Experimental.Flow

An overview of experimental Elixir Flow module that allows developers to express processing steps for collections, like Stream, but utilizing the power of parallel execution.

Resources

ElixirConf 2016 - Keynote by José Valim

www.youtube.com/watch?v=srtMWzyqdp8

Kyiv Elixir Meetup 3 - Flow-Based Programming with Elixir, Anton Mishchuk

www.youtube.com/watch?v=yDaPaCxAVq8

www.slideshare.net/AntonMishchuk/flowbased-programming-with-elixir

Resources

Announcing GenStage

elixir-lang.org/blog/2016/07/14/announcing-genstage/

gen_stage

hex.pm/packages/gen_stage

https://hexdocs.pm/gen_stage/

https://hexdocs.pm/gen_stage/Experimental.Flow.html

Task

Implement word counting algorithm using Eager, Lazy and Concurrent approaches

- Eager – Enum
- Lazy – Stream
- Concurrent - **Flow**

Eager

+ High processing speed (for small collections)
- May require large amounts of memory

Good for fast processing of small collections

Eager (Enum)

```
def process_eager(path_to_file) do
     path_to_file
     |> File.read!()
     |> String.split()
     |> Enum.reduce(%{}, &words_to_map/2)
end
```

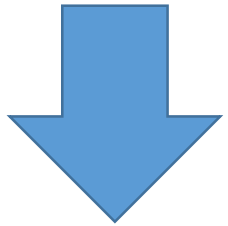
Helper functions

```
defp words_to_map(word, map) do
      word
      |> String.replace(~r/\W/u, "")
      |> filter_map(map)
end

defp filter_map("", map), do: map
defp filter_map(word, map) do
      word = String.downcase(word)
      Map.update(map, word, 1, &(&1 + 1))
end
```

Eager (Enum)

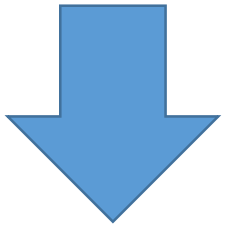
```
def process_eager(path_to_file) do
      path_to_file
      |> File.read!()
```



"The Project Gutenberg EBook of The Complete Works of William Shakespeare, by\r\nWilliam Shakespeare\r\n\r\n …”

Eager (Enum)

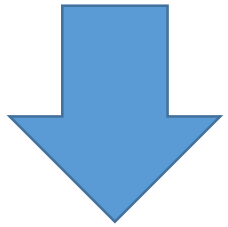
```
def process_eager(path_to_file) do
      path_to_file
      |> File.read!()
      |> String.split()
```



["The", "Project", "Gutenberg", "EBook", "of", "The", "Complete", "Works", "of", "William", "Shakespeare,", "by", "William", "Shakespeare", …]

Eager (Enum)

```
def process_eager(path_to_file) do
      path_to_file
      |> File.read!()
      |> String.split()
      |> Enum.reduce(%{}, &words_to_map/2)
```



%{"citizens" => 82, "bestrides" => 1, "oercoverd" => 1, "roots" => 10, "quintus" => 20, "ordain" => 1, "sop" => 3, "inset" => 1, …}

Lazy (Stream)

+ Allows us to “control” memory consumption
- Processing overhead

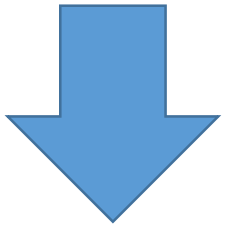
Allows us to work with large datasets without loading them all into memory

Lazy (Stream)

```elixir
def process_lazy(path_to_file) do
    path_to_file
    |> File.stream!()
    |> Stream.flat_map(&String.split/1)
    |> Enum.reduce(%{}, &words_to_map/2)
end
```

Lazy (Stream)

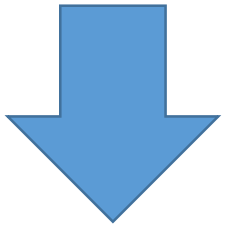
```
def process_lazy(path_to_file) do
    path_to_file
    |> File.stream!()
```



%File.Stream{line_or_bytes: **:line**, modes: [:raw, :read_ahead, {:read_ahead, 100000}, :binary], path: “…/small.txt", raw: true}

Lazy (Stream)

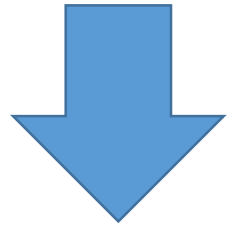
```
def process_lazy(path_to_file) do
      path_to_file
      |> File.stream!()
      |> Stream.flat_map(&String.split/1)
```



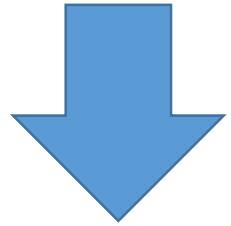
#Function<57.64528250/2 in Stream.transform/3>

Lazy (Stream)

```
"The Project Gutenberg EBook of The Complete Works
of William Shakespeare, by\n"
```



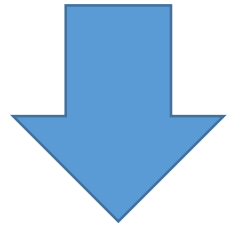
```
"The", "Project", "Gutenberg", "EBook", "of", "The",
"Complete", "Works", "of", "William", "Shakespeare",
"by"
```



