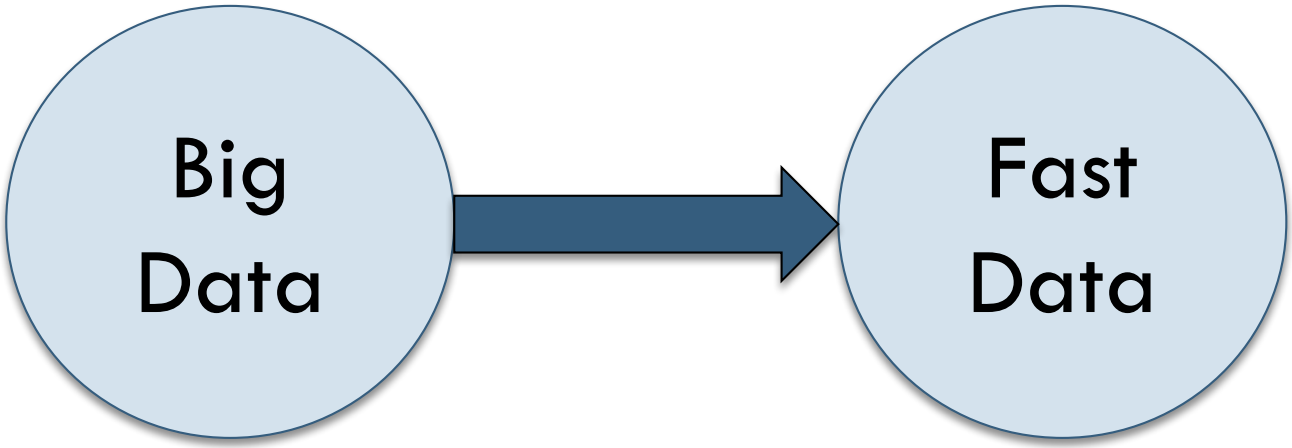
INTRODUCTION TO SPARK WITH SCALA

Spark Streaming

Agenda

- Streaming Processing
- Spark Streaming Concepts
- Working with DStream APIs
- Spark Streaming Fault Tolerance
- Spark Streaming Application

Streaming Processing

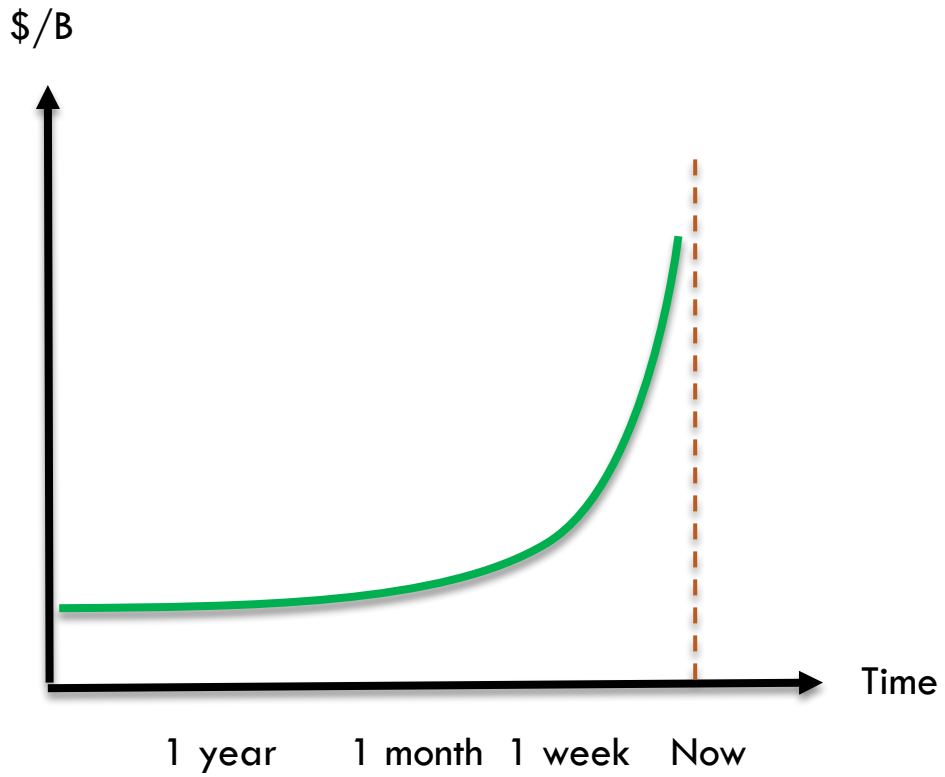


Data in motion has equal or greater value than "historical" data

Streaming Processing

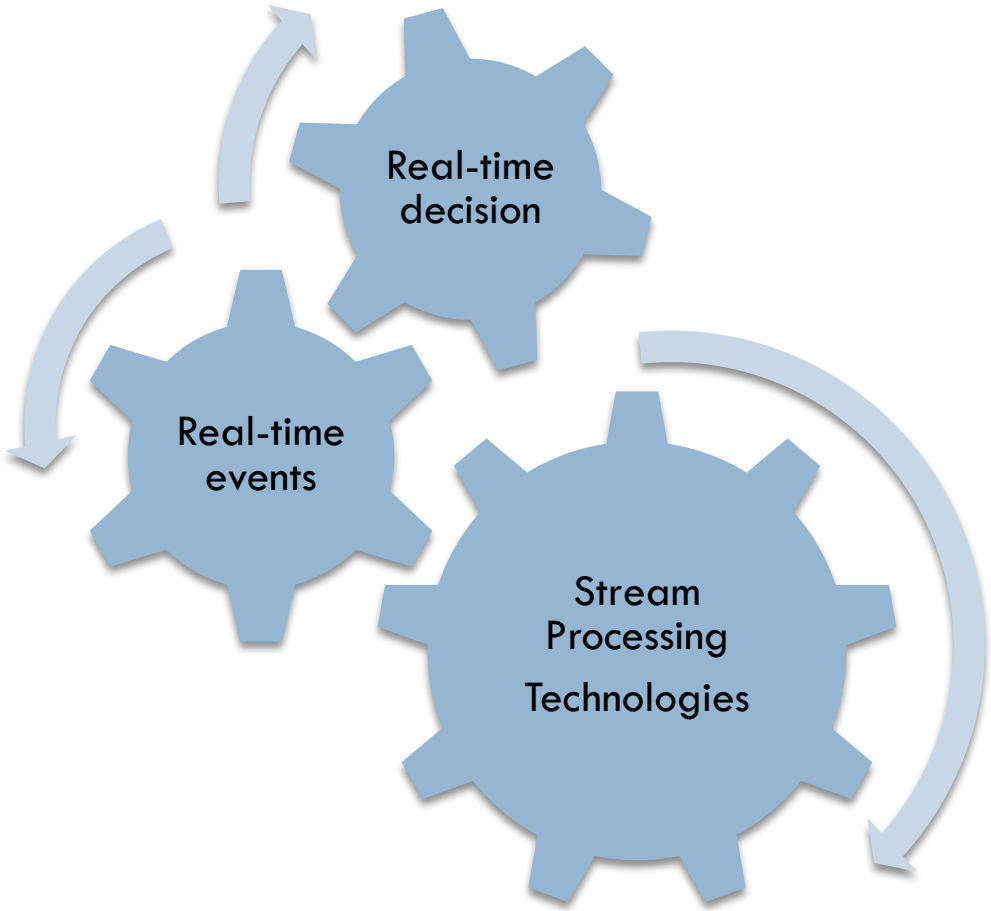
Value of Data

- Recent data is valuable
  - if act on it
- Historical + new data
  - More valuable
- Technologies are maturing
  - To enable this

