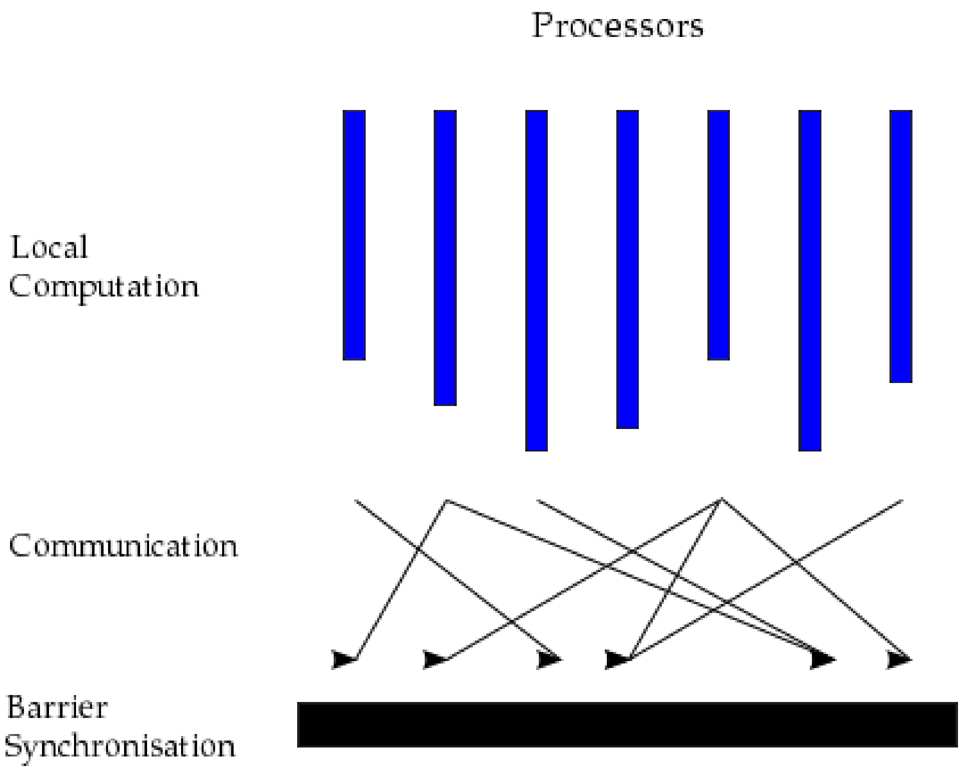
Whatever is left

Alexander Lazovik
a.lazovik@rug.nl

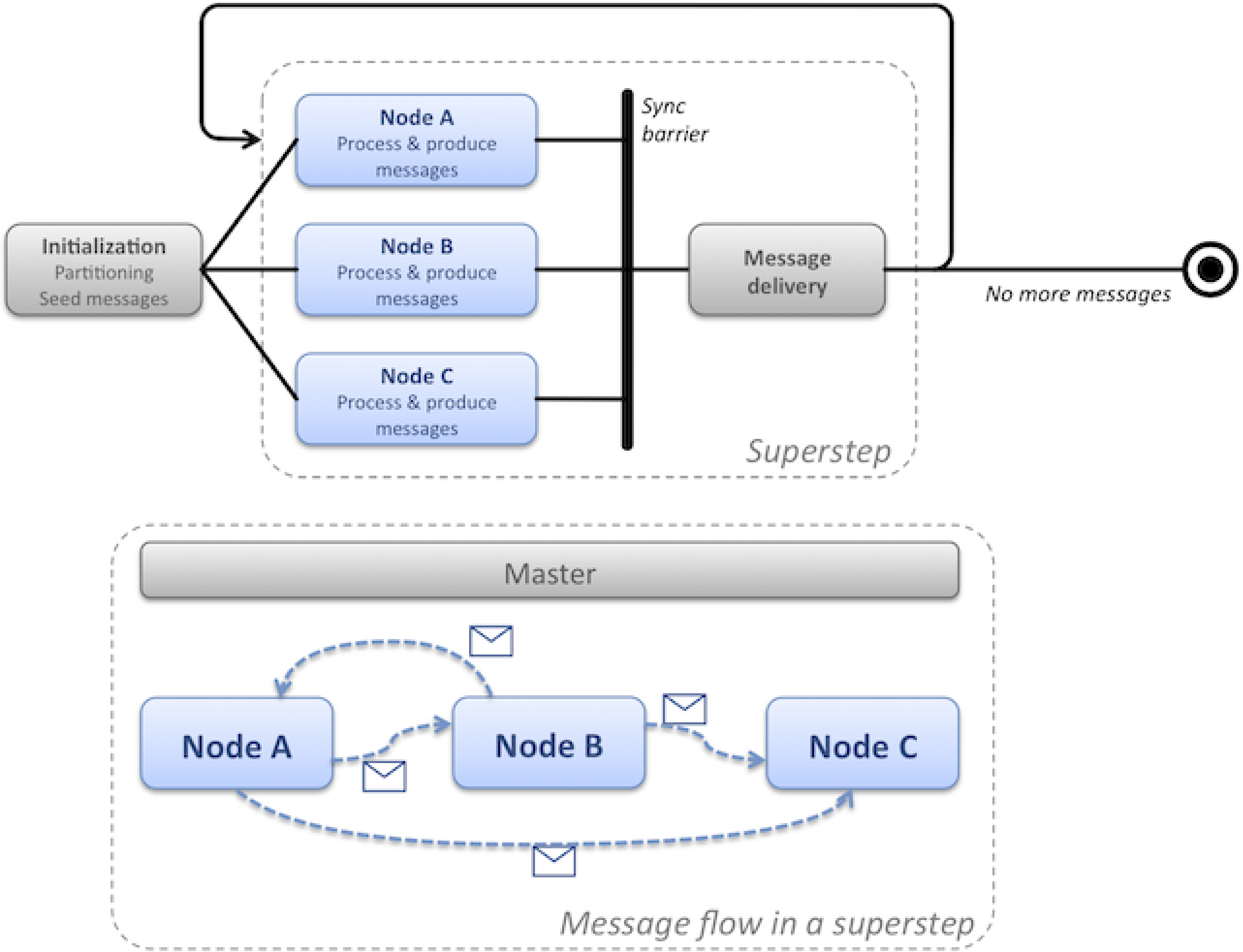
http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Bulk Synchronous Parallel (1990)

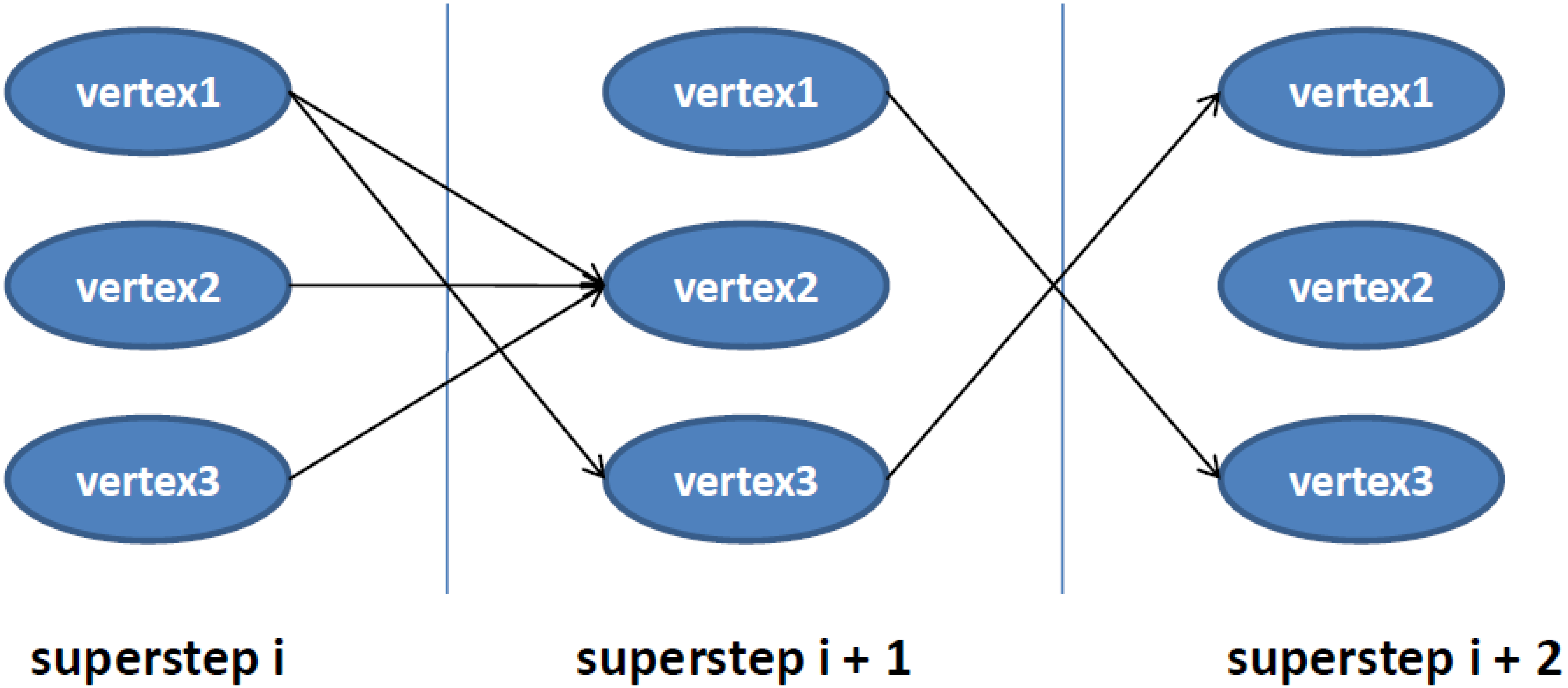


http://blog.octo.com/wp-content/uploads/2012/06/bsp1.png

BSP



BSP on distributed graphs



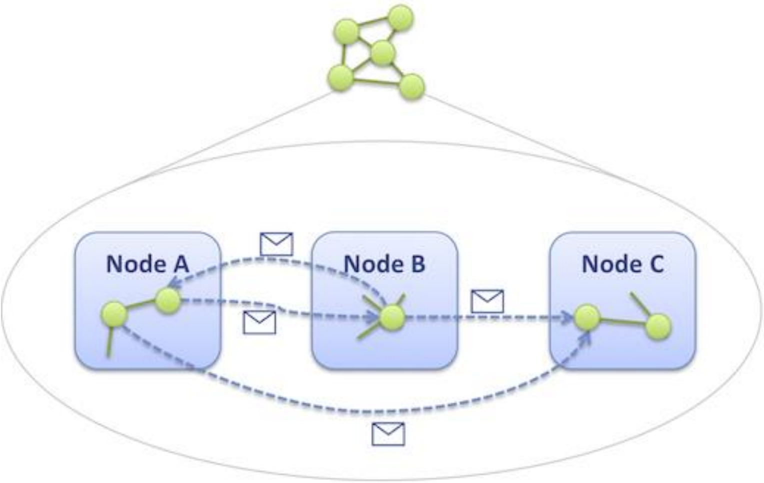
Map/Reduce implementation

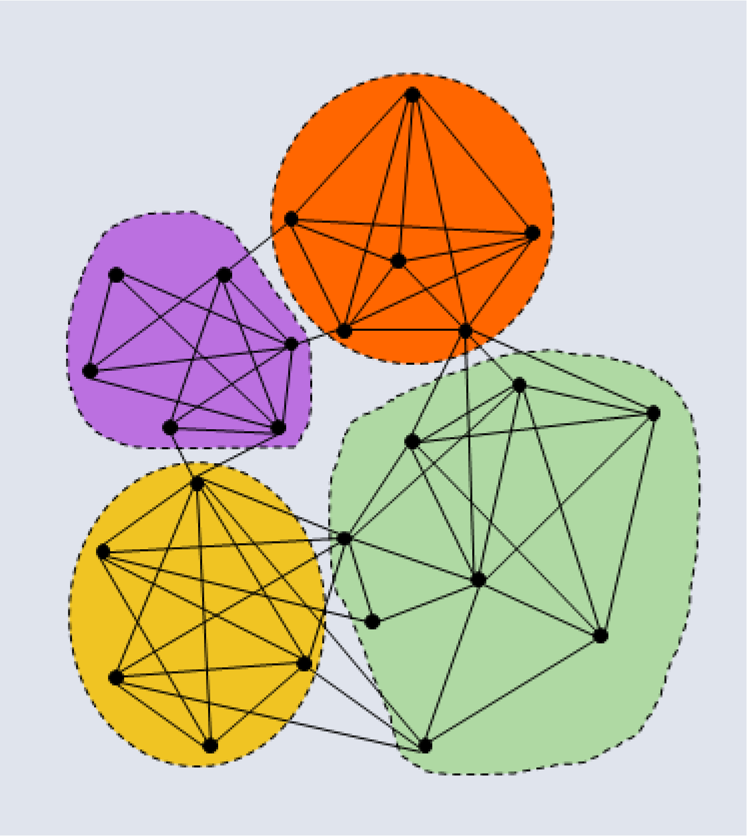
- *https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/*

http://www.slideshare.net/nitayj/20130910-giraph-at-london-hadoop-users-group

Pregel: BSP applied to distributed graphs

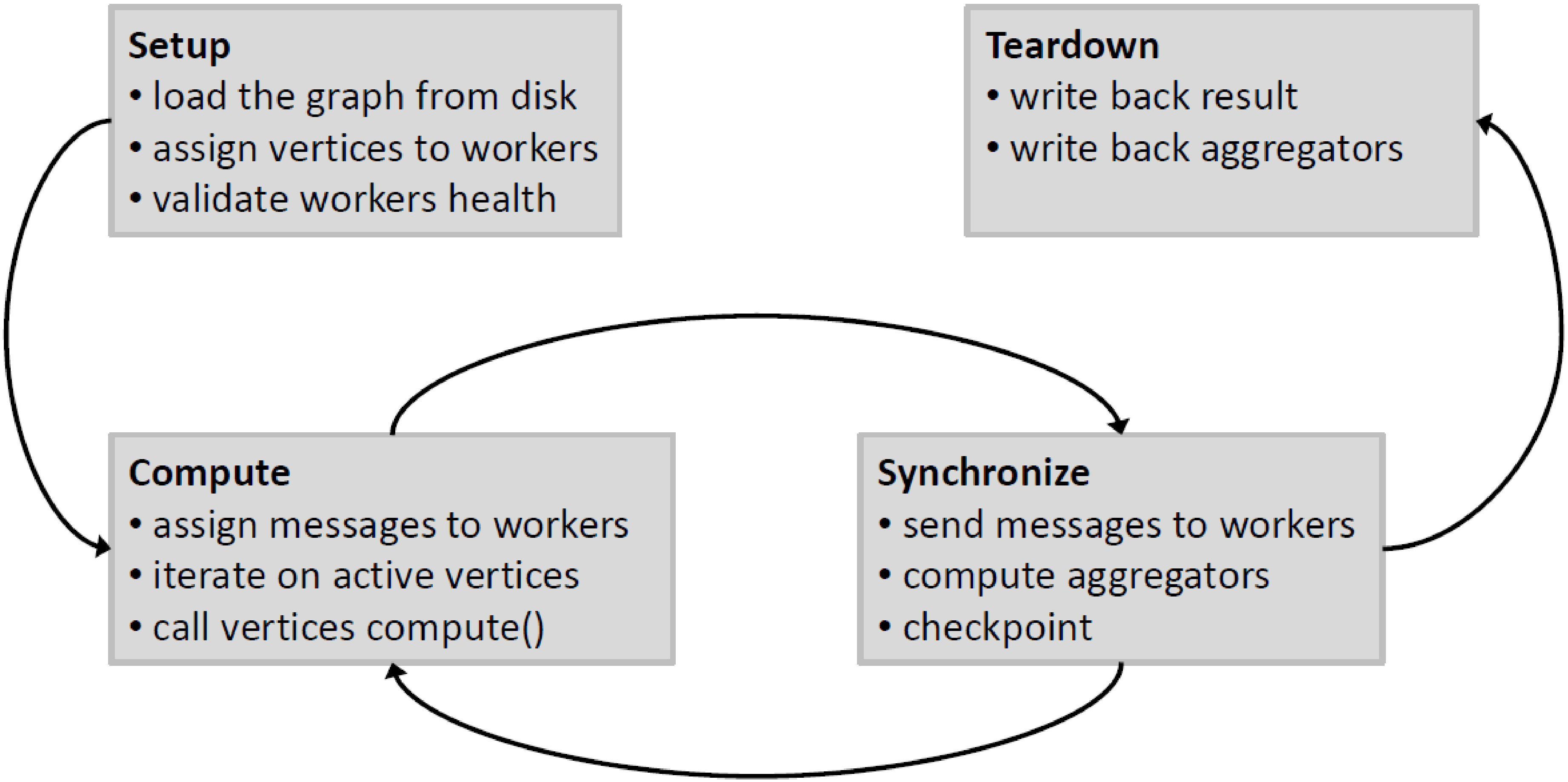
- Apache Giraph: an open source implementation of Pregel
  - graph is distributed across several machines
  - computation "node" is a vertex





http://www.slideshare.net/sscdotopen/introducing-apache-giraph-for-large-scale-graph-processing

Apache Giraph



Setup
- load the graph from disk
- assign vertices to workers
- validate workers health

