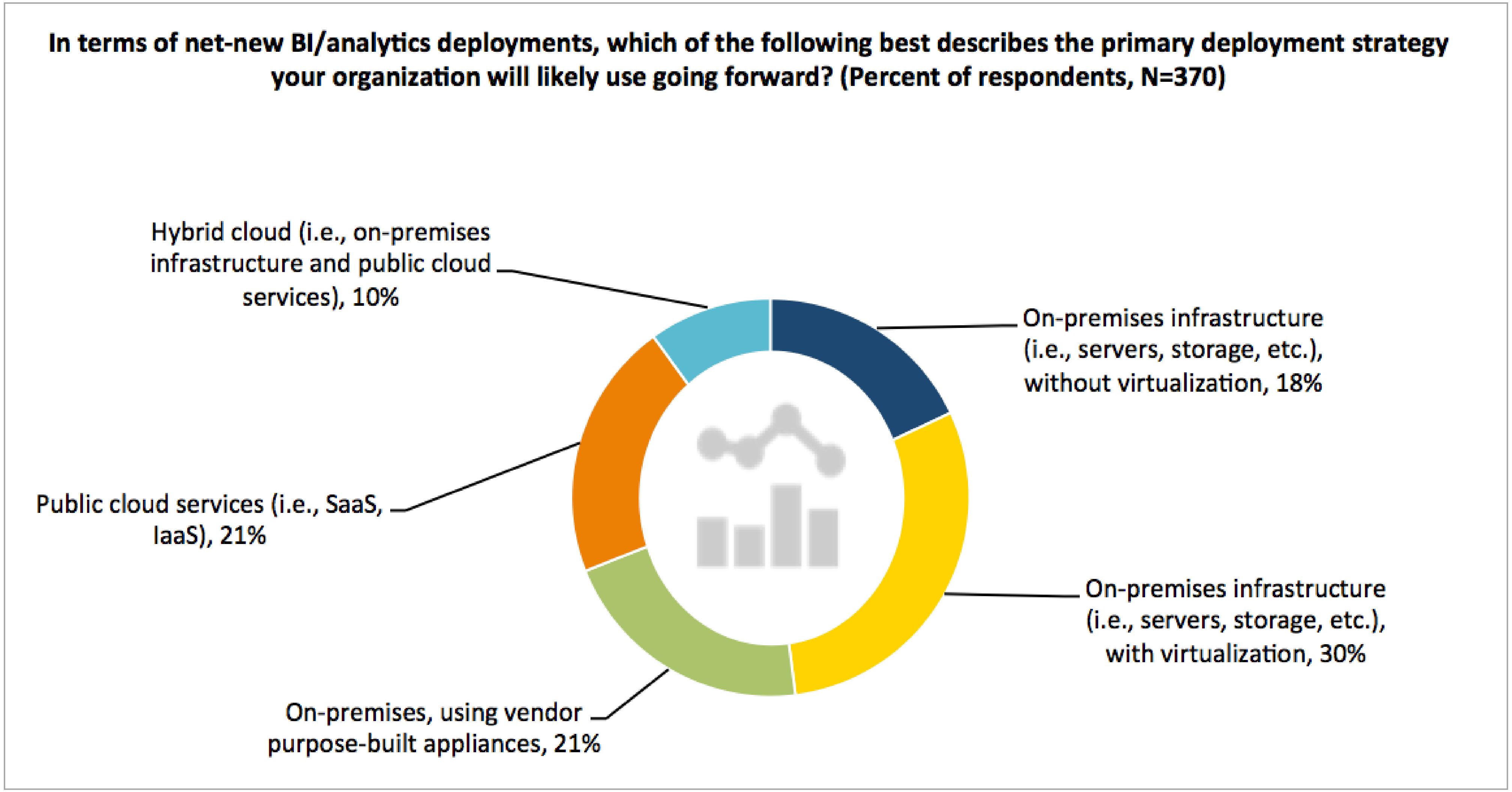
MITS6005

Big Data

Session 10 & 11

Spark Deployments

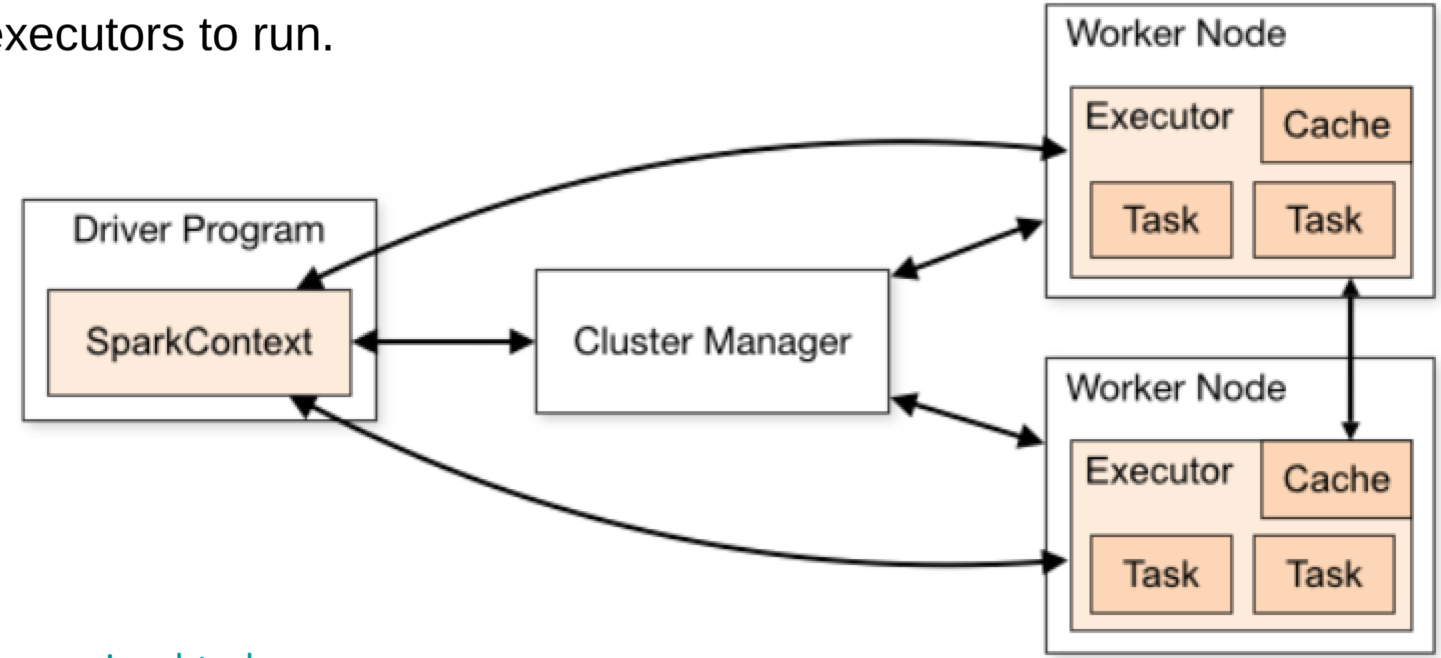
Big Data Deployment Options



Source: Enterprise Strategy Group (ESG) Survey, 2015

How Spark runs on clusters

- Spark applications run as independent sets of processes on a cluster, coordinated by the `SparkContext` object in your main program (called the *driver program*).
- To run on a cluster, the SparkContext can connect to several types of *cluster managers*, which allocate resources across applications.
- Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors.
- Finally, SparkContext sends *tasks* to the executors to run.



Source: https://spark.apache.org/docs/latest/cluster-overview.html

Spark Cluster Management Types

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN – the resource manager in Hadoop 2.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

Common Spark Deployment Patterns

Most Common Spark Deployment Environments (Cluster Managers)

