

# How We Built an Event-Time Merge of Two Kafka-Streams with Spark Streaming

Ralf Sigmund, Sebastian Schröder  
Otto GmbH & Co. KG



# Agenda

- Who are we?
- Our Setup
- The Problem
- Our Approach
- Requirements
- Advanced Requirements
- Lessons learned

# Who are we?

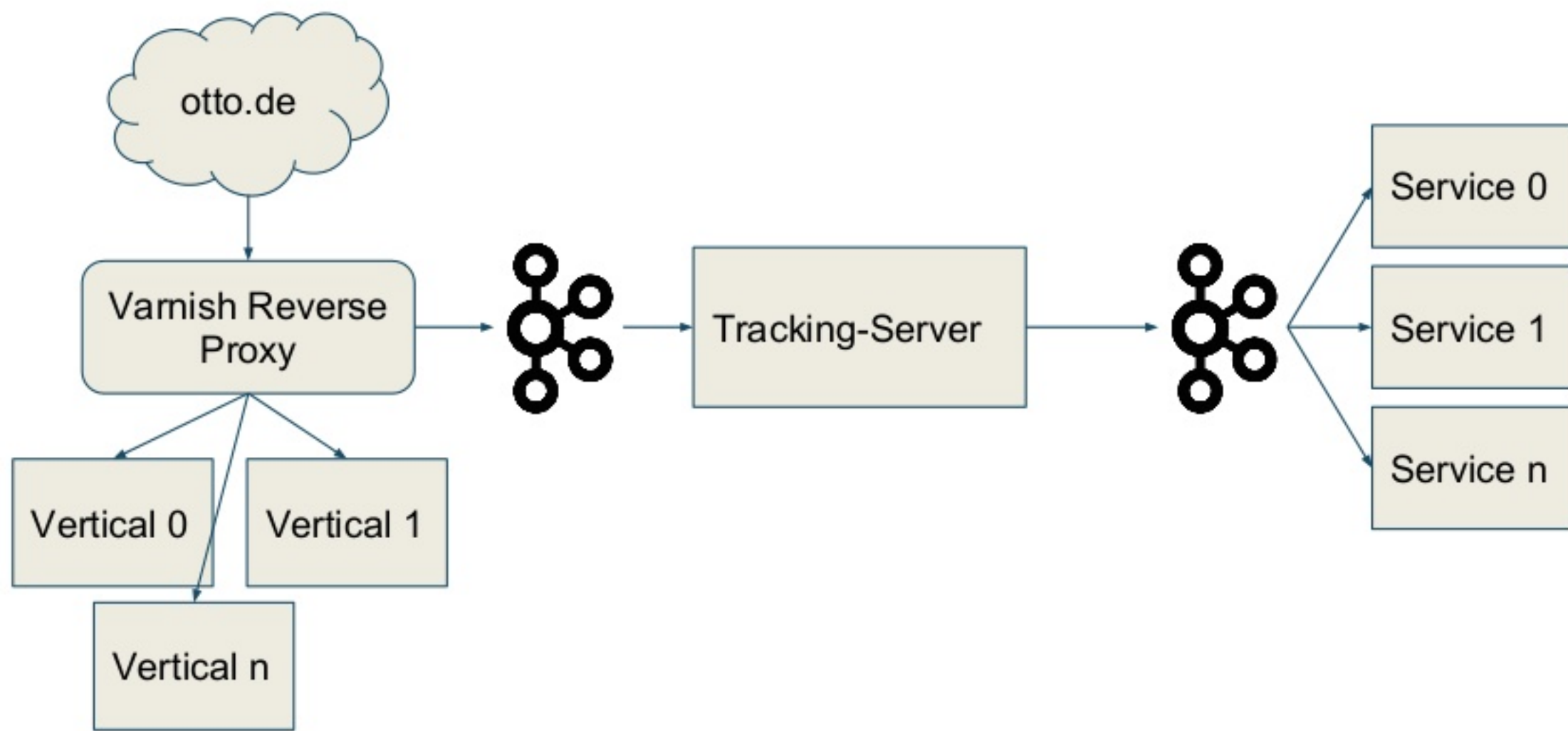
- Ralf
  - Technical Designer Team Tracking
  - Cycling, Surfing
  - Twitter: @sistar\_hh
- Sebastian
  - Developer Team Tracking
  - Taekwondo, Cycling
  - Twitter: @Sebasti0n

