Versioned Boxes as the Basis for Memory Transactions

João Cachopo and António Rito-Silva

Inesc-ID/Technical University of Lisbon

October 16, 2005

Abstract

Context

Optimistic Software Transactional Memory.

Problem

How can we reduce conflicts?

Proposals

- Use versioned boxes to hold the shared state.
- Delay reads for high-contention boxes.
- Use restartable nested transactions for high-contention sections of code.

Abstract

Context

Optimistic Software Transactional Memory.

Problem

How can we reduce conflicts?

Proposals

- Use versioned boxes to hold the shared state.
- Delay reads for high-contention boxes.
- Use restartable nested transactions for high-contention sections of code.

Abstract

Context

Optimistic Software Transactional Memory.

Problem

How can we reduce conflicts?

Proposals

- Use versioned boxes to hold the shared state.
- Delay reads for high-contention boxes.
- Use restartable nested transactions for high-contention sections of code.

Abstract

Context

Optimistic Software Transactional Memory.

Problem

How can we reduce conflicts?

Proposals

- Use versioned boxes to hold the shared state.
- Delay reads for high-contention boxes.
- Use restartable nested transactions for high-contention sections of code.

inesc id lisboa

Abstract

Context

Optimistic Software Transactional Memory.

Problem

How can we reduce conflicts?

Proposals

- Use versioned boxes to hold the shared state.
- Delay reads for high-contention boxes.
- Use restartable nested transactions for high-contention sections of code.

inesc id lisboa

Outline

Outline

Outline

Outline

Outline

Implemented as a Java library

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

Implemented as a Java library

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

Implemented as a Java library

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

Implemented as a Java library

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

Implemented as a Java library

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

Implemented as a Java library

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

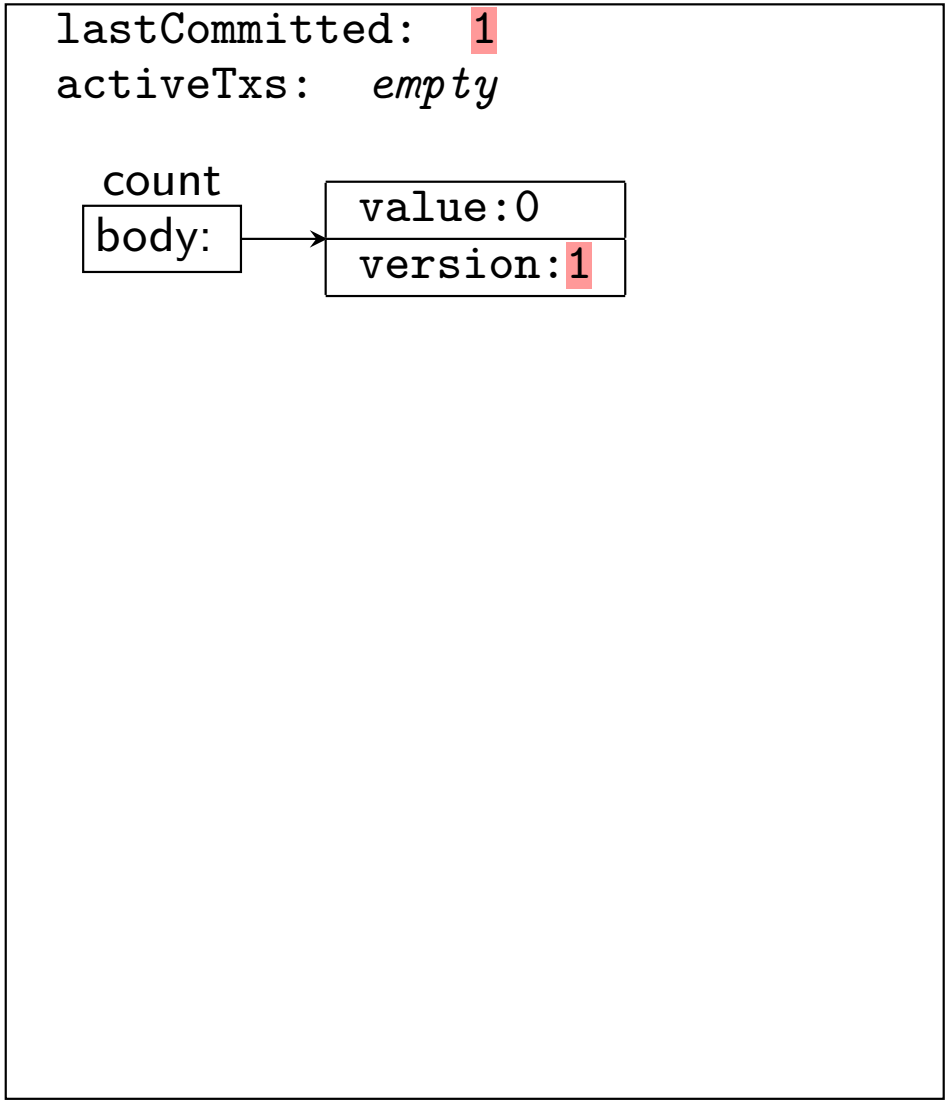
```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

In Action

lastCommitted: 0
activeTxs: *empty*

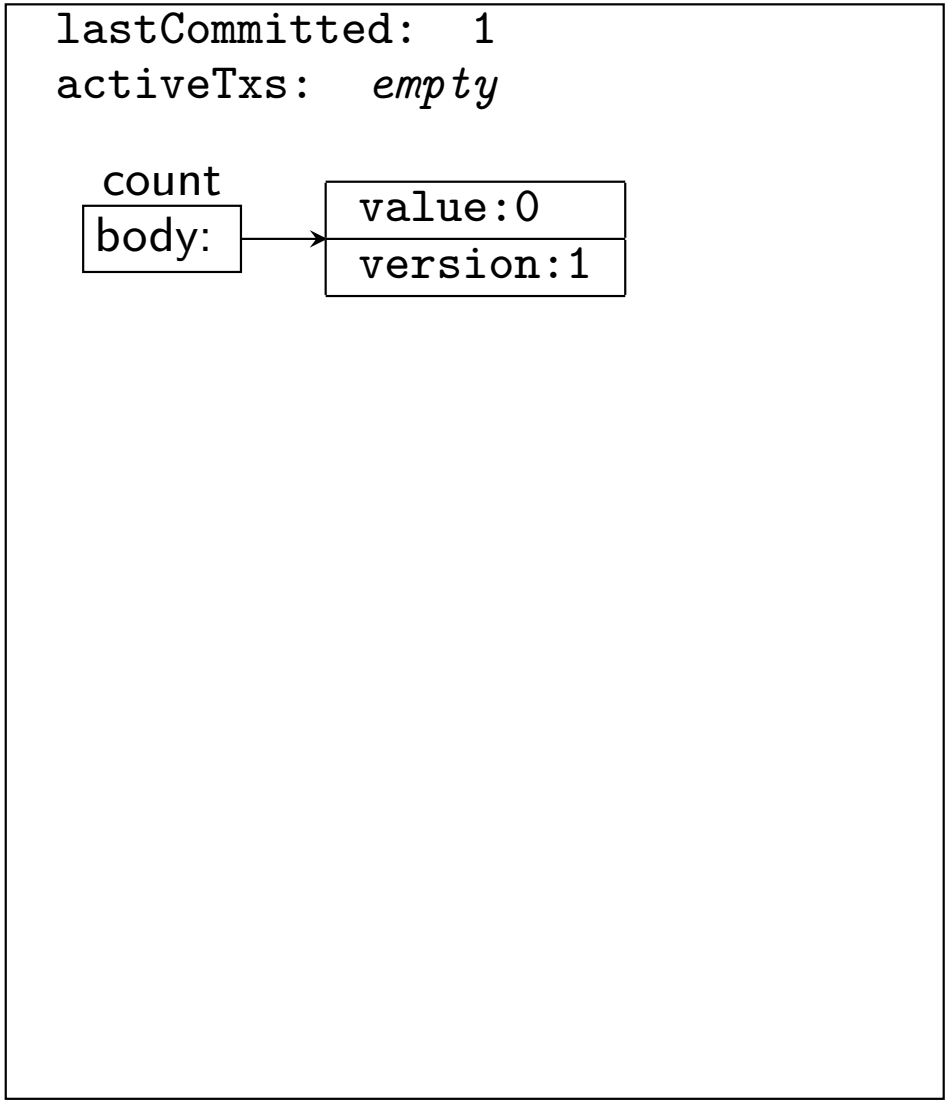
```
Counter counter = new Counter();
spawnThreads(counter);
```

In Action




```
Counter counter = new Counter();
spawnThreads(counter);
```

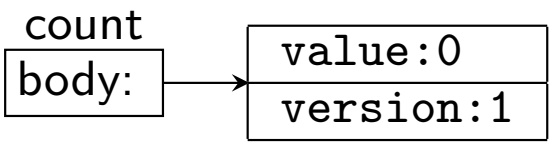
In Action




```
Counter counter = new Counter();
spawnThreads(counter);
```

In Action

lastCommitted: 1
activeTxs: *empty*



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

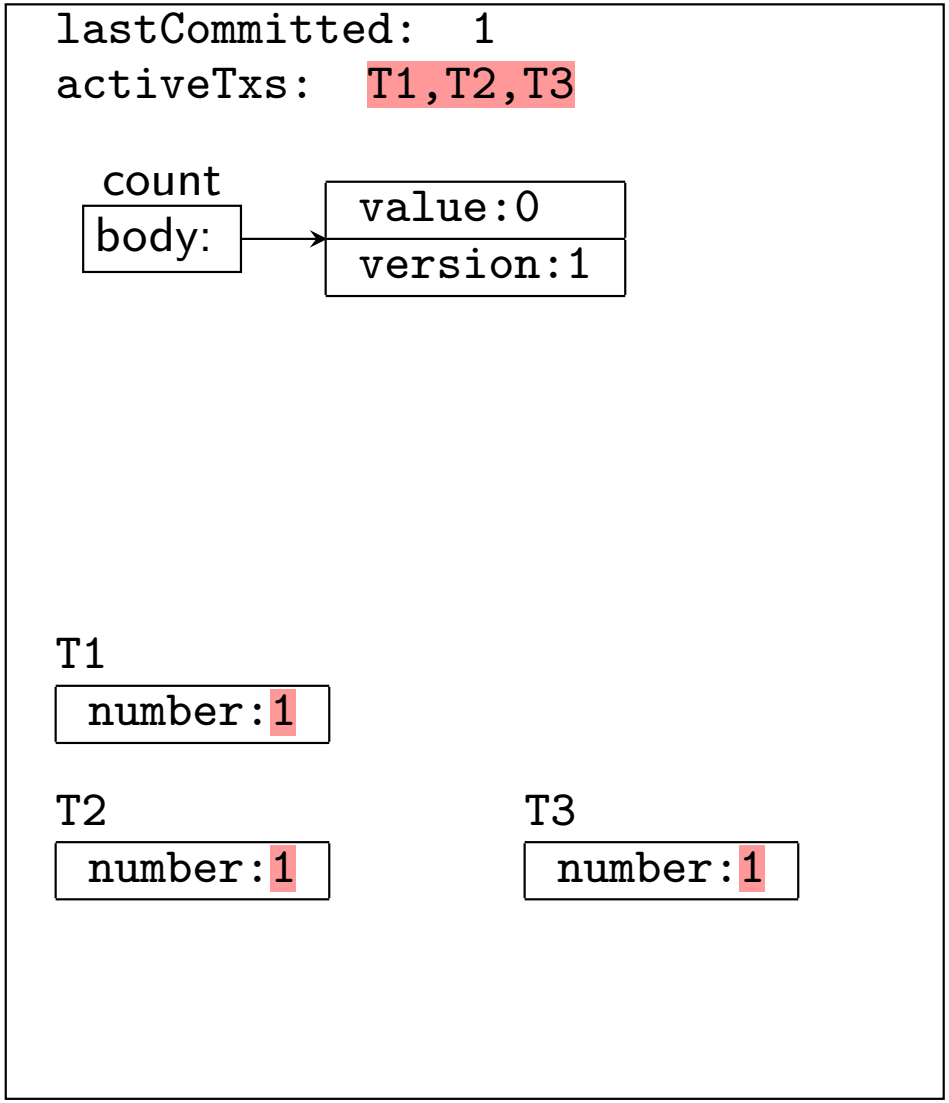
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

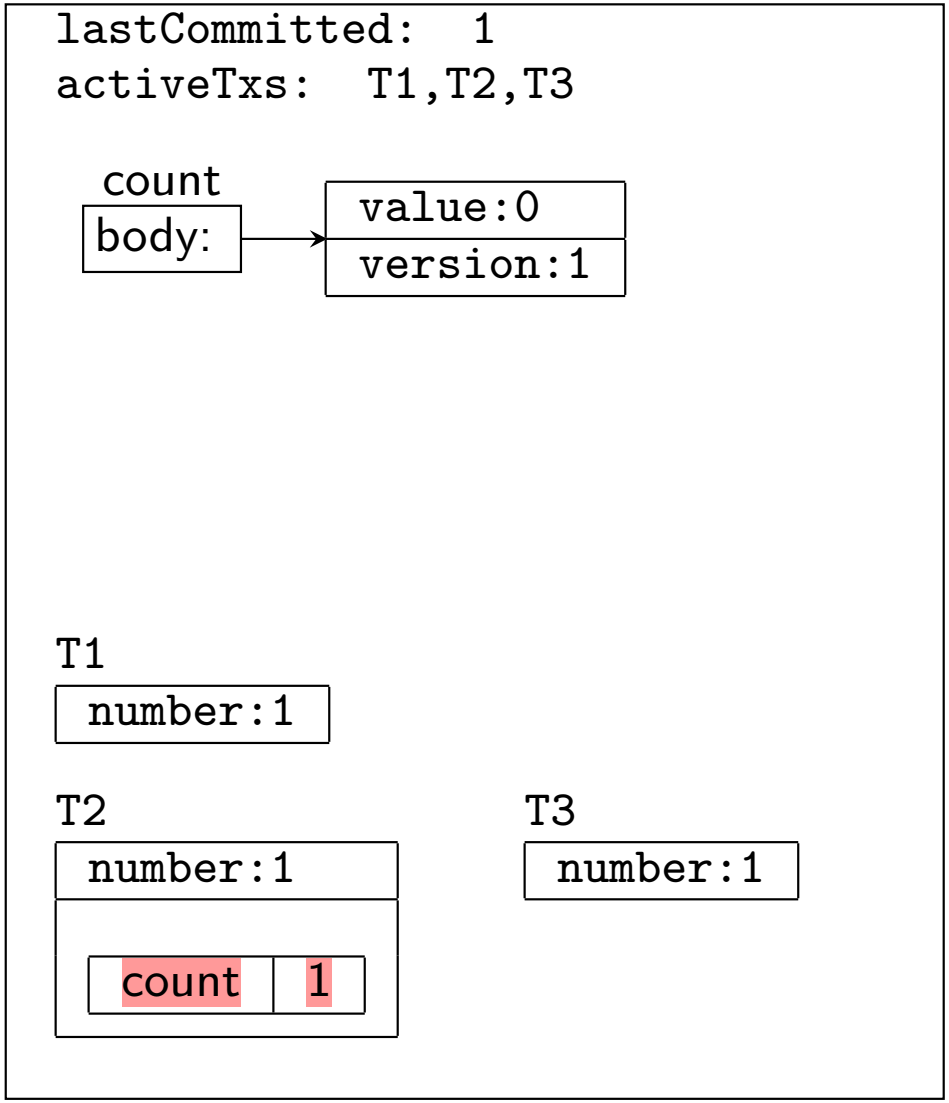
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

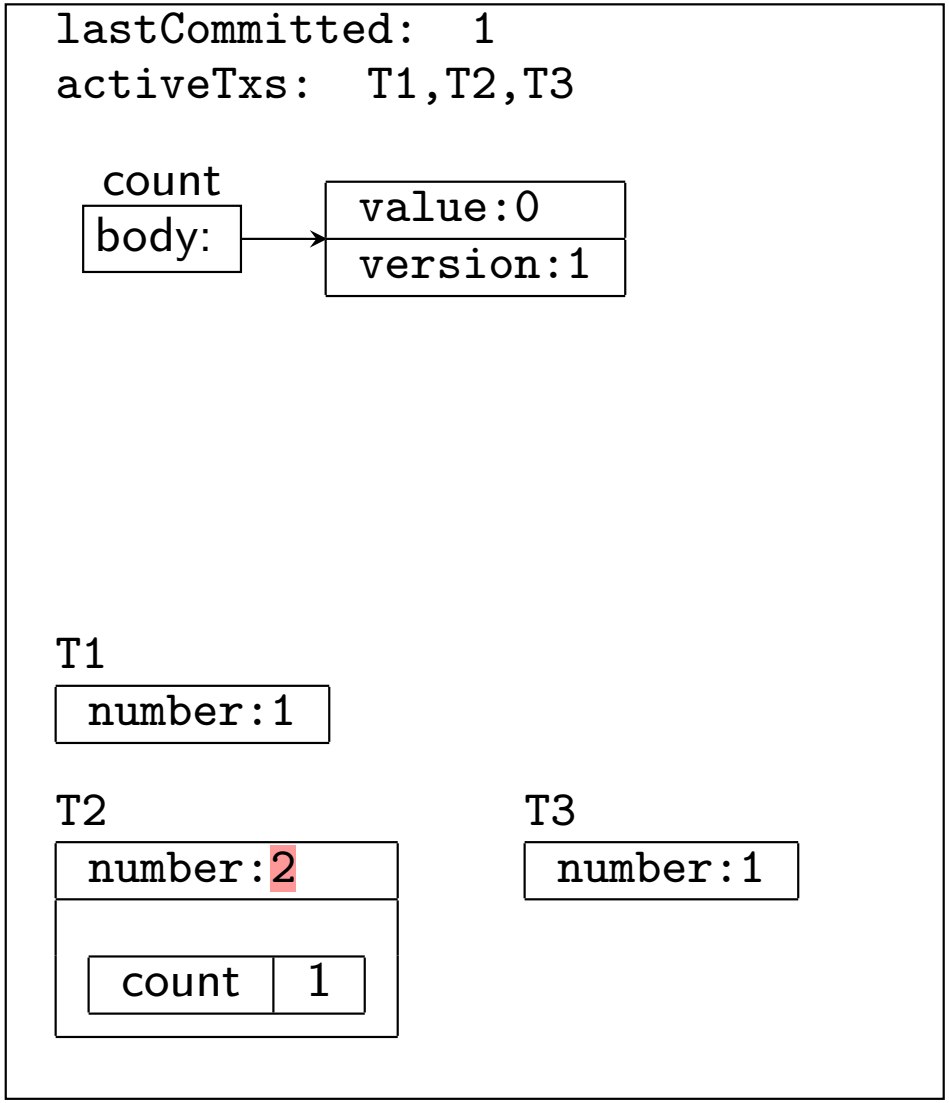
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

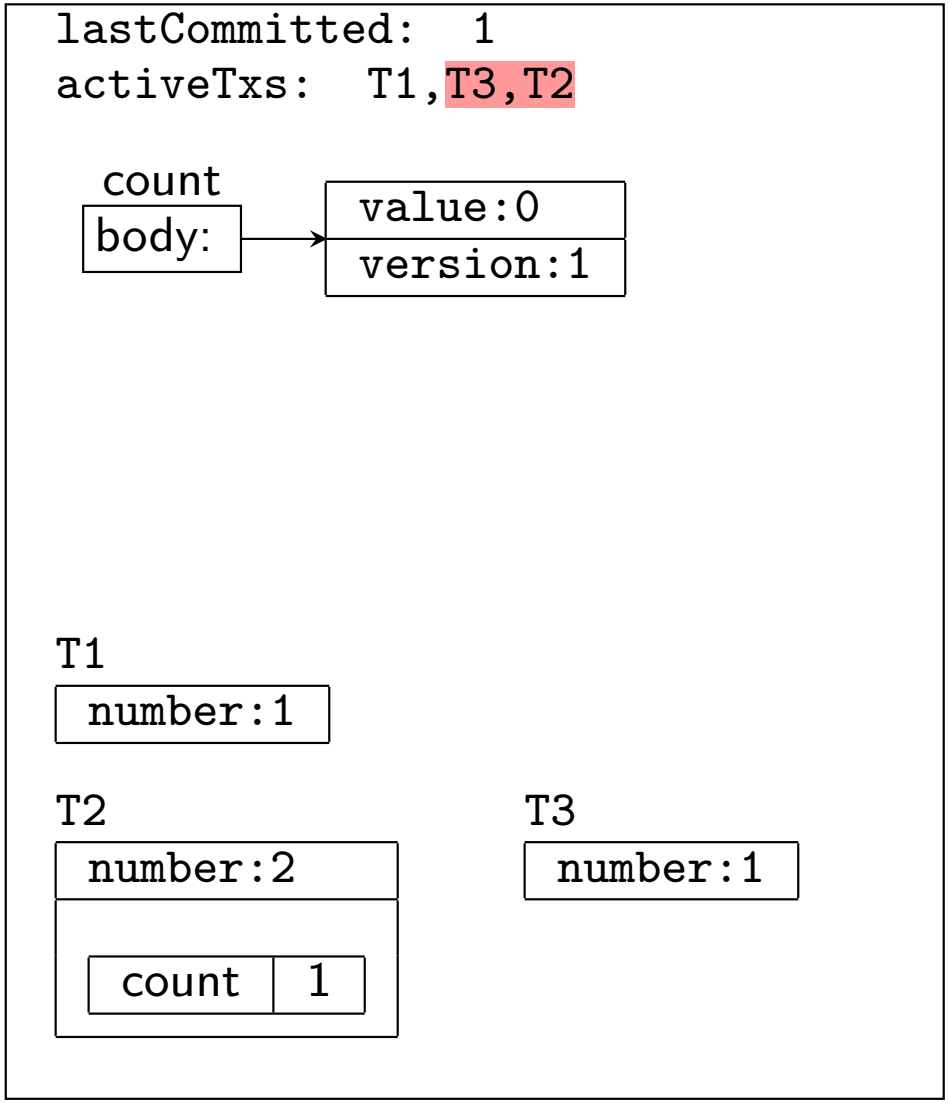
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

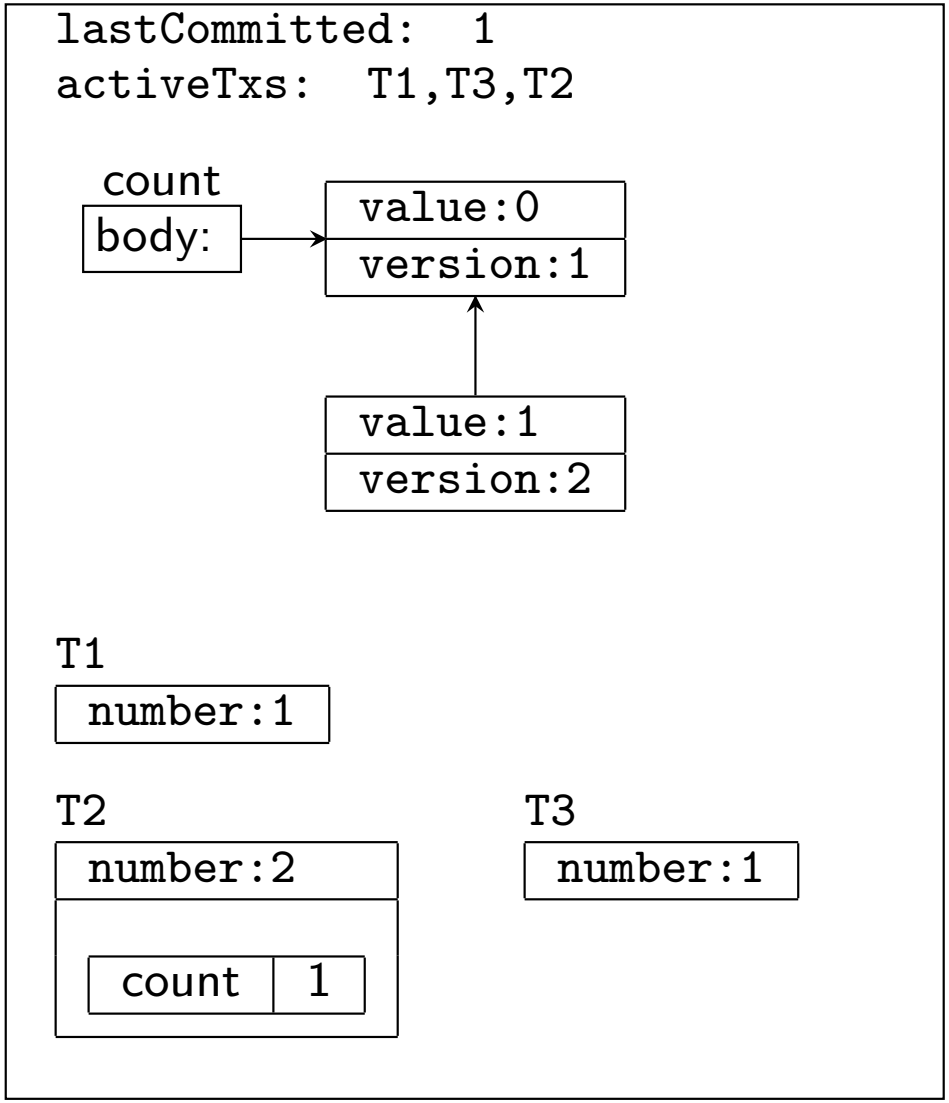
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

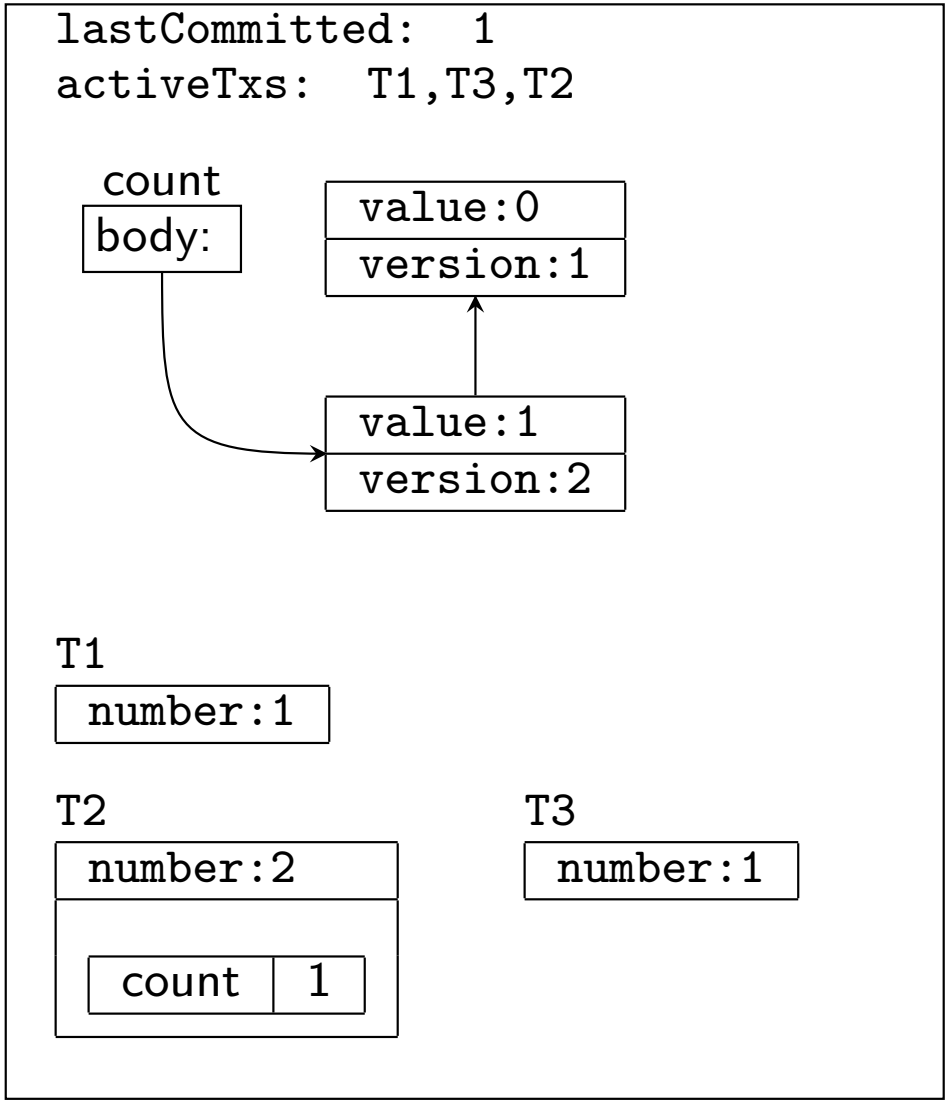
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

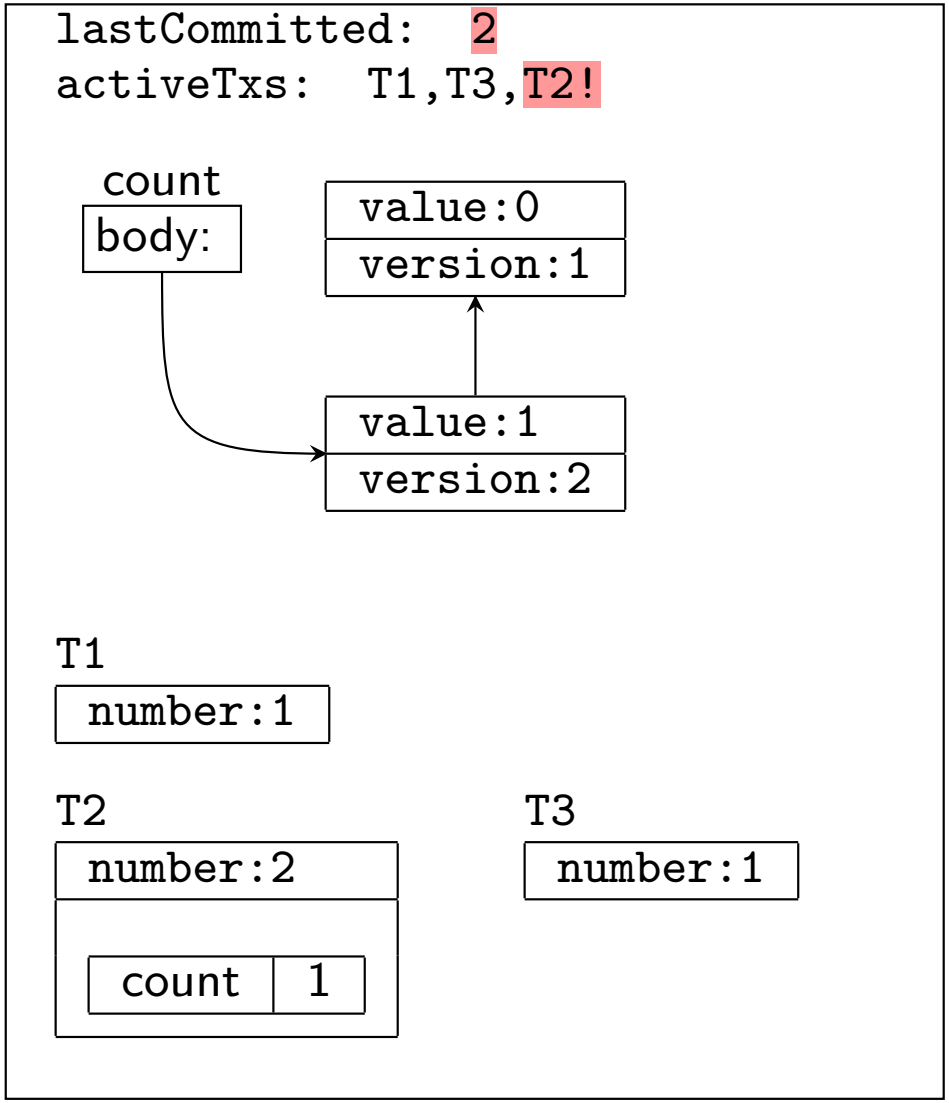
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

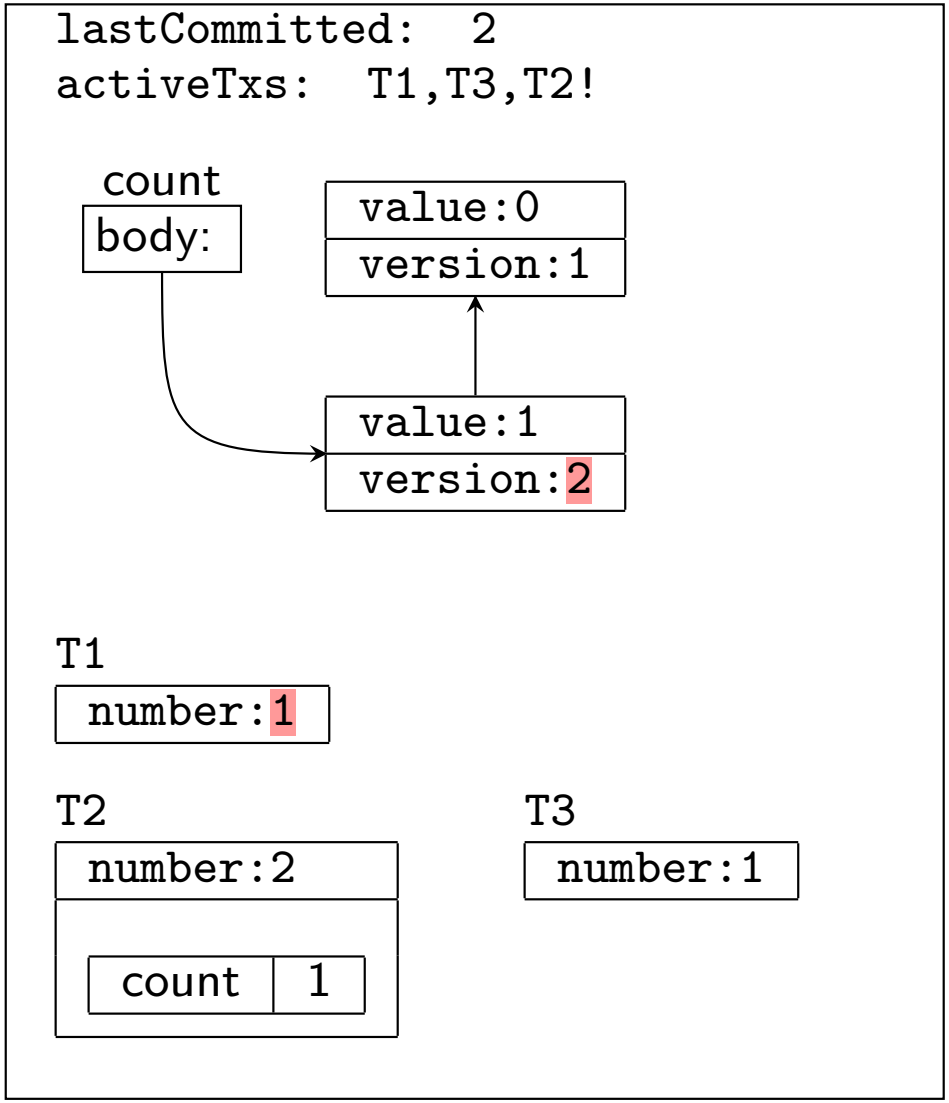
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

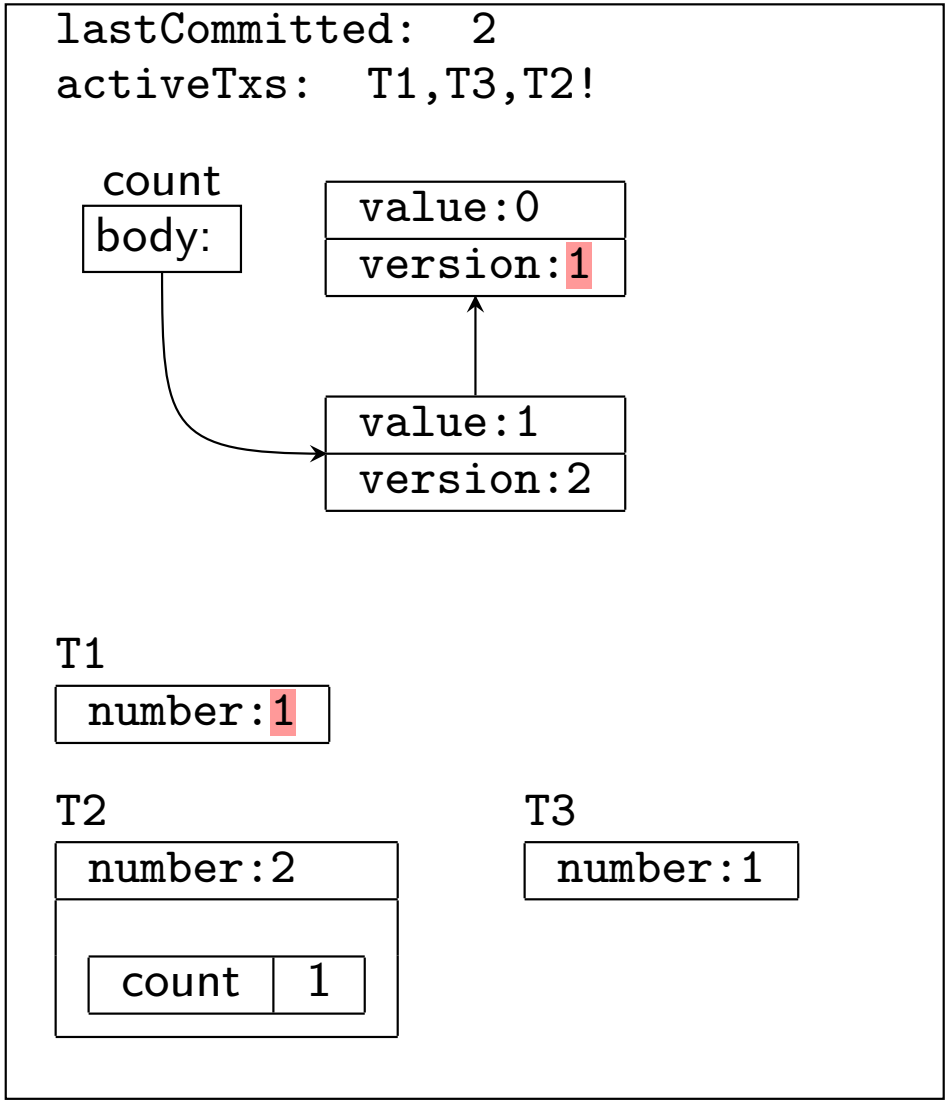
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

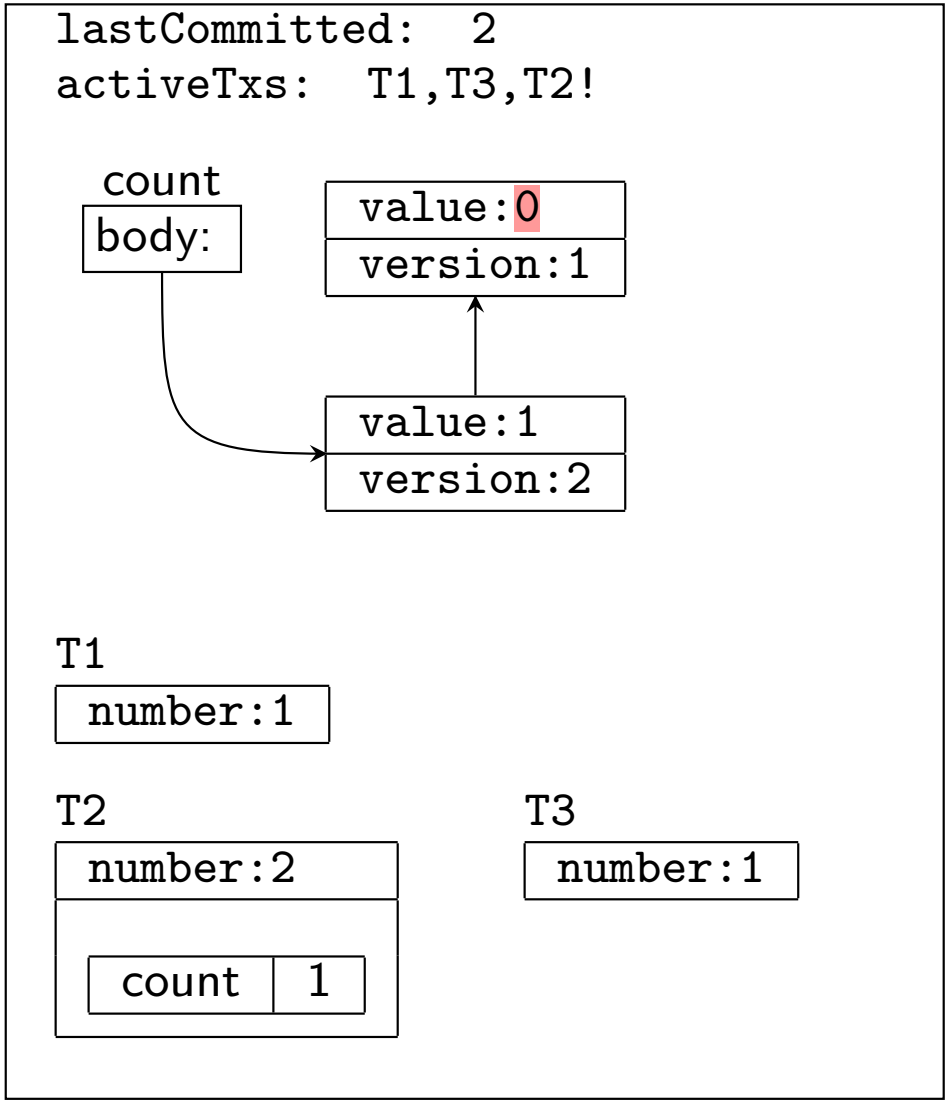
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

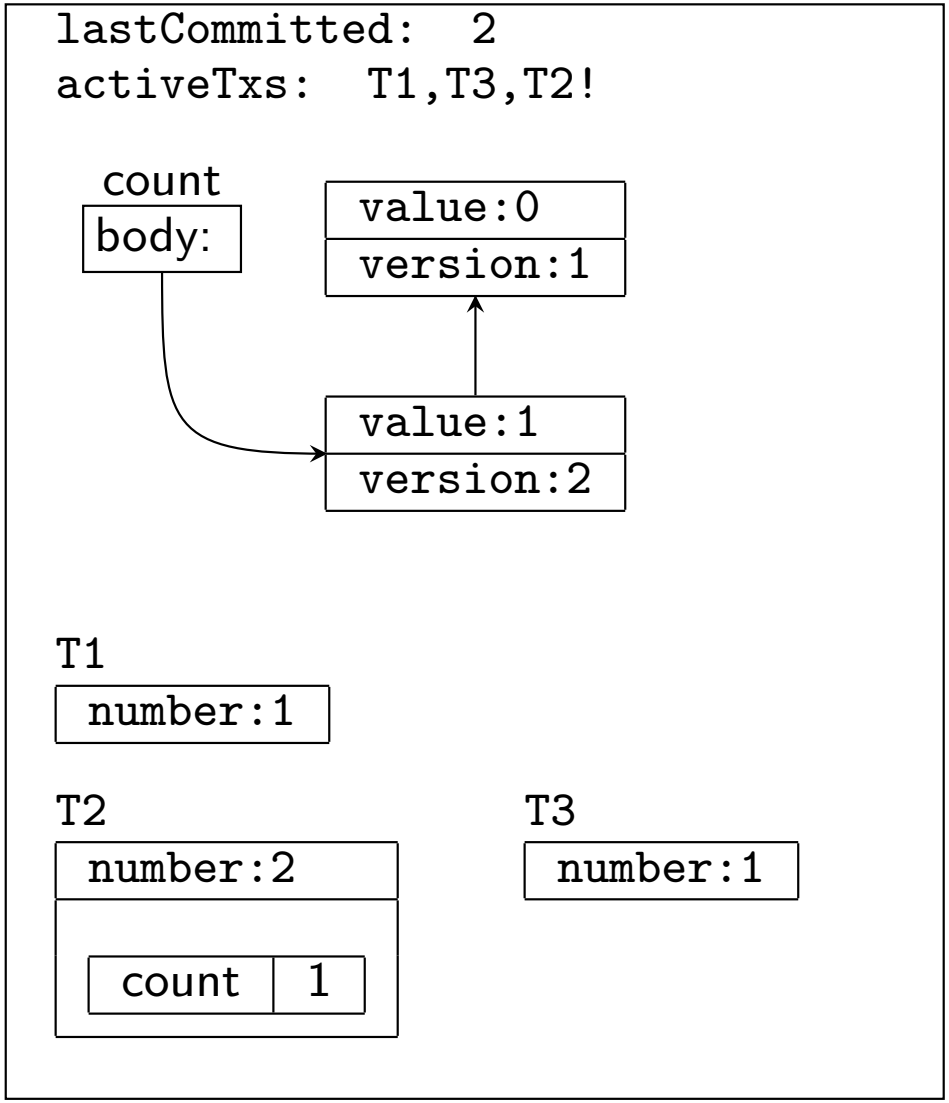
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

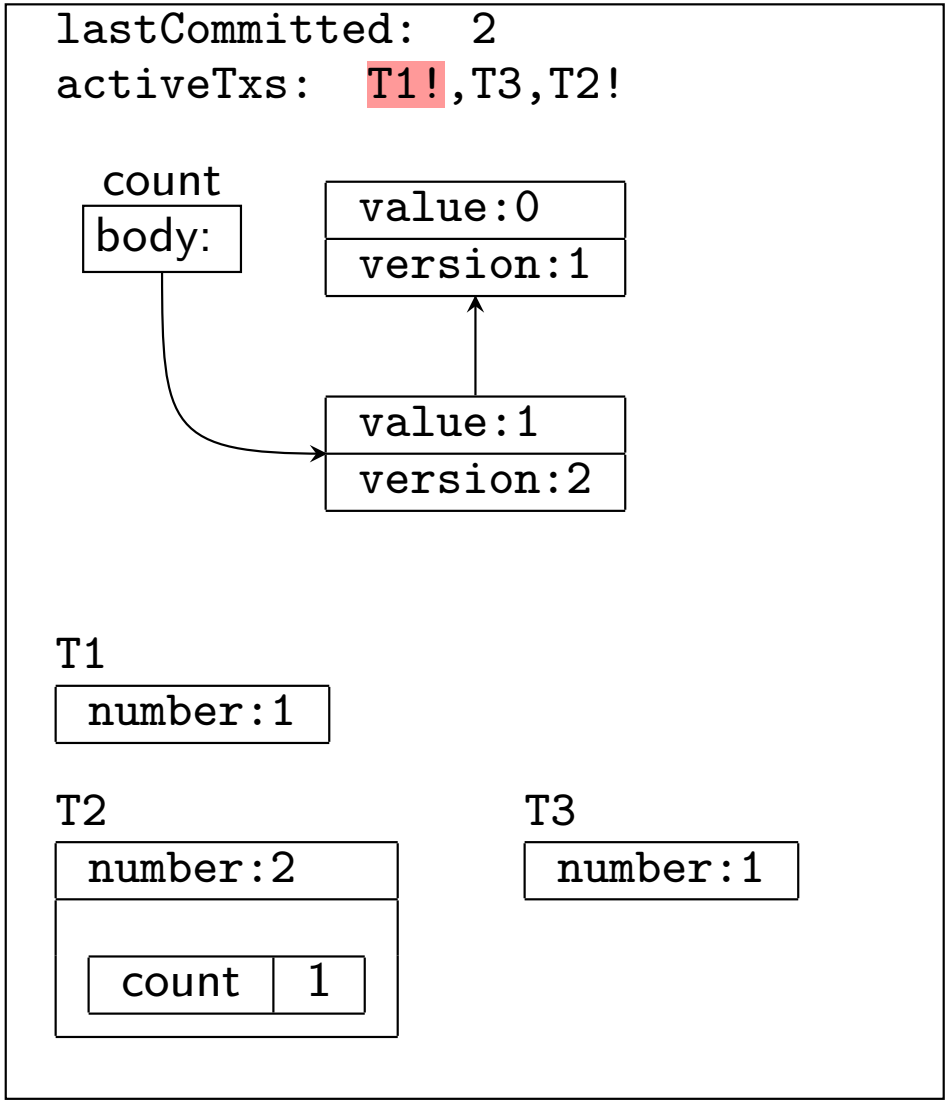
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

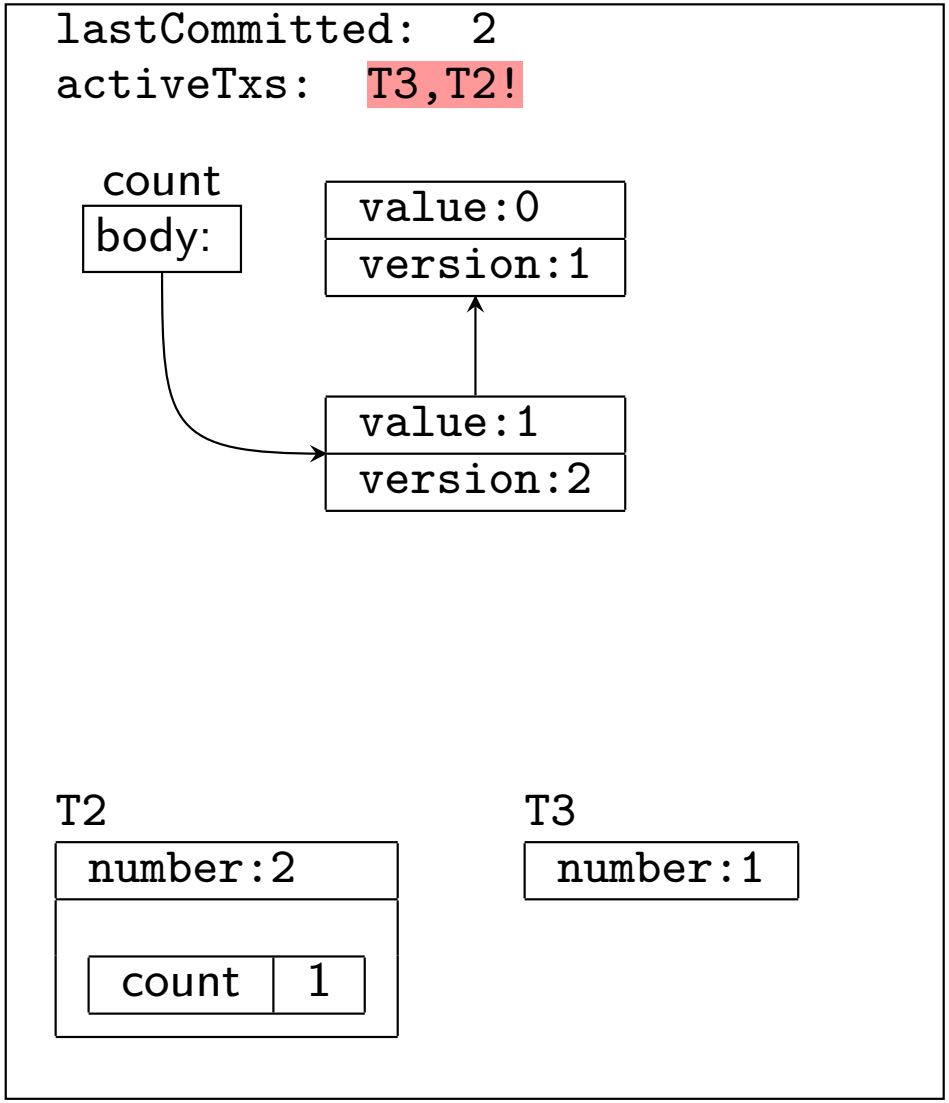
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

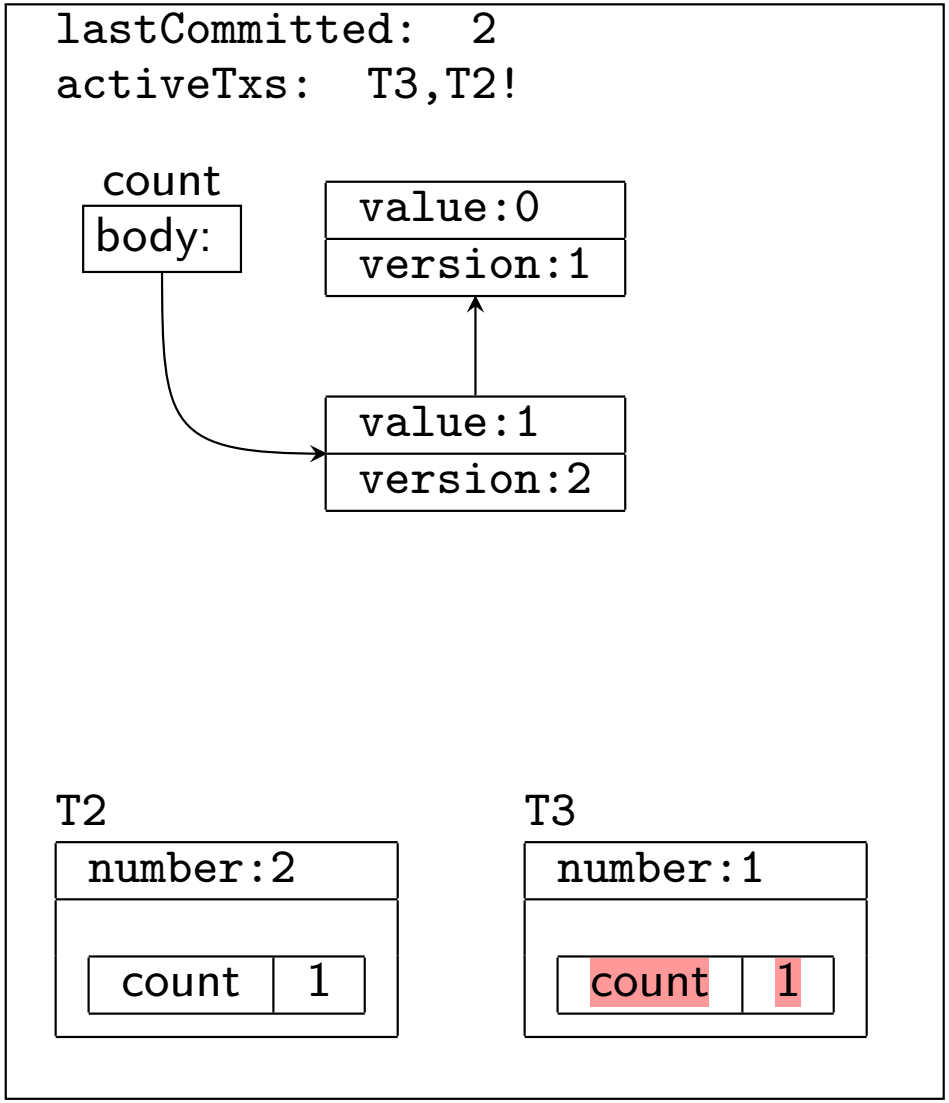
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

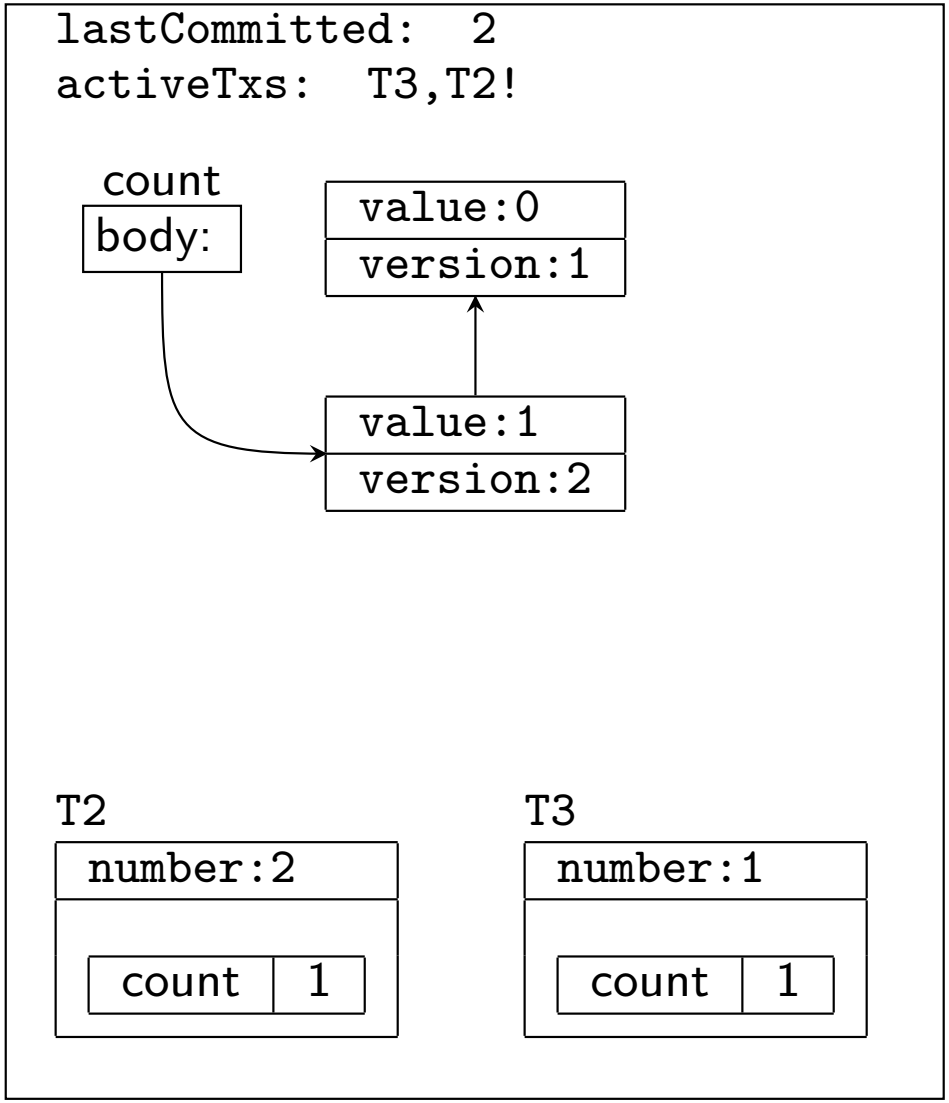
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

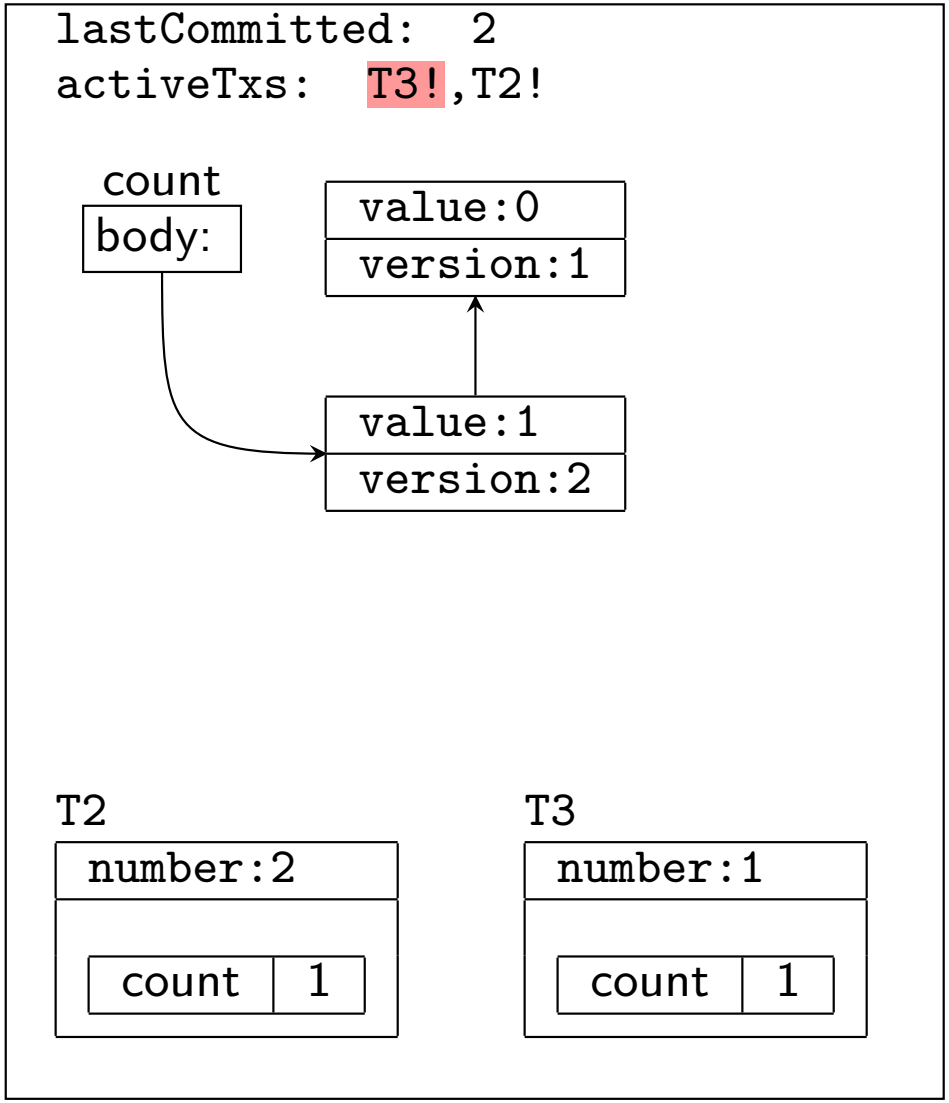
