B	{T2}	X	T1(S)
C	{T3}	S	



T1	S (A)	S(D)		S(B)				
T2			X(B)				X(C)	
T3					S(D)	S(C)		
T4								X(B)



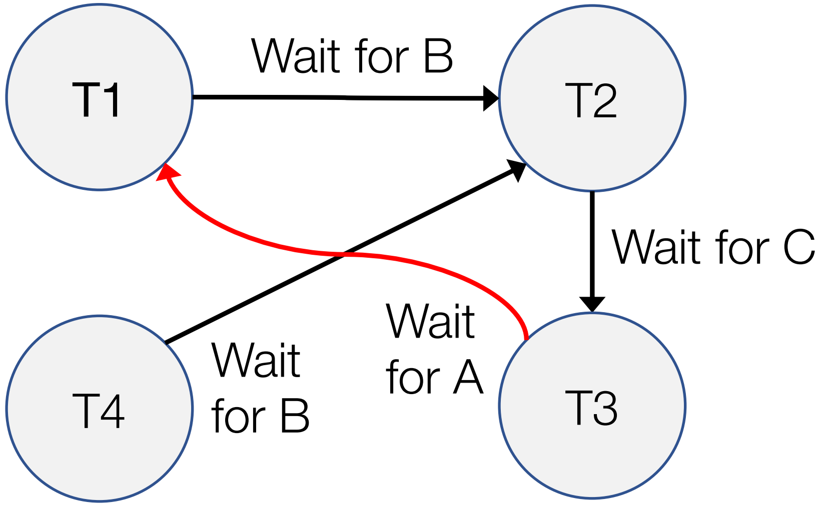
	Granted Set	Mode	Wait Queue
A	{T1}	S	
D	{T1,T3}	S	
B	{T2}	X	T1(S) ←T4(X)
C	{T3}	S	T2(X)

Deadlock Detection in Action

Deadlock Detection in Action

T1	S (A)	S(D)		S(B)					
T2			X(B)				X(C)		
T3					S(D)	S(C)			X(A)
T4								X(B)	

	Granted Set	Mode	Wait Queue
A	{T1}	S	T3(X)
D	{T1,T3}	S	
B	{T2}	X	T1(S) ←T4(X)
C	{T3}	S	T2(X)

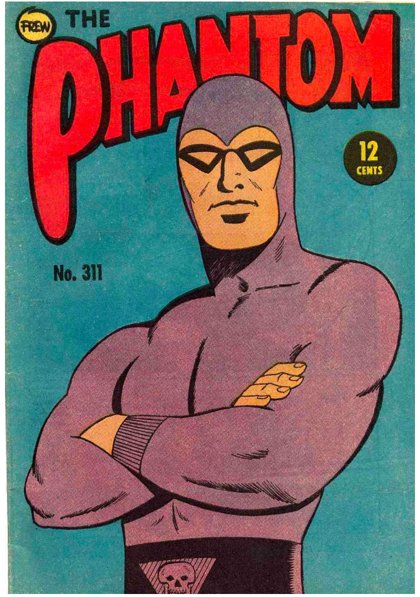


Deadlock!

For Your Information: Indexes

- **2PL on B+ tree pages is a rotten idea.**
  - Think about the first thing you would lock, and how that affects other xacts!
- **Instead, do short locks (latches) in a clever way**
  - Idea: Upper levels of B+ tree just need to direct traffic correctly. Don’t need serializability or 2PL!
  - Different tricks to exploit this
    - The *B-link* tree is elegant
    - The *Bw-tree* is a recent variant for main memory DBs
- **Note: this is pretty complicated!**

For Your Information: Phantoms



- **Suppose you query for sailors with rating between 10 and 20, using an Alternative 2 B+ tree**
  - You set tuple-level locks in the Heap File
- **I insert “Dread Pirate Roberts”, with rating 12**
- **You do your query again via the index**
  - Yikes! A phantom
- **Problem: Serializability assumed a static DB**
- **What we want: lock the logical range 10-20**
  - Hard to imagine that lock table! Doesn’t work well.
- **What is done: set locks in indexes cleverly**
  - So-called “next key locking”

Summary, cont.

- **Correctness criterion for isolation is “serializability”.**
  - In practice, we use “conflict serializability” which is conservative but easy to enforce
- **Two Phase Locking and Strict 2PL: Locks implement the notions of conflict directly**
  - The lock manager keeps track of the locks issued.
  - **Deadlocks** may arise; can either be prevented or detected.
- **Multi-Granularity Locking:**
  - Allows flexible tradeoff between lock “scope” in DB, and # of lock entries in lock table
- **More to the story**
  - Optimistic/Multi-version/Timestamp CC
  - Index “latching”, phantoms
  - Actually, there’s much much more :-)