FlowPools

A Lock-Free Deterministic Concurrent Dataflow Abstraction

Aleksandar Prokopec Heather Miller Tobias Schlatter
Philipp Haller Martin Odersky

LAMP, EPFL

September 12, 2012

Scala

- Functional
- Object-Oriented
- Runs on JVM

Outline

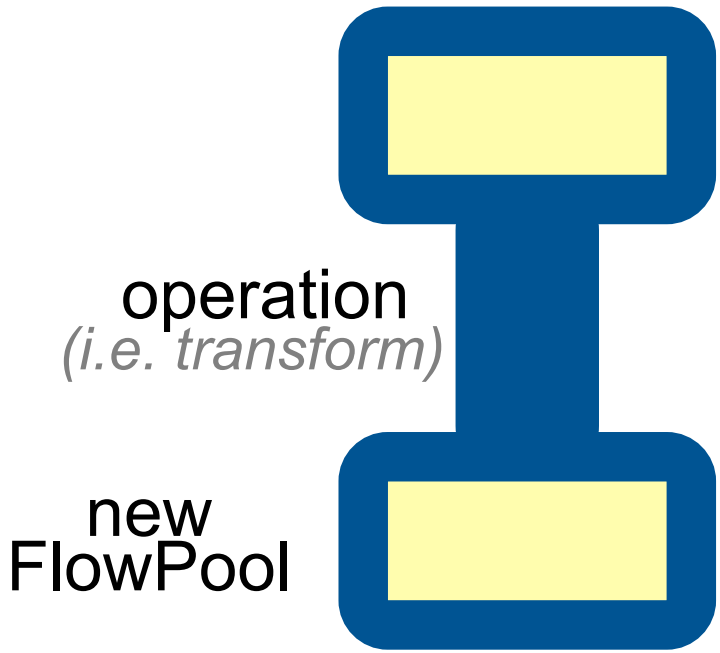
What is a FlowPool

Flow Graph

FlowPool
(collection)