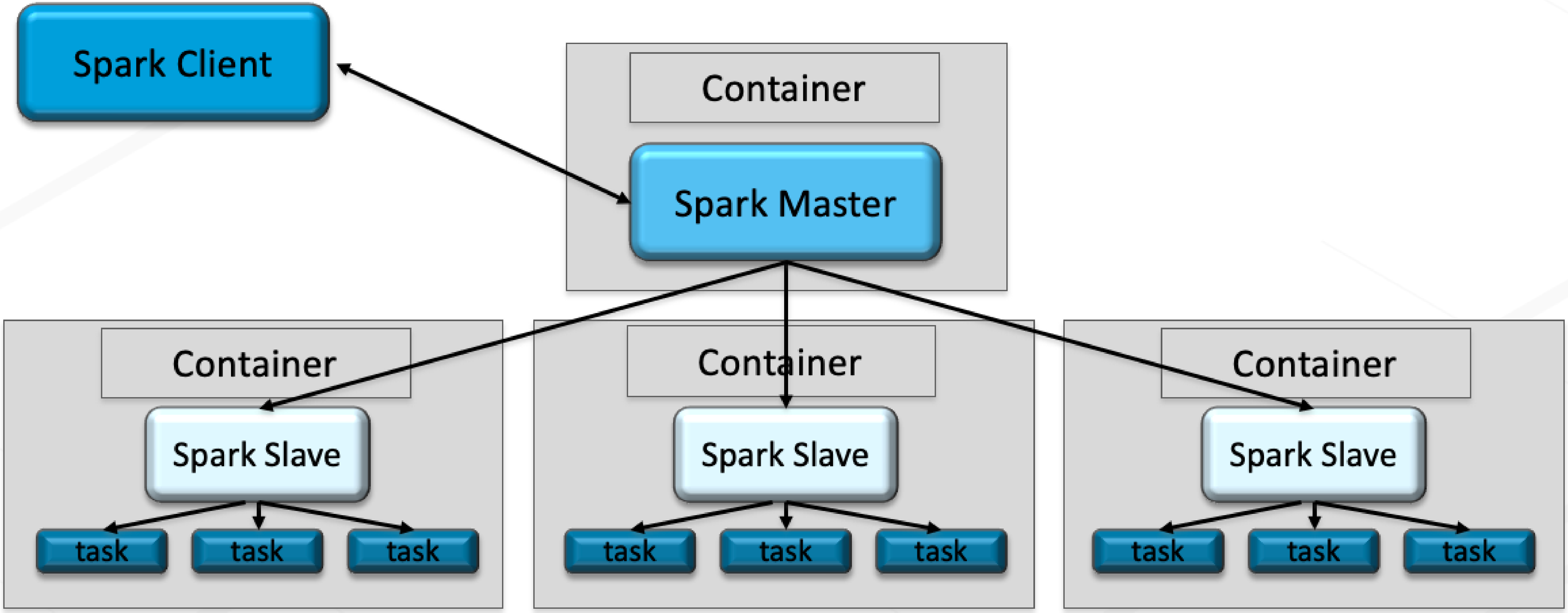
48%
Standalone mode

40%
YARN

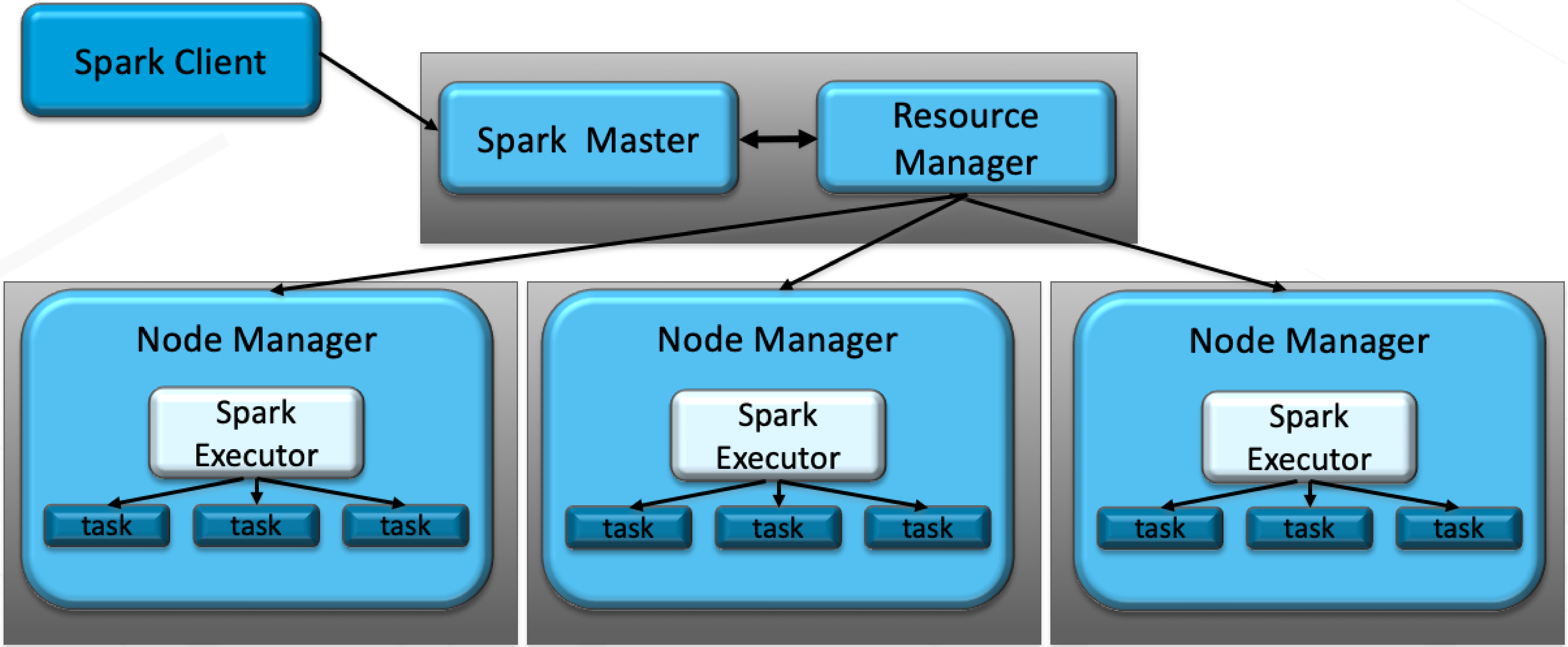
11%
Mesos

Source: Spark Survey Report, 2015 (Databricks)

Spark Cluster – Standalone



Spark Cluster – Hadoop YARN

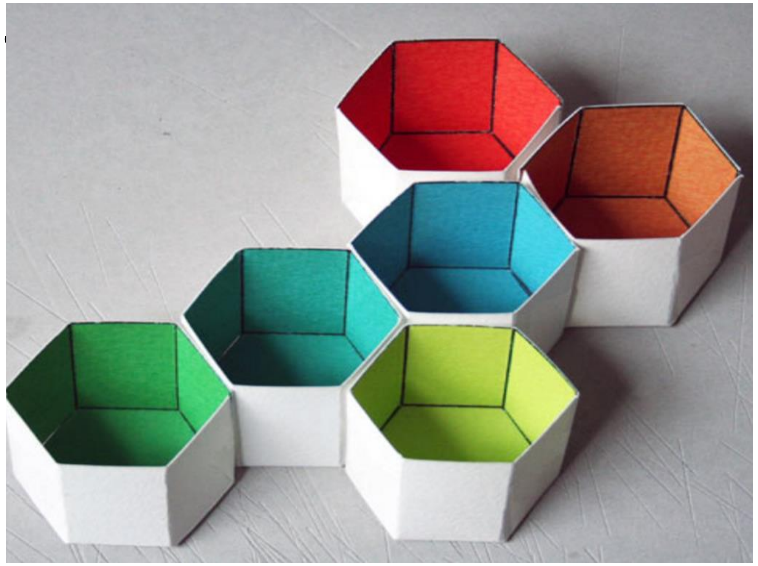


What is Real-Time Data Analysis?

- Spot risk as it happens across multiple data sets
- Tap into data that is always on (sensor data, machine logs, web logs, etc.)
- React to changing business conditions in real time
- Spot opportunities before it is too late

Visualizing Different Analytics

Data Marts	Data Lake	Real-



