

We are all to gather

Rémi Forax

Université Gustave Eiffel – June 2024

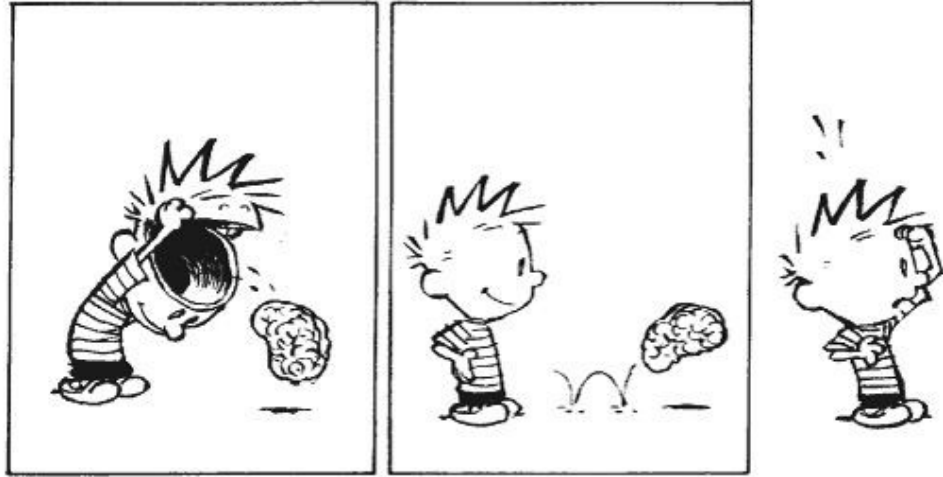
We are all together

Rémi Forax
Université Gustave Eiffel – June 2024

We are all to **gather**

Rémi Forax

Université Gustave Eiffel – June 2024



CALVIN & HOBBS © BIL WATTERSON

Don't believe what I'm saying !

Me, myself and I

Rémi Forax

- Assistant Prof at University Gustave Eiffel
- Expert for lambda, module, record, etc

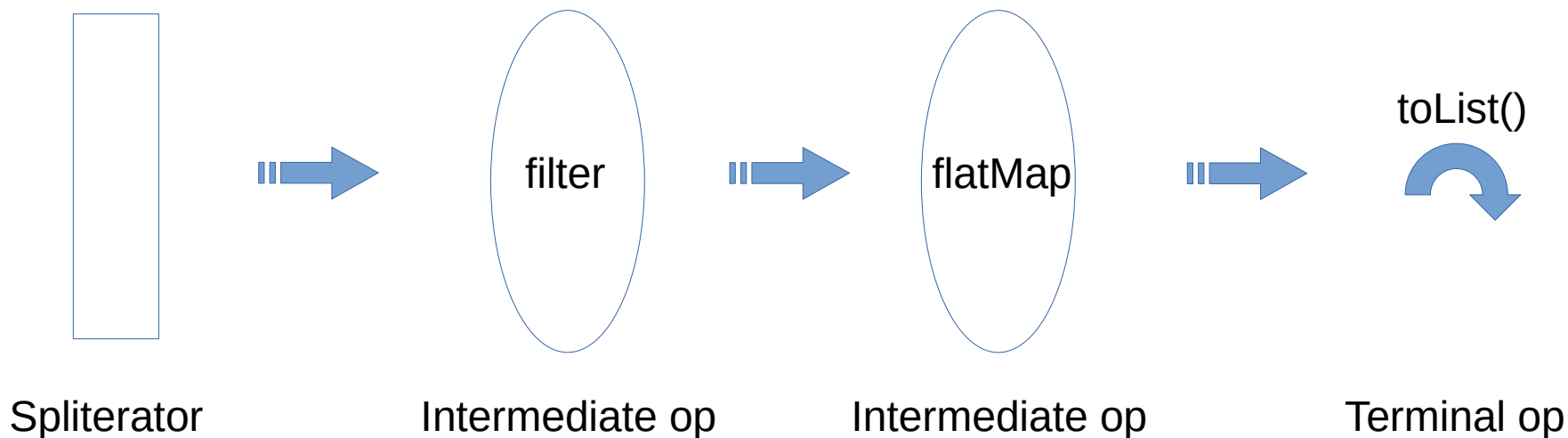
Feel free to google me if you want to know more ...

