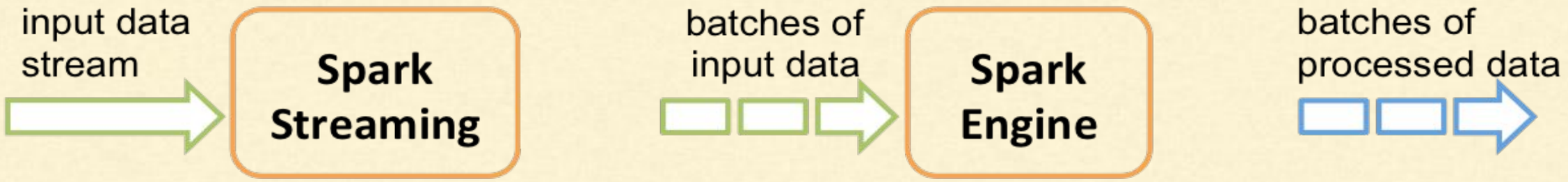
Welcome to Spark Streaming

Spark Streaming - Introduction

- Extension of the core Spark API
- Enable Stream Processing
  - Scalable
  - High-throughput
  - Fault-tolerant

Spark Streaming - Introduction

Spark Streaming - Workflow

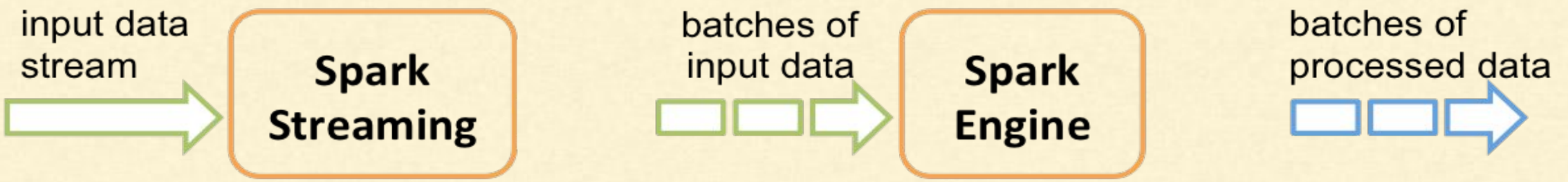


Spark Streaming receives input data streams

Divides data into batches

These batches are processed by Spark engine

Spark Streaming - Workflow



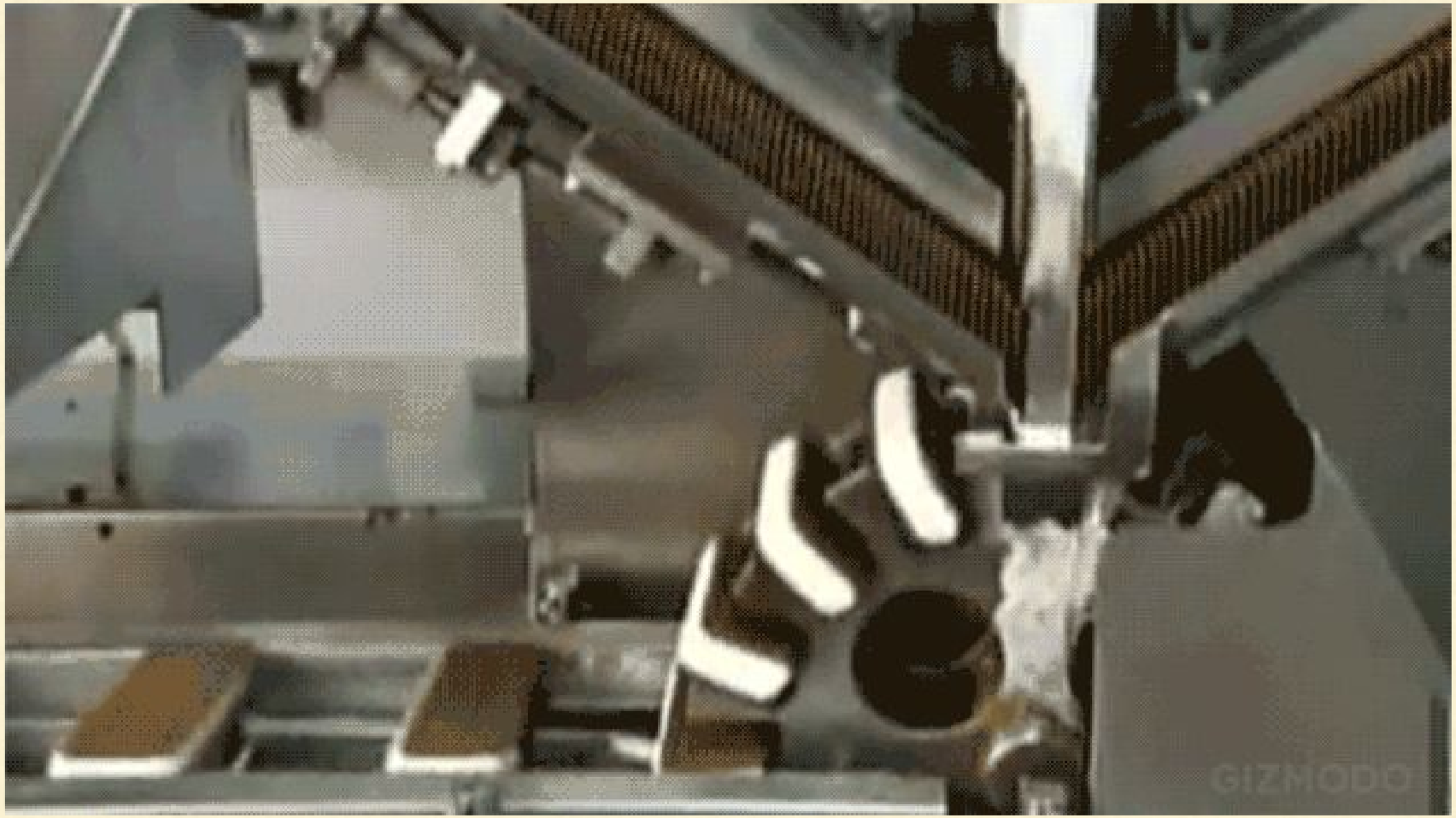
Spark Streaming receives input data streams

