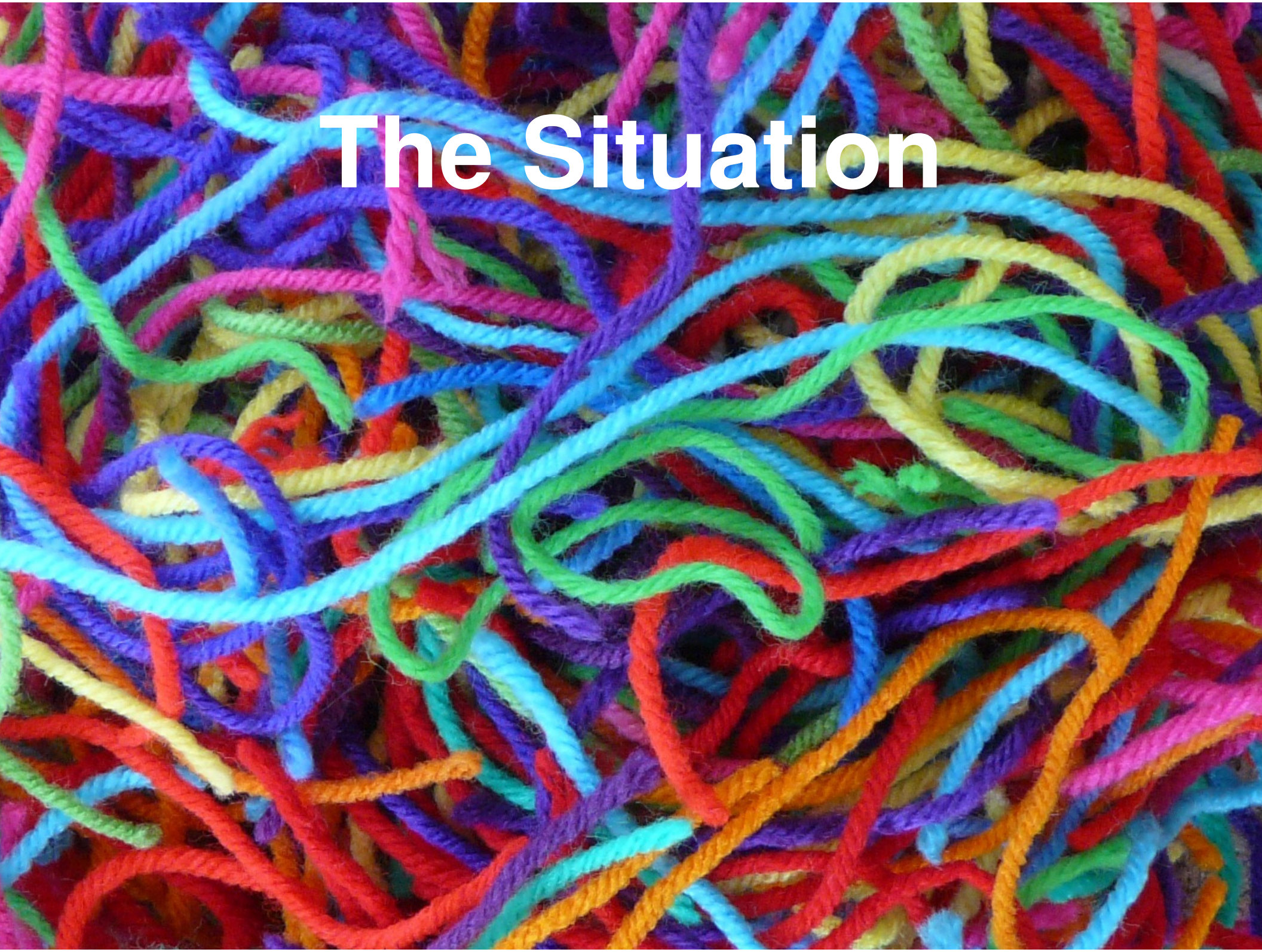
Akka Streams and Bloom Filters

Or: How I learned to stop worrying and love resilient elastic distributed real-time transaction processing using space efficient probabilistic data structures…