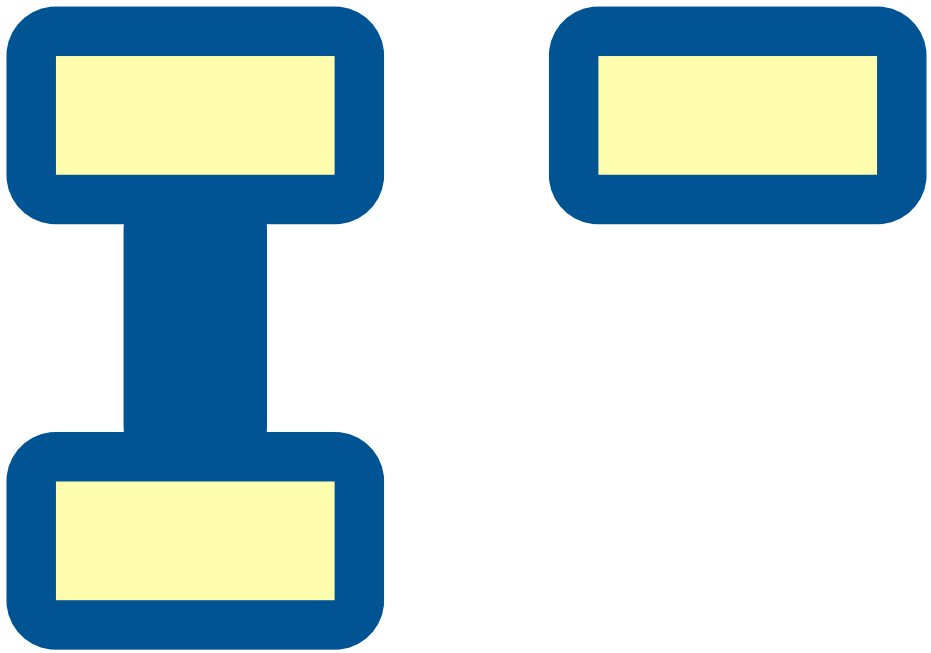
What is a FlowPool

Flow Graph



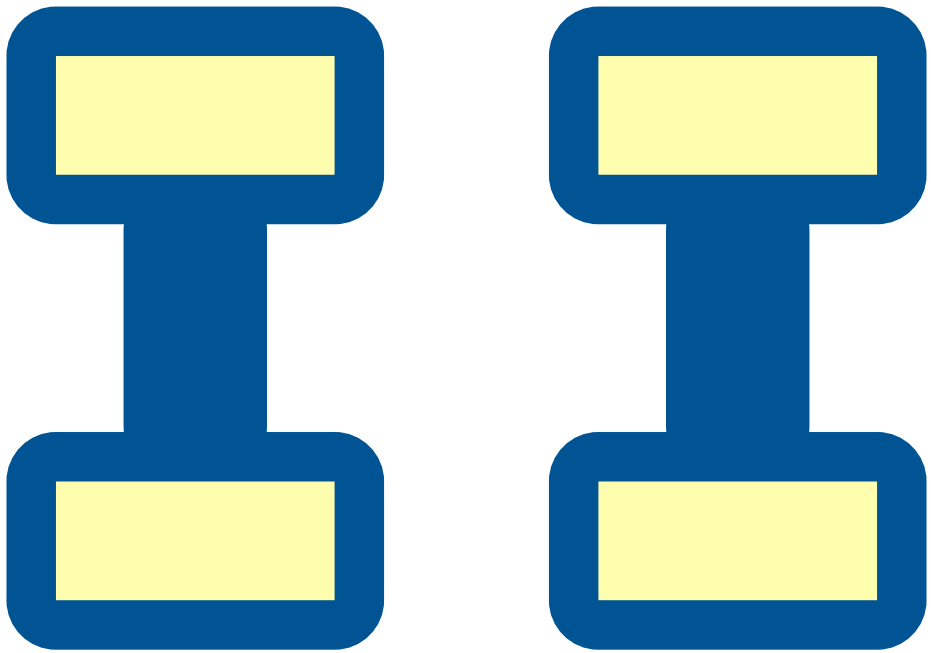
What is a FlowPool

Flow Graph



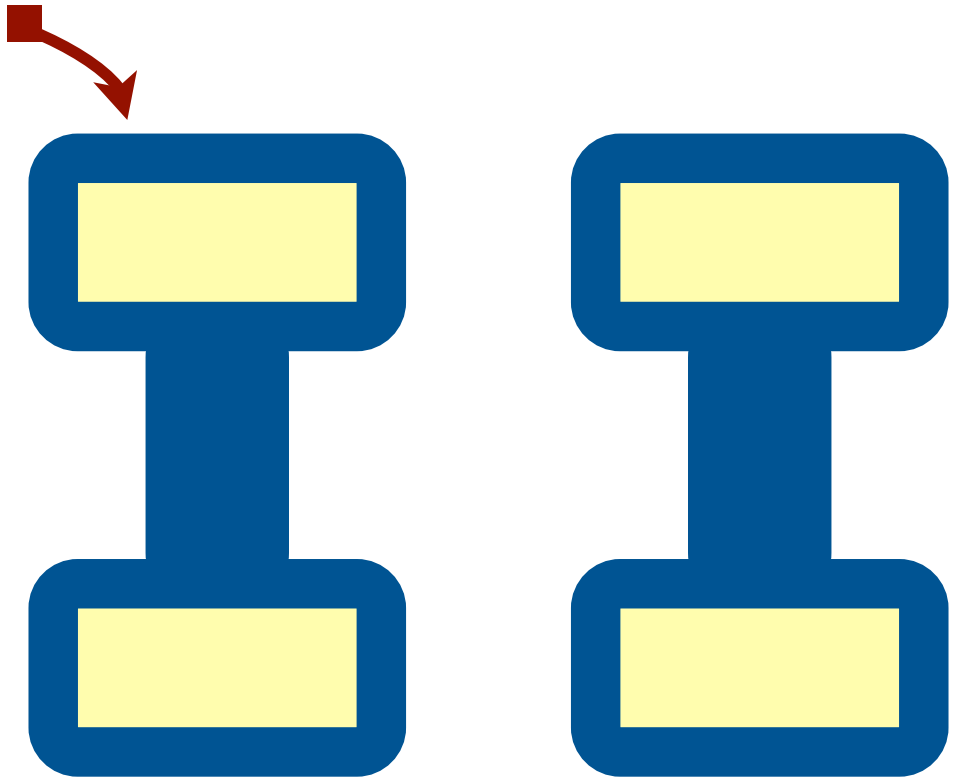
What is a FlowPool

Flow Graph



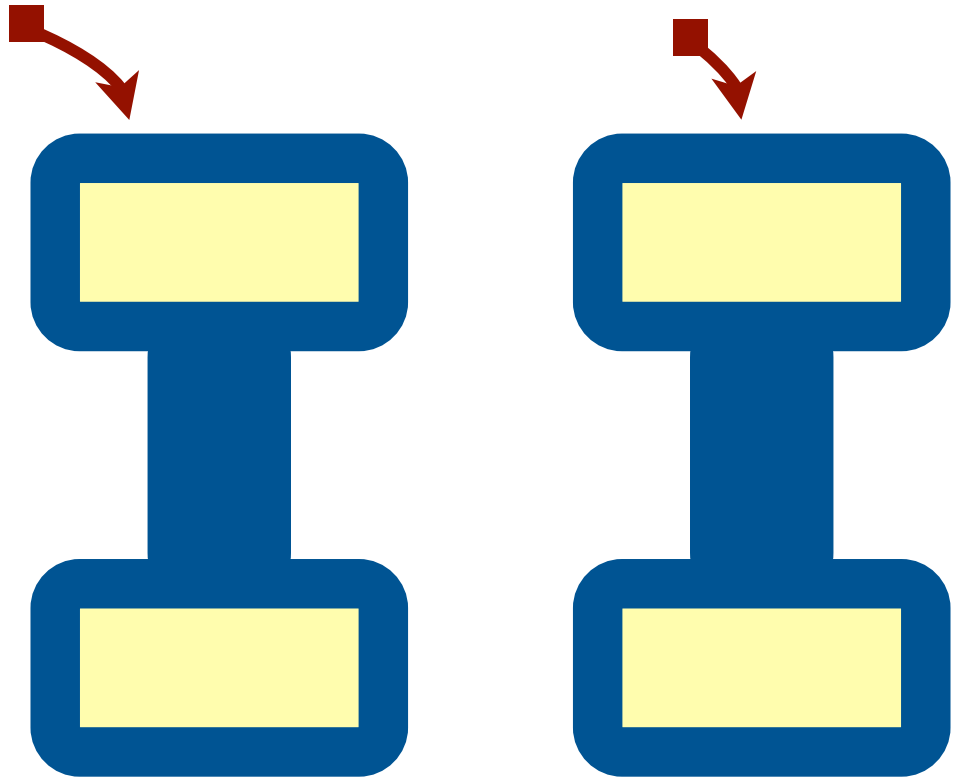
What is a FlowPool

Flow Graph



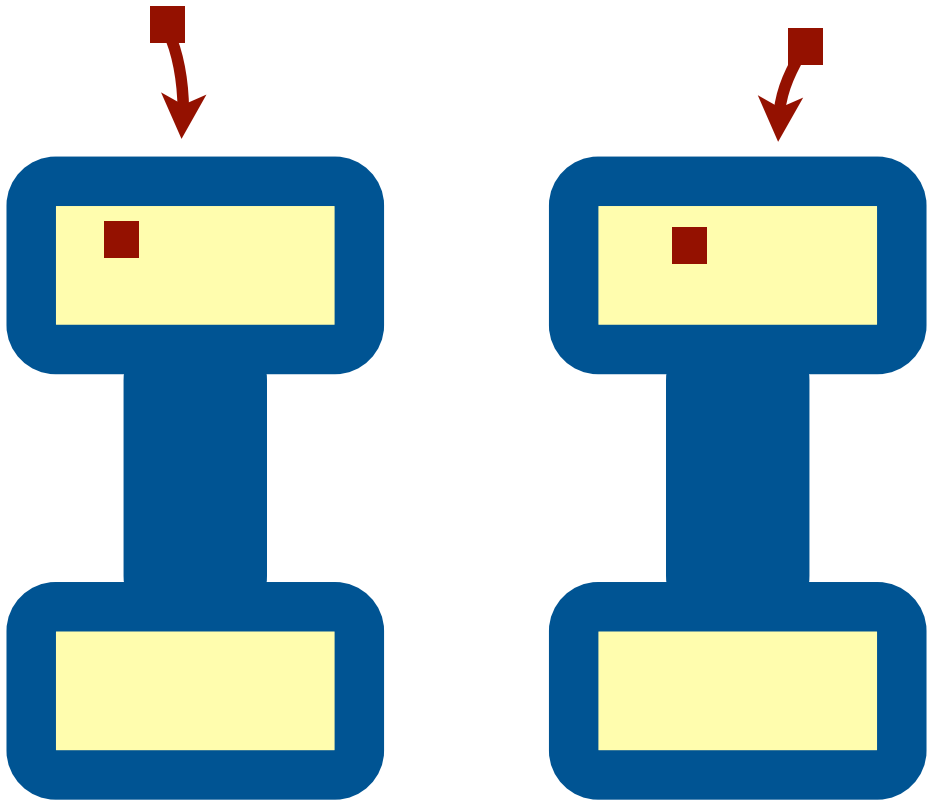
What is a FlowPool

Flow Graph



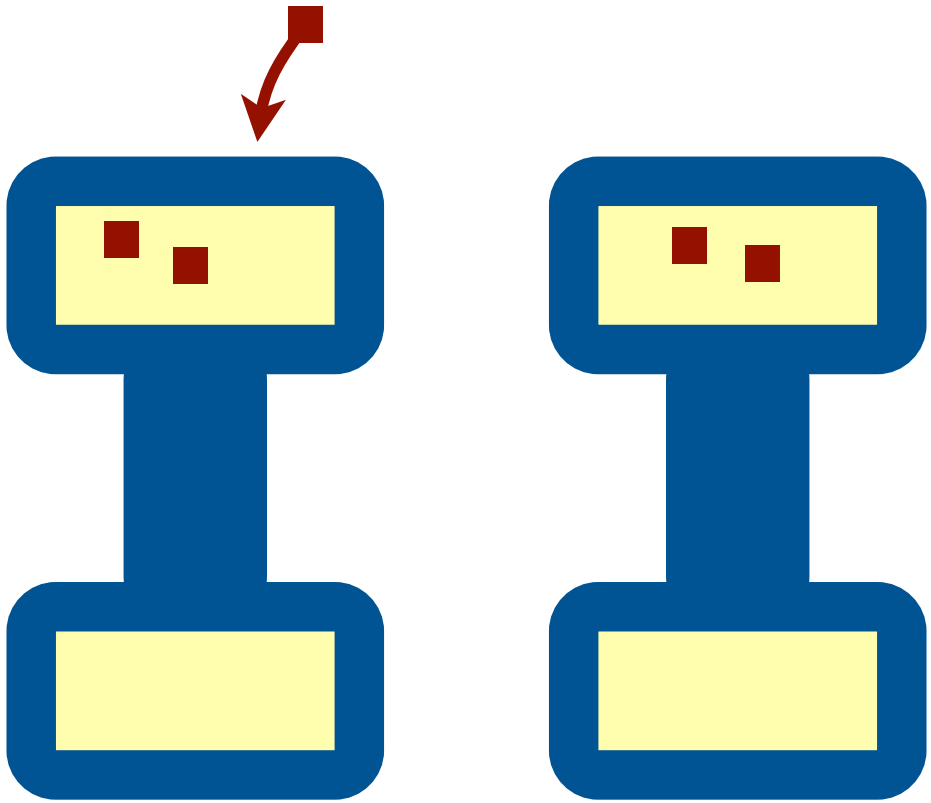
What is a FlowPool

Flow Graph



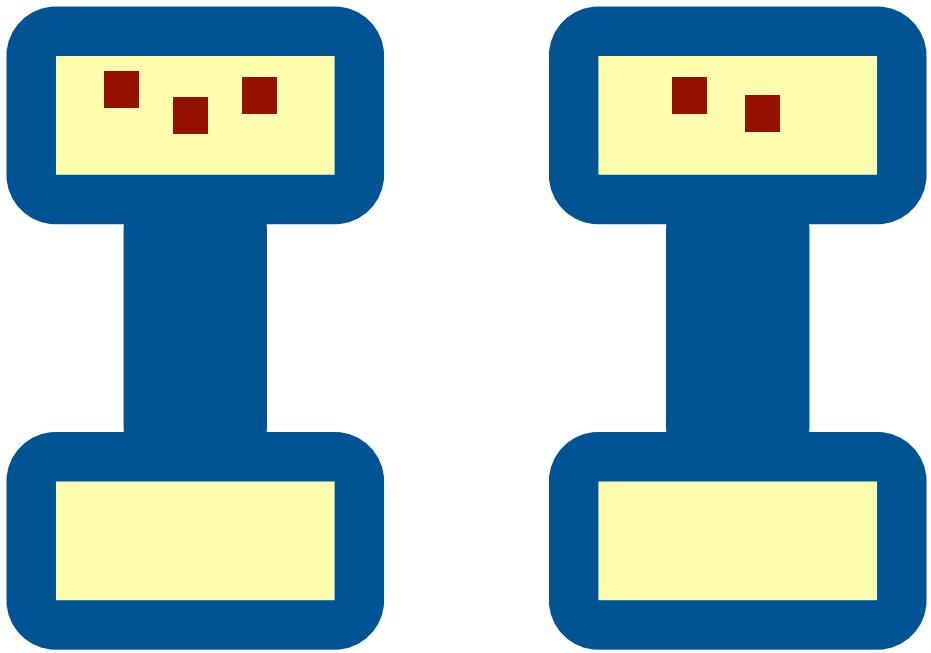
What is a FlowPool

Flow Graph



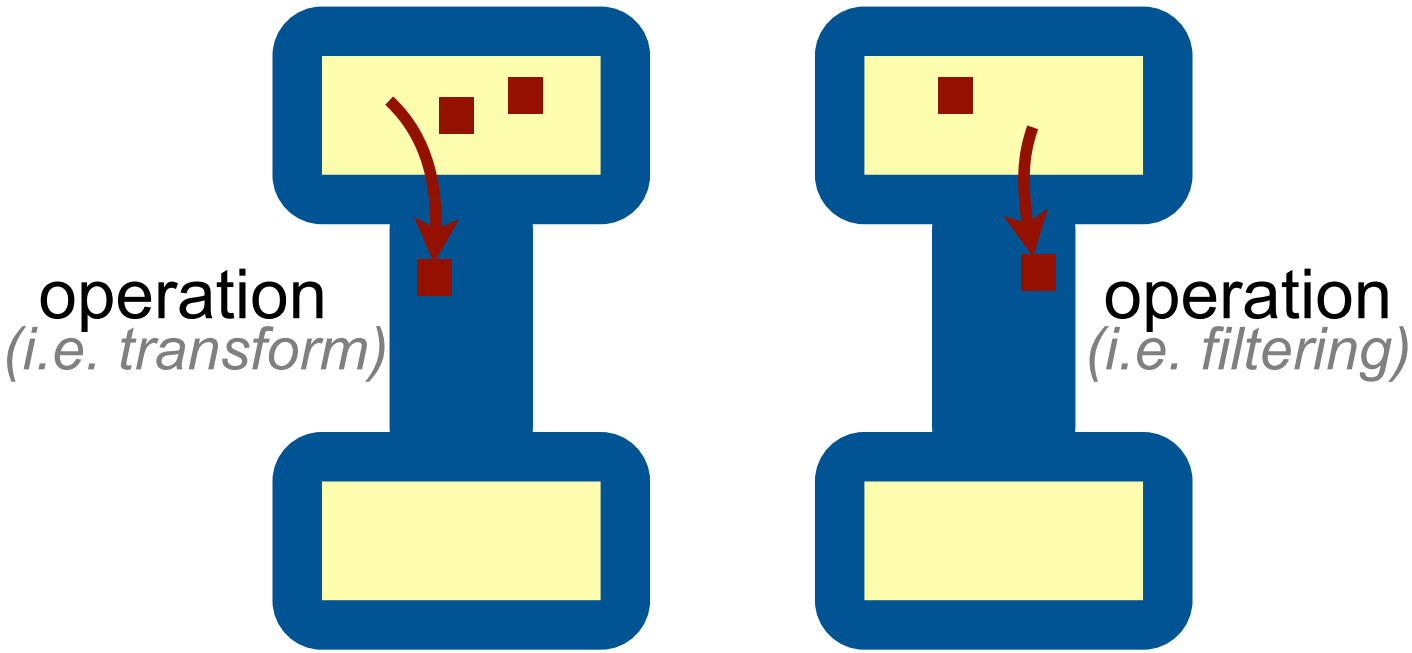
What is a FlowPool

Flow Graph



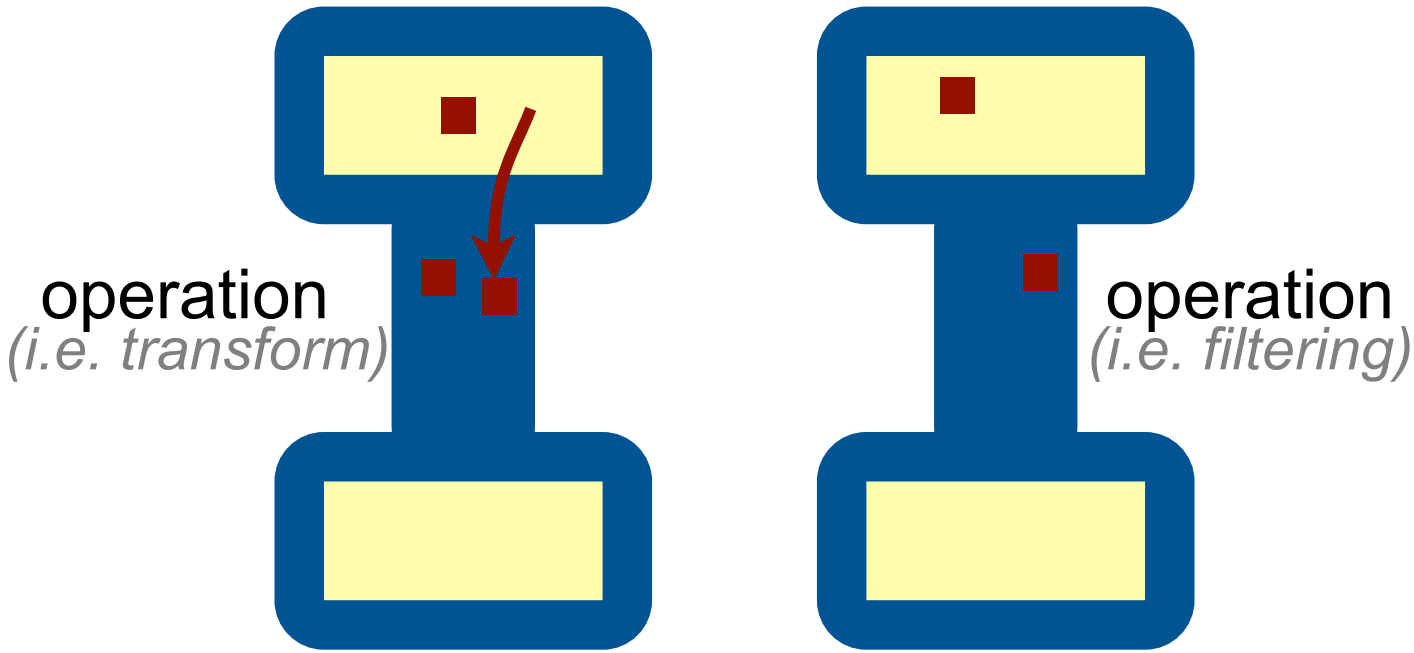
What is a FlowPool

Flow Graph



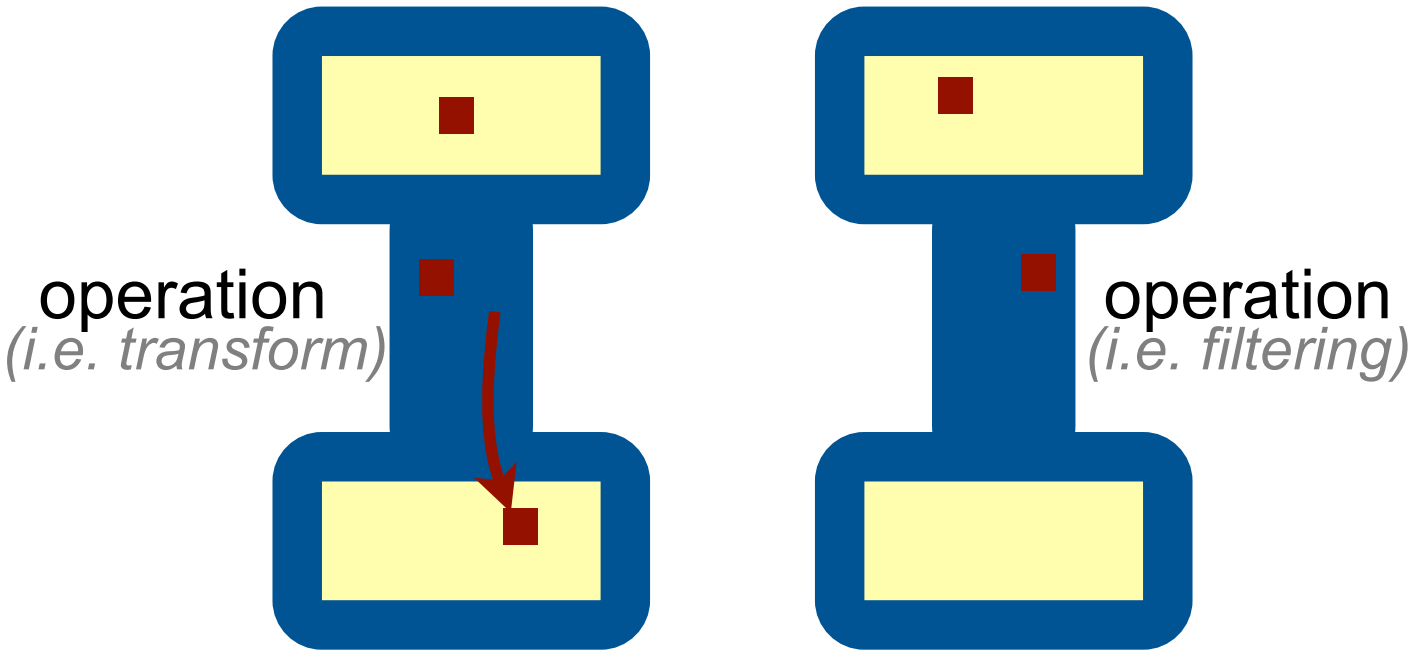
What is a FlowPool

Flow Graph



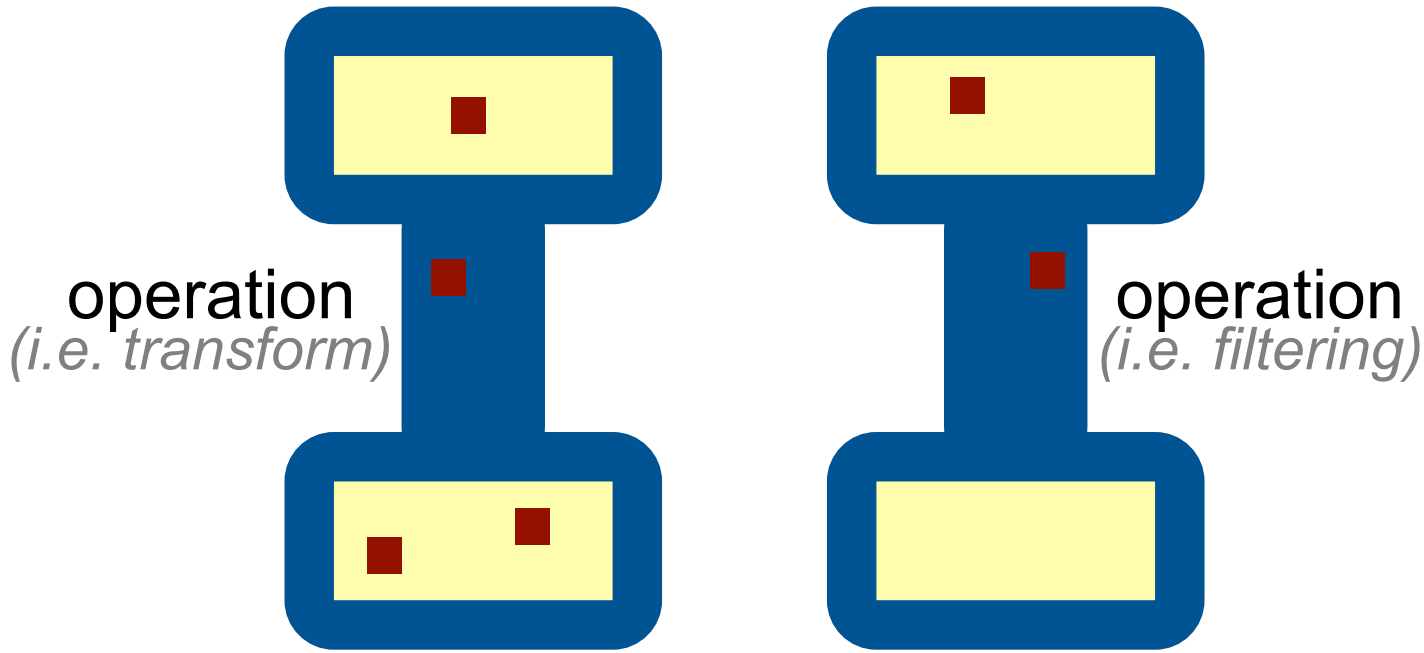
What is a FlowPool

Flow Graph



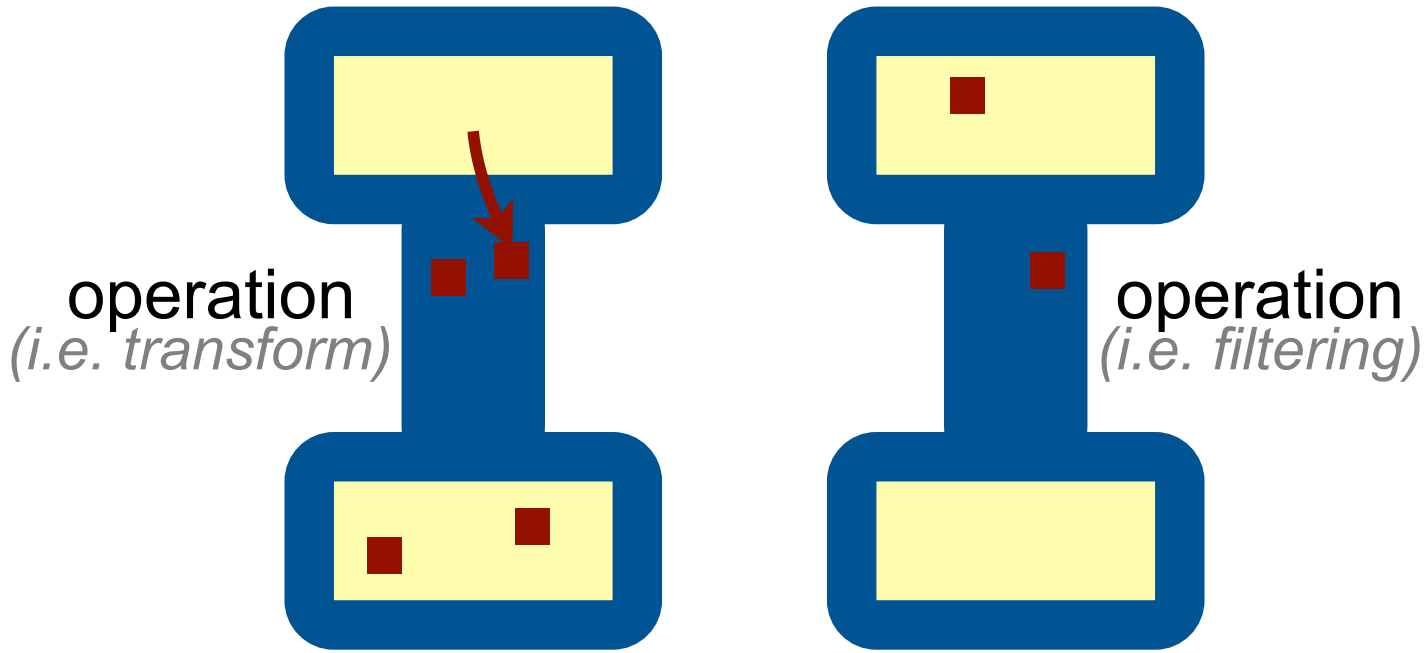
What is a FlowPool

Flow Graph



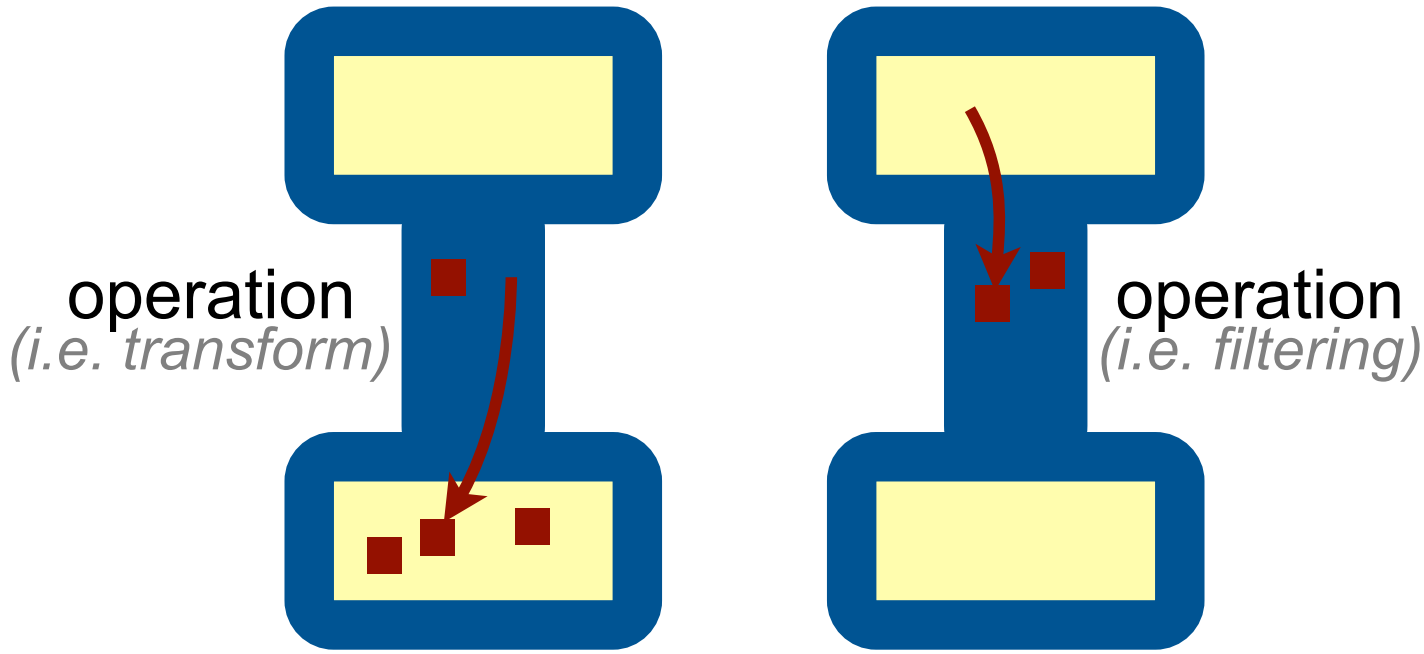
What is a FlowPool

Flow Graph



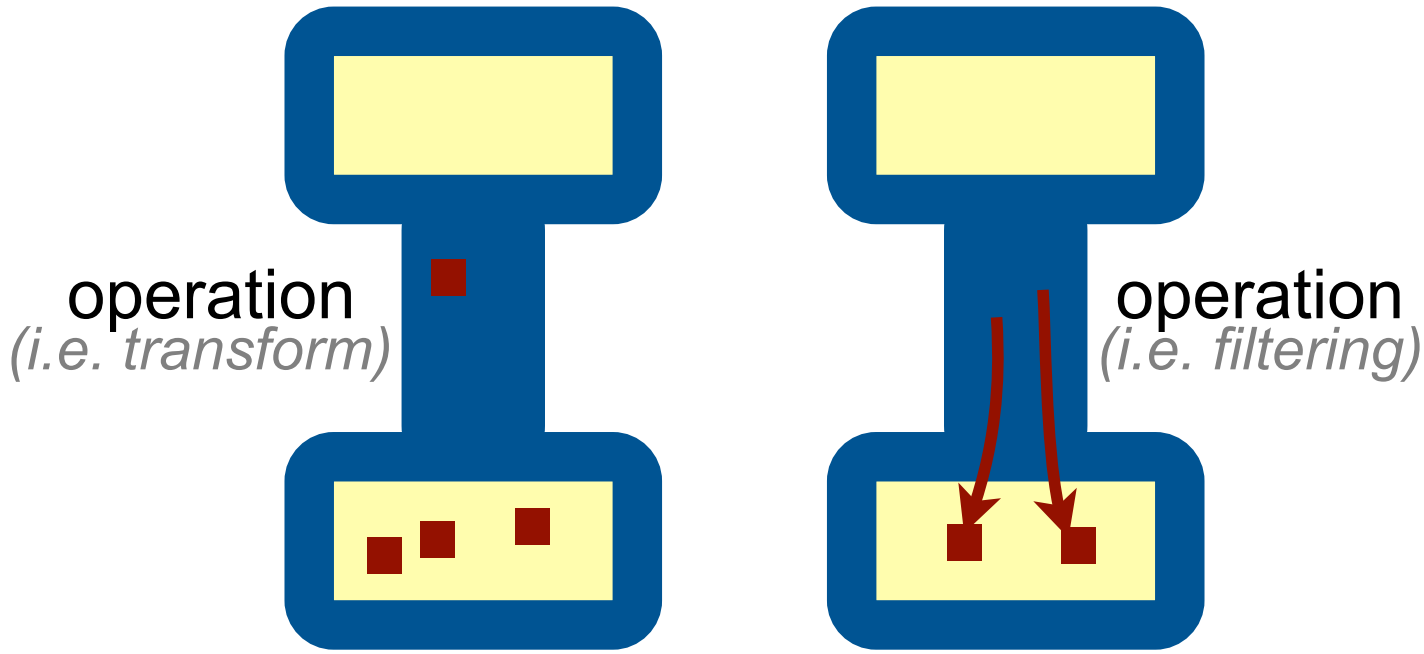
What is a FlowPool

Flow Graph



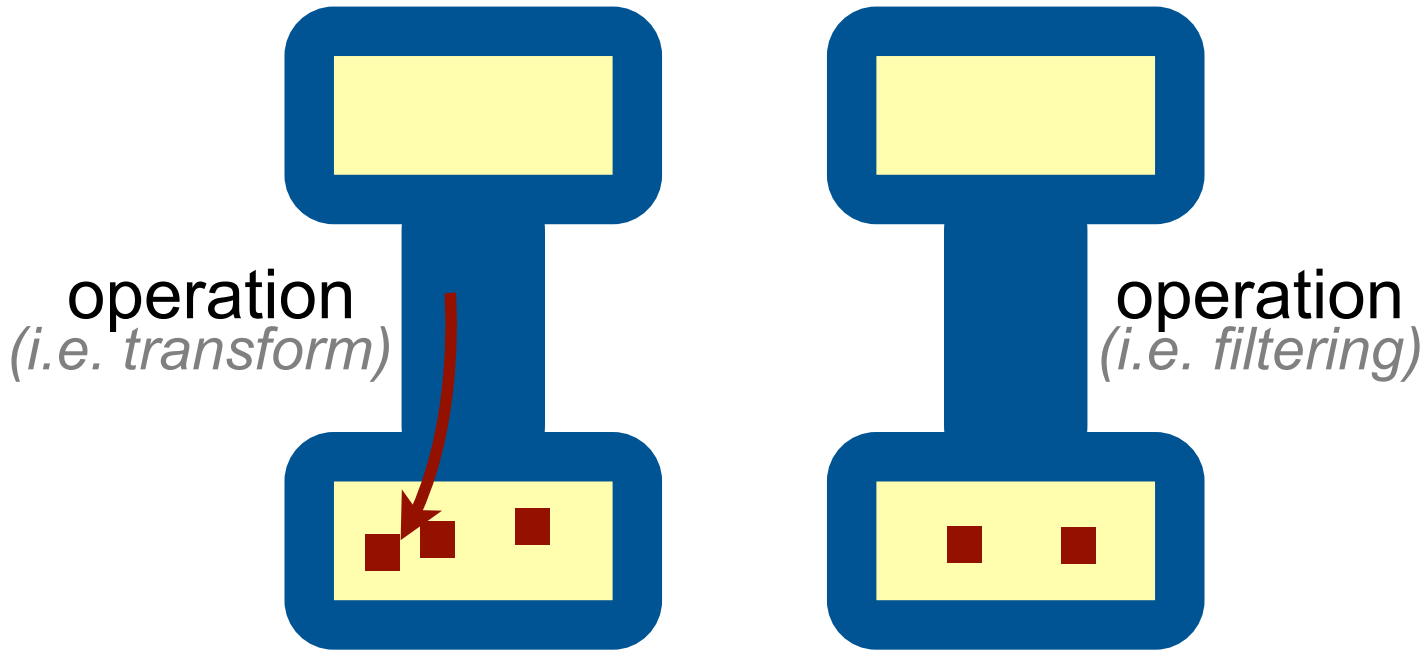
What is a FlowPool

Flow Graph



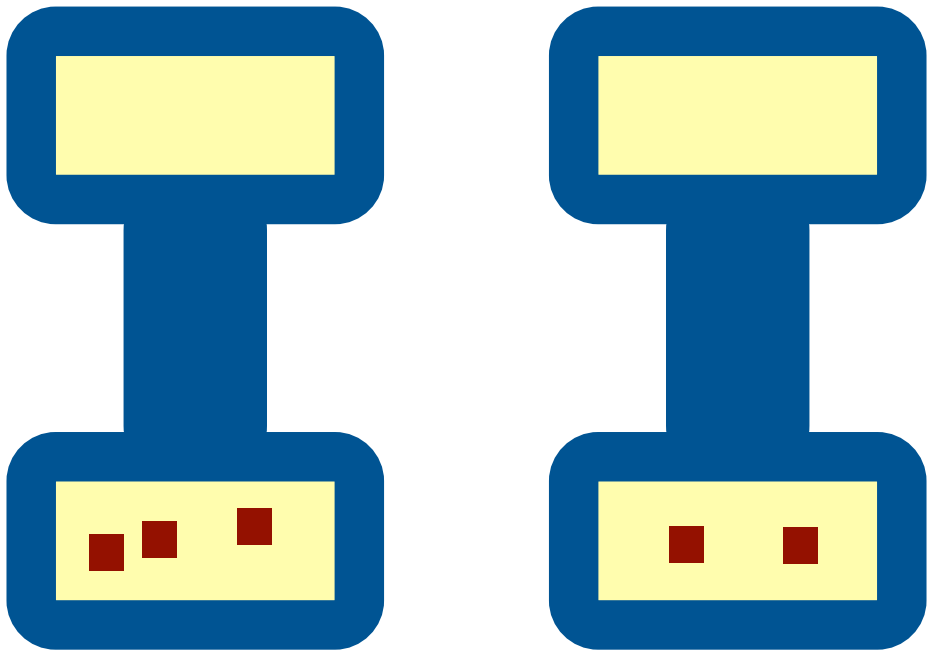
What is a FlowPool

Flow Graph



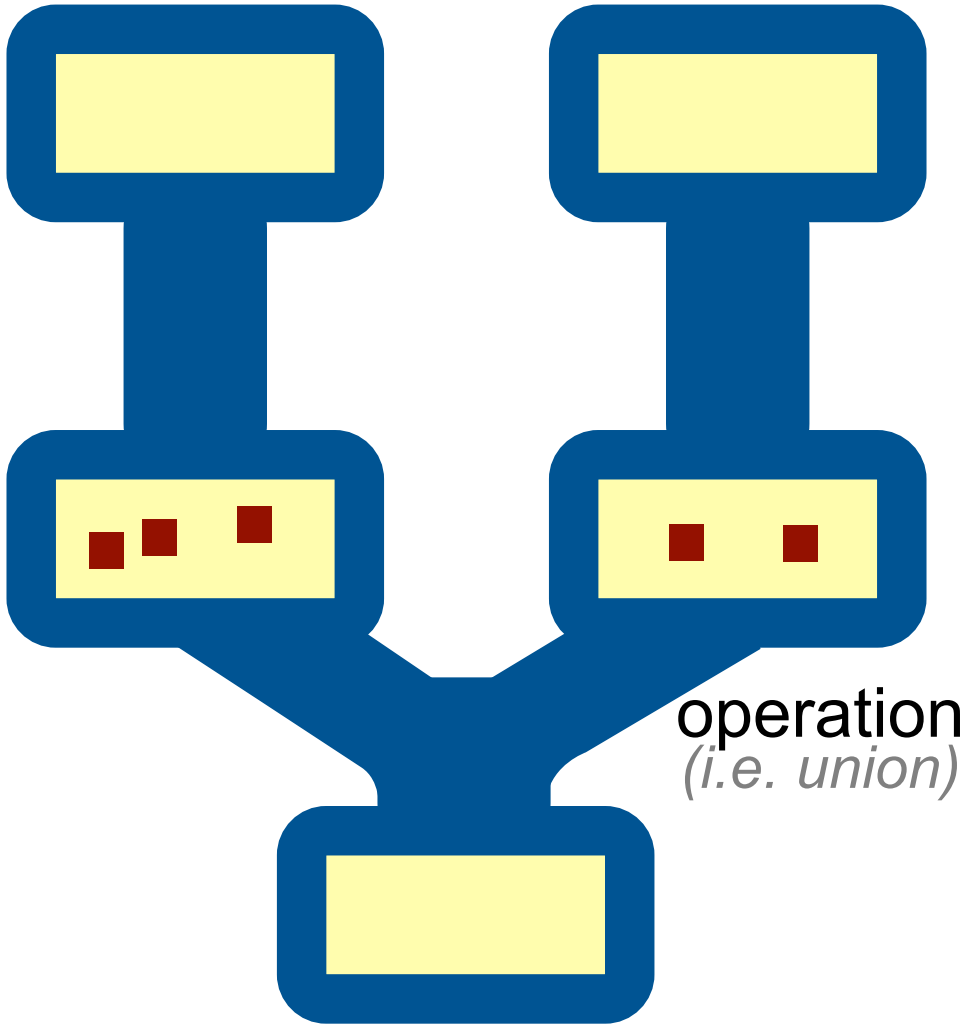
What is a FlowPool

Flow Graph



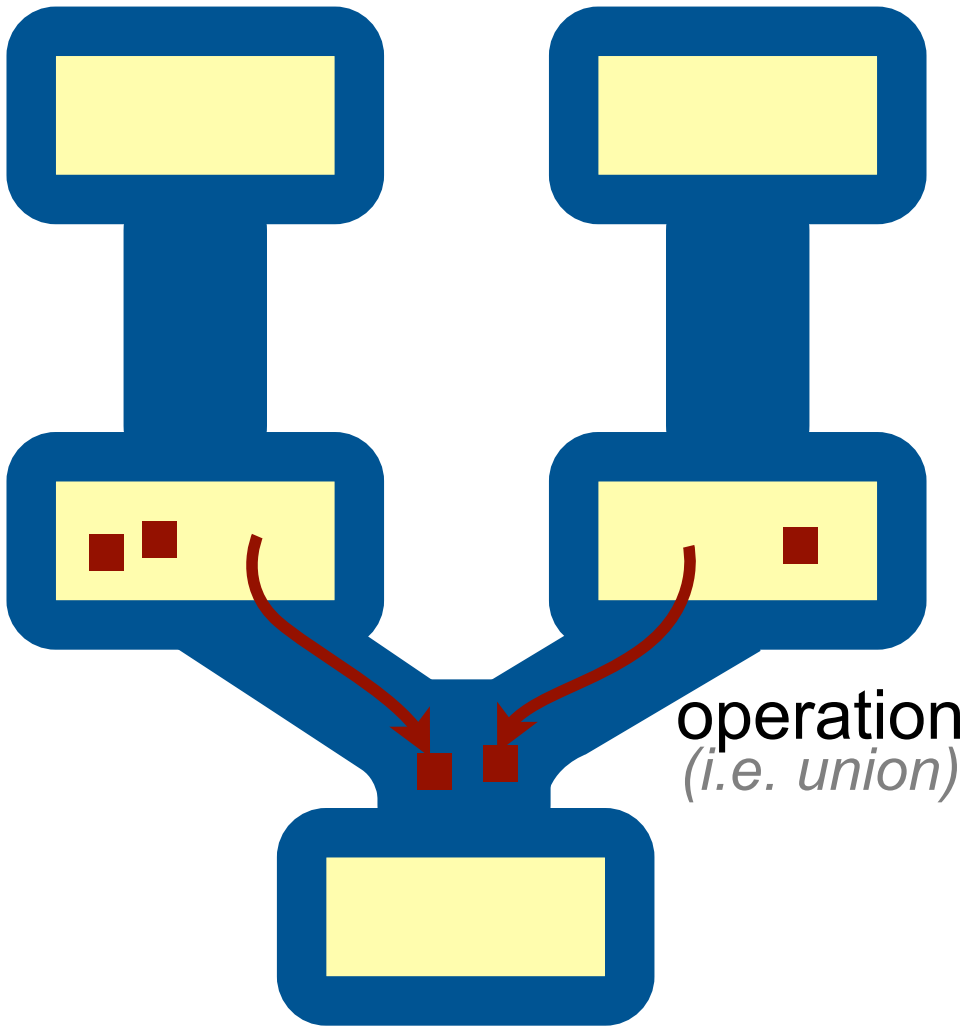
What is a FlowPool

Flow Graph



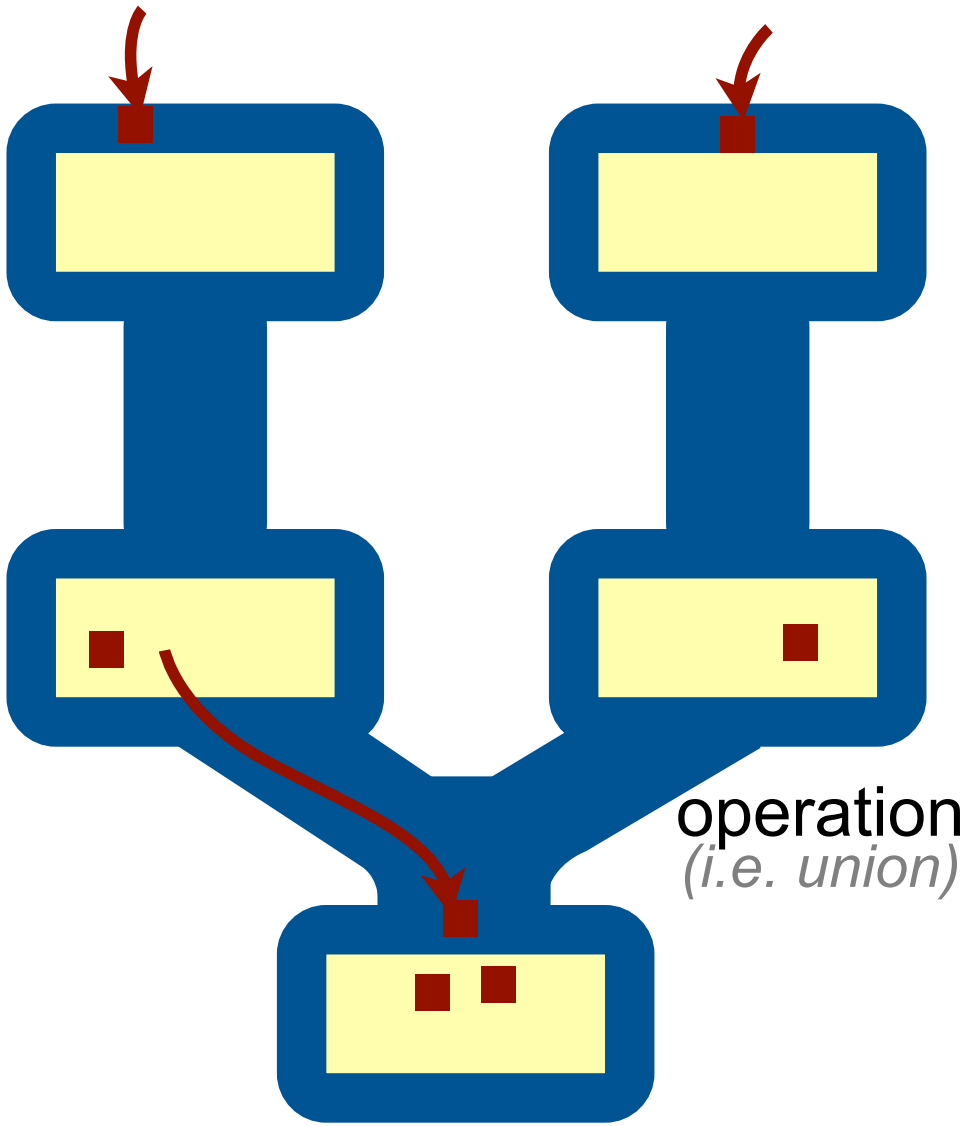
What is a FlowPool

Flow Graph



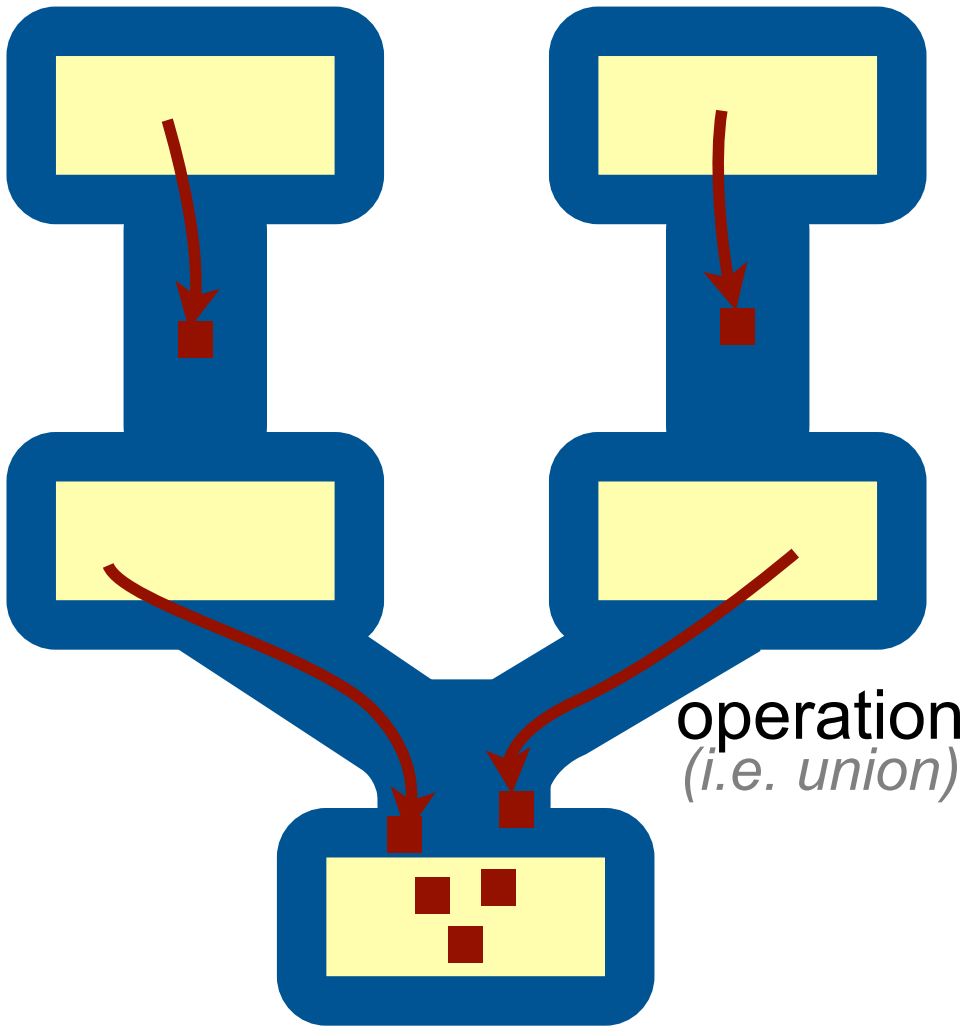
What is a FlowPool

Flow Graph



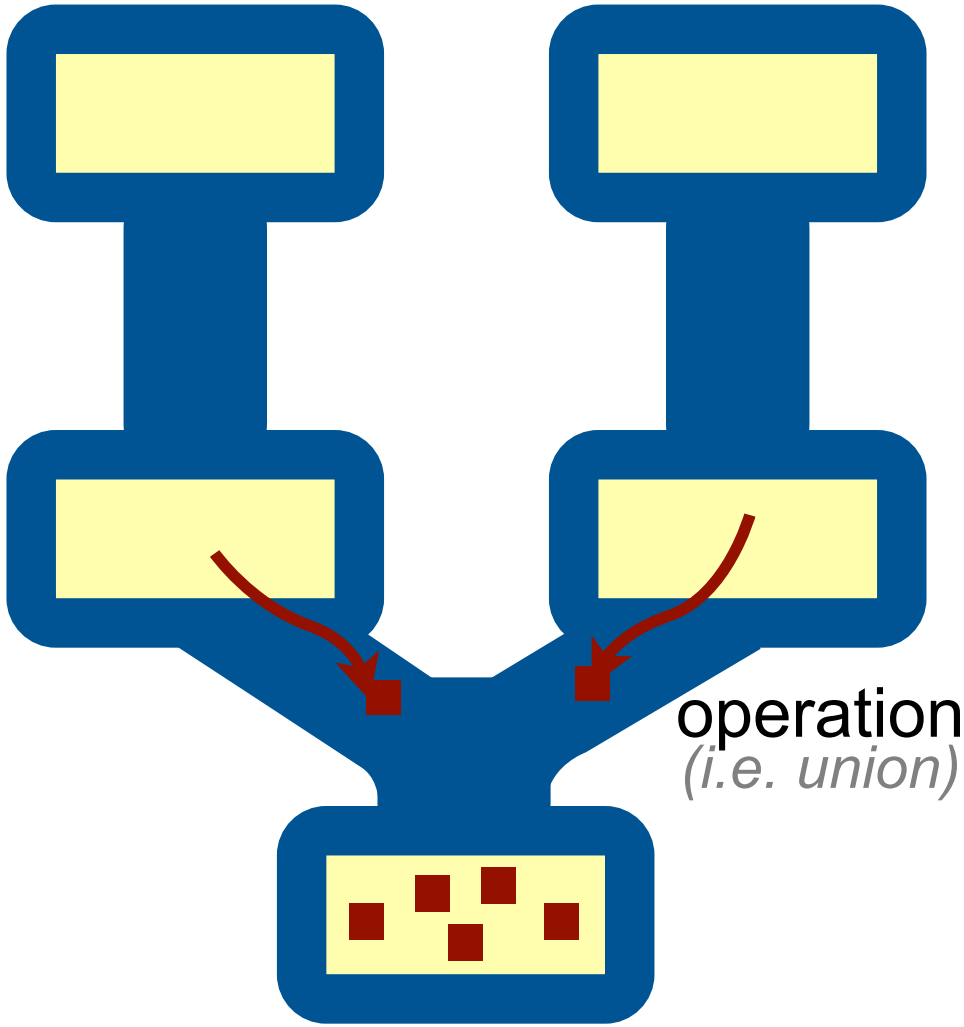
What is a FlowPool

Flow Graph



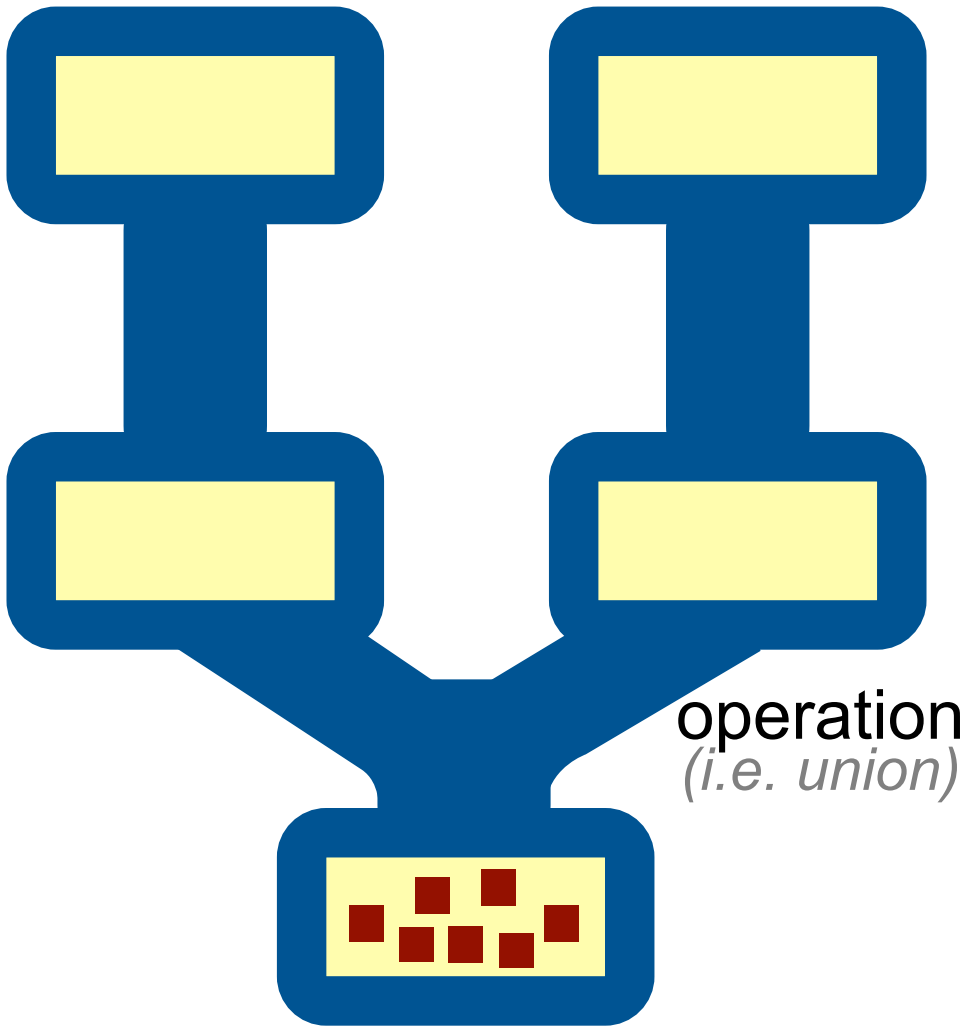
What is a FlowPool

Flow Graph



What is a FlowPool

Flow Graph



What is a FlowPool

Big Picture

Pool: concurrent collections abstraction for Scala

FlowPool Properties

- Pool semantics
- Asynchronous
- Deterministic
- Lock-Free
- Garbage collection of unused elements

Pool semantics

- Unordered
- Multiple occurrences
- Insertion (<<)
- Traversal (`foreach`)

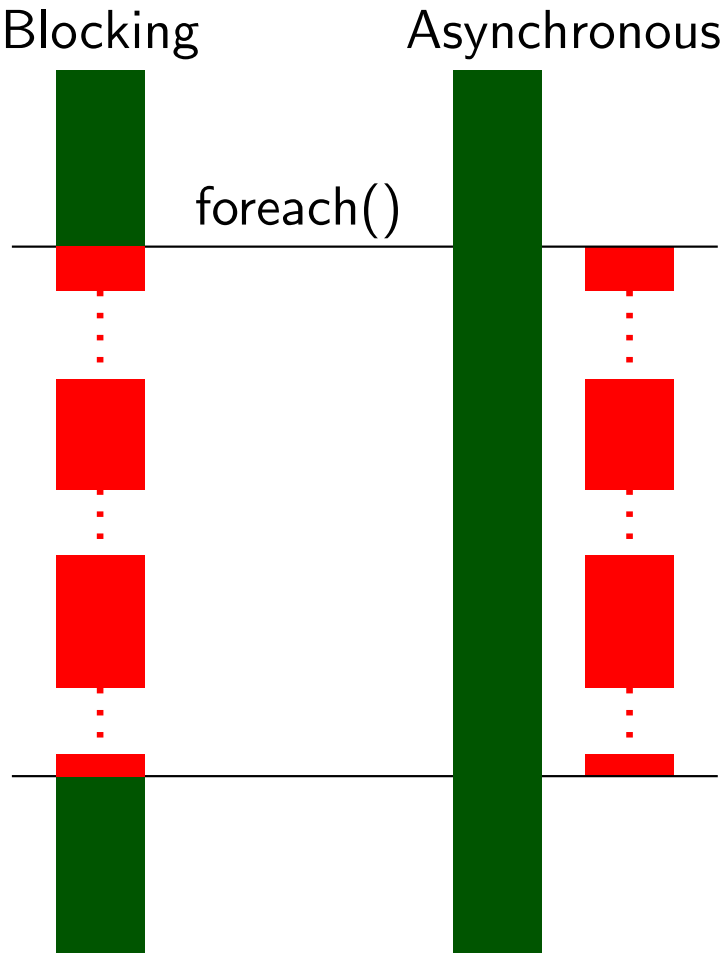
What is a FlowPool

Big Picture

Pool: concurrent collections abstraction for Scala

FlowPool Properties

- Pool semantics
- Asynchronous
- Deterministic
- Lock-Free
- Garbage collection of unused elements