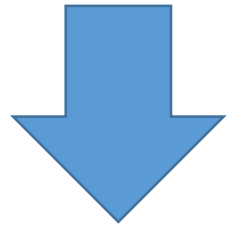
```
|> Enum.reduce(%{}, &words_to_map/2)
```

Lazy (Stream)

```
"William Shakespeare\n"
```



```
"William", "Shakespeare“
```



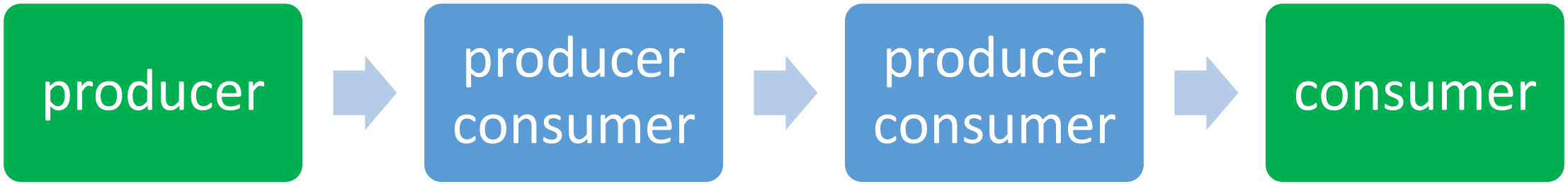
```
    |> Enum.reduce(%{}, &words_to_map/2)
```

Concurrent (Flow)

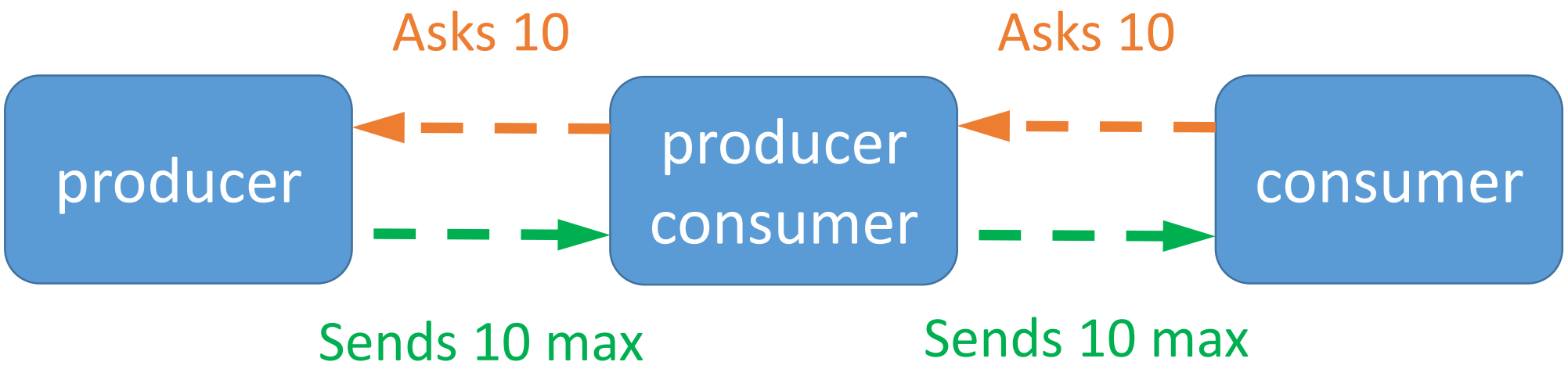
+ Concurrency
+ Allows us to “control” memory consumption
- Processing overhead
- Processing order

Allows us to process large or infinite collections **concurrently** (on multicore machines)

