Transactions: Pessimistic

CSE 4/562: Database Systems | Lecture 19

DB. Sys.: T.C.B.: Ch. 18.1-18.2, 19.1

Announcements

1. Checkpoint 4 released
2. Schedule checkpoint 3 code reviews

Quiz

Conflict Equivalence

Two schedules are conflict equivalent if there is a sequence of pairwise “flips” of reads or operations on different objects that transforms one schedule into the other.

Conflict Serializability

A schedule is conflict serializable if it is conflict equivalent to a serial schedule.

How do we decide whether a schedule is conflict serializable?

Conflicts

write/write, read/write, and write/read operation pairs can't be flipped.

Let's give these types of operation pairs a name: **Conflicts**[1].

[1] A note on language: A conflict does **not** indicate a problem.

Detecting Conflict Serializability

Time	T1	T2
↓	W(A)	
↓		W(A)

Detecting Conflict Serializability

Time	T1	T2
↓	W(A)	
↓		W(A)

This conflict requires that T1 **happens before** T2

Detecting Conflict Serializability

Conflicts create a partial order over the conflict-equivalent serial schedules.

Detecting Conflict Serializability

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

In this schedule...

Detecting Conflict Serializability

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

In this schedule...

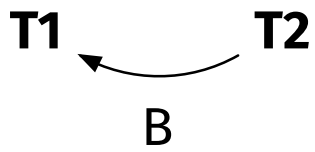
- **T2**′s write to B “happens before” **T1**′s read.

Detecting Conflict Serializability

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

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.

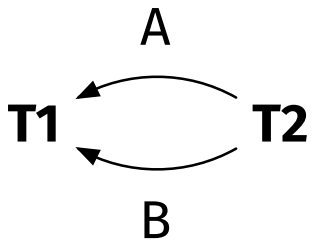


Detecting Conflict Serializability

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

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.
- **T2**'s write to A "happens before" **T1**'s write.

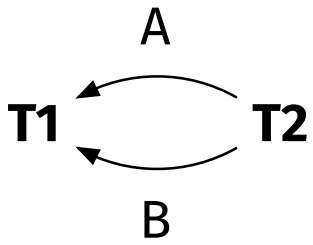


Detecting Conflict Serializability

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

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.
- **T2**'s write to A "happens before" **T1**'s write.



No cycles in the "conflict graph"
Schedule is Conflict Serializable

Detecting Conflict Serializability

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

In this schedule...

Detecting Conflict Serializability

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

In this schedule...

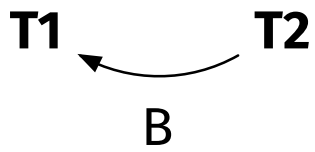
- **T2**′s write to B “happens before” **T1**′s read.

Detecting Conflict Serializability

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

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.

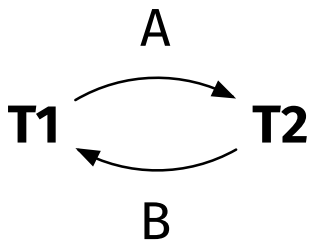


Detecting Conflict Serializability

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

In this schedule...

- **T2**′s write to B “happens before” **T1**′s read.
- **T1**′s write to A “happens before” **T2**′s write.

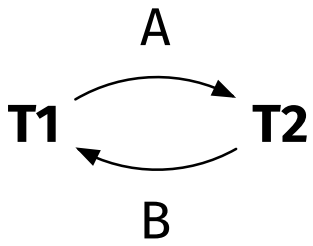


Detecting Conflict Serializability

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

In this schedule...

- **T2**'s write to B "happens before" **T1**'s read.
- **T1**'s write to A "happens before" **T2**'s write.



Cycle = Schedule is **not** Conflict Serializable

Detecting Conflict Serializability

Time	T1	T2	T3
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓			W(A)
↓	W(B)		

In this schedule...

T1 **T2**

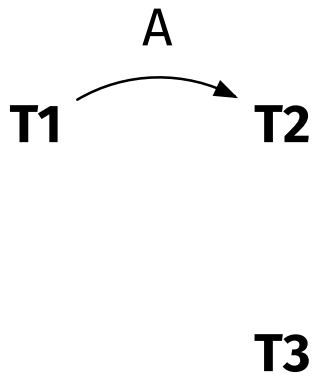
T3

Detecting Conflict Serializability

Time	T1	T2	T3
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓			W(A)
↓	W(B)		

In this schedule...

- **T1**'s read of A "happens before" **T2**'s write

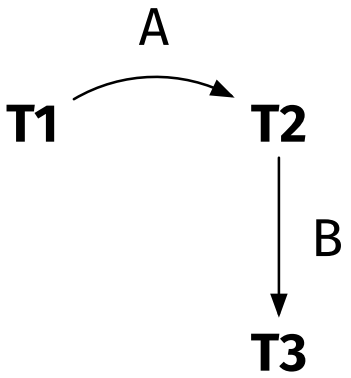


Detecting Conflict Serializability

Time	T1	T2	T3
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓			W(A)
↓	W(B)		

In this schedule...

- **T1**'s read of A "happens before" **T2**'s write
- **T2**'s read of B "happens before" **T3**'s write

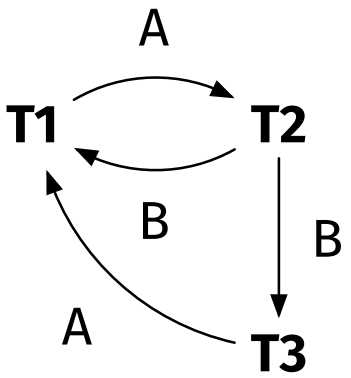


Detecting Conflict Serializability

Time	T1	T2	T3
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓			W(A)
↓	W(B)		

In this schedule...

- **T1**'s read of A "happens before" **T2**'s write
- **T2**'s read of B "happens before" **T3**'s write
- **T2**'s write to B and **T3**'s write to A "happen before" **T1**'s write



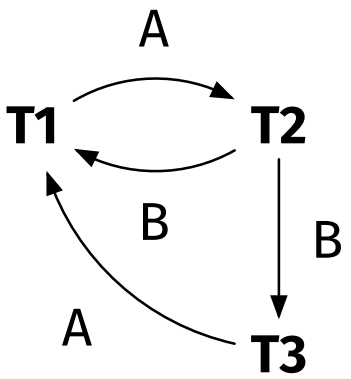
Detecting Conflict Serializability

Time	T1	T2	T3
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓			W(A)
↓	W(B)		

Cycle!

In this schedule...

- **T1**'s read of A "happens before" **T2**'s write
- **T2**'s read of B "happens before" **T3**'s write
- **T2**'s write to B and **T3**'s write to A "happen before" **T1**'s write



Happens Before (aka Dependency) Graphs

1. Add one node for every transaction
2. For every pair of operations, add an edge from the later operation's transaction to the earlier operation's transaction if:
   - ... the operations are from different transactions
   - ... the operations act on the same object
   - ... at least one operation is a write

If the resulting schedule is acyclic, the schedule is **conflict serializable**

Side note...

If the schedule is conflict serializable, it is serializable.

If the schedule is not conflict serializable, it may still be serializable...
... but we don't have the tools to prove it

Side note 2...

You will be expected to know whether a schedule is (conflict) serializable...

Side note 2...

You will be expected to know whether a schedule is (conflict) serializable...

... but this is usually not the point.

Side note 2...

You will be expected to know whether a schedule is (conflict) serializable...