- Data warehouse
- High response time
- Low latency
- Purpose-built
- No room for innovation

- Batch data
- High latency
- Iterative
- Exploratory
- Try a few different ways

- Real-time data
- Low latency
- High throughput
- More unknowns
- No room for error

Real-Time Analysis Use Cases

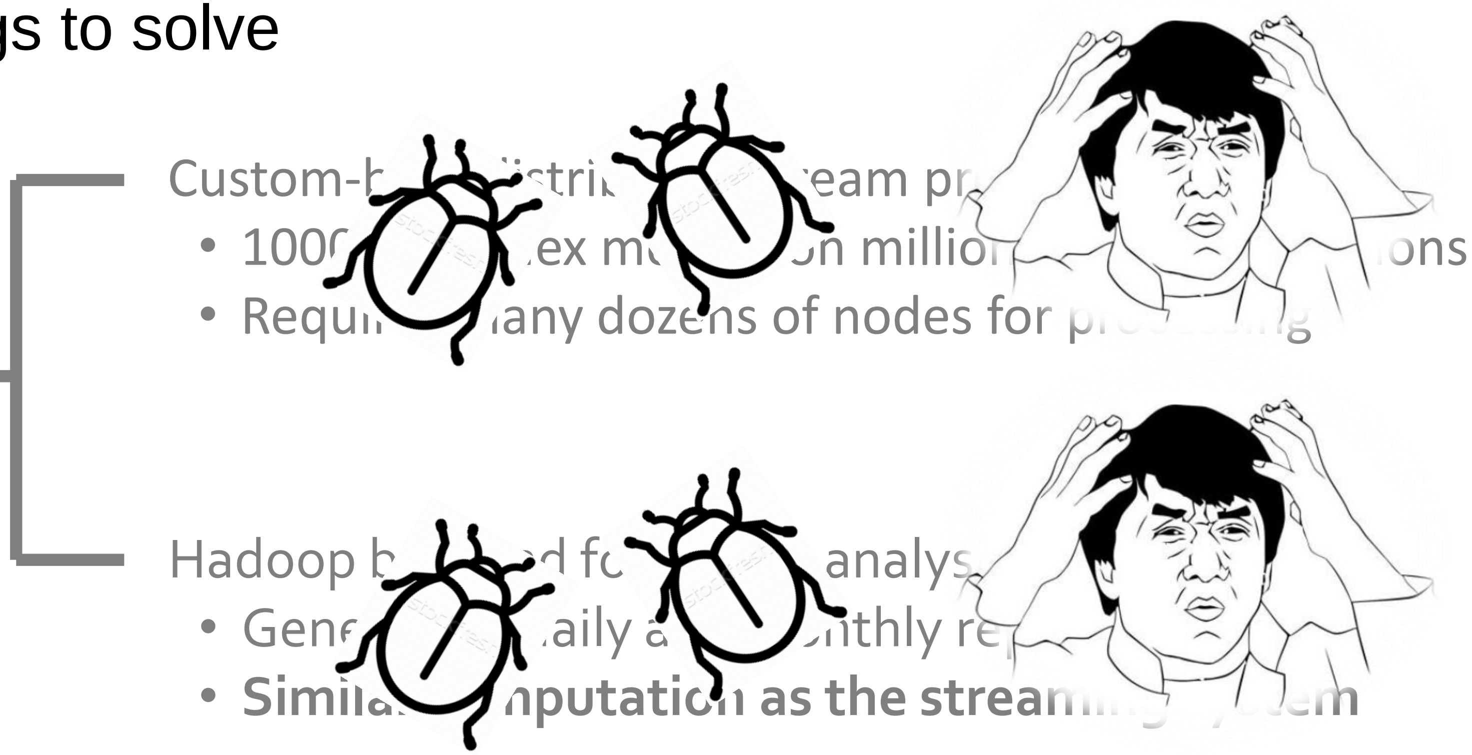
applications	sensors	web	mobile phones
intrusion detection	malfunction detection	site analytics	network metrics analysis
fraud detection	dynamic process optimisation	recommendations	location based ads
log processing	supply chain planning	sentiment analysis	...