Divides data into batches

These batches are processed by Spark engine

Spark Streaming - Workflow

Continuous Stream



Batches of Data

Spark Streaming - Use Case

Real-time Analytics

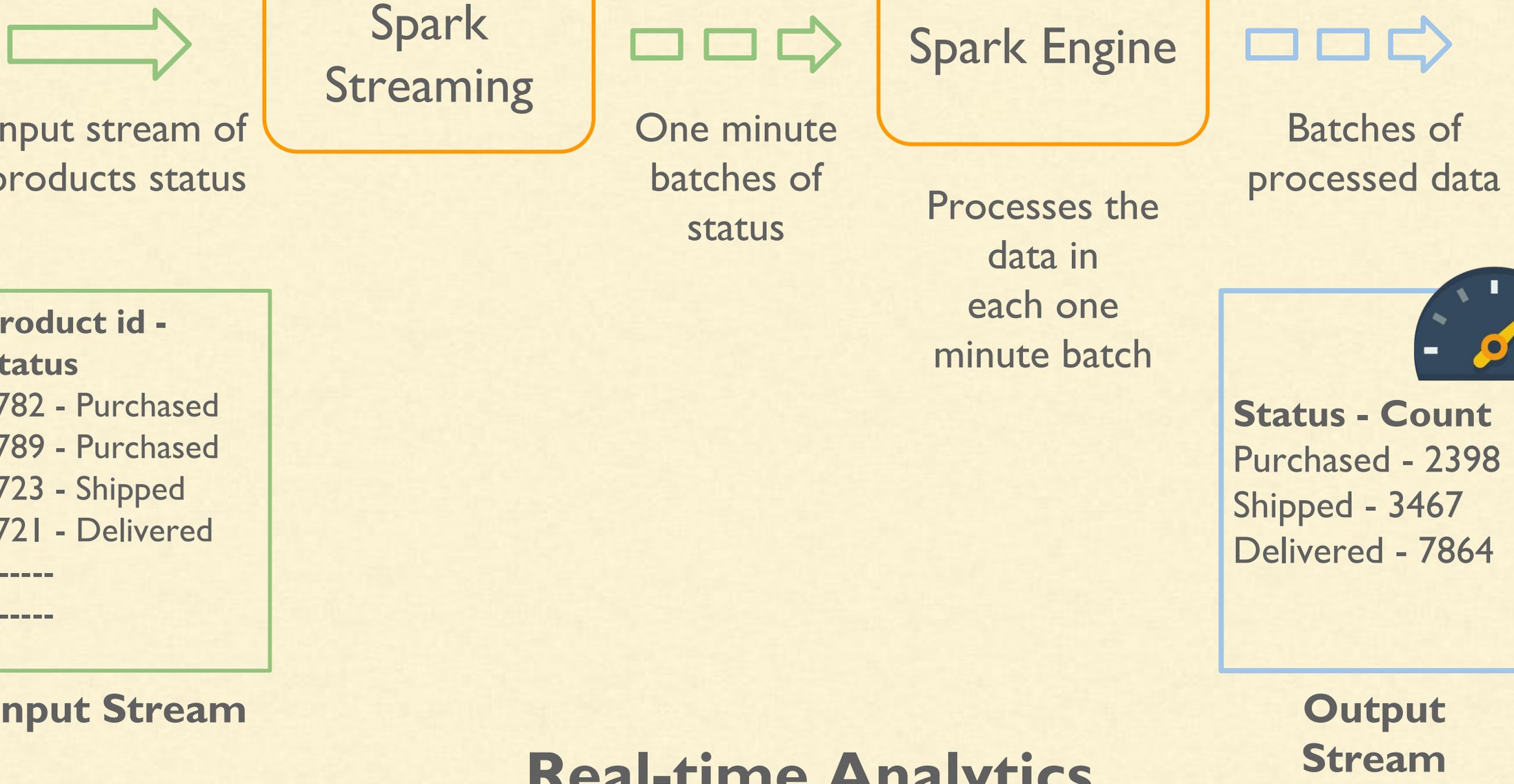
Spark Streaming - Use Case

Real-time Analytics

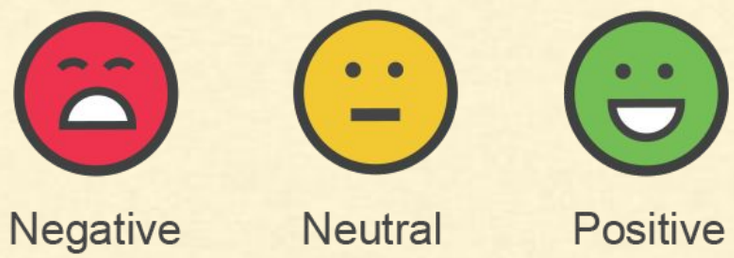
Problem: Build real-time analytics dashboard with information on how many products are getting

- Purchased
- Shipped
- Delivered every minute

Spark Streaming - Use Case - Ecommerce



Spark Streaming - Use Case



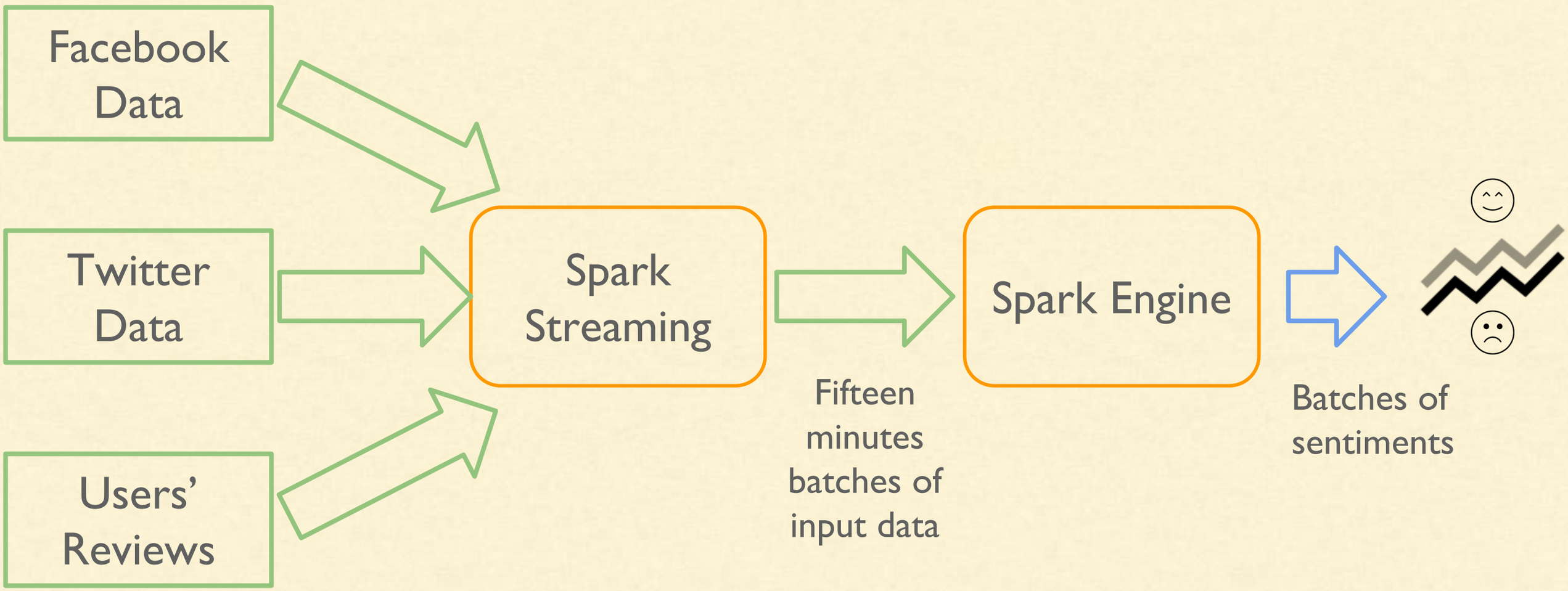
Real-time Sentiment Analysis

Spark Streaming - Use Case

Real-time Sentiment Analysis

Problem: Build Real-time sentiment analytics system to find out sentiment of users every fifteen minute by analyzing data from various sources such as Facebook, Twitter, users' feedback, comments and reviews

