CPSC 436C
Cloud Computing for Data Science

Stream Processing - Part 2
Spark Streaming

Maryam R.Aliabadi

mraiyata@cs.ubc.ca

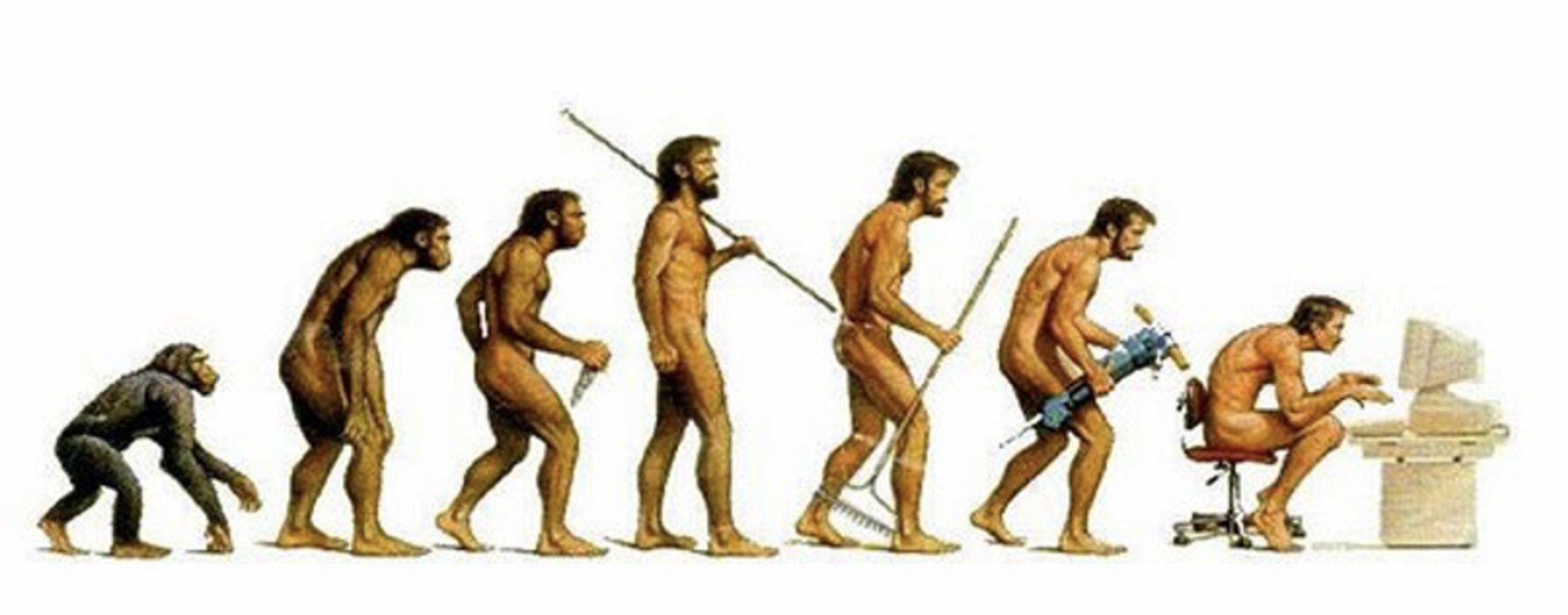
Spring 2024

Last Class review

- Messaging system and partitioned logs
- Decoupling producers and consumers
- Kafka: Distributed, topic oriented, partitioned, replicated log service
- Data stream, unbounded data, tuples
- Event-time vs. processing time
- Micro-batch vs. continues processing (windowing)

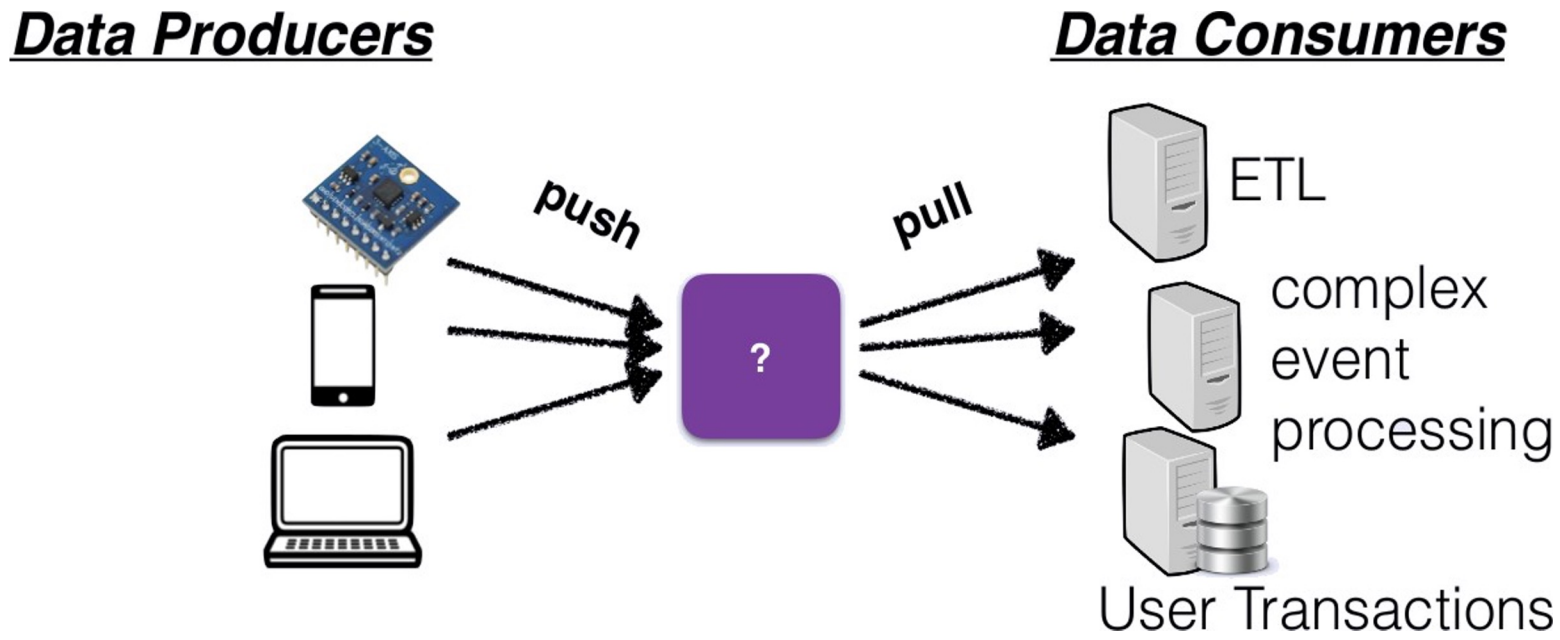
Data Stream Management Systems

- An evolution of traditional data processing, as supported by DBMSs.