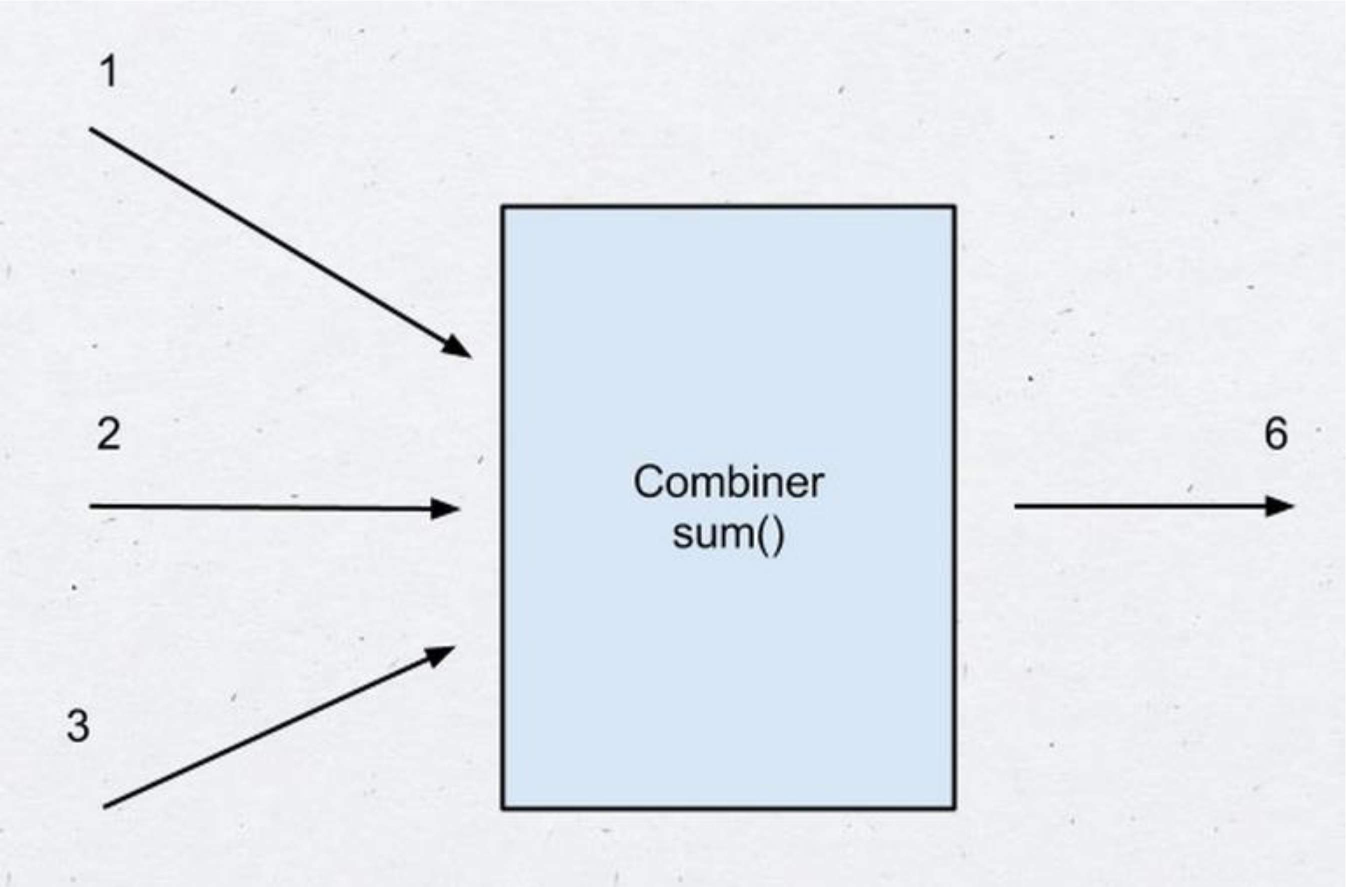
Teardown
- write back result
- write back aggregators

Compute
- assign messages to workers
- iterate on active vertices
- call vertices compute()

Synchronize
- send messages to workers
- compute aggregators
- checkpoint

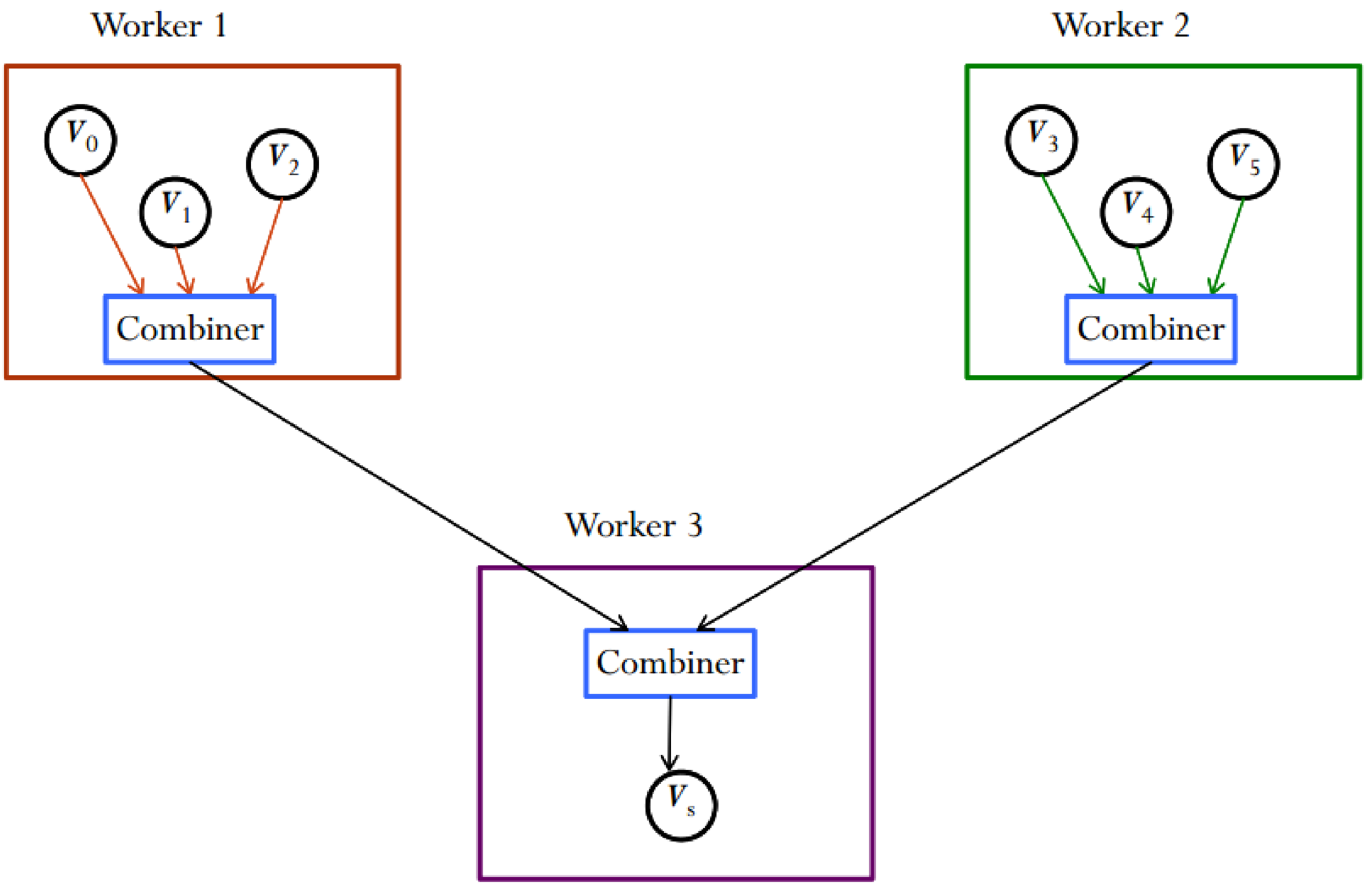
http://www.slideshare.net/juliaproskurnia/pregel

Combiners (user-defined)



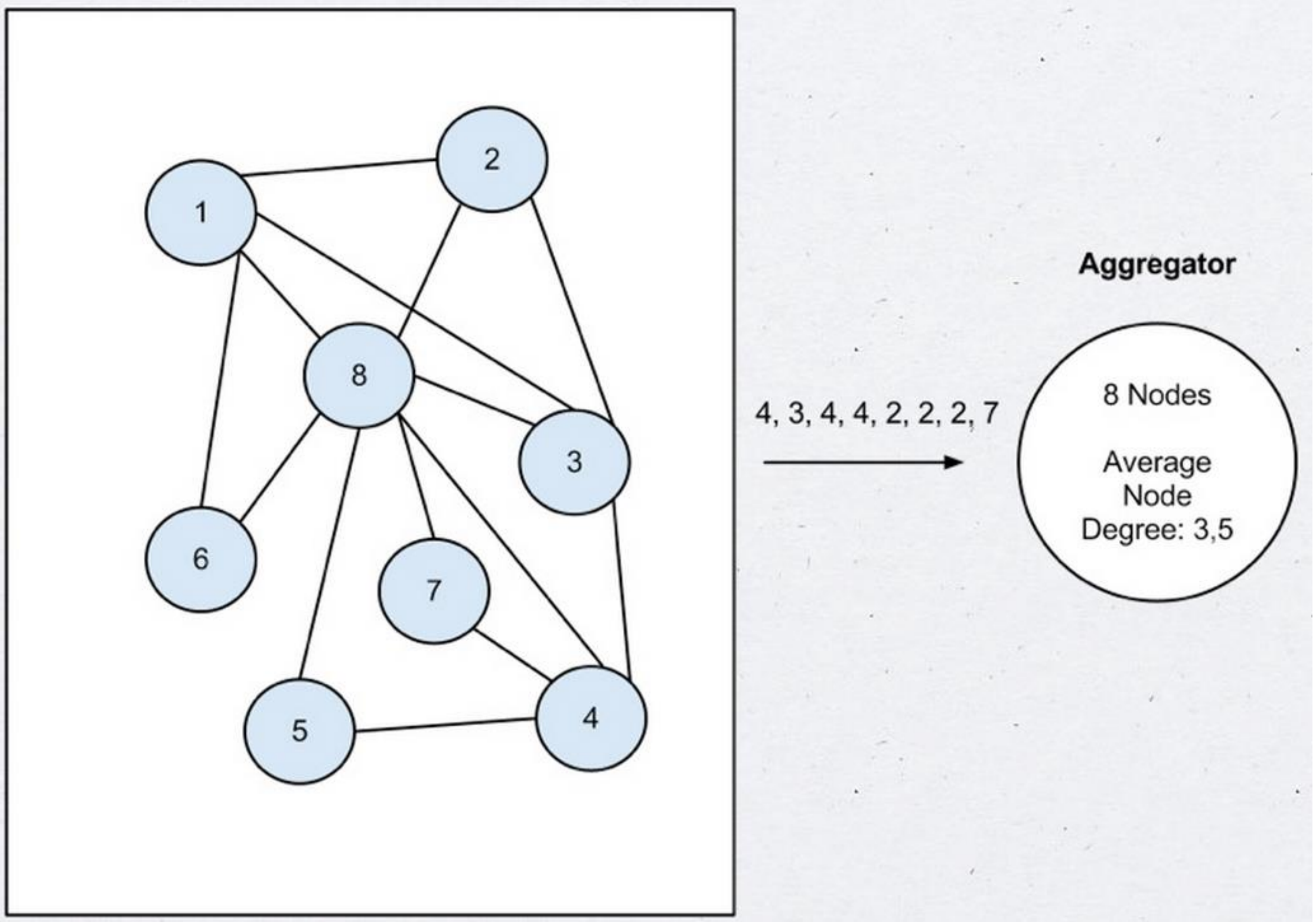
https://wiki.engr.illinois.edu/download/attachments/188588798/pregel.pdf?version=1

Combiners



main goal: reduce network bandwidth / number of supersteps http://www.slideshare.net/juliaproskurnia/pregel

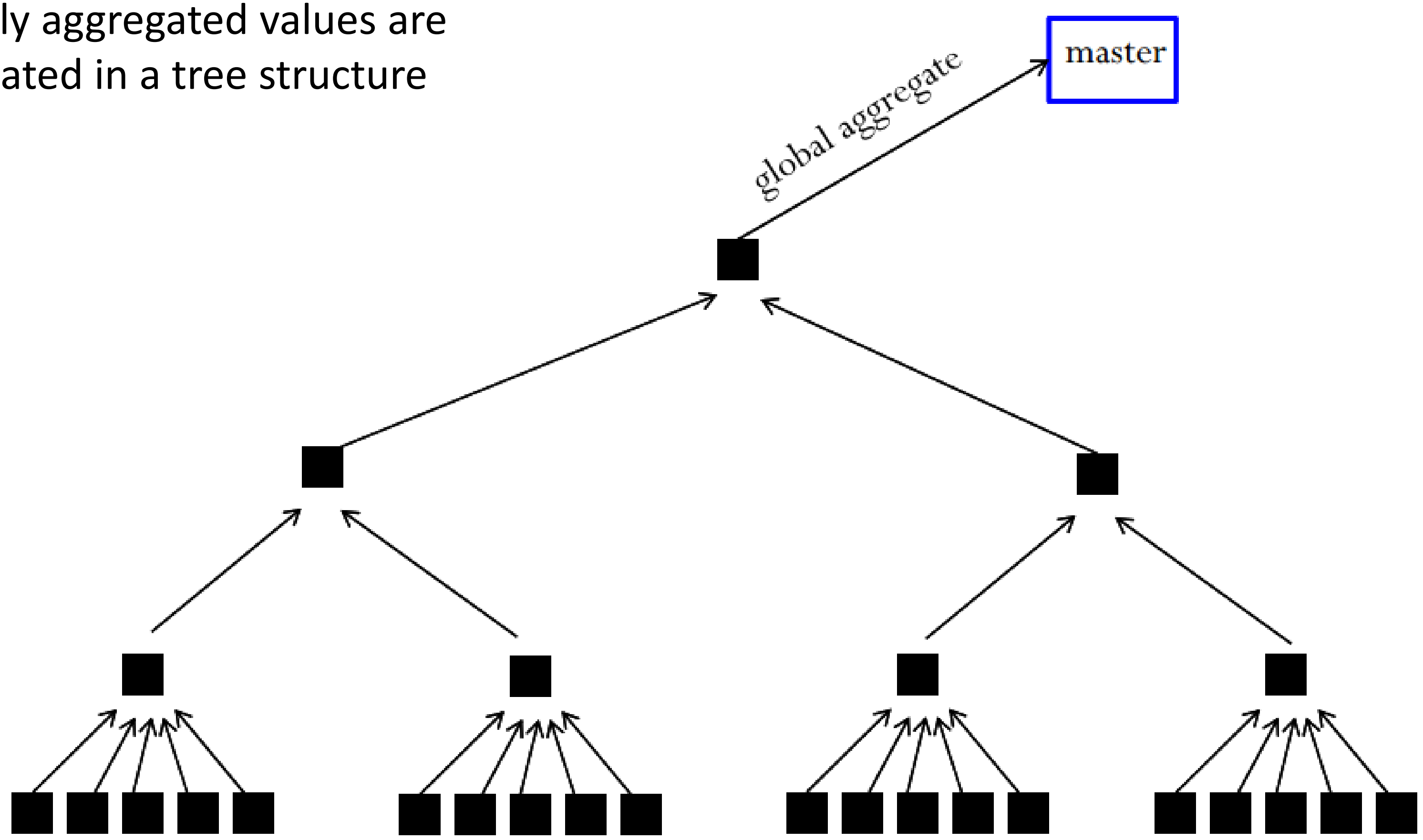
Aggregators



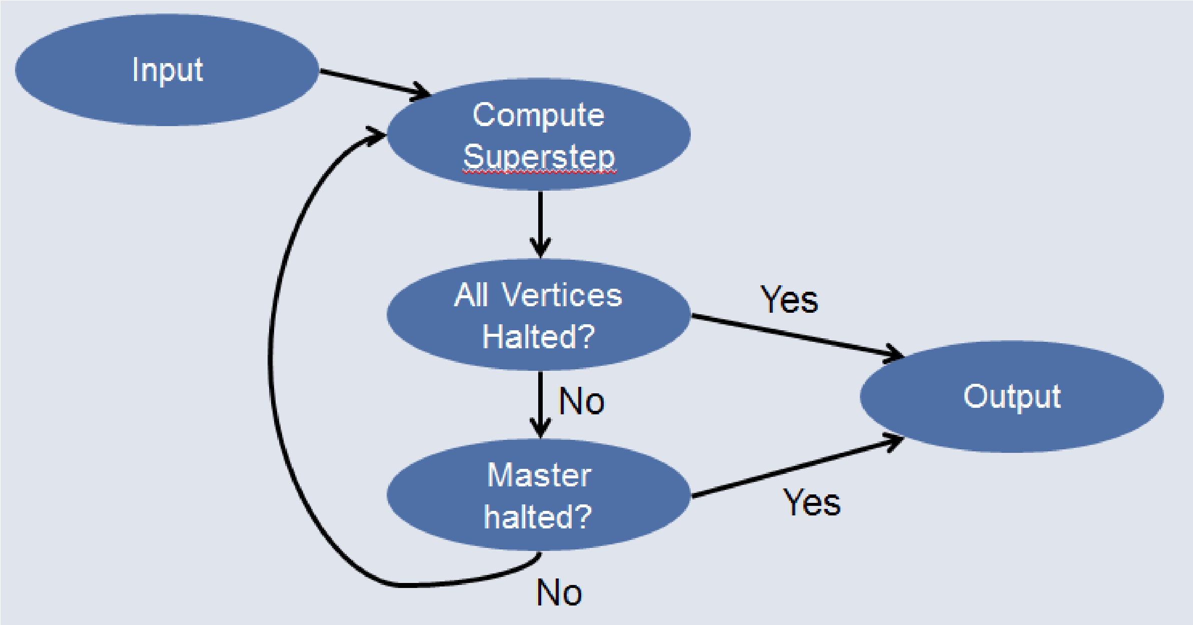
https://wiki.engr.illinois.edu/download/attachments/188588798/pregel.pdf?version=1

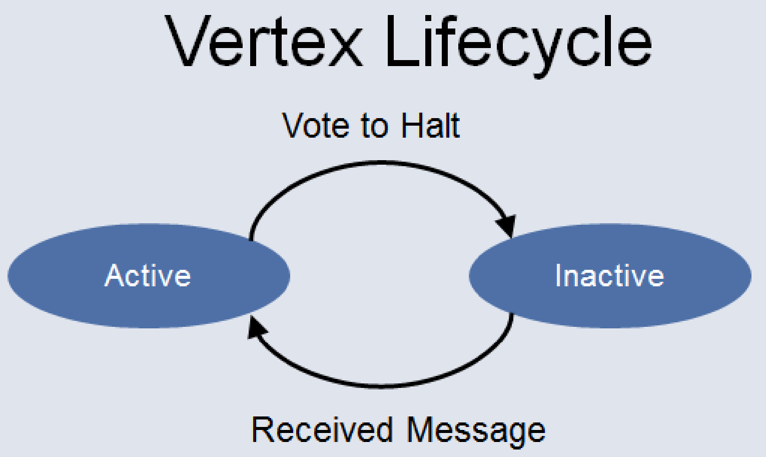
Aggregators

- each worker aggregates values from its vertices
- partially aggregated values are aggregated in a tree structure