Spark Streaming - Use Case



Real-time Sentiment Analysis

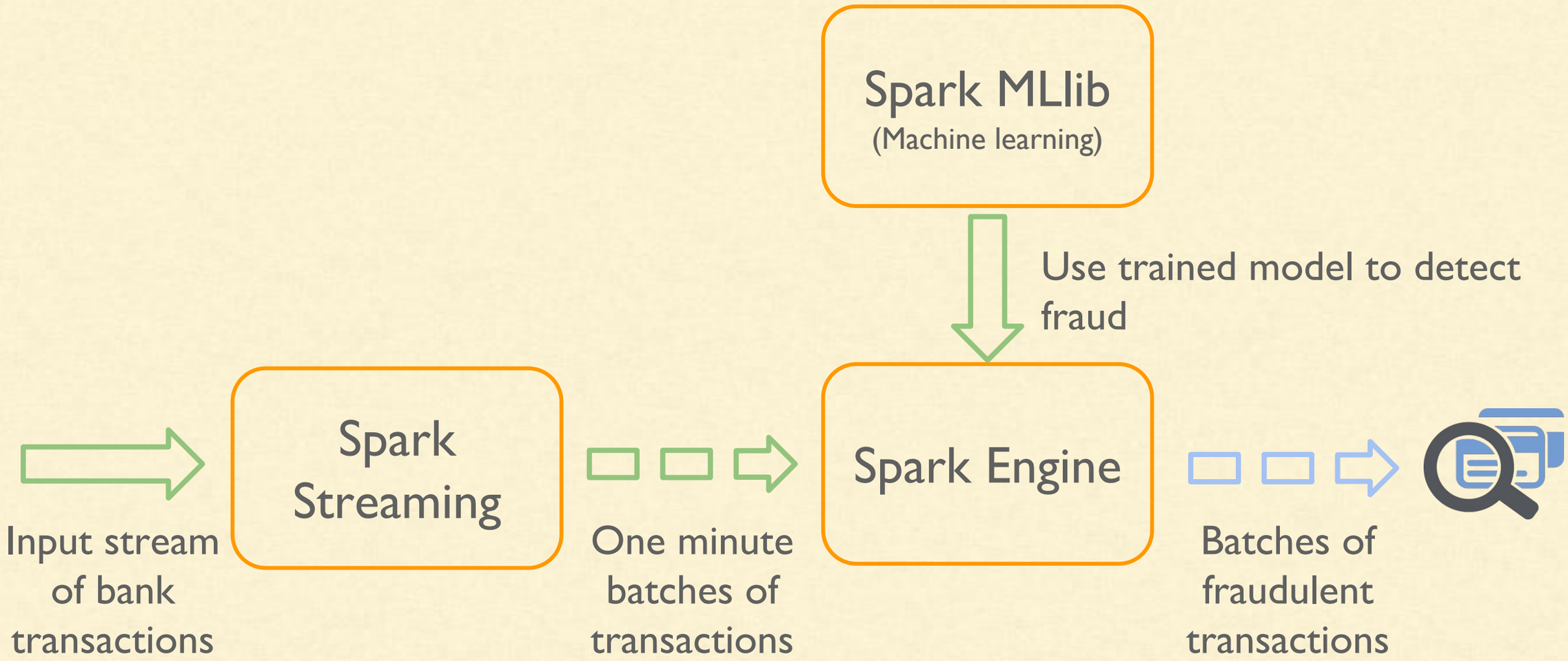
Spark Streaming - Use Case

Real-time Fraud Detection

Problem: Build a real-time fraud detection system for a bank to find out the fraudulent transactions

Spark Streaming - Use Case

Real-time Fraud Detection



Spark Streaming - Use Cases - Uber

UBER

Uber uses Spark Streaming for real-time telemetry analytics

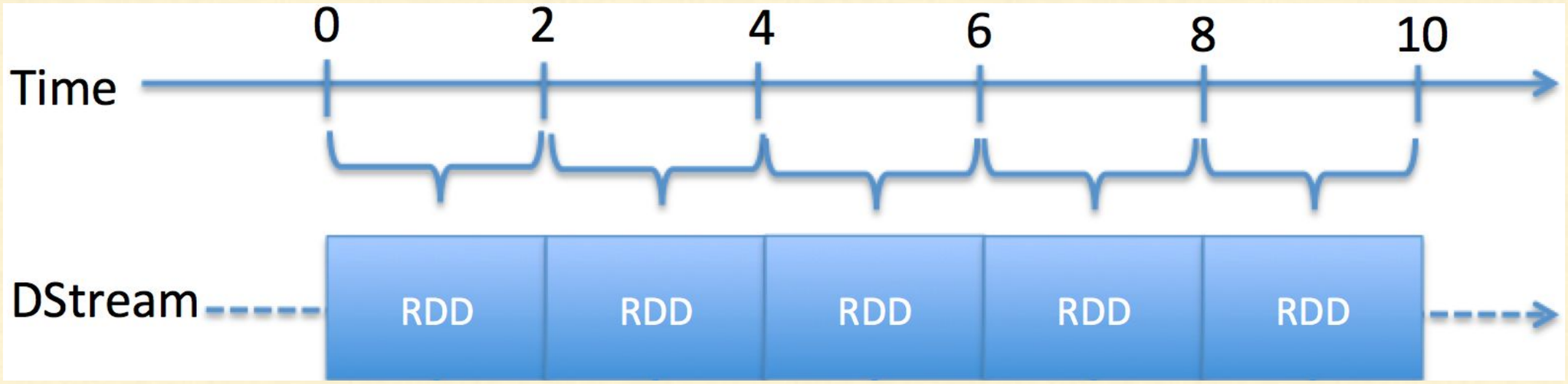
Spark Streaming - Use Cases - Pinterest

Pinterest uses Spark Streaming to provide immediate insight into how users are engaging with pins across the globe in real-time

