scala wave

GFT

SCALAWAVE.IO 2016

Applied Reactive Streaming Fast Data Processing with Akka

Jan Pustelnik @gosubpl

24 November 2016

Why GoSub?

Image source: Sinclair ZX Spectrum keys with BASIC keywords: Bill Bertram CC-BY SA 2.5, via Wikimedia Commons.

GoSub strikes back ☺

gosubpl / akka-online

Online and streaming algorithms with Akka

For good overview of streaming algos go to this debasishg's gist.

In `org.apache.spark.streamdm` there is code copied from Huawei Noah's Ark Lab streamDM project (the licencse is Apache 2.0 too). The code has been adapted to work with Akka streams instead of Spark streaming module by removing dependencies on the Spark stuff. Tested path is the `HoeffdingTree` model usage, which works best with input sourced from `Arff` files via `SpecificationParser` / `ExampleParser` - see see other parts of `akka-online` for usage examples. If you'd like to know more about HoeffdingTree on-line classifier, please read the HDT docs or go to Massive Online Analysis website that contains more information on the context.

Sample `Arff` files can be found in the MOA dataset repository.

For theoretical exposition to `HoeffdingTree` usage go to the original paper.

If you are asking yourself question *why Akka, not Spark?* - please read the classic bigger data, same laptop to see what we can gain *going lightweight*. In previous century, 4.5k records machine learning data set could be considered sizeable and hence all the growth of *clustering* for data processing. But with today's hardware I believe *we can do better*.

In the `/lib` directory of the project I have put `suffixtree-1.0.0-SNAPSHOT.jar` which is a compiled artifact of the Ukkonen's on-line Suffix Tree implementation. The licensse for this project is also Apache 2.0.

In this project I also use Guava implementation of `Bloom Filter` (see project dependencies) but there are other options.

Community, for teh win!

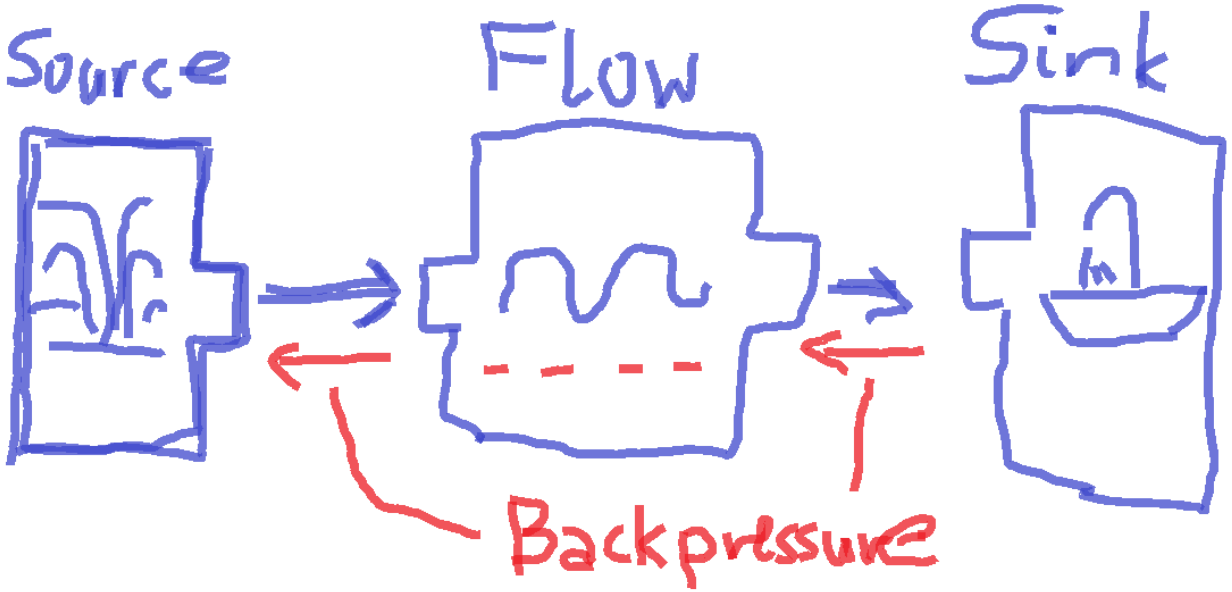
scala wave

GFT

- While preparing this slide deck I have received help from the wide Akka community:
  - all helpful people on *akka/akka* and *akka/dev* gitter chats
  - *@ktoso*, *@hseeberger*, *@rbudzko* specifically

THANKS!

Reactive Streams Made Easy



Backpressure ftw!

scala wave

GFT

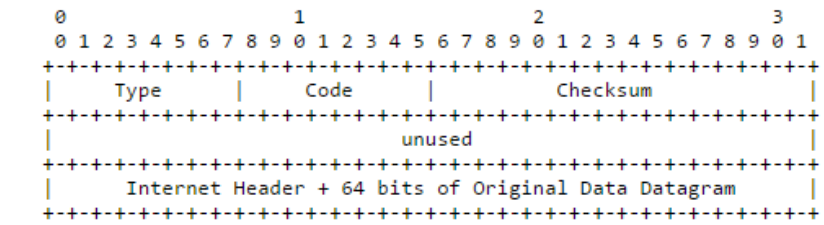
September 1981

RFC 792

Flow Control:

TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

Source Quench Message



gateway may send a source quench message for every message that it discards. On receipt of a source quench message, the source host should cut back the rate at which it is sending traffic to the specified destination until it no longer receives source quench messages from the gateway. The source host can then gradually increase the rate at which it sends traffic to the destination until it again receives source quench messages.

The gateway or host may send the source quench message when it approaches its capacity limit rather than waiting until the capacity is exceeded. This means that the data datagram which triggered the source quench message may be delivered.

Staged event-driven architecture, ftw! (2001)

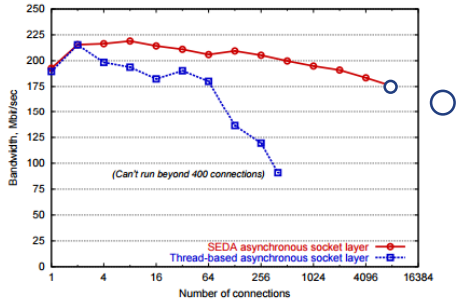


Figure 11: **Asynchronous sockets layer performance:** *This graph shows the performance of the SEDA-based asynchronous socket layer as a function*

SEDA is fast because async and has buffers and stages are single threaded

SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

Matt Welsh, David Culler, and Eric Brewer
Computer Science Division
University of California, Berkeley
{mdw,culler,brewer}@cs.berkeley.edu

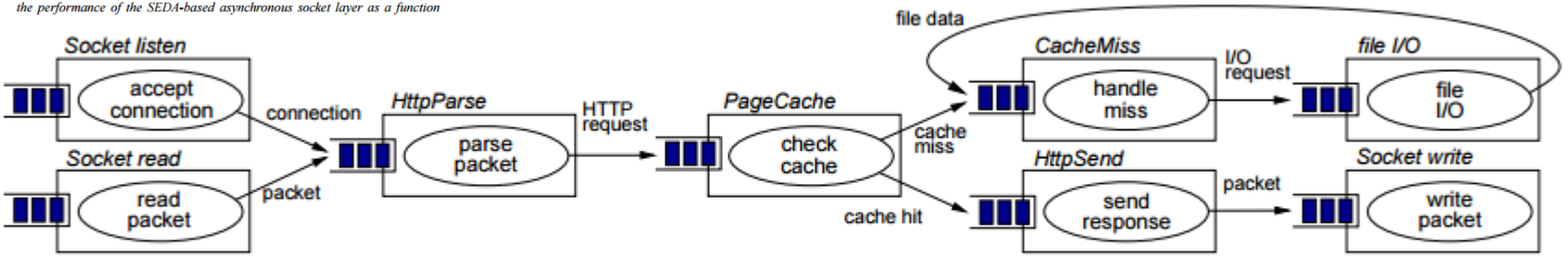
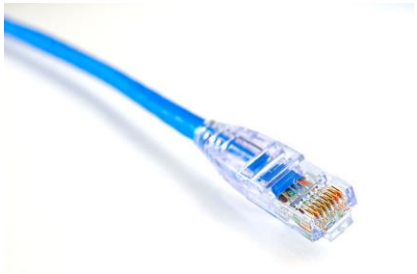


Figure 5: **Staged event-driven (SEDA) HTTP server:** *This is a structural representation of the SEDA-based Web server, described in detail in Section 5.1. The application is composed as a set of* stages *separated by* queues. *Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue. For simplicity, some event paths and stages have been elided from this figure.*

On-line algorithms / data structures



On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future?

Richard M. Karp*

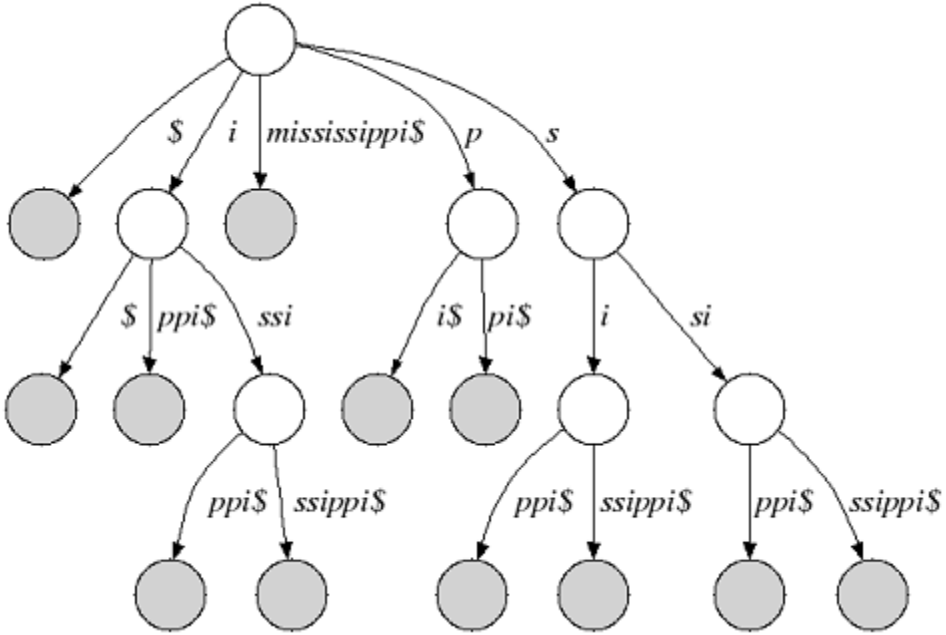
TR-92-044

July 1992

Abstract

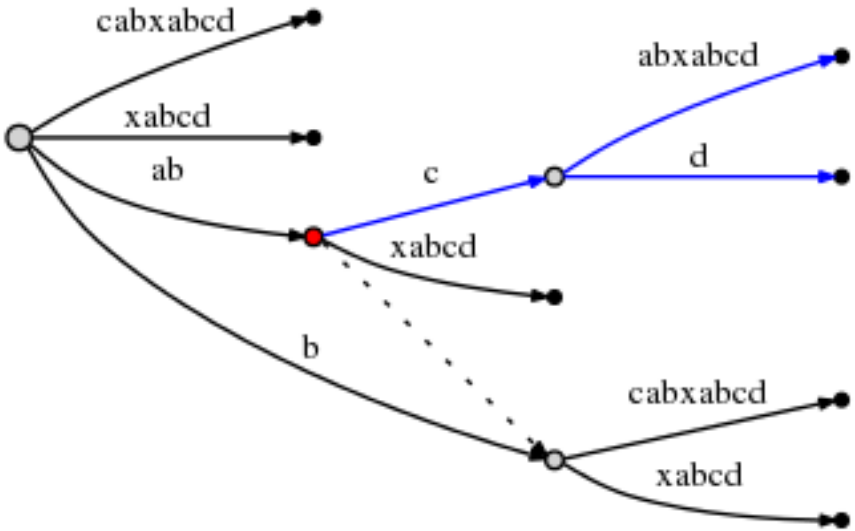
An *on-line algorithm* is one that receives a sequence of requests and performs an immediate action in response to each request. On-line algorithms arise in any situation where decisions must be made and resources allocated without knowledge of the future. The effectiveness of an on-line algorithm may be measured by its *competitive ratio*, defined as the worst-case ratio between its cost and that of a hypothetical off-line algorithm which knows the entire sequence of requests in advance and chooses its actions optimally. In a variety of settings, we discuss techniques for proving upper and lower bounds on the competitive ratios achievable by on-line algorithms. In particular, we discuss the advantages of randomized on-line algorithms over deterministic ones.

Suffix tree



Source: http://docs.seqan.de/seqan/1.2/PAGE_Indices.html

Ukkonen algorithm for on-line Suffix tree construction



Algorithmica (1995) 14: 249–260

Algorithmica

On-Line Construction of Suffix Trees[1]

E. Ukkonen[2]

Abstract. An on-line algorithm is presented for constructing the suffix tree for a given string in time linear in the length of the string. The new algorithm has the desirable property of processing the string symbol by symbol from left to right. It always has the suffix tree for the scanned part of the string ready. The method is developed as a linear-time version of a very simple algorithm for (quadratic size) suffix *tries*. Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long. Another variation of this method is shown to give, in a natural way, the well-known algorithms for constructing suffix automata (DAWGs).

Key Words. Linear-time algorithm, Suffix tree, Suffix trie, Suffix automaton, DAWG.

Source: http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english

Genome samples contamination - algorithm

scala wave

GFT

- Build *on-line* **generalized suffix tree** based on the genome signatures of known contaminants

  {aggtca, attgca, gatttaca}

- Look-up whether our specimen does not contain a substring common with one of the known contaminants

  ttg -> {aggtca, a**ttg**ca, gatttaca}

- There can be new known contaminants… add them as they go
- Repeat search if necessary…

Genome samples contamination – demo

Streaming algorithms

Jérémie Lumbroso's talk at the AK Data Science Summit on Streaming and Sketching in Big Data and Analytics.

Source: https://speakerdeck.com/timonk/philippe-flajolets-contribution-to-streaming-algorithms

- ONE PASS (on-line algorithm)
- **constant/logarithmic memory**

  ...
- *1970, 1983 … old stuff* ☺

For more: https://gist.github.com/debasishg/8172796

Historical context

- **1970**: average-case → deterministic algorithms on random input
- **1976-78**: first randomized algorithms (primality testing, matrix multiplication verification, find nearest neighbors)
- **1979**: Munro and Paterson, find median in one pass with $\Theta(\sqrt{n})$ space with high probability
  ⇒ (almost) first streaming algorithm

In **1983**, Probabilistic Counting by Flajolet and Martin is (more or less) the first streaming algorithm (one pass + constant/logarithmic memory).

