Shooting the Rapids: Getting the Best from Java 8 Streams

Kirk Pepperdine
@kcpeppe
Maurice Naftalin
@mauricenaftalin

About Kirk

- Specialises in performance tuning
  - speaks frequently about performance
  - author of performance tuning workshop
- Co-founder jClarity
  - performance diagnositic tooling
- Java Champion (since 2006)

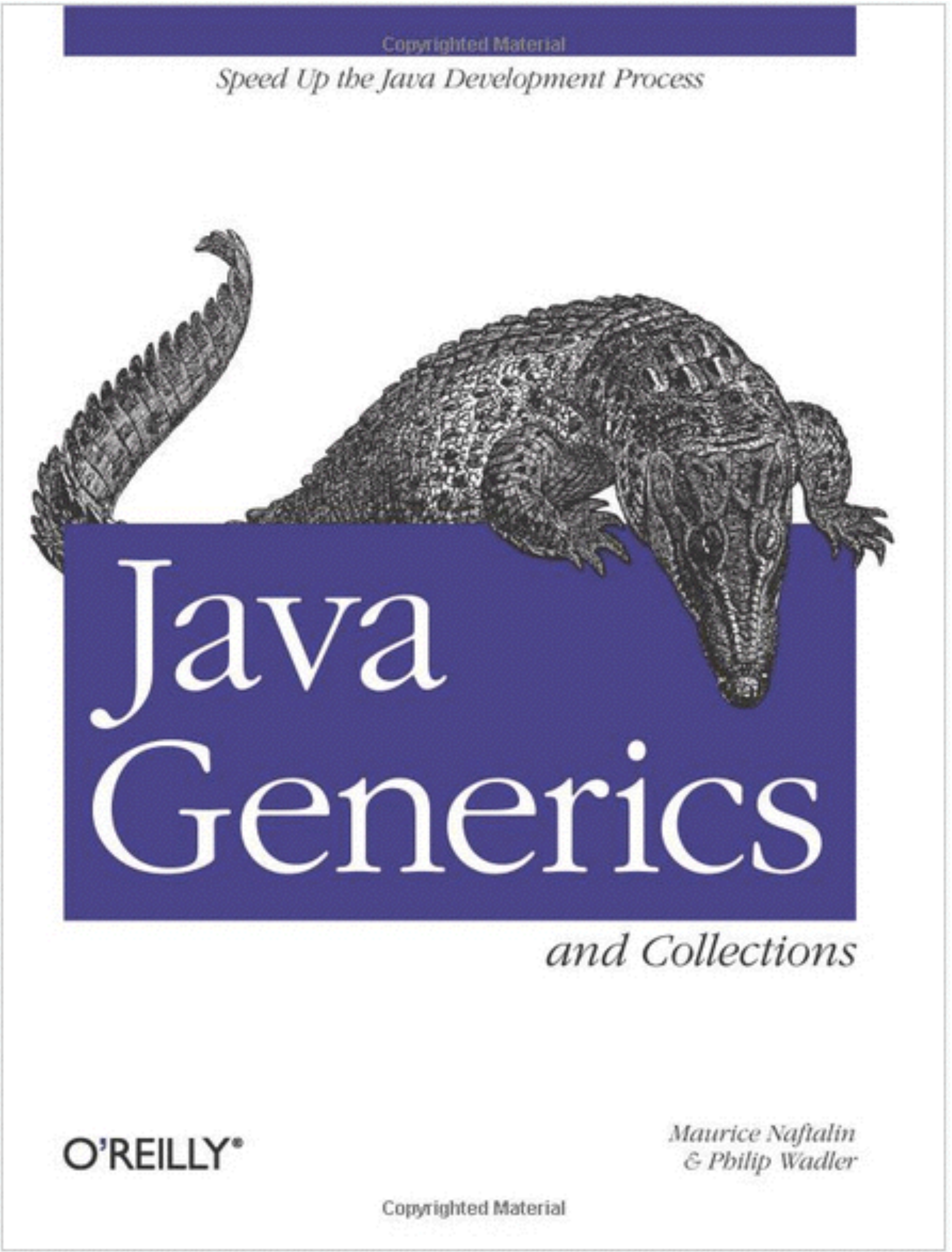
About Kirk

- Specialises in performance tuning
  - speaks frequently about performance
  - author of performance tuning workshop
- Co-founder jClarity
  - performance diagnositic tooling
- Java Champion (since 2006)

