elixir

@elixirlang / elixir-lang.org

# GenStage & Flow

github.com/elixir-lang/gen_stage

## Prelude:
## From eager,
## to lazy,
## to concurrent,
## to distributed

## Example problem: word counting

```
“roses are red\n
violets are blue\n"
```

$\downarrow$

```
%{“are” => 2,
  “blue” => 1,
  “red” => 1,
  “roses" => 1,
  “violets" => 1}
```

# Eager

```
File.read!("source")
```

↓

```
"roses are red\n
 violets are blue\n"
```

```
File.read!("source")
|> String.split("\n")
```

↓

```
["roses are red",
 "violets are blue"]
```

```
File.read!("source")
|> String.split("\n")
|> Enum.flat_map(&String.split/1)
```

↓

```
["roses", “are", “red",
 "violets", "are", "blue"]
```

```
File.read!("source")
|> String.split("\n")
|> Enum.flat_map(&String.split/1)
|> Enum.reduce(%{}, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```

↓

```
         %{“are” => 2,
           “blue” => 1,
           “red” => 1,
           “roses" => 1,
           “violets" => 1}
```

## Eager

- Simple
- Efficient for small collections
- Inefficient for large collections with multiple passes

```
File.read!(“really large file")
|> String.split("\n")
|> Enum.flat_map(&String.split/1)
```

# Lazy

```elixir
File.stream!("source", :line)
```

↓

```
#Stream<...>
```

```
File.stream!("source", :line)
|> Stream.flat_map(&String.split/1)
```

↓

```
#Stream<...>
```

```
File.stream!("source", :line)
|> Stream.flat_map(&String.split/1)
|> Enum.reduce(%{}, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```

```
%{“are” => 2,
  “blue” => 1,
  “red” => 1,
  “roses" => 1,
  “violets" => 1}
```

## Lazy

- Folds computations, goes item by item
- Less memory usage at the cost of computation
- Allows us to work with large or infinite collections

## Concurrent

```
File.stream!("source", :line)
```

↓

```
#Stream<...>
```

```elixir
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
|> Flow.partition()
|> Flow.reduce(fn -> %{} end, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```

↓

```
#Flow<...>
```

```
File.stream!("source", :line)
 |> Flow.from_enumerable()
 |> Flow.flat_map(&String.split/1)
 |> Flow.partition()
 |> Flow.reduce(fn -> %{} end, fn word, map ->
   Map.update(map, word, 1, & &1 + 1)
end)
 |> Enum.into(%{})
```

↓

```
%{“are” => 2,
  “blue” => 1,
  “red” => 1,
  “roses" => 1,
  “violets" => 1}
```

## Flow

- We give up ordering and process locality for concurrency
- Tools for working with bounded and unbounded data

## Flow

- It is not magic! There is an overhead when data flows through processes
- Requires volume and/or cpu/io-bound work to see benefits

## Flow stats

- ~1200 lines of code (LOC)
- ~1300 lines of documentation (LOD)

## Topics

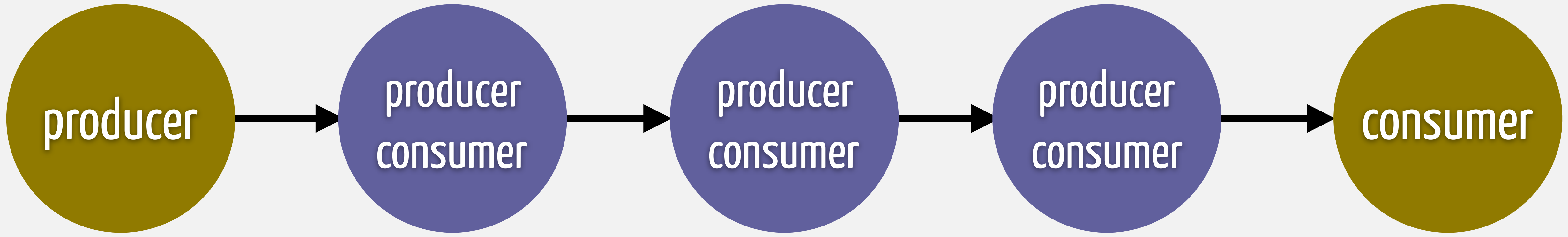
- 1200LOC: How is flow implemented?
- 1300LOD: How to reason about flows?

# GenStage

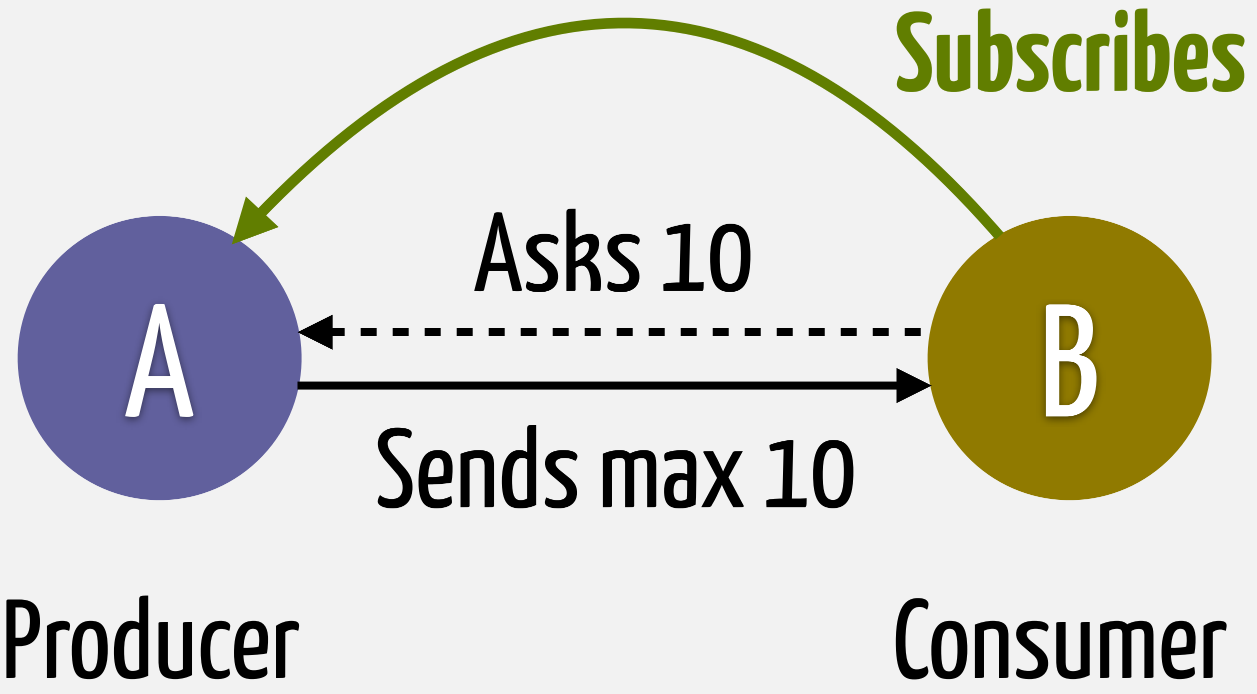
# GenStage

- It is a new behaviour
- Exchanges data between stages transparently with back-pressure
- Breaks into producers, consumers and producer_consumers

# GenStage

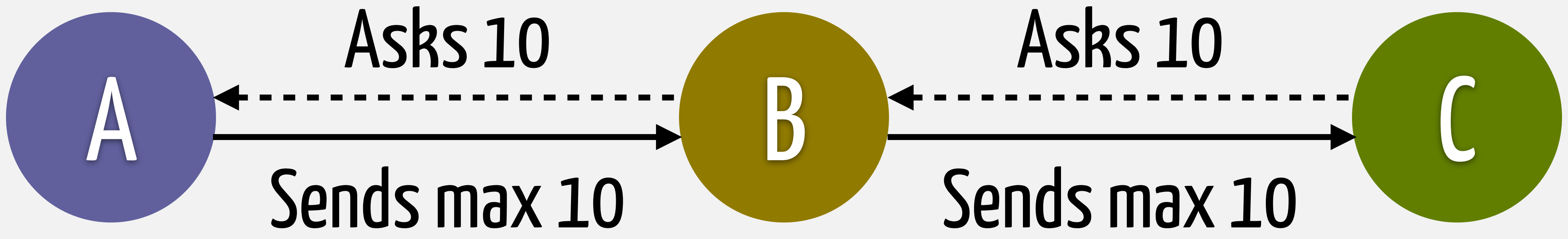

producer
producer
consumer
producer
consumer
producer
consumer
consumer

# GenStage: Demand-driven



1. consumer subscribes to producer
2. consumer sends demand
3. producer sends events

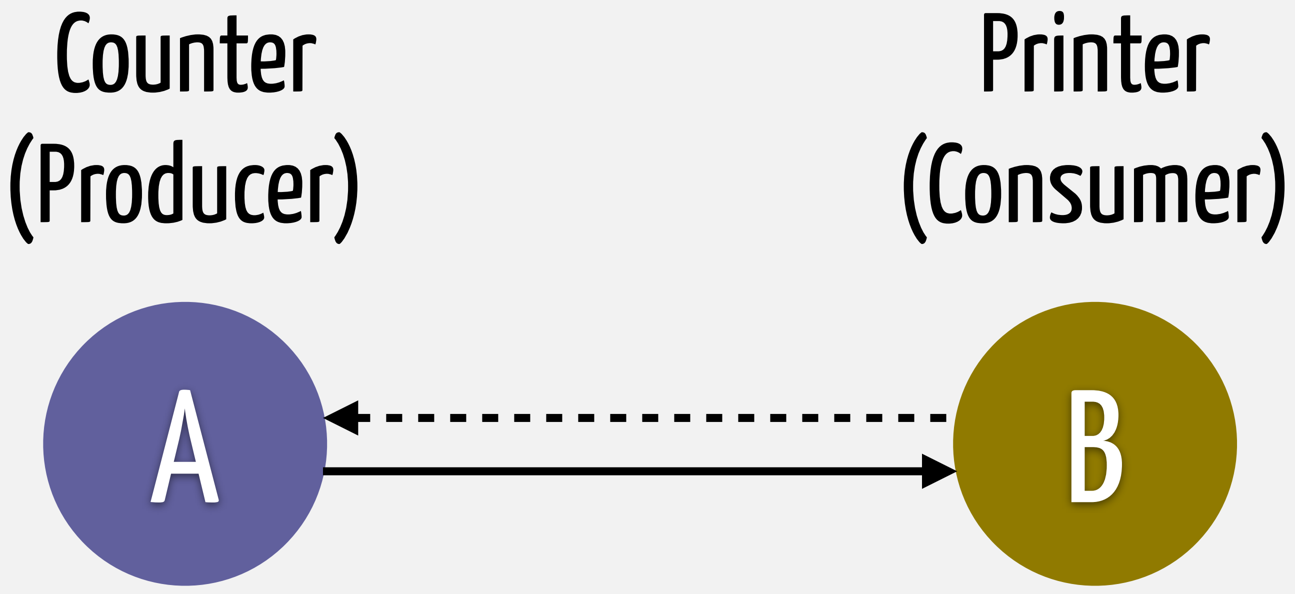
# GenStage: Demand-driven



## GenStage: Demand-driven

- It is a message contract
- It pushes back-pressure to the boundary
- GenStage is one impl of this contract

# GenStage example


Counter
(Producer)
Printer
(Consumer)
A
B

```elixir
defmodule Producer do
  use GenStage

  def init(counter) do
    {:producer, counter}
  end

  def handle_demand(demand, counter) when demand > 0 do
    events = Enum.to_list(counter..counter+demand-1)
    {:noreply, events, counter + demand}
  end
end
```

| state | demand | handle_demand |
|---|---|---|
| 0 | 10 | {:noreply, [0, 1, …, 9], 10} |
| 10 | 5 | {:noreply, [10, 11, 12, 13, 14], 15} |
| 15 | 5 | {:noreply, [15, 16, 17, 18, 19], 20} |

```elixir
defmodule Consumer do
  use GenStage

  def init(:ok) do
    {:consumer, :the_state_does_not_matter}
  end

  def handle_events(events, _from, state) do
    Process.sleep(1000)
    IO.inspect(events)
    {:noreply, [], state}
  end
end
```

```
{:ok, counter} =
  GenStage.start_link(Producer, 0)

{:ok, printer} =
  GenStage.start_link(Consumer, :ok)

GenStage.sync_subscribe(printer, to: counter)
```

```
(wait 1 second)
[0, 1, 2, ..., 499] (500 events)
(wait 1 second)
[500, 501, 502, ..., 999] (500 events)
```

## Subscribe options

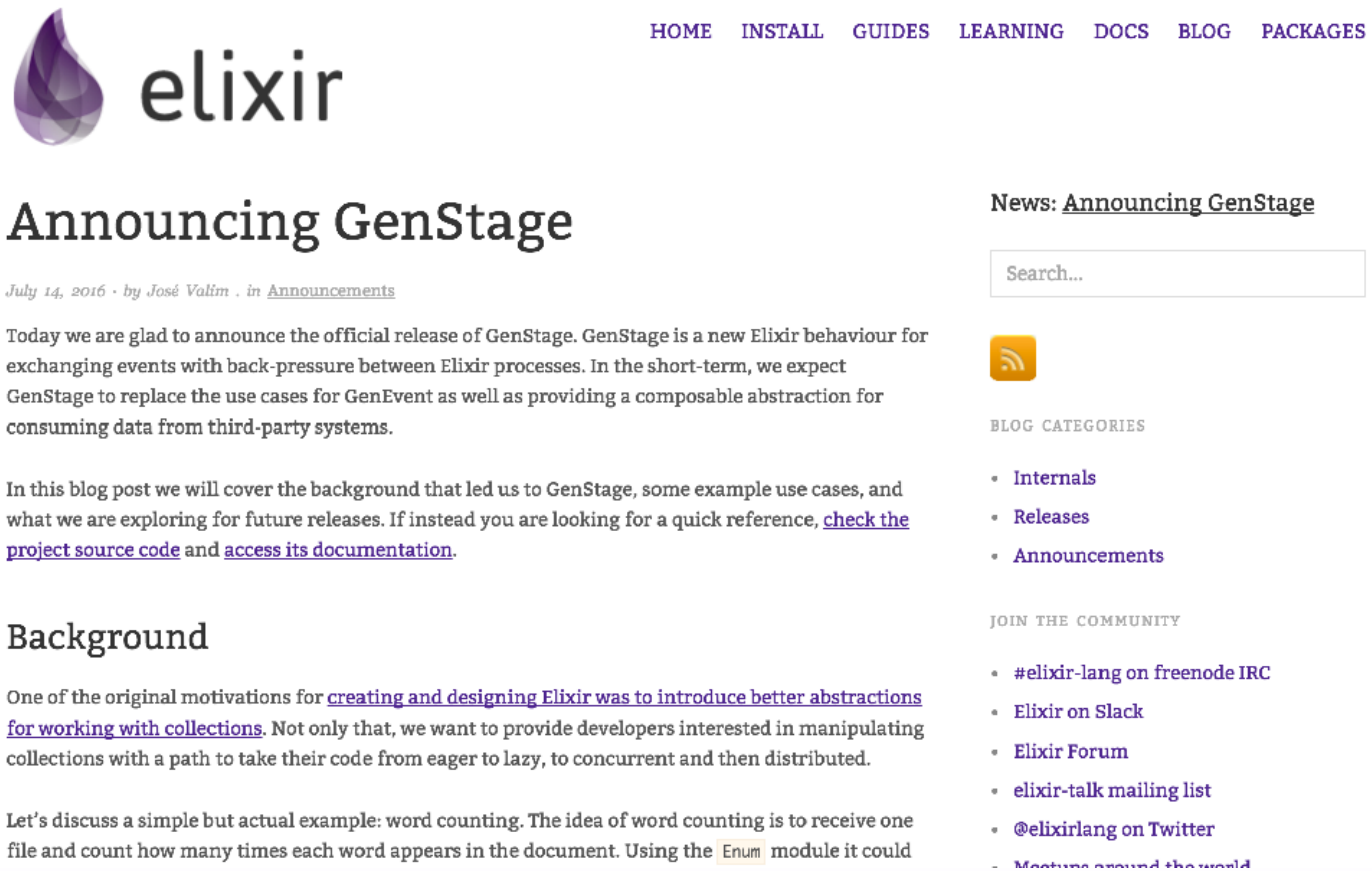
- max_demand: the maximum amount of events to ask (default 1000)
- min_demand: when reached, ask for more events (default 500)

## max_demand: 10, min_demand: 0

1. the consumer asks for 10 items
2. the consumer receives 10 items
3. the consumer processes 10 items
4. the consumer asks for 10 more
5. the consumer waits

## max_demand: 10, min_demand: 5

1. the consumer asks for 10 items
2. the consumer receives 10 items
3. the consumer processes 5 of 10 items
4. the consumer asks for 5 more
5. the consumer processes the remaining 5



Announcing GenStage

*July 14, 2016 · by José Valim . in* Announcements

Today we are glad to announce the official release of GenStage. GenStage is a new Elixir behaviour for exchanging events with back-pressure between Elixir processes. In the short-term, we expect GenStage to replace the use cases for GenEvent as well as providing a composable abstraction for consuming data from third-party systems.

In this blog post we will cover the background that led us to GenStage, some example use cases, and what we are exploring for future releases. If instead you are looking for a quick reference, check the project source code and access its documentation.

Background

One of the original motivations for creating and designing Elixir was to introduce better abstractions for working with collections. Not only that, we want to provide developers interested in manipulating collections with a path to take their code from eager to lazy, to concurrent and then distributed.

Let's discuss a simple but actual example: word counting. The idea of word counting is to receive one file and count how many times each word appears in the document. Using the `Enum` module it could

News: Announcing GenStage

BLOG CATEGORIES

- Internals
- Releases
- Announcements

JOIN THE COMMUNITY

- #elixir-lang on freenode IRC
- Elixir on Slack
- Elixir Forum
- elixir-talk mailing list
- @elixirlang on Twitter

http://bit.ly/genstage

## Topics

- 1200LOC: How is flow implemented?
    - Its core is a 80LOC stage
- 1300LOD: How to reason about flows?

# Flow

```
“roses are red\n
violets are blue\n"
```

↓

```
%{“are” => 2,
  “blue” => 1,
  “red” => 1,
  “roses" => 1,
  “violets" => 1}
```

```
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
|> Flow.partition()
|> Flow.reduce(fn -> %{} end, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```

↓

```
#Flow<...>
```

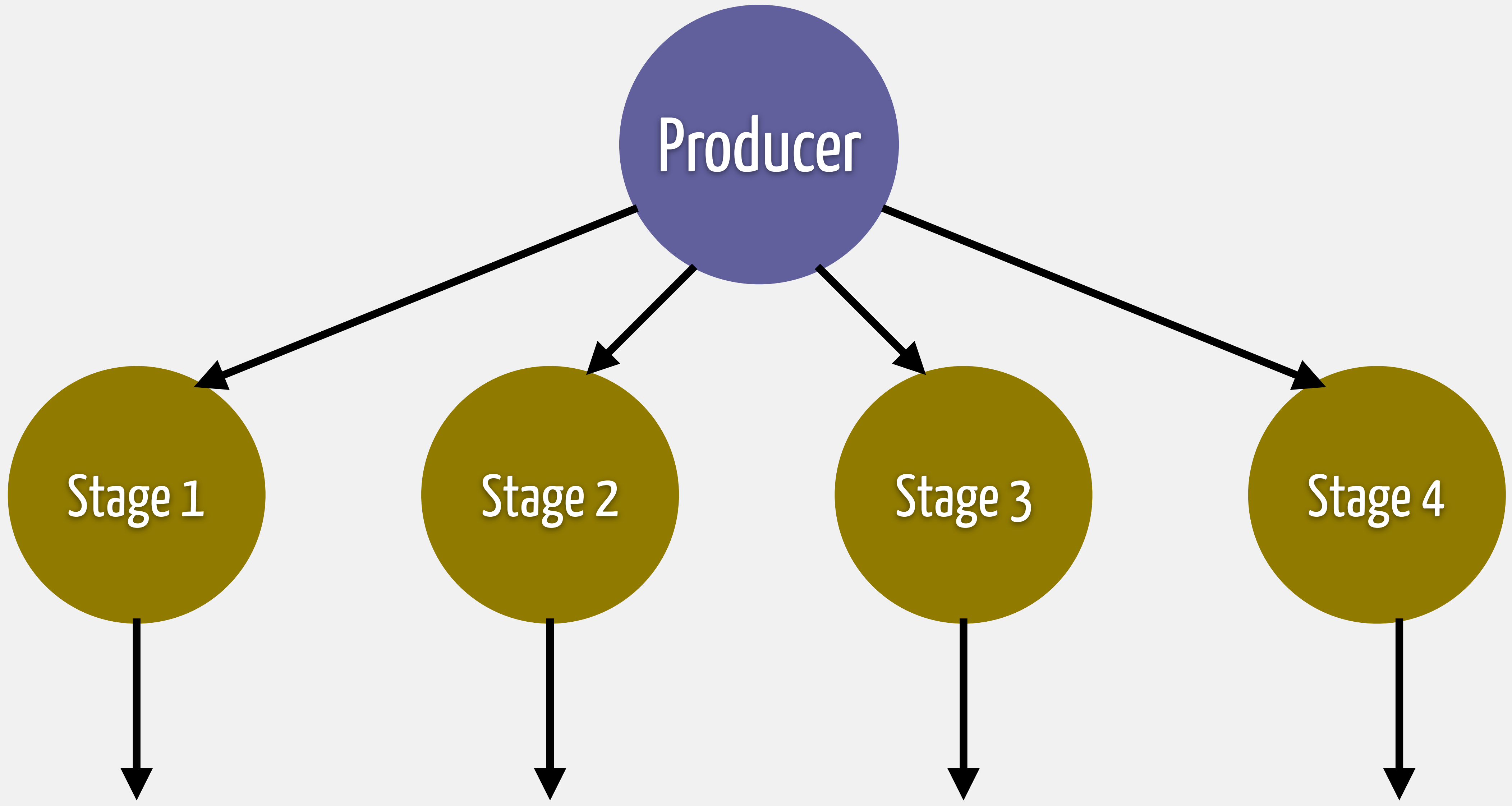
```
File.stream!("source", :line)
 |> Flow.from_enumerable()
 |> Flow.flat_map(&String.split/1)
 |> Flow.partition()
 |> Flow.reduce(fn -> %{} end, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
 |> Enum.into(%{})
```