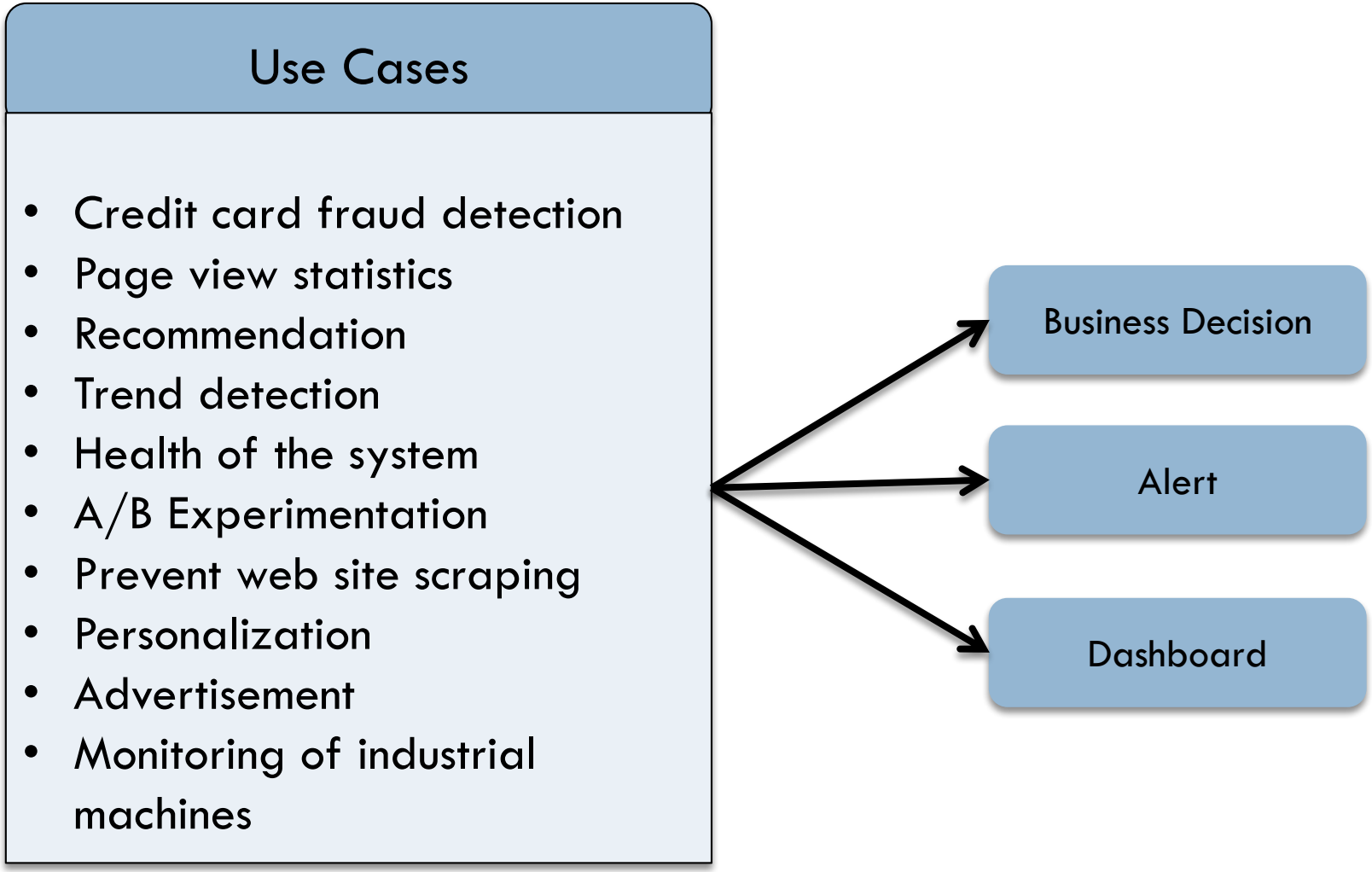


Streaming Processing

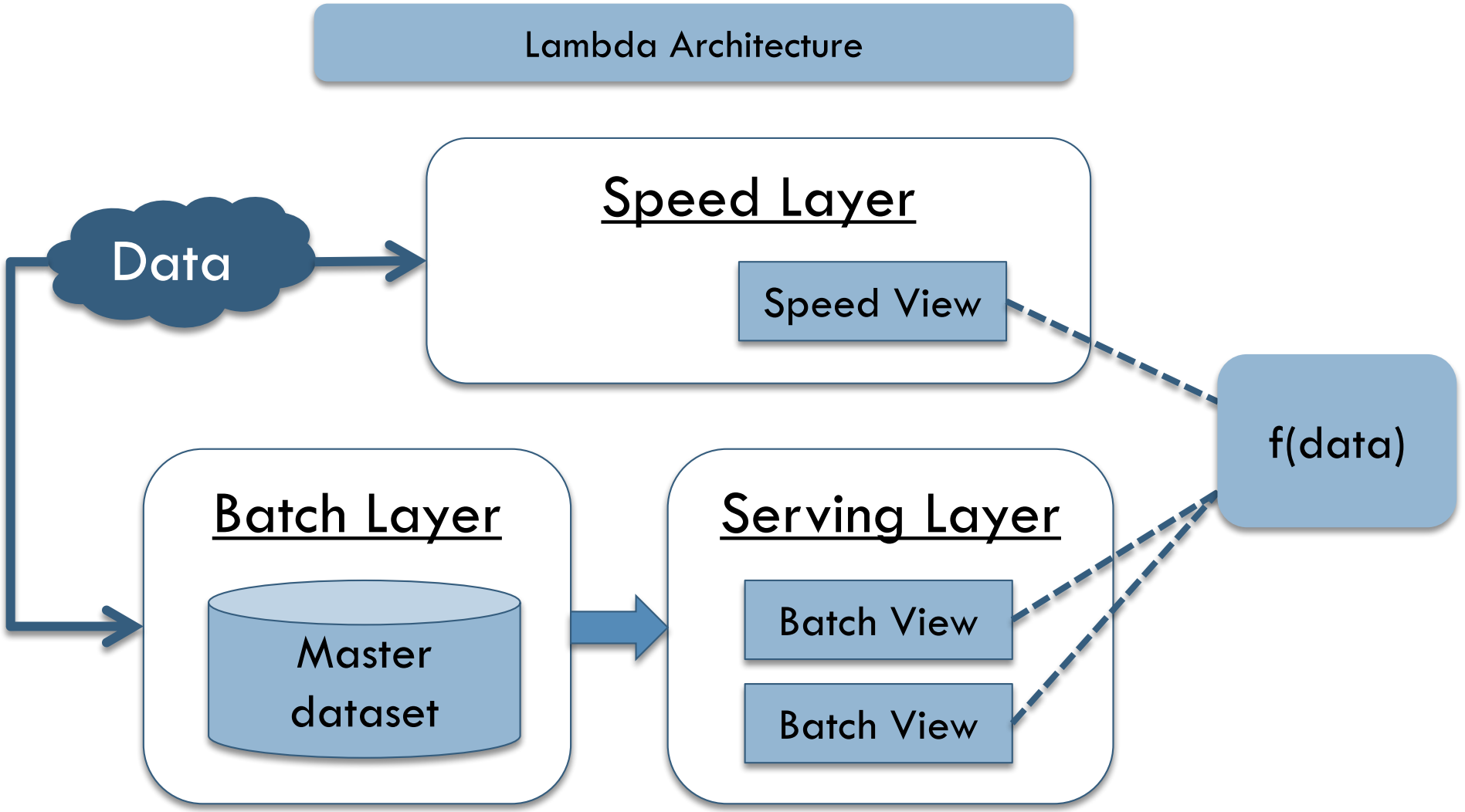
The Rise of Real Time



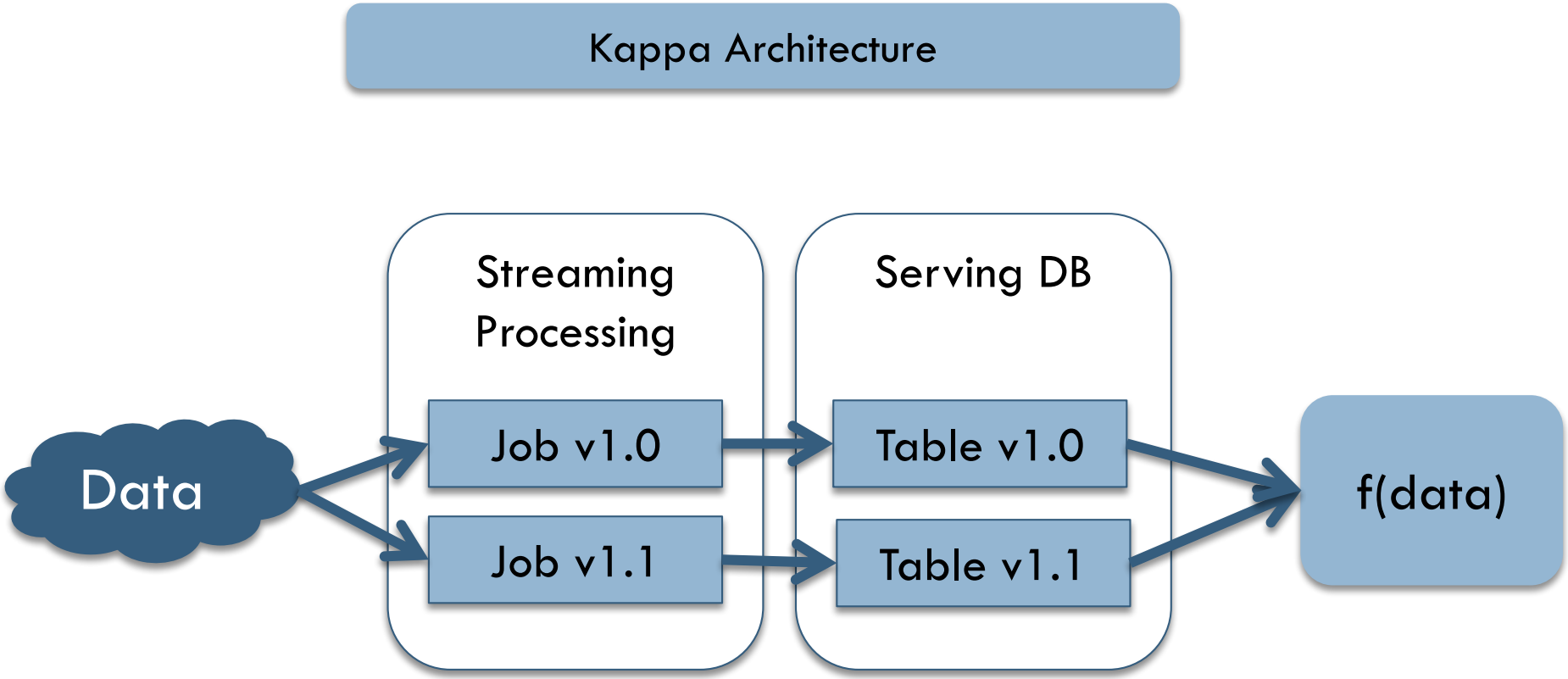
Streaming Processing



Streaming Processing



Streaming Processing

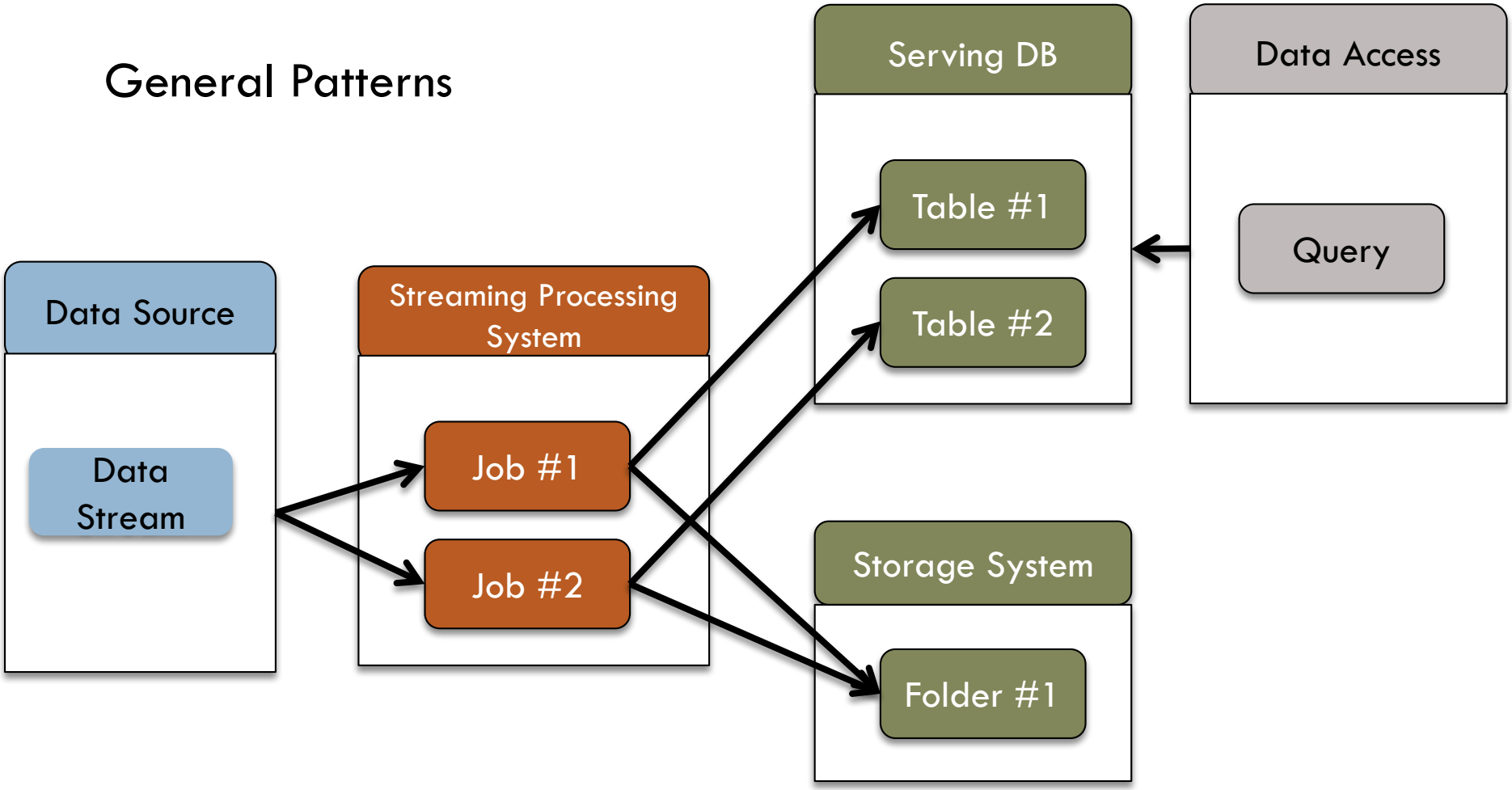


Streaming Processing

Continued processing on data that continuously produced

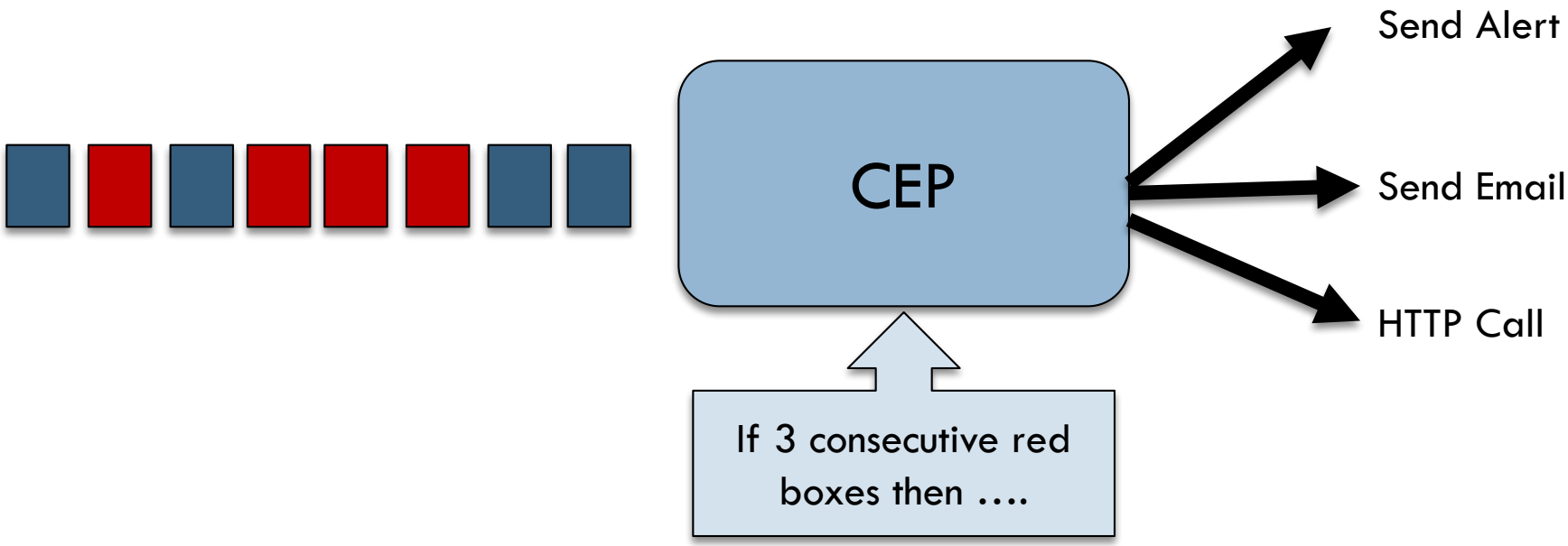
Streaming Processing

General Patterns



Streaming Processing

- Complex Event Processing
  - Detect complex event patterns in streams of data



Streaming Processing

- Challenges
  - Low latency & Scalable
  - Fault tolerance
  - Adaptive
  - Unpredictable incoming data patterns
  - Correlating multiple streams
  - Integrate with batching and interactive processing
  - Out of sequence or late late events
  - Stateful processing
    - Traditional model – in-memory tracking of mutable state
    - Must be fault tolerant