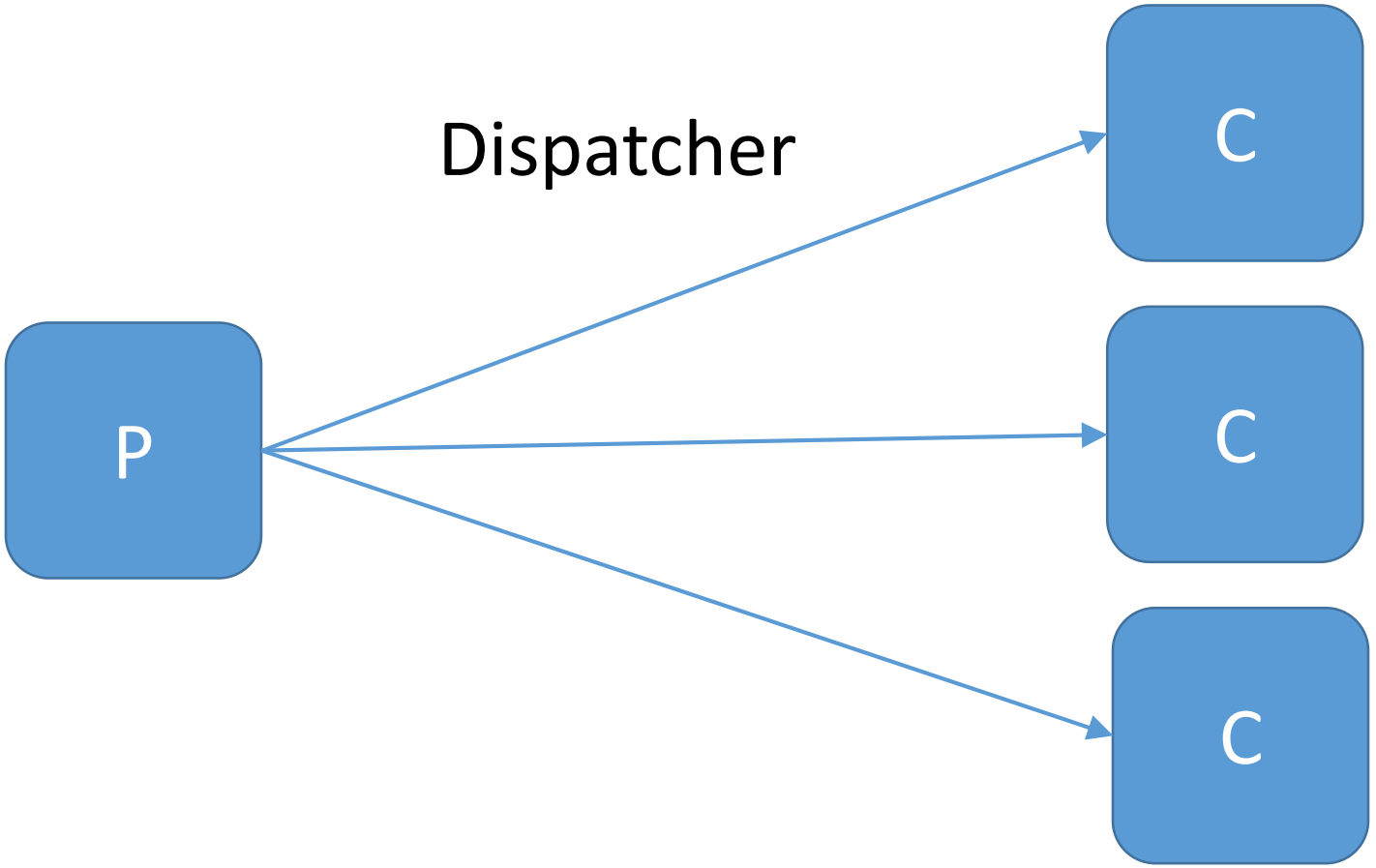
GenStage is a new Elixir behaviour for exchanging events with back-pressure between Elixir processes



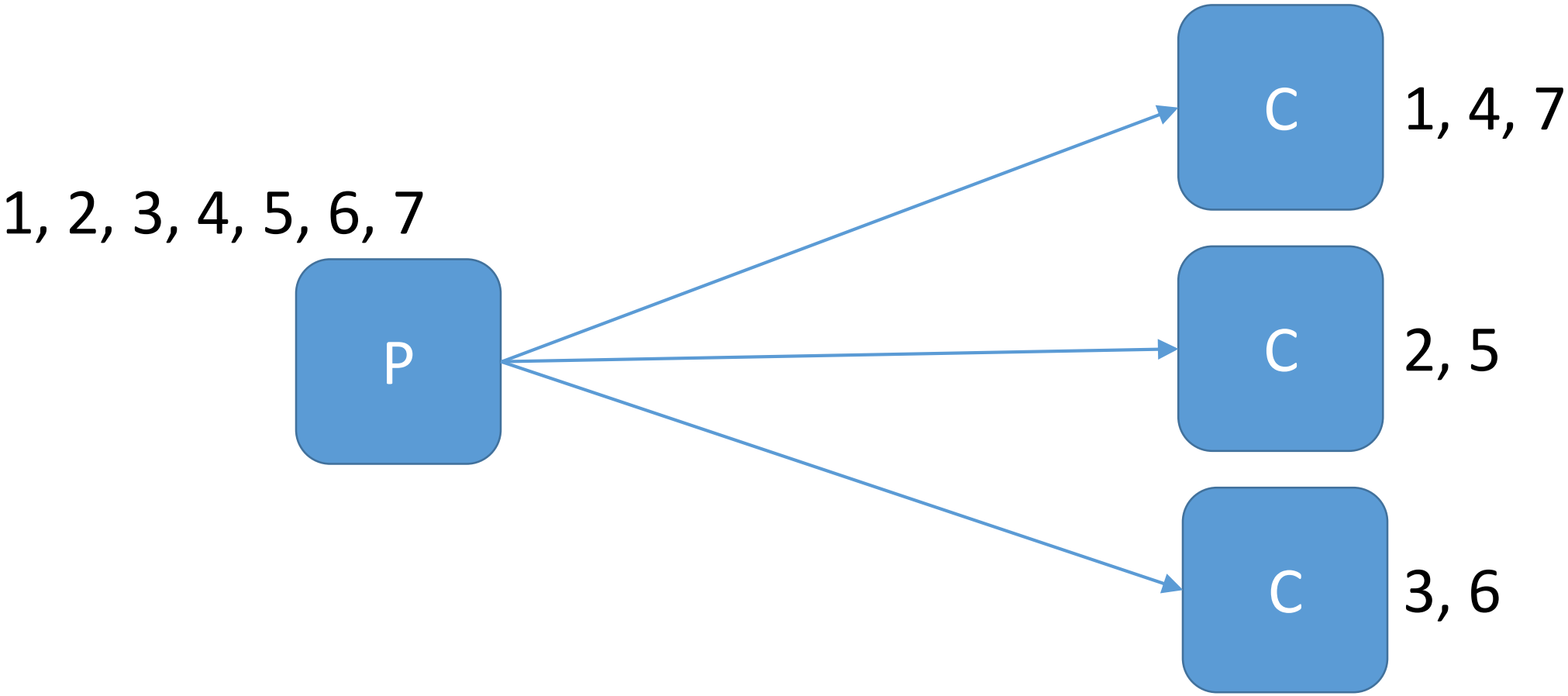
GenStage: demand-driven message exchange