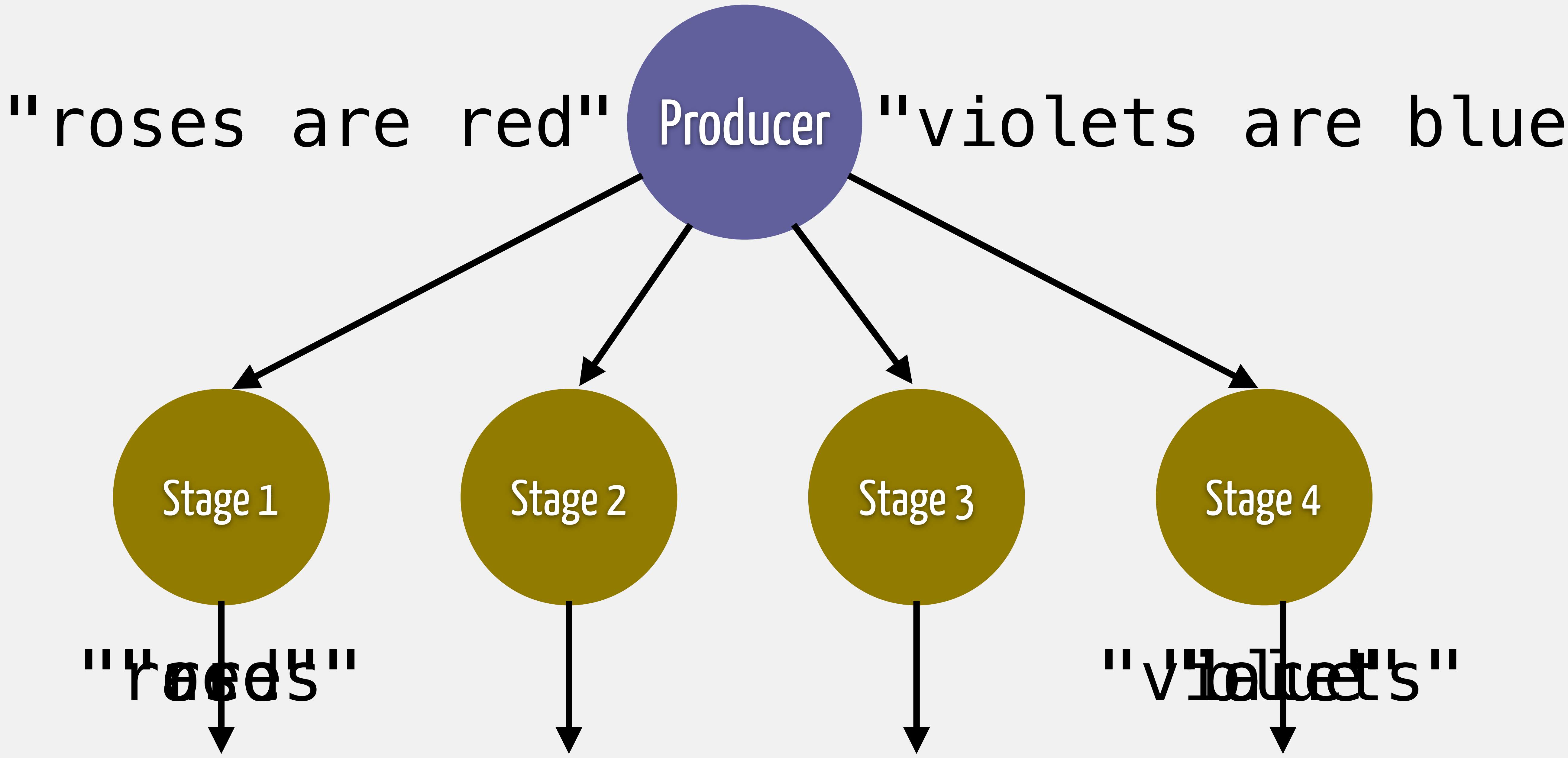
↓

```
%{“are” => 2,
  “blue” => 1,
  “red” => 1,
  “roses" => 1,
  “violets" => 1}
```