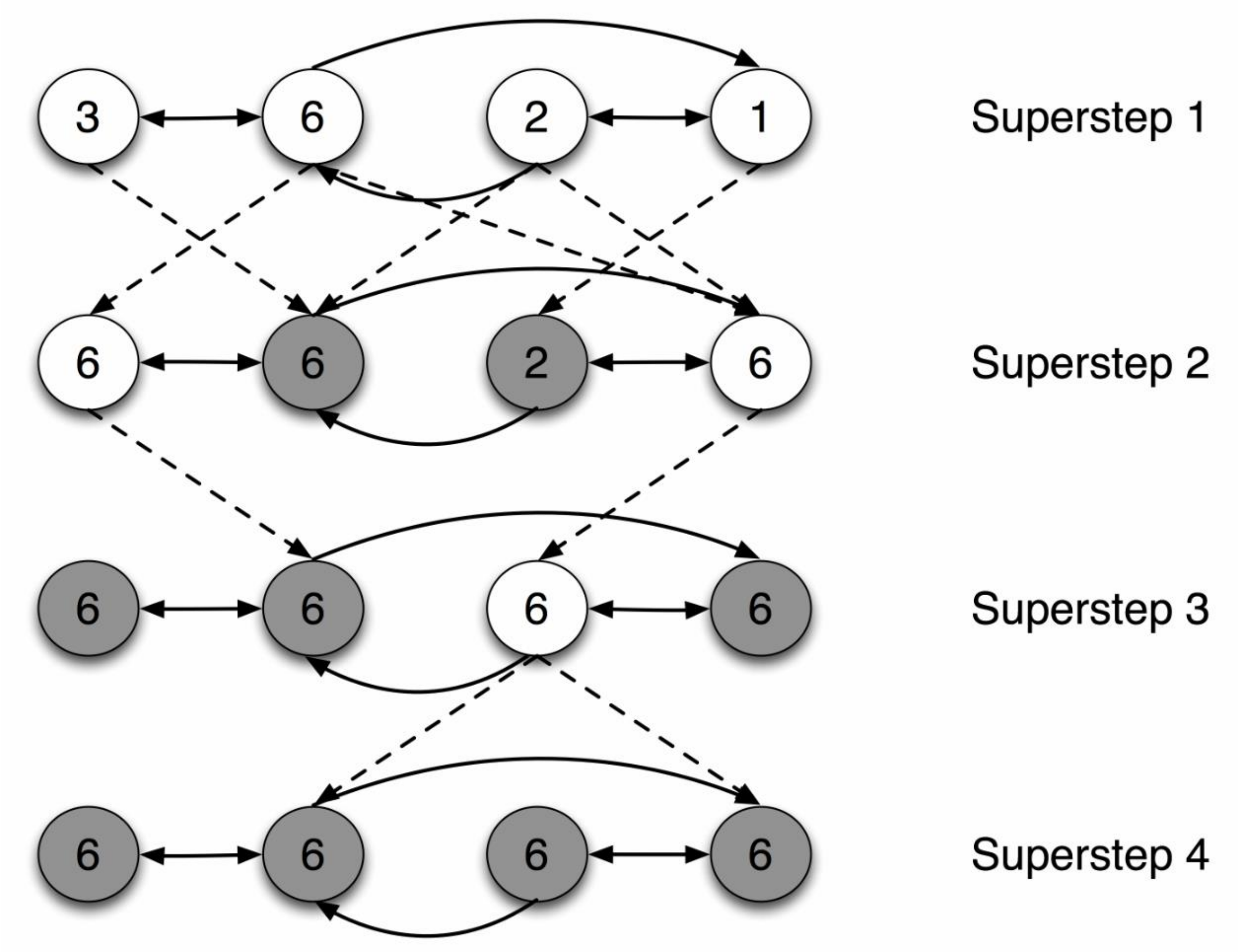


Apache Giraph Lifecycle





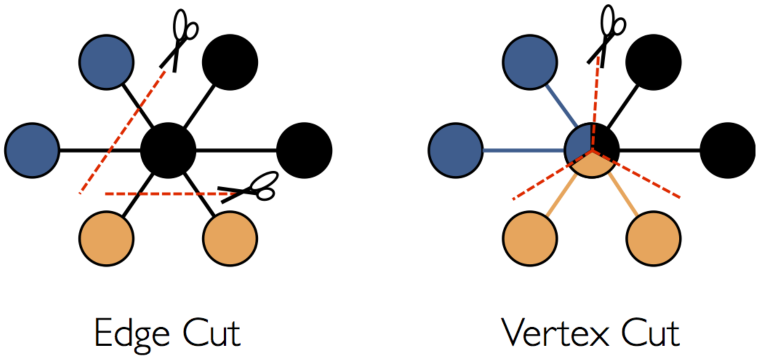
Finding max


3
6
2
1
Superstep 1
6
6
2
6
Superstep 2
6
6
6
6
Superstep 3
6
6
6
6
Superstep 4

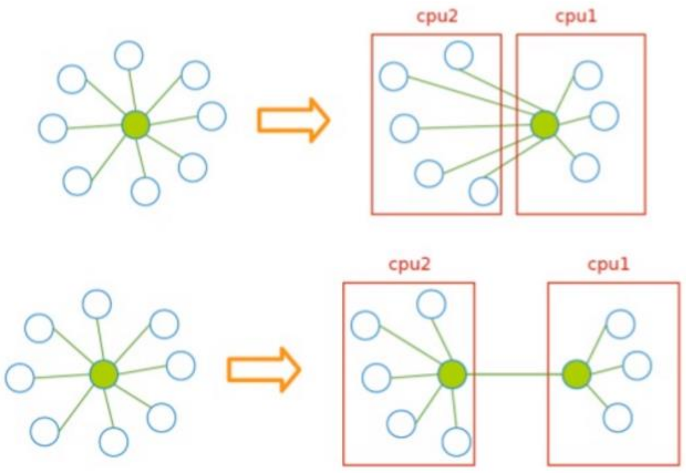
https://pdfs.semanticscholar.org/6219/cb8ffc5d1ca18231d1255706caac81fd6858.pdf?_ga=1.172330662.468480408.1490021126

https://www.slideshare.net/payberah/graph-processing-powergraph-and-graphx

Spark GraphX: Resilient Distributed Graphs



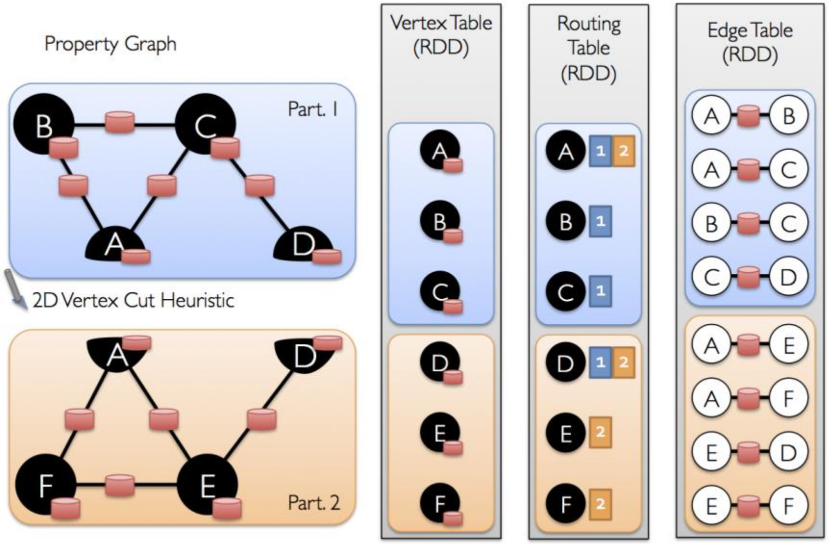
vertex vs. edge cutting

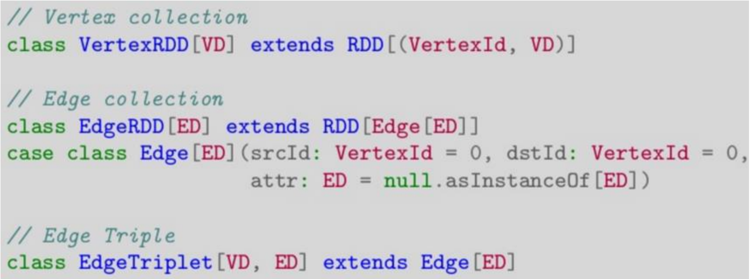


https://pdfs.semanticscholar.org/6219/cb8ffc5d1ca18231d1255706caac81fd6858.pdf?_ga=1.172330662.468480408.1490021126

https://www.slideshare.net/payberah/graph-processing-powergraph-and-graphx

Spark GraphX: Resilient Distributed Graphs

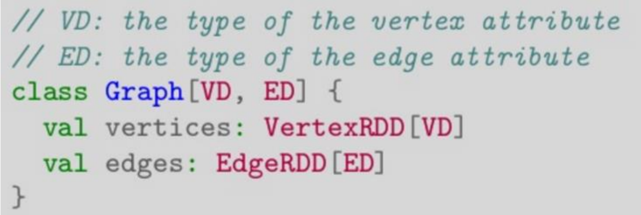




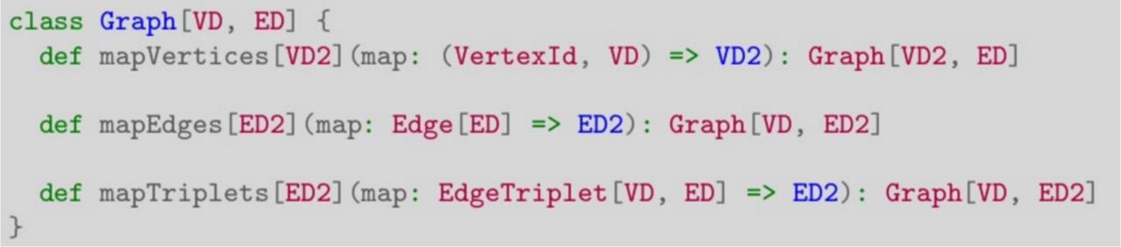
```
// Vertex collection
class VertexRDD[VD] extends RDD[(VertexId, VD)]

// Edge collection
class EdgeRDD[ED] extends RDD[Edge[ED]]
case class Edge[ED](srcId: VertexId = 0, dstId: VertexId = 0,
                    attr: ED = null.asInstanceOf[ED])

// Edge Triple
class EdgeTriplet[VD, ED] extends Edge[ED]
```



```
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```



```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]

  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]

  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

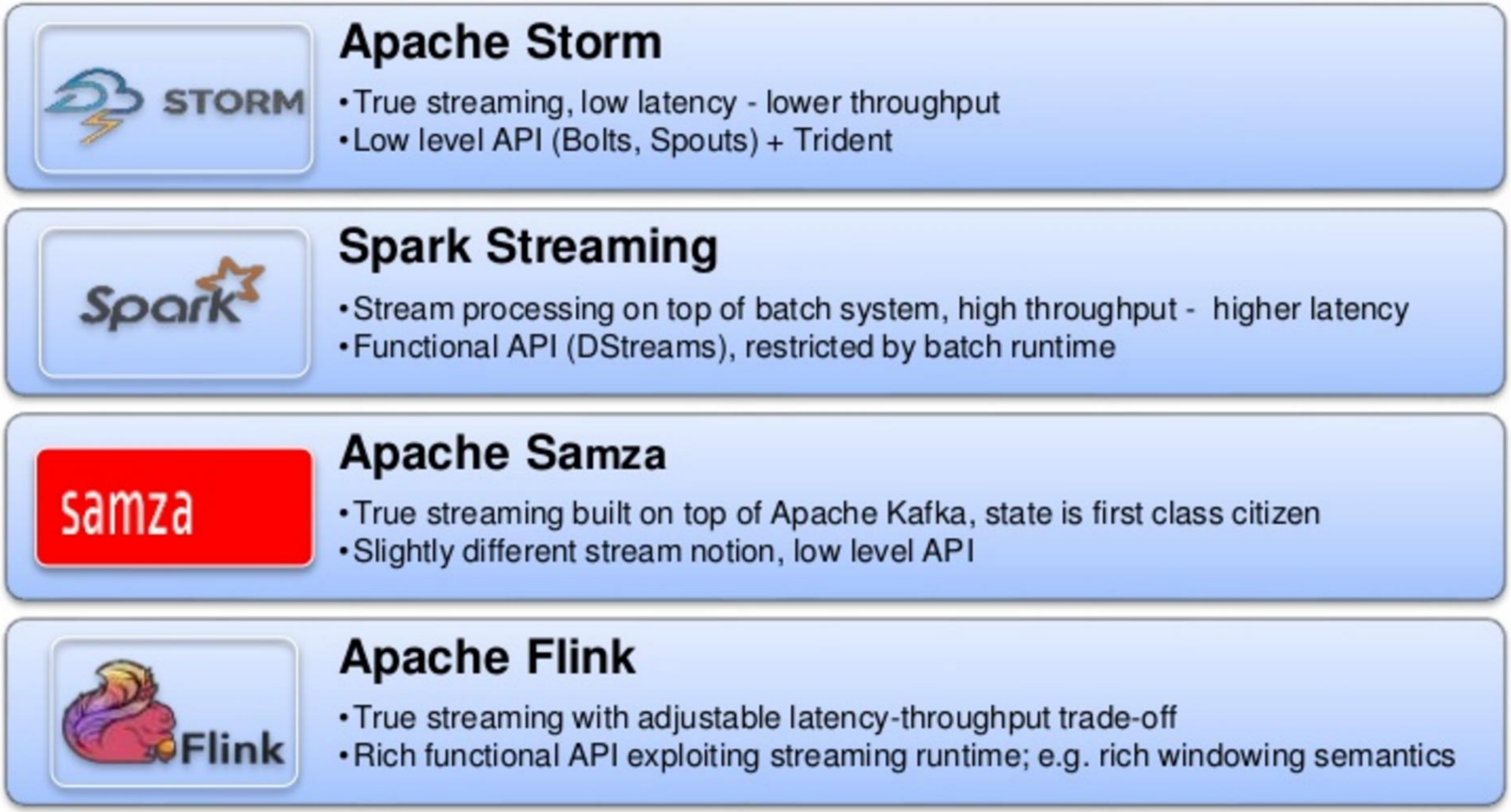
Streaming (unbounded)

- You cannot get answers to some questions
  - e.g., what is the average of all elements?
  - you can answer them for a given subset though
    - last 20 elements, last 20 seconds
  - effectively, generating a new stream
    - not necessarily after each 20 elements

- Two approaches:
  - Microbatching (combine elements and then use existing tools)
    - e.g., Spark
    - additional buffering may help better distribute load
      - do not have a convincing example (though, you can think of some)
  - Process-each-element
    - e.g., Twitter Storm, Apache Flink
    - response time/availability of the first processed element is better

https://www.slideshare.net/Hadoop_Summit/realtime-stream-processing-with-apache-flink

Streaming frameworks



Apache Storm

- True streaming, low latency - lower throughput
- Low level API (Bolts, Spouts) + Trident

Spark Streaming

- Stream processing on top of batch system, high throughput - higher latency
- Functional API (DStreams), restricted by batch runtime

Apache Samza

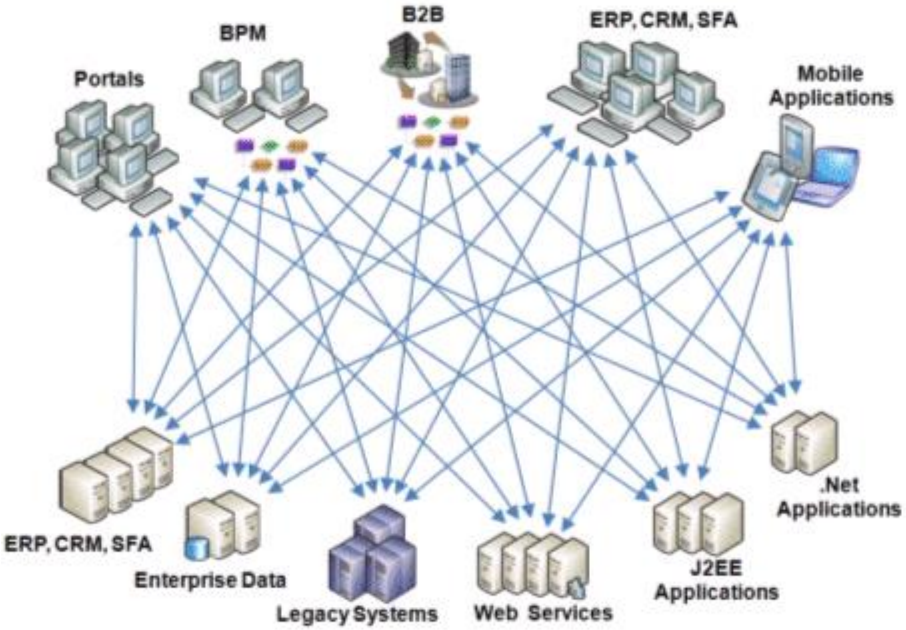
- True streaming built on top of Apache Kafka, state is first class citizen
- Slightly different stream notion, low level API