... but this is usually not the point.

By learning to work with serializability, we can prove to ourselves that a new concurrency control scheme can **guarantee** that it will only ever permit transactions to execute according to a (conflict) serializable schedule.

Allow concurrency, but...

Locking (Pessimistic)
... pause a transaction before it ever risks non-serializable behavior.

Snapshot Isolation (Optimistic)
... check whether a transaction had a serializable schedule prior to committing and abort it if not.

Timestamp Concurrency Control (Optimistic)
... detect serializability violations while the transaction is running and abort.

Allow concurrency, but...

Locking (Pessimistic)

... pause a transaction before it ever risks non-serializable behavior.

Snapshot Isolation (Optimistic)

... check whether a transaction had a serializable schedule prior to committing and abort it if not.

Timestamp Concurrency Control (Optimistic)

... detect serializability violations while the transaction is running and abort.

Observation: All conflicts require access to the same object.

Idea: When a second transaction tries to access an object, block it until the first `COMMIT`s or `ABORT`s first.

Idea: When a second transaction tries to access an object, block it until the first `COMMIT`s or `ABORT`s first.

Locking

1. A transaction must **Lock** every object it accesses before the access
2. All locks are released when the transaction `COMMIT`s or `ABORT`s

Time	T1	T2	T3
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓		COMMIT	
↓			W(A)
↓			COMMIT
↓	W(B)		
↓	COMMIT		

Time	T1	T2	T3
↓	Lock(A)		
↓	R(A)		
↓		R(B)	
↓		W(A)	
↓		COMMIT	
↓			W(A)
↓			COMMIT
↓	W(B)		
↓	COMMIT		

Time	T1	T2	T3
↓	Lock(A)		
↓	R(A)		
↓		Lock(B)	
↓		R(B)	
↓		W(A)	
↓		COMMIT	
↓			W(A)
↓			COMMIT
↓	W(B)		
↓	COMMIT		

Time	T1	T2	T3
↓	Lock(A)		
↓	R(A)		
↓		Lock(B)	
↓		R(B)	
↓		Lock(A)	
↓		W(A)	
↓		COMMIT	
↓			W(A)
↓			COMMIT
↓	W(B)		
↓	COMMIT		

Time	T1	T2	T3
↓	Lock(A)		
↓	R(A)		
↓		Lock(B)	
↓		R(B)	
↓		😢	
↓		W(A)	
↓		COMMIT	
↓			W(A)
↓			COMMIT
↓	W(B)		
↓	COMMIT		

Time	T1	T2	T3
↓	Lock(A)		
↓	R(A)		
↓		Lock(B)	
↓		R(B)	
↓		😢	
↓		W(A)	
↓		COMMIT	
↓			W(A)
↓			COMMIT
↓	W(B)		
↓	COMMIT		

This schedule is not permitted!

Time	T1	T2	T3	Locks
↓	R(A)			
↓	W(B)			
↓	COMMIT			
↓		R(B)		
↓			W(A)	
↓			COMMIT	
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	W(B)			
↓	COMMIT			
↓		R(B)		
↓			W(A)	
↓			COMMIT	
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			
↓		R(B)		
↓			W(A)	
↓			COMMIT	
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		R(B)		
↓			W(A)	
↓			COMMIT	
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		Lock(B)		**T1** holds ∅, **T2** holds { B }
↓		R(B)		
↓			W(A)	
↓			COMMIT	
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		Lock(B)		**T1** holds ∅, **T2** holds { B }
↓		R(B)		
↓			Lock(A)	**T1** holds ∅, **T2** holds { B }, **T3** holds { A }
↓			W(A)	
↓			COMMIT	
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		Lock(B)		**T1** holds ∅, **T2** holds { B }
↓		R(B)		
↓			Lock(A)	**T1** holds ∅, **T2** holds { B }, **T3** holds { A }
↓			W(A)	
↓			COMMIT	**T1** holds ∅, **T2** holds { B }, **T3** holds ∅
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		Lock(B)		**T1** holds ∅, **T2** holds { B }
↓		R(B)		
↓			Lock(A)	**T1** holds ∅, **T2** holds { B }, **T3** holds { A }
↓			W(A)	
↓			COMMIT	**T1** holds ∅, **T2** holds { B }, **T3** holds ∅
↓		Lock(A)		**T1** holds ∅, **T2** holds { B, A }, **T3** holds ∅
↓		W(A)		
↓		COMMIT		

Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		Lock(B)		**T1** holds ∅, **T2** holds { B }
↓		R(B)		
↓			Lock(A)	**T1** holds ∅, **T2** holds { B }, **T3** holds { A }
↓			W(A)	
↓			COMMIT	**T1** holds ∅, **T2** holds { B }, **T3** holds ∅
↓		Lock(A)		**T1** holds ∅, **T2** holds { B, A }, **T3** holds ∅
↓		W(A)		
↓		COMMIT		**T1** holds ∅, **T2** holds ∅, **T3** holds ∅

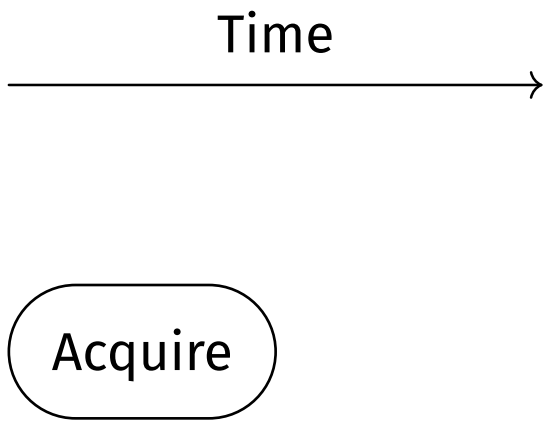
Time	T1	T2	T3	Locks
↓	Lock(A)			**T1** holds { A }
↓	R(A)			
↓	Lock(B)			**T1** holds { A, B }
↓	W(B)			
↓	COMMIT			**T1** holds ∅
↓		Lock(B)		**T1** holds ∅, **T2** holds { B }
↓		R(B)		
↓			Lock(A)	**T1** holds ∅, **T2** holds { B }, **T3** holds { A }
↓			W(A)	
↓			COMMIT	**T1** holds ∅, **T2** holds { B }, **T3** holds ∅
↓		Lock(A)		**T1** holds ∅, **T2** holds { B, A }, **T3** holds ∅
↓		W(A)		
↓		COMMIT		**T1** holds ∅, **T2** holds ∅, **T3** holds ∅

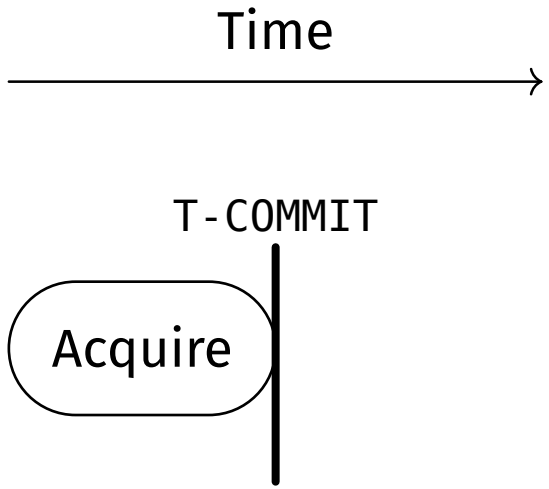
This schedule **is** permitted!