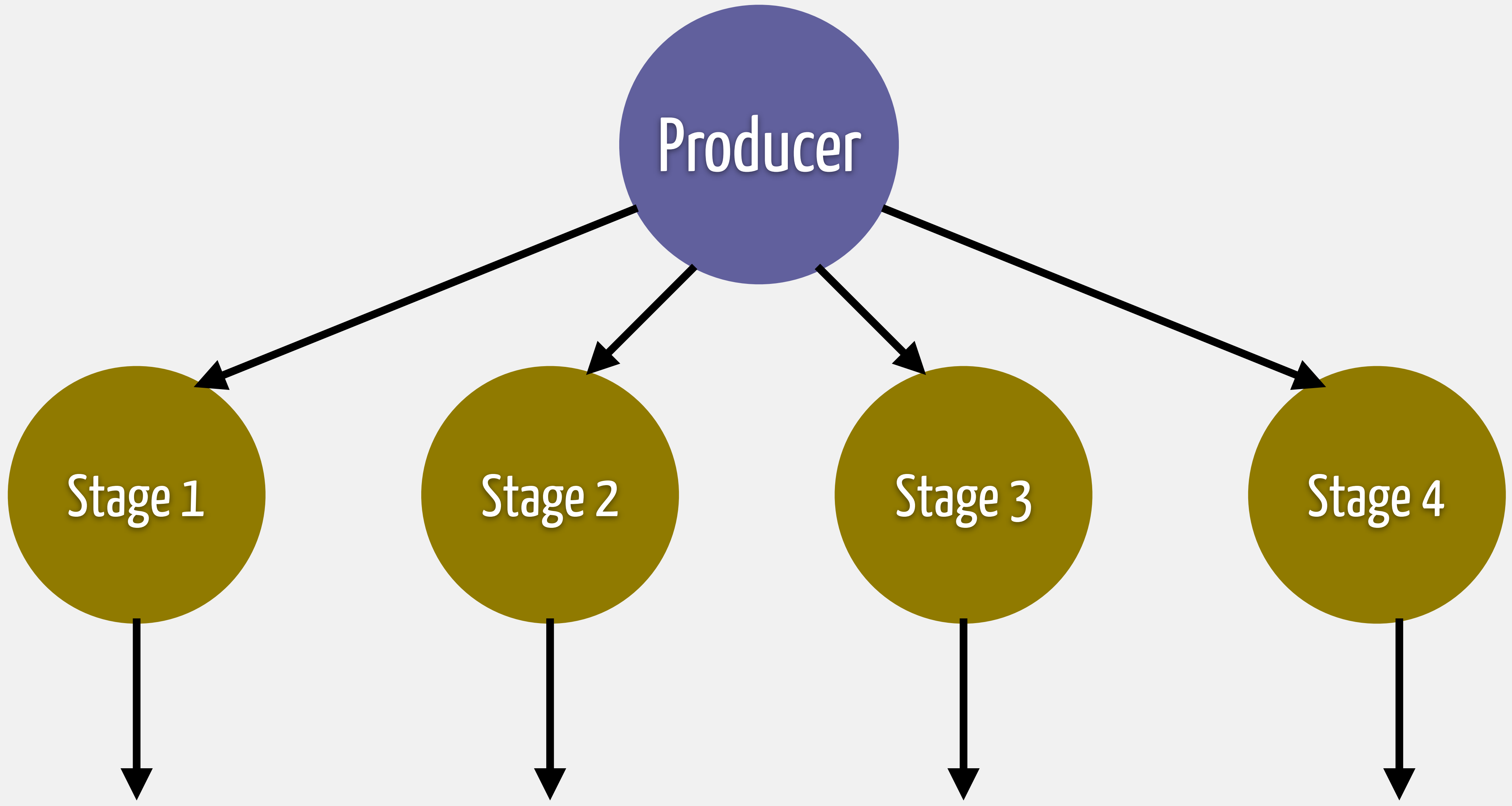
```elixir
File.stream!("source", :line)
|> Flow.from_enumerable()
```

```
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
```