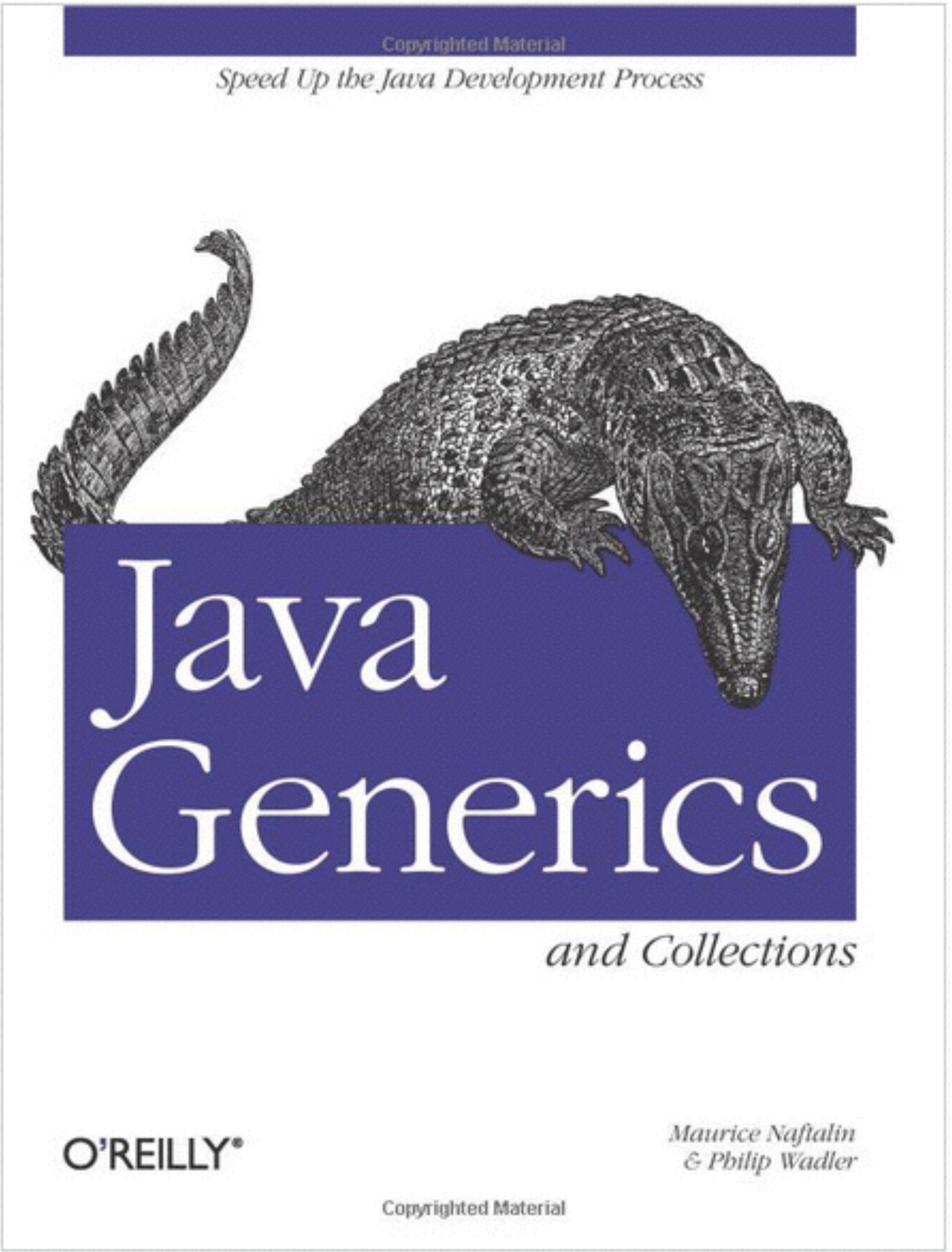
Java™ Champions

About Maurice

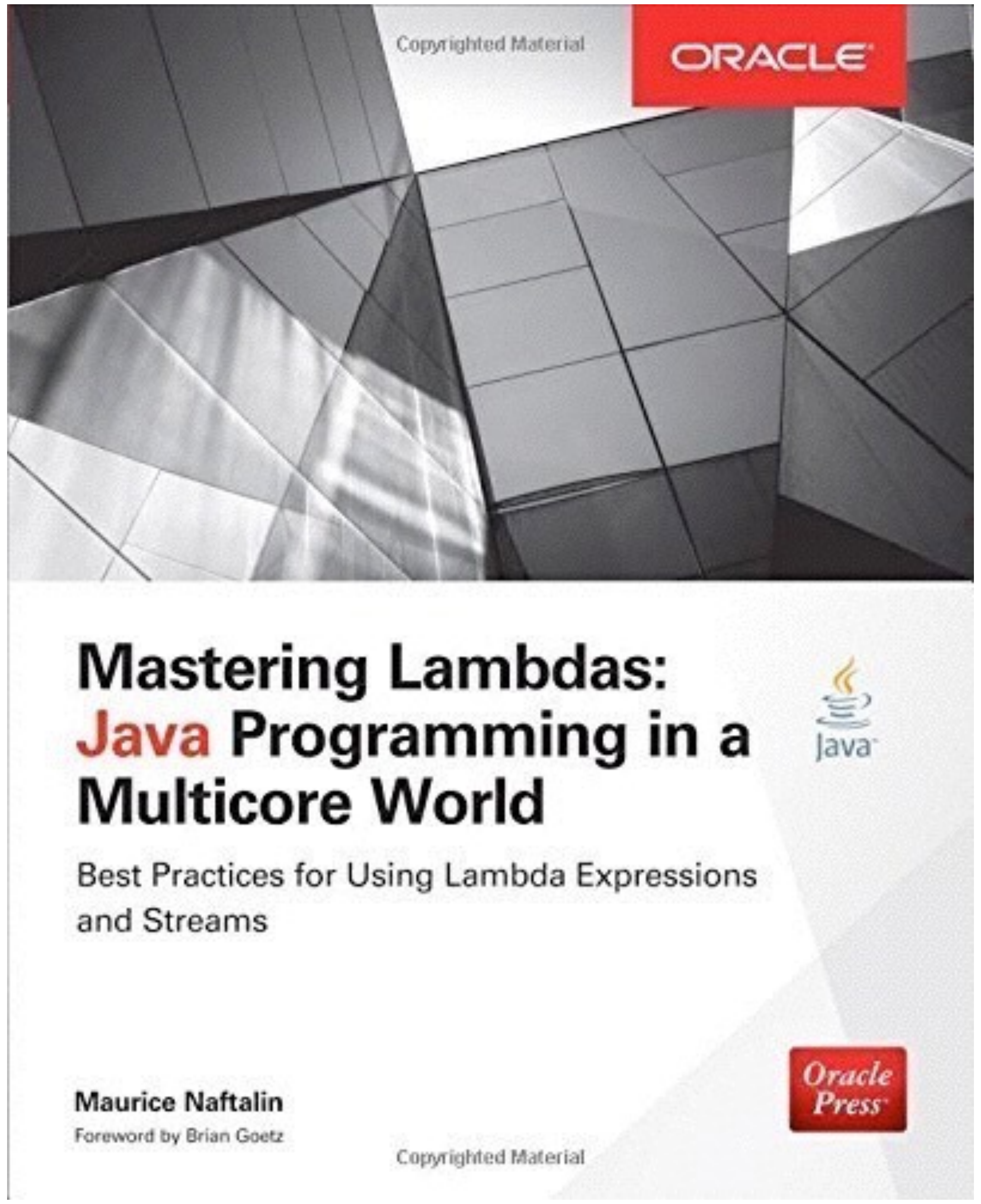
About Maurice



About Maurice

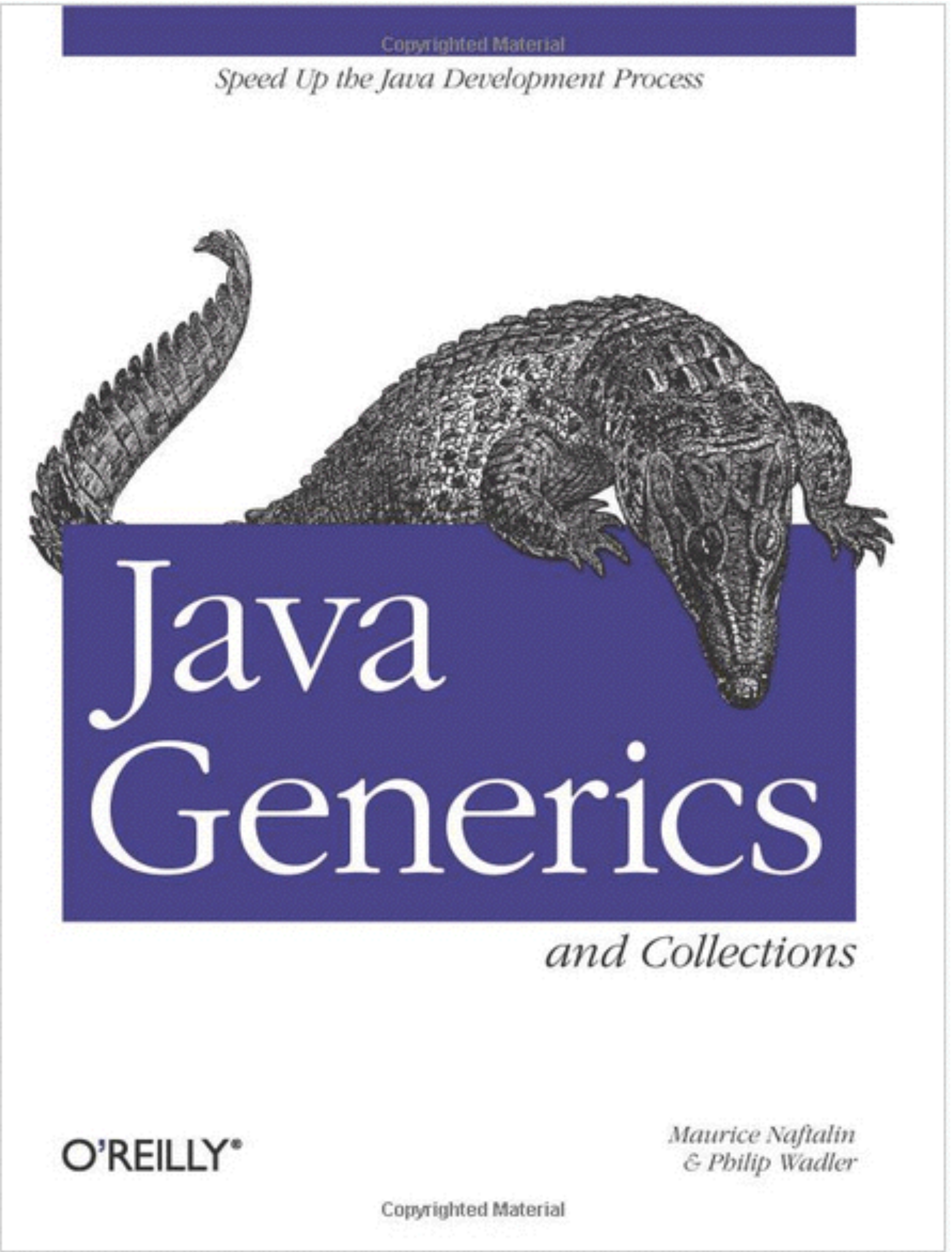


Co-author

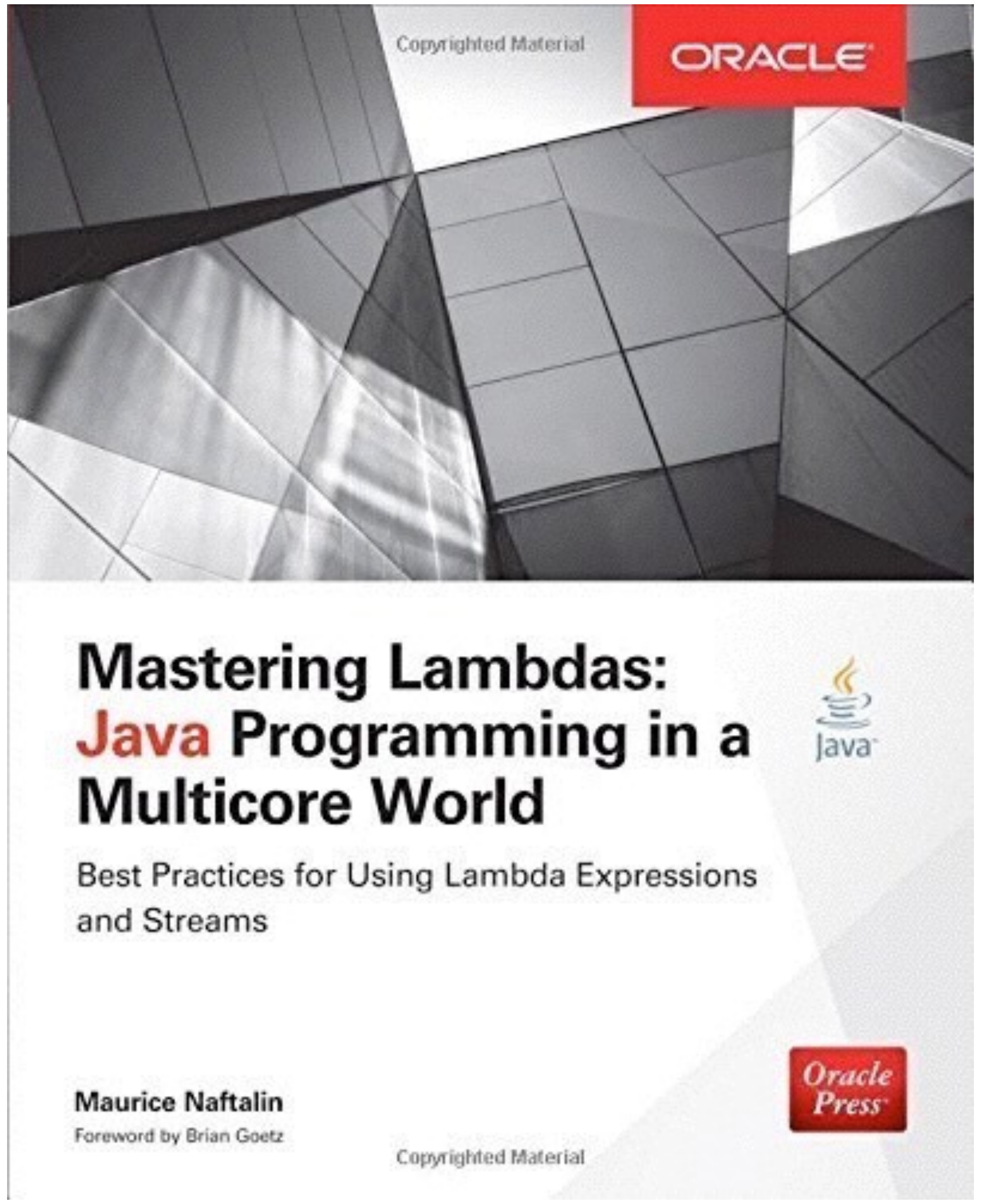


Author

About Maurice



Co-author



Author

Java
Champion

JavaOne
Rock Star

Agenda

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem
- Optimizing stream sources

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons
- Justifying the Overhead

Example: Processing GC Logfile

```
⋮
2.869: Application time: 1.0001540 seconds
5.342: Application time: 0.0801231 seconds
8.382: Application time: 1.1013574 seconds
⋮
```

Example: Processing GC Logfile

```
⋮
2.869: Application time: 1.0001540 seconds
5.342: Application time: 0.0801231 seconds
8.382: Application time: 1.1013574 seconds
⋮
```

Example: Processing GC Logfile

```
:
2.869: Application time: 1.0001540 seconds
5.342: Application time: 0.0801231 seconds
8.382: Application time: 1.1013574 seconds
:
```

```
sum=2.181635
```

Example: Processing GC Logfile

```
:
2.869: Application time: 1.0001540 seconds
5.342: Application time: 0.0801231 seconds
8.382: Application time: 1.1013574 seconds
:
```

```
DoubleSummaryStatistics
{count=3, sum=2.181635, min=0.080123, average=0.727212,
max=1.101357}
```

Example: Processing GC Logfile

```
⋮
2.869: Application time: 1.0001540 seconds
5.342: Application time: 0.0801231 seconds
8.382: Application time: 1.1013574 seconds
⋮
```

Regex: `Application time: (\\d+\\.\\d+)`

Example: Processing GC Logfile

```
Pattern stoppedTimePattern =
            Pattern.compile(" Application time: (\\d+\\.\\d+)");

        ⋮

Matcher matcher = stoppedTimePattern.matcher(logRecord);
String value = matcher.group(1);
```

Processing GC Logfile: Old School Code

```
Pattern stoppedTimePattern =
        Pattern.compile("Application time: (\\d+\\.\\d+)");

String logRecord;
double value = 0;
while ( ( logRecord = logFileReader.readLine()) != null) {
  Matcher matcher = stoppedTimePattern.matcher(logRecord);
  if ( matcher.find()) {
    value += (Double.parseDouble( matcher.group(1)));
  }
}
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

```
Predicate<Matcher> matches =
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

```
Predicate<Matcher> matches = matcher
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

```
Predicate<Matcher> matches = matcher ->
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

```
Predicate<Matcher> matches = matcher -> matcher.find()
```

What is a Lambda?

```
Predicate<Matcher> matches = new Predicate<Matcher>() {
    @Override
    public boolean test(Matcher matcher) {
        return matcher.find();
    }
};
```

```
Predicate<Matcher> matches = matcher -> matcher.find()
```

A lambda is a function
from arguments to result

Processing Logfile: Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

Processing Logfile: Stream Code

data source

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

Processing Logfile: Stream Code

start streaming

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

Processing Logfile: Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

map to Matcher

Processing Logfile: Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

filter out uninteresting bits

Processing Logfile: Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

extract group

Processing Logfile: Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

map String to Double

Processing Logfile: Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines()
        .map(input -> stoppedTimePattern.matcher(input))
        .filter(matcher -> matcher.find())
        .map(matcher -> matcher.group(1))
        .mapToDouble(s -> Double.parseDouble(s))
        .summaryStatistics();
```

aggregate results

What is a Stream?

- A sequence of values
  - *source* and *intermediate operations* set the stream up lazily:

Source