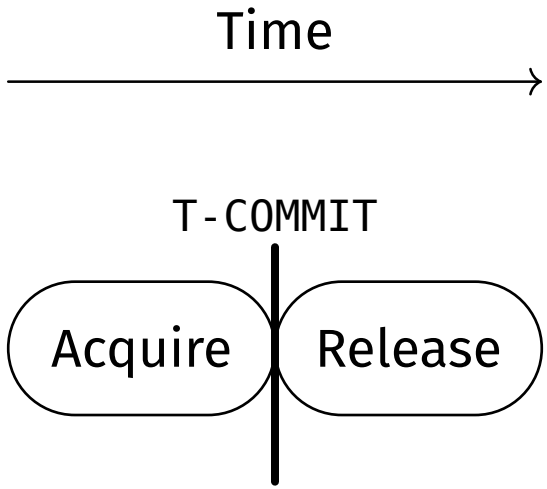
Locking is usually used to protect critical sections.

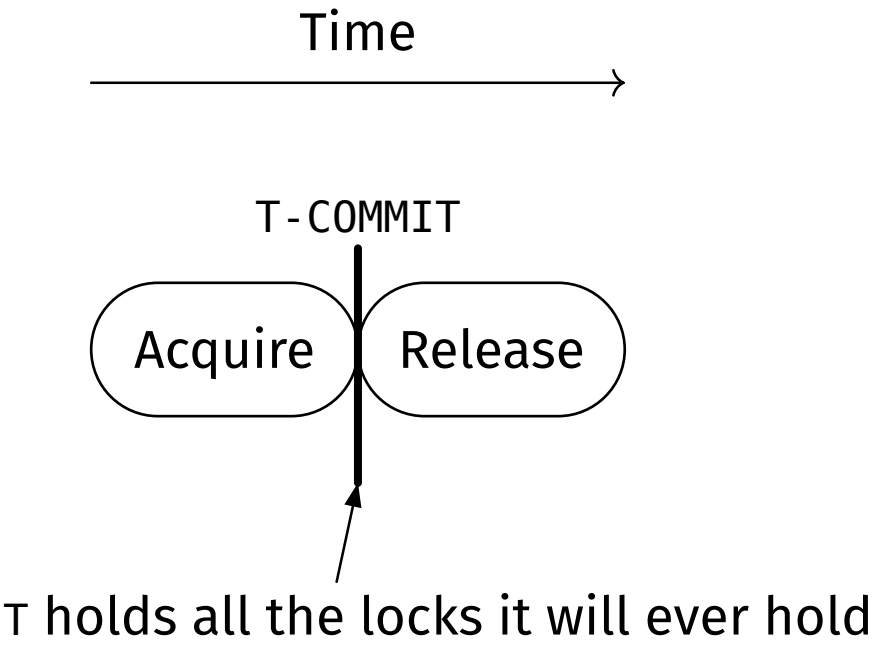
That is NOT what is happening here

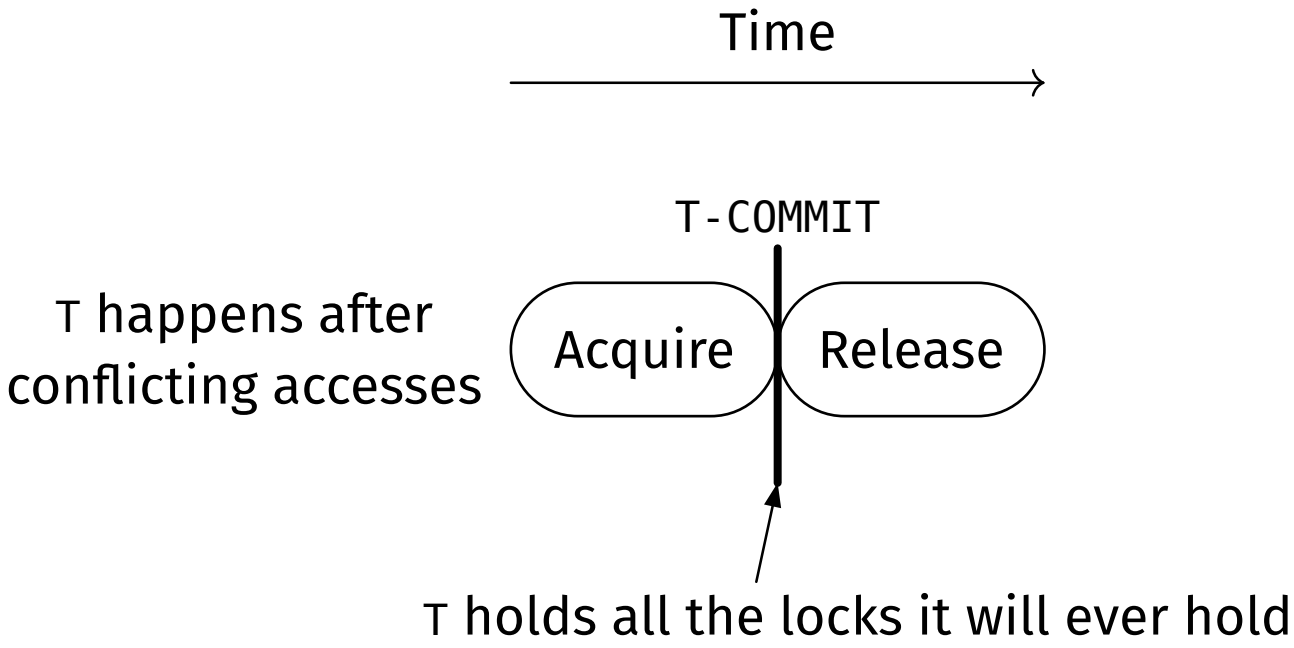
We are using locking to enforce an order over the transactions.