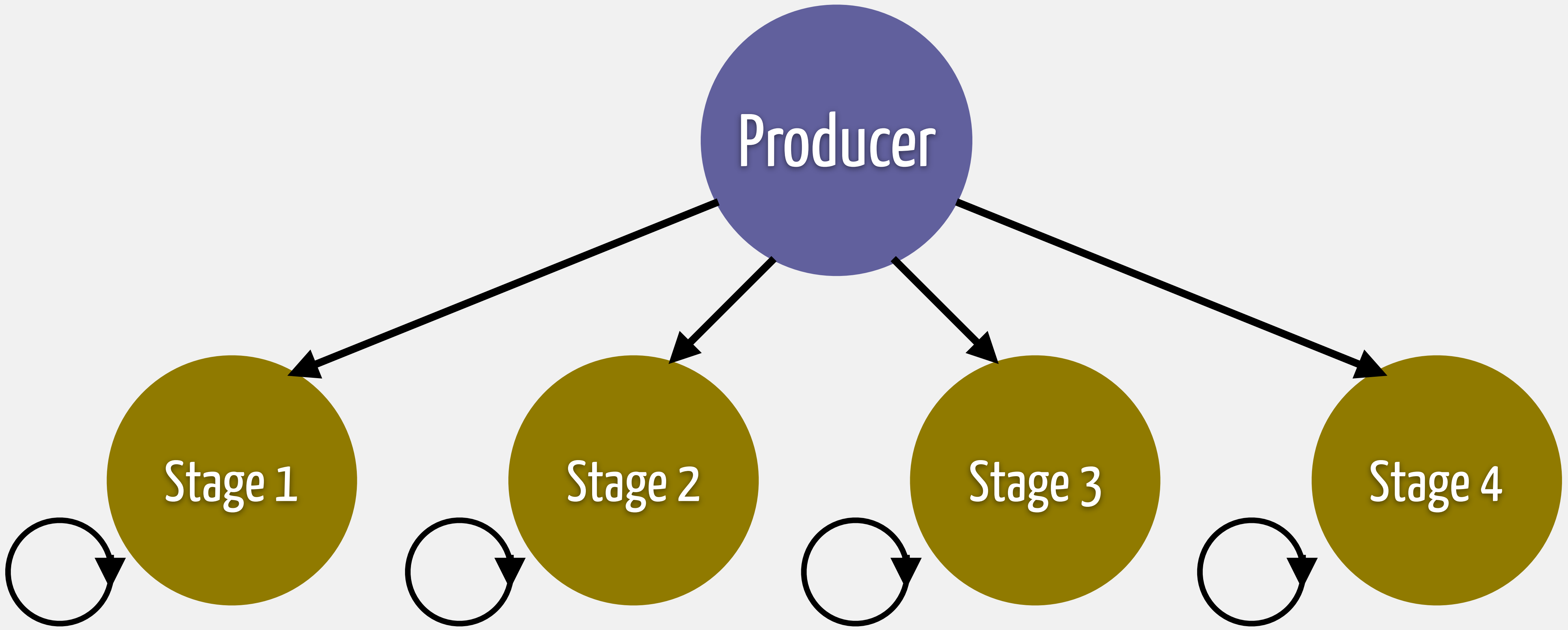


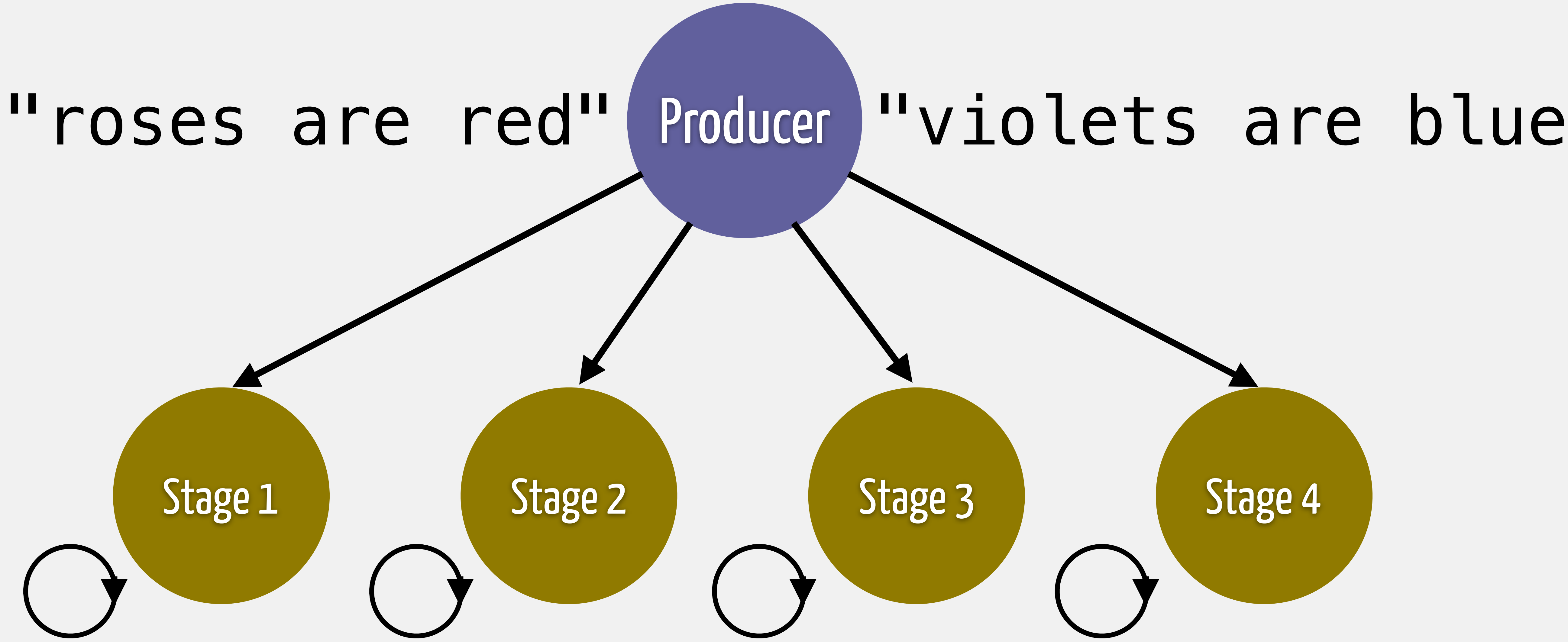

"roses are red"
Producer
"violets are blue
Stage 1
Stage 2
Stage 3
Stage 4
"roses"
"violets"

```elixir
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
```



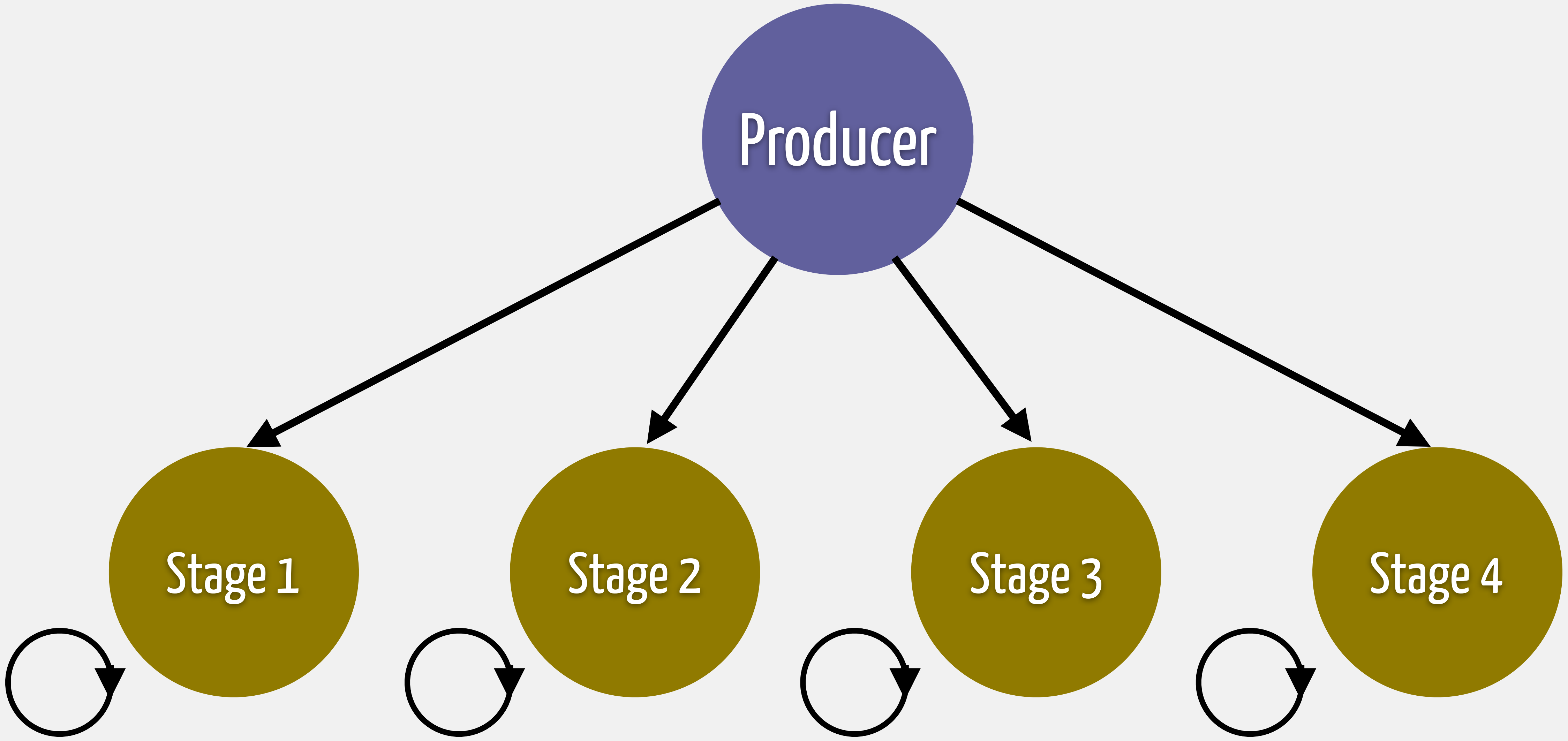
```
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
|> Flow.reduce(fn -> %{} end, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```





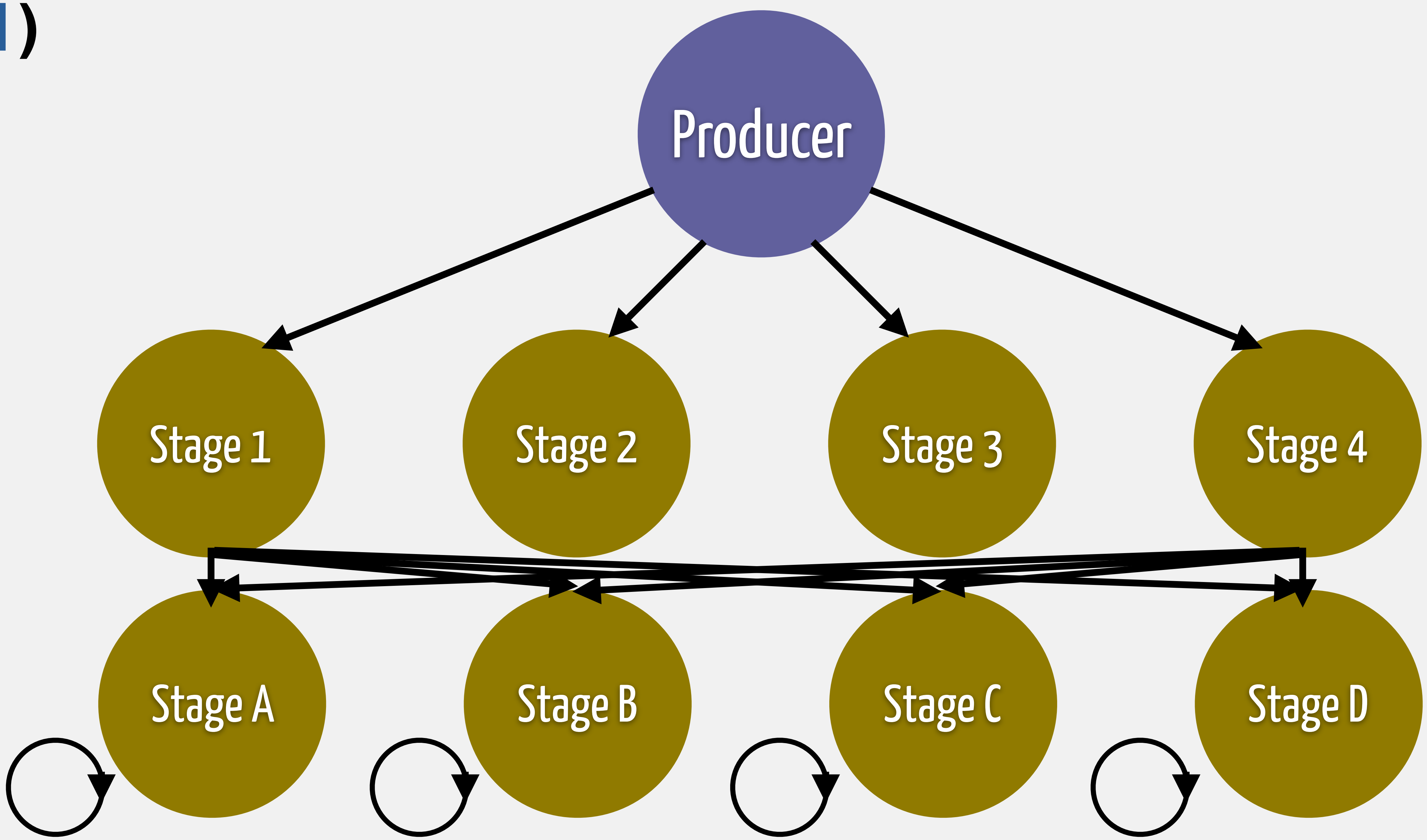