The Situation

The Situation

- Existing analytics data ingestion system based on batch ETL
- Cascading/Map Reduce
- Ingests files from S3, problematic to support from upstream root data ingestion system.
- Costly dedicated EC2 hosts
- Expensive Cassandra and file joins

The Desired Outcome

- Monetize the data faster
- Eliminate problematic S3 dependency
- Minimize Cassandra queries and eliminate file joins
- Efficient use of memory and compute resources
- Auto-scaling and error recovery

Low Hanging Fruit

- Streaming instead of Batch
- Kafka instead of ingesting from S3 files
- Docker and Mesos/Marathon instead of dedicated EC2 Instances

Open Questions

- What to do about Cassandra and file joins?
- Deduplication of the stream?
- Spark Streaming?

Cassandra and Files

- The legacy system joins files or Cassandra data to determine household presence in an audience
- This is….slooooow and is redone every time
- However this can be precomputed, added to a Bloom Filter and reused

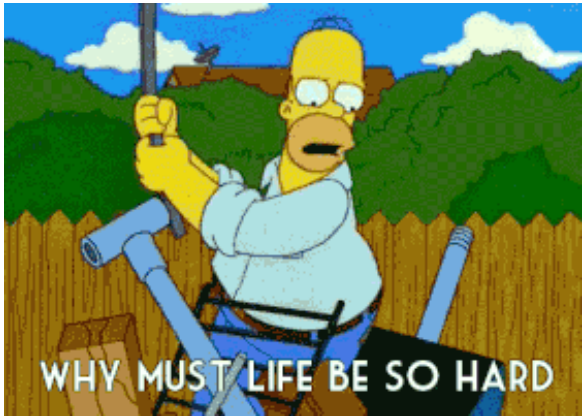
When 99% is accurate enough

- Bloom filters are used to determine the presence of an element in a set
- Bloom filters sacrifice 100% certainty for space efficiency
- False positives are possible. False negatives are impossible.
- Only ~9.6 bits per element for ~99% certainty. ~27 bits for ~99.99%
- That's a set of 3,000,000 elements in ~3.6Mb or 30X compression factor if elements are standard guids, for 99% certainty
- Checking presence in a bloom filter is fast!
- Bloom filters can be merged together (perfect for Map Reduce)

Twitter Algebird Bloom Filters

```
import com.twitter.algebird._

val NUM_HASHES = 6
val WIDTH = 64
val SEED = 1

val bfMonoid = new BloomFilterMonoid(NUM_HASHES, WIDTH, SEED)
val bf = bfMonoid.create("1", "2", "3", "4", “100") // 49 bits
bf.contains("1")

//Results: ApproximateBoolean(true,0.995724300187435)

val WIDTH = 128

val bfMonoid = new BloomFilterMonoid(NUM_HASHES, WIDTH, SEED)
val bf = bfMonoid.create("1", "2", "3", "4", “100") // 113 bits
bf.contains(“1")

//Results: ApproximateBoolean(true,0.999862235670191)

bf.contains(“70")

//Results: ApproximateBoolean(false,1.0)
```

Deduplication of a stream is hard



- The space efficient property also promised to be useful for fast in-memory deduplication
- However normal bloom filters are bounded in size while the stream is not
- No mature Stable Bloom Filter or distributed Bloom Filter implementations
- Problem synchronizing across distributed nodes
- Kafka/node failures downstream can reintroduce duplicates anyway
- So we pushed deduplication downstream to right before it’s needed

Spark Streaming?

- Mature product
- A lot of industry and team experience
- Mature supporting libraries
- Some very promising experimental features (e.g. structured streaming)
- However, handling complex state is not a well supported use case
- Juggling scores of bloom filters without causing OOM exceptions and updating them when they change is really complex.
- Well, complex for Spark but not for Akka Streams!

Akka Streams

- Built on Akka
- “…move data across an asynchronous boundary, without loss, buffering or resource exhaustion.”
- Actor Model
- Reactive
- Distributed by default
- Source, Sink, Flow abstractions
- Graph DSL
- Backpressure

State Management In Flow

```
object FindPrimes {
  case class MaybePrime(num: Int, isPrime: Boolean)

  def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem("ExampleSystem")
    implicit val materializer = ActorMaterializer()

    val findPrimes = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder => {
      import GraphDSL.Implicits._
      val source = builder.add(Source(1 to 100))
      val flow = builder.add(DetectPrimesFlow())
      val sink = builder.add(Sink.foreach[MaybePrime](maybePrime => if (maybePrime.isPrime) println(maybePrime.num)))

      source ~> flow ~> sink

      ClosedShape
    } })
    findPrimes.run()
  }

  def calculateIfHasPrimeFactor(num: Int, primes: mutable.MutableList[Int]): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })}

  def DetectPrimesFlow()(implicit materializer: ActorMaterializer): Flow[Int, MaybePrime, NotUsed] = {
    Flow[Int].statefulMapConcat[MaybePrime](() => {
      var primes = mutable.MutableList[Int]()
      (num: Int) => {
        if (num == 1) List(MaybePrime(num, isPrime = false))
        else {
          val hasPrimeFactor = calculateIfHasPrimeFactor(num, primes)
          if (!hasPrimeFactor) primes += num
          List(MaybePrime(num, !hasPrimeFactor))
        }
      }})}
}
```

https://gist.github.com/nyokodo/3f20dca5644b02f852cfd00515420d29

State Management In Flow

```
object FindPrimes {
  case class MaybePrime(num: Int, isPrime: Boolean)

  def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem("ExampleSystem")
    implicit val materializer = ActorMaterializer()

    val findPrimes = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder => {
      import GraphDSL.Implicits._
      val source = builder.add(Source(1 to 100))
      val flow = builder.add(DetectPrimesFlow())
      val sink = builder.add(Sink.foreach[MaybePrime](maybePrime => if (maybePrime.isPrime) println(maybePrime.num)))

      source ~> flow ~> sink

      ClosedShape
    } })
    findPrimes.run()
  }

  def calculateIfHasPrimeFactor(num: Int, primes: mutable.MutableList[Int]): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })}

  def DetectPrimesFlow()(implicit materializer: ActorMaterializer): Flow[Int, MaybePrime, NotUsed] = {
    Flow[Int].statefulMapConcat[MaybePrime](() => {
      var primes = mutable.MutableList[Int]()
      (num: Int) => {
        if (num == 1) List(MaybePrime(num, isPrime = false))
        else {
          val hasPrimeFactor = calculateIfHasPrimeFactor(num, primes)
          if (!hasPrimeFactor) primes += num
          List(MaybePrime(num, !hasPrimeFactor))
        }
      }})}
}
```

https://gist.github.com/nyokodo/3f20dca5644b02f852cfd00515420d29

State Management In Flow

```
object FindPrimes {
  case class MaybePrime(num: Int, isPrime: Boolean)

  def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem("ExampleSystem")
    implicit val materializer = ActorMaterializer()

    val findPrimes = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder => {
      import GraphDSL.Implicits._
      val source = builder.add(Source(1 to 100))
      val flow = builder.add(DetectPrimesFlow())
      val sink = builder.add(Sink.foreach[MaybePrime](maybePrime => if (maybePrime.isPrime) println(maybePrime.num)))

      source ~> flow ~> sink

      ClosedShape
    } })
    findPrimes.run()
  }

  def calculateIfHasPrimeFactor(num: Int, primes: mutable.MutableList[Int]): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })}

  def DetectPrimesFlow()(implicit materializer: ActorMaterializer): Flow[Int, MaybePrime, NotUsed] = {
    Flow[Int].statefulMapConcat[MaybePrime](() => {
      var primes = mutable.MutableList[Int]()
      (num: Int) => {
        if (num == 1) List(MaybePrime(num, isPrime = false))
        else {
          val hasPrimeFactor = calculateIfHasPrimeFactor(num, primes)
          if (!hasPrimeFactor) primes += num
          List(MaybePrime(num, !hasPrimeFactor))
        }
      }})}
}
```

https://gist.github.com/nyokodo/3f20dca5644b02f852cfd00515420d29

State Management In Flow

```
object FindPrimes {
  case class MaybePrime(num: Int, isPrime: Boolean)

  def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem("ExampleSystem")
    implicit val materializer = ActorMaterializer()

    val findPrimes = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder => {
      import GraphDSL.Implicits._
      val source = builder.add(Source(1 to 100))
      val flow = builder.add(DetectPrimesFlow())
      val sink = builder.add(Sink.foreach[MaybePrime](maybePrime => if (maybePrime.isPrime) println(maybePrime.num)))

      source ~> flow ~> sink

      ClosedShape
    } })
    findPrimes.run()
  }

  def calculateIfHasPrimeFactor(num: Int, primes: mutable.MutableList[Int]): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })}

  def DetectPrimesFlow()(implicit materializer: ActorMaterializer): Flow[Int, MaybePrime, NotUsed] = {
    Flow[Int].statefulMapConcat[MaybePrime](() => {
      var primes = mutable.MutableList[Int]()
      (num: Int) => {
        if (num == 1) List(MaybePrime(num, isPrime = false))
        else {
          val hasPrimeFactor = calculateIfHasPrimeFactor(num, primes)
          if (!hasPrimeFactor) primes += num
          List(MaybePrime(num, !hasPrimeFactor))
        }
      }})}
}
```

https://gist.github.com/nyokodo/3f20dca5644b02f852cfd00515420d29

Encapsulating State in Actor

```
class IsPrimeNumber extends Actor {
  var primes = mutable.MutableList[Int]()

  def calculateIfHasPrimeFactor(num: Int): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })
  }

  def isPrimeNumber(num: Int) = {
    if (num == 1) false
    else {
      val hasPrimeFactor = calculateIfHasPrimeFactor(num)
      if (!hasPrimeFactor) primes += num
      !hasPrimeFactor
    }
  }

  def receive = {
    case IsPrimeNumberRequest(num) => sender ! IsPrimeNumberResponse(isPrimeNumber(num))
    case _ => sender ! None
  }
}
```

https://gist.github.com/nyokodo/3b1a15beb3d71932bbbf0673d2bc6b67

Encapsulating State in Actor

```
class IsPrimeNumber extends Actor {
  var primes = mutable.MutableList[Int]()

  def calculateIfHasPrimeFactor(num: Int): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })
  }

  def isPrimeNumber(num: Int) = {
    if (num == 1) false
    else {
      val hasPrimeFactor = calculateIfHasPrimeFactor(num)
      if (!hasPrimeFactor) primes += num
      !hasPrimeFactor
    }
  }

  def receive = {
    case IsPrimeNumberRequest(num) => sender ! IsPrimeNumberResponse(isPrimeNumber(num))
    case _ => sender ! None
  }
}
```

https://gist.github.com/nyokodo/3b1a15beb3d71932bbbf0673d2bc6b67

Encapsulating State in Actor

```
class IsPrimeNumber extends Actor {
  var primes = mutable.MutableList[Int]()

  def calculateIfHasPrimeFactor(num: Int): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })
  }

  def isPrimeNumber(num: Int) = {
    if (num == 1) false
    else {
      val hasPrimeFactor = calculateIfHasPrimeFactor(num)
      if (!hasPrimeFactor) primes += num
      !hasPrimeFactor
    }
  }

  def receive = {
    case IsPrimeNumberRequest(num) => sender ! IsPrimeNumberResponse(isPrimeNumber(num))
    case _ => sender ! None
  }
}
```

https://gist.github.com/nyokodo/3b1a15beb3d71932bbbf0673d2bc6b67

Encapsulating State in Actor

```
class IsPrimeNumber extends Actor {
  var primes = mutable.MutableList[Int]()

  def calculateIfHasPrimeFactor(num: Int): Boolean = {
    primes.filter(prime => prime.toDouble <= Math.sqrt(num.toDouble))
      .foldLeft(false)((primeDivisible, prime) => {
        if (primeDivisible) primeDivisible else num % prime == 0
      })
  }

  def isPrimeNumber(num: Int) = {
    if (num == 1) false
    else {
      val hasPrimeFactor = calculateIfHasPrimeFactor(num)
      if (!hasPrimeFactor) primes += num
      !hasPrimeFactor
    }
  }

  def receive = {
    case IsPrimeNumberRequest(num) => sender ! IsPrimeNumberResponse(isPrimeNumber(num))
    case _ => sender ! None
  }
}
```

https://gist.github.com/nyokodo/3b1a15beb3d71932bbbf0673d2bc6b67

Flow calling Actor

```
object FindPrimes {

  case class MaybePrime(num: Int, isPrime: Boolean)

  def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem("ExampleSystem")
    implicit val materializer = ActorMaterializer()

    val isPrimeNumber = system.actorOf(Props[IsPrimeNumber], "IsPrimeNumber")

    val findPrimes = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder => {
      import GraphDSL.Implicits._
      val source = builder.add(Source(1 to 100))
      val flow = builder.add(DetectPrimesFlow(isPrimeNumber))
      val sink = builder.add(Sink.foreach[MaybePrime](maybePrime => if (maybePrime.isPrime) println(maybePrime.num)))

      source ~> flow ~> sink

      ClosedShape
    } })
    findPrimes.run()
  }

  def DetectPrimesFlow(isPrimeNumber: ActorRef)(implicit materializer: ActorMaterializer): Flow[Int, MaybePrime, NotUsed] = {
    Flow[Int].mapAsync[MaybePrime](5) { num =>
      import scala.concurrent.ExecutionContext.Implicits.global
      implicit val timeout = Timeout(5 seconds)

      (isPrimeNumber ? IsPrimeNumberRequest(num)).map {
        case IsPrimeNumberResponse(isPrime) => MaybePrime(num, isPrime)
      }
    }
  }
}
```

https://gist.github.com/nyokodo/3b1a15beb3d71932bbbf0673d2bc6b67

Flow calling Actor

```
object FindPrimes {

  case class MaybePrime(num: Int, isPrime: Boolean)

  def main(args: Array[String]): Unit = {
    implicit val system = ActorSystem("ExampleSystem")
    implicit val materializer = ActorMaterializer()

    val isPrimeNumber = system.actorOf(Props[IsPrimeNumber], "IsPrimeNumber")

    val findPrimes = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder => {
      import GraphDSL.Implicits._
      val source = builder.add(Source(1 to 100))
      val flow = builder.add(DetectPrimesFlow(isPrimeNumber))
      val sink = builder.add(Sink.foreach[MaybePrime](maybePrime => if (maybePrime.isPrime) println(maybePrime.num)))

      source ~> flow ~> sink

      ClosedShape
    } })
    findPrimes.run()
  }

  def DetectPrimesFlow(isPrimeNumber: ActorRef)(implicit materializer: ActorMaterializer): Flow[Int, MaybePrime, NotUsed] = {
    Flow[Int].mapAsync[MaybePrime](5) { num =>
      import scala.concurrent.ExecutionContext.Implicits.global
      implicit val timeout = Timeout(5 seconds)

      (isPrimeNumber ? IsPrimeNumberRequest(num)).map {
        case IsPrimeNumberResponse(isPrime) => MaybePrime(num, isPrime)
      }
    }
  }
}
```

https://gist.github.com/nyokodo/3b1a15beb3d71932bbbf0673d2bc6b67

Our Solution

- Encapsulate Bloom Filter cache in Actor
- Cache loads serialized Bloom Filters from S3
- Cache is limited by size and elements have timeout
- Memory safe and Bloom Filters updated when needed
- Actor called Asynchronously from Flow limiting performance impact of I/O

Results

- Akka Streams has proven extremely effective at encapsulating complex state
- Comparisons are difficult but early indications are that it’s very fast
- Still need to solve Deduplication problem
- Still need a robust solution to auto-scaling

Questions

References

- Bloom Filters: http://dmod.eu/deca/ft_gateway.cfm.pdf
- Stable Bloom Filters: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.1569&rep=rep1&type=pdf
- Twitter Algebird: https://github.com/twitter/algebird
- Reactive Manifesto: http://www.reactivemanifesto.org/
- Akka: http://akka.io/
- Akka Streams: http://doc.akka.io/docs/akka/current/scala.html