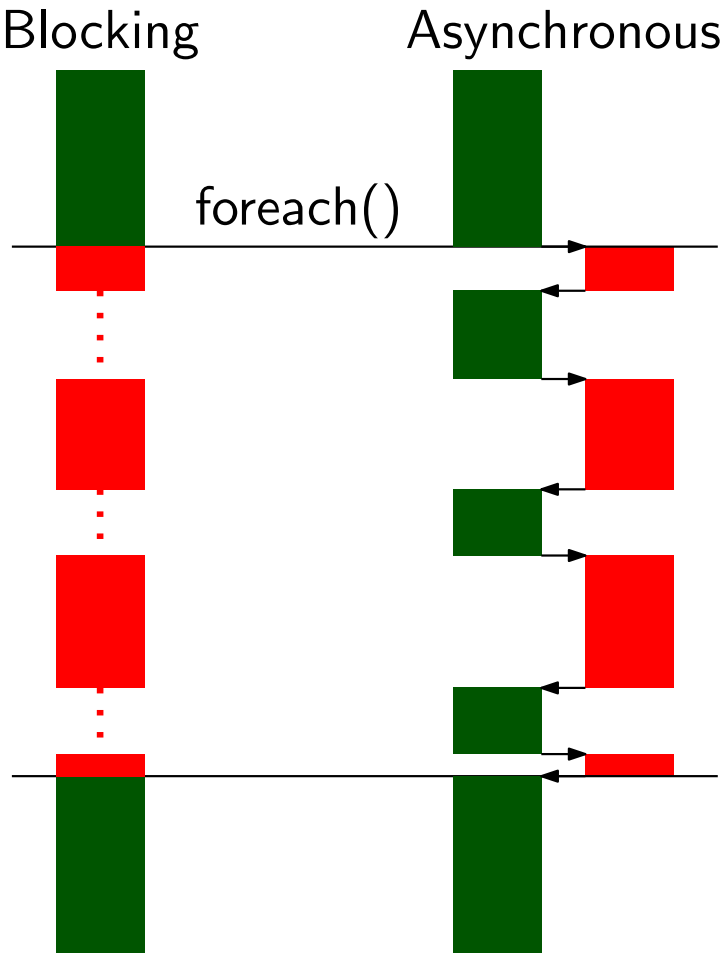
What is a FlowPool

Big Picture

Pool: concurrent collections abstraction for Scala

FlowPool Properties

- Pool semantics
- Asynchronous
- Deterministic
- Lock-Free
- Garbage collection of unused elements



What is a FlowPool

Big Picture

Pool: concurrent collections abstraction for Scala

FlowPool Properties

- Pool semantics
- Asynchronous
- Deterministic
- Lock-Free
- Garbage collection of unused elements

Determinism

Every execution of a given program with given input eventually

- Always reaches same state

  or

- Always fails

What is a FlowPool

Big Picture

Pool: concurrent collections abstraction for Scala

FlowPool Properties

- Pool semantics
- Asynchronous
- Deterministic
- Lock-Free
- Garbage collection of unused elements

Lock-Freedom

At least one operation will complete in a finite number of steps.

What is a FlowPool

Big Picture

Pool: concurrent collections abstraction for Scala

FlowPool Properties

- Pool semantics
- Asynchronous
- Deterministic
- Lock-Free
- Garbage collection of unused elements

What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

```
Thread 1:
  p << x
```

```
Thread 2:
  p.seal()
```

What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

```
Thread 1:
  p << x
```

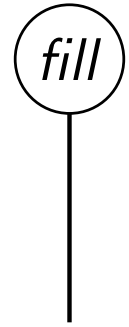
```
Thread 2:
  p.seal(1)
```

What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

```
def fill(n: Int, el: T) {
  p = new FlowPool[T]
  for (i = 1 to n) { p << el }
  p.seal(n)
  return p
}
```

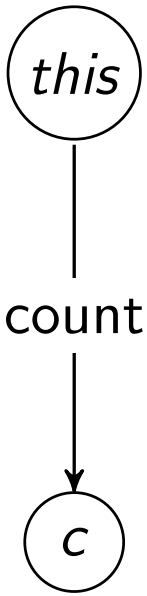


What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

```
def count(cond) =
  this.aggregate(0)(_ + _) { x =>
    if (cond(x)) cnt + 1
    else cnt
  }
```



What is a FlowPool

Programming Model – Elementary Operations

- Insertion (<<)
- Traversal (`foreach`)
- Sealing (`seal(n)`)
- Aggregation (`aggregate`)

```
def foreach[U](f: T => U): Future[Unit] = {
  aggregate(())((_, _) => ()) {
    (acc, x) => f(x); ()
  }
}
```

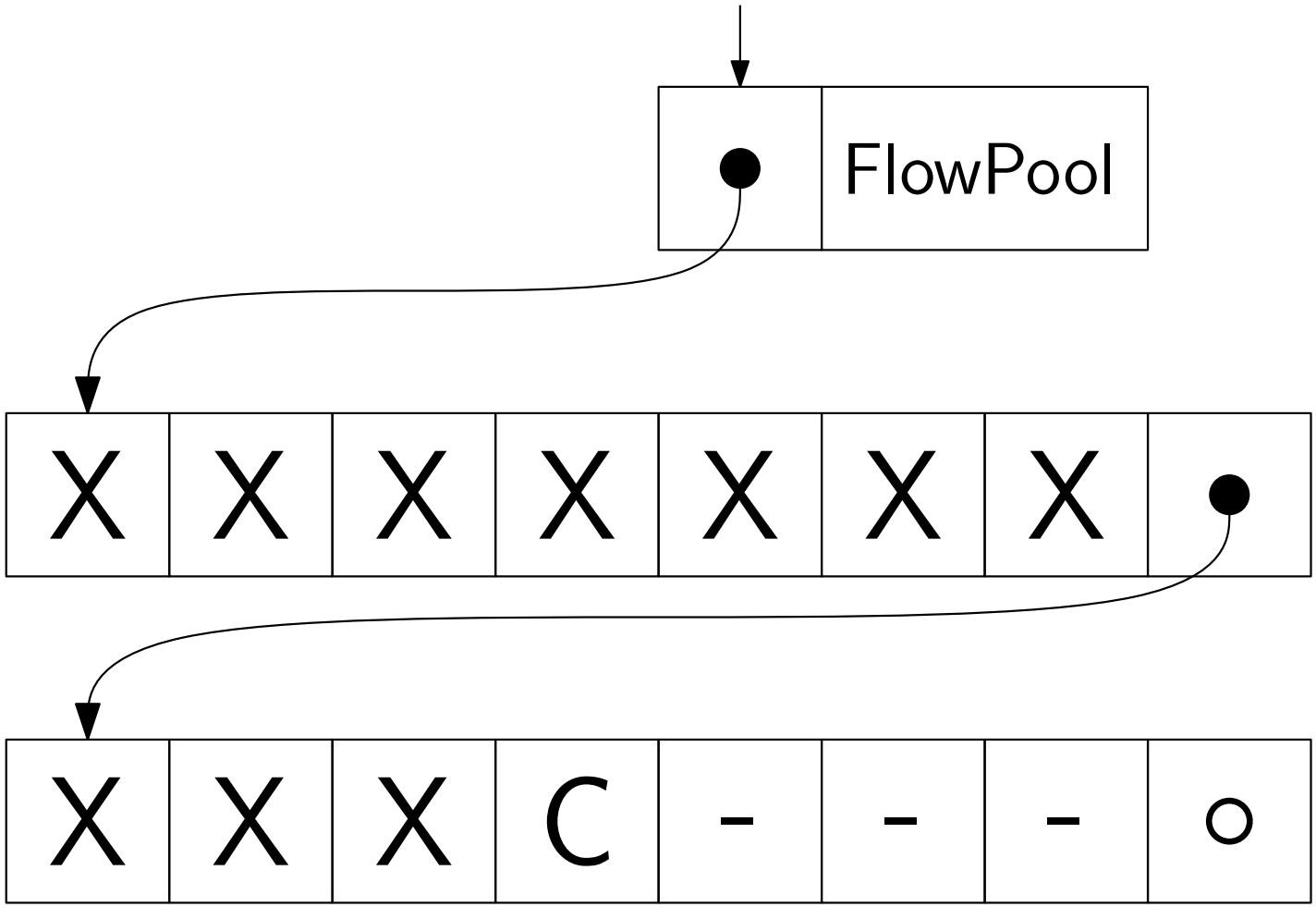
What is a FlowPool

Programming Model – Higher Level Operations

- Union (++)
- `filter`
- Transformation (`map`, `flatMap`)
- Reduction (`fold`)
- Generation (`fill`, `iterate`)