Outline

- The Gatherer API
- Performance and limitations

A stream is defined by ...

- a source, abstracted as a Splitterator
- some intermediate operations
- a terminal operation that drives the pipeline



A Gatherer



Abstracts any intermediate operation

- Users can define their own

New intermediate method `Stream.gather()`

- `stream.gather(Gatherers.fold(...)).toList()`

Modeled like a Collector

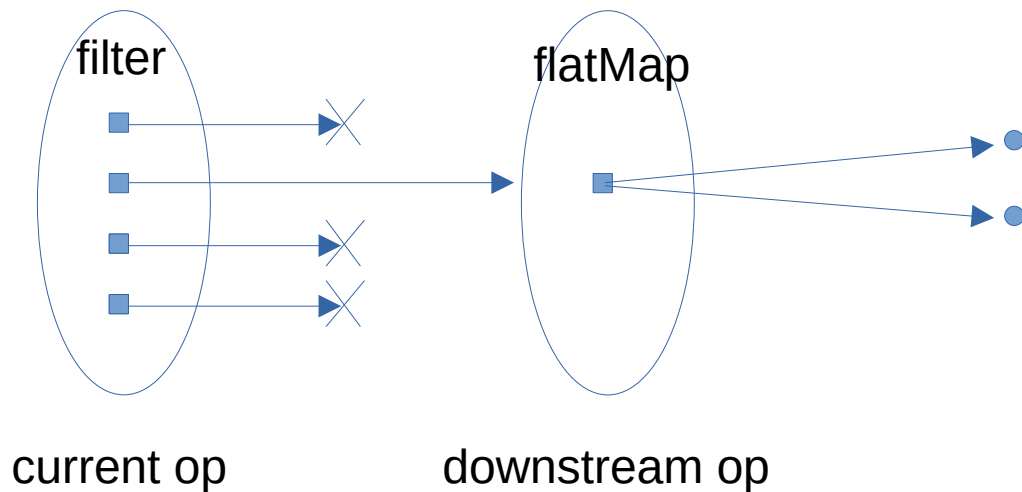
- `j.u.s.Gatherers` contains predefined Gatherer

An intermediate operation

Push (or not) transformed elements to the next stage

- Back-propagate “I want more !”

May have state (limit, distinct, sorted, ...)

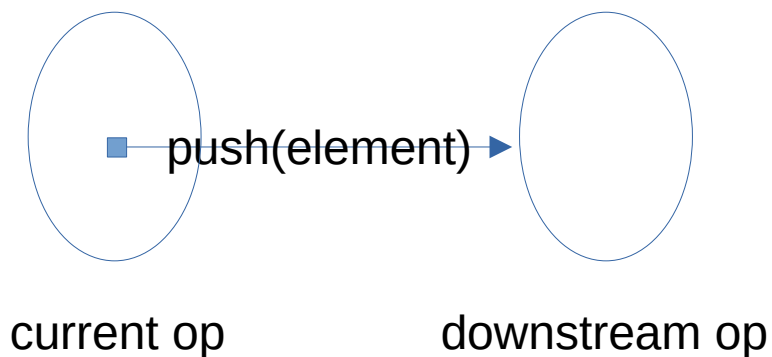


Modeling an intermediate op

```
@FunctionalInterface  
interface Integrator {
```

```
    boolean integrate (???, E element, Downstream<T>  
        downstream)
```

- May push downstream + “I want more”



Interface Downstream<T>

Conceptually a Consumer<T>

- boolean push(T element)
 - Returns true → “I want more”
- isRejecting()
 - Is the next stage rejecting elements (will return false) ?