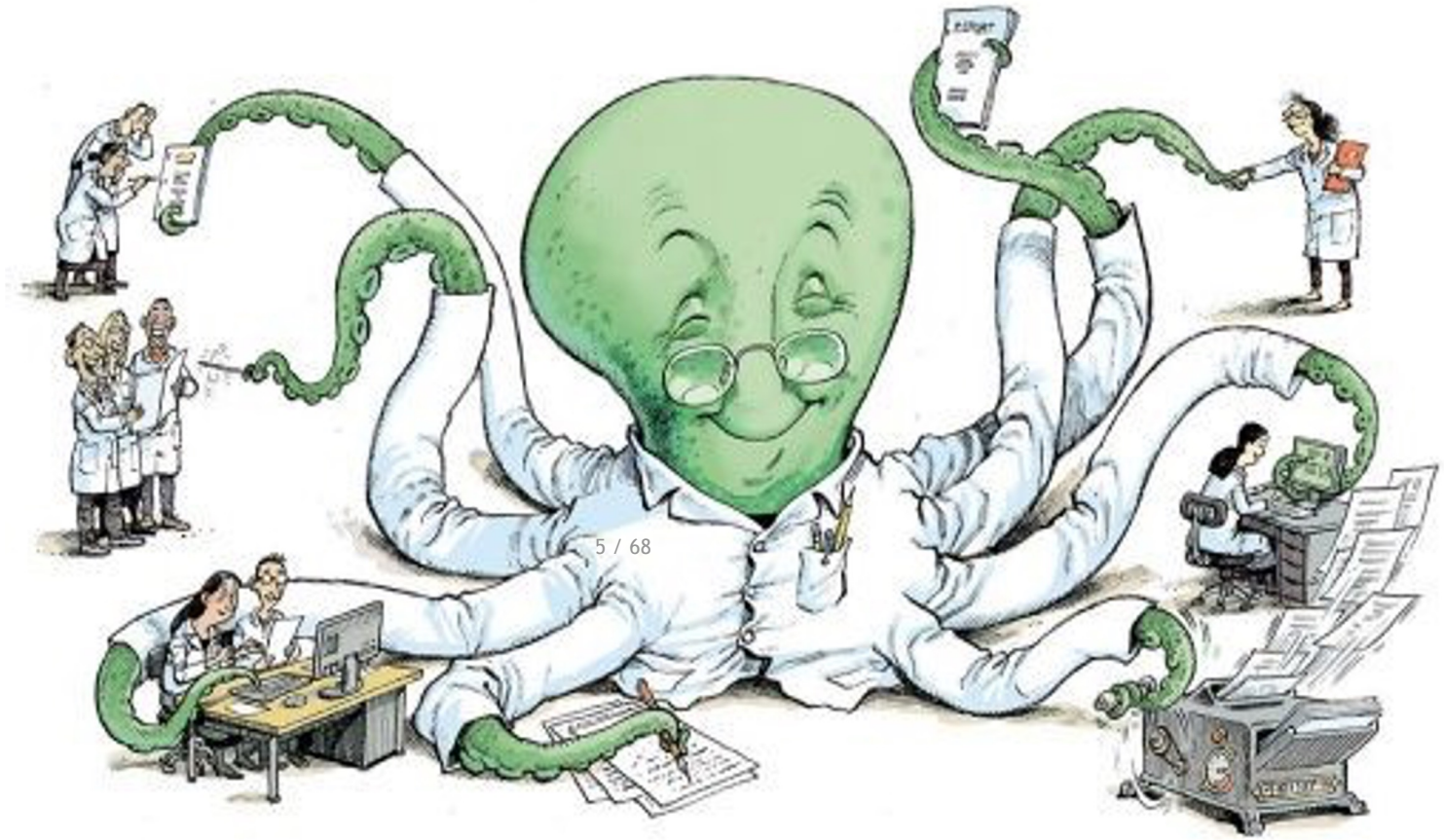


Data streaming

- We need disseminate streams of events from various producers to various consumers.



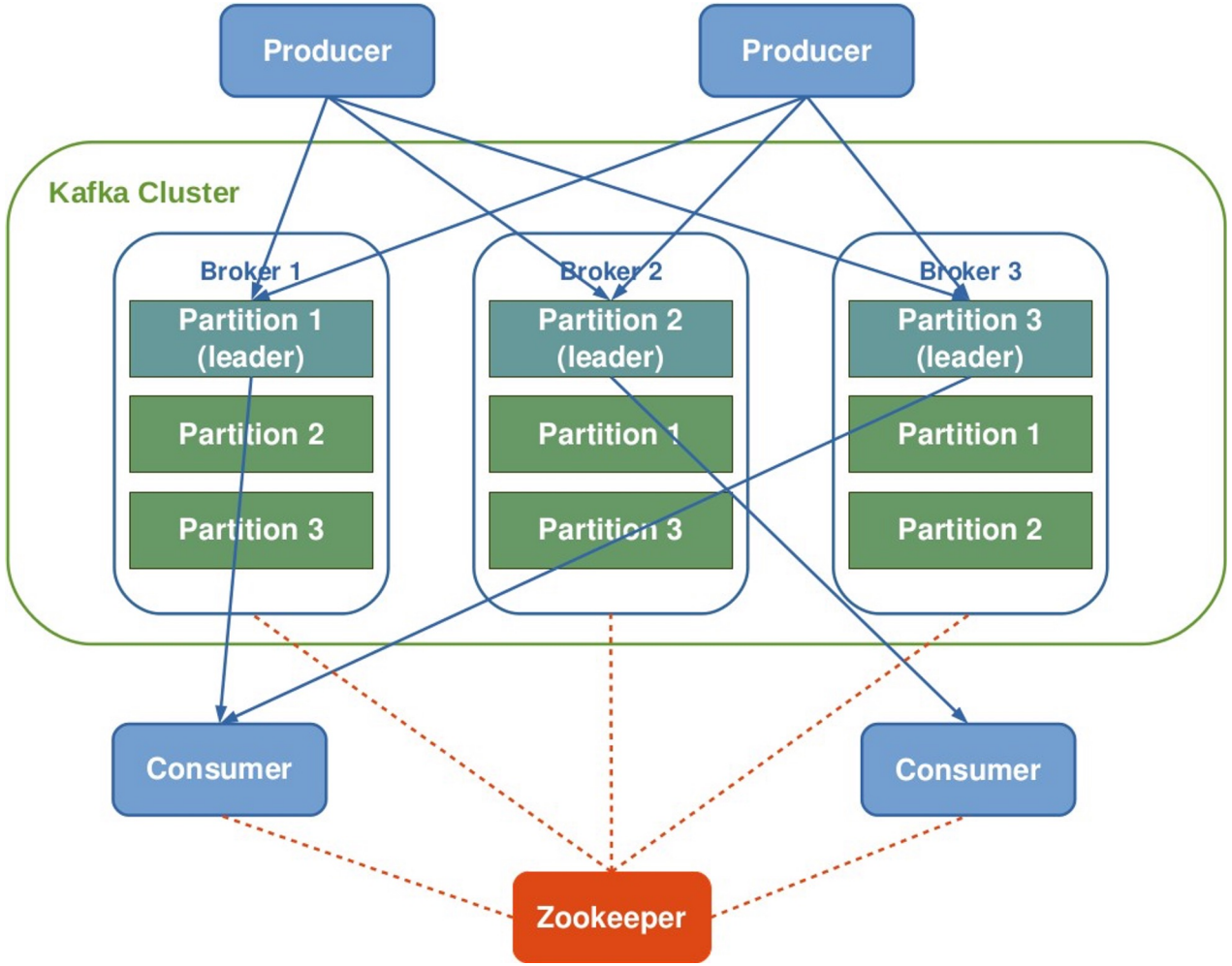
Message Broker



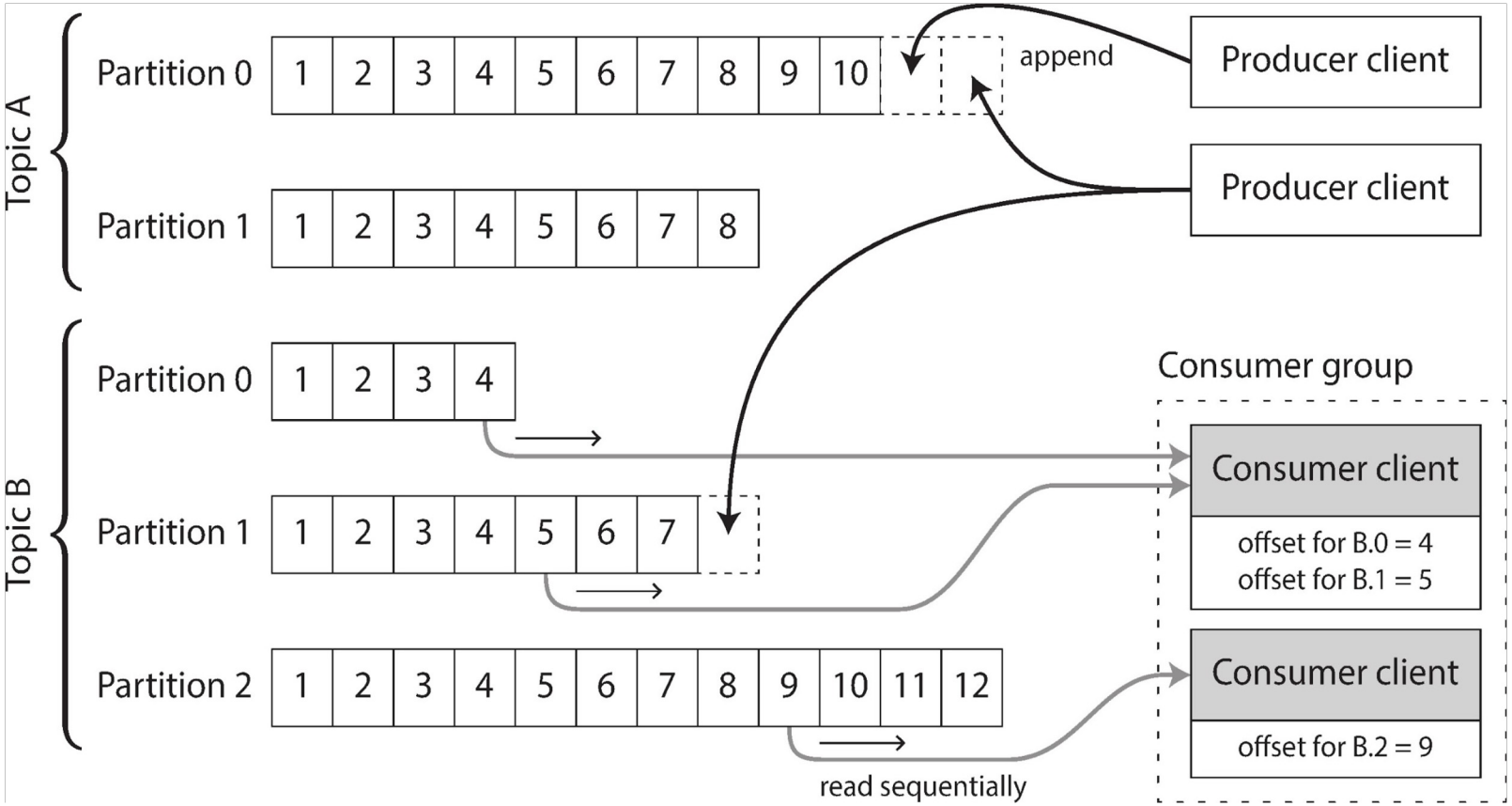
[https://bluesyemre.com/2018/10/16/thousands-of-scientists-publish-a-paper-every-five-days]