Apache Flink

- True streaming with adjustable latency-throughput trade-off
- Rich functional API exploiting streaming runtime; e.g. rich windowing semantics

http://msdn.microsoft.com/en-us/library/bb220799.aspx

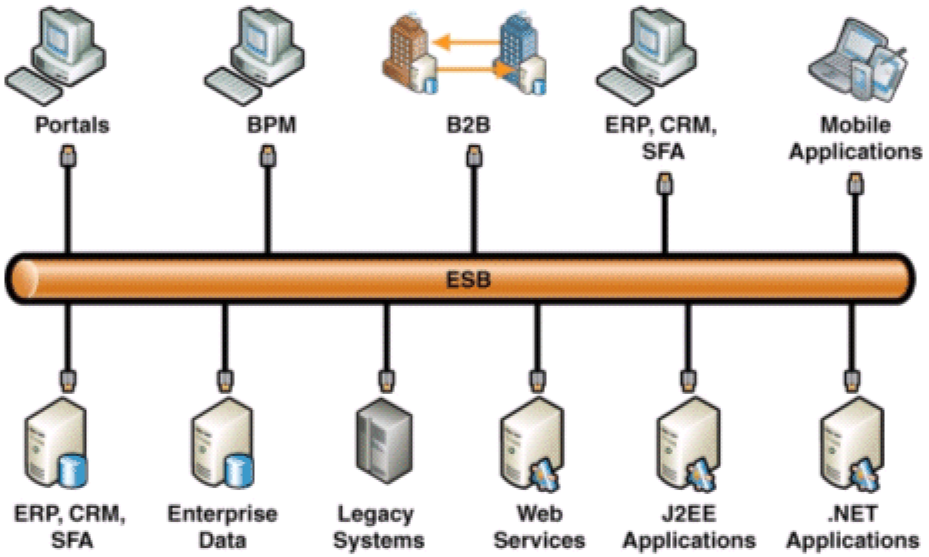
Enterprise Integration



point-to-point communication

http://msdn.microsoft.com/en-us/library/bb220799.aspx

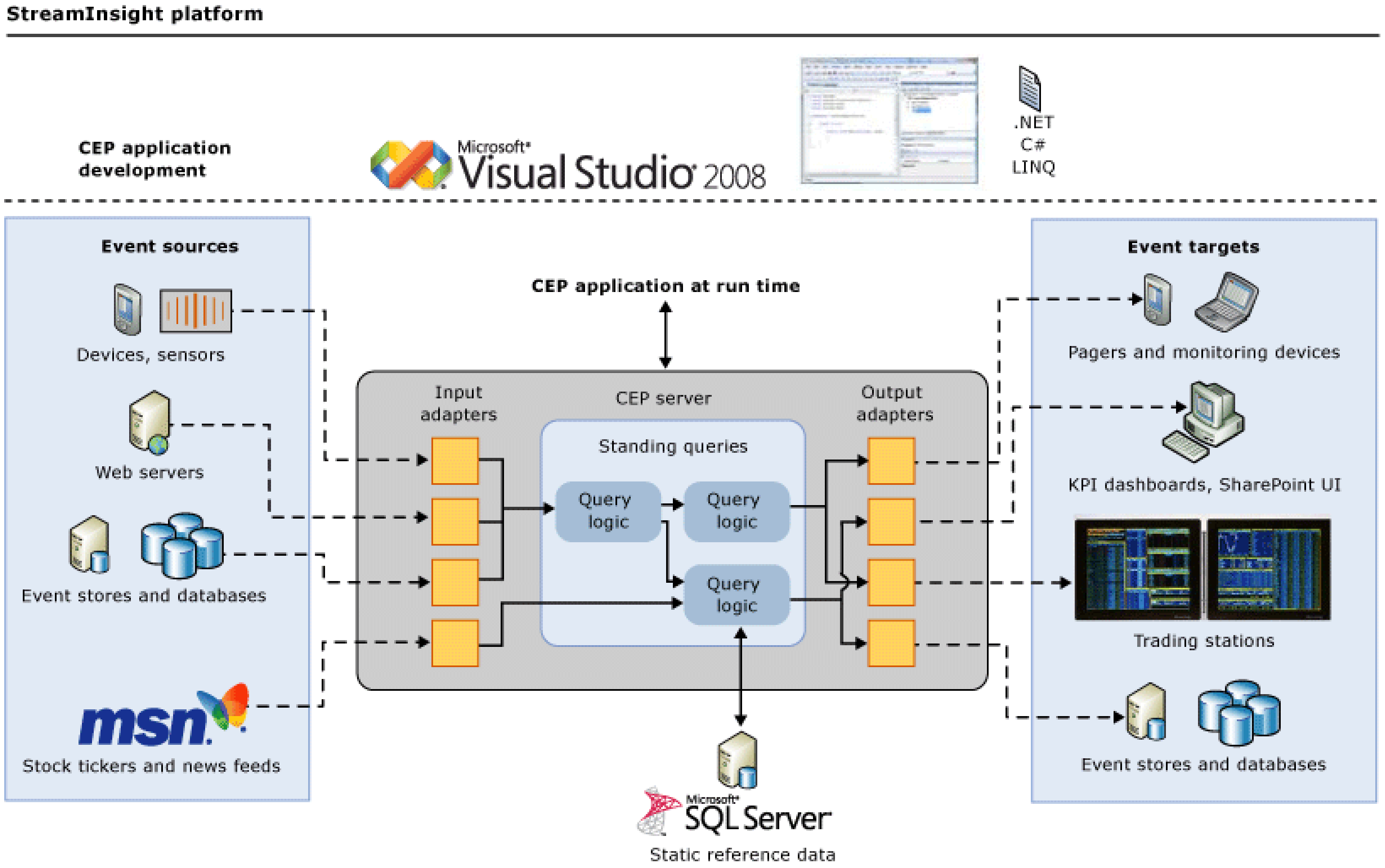
Enterprise Integration



enterprise service bus

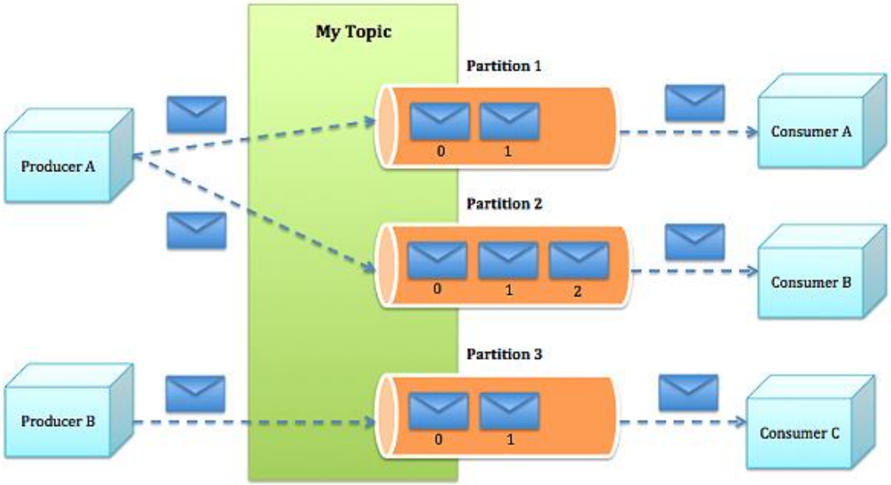
consider integration across organizations

Complex Event Processing (CEP)

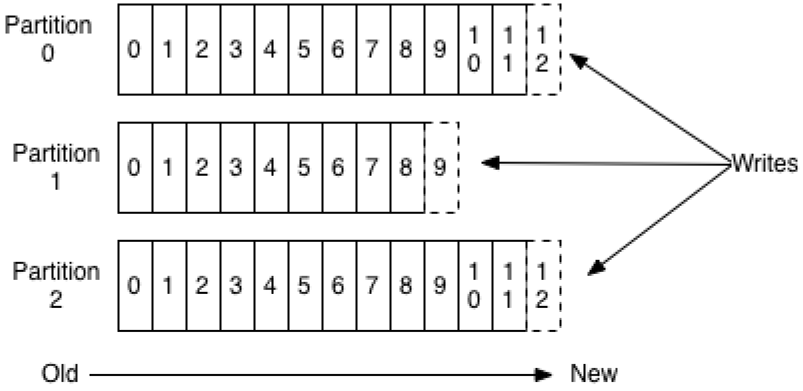


https://www.infoq.com/articles/apache-kafka https://kafka.apache.org/intro

Kafka



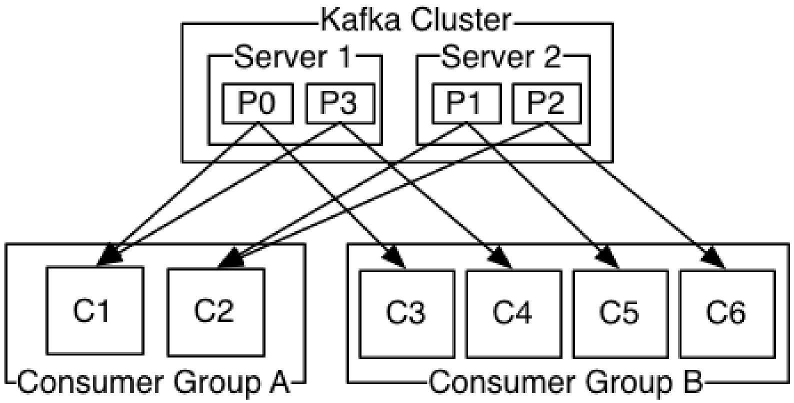
Anatomy of a Topic



- partitions are immutable sequences
  - and stored (on disk) for a configurable retention period
- producer is responsible to chose topic/partition
- within a partition, order is preserved
- partitions are replicated over several servers (one acting as a leader)

https://www.infoq.com/articles/apache-kafka https://kafka.apache.org/intro

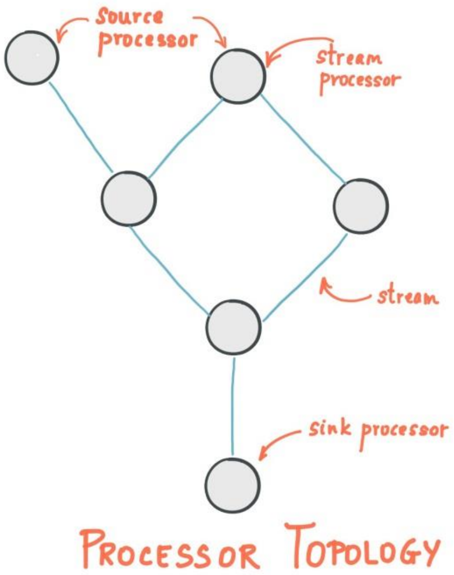
Kafka

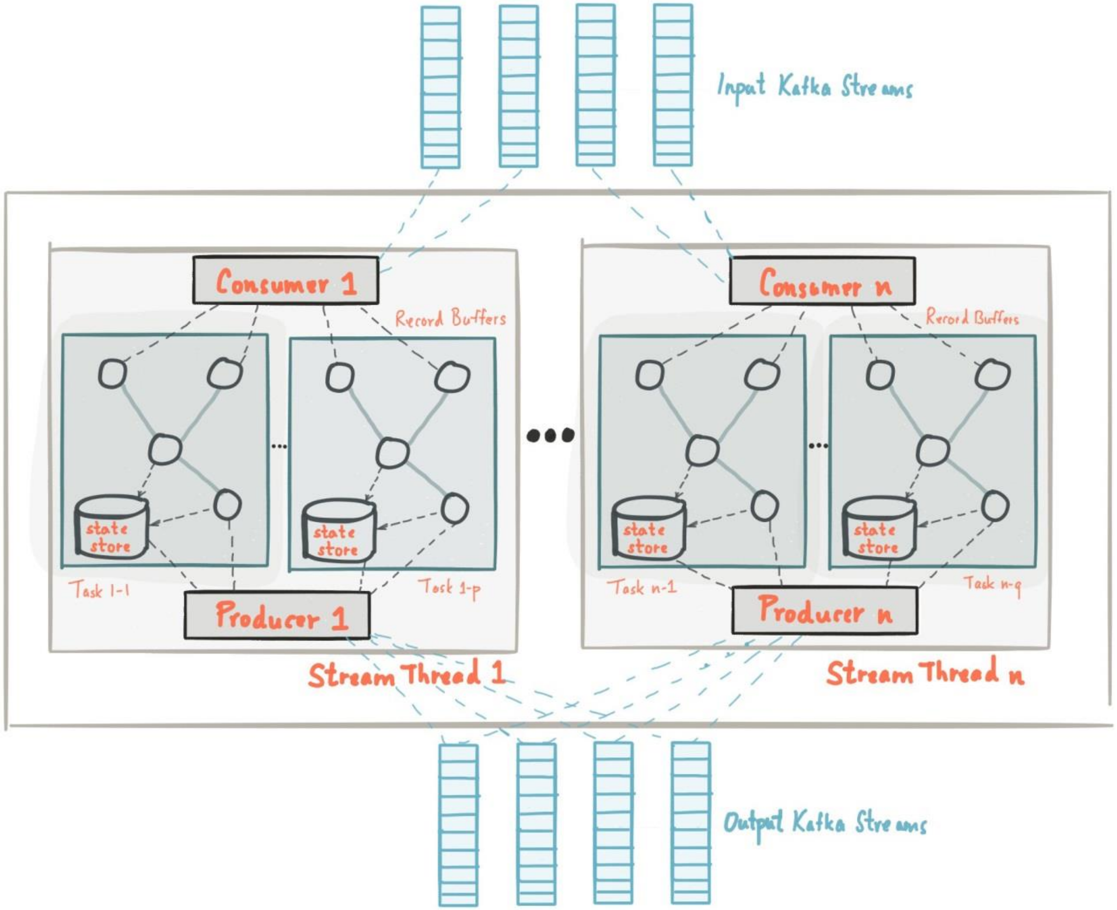


- each record is delivered to one consumer from each consumer group
  - one consumer group => load balancing
  - consumer group per consumer => broadcasting
- each consumer is assigned a partition exclusively
  - one partition => max one consumer per consumer group

http://docs.confluent.io/3.0.0/streams/architecture.html

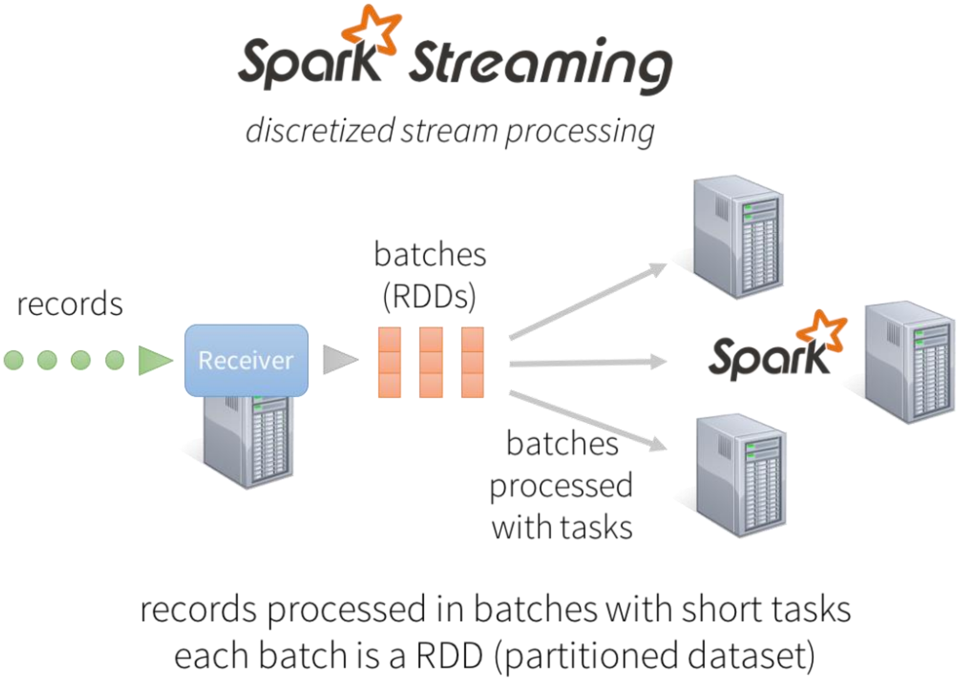
Kafka Streaming API





map, filter, reduceByKey, …

Spark Microbatching approach


Spark Streaming
discretized stream processing
records
Receiver
batches (RDDs)
Spark
batches processed with tasks
records processed in batches with short tasks
each batch is a RDD (partitioned dataset)

- DStream is a stream of RDDs
  - each element (RDD) representing one batch

https://flink.apache.org/news/2015/12/04/Introducing-windows.html

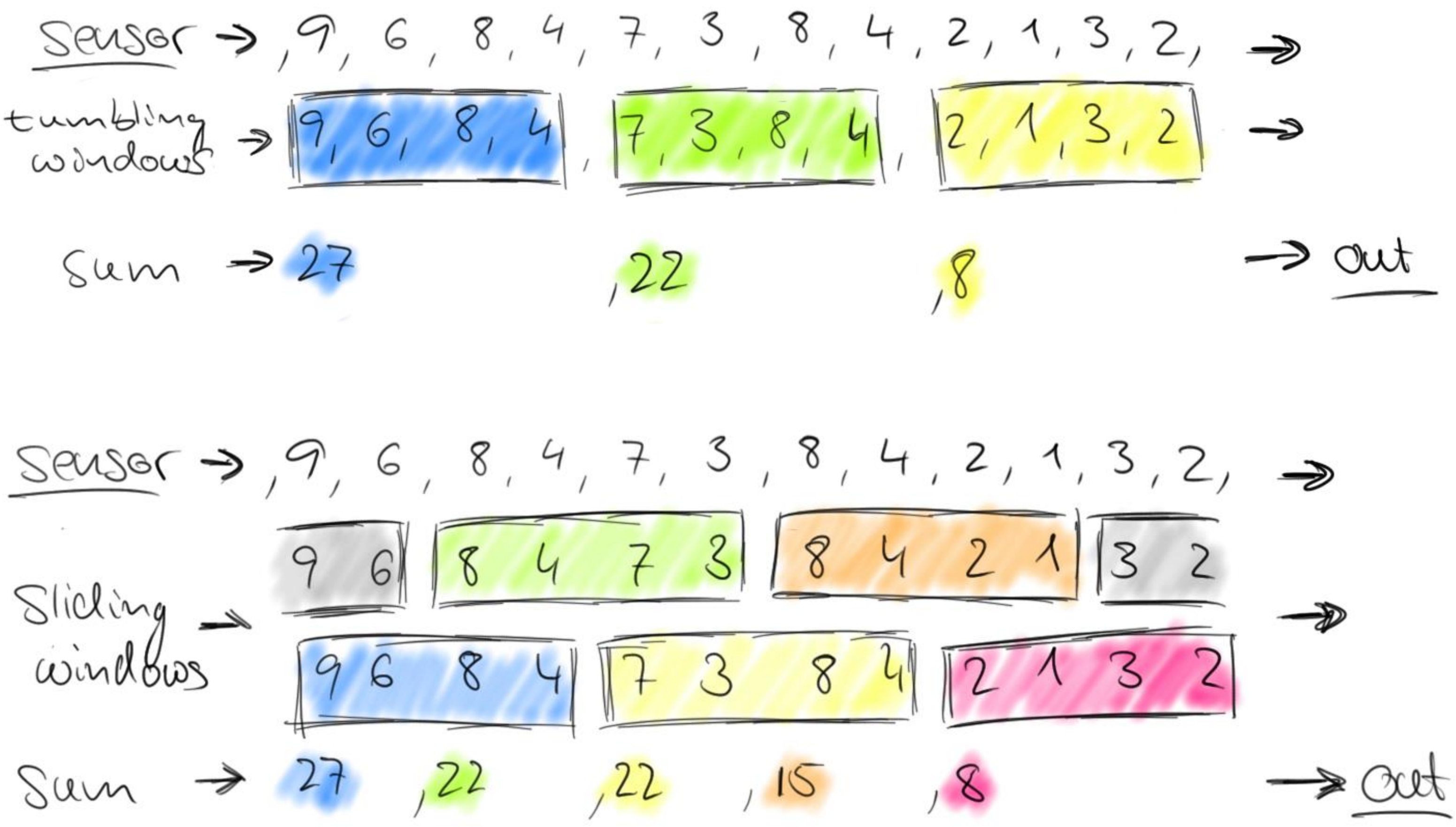
Windows and Sliding

sensor → ,9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, → out

sensor → ,9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →

rolling sum → ,57, 48, 42, 34, 30, 23, 20, 12, 8, 6, 5, 2, → out https://flink.apache.org/news/2015/12/04/Introducing-windows.html

