
# PSUG #52
# Dataflow and simplified reactive programming with Akka-streams

Stéphane Manciot
26/02/2015

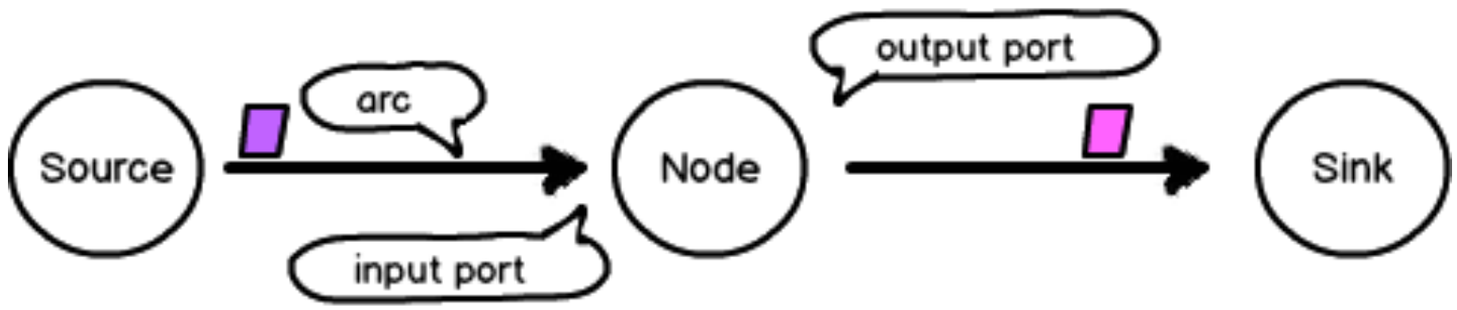
# Plan

- Dataflow main features
- Reactive Streams
- Akka-streams - Pipeline Dataflow
- Akka-streams - working with graphs

# Dataflow - main features

# Dataflow Nodes and data

- A node is a processing element that takes inputs, does some operation and returns the results on its outputs
- Data (referred to as tokens) flows between nodes through arcs
- The only way for a node to send and receive data is through ports
- A port is the connection point between an arc and a node



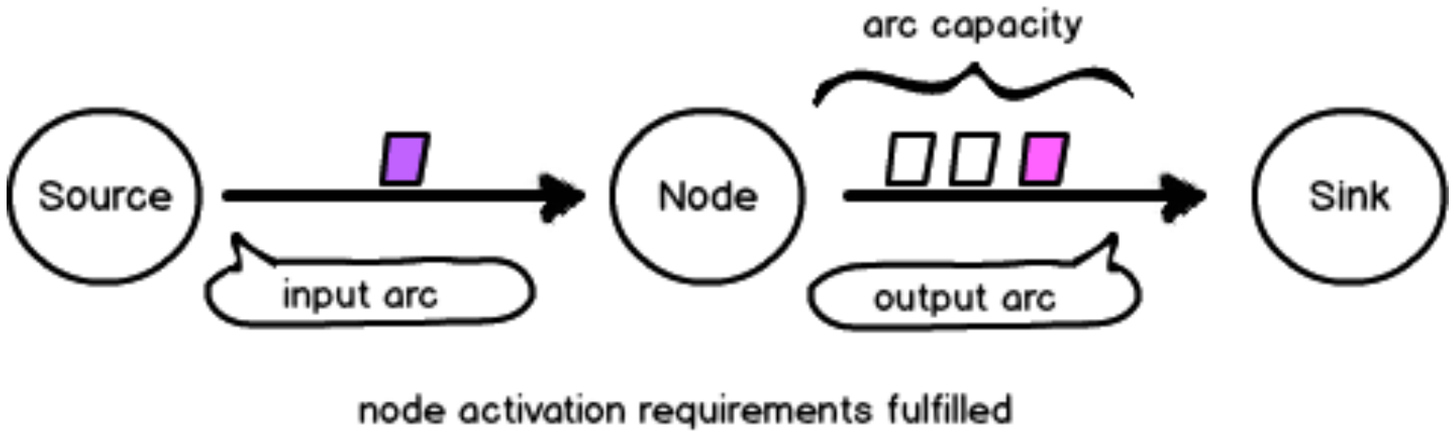

A data flow node with one input port and one output port

# Dataflow Arcs

- Always connects one output port to one input port
- Has a capacity : the maximum amount of tokens that an arc can hold at any one time
- Node activation requirements :
  - at least one token waiting on the input arc
  - space available on the output arc
- Reading data from the input arc frees space on the latter
- Arcs may contain zero or more tokens as long as it is less than the arc’s capacity

# Dataflow graph

- Explicitly states the connections between nodes
- Dataflow graph execution :
  - Node activation requirements are fulfilled
  - A token is consumed from the input arc
  - The node executes
  - If needed, a new token is pushed to the output arc

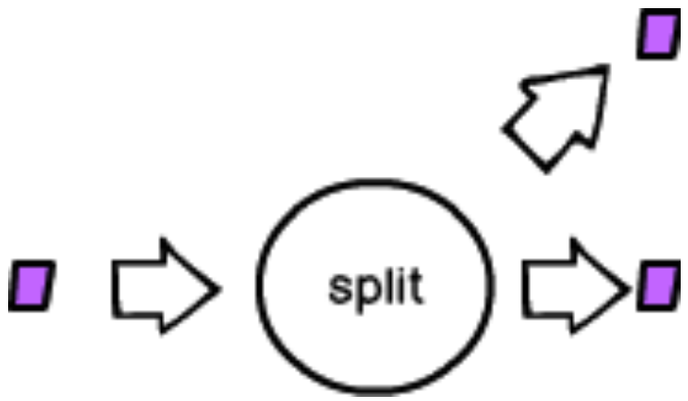


# Dataflow features - push or pull data

- Push :
  - nodes send token to other nodes whenever they are available
  - the data producer is in control and initiates transmissions
- Pull :
  - the demand flows upstream
  - allows a node to lazily produce an output only when needed
  - the data consumer is in control

# Dataflow features - mutable or immutable data

- splits require to copy mutable data




- Immutable data is preferred any time parallel computations exist

# Dataflow features - multiple inputs and/or outputs

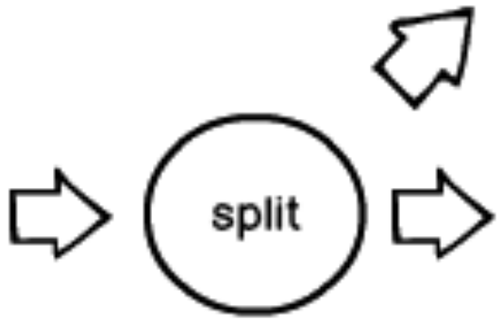
- multiple inputs
  - node “firing rule”
    - ALL of its inputs have tokens (zip)
    - ANY of its inputs have tokens waiting (merge)
  - activation preconditions
    - a firing rule for the node must match the available input tokens
    - space for new token(s) are available on the outputs
- multiple outputs

# Dataflow features - compound nodes

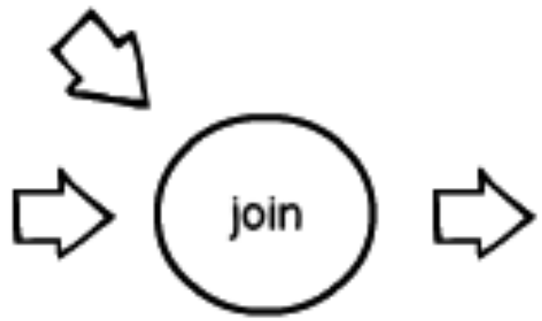
- the ability to create new nodes by using other nodes
- a smaller data flow program
- the interior nodes
  - should not know of anything outside of the compound node
  - must be able to connect to the ports of the parent compound node

# Dataflow features - arc joins and/or splits

- Fan-in : merge, zip, concat
- Fan-out : broadcast, balance, unzip




1 input port and 2 or more output ports

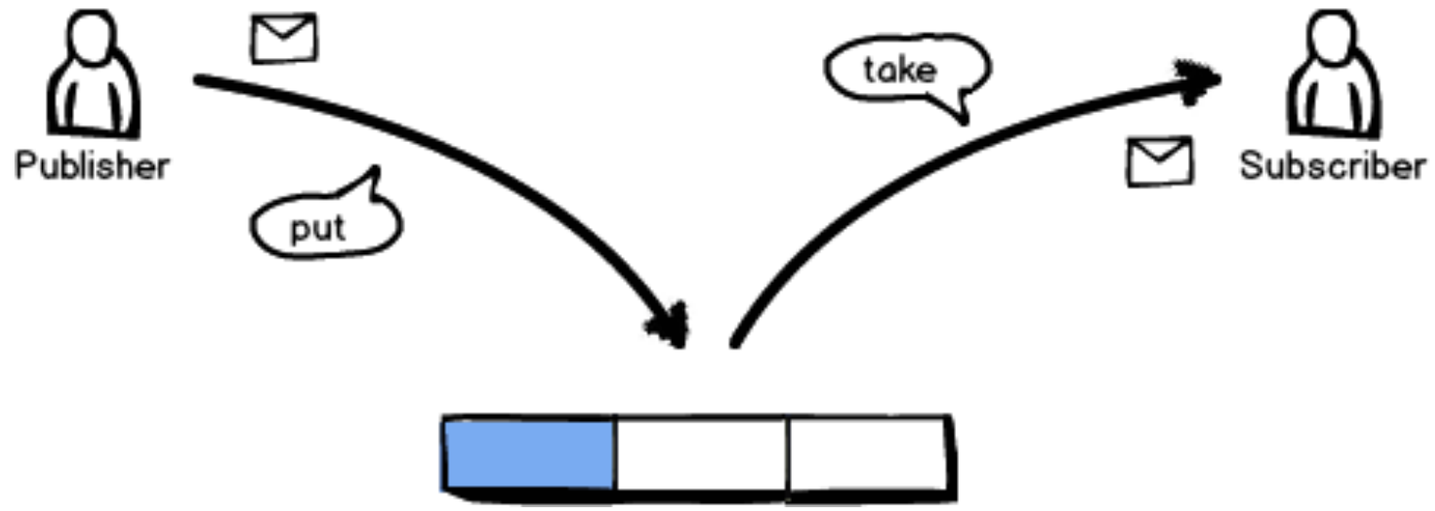



2 or more input ports and 1 output port

# Reactive Streams

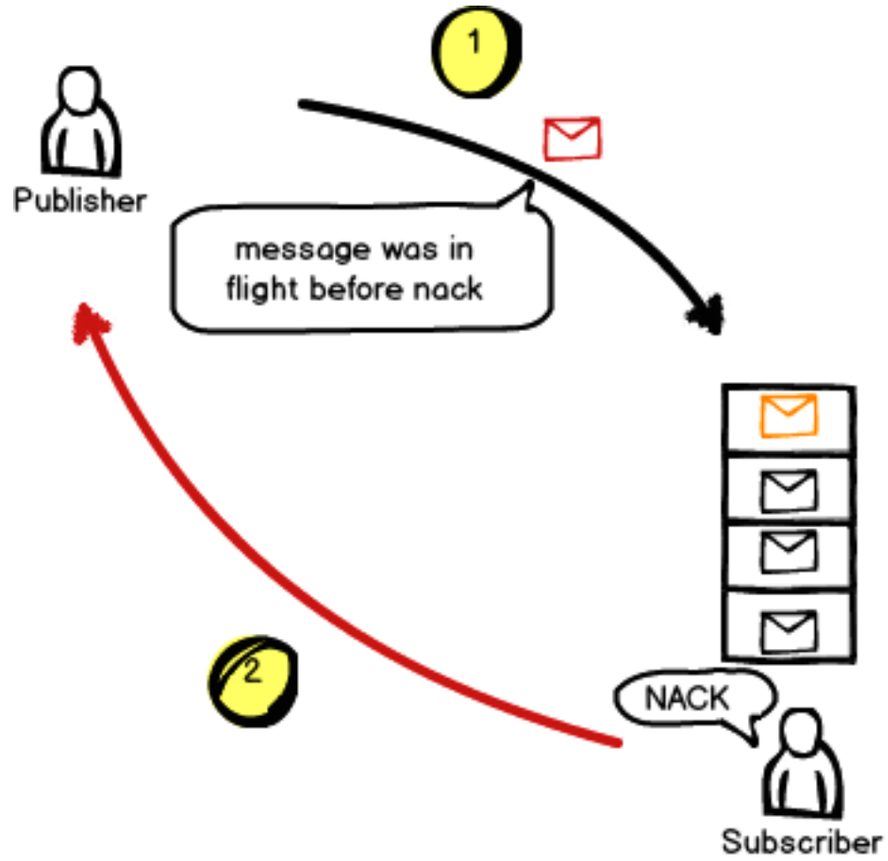
# Reactive Streams - back-pressure

- How handling data across asynchronous boundary ?
  - Blocking calls : queue with a bounded size



# Reactive Streams - back-pressure

- How handling data across asynchronous boundary ?
  - The Push way
    - unbounded buffer : Out Of Memory error
    - bounded buffer with NACK




NACK problem

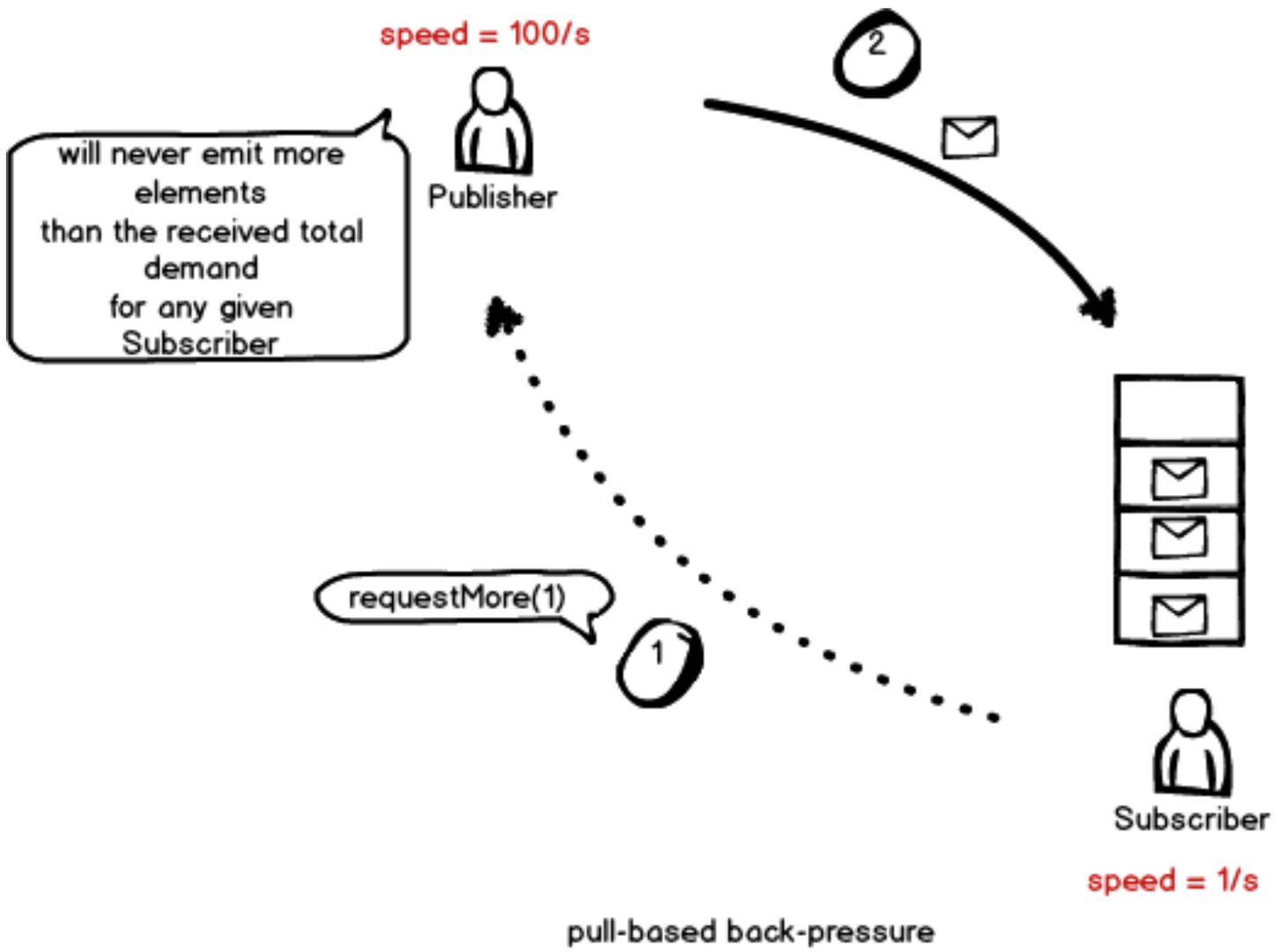
# Reactive Streams - back-pressure

- How handling data across asynchronous boundary ?
  - The reactive way : non-blocking, non-dropping & bounded
    - data items flow downstream
    - demand flows upstream
    - data items flow only when there is demand

# Reactive Streams - dynamic push / pull model

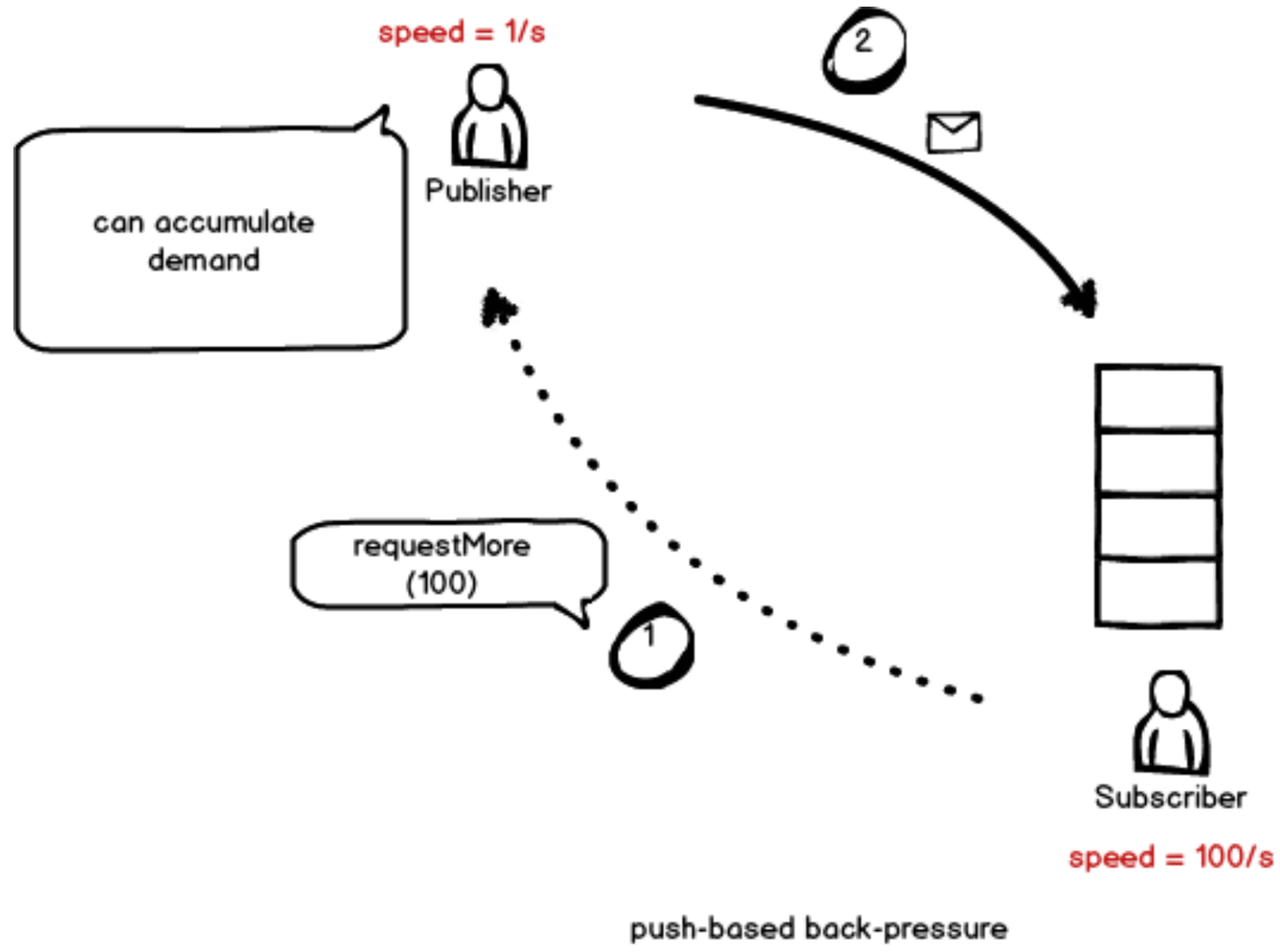
- “push” behaviour when Subscriber is faster
- “pull” behaviour when Publisher is faster
- switches automatically between those behaviours in runtime
- batching demand allows batching data

# Reactive Streams - back-pressure




pull-based back-pressure

# Reactive Streams - back-pressure

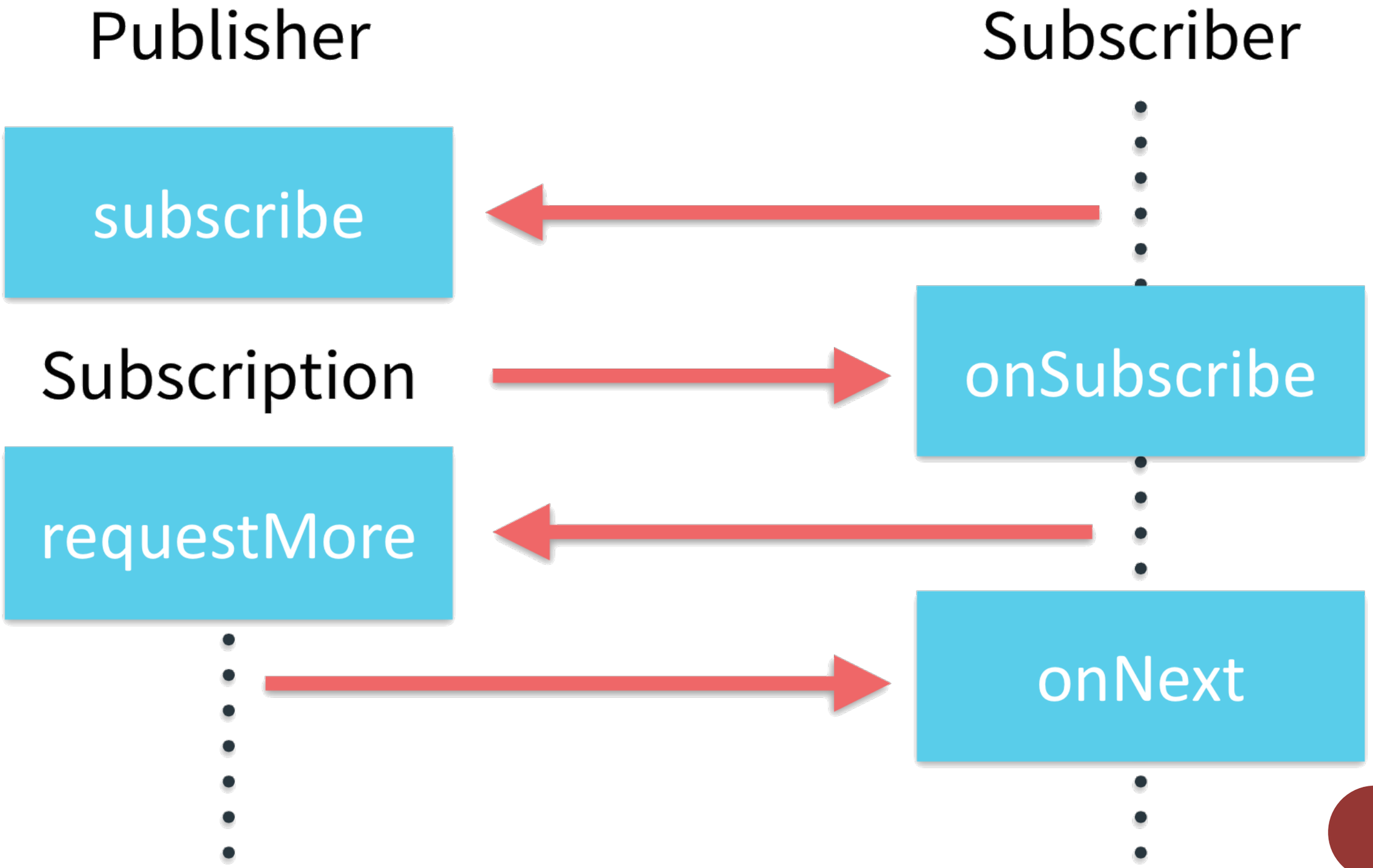



push-based back-pressure

# Reactive Streams - SPI

```
trait Publisher[T] {
  def subscribe(sub: Subscriber[T]): Unit
}

trait Subscription{
  def requestMore(n: Int): Unit
  def cancel(): Unit
}

trait Subscriber[T] {
  def onSubscribe(s: Subscription): Unit
  def onNext(elem: T): Unit
  def onError(thr: Throwable): Unit
  def onComplete(): Unit
}
```

# Reactive Streams - asynchronous non-blocking protocol



# Akka Streams - Pipeline Dataflow

# Akka Streams - Core Concepts

- a DSL implementation of Reactive Streams relying on Akka actors
- Stream which involves moving and/or transforming data
- Element : the processing unit of streams
- Processing Stage : building blocks that build up a Flow or FlowGraph (map(), filter(), transform(), junctions …)

# Akka Streams - Pipeline Dataflow

- Source : a processing stage with exactly one output, emitting data elements when downstream processing stages are ready to receive them
- Sink : a processing stage with exactly one input, requesting and accepting data elements
- Flow : a processing stage with exactly one input and output, which connects its up- and downstream by moving / transforming the data elements flowing through it
- Runnable Flow : a Flow that has both ends “attached” to a Source and Sink respectively
- Materialized Flow : a Runnable Flow that ran

# Akka Streams - Pipeline Dataflow

```
/**
 * Construct the ActorSystem we will use in our application
 */
implicit lazy val system = ActorSystem("DataFlow")

implicit val _ = ActorFlowMaterializer()

val source = Source(1 to 10)

val sink = Sink.fold[Int, Int](0)(_ + _)

val transform = Flow[Int].map(_ + 1).filter(_ % 2 == 0)

// connect the Source to the Sink, obtaining a runnable flow
val runnable: RunnableFlow = source.via(transform).to(sink)

// materialize the flow
val materialized: MaterializedMap = runnable.run()

implicit val dispatcher = system.dispatcher

// retrieve the result of the folding process over the stream
val result: Future[Int] = materialized.get(sink)
result.foreach(println)
```

# Akka Streams - Pipeline Dataflow

- Processing stages are immutable (connecting them returns a new processing stage)

```
val source = Source(1 to 10)
source.map(_ => 0) // has no effect on source, since it's immutable
source.runWith(Sink.fold(0)(_ + _)) // 55
```

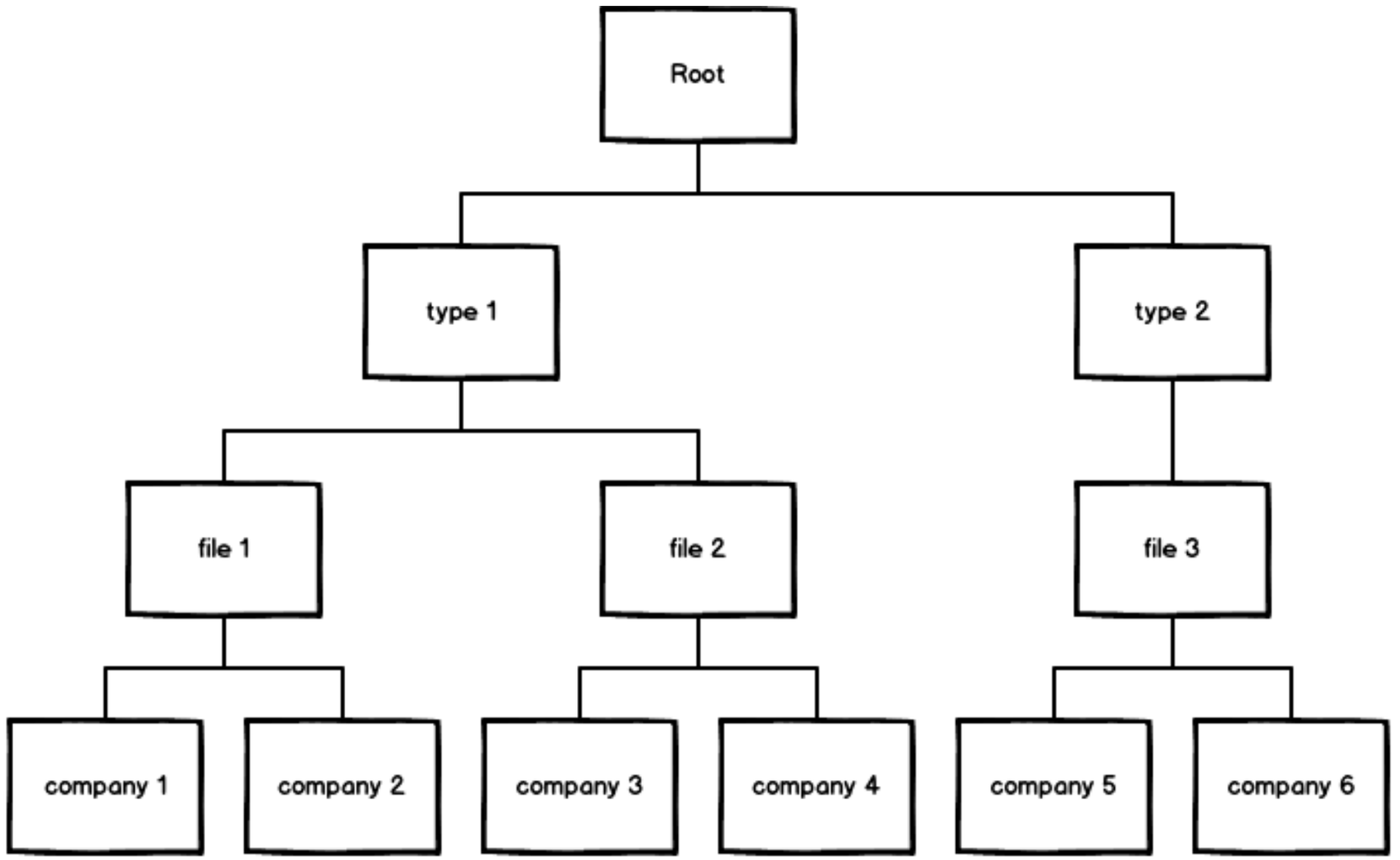
- Processing stages preserve input order of elements

- A stream can be materialized multiple times

```
// connect the Source to the Sink, obtaining a RunnableFlow
val sink = Sink.fold[Int, Int](0)(_ + _)
val runnable: RunnableFlow = Source(1 to 10).to(sink)
// get the materialized value of the FoldSink
val sum1: Future[Int] = runnable.run().get(sink)
val sum2: Future[Int] = runnable.run().get(sink)
// sum1 and sum2 are different Futures!
for(a <- sum1; b <- sum2) yield println(a + b) //110
```