Combining both versions: cited about 750 times = **second most cited** element of Philippe's bibliography, after only *Analytic Combinatorics*.

4/22

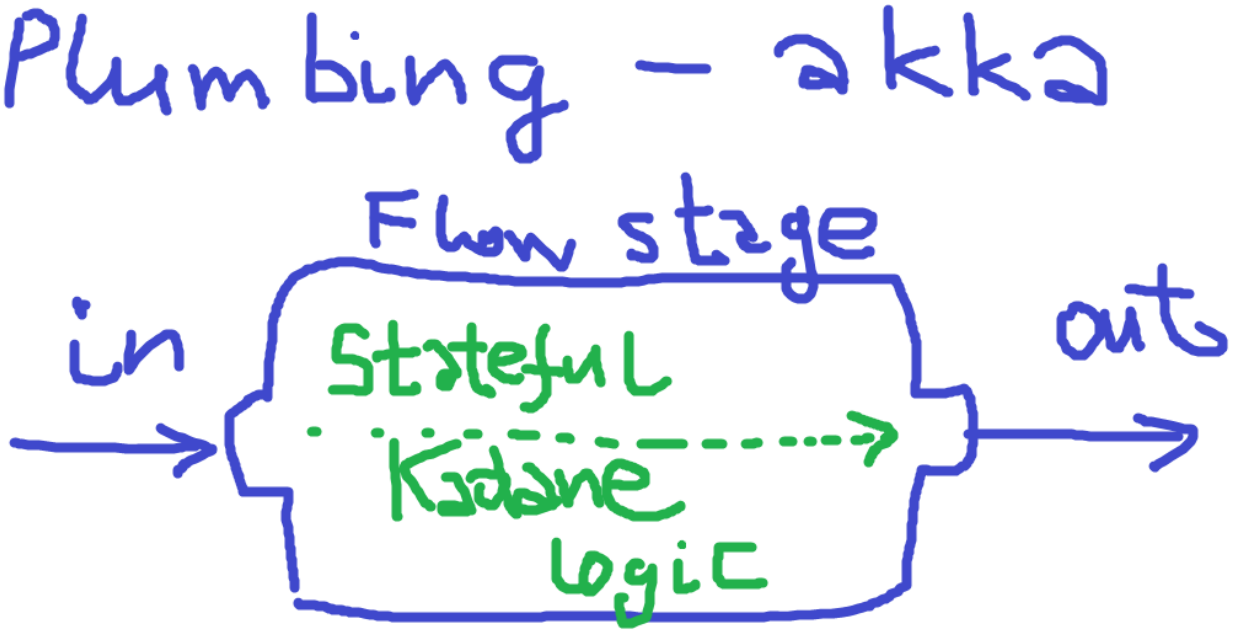
Example: Kadane algorithm (on-line, streaming, 1984)

(…) the **maximum subarray problem** is the task of finding the contiguous subarray within a one-dimensional array of numbers which has the largest sum. For example, for the sequence of values −2, 1, −3, 4, −1, 2, 1, −5, 4; the contiguous subarray with the largest sum is 4, −1, 2, 1, with sum 6.

```
def max_subarray(A):
    max_ending_here = max_so_far = 0
    for x in A:
        max_ending_here = max(0, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

Source: https://en.wikipedia.org/wiki/Maximum_subarray_problem

KadaneFlowStage

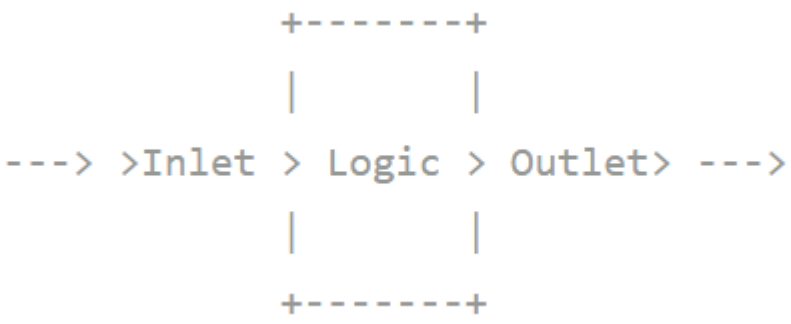


Flow Shape (plumbing)

scala wave

GFT

```
// Shape definition
val in: Inlet[Int] = Inlet("KadaneFlowStage.in")
val out: Outlet[Int] = Outlet("KadaneFlowStage.out")
override val shape: FlowShape[Int, Int] = FlowShape(in, out)
```



Flow Shape – output handler (plumbing)

GFT