Windows and Sliding

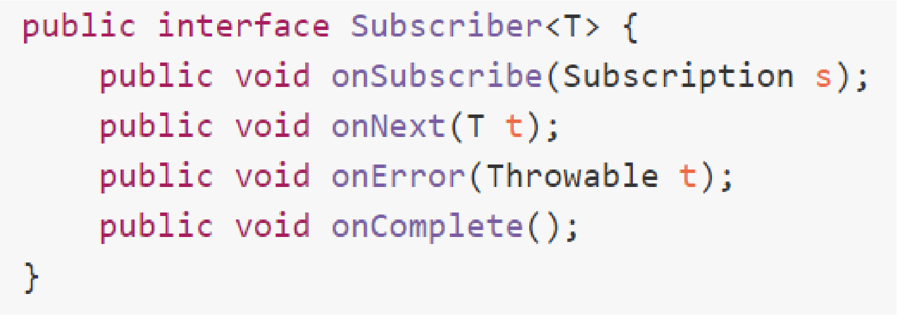

Sensor → ,9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →
tumbling windows → 9, 6, 8, 4 , 7, 3, 8, 4 , 2, 1, 3, 2 →
sum → 27 , 22 , 8 → out
Sensor → ,9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →
Sliding windows →
9 6
8 4 7 3
8 4 2 1
3 2
9 6 8 4
7 3 8 4
2 1 3 2
→
Sum → 27 ,22 ,22 ,15 ,8 → Out

Tumbling vs hopping vs sliding windows

- Tumbling
  - non-overlapping windows
- Hopping
  - overlapping windows with a fixed jump-ahead period
- Sliding
  - overlapping “continuous” window
  - all possible windows of a given size