```
Stream<String> groupStream =
    logFileReader.lines()
        .map(stoppedTimePattern::matcher)
        .filter(Matcher::find)
        .map(matcher -> matcher.group(1))
        .mapToDouble(Double::parseDouble);
```

What is a Stream?

- A sequence of values
  - *source* and *intermediate operations* set the stream up lazily:

Intermediate Operations

```
Stream<String> groupStream =
    logFileReader.lines()
        .map(stoppedTimePattern::matcher)
        .filter(Matcher::find)
        .map(matcher -> matcher.group(1))
        .mapToDouble(Double::parseDouble);
```

What is a Stream?

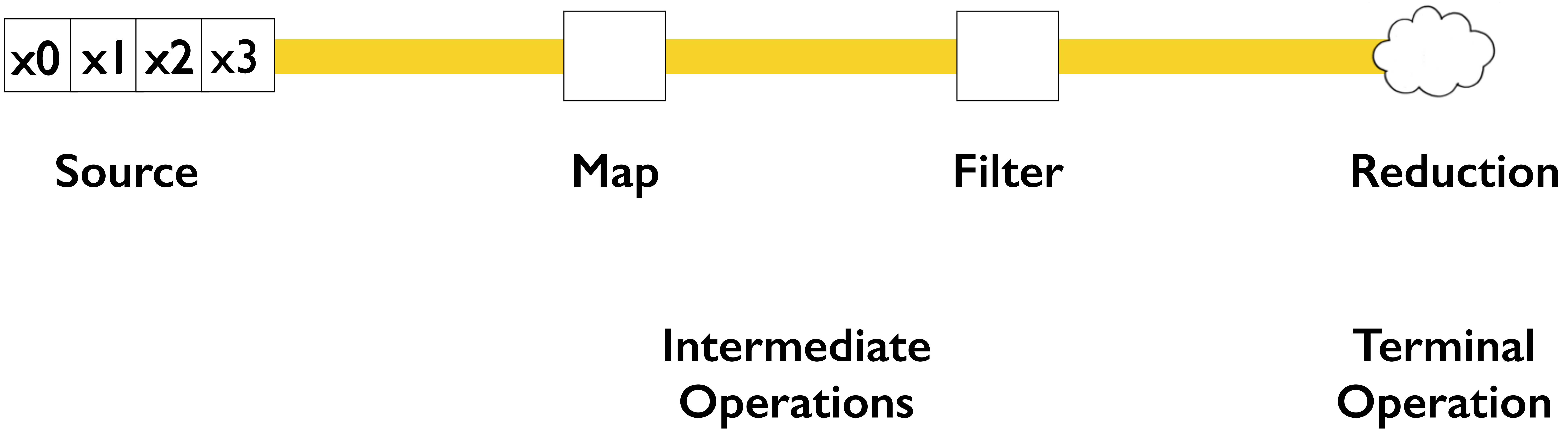
- The *terminal operation* pulls the values down the stream:

```
SummaryStatistics statistics =
    logFileReader.lines()
        .map(stoppedTimePattern::matcher)
        .filter(Matcher::find)
        .map(matcher -> matcher.group(1))
        .mapToDouble(Double::parseDouble)
        .summaryStatistics();
```

Terminal Operation

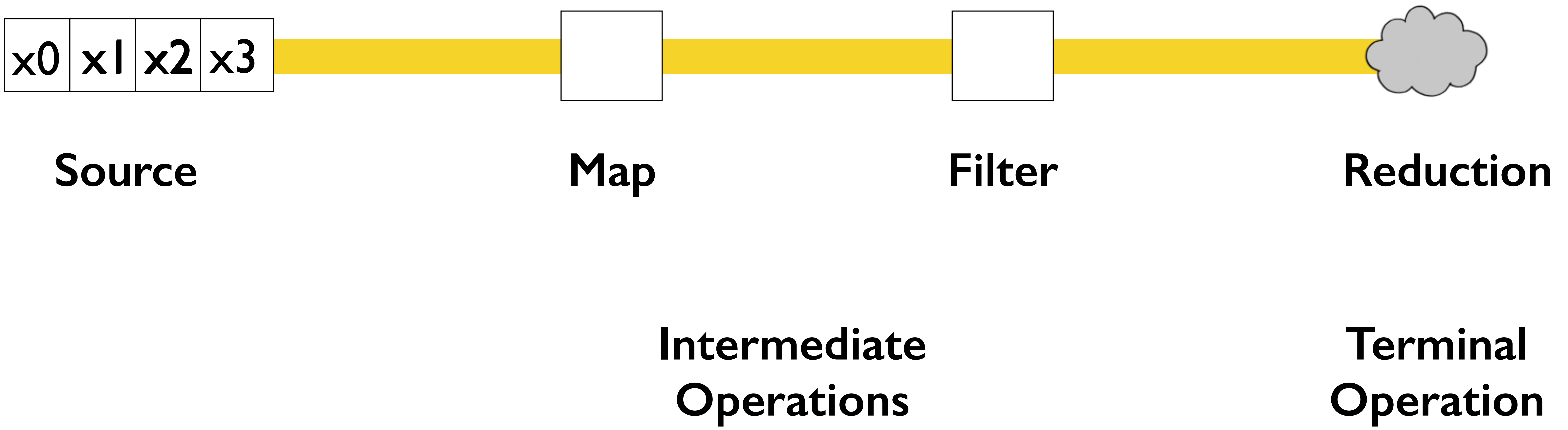
Visualising Sequential Streams

"Values in Motion"



Visualising Sequential Streams

"Values in Motion"



Visualising Sequential Streams

"Values in Motion"

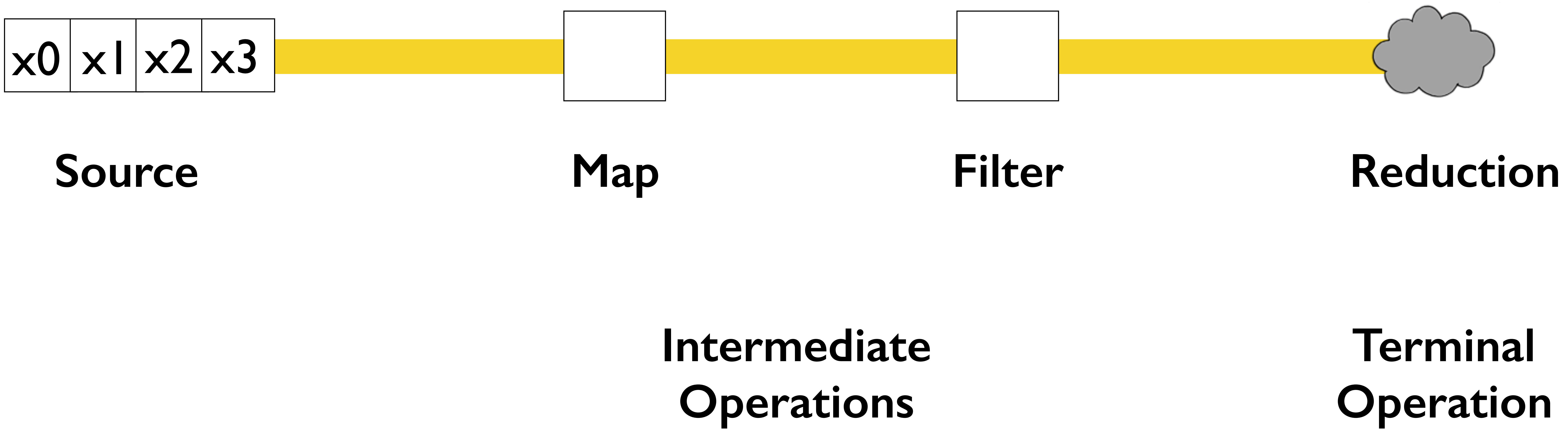
Source **Map** **Filter** **Reduction**

Intermediate Operations

Terminal Operation

Visualising Sequential Streams

"Values in Motion"



How Does That Perform?

Old School: 80200ms

Sequential: 25800ms