```
%{“are” => 1,
  “red” => 1,
  “roses" => 1}
```

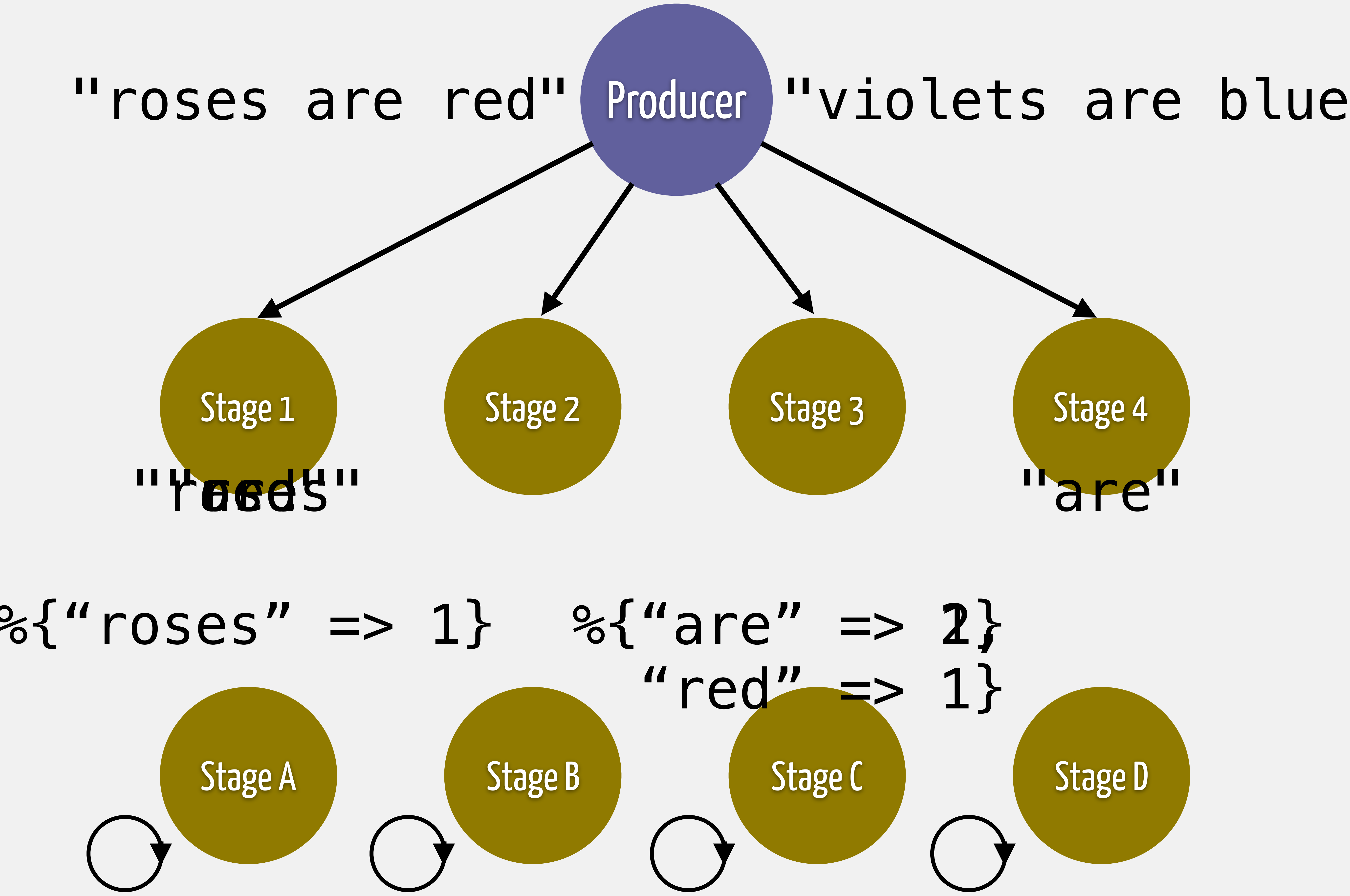
```
%{“are” => 1,
  “blue” => 1,
  “violets" => 1}
```

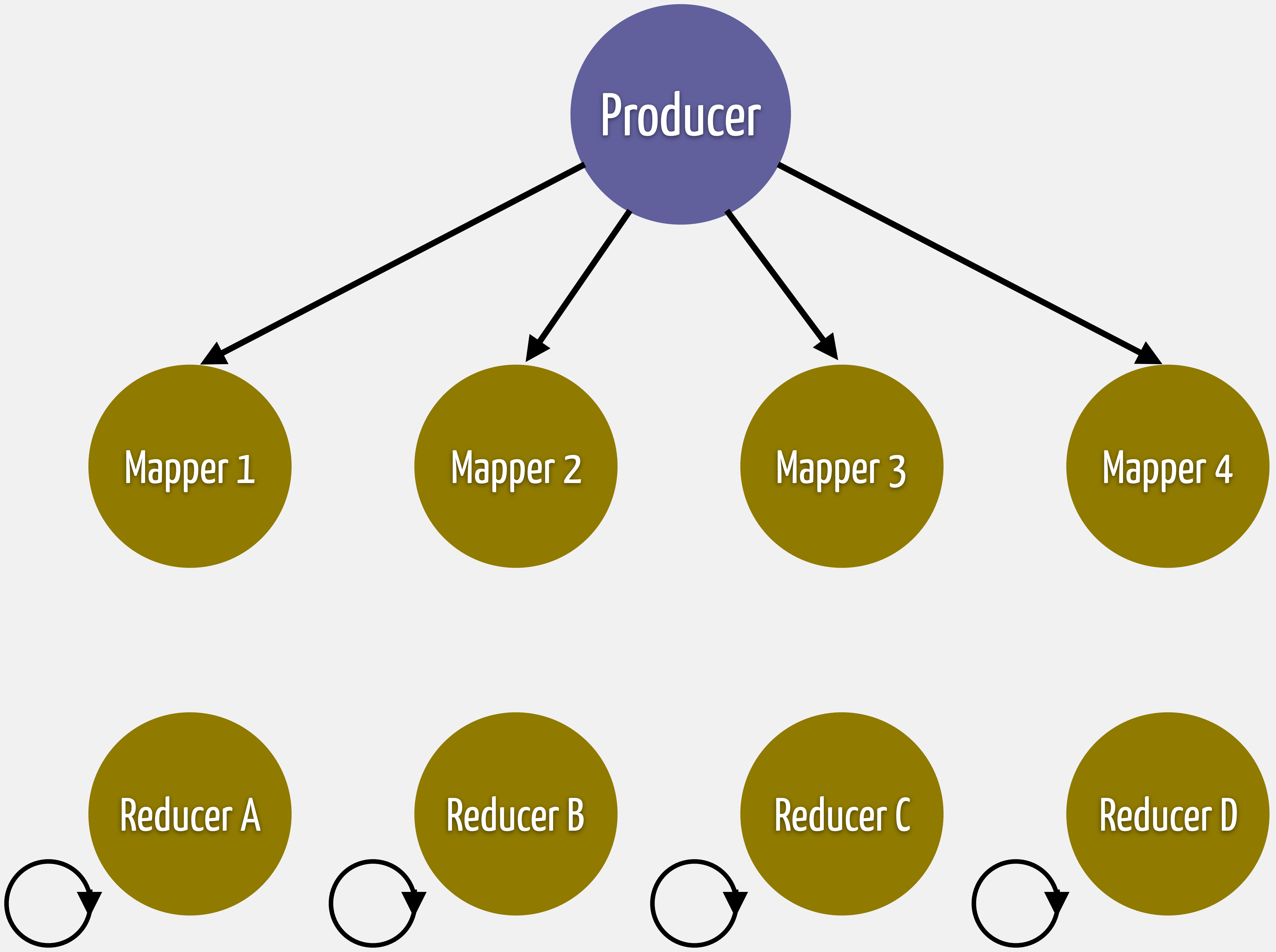