Reactive Streams

- (Akka) Reactive Streams
  - good for streaming-like APIs



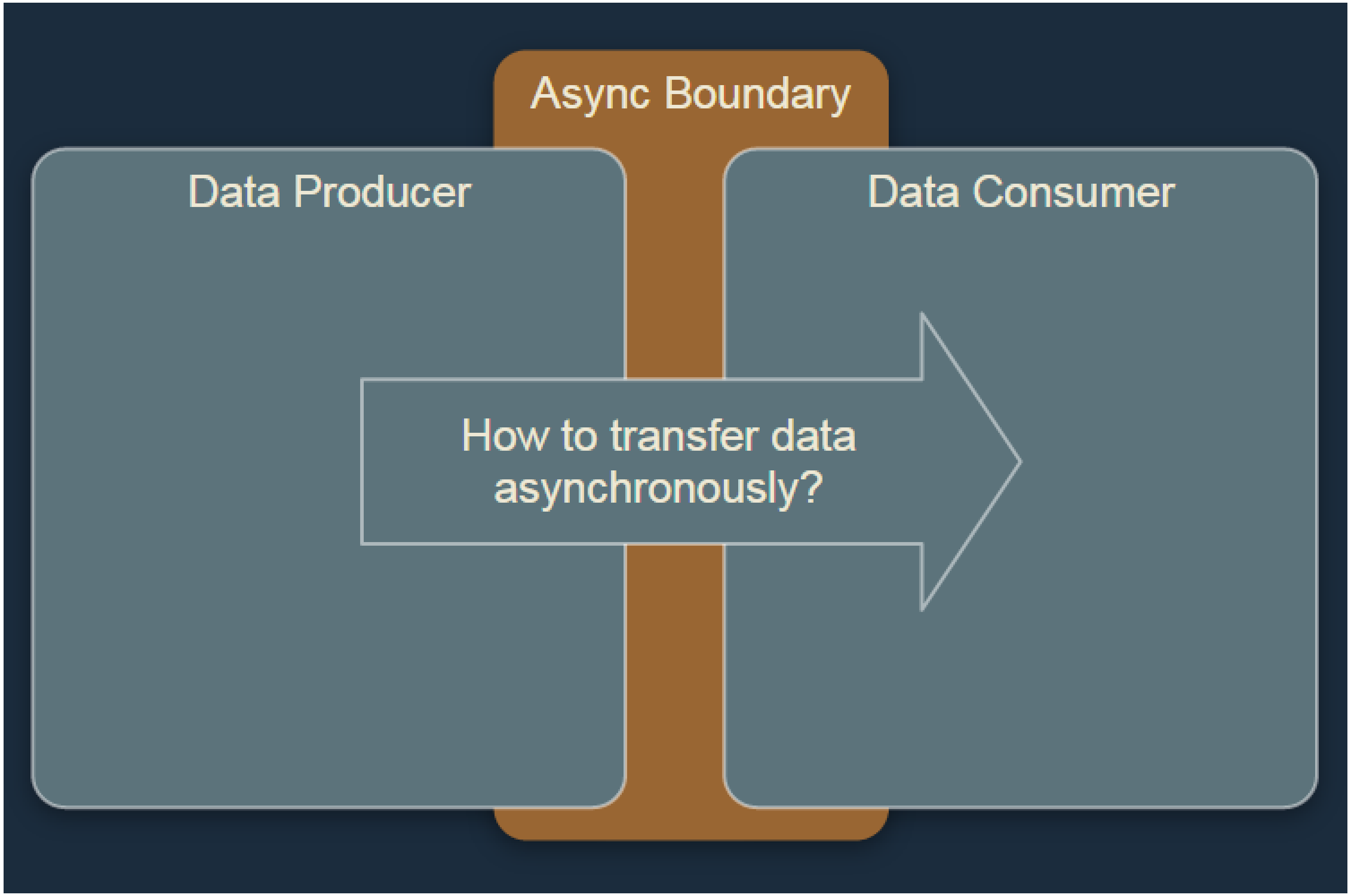
```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

- Goal:

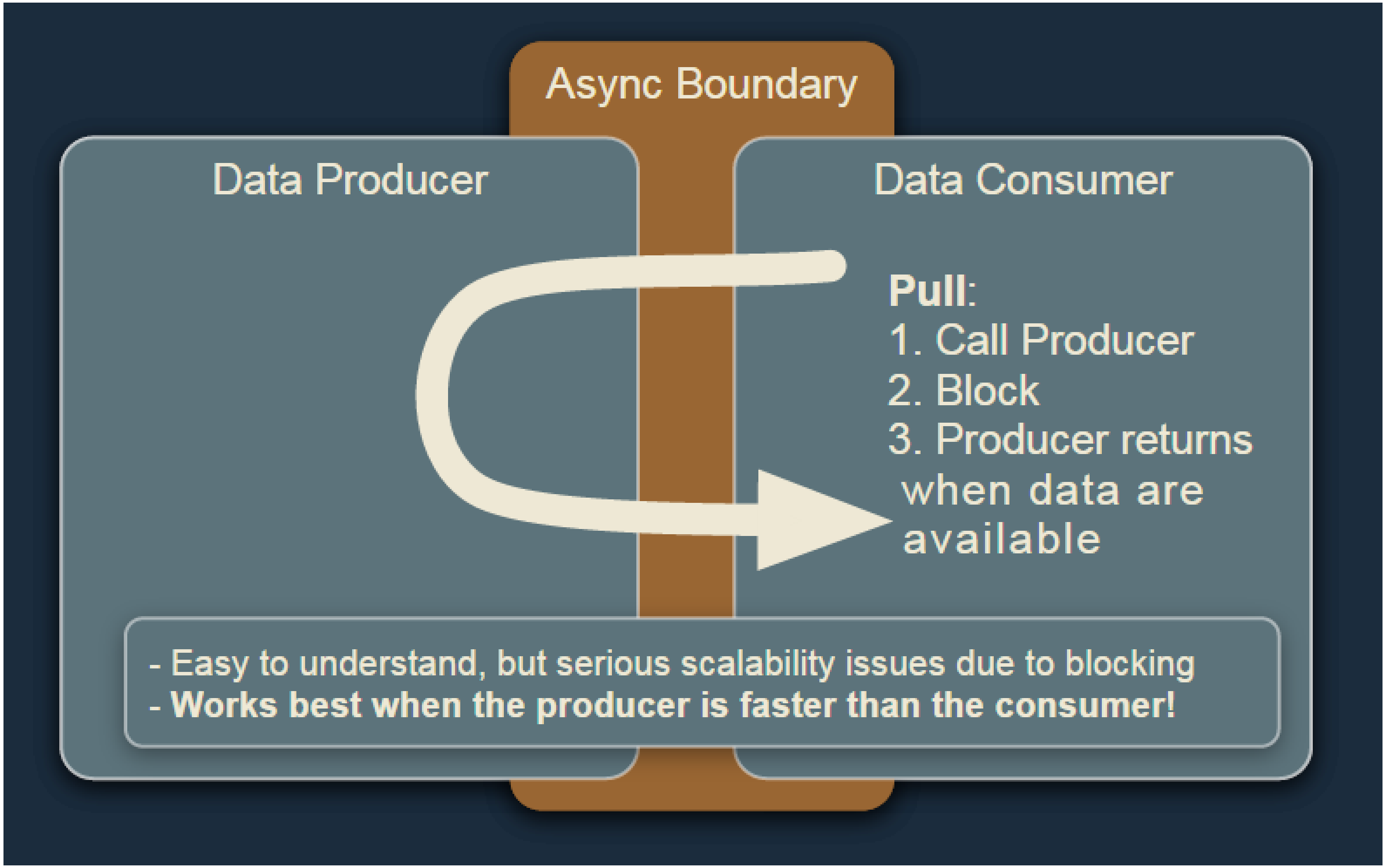
let the consumer explicitly tell the producer

"I am ready to accept new data"

Slides from Mathias Doenitz (spray.io, akka-http)



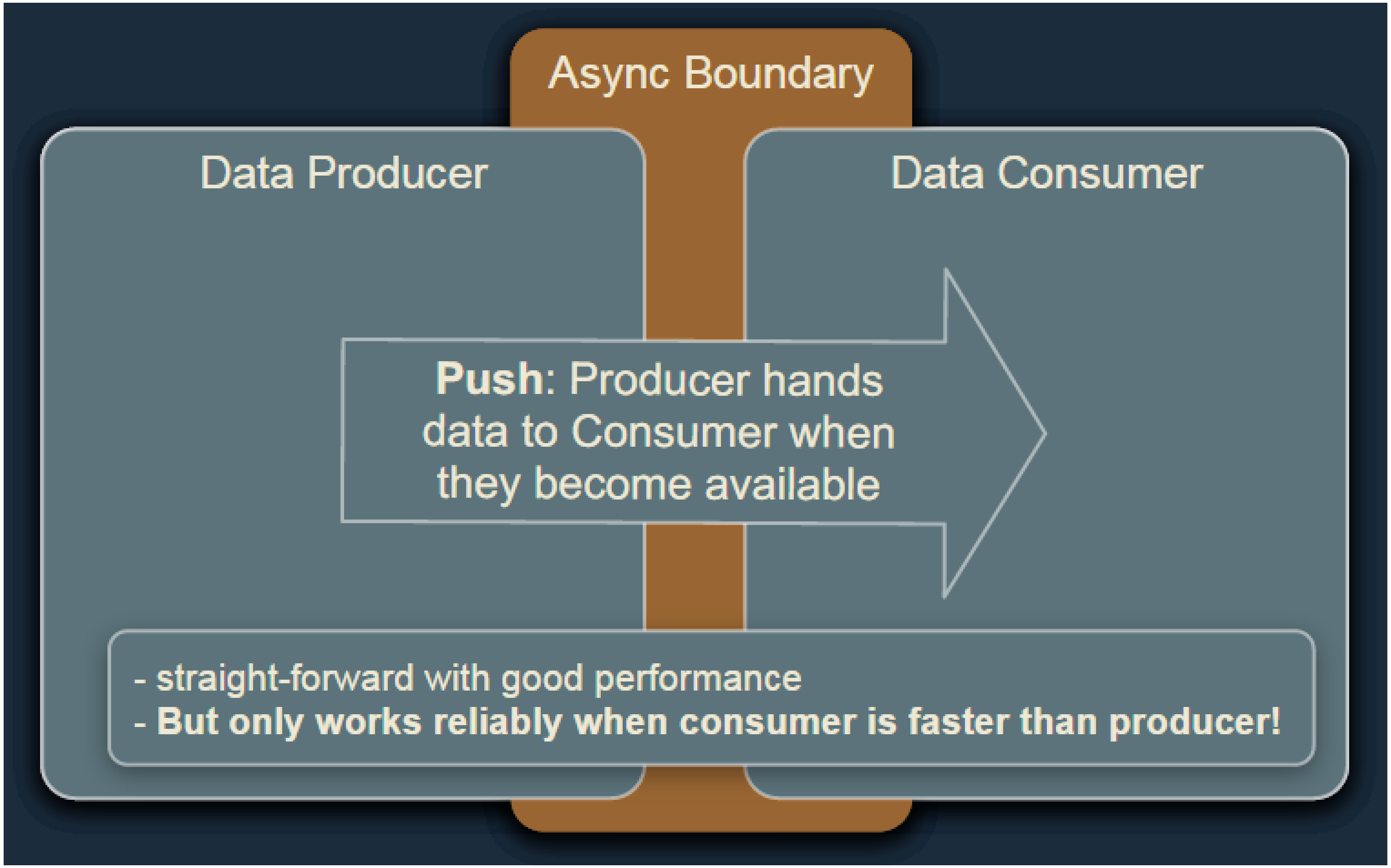
Slides from Mathias Doenitz (spray.io, akka-http)



Pull:
1. Call Producer
2. Block
3. Producer returns when data are available

- Easy to understand, but serious scalability issues due to blocking
- **Works best when the producer is faster than the consumer!**

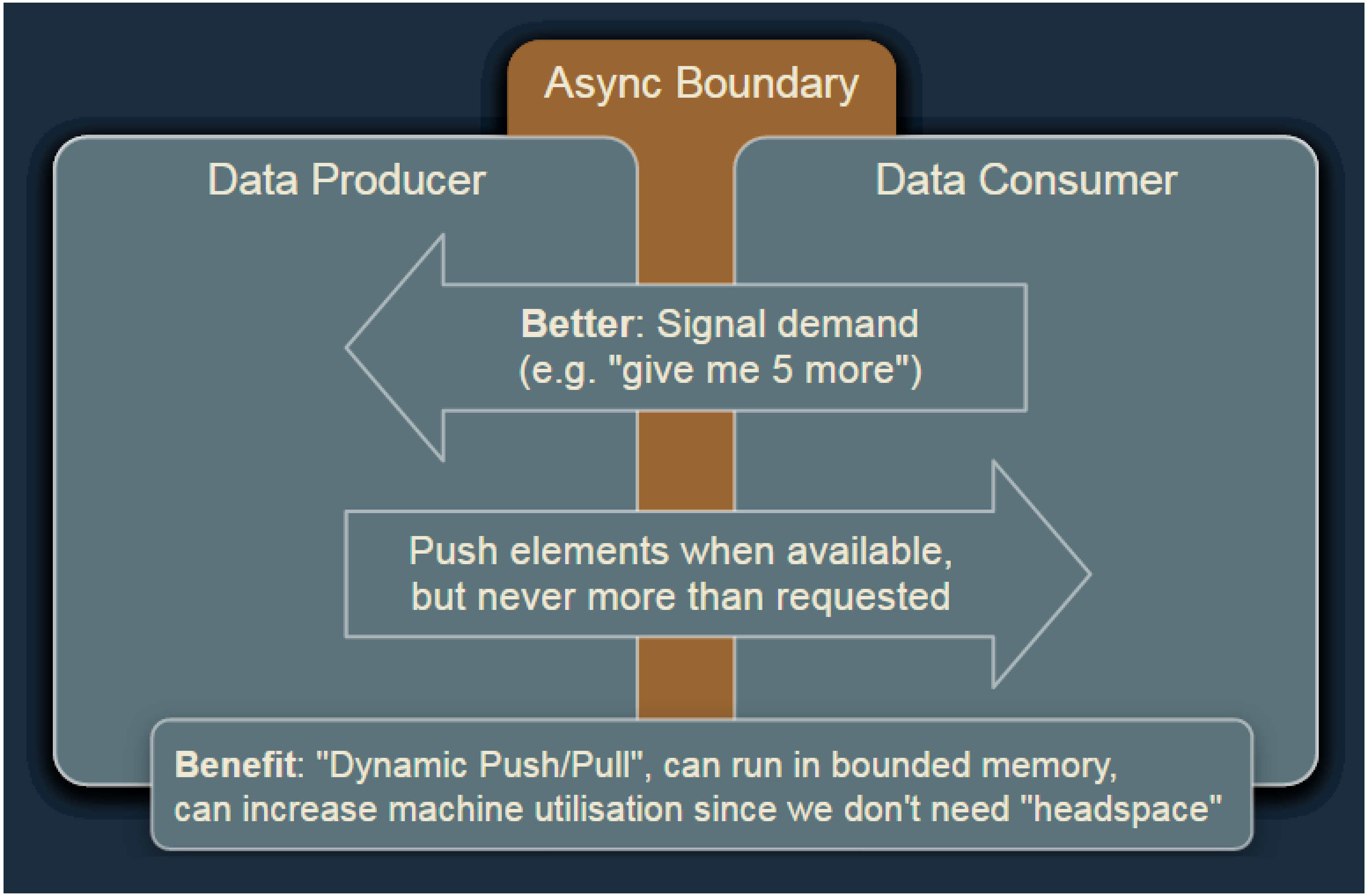
Slides from Mathias Doenitz (spray.io, akka-http)



Push: Producer hands data to Consumer when they become available

- straight-forward with good performance
- **But only works reliably when consumer is faster than producer!**

Slides from Mathias Doenitz (spray.io, akka-http)



Async Boundary

Data Producer

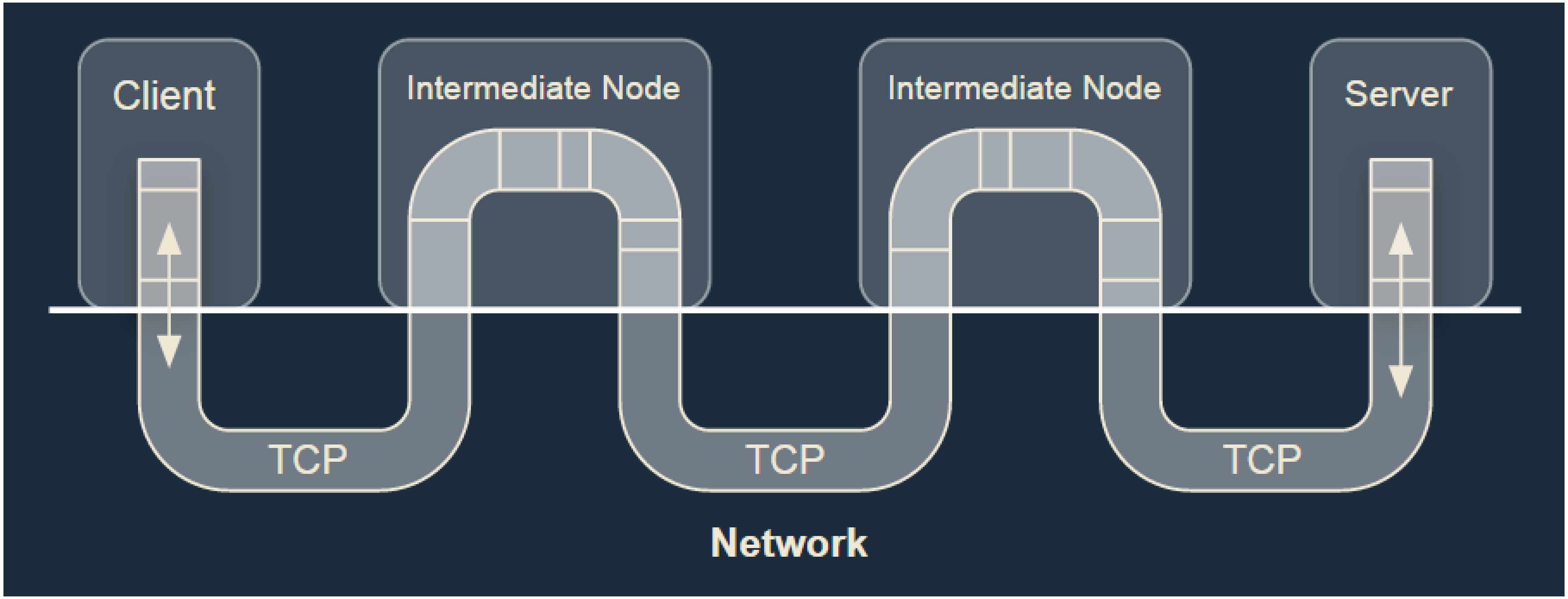
Data Consumer

Better: Signal demand
(e.g. "give me 5 more")

Push elements when available,