(>9m lines, MacBook Pro, Haswell i7, 4 cores, hyperthreaded)

Stream code is faster because operations are fused

Can We Do Better?

Parallel streams make use of multiple cores

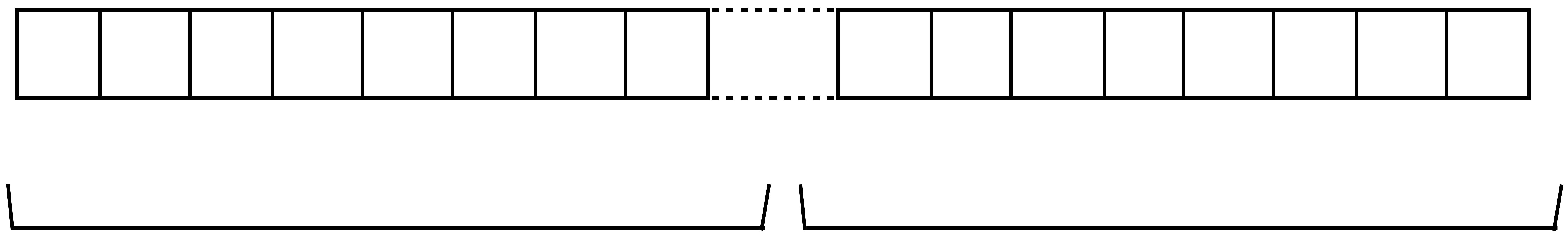
- split the data into segments
- each segment processed by its own thread
  - on its own core – if possible

Splitting the Data

Implemented by a `Spliterator`:

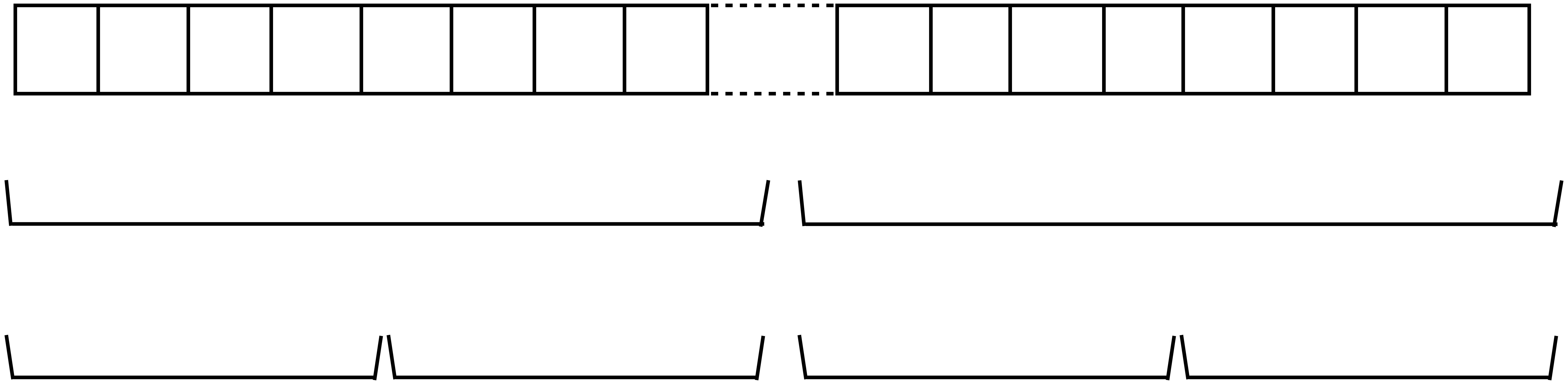
Splitting the Data

Implemented by a `Spliterator`:



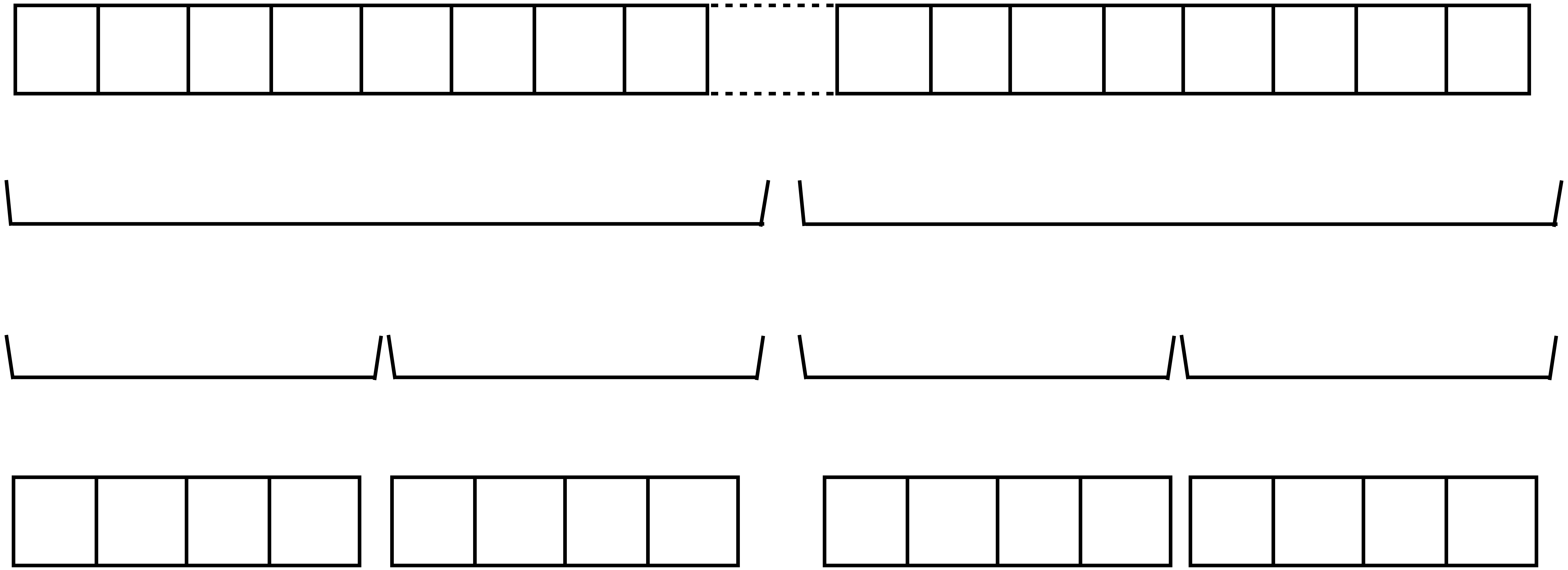
Splitting the Data

Implemented by a `Spliterator`:



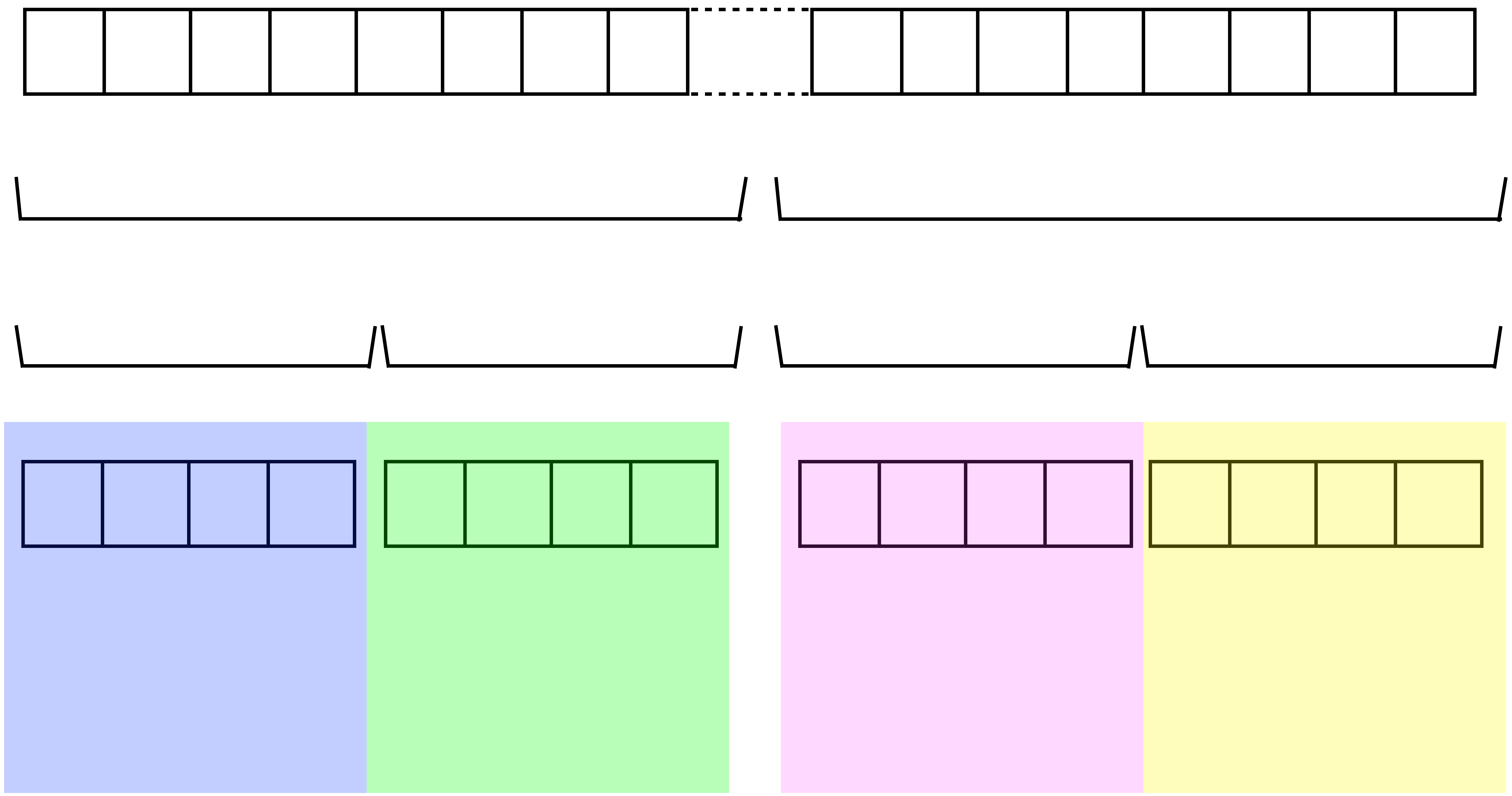
Splitting the Data

Implemented by a `Spliterator`:



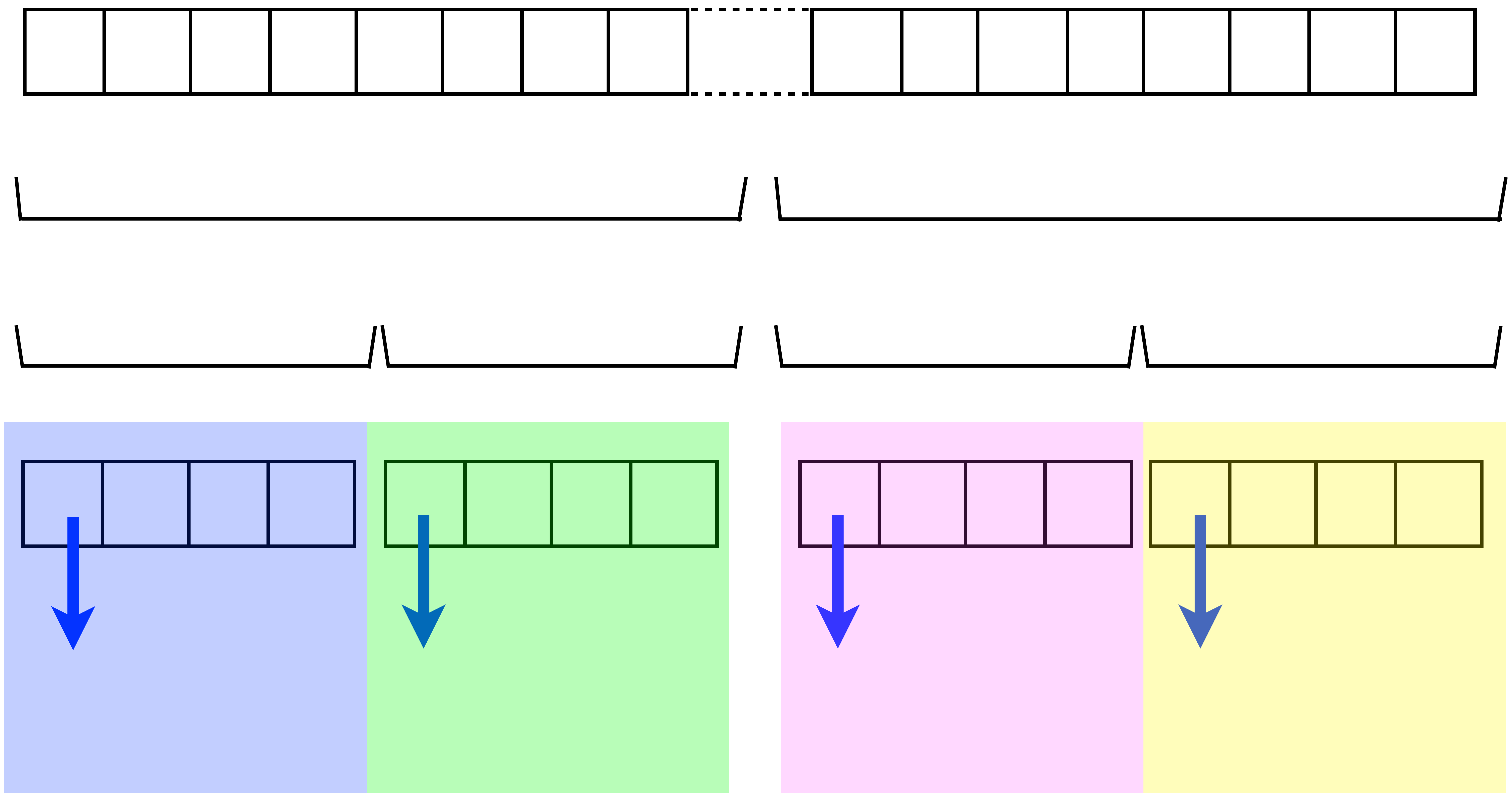