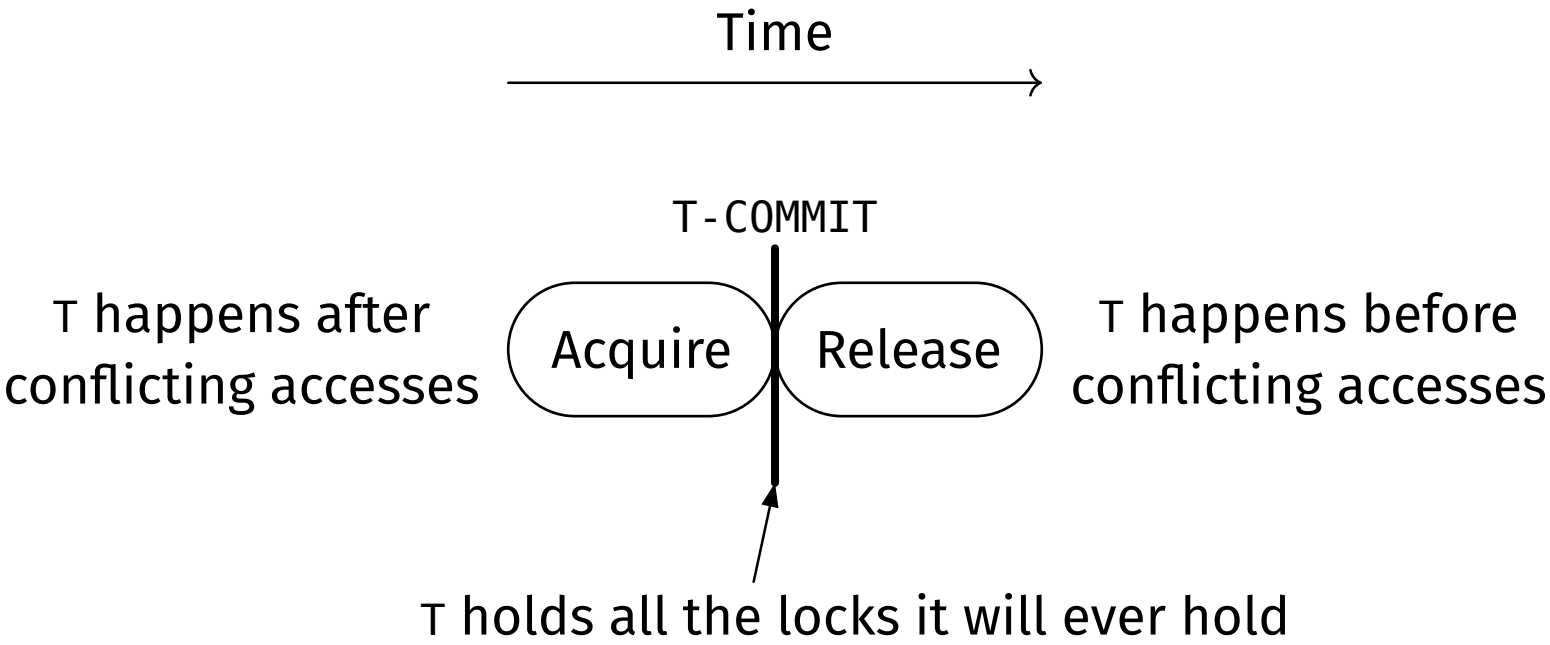

Time
Acquire


Time
T-COMMIT
Acquire


Time
T-COMMIT
Acquire
Release


Time
T-COMMIT
Acquire
Release
T holds all the locks it will ever hold


Time
T-COMMIT
T happens after
conflicting accesses
Acquire
Release
T holds all the locks it will ever hold


Time
T-COMMIT
T happens after conflicting accesses
Acquire
Release
T happens before conflicting accesses
T holds all the locks it will ever hold

2-Phase Locking

We can be a bit more general...

A transaction must hold (have acquired, but not released) a lock to access a variable.

1. **Acquire Phase**: Transaction can only acquire locks (but not release them)
   - T "Happens after" any permitted concurrent writes
2. **Release Phase**: Transaction can only release locks (but not acquire them)
   - T "Happens before" any permitted concurrent writes

This is called 2-Phase Locking

2-Phase Locking only permits conflict serializable schedules

2-Phase Locking

General 2-Phase locking is stronger than “Release at COMMIT” locking.

Time	T1	T2	T3
↓	W(A)		
↓		W(A)	
↓	W(B)		
↓			W(A)
↓		W(B)	
↓			W(B)

2-Phase Locking

General 2-Phase locking is stronger than “Release at COMMIT” locking.

Time	T1	T2	T3
↓	Lock(A)		
↓	Lock(B)		
↓	W(A)		
↓	Unlock(A)		
↓		Lock(A)	
↓		W(A)	
↓	W(B)		
↓	Unlock(B)		
↓		Lock(B)	
↓		Unlock(A)	
↓			Lock(A)
↓			W(A)
↓		W(B)	
↓		Unlock(B)	
↓			Lock(B)
↓			W(B)
↓			Unlock(A)
↓			Unlock(B)

... although some conflict serializable schedules may not be permitted

Time	T1	T2	T3
↓	W(A)		
↓		W(A)	
↓			W(A)
↓	W(B)		
↓		W(B)	
↓			W(B)

A schedule admitted by 2-Phase locking

... is always ...

A conflict serializable schedule

... is always ...

A serializable schedule

(But the reverse is not true)

Optimizations

Observation: Read-Read conflicts shouldn't be a problem

Reader/Writer Locks

Observation: Read-Read conflicts shouldn't be a problem

Solution: Reader/Writer Locks

(Any number of readers **or** one writer can hold the lock simultaneously)

Reader/Writer Locks

Observation: Read-Read conflicts shouldn't be a problem

Solution: Reader/Writer Locks

(Any number of readers **or** one writer can hold the lock simultaneously)

... also called Shared (S) / Exclusive (X) locks

Reader/Writer Locks

Held \ Requested	S	X
None	Allow	Allow
S	Allow	Block
X	Block	Block

A little abstraction

I'm going to use the word "object" to mean some database "thing".

- ... a table
- ... a record
- ... a collection of records (e.g., a page)
- ... a column of data
- ... a specific field in a specific record

A little abstraction

I'm going to use the word "object" to mean some database "thing".

- ... a table
- ... a record
- ... a collection of records (e.g., a page)
- ... a column of data
- ... a specific field in a specific record

Which should we actually implement?

Observation 1: If the locked objects are too coarse, we lose concurrency opportunities.

Observation 2: If the locked objects are too fine, we have to do more work locking them.

Locking Granularity

Idea: Separate locks for tables, pages, rows, cells, etc...

Locking Granularity

~~**Idea**: Separate locks for tables, pages, rows, cells, etc...~~

Doesn't work: Locking row on a page should block an attempt to lock the page.

Locking Granularity

~~**Idea**: Separate locks for tables, pages, rows, cells, etc...~~

Doesn't work: Locking row on a page should block an attempt to lock the page.

Better Idea: Lock the page (and table) before locking the row.

Locking Granularity

~~**Idea**: Separate locks for tables, pages, rows, cells, etc...~~

Doesn't work: Locking row on a page should block an attempt to lock the page.

~~**Better Idea**: Lock the page (and table) before locking the row.~~

Doesn't work: Kills concurrent access to different rows.

Exclusive (X)

The holding thread can safely modify the object

Shared (S)

The holding thread can safely read the object

Exclusive (X)
The holding thread can safely modify the object

Shared (S)
The holding thread can safely read the object

Intent-Exclusive (IX)
The holding thread can safely acquire an exclusive lock on a smaller component of the object

Intent-Shared (IS)
The holding thread can safely acquire a shared lock on a smaller component of the object

New Lock Modes

To acquire a shared (exclusive) lock, first acquire an intent-shared (intent-exclusive) lock on all containing objects.

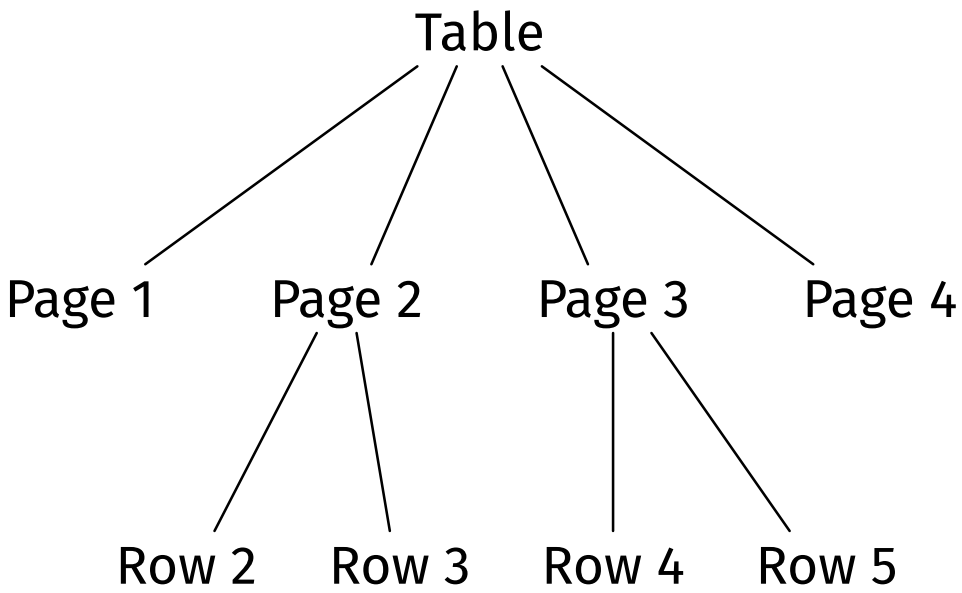
New Lock Modes

1. Acquire-IX(Table)
2. Acquire-IX(Page)
3. Acquire-IX(Row)
4. Acquire-X(Cell)
5. Modify Cell
6. Release all locks in reverse order

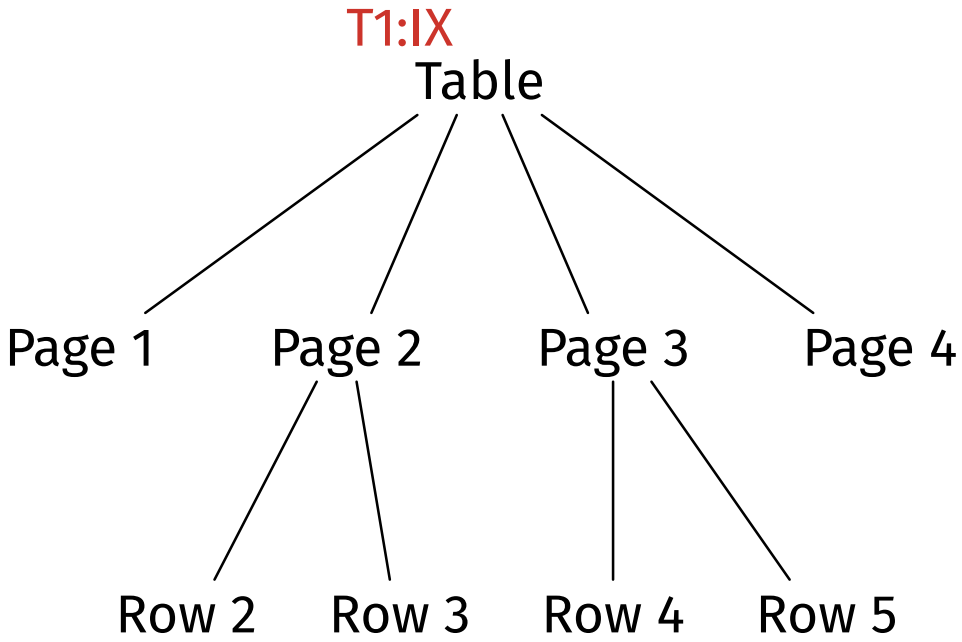
Reader/Writer Locks

Held \ Requested	IS	IX	S	X
None	Allow	Allow	Allow	Allow
IS	Allow	Allow	Allow	Block
IX	Allow	Allow	Block	Block
S	Allow	Block	Allow	Block
X	Block	Block	Block	Block

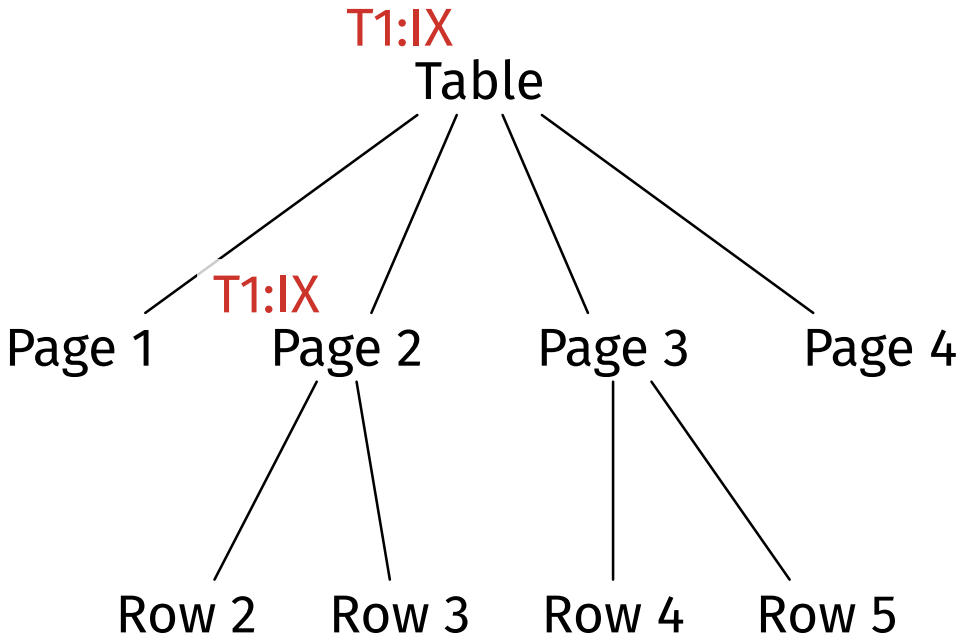
Example 1



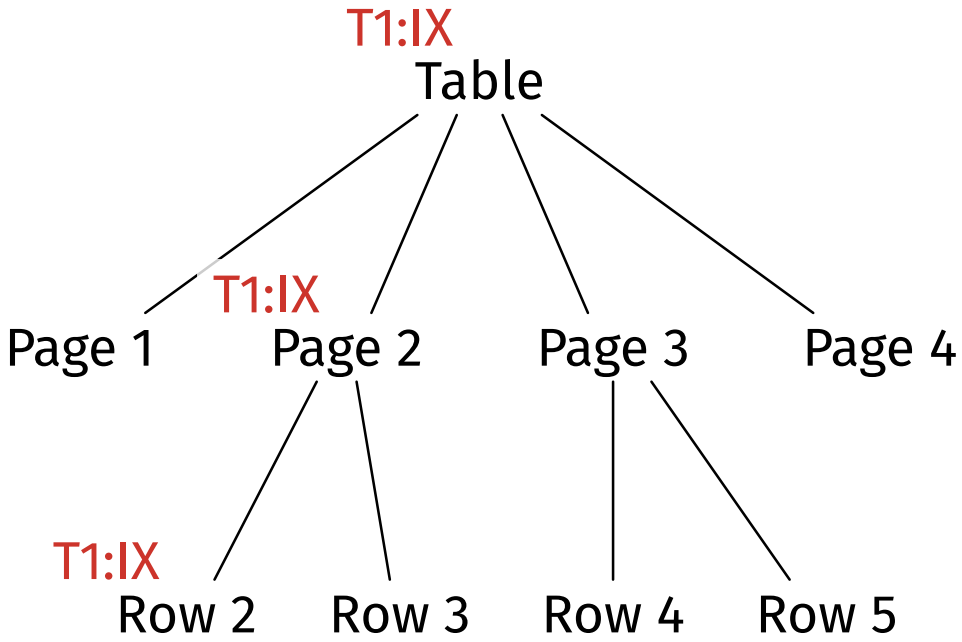
Example 1



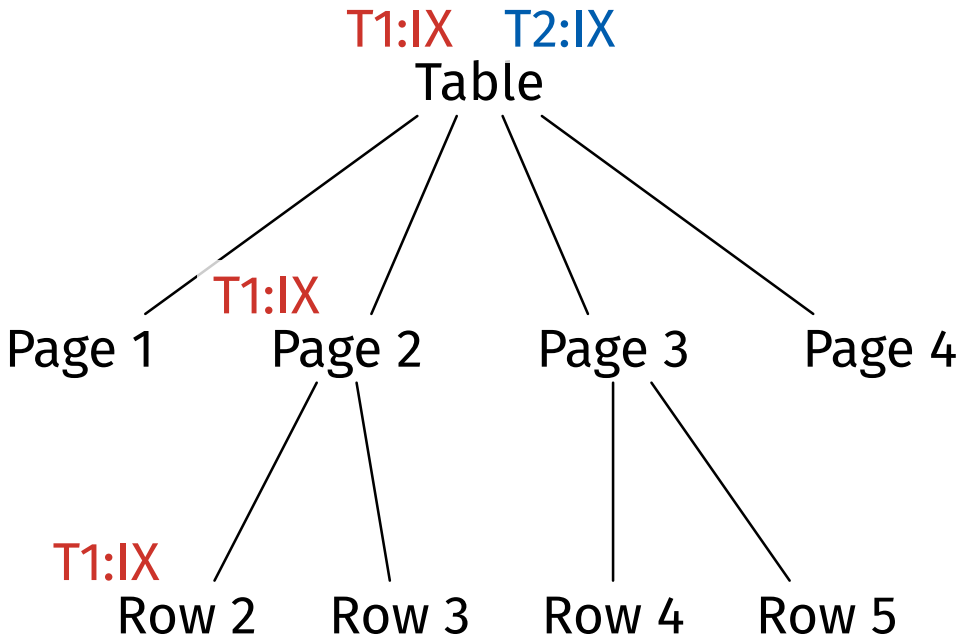
Example 1



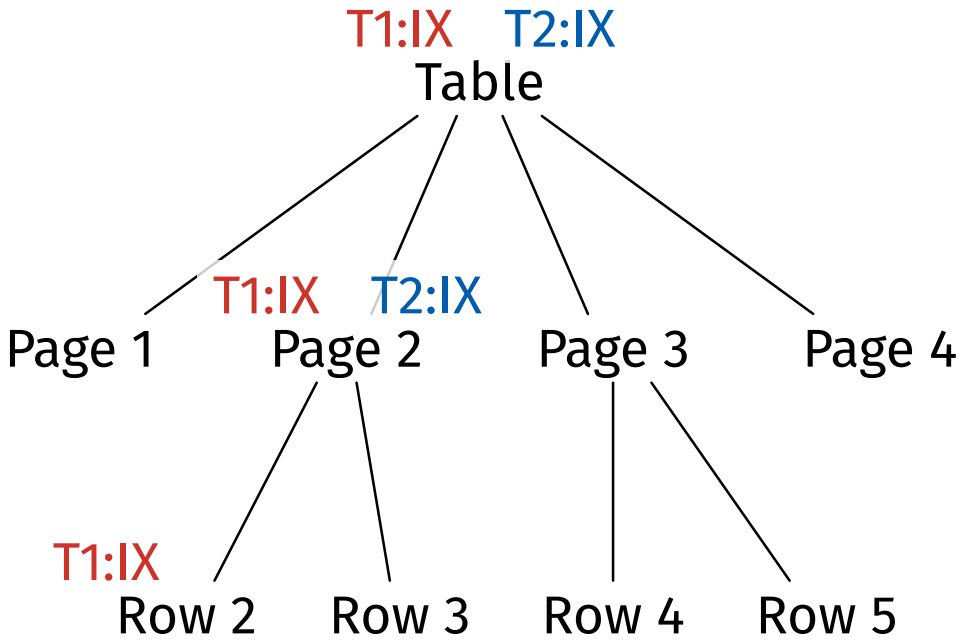
Example 1


T1:IX
Table
T1:IX
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

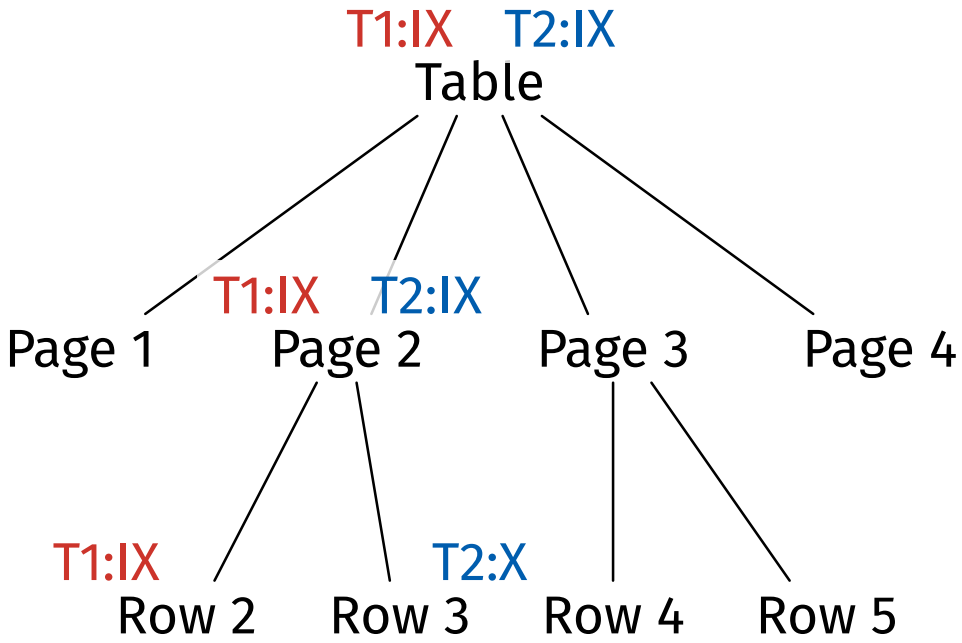
Example 1


T1:IX
T2:IX
Table
T1:IX
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

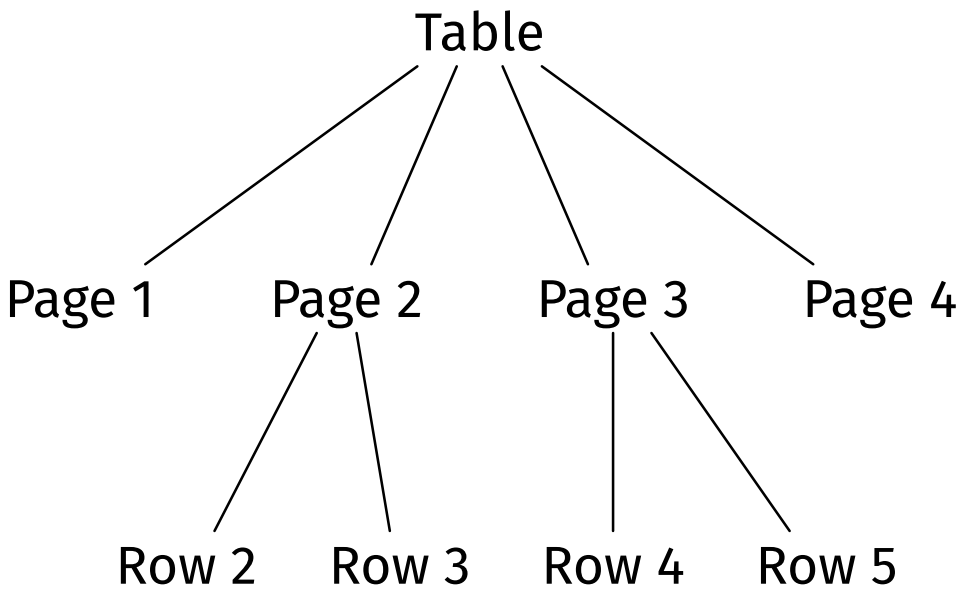
Example 1


T1:IX
T2:IX
Table
T1:IX
T2:IX
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

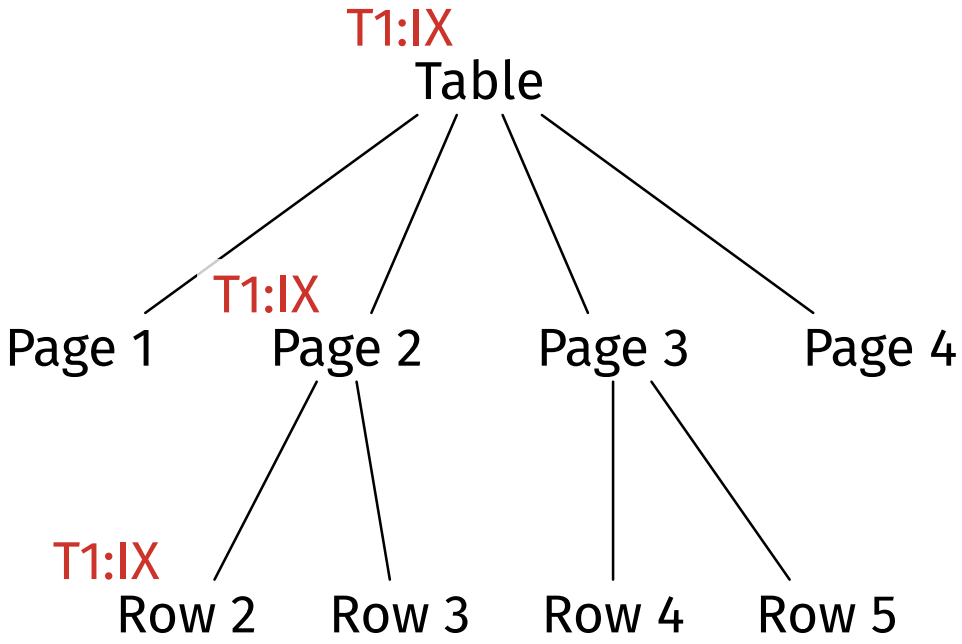
Example 1


T1:IX
T2:IX
Table
T1:IX
T2:IX
Page 1
Page 2
Page 3
Page 4
T1:IX
T2:X
Row 2
Row 3
Row 4
Row 5