Streaming Processing

Message processing semantics

Exactly once => 1 only

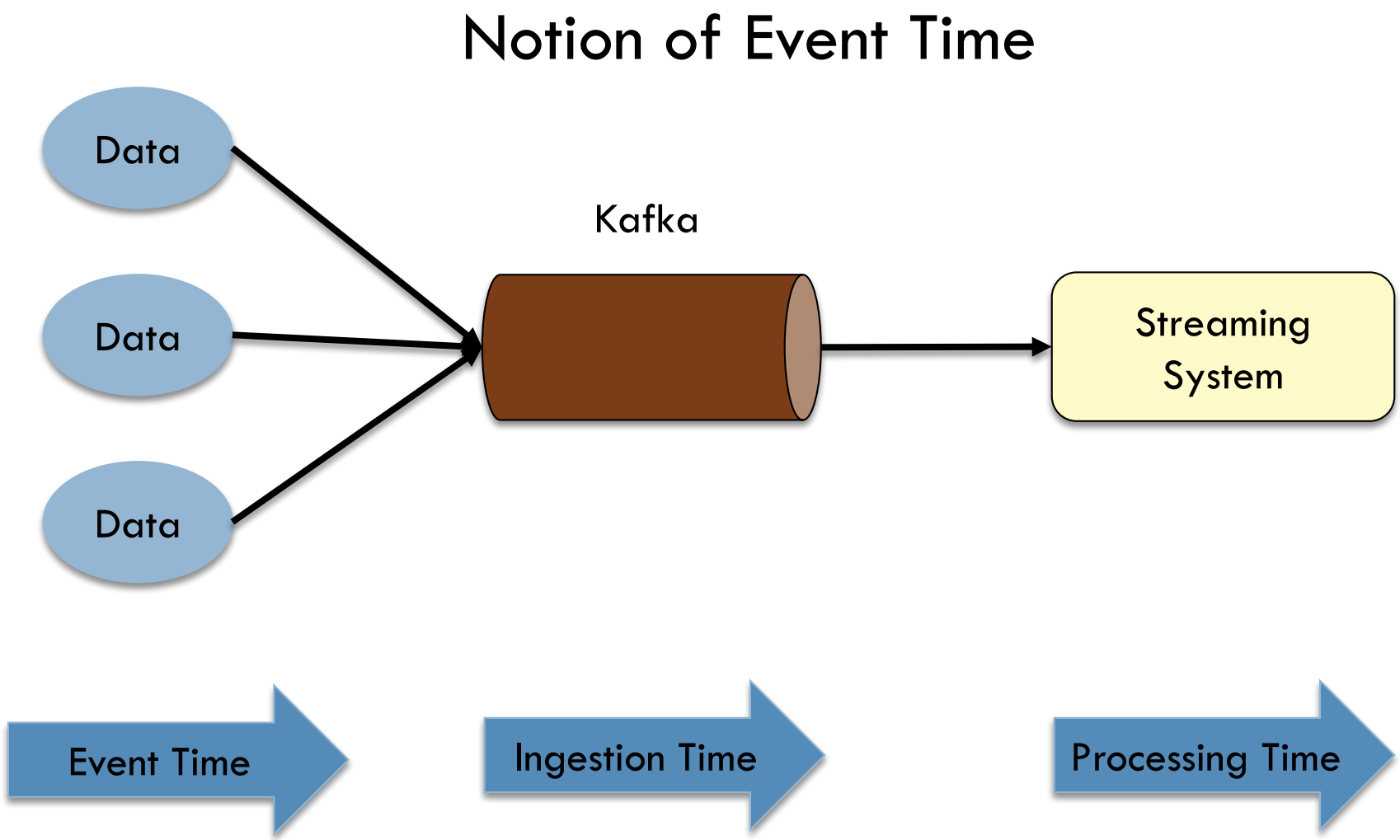
At least once => 1 or n

At most once => 1 or 0

Strongest

Weak

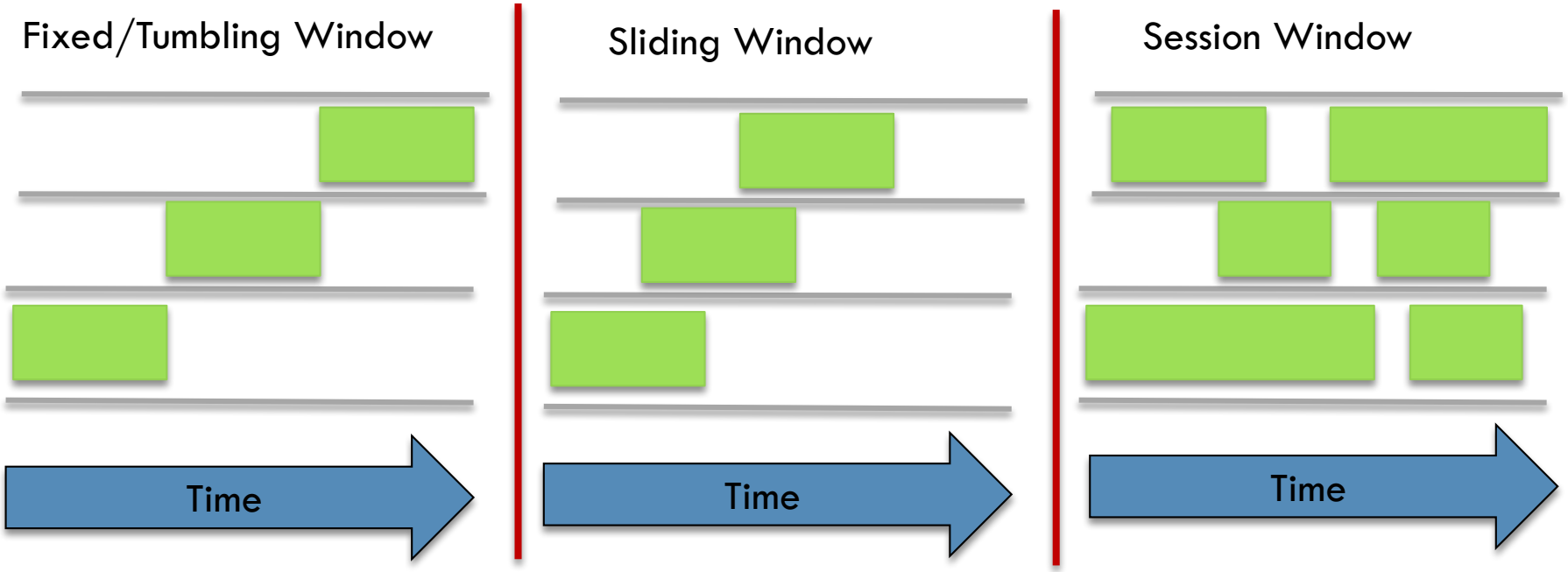
Streaming Processing



Streaming Processing

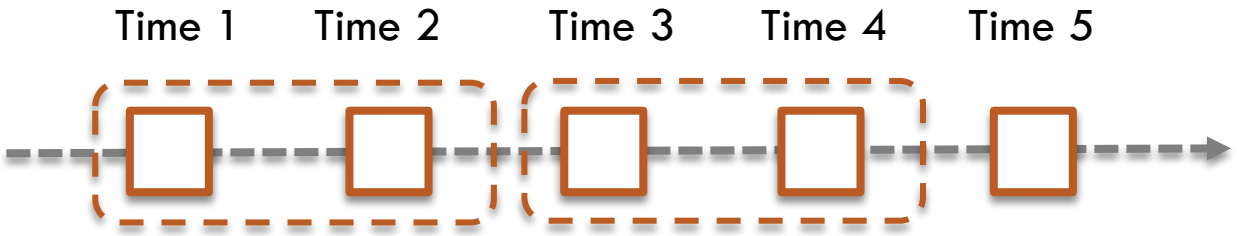
Windowing

- Slicing up incoming data into chunks for processing
- Delineate finite boundaries for grouping

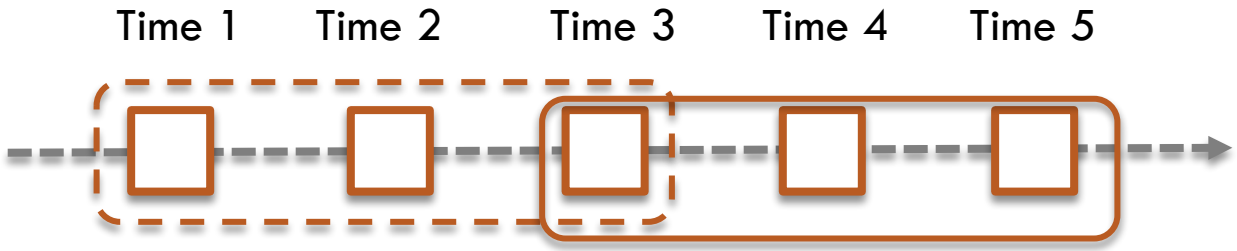


Streaming Processing

- Streaming Windows
  - Tumbling/Fixed



  - Sliding (smoother aggregation)



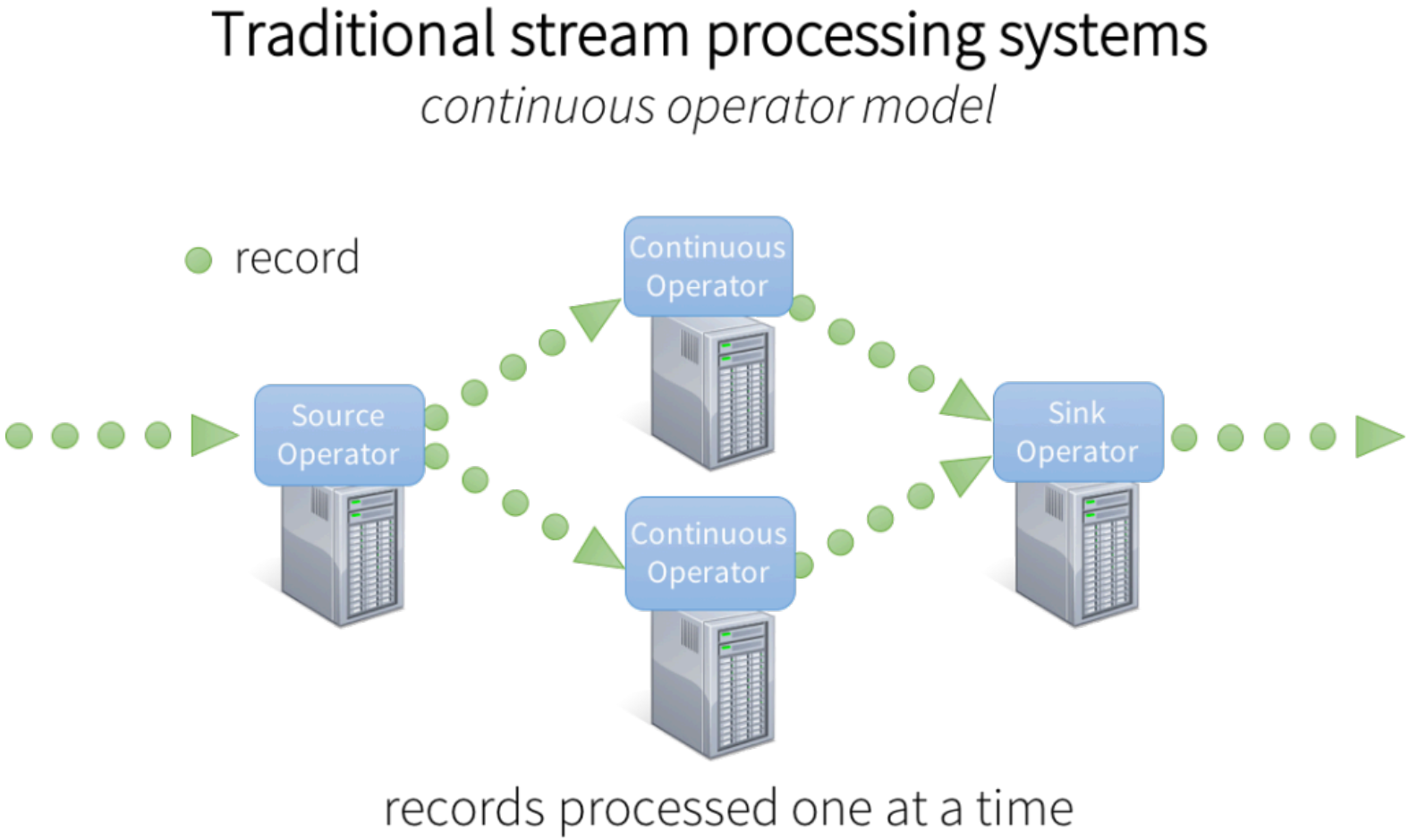
Streaming Processing

Data Streaming Processing Systems

(Heron)

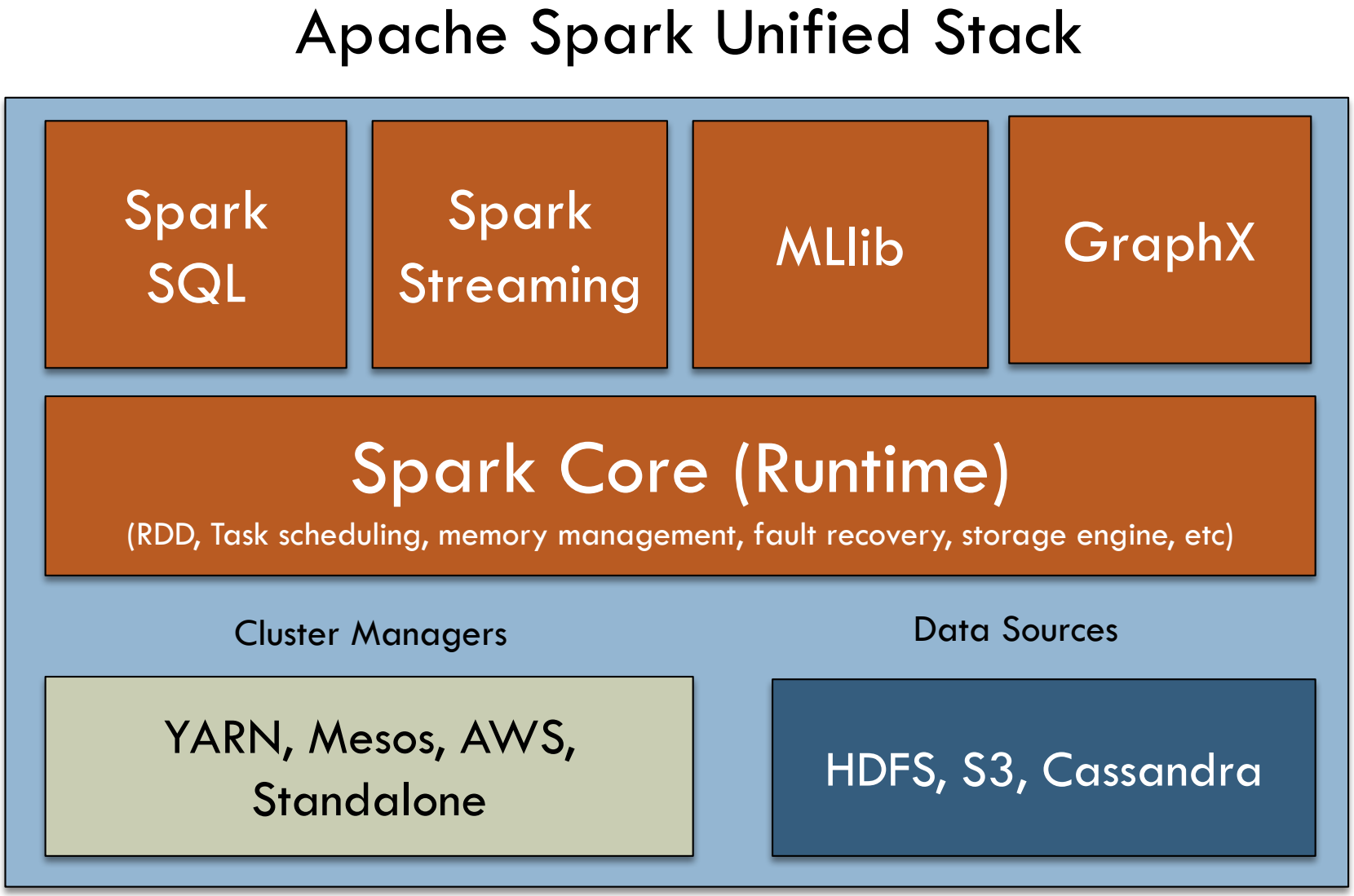
Apache Beam

Streaming Processing

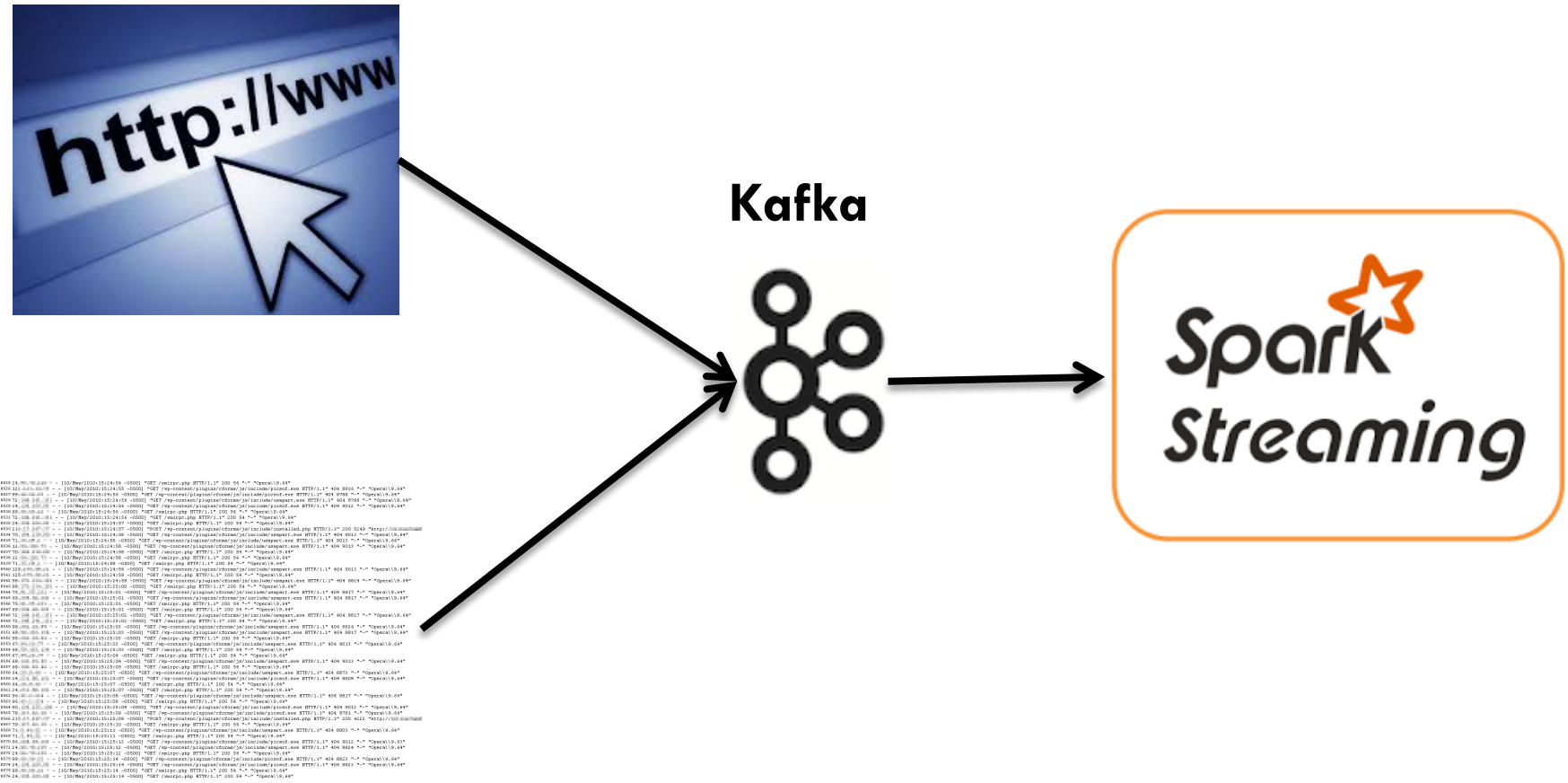


https://databricks.com/blog/2015/07/30/diving-into-spark-streamings-execution-model.html

Spark Streaming Concepts



Spark Streaming Concepts



Event data

Buffering

Processing

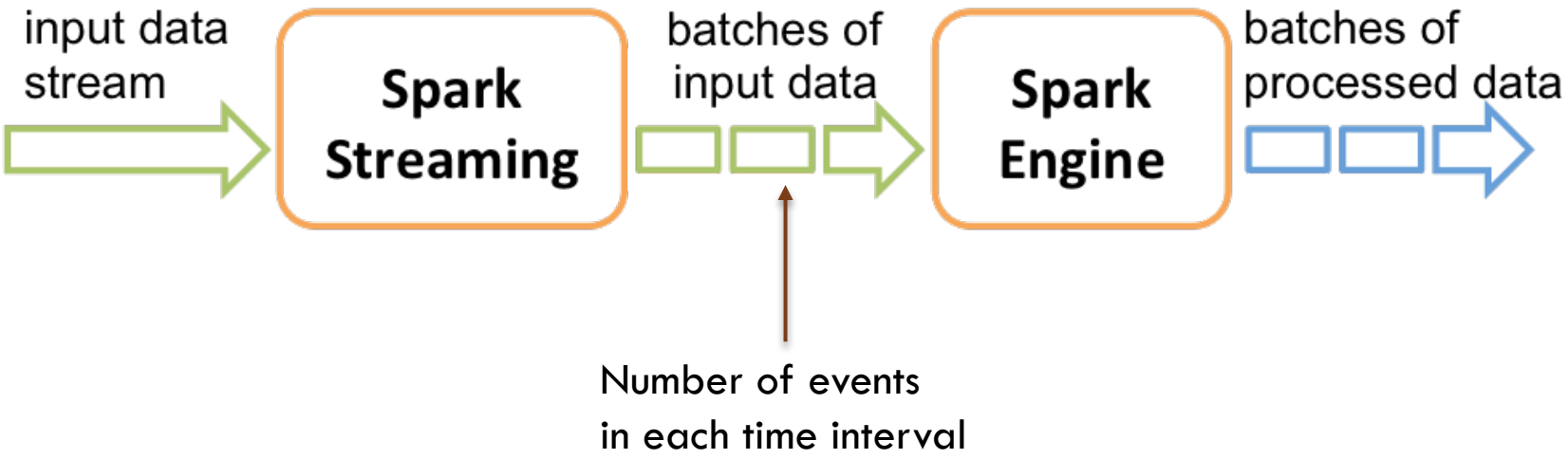
Spark Streaming Concepts

High-level APIs Fault-tolerant Flexible Integration

https://spark.apache.org/docs/latest/streaming-programming-guide.html

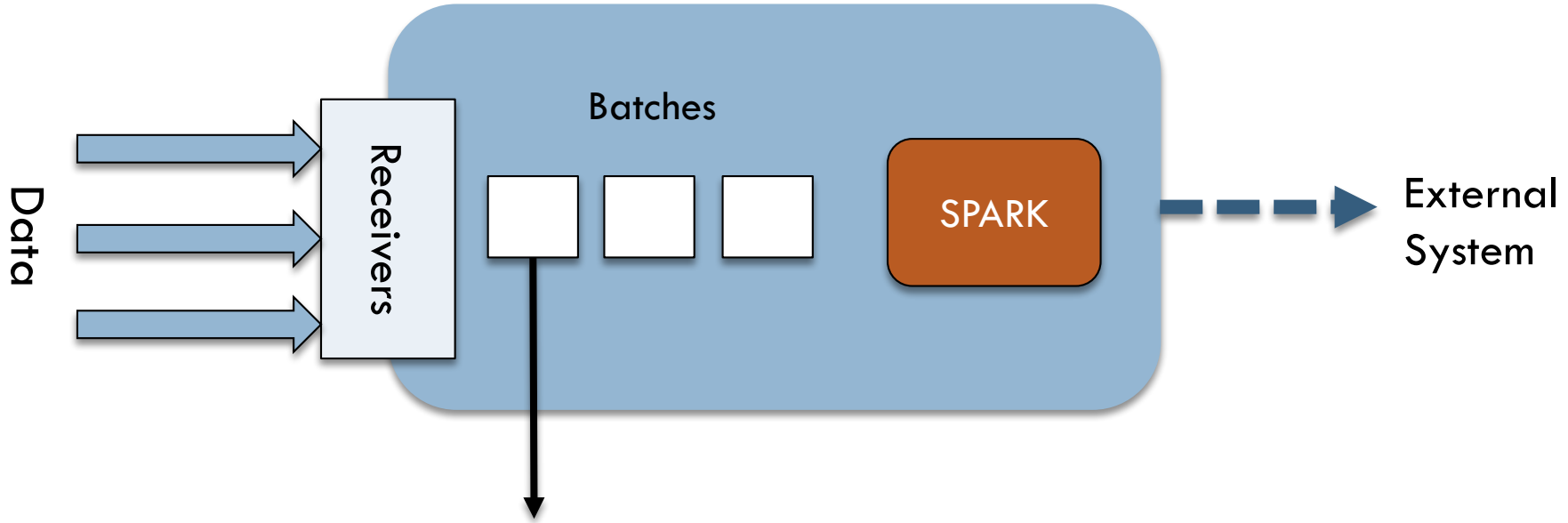
Spark Streaming Concepts

Treat streaming computations as a series of deterministic batch computations on small time intervals



