Big Data Computing

Master's Degree in Computer Science

2019-2020

Gabriele Tolomei

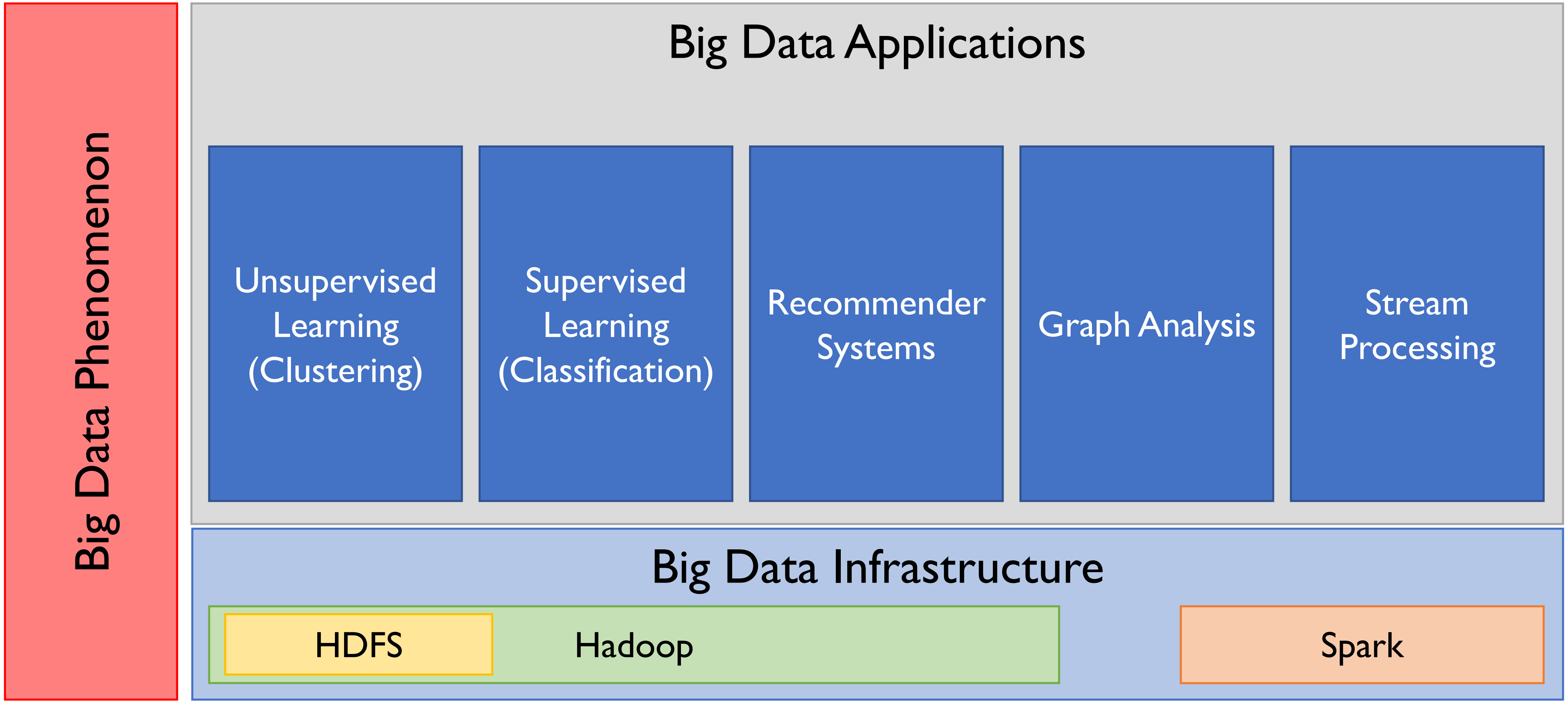
Department of Computer Science

Sapienza Università di Roma

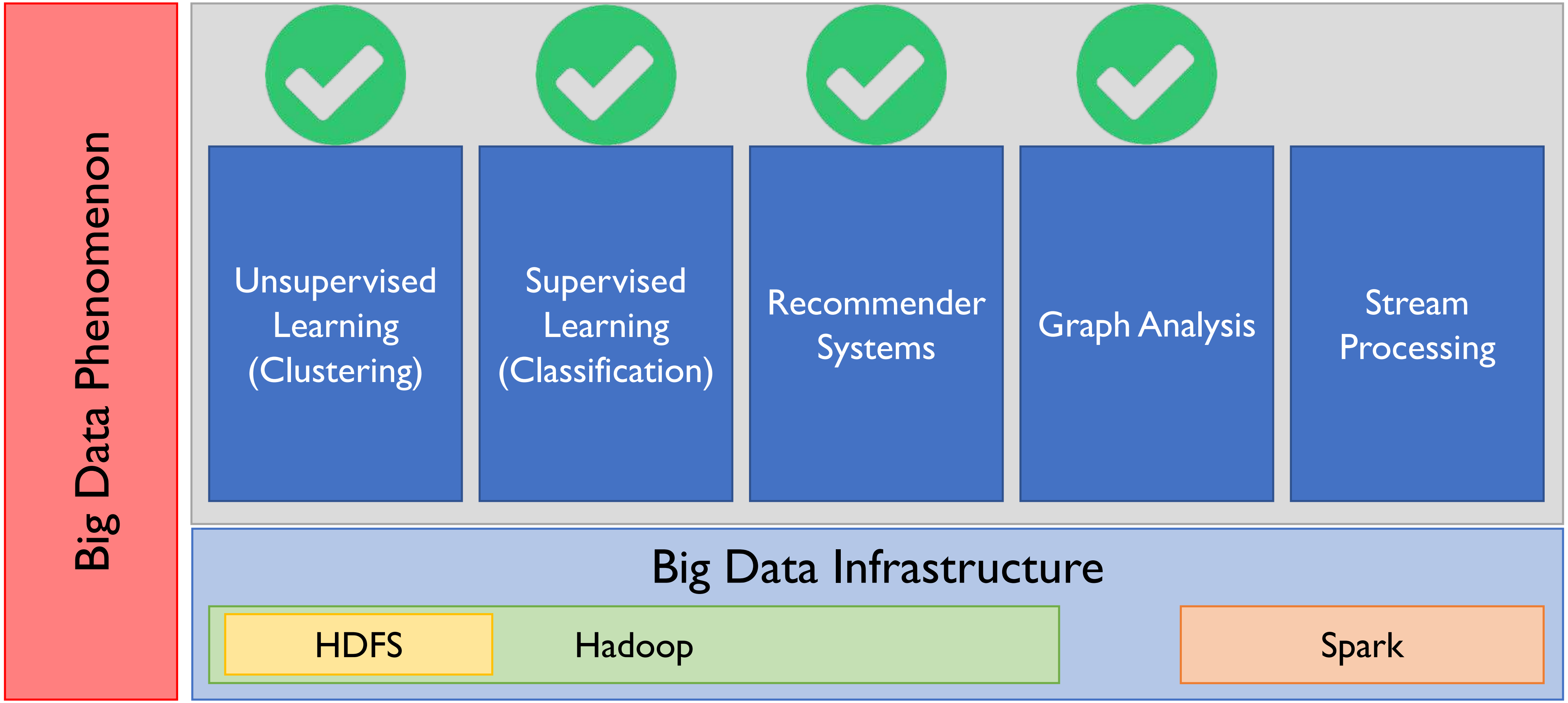
tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

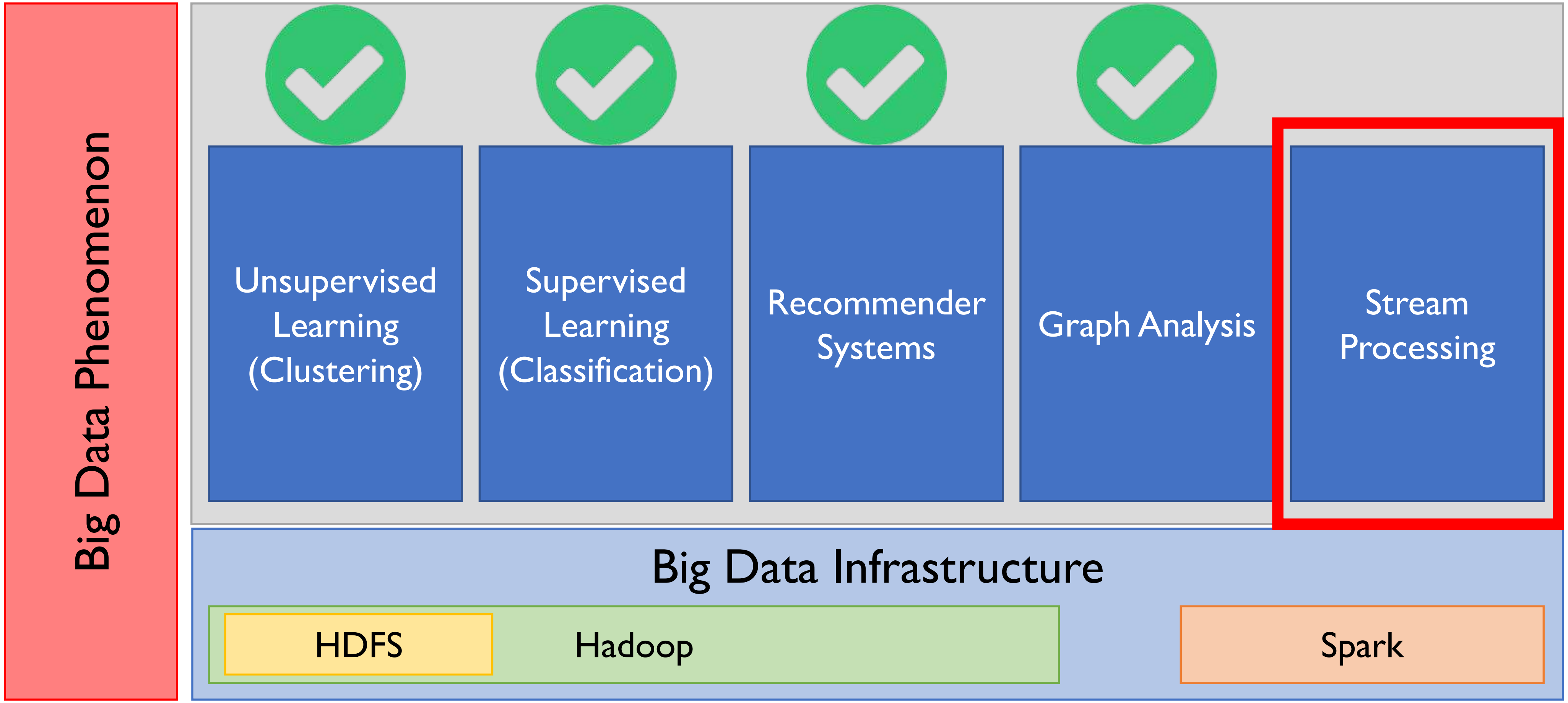
Outline of the Course



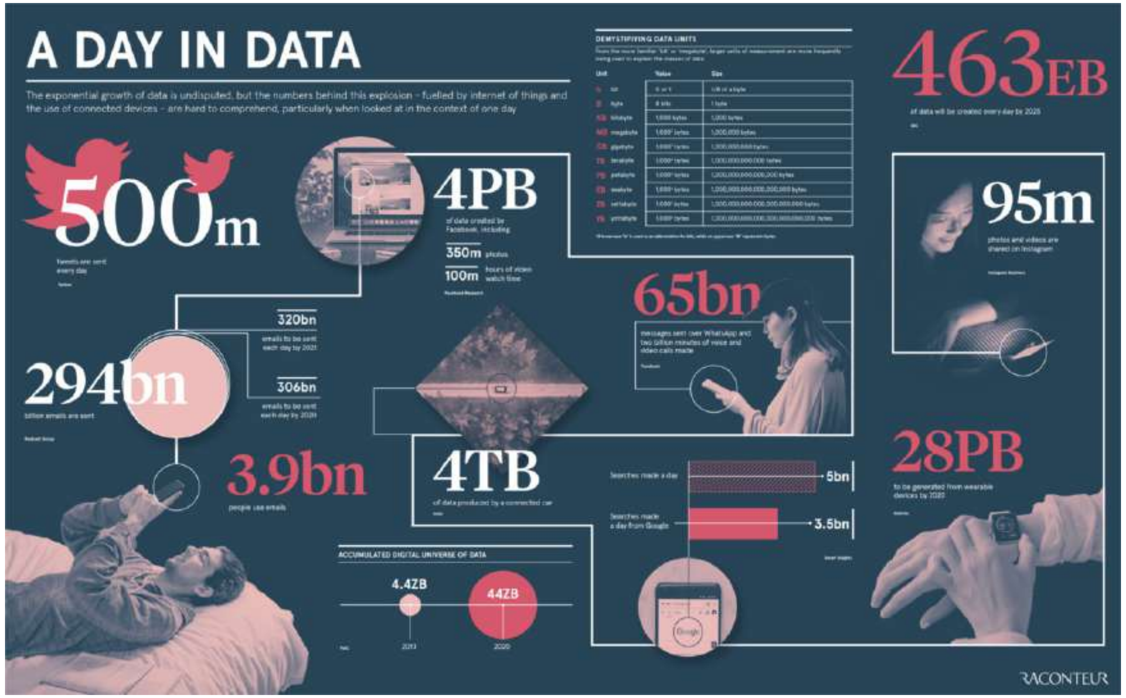
Outline of the Course



Outline of the Course



Do You Remember All This?



https://www.dailyinfographic.com/worlds-internet-activity-for-one-minute

https://www.visualcapitalist.com/how-much-data-is-generated-each-day/

Just Crunch Some Numbers…

- Every second:
  - Around **9,000 tweets** are sent on **Twitter**
  - About **1,000 pictures** are uploaded on **Instagram**
  - More than **82,000 queries** are submitted to **Google**
  - Approximately **84,000 videos** are watched on **YouTube**
  - Roughly **3,000,000 emails** are sent

Just Crunch Some Numbers…

- Every second:
  - Around **9,000 tweets** are sent on **Twitter**
  - About **1,000 pictures** are uploaded on **Instagram**
  - More than **82,000 queries** are submitted to **Google**
  - Approximately **84,000 videos** are watched on **YouTube**
  - Roughly **3,000,000 emails** are sent
- Curious about that? Just check Internet Live Stats website

Opportunities and Challenges

- Great time to be working in the data analysis/science landscape
    - Data is generated at an unprecedented pace and scale

Opportunities and Challenges

- Great time to be working in the data analysis/science landscape
  - Data is generated at an unprecedented pace and scale
- However, hard challenges are right behind the corner:
  - How do we collect data at this scale?
  - How do we ensure that our data analytics pipelines are continuously updated by feeding them with the freshest data?

Opportunities and Challenges

- Great time to be working in the data analysis/science landscape
  - Data is generated at an unprecedented pace and scale
- However, hard challenges are right behind the corner:
  - How do we collect data at this scale?
  - How do we ensure that our data analytics pipelines are continuously updated by feeding them with the freshest data?
- In other words, how do we handle with **streaming data**?

What is Streaming Data?

- Data that is generated **continuously** by many different sources

What is Streaming Data?

- Data that is generated **continuously** by many different sources
- Each source sends small "chunks" of records simultaneously (order of kB) at **very high speed**

What is Streaming Data?

- Data that is generated **continuously** by many different sources
- Each source sends small "chunks" of records simultaneously (order of kB) at **very high speed**
- Includes a wide variety of data such as:
  - log files generated by customers using your mobile or web applications
  - e-commerce purchases
  - information from social networks
  - financial trading floors

How Streaming Data is Processed?

- Sequentially and **incrementally** on a record-by-record basis or over **sliding time** windows using **stream processing** techniques

How Streaming Data is Processed?

- Sequentially and **incrementally** on a record-by-record basis or over **sliding time** windows using **stream processing** techniques
- Used for a wide variety of analytics including: **correlations**, **aggregations**, **filtering**, and **sampling**

How Streaming Data is Processed?

- Sequentially and **incrementally** on a record-by-record basis or over **sliding time** windows using **stream processing** techniques
- Used for a wide variety of analytics including: **correlations**, **aggregations**, **filtering**, and **sampling**
- Information derived from such analysis gives companies visibility into many aspects of their business and customer activity

How Streaming Data is Processed?

- Sequentially and **incrementally** on a record-by-record basis or over **sliding time** windows using **stream processing** techniques
- Used for a wide variety of analytics including: **correlations**, **aggregations**, **filtering**, and **sampling**
- Information derived from such analysis gives companies visibility into many aspects of their business and customer activity
- Enabling companies to **respond promptly** to emerging situations

Streaming Data Examples

- A company tracks changes in public **sentiment** on their products by analyzing social media streams to adapt its marketing strategy

Streaming Data Examples

- A company tracks changes in public **sentiment** on their products by analyzing social media streams to adapt its marketing strategy
- Sensors in **vehicles** send data to a streaming application to monitor performance and detect any potential defects in advance

Streaming Data Examples

- A company tracks changes in public **sentiment** on their products by analyzing social media streams to adapt its marketing strategy
- