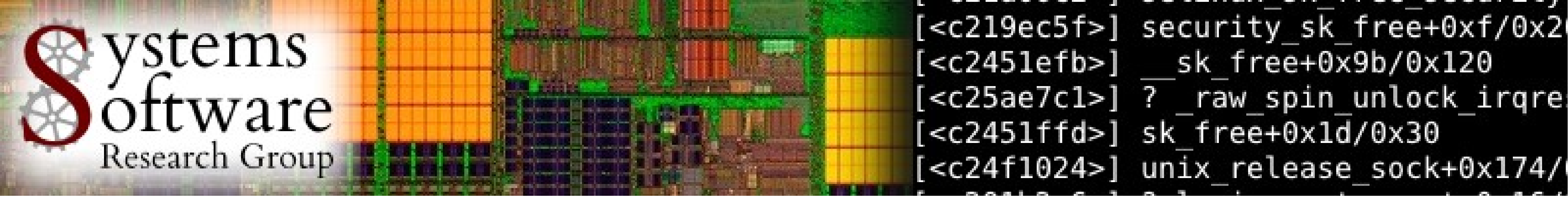


Designing, Modeling, and Optimizing Transactional Data Structures

PhD Dissertation Defense

Ahmed Hassan

Electrical and Computer Engineering Department
Virginia Tech

September 1, 2015

VT MENA EGYPT 2008

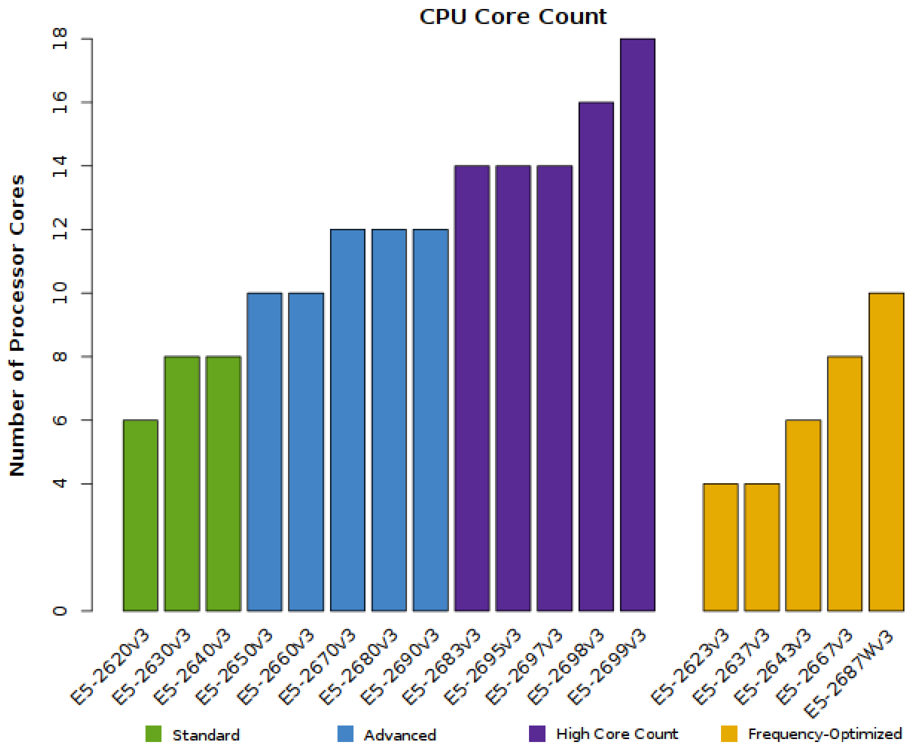
Virginia Tech

State of the Art

- Multi-core architectures.

- Synchronization
  - Critical sections.
  - Using locks.

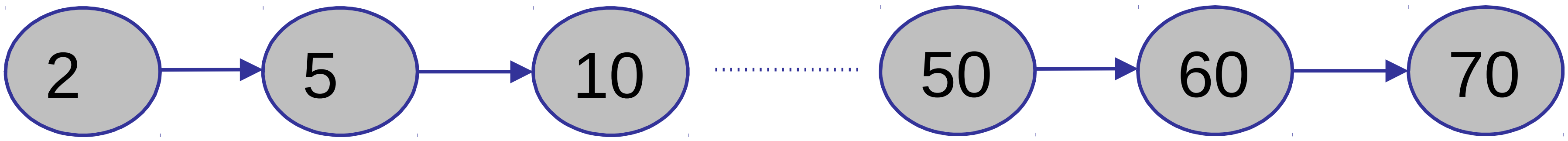
- Efficient synchronization:
  - Cache coherence protocols.
  - Atomic instructions (e.g. CAS operations).



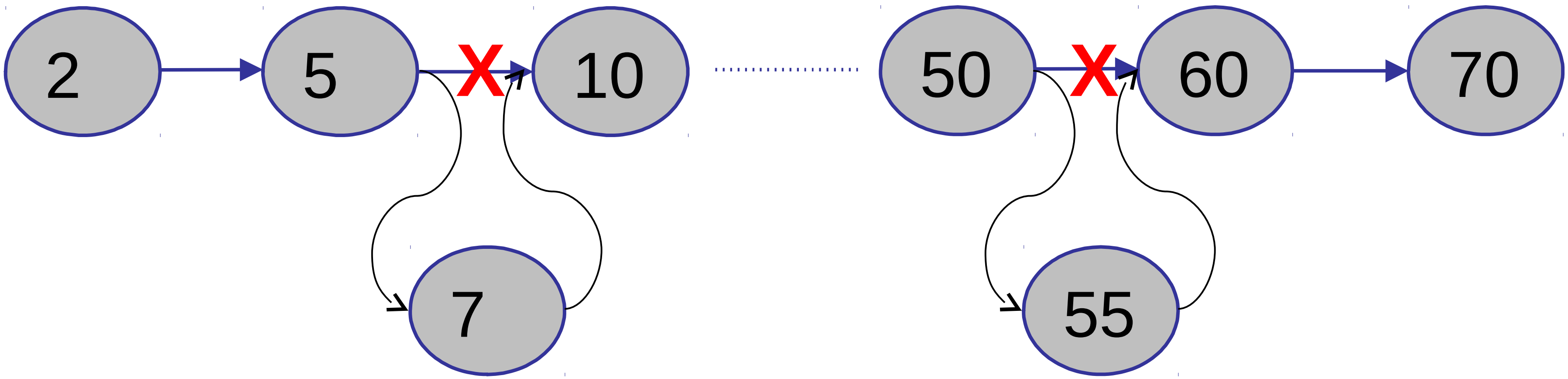
From *www.microway.com*

Concurrent Data Structures

Example: Linked List



Example: Linked List



T_1 T_2

Synchronization in Concurrent Data Structures

- Coarse-grained locking
  - Easy to implement, good for low number of small threads .
  - **But** minimizes concurrency.

- Fine-grained locking
  - Allows more concurrency.
  - **But** error prone.

- Non-Blocking Designs
  - Lock-free, obstruction-free, wait-free, …
  - Progress guarantees **But** more complex designs.

Transactional Memory

- Use an underlying TM framework to guarantee consistency, atomicity, and isolation.

```
Thread 1

@Atomic
foo1()
{
   seqList.add(5)
}
```

- Programmable (like coarse-grained locking).
- Allows concurrency (like fine-grained locking).

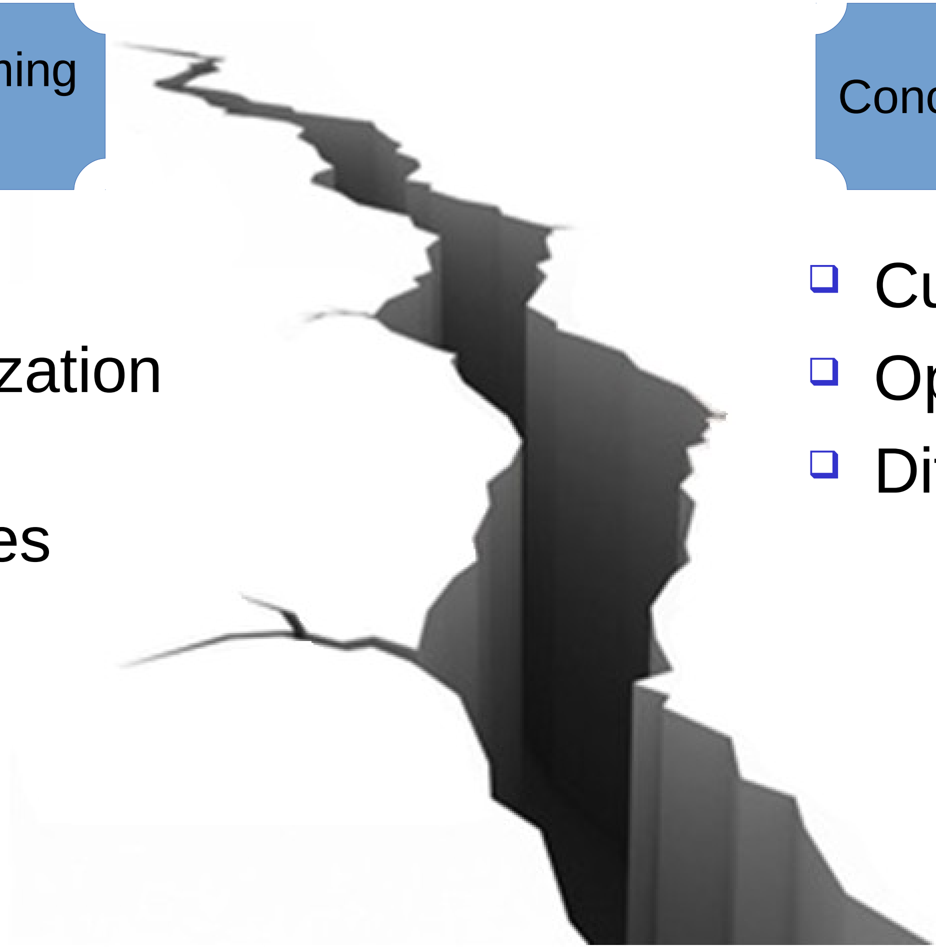
Transactional Memory Gains Traction!!

- Intel Haswell's TSX Extensions.
- IBM Power8.
- STM support in C++ and GCC.

What about data structures?

Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.



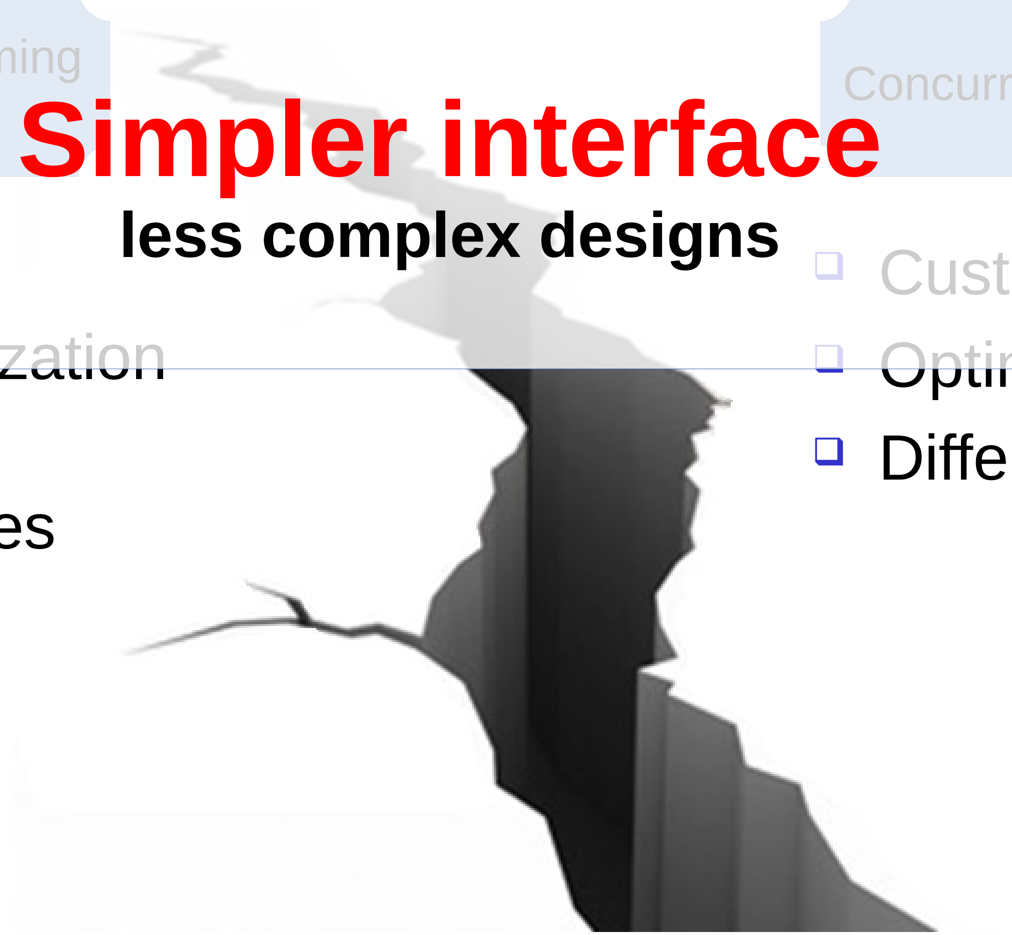
Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives

What about data structures?

Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.



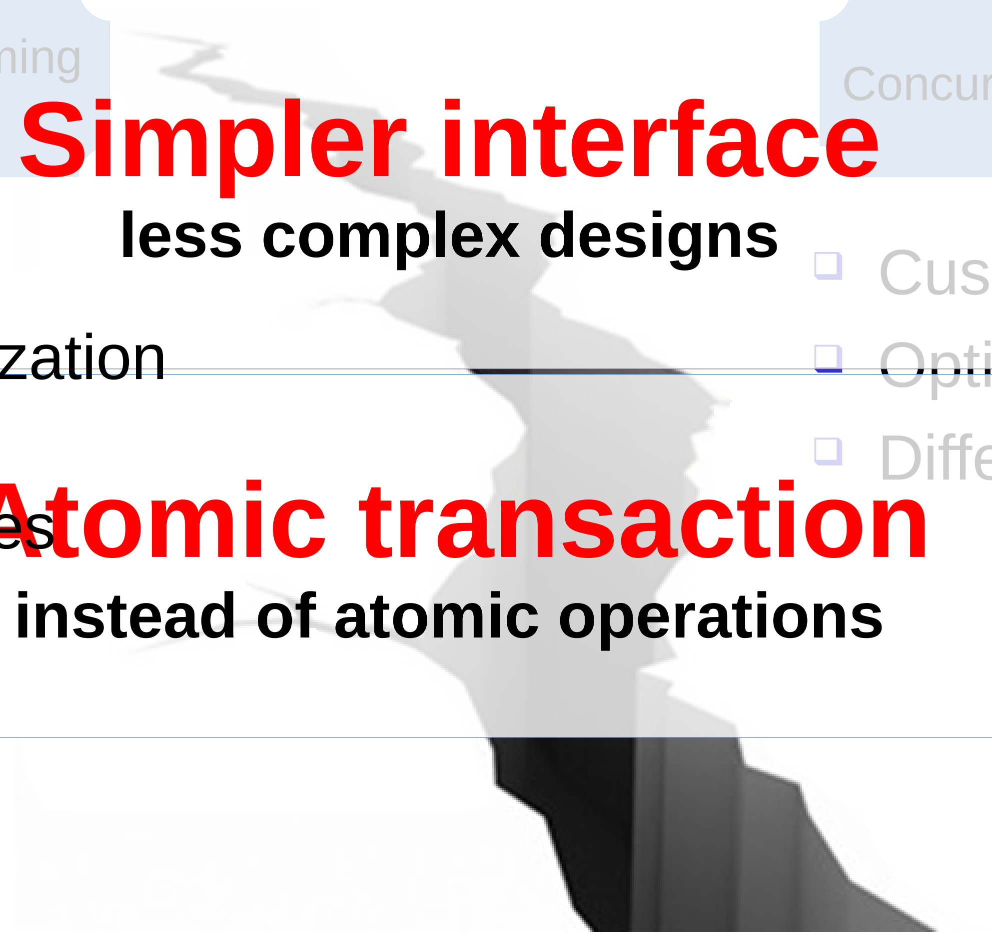
Simpler interface

less complex designs

Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives

What about data structures?



Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.

Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives

Simpler interface

less complex designs

Atomic transaction

instead of atomic operations

What about data structures?

Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.

Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives

Simpler interface

less complex designs

Atomic transaction

instead of atomic operations

Hardware Support

possible performance improvement

Our Goal

From Concurrent to Transactional Data Structures

Challenges

- Composability.
- Integration with generic transactions.
- Modeling.

Composability

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

Composability

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

Modify the design of concurrentList?

More complex designs

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

Composability

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

Modify the design of concurrentList?

More complex designs

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

Transactional Memory?

Lose optimizations of concurrentList

Integration

```
Shared data: concurrentList

atomicFoo()
{
        If(concurrentList.add(x))
                n1++;
        Else
                n2++;
}
```

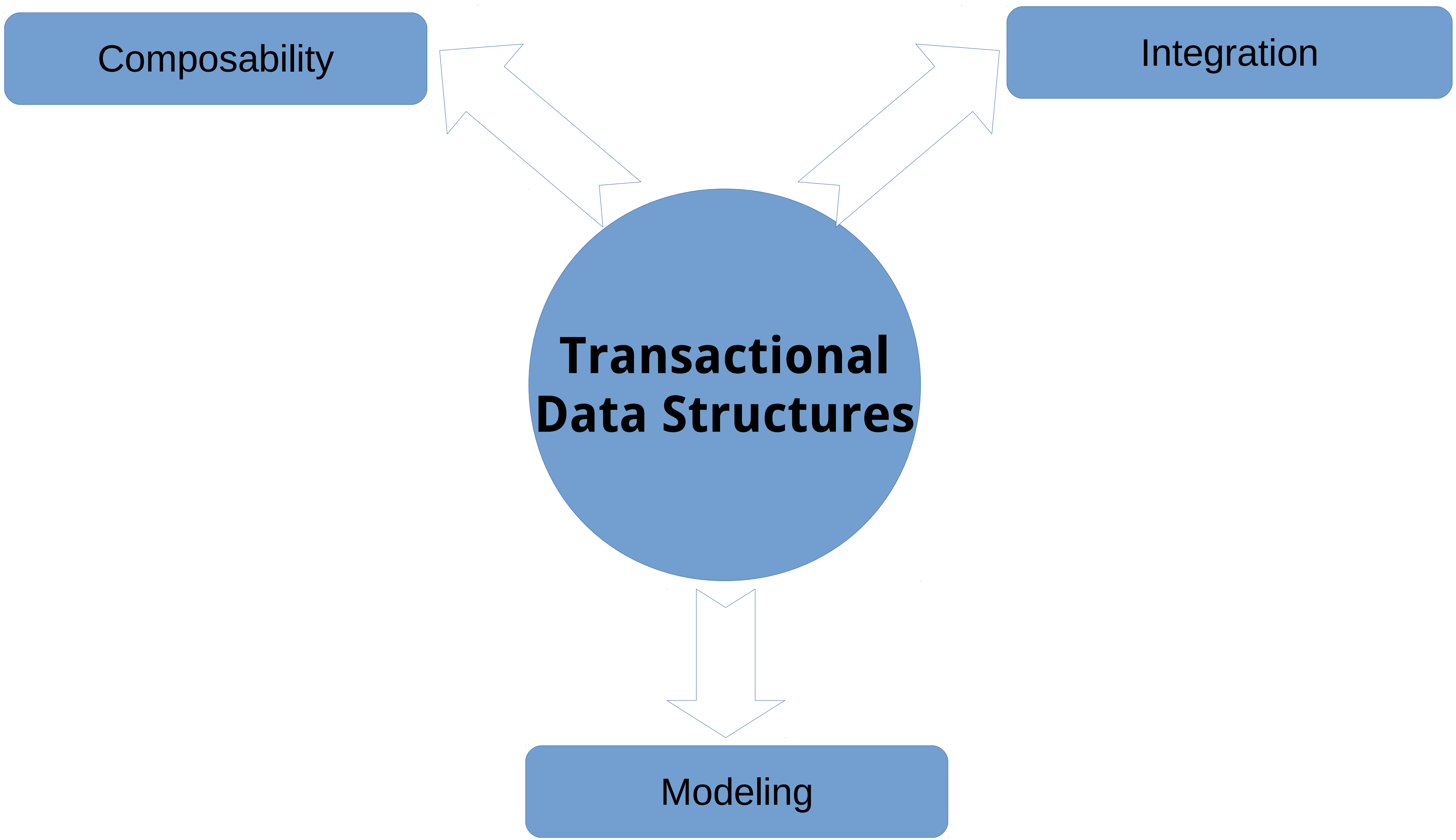
Modeling

- Different Designs and Implementations
  - Different ad-hoc approaches for proving correctness.

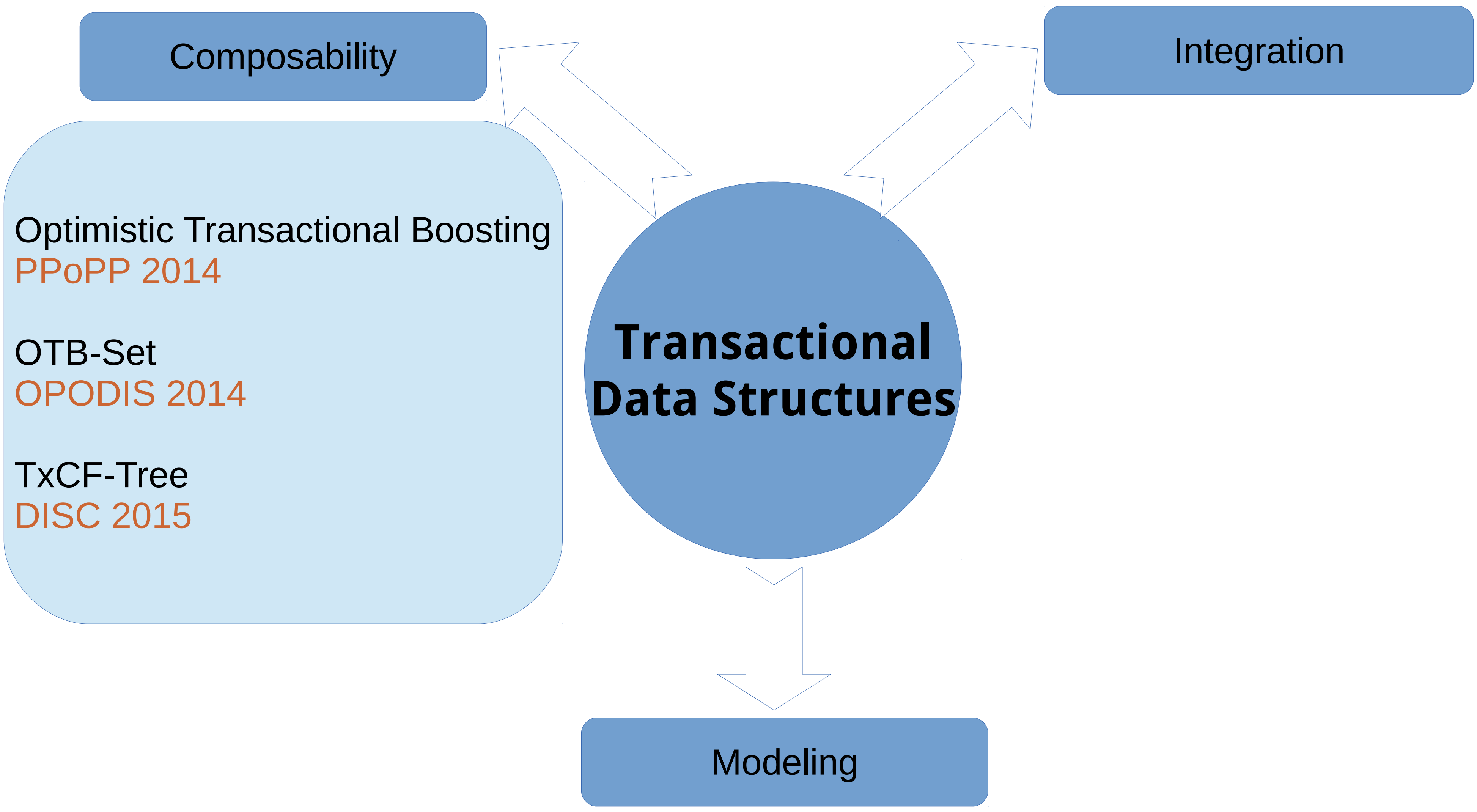
- Is there a unified model for concurrent data structures?
  - General enough
  - Easy to use
  - Includes composability and integration

Our Contributions

Our Contributions



Our Contributions



Our Contributions

Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

Our Contributions

Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

SWC and C-SWC Models
WTTM 2015, under submission

Our Contributions

Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

SWC and C-SWC Models
WTTM 2015, under submission

Past and Related Work

Past and Related Work

- Composability and Integration
  - Transactional Memory.
  - Transactional Boosting.
- Modeling
  - SWMR Model

Past and Related Work

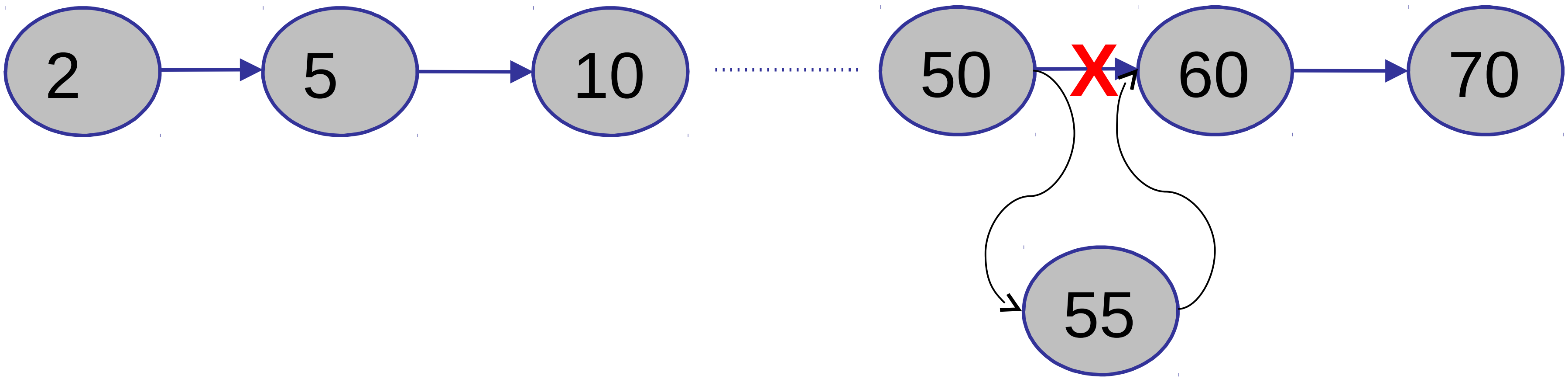
- Composability and Integration
  - Transactional Memory.
  - Transactional Boosting.
- Modeling
  - SWMR Model

Transactional Memory

- Software Transactional Memory (STM)
  - SW meta-data (e.g. read-sets and write-sets) on the current HW.
- Hardware Transactional Memory (HTM)
  - New HW (modify cache coherency protocols).
- Hybrid Transactional Memory (Hybrid TM)
  - HTM transactions fall-back to STM

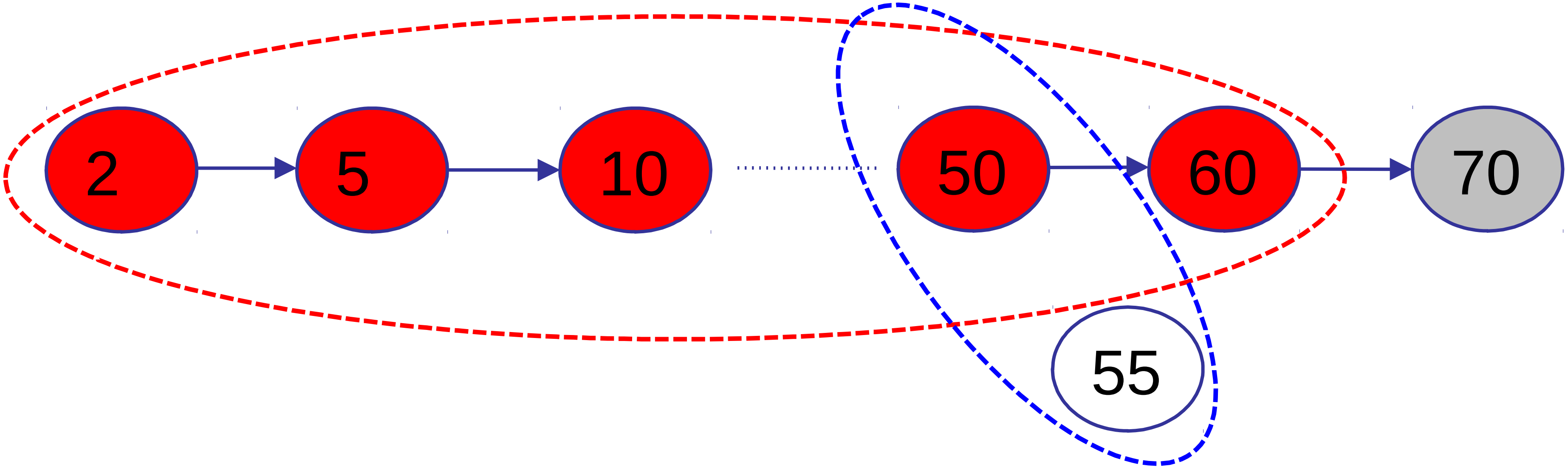
TM Limitation: False Conflict

Example: Linked list (Insert "55")



TM Limitation: False Conflict

Example: Linked list (Insert “55”)

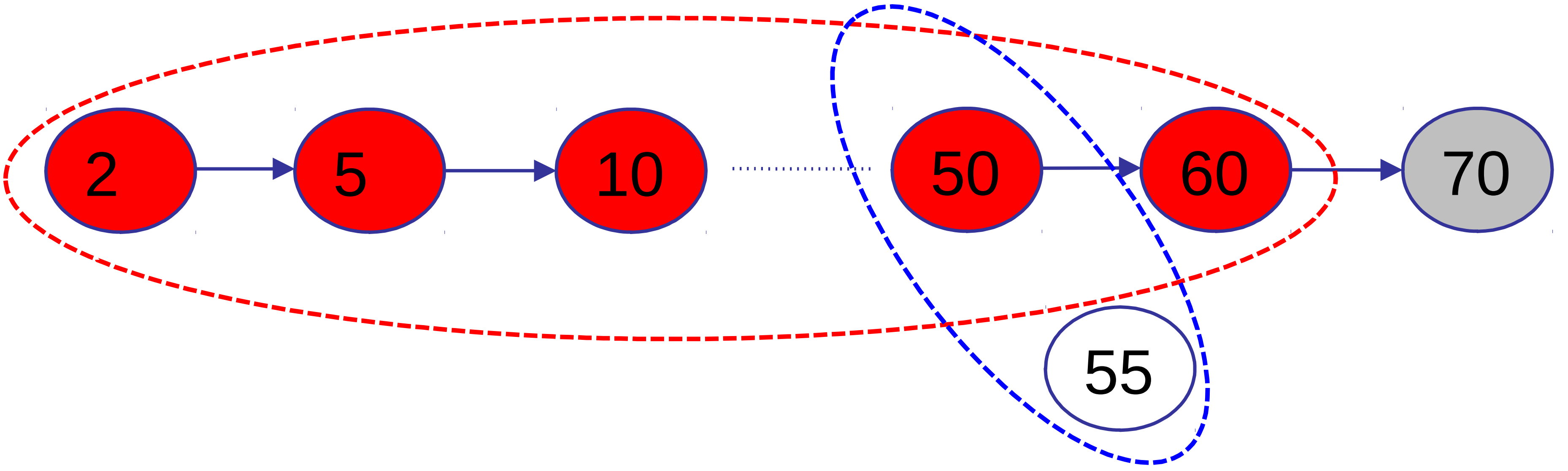


All “red” nodes are in the read-set

“50” and “55” are in the write-set

TM Limitation: False Conflict

Example: Linked list (Insert “55”)



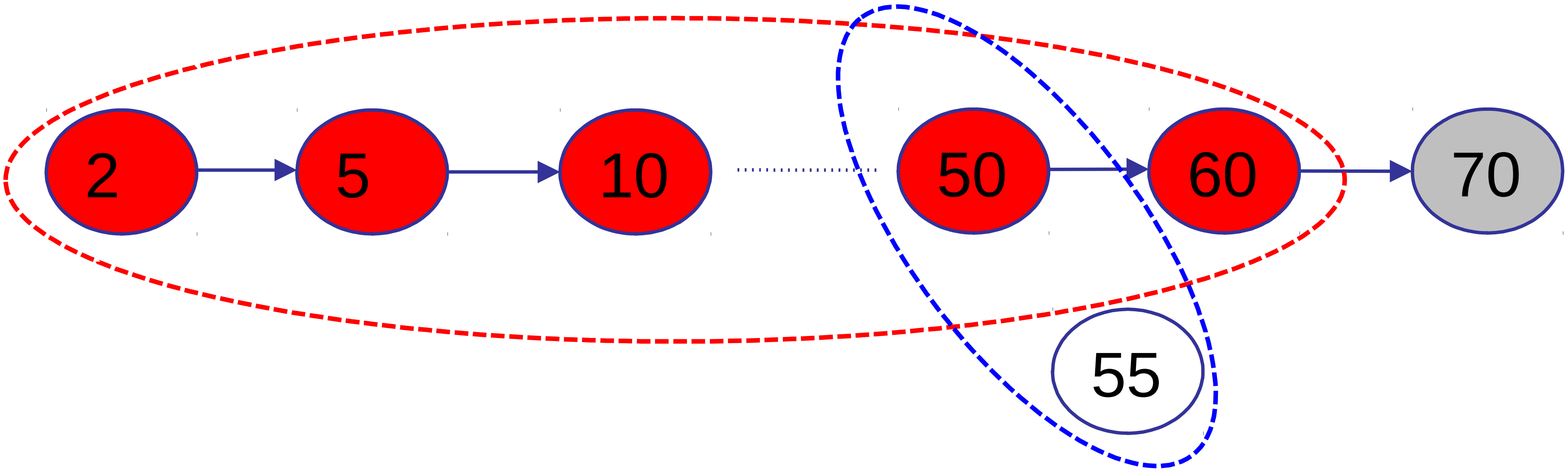
All “red” nodes are in the read-set

“50” and “55” are in the write-set

What if a concurrent transaction deletes “5”??

TM Limitation: False Conflict

Example: Linked list (Insert "55")



All "red" nodes are in the read-set

"50" and "55" are in the write-set

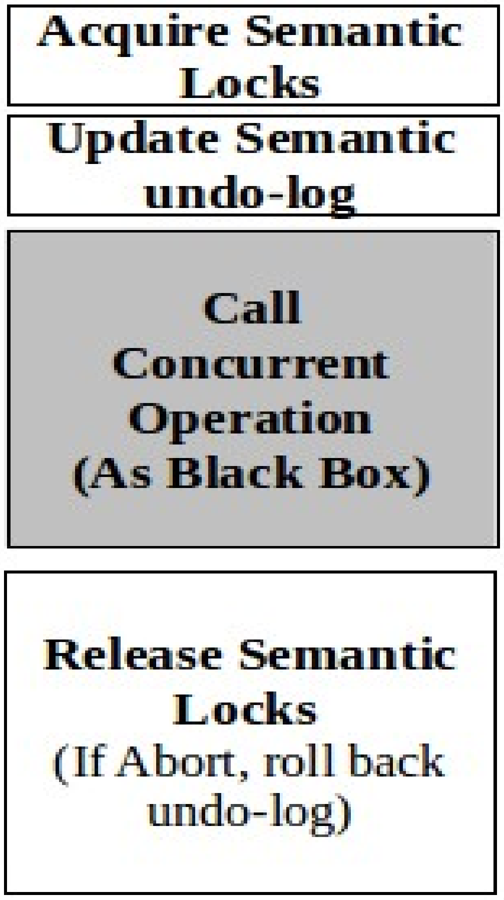
What if a concurrent transaction deletes "5"??

False Conflict

Transactional Boosting

- Convert highly concurrent data structures to be transactional.

- Composable (like STM)
- And efficient (like lazy/lock-free linked-list)

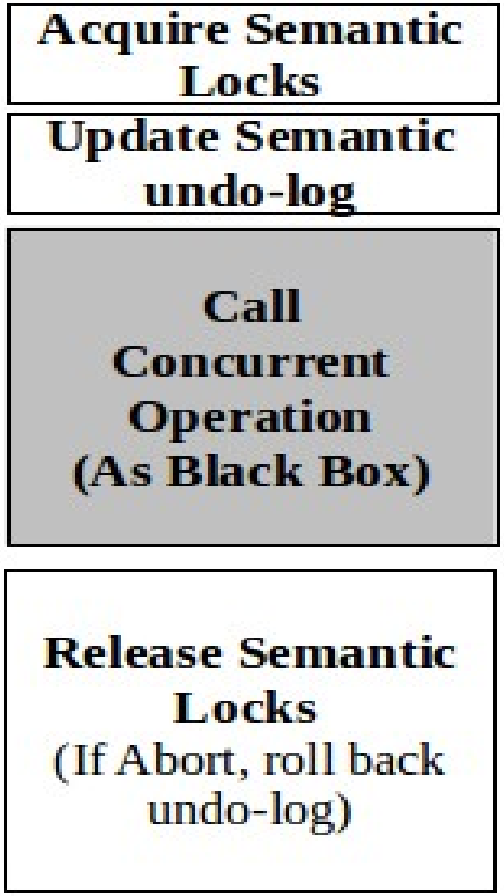


Transactional Boosting

- Convert highly concurrent data structures to be transactional.

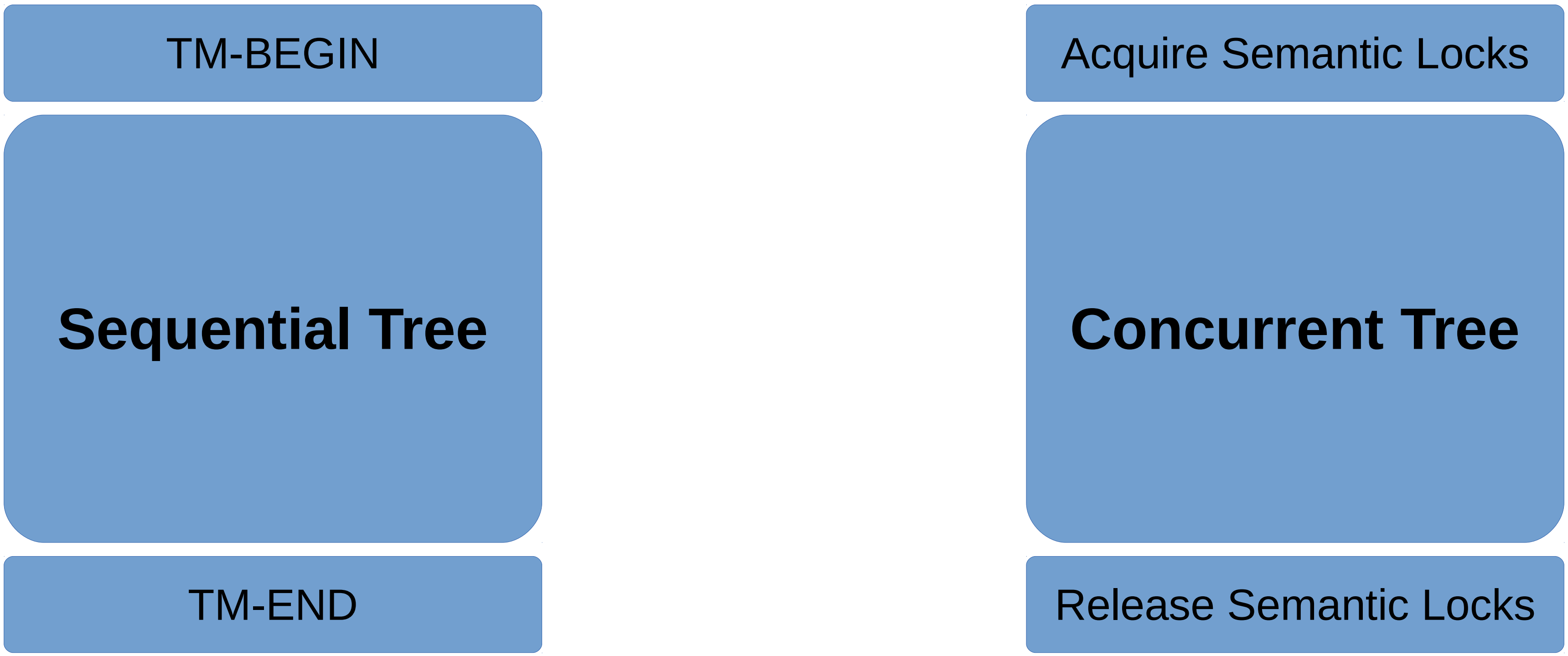
- Composable (like STM)
- And efficient (like lazy/lock-free linked-list)

- Issues:
  - Eager locking.
  - Inverse operations.
  - Black-box concurrent data structure.
  - No Straightforward Integration



Optimistic Transactional Boosting

Past Solutions



Transactional Memory

Transactional Boosting

Past Solutions

TM-BEGIN

Sequential Tree

TM-END

Transactional Memory

Acquire Semantic Locks

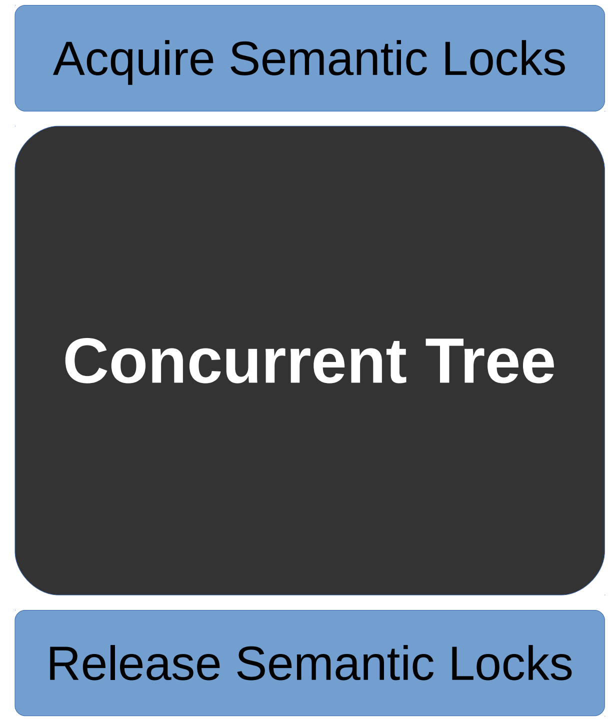
Concurrent Tree

Release Semantic Locks

Transactional Boosting

Past Solutions

Transactional Memory



Transactional Boosting

General, BUT not optimized.

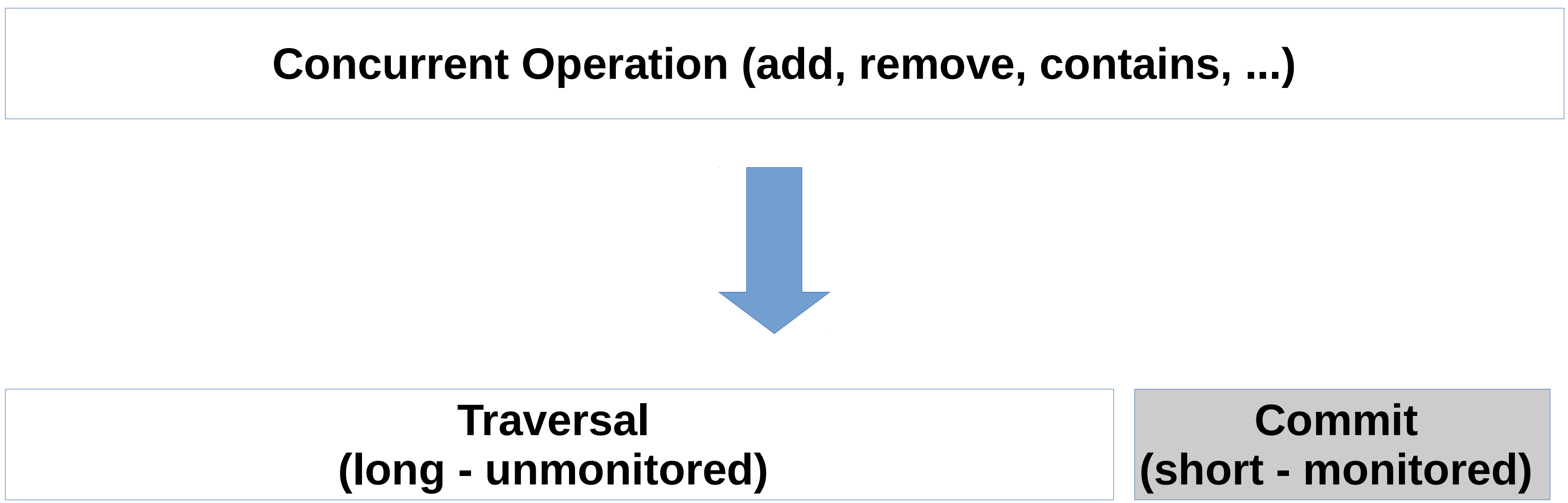
OTB's Three Guidelines

- G1: Split operation

Concurrent Operation (add, remove, contains, ...)

OTB's Three Guidelines

- G1: Split operation

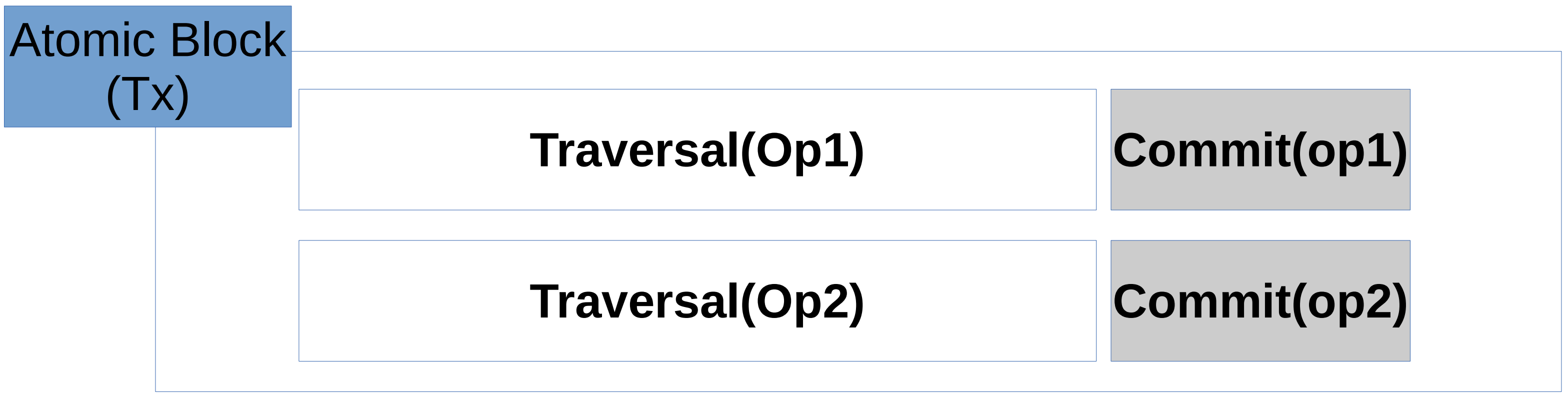


OTB's Three Guidelines

- G2: Compose phases.

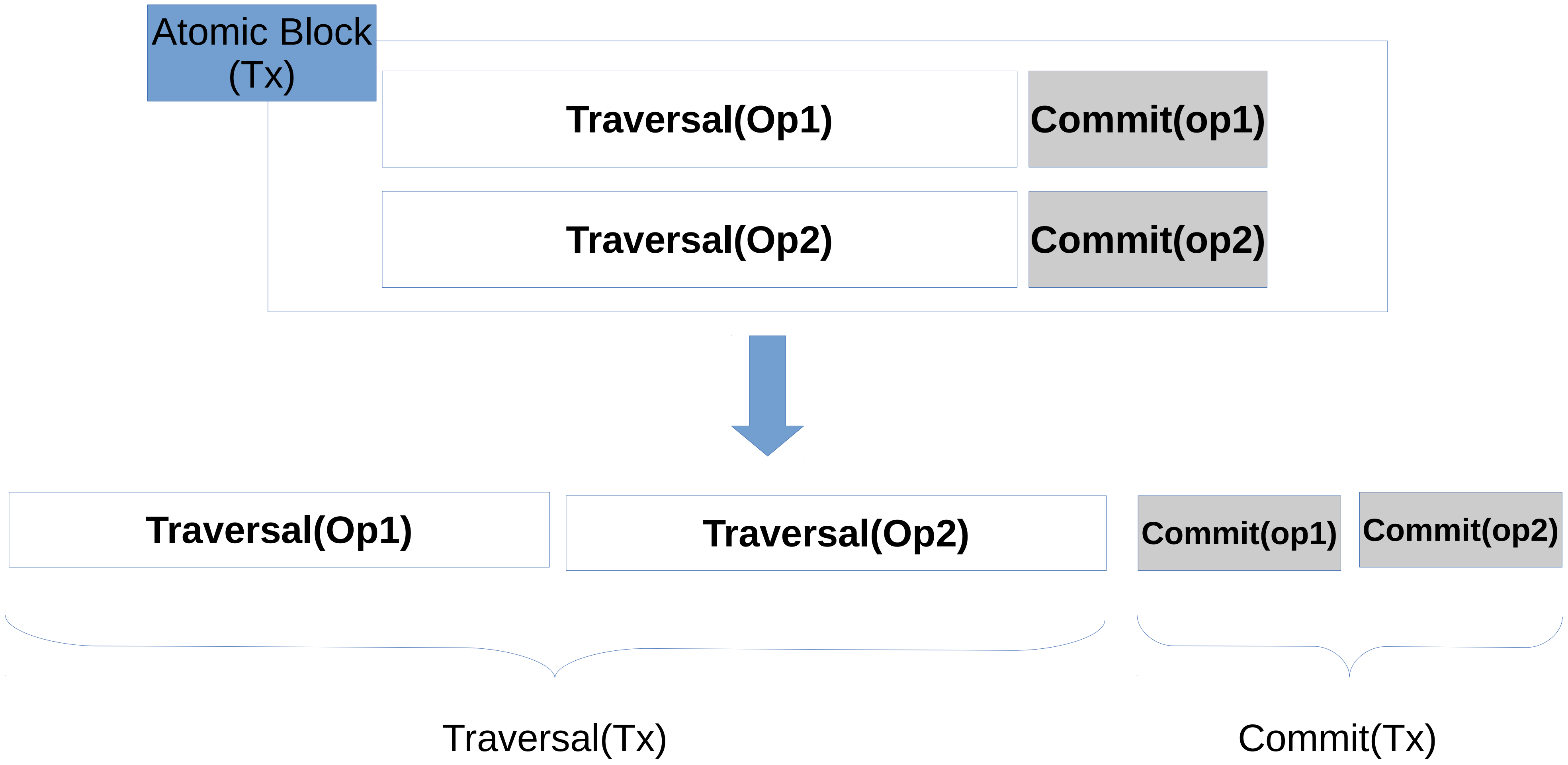
OTB's Three Guidelines

- G2: Compose phases



OTB's Three Guidelines

- G2: Compose phases



OTB's Three Guidelines

- G3: Optimize

OTB's Three Guidelines

- G3: Optimize
  - Specific to each data structure.