Kafka

- Kafka:
  - Distributed,
  - Topic oriented,
  - Partitioned,
  - Replicated log service

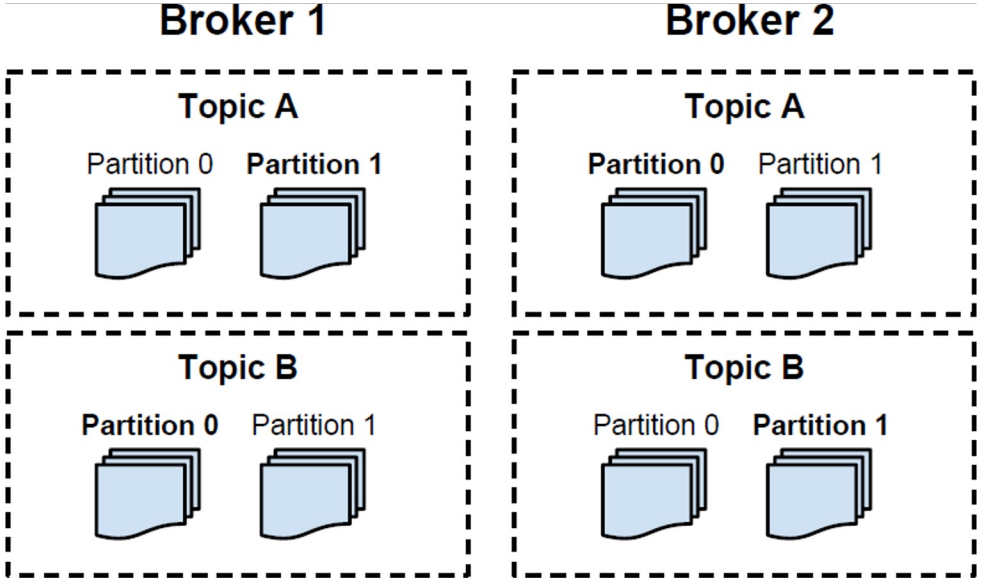


Partition Logs



Fault tolerance

- Partitions of a topic are replicated
- One replica is partition leader
- The broker where the partition leader is seated is also called broker leader- all writes and reads must go to the leader.

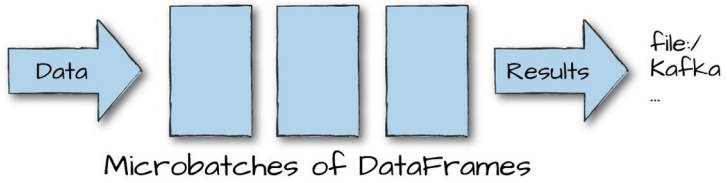


Streaming Data Processing

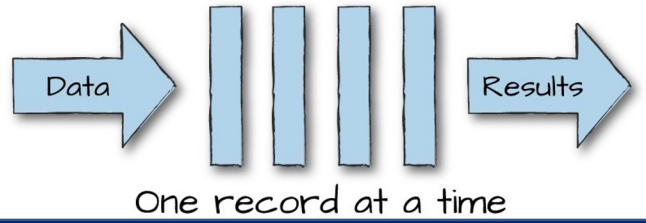
- Data stream is unbound data, which is broken into a sequence of individual tuples.
- A data tuple is the atomic data item in a data stream.
- Can be structured, semi-structured, and unstructured.

Continuous vs. micro-batch processing

- Micro-batch systems
  - Batch engines
  - Slicing up the unbounded data into a sets of bounded data, then process each batch.

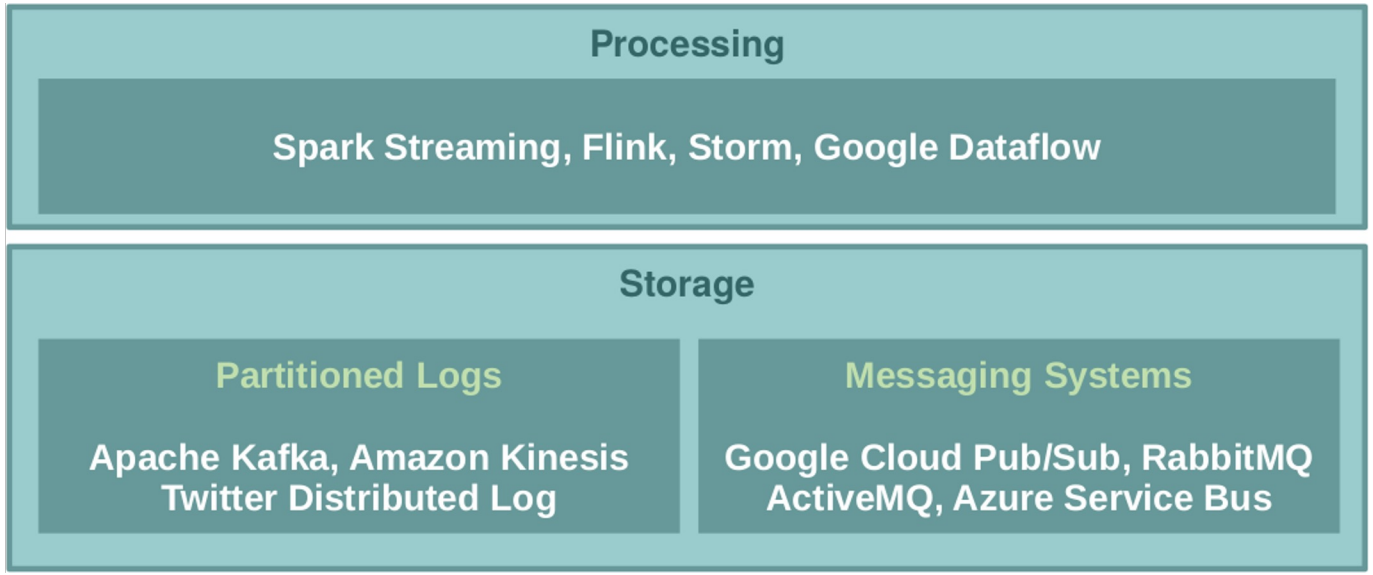


- Continuous processing-based systems
  - Each node in the system continually listens to messages from other nodes and outputs new updates to its child nodes.



Today's Topics

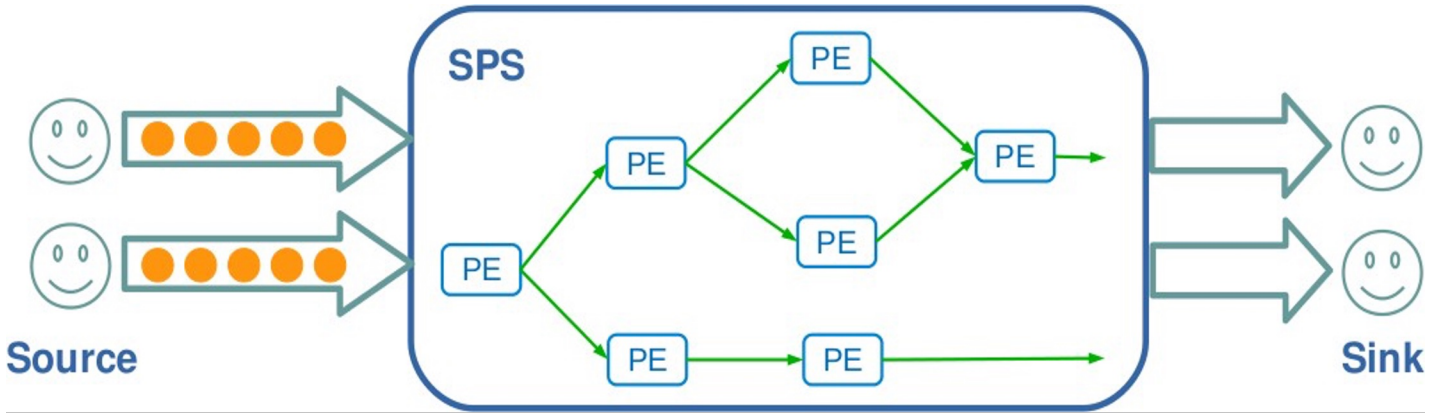
- Streaming data processing model
- Introduction to Spark Streaming



Streaming Data Processing Model

Streaming Data Processing

- The tuples are processed by the application's operators or processing element (PE).
- A PE is the basic functional unit in an application.
  - A PE processes input tuples, applies a function, and outputs tuples.
  - A set of PEs and stream connections, organized into a data flow graph.