# Akka Streams - working with graphs

# Akka Streams - bulk export to es

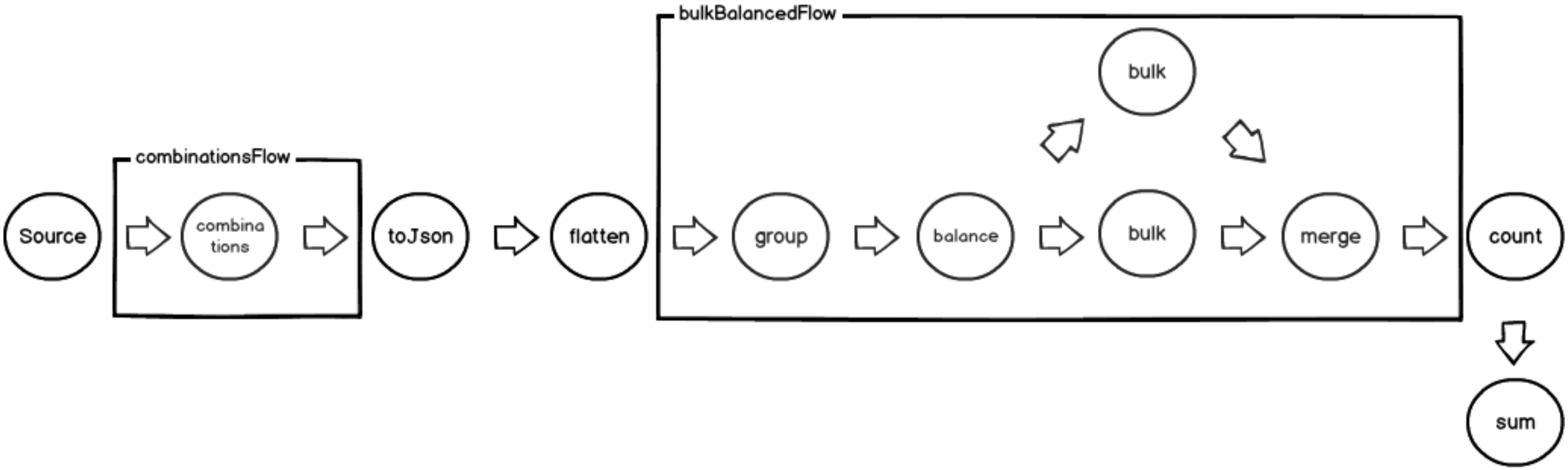



# Akka Streams - bulk export to es

```
val bulkSize: Int = 100

val source: Source[String] = Source(types.toList)

val sum: Sink[Int] = FoldSink[Int, Int](0)(_ + _)

val g:FlowGraph = FlowGraph{ implicit builder =>
  import FlowGraphImplicits._

  val walk: Flow[String, Companies2Export] = Flow[String]
    .map(fileTree(_).toList)
    .mapConcat[Companies2Export](identity)

  val parse: Flow[Companies2Export, List[Company]] = Flow[Companies2Export]
    .transform(() => new LoggingStage("ExportService"))
    .map[List[Company]](f => parseFile(f.source, index, f.`type`).toList)

  val flatten: Flow[List[Company], Company] = Flow[List[Company]].mapConcat[Company](identity)

  val group: Flow[Company, Seq[Company]] = Flow[Company].grouped(bulkSize)

  val toJson: Flow[Seq[Company], String] = Flow[Seq[Company]]
    .map(_.map(_.toBulkIndex(index.name)).mkString(crlf) + crlf)

  val bulkIndex: Flow[String, EsBulkResponse] = Flow[String]
    .mapAsyncUnordered[EsBulkResponse](esBulk(index.name, _))

  val count: Flow[EsBulkResponse, Int] = Flow[EsBulkResponse].map[Int]((b)=>{
    logger.debug(s"index ${b.items.size} companies within ${b.took} ms")
    b.items.size
  })

  // connect the graph
  source ~> walk ~> parse ~> flatten ~> group ~> toJson ~> bulkIndex ~> count ~> sum
}

val result: Future[Int] = g.runWith(sum)

result.foreach(c => logger.info(s"*** total companies indexed: $c"))

result.onComplete(_aliases(index.name, before))
```

# Akka Streams - frequent item set


(1, 2, 3, 4) (1), (1, 2), (1, 3), (1, 4), (1, 2, 3), (1, 2, 4), (1, 3, 4),...



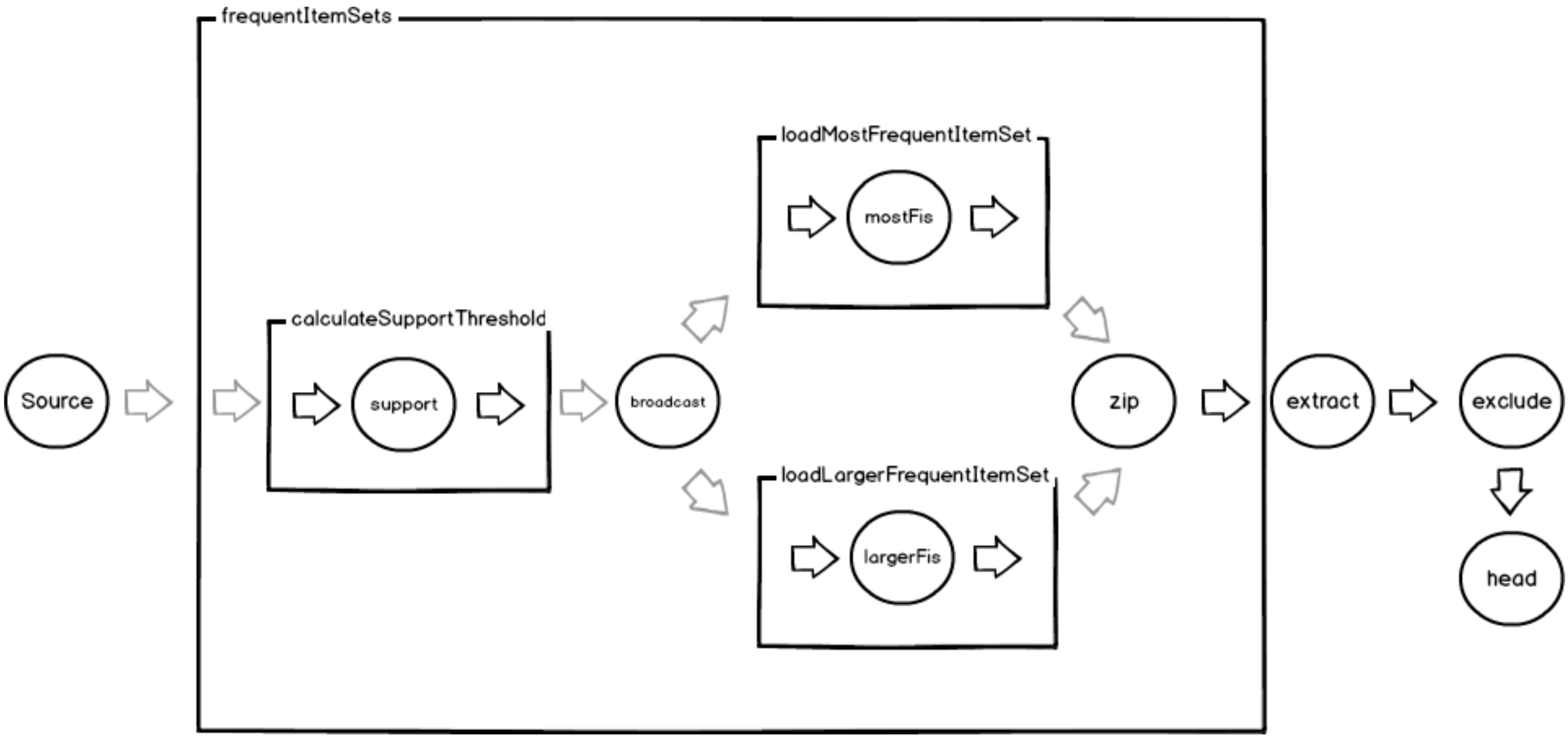
# Akka Streams - frequent item set

```
val g:FlowGraph = FlowGraph{ implicit builder =>
  import FlowGraphImplicits._

  val source = Source.single(sorted)

  import com.sksamuel.elastic4s.ElasticDsl._

  val toJson = Flow[List[Seq[Long]]].map(_.map(seq => {
    val now = Calendar.getInstance().getTime
    val uuid = seq.mkString("-")
    update(uuid)
      .in(s"${esInputStore(store)}/CartCombination")
      .upsert(
        "uuid" -> uuid,
        "combinations" -> seq.map(_.toString),
        "counter" -> 1,
        "dateCreated" -> now,
        "lastUpdated" -> now)
      .script("ctx._source.counter += count;ctx._source.lastUpdated = now")
      .params("count" -> 1, "now" -> now)
  }))

  val flatten = Flow[List[BulkCompatibleDefinition]].mapConcat[BulkCompatibleDefinition](identity)

  val count = Flow[BulkResponse].map[Int]((resp)=>{
    val nb = resp.getItems.length
    logger.debug(s"index $nb combinations within ${resp.getTookInMillis} ms")
    nb
  })

  import EsClient._

  source ~> combinationsFlow ~> toJson ~> flatten ~> bulkBalancedFlow() ~> count ~> sum
}

val start = new Date().getTime

val result: Future[Int] = g.runWith(sum)

result.foreach(c => logger.debug(s"*** $c combinations indexed within ${new Date().getTime - start} ms"))
```

# Akka Streams - frequent item set

```
val combinationsFlow = Flow(){implicit b =>
  import FlowGraphImplicits._

  val undefinedSource = UndefinedSource[Seq[Long]]
  val undefinedSink = UndefinedSink[List[Seq[Long]]]

  undefinedSource ~> Flow[Seq[Long]].map[List[Seq[Long]]](s => {
    val combinations : ListBuffer[Seq[Long]] = ListBuffer.empty
    1 to s.length foreach {i => combinations ++= s.combinations(i).toList}
    combinations.toList
  }) ~> undefinedSink

  (undefinedSource, undefinedSink)
}
def bulkBalancedFlow(bulkSize: Int = Settings.ElasticSearch.bulkSize, balanceSize: Int = 2) =
  Flow(){implicit b =>
    import FlowGraphImplicits._

    val in = UndefinedSource[BulkCompatibleDefinition]
    val group = Flow[BulkCompatibleDefinition].grouped(bulkSize)
    val bulkUpsert = Flow[Seq[BulkCompatibleDefinition]].map(bulk)
    val out = UndefinedSink[BulkResponse]

    if(balanceSize > 1){

      val balance = Balance[Seq[BulkCompatibleDefinition]]
      val merge = Merge[BulkResponse]

      in ~> group ~> balance
      1 to balanceSize foreach { _ =>
        balance ~> bulkUpsert ~> merge
      }
      merge ~> out

    }
    else{
      in ~> group ~> bulkUpsert ~> out
    }

    (in, out)
  }
```

# Akka Streams - frequent item set



# Akka Streams - frequent item set

```
def fis(store: String, productId: String, frequency: Double = 0.2): Future[(Seq[String], Seq[String])] = {
  implicit val _ = ActorFlowMaterializer()

  import com.mogobiz.run.learning._

  val source = Source.single((productId, frequency))

  val extract = Flow[(Option[CartCombination], Option[CartCombination])].map((x) => {
    (
      x._1.map(_.combinations.toSeq).getOrElse(Seq.empty),
      x._2.map(_.combinations.toSeq).getOrElse(Seq.empty))
  })

  val exclude = Flow[(Seq[String], Seq[String])]
    .map(x => (x._1.filter(_ != productId), x._2.filter(_ != productId)))

  val head = Sink.head[(Seq[String], Seq[String])]

  val runnable:RunnableFlow = source
    .transform(() => new LoggingStage[(String, Double)]("Learning"))
    .via(frequentItemSets(store))
    .via(extract)
    .via(exclude)
    .to(head)

  runnable.run().get(head)
}

def frequentItemSets(store: String) = Flow() { implicit builder =>
  import FlowGraphImplicits._

  val undefinedSource = UndefinedSource[(String, Double)]
  val broadcast = Broadcast[Option[(String, Long)]]
  val zip = Zip[Option[CartCombination], Option[CartCombination]]
  val undefinedSink = UndefinedSink[(Option[CartCombination], Option[CartCombination])]

  undefinedSource ~> calculateSupportThreshold(store) ~> broadcast
  broadcast ~> loadMostFrequentItemSet(store)   ~> zip.left
  broadcast ~> loadLargerFrequentItemSet(store) ~> zip.right
  zip.out ~> undefinedSink

  (undefinedSource, undefinedSink)
}
```

# Akka Streams - frequent item set

```
def calculateSupportThreshold(store:String) = Flow[(String, Double)].map { tuple =>
  load[CartCombination](esInputStore(store), tuple._1).map { x =>
    Some(tuple._1, Math.ceil(x.counter * tuple._2).toLong)
  }.getOrElse(None)
}

def loadMostFrequentItemSet(store: String) = Flow[Option[(String, Long)]].map(_.map((x) =>
  searchAll[CartCombination](cartCombinations(store, x) sort {
    by field "counter" order SortOrder.DESC
  }).headOption).getOrElse(None))

def loadLargerFrequentItemSet(store: String) = Flow[Option[(String, Long)]].map(_.map((x) =>
  searchAll[CartCombination](cartCombinations(store, x) sort {
    by script "doc['combinations'].values.size()" order SortOrder.DESC
  }).headOption).getOrElse(None))
```