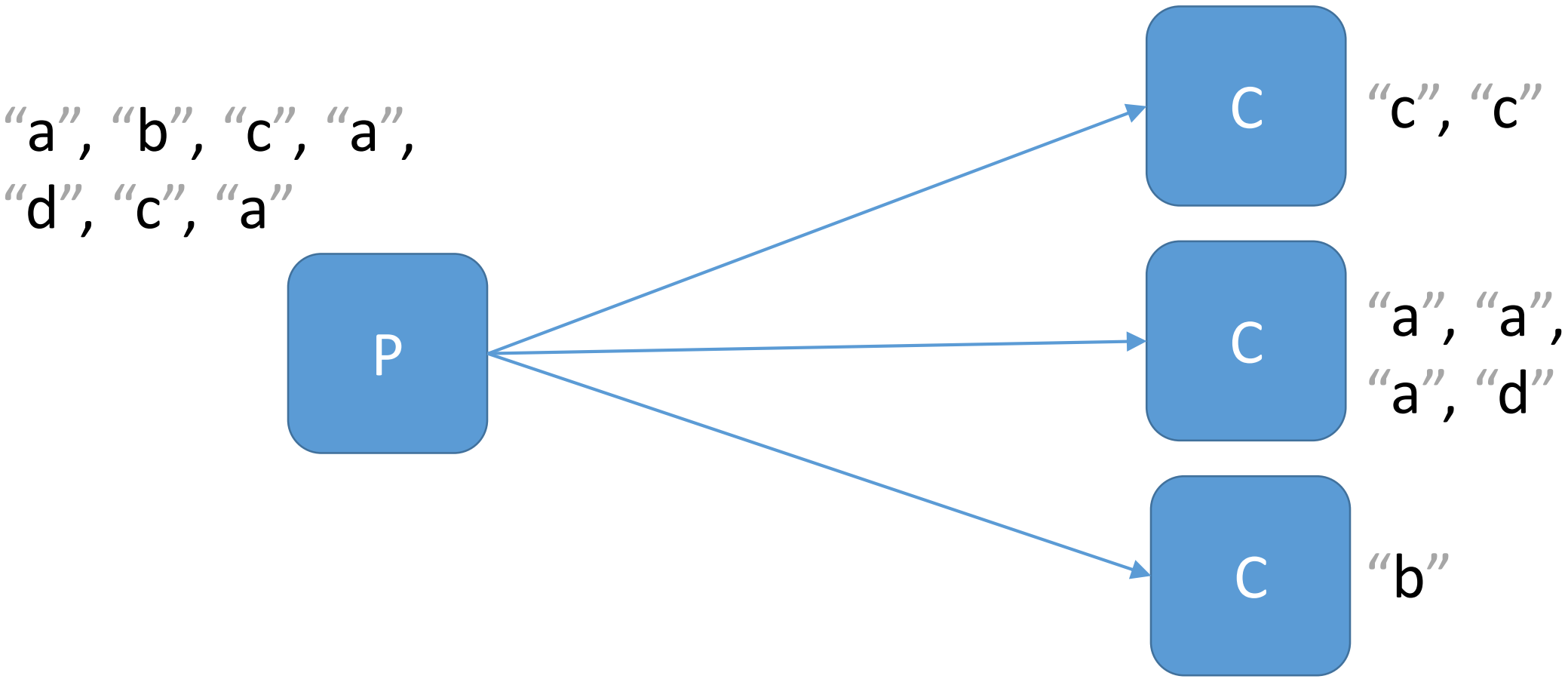
Dispatcher defines how the events are dispatched to multiple consumers



DemandDispatcher - dispatches events according to a demand



PartitionDispatcher - dispatches events according to a hash

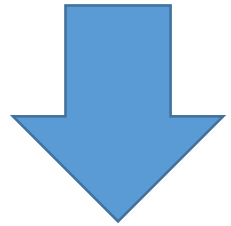


Concurrent (Flow)

```elixir
def process_flow(path_to_file) do
  path_to_file
  |> File.stream!()
  |> Flow.from_enumerable()
  |> Flow.flat_map(&String.split/1)
  |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
  |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
  |> Flow.partition()
  |> Flow.reduce(fn -> %{} end, fn word, map ->
       Map.update(map, word, 1, &(&1 + 1))
     end)
  |> Enum.into(%{})
end
```

Concurrent (Flow)

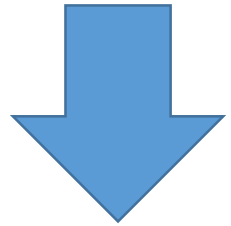
```
def process_flow(path_to_file) do
    path_to_file
    |> File.stream!()
    |> Flow.from_enumerable()
```



%Experimental.Flow{operations: [], options: [stages: 4], producers: {:enumerables, [%File.Stream{line_or_bytes: :line, modes: [:raw, :read_ahead, {:read_ahead, 100000}, :binary], path: "…/small.txt", raw: true}]}, window: %Experimental.Flow.Window.Global{periodically: [], trigger: nil}}

Concurrent (Flow)

```
def process_flow(path_to_file) do
    path_to_file
    |> File.stream!()
    |> Flow.from_enumerable()
    |> Flow.flat_map(&String.split/1)
```



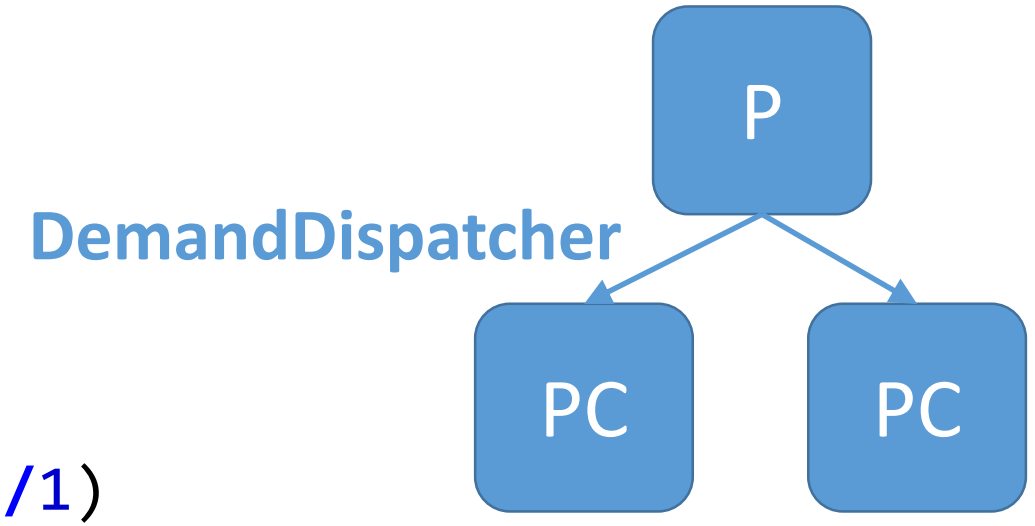
%Experimental.Flow{operations: [**{:mapper, :flat_map, [&String.split/1]}**], options: [stages: 4], producers: {:enumerables, [%File.Stream{line_or_bytes: :line, modes: [:raw, :read_ahead, {:read_ahead, 100000}, :binary], path: "d:/Elixir/flow/files/small.txt", raw: true}]}, window: %Experimental.Flow.Window.Global{periodically: [], trigger: nil}}

Concurrent (Flow)


```
def process_flow(path_to_file) do
      path_to_file
      |> File.stream!()
      |> Flow.from_enumerable()
      |> Flow.flat_map(&String.split/1)
      |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
      |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
      |> Flow.partition()
      |> Flow.reduce(fn -> %{} end, fn word, map ->
            Map.update(map, word, 1, &(&1 + 1))
         end)
      |> Enum.into(%{})
end
```