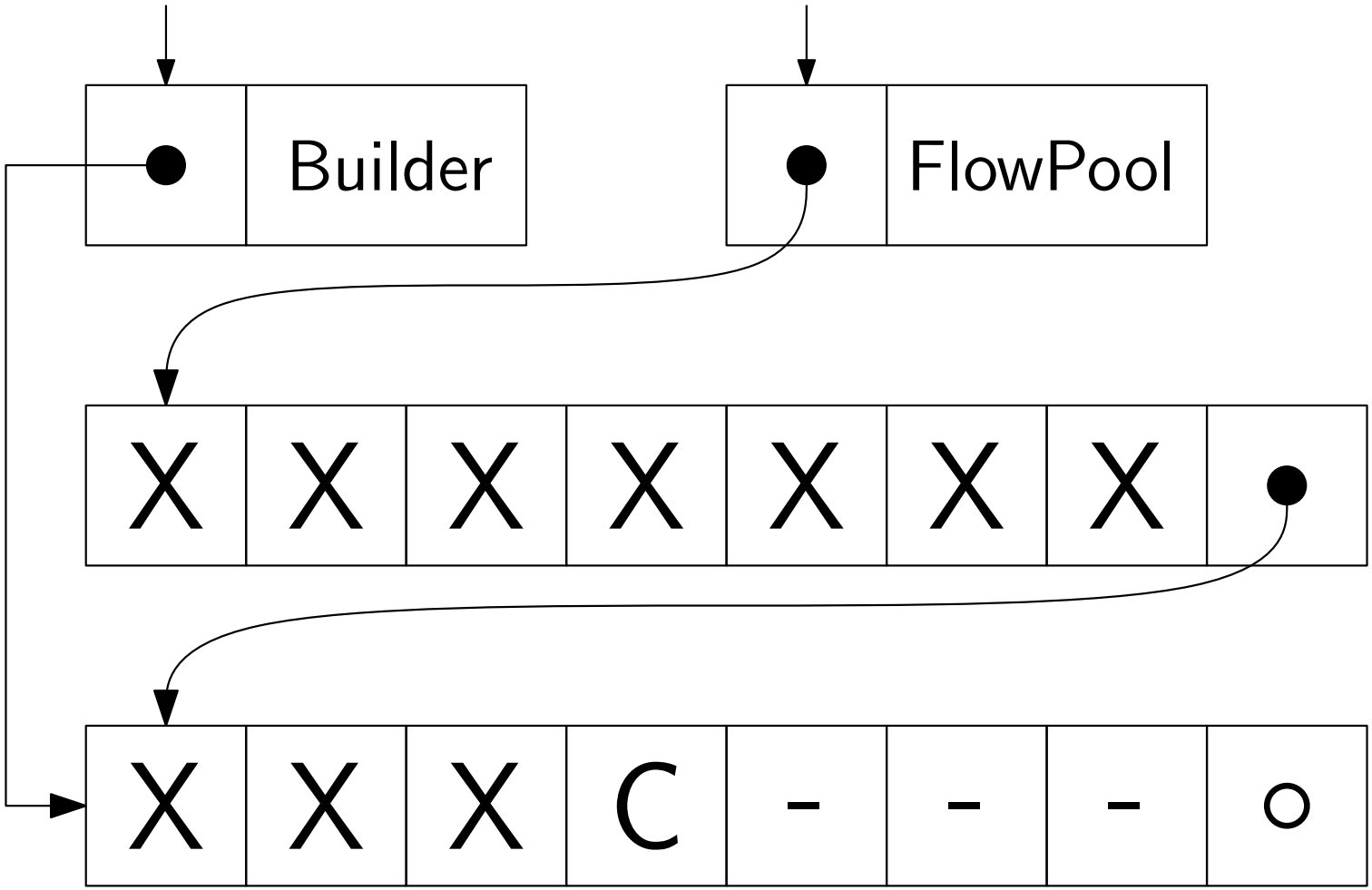
Implementation

Basic Structure / Garbage Collection of Unneeded Elements



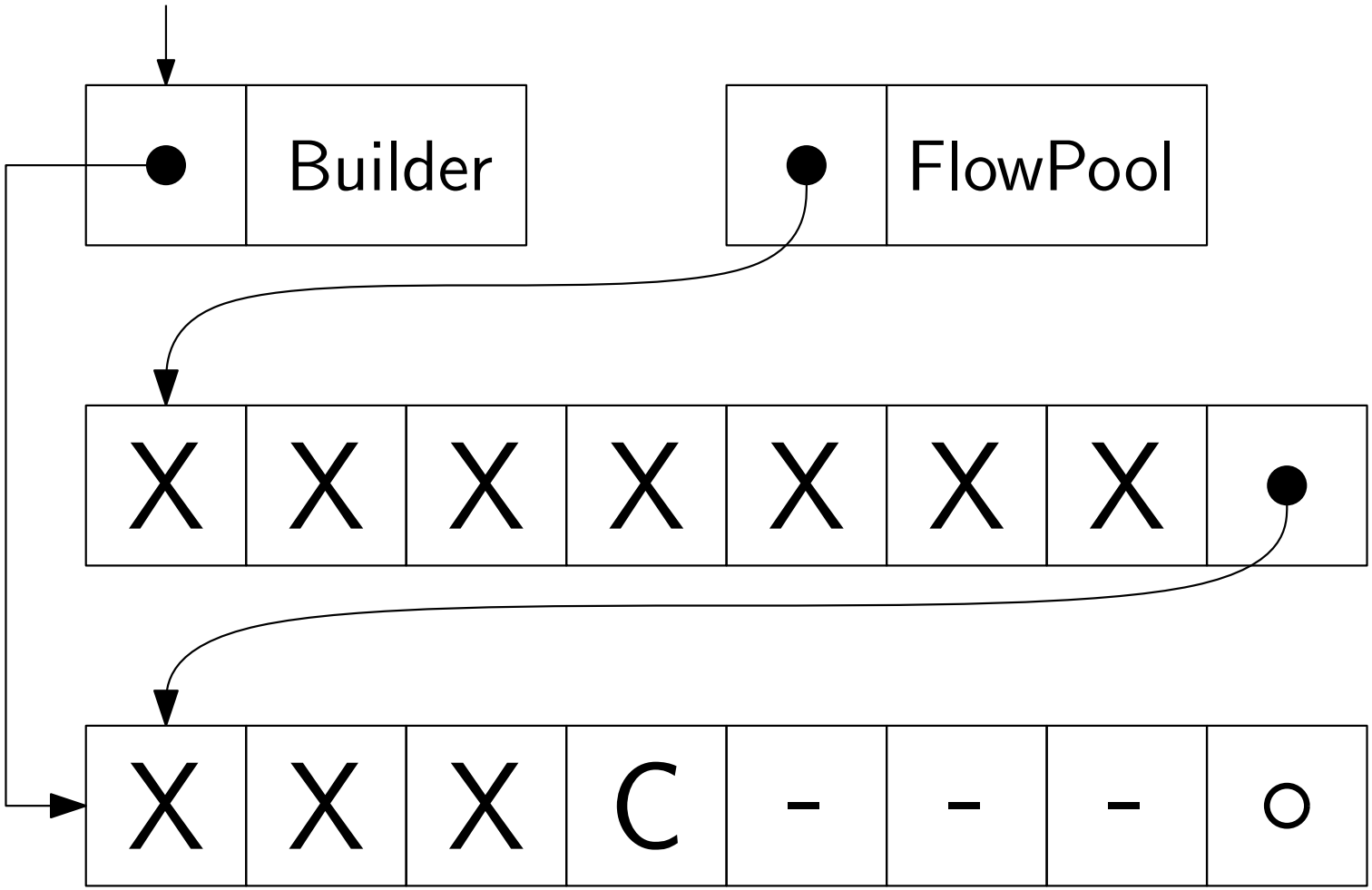
Implementation

Basic Structure / Garbage Collection of Unneeded Elements



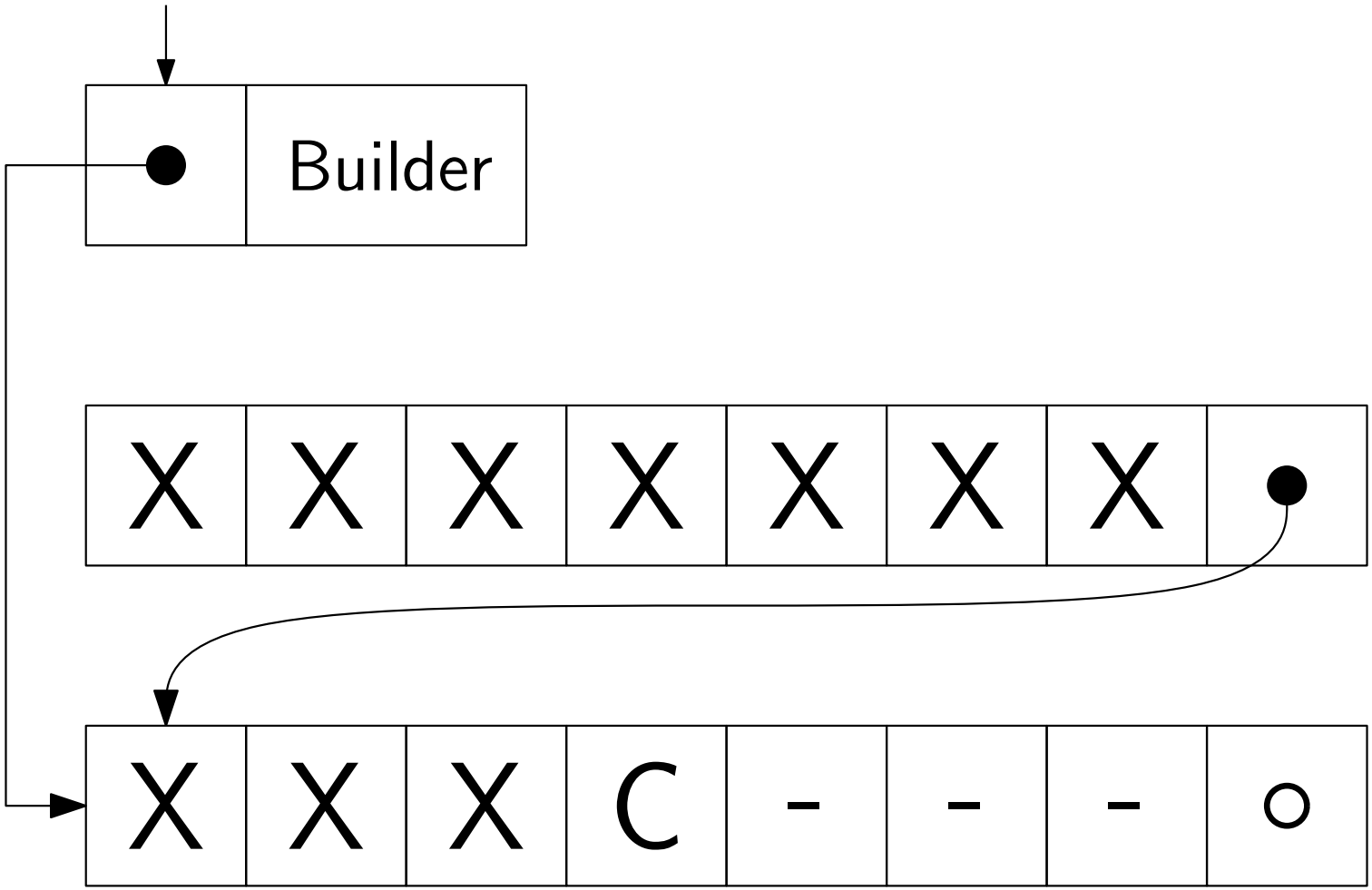
Implementation

Basic Structure / Garbage Collection of Unneeded Elements



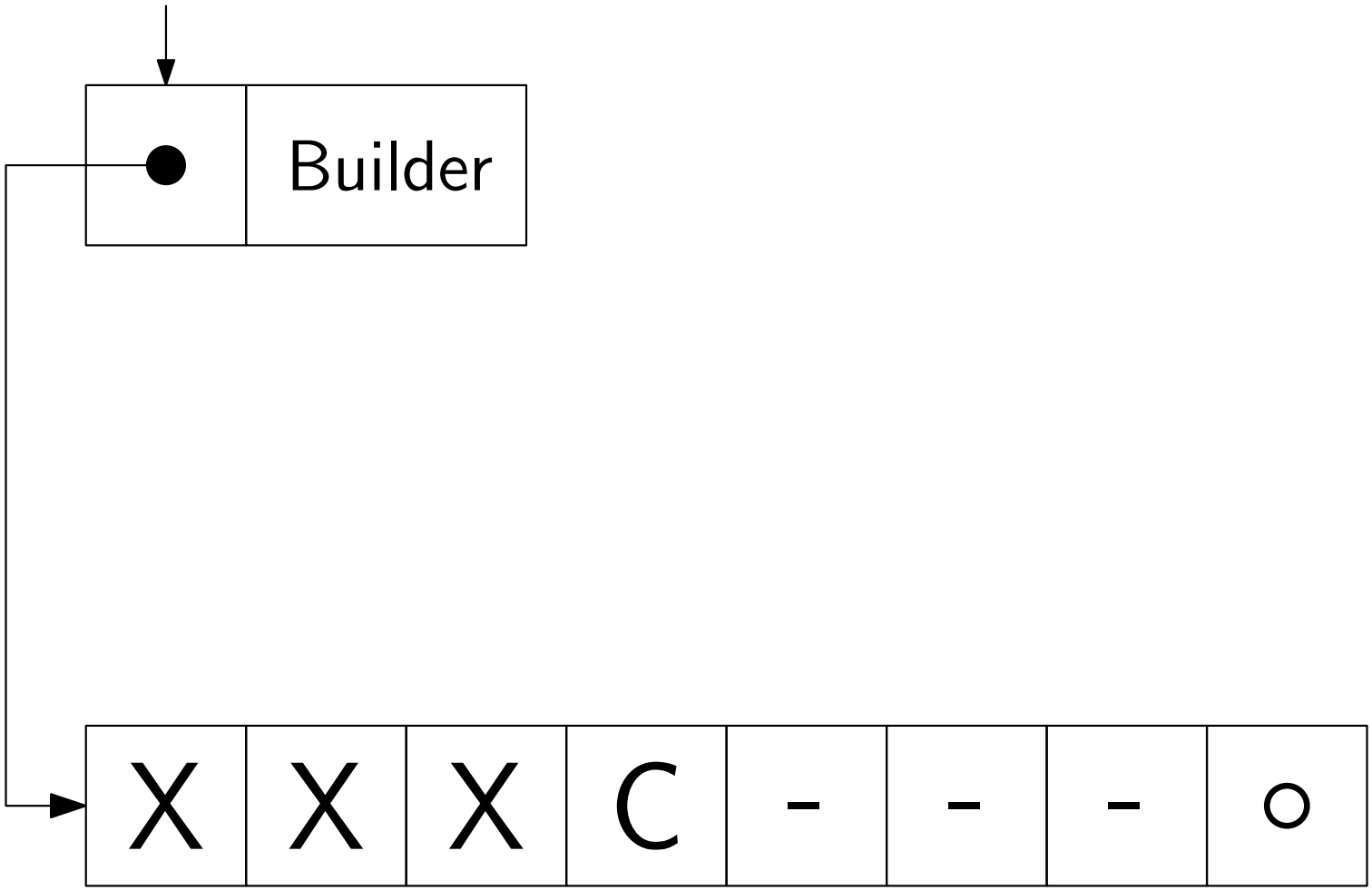
Implementation

Basic Structure / Garbage Collection of Unneeded Elements



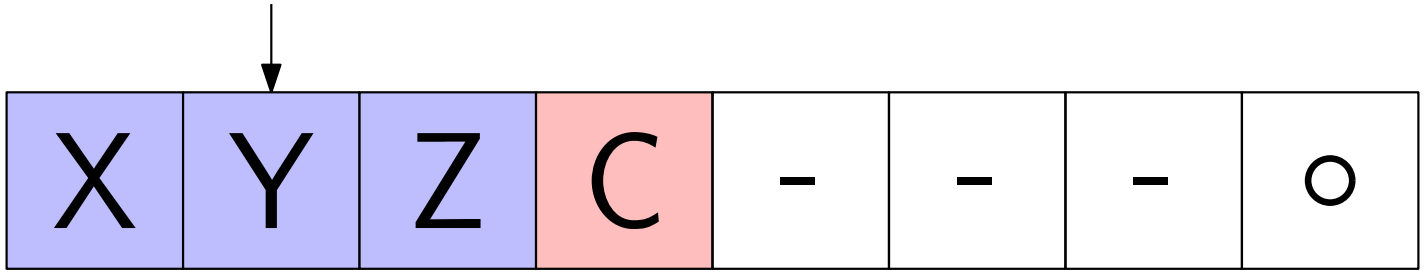
Implementation

Basic Structure / Garbage Collection of Unneeded Elements



Implementation

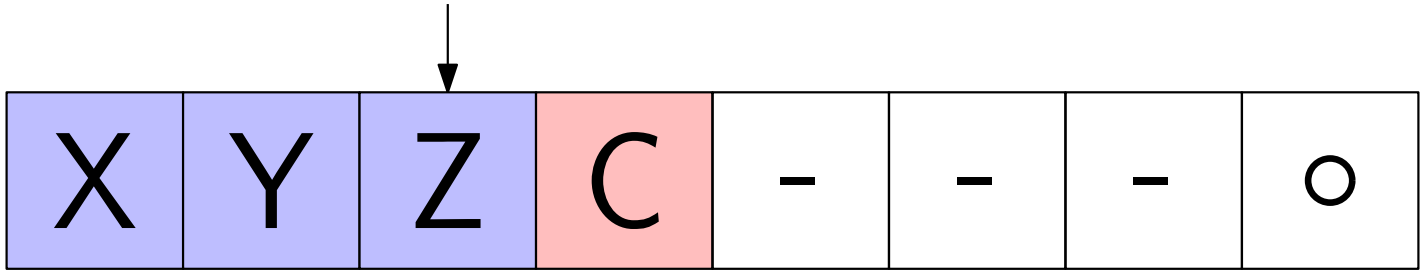
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

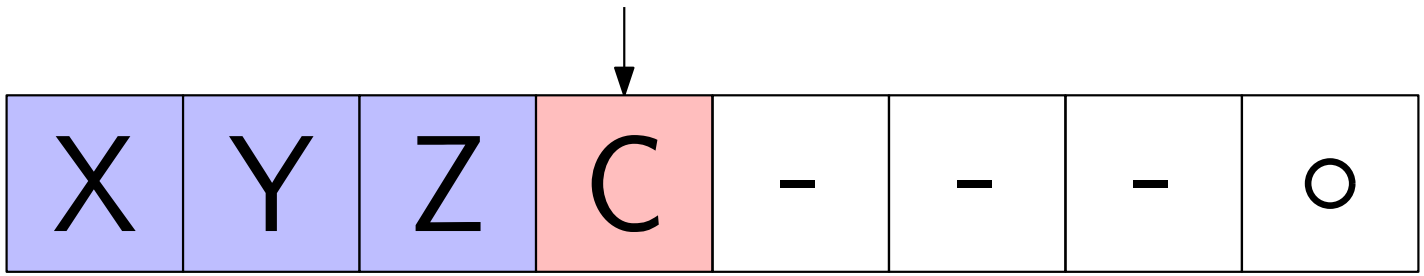
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

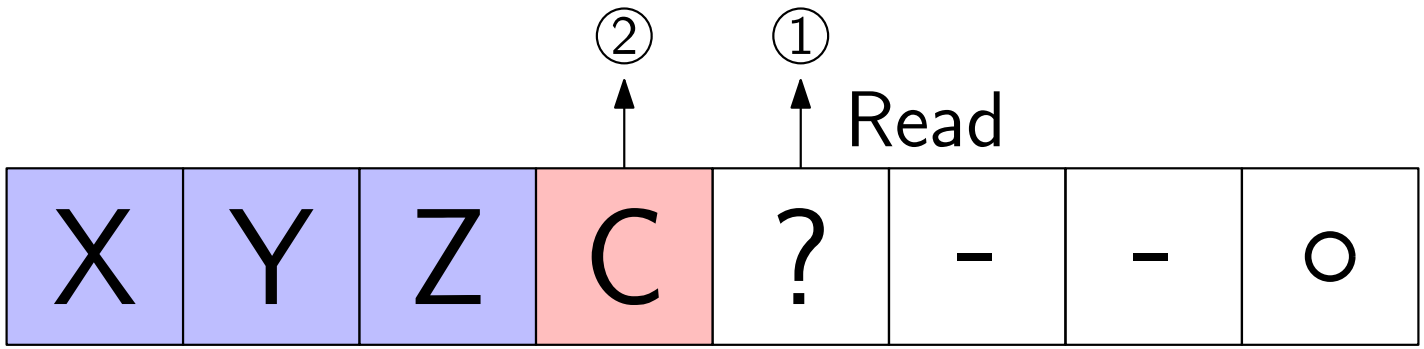
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

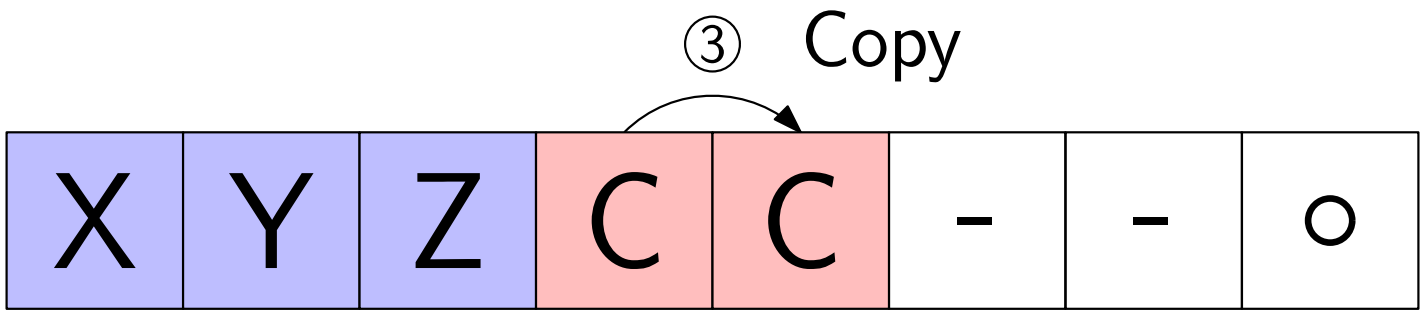
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

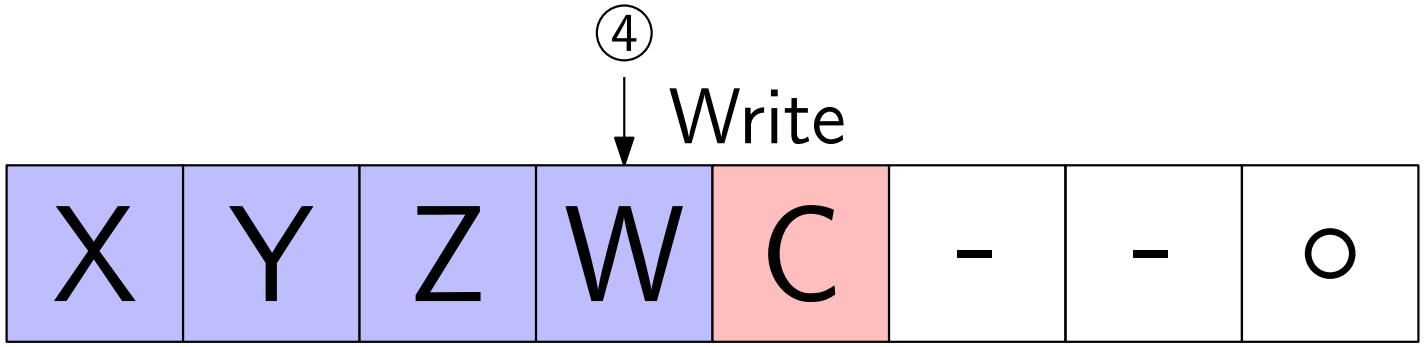
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

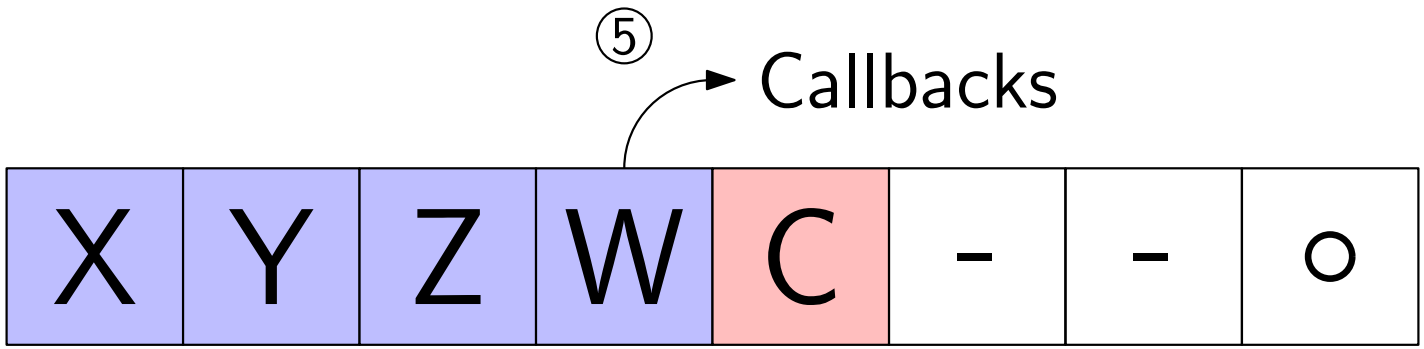
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