Spark Streaming - Use Cases - Netflix

Netflix uses Spark Streaming to provide movie recommendations to its users

Spark Streaming - DStream



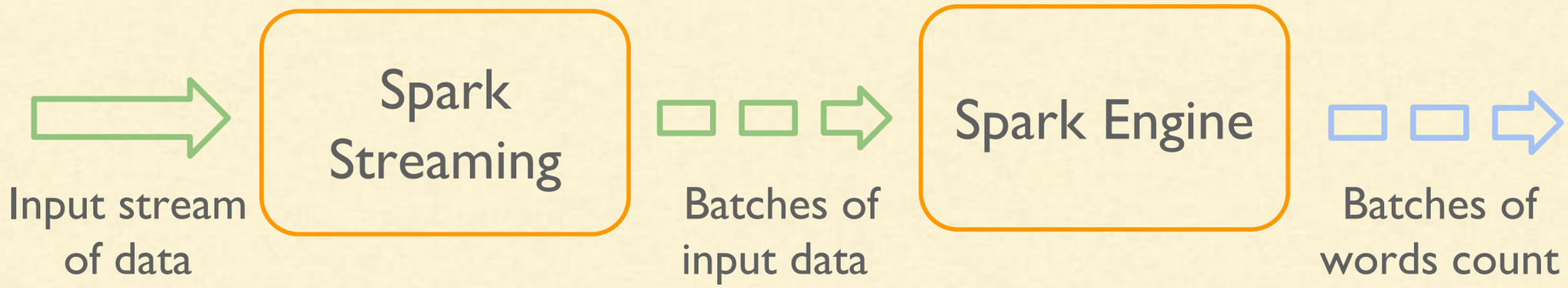
Discretized stream or DStream:

- Represents continuous stream of data
- Can be created
  - From input data streams
  - Or from other DStreams
- Is represented as a sequence of RDDs internally
- Is used to write Spark Streaming programs

Spark Streaming - Hands-on - Word Count

Problem: Count the number of words in the text data received from a server listening on a host and a port

Spark Streaming - Hands-on - Word Count



The quick
brown fox
jumps over
the lazy
dog

Input Stream

(lazy,1)
(dog,1)
(over,1)
(The,1)
(jumps,1)
(quick,1)
(the,1)
(brown,1)
(fox,1)

Output Stream

Spark Streaming - Hands-on - Word Count

Spark Streaming code listens to this host and port

Server generates data on a host and a port. This server will work like a producer

Spark Streaming - Word Count - Code

- Sample code is at CloudxLab GitHub repository

  https://github.com/singhabhinav/cloudxlab/blob/master/spark/examples/streaming/word_count/word_count.scala

Spark Streaming - Word Count - Code

- Clone the repository

  git clone https://github.com/singhabhinav/cloudxlab.git ~/cloudxlab

- Or update the repository if already cloned

  cd ~/cloudxlab && git pull origin master

Spark Streaming - Word Count - Code

- Go to word_count directory

  cd ~/cloudxlab/spark/examples/streaming/word_count

- There are word_count.scala and word_count.py files having Scala and Python code for the word count problem
- Open word_count.scala

  vi word_count.scala

- Copy the code and paste in spark-shell

Spark Streaming - Word Count - Producer

Create the data producer

- Open a new web console
- Run the following command to start listening to 9999 port

  nc -lk 9999
- Whatever you type here would be passed to a process connecting at 9999 port

Spark Streaming - Word Count - Producer

Type in data

- The quick
brown fox
jumps over
the lazy
dog

- my first Spark
Streaming code

Spark Streaming - Word Count - SBT

- *cd ~/cloudxlab/spark/examples/streaming/word_count_sbt*
- # Build the JAR
- *sbt package*
- # Run the JAR
- *spark-submit --class "WordCount" --master "local[2]"*

  target/scala-2.10/word-count_2.10-1.0.jar

Spark Streaming - Word Count - Python

- cd ~/cloudxlab/spark/examples/streaming/word_count
- # Run the code
- *spark-submit --master "local[2]" word_count.py*

Spark Streaming - Adding Dependencies

- *libraryDependencies += "org.apache.spark" % "spark-streaming_2.10" % "1.5.2"*
- 1.5.2 is the Spark version
- You can change 1.5.2 to your Spark version

Spark Streaming - Adding Dependencies

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10 [Amazon Software License]
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

Spark Streaming - Adding Dependencies

For python, it is better to download the jars binaries from the maven repository directly

g:"org.apache.spark" AND v:"1.5.1"

SEARCH

New: About Central

Advanced Search | API Guide | Help

Search Results

< 1 2 3 > displaying 1 to 20 of 57

GroupId	ArtifactId	Version	Updated	Download
org.apache.spark	spark-streaming-kinesis-asl-assembly_2.11	1.5.1	24-Sep-2015	pom jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-streaming-flume_2.10	1.5.1	24-Sep-2015	pom jar javadoc.jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-yarn_2.11	1.5.1	24-Sep-2015	pom jar javadoc.jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-streaming-kafka-assembly_2.10	1.5.1	24-Sep-2015	pom jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-streaming-flume-assembly_2.11	1.5.1	24-Sep-2015	pom jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-network-shuffle_2.10	1.5.1	24-Sep-2015	pom jar javadoc.jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-parent_2.10	1.5.1	24-Sep-2015	pom tests.jar

Spark Streaming - A Quick Recap

1. First initialize the StreamingContext. It is initialized in ***ssc*** variable in our code
2. Define the input sources by creating input DStreams. It is defined in ***lines*** variable in our code
3. Define the streaming computations by applying transformations to DStreams. It is defined in ***words***, ***pairs*** and ***wordCounts*** variables in our code

Spark Streaming - A Quick Recap

4. Start receiving data and processing it using streamingContext.start().

5. Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().

6. The processing can be manually stopped using streamingContext.stop().

Spark Streaming - Running Locally

For running locally,

- Do not use “local” or “local[1]” as the master URL.
  - As it uses only one thread for receiving the data
  - Leaves no thread for processing the received data
- So, Always use “local[n]” as the master URL , where n > no. of receivers

Spark Streaming - Running on Cluster

For running on cluster

- Number of cores allocated must be > no. of receivers
- Else system will receive data, but not be able to process it

Apache Kafka - Introduction

Kafka is used for building real-time data pipelines and streaming applications.