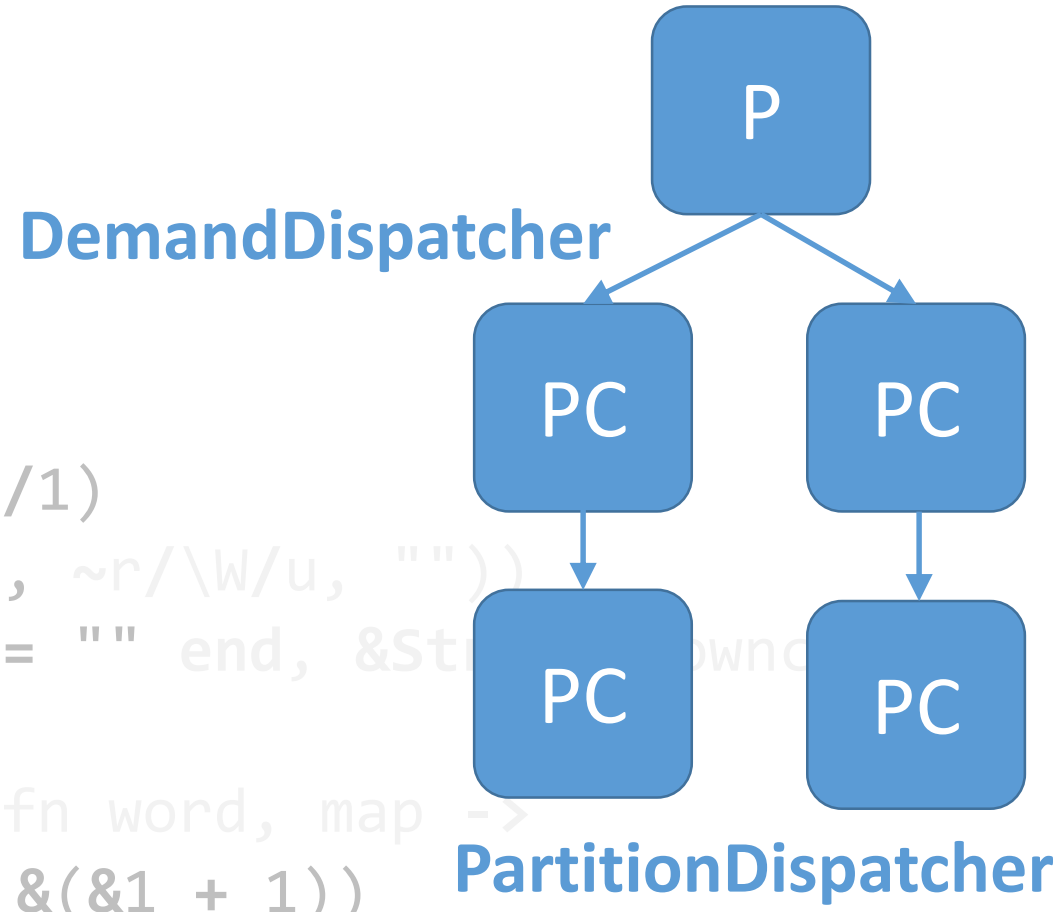
Concurrent (Flow)

```
def process_flow(path_to_file) do
  path_to_file
  |> File.stream!()
  |> Flow.from_enumerable()
  |> Flow.flat_map(&String.split/1)
  |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
  |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
  |> Flow.partition()
  |> Flow.reduce(fn -> %{} end, fn word, map ->
       Map.update(map, word, 1, &(&1 + 1))
     end)
  |> Enum.into(%{})
end
```



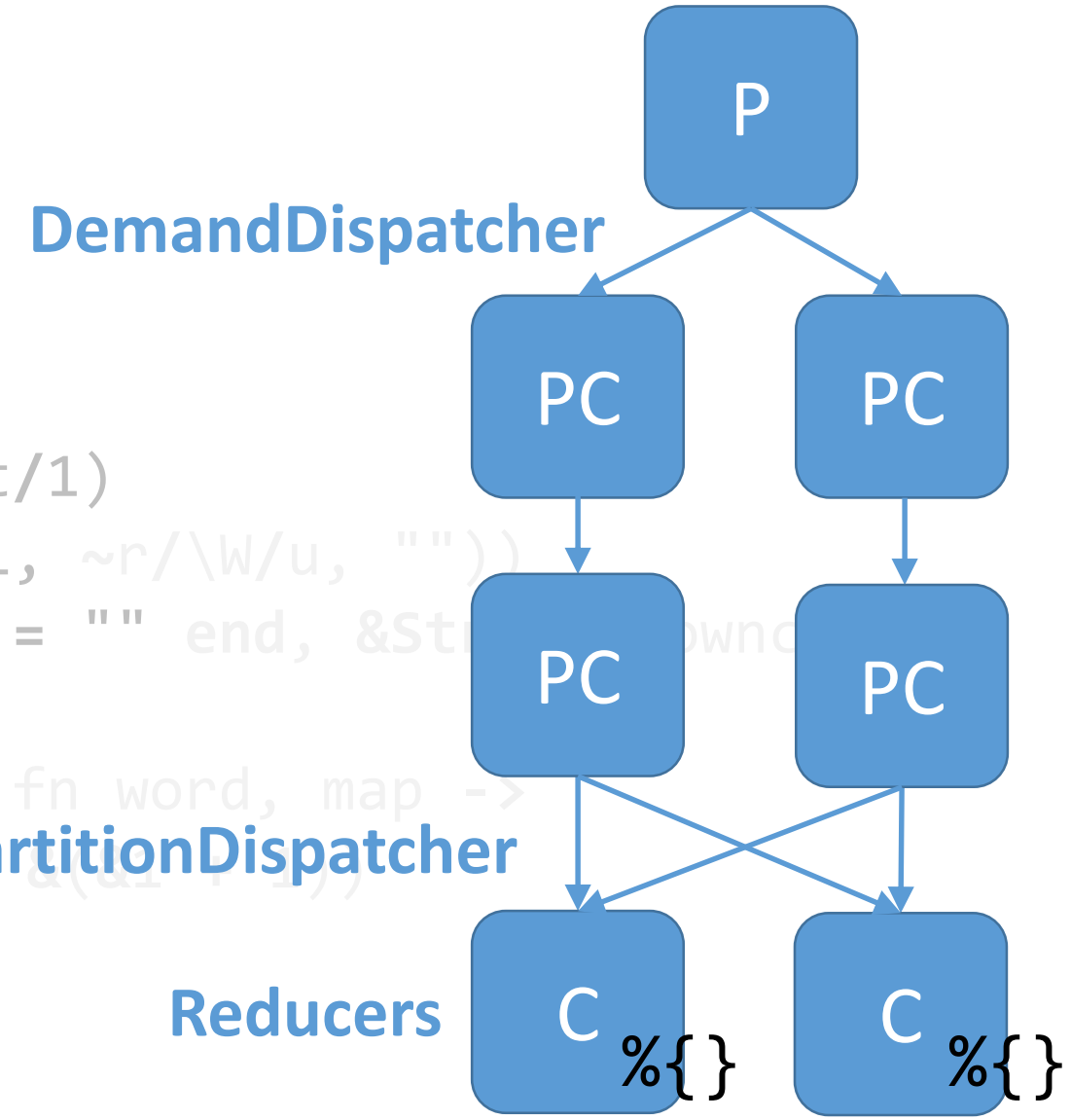
Concurrent (Flow)

```
def process_flow(path_to_file) do
      path_to_file
      |> File.stream!()
      |> Flow.from_enumerable()
      |> Flow.flat_map(&String.split/1)
      |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
      |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
      |> Flow.partition()
      |> Flow.reduce(fn -> %{} end, fn word, map ->
           Map.update(map, word, 1, &(&1 + 1))
         end)
      |> Enum.into(%{})
end
```