What is Spark Streaming?

- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Integrates with Spark’s batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.

Case study: XYZ, Inc.

- Any company who wants to process live streaming data has this problem
- Twice the effort to implement any new function
- Twice the number of bugs to solve
- Twice the headache
- Two processing stacks

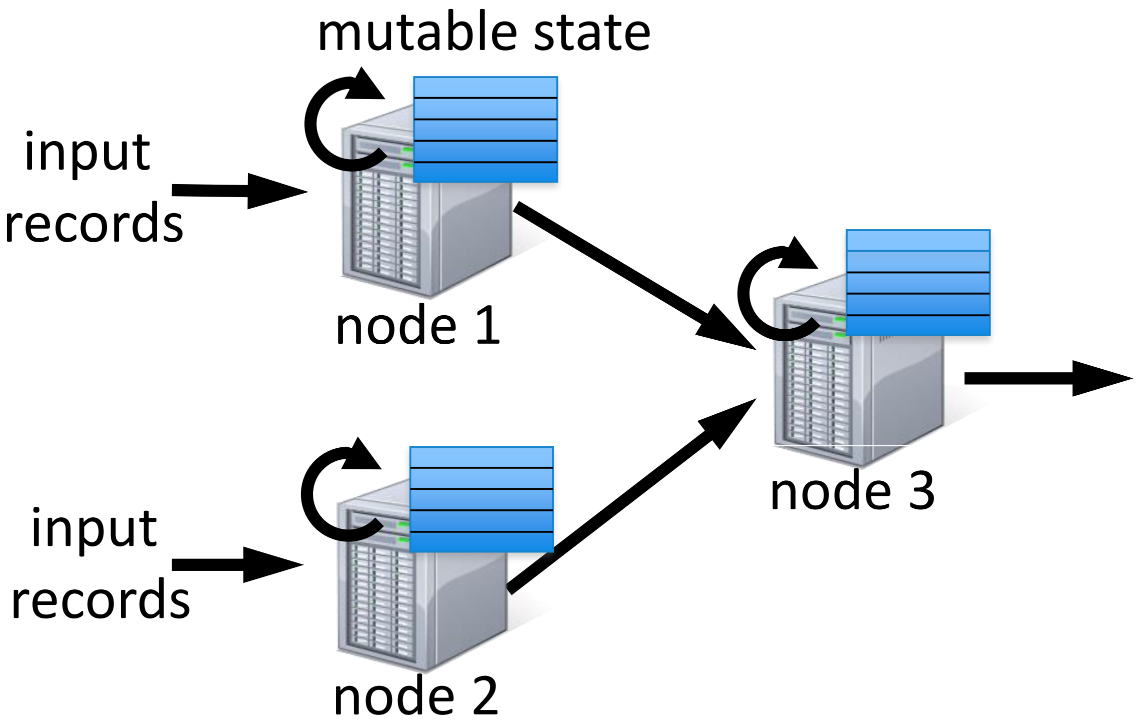


Stateful Stream Processing

- Traditional streaming systems have a event-driven **record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records

- State is lost if node dies!