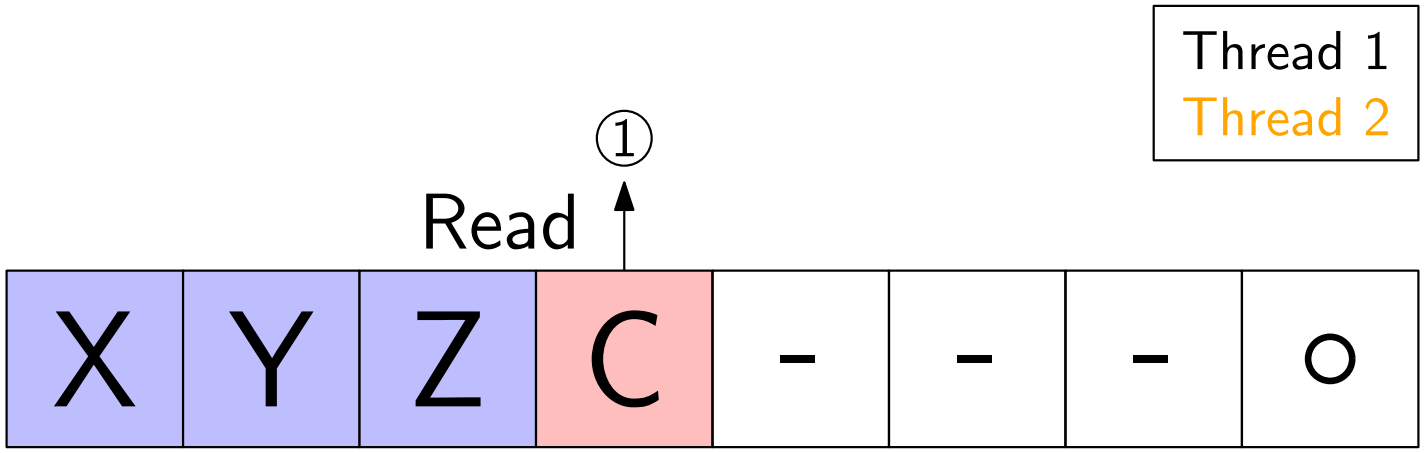
Insert `<<(x: T)`



1. `next = block(i+1)`
2. `curo = block(i)`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

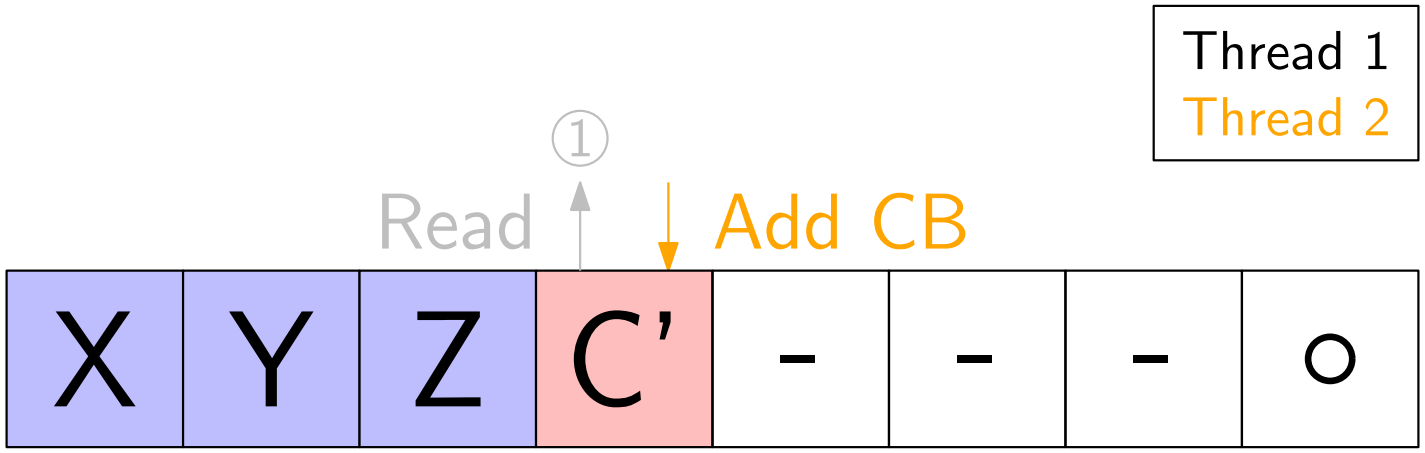
Wrong Insert `<<(x: T)`



1. `curo = block(i)` `(= C)`
2. `next = block(i+1)` `(= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

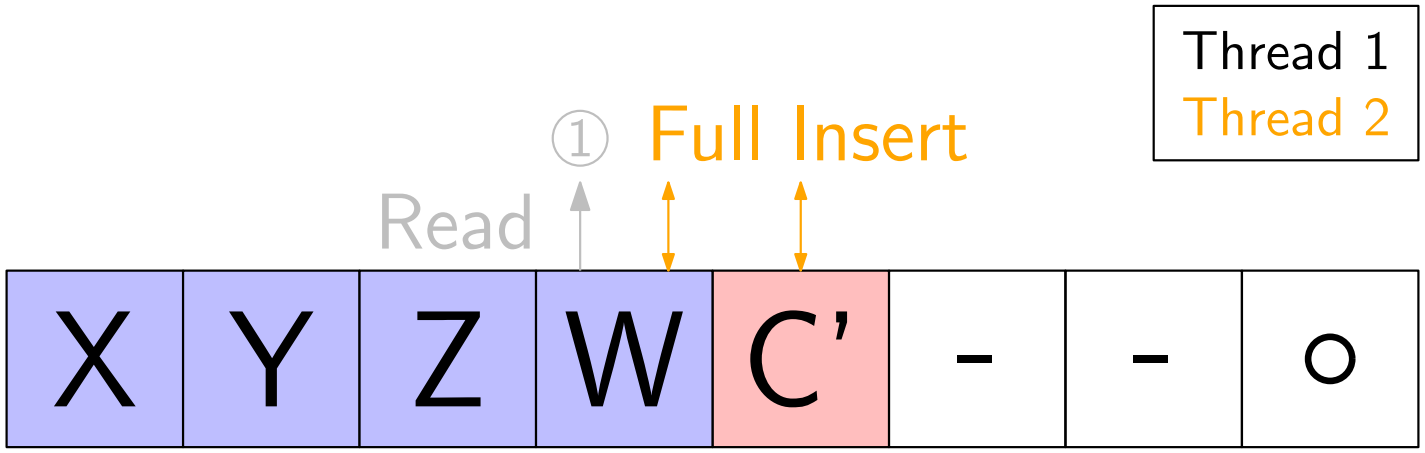
Wrong Insert `<<(x: T)`



1. `curo = block(i)` `(= C)`
2. `next = block(i+1)` `(= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

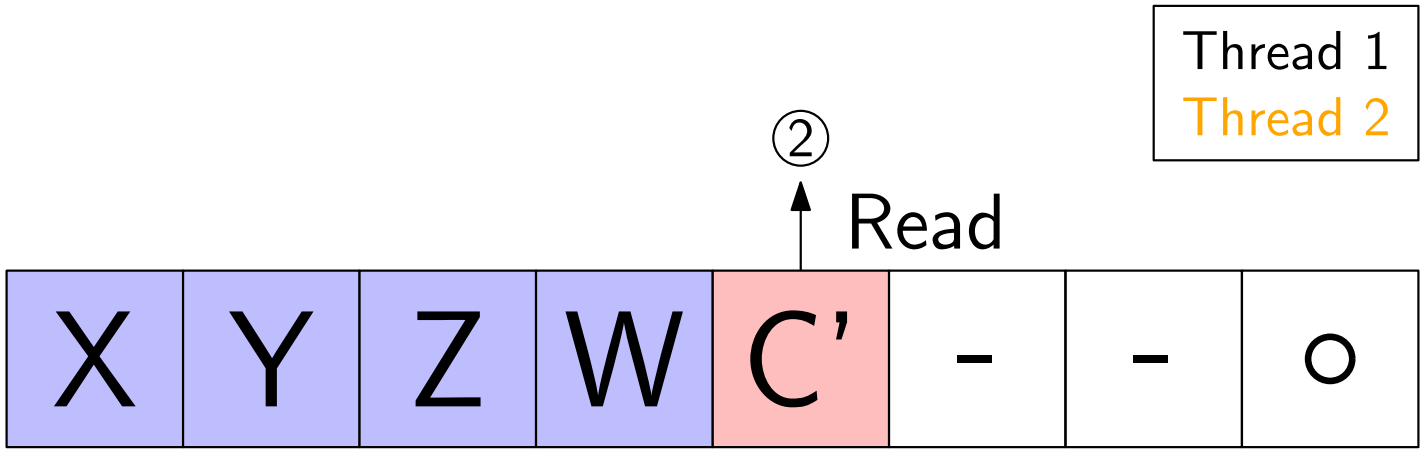
Wrong Insert `<<(x: T)`



1. `curo = block(i)` `(= C)`
2. `next = block(i+1)` `(= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

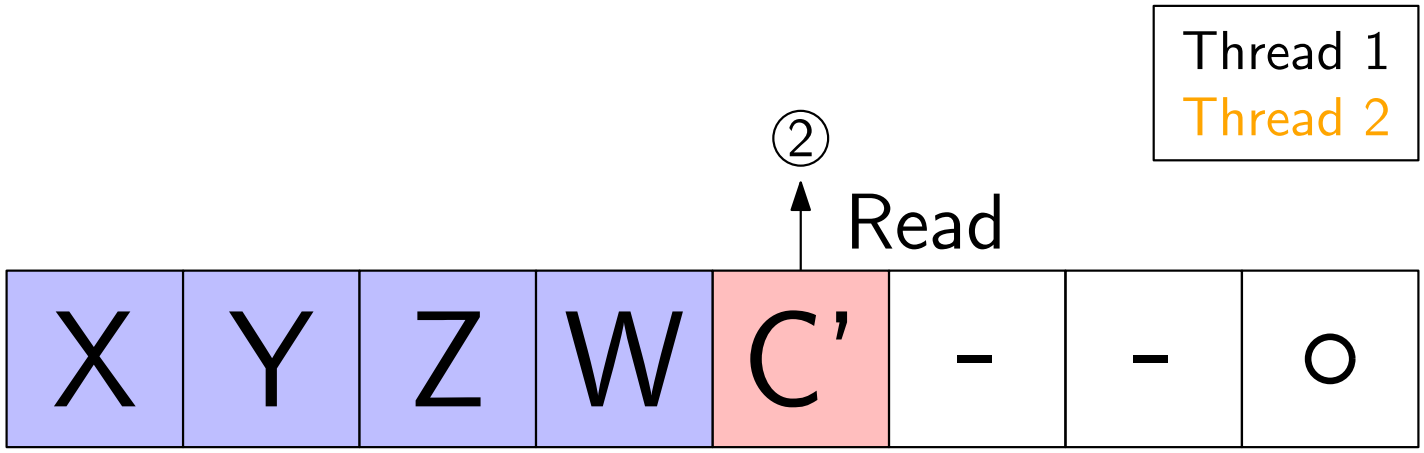
Wrong Insert `<<(x: T)`



1. `curo = block(i)` `(= C)`
2. `next = block(i+1)` `(= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

Wrong Insert `<<(x: T)`



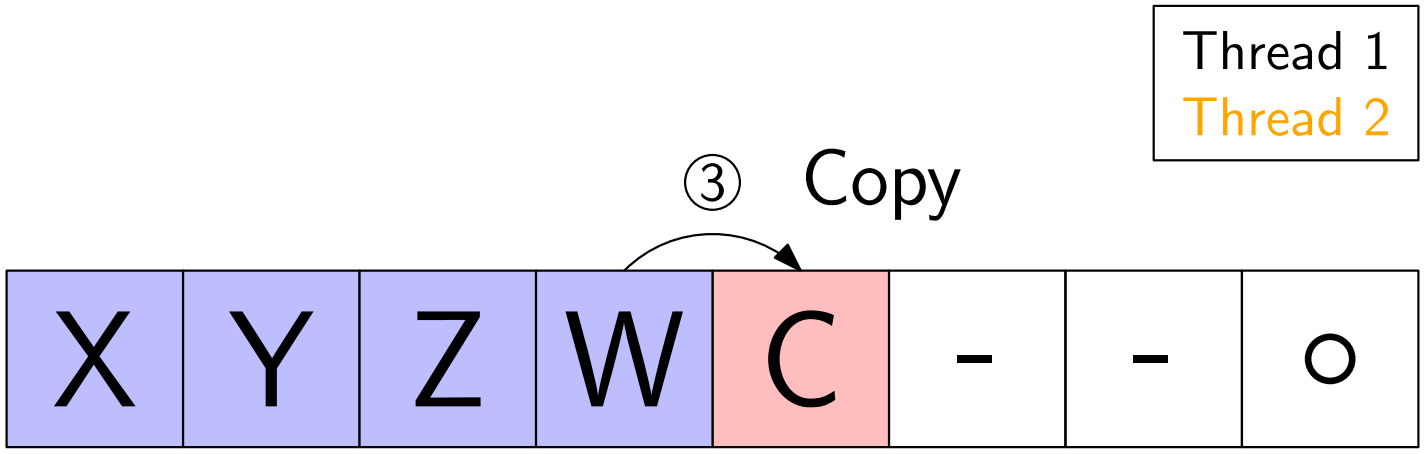
Observed State (inconsistent)

X	Y	Z	C	C'	-	-	○

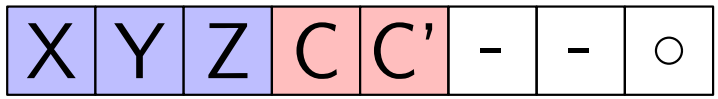
1. `curo = block(i)  (= C)`
2. `next = block(i+1)  (= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

Wrong Insert `<<(x: T)`



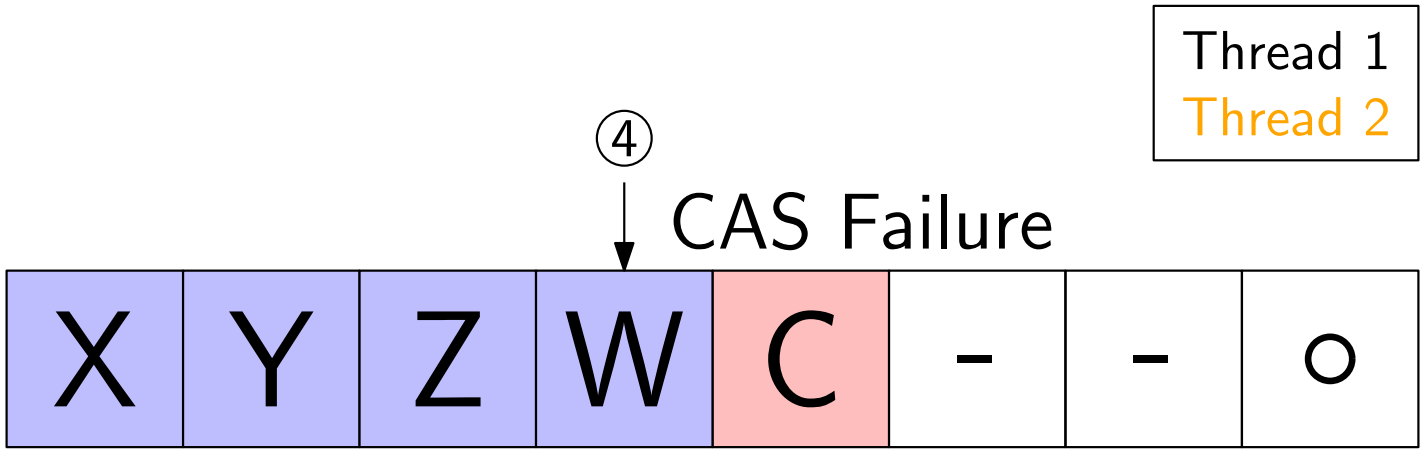
Observed State (inconsistent)



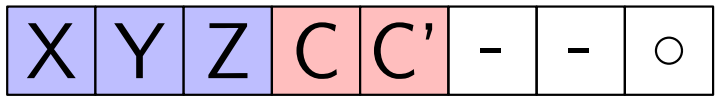
1. `curo = block(i)  (= C)`
2. `next = block(i+1)  (= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

Wrong Insert `<<(x: T)`



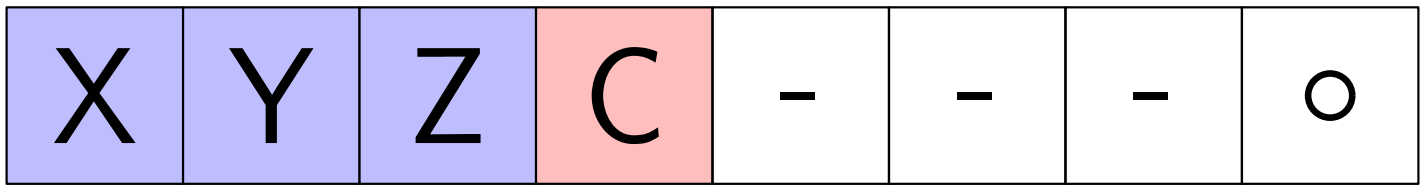
Observed State (inconsistent)



1. `curo = block(i)  (= C)`
2. `next = block(i+1)  (= C')`
3. `CAS(block(i+1), next, curo)`
4. `CAS(block(i), curo, W)`
5. `invokeCallbacks(W, curo)`

Implementation

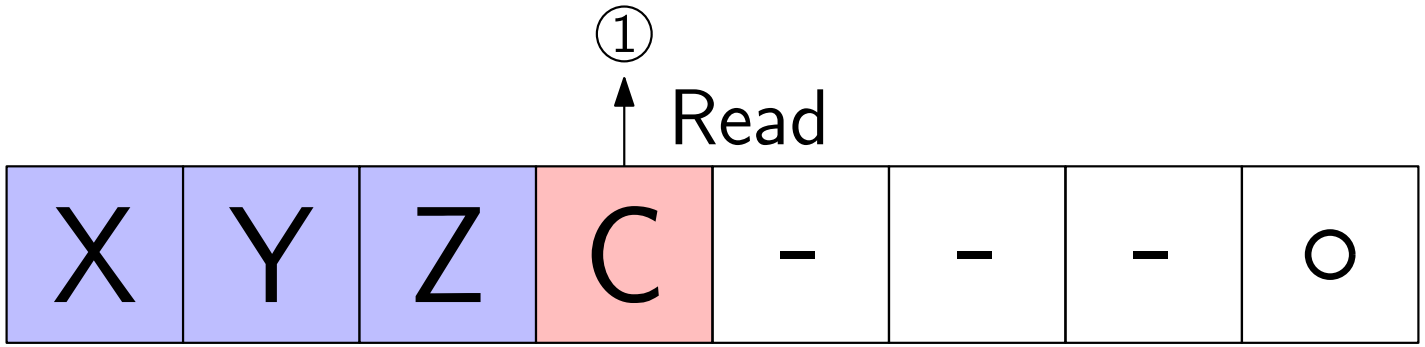
Seal `seal(n: Int)`



1. `cbs = block(i)`
   `s = Seal(sealsize, cbs)`
2. `CAS(block(i), cbs, s)`

Implementation

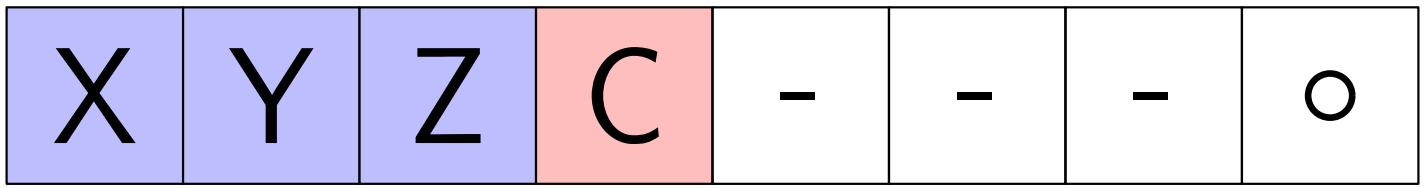
Seal `seal(n: Int)`



1. `cbs = block(i)`
   `s = Seal(sealsize, cbs)`
2. `CAS(block(i), cbs, s)`

Implementation

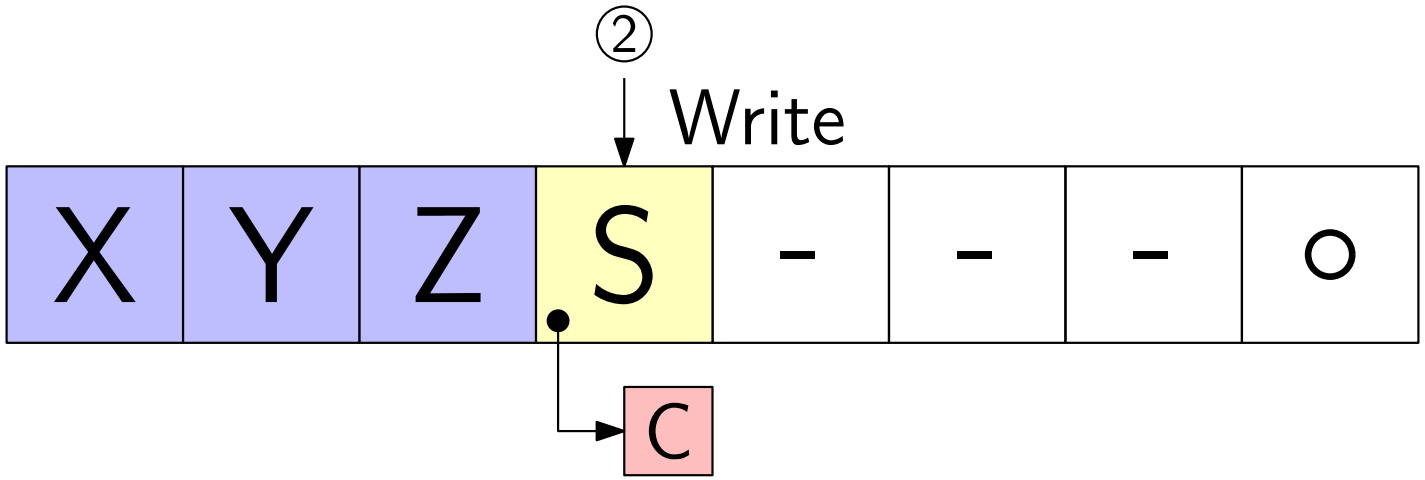
Seal `seal(n: Int)`



1. `cbs = block(i)`
   `s = Seal(sealsize, cbs)`
2. `CAS(block(i), cbs, s)`

Implementation

Seal `seal(n: Int)`



1. `cbs = block(i)`

   `s = Seal(sealsize, cbs)`
2. `CAS(block(i), cbs, s)`

Multi-Lane FlowPools

Single-Lane FlowPools: Issues

- Bad scaling (insertions)
  - CAS failures
  - Cache contention

Solution

- Use unordered property
- Extend to multiple lanes
- Scales nicely
- BUT: Seal is complex

Multi-Lane FlowPools

Single-Lane FlowPools: Issues

- Bad scaling (insertions)
  - CAS failures
  - Cache contention

Solution

- Use unordered property
- Extend to multiple lanes
- Scales nicely
- BUT: Seal is complex

Benchmarks

CPU-Scaling – Insertions

Benchmarks

CPU-Scaling – Insert & Map

Benchmarks

CPU-Scaling – Insert & Reduce

Benchmarks

CPU-Scaling – Communication/Garbage Collection

Conclusion

FlowPools are . . .

Basic Properties

- Flow-Based Collection
- Asynchronous
- Deterministic

Performance & Scalability

- Speed comparable to Java standard queues
- Scalable
- Composable
- Unneeded elements garbage collected

¿Questions?

Conclusion

FlowPools are . . .

Basic Properties

- Flow-Based Collection
- Asynchronous
- Deterministic

Performance & Scalability

- Speed comparable to Java standard queues
- Scalable
- Composable
- Unneeded elements garbage collected

¿Questions?

Conclusion

FlowPools are . . .

Basic Properties

- Flow-Based Collection
- Asynchronous
- Deterministic

Performance & Scalability

- Speed comparable to Java standard queues
- Scalable
- Composable
- Unneeded elements garbage collected

¿Questions?

Conclusion

FlowPools are . . .

Basic Properties

- Flow-Based Collection
- Asynchronous
- Deterministic

Performance & Scalability

- Speed comparable to Java standard queues
- Scalable
- Composable
- Unneeded elements garbage collected

¿Questions?