Splitting the Data

Implemented by a `Spliterator`:



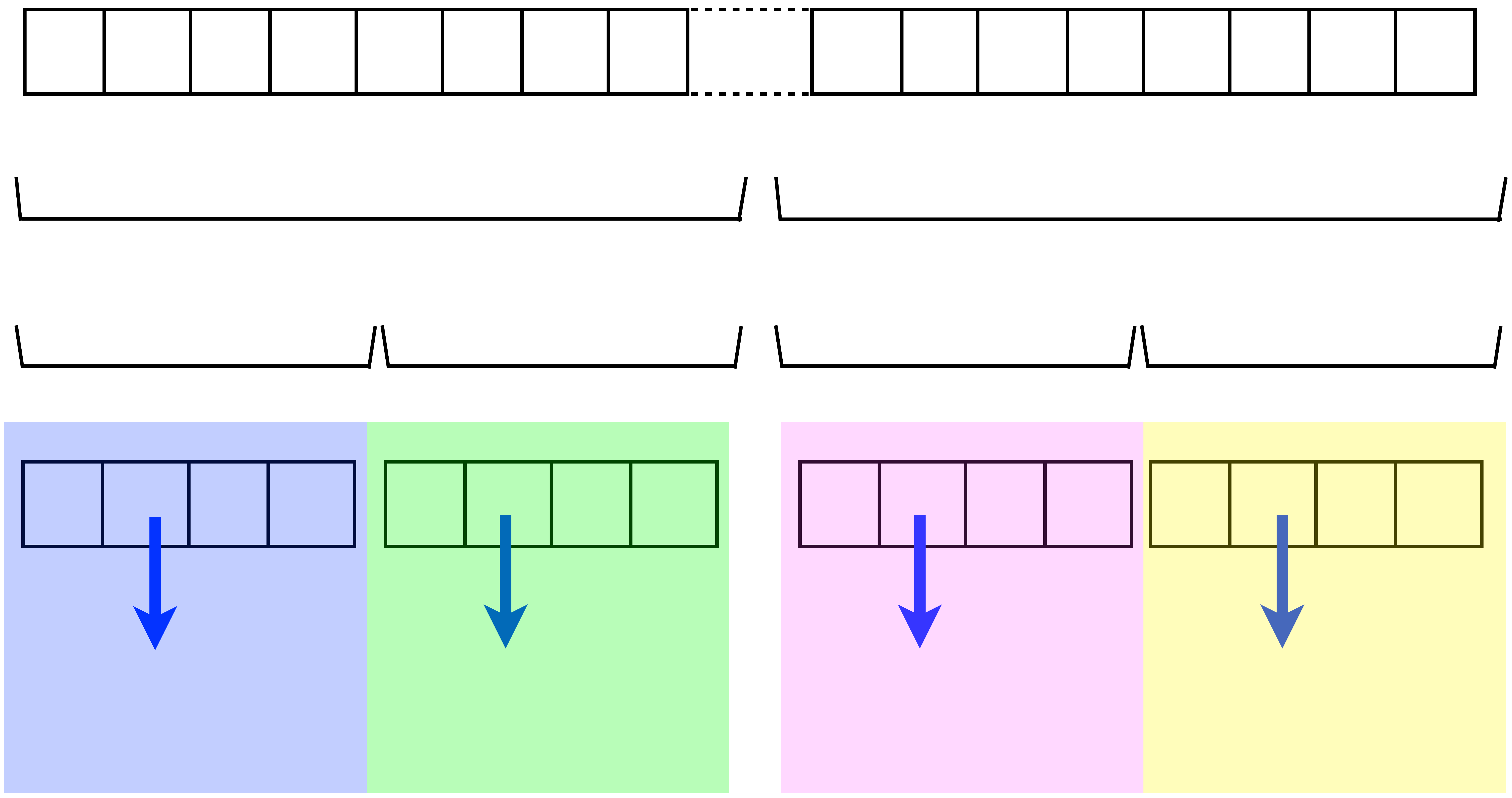
Splitting the Data

Implemented by a `Spliterator`:



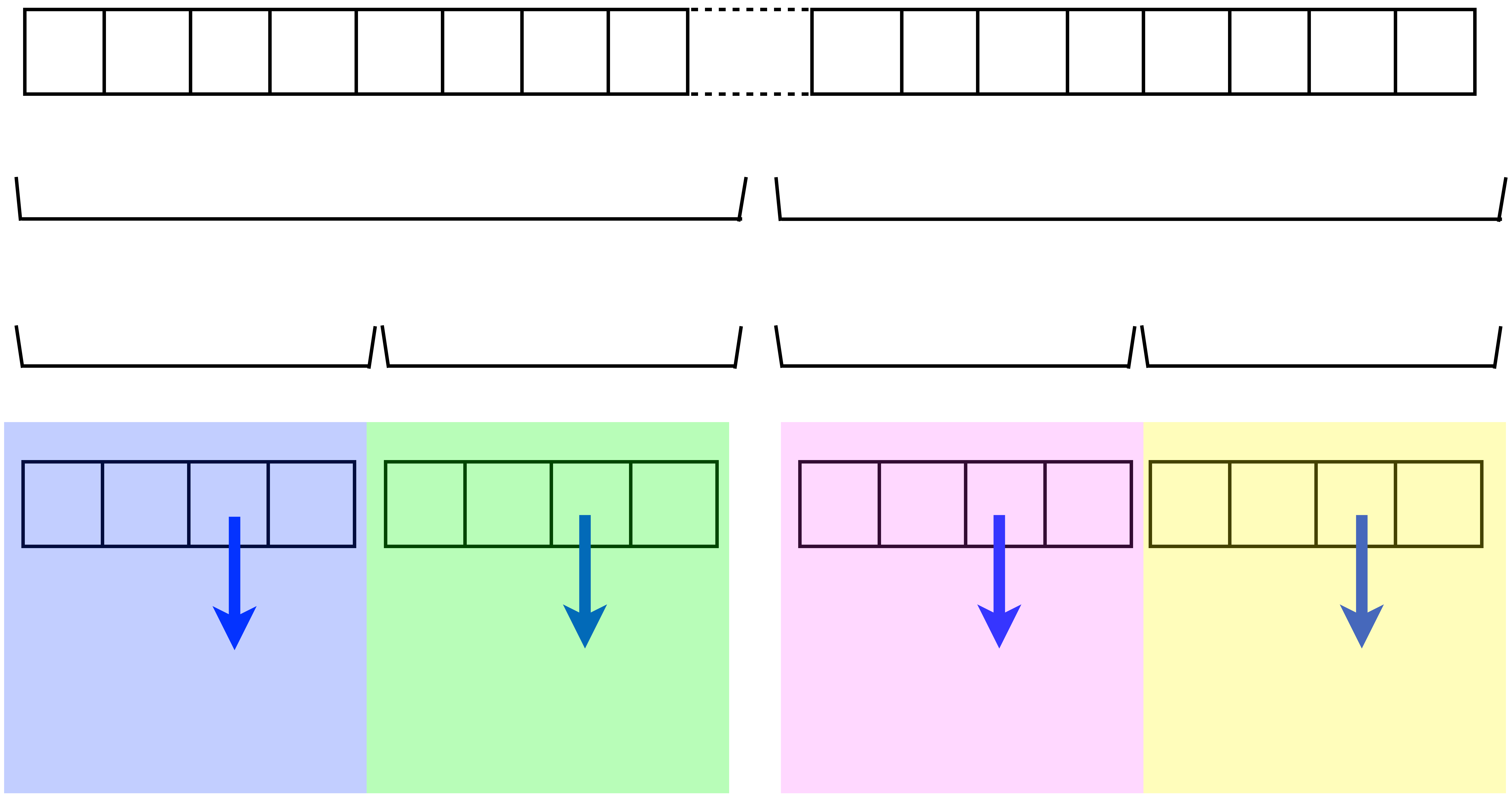
Splitting the Data

Implemented by a `Spliterator`:



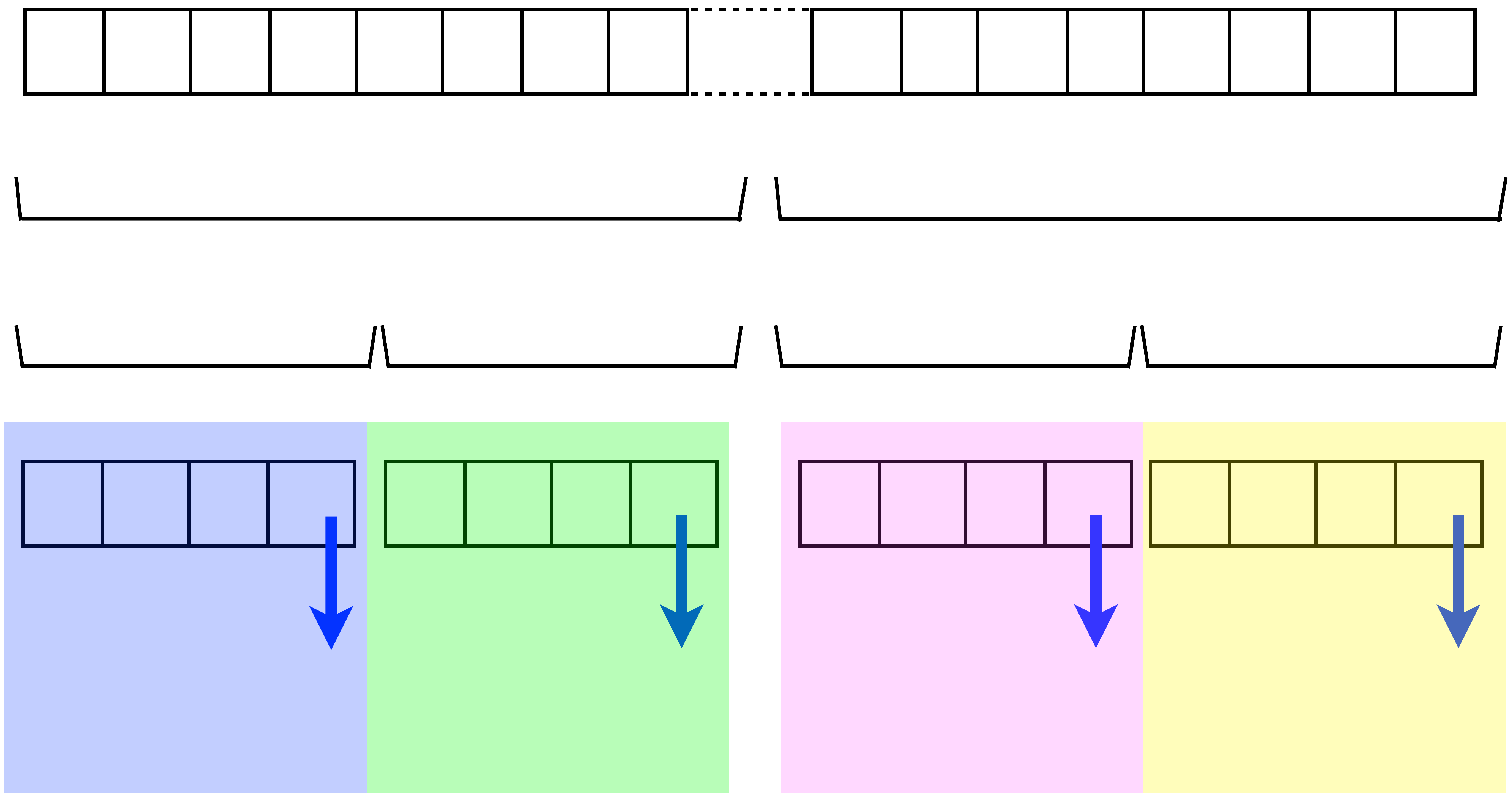
Splitting the Data

Implemented by a `Spliterator`:

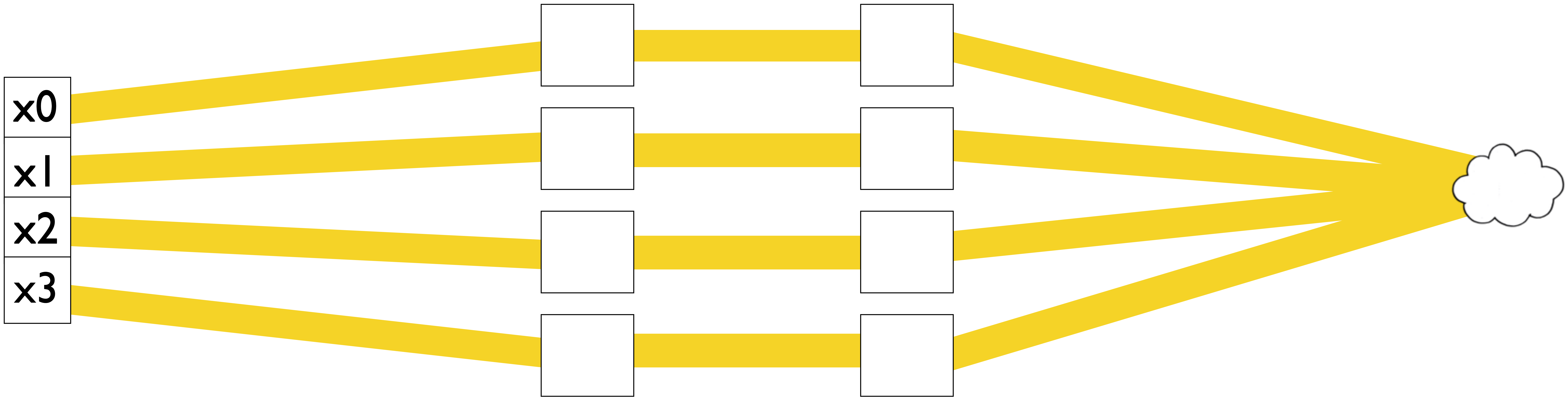


Splitting the Data

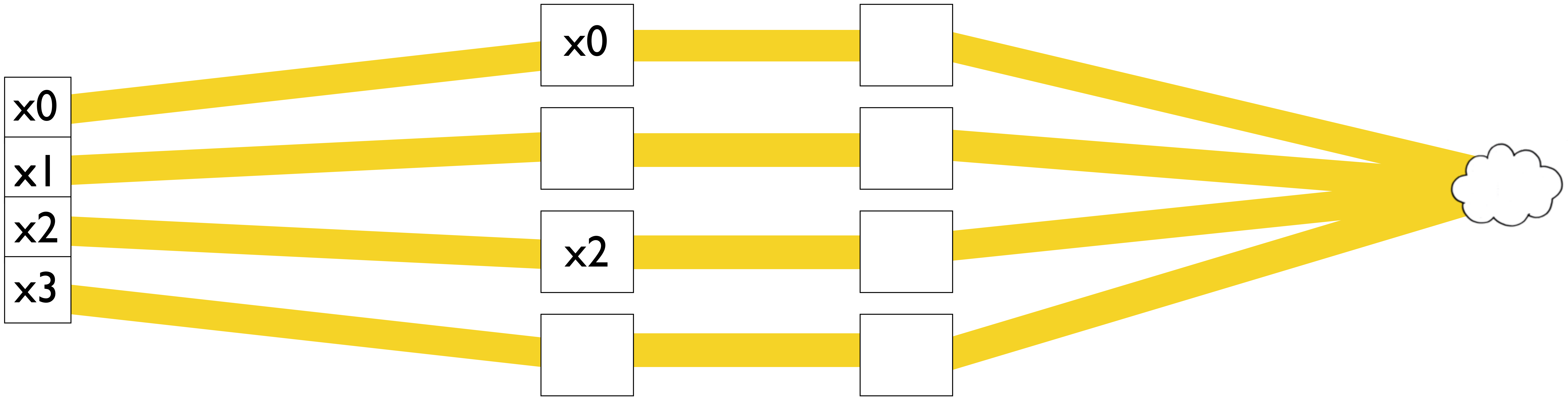
Implemented by a `Spliterator`:



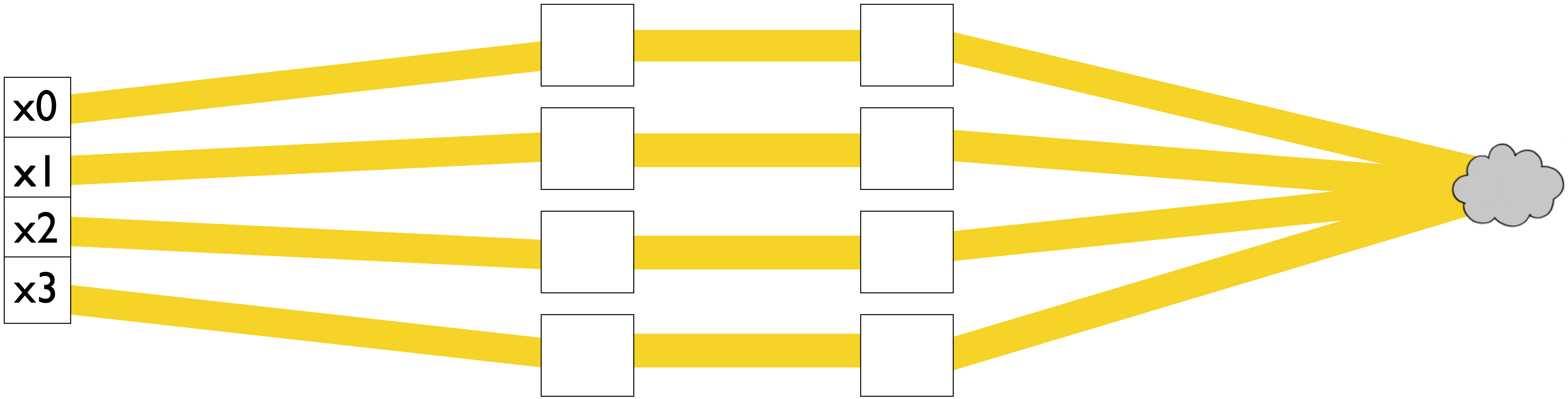
Visualizing Parallel Streams



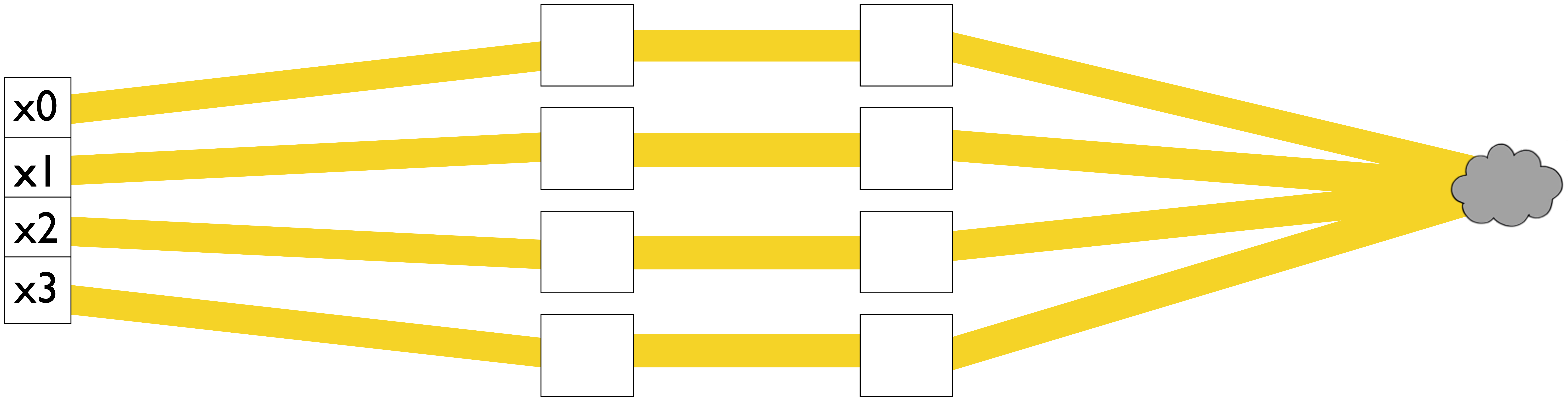
Visualizing Parallel Streams



Visualizing Parallel Streams



Visualizing Parallel Streams



Stream Code