Spark Streaming Concepts

Spark Streaming uses
"micro-batch" architecture



Each batch is created at
regular time interval,
aka batch interval

Spark Streaming Concepts

Word Count Example

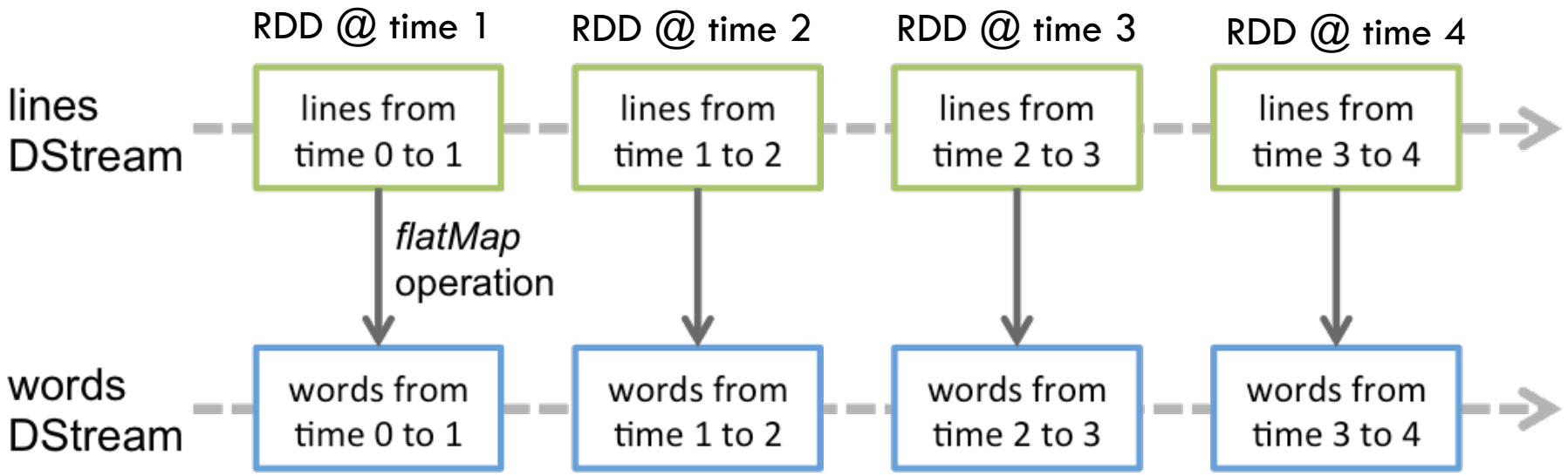
```
val batchInterval = Seconds(1)

// Entry point for streaming functionality
val ssc = new StreamingContext(new SparkConf(), batchInterval)

// create a DStream
val lines = ssc.socketTextStream("localhost",9888)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x,1)).reduceByKey(_ + _)

wordCounts.print()

ssc.start()  // start the computation

// wait for computation to terminate
ssc.awaitTermination()
```

https://github.com/ - NetworkWordCount.scala

Spark Streaming Concepts

Discretized Streams (DStream) Abstraction

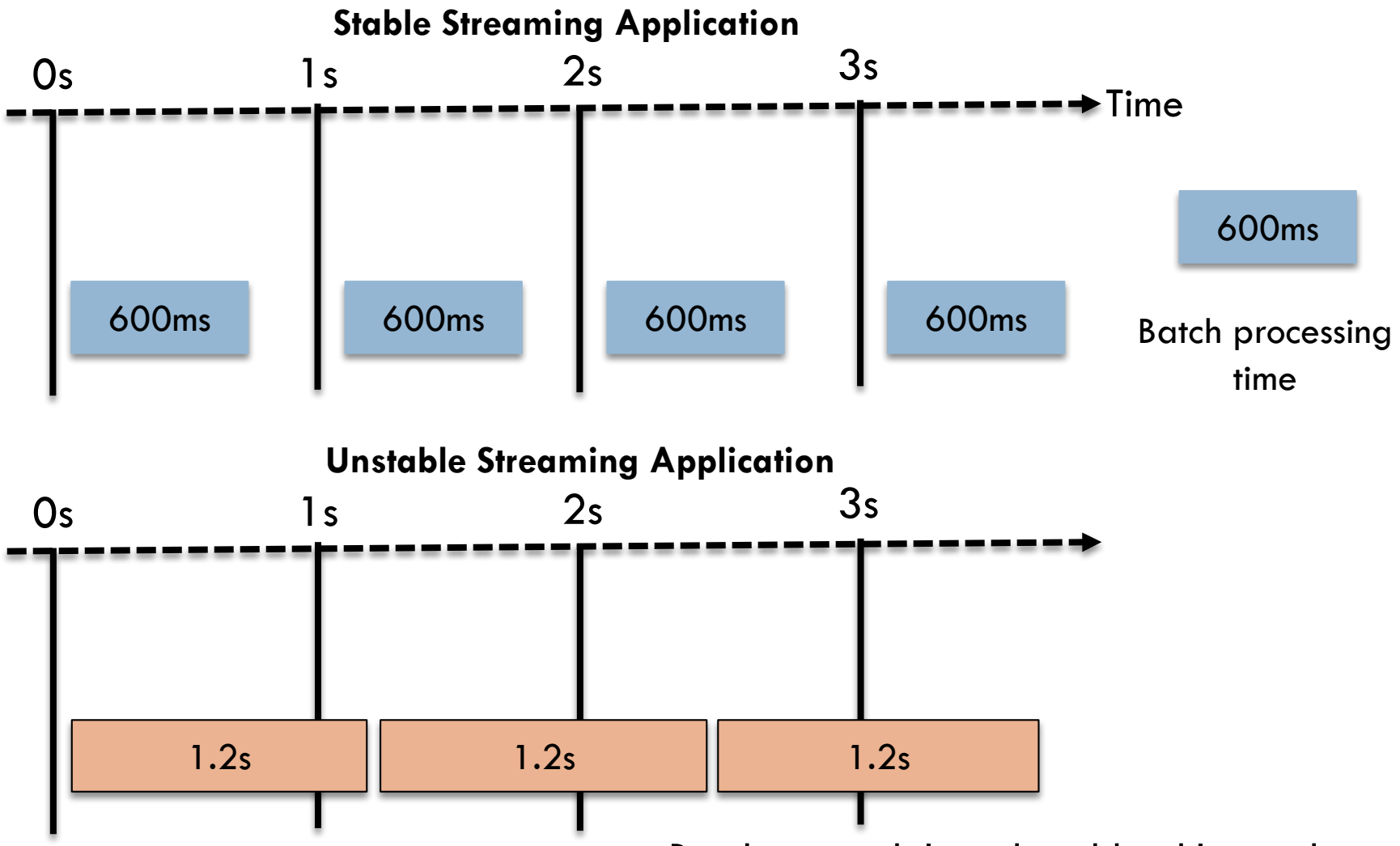
```
val context = new StreamContext(new SparkConf(), Seconds(1))
```



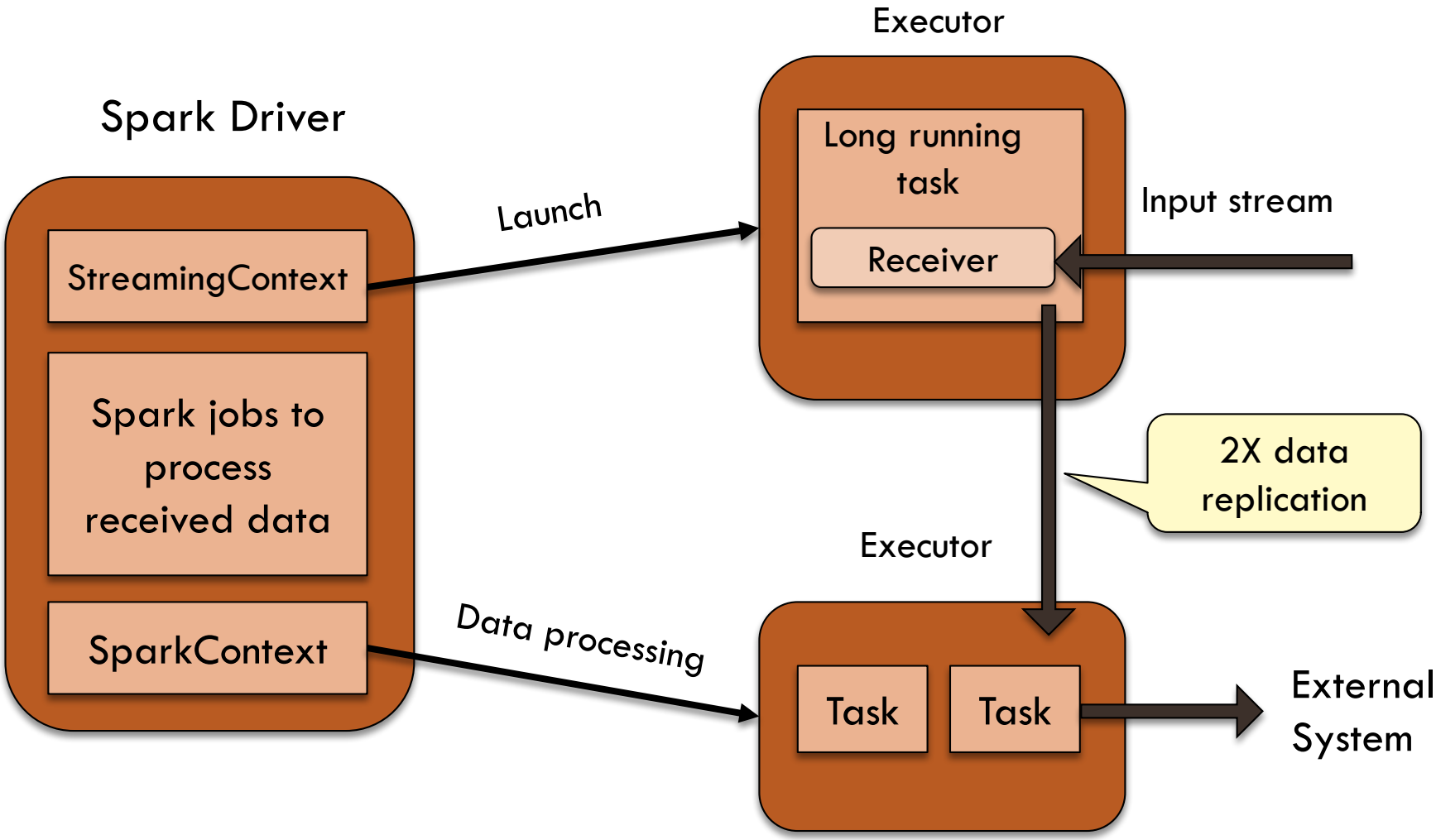
Continuous series of RDDs

http://spark.apache.org/docs/latest/streaming-programming-guide.html

Spark Streaming Concepts



Spark Streaming Concepts



Spark Streaming Concepts

- DStream
  - One per streaming source
  - An application can have multiple DStreams
- Streaming Sources
  - Basic sources
    - file systems, socket connections, Akka actors
  - Advanced sources
    - Kafka, Flume, Kinesis, Twitter
    - Extra utility classes
  - Custom sources