PE's State

- A PE can either maintain internal state across tuples while processing them, or process tuples independently of each other.
- Stateful vs. stateless tasks
- Stateless tasks: do not maintain state and process each tuple independently of prior history, or even from the order of arrival of tuples.
- Easily parallelized.
- No synchronization.
- Restart upon failures without the need of any recovery procedure

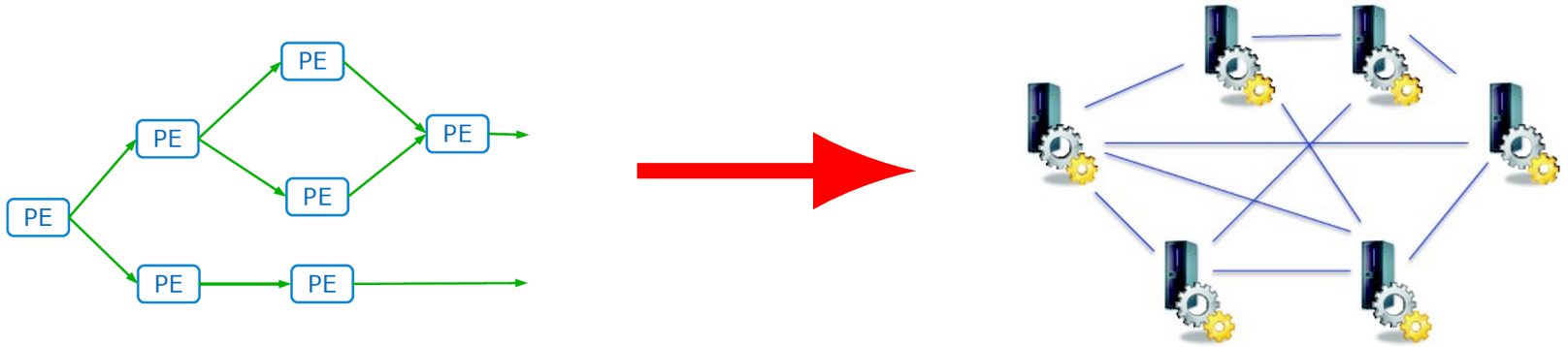
Job and Job Management

- At runtime, an application is represented by one or more jobs.
- Jobs are deployed as a collection of PEs.
- Job management component must identify and track individual PEs, the jobs they belong to, and associate them with the user that instantiated them.

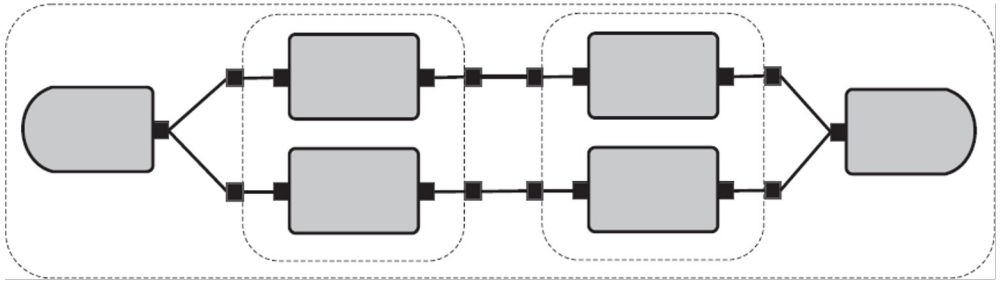
Logical Vs. Physical Plans

- Logical plan: a data flow graph, where the vertices correspond to PEs, and the edges to stream connections.

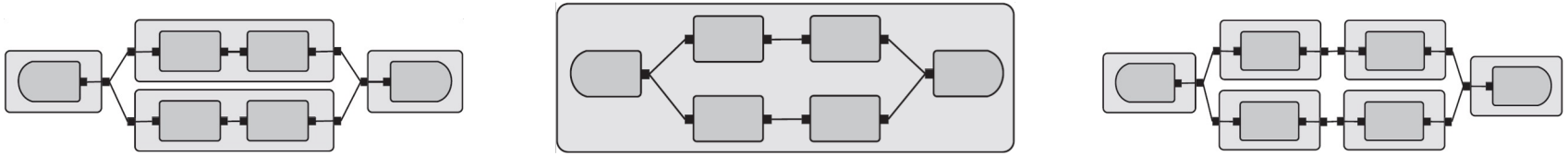
- Physical plan: a data flow graph, where the vertices correspond to OS processes, and the edges to transport connections.



Logical Vs. Physical Plans



Logical plan



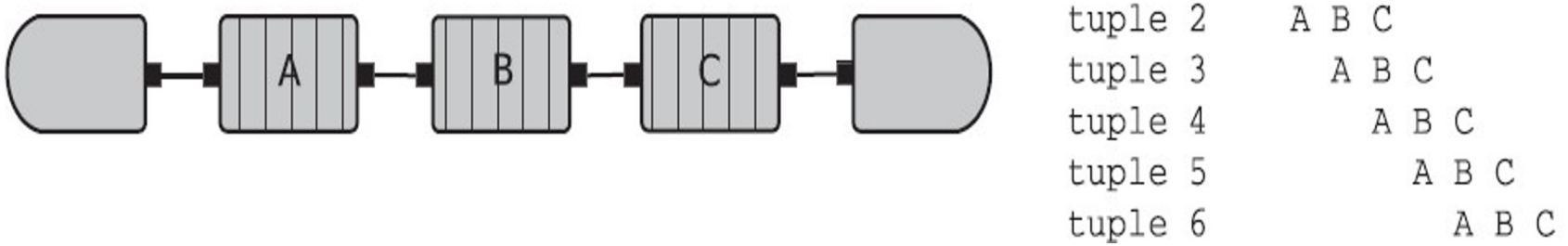
Different physical plans

Parallelization

- How to scale with increasing the number queries and the rate of incoming events?
- Three forms of parallelisms.
  - Pipelined parallelism
  - Task parallelism
  - Data parallelism

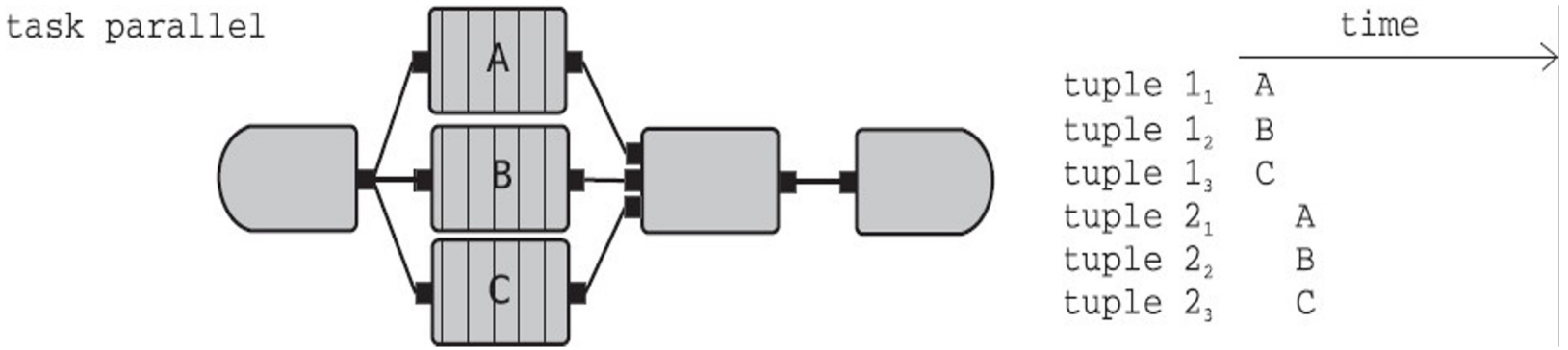
Pipeline Parallelism

- Sequential stages of a computation execute concurrently for different data items.



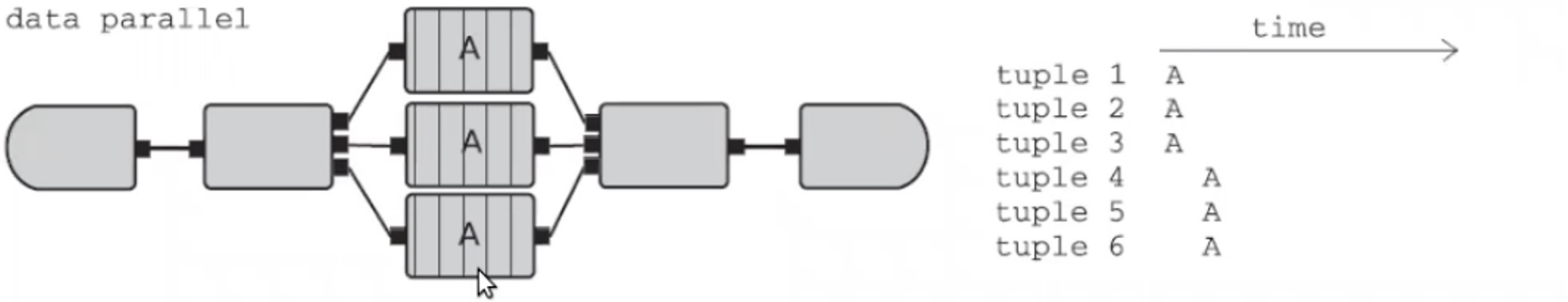
Task Parallelism

- Independent processing stages of a larger computation are executed concurrently on the same or distinct data items.



Data Parallelism

- The same computation takes place concurrently on distinct data items.