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

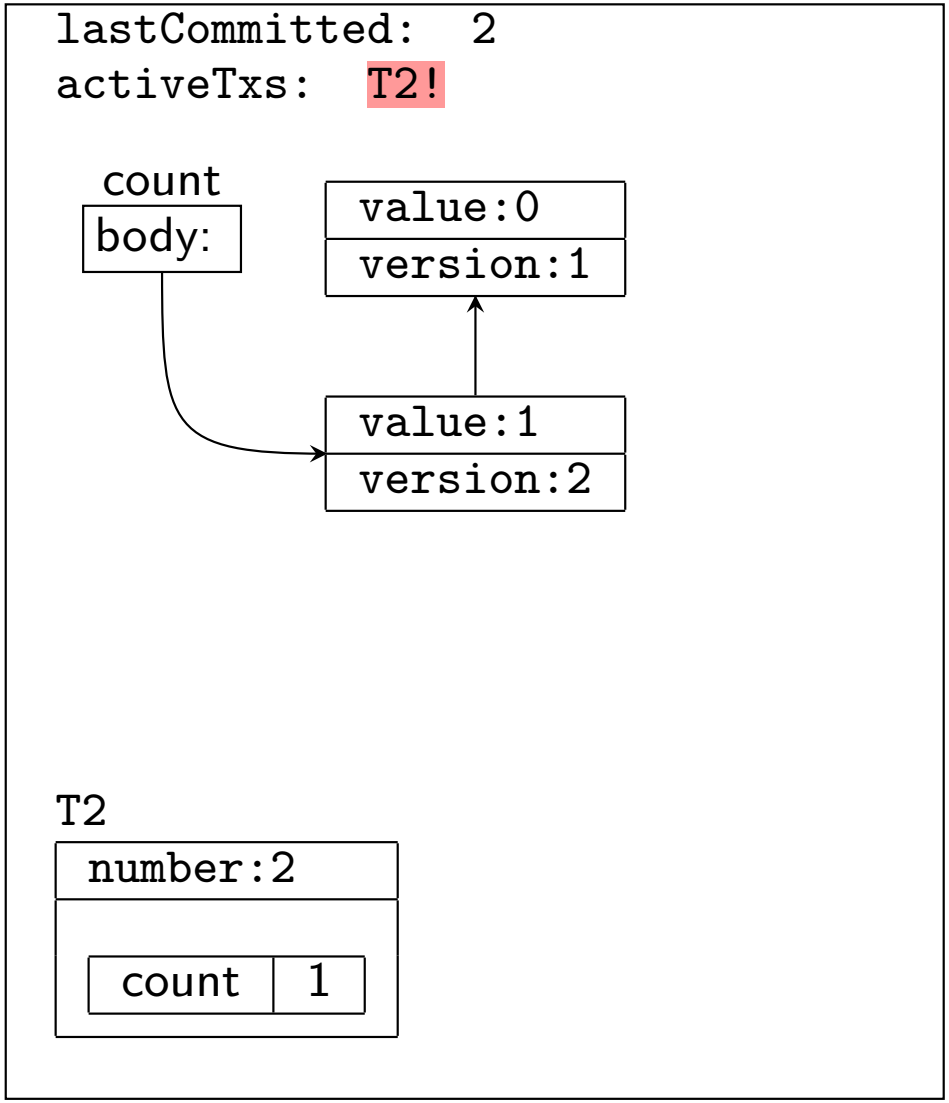
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

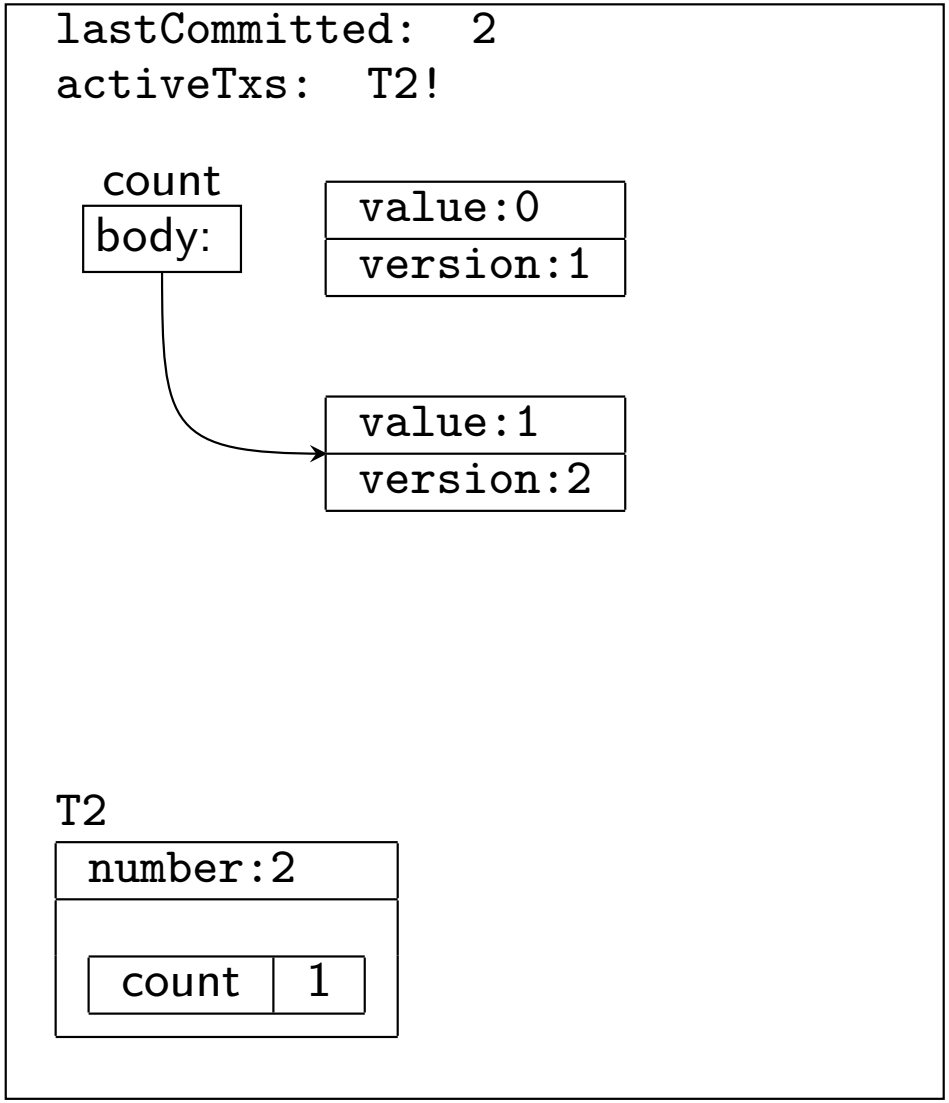
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

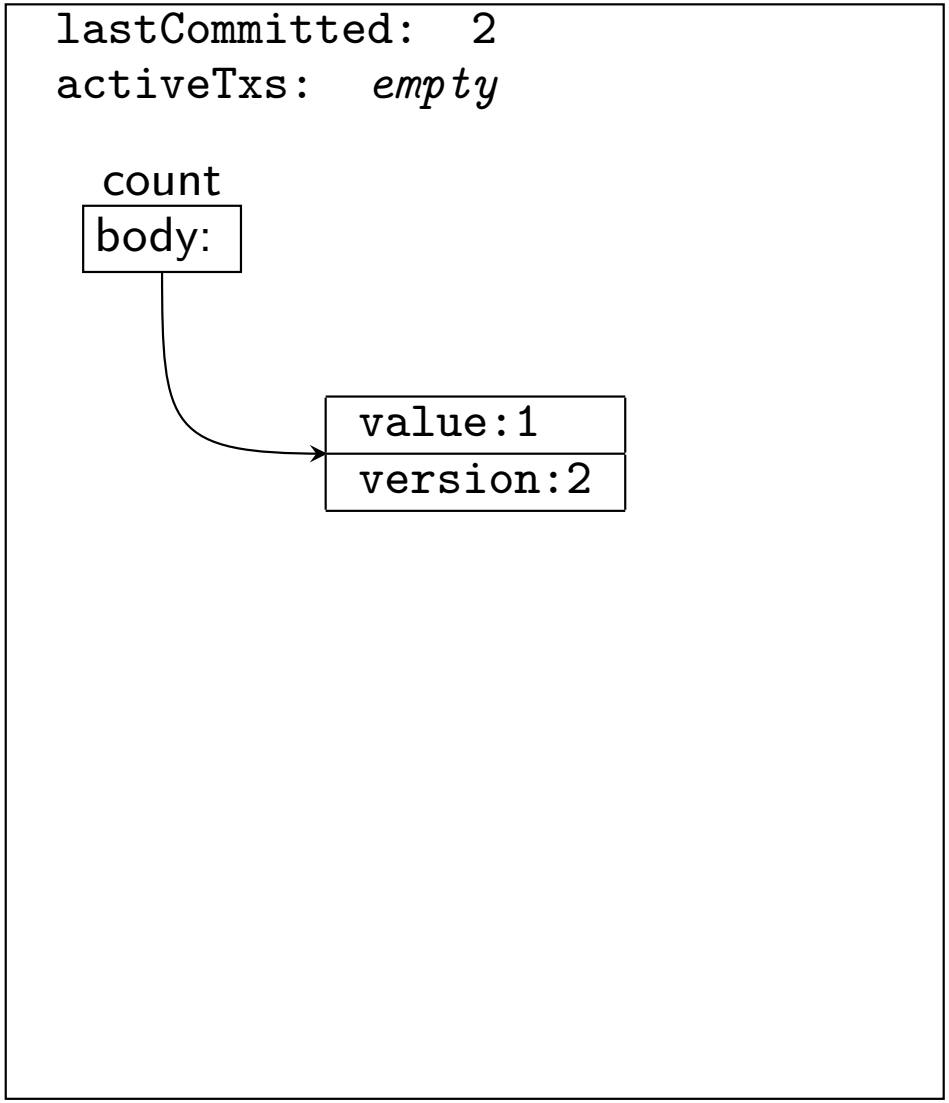
T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

In Action



T1

