Transactions: Optimistic

CSE 4/562: Database Systems | Lecture 20

DB. Sys.: T.C.B.: Ch. 18.8-18.9

Schedule

An order over the reads and writes of a transaction

Serial Schedule

A schedule with no interleaving

Serializable Schedule

A schedule guaranteed to produce the same output as a serial schedule

Conflict Serializable Schedule

A schedule with the same “happens before” constraints as a serial schedule

Acquire Phase

A transaction in the acquire phase may...

- Acquire Shared or eXclusive locks for any object
- Upgrade any shared lock into an exclusive one
- Read any object for which it holds a shared or exclusive lock
- Write any object for which it holds an exclusive lock
- Move to the release phase.

Release Phase

A transaction in the release phase may...

- Read any object for which it holds a shared or exclusive lock
- Write any object for which it holds an exclusive lock
- Release any lock it holds
- End (Commit or Abort)

2-Phase Locking

Any schedule created by 2-phase locking must be conflict serializable.

(but the reverse is not true)

Example

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

Example

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

3 $\times$ T1 “happens before” T2
“happens before” T3

Example

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

3 $\times$ T1 “happens before” T2
“happens before” T3

The schedule is conflict-serializable.

Example

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

3 $\times$ T1 “happens before” T2
“happens before” T3

The schedule is conflict-serializable.

... but there is no way to create this schedule with 2-Phase Locking.

Locking is...

... expensive

Costs are still incurred even if there are no problematic conflicts.

... restrictive

We don’t know what the transaction will do ahead of time, so we sometimes need to block unnecessarily.

Optimism!

We don’t know what a transaction will do...

Optimism!

We don’t know what a transaction will do… until it does.

Optimism!

Idea: Let the transaction do it!

Optimism!

Idea: Let the transaction do it!

(then fix anything it breaks, later)

Locking

Pessimistically block transactions before they can act out of order.

Snapshot Isolation

Run transactions on an image of the data, and revert changes if they accidentally ran out-of-order.

Timestamp Concurrency Control

Enforcing Serializability

Locking

Pessimistically block transactions before they can act out of order.

Snapshot Isolation

Run transactions on an image of the data, and revert changes if they accidentally ran out-of-order.

Timestamp Concurrency Control

Snapshot Isolation

Snapshot Isolation

Phase 1: Read

- The transaction executes on a private copy of all accessed objects
- The system records each of the transaction's reads and writes

Phase 1: Read

- The transaction executes on a private copy of all accessed objects
- The system records each of the transaction's reads and writes

Phase 2: Validate

- The system checks whether the transaction's writes can be applied without violating isolation.

Phase 1: Read

- The transaction executes on a private copy of all accessed objects
- The system records each of the transaction's reads and writes

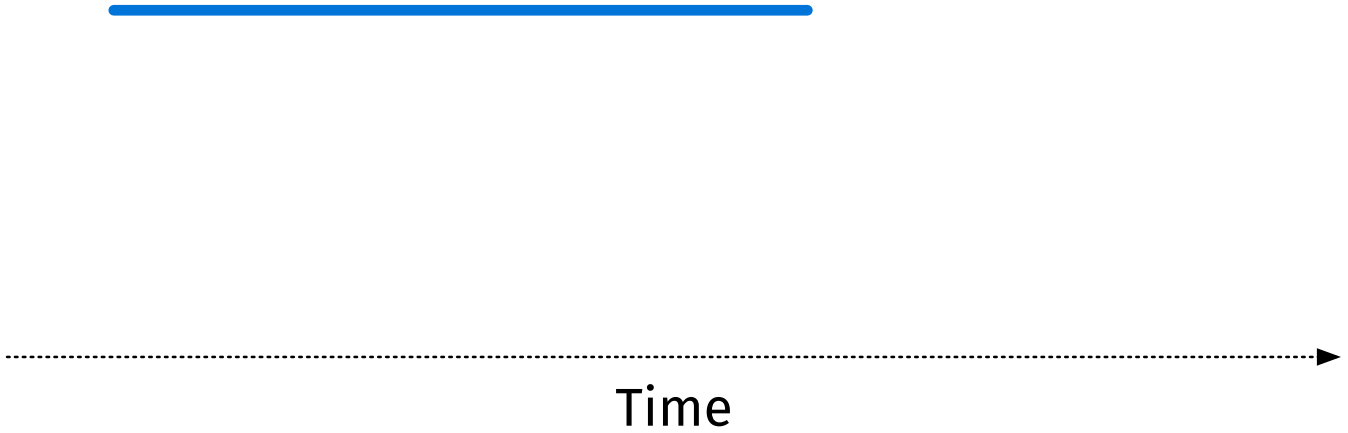
Phase 2: Validate

- The system checks whether the transaction's writes can be applied without violating isolation.

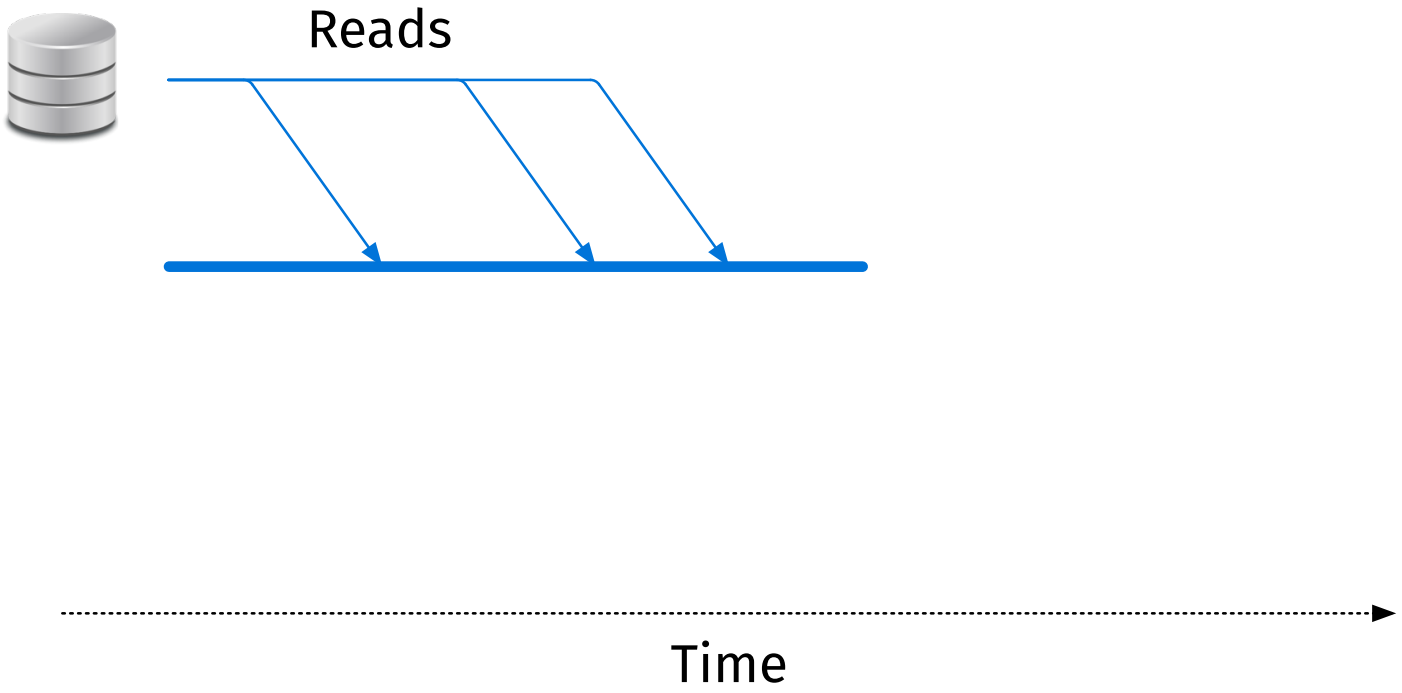
Phase 3: Write (Merge)

- The system replaces all written objects with the transaction's private copy.