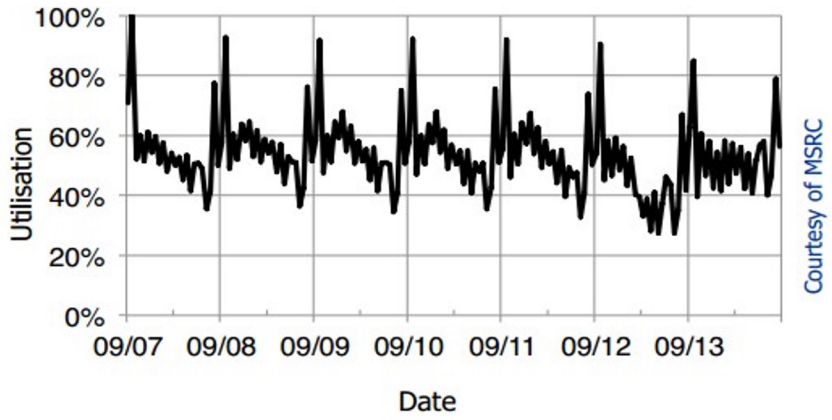
Challenges

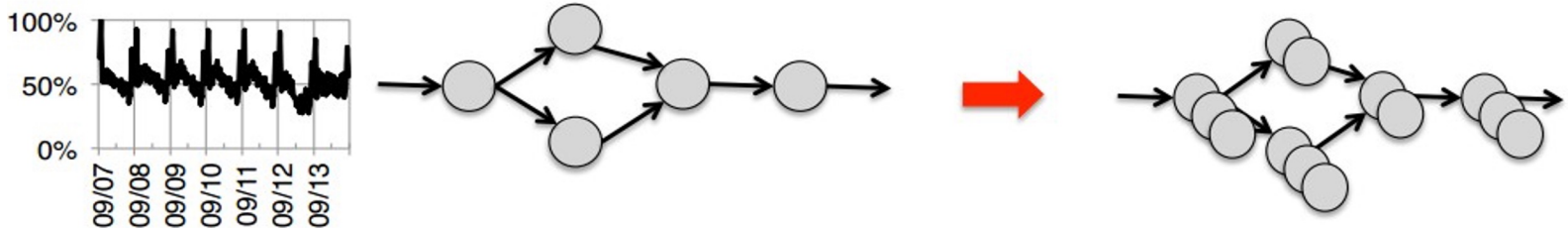
- Elastic data-parallel processing

- Fault-tolerant processing

Elastic Data-Parallel Processing

- Typical stream processing workloads are bursty.
- High and bursty input rates → detect bottleneck + parallelize



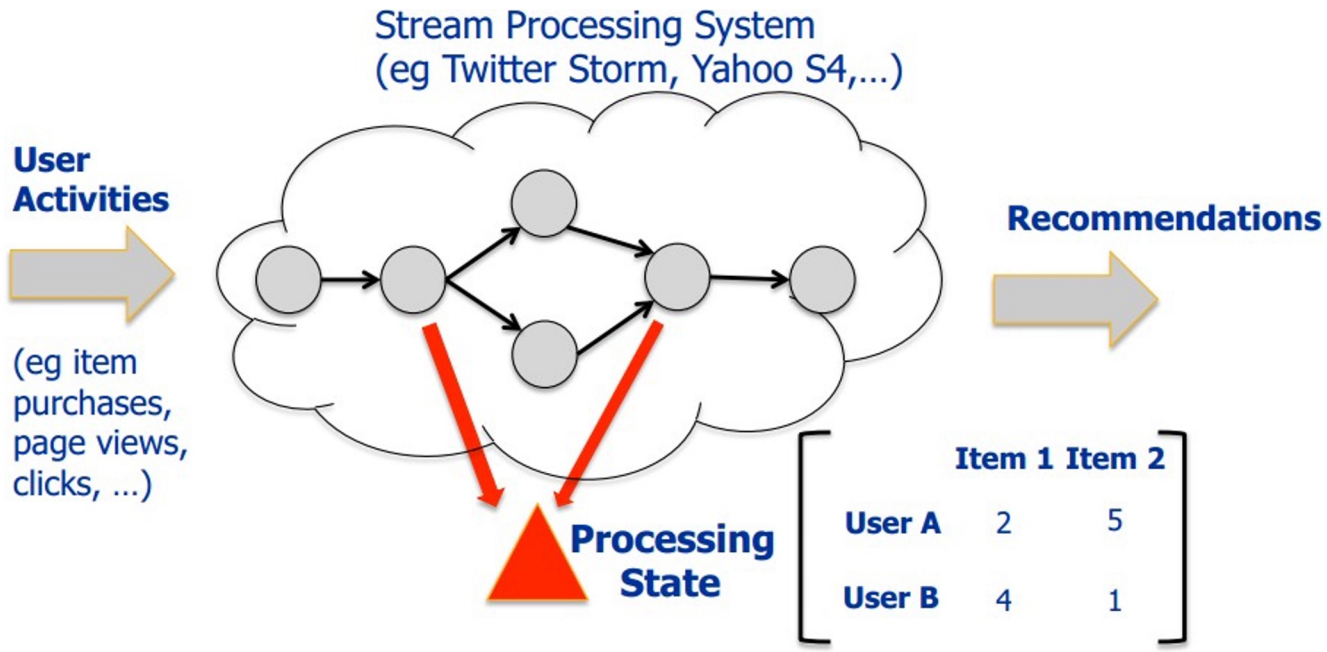


Fault Tolerant Processing

- Large scale deployment → handle node failures.

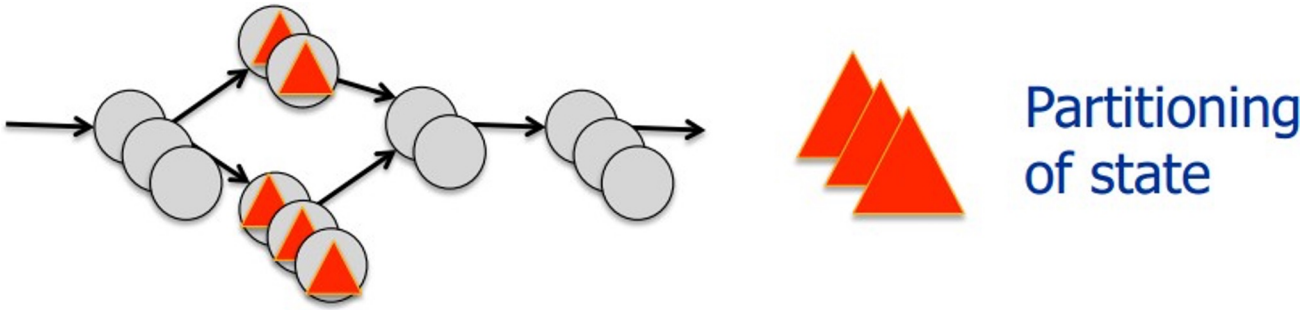
States in Stream Processing

- Many online applications, like machine learning algorithms, require state.

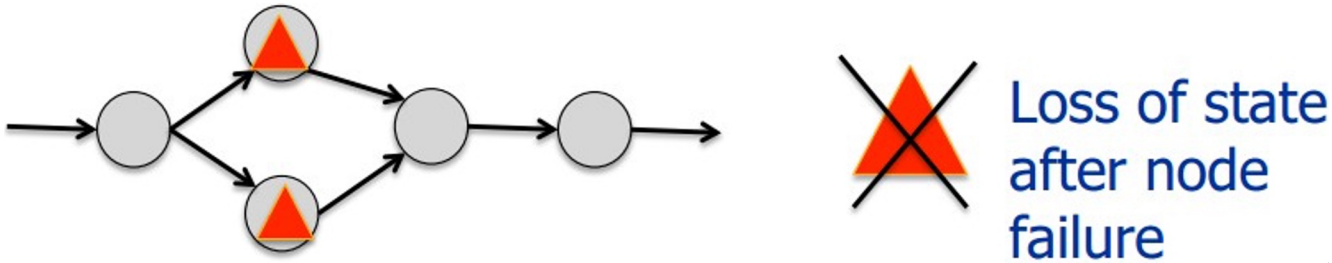


State Compliance Thing

- Dynamic scale out impacts state.

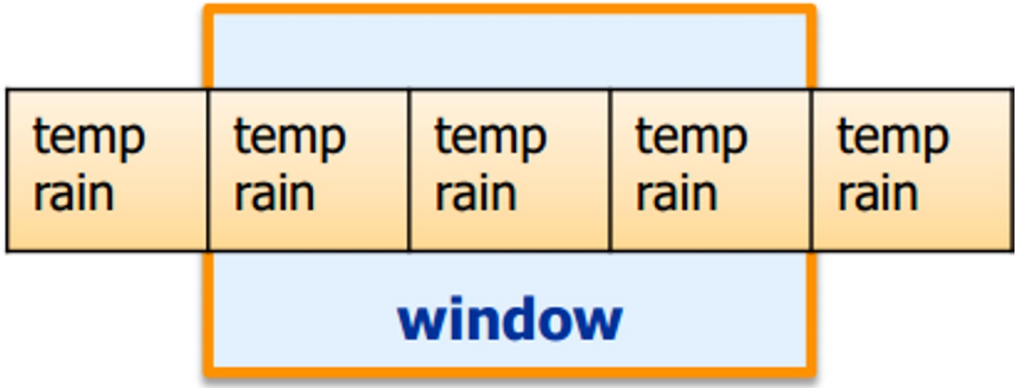


- Recovery from failures.