+ State management

Gatherer.of() uses 4 functions

initializer: Supplier<A>

- Create a state

integrator: (A state, E element, Downstream<T> downstream) → boolean

- Modify state and/or push downstream (+ back-propagate return type)

combiner: BinaryOperator<A>

- Combine two states, return a new state

finisher: BiConsumer<A, Downstream<T>>

- Push downstream at the end

```
interface Downstream<T> {  
    boolean push(T element);  
    boolean isRejecting();  
}
```

Live Code !

Gatherer API

Gatherer<E, A, T>

initializer: Supplier<A>

- Create a state

integrator (A state, E element, Downstream<T> downstream) → boolean

- Modify state and/or push downstream (+ back-propagate return type)

combiner: BinaryOperator<A>

- Combine two states, return a new state

finisher: BiConsumer<A, Downstream<T>>

- Push downstream at the end

```
interface Downstream<T> {  
    boolean push(T element);  
    boolean isRejecting();  
}
```

Creating a Gatherer : 3 axis

Only Sequential vs Parallelizable

- Gatherer.ofSequential() vs Gatherer.of() + combiner?

Stateless vs Stateful

- integrator vs initializer + integrator + finisher?

Short-circuit vs Greedy

- integrator vs Integrator.ofGreedy()

Greedy vs short-circuit

If a Gatherer does not short-circuit

- It can be merged with a subsequent gatherer
- It can be merged with a subsequent collector

Wraps the Integrator into `Greedy.of(integrator)`

Performance ?

WARNING ! WARNING !

The stream API relies heavily on the VM being able to correctly propagate type profiles

- If that strategy does not work, the VM records type profiles

There is usually no profile pollution in a micro-benchmark so real world scenario performance may differ !

Performance map + sum

```
public int stream_map_sum() {  
    return values.stream().map(String::length).reduce(0, Integer::sum);  
}    // 481.222 ± 1.560 us/op  
  
public int stream_mapToInt_sum() {  
    return values.stream().mapToInt(String::length).sum();  
}    // 102.089 ± 0.672 us/op  
  
public int gatherer_map_sum() {  
    return values.stream().gather(map(String::length)).reduce(0, Integer::sum);  
}    // 552.384 ± 3.405 us/op
```

No primitive specialization ...

* `values` is a List of 100_000 strings

Performance map + toList

```
public List<Integer> stream_map_toList() {  
    return values.stream().map(String::length).toList();  
}    // 332.322 ± 0.512 us/op  
  
public List<Integer> gatherer_map_toList() {  
    return values.stream().gather(map(String::length)).toList();  
}    // 558.873 ± 6.200 us/op
```

Why using a Gatherer is slower ? ...

Performance map + count

```
public long stream_map_count() {  
    return values.stream().map(String::length).count();  
}    // 0.009 ± 0.001 us/op  
  
public long stream_mapToInt_count() {  
    return values.stream().mapToInt(String::length).count();  
}    // 0.009 ± 0.001 us/op  
  
public long gatherer_map_count() {  
    return values.stream().gather(map(String::length)).count();  
}    // 101.993 ± 0.105 us/op
```

=> spliterator characteristics are not propagated !

Performance map + collect(summing)

```
public int stream_map_collect() {  
    return integers.stream().map(v -> v + 1).collect(summingInt(v -> v));  
}    // 311.241 ± 62.204 us/op  
  
public int gatherer_map_collect() {  
    return integers.stream().gather(map(v -> v + 1)).collect(summingInt(v -> v));  
}    // 307.455 ± 0.222 us/op
```

gather + collect are fused into one operation, yeah !

* `integers` is a List of 100_000 Integers

Performance issues

No primitive specialization

- mapToInt/flatMapToInt, etc
 - Same issue with collectors
 - Valhalla generics to the rescue ??

Splititerator characteristics are not propagated

- Same issue with collectors
 - For ex: Stream.toList() can pre-size, not Collectors.toList()

Executive Summary

Gatherer API

User defined intermediary operations

- 4 functions: initializer, integrator, combiner, finisher
- 3 axis: only-sequential / statefulness / short-circuitability

Still in preview in Java 23

- more predefined Gatherers ?
- Splititerator characteristics could be propagated ?

Questions ?

https://github.com/forax/we_are_all_to_gather