Concurrent (Flow)

```
def process_flow(path_to_file) do
  path_to_file
  |> File.stream!()
  |> Flow.from_enumerable()
  |> Flow.flat_map(&String.split/1)
  |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
  |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
  |> Flow.partition()
  |> Flow.reduce(fn -> %{} end, fn word, map ->
       Map.update(map, word, 1, &(&1 + 1))
     end)
  |> Enum.into(%{})
end
```

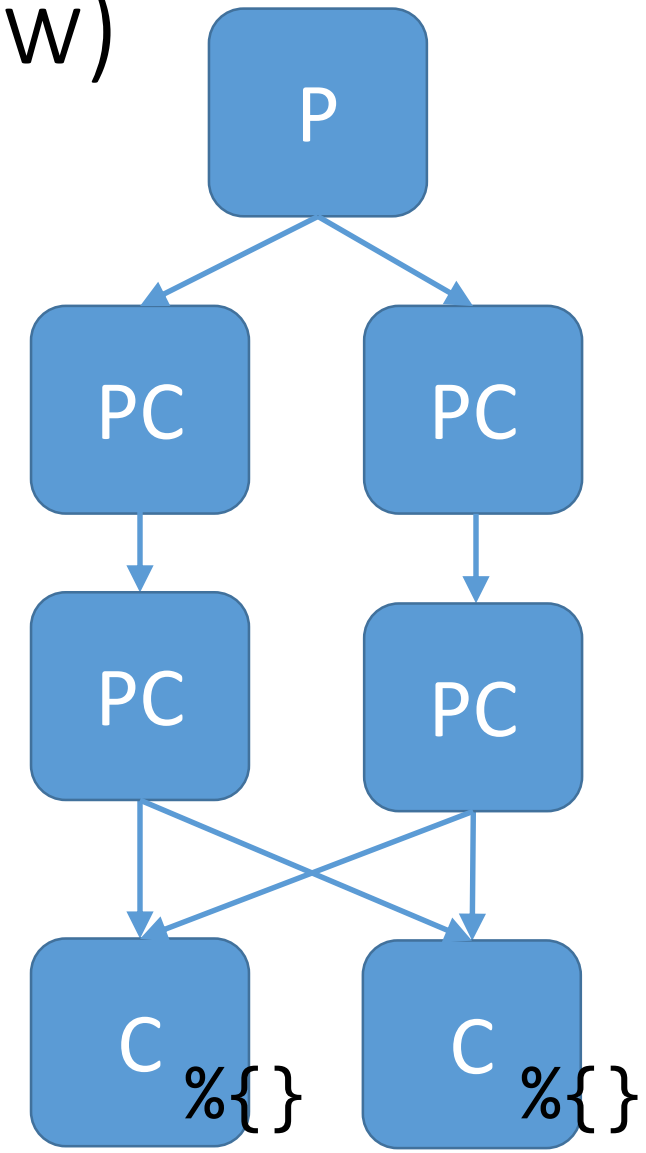


Concurrent (Flow)

"The Project Gutenberg EBook of The Complete Works of William Shakespeare, by\n"

"The", "Project", "Gutenberg", "EBook", "of", "The", "Complete", "Works", "of", "William", "Shakespeare,", "by"

"the", "project", "of", “the", "william", "of", "by ", "william"



"William Shakespeare\n"

"William", "Shakespeare"

"gutenberg", "ebook", "complete", "shakespeare", "works", "shakespeare"

Concurrent (Flow)

```elixir
def process_flow(path_to_file) do
  path_to_file
  |> File.stream!()
  |> Flow.from_enumerable()
  |> Flow.flat_map(&String.split/1)
  |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
  |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
  |> Flow.partition()
  |> Flow.reduce(fn -> %{} end, fn word, map ->
       Map.update(map, word, 1, &(&1 + 1))
     end)
  |> Enum.into(%{})
end
```

Concurrent (Flow): multiple sources

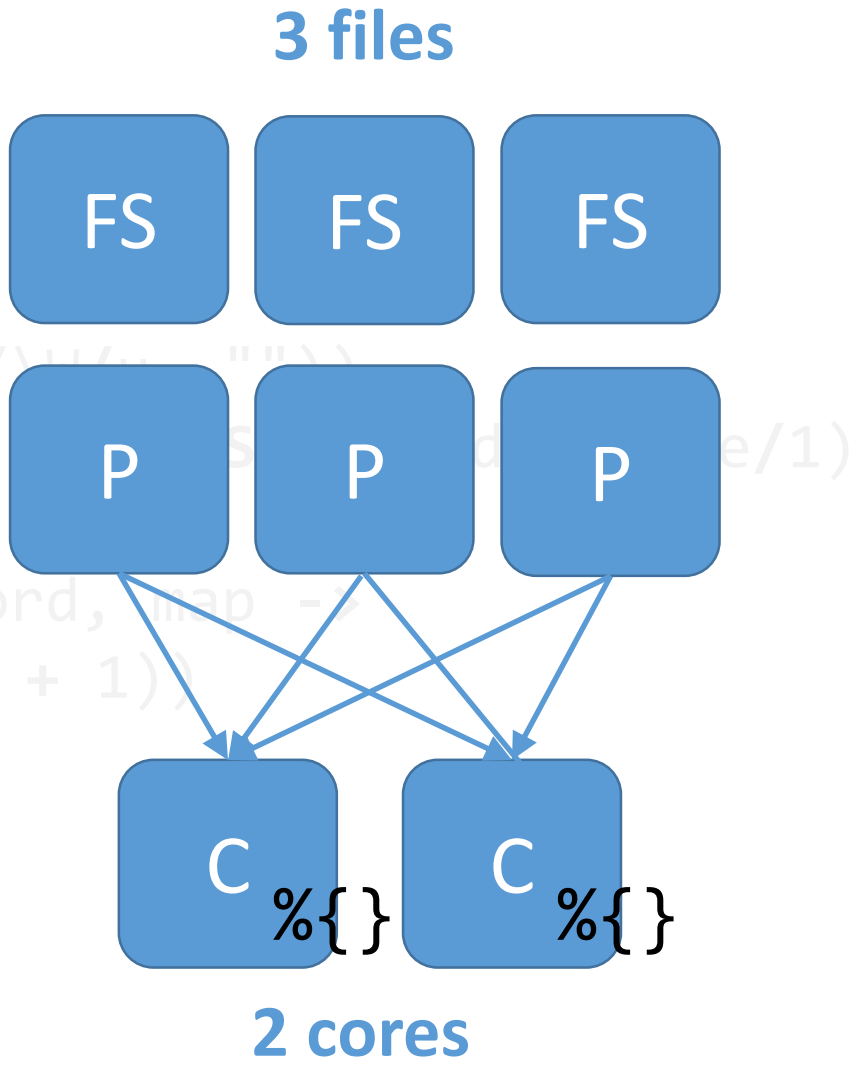
```
streams =
for file <- File.ls!(path_to_dir) do
     File.stream!(path_to_dir <> "/" <> file,
          read_ahead: 100_000)
end
```

Concurrent (Flow): multiple sources

```
streams
      |> Flow.from_enumerables()
      |> Flow.flat_map(&String.split/1)
      |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
      |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
      |> Flow.partition()
      |> Flow.reduce(fn -> %{} end, fn word, map ->
           Map.update(map, word, 1, &(&1 + 1))
         end)
      |> Enum.into(%{})
```

Concurrent (Flow): multiple sources

```
streams
      |> Flow.from_enumerables()
      |> Flow.flat_map(&String.split/1)
      |> Flow.map(&String.replace(&1, ~r/\W/u, ""))
      |> Flow.filter_map(fn w -> w != "" end, &String.downcase/1)
      |> Flow.partition()
      |> Flow.reduce(fn -> %{} end, fn word, map ->
           Map.update(map, word, 1, &(&1 + 1))
         end)
      |> Enum.into(%{})
```



Configuration (demand, the number of stages)

```
Flow.partition(stages: 8)
```

- :stages - the number of partitions (reducer stages)
- :hash - the hashing function
- :max_demand - the maximum demand for this subscription
- :min_demand - the minimum demand for this subscription
- …

Experimental.Flow.Window

Splits a flow into **windows** that are materialized at certain **triggers**.

```
window = Flow.Window.global
     |> Flow.Window.trigger_every(10, :keep)

window = Flow.Window.global
     |> Flow.Window.trigger_every(10, :reset)
```

Experimental.Flow.Window

```
Flow.from_enumerable(1..100)
|> Flow.partition(window: window, stages: 1)
|> Flow.reduce(fn -> 0 end, & &1 + &2)
|> Flow.emit(:state)
|> Enum.to_list()
```

```
keep> [55, 210, 465, 820, 1275, 1830, 2485, 3240, 4095, 5050,
5050]
```

```
reset> [55, 155, 255, 355, 455, 555, 655, 755, 855, 955, 0]
```

Questions?

twitter.com/bodarev_yurii

yuriibodarev@gmail.com

THANK YOU!!!

https://github.com/yuriibodarev/elixir_flow