# Who are we working for?

# OTTO

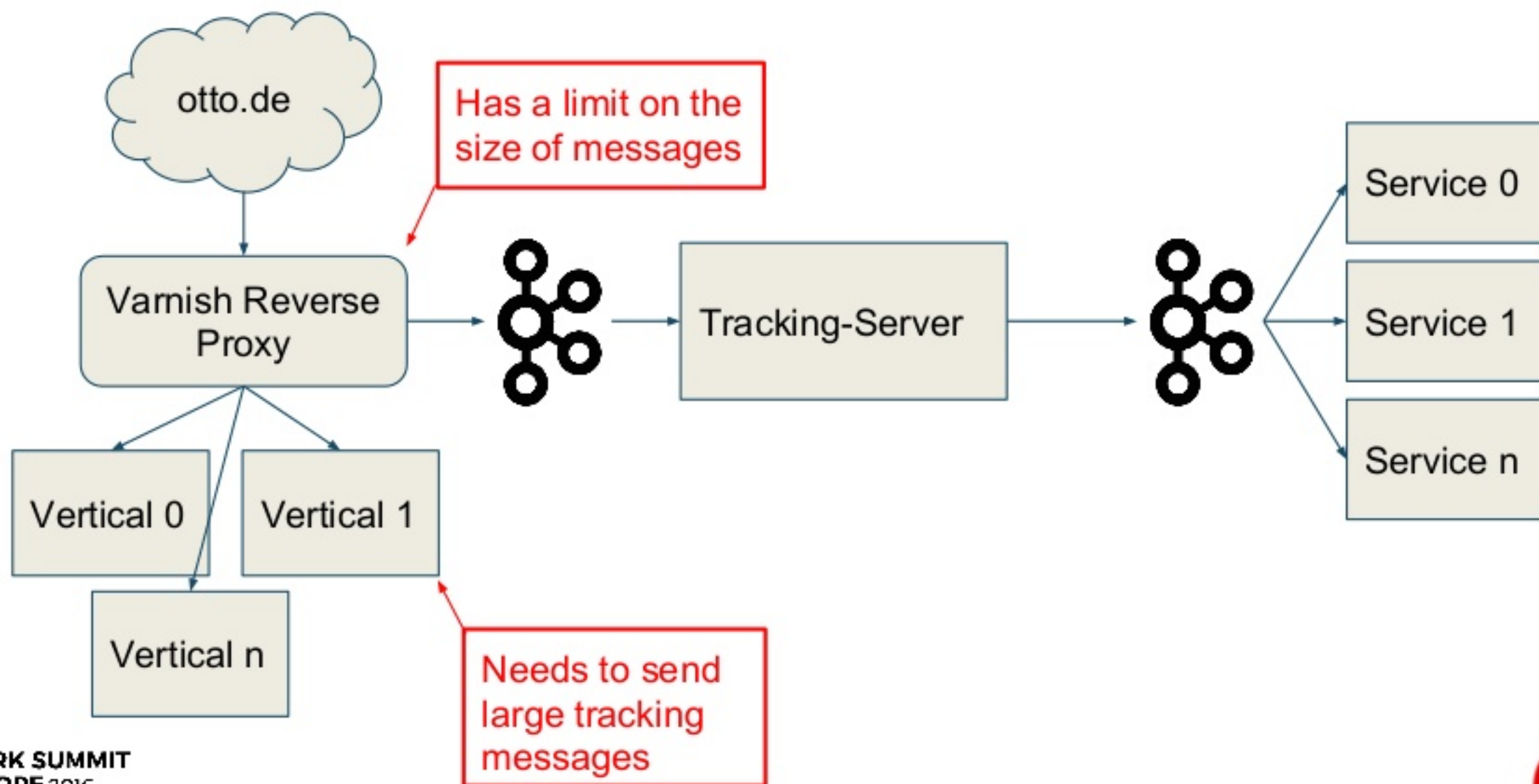
- 2nd largest ecommerce company in Germany
- more than 1 million visitors every day
- our blog: <https://dev.otto.de>
- our jobs: [www.otto.de/jobs](http://www.otto.de/jobs)