```
%{“are” => 1,                %{“are” => 1,
  “red” => 1,                  “blue” => 1,
  “roses" => 1}                “violets" => 1}
```

```
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
|> Flow.partition()
|> Flow.reduce(fn -> %{} end, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```




"roses are red"
Producer
"violets are blue
Stage 1
Stage 2
Stage 3
Stage 4
"roses"
"red"
"are"
%{“roses” => 1}
%{“are” => 1}
%{“are” => 2,
“red” => 1}
Stage A
Stage B
Stage C
Stage D


Producer
Mapper 1
Mapper 2
Mapper 3
Mapper 4
Reducer A
Reducer B
Reducer C
Reducer D

```elixir
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
|> Flow.partition()
|> Flow.reduce(fn -> %{} end, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
|> Enum.into(%{})
```

- reduce/3 collects all data into maps
- when it is done, the maps are streamed
- into/2 collects the state into a map

## Postlude: From eager, to lazy, to concurrent, to distributed

# Enum (eager)

```
File.read!("source")
|> String.split("\n")
|> Enum.flat_map(&String.split/1)
|> Enum.reduce(%{}, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```

# Stream (lazy)

```
File.stream!("source", :line)
|> Stream.flat_map(&String.split/1)
|> Enum.reduce(%{}, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
```

# Flow (concurrent)

```
File.stream!("source", :line)
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split/1)
|> Flow.partition()
|> Flow.reduce(%{}, fn word, map ->
  Map.update(map, word, 1, & &1 + 1)
end)
|> Enum.into(%{})
```

## Flow features

- Provides map and reduce operations, partitions, flow merge, flow join
- Configurable batch size (max & min demand)
- Data windowing with triggers and watermarks

## Distributed?

- Flow API has feature parity with frameworks like Apache Spark
- However, there is no distribution nor execution guarantees

“small inputs are common in practice: 40–80% of Cloudera customers’ MapReduce jobs and 70% of jobs in a Facebook trace have ≤ 1GB of input”

CHEN, Y., ALSPAUGH, S., AND KATZ, R.
Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads

“For between 40-80% of the jobs submitted to MapReduce systems, you’d be better off just running them on a single machine”

GOG, I., SCHWARZKOPF, M., CROOKS, N., GROSVENOR, M. P., CLEMENT, A., AND HAND, S. Musketeer: all for one, one for all in data processing systems.

## Distributed?

- Single machine matters - try it!
- The gap between concurrent and distributed in Elixir is small
- Durability concerns will be tackled next

## Inspirations

- Akka Streams - back pressure contract
- Apache Spark - map reduce API
- Apache Beam - windowing model
- Microsoft Naiad - stage notifications

## Built and designed at

plataformatec

consulting and software engineering

elixir

@elixirlang / elixir-lang.org