- Making stateful stream processing be fault-tolerant is challenging

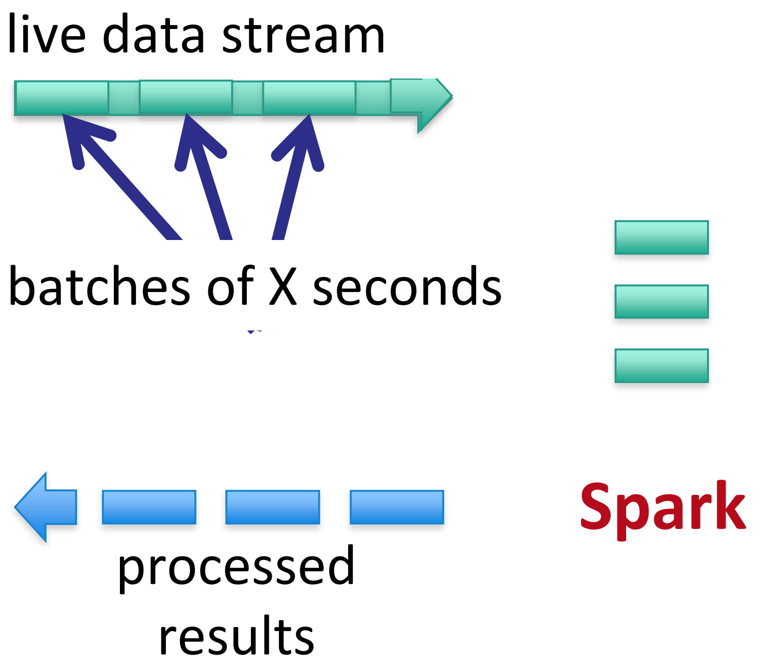


14

Discretized Stream Processing

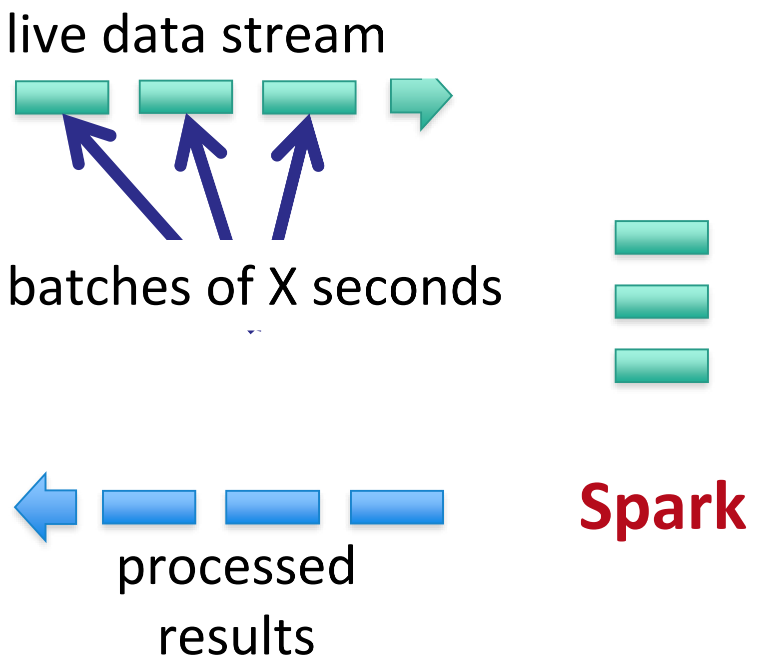
Run a streaming computation as a **series of very small, deterministic batch jobs**

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**



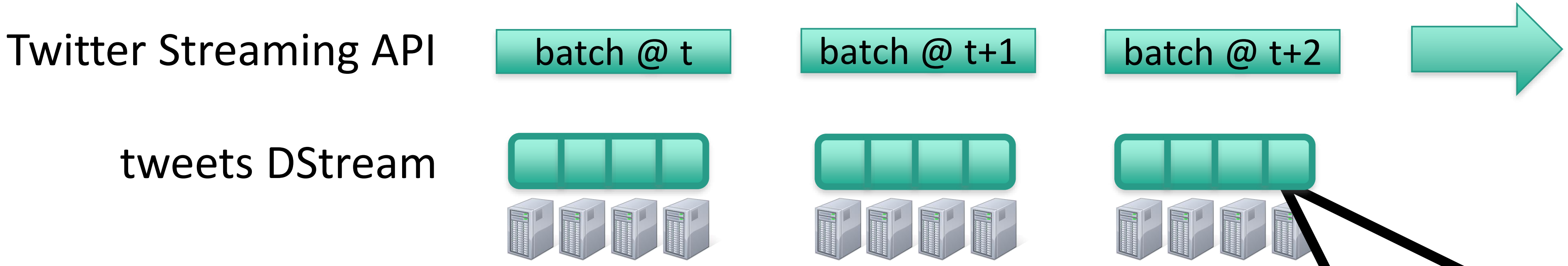
- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system

16

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data



stored in memory as an RDD (immutable, distributed)