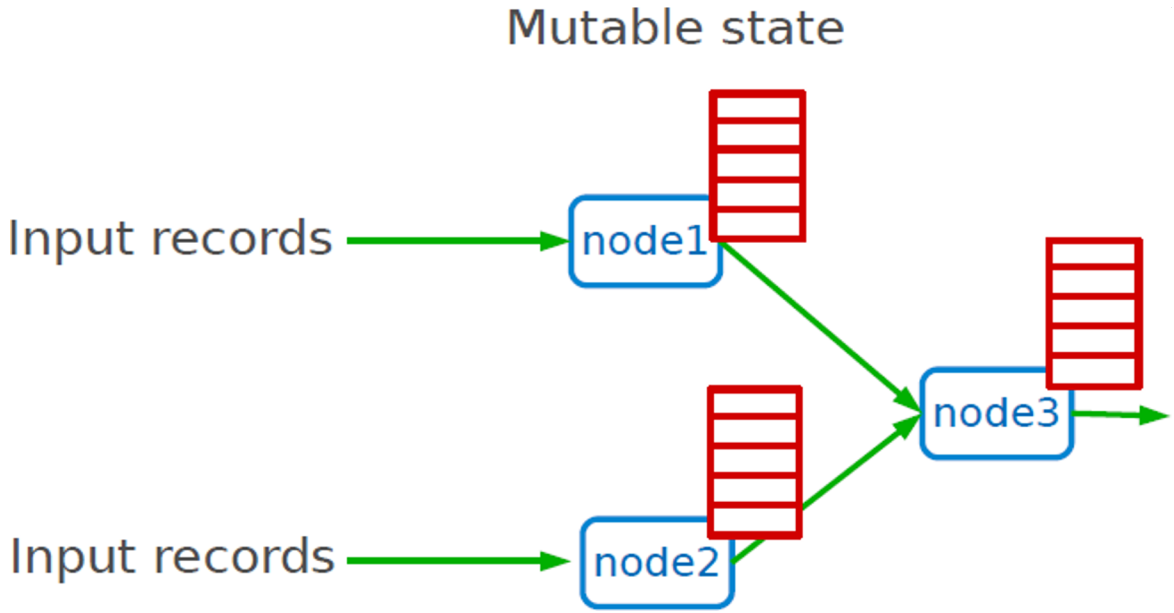
Operator States

- Stateless operators, e.g., filter and map
- Stateful operators, e.g., join and aggregate
- Window operators, use the concept of a finite window of tuples.



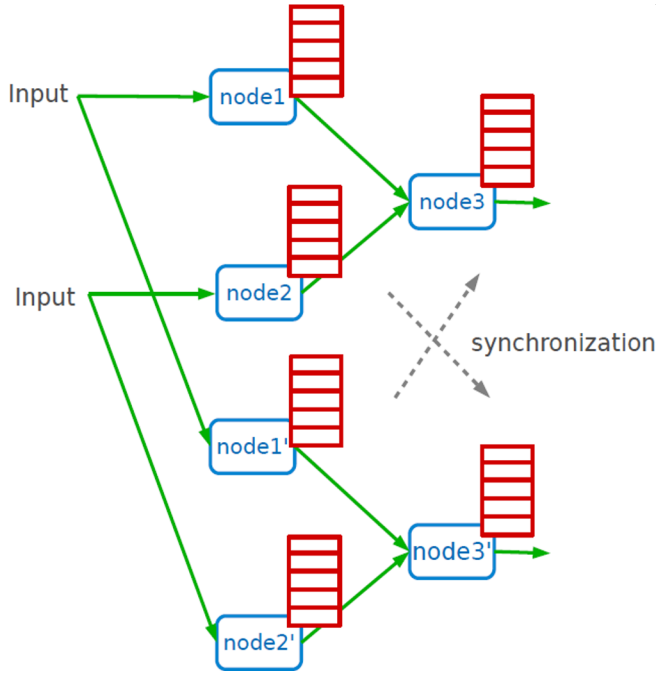
Traditional Streaming Systems

- Record-at-a-time processing model:
  - Each node has mutable state.
  - For each record, updates state and sends new records.
  - State is lost if node dies.

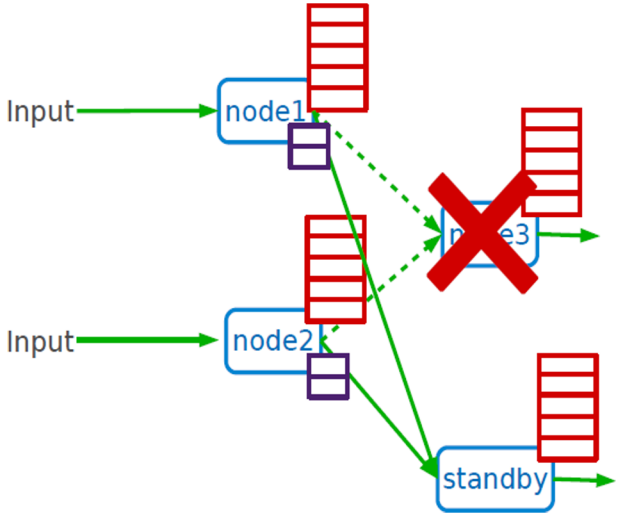


Traditional streaming systems

- Fault tolerance via replication or upstream backup.



Fast recovery, but 2x hardware cost



Only need one standby, but slow to recover

Observation

- Batch processing models for clusters provide fault tolerance efficiently.
- Divide job into deterministic tasks.
- Rerun failed/slow tasks in parallel on other nodes.

Scalable Stream Processing

Spark Streaming

Spark Streaming

- Design considerations:
  - Continues Vs. **Micro-batch processing**
  - Record_at_a_Time API Vs. **Declarative API**

Spark Streaming

- Core idea: Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

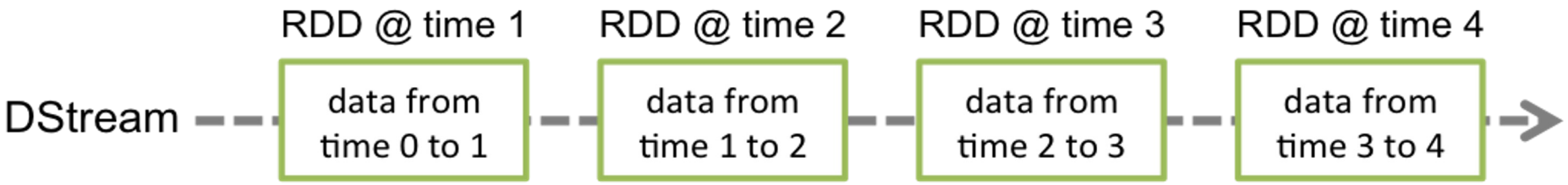
- Core idea: Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.

Spark Streaming

- Core idea: 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 (D-Stream) Processing

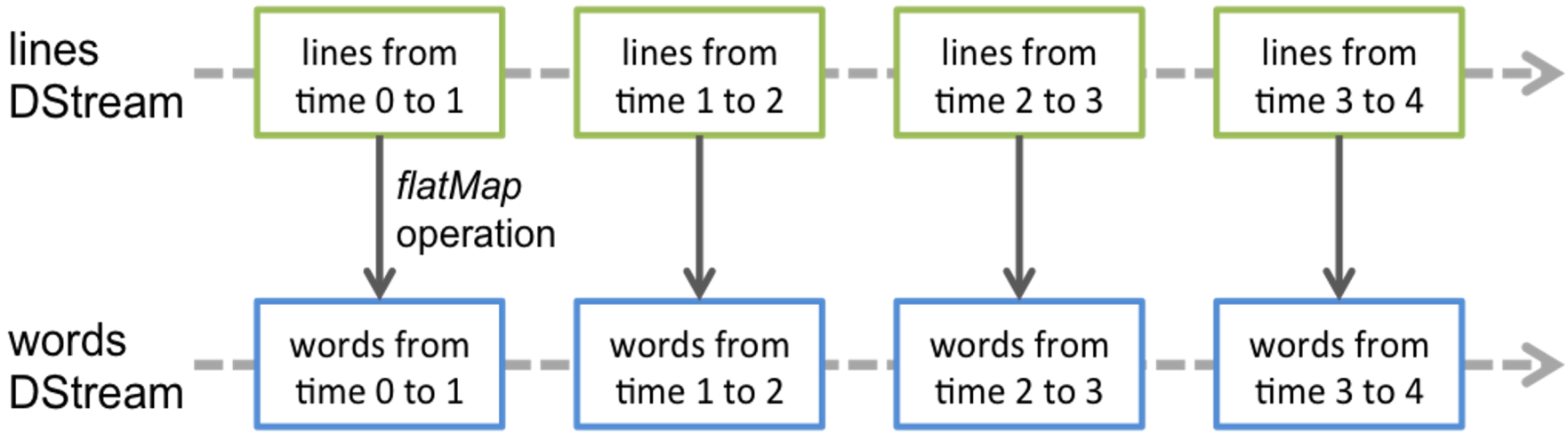
DStream

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...