Working with DStream APIs

- DStream transformation types
  - Stateless and stateful
- Stateless transformations
  - Processing of each batch doesn't depend on data from previous batches
  - map(), filter(), reduceByKey()
- Stateful transformations
  - Use data or intermediate results from previous batches
  - Sliding windows or tracking state across time

Working with DStream APIs

DStream Transformations

Transformation	Description
map(func)	Passing each element through given function
filter(func)	Select those elements which func returns true
flatMap(func)	Each input item can be mapped to 0 or more output items
repartition(numParitions)	Change level of parallelism
union(sataSet)	Create a union of two data sets
reduce(func)	Create an intersection between two data sets
countByValue()	Count values per key, returns (K, Long) pairs
reduceByKey(func)	Combine the values of the same key
join(otherStream), cogroup	Return (K,(V,W)), (K, Seq(V), Seq(W))
updateStateByKey(func)	Return new Dstream where state of each key is updated
transform(func)	Apply func to every RDD in the source stream

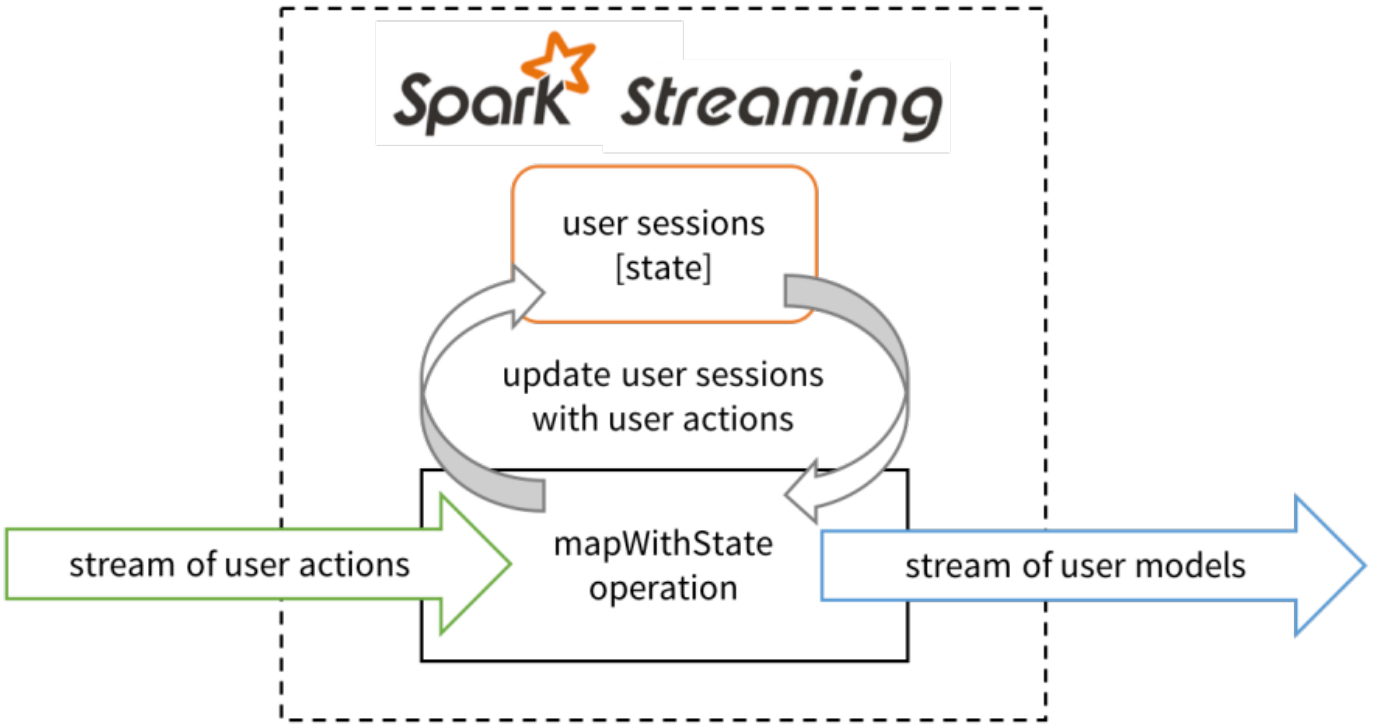
http://spark.apache.org/docs/latest/streaming-programming-guide.html

Working with DStream APIs

- State Management
  - A very useful and powerful feature of Spark Streaming
  - Maintain state across a period time
    - Maintain running count of each word
    - Maintain last 10 pages each user visited
    - Maintain user sessions and continuously update them
  - Seamless fault-tolerant & recovery
  - Two options
    - updateStateByKey – pre Spark 1.6
    - mapWithState – Spark 1.6

Working with DStream APIs

Maintaining User Sessions



https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html

Working with DStream APIs

- mapWithState Features
  - Applicable to only K,V Dstream
  - Maintain states between batches, continuously updating
  - Update for each key's new values
  - Support for timeout – remove the state automatically
  - Two step process
    - Define the spec
    - Define the update
  - Intelligent partial update for keys with new values
  - Can specify initial state

Working with DStream APIs

- mapWithState(func)

```
def updateState(….)


val stateSpec = StateSpec.function(updateState _)
      .initialState(initialRDD)
      .timeout(Minutes(120))

val wordCountState = wordStream.mapWithState(stateSpec)
```

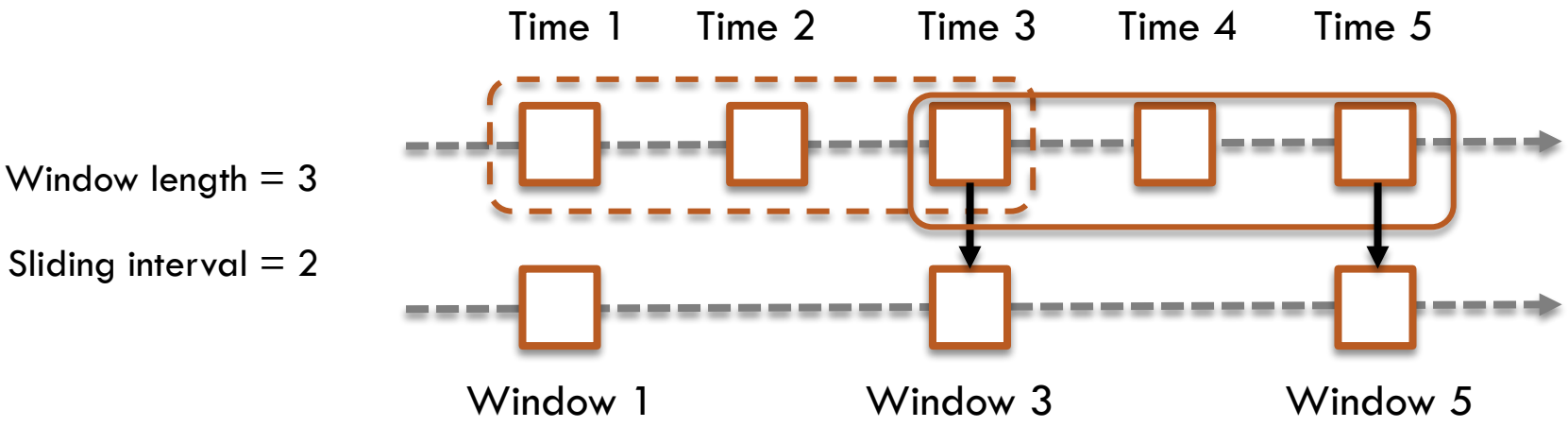
StatefulNetworkWordCount.scala

Working with DStream APIs


```
def updateState(key: String,              // key for state
                value: Option[Int],       // value at batch time
                state: State[Long]        // current state
               ): Option[(String, Long)] = {

 val sum = value.getOrElse(0).toLong +
           state.getOption.getOrElse(0L)
 state.update(sum)
 Some((key, sum))
}

val initialRDD = ssc.sparkContext.parallelize(List(("hello", 1),
("world", 1)))
val stateSpec = StateSpec.function(updateState _)
                  .initialState(initialRDD)
                  .timeout(Minutes(120))

val wordCountState = wordStream.mapWithState(stateSpec)
```

Working with DStream APIs

- Windowed Transformations
  - Compute results across batch intervals (sliding window)
  - Combine results from multiple batches
  - All windowed operations requires two parameters
    - Window length – size of the window
    - Sliding interval – how frequent to compute result
    - Multiple of batch interval