but never more than requested

Benefit: "Dynamic Push/Pull", can run in bounded memory,
can increase machine utilisation since we don't need "headspace"

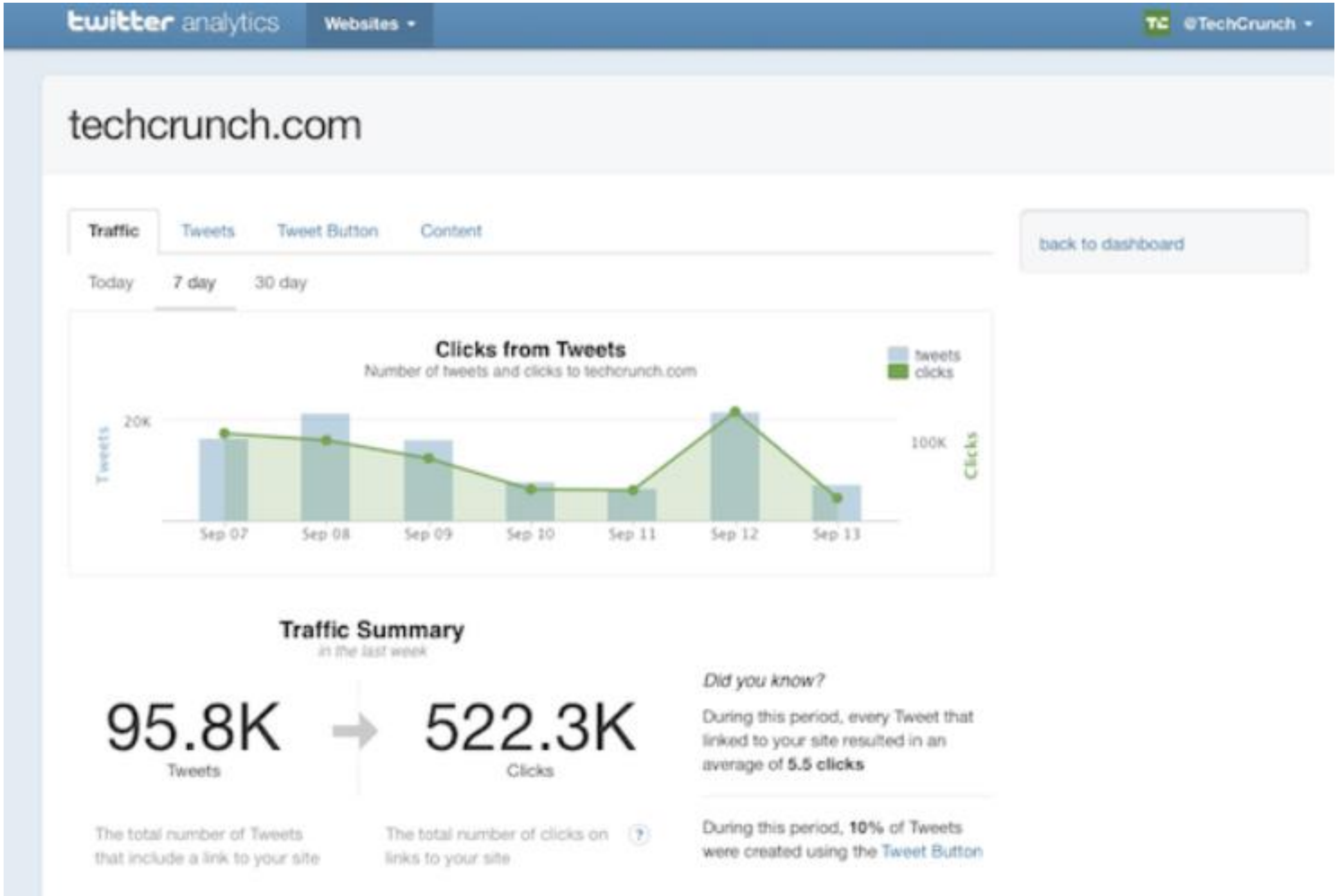
Slides from Mathias Doenitz (spray.io, akka-http)



full resource utilization if used across the whole pipeline!

for Storm slides, I re-use slides from a presentation of Nathan Marz, the creator of Storm

Storm @ Twitter



twitter analytics | Websites ▾ | TC @TechCrunch ▾

techcrunch.com

Traffic | Tweets | Tweet Button | Content

back to dashboard

Today | 7 day | 30 day

Clicks from Tweets

Number of tweets and clicks to techcrunch.com

Traffic Summary

in the last week

95.8K Tweets → 522.3K Clicks

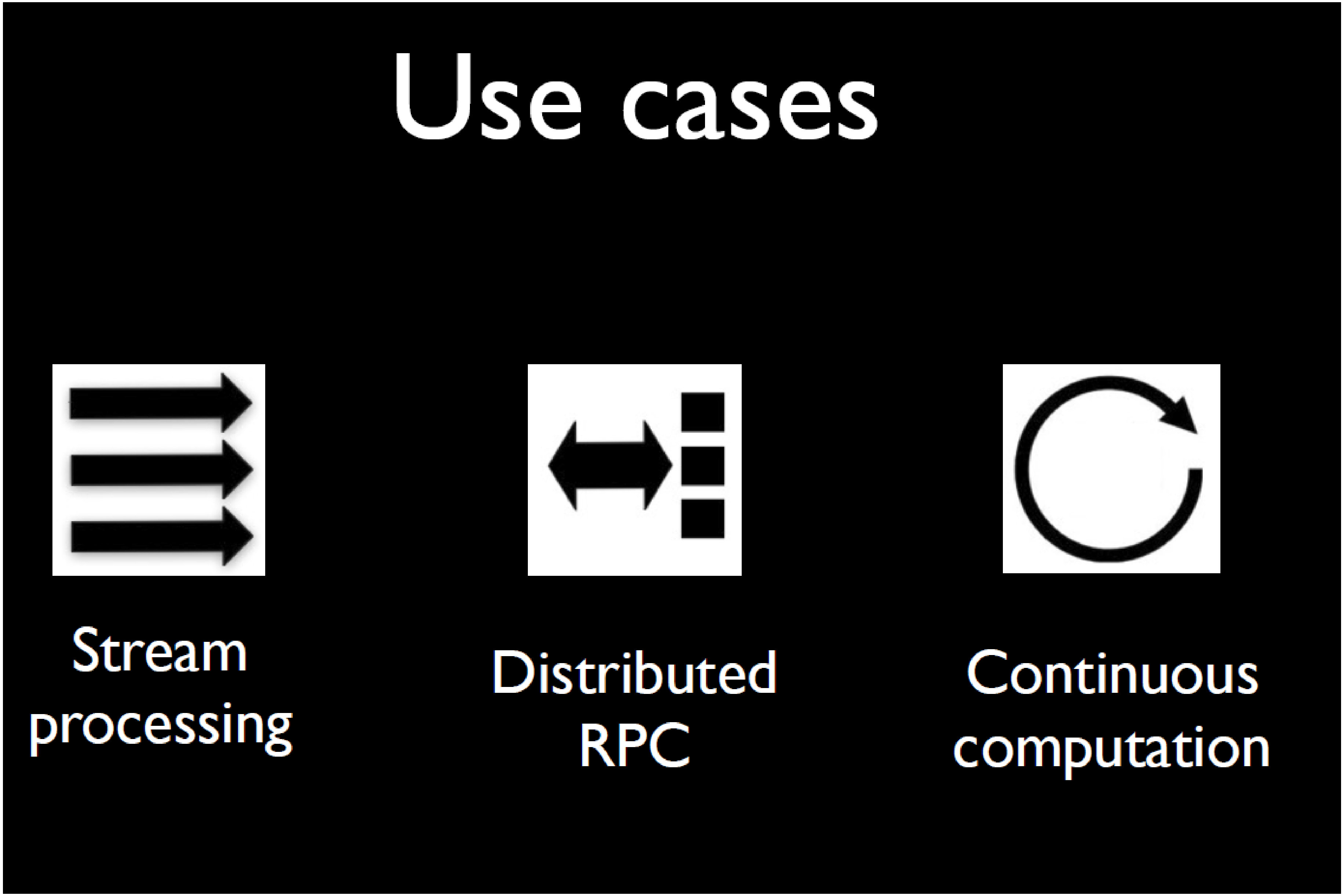
The total number of Tweets that include a link to your site

The total number of clicks on links to your site ?

Did you know?

During this period, every Tweet that linked to your site resulted in an average of **5.5 clicks**

During this period, **10%** of Tweets were created using the Tweet Button



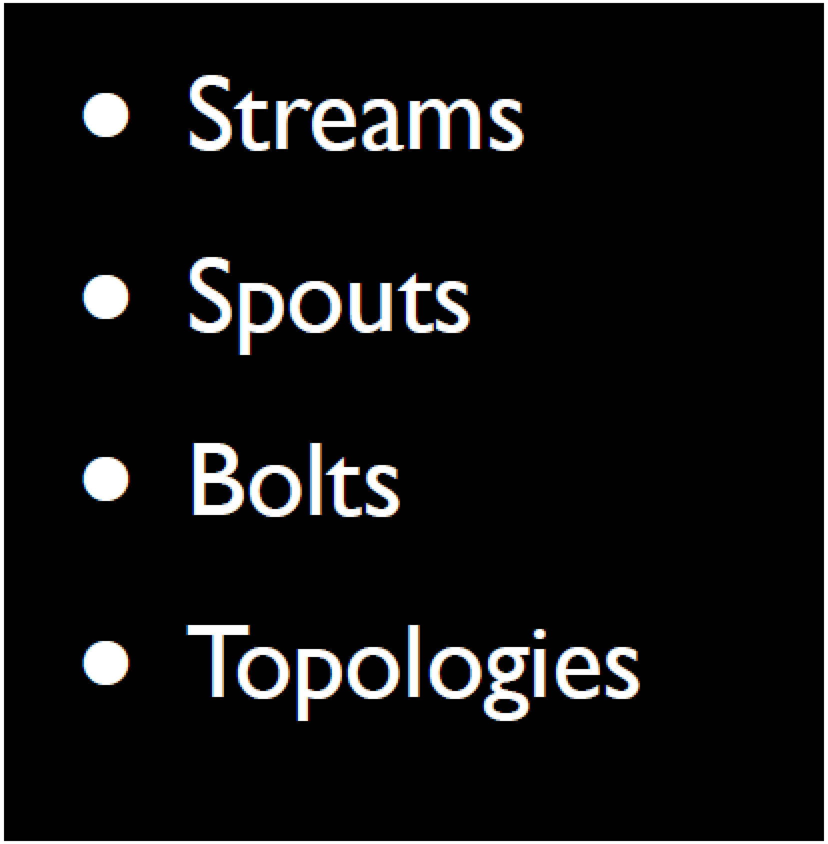
Use cases

Stream processing

Distributed RPC