Dstream

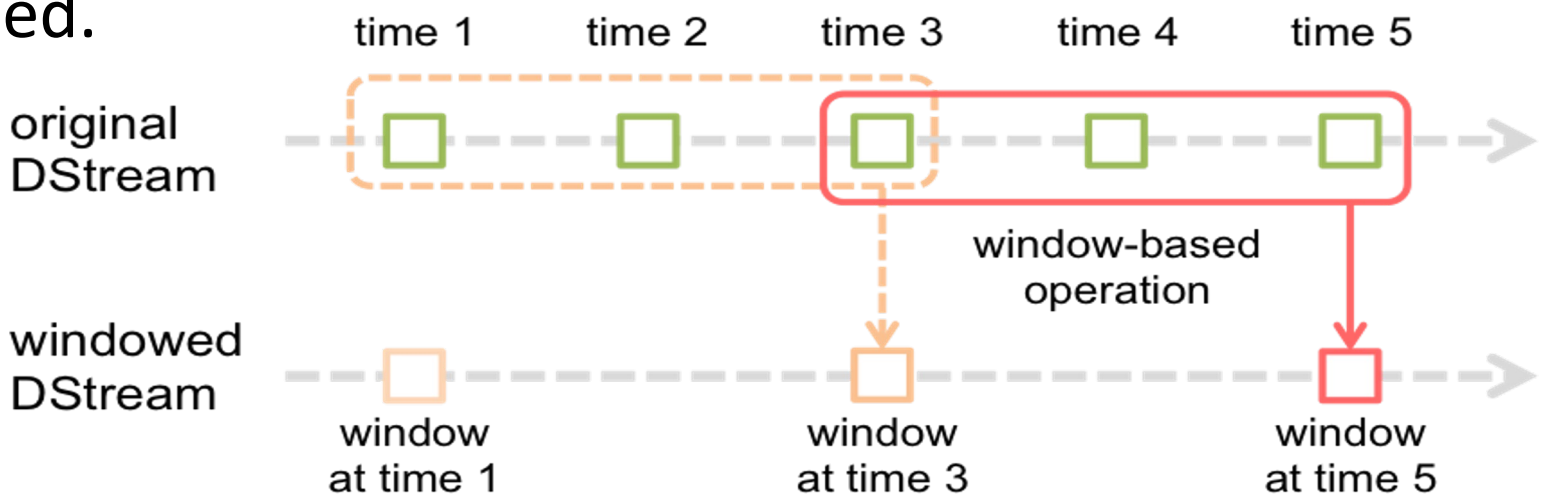
- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless/stateful operations): map, join, ...



Windowing for DStream

- Window operations: group all the records from a sliding window of the past time intervals into one RDD: window, reduceByAndWindow,

- Window length: the duration of the window.
- Slide interval: the interval at which the operation is performed.



StreamingContext

- StreamingContext: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...

- The second parameter, Seconds(1), represents the time interval at which streaming data will be divided into batches.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

Input operations

- Every input Dstream is associated with a receiver object
  - It receives the data from a source and stores it in Spark's memory for processing.

- Basic sources directly available in the StreamingContext API, e.g., file systems, Socket connections

- Advanced sources, e.g., Kafka, Flume, Twitter

Input Operations

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

Input Operations

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

- File stream
  - Reads data from files.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)

streamingContext.textFileStream(dataDirectory)
```

Input Operations

- Socket connection
  - Creates a DStream from text data received over a TCP socket connection.

```
ssc.socketTextStream("localhost", 9999)
```

- File stream
  - Reads data from files.

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)

streamingContext.textFileStream(dataDirectory)
```

- Connectors with external sources, e.g., Twitter, Kafka, Flume, Kinesis,

```
TwitterUtils.createStream(ssc, None)

KafkaUtils.createStream(ssc, [ZK quorum], [consumer group id], [number of partitions])
```

Transformations

- Transformations on DStreams are still lazy!
- DStreams support many of the transformations available on normal Spark RDDs.
- Computation is kicked off explicitly by a call to the start() method.

Transformations

- map: a new DStream by passing each element of the source DStream through a given function.

- reduce: a new DStream of single-element RDDs by aggregating the elements in each RDD using a given function.

- reduceByKey: a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

Spark Streaming Hands-On Exercise 1

- Count frequency of words received on TCP port in one minute with batch intervals of 1 second

Spark Streaming Hands-On Exercise 1

- import the streaming libraries

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
```

- Create StreamingContext with batch interval of 1 second

```
// Create a local StreamingContext with two working threads and batch interval of 1 second.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

Spark Streaming Hands-On Exercise 1

- Create a DStream that represents streaming data from a TCP source.

```
nc -lk 9999
```

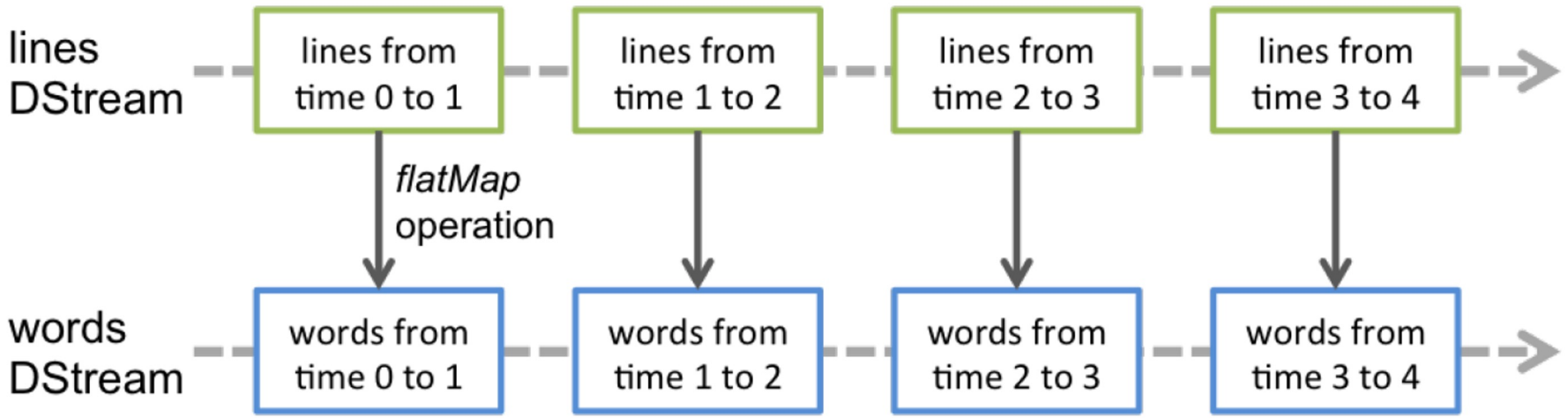
- Specify hostname (e.g., localhost) and port (e.g., 9999)

```
val lines = ssc.socketTextStream("localhost", 9999)
```

Spark Streaming Hands-On Exercise 1

- Use flatMap on the stream to split the records text to words.
- It creates a new DStream.

```
val words = lines.flatMap(_.split(" "))
```

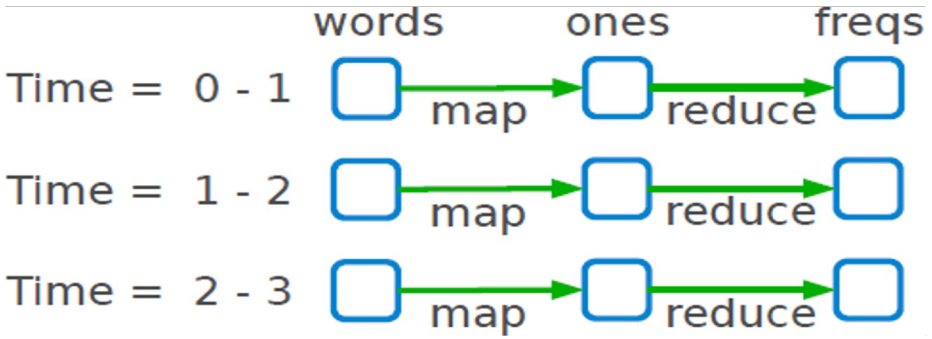


Spark Streaming Hands-On Exercise 1

- Map the words DStream to a DStream of (word, 1).
- Get the frequency of words in each batch of data.

```
val pairs = words.map(word => (word, 1))

val wordCounts = pairs.reduceByKey(_ + _)
```