Example 1 – Get hashtags from Twitter

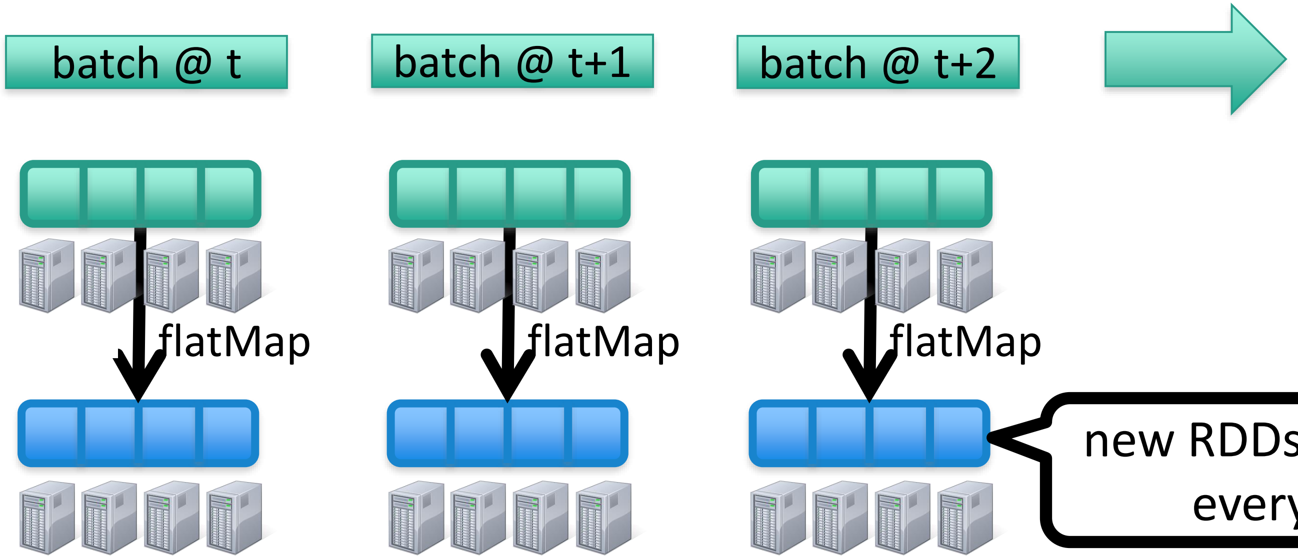
```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

transformation: modify data in one Dstream to create another DStream

tweets DStream

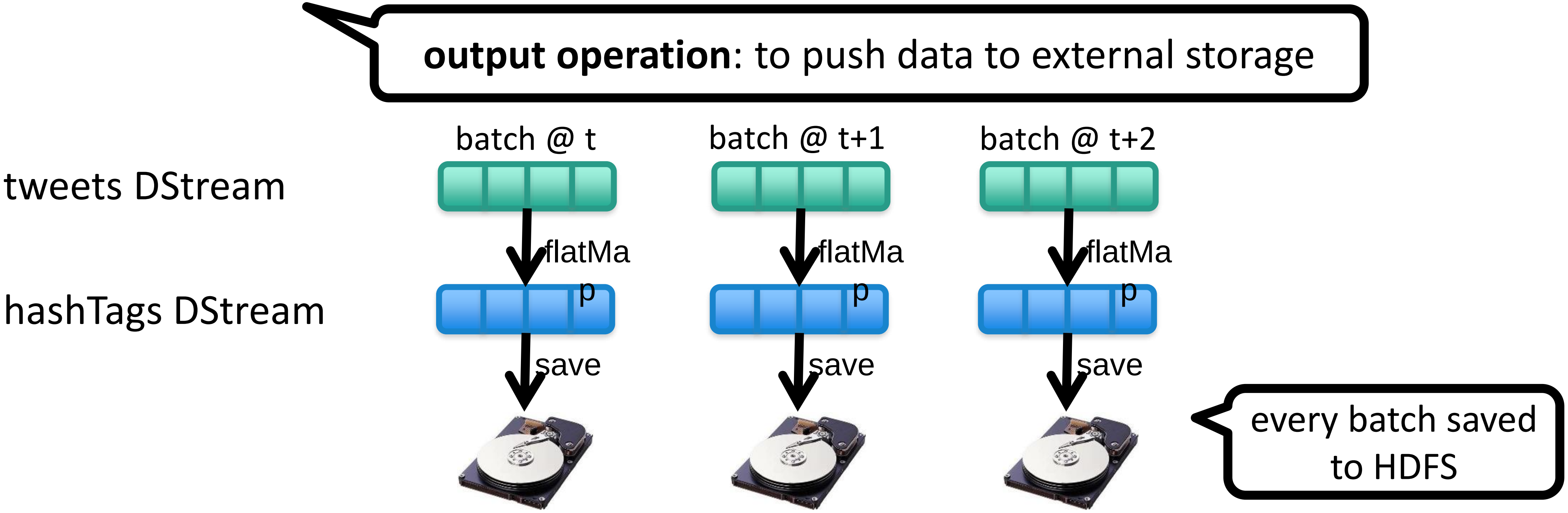
hashTags Dstream
[#cat, #dog, …]



new RDDs created for every batch

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

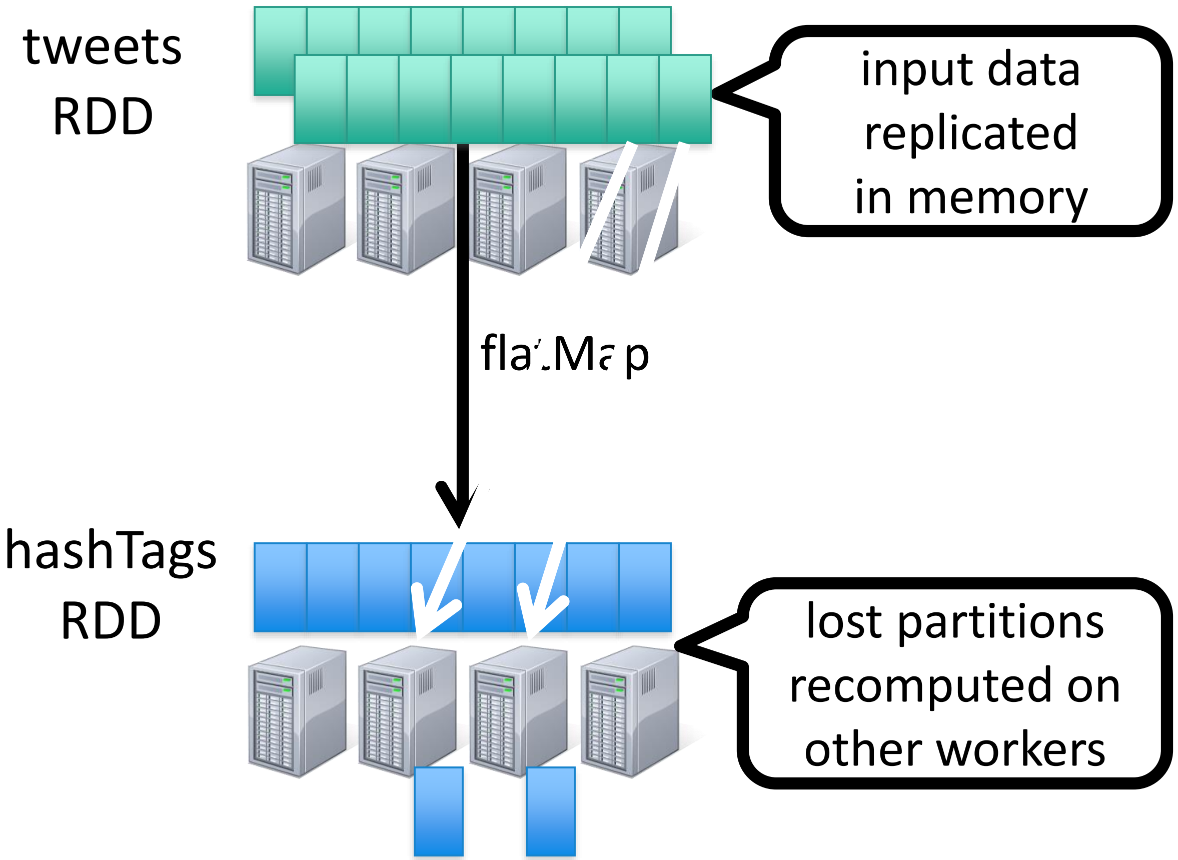


Fault-tolerance

- RDDs are remember the sequence of operations that created it from the original fault-tolerant input data

- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant

- Data lost due to worker failure, can be recomputed from input data

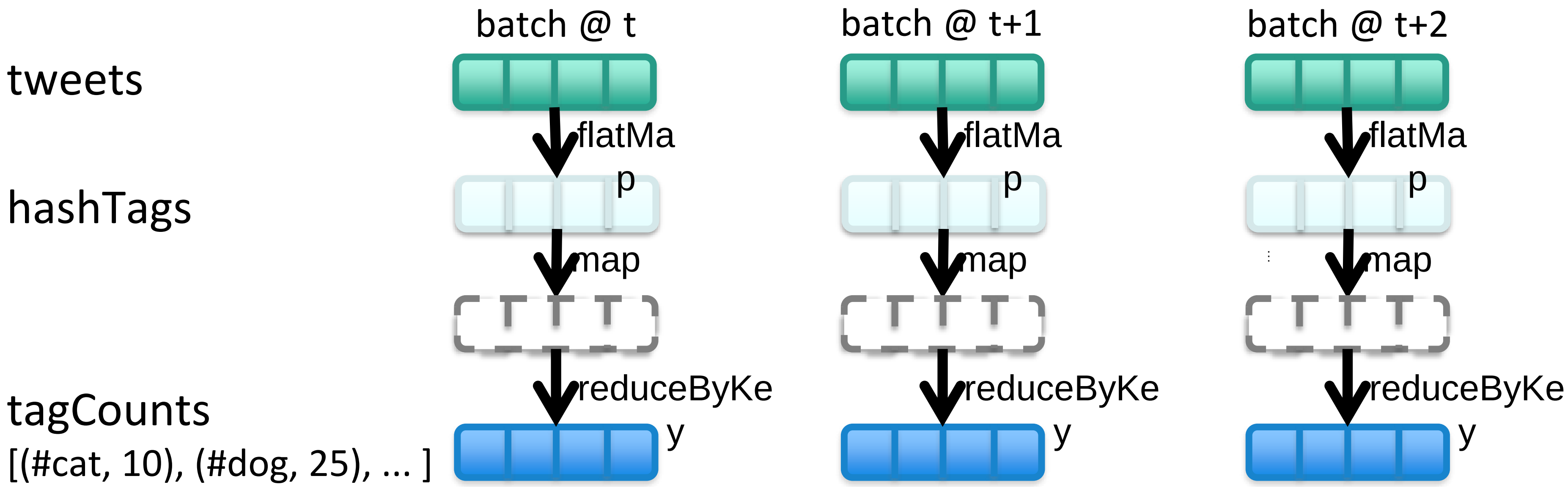


Key concepts

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- **Transformations** – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …
- **Output Operations – send data to external entity**