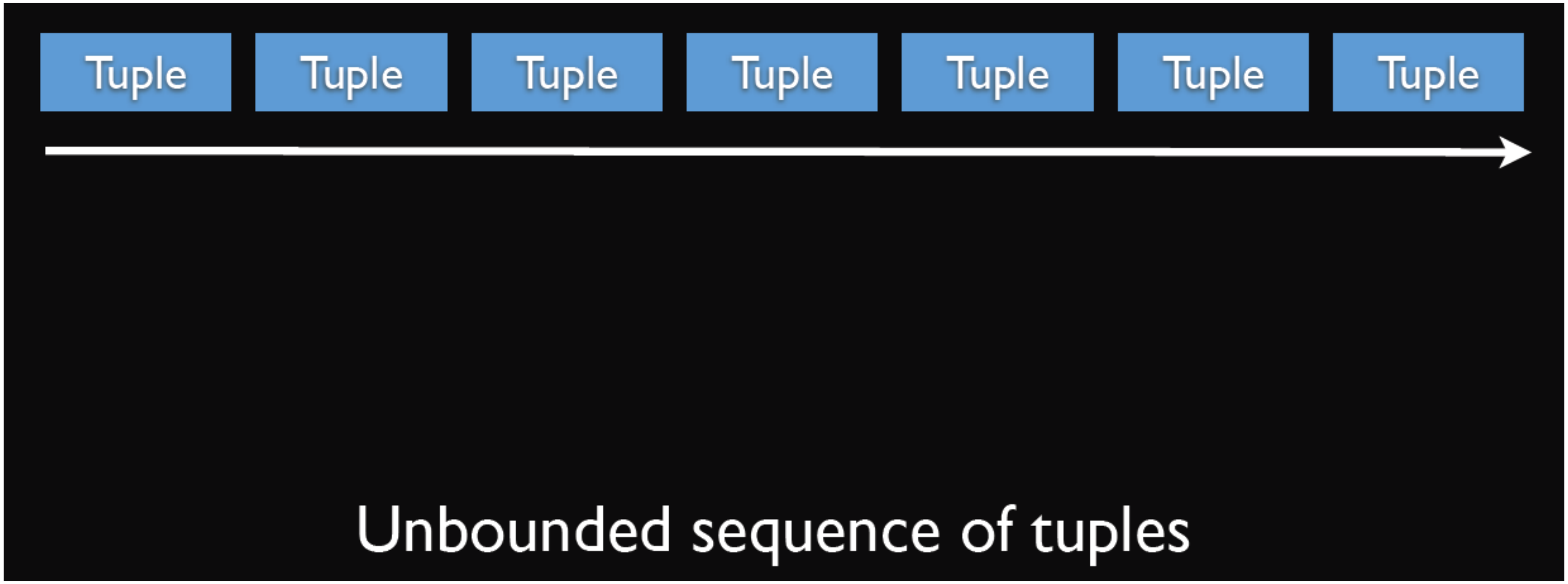
Continuous computation

Concepts



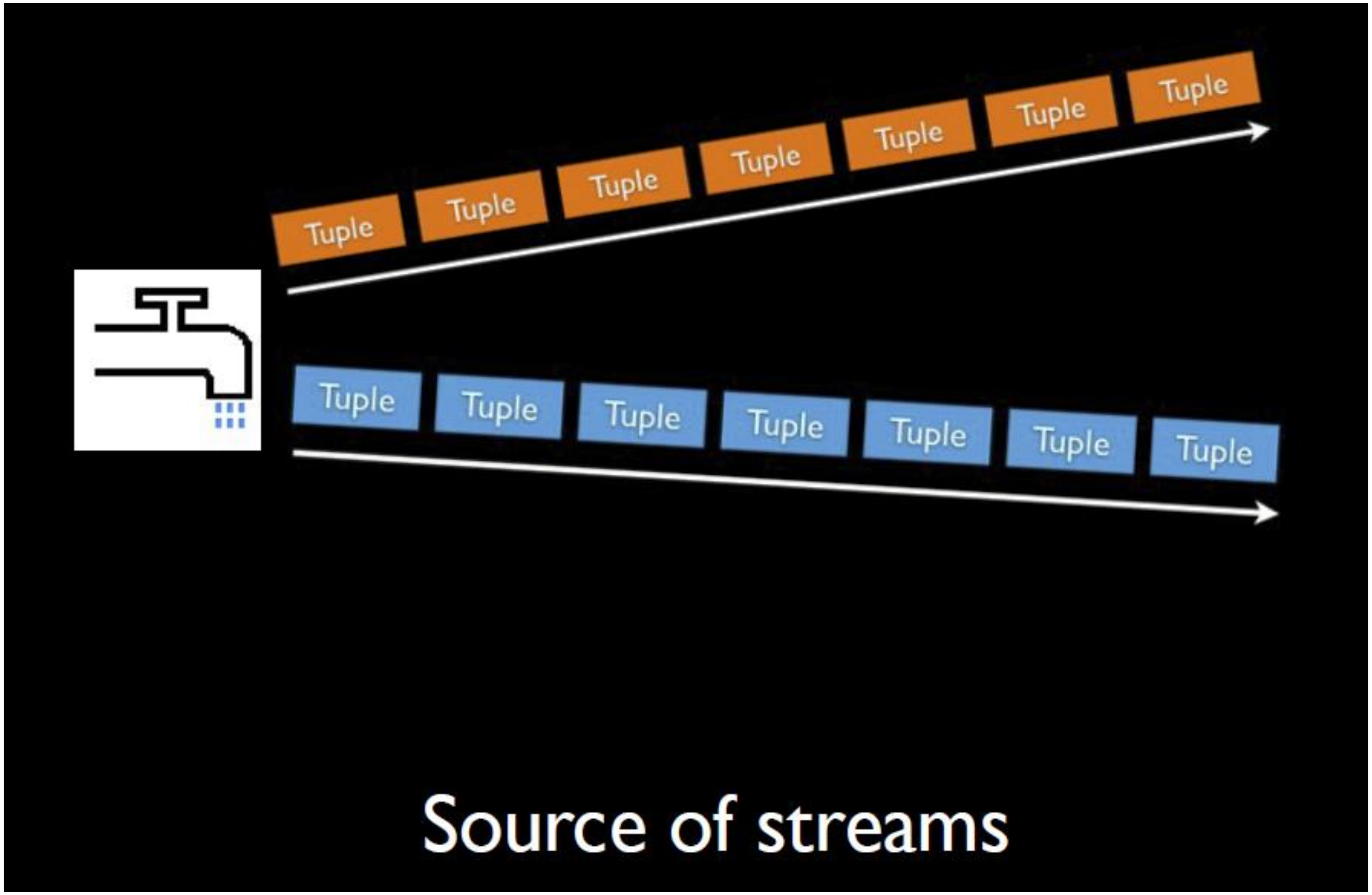
- Streams
- Spouts
- Bolts
- Topologies

Streams



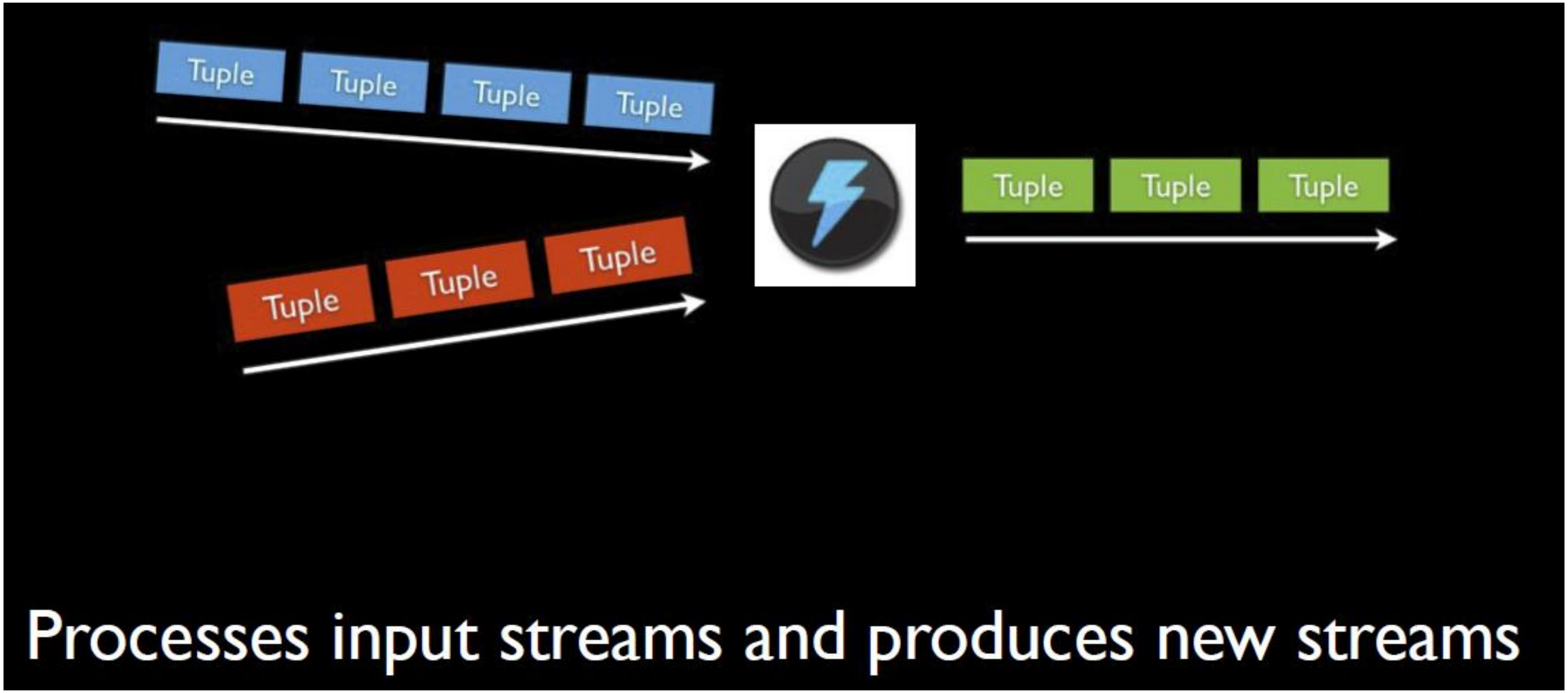
Unbounded sequence of tuples

Spouts



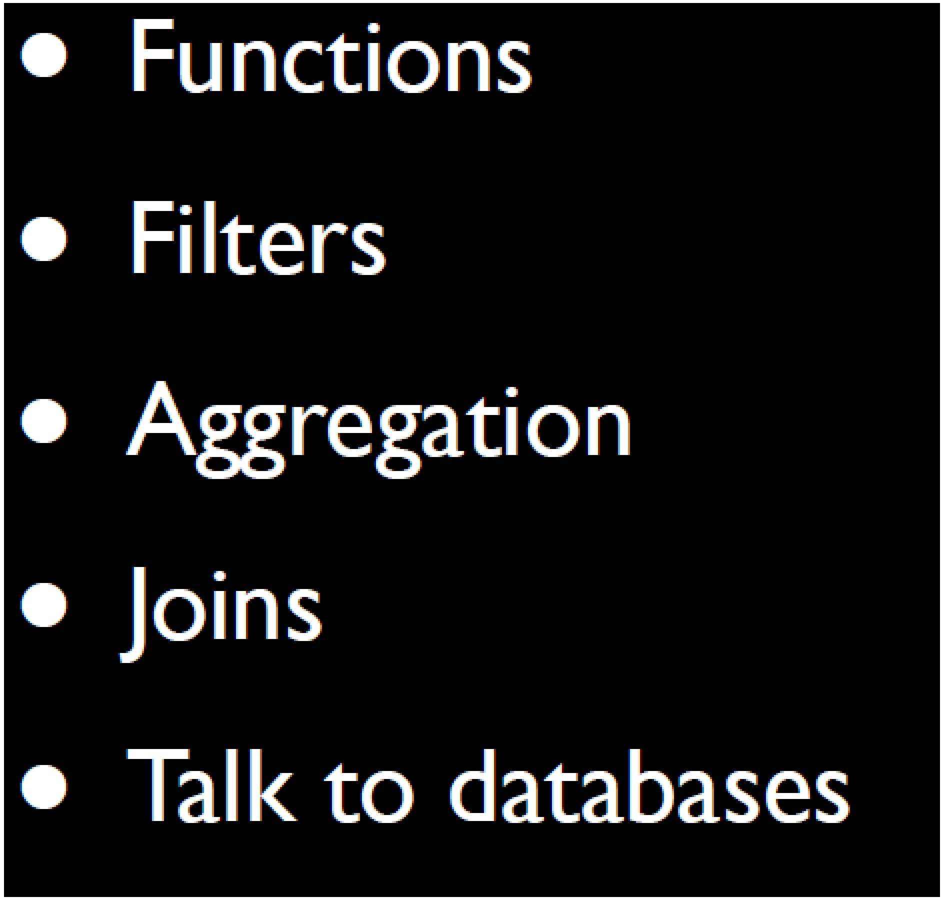
Source of streams

Bolts



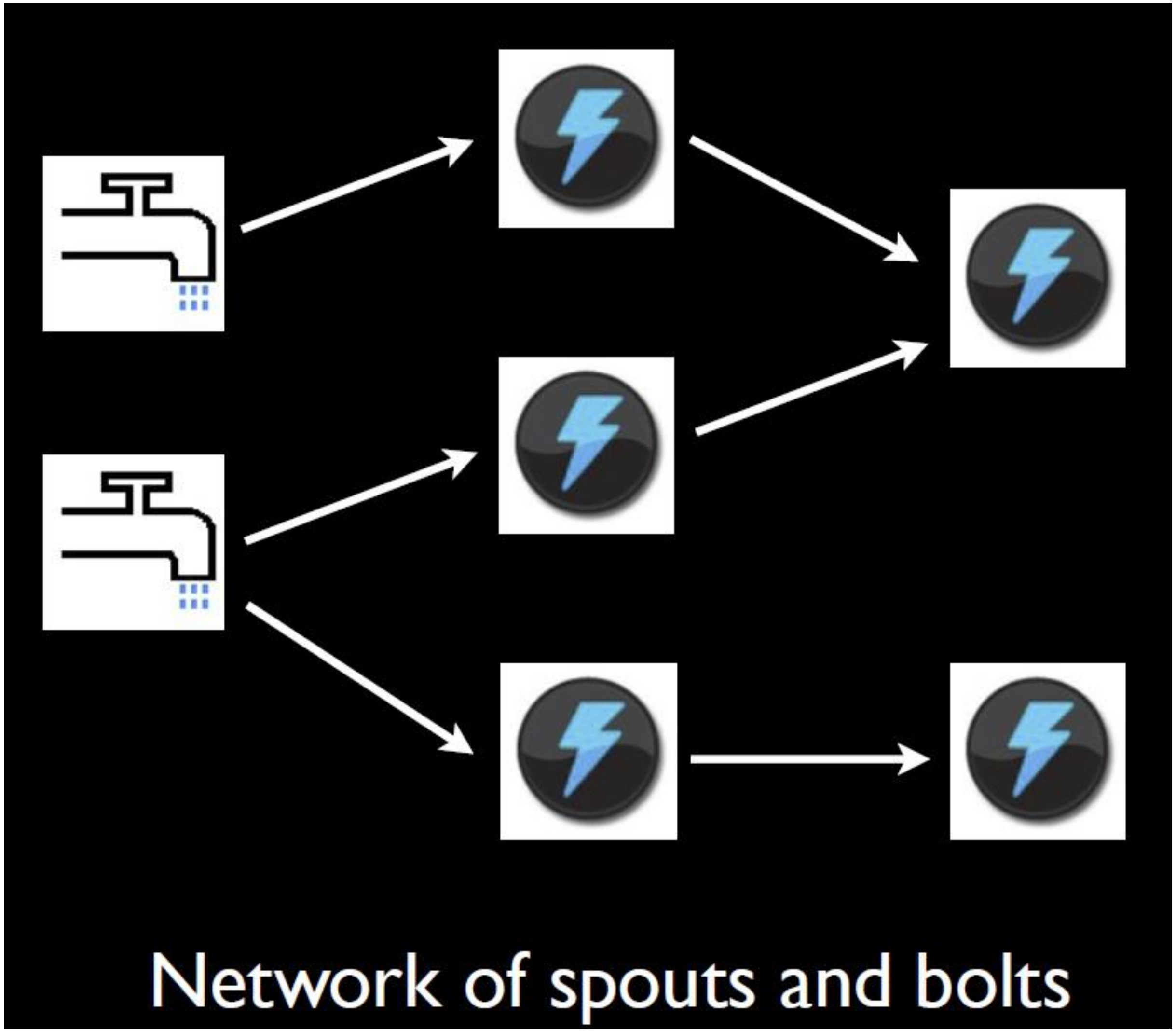
Processes input streams and produces new streams

Bolts

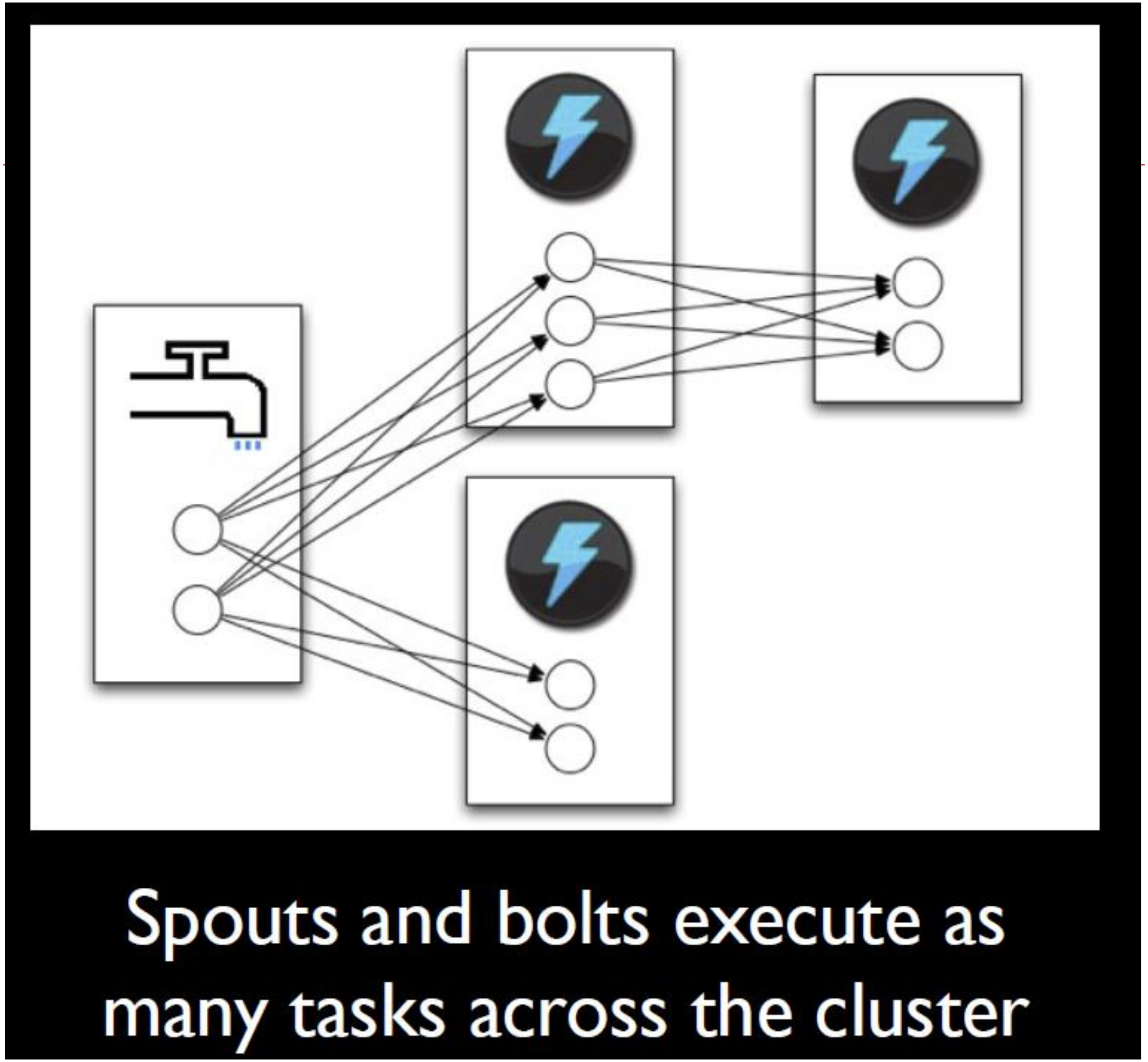


- Functions
- Filters
- Aggregation
- Joins
- Talk to databases

Topology

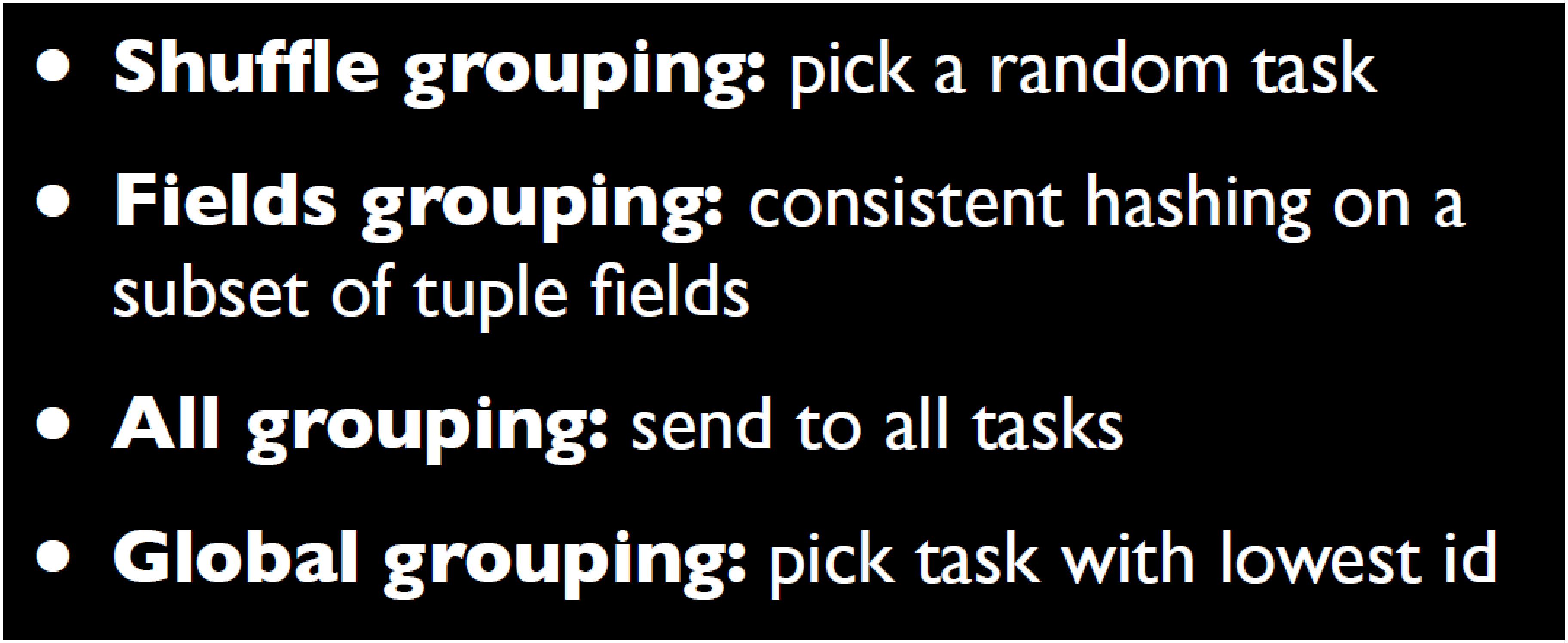


Tasks



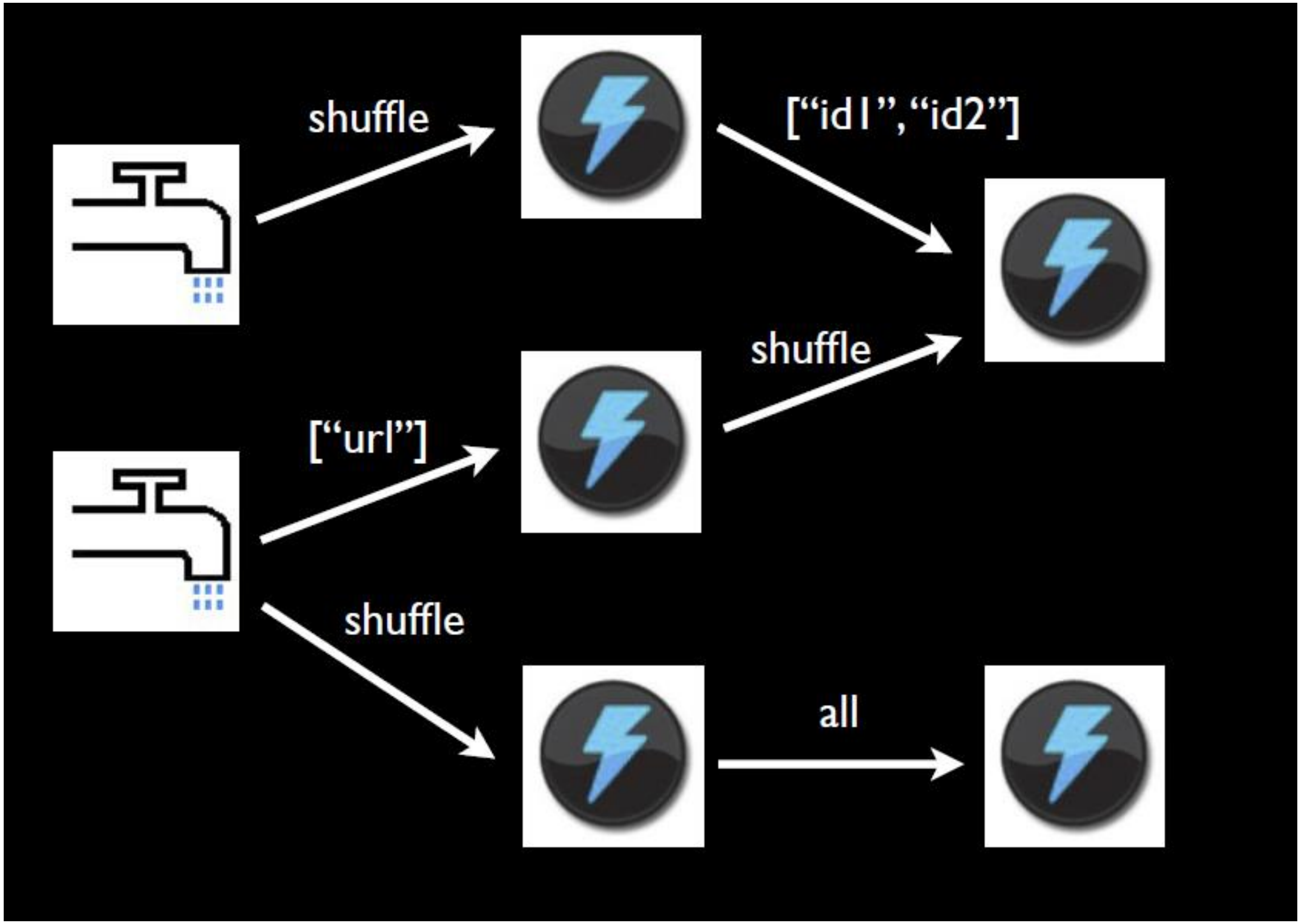
Spouts and bolts execute as many tasks across the cluster

Stream grouping

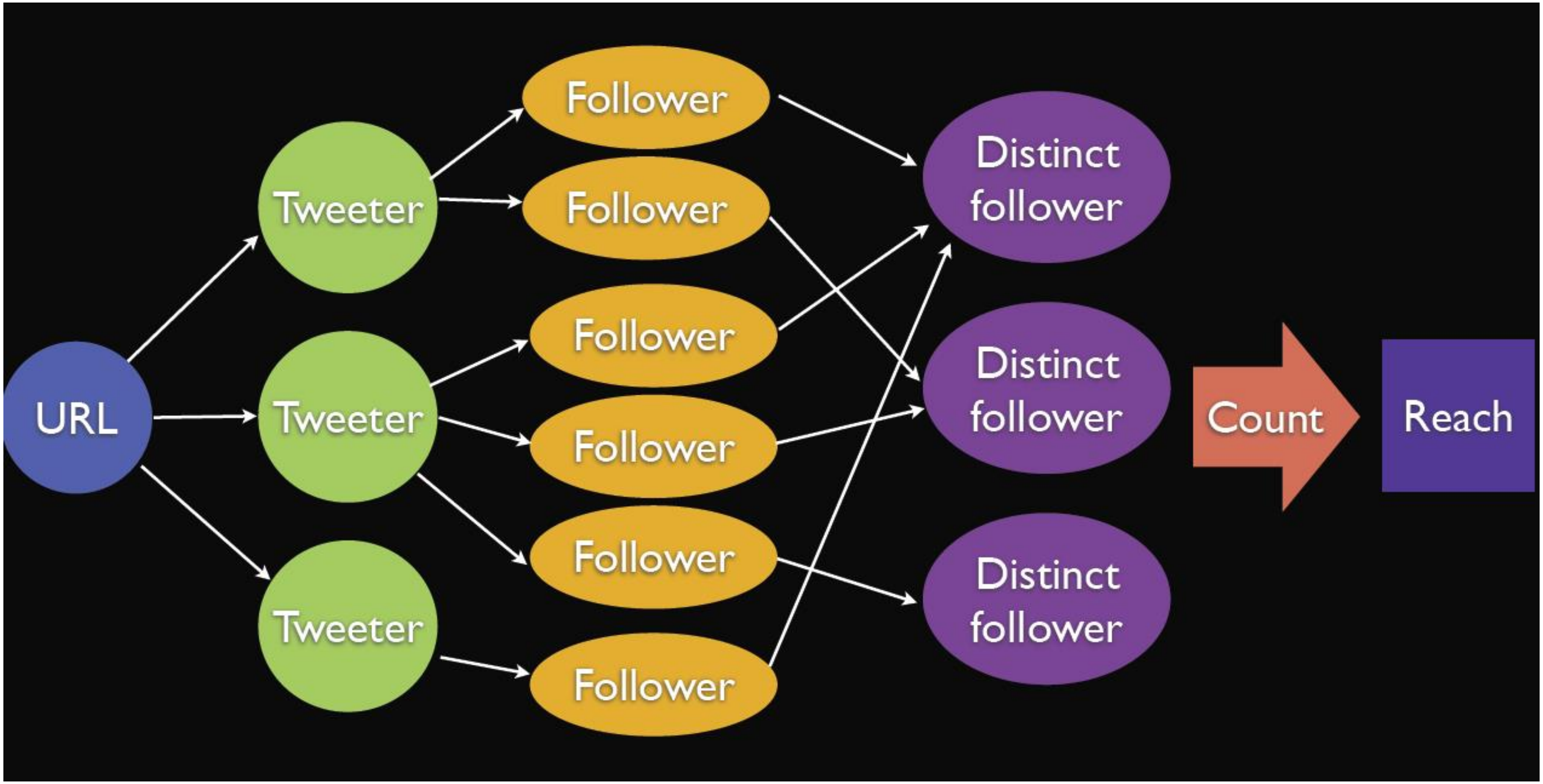


- **Shuffle grouping:** pick a random task
- **Fields grouping:** consistent hashing on a subset of tuple fields
- **All grouping:** send to all tasks
- **Global grouping:** pick task with lowest id

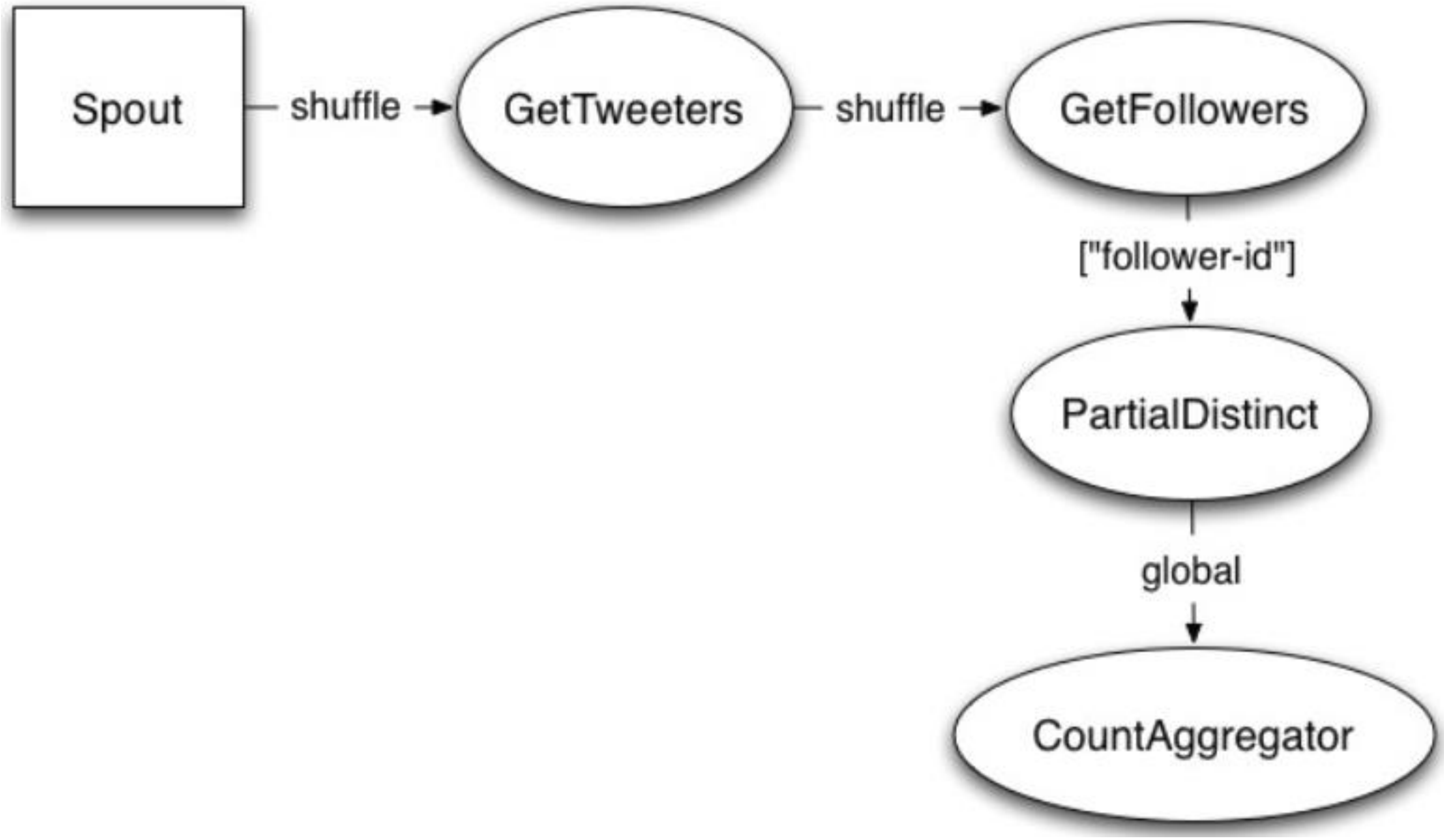
Filtering and grouping



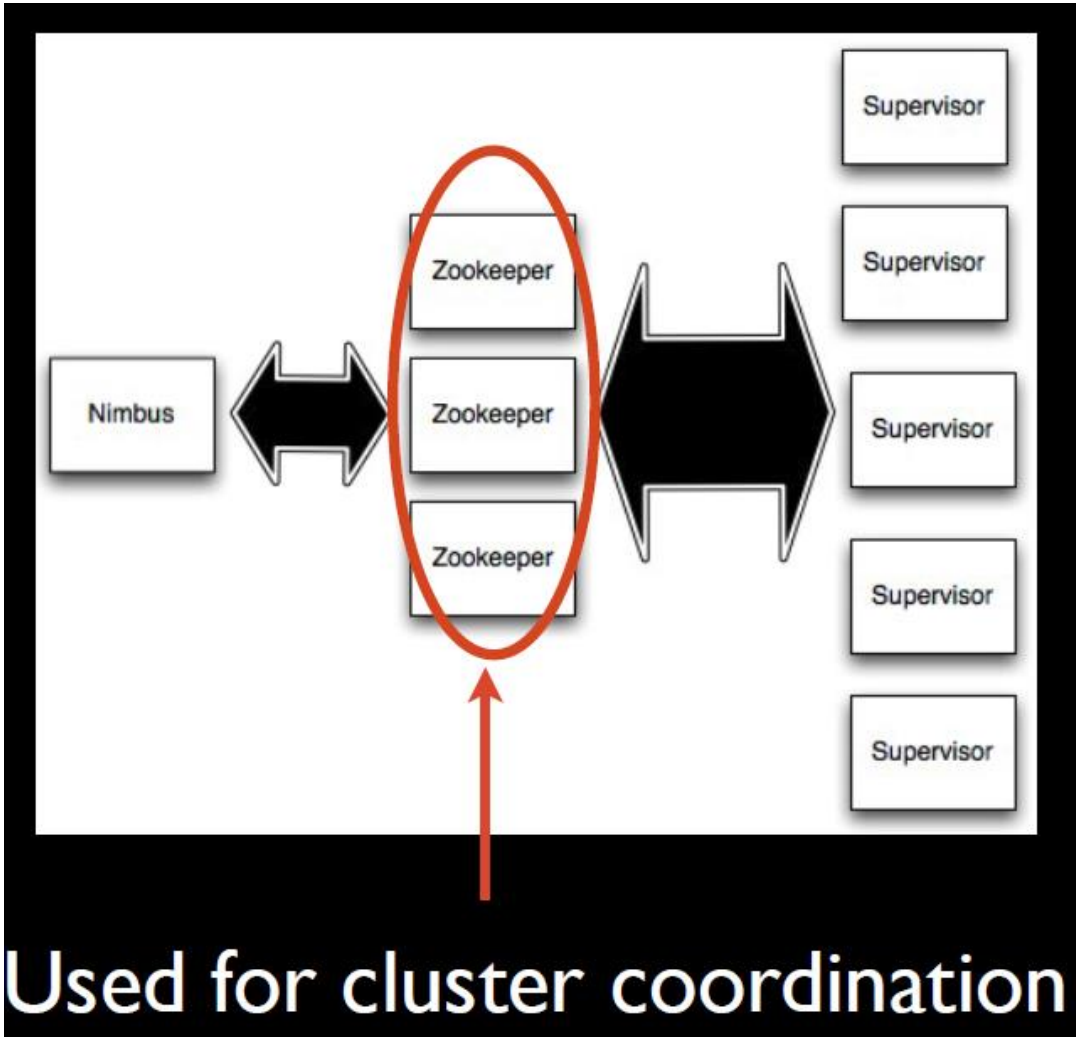
Sample Application



Filtering and grouping



Cluster Coordination



Used for cluster coordination