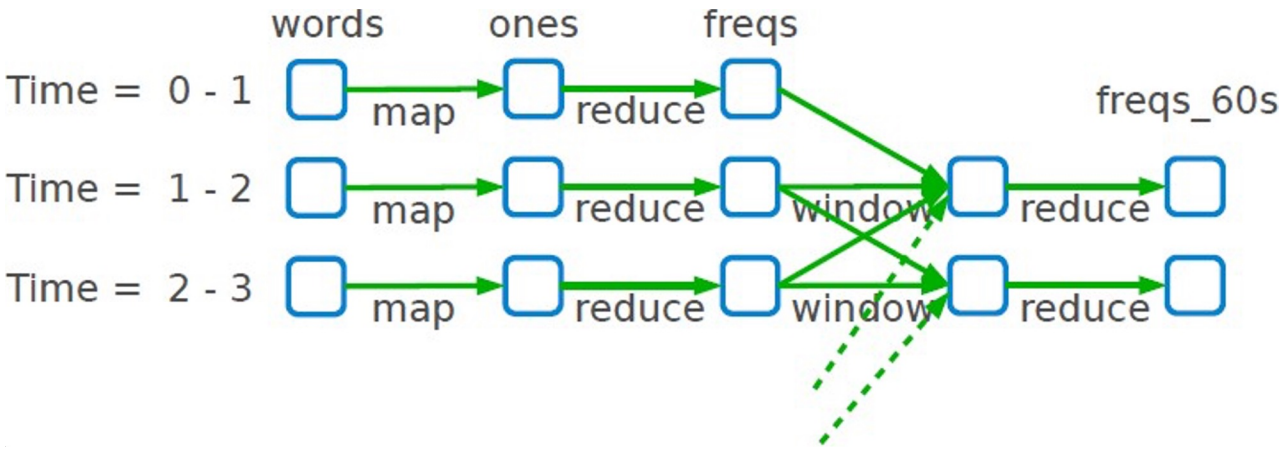
Spark Streaming Hands-On Exercise 1

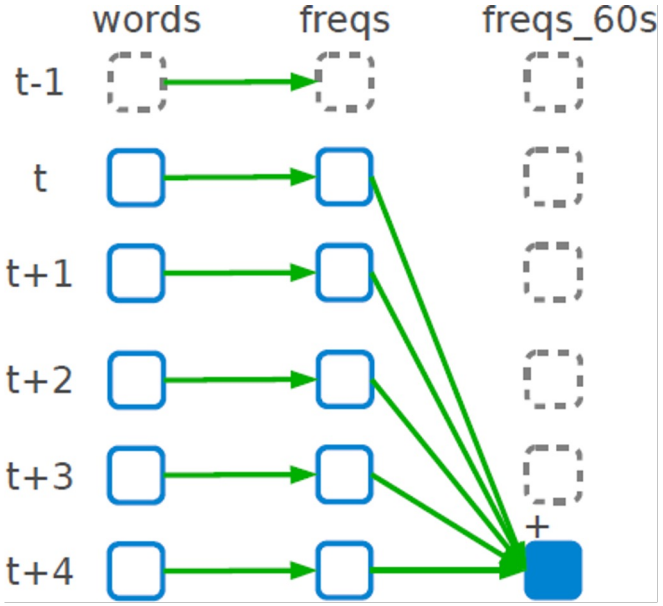
- Count frequency of words received in one minute.

```
val freqs_60s =  freqs.window(Seconds(60), Second(1)).reduceByKey(_ + _)
```

window length

window movement





Spark Streaming Hands-On Exercise 1

- Finally, print the result.

```
Freq_60s.print()
```

Spark Streaming Hands-On Exercise 1

- Start the computation and wait for it to terminate.

```
// Start the computation
ssc.start()

// Wait for the computation to terminate
ssc.awaitTermination()
```

Spark Streaming Hands-On Exercise 1

- Entire stream processing program for word counting

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
```

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
val freqs_60s = freqs.window(Seconds(60), Second(1)).reduceByKey(_ + _)
Freq_60s.print()

ssc.start()
ssc.awaitTermination()
```

Spark Streaming Hands-On Exercise 2

- Utilizing the Twitter API, establish a data stream and determine the ten most frequently occurring hashtags within a 60-second timeframe. You have the option to configure a batch duration of 5 seconds.

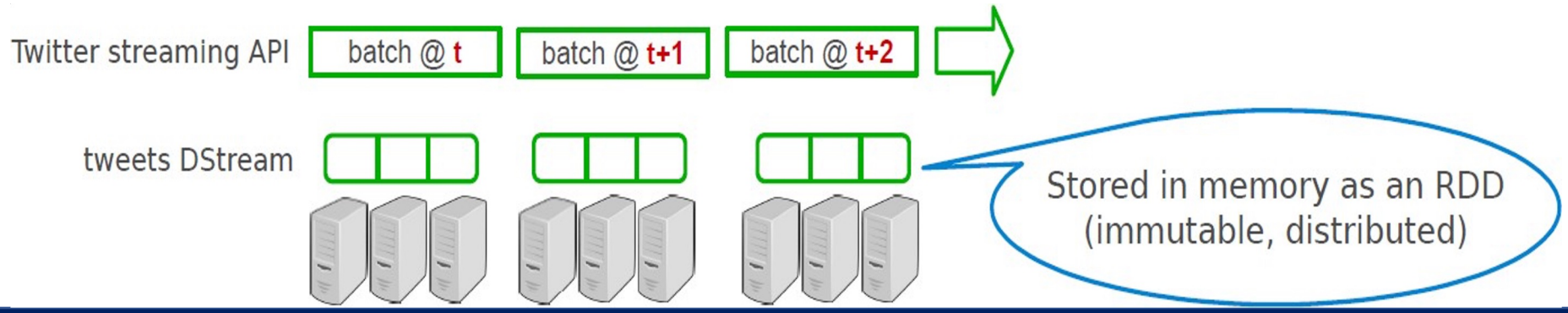
Spark Streaming Hands-On Exercise 2

- Create an StreamingContext for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))
 val tweets = TwitterUtils.createStream(ssc, None)
```

DStream: a sequence of RDD representing a stream of data



Spark Streaming Hands-On Exercise 2

- Create an StreamingContext for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))  val tweets =
TwitterUtils.createStream(ssc, None)
```

- Print the status text of the tweets

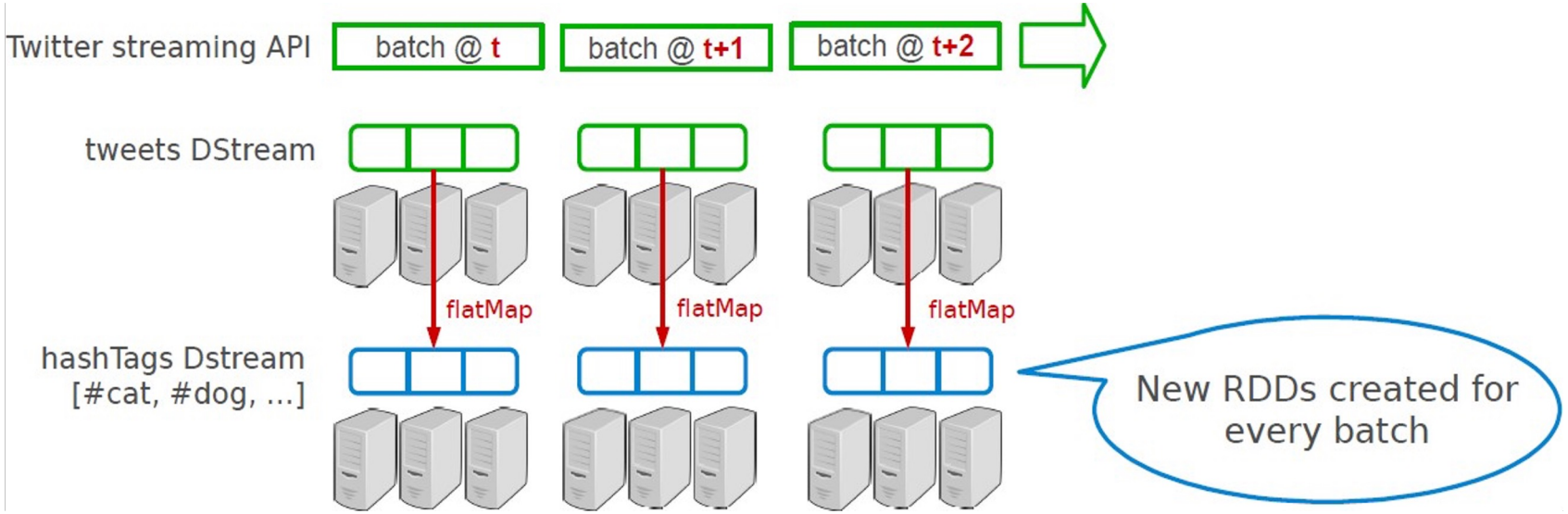
```
val statuses = tweets.map(status => status.getText())  statuses.print()
```

Spark Streaming Hands-On Exercise 2

- Get the stream of hashtags from the stream of tweets

```
val words = statuses.flatMap(status => status.split(" "))
val hashtags = words.filter(word => word.startsWith("#"))
```

transformation: modify data in one DStream to create another DStream



Spark Streaming Hands-On Exercise 2

- Set a path for periodic checkpointing of the intermediate data, and then count the hashtags over a one minute window

```
ssc.checkpoint("/home/sics/temp")
val counts =  hashtags.map(tag =>  (tag, 1))
   .reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(5))  counts.print()
```

Spark Streaming Hands-On Exercise 2

- Set a path for periodic checkpointing of the intermediate data, and then count the hashtags over a one minute window

```
ssc.checkpoint("/home/sics/temp")
val counts =  hashtags.map(tag =>  (tag, 1))
   .reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(5))  counts.print()
```

- Find the top 10 hashtags based on their counts

```
val sortedCounts =  counts.map { case (tag, count) =>  (count, tag) }
   .transform(rdd => rdd.sortByKey(false))
sortedCounts.foreachRDD(rdd =>
   println("\nTop 10  hashtags:\n" +  rdd.take(10).mkString("\n")))
```

Recap

- Spark Streaming

  - Run a streaming computation as a series of very small, deterministic batch jobs.
  - Discretized Stream (D-Stream): sequence of RDDs
  - Operators: Transformations (stateless, stateful, and window) and output operations

Next class:

Graph Processing