Working with DStream APIs

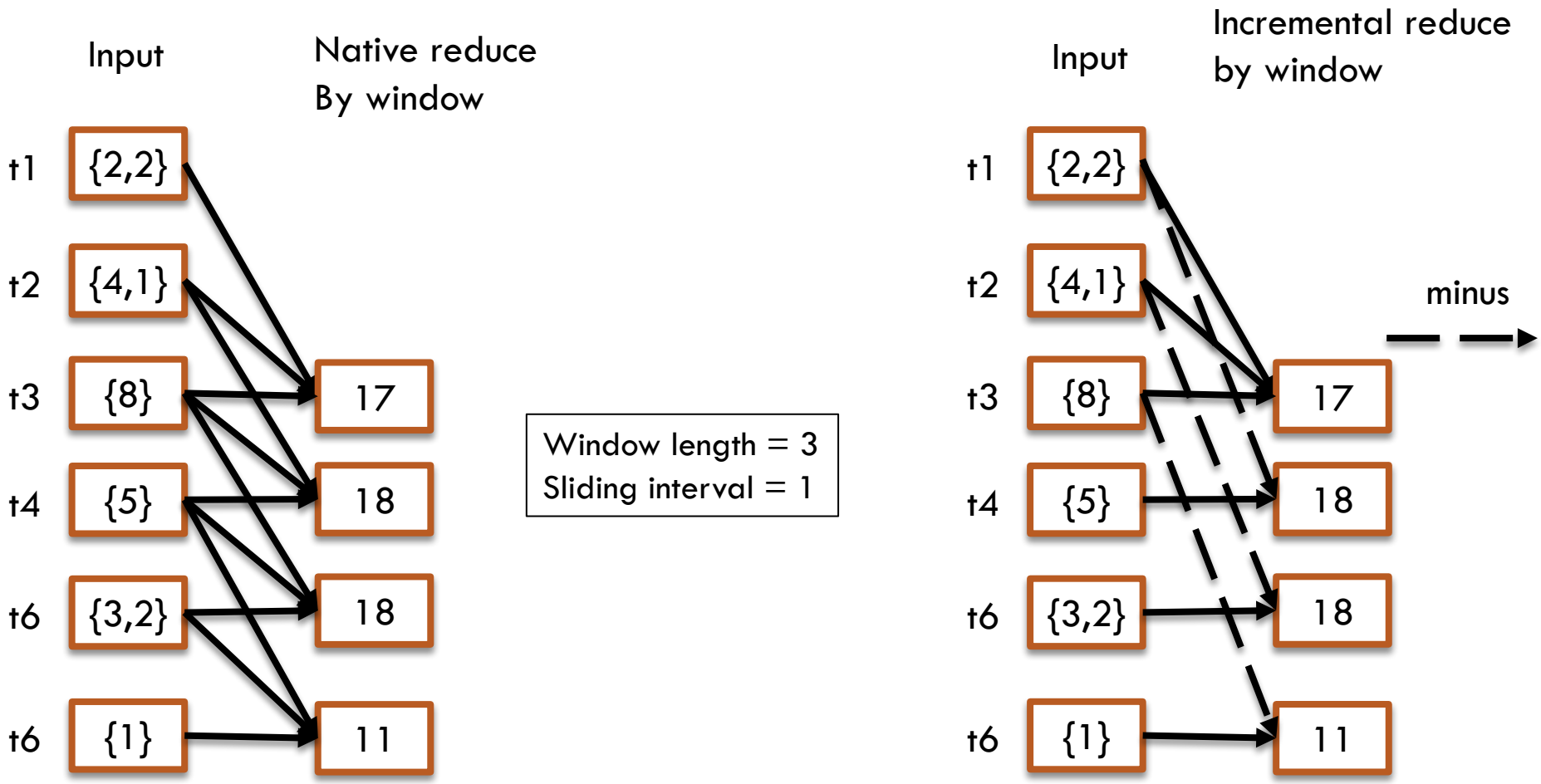
Windowed DStream Transformations

Transformation	Description
window(wl, si)	New Dstream contains data from multiple batches
countByWindow(wl, si)	Sliding window count of elements in the DStream
reduceByWindow(wl, si)	Aggregate elements in the sliding interval using provided func
reduceByKeyAndWindow(func,wl, si)	Aggregate (K,V) elements in the sliding interval using provided func
reduceByKeyAndWindow (func, invFunc,wl, si)	Aggregate (K,V) elements incrementally in the sliding interval using provided func and invFunc
countByValueAndWindow(wl, si)	Count values per key, returns (K, Long) pairs within a sliding window

http://spark.apache.org/docs/latest/streaming-programming-guide.html

Working with DStream APIs

- reduceByKeyWindow vs incremental reduceByKeyWindow



Working with DStream APIs

- reduceByKeyWindow vs incremental reduceByKeyWindow

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow(
  (a:Int,b:Int) => (a + b),
  Seconds(30), Seconds(10))



// incremental reduce by key
val windowedWordCounts = pairs.reduceByKeyAndWindow(
  (a:Int,b:Int) => (a + b), // adding elements to new batch
  (a:Int,b:Int) => (a - b), // remove elements from old batch
  Seconds(30),              // window length
  Seconds(10))              // sliding interval
```

Working with DStream APIs

- Stream to Stream join
  - Join happens in each batch interval

```
val s1: DStream = …
val s2: DStream = …
val s3 = s1.join(s2)
```

- Stream and Regular RDD join

```
val dataSet: RDD = …
val wStream: DStream = stream.window(Seconds(10))
val s3 = wStream.join(dataSet)
```

Working with DStream APIs

- DStream Output Operations
  - Trigger the evaluation of DStream lazy-evaluated operations
  - Output the final transformed data in a stream at each batch interval

Output Operation	Description
print()	Print the first 10 elements of every batch of Dstream on driver side
saveAsTextFiles(prefix,[suffix])	Save Dstream at each batch interval – prefix-TIME.[suffix]
saveAsObjectFiles(prefix,[suffix])	Save Dstream at each batch interval as sequence files – prefix-TIME.[suffix]
foreachRDD(func)	Most generic output generator – apply func to each RDD generated from the stream.

http://spark.apache.org/docs/latest/streaming-programming-guide.html

Working with DStream APIs

- foreachRDD best practices
  - Avoid transferring objects from driver to worker
  - Avoid creating objects per record
  - Create an object per partition

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}

dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```

Working with DStream APIs

- foreachRDD best practices
  - Avoid transferring objects from driver to worker
  - Avoid creating objects per record
  - Create an object per partition

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // Leverage connection
    val connection = ConnectionPool.getConnection()

    partitionOfRecords.foreach(record => connection.send(record))

    // return to the pool for future reuse
    ConnectionPool.returnConnection(connection)
  }
}
```

Spark Streaming Fault Tolerance

- Data streaming application
  - Long running – 24/7 operation
  - Resilient to failure (driver, executor), upgrade
- Checkpointing
  - Saving information to fault-tolerant storage system
  - Metadata
    - Configuration, operations, incomplete batch
    - To recover the driver
  - Received Data
    - Generated RDD for stateful computations (`updateStateByKey` or `reduceByKeyAndWindow`)
- Write head logs
  - All data received from a receiver gets written into a write ahead log

Spark Streaming Fault Tolerance

- Checkpointing
  - Provide a directory on HDFS or S3 to write data to
  - Create new streaming context or recover from previous one
  - Additional costs for checkpointing
    - Determine appropriate checkpoint interval
    - For stateful transformations, set to multiple batch intervals

Spark Streaming Fault Tolerance

- Checkpointing
  - Create new streaming context or recover from previous one

```
// Function to create and setup a new StreamingContext
def createContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ssc.checkpoint(checkpointDir)   // set checkpoint directory
    ssc
}

val context = StreamingContext.getOrCreate(checkpointDir, createContext _)

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted

// Start the context
context.start()
context.awaitTermination()
```

RecoverableNetworkWordCount.scala

Spark Streaming Fault Tolerance

- Failure Modes
  - Driver
    - Create new or reuse existing streaming context from checkpoint
    - Must monitor and restart the driver
    - Configure write ahead logs
  - Worker
    - All data received is replicated among the workers
    - Use same techniques as Spark RDD for fault tolerance
  - Receiver
    - Restart failed receivers on other nodes
    - Data loss depending on source and receiver implementation
    - Best to use reliable input source – HDFS, Kafka directly

Spark Streaming Fault Tolerance

- Processing Guarantee
  - Exactly-one semantics for all transformations
  - Some data may get pushed out to external system multiple times
    - Unless use transactions to push to external system
    - Design update to be idempotent (multiple updates produce same result)

Spark Streaming – Robust, Scalable & Adaptive

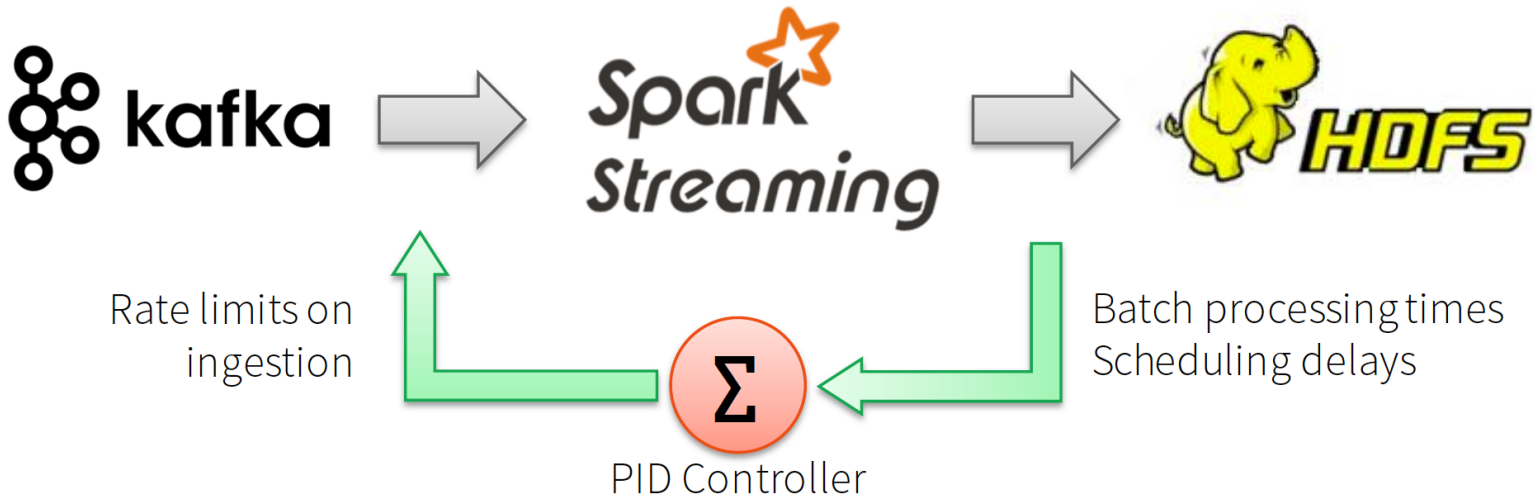
Adaptive

Backpressure
Robust against data surges

Elastic Scaling
Scale with load variations

Spark Streaming – Robust, Scalable & Adaptive

- Backpressure – feedback loop
  - Use current operating condition to determine the rate of data ingestion
  - Enable through SparkConf
    - `spark.streaming.backpressure.enabled = true`



Spark Streaming – Robust, Scalable & Adaptive

- Elastic Scaling (aka Dynamic Allocation) – 2.0
  - Scaling # of Spark executors according to load
  - Already support for batch jobs
  - Scale up and scale down

If Kafka gets data faster than what backpressure allows → SS scales up cluster to increase processing rate

Spark Streaming – Robust, Scalable & Adaptive

Demo

Backpressure relaxes limits to allow higher ingestion rate

But still less than 20x as cluster is fully utilized

databricks

Spark Streaming Application

- Can't run in spark-shell
- Build as Spark application
- Run from command line using spark-submit

```
bin/run-example streaming.NetworkWordCount localhost 9999

bin/run-example streaming.StatefulNetworkWordCount localhost
9999
```

Resources

- Discretized Streams
  - https://www.cs.berkeley.edu/~matei/papers/2012/hotcloud_spark_streaming.pdf