Details about Benchmarks

Insert / Reduce

- $5 \cdot 10^6$ elements
- 20 measurements

Communication / GC

- Parallelization level: 1
- Measurements
  - UltraSPARC T2: 2
  - 4-core i7: 10
  - 32-core Xeon: 3

Java Command

```
-Xmx2048m -Xms2048m
-XX:+UseCondCardMark
-verbose:gc
-XX:+PrintGCDetails -server.
```

Java Version

- Intel `1.7.0_04-ea-b15 HotSpot 64-Bit Server VM (build 23.0-b16, mixed mode)`
- SPARC `1.7.0_03-b04 HotSpot Server VM (build 22.1-b02, mixed mode)`

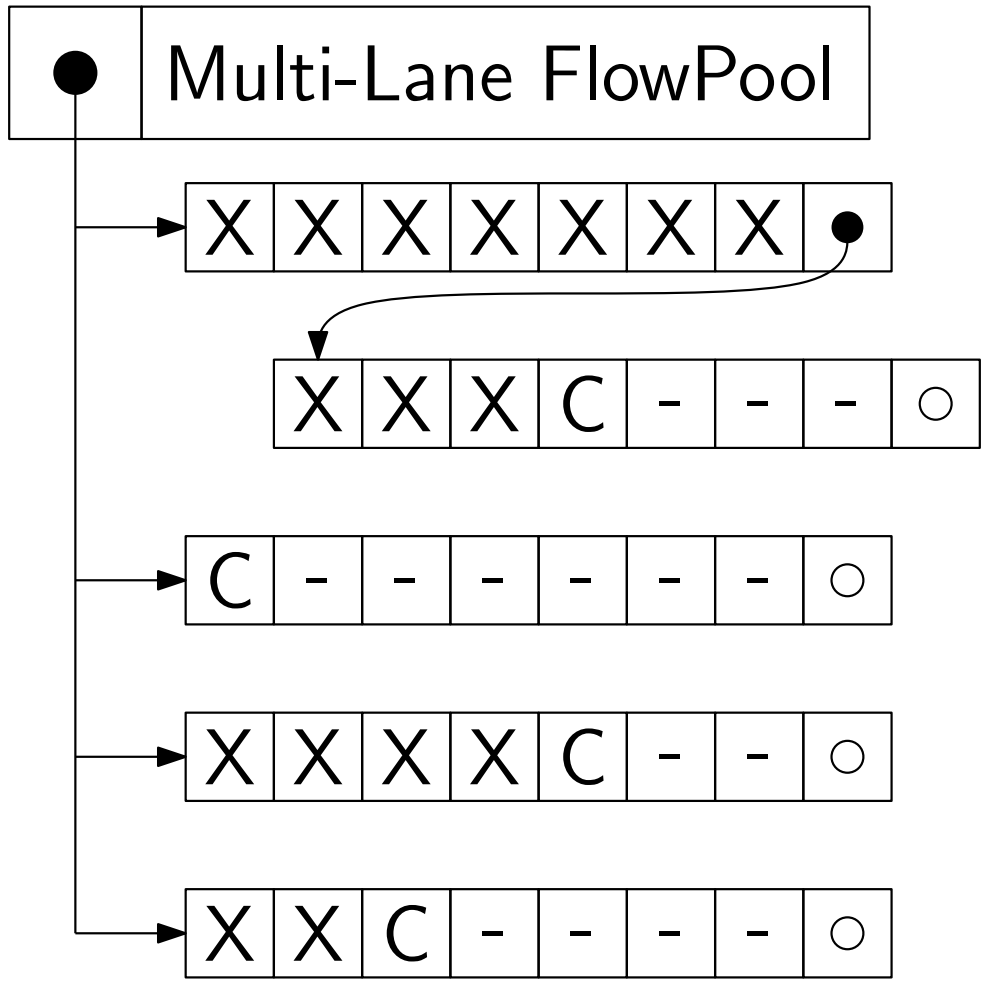
Details about Benchmarks

Architectures

- octa-core 1.2GHz UltraSPARC T2 w/ 64 HW threads
- quad-core 3.4 GHz i7-2600 (w/ HT)
- 4x octa-core 2.27 GHz Intel Xeon x7560 (w/ HT)