Apache Kafka - Introduction

- Distributed publish (write) - subscribe (consume) messaging system, similar to a message queue or enterprise messaging system

Apache Kafka - Introduction

- Distributed publish-subscribe messaging system, similar to a message queue or enterprise messaging system
- Originally developed at LinkedIn and later on became part of Apache project

Apache Kafka - Introduction

- Distributed publish-subscribe messaging system, similar to a message queue or enterprise messaging system
- Originally developed at LinkedIn and later on became part of Apache project.
- It is fast, scalable, durable and distributed by design

Apache Kafka - Key Concepts

- Runs on a cluster of one or more servers. Each node in the cluster is called **broker**

Apache Kafka - Key Concepts

- Runs on a cluster of one or more servers. Each node in the cluster is called **broker**.
- Stores records in the categories called **topics**

Apache Kafka - Key Concepts

- Runs on a cluster of one or more servers. Each node in the cluster is called **broker**
- Stores records in the categories called **topics**
- Kafka topics are divided into a number of partitions

Apache Kafka - Key Concepts

- Runs on a cluster of one or more servers. Each node in the cluster is called **broker**
- Stores records in the categories called **topics**
- Kafka topics are divided into a number of partitions
- Partitions split the data in a particular topic across multiple brokers

Apache Kafka - Key Concepts

- Each topic partition in Kafka is replicated “n” times where “n” is the replication factor of topic

Apache Kafka - Key Concepts

- Each topic partition in kafka is replicated “n” times where “n” is the replication factor of topic
  - Automatic failover to replicas when a server in the cluster fails

Apache Kafka - Key Concepts

- Each topic partition in kafka is replicated “n” times where “n” is the replication factor of topic
  - Automatic failover to replicas when a server in the cluster fails
- There can be multiple topics in the Kafka cluster

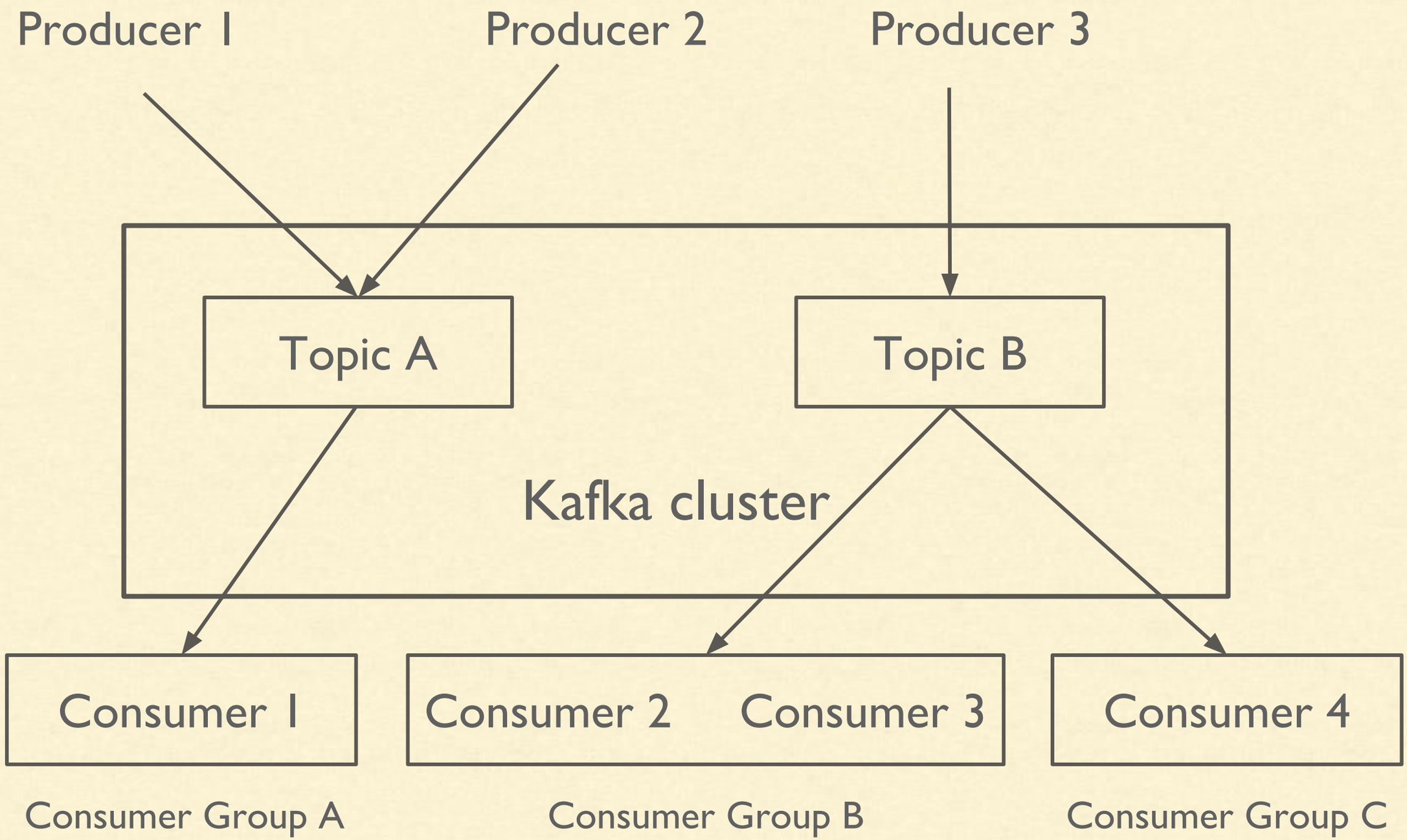
Apache Kafka - Key Concepts

- Each topic partition in kafka is replicated “n” times where “n” is the replication factor of topic
  - Automatic failover to replicas when a server in the cluster fails
- There can be multiple topics in the Kafka cluster
  - One topic for website activity tracking

Apache Kafka - Key Concepts

- Each topic partition in kafka is replicated “n” times where “n” is the replication factor of topic
  - Automatic failover to replicas when a server in the cluster fails
- There can be multiple topics in the Kafka cluster
  - One topic for website activity tracking
  - Another topic for storing application performance metrics

Apache Kafka - Producers and Consumers



Apache Kafka - Hands-on

Gist -

https://gist.github.com/singhabhinav/1003a2a473l8d85a222b4f51c3f79cf7

Spark Streaming + Kafka Integration

Problem - Count the words from the messages stored in Kafka every 10 seconds

Steps

- Publish stream of “y” using yes command to Kafka topic
- Spark streaming code consumes the stream of “y” from the Kafka topic in the batch interval of 10 seconds
- Print number of “y” consumed or processed

Spark Streaming + Kafka Integration

Gist -

https://gist.github.com/singhabhinav/0ab4f33f5da16363ef9bba5b057c6465

Spark Streaming + Kafka Integration

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

Spark Streaming + Kafka Integration

Transformation	Meaning
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config propertyspark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

Spark Streaming + Kafka Integration

Transformation	Meaning
cogroup(*otherStream*, [*numTasks*])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

Spark Streaming - updateStateByKey - Why?

- In the last hands-on we just printed the word count in interval of 10 seconds

Spark Streaming - updateStateByKey - Why?

- In the last hands-on we just printed the word count in interval of 10 seconds
- What if we also want to count the each word seen in the input data stream in last 24 hours

Spark Streaming - updateStateByKey - Why?

- In the last hands-on we just printed the word count in interval of 10 seconds
- What if we also want to count the each word seen in the input data stream in last 24 hours
- How do we maintain the running count of each word in last 24 hours?

Spark Streaming - updateStateByKey - Why?

- To keep track of statistics, a state must be maintained while processing RDDs in the DStream

Spark Streaming - updateStateByKey - Why?

