# Akka Streams

Franz Thoma

BOB 2018, Berlin, 2018-02-23

# Streaming for Big Data Applications

# Streaming Big Data

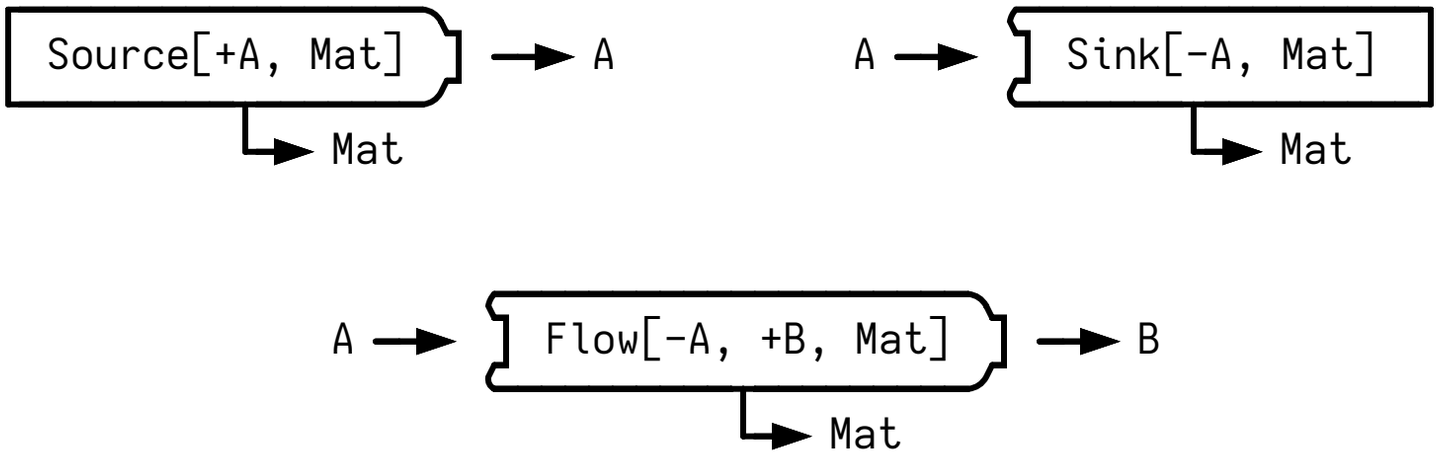
- Billions of events per day (Terabytes!)
- (Near) real-time processing
- Fault tolerance
- Bounded: Batch processing
- Unbounded: Stream processing

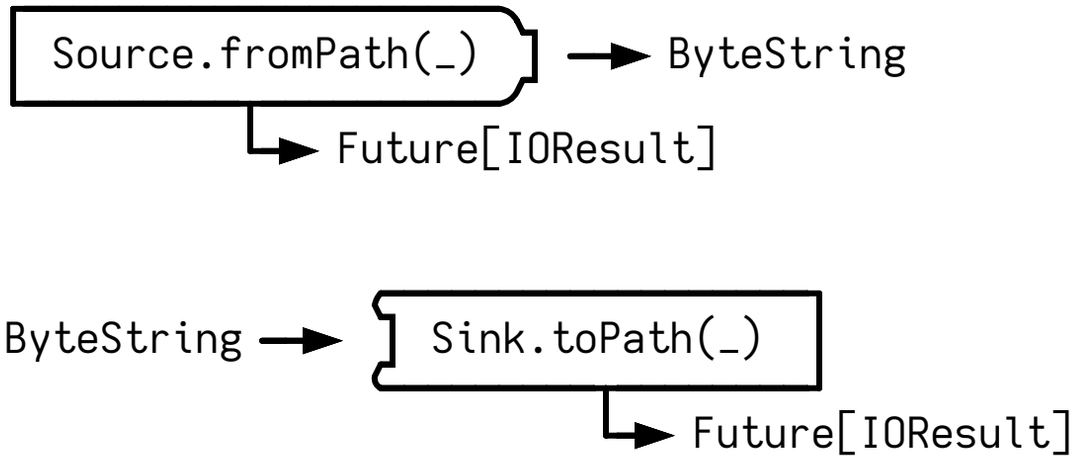
# Why Use Akka Streams?

- Type-safe
- Compositional
- High-level
- Explicit semantics
- Integrates well (Alpakka)
- Fast (fusion & other optimizations!)

# Building Blocks of Akka Streams

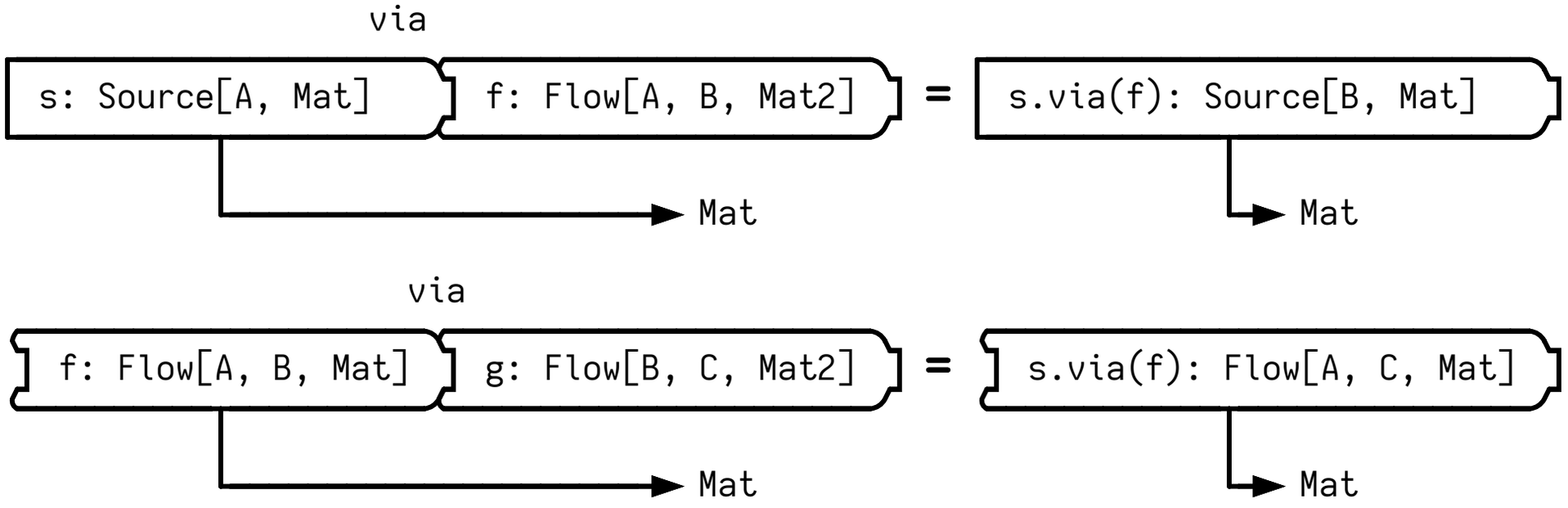
# Sources, Sinks and Flows




Source.fromPath(_)
ByteString
Future[IOResult]
ByteString
Sink.toPath(_)
Future[IOResult]

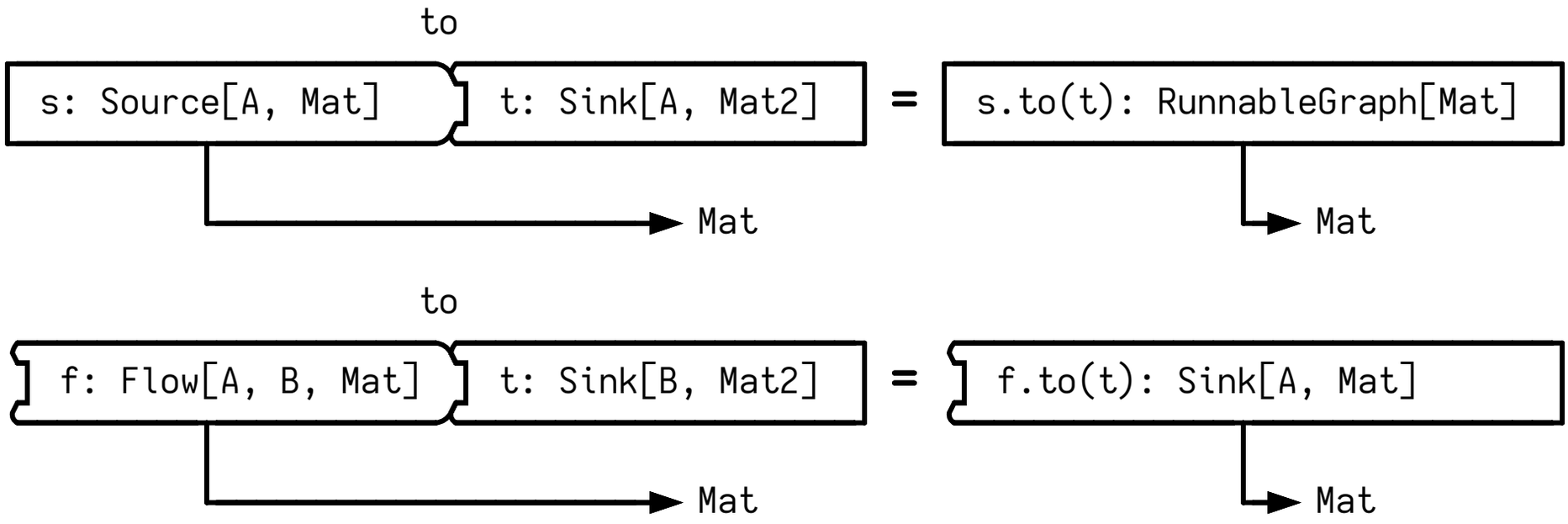
# Connecting Stream Elements: `via`

`via` composes the the outlet of a `Source` or `Flow` with another `Flow`, keeping the materialized value:

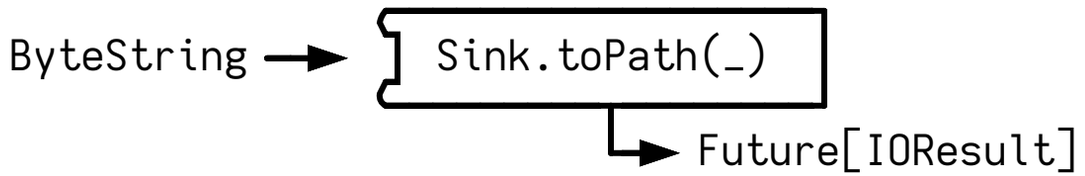


# Connecting Stream Elements: `to`

`to` connects the the outlet of a `Source` or `Flow` to a `Sink`, keeping the materialized value:



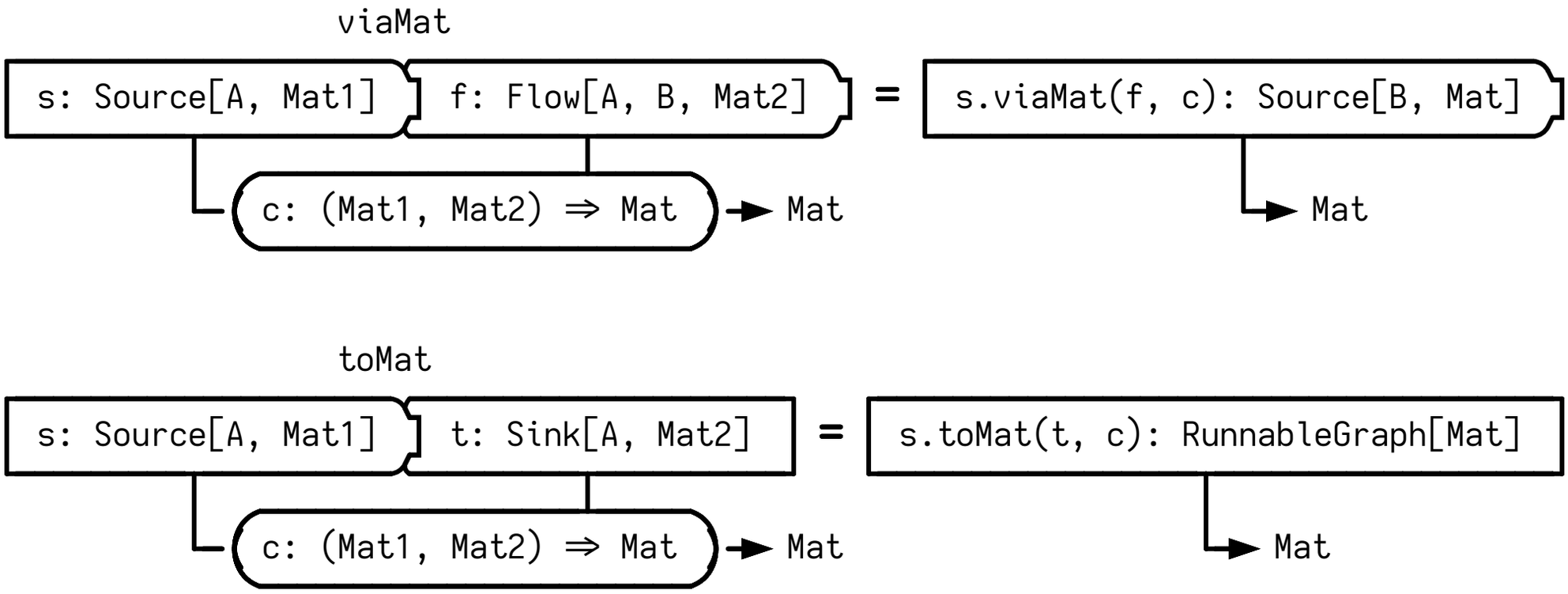
# Materialized Values



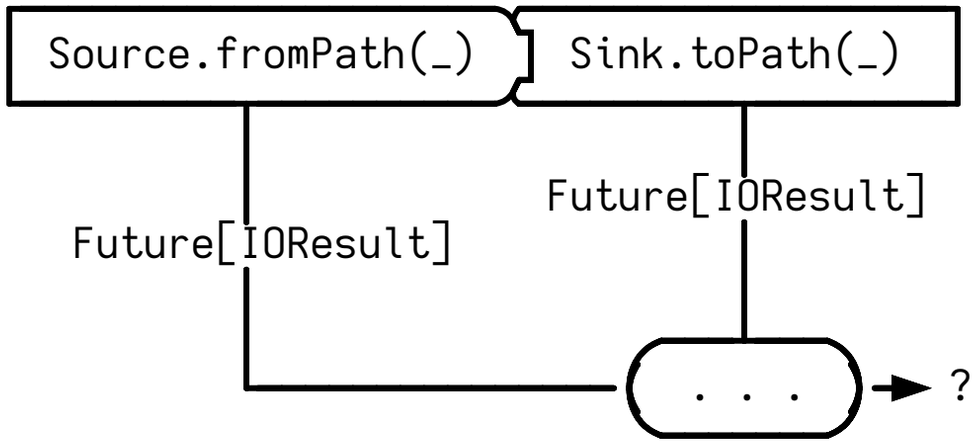

Each stream element allows to return some information on the items processed.

- `NotUsed` (no information available)
- The number of elements processed
- The result (Success/Failure) of an IO action (`Future[IOResult]`)
- Items collected from the stream (`Future[Seq[A]]`)

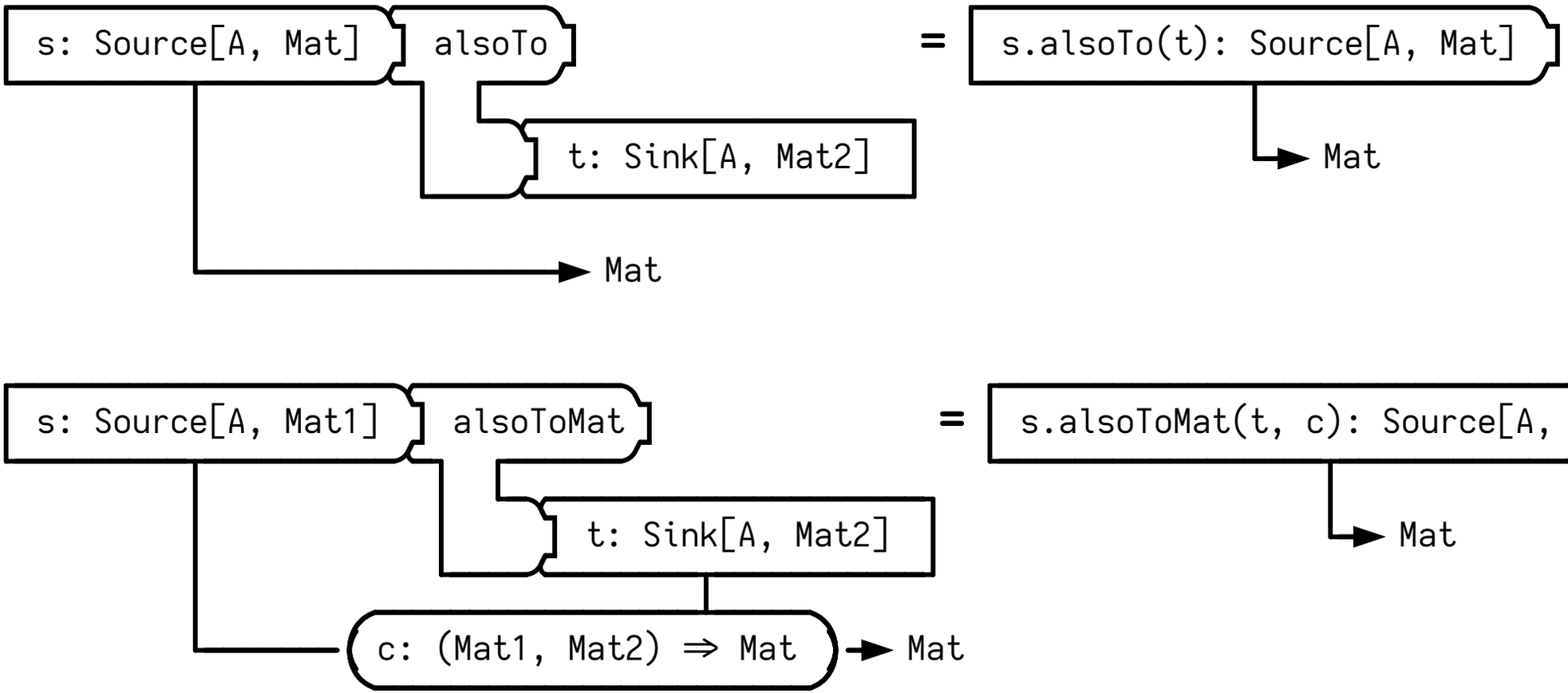
# But What About the Materialized Value?




- Keep.left: (L, R) → L
- Keep.right: (L, R) → R


Source.fromPath(_)
Sink.toPath(_)
Future[IOResult]
Future[IOResult]
. . .
?

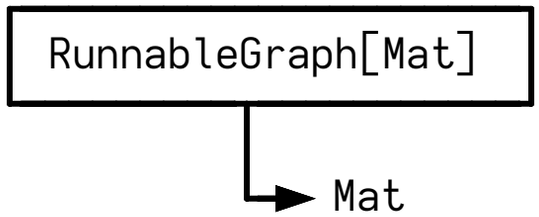
# Tee Pieces: `alsoTo` and `alsoToMat`



# Some `Source`, `Flow` and `Sink` examples

```
Source[T](xs: Iterable[T]): Source[T, NotUsed]

Sink.ignore: Sink[Any, Future[Done]]

Sink.foreach[T](f: T ⇒ Unit): Sink[T, Future[Done]]
// e.g. Sink.foreach(System.out.println(_))

Sink.fold[U, T](zero: U)(f: (U, T) ⇒ U): Sink[T, Future[U]]

Flow.fromFunction[A, B](f: A ⇒ B): Flow[A, B, NotUsed]

FileIO.fromPath(f: Path): Source[ByteString, Future[IOResult]]
FileIO.toPath(f: Path): Sink[ByteString, Future[IOResult]]

// Akka HTTP client/server is a Flow[HttpRequest, HttpResponse, …]:
Http.outgoingConnection(…): Flow[HttpRequest, HttpResponse, Future[OutgoingConne
Http.bindAndHandle(handler: Flow[HttpRequest, HttpResponse, Any], …)
```

# Runnable Graphs




A `RunnableGraph` is a black box that neither consumes nor produces items, but it still returns a materialized value.

# Materialization

`Sources`, `Sinks` and `Flows` are just *blueprints*.
`RunnableGraph.run` builds and optimizes the actual stream.

```
implicit val system : ActorSystem = ActorSystem()
implicit val materializer : Materializer = ActorMaterializer()

val blueprint: RunnableGraph[Future[Int]] =
  Source(List(1, 2, 3)).toMat(Sink.fold(0)(_ + _))(Keep.right)

val result: Future[Int] = blueprint.run
```

# Backpressure

# What is Backpressure?

```
FileIO.fromPath(Paths.get("requests.txt"))                          // ← fast-ish
    .via(Framing.delimiter("\n", 1024))                             // ← fast
    .via(Fold.fromFunction(request ⇒ send(request)))                // ← slow
    .to(Sink.foreach(response ⇒ System.out.println(response)))      // ← fast
```

Default backpressuring: Only produce/consume as fast as the slowest link in the chain.

# Backpressure Boundaries

## What if I can't (or don't want to) control the speed of a Source?

```
incomingRequests       // ← will turn away requests if they come too fast
    .to(slowSink)
```

```
incomingRequests                                    // ← won't turn away requests
    .buffer(50, OverflowStrategy.dropNew)           // ← may lose requests
    .to(slowSink)
```

```
incomingRequests                                     // ← will turn away requests
    .buffer(50, OverflowStrategy.backpressure)       //    if buffer is full
    .to(slowSink)
```

# Throttling

## What if a `Sink` chokes if items come in too fast?

```
fastSource                                                    // ← fast
    .to(chokingSink)                                          // ← chokes :-(
```

```
fastSource                                                    // ← fast
    .throttle(elements = 5, per = 1 second, mode = shaping)   // ← slow down!
    .to(chokingSink)                                          // ← doesn't choke
```

# Backpressure Boundaries (II)

```
fastSource
    .alsoTo(slowSink)        // ← slows everything down :-(
    .to(fastSink)
```

```
fastSource
    .alsoTo(Flow()
        .buffer(50, backpressure)  // ← Tries to buffer
        .to(slowSink))             //    before slowing everything down
    .to(fastSink)
```

Particularly useful if the source produces at irregular intervals.

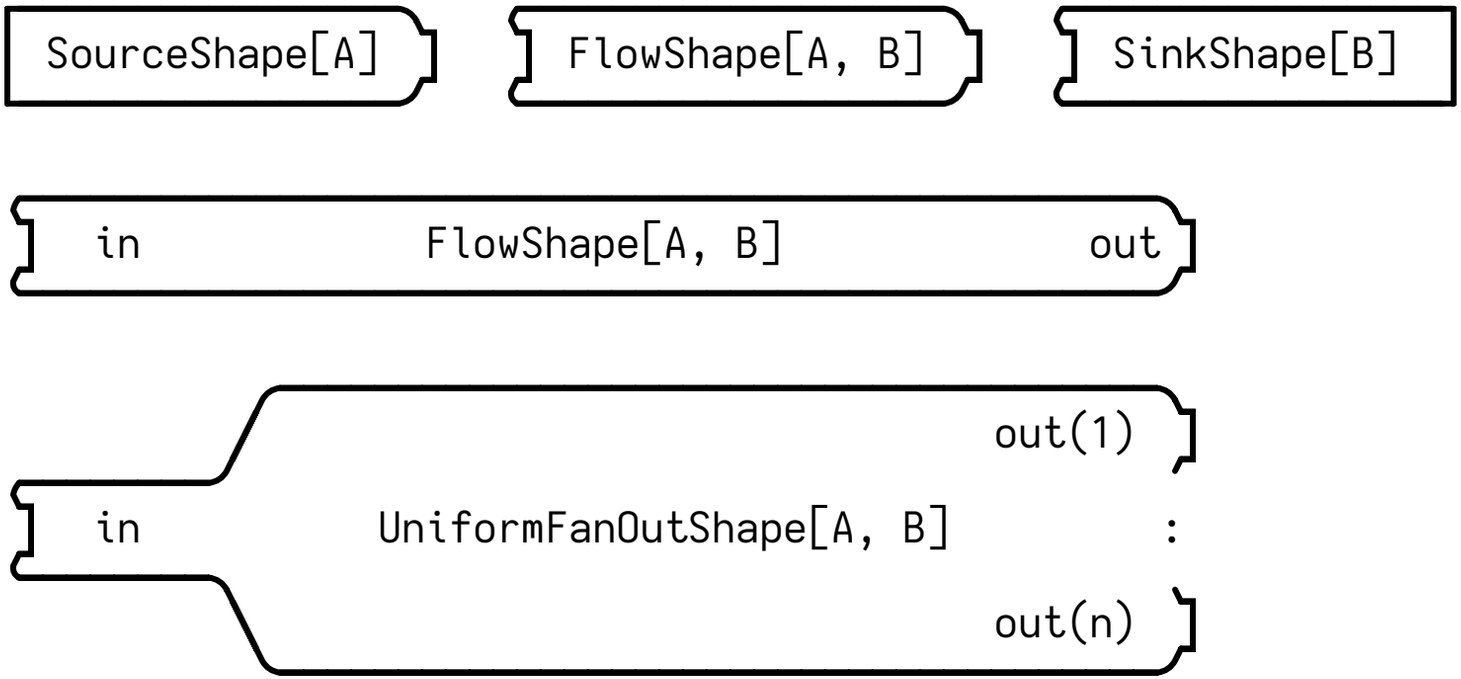
# Batching

Another way to connect a fast `Source` to a slow `Sink`:

```
fastSource
    .batch(max = 10, seed = List(_))(_ :+ _)   // ← backpressures only if
                                               //    batch size is exceeded
    .to(slowBatchSink)                         // ← consumes batches faster
                                               //    than individual elements
```

# Graph DSL for More Complex Streams

# Talk About Shapes



# Shapes & Graphs

`Flow[A, B, Mat]`
is just a decorator around
`Graph[FlowShape[A, B], Mat]`

# Some Useful Shapes

- `SourceShape[A]`(`Source[A, Mat]`)
- `SinkShape[A]`(`Sink[A, Mat]`)
- `FlowShape[A, B]`(`Flow[A, B, Mat]`)
- `ClosedShape`(`RunnableGraph[Mat]`)
- `FanOutShape2[A, B1, B2]`, `FanInShape2[A1, A2, B]` (up to 22 inlets/outlets)
- `BidiShape[In1, Out1, In2, Out2]`

# Constructing Graphs from Shapes

## Balancing between workers:

```
def balanced[S, T, Mat >: Any](workers: Seq[Graph[FlowShape[S, T], Mat]]): Flow[
  Flow.fromGraph(GraphDSL.create() { implicit builder ⇒

    import GraphDSL.Implicits._

    val n = workers.length
    val balance: UniformFanOutShape[S, S] = builder.add(Balance[S](n))
    val merge: UniformFanInShape[T, T] = builder.add(Merge[T](n))

    for (i ← 0 until n) {
      balance.out(i) ~> workers(i).async ~> merge.in(i)
    }

    FlowShape(balance.in, merge.out)
  })
```

## Also one way to speed up slow `Flow` elements!

# Connecting to the World

# Akka Streams Connectors

- Akka HTTP
- Slick (JDBC, Functional Relational Mapping)
- Apache Kafka
- Apache Camel
- AWS (S3, Kinesis, …)
- Have a look at Alpakka for more connectors

Thank you!

# Questions?

Slides on Github: TBD

fmthoma on Github

fmthoma on keybase.io

franz.thoma@tngtech.com