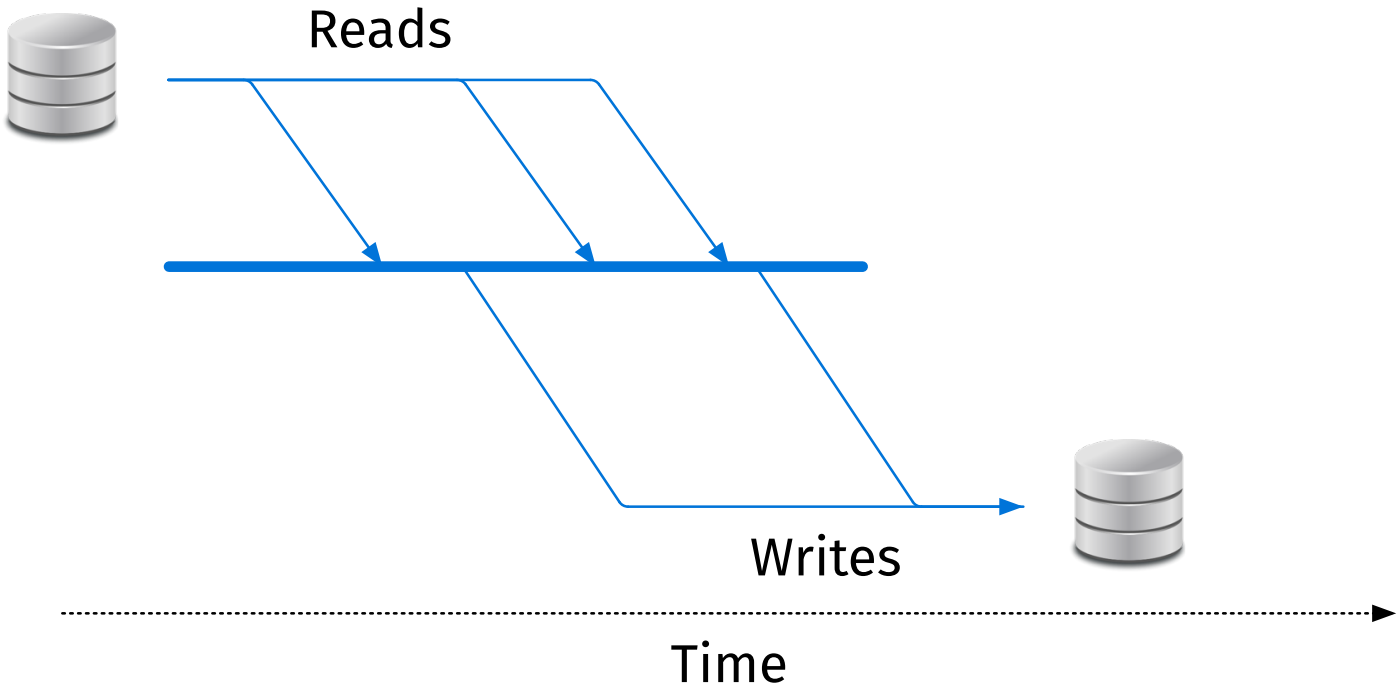
Serial Transactions



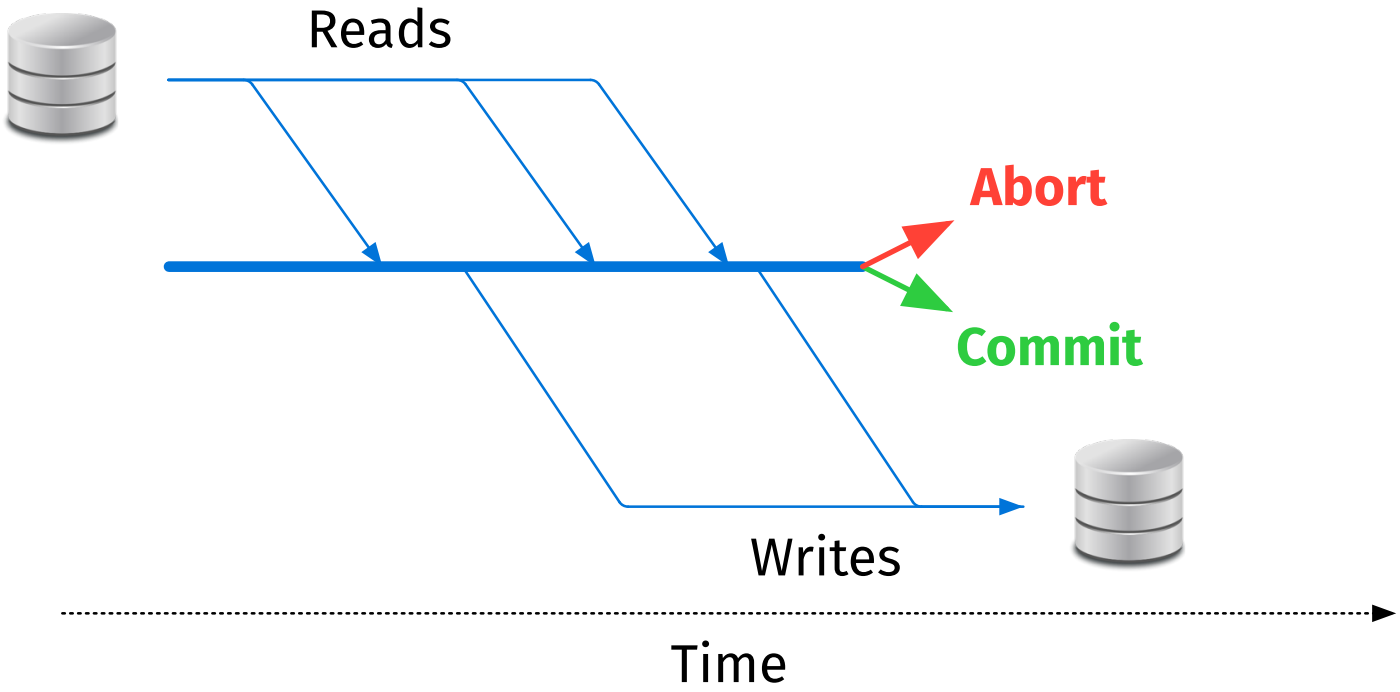
Serial Transactions



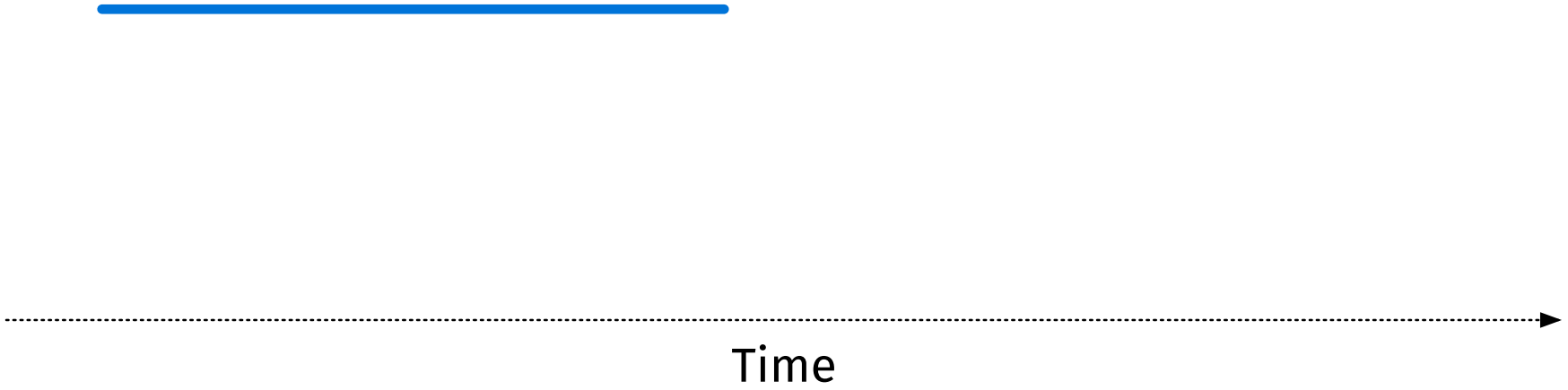
Serial Transactions



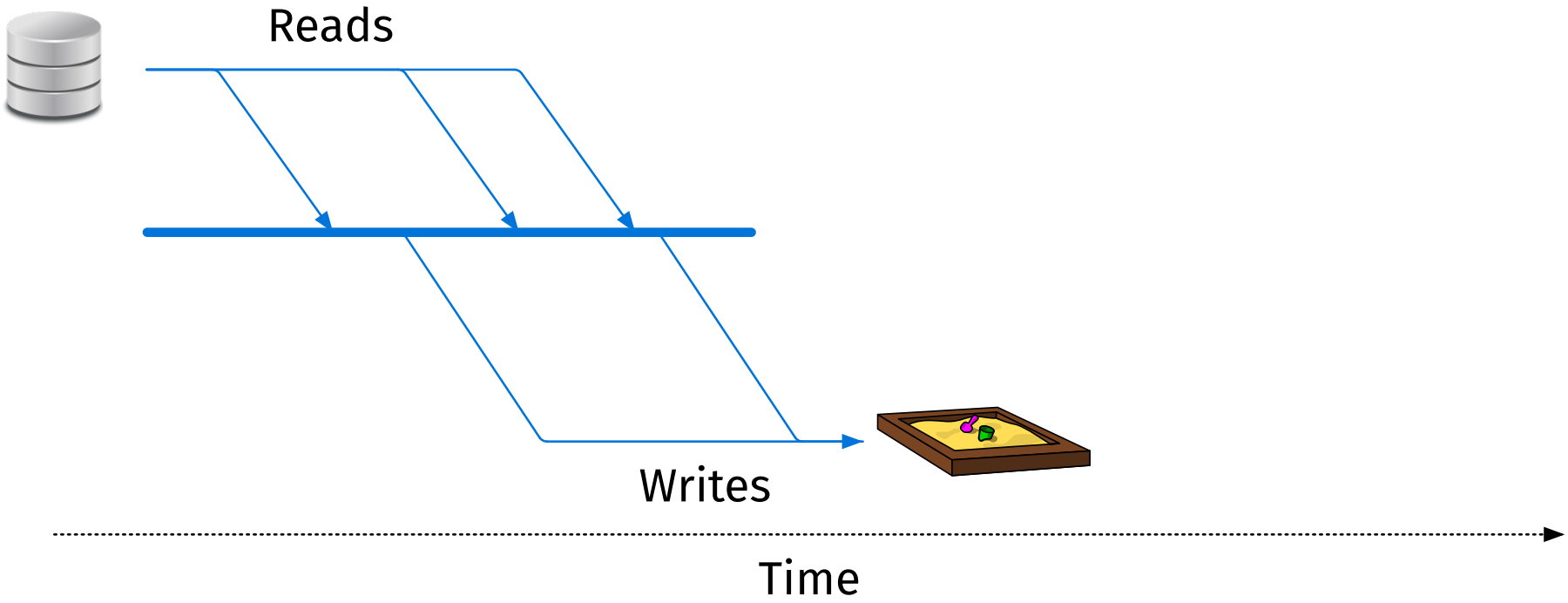
Serial Transactions



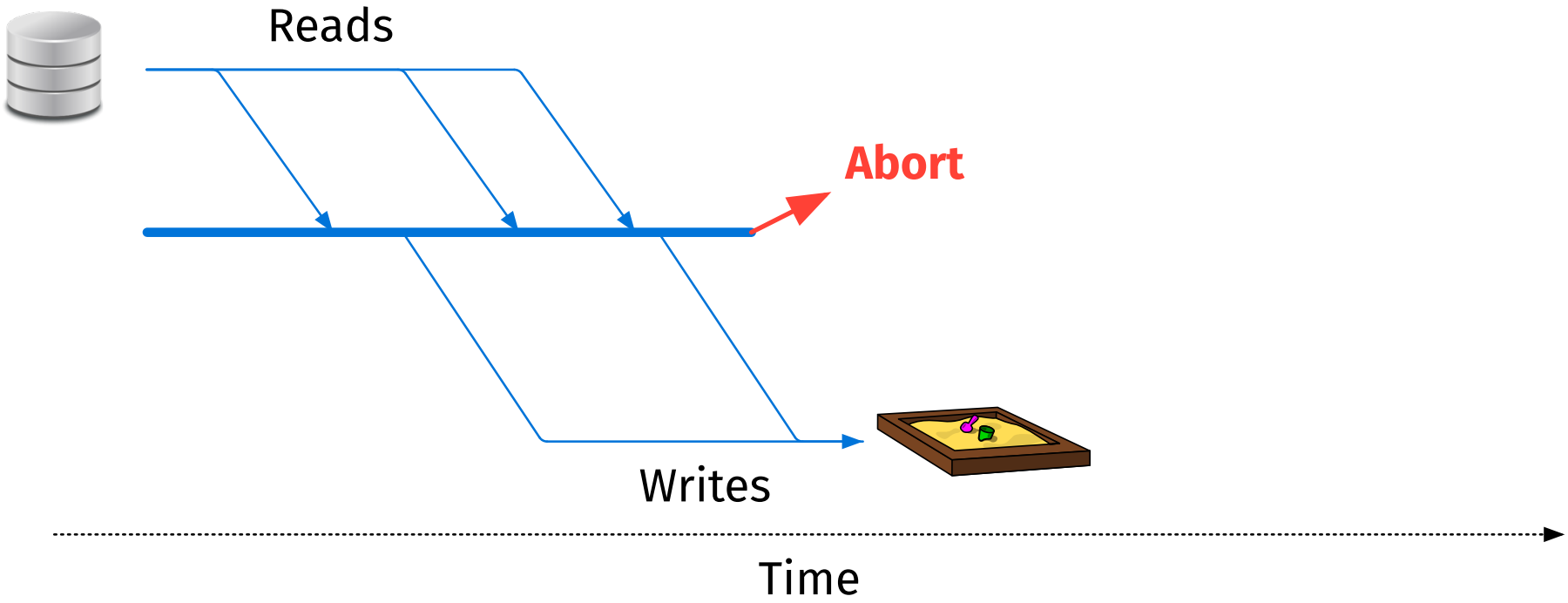
Sandboxed Transactions



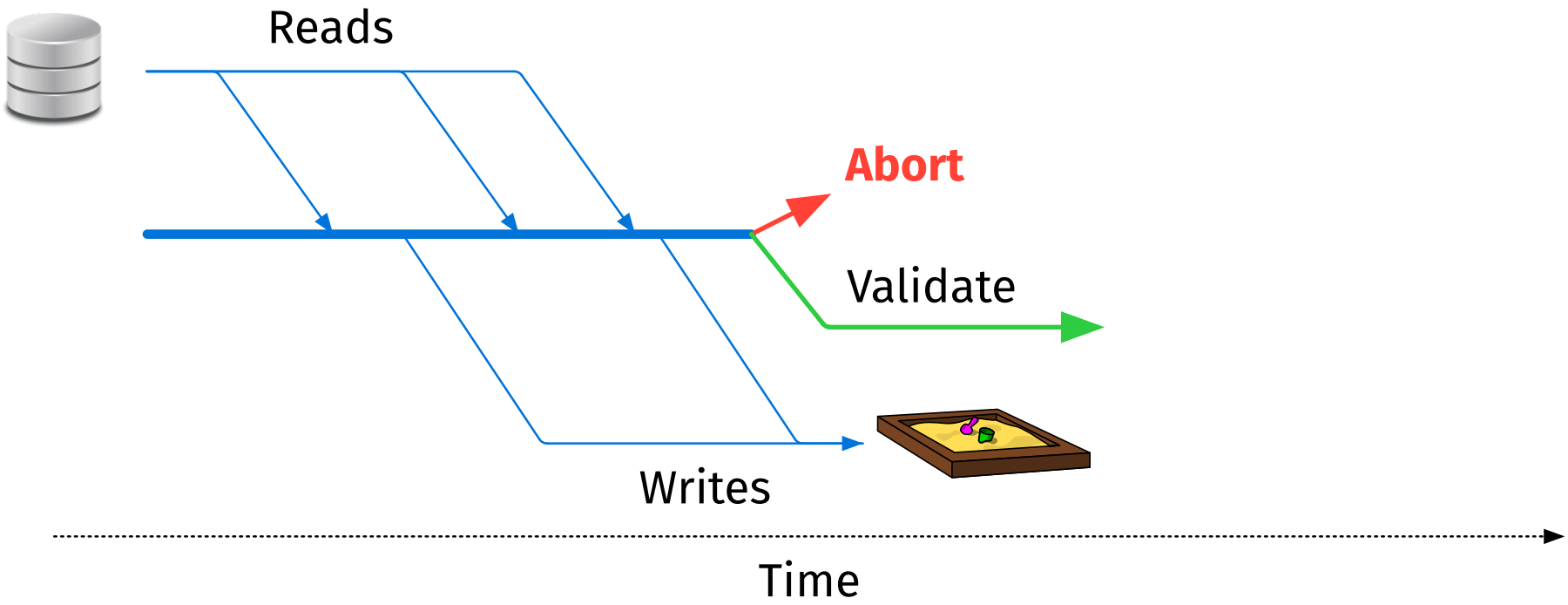
Sandboxed Transactions



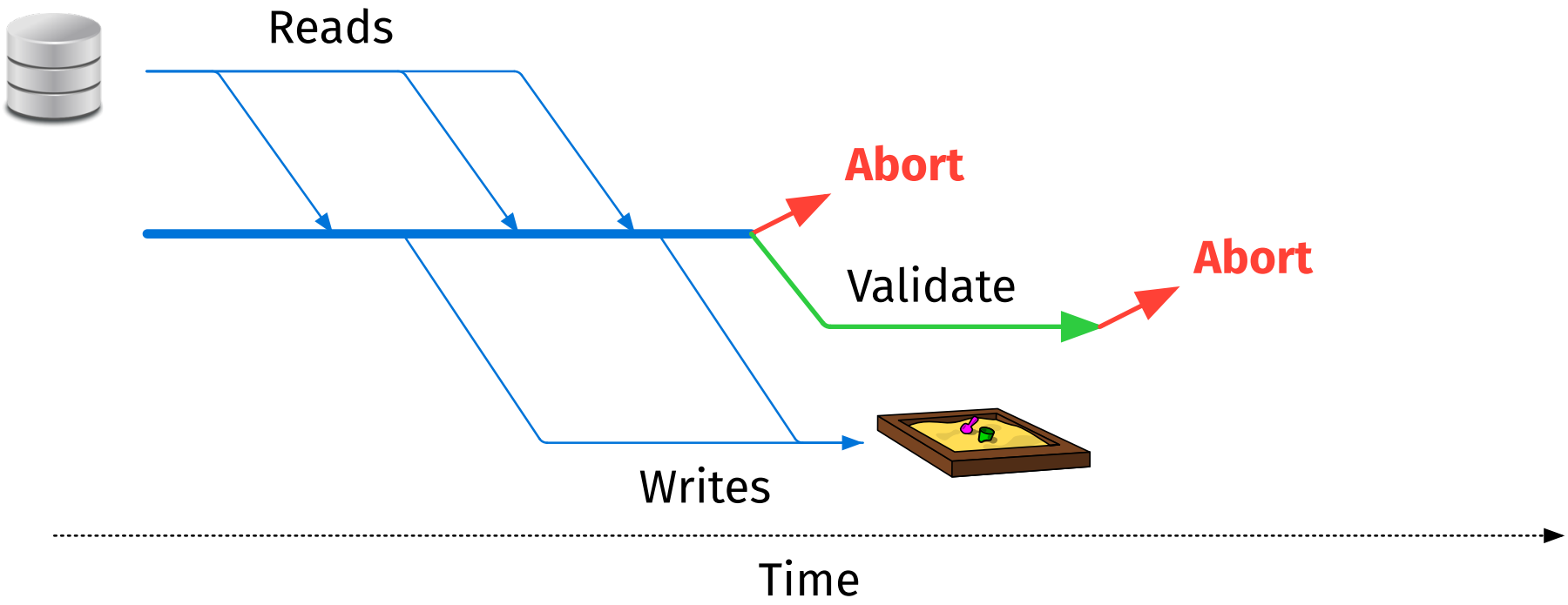
Sandboxed Transactions



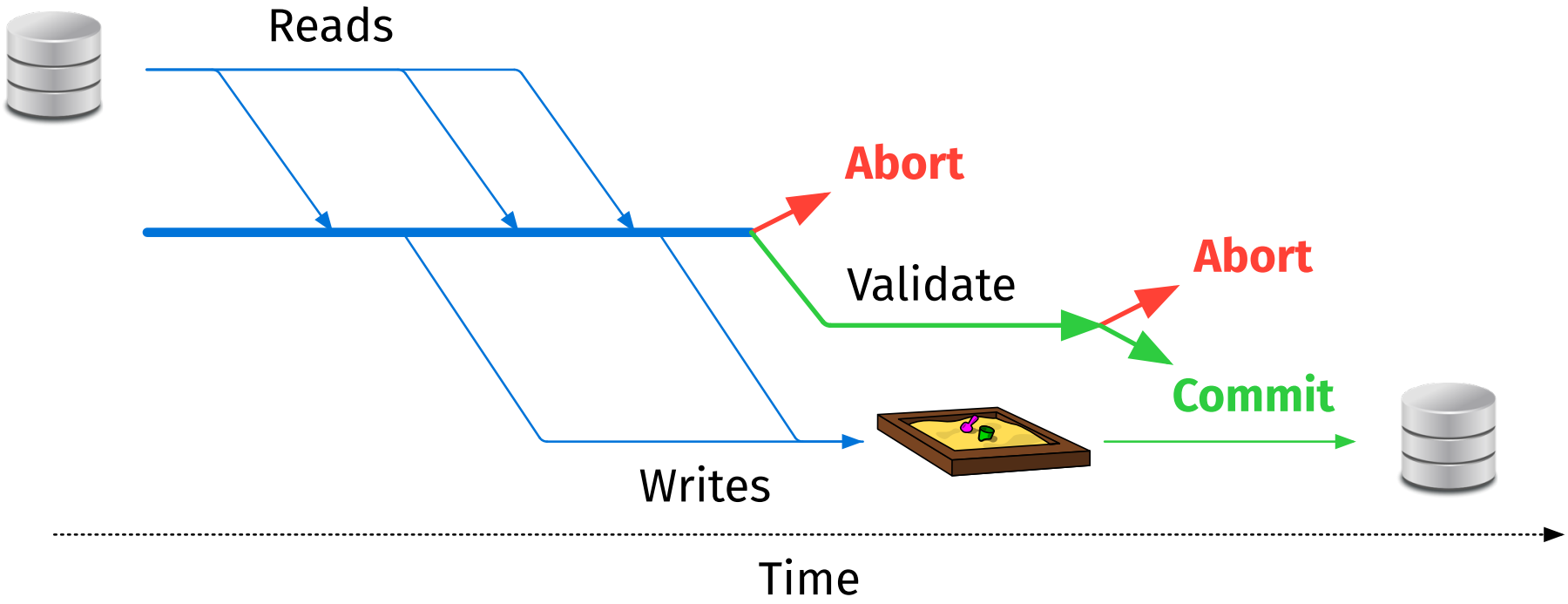
Sandboxed Transactions



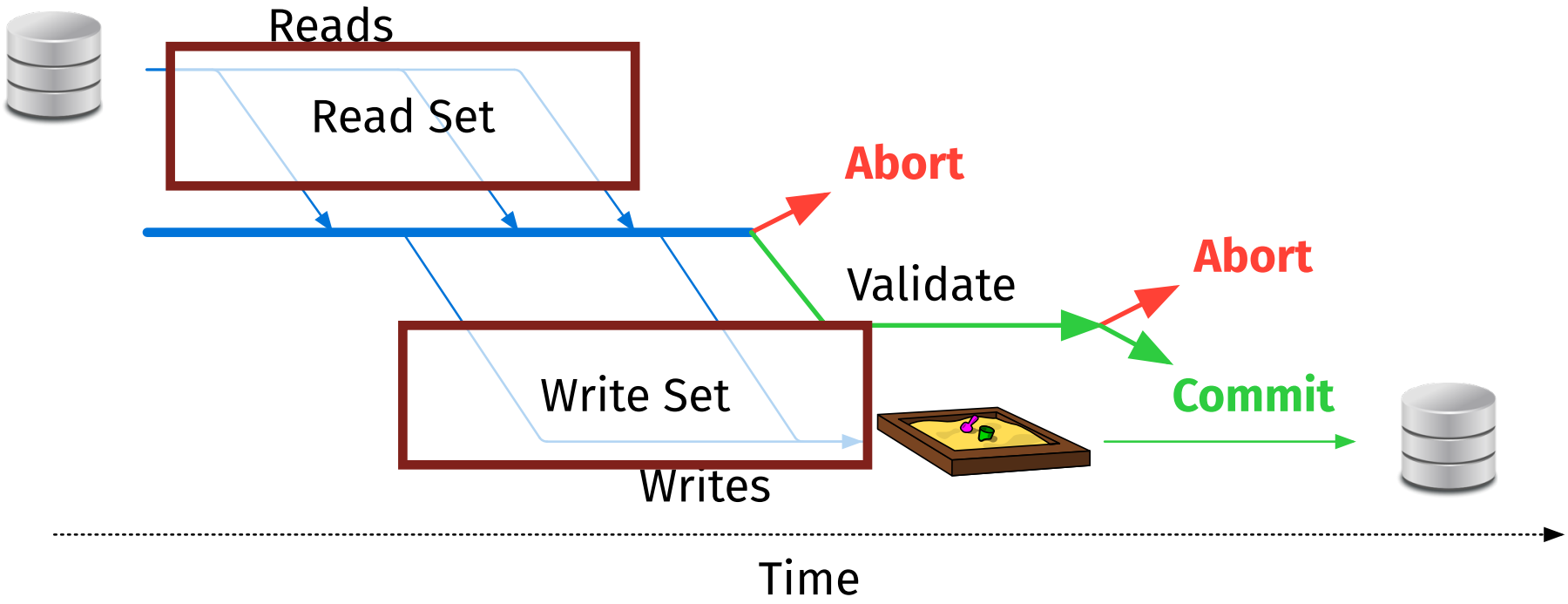
Sandboxed Transactions



Sandboxed Transactions



Sandboxed Transactions



Read Phase

Run the transaction on a snapshot of the data and collect:

$\mathrm{ReadSet}(\mathcal{T})$

The set of objects read by $\mathcal{T}$

$\mathrm{WriteSet}(\mathcal{T})$

The set of objects written by $\mathcal{T}$

Validation Phase

1. Pick a serial order
   (e.g., the order in which transactions reach the validation phase)
2. Make sure that the transaction's operations are consistent with this order

Conflicts

T1 scheduled before T2 (So T1′s validation step happens before T2′s)

T1 and T2 read the same object

T1 reads an object written by T2 (read-write conflict)

T2 reads an object written by T1 (write-read conflict)

T1 and T2 write the same object (write-write conflict)

Conflicts

T1 scheduled before T2 (So T1′s validation step happens before T2′s)

T1 and T2 read the same object

Ok!

T1 reads an object written by T2 (read-write conflict)

T2 reads an object written by T1 (write-read conflict)

T1 and T2 write the same object (write-write conflict)

T1 scheduled before T2 (So T1′s validation step happens before T2′s)

T1 and T2 read the same object

Ok!

T1 reads an object written by T2 (read-write conflict)

Ok! (T2′s write was sandboxed and invisible to T1)

T2 reads an object written by T1 (write-read conflict)

T1 and T2 write the same object (write-write conflict)

T1 scheduled before T2 (So T1′s validation step happens before T2′s)

T1 and T2 read the same object

Ok!

T1 reads an object written by T2 (read-write conflict)

Ok! (T2′s write was sandboxed and invisible to T1)

T2 reads an object written by T1 (write-read conflict)

Ok if T1′s write phase finishes before T2′s read phase starts

T1 and T2 write the same object (write-write conflict)

T1 scheduled before T2 (So T1′s validation step happens before T2′s)

T1 and T2 read the same object

Ok!

T1 reads an object written by T2 (read-write conflict)

Ok! (T2′s write was sandboxed and invisible to T1)

T2 reads an object written by T1 (write-read conflict)

Ok if T1′s write phase finishes before T2′s read phase starts

T1 and T2 write the same object (write-write conflict)

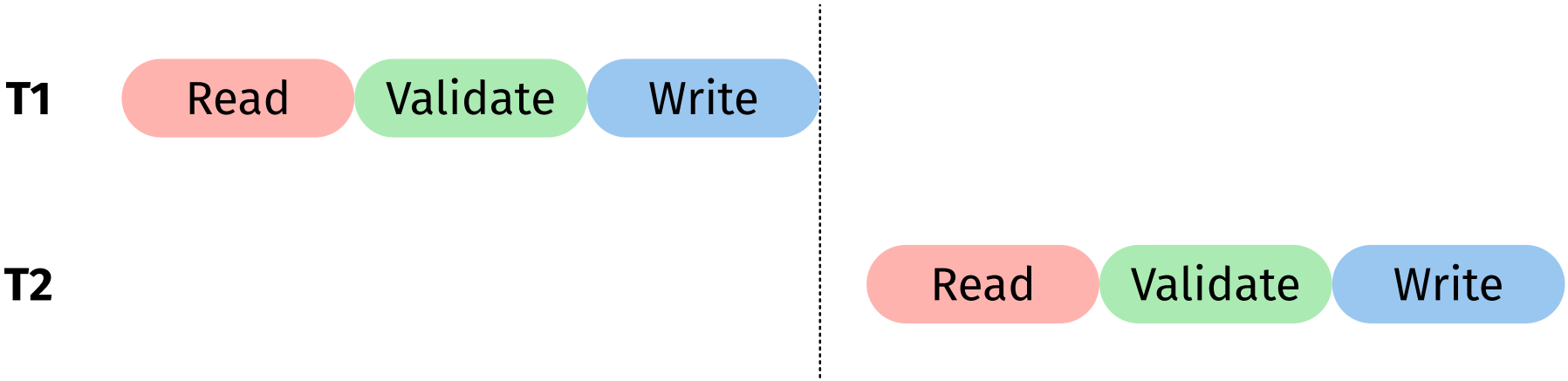
Ok if T1′s write phase finishes before T2′s write phase starts

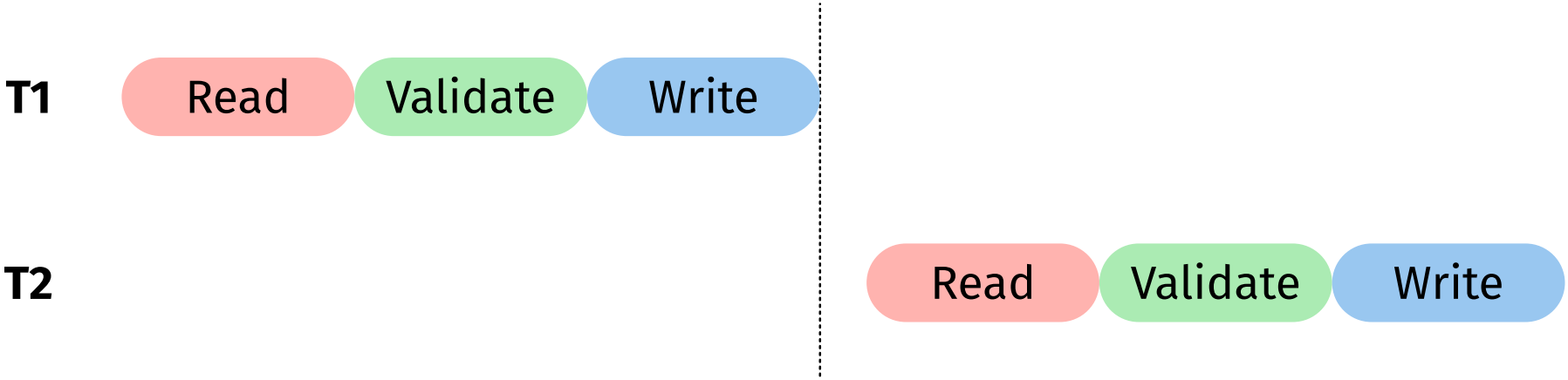
Depending on how transaction phases overlap, we need to check for different conflicts.

Note: The validation and write phases are not instantaneous.

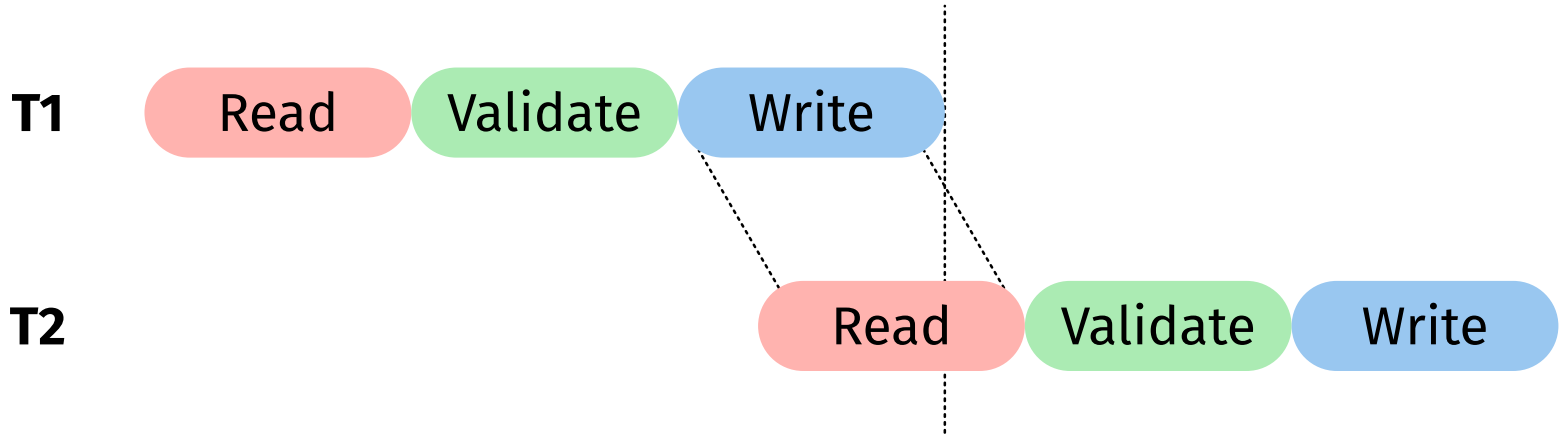
- Validation (and serial order selection) are a critical section, only one transaction is permitted at a time.
- (Compatible) write phases can proceed concurrently.

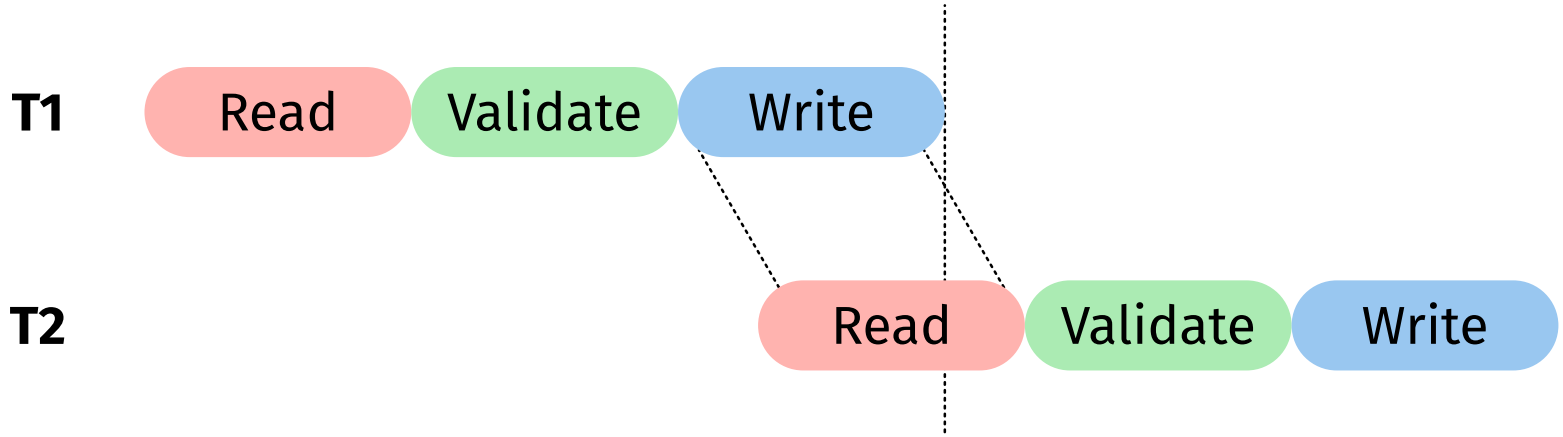
Test 1





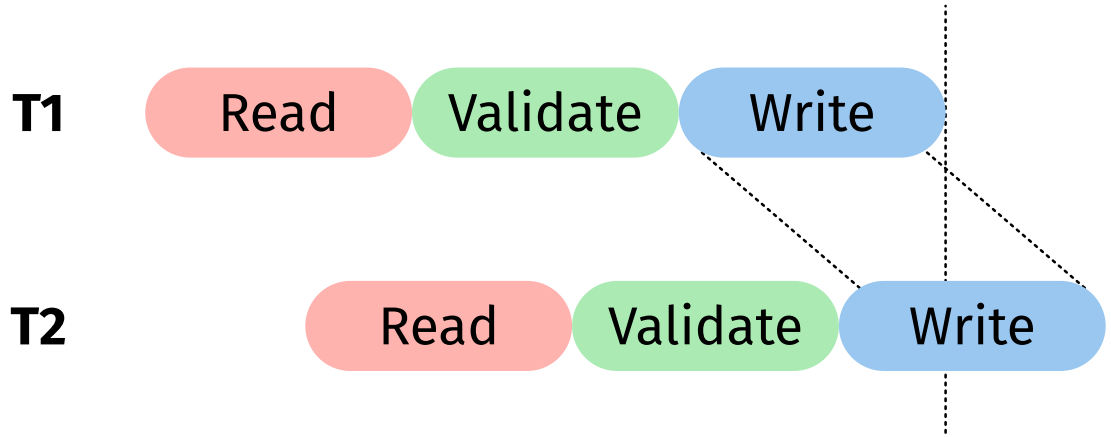
If **T1**′s write phase ends before **T2** starts, then **T2** sees all of **T1**′s writes

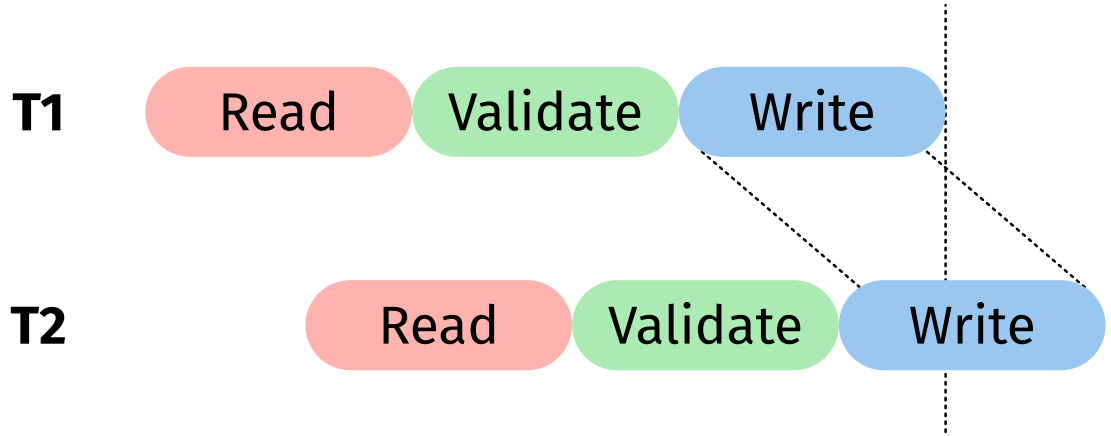

T1
Read
Validate
Write
T2
Read
Validate
Write



If **T1**'s write phase ends after **T2**'s read phase starts, there is a possibility of write-read conflicts. Abort (and restart) **T2** if:

$$\mathrm{WriteSet}(\mathcal{T}_1) \cap \mathrm{ReadSet}(\mathcal{T}_2) \neq \emptyset$$


T1
Read
Validate
Write
T2
Read
Validate
Write



If **T1**′s write phase overlaps with **T2**′s write phase, there is also a possibility of write-write conflicts. Abort (and restart) **T2** if either:

$$\text{WriteSet}(\mathcal{T}_1) \cap \text{ReadSet}(\mathcal{T}_2) \neq \emptyset$$

$$\text{WriteSet}(\mathcal{T}_1) \cap \text{WriteSet}(\mathcal{T}_2) \neq \emptyset$$

Locking vs Snapshot Isolation

Locking

Snapshot Isolation

Locking vs Snapshot Isolation

Locking

- **Pro**: No need to restart failed transactions in general.
- **Pro**: Low overheads when there are no conflicts.

Snapshot Isolation

Locking vs Snapshot Isolation

Locking

- **Pro**: No need to restart failed transactions in general.
- **Pro**: Low overheads when there are no conflicts.

Snapshot Isolation

- **Pro**: Even lower overheads if transactions are “safe” by overhead.

Locking

- **Pro**: No need to restart failed transactions in general.
- **Pro**: Low overheads when there are no conflicts.

Snapshot Isolation

- **Pro**: Even lower overheads if transactions are “safe” by overhead.
- **Con**: Validation doesn’t happen until the validation step.

Snapshot Isolation Flaws

Idea: Check for conflicts while the transaction is running

(abort and restart immediately if a violation is detected)

Timestamp Concurrency Control

Timestamp Concurrency Control (Idealized)

Each object A gets:

- A read timestamp $\mathrm{RTS}(A)$
- A write timestamp $\mathrm{WTS}(A)$

Each transaction $\mathcal{T}$ gets:

- A timestamp $\mathrm{TS}(\mathcal{T})$

(note that timestamps can be logical timestamps like sequence numbers)

Timestamp Concurrency Control

We require transactions to follow the serial order defined by $\mathrm{TS}(\mathcal{T})$

Transactions that violate this order are restarted with a fresh timestamp

When $\mathcal{T}$ reads object A

When $\mathcal{T}$ reads object A

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ then

When $\mathcal{T}$ reads object A

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. The value stored is a “later” version (write-read), so $\mathcal{T}$ is aborted.

When $\mathcal{T}$ reads object A

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. The value stored is a “later” version (write-read), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) \leq \mathrm{TS}(\mathcal{T})$ **then**

When $\mathcal{T}$ reads object A

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. The value stored is a “later” version (write-read), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) \leq \mathrm{TS}(\mathcal{T})$ **then**

1. The value stored is an earlier version, so is safe to read.
2. Update $\mathrm{RTS}(A) \leftarrow \max(\mathrm{RTS}(A), \mathrm{TS}(\mathcal{T}))$

When $\mathcal{T}$ writes object A

When $\mathcal{T}$ writes object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ then

When $\mathcal{T}$ writes object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

When $\mathcal{T}$ writes object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

When $\mathcal{T}$ writes object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already written a new version (write-write), so $\mathcal{T}$ is aborted.

When $\mathcal{T}$ writes object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already written a new version (write-write), so $\mathcal{T}$ is aborted.

Otherwise

When $\mathcal{T}$ writes object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already written a new version (write-write), so $\mathcal{T}$ is aborted.

Otherwise

1. The value is safe to write.
2. Update $\mathrm{WTS}(A) \leftarrow \max(\mathrm{WTS}(A), \mathrm{TS}(\mathcal{T}))$

Optimization

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

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

This schedule is not conflict serializable

Optimization

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

This schedule is not conflict serializable … but it could be

Optimization

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

This schedule is not conflict serializable ... but it could be

When $\mathcal{T}$ reads object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already written a new version (write-write), so $\mathcal{T}$ is aborted.

Otherwise

1. The value is safe to write.
2. Update $\mathrm{WTS}(A) \leftarrow \max(\mathrm{WTS}(A), \mathrm{TS}(\mathcal{T}))$

When $\mathcal{T}$ reads object A

If $\mathrm{RTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already read the stored version (read-write), so $\mathcal{T}$ is aborted.

If $\mathrm{WTS}(A) > \mathrm{TS}(\mathcal{T})$ **then**

1. A later transaction has already written a new version (write-write), so **the write is ignored**

Otherwise

1. The value is safe to write.
2. Update $\mathrm{WTS}(A) \leftarrow \max(\mathrm{WTS}(A), \mathrm{TS}(\mathcal{T}))$

Optimization

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

Optimization

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

Optimization

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

Optimization

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

… but this schedule isn’t conflict serializable

View Serializability

Conflict Equivalence

The order of two operations has no effect on the transaction's visible if the operations:

1. ... act on different objects
2. ... or are both reads

View Equivalence

The order of two operations has no effect on the transaction's visible if the operations:

1. ... act on different objects
2. ... or are both reads
3. ... are both writes and their effects are hidden by a third write

View Equivalence

The order of two operations has no effect on the transaction's visible if the operations:

1. ... act on different objects
2. ... or are both reads
3. ... are both writes and their effects are hidden by a third write

A schedule is **view serializable** if it is view equivalent to a serial schedule

View Serializability

Timestamp Concurrency Control guarantees view serializable schedules.

Serializability

- Serializability is guaranteed by...
  - View serializability, which is guaranteed by...
    - Timestamp concurrency control
    - Conflict serializability, which is guaranteed by...
      - 2-Phase Locking
      - Optimistic Concurrency Control (aka Snapshot Isolation[1])

[1] Database vendors, in their markety wisdom (incorrectly) refer to a weaker form of TSCC that we'll talk about next as "Snapshot Isolation". This term is, accordingly, ambiguous.

Optimization?

Problem: Adding timestamp checks on reads is **expensive**.

Optimization?

Solution?: Ignore read timestamps.

(Major database vendors (incorrectly) call this “snapshot isolation”)

Write-Write conflicts

Write-Read conflicts

Read-Write conflicts

Write-Write conflicts

Detected!

Write-Read conflicts

Read-Write conflicts

Write-Write conflicts

Detected!

Write-Read conflicts

Detected!

Read-Write conflicts

Write-Write conflicts

Detected!

Write-Read conflicts

Detected!

Read-Write conflicts

Ignored!

```
A = 100          def T1():      def T2():
B = 200            A = B+1        B = A+1
```

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

Errors

```
A = 100          def T1():      def T2():
B = 200            A = B+1        B = A+1
```

Time	T1	T2	
↓		R(A)	$A \rightarrow 100$
↓	R(B)		
↓	W(A)		
↓		W(B)	

```
A = 100          def T1():      def T2():
B = 200            A = B+1        B = A+1
```

Time	T1		T2	
↓			R(A)	$A \to 100$
↓	R(B)	$B \to 200$		
↓	W(A)			
↓			W(B)	

```
A = 100          def T1():      def T2():
B = 200            A = B+1        B = A+1
```

Time	T1		T2	
↓			R(A)	$A \rightarrow 100$
↓	R(B)	$B \rightarrow 200$		
↓	W(A)	$A \leftarrow 201$		
↓			W(B)	

```
A = 100          def T1():      def T2():
B = 200            A = B+1        B = A+1
```

Time	T1		T2	
↓			R(A)	$A \rightarrow 100$
↓	R(B)	$B \rightarrow 200$		
↓	W(A)	$A \leftarrow 201$		
↓			W(B)	$B \leftarrow 101$

```
A = 100          def T1():      def T2():
B = 200            A = B+1        B = A+1
```

Time	T1		T2	
↓			R(A)	$A \rightarrow 100$
↓	R(B)	$B \rightarrow 200$		
↓	W(A)	$A \leftarrow 201$		
↓			W(B)	$B \leftarrow 101$

This schedule would be permitted!

Recoverability

Recoverability

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

Recoverability

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

Oops...

Until a transaction $\mathcal{T}$ commits successfully...

Until a transaction $\mathcal{T}$ commits successfully...

1. Buffer all writes (but update WTS immediately)

Until a transaction $\mathcal{T}$ commits successfully...

1. Buffer all writes (but update WTS immediately)
2. Block transactions reading from $\mathcal{T}$'s writes from committing

Until a transaction $\mathcal{T}$ commits successfully...

1. Buffer all writes (but update WTS immediately)
2. Block transactions reading from $\mathcal{T}$'s writes from committing
3. Abort transactions reading from $\mathcal{T}$'s writes if $\mathcal{T}$ aborts

Multiversion Timestamp Concurrency Control

Write-Read conflicts are avoidable

(the necessary data was available at one point)

Idea: Keep old versions of objects around

Each object version tracks...

- The writing transaction's WTS
- The greatest TS of any transaction to read it as an RTS

When $\mathcal{T}$ reads an object A

1. Find the newest version A_i such that $\mathrm{WTS}(A_i) < \mathrm{TS}(\mathcal{T})$
   This is always safe!
2. Update $\mathrm{RTS}(A_i) \leftarrow \max(\mathrm{RTS}(A_i), \mathrm{TS}(\mathcal{T}))$

When $\mathcal{T}$ writes an object A

1. Find the newest version A_i such that $\mathrm{WTS}(A_i) < \mathrm{TS}(\mathcal{T})$

If $\mathrm{RTS}(A_i) < \mathrm{TS}(\mathcal{T})$

2. The read is unrecoverable, as a later transaction should have read this version. Abort $\mathcal{T}$

Otherwise

2. Write the new version in between A_i and A_{i+1}

Next time...