- To keep track of statistics, a state must be maintained while processing RDDs in the DStream
- If we maintain state for key-value pairs, the data may become too big to fit in memory on one machine

Spark Streaming - updateStateByKey - Why?

- To keep track of statistics, a state must be maintained while processing RDDs in the DStream
- If we maintain state for key-value pairs, the data may become too big to fit in memory on one machine
- We can use updateStateByKey function of Spark Streaming library

Spark Streaming - updateStateByKey - How?

- The updateStateByKey operation allows us to maintain arbitrary state while continuously updating it with new information

Spark Streaming - updateStateByKey - How?

- The updateStateByKey operation allows us to maintain arbitrary state while continuously updating it with new information
- To use this, we will have to do two steps

Spark Streaming - updateStateByKey - How?

- The updateStateByKey operation allows us to maintain arbitrary state while continuously updating it with new information
- To use this, we will have to do two steps
  - Define the state - The state can be an arbitrary data type

Spark Streaming - updateStateByKey - How?

- The updateStateByKey operation allows us to maintain arbitrary state while continuously updating it with new information
- To use this, we will have to do two steps
  - Define the state - The state can be an arbitrary data type
  - Define the state update function - Specify with a function how to update the state using the previous state and the new values from an input stream

Spark Streaming - updateStateByKey - How?

- In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not

Spark Streaming - updateStateByKey - How?

- In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not
- If the update function returns None then the key-value pair will be eliminated

Maintain a running count of each word seen in a input data stream

The running count is the state and it is an integer

Spark Streaming - updateStateByKey

```
# Python code
def updateFunction(newValues, runningCount):
    if runningCount is None:
       runningCount = 0
    # add the new values with the previous running
    # count to get the new count
    return sum(newValues, runningCount)

runningCounts = pairs.updateStateByKey(updateFunction)
```

Spark Streaming - updateStateByKey

```
# Python code
def updateFunction(newValues, runningCount):
    if runningCount is None:
       runningCount = 0
    # add the new values with the previous running
    # count to get the new count
    return sum(newValues, runningCount)

runningCounts = pairs.updateStateByKey(updateFunction)
```

Spark Streaming - updateStateByKey

Read more on updateStateByKey here

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

Transform Operation

- Allows arbitrary RDD-to-RDD functions to be applied on a DStream

Transform Operation

- Allows arbitrary RDD-to-RDD functions to be applied on a DStream
- Apply any RDD operation that is not available in the DStream API

Transform Operation

- Allows arbitrary RDD-to-RDD functions to be applied on a DStream
- Apply any RDD operation that is not available in the DStream API
- Perfect for reusing any RDD to RDD functions that you may have written in batch code and want to port over to streaming

Transform Operation

- Allows arbitrary RDD-to-RDD functions to be applied on a DStream
- Apply any RDD operation that is not available in the DStream API
- Perfect for reusing any RDD to RDD functions that you may have written in batch code and want to port over to streaming
- For example, the functionality of joining every batch in a data stream with another dataset is not directly exposed in the DStream API

Transform Operation - Use case

Real-time data cleaning by joining the input data stream with precomputed spam information and filtering based on it

```
// RDD containing spam information

val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...)

val cleanedDStream = wordCounts.transform { rdd =>

rdd.join(spamInfoRDD).filter(...)

// join data stream with spam information to do data cleaning
…
}
```

Transform Operation

Read more about transform operation here

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

Window Operations

- Apply transformations over a sliding window of data

Window Operations

- Apply transformations over a sliding window of data
- Use case in monitoring web server logs
    - Find out what happened in the last one house and refresh that statistics every one minute

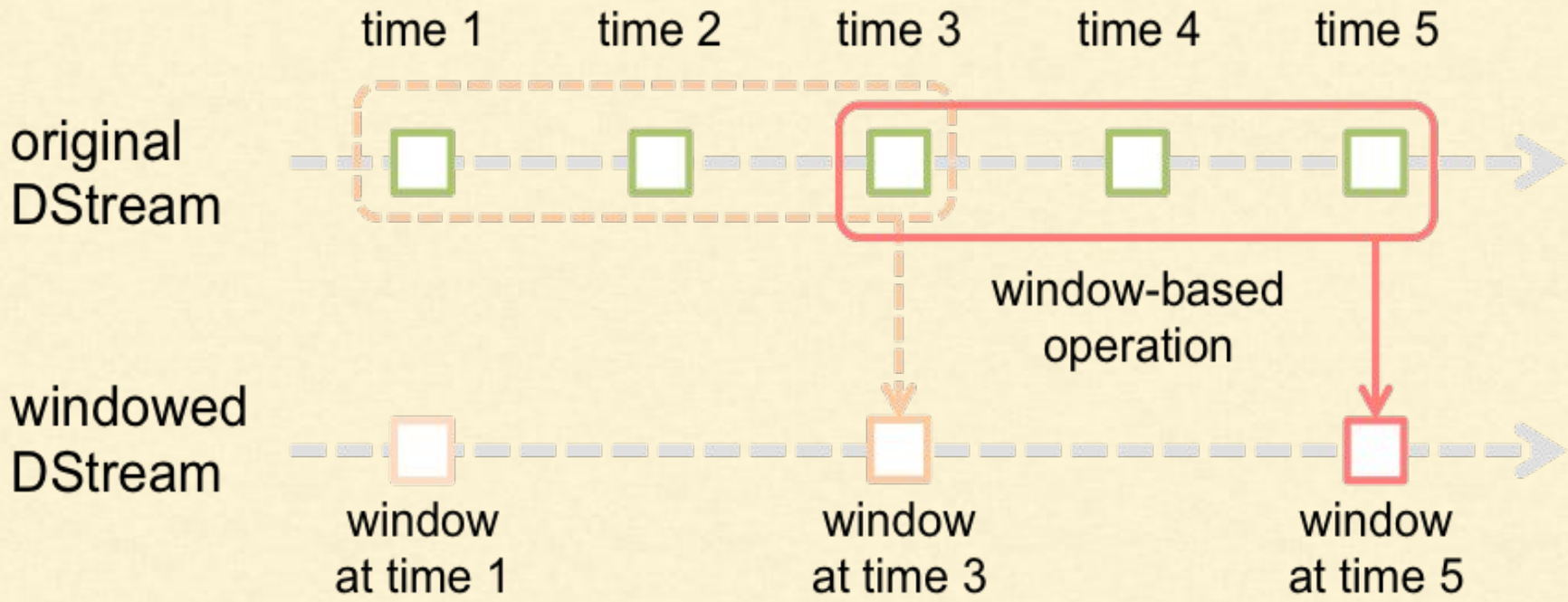
Window Operations

- Apply transformations over a sliding window of data
- Use case in monitoring web server logs
    - Find out what happened in the last one house and refresh that statistics every one minute
- **Window length - 1 hour**