# Our Setup

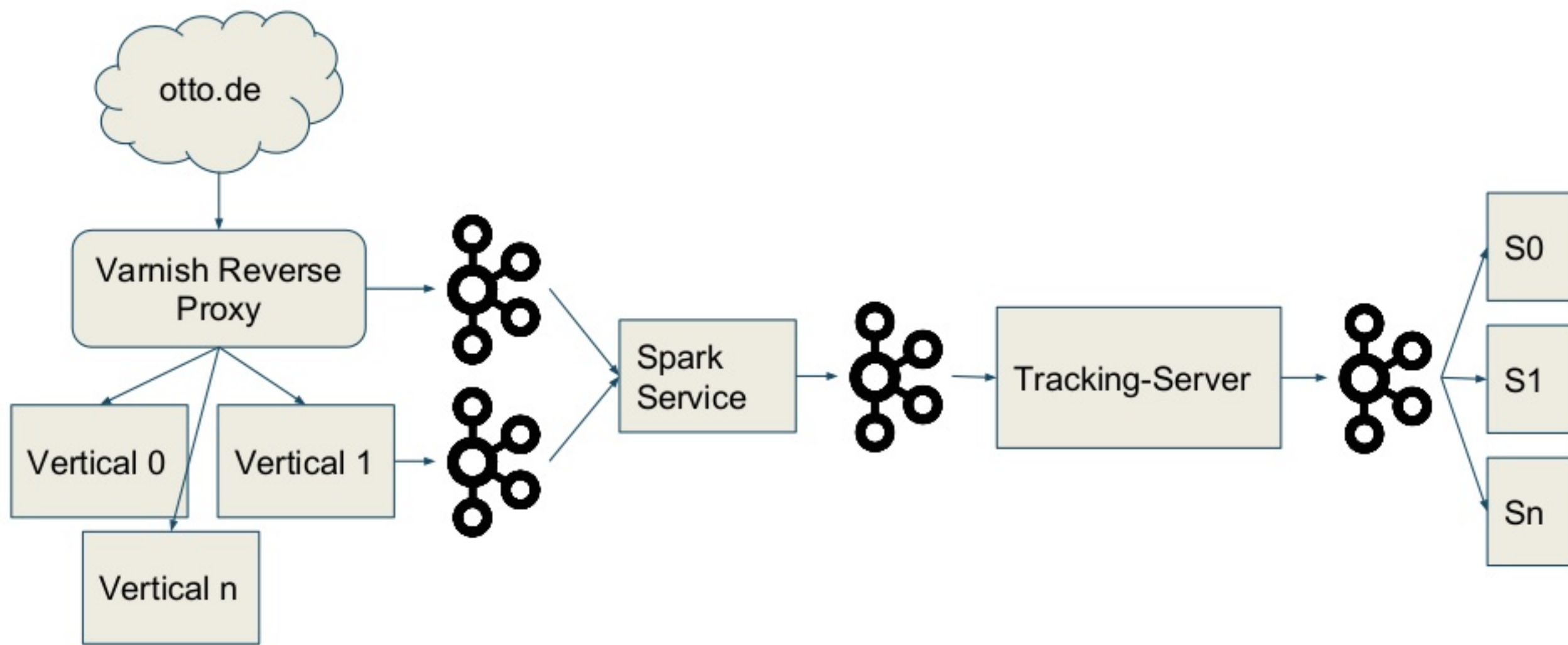




# The Problem



# Our Approach

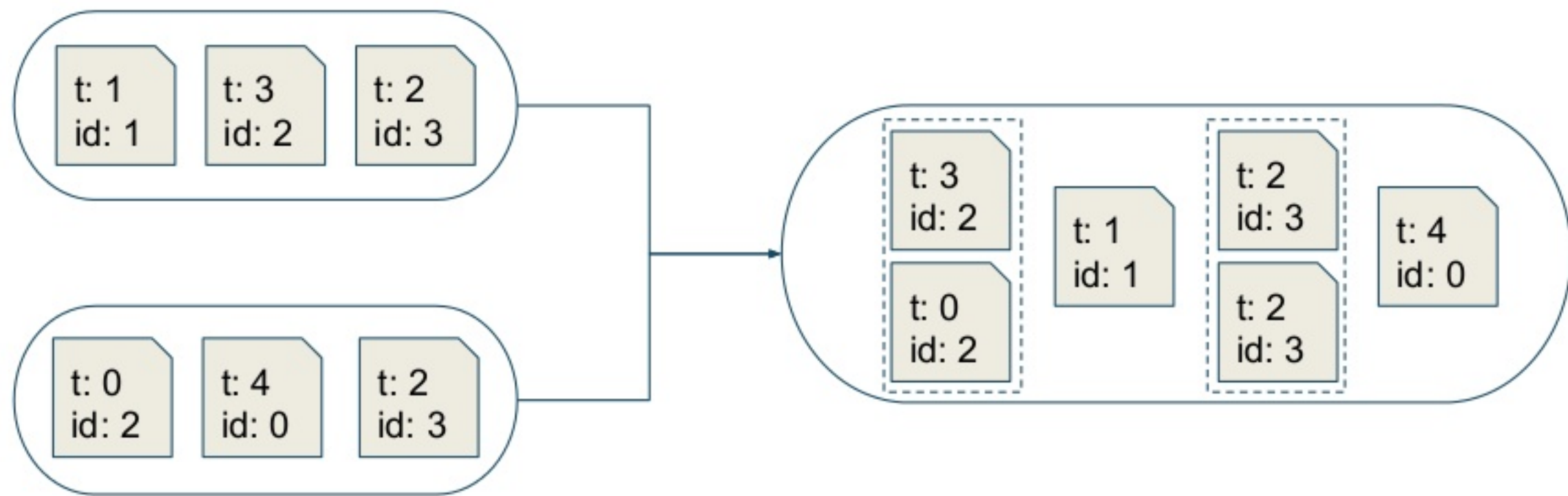


# Requirements

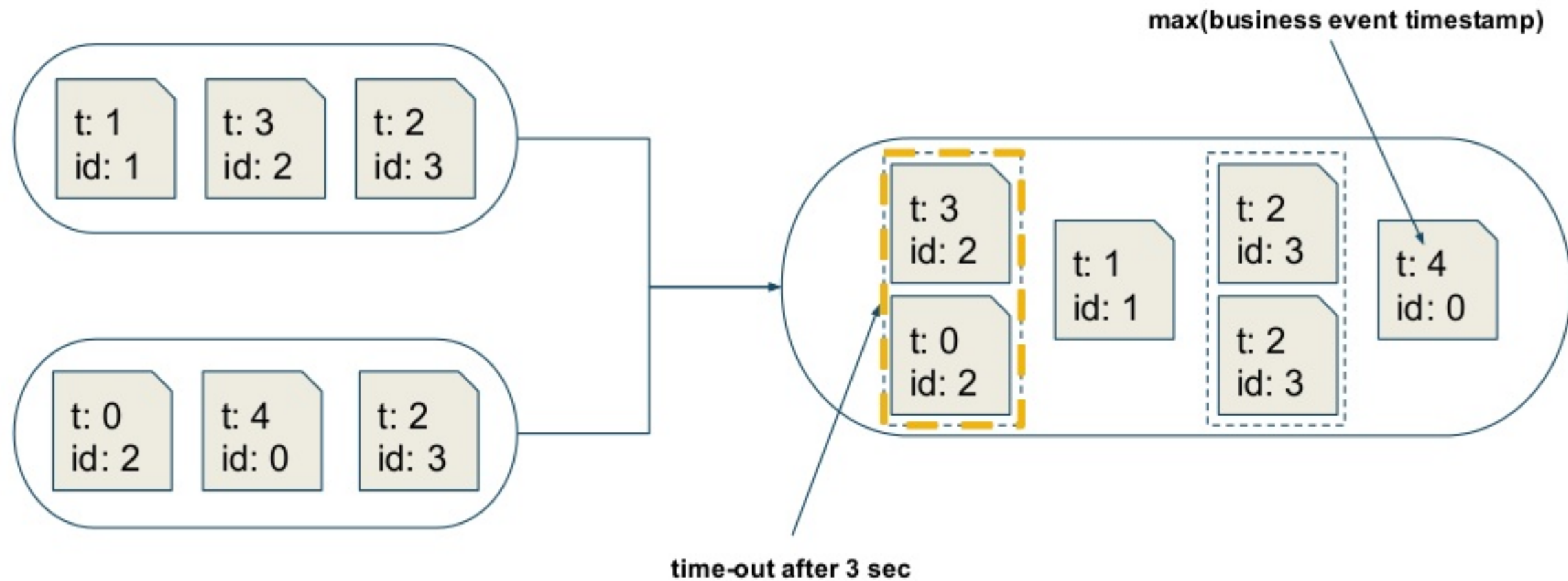




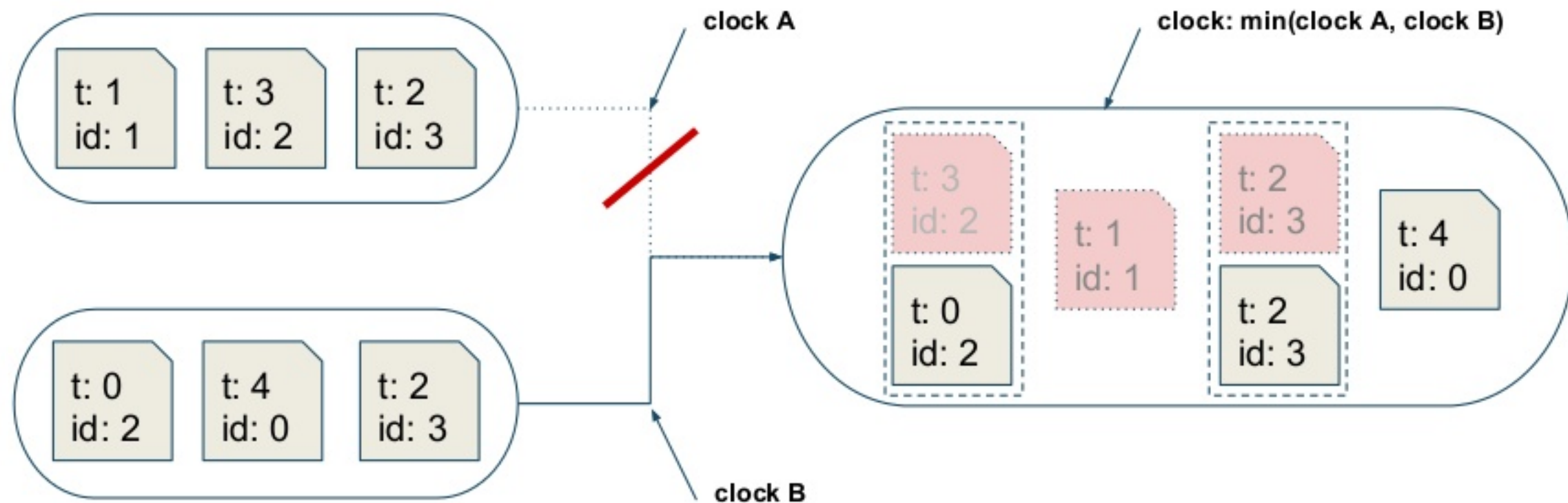
# Messages with the same key are merged



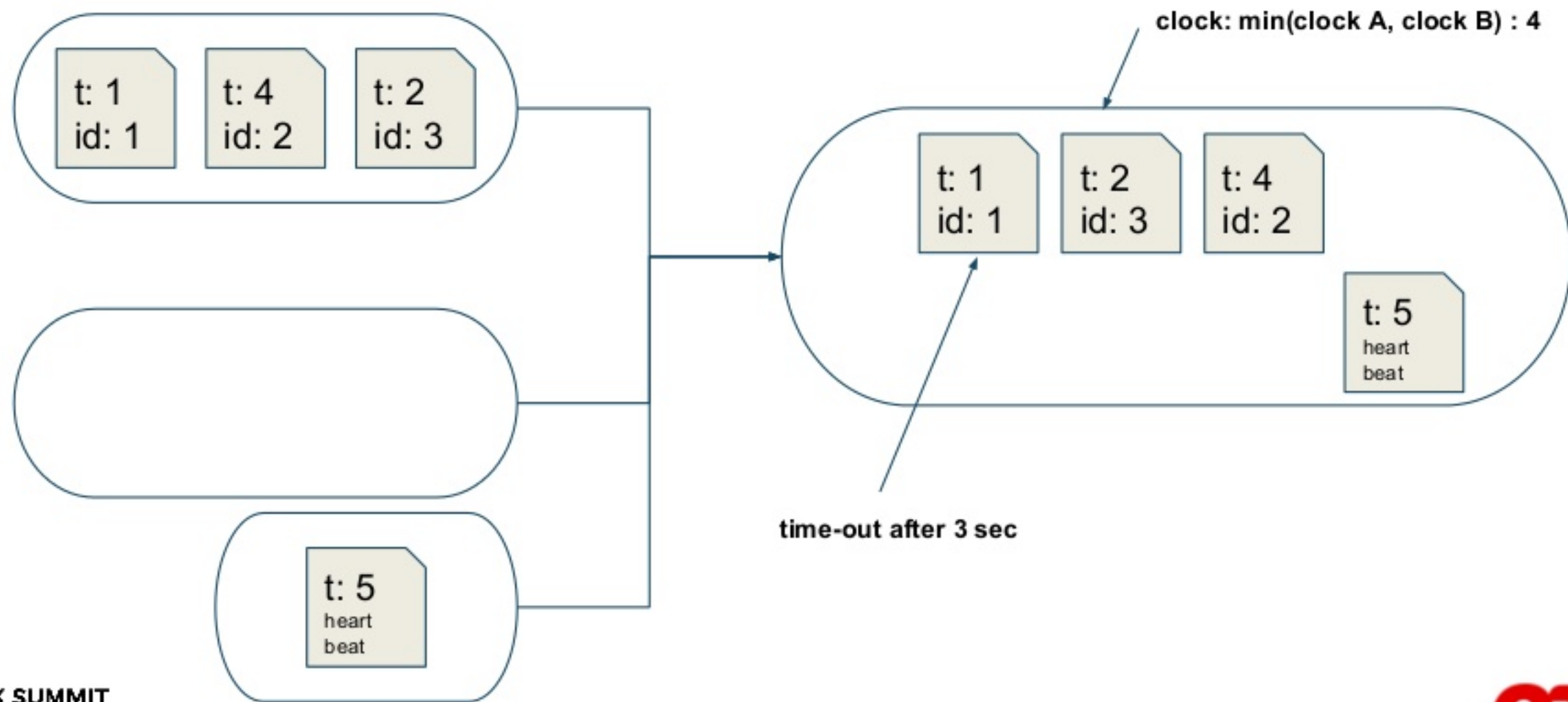
# Messages have to be timed out



# Event Streams might get stuck



# There might be no messages



# Requirements Summary

- Messages with the same key are merged
- Messages are timed out by the combined event time of both source topics
- Timed out messages are sorted by event time



# Solution



# Merge messages with the same Id

- UpdateStateByKey, which is defined on pairwise RDDs
- It applies a function to all new messages and existing state for a given key
- Returns a StateDStream with the resulting messages

# UpdateStateByKey

DStream  
[InputMessage]



HistoryRDD  
[MergedMessage]



DStream  
[MergedMessage]



**UpdateFunction:**

(Seq[InputMessage], Option[MergedMessage]) =>  
Option[MergedMessage]

# Merge messages with the same Id

```
val merge: (DStream[InputMessage]) => DStream[(String, MergedMessage)] =  
  (input) => {  
    input  
      .map(msg => (msg.ssid, msg))  
      .updateStateByKey(updateState(_, _))  
  }
```



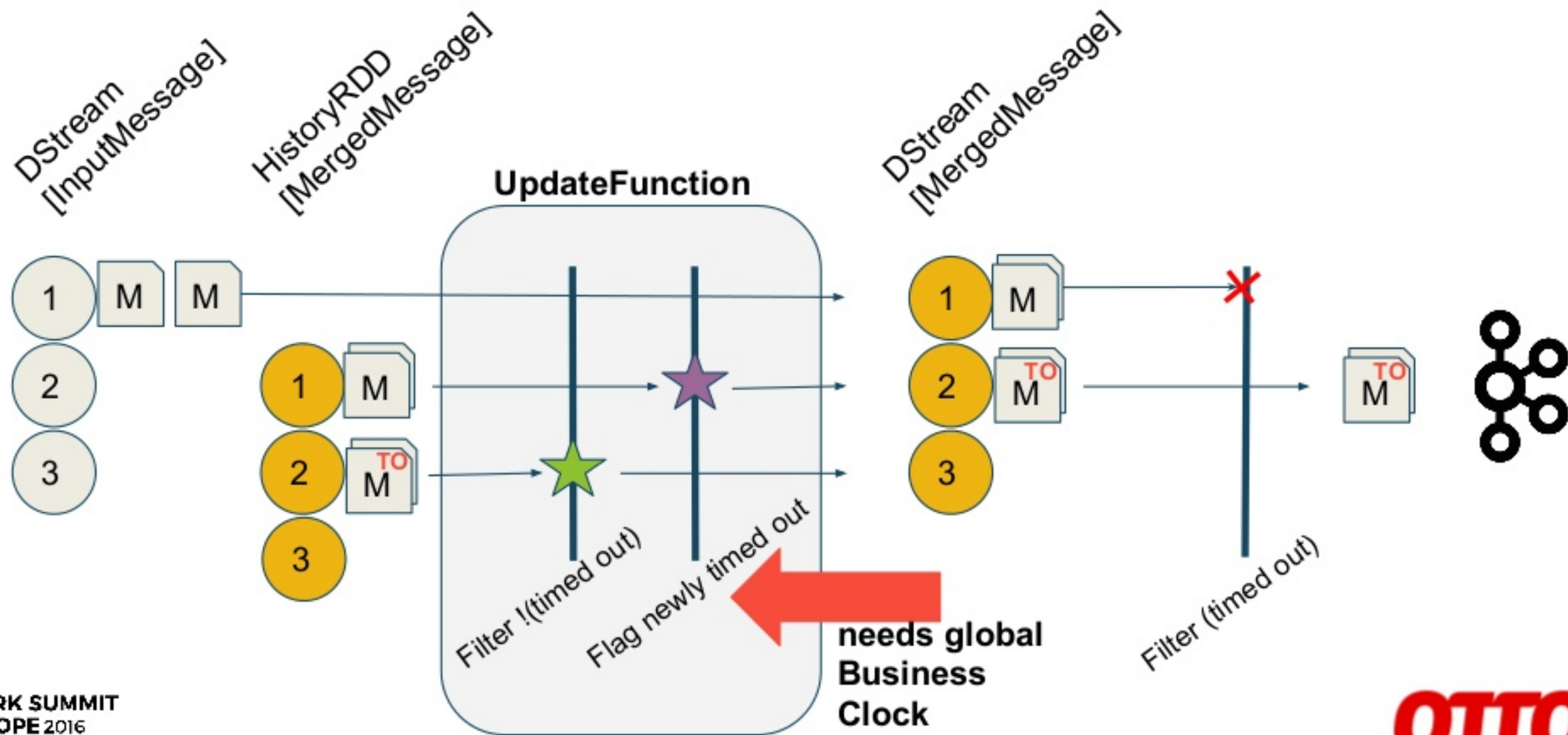
# Merge messages with the same Id

@tailrec

```
def updateState(inputMessages: Seq[InputMessage],  
               history: Option[MergedMessage]): Option[MergedMessage] = {  
  
  def combine(mm: MergedMessage, im: InputMessage): MergedMessage = {...}  
  
  (inputMessages, history) match {  
    case (seq, _) if seq.isEmpty => // no input messages  
      history  
    case (seq, None) => // no history (state), new input message(s)  
      updateState(seq.tail, Some(MergedMessage.fromInputMessage(seq.head)))  
    case (seq, Some(h)) => // existing history (state), new input messages  
      updateState(seq.tail, Some(combine(h, seq.head)))  
  }  
}
```



# Flag And Remove Timed Out Msgs



# Messages are timed out by event time

- We need access to all messages from the current micro batch and the history (state) => Custom StateDStream
- Compute the maximum event time of both topics and supply it to the updateStateByKey function

# Messages are timed out by event time

```
val merge: (DStream[InputMessage]) => DStream[(String, MergedMessage)] =  
  (input) => {  
    input  
      .map(msg => (msg.ssid, msg))  
      .updateStateByKeyMerge(createUpdateStateFunction(_, _))  
  }
```



# Messages are timed out by event time

```
def createUpdateStateFun(inputMsgs: Option[RDD[(String, InputMessage)]],  
                        historyMsgs: Option[RDD[(String, MergedMessage)]]):  
  (Seq[InputMessage], Option[MergedMessage]) => Option[MergedMessage] = {  
  
    val minBusinessClockPerPartition = calculateMbc(inputMsgs, historyMsgs)  
    val minOffsetPerPartition = calculateMinOffset(inputMsgs, historyMsgs)  
  
    updateState(minBusinessClockPerPartition, minOffsetPerPartition)  
  }
```

# Messages are timed out by event time

```
def updateState(minBusinessClocks: Map[Int, Long],
               minSourceTopicOffsets: Map[Int, MinSourceTopicOffsets])
               (inputMsgs: Seq[InputMessage], history: Option[MergedMessage]):
Option[MergedMessage] = {

  def flagIfTimedOut(mergedMessage: MergedMessage): MergedMessage = {...}

  inputMsgs
    .filterNot(isHeartbeatMessage)
    .mergeWith(history)
    .filterNot(_.meta.timedOut)
    .map(flagIfTimedOut)
}
```



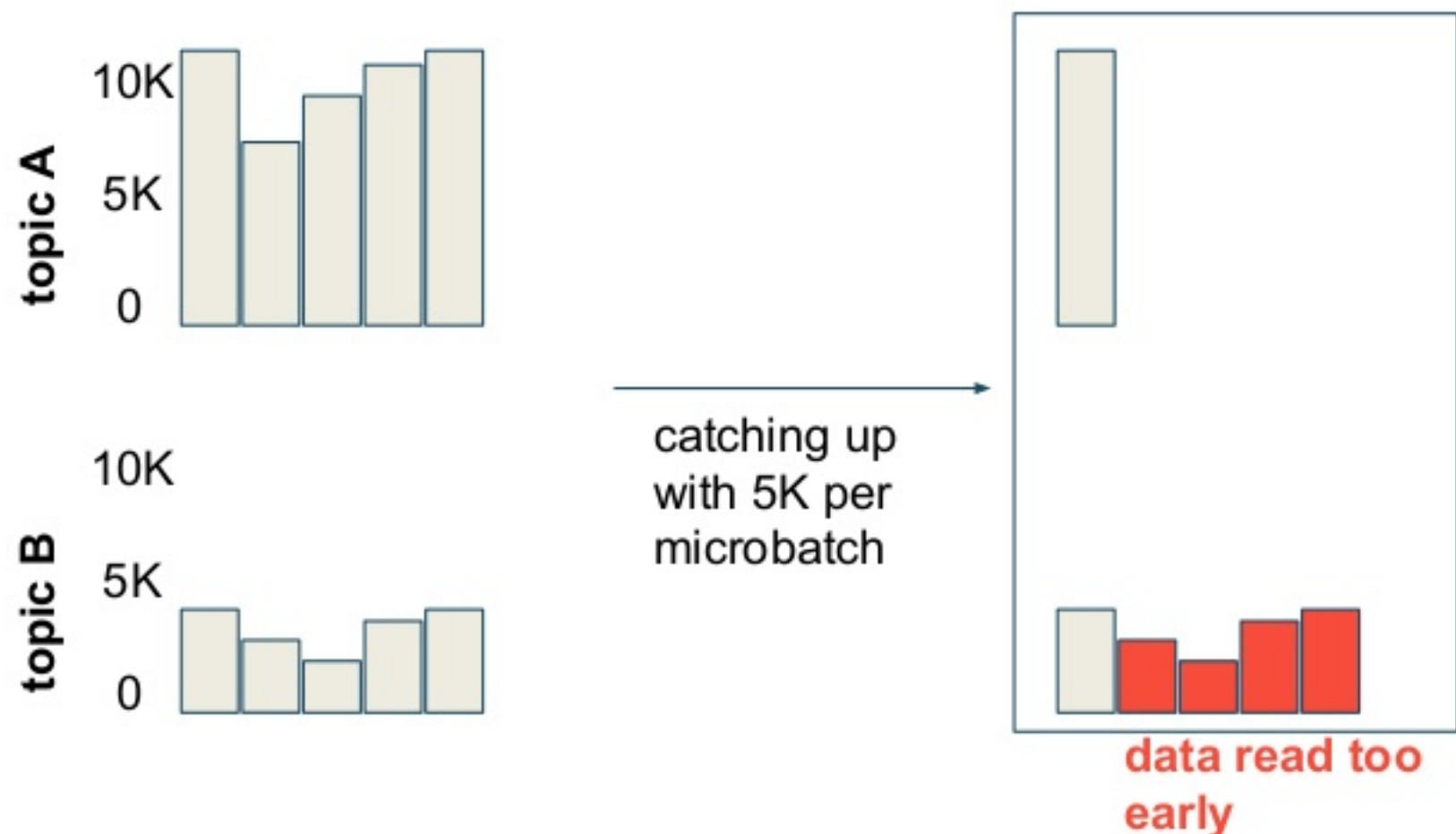
# Messages are timed out by event time

```
def flagIfTimedOut(mergedMessage: MergedMessage): MergedMessage = {  
  val isTimedOut = minBusinessClocks  
    .get(mergedMessage.meta.partition)  
    .exists(mbc => mergedMessage.timestamp < (mbc - TimeoutPeriod))  
  
  if (isTimedOut) {  
    mergedMessage.asTimedOut(minSourceTopicOffsets.  
      get(mergedMessage.meta.partition))  
  } else {  
    mergedMessage  
  }  
}
```

# Advanced Requirements



# Effect of different request rates in source topics



## Handle different request rates in topics

- Stop reading from one topic if the event time of other topic is too far behind
- Store latest event time of each topic and partition on driver
- Use custom KafkaDStream with clamp method which uses the event time



# Handle different request rates in topics

```
private var latestTracking: Map[Int, Long] = Map.empty
private var latestOrder: Map[Int, Long] = Map.empty

def createStreams(ssc: StreamingContext,
                  cProps: Map[String, String],
                  pProps: Map[String, String],
                  readOffsets: Map[TopicAndPartition, Long]):
(DStream[InputMessage], DStream[InputMessage]) = {

  val trackingStream: DStream[InputMessage] = createStream(cProps, pProps,
    readOffsets.filter(_._1.topic == Settings.topic), ssc,
    (raw, p, o) => TrackingMessage(TrackingMessage.Meta(p, o), raw),
    partition => latestTracking.get(partition),
    partition => latestOrder.get(partition))
  val orderStream: DStream[InputMessage] = createStream(...)
```



# Handle different request rates in topics

```
val trackingStream: DStream[InputMessage] = createStream(...)
val orderStream: DStream[InputMessage] = createStream(...)

storeLatestTimestamp(trackingStream) { maxTimestamps =>
  latestTracking = latestTracking ++ maxTimestamps }
storeLatestTimestamp(orderStream) { maxTimestamps =>
  latestOrder = latestOrder ++ maxTimestamps }

(trackingStream, orderStream)
}
```

# Handle different request rates in topics

```
override def clamp(leaderOffsets: Map[TopicAndPartition, LeaderOffset]):  
Map[TopicAndPartition, LeaderOffset] = {  
  
  val ownTimestampValues = for {  
    (tp@TopicAndPartition(_, partition), _) <- leaderOffsets  
    ownTimestamp <- this.obtainOwnTimestamp(partition)  
  } yield (tp, ownTimestamp)  
  
  val otherTimestampValues = for {  
    (tp@TopicAndPartition(_, partition), _) <- leaderOffsets  
    otherTimestamp <- this.obtainOtherTimestamp(partition)  
  } yield (tp, otherTimestamp)
```

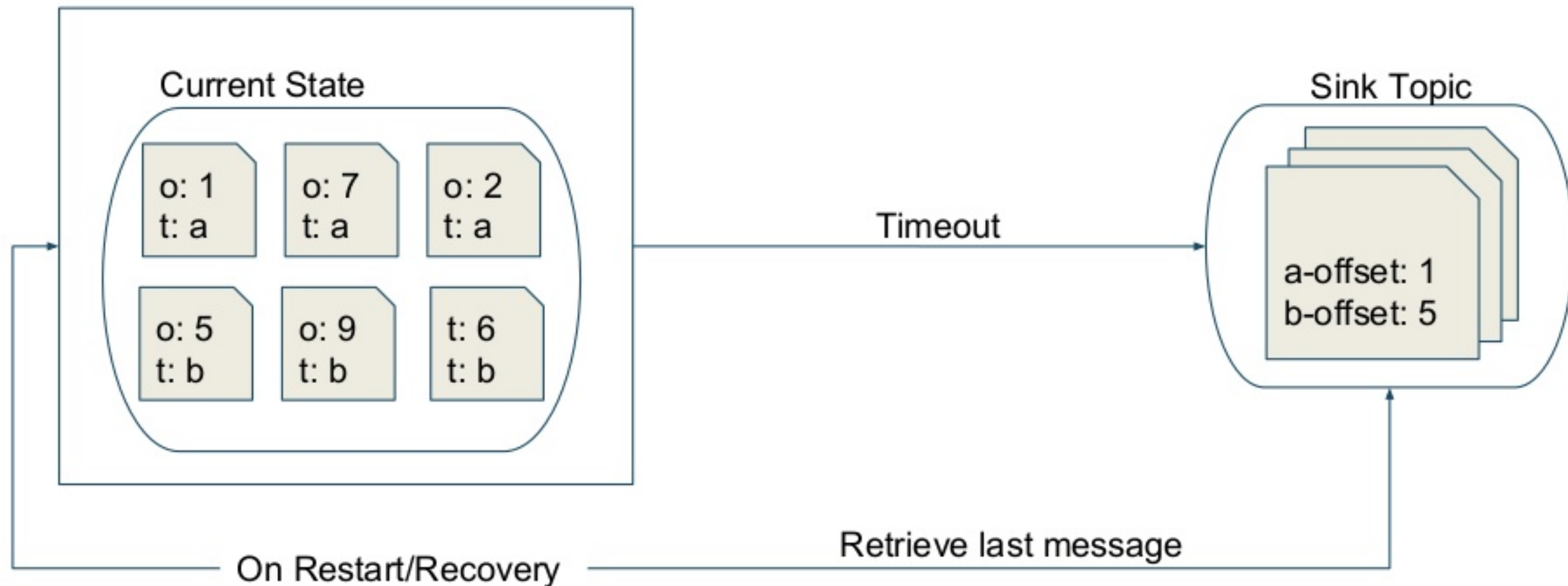


# Handle different request rates in topics

```
lazy val standardOffsets = super.clamp(leaderOffsets)
leaderOffsets.map { case (tP, leaderOffset) =>
  def stopReading = (tP, leaderOffset.copy(offset = currentOffsets(tP)))
  def continueReading = (tP, standardOffsets(tP))
  (ownTimestampValues.get(tP), otherTimestampValues.get(tP)) match {
    case (None, _) => // no values in this stream yet
      continueReading
    case (_, None) => // stop reading until other stream got values
      stopReading
    case (Some(ownValue), Some(oValue)) if ownValue - oValue > maxLead =>
      stopReading
    case (Some(ownValue), Some(oValue)) =>
      continueReading
  }
}
```

# Guarantee at least once semantics

## Spark Merge Service





# Guarantee at least once semantics

```
def lookupOffsets(cProps: Map[String, String]):  
  Map[TopicAndPartition, Long] = {  
  
    val sourceOffsetsFromSink =  
      for {  
        (partition, mergedMessage) <- retrieveLastMergedMessage(cProps)  
        offsets <- mergedMessage.meta.minSourceTopicOffsets.toList  
        (topic, offset) <- List(  
          S.topic -> offsets.tracking,  
          S.orderTopic -> offsets.order)  
      } yield TopicAndPartition(topic, partition) -> offset  
  
    val tPs = retrievePartitions(cProps, Set(S.topic, S.orderTopic))  
    val earliestSourceOffsets = retrieveEarliestOffsets(cProps, tPs)  
    val latestSourceOffsets = retrieveLatestOffsets(cProps, tPs)
```

# Guarantee at least once semantics

```
tPs.map { topicAndPartition =>
  val offset: Long =
    (earliestSourceOffsets.get(topicAndPartition),
     sourceOffsetsFromSink.get(topicAndPartition),
     latestSourceOffsets.get(topicAndPartition)) match {
      case (Some(earliestOffset), Some(offsetFromSink), _)
        if earliestOffset <= offsetFromSink =>
        offsetFromSink
      case (Some(earliestOffset), Some(earliestFromSink), Some(latest)) =>
        earliestOffset //offset already gone in source
      case (_, _, Some(latest)) =>
        latest //min offset could not be retrieved
    }
  topicAndPartition -> offset
}.toMap
```



# Everything put together

```
def work(appInfo: AppInfo,
        ssc: StreamingContext,
        cProps: Map[String, String],
        pProps: Map[String, String],
        customReadOffsets: Option[Map[TopicAndPartition, Long]]): Unit = {

  val readOffsets = customReadOffsets.getOrElse(lookupOffsets(cProps))
  val (streamTracking, streamOrder) = createStreams(ssc, cProps, pProps, readOffsets)
  val state = Merger.merge(streamTracking.union(streamOrder))
  val timedOutMessages = state.filter { case (_, message) => message.meta.timedOut }
  val validTimedOutMessages = timedOutMessages.filter(isValid)
  validTimedOutMessages.foreachRDD(MessageSender.send(cProps, pProps))
}
```

# Lessons learned

- Excellent Performance and Scalability
- Extensible via Custom RDDs and Extension to DirectKafka
- No event time windows in Spark Streaming  
See: <https://github.com/apache/spark/pull/2633>
- Checkpointing cannot be used when deploying new artifact versions
- Driver/executor model is powerful, but also complex



# THANK YOU.

Twitter:

@sistar\_hh

@Sebasti0n

Blog: [dev.otto.de](http://dev.otto.de)

Jobs: [otto.de/jobs](http://otto.de/jobs)