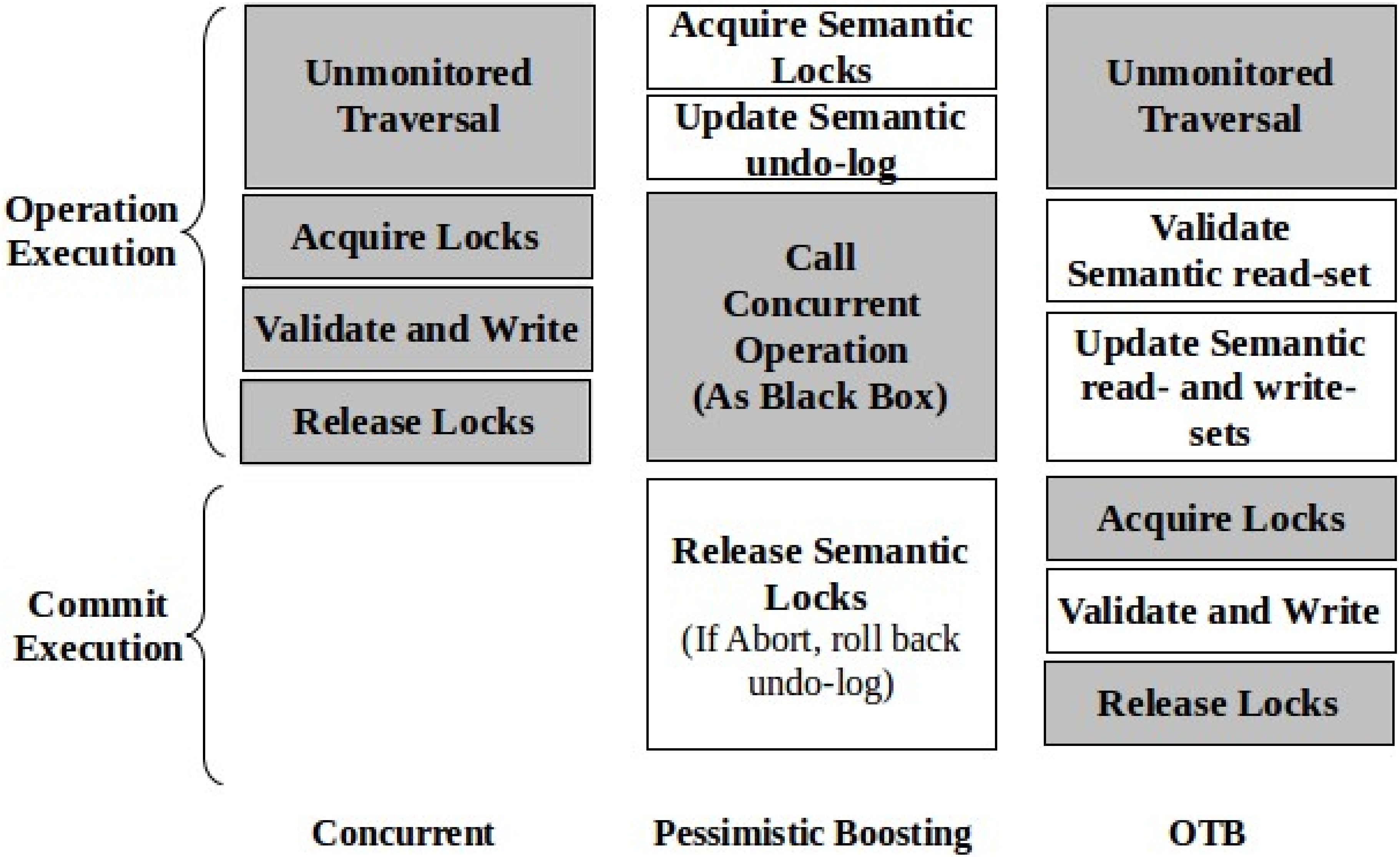
- Our Contribution:
  - Linked-list-based Set.
  - Skip-list-based Set.
  - Skip-list-based Priority Queue.
  - Balanced Tree

OTB's Three Guidelines

- G3: Optimize
  - Specific to each data structure.

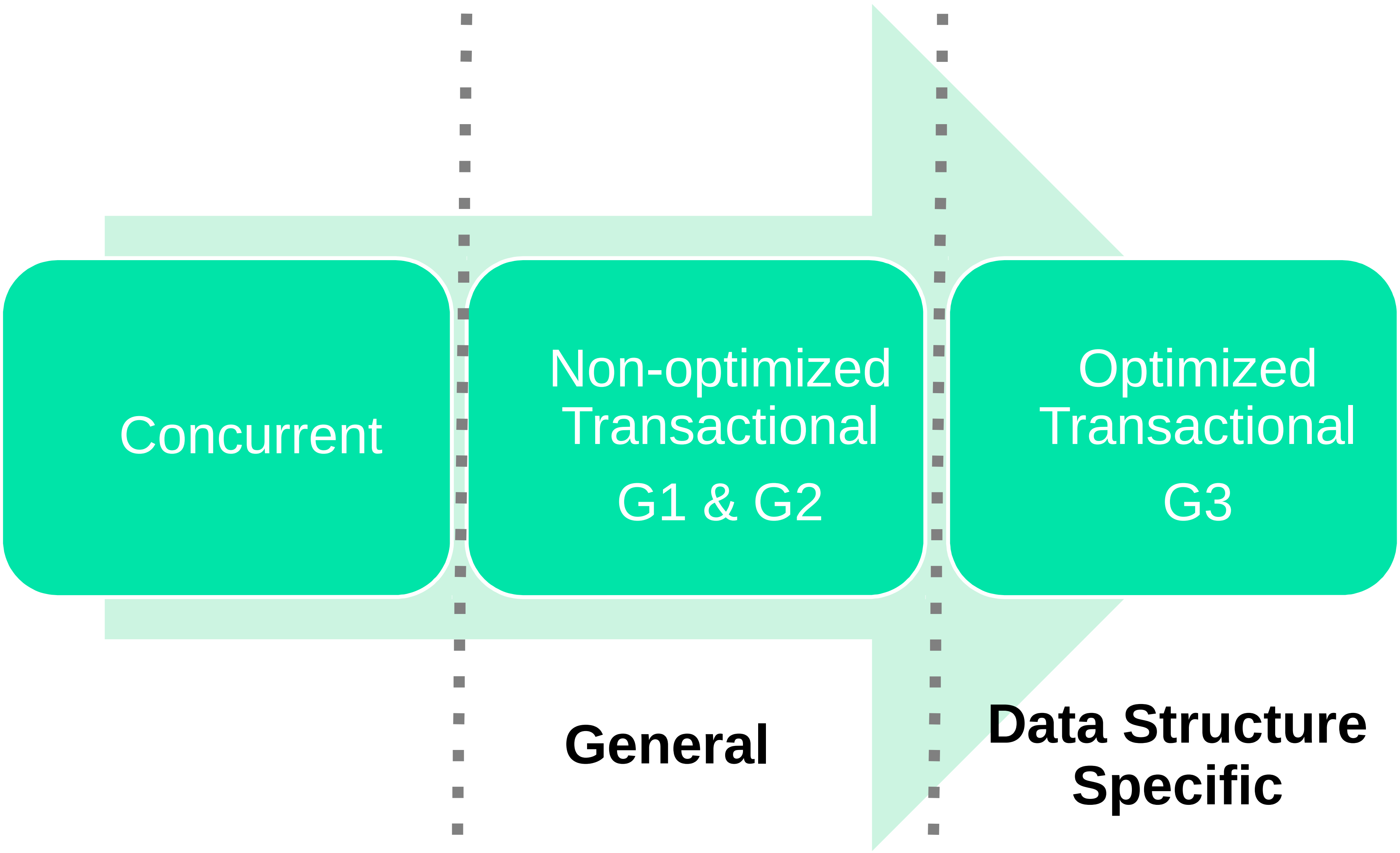
- Our Contribution:
  - Linked-list-based Set.
  - Skip-list-based Set.
  - Skip-list-based Priority Queue.
  - Balanced Tree

Lazy Vs Boosting Vs Optimistic Boosting



Example
Bootsing “lazy” concurrent linked list

OTB Methodology

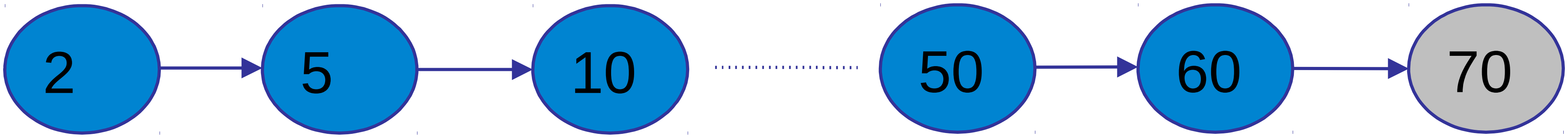


Example

- Lazy Linked list (Insert “55”)

Example

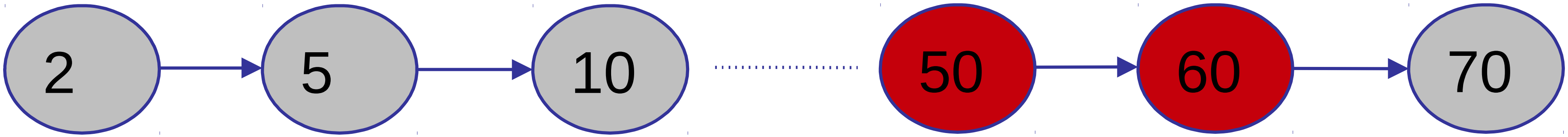
- Lazy Linked list (Insert “55”)



- Traversal (unmonitored)

Example

- Lazy Linked list (Insert “55”)

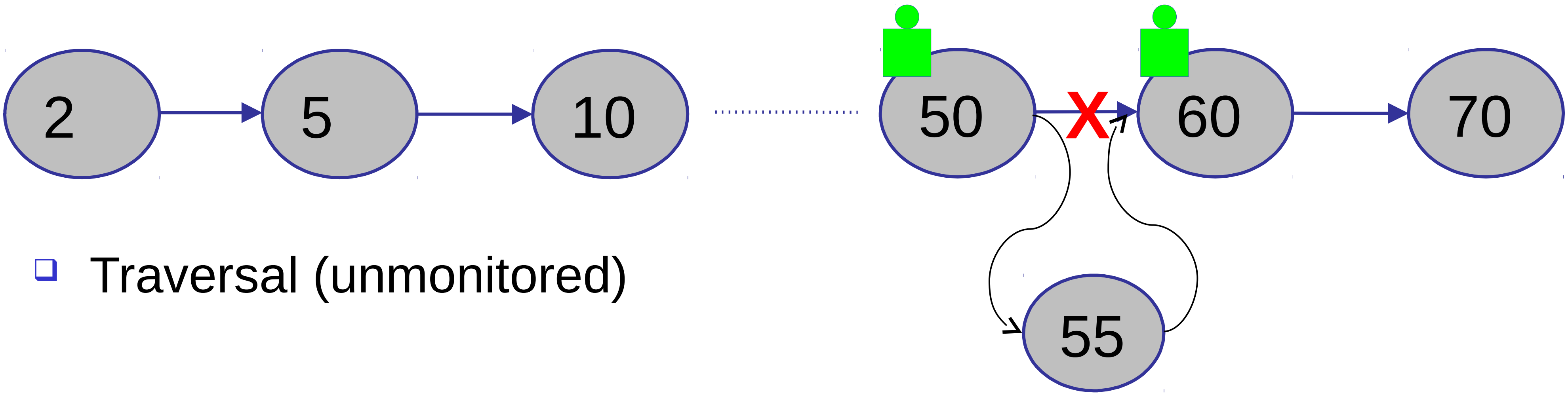


- Traversal (unmonitored)

- Validation

Example

- Lazy Linked list (Insert "55")



- Traversal (unmonitored)
- Validation
- Commit

To Make it Transactional

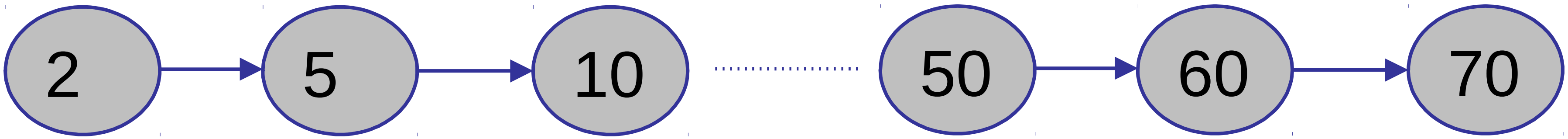
- Results of traversal are saved in local objects:
  - Semantic read-set: to be validated.
  - Semantic write-set: to be published at commit.

To Make it Transactional

- Example: Linked list (Insert “55”)

To Make it Transactional

- Example: Linked list (Insert “55”)



read-set entry

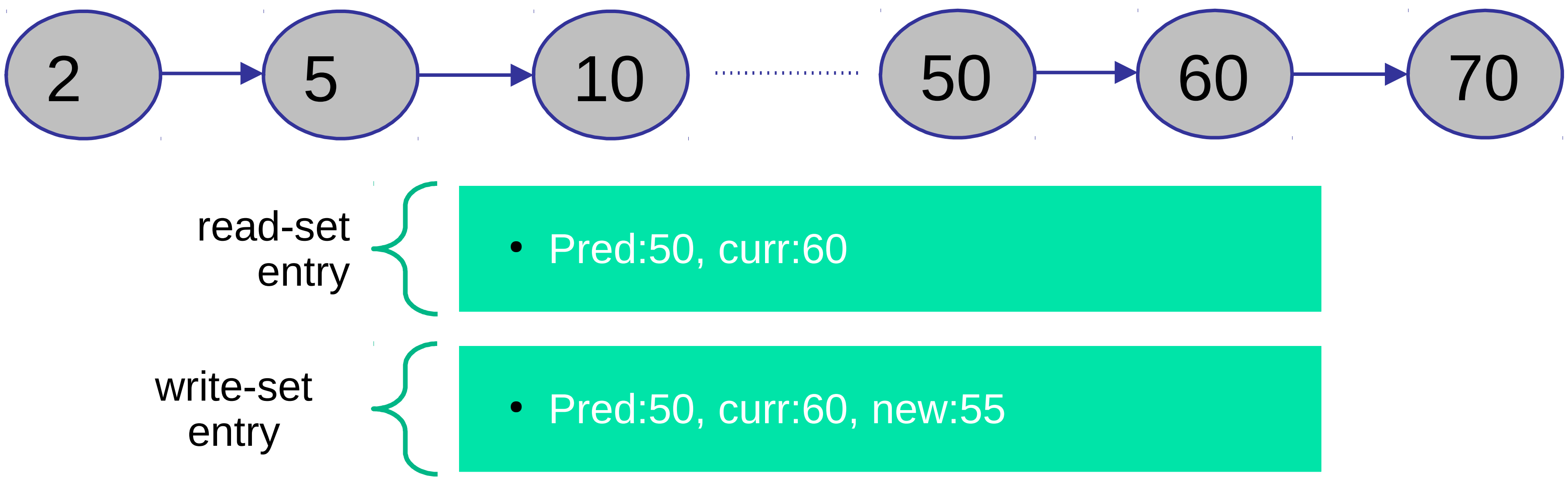
- Pred:50, curr:60

write-set entry

- Pred:50, curr:60, new:55

To Make it Transactional

- Example: Linked list (Insert “55”)

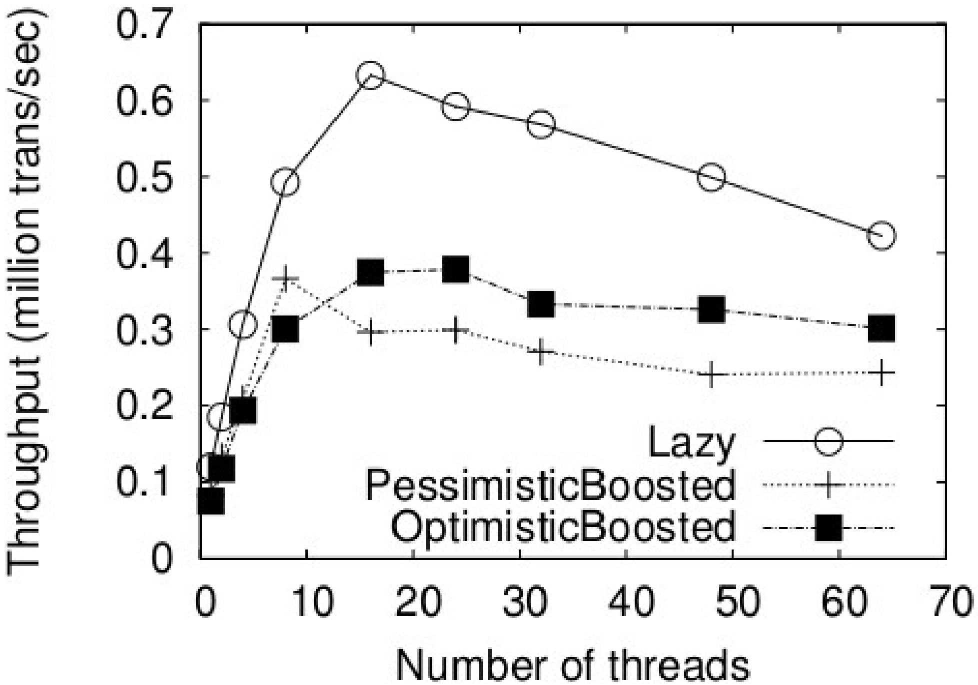


- Guidelines to guarantee opacity (see OPODIS'14 paper)

Specific Optimizations

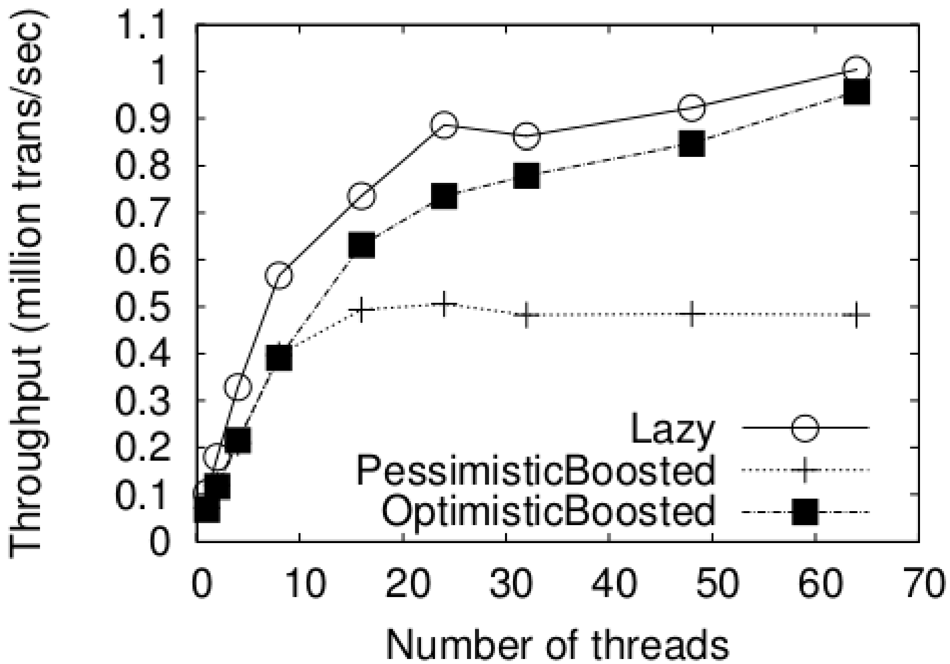
- Example optimizations on Linked-List and Skip-List
  - Local elimination:
    - Ex. Add(x) then Remove(x).
    - No need to access the shared data structure.

Results



Skip-list 512 Nodes

5 ops/transaction



Skip-list 64K Nodes

5 ops/transaction

Transactional Interference-less Balanced Tree

Transactional Interference-less Balanced Trees

- Transactional: Functionality (following OTB's G1, G2).
- Interference-less: Performance (following OTB's G3).

The Next Question

- Which concurrent balanced tree design fits OTB?

The Next Question

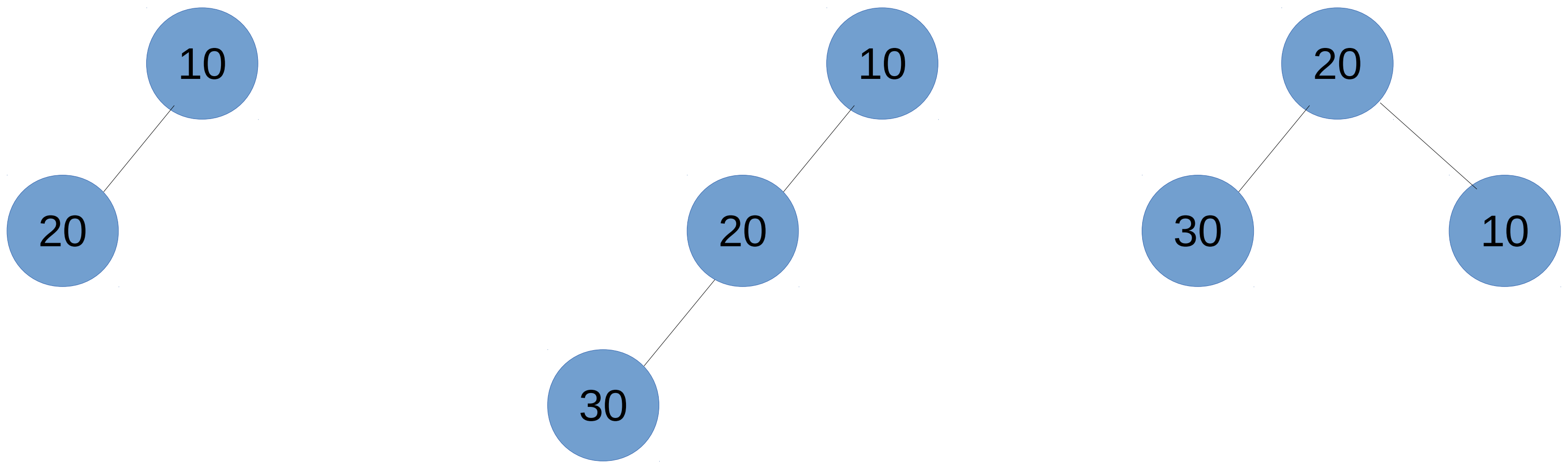
- Which concurrent balanced tree design fits OTB?

Contention-Friendly Tree

Crain, Gramoli, & Raynal'13

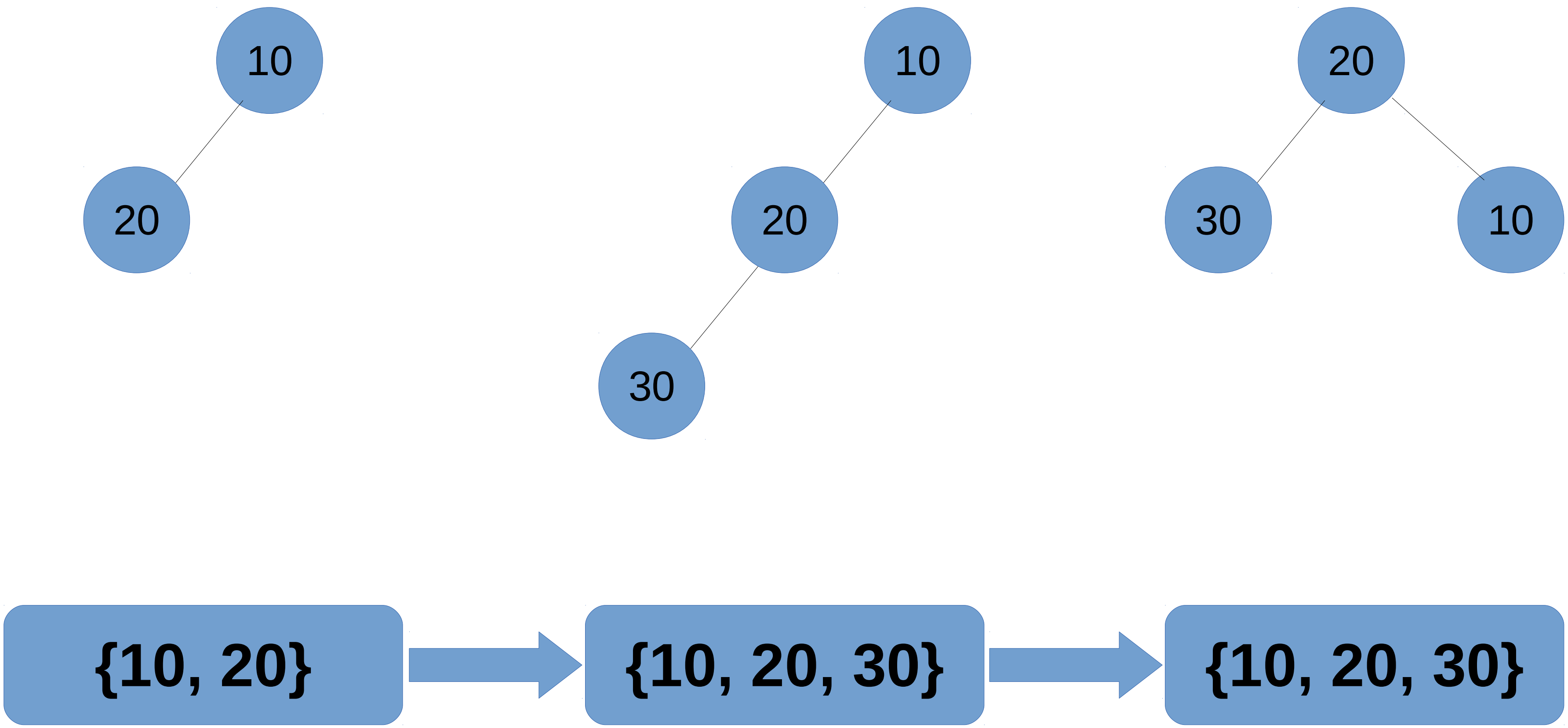
CF-Tree

- Example: Insert 30.



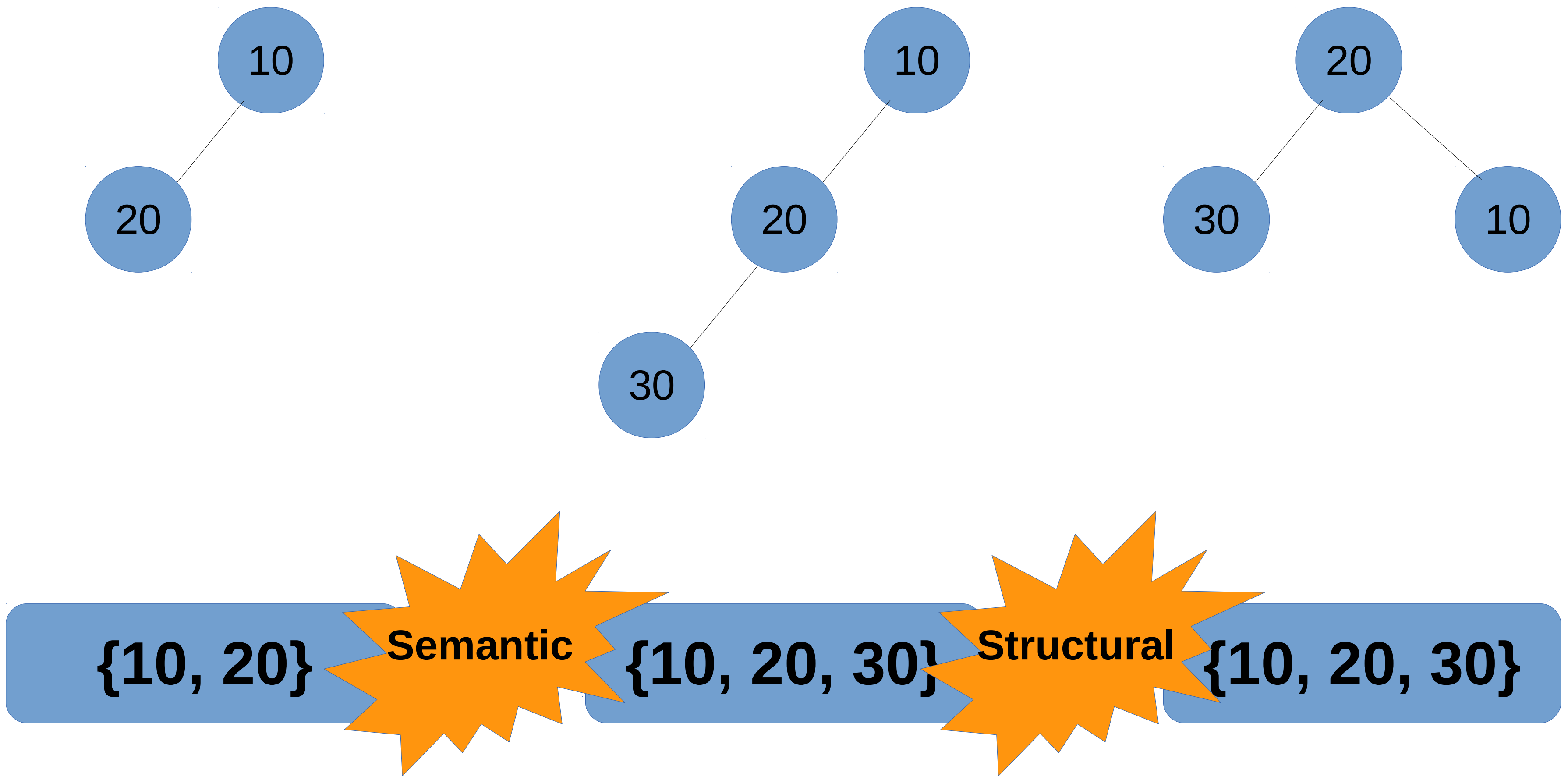
CF-Tree

- Example: Insert 30.



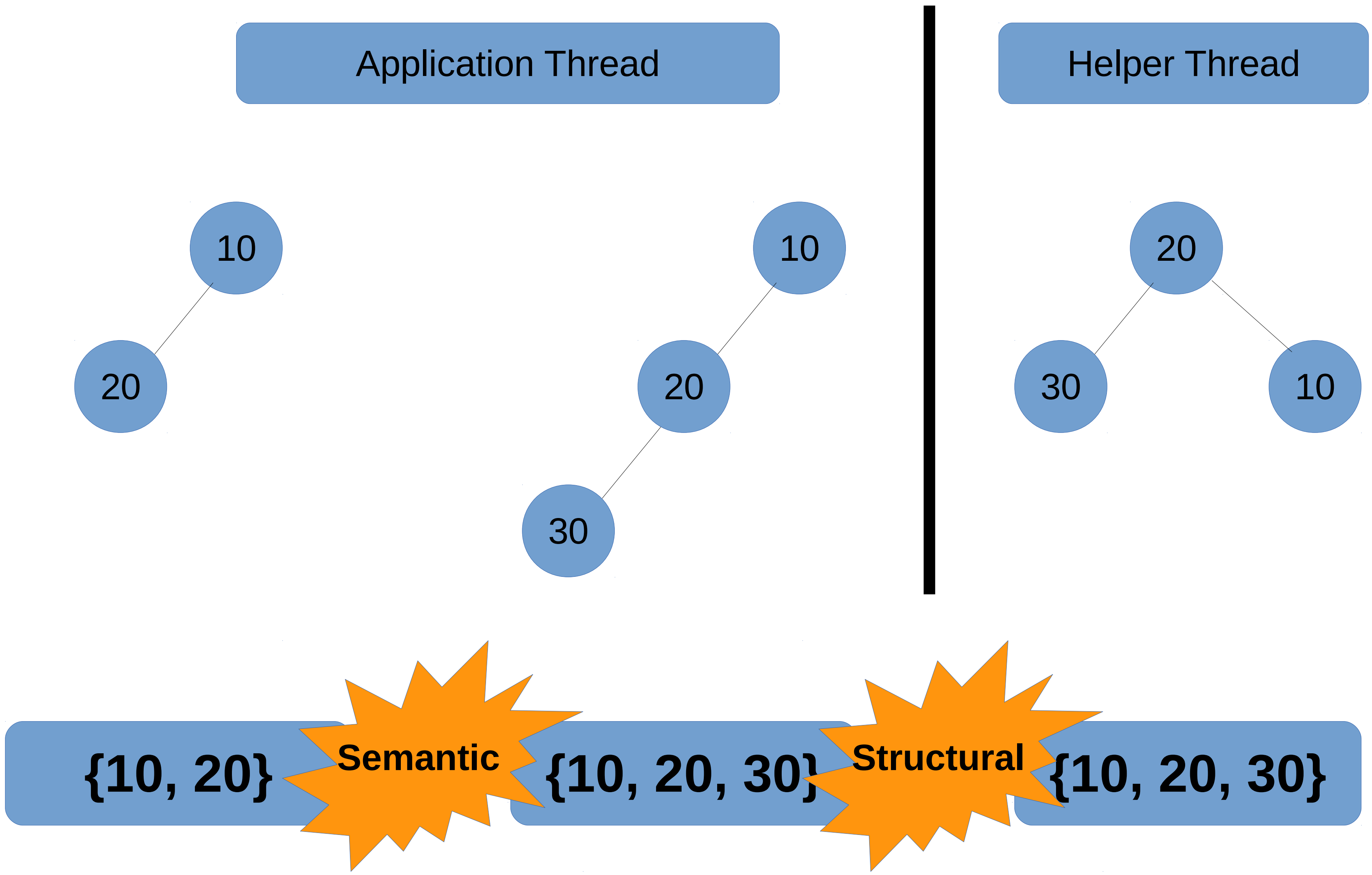
CF-Tree

- Example: Insert 30.



CF-Tree

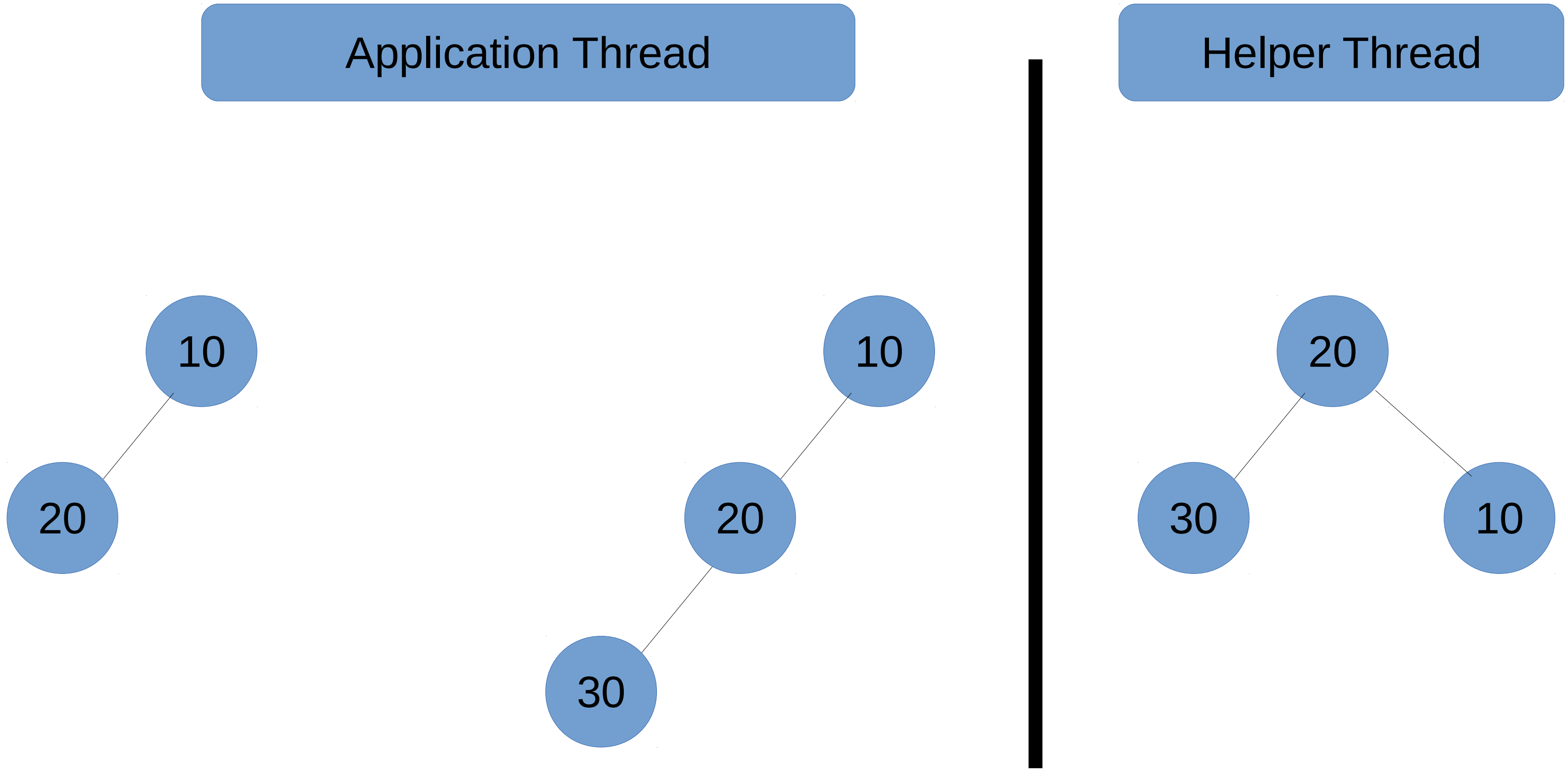
- Example: Insert 30.



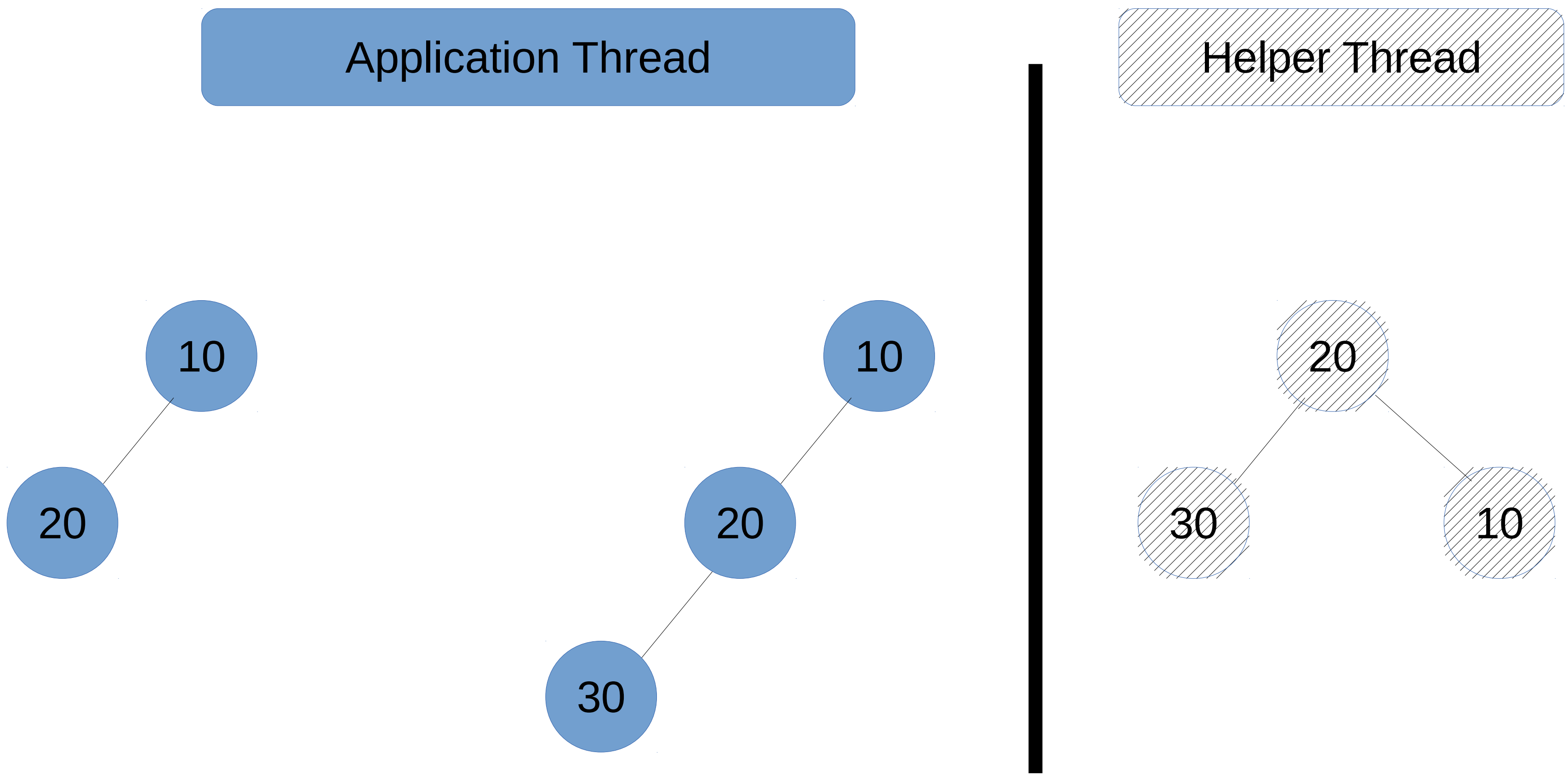
Our Proposal

Transactionalizing CF-Tree using OTB
(TxCF-Tree)

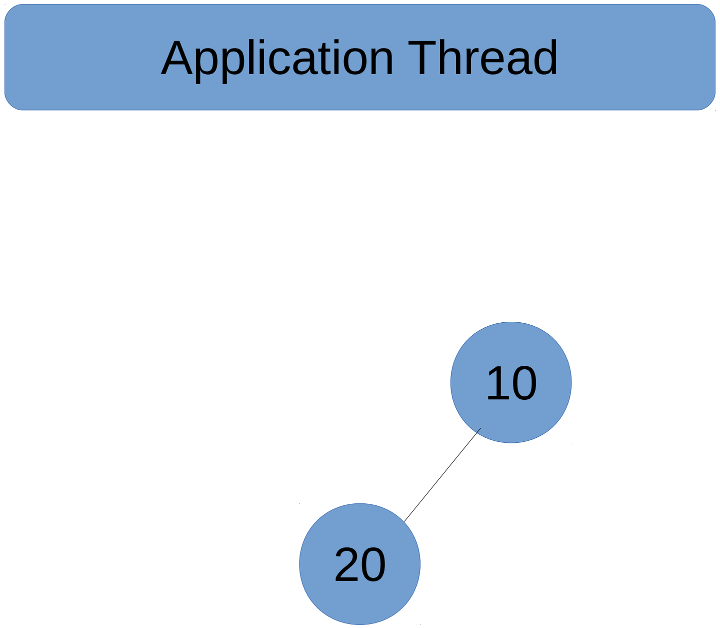
TxCF-Tree



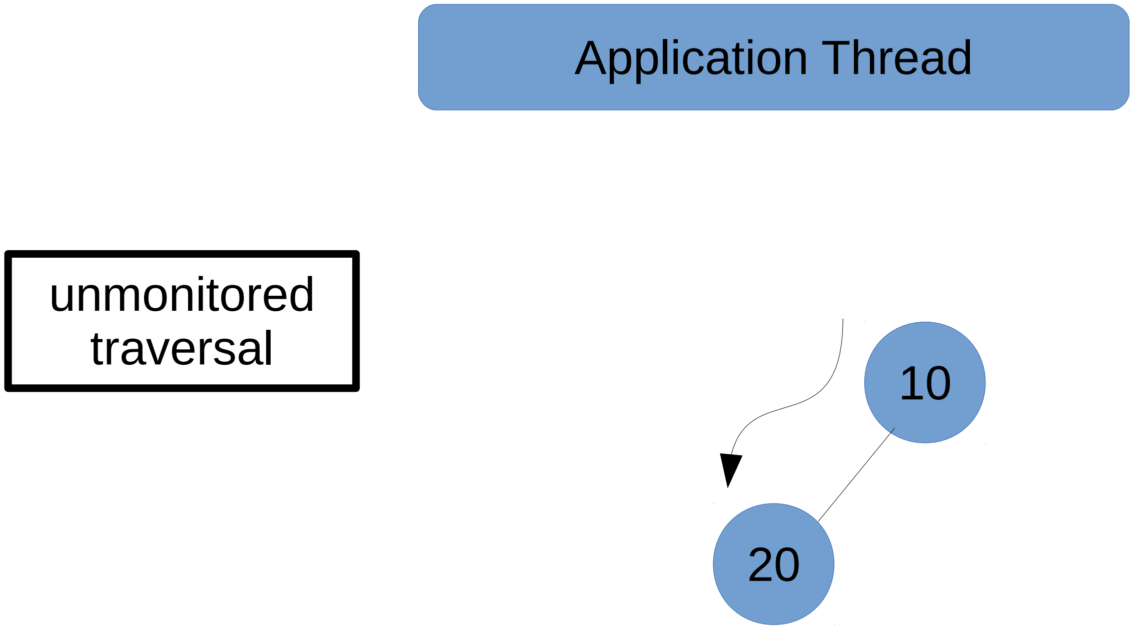
TxCF-Tree



TxCF-Tree



TxCF-Tree

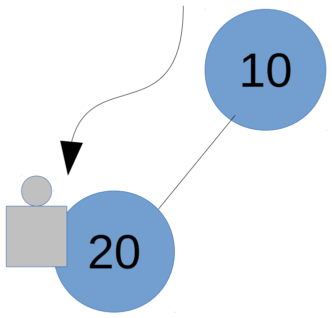


TxCF-Tree

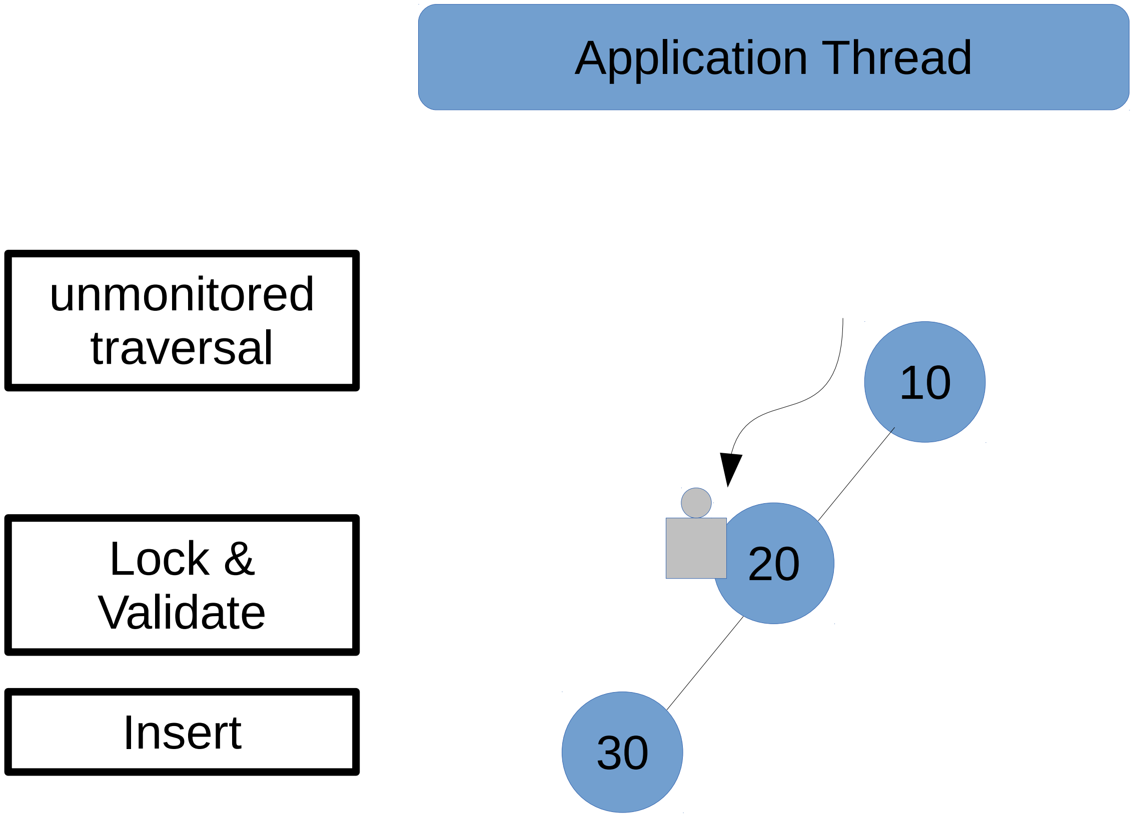
Application Thread

unmonitored traversal

Lock & Validate

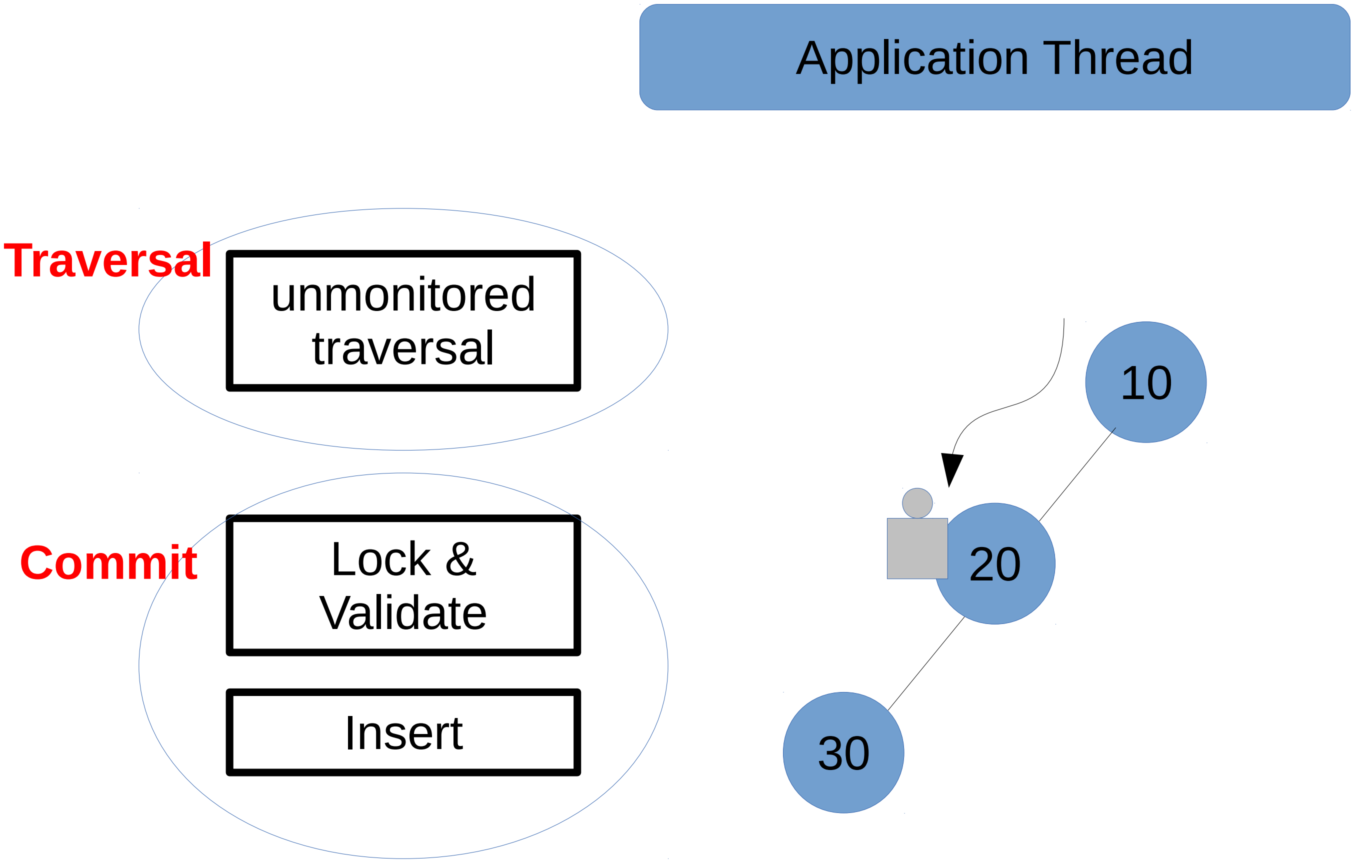


TxCF-Tree

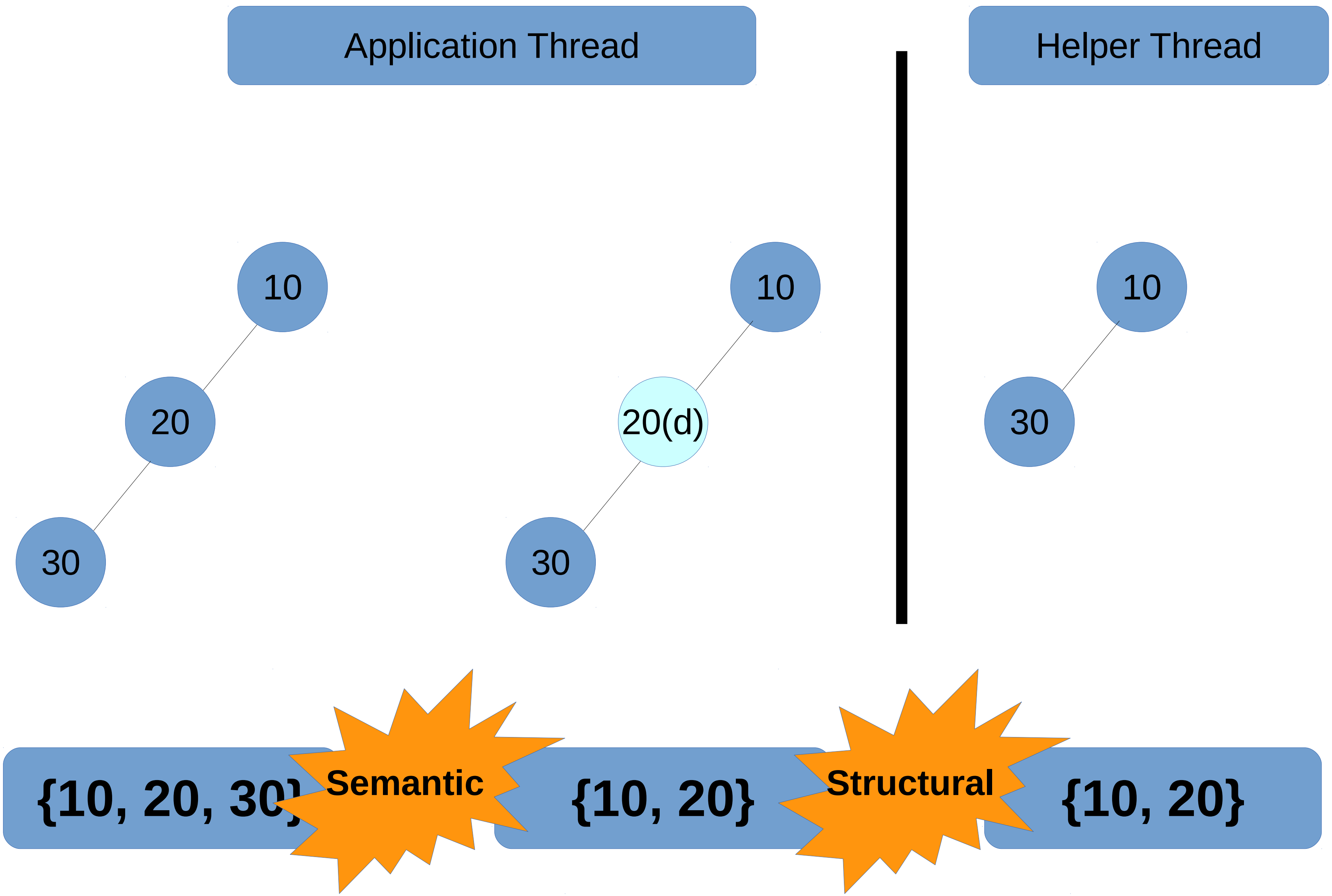


Application Thread
unmonitored traversal
Lock & Validate
Insert
10
20
30

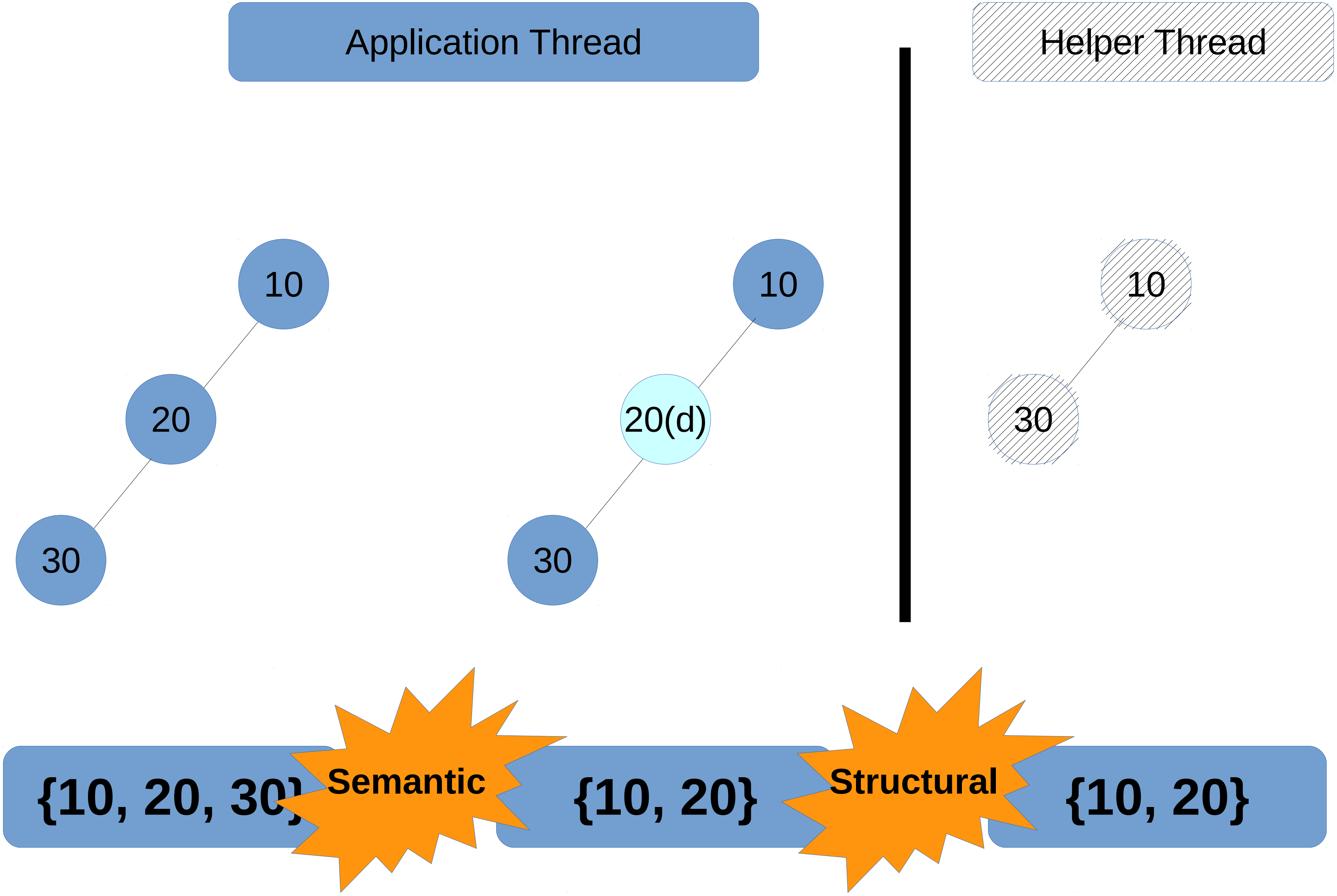
TxCF-Tree


Application Thread
Traversal
unmonitored traversal
Commit
Lock & Validate
Insert
10
20
30

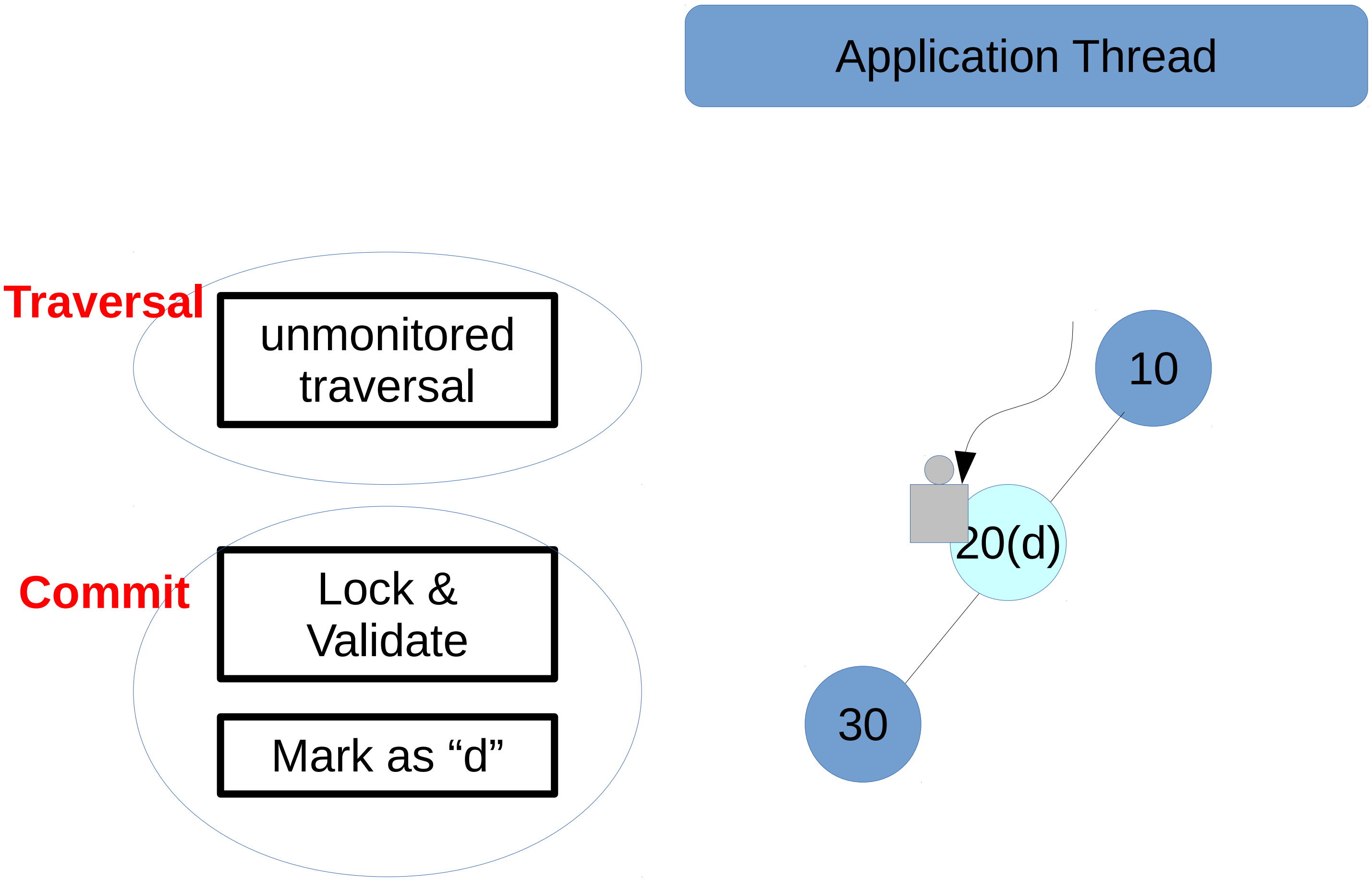
Remove is similar...



Remove is similar...



Remove is similar...



Transactional Interference-less Tree

Transactional Interference-less Tree

- How
  - Step 1: CF-Tree!!
  - Step 2: Always give the highest priority to semantic operations over structural operations.

Transactional Interference-less Tree

- How
  - Step 1: CF-Tree!!
  - Step 2: Always give the highest priority to semantic operations over structural operations.

- Why
  - Aborting transactions rolls back all its operations (including the non-conflicting ones).
  - Long transactions are more prone to interfere with the helper thread.

Two building blocks

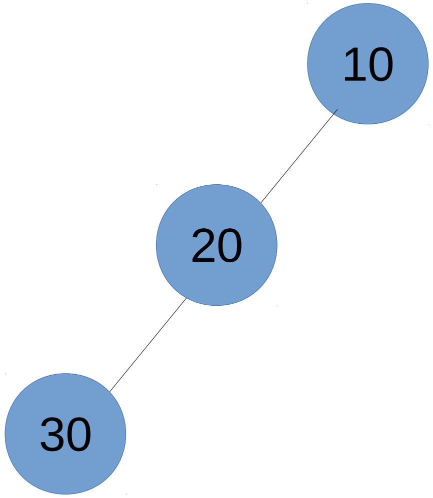
- Structural Locks.
- Structural Invalidation.

Structural Locks

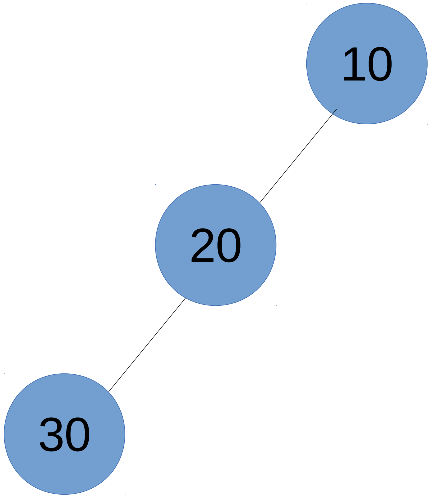
Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



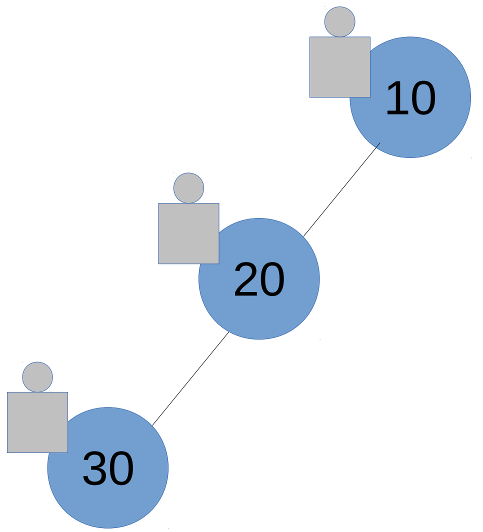
A concurrent delete(30)



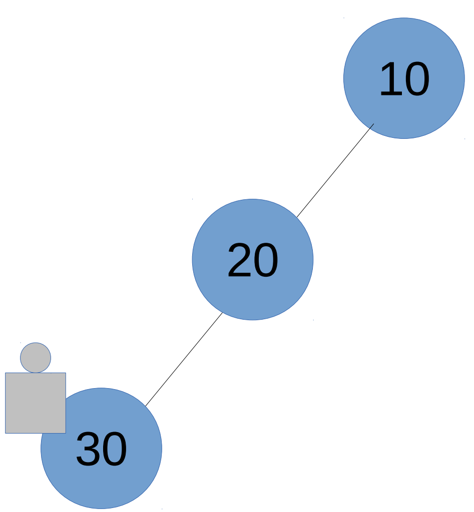
Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



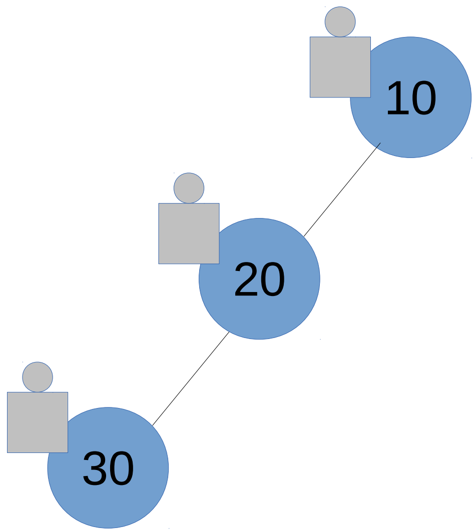
A concurrent delete(30)



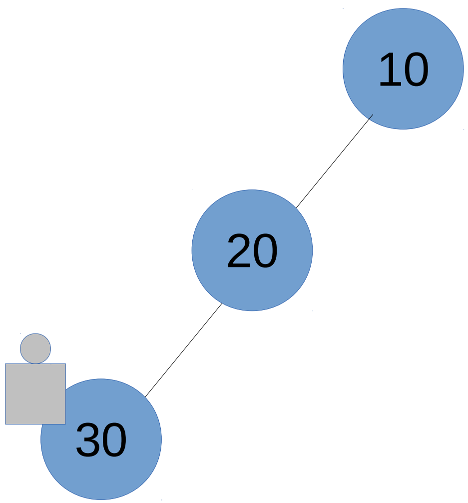
Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent delete(30)



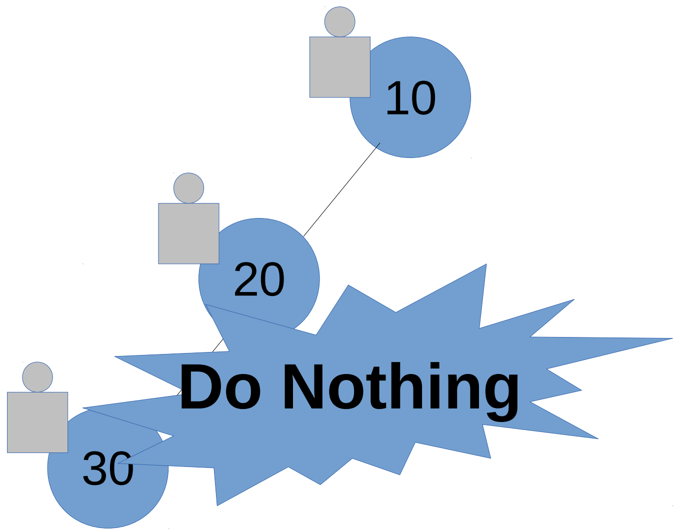
T1 observes that "30" is locked

What is the best to do in both cases?

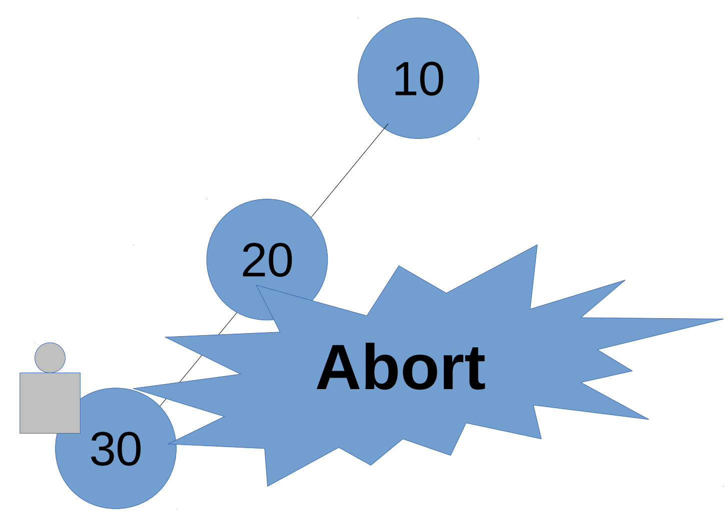
Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent delete(30)



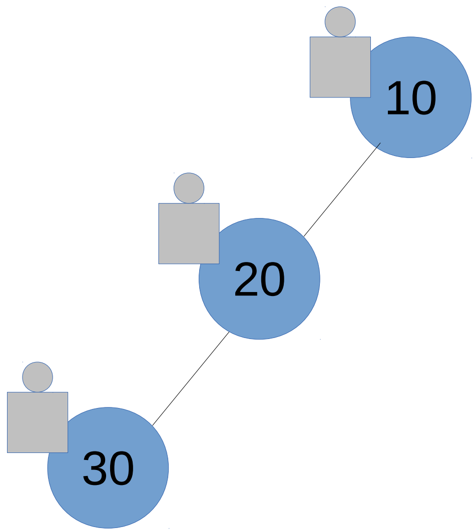
T1 observes that "30" is locked

What is the best to do in both cases?

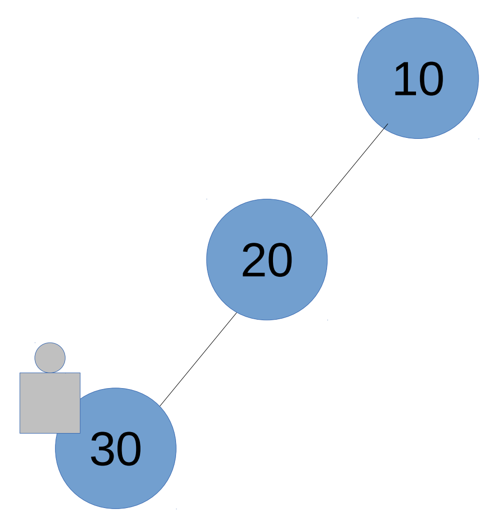
Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent delete(30)

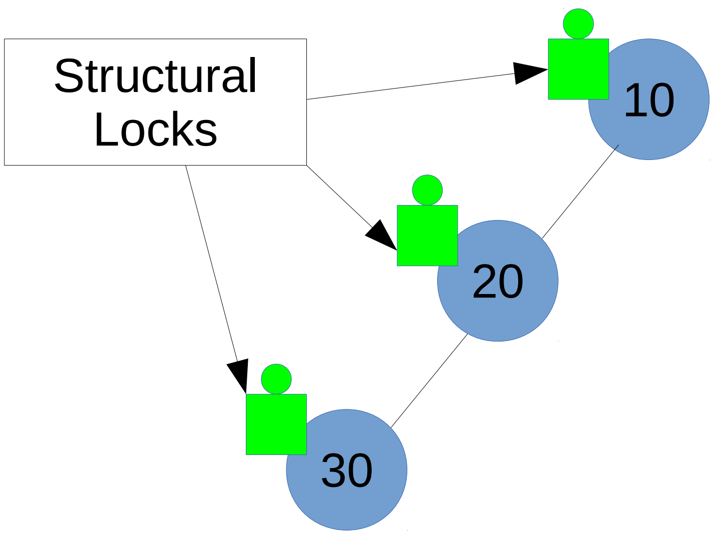


Solution?

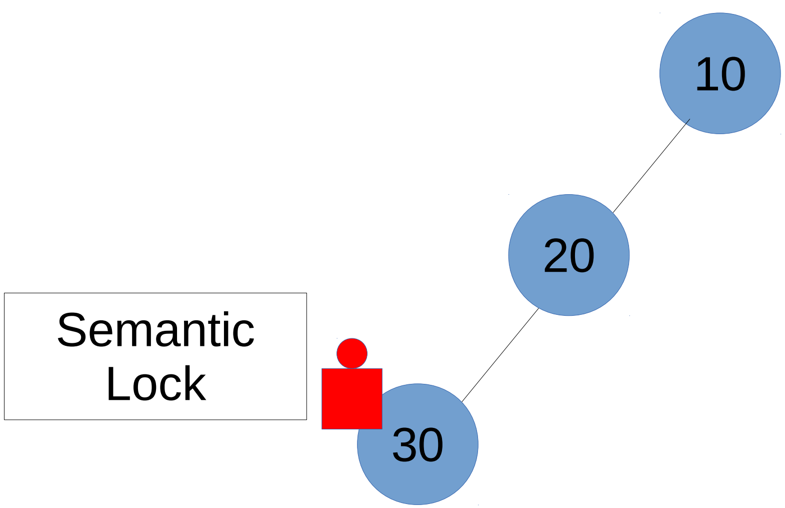
Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent delete(30)



Solution?

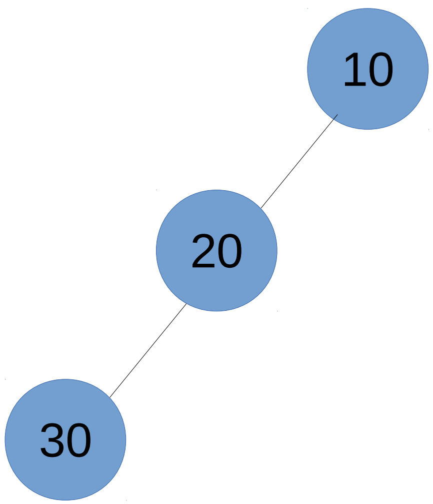
Two types of locks

Structural Invalidation

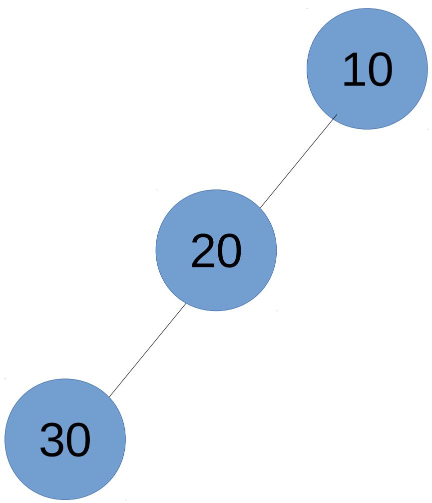
Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



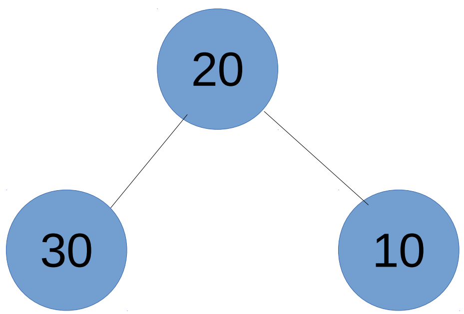
A concurrent insert(15)



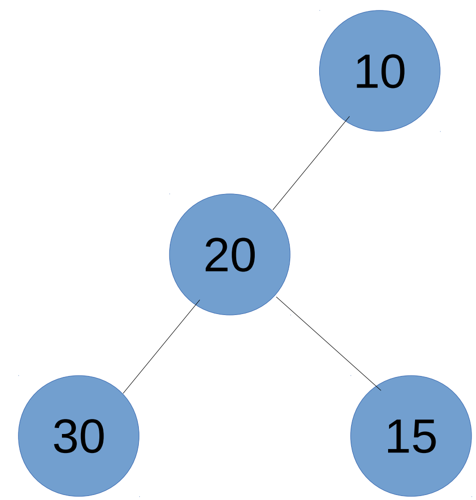
Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



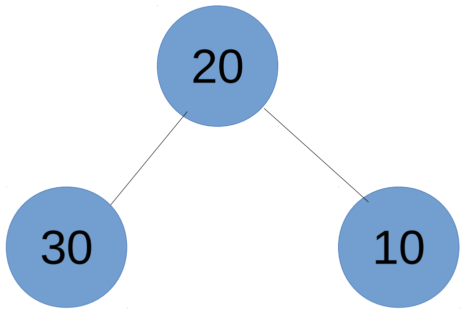
A concurrent insert(15)



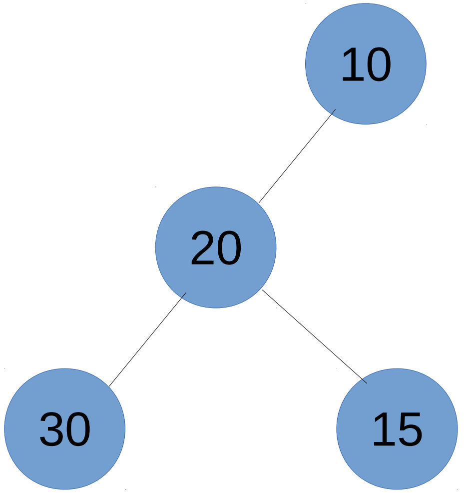
Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent insert(15)



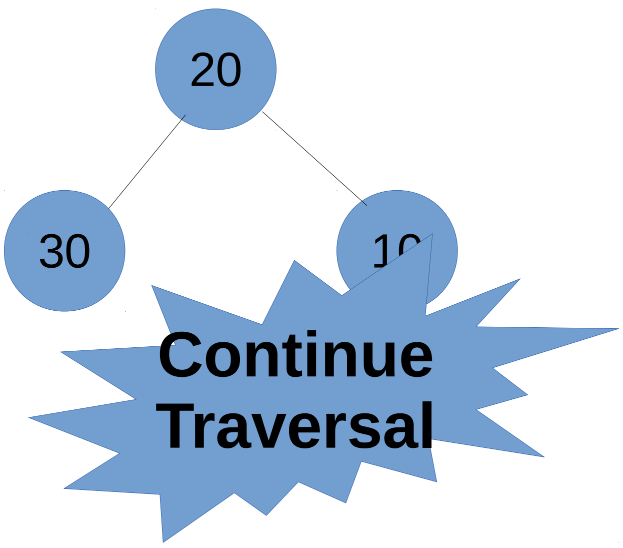
T1 observes that the right child of “20” is not NULL

What is the best to do in both cases?

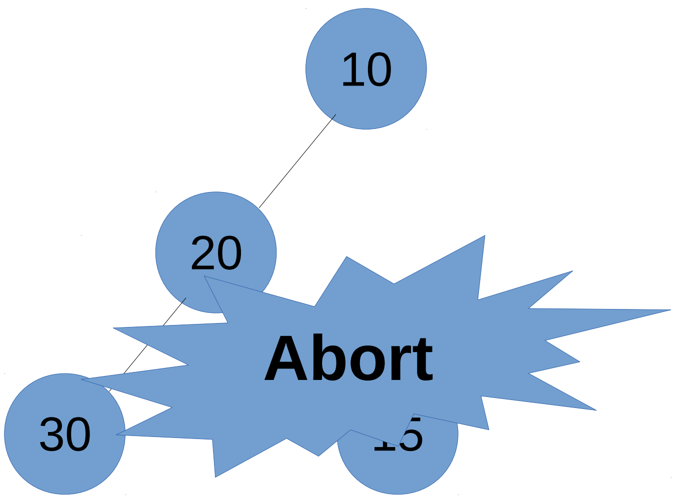
Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent insert(15)



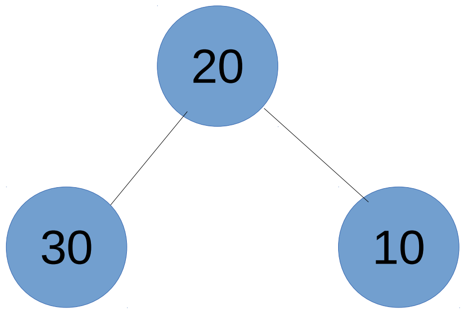
T1 observes that the right child of "20" is not NULL

What is the best to do in both cases?

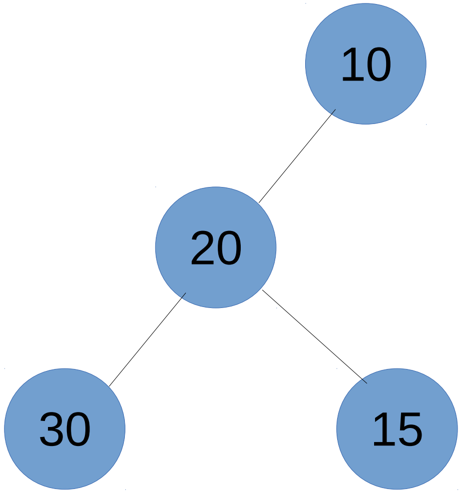
Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



A concurrent insert(15)

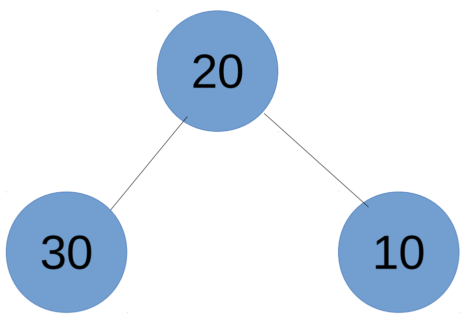


Solution?

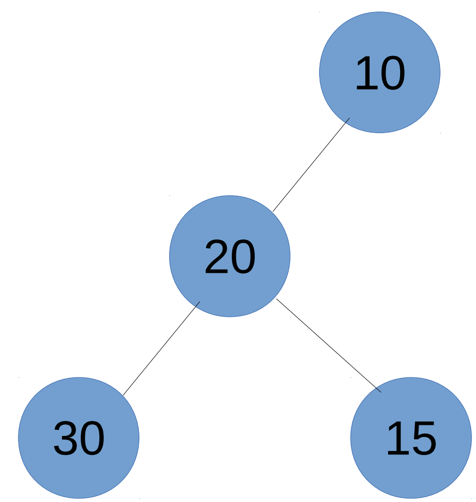
Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



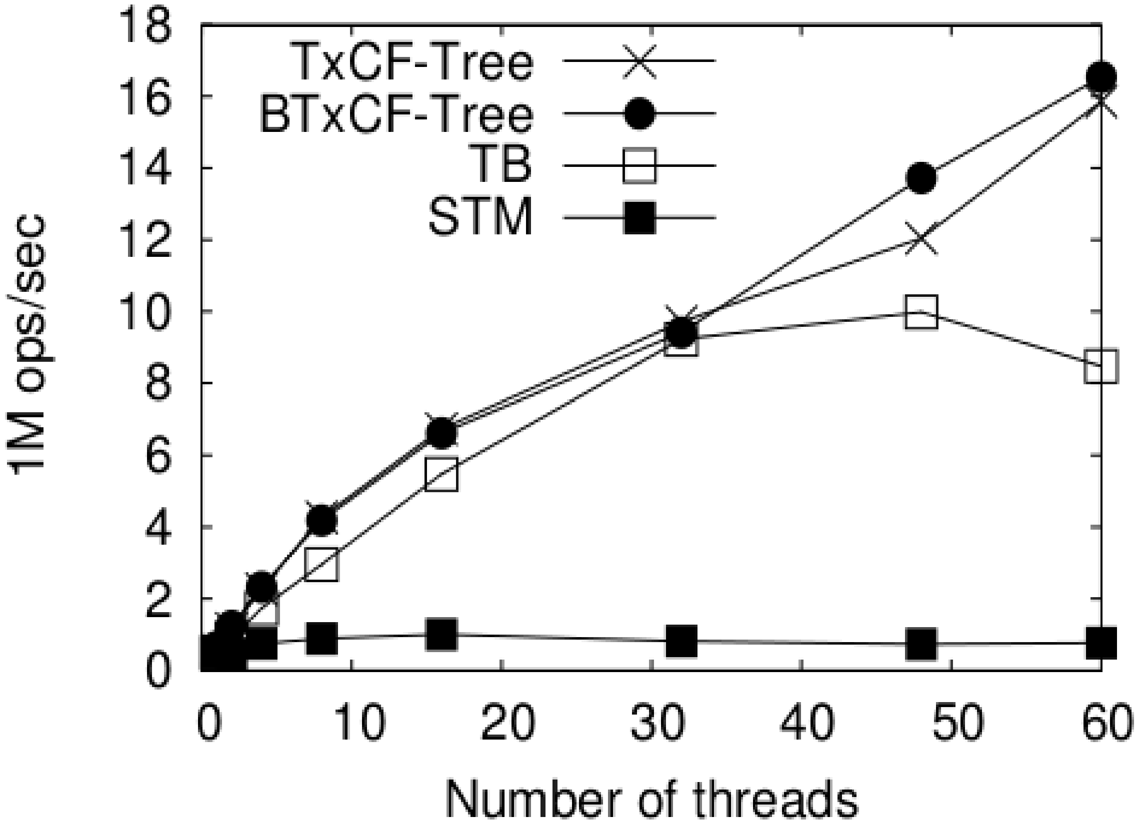
A concurrent insert(15)



Solution?

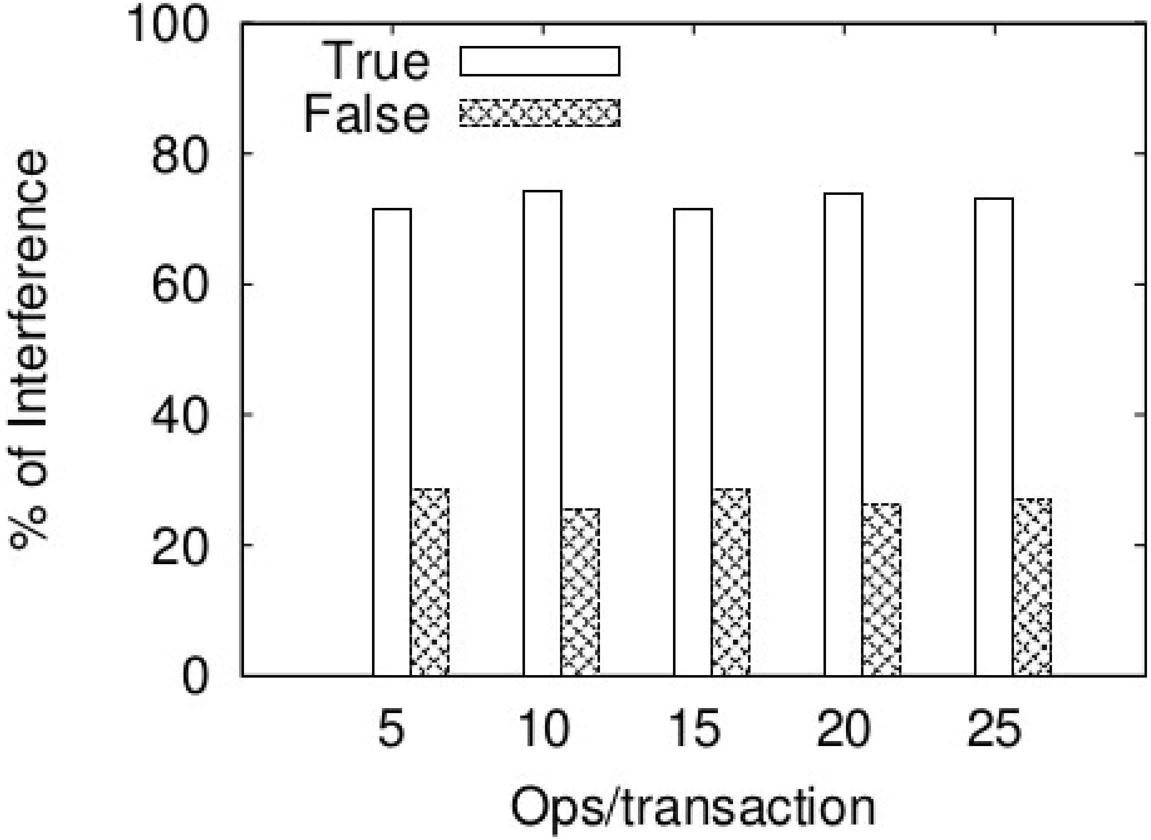
Continue Traversal anyway

Evaluation



AMD 64-cores, size 10K nodes, 50% reads, 5 ops/transaction

Evaluation



AMD 64-cores, size: 10K nodes , 32 threads, 50% reads, 5 ops/transaction

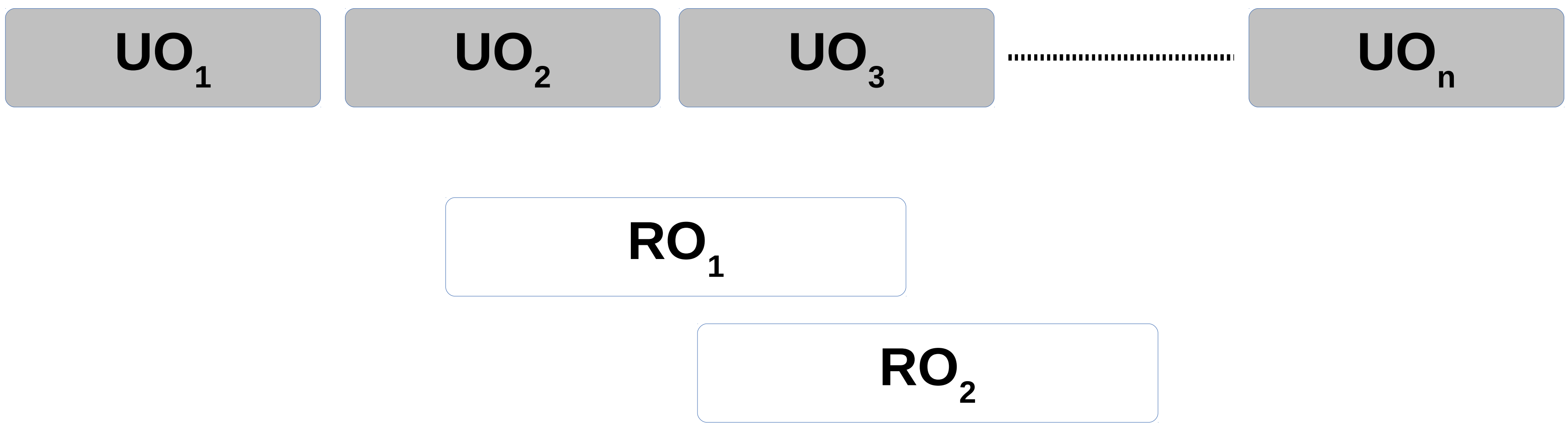
Modeling Transactional Data Structures

Concurrent Data Structures

- Different Designs and Implementations
  - Different ad-hoc approaches for proving correctness.

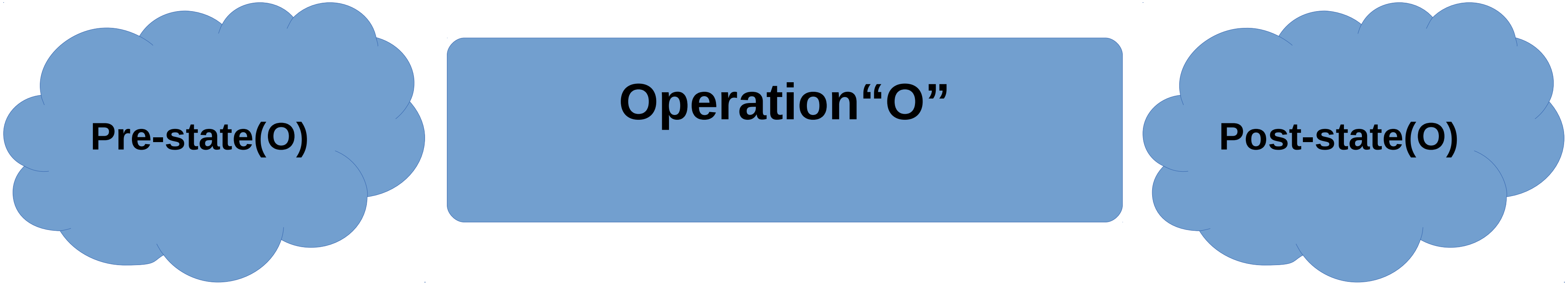
- Is there a unified model for concurrent data structures?
  - General enough.
  - Easy to use.

SWMR Model (Lev-Ari et. Al, DISC'14)



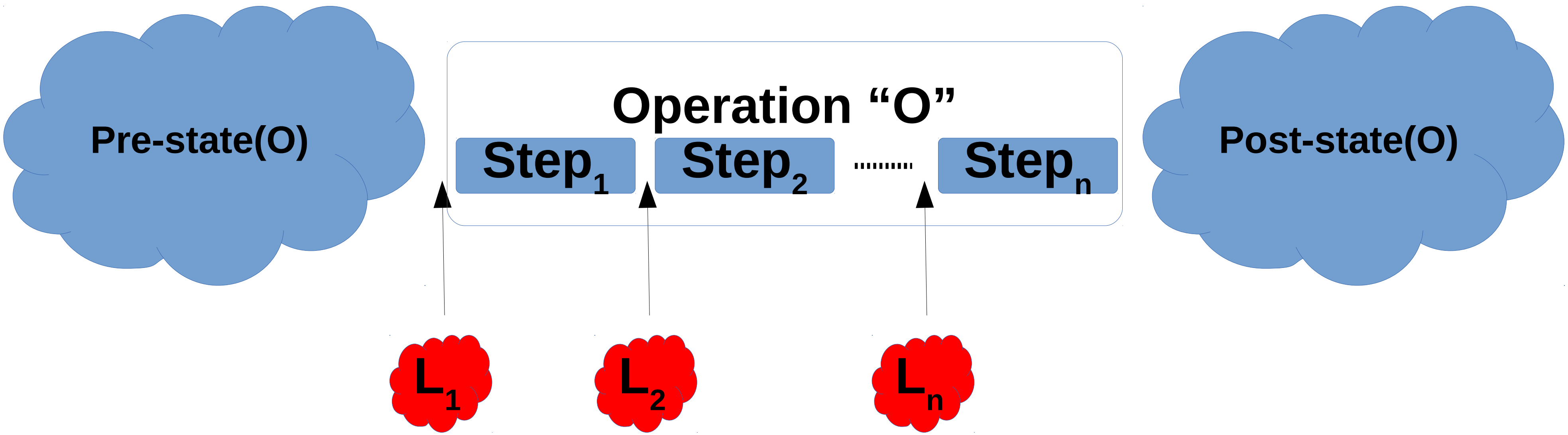
Shared States

- Data Structure is represented as a set of shared variables.
- The values of those variables is the shared state of the data structure.

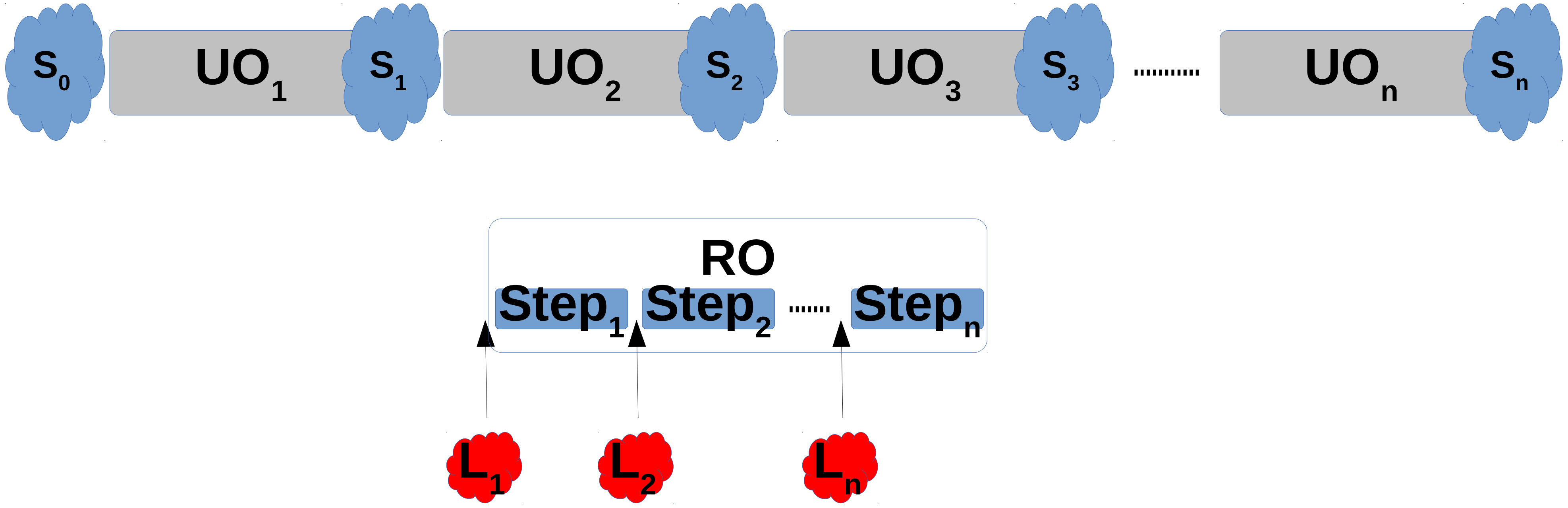


Local States

- Operation is represented as a set of steps.
- The values of the operation's local variables before any step is the local state of the step.

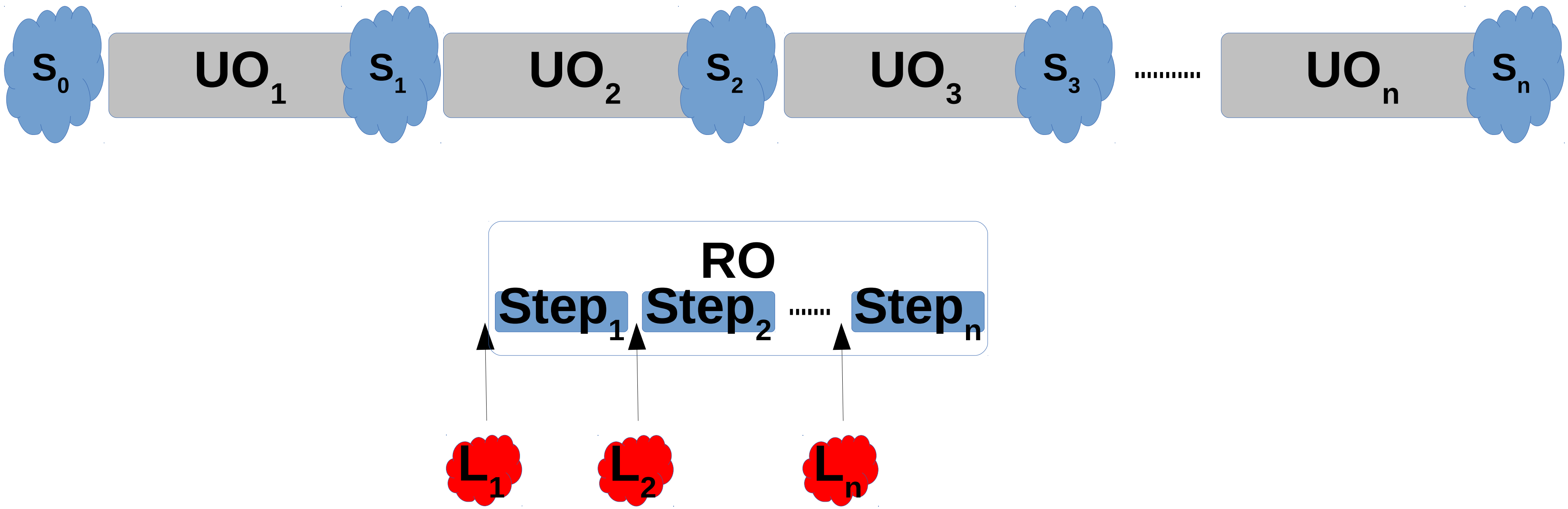


SWMR Scenario

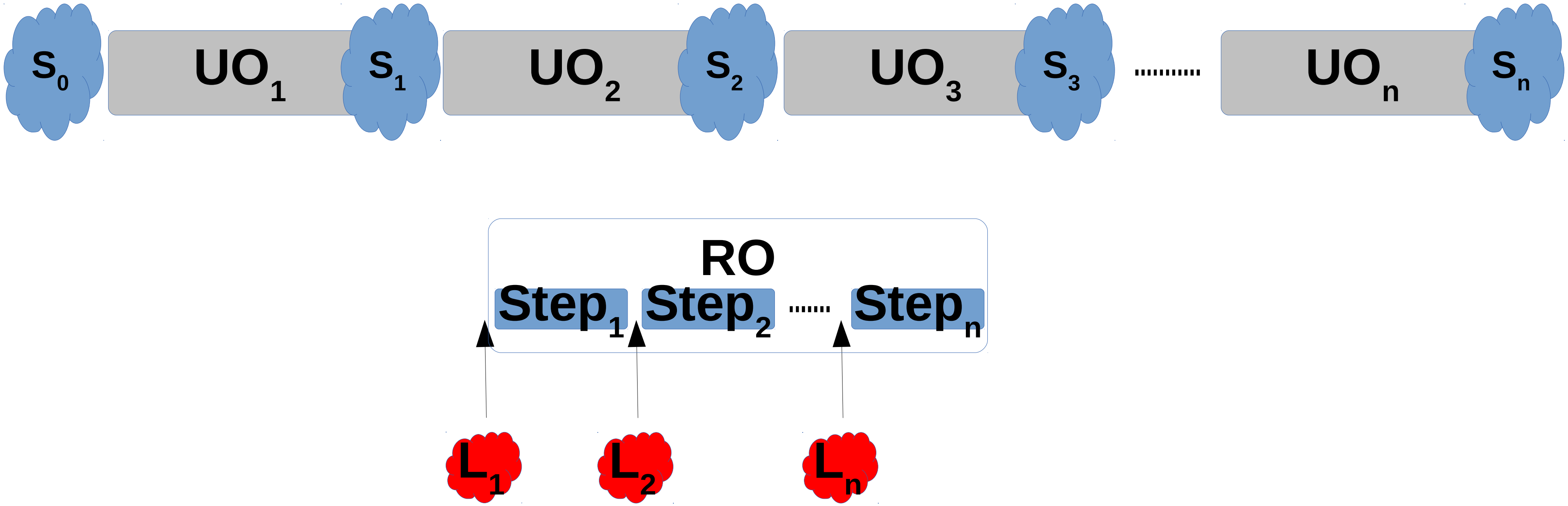


Validity

Validity

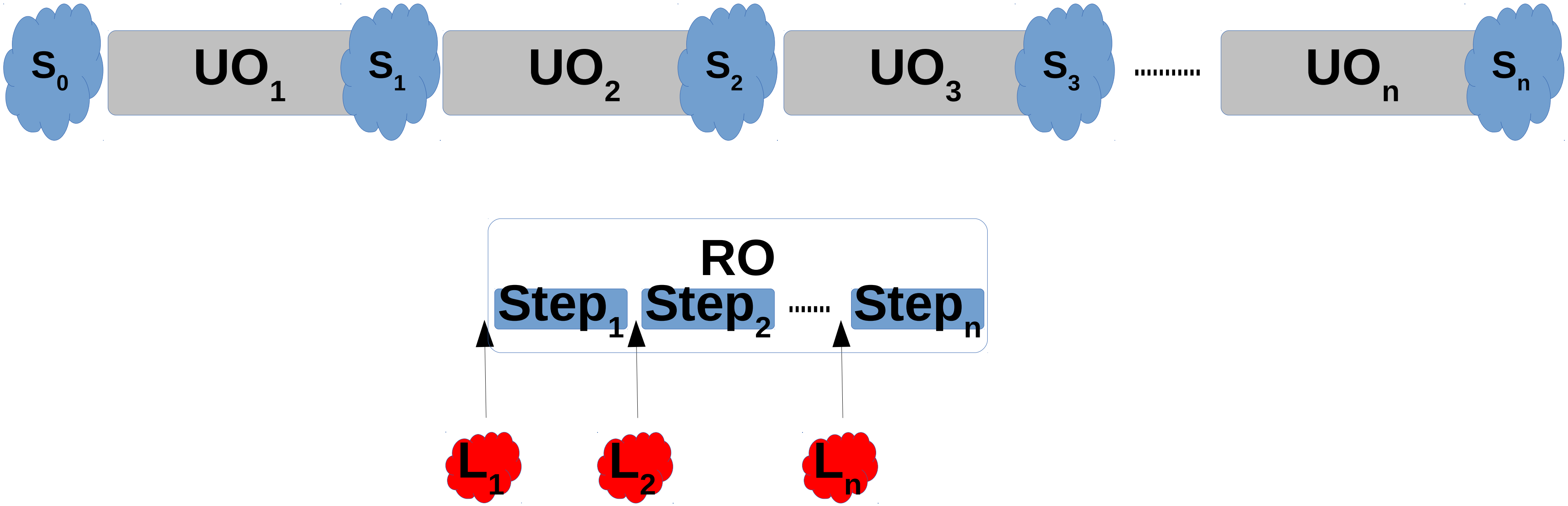


Validity



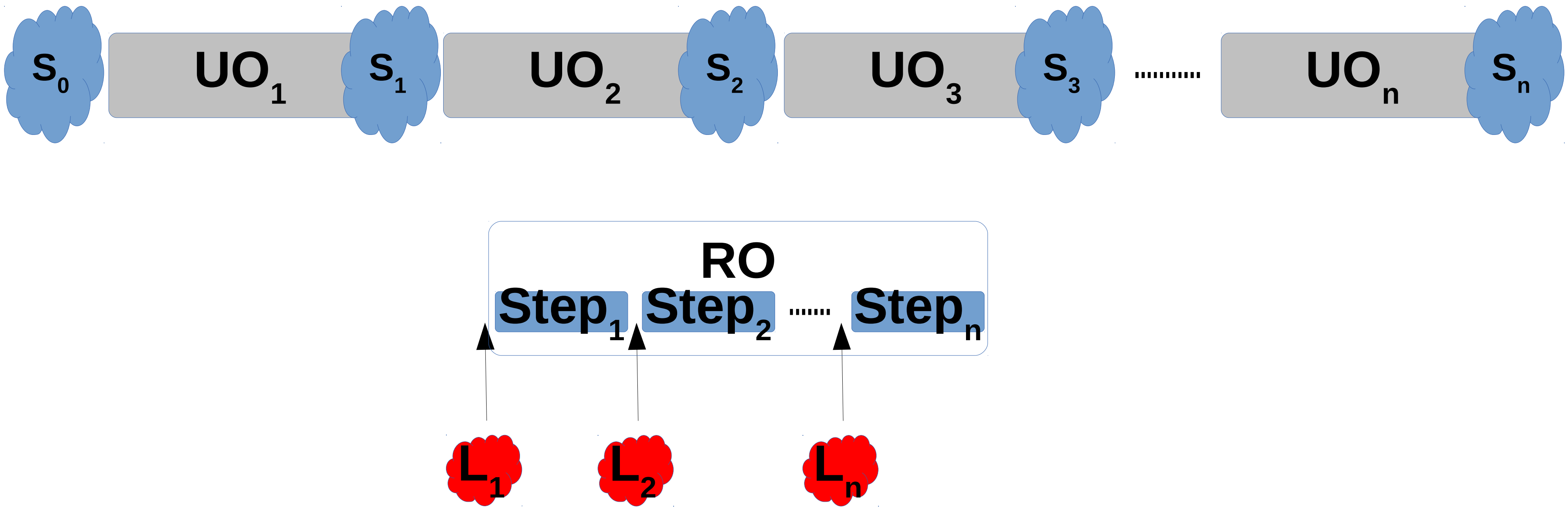
- All S_i are sequentially reachable, so all UO_i are valid.

Validity



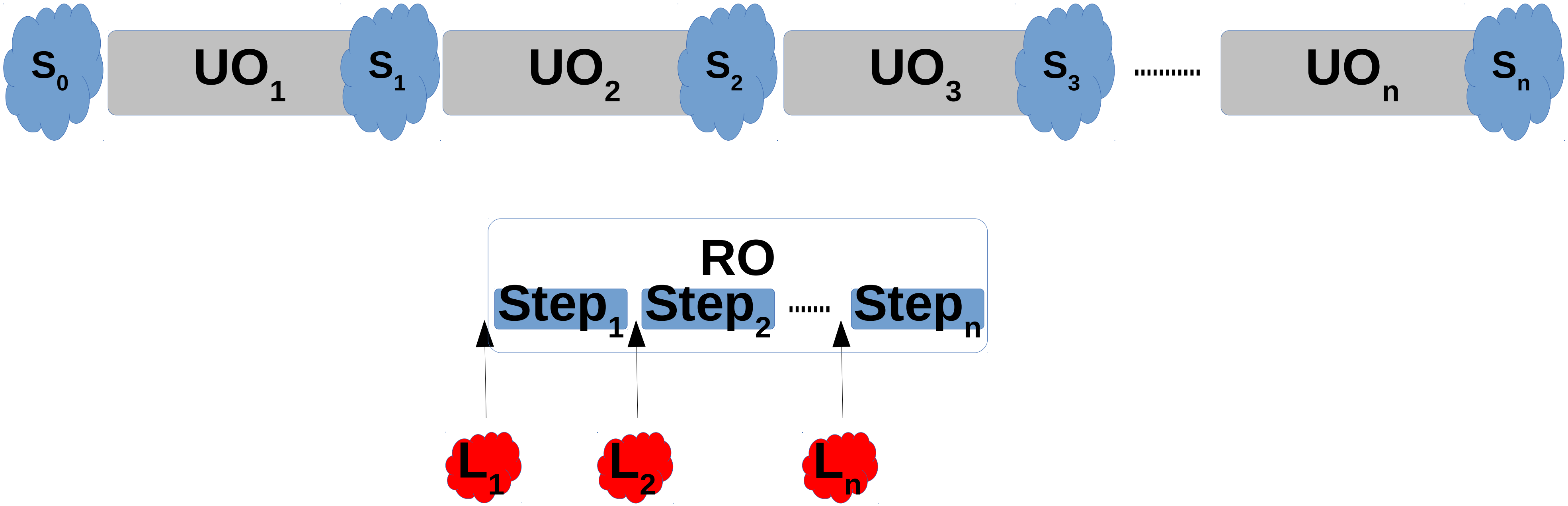
- All S_i are sequentially reachable, so all UO_i are valid.
- $Step_i$ in RO is valid if there is S_j such that a sequential execution of RO starting from S_j reaches L_i.

Validity

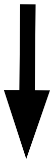


- All S_i are sequentially reachable, so all UO_i are valid.
- $Step_j$ in RO is valid if there is S_i such that a sequential execution of RO starting from S_i reaches L_j.

Validity



- All S_i are sequentially reachable, so all UO_i are valid.
- $Step_j$ in RO is valid if there is S_i such that a sequential execution of RO starting from S_i reaches L_j.



- $Step_j$ in RO is valid if there is a “base point” where the “base condition” of $step_j$ holds.

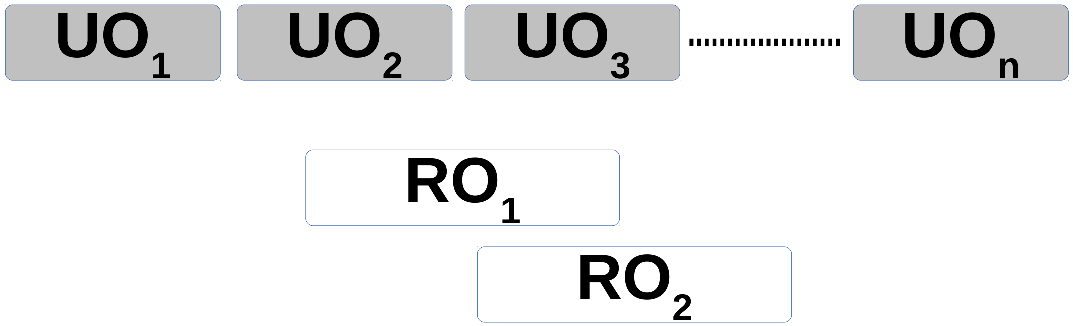
Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.

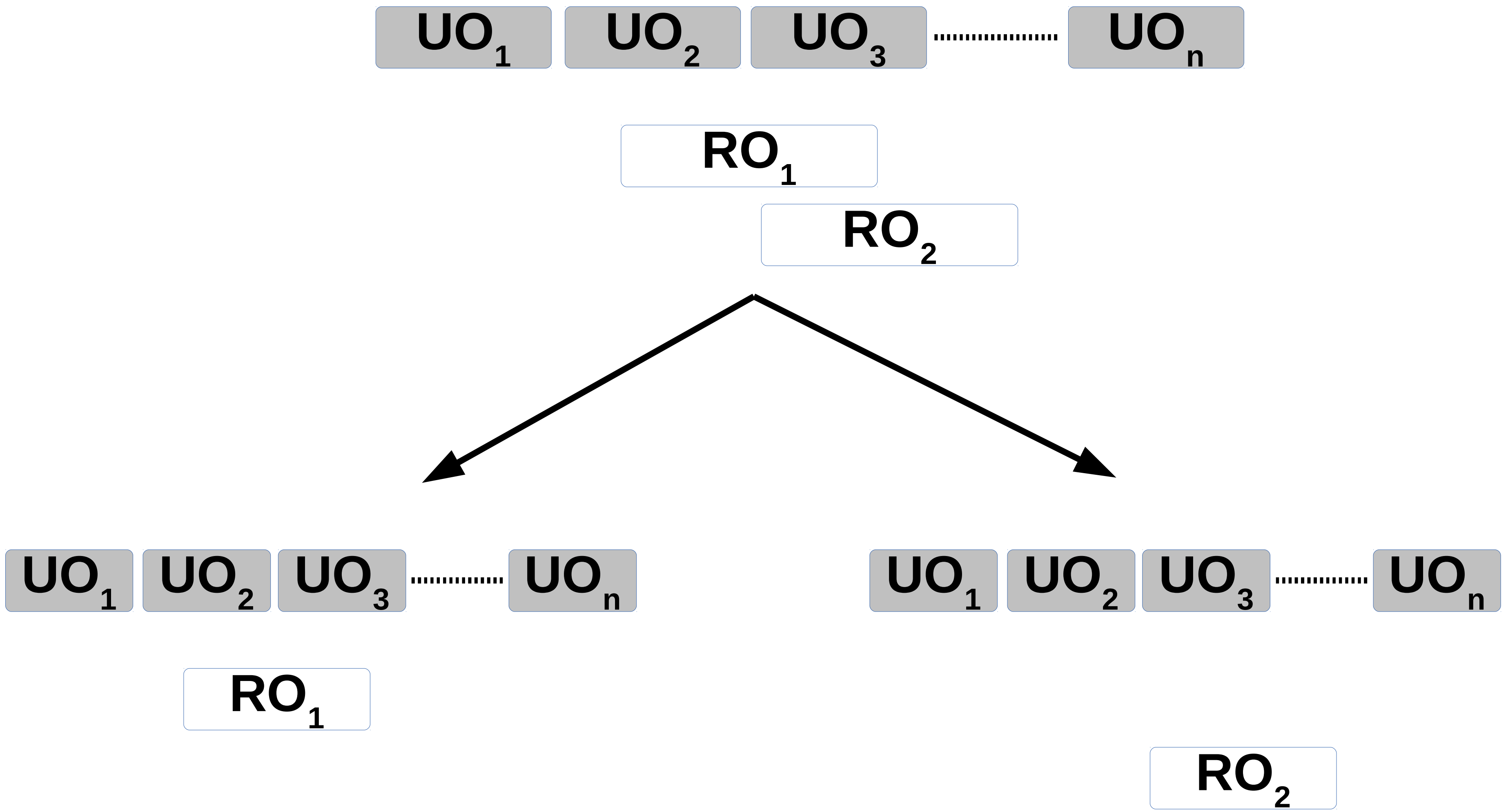
Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.

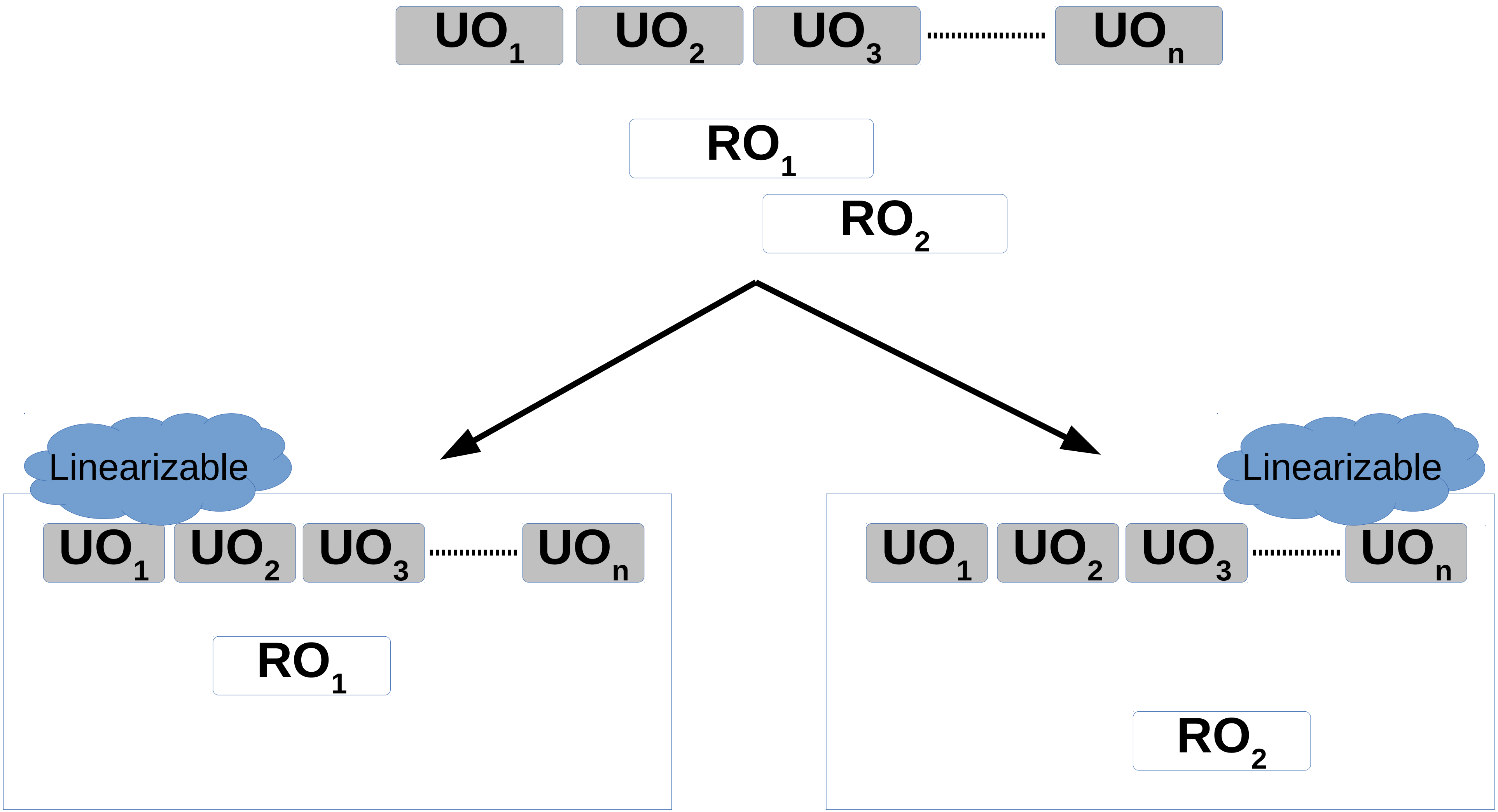
Regularity



Regularity

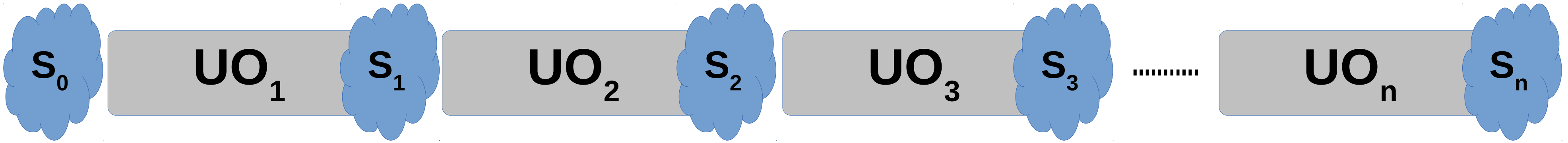


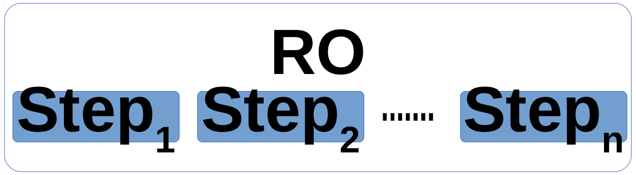
Regularity



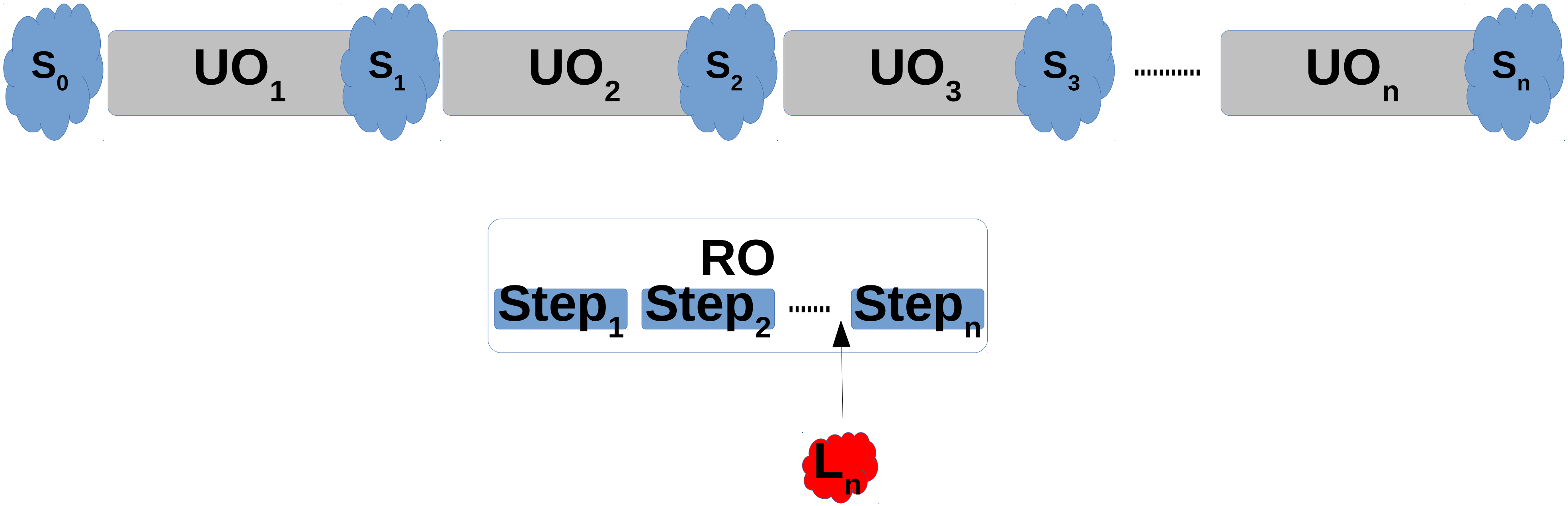
UO_1 UO_2 UO_3 UO_n
RO_1
RO_2
Linearizable
UO_1 UO_2 UO_3 UO_n
RO_1
Linearizable
UO_1 UO_2 UO_3 UO_n
RO_2

Regularity





Regularity



- Acceptable base points for RO's return step are only S_1, S_2, S_3.
  - Observes either the last update or a concurrent update.

Example

Function *remove(n)*

$p \leftarrow \perp$

$next \leftarrow \boldsymbol{read}(head.next)$

while $next \neq n$

$\quad p \leftarrow next$

$\quad next \leftarrow \boldsymbol{read}(p.next)$

$\boldsymbol{write}(p.next,\ n.next)$

Function *insertLast(n)*

$last \leftarrow readLast()$

$\boldsymbol{write}(last.next,\ n)$

Base conditions:

$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$

$\Phi_3 : head \overset{*}{\Rightarrow} n$

Function *readLast()*

$n \leftarrow \perp$

$next \leftarrow \boldsymbol{read}(head.next)$

while $next \neq \perp$

$\quad n \leftarrow next$

$\quad next \leftarrow \boldsymbol{read}(n.next)$

$\boldsymbol{return}(n)$

Example

Function *remove(n)*
- $p \leftarrow \perp$
- $next \leftarrow \textbf{\textit{read}}(head.next)$
- **while** $next \neq n$
  - $p \leftarrow next$
  - $next \leftarrow \textbf{\textit{read}}(p.next)$
- $\textbf{\textit{write}}(p.next,\ n.next)$

Function *insertLast(n)*
- $last \leftarrow readLast()$
- $\textbf{\textit{write}}(last.next,\ n)$

Base conditions:

$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$

$\Phi_3 : head \overset{*}{\Rightarrow} n$

Function *readLast()*
- $n \leftarrow \perp$
- $next \leftarrow \textbf{\textit{read}}(head.next)$
- **while** $next \neq \perp$
  - $n \leftarrow next$
  - $next \leftarrow \textbf{\textit{read}}(n.next)$
- $\textbf{\textit{return}}(n)$

Example

Function *remove(n)*

$p \leftarrow \bot$

$next \leftarrow \boldsymbol{read}(head.next)$

while $next \neq n$

$p \leftarrow next$

$next \leftarrow \boldsymbol{read}(p.next)$

$\boldsymbol{write}(p.next,\ n.next)$

Function *insertLast(n)*

$last \leftarrow readLast()$

$\boldsymbol{write}(last.next,\ n)$

Base conditions:

Function *readLast()*

$n \leftarrow \bot$

$\Phi_1 : true$ — $next \leftarrow \boldsymbol{read}(head.next)$

while $next \neq \bot$

$n \leftarrow next$

$\Phi_2 : head \overset{*}{\Rightarrow} n$ — $next \leftarrow \boldsymbol{read}(n.next)$

$\Phi_3 : head \overset{*}{\Rightarrow} n$ — $\boldsymbol{return}(n)$

Where is the Problem?

It covers only single-writer designs

It does not cover composable designs

Can we cover a wider set?

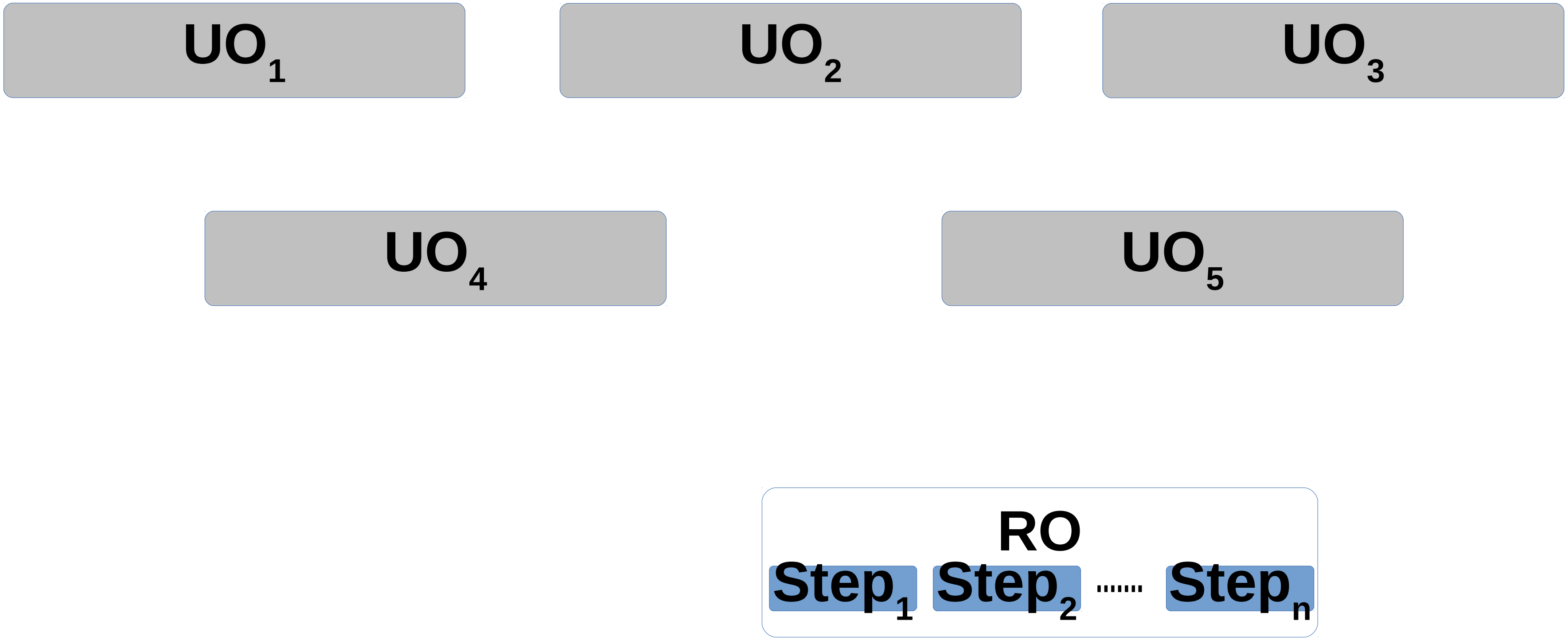
Optimistic Composable Data Structures

Our Models

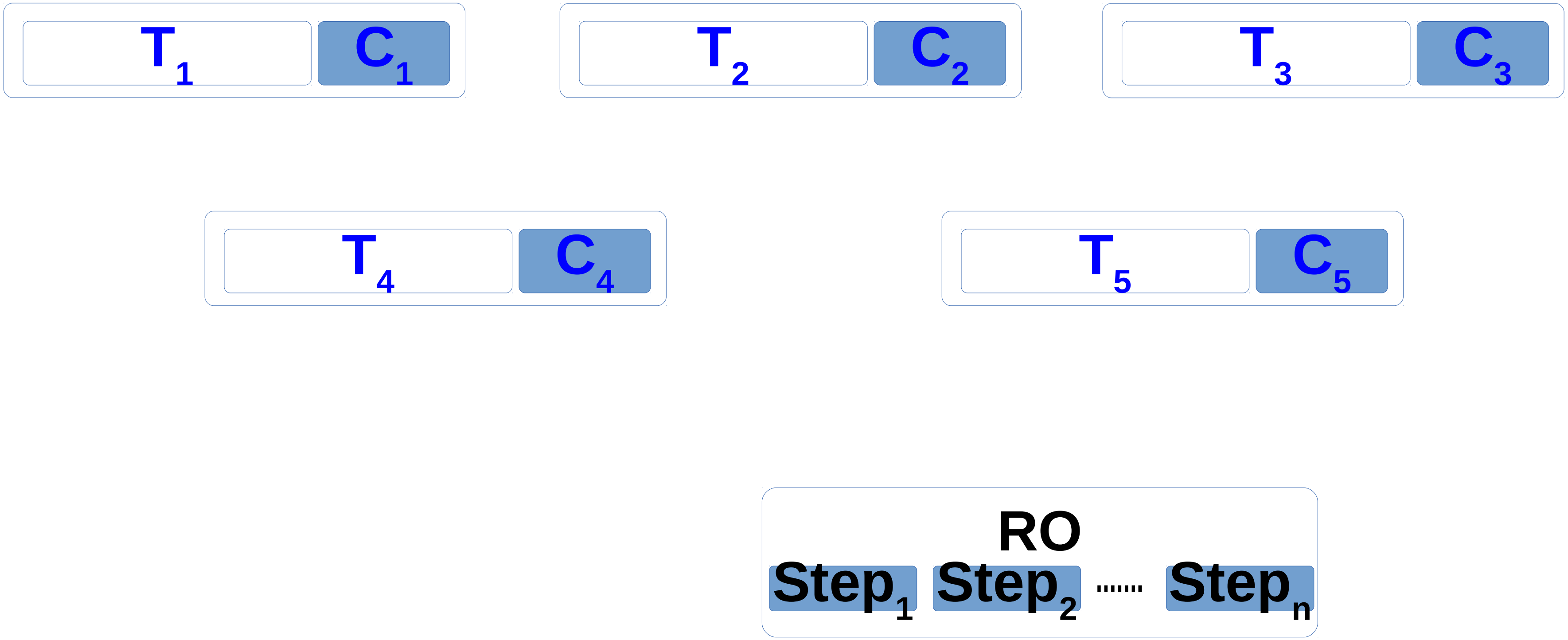
Single Writer Commit (SWC)

Composable SWC (C-SWC)

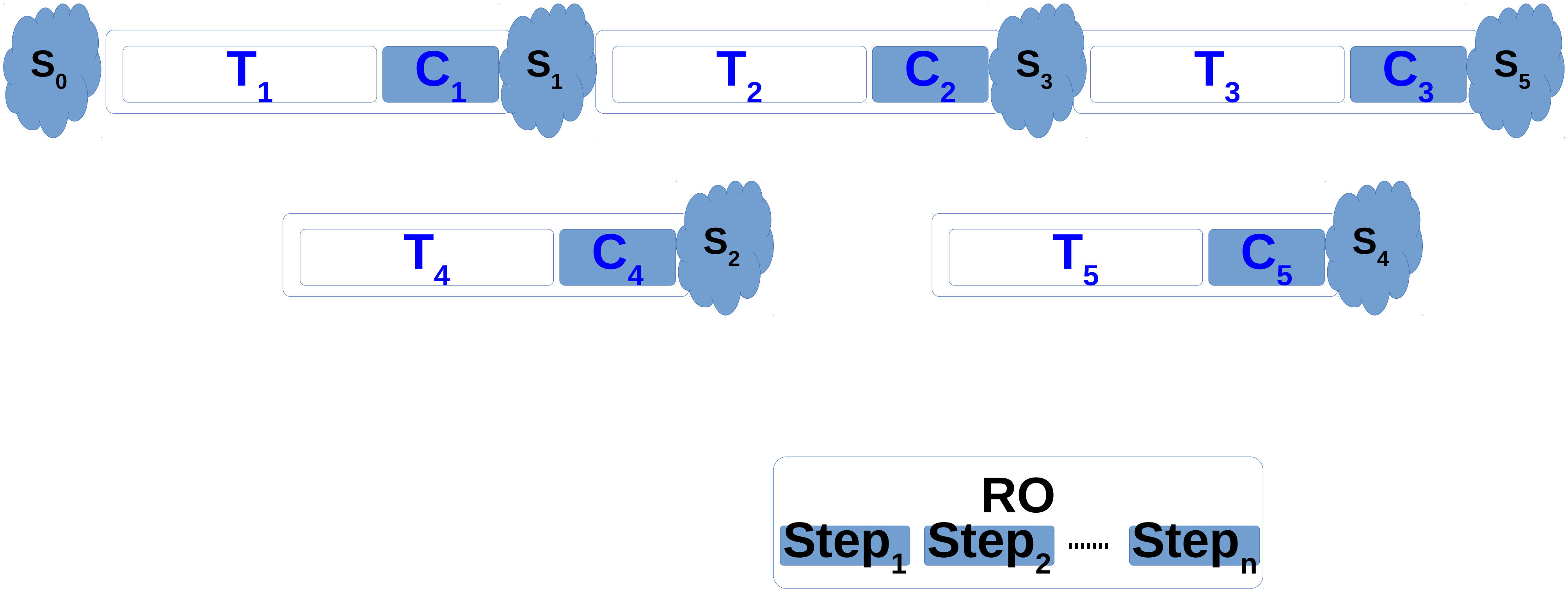
SWC Model



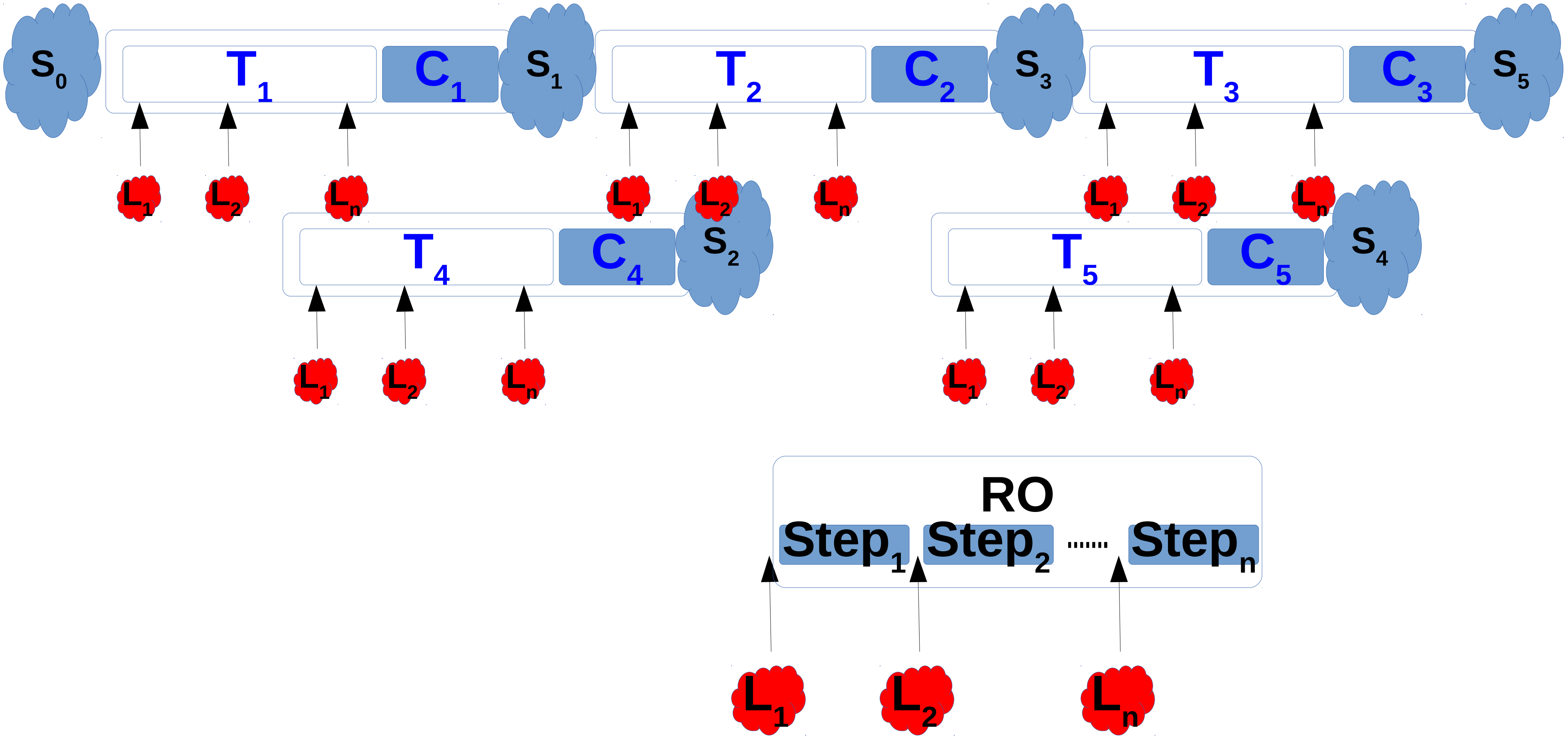
SWC Model



SWC Model



SWC Model



S0
T1
C1
S1
T2
C2
S3
T3
C3
S5
L1
L2
Ln
L1
L2
Ln
L1
L2
Ln
T4
C4
S2
T5
C5
S4
L1
L2
Ln
L1
L2
Ln
RO
Step1
Step2
....
Stepn
L1
L2
Ln

Even More...

- Do we really need single commit at a time:
  - NO!!!
  - It is enough to execute commit phases atomically with *single lock atomicity (SLA)* guarantees.
  - More practical alternatives:
    - HTM (e.g. Intel TSX).
    - STM (e.g. NOrec “the SLA version”).

Example

```
procedure READLAST
    last ← ⊥
    next ← read(head.next)                         ▷ φ1 : true
    while next ≠ ⊥ do
        last ← next
        next ← read(last.next)                     ▷ φ2 : head ⇒* last
    return(last)                                   ▷ φ3 : head ⇒* last
end procedure

procedure INSERTLAST(n)
    last ← ⊥
    next ← read(head.next)                         ▷ φ4 : true
    while next ≠ ⊥ do
        last ← next
        next ← read(last.next)                     ▷ φ5 : head ⇒* last
    lockAcquire(gl)
                                                   ▷ φ6 : head ⇒* last
    if read(last.next) ≠ ⊥ then
        lockRelease(gl)
        go to 10
    write(last.next, n)
    lockRelease(gl)
end procedure
```

Example

```
1:  procedure READLAST
2:      last ← ⊥
3:      next ← read(head.next)                     ▷ φ1 : true
4:      while next ≠ ⊥ do
5:          last ← next
6:          next ← read(last.next)                 ▷ φ2 : head ⇒* last
7:      return(last)                               ▷ φ3 : head ⇒* last
8:  end procedure

9:  procedure INSERTLAST(n)
10:     last ← ⊥
11:     next ← read(head.next)                     ▷ φ4 : true
12:     while next ≠ ⊥ do
13:         last ← next
14:         next ← read(last.next)                 ▷ φ5 : head ⇒* last
15:     lockAcquire(gl)
                                                   ▷ φ6 : head ⇒* last
16:     if read(last.next) ≠ ⊥ then
17:         lockRelease(gl)
18:         go to 10
19:     write(last.next, n)
20:     lockRelease(gl)
21: end procedure
```

Example

```
procedure READLAST
    last ← ⊥
    next ← read(head.next)                ▷ φ1 : true
    while next ≠ ⊥ do
        last ← next
        next ← read(last.next)            ▷ φ2 : head ⇒* last
    return(last)                          ▷ φ3 : head ⇒* last
end procedure

procedure INSERTLAST(n)
    last ← ⊥
    next ← read(head.next)                ▷ φ4 : true
    while next ≠ ⊥ do
        last ← next
        next ← read(last.next)            ▷ φ5 : head ⇒* last
    lockAcquire(gl)
                                          ▷ φ6 : head ⇒* last
    if read(last.next) ≠ ⊥ then
        lockRelease(gl)
        go to 10
    write(last.next, n)
    lockRelease(gl)
end procedure
```

Composable SWC Model (C-SWC)

procedure ATOMIC:T_1

  $x = 5$

  if $readLast() \neq x$ **then**

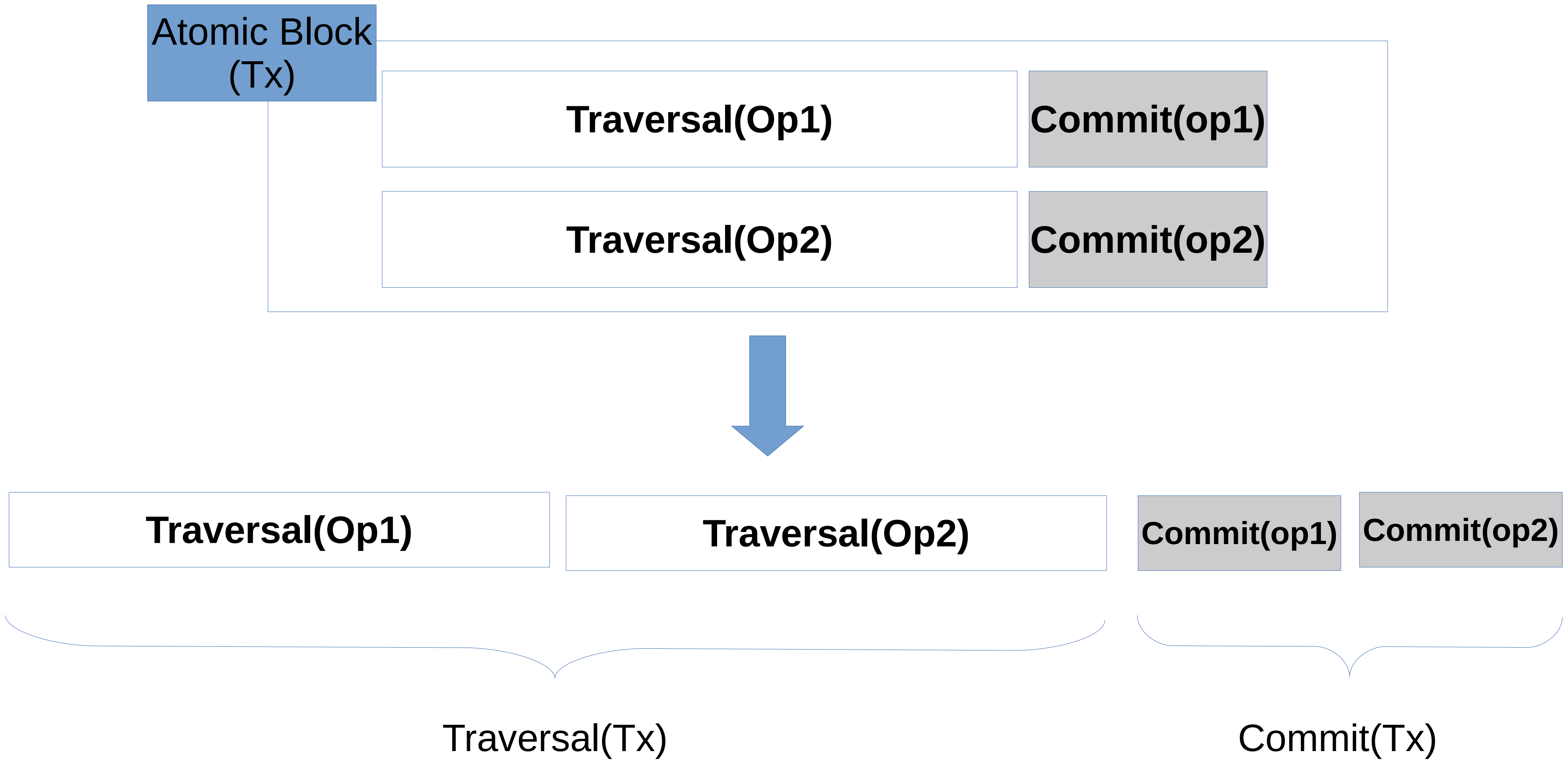
    $insertLast(x)$

  if $readLast() \neq x$ **then**

    ... // illegal execution

end procedure

Composable SWC Model (C-SWC)



What is remaining?

- Internal Consistency.
  - The commit phase of each operation reflects what the operation observed in its traversal.
  - The shared state of an operation is visible to subsequent operations in the same transaction.

How to prove internal consistency?

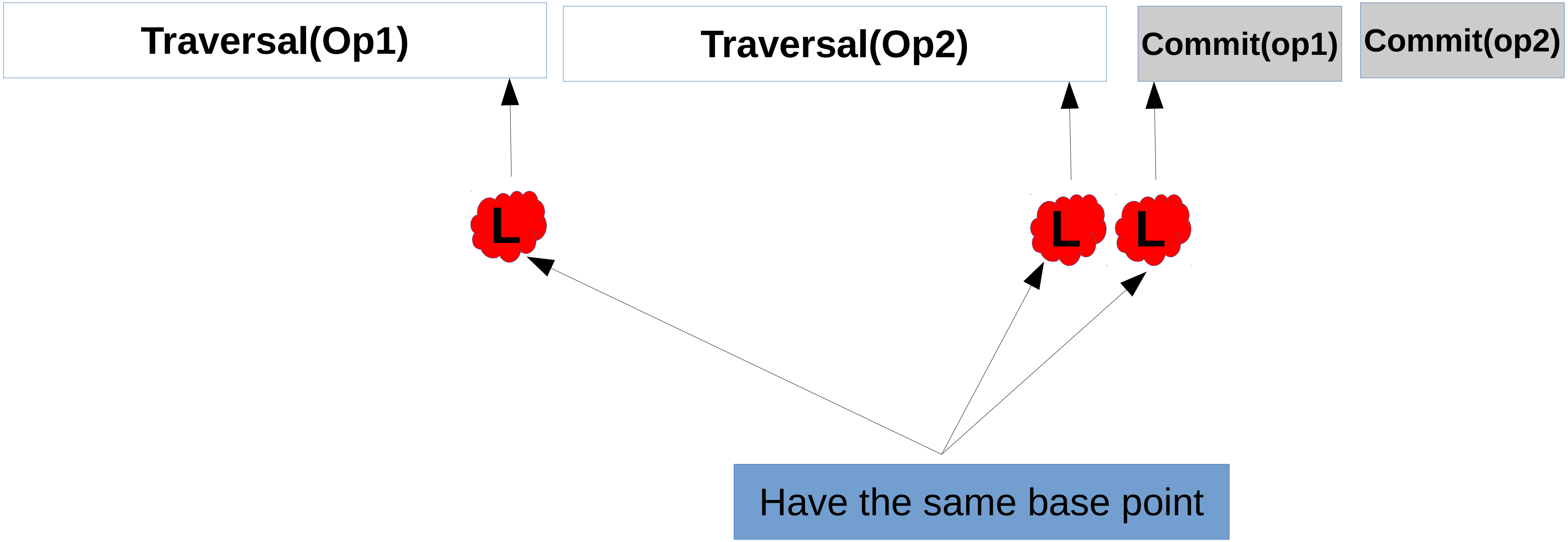
Traversal(Op1)

Traversal(Op2)

Commit(op1)

Commit(op2)

How to prove internal consistency?



Conclusions

Our Contributions

Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

SWC and C-SWC Models
WTTM 2015, under submission

Thanks!

Questions?