```
Transaction.start();
print(counter.getCount());
Transaction.commit();
```

T2

```
Transaction.start();
counter.inc();
Transaction.commit();
```

T3

```
Transaction.start();
counter.inc();
Transaction.commit();
```

Outline

When We Can Delay the Reads

Consider a shared `Counter` instance that is incremented in all transactions.

Then, all transactions conflict with each other if executed concurrently.

What can we do about it?

If the value of the counter is not used in the transaction, then we can delay the increment, thereby avoiding the read that causes the conflict.

When We Can Delay the Reads

Consider a shared `Counter` instance that is incremented in all transactions.

Then, all transactions conflict with each other if executed concurrently.

What can we do about it?

If the value of the counter is not used in the transaction, then we can delay the increment, thereby avoiding the read that causes the conflict.

When We Can Delay the Reads

Consider a shared `Counter` instance that is incremented in all transactions.

Then, all transactions conflict with each other if executed concurrently.

What can we do about it?

If the value of the counter is not used in the transaction, then we can delay the increment, thereby avoiding the read that causes the conflict.

When We Can Delay the Reads

Consider a shared `Counter` instance that is incremented in all transactions.

Then, all transactions conflict with each other if executed concurrently.

What can we do about it?

If the value of the counter is not used in the transaction, then we can delay the increment, thereby avoiding the read that causes the conflict.

Using Per-Transaction Boxes to Delay Operations

```
public class CFOCounter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

Using Per-Transaction Boxes to Delay Operations

```
public class CFOCounter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

Using Per-Transaction Boxes to Delay Operations

```
public class CFOCounter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

Using Per-Transaction Boxes to Delay Operations

```
public class CFOCounter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

Using Per-Transaction Boxes to Delay Operations

```
public class CFOCounter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

When We Cannot Delay the Reads

What if we cannot delay the reads?

Restarting the whole transaction because a small part of it caused the conflict is rather drastic.

If the re-execution of the conflicting part, with the commit-time values, produces the same results, then it is safe to commit.

When We Cannot Delay the Reads

What if we cannot delay the reads?

Restarting the whole transaction because a small part of it caused the conflict is rather drastic.

If the re-execution of the conflicting part, with the commit-time values, produces the same results, then it is safe to commit.

When We Cannot Delay the Reads

What if we cannot delay the reads?

Restarting the whole transaction because a small part of it caused the conflict is rather drastic.

If the re-execution of the conflicting part, with the commit-time values, produces the same results, then it is safe to commit.

Using Restartable Transactions

```
class List {
  ...
  @Restartable boolean contains(Object obj) {
    ...contains body...
  }
}
```

A transaction that uses the `List.contains` method can re-execute the contains method, if a conflict is on a box that was read only by this method.

If the result of the method is the same, then it is safe to commit.

Using Restartable Transactions

```
class List {
  ...
  @Restartable boolean contains(Object obj) {
    ...contains body...
  }
}
```

A transaction that uses the `List.contains` method can re-execute the contains method, if a conflict is on a box that was read only by this method.

If the result of the method is the same, then it is safe to commit.

Using Restartable Transactions

```
class List {
  ...
  @Restartable boolean contains(Object obj) {
    ...contains body...
  }
}
```

A transaction that uses the `List.contains` method can re-execute the contains method, if a conflict is on a box that was read only by this method.

If the result of the method is the same, then it is safe to commit.

Using Restartable Transactions

```
class List {
  ...
  @Restartable boolean contains(Object obj) {
    ...contains body...
  }
}
```

A transaction that uses the `List.contains` method can re-execute the contains method, if a conflict is on a box that was read only by this method.

If the result of the method is the same, then it is safe to commit.

Outline

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes. . .
. . . and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

- the read-set is still valid
- the read-set is not invalidated by the restarted transaction, but the read-set is still valid
- the read-set is invalidated by the restarted transaction, but the read-set is still valid and the read-set is equal

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes...
...and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes...
...and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

- It produces the same return value
- Each box written by the restartable transaction was read only by restartable transactions, which should succeed in their re-execution

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes...
...and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

- It produces the same return value.
- Each box written by the restartable transaction was read only by restartable transactions, which should succeed in their re-execution.

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes...
...and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

- It produces the same return value.
- Each box written by the restartable transaction was read only by restartable transactions, which should succeed in their re-execution.

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes...
...and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

- It produces the same return value.
- Each box written by the restartable transaction was read only by restartable transactions, which should succeed in their re-execution.

Unifying both Approaches

In both cases, we are moving for commit-time some computation.

However, in the `Counter` example, the operation also writes...
...and the result of the re-execution will not be the same.

Still, this different result does not interfere with the transaction execution flow.

A restartable transaction succeeds if:

- It produces the same return value.
- Each box written by the restartable transaction was read only by restartable transactions, which should succeed in their re-execution.

Towards Fine-grained Restarts

The main difficulties in the implementation of restartable transactions are:

- Freezing the execution context of each restartable transaction.
- Keeping track of dependencies between transactions that read and write the same boxes.

This is hard because the results of nested transactions are merged with their parents.

Fundamental change

Extend the versioned boxes model, so that nested transactions behave in the same way as top-level transactions.

Towards Fine-grained Restarts

The main difficulties in the implementation of restartable transactions are:

- Freezing the execution context of each restartable transaction.
- Keeping track of dependencies between transactions that read and write the same boxes.

This is hard because the results of nested transactions are merged with their parents.

Fundamental change

Extend the versioned boxes model, so that nested transactions behave in the same way as top-level transactions.

Towards Fine-grained Restarts

The main difficulties in the implementation of restartable transactions are:

- Freezing the execution context of each restartable transaction.
- Keeping track of dependencies between transactions that read and write the same boxes.

This is hard because the results of nested transactions are merged with their parents.

Fundamental change

Extend the versioned boxes model, so that nested transactions behave in the same way as top-level transactions.

Towards Fine-grained Restarts

The main difficulties in the implementation of restartable transactions are:

- Freezing the execution context of each restartable transaction.
- Keeping track of dependencies between transactions that read and write the same boxes.

This is hard because the results of nested transactions are merged with their parents.

Fundamental change

Extend the versioned boxes model, so that nested transactions behave in the same way as top-level transactions.

Towards Fine-grained Restarts

The main difficulties in the implementation of restartable transactions are:

- Freezing the execution context of each restartable transaction.
- Keeping track of dependencies between transactions that read and write the same boxes.

This is hard because the results of nested transactions are merged with their parents.

Fundamental change

Extend the versioned boxes model, so that nested transactions behave in the same way as top-level transactions.

Extended Model

In this extended model:

- Transactions form a tree, where each node keeps the information of what was read and written.
- The execution context for a restartable transaction is the sub-tree that is to the left and above the transaction.
- The re-execution of a restartable transaction updates its local information only.
- Later transactions will notice that some values where changed and restart or abort.

Extended Model

In this extended model:

- Transactions form a tree, where each node keeps the information of what was read and written.
- The execution context for a restartable transaction is the sub-tree that is to the left and above the transaction.
- The re-execution of a restartable transaction updates its local information only.
- Later transactions will notice that some values where changed and restart or abort.

Extended Model

In this extended model:

- Transactions form a tree, where each node keeps the information of what was read and written.
- The execution context for a restartable transaction is the sub-tree that is to the left and above the transaction.
- The re-execution of a restartable transaction updates its local information only.
- Later transactions will notice that some values where changed and restart or abort.

Extended Model

In this extended model:

- Transactions form a tree, where each node keeps the information of what was read and written.
- The execution context for a restartable transaction is the sub-tree that is to the left and above the transaction.
- The re-execution of a restartable transaction updates its local information only.
- Later transactions will notice that some values where changed and restart or abort.

Outline

Conclusions

- By using versioned boxes, long running read-only transactions never fail.
- We can avoid conflicts by delaying computations until commit-time.
- We can resolve conflicts by re-executing the conflicting part of a transaction.
- An extended versioned model supports fine-grained restarts.

Our versioned boxes implementation is in use on a production environment, with more than 200 transactional classes, and millions of versioned boxes.

Conclusions

- By using versioned boxes, long running read-only transactions never fail.
- We can avoid conflicts by delaying computations until commit-time.
- We can resolve conflicts by re-executing the conflicting part of a transaction.
- An extended versioned model supports fine-grained restarts.

Our versioned boxes implementation is in use on a production environment, with more than 200 transactional classes, and millions of versioned boxes.

Conclusions

- By using versioned boxes, long running read-only transactions never fail.
- We can avoid conflicts by delaying computations until commit-time.
- We can resolve conflicts by re-executing the conflicting part of a transaction.
- An extended versioned model supports fine-grained restarts.

Our versioned boxes implementation is in use on a production environment, with more than 200 transactional classes, and millions of versioned boxes.

Conclusions

- By using versioned boxes, long running read-only transactions never fail.
- We can avoid conflicts by delaying computations until commit-time.
- We can resolve conflicts by re-executing the conflicting part of a transaction.
- An extended versioned model supports fine-grained restarts.

Our versioned boxes implementation is in use on a production environment, with more than 200 transactional classes, and millions of versioned boxes.

Conclusions

- By using versioned boxes, long running read-only transactions never fail.
- We can avoid conflicts by delaying computations until commit-time.
- We can resolve conflicts by re-executing the conflicting part of a transaction.
- An extended versioned model supports fine-grained restarts.

Our versioned boxes implementation is in use on a production environment, with more than 200 transactional classes, and millions of versioned boxes.