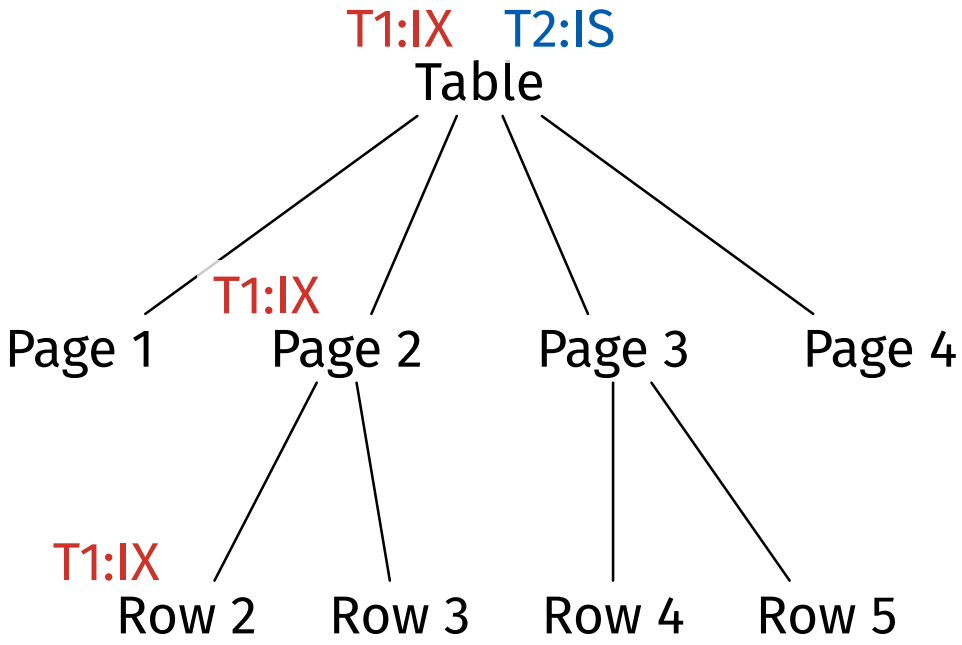
Example 2



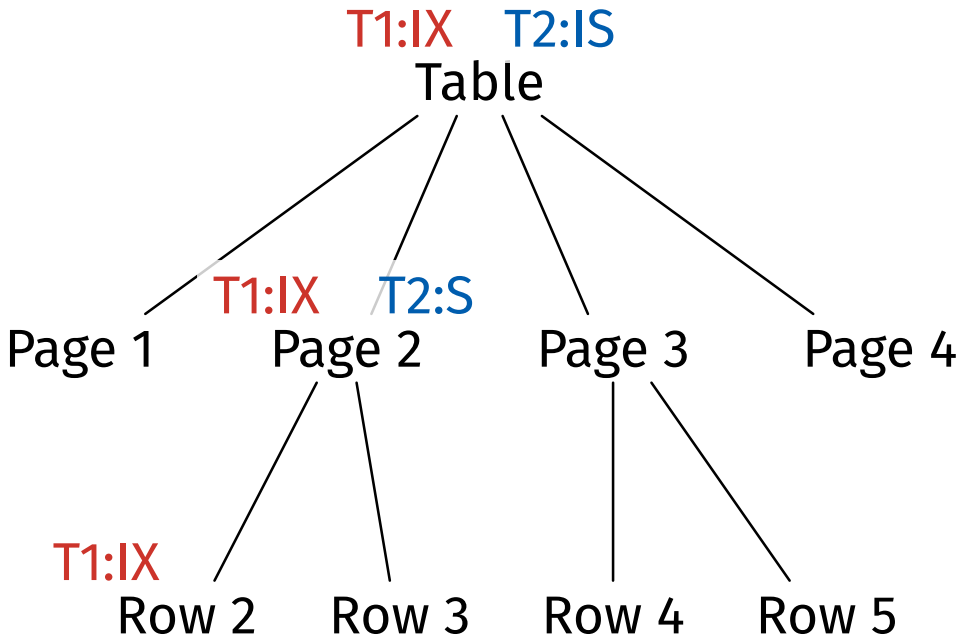
Example 2


T1:IX
Table
T1:IX
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

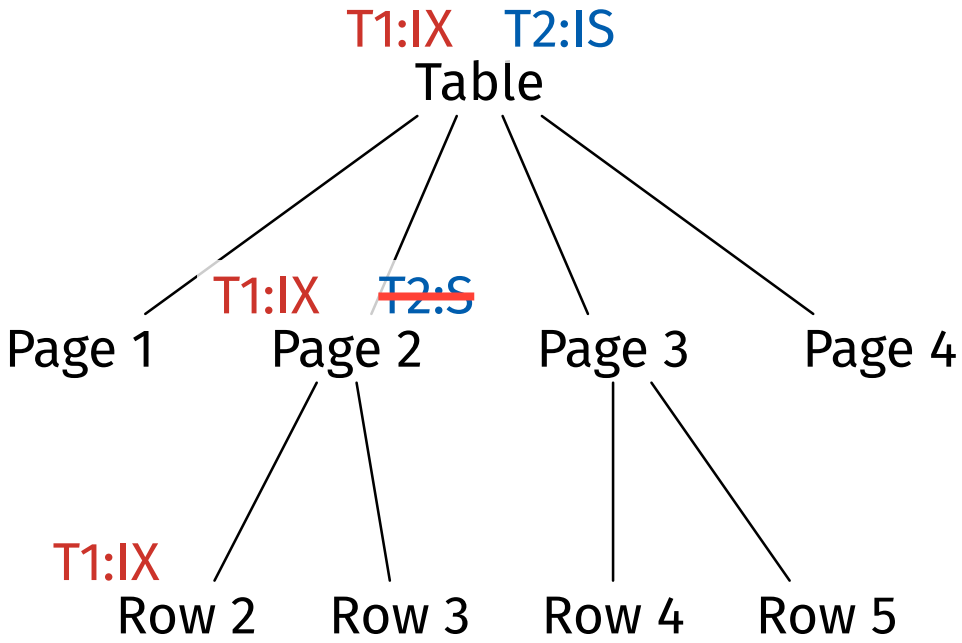
Example 2


T1:IX
T2:IS
Table
T1:IX
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

Example 2


T1:IX
T2:IS
Table
T1:IX
T2:S
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

Example 2


T1:IX
T2:IS
Table
T1:IX
T2:S
Page 1
Page 2
Page 3
Page 4
T1:IX
Row 2
Row 3
Row 4
Row 5

Deadlock

Deadlock

Time	T1	T2
↓	Lock(A)	
↓	R(A)	
↓		Lock(B)
↓		R(B)
↓	???	
↓	W(A)	
↓		???
↓		W(B)

Optimistic

Detect deadlock situations when they happen and abort the deadlocked transaction.

Pessimistic

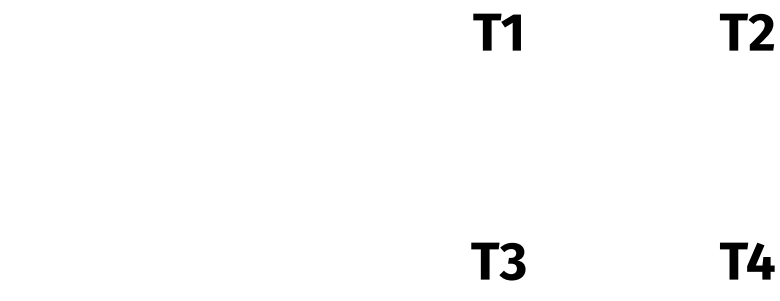
Anticipate deadlock situations and abort preemptively

The Waits-For Graph

- Add one node for every **running** transaction
- Add one edge from a blocked transaction to the transaction(s) holding the conflicting lock.

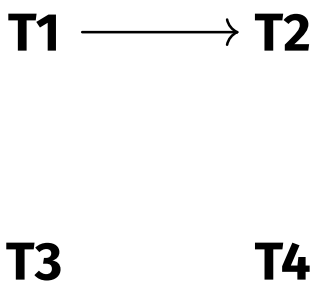
Waits For Example

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	



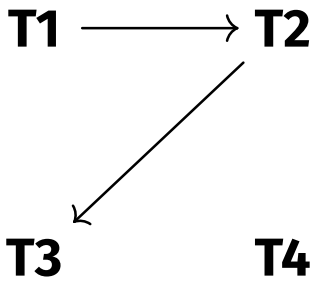
Waits For Example

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	



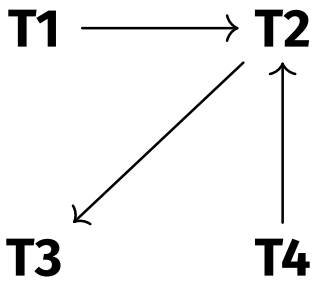
Waits For Example

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	



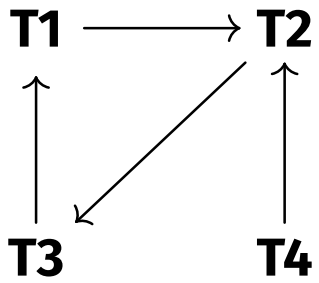
Waits For Example

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	



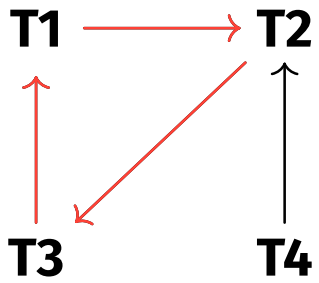
Waits For Example

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	



Waits For Example

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	



Waits For

A cycle is a set of transactions that will never finish.

1. Periodically check for cycles in the Waits For Graph

Waits For

1. Periodically check for cycles in the Waits For Graph
2. Abort one transaction from each cycle until there are no more

Cycle detection is expensive! $O(N + E)$

Proactive Deadlock Detection

Idea: Define an order over objects and only allow locks to be acquired in order

Waits For Example

Only allow locks to be acquired in alphabetical order

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	

Waits For Example

Only allow locks to be acquired in alphabetical order

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	

T3 already has **C**, so it can't wait for **A**

Waits For Example

Only allow locks to be acquired in alphabetical order

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	

T3 already has **C**, so it can't wait for **A**

Waits For Example

Only allow locks to be acquired in alphabetical order

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	

T3 already has **C**, so it can't wait for **A**

Waits For Example

Only allow locks to be acquired in alphabetical order

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	

T3 already has **C**, so it can't wait for **A**

Waits For Example

Only allow locks to be acquired in alphabetical order

Time	T1	T2	T3	T4
↓	S(A)			
↓	R(A)			
↓		X(B)		
↓		W(B)		
↓	Wait for S(B)			
↓			S(C)	
↓			R(C)	
↓		Wait for X(C)		
↓				Wait for X(B)
↓			Wait for X(A)	

T3 already has **C**, so it can't wait for **A**

Proactive Deadlock Detection

Variant Idea: Acquire all locks at the start of the transaction

Proactive Deadlock Detection

Pros

- No deadlocks… ever
- No expensive cycle detection

Cons

- Not all transactions can bound the set of objects they'll need upfront

Pessimistic Deadlock Detection

Idea: (Rare) False positives are ok

Pessimistic Deadlock Detection

1. Define an order over transactions as they arrive
2. Only allow newer transactions to block on older transactions

Variant 1

Variant 1

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

Variant 1

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

Variant 1

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

Variant 1

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

- `ABORT` T1 and restart it as a “younger” transaction

Variant 1

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

- `ABORT` T1 and restart it as a “younger” transaction

“Wait-Die”

Variant 2

Variant 2

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

Variant 2

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

Variant 2

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

- `ABORT` T2 and restart it at the same age

Variant 2

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

- `ABORT` T2 and restart it at the same age

Variant 2

T1 holds a lock on A

T2 tries to acquire a lock on A and blocks

- Ok because T1 is “older”

T2 holds a lock on A

T1 tries to acquire a lock on A and blocks

- `ABORT` T2 and restart it at the same age

“Wait-Wound”

Wait Die

Abort older transactions that block on younger ones

Wait Wound

Abort younger transactions if an older transaction would block on it

Remember serializability containment

A schedule admitted by 2-Phase locking

... is always ...

A conflict serializable schedule

... is always ...

A serializable schedule

(But the reverse is not true)