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Example 2 – Count the hashtags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```



Example 3 – Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

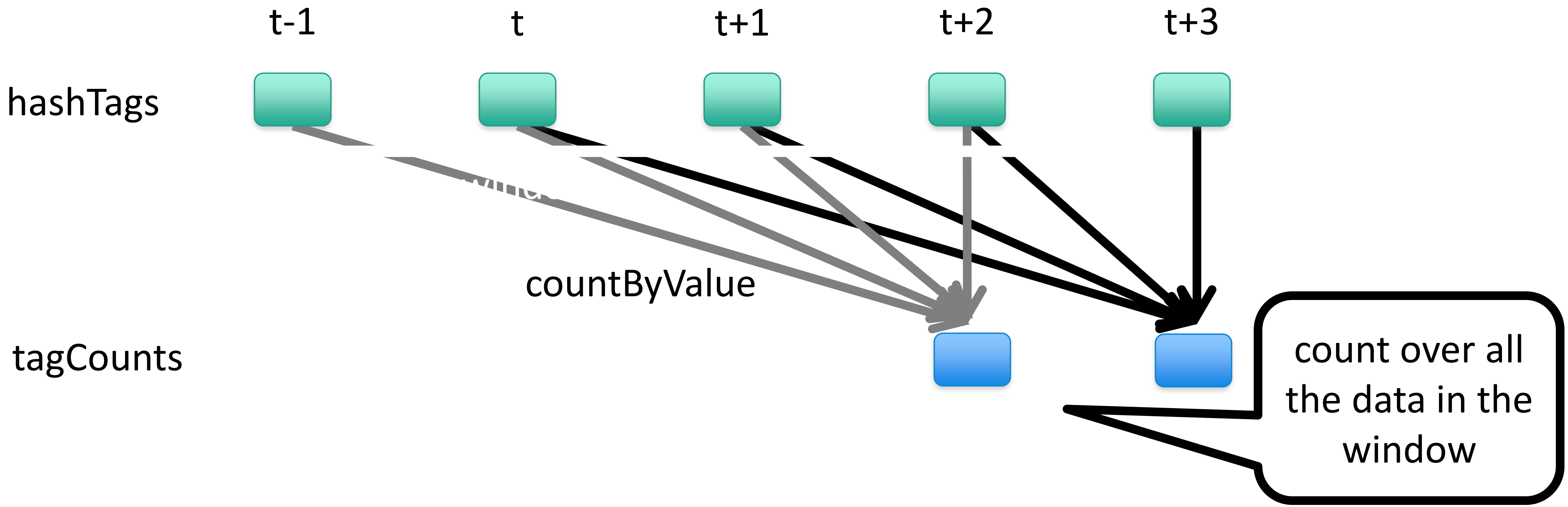
sliding window operation

window length

sliding interval

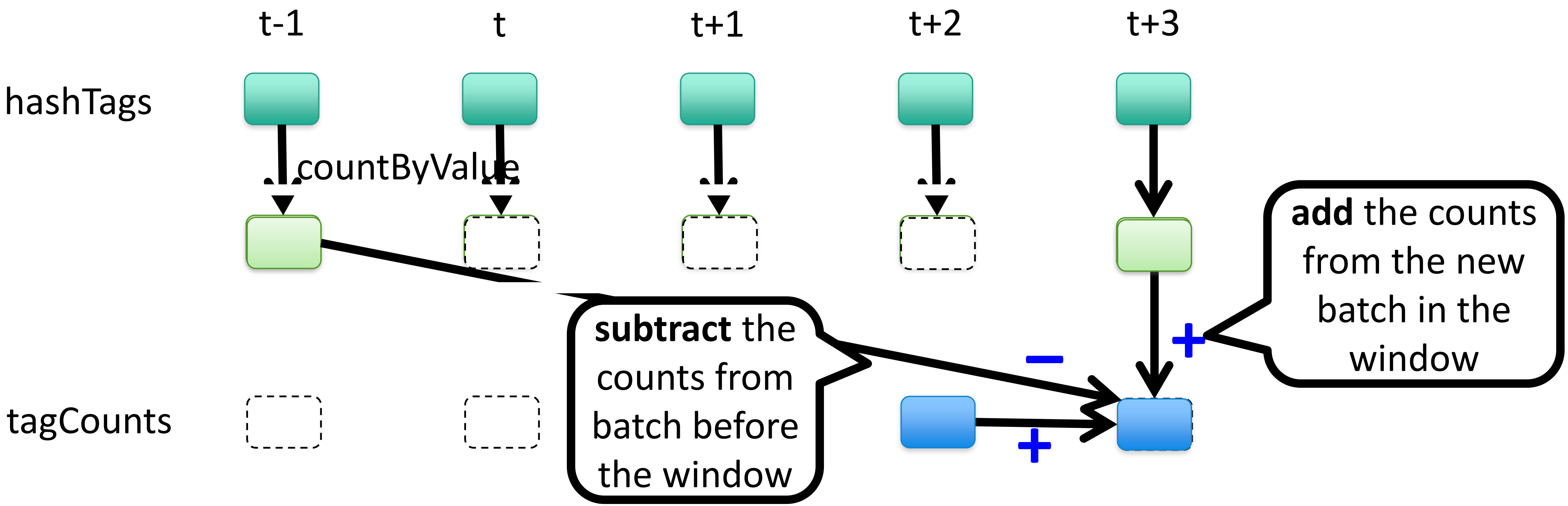
Example 3 – Counting the hashtags over last 10 mins

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



Smart window-based *reduce*

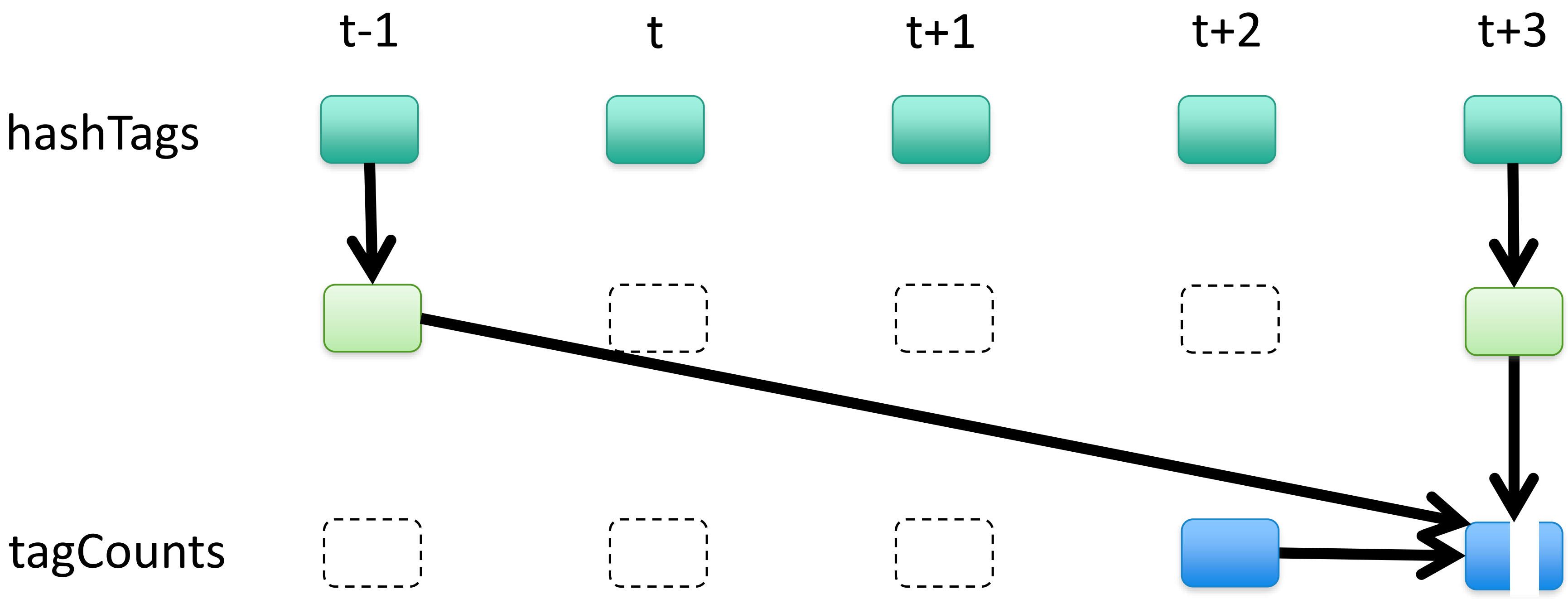
- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)
```

Fault-tolerant Stateful Processing

All intermediate data are RDDs, hence can be recomputed if lost



Fault-tolerant Stateful Processing

- State data not lost even if a worker node dies
  - Does not change the value of your result

- *Exactly once* semantics to all transformations
  - No double counting!

Other Interesting Operations

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet => updateMood(tweet))
```

- Do arbitrary Spark RDD computation within DStream
  - Join incoming tweets with a spam file to filter out bad tweets

```
tweets.transform(tweetsRDD => {
        tweetsRDD.join(spamHDFSFile).filter(...)
})
```

An example - Sentiment Analysis with PySpark

First step in any Apache programming is to create a SparkContext.

```
import findspark
findspark.init()
import pyspark as ps
import warnings
from pyspark.sql import SQLContext

try:
  # create SparkContext on all CPUs available: in my case I have 4 CPUs on my laptop
  sc = ps.SparkContext('local[4]')
  sqlContext = SQLContext(sc)
  print("Just created a SparkContext")
except ValueError:
  warnings.warn("SparkContext already exists in this scope")
```

For full details on sentiment analysis: https://towardsdatascience.com/sentiment-analysis-with-pyspark-bc8e83f80c35


The End