```
// Handler for the Outlet
setHandler(out, new OutHandler {
  override def onPull(): Unit = {
    if (!hasBeenPulled(in))
      pull(in)
  }
})
```

Flow Shape – output handler (plumbing)

scala wave

GFT

```
// Handler for the Outlet
setHandler(out, new OutHandler {
  override def onPull(): Unit = {
    if (!hasBeenPulled(in))
      pull(in)
  }
})
```

PDD – Pull Driven Development

Flow Shape – output handler (plumbing)

scala wave

GFT

```
// Handler for the Outlet
setHandler(out, new OutHandler {
  override def onPull(): Unit = {
    if (!hasBeenPulled(in))
      pull(in)
  }
})
```

Captain: for great backpressure!

艦長：我々の未来に希望を・・・

Flow stage – “Business logic”

```
new GraphStageLogic(shape) {
  // state
  var maxEndingHere = 0
  var maxSoFar = 0

  // Handler(s) for the Inlet
  setHandler(in, new InHandler {
    // what to do when a new element is ready to be consumed
    override def onPush(): Unit = {
      val elem = grab(in)

      // "Business" logic
      maxEndingHere = Math.max(0, maxEndingHere + elem)
      maxSoFar = Math.max(maxSoFar, maxEndingHere)

      // this should never happen
      // we decide to not push the value, avoiding the error
      // but potentially losing the value
      if (isAvailable(out))
        push(out, maxSoFar)
    }
```

Flow stage – “Business logic”

```
new GraphStageLogic(shape) {
  // state
  var maxEndingHere = 0
  var maxSoFar = 0

  // Handler(s) for the Inlet
  setHandler(in, new InHandler {
    // what to do when a new element is ready to be consumed
    override def onPush(): Unit = {
      val elem = grab(in)

      // "Business" logic
      maxEndingHere = Math.max(0, maxEndingHere + elem)
      maxSoFar = Math.max(maxSoFar, maxEndingHere)

      // this should never happen
      // we decide to not push the value, avoiding the error
      // but potentially losing the value
      if (isAvailable(out))
        push(out, maxSoFar)
    }
```

Proper Kadane algo logic here

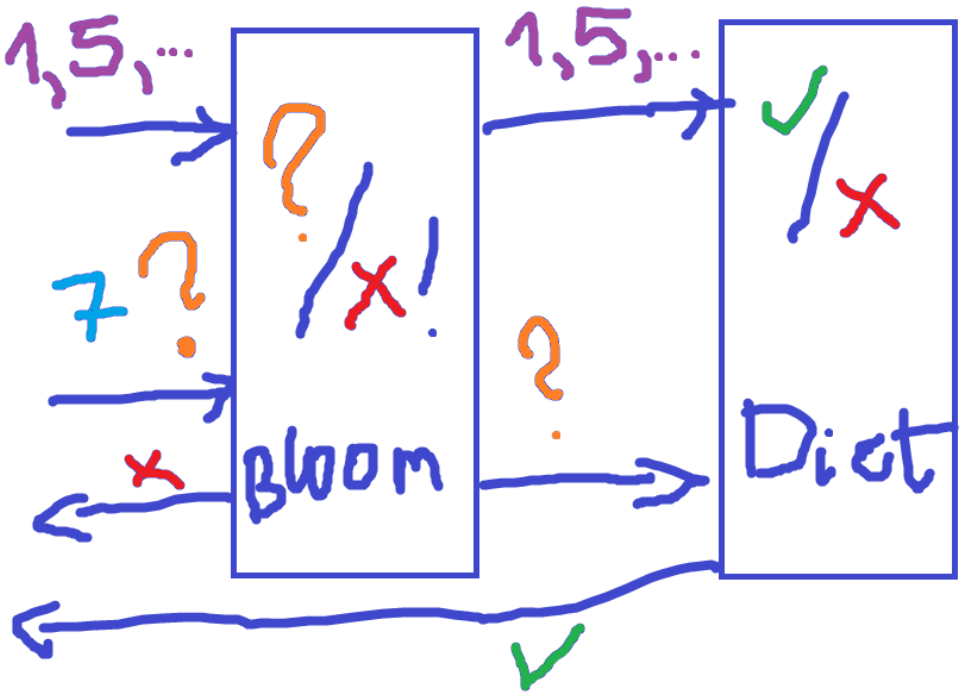
Pro Tip: replace plumbing with statefulMapConcat

Let the flow flow…

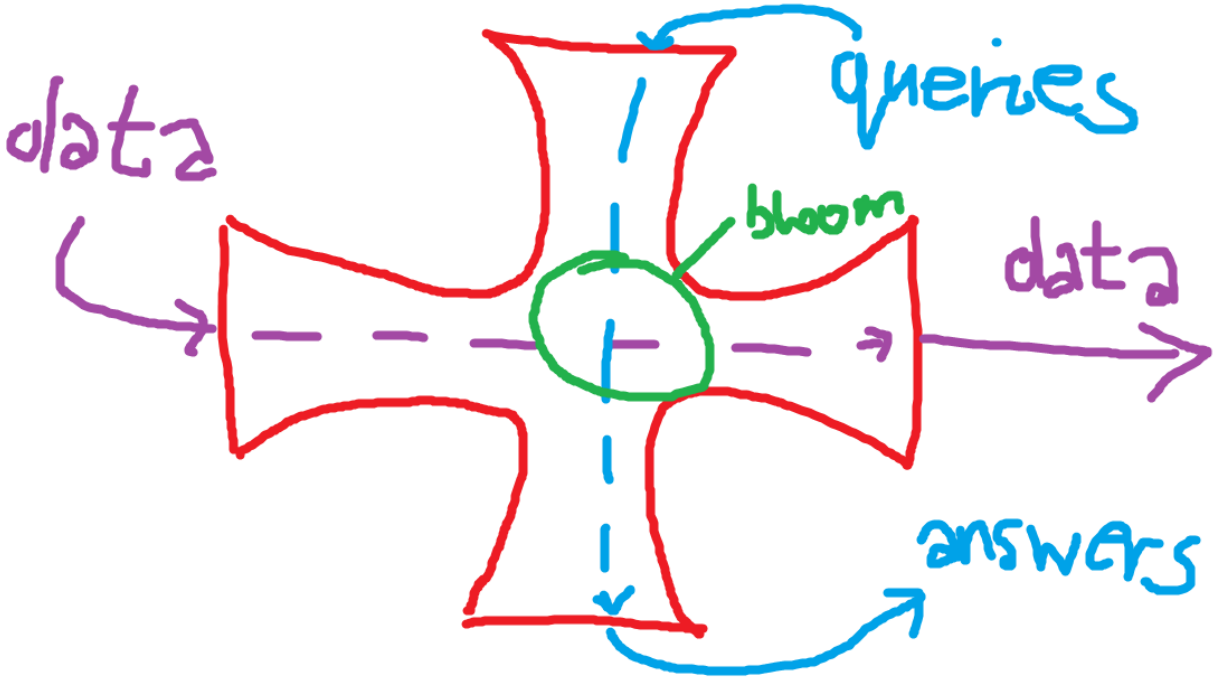
```
object KadaneFlowMain extends App {
  implicit val system = ActorSystem()
  implicit val mat = ActorMaterializer()

  val kadaneFlowStage = new KadaneFlowStage

  val done = Source.repeat(1).take(100).map(_ => Random.nextInt(1100) - 1000)
//    .throttle(1, Duration(100, "millisecond"), 1, ThrottleMode.shaping)
    .via(kadaneFlowStage)
    .throttle(1, Duration(100, "millisecond"), 1, ThrottleMode.shaping)
    .runWith(Sink.foreach(println))

  import scala.concurrent.ExecutionContext.Implicits.global

  done.onComplete(_ => system.terminate())
  Await.ready(system.whenTerminated, Duration.Inf)
}
```

Kadane Flow – demo

Bloom filter (on-line, streaming, 1970)



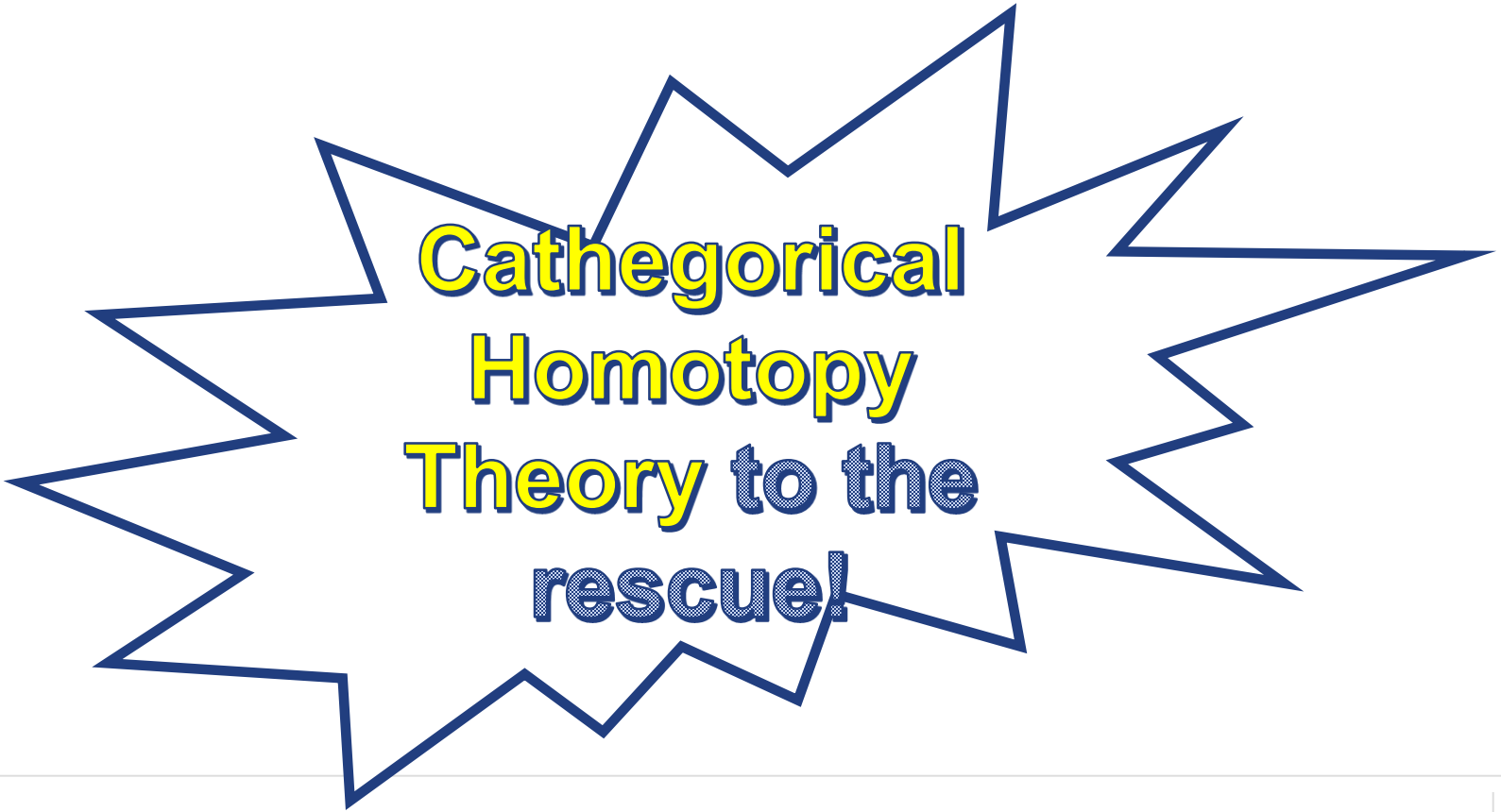
A good shape for bloom filter? CrossShape!



Shapes are just … shapes ☺

scala wave

GFT



Remember, a shape is just a shape

```
/**
 * A bidirectional flow of elements that consequently has two inputs and two
 * outputs, arranged like this:
 *
 * {{{
 *        +------+
 *  In1 ~>|      |~> Out1
 *        | bidi |
 * Out2 <~|      |<~ In2
 *        +------+
 * }}}
 */
final case class BidiShape[-In1, +Out1, -In2, +Out2](
  in1:  Inlet[In1 @uncheckedVariance],
  out1: Outlet[Out1 @uncheckedVariance],
  in2:  Inlet[In2 @uncheckedVariance],
  out2: Outlet[Out2 @uncheckedVariance]) extends Shape {
```

=

```
          In2
           |
           v
      +---------+
In1 ~> |  cross  | ~> Out1
      +---------+
           |
           v
          Out2
```

Homotopic!

BloomFilterCrossStage, ftw!

scala wave

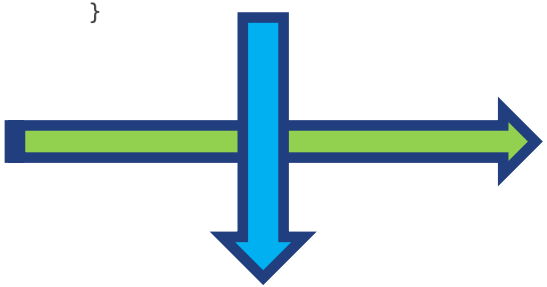
GFT

```
// Cross Shape is actually BidiShape - for shape the semantics doesn't count, only syntax
class BloomFilterCrossStage extends GraphStage[BidiShape[Int, Int, Int, String]] {

  // Stage syntax
  val dataIn: Inlet[Int] = Inlet("BloomFilterCrossStage.dataIn")
  val dataOut: Outlet[Int] = Outlet("BloomFilterCrossStage.dataOut")
  val queriesIn: Inlet[Int] = Inlet("BloomFilterCrossStage.queriesIn")
  val answersOut: Outlet[String] = Outlet("BloomFilterCrossStage.answersOut")
  override val shape: BidiShape[Int, Int, Int, String] = BidiShape(dataIn, dataOut, queriesIn, answersOut)
```

Two crossing flows…

```
setHandler(queriesIn, new InHandler {
  override def onPush(): Unit = {
    val x = grab(queriesIn)
    val answer = if (filter.mightContain(x)) {
      s"MAYBE, filter probably contains $x"
    } else {
      s"NO, filter definitely does not contain $x"
    }
    if (isAvailable(answersOut))
      push(answersOut, answer)
  }
```

```
setHandler(dataIn, new InHandler {
  override def onPush(): Unit = {
    val elem = grab(dataIn)
    filter.put(elem)
    if (isAvailable(dataOut))
      push(dataOut, elem)
  }
```



```
setHandler(dataOut, new OutHandler {
  override def onPull(): Unit = {
    if (!hasBeenPulled(dataIn))
      pull(dataIn)
  }
})
```

```
setHandler(answersOut, new OutHandler {
  override def onPull(): Unit = {
    if (!hasBeenPulled(queriesIn))
      pull(queriesIn)
  }
```

Akka allows you to put your **on-line / streaming data structure / algorithm** in context. You don't need to think about how to take care of data flow and backpressure.

Well thought out architecture of **Akka** lets you **concentrate on the stuff *relevant to your problem domain*.**

Akka is **geared towards high performance**, and **can be used in IoT setup**, where e.g. Spark streaming would not fit.

It is **easy to port** e.g. machine learning **algorithms from other streaming setups** (like Spark streaming) **to Akka**.

ProTip #1 – you can use actor to better manage state for on-line algos

scala wave

GFT

```
val actorQueue = Source.queue[Int](10, OverflowStrategy.backpressure)
val actorSink = actorQueue
  .throttle(1, Duration(100, "millisecond"), 1, ThrottleMode.shaping)
  .to(Sink.foreach((x) => println(x))).run()

val done = actorSink.watchCompletion()

val kadaneFlowActor = system.actorOf(Props(new KadaneFlowActor(actorSink)))

Source.repeat(1).take(100).map(_ => Random.nextInt(1100) - 1000)
  .runWith(Sink.actorRefWithAck(kadaneFlowActor, "TEST", "ACK", "END", _ => "FAIL"))
```

ProTip #1 – you can use actor to better manage state for on-line algos

scala wave

GFT

```
    case elem : Int     => // onPush() + grab(in)
      // "Business" logic
      max_ending_here = Math.max(0, max_ending_here + elem)
      max_so_far = Math.max(max_so_far, max_ending_here)

//      Thread.sleep(100) // FIXME: don't do this at home

      log.info(s"Received: $elem, sending out: $max_so_far")

      val offF = queue.offer(max_so_far)  // push element downstream
      offF pipeTo self                    // generate backpressure in the Actor

    case e : QueueOfferResult =>
      upStream ! "ACK" // ask for next element
```

ProTip #2 – it is (too) easy to create new shapes

scala wave

GFT

```
/**
  * A Y-shaped flow of elements that consequently has two inputs
  * and one output, arranged like this:
  *
  * {{{
  *        +--------+
  *  In1 ~>|        |
  *        | tripod |~> Out
  *  In2 ~>|        |
  *        +--------+
  * }}}
  */
```

Tripod? Just like the in old days ☺

scala wave

GFT

```
final case class TripodShape[-In1, -In2, +Out](
  in1:  Inlet[In1 @uncheckedVariance],
  in2:  Inlet[In2 @uncheckedVariance],
  out: Outlet[Out @uncheckedVariance]) extends Shape {
    override val inlets: immutable.Seq[Inlet[_]] = List(in1, in2)
    override val outlets: immutable.Seq[Outlet[_]] = List(out)

    override def deepCopy(): TripodShape[In1, In2, Out] =
      TripodShape(in1.carbonCopy(), in2.carbonCopy(), out.carbonCopy())
    override def copyFromPorts(inlets: immutable.Seq[Inlet[_]], outlets: immutable.Seq[Outlet[_]]): Shape = {
      require(inlets.size == 2, s"proposed inlets [${inlets.mkString(", ")}] do not fit TripodShape")
      require(outlets.size == 1, s"proposed outlets [${outlets.mkString(", ")}] do not fit TripodShape")
      TripodShape(inlets(0), inlets(1), outlets(0))
    }
  def reversed: Shape = copyFromPorts(inlets.reverse, outlets.reverse)
}
```

Remember your Topology class? A shape is just a shape…

scala wave

GFT

ProTip #3 – you can return materialized value from GraphDSL built shape

scala wave

GFT

```
val crossStage = new BloomFilterCrossStage

val graph = RunnableGraph.fromGraph(GraphDSL.create(Sink.foreach(println)) { implicit builder => outControl =>
  import GraphDSL.Implicits._
  val inData = Source.repeat(1).take(100).map(_ => Random.nextInt(1000)).throttle(1, Duration(100, "millisecond
  val outData = Sink.foreach(println)
  val inControl = Source.repeat(1).take(10).map(_ => Random.nextInt(2000) - 1000).throttle(1, Duration(1500, "m
  //val outControl = Sink.foreach(println) // Moved to foreach/builder construct

  val cross = builder.add(crossStage)

  inData ~> cross.in1; cross.out1 ~> outData
  inControl ~> cross.in2; cross.out2 ~> outControl
  ClosedShape
}).run()
```


Demo //FIXME

That's it…

https://en.wikipedia.org/wiki/Banner_Mania