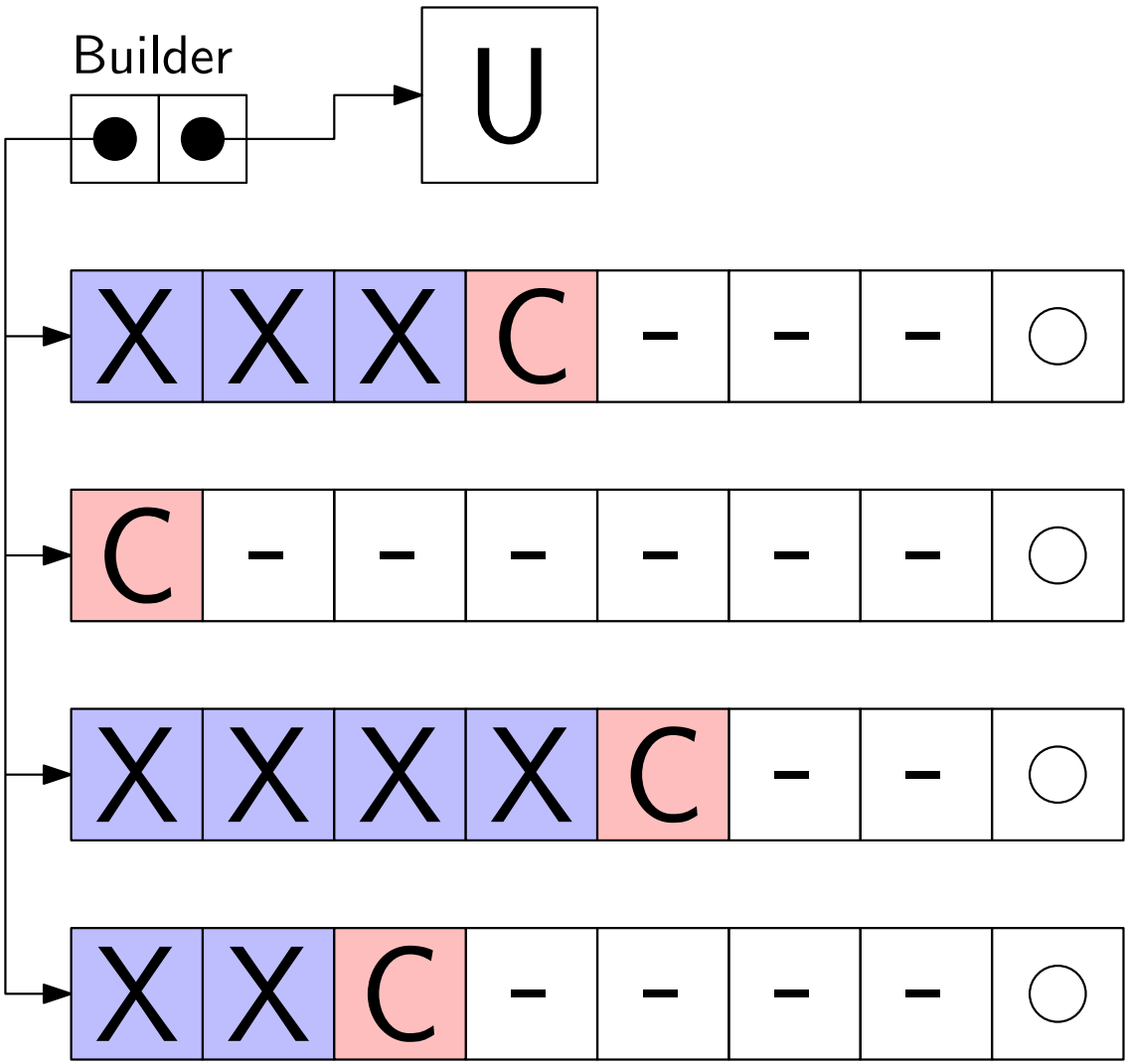
Multi-Lane FlowPools

Basic Structure



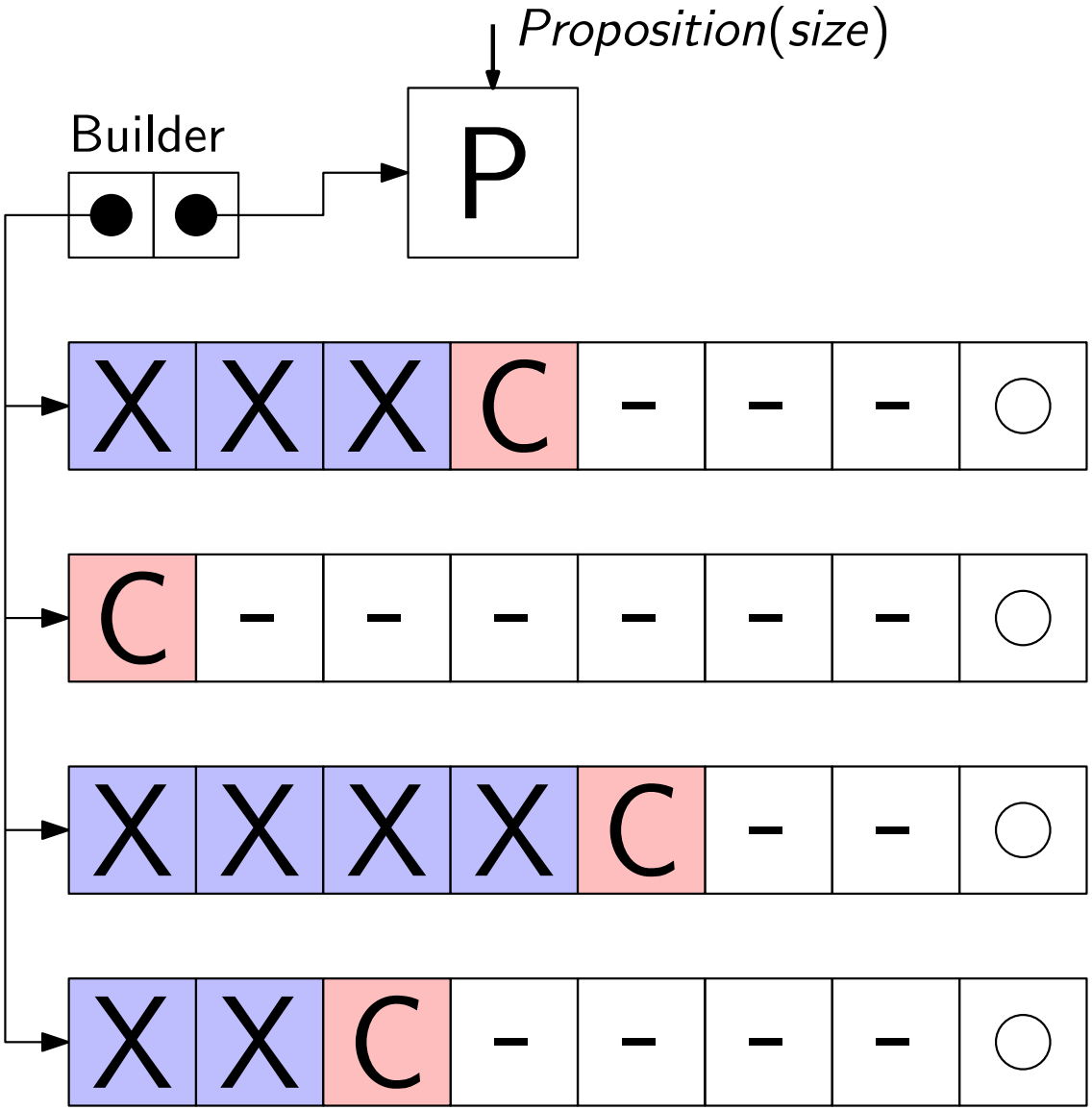
Multi-Lane FlowPools

Seal `seal(n: Int)`



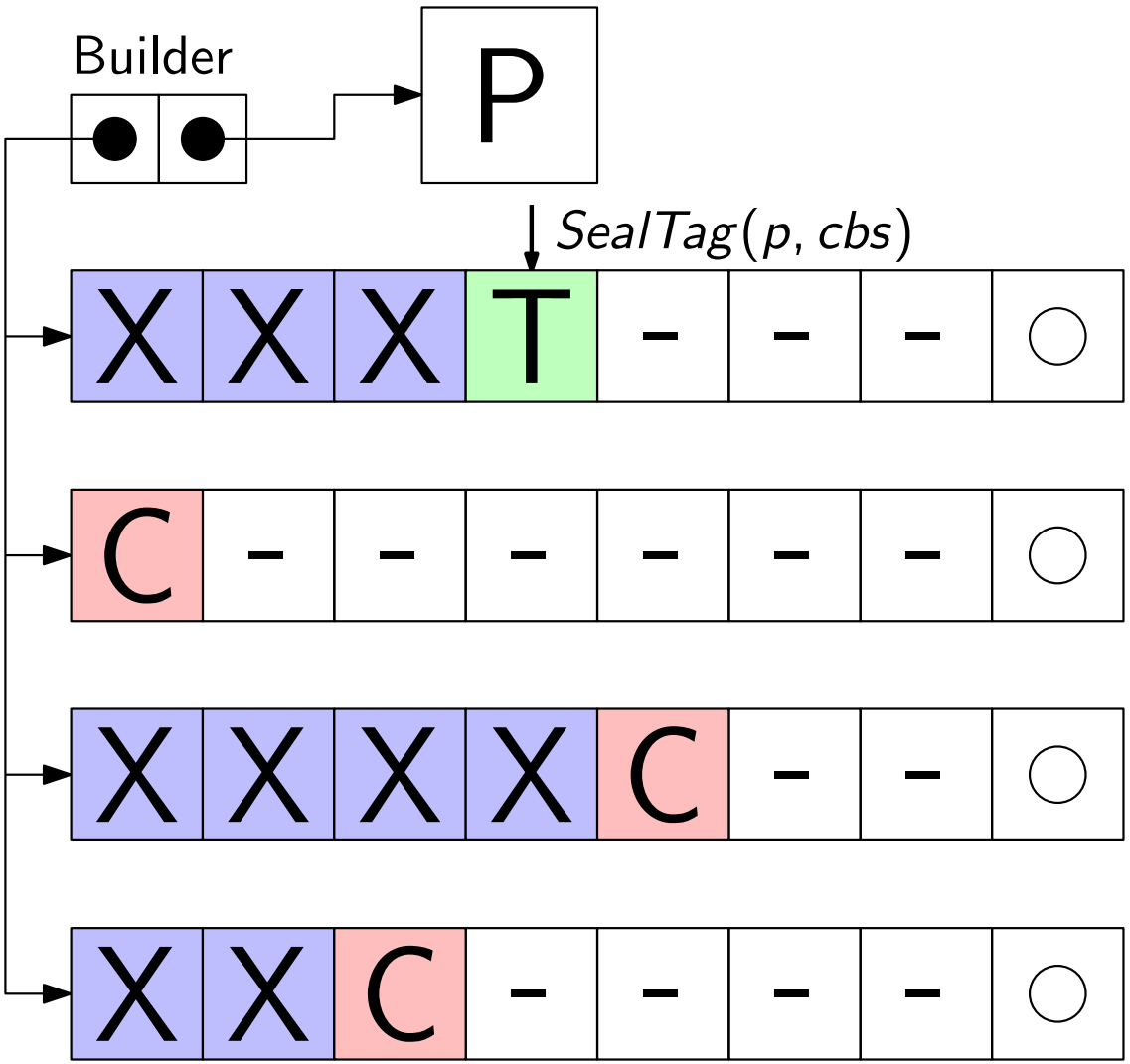
Multi-Lane FlowPools

Seal `seal(n: Int)`



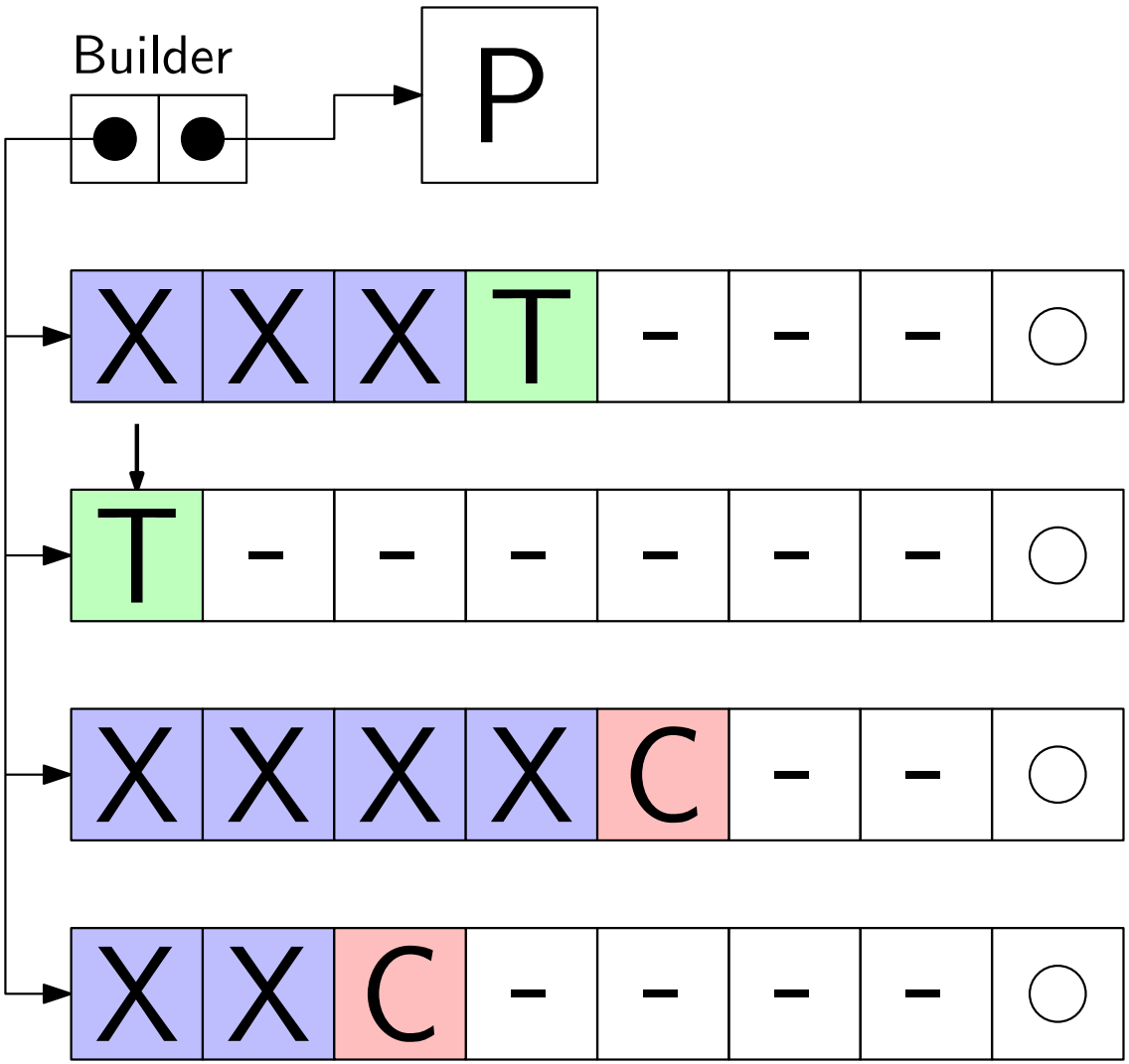
Multi-Lane FlowPools

Seal `seal(n: Int)`



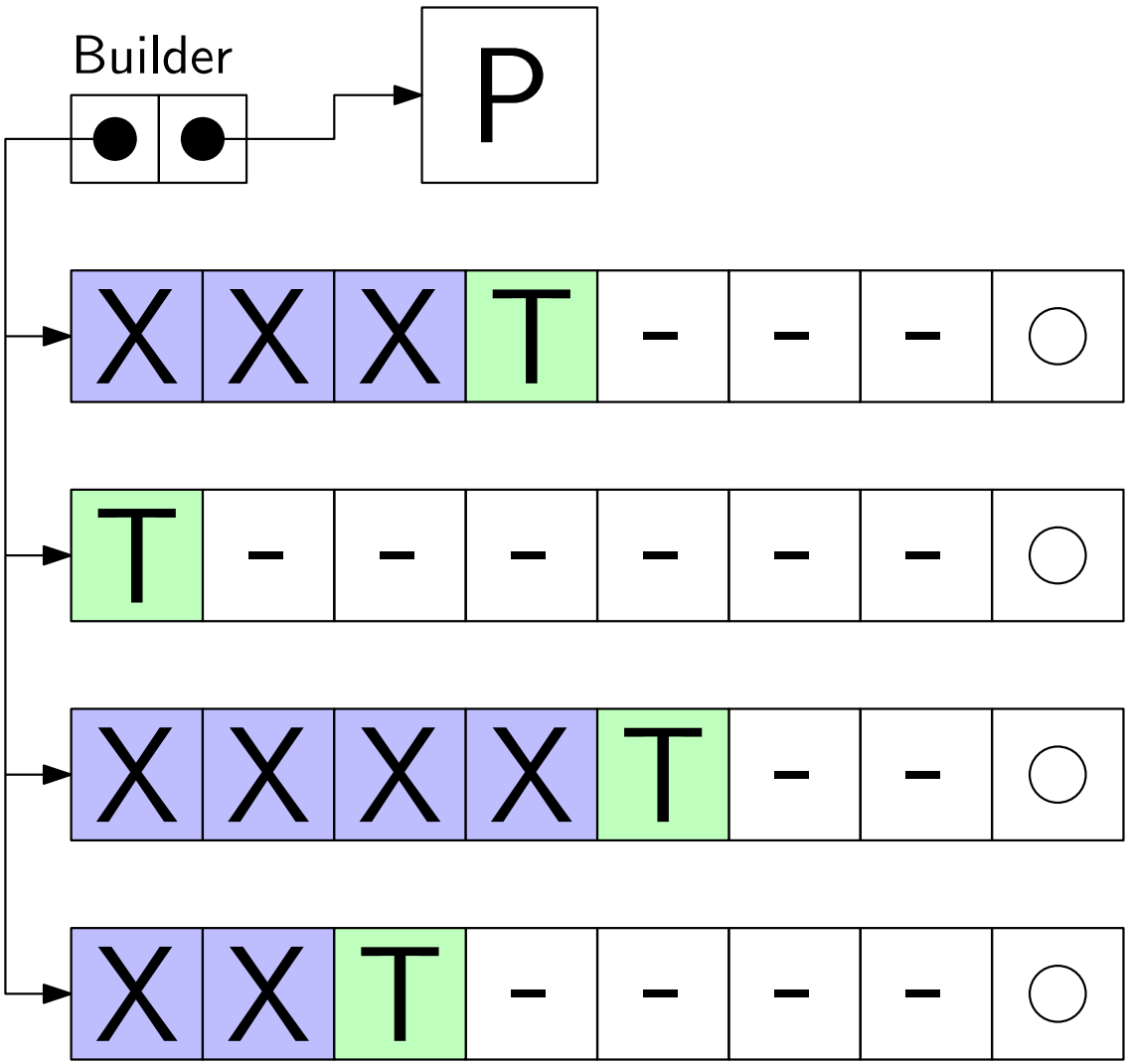
Multi-Lane FlowPools

Seal `seal(n: Int)`



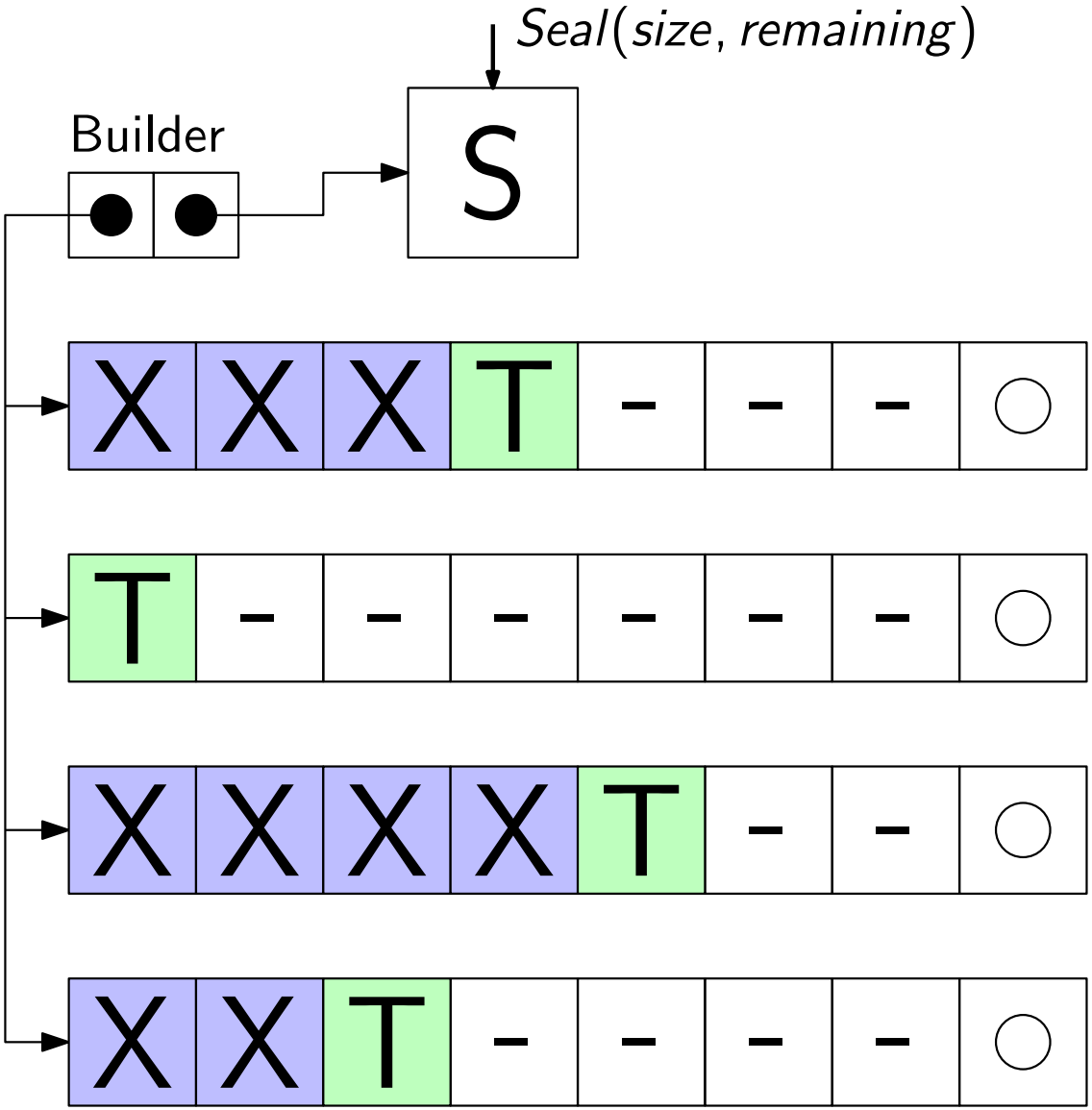
Multi-Lane FlowPools

Seal `seal(n: Int)`



Multi-Lane FlowPools

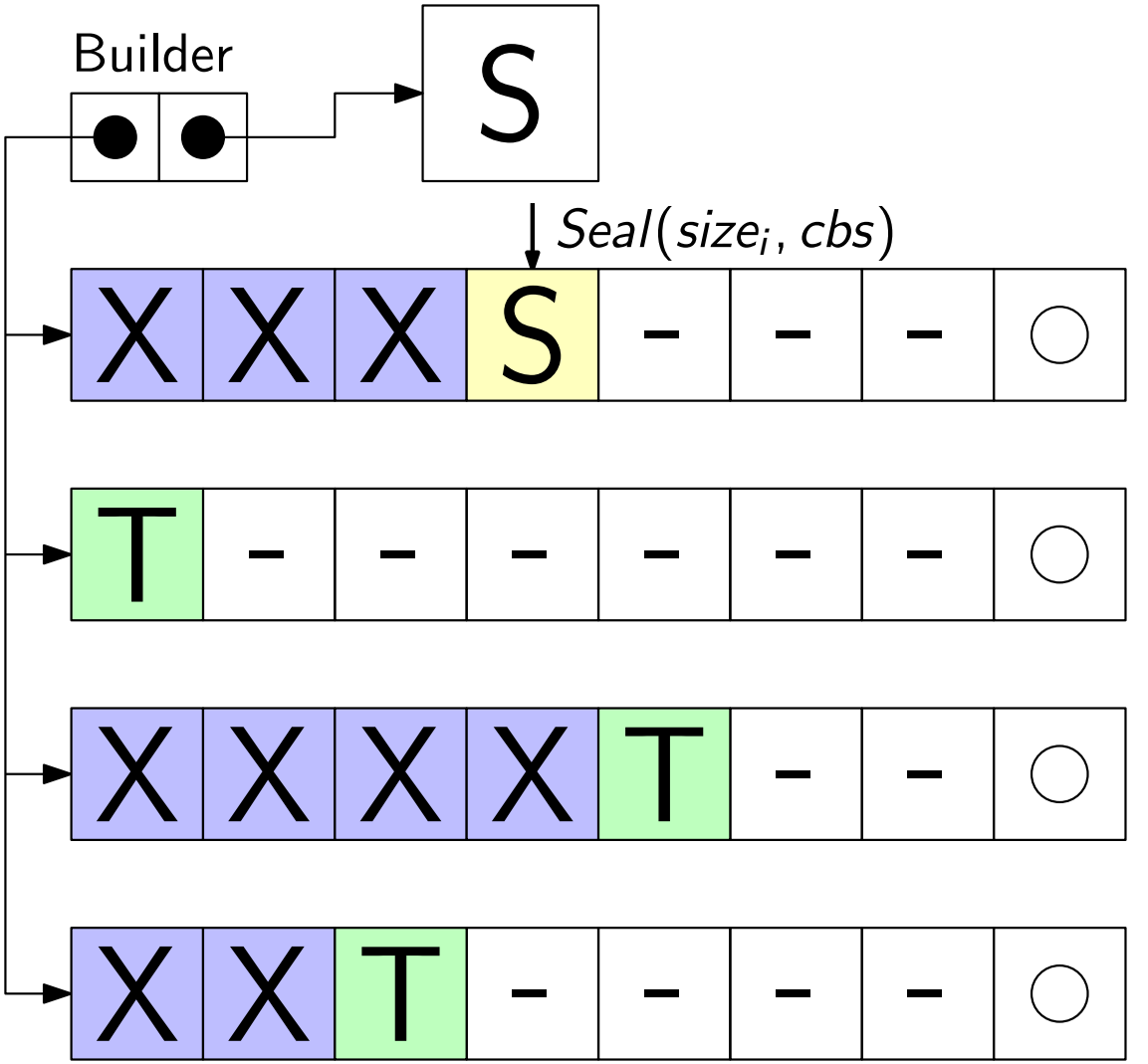
Seal `seal(n: Int)`



Multi-Lane FlowPools

Seal `seal(n: Int)`

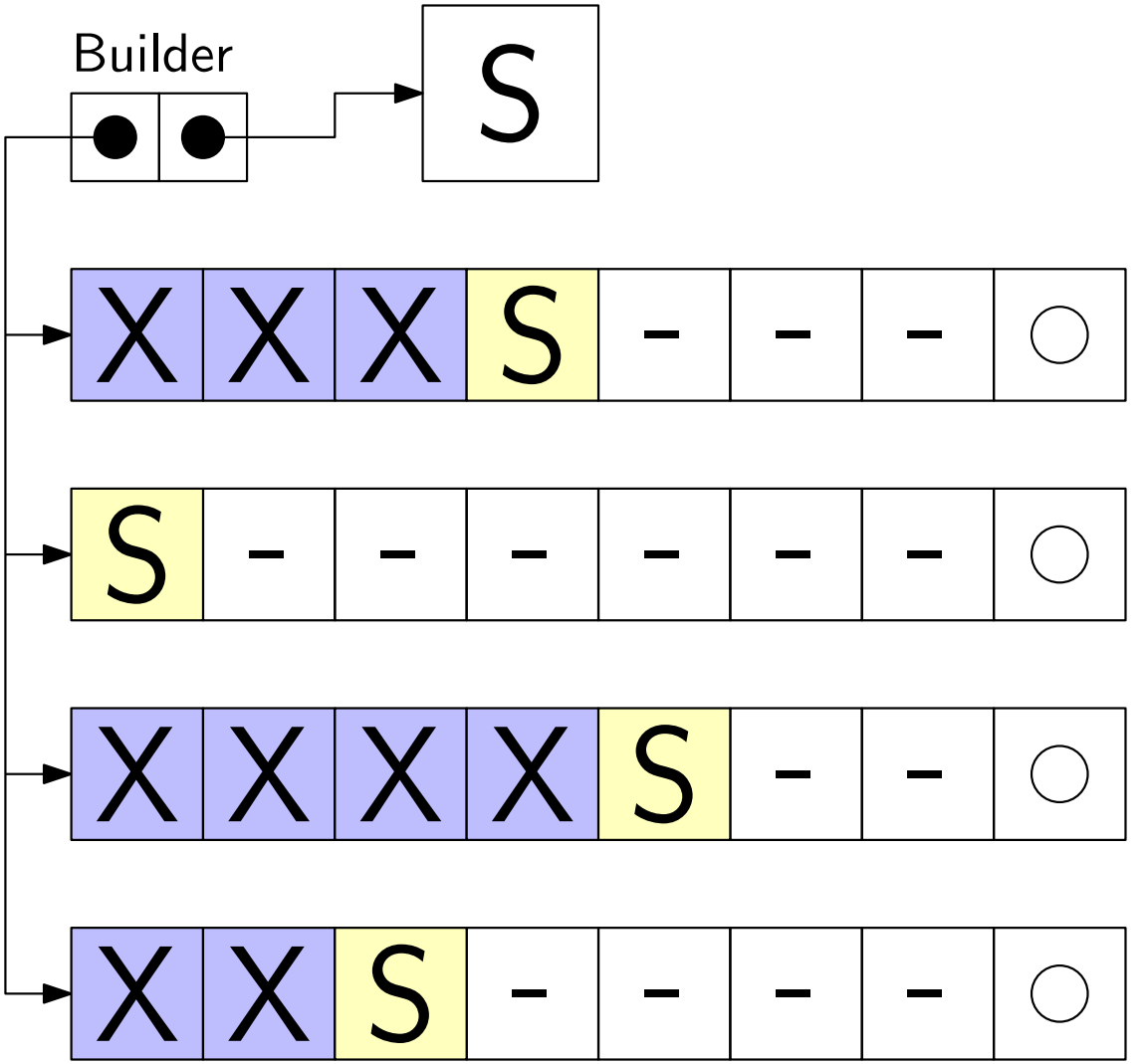
$$size_i = size_{cur} + \frac{remaining}{L_{tot}} + 1_{\{i < remaining \bmod L_{tot}\}}$$



Multi-Lane FlowPools

Seal `seal(n: Int)`

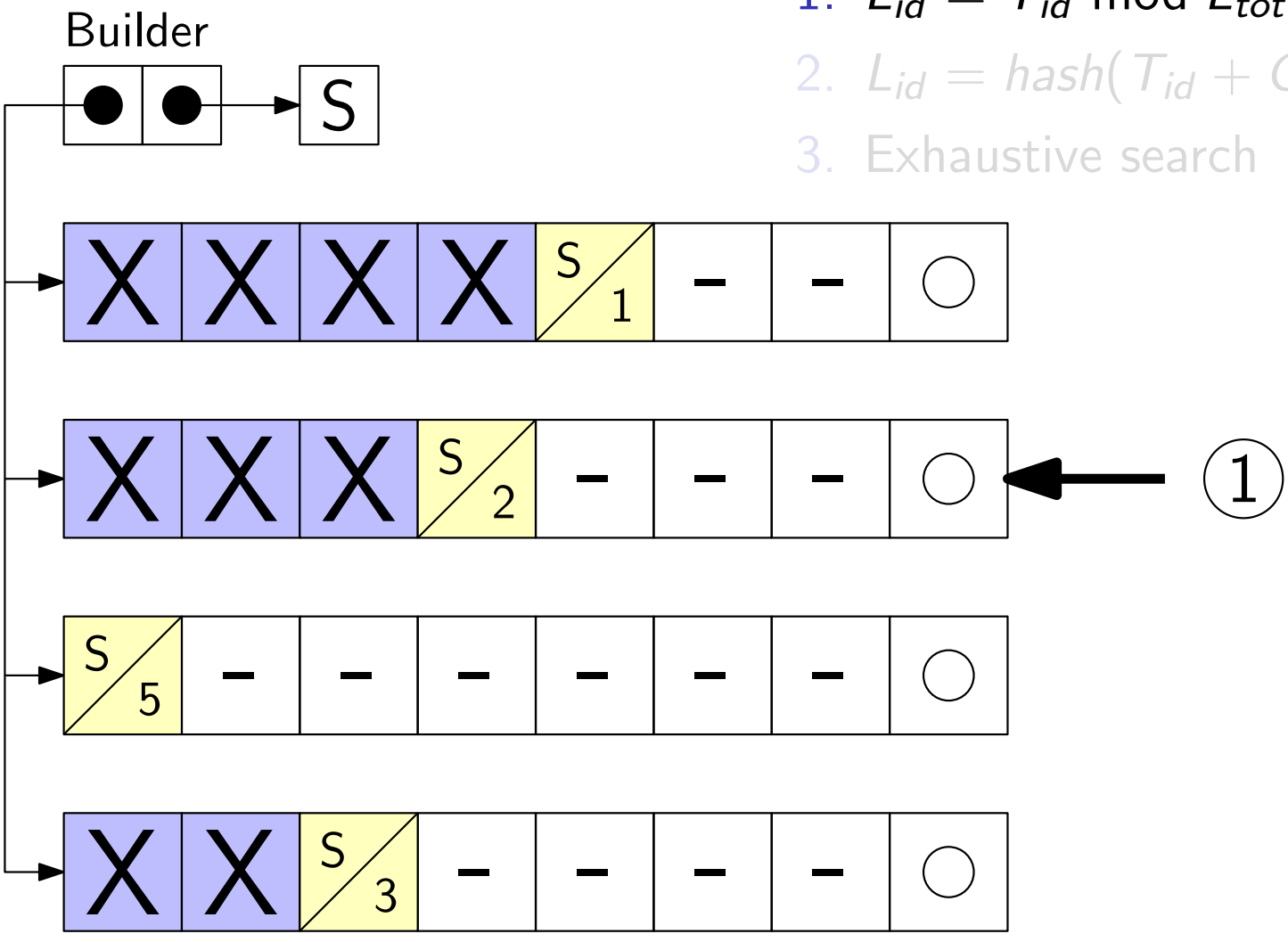
$$size_i = size_{cur} + \frac{remaining}{L_{tot}} + 1_{\{i < remaining \bmod L_{tot}\}}$$



Multi-Lane FlowPools

Insert / Choice of Lane

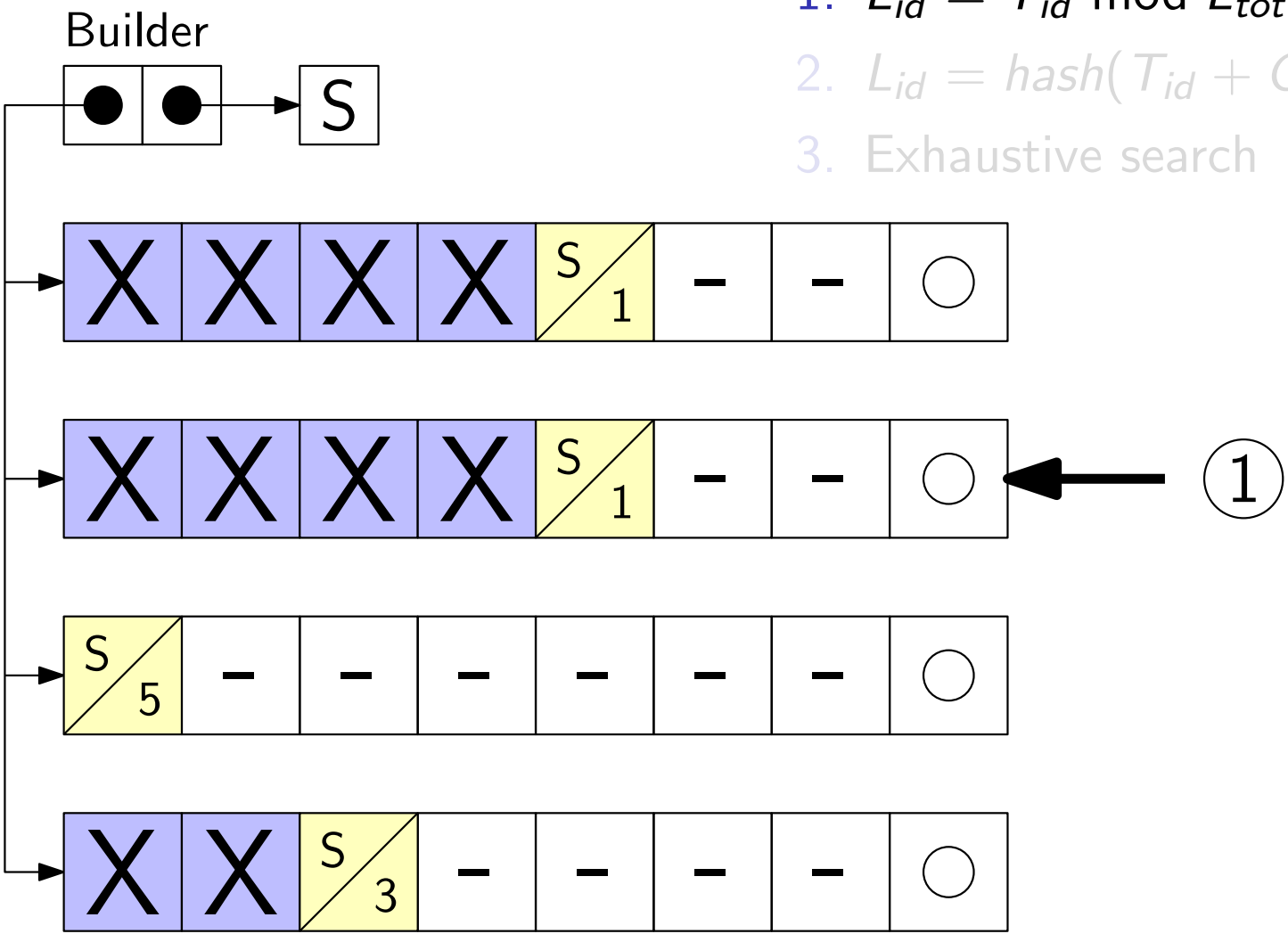
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

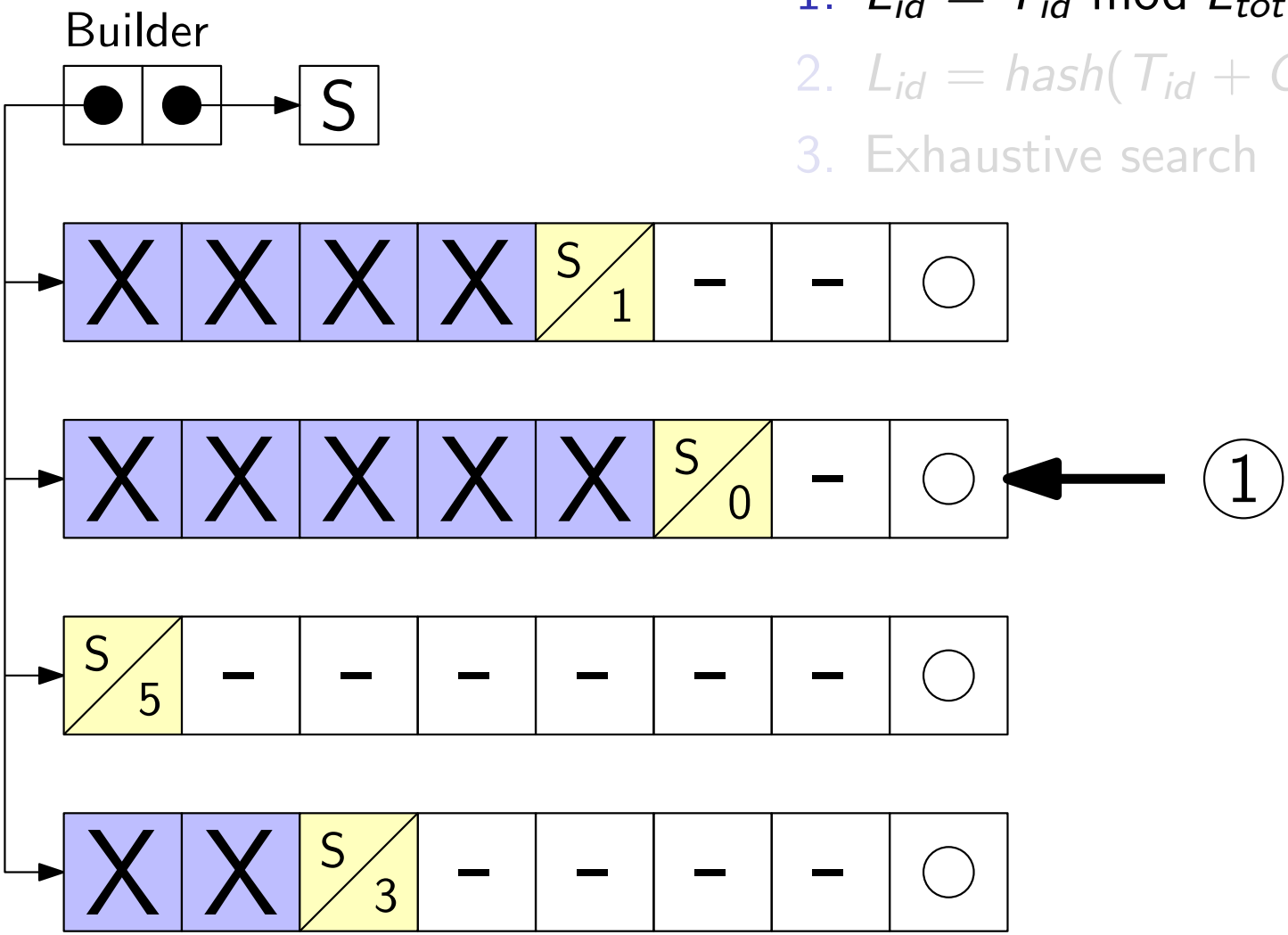
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

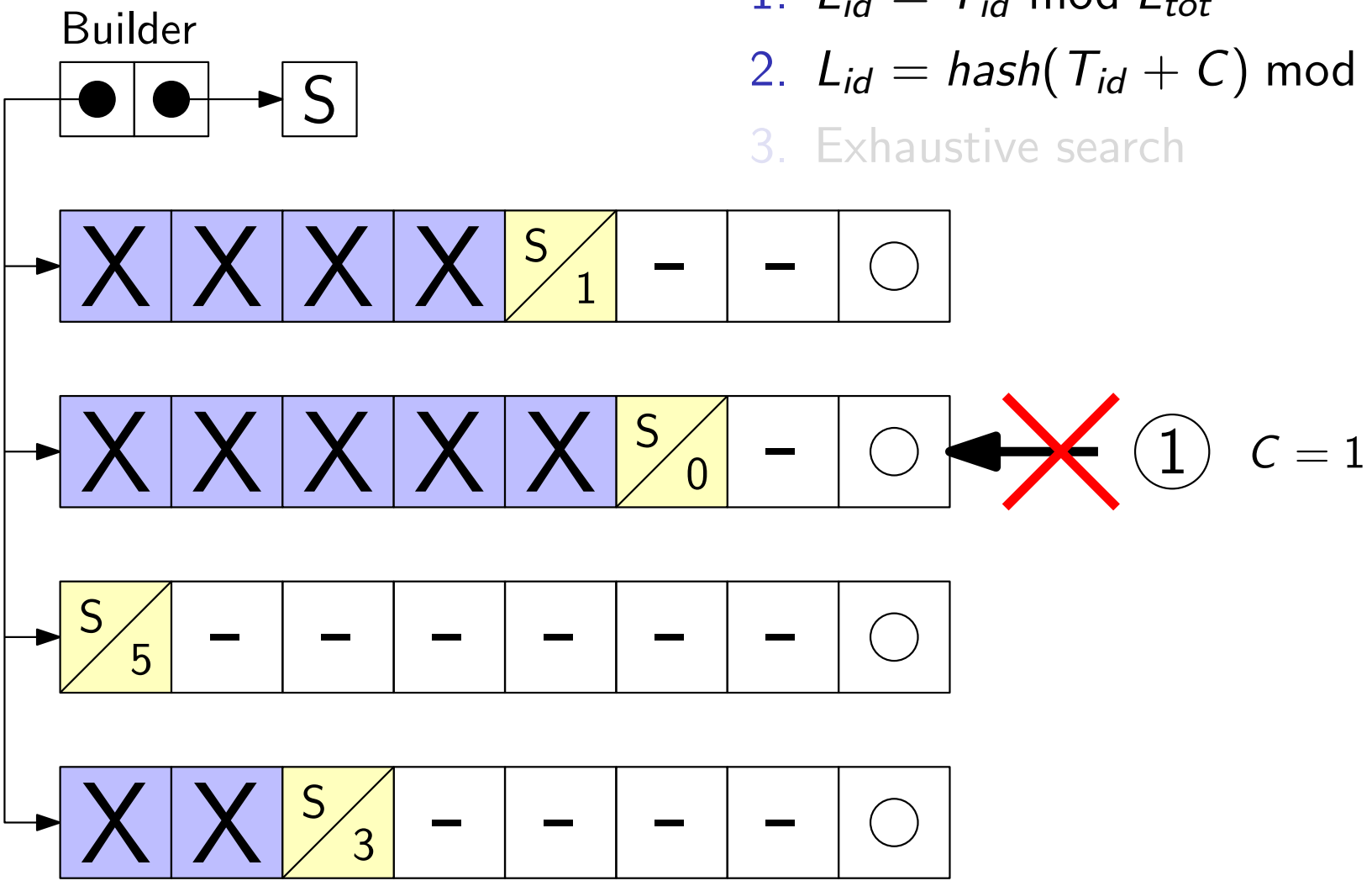
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

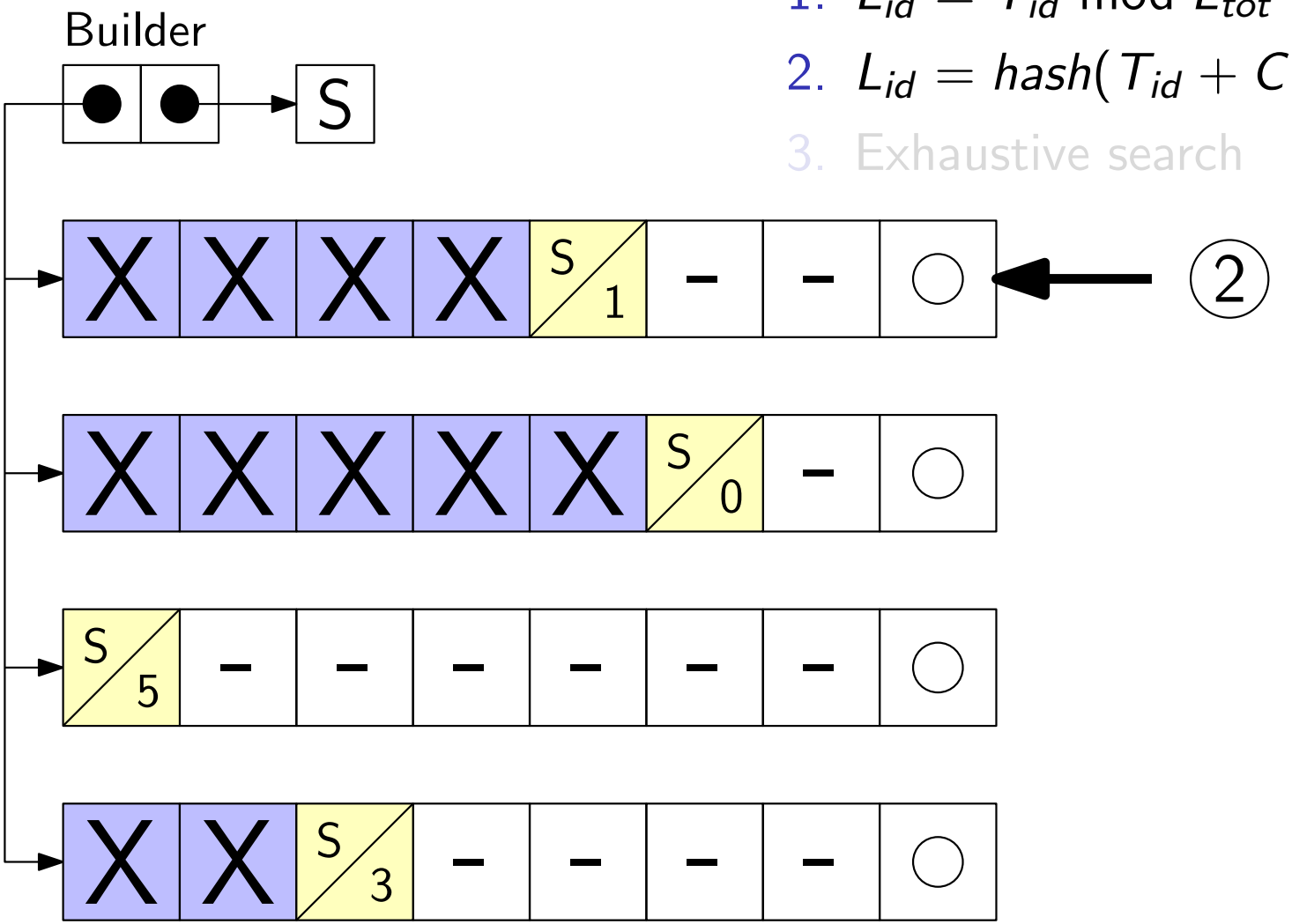
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

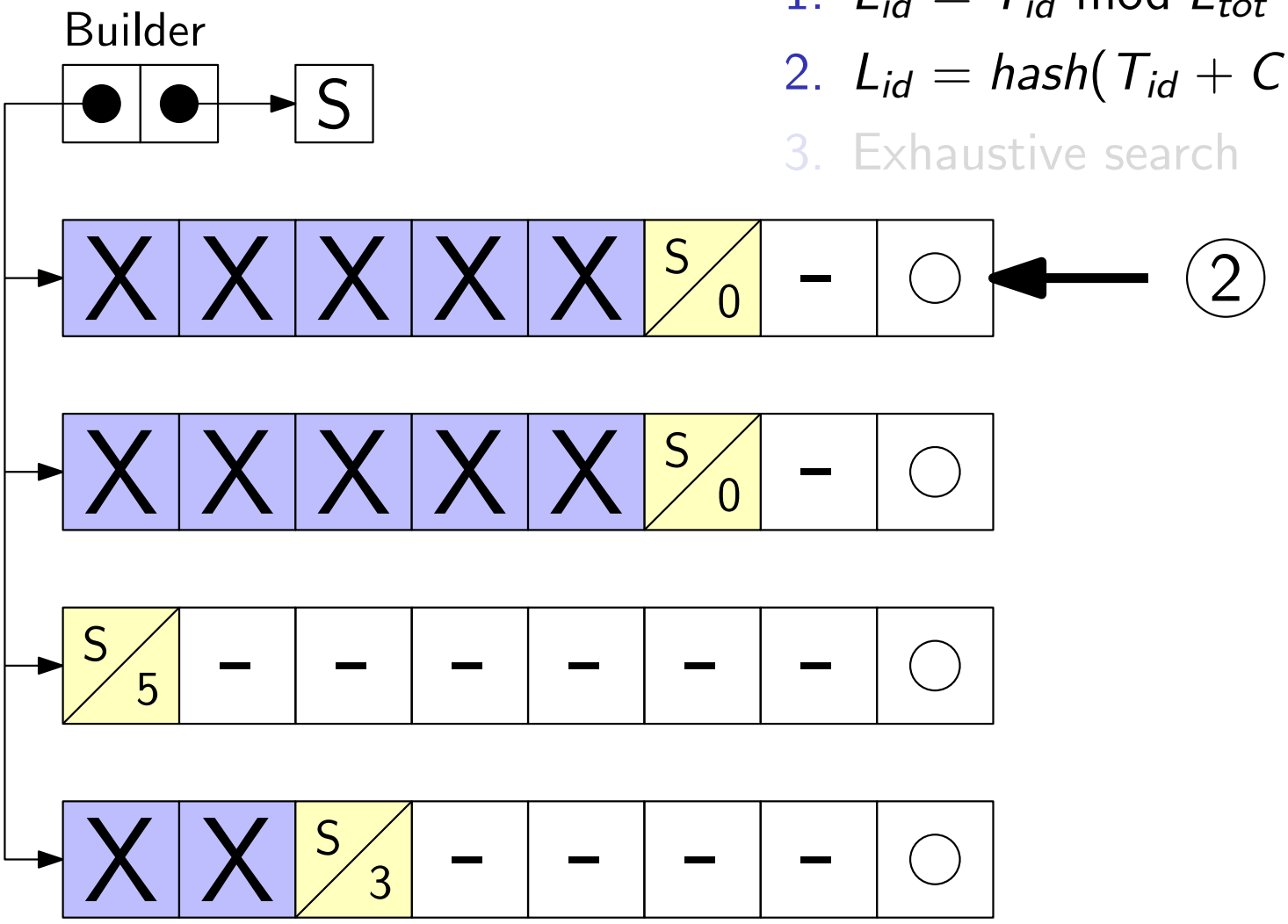
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

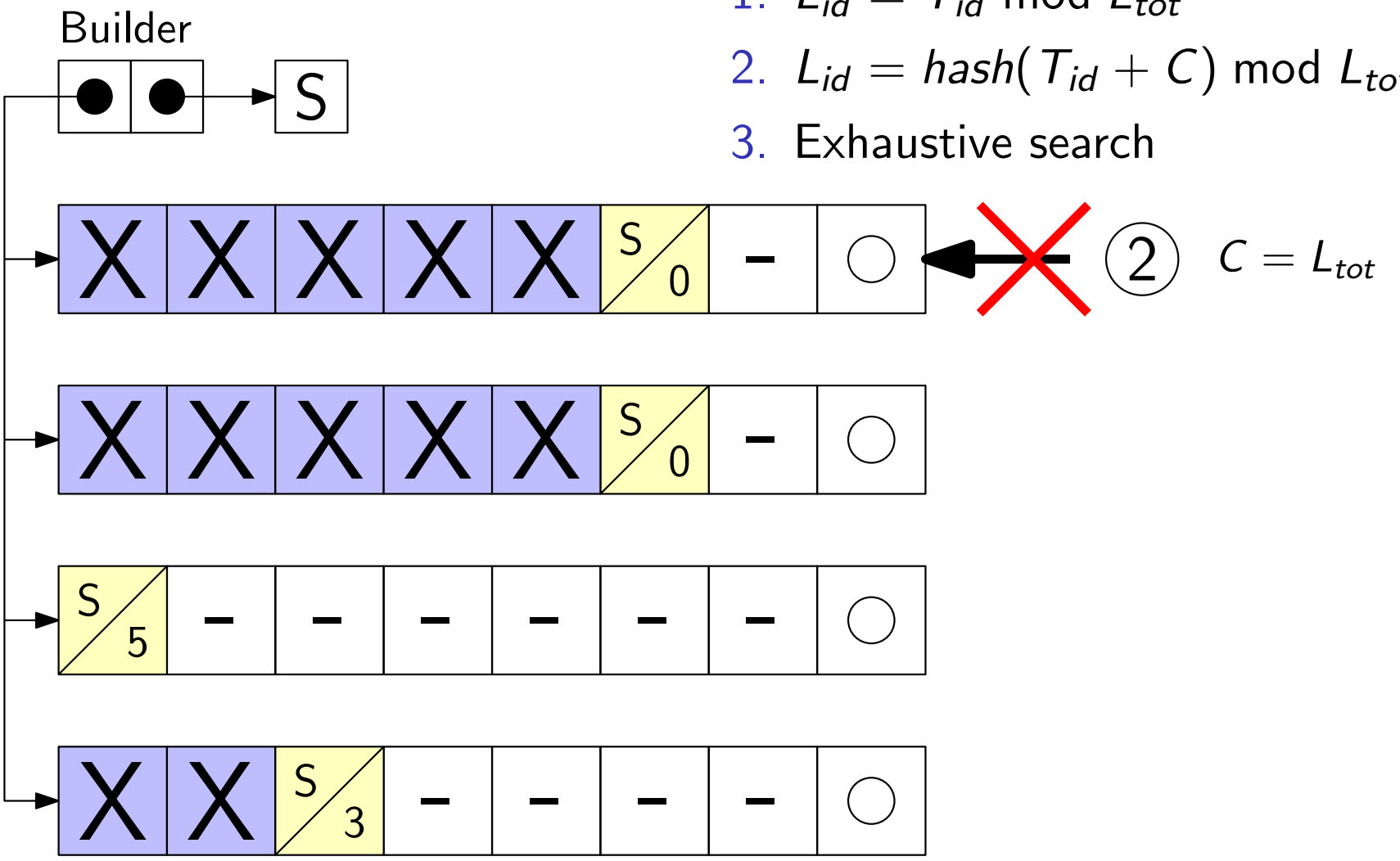
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

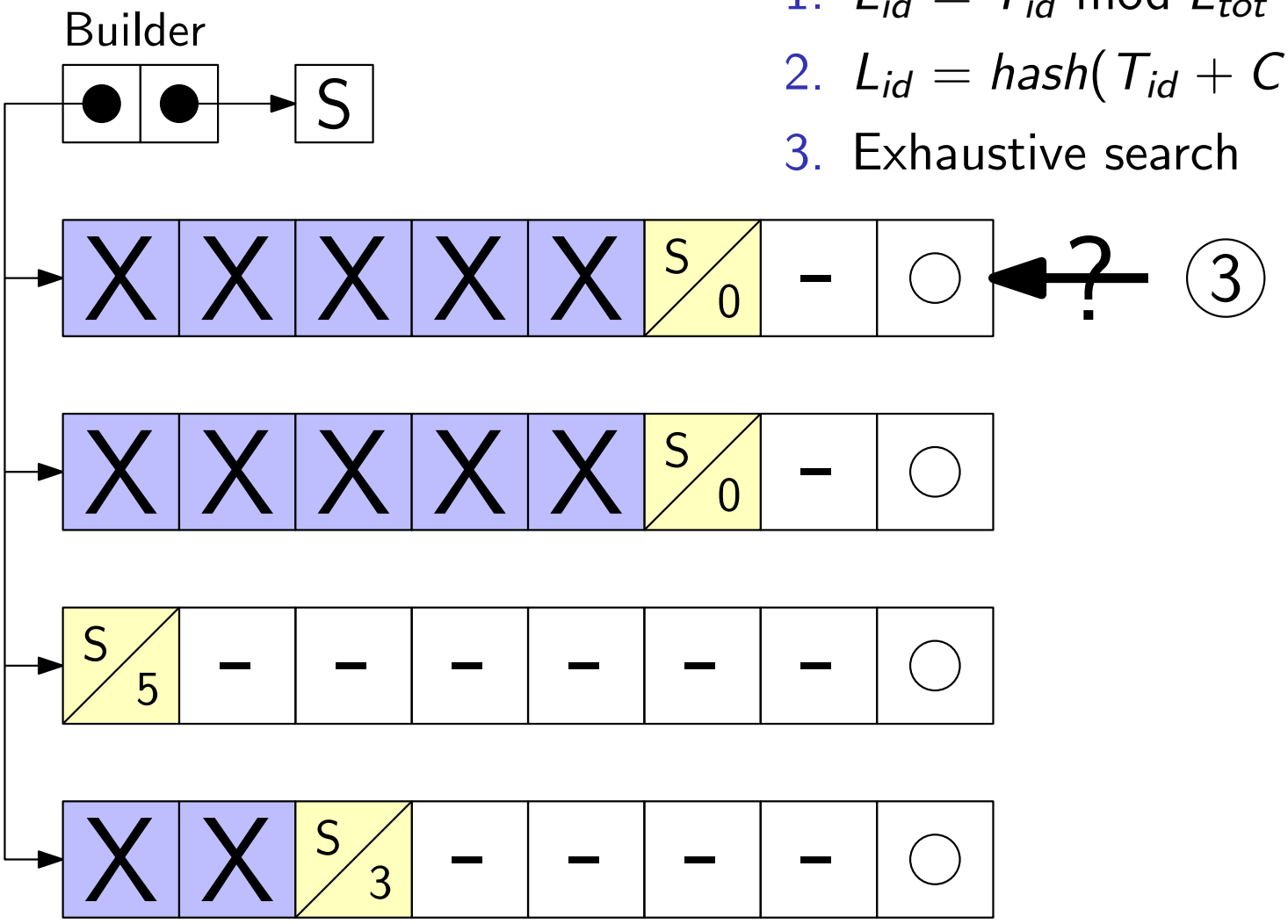
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

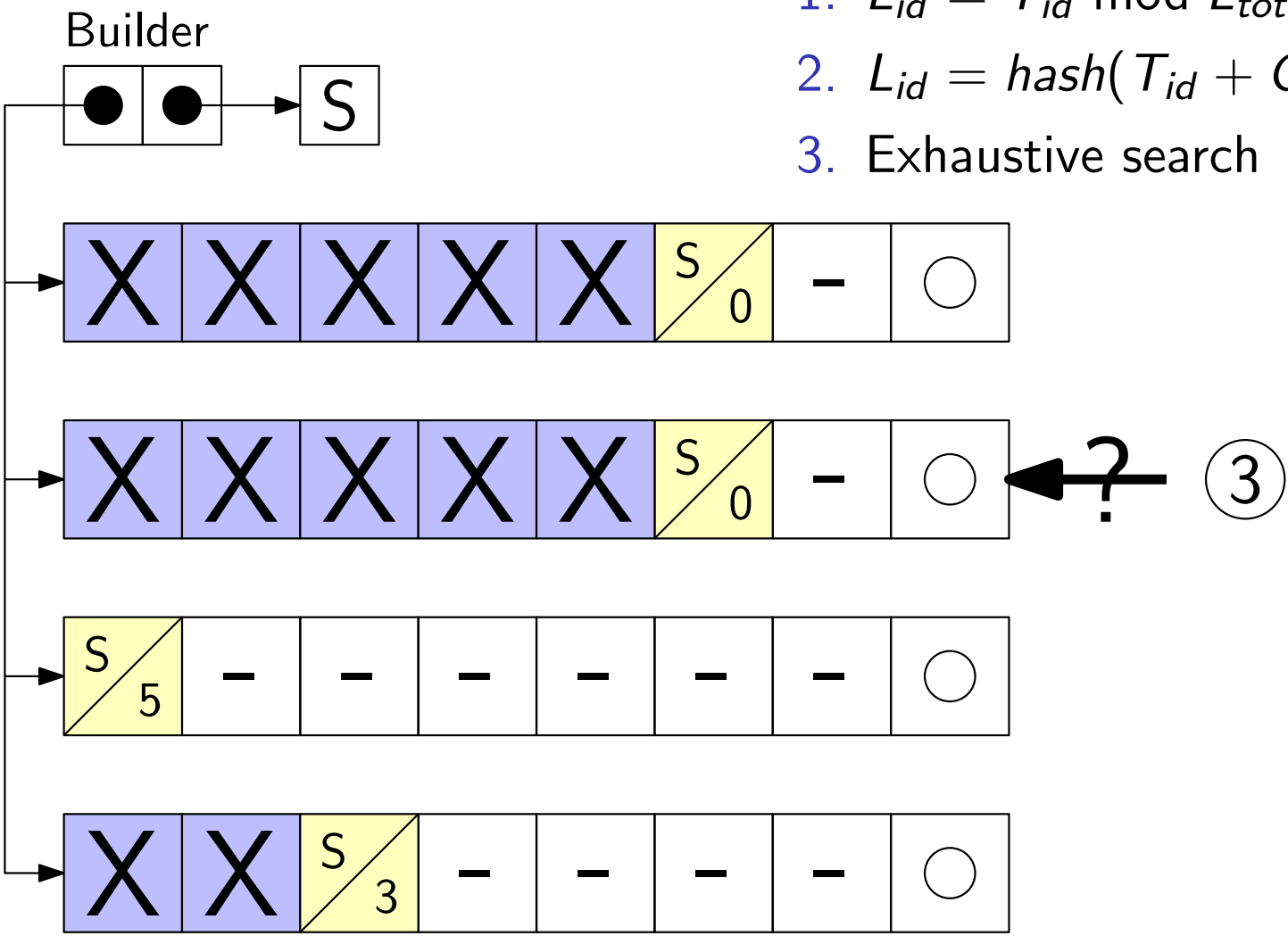
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

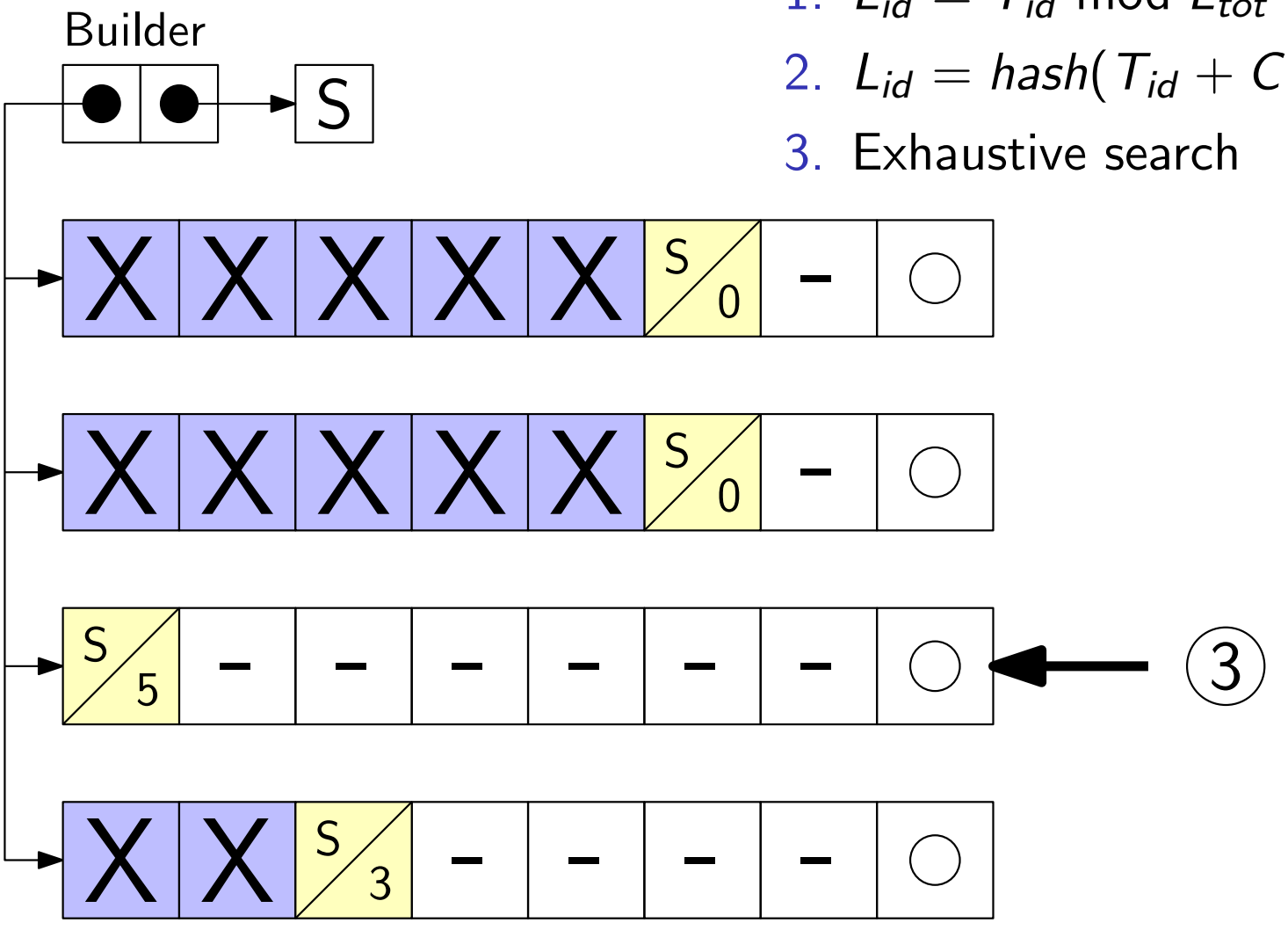
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

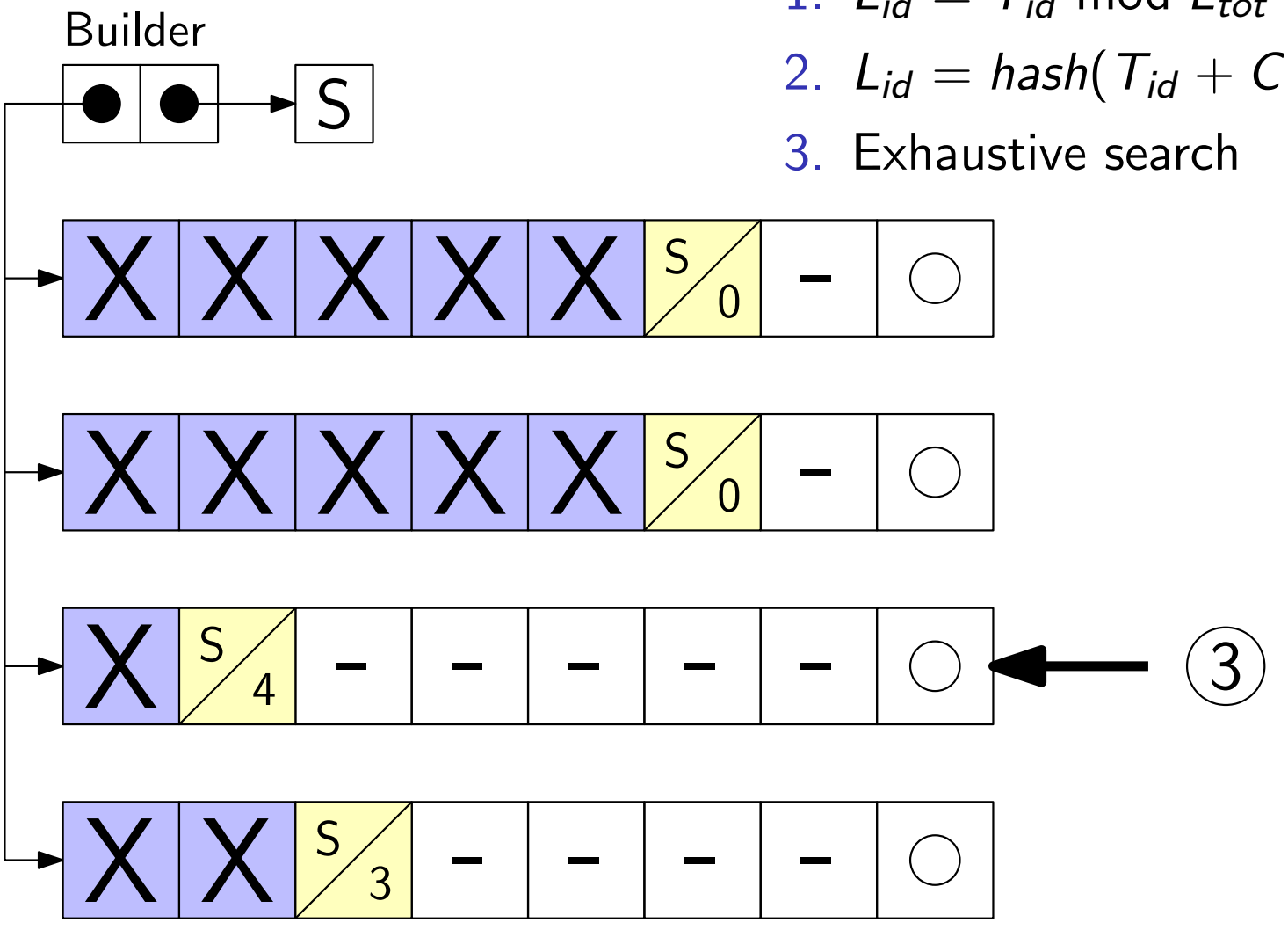
1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Multi-Lane FlowPools

Insert / Choice of Lane

1. $L_{id} = T_{id} \bmod L_{tot}$
2. $L_{id} = hash(T_{id} + C) \bmod L_{tot}$
3. Exhaustive search



Hash Function

Byte-swap Hashing

$$hash(x) = \mathrm{rb}(x \cdot 9e3775cd_{16}) \cdot 9e3775cd_{16}$$

Efficiency of Hashing

$\times$ UltraSPARC T2, $\triangle$ 4-core i7