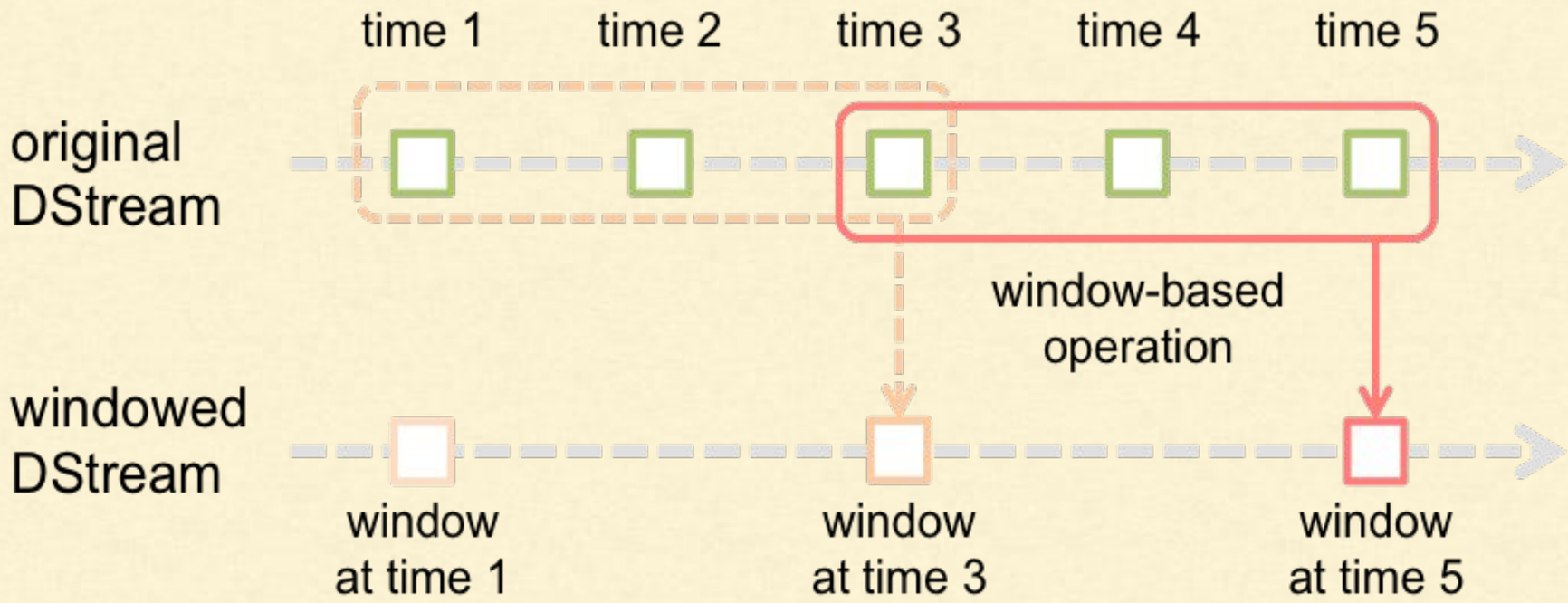
Window Operations

- Apply transformations over a sliding window of data
- Use case in monitoring web server logs
    - Find out what happened in the last one house and refresh that statistics every one minute
- **Window length - 1 hour**
- **Slide interval - 1 minute**

Window Operations



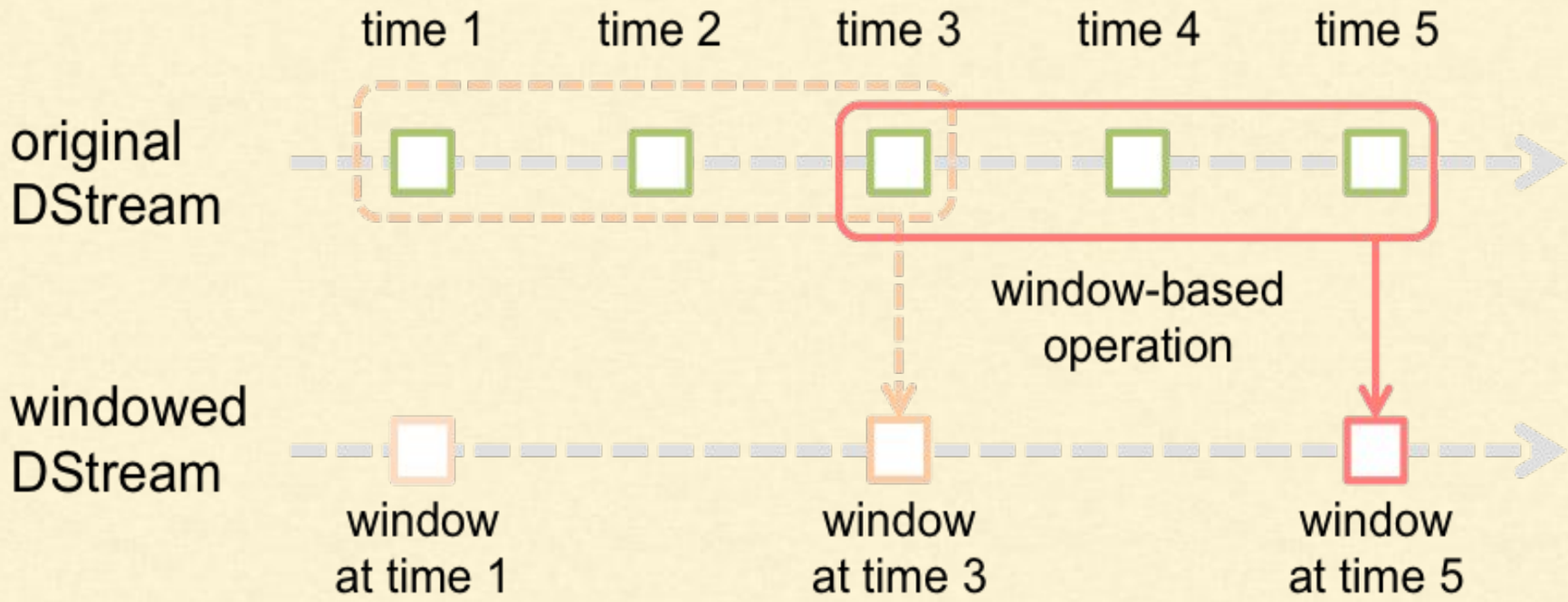
Window Operations



Needs to specify two parameters:

- **Window length -** The duration of the window (3 in the figure)

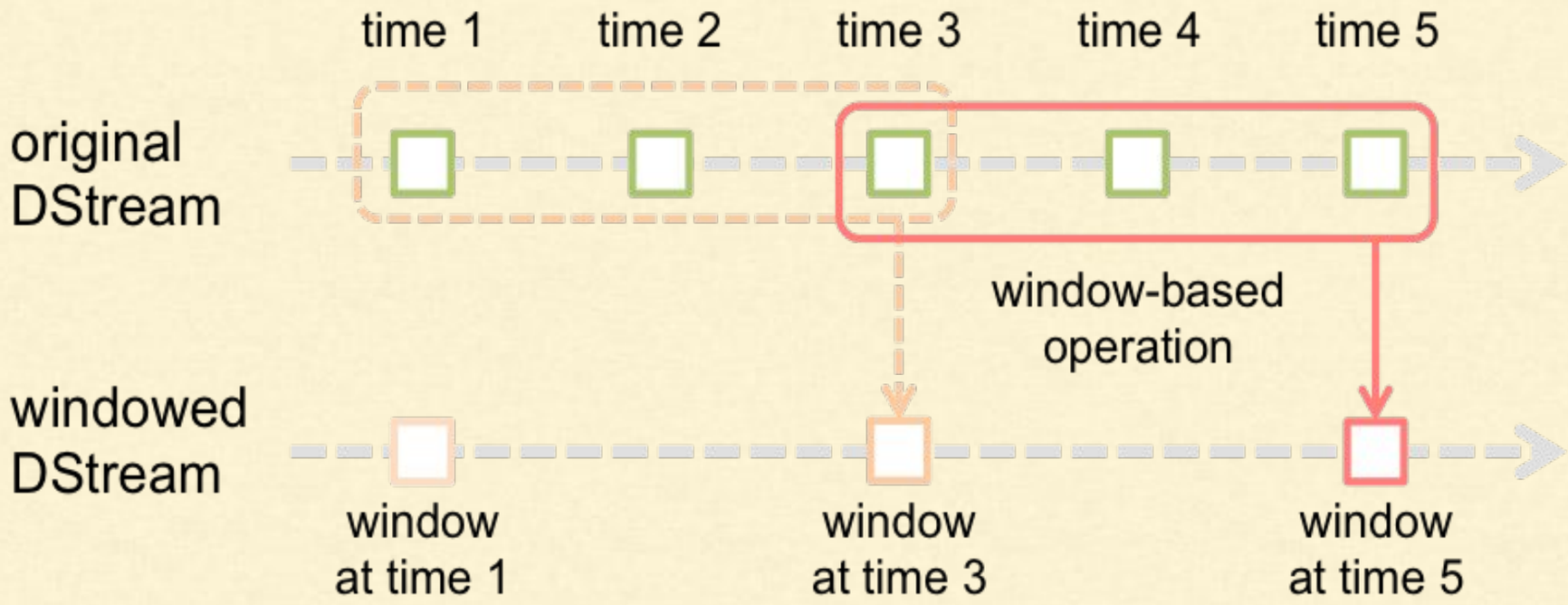
Window Operations



Needs to specify two parameters:

- **Window length -** The duration of the window (3 in the figure)
- **Sliding interval -** The interval at which the window operation is performed (2 in the figure)

Window Operations



Needs to specify two parameters:

- **Window length -** The duration of the window (3 in the figure)
- **Sliding interval -** The interval at which the window operation is performed (2 in the figure)

These two parameters must be multiples of the batch interval of the source DStream

Window Operations - Use case

Count the words in the 30 seconds of input data, every 10 seconds

```
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) =>
(a + b), Seconds(30), Seconds(10))
```

Window Operations - Use case

Read more about window operations here

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

Join Operations

Stream-stream joins

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

Join Operations

Stream-stream joins

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

- Here is each interval, the RDD generated by stream1 will be joined with the RDD generated by stream2

Join Operations

Stream-stream joins

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

- Here is each interval, the RDD generated by stream1 will be joined with the RDD generated by stream2
- We can also do leftOuterJoin, rightOuterJoin and fullOuterJoin

Join Operations

Windowed Stream-stream joins

```
val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)
```

Join Operations

Stream-dataset joins

```
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd =>
rdd.join(dataset) }
```

Join Operations

Read more about join operations here

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

Output Operations on DStreams

- Output operations allow DStream's data to be pushed out to external systems like
  - a database
  - or a file system

Output Operations on DStreams

- Output operations allow DStream's data to be pushed out to external systems like
  - a database
  - or a file system
- Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs)

Output Operations on DStreams

`print()`

- Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application.
- This is useful during development and debugging

Output Operations on DStreams

`print()`

- Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application.
- This is useful during development and debugging

`saveAsTextFiles(prefix, [suffix])`

- Saves DStream's contents as text files.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]"

Output Operations on DStreams

`saveAsObjectFiles(prefix, [suffix])`

- Saves DStream's contents as SequenceFiles of serialized Java objects.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

Output Operations on DStreams

`saveAsObjectFiles(prefix, [suffix])`

- Save this DStream's contents as SequenceFiles of serialized Java objects.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

`saveAsHadoopFiles(prefix, [suffix])`

- Save this DStream's contents as Hadoop files.
- The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

Output Operations on DStreams

`foreachRDD(func)`

- The most generic output operator that applies a function, func, to each RDD in the stream.

Output Operations on DStreams

`foreachRDD(func)`

- The most generic output operator that applies a function, func, to each RDD in the stream.
- This function should push the data in each RDD to an external system, such as
  - Saving the RDD to files
  - Or writing it over the network to a database

Output Operations on DStreams

`foreachRDD(func)`

- The most generic output operator that applies a function, func, to each RDD in the stream.
- This function should push the data in each RDD to an external system, such as
  - Saving the RDD to files
  - Or writing it over the network to a database
- Note that the function func is executed in the driver node running the streaming application

Output Operations on DStreams

`foreachRDD(func) - Design Pattern`

- Allows data to be sent out to external systems

Output Operations on DStreams

`foreachRDD(func) - Design Pattern`

- Allows data to be sent out to external systems
- Often writing data to external system requires creating a connection object (e.g. TCP connection to a remote server) and using it to send data to a remote system

Output Operations on DStreams

`foreachRDD(func) - Design Pattern`

- Allows data to be sent out to external systems
- Often writing data to external system requires creating a connection object (e.g. TCP connection to a remote server) and using it to send data to a remote system
- A developer may try creating a connection object at the Spark driver, and then try to use it in a Spark worker to push data in the RDDs to remote systems

Output Operations on DStreams

foreachRDD(func) - Design 1

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

Output Operations on DStreams

foreachRDD(func) - Design 2

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

Output Operations on DStreams

foreachRDD(func) - Design 3

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

Output Operations on DStreams

foreachRDD(func) - Design 4

```
   dstream.foreachRDD { rdd =>
      rdd.foreachPartition { partitionOfRecords =>
         // ConnectionPool is a static, lazily initialized pool of
connections
         val connection = ConnectionPool.getConnection()
         partitionOfRecords.foreach(record =>
            connection.send(record))
         ConnectionPool.returnConnection(connection)
         // return to the pool for future reuse
      }
   }
```

Output Operations on DStreams

Read more about output operations on DStreams here

https://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations-on-dstreams