```
DoubleSummaryStatistics summaryStatistics =
    logFileReader.lines().parallel()
        .map(stoppedTimePattern::matcher)
        .filter(Matcher::find)
        .map(matcher -> matcher.group(1))
        .mapToDouble(Double::parseDouble)
        .summaryStatistics();
```

Results of Going Parallel:

- No benefit from using parallel streams while streaming data

Agenda

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem
- Optimizing stream sources

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons
- Justifying the Overhead

Poorly Splitting Sources

Poorly Splitting Sources

- Some sources split much worse than others
  - `LinkedList` vs. `ArrayList`

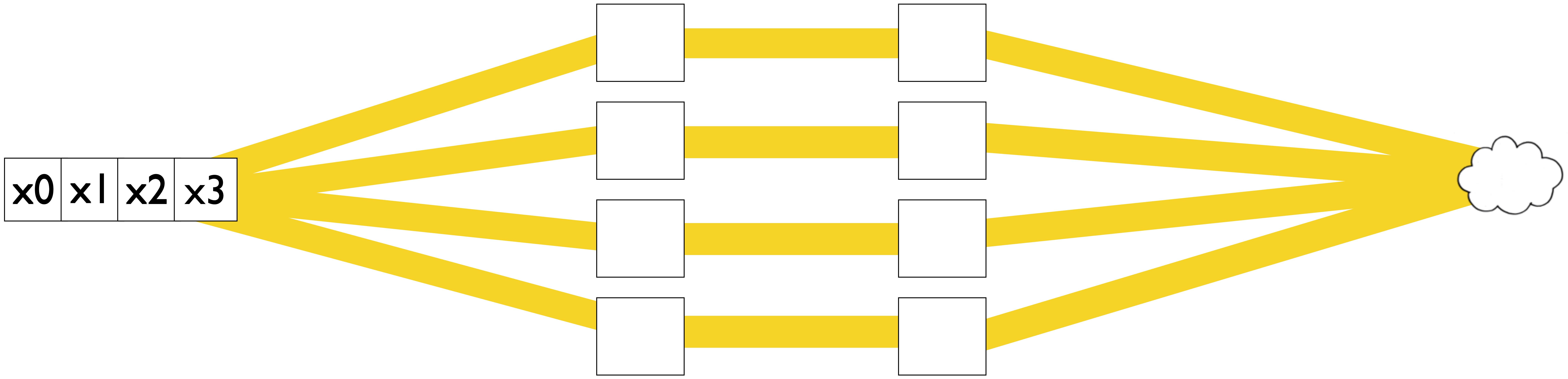
Poorly Splitting Sources

- Some sources split much worse than others
  - `LinkedList` vs. `ArrayList`
- Streaming I/O is bad.
  - kills the advantage of going parallel

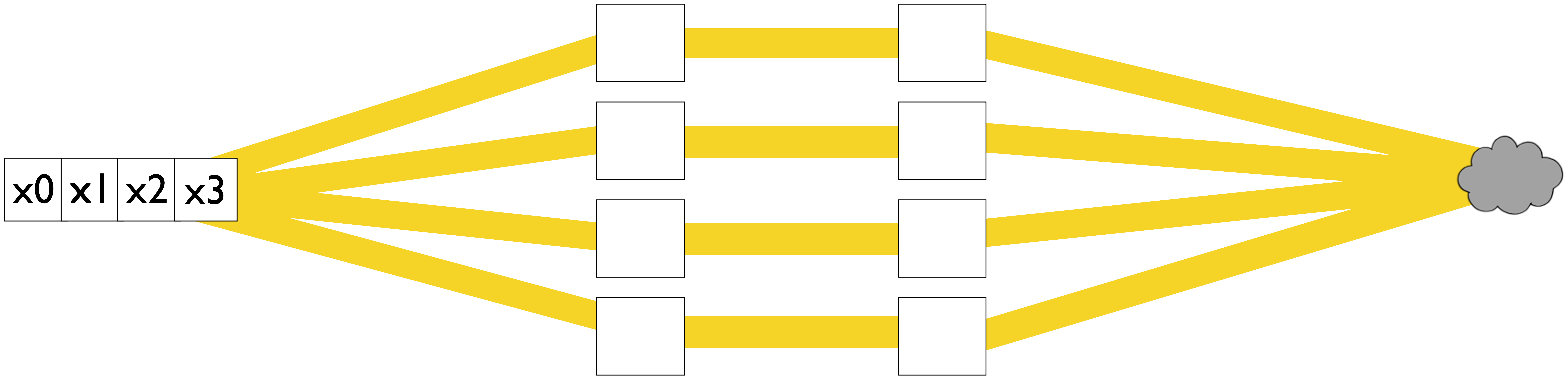
Poorly Splitting Sources

- Some sources split much worse than others
  - `LinkedList` vs. `ArrayList`
- Streaming I/O is bad.
  - kills the advantage of going parallel

Streaming I/O Bottleneck



Streaming I/O Bottleneck

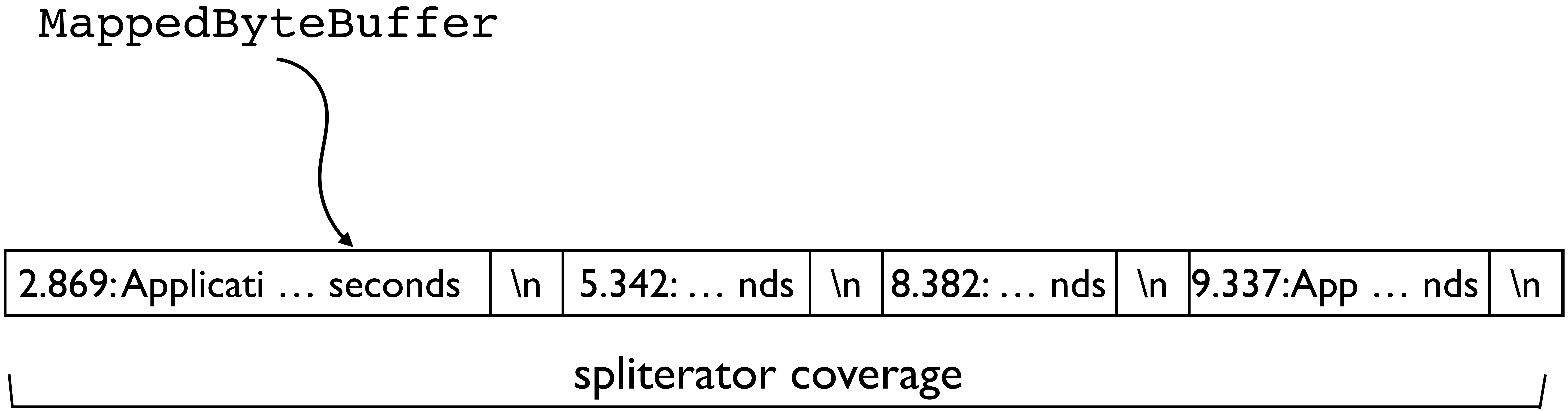


LineSpliterator

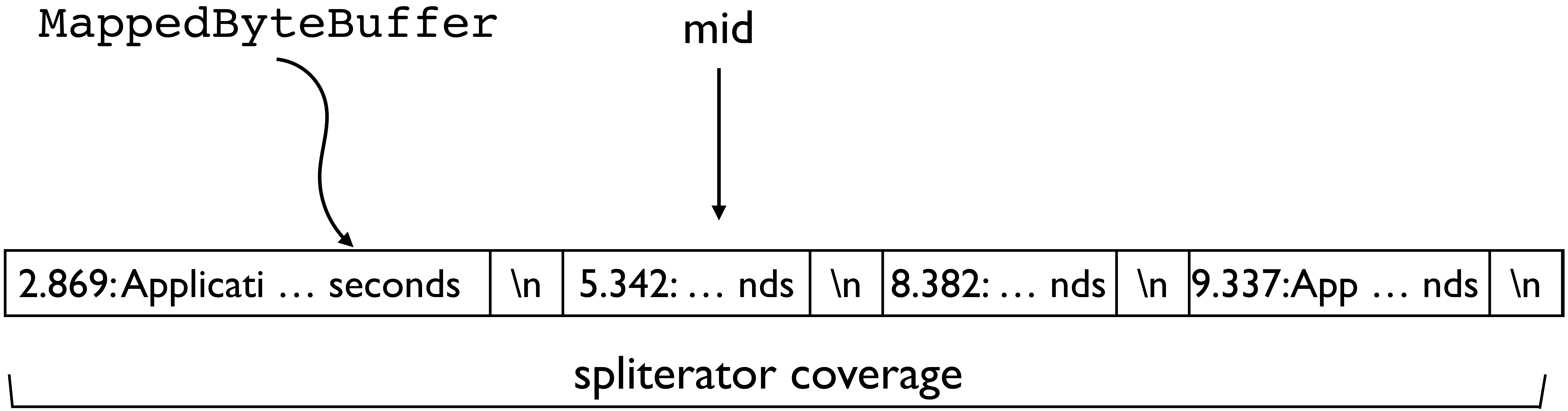
2.869: Applicati … seconds	\n	5.342: … nds	\n	8.382: … nds	\n	9.337:App … nds	\n

spliterator coverage

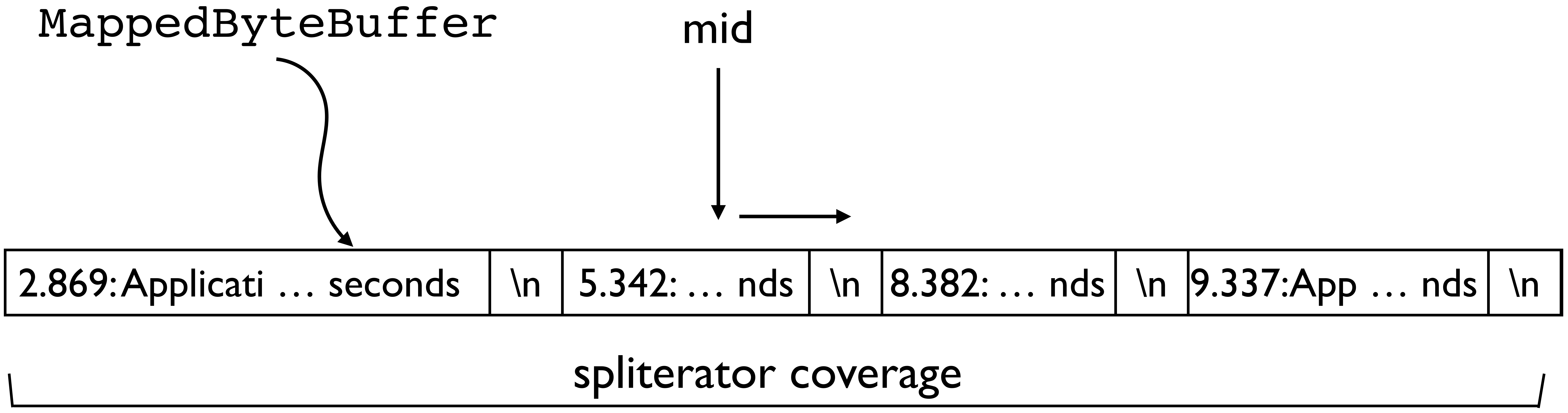
LineSpliterator



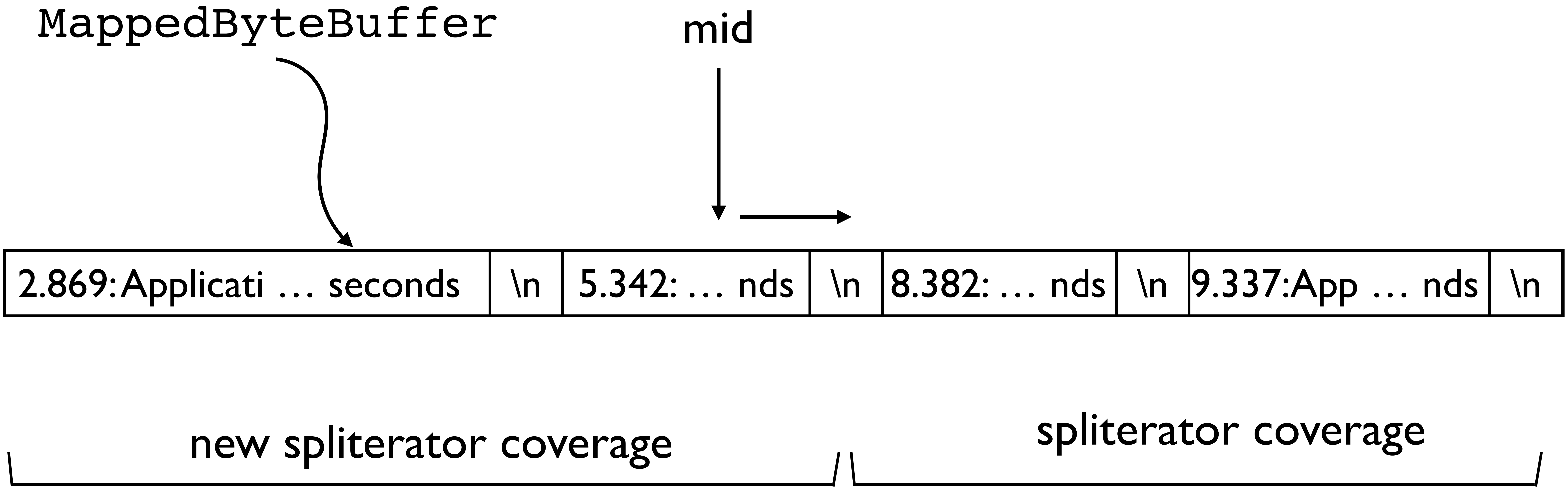
LineSpliterator



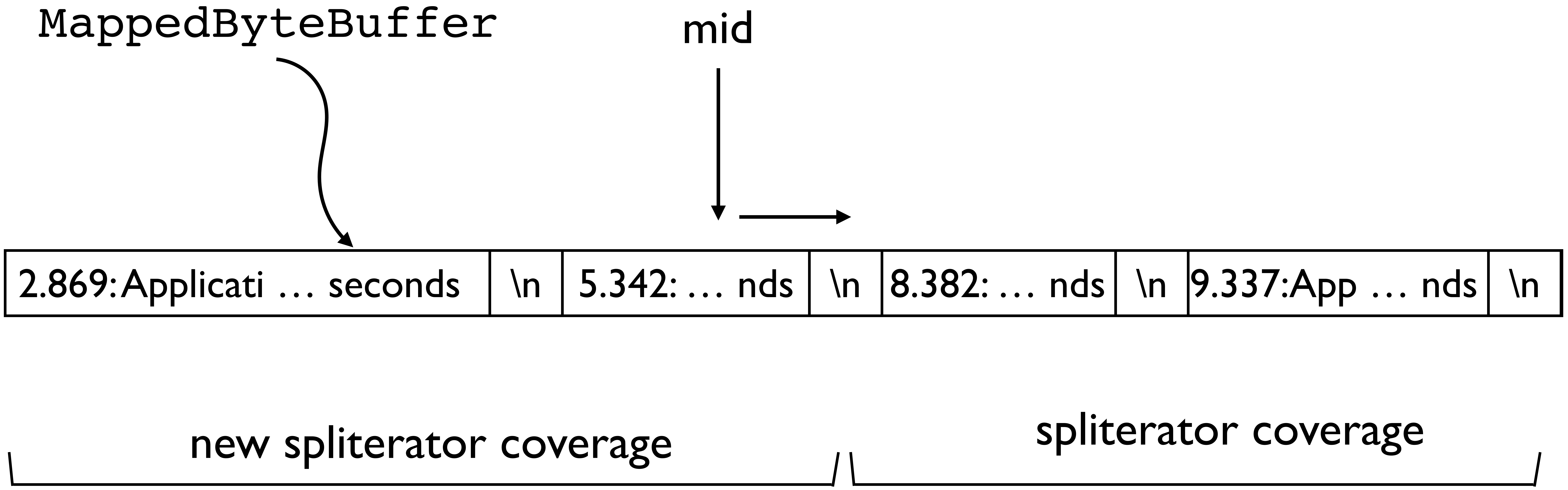
LineSpliterator



LineSpliterator



LineSpliterator



Included in JDK9 as `FileChannelLinesSpliterator`

`LineSpliterator` – results

StreamingIO: 56s

Spliterator: 88s

(>9m lines, MacBook Pro, Haswell i7, 4 cores, hyperthreaded)

Stream code is faster because operations are fused

When to Use Parallel Streams?

When to Use Parallel Streams?

- Task must be recursively decomposable
  - subtasks for each data segment must be independent

When to Use Parallel Streams?

- Task must be recursively decomposable
  - subtasks for each data segment must be independent
- Source must be well-splitting

When to Use Parallel Streams?

- Task must be recursively decomposable
  - – subtasks for each data segment must be independent
- Source must be well-splitting
- Enough hardware to support all VM needs
  - – there may be other business afoot

When to Use Parallel Streams?

- Task must be recursively decomposable
  - – subtasks for each data segment must be independent
- Source must be well-splitting
- Enough hardware to support all VM needs
  - – there may be other business afoot
- Overhead of splitting must be justified
  - – intermediate operations need to be expensive
  - – and CPU-bound

When to Use Parallel Streams?

- Task must be recursively decomposable
  - – subtasks for each data segment must be independent
- Source must be well-splitting
- Enough hardware to support all VM needs
  - – there may be other business afoot
- Overhead of splitting must be justified
  - – intermediate operations need to be expensive
  - – and CPU-bound

http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

Agenda

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem

Agenda

- Introduction
    - lambdas, streams, and a logfile processing problem
- Optimizing stream sources

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons
- Justifying the Overhead

Tragedy of the Commons

Tragedy of the Commons

You have a finite amount of hardware

- it might be in your best interest to grab it all
- but if everyone behaves the same way…

Agenda

Agenda

- Introduction
  - lambdas, streams, and a logfile processing problem

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem
- Optimizing stream sources

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons

Agenda

- Introduction
  - – lambdas, streams, and a logfile processing problem
- Optimizing stream sources
- Tragedy Of The Commons
- Justifying the Overhead

Justifying the Overhead

CPNQ performance model:

C - number of submitters

P - number of CPUs

N - number of elements

Q - cost of the operation

Justifying the Overhead

Need to amortize setup costs

– N*Q needs to be large

– Q can often only be estimated

– N often should be >10,000 elements

If P is the number of processors, the formula assumes that intermediate tasks are CPU bound

Don’t Have Too Many Threads!

- Too many threads cause frequent handoffs
- It costs ~80,000 cycles to handoff data between threads
- You can do a lot of processing in 80,000 cycles!

Fork/Join

Fork/Join

- Parallel streams implemented by Fork/Join framework
  - added in Java 7, but difficult to code
  - parallel streams are more usable

Fork/Join

- Parallel streams implemented by Fork/Join framework
  - added in Java 7, but difficult to code
  - parallel streams are more usable

Fork/Join

- Parallel streams implemented by Fork/Join framework
  - added in Java 7, but difficult to code
  - parallel streams are more usable

- Each segment of data is submitted as a `ForkJoinTask`
  - `ForkJoinTask.invoke()` spawns a new task
  - `ForkJoinTask.join()` retrieves the result

Fork/Join

- Parallel streams implemented by Fork/Join framework
  - added in Java 7, but difficult to code
  - parallel streams are more usable
- Each segment of data is submitted as a `ForkJoinTask`
  - `ForkJoinTask.invoke()` spawns a new task
  - `ForkJoinTask.join()` retrieves the result
- How Fork/Join works and performs is important to your latency picture

Common Fork/Join Pool

Fork/Join by default uses a common thread pool

- default number of worker threads == number of logical cores - 1
  - (submitting thread is pressed into service)
- can configure the pool via system properties:

```
java.util.concurrent.ForkJoinPool.common.parallelism
java.util.concurrent.ForkJoinPool.common.threadFactory
java.util.concurrent.ForkJoinPool.common.exceptionHandler
```

- or create our own pool…

Custom Fork/Join Pool

When used inside a `ForkJoinPool`, the `ForkJoinTask.fork()` method uses the *current* pool:

```
ForkJoinPool ourOwnPool = new ForkJoinPool(10);

ourOwnPool.invoke(
     () -> stream.parallel().
                   ⋮
```

Don’t Have Too Few Threads!

- Fork/Join pool uses a work queue
  - If tasks are CPU bound, no use increasing the size of the thread pool
- But if not CPU bound, they are sitting in queue accumulating dead time
  - Can make thread pool bigger to reduce dead time
  - Little’s Law tells us

Number of tasks in the system =
Arrival rate * Average service time

Little's Law Example

System receives 400 Txs and it takes 100ms to clear a request

- Number of tasks in system = 0.100 * 400 = 40

On an 8 core machine with a CPU bound task

- implies 32 tasks are sitting in queue accumulating dead time
- Average response time 600 ms of which 500ms is dead time
  - ~83% of service time is in waiting

ForkJoinPool Observability

`ForkJoinPool` comes with no visibility

- need to instrument `ForkJoinTask.invoke()`
  - gather data from `ForkJoinPool` to feed into Little's Law

```
public final V invoke() {
    ForkJoinPool.common.getMonitor().submitTask(this);
    int s;
    if ((s = doInvoke() & DONE_MASK) != NORMAL) reportException(s);
    ForkJoinPool.common.getMonitor().retireTask(this);
    return getRawResult();
}
```

Conclusions

Sequential stream performance comparable to imperative code

Going parallel is worthwhile IF

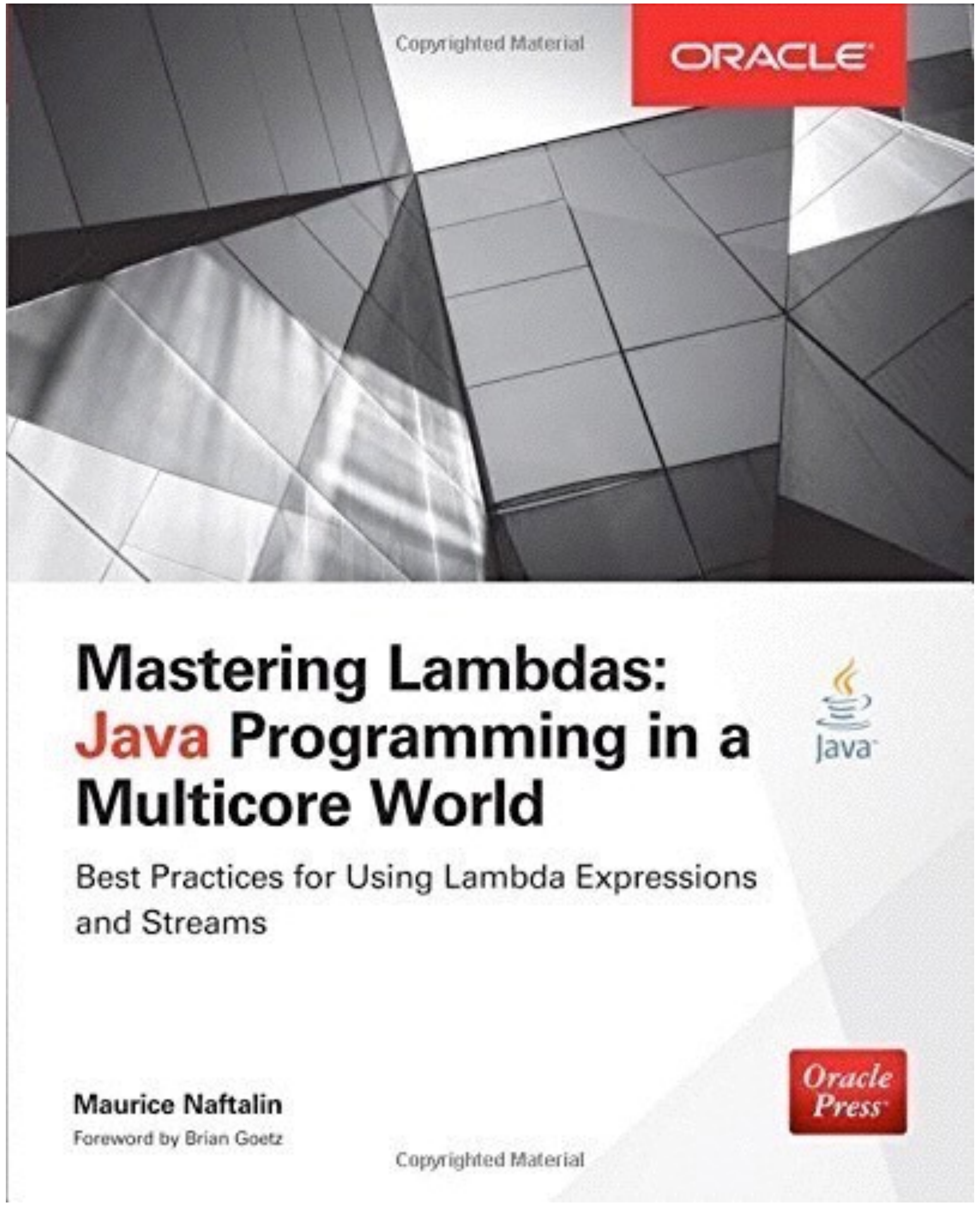
- task is suitable
- data source is suitable
- environment is suitable

Need to monitor JDK to understanding bottlenecks

- Fork/Join pool is not well instrumented

Questions?

Questions?


Copyrighted Material
ORACLE
Mastering Lambdas:
Java Programming in a
Multicore World
Best Practices for Using Lambda Expressions
and Streams
Java
Maurice Naftalin
Foreword by Brian Goetz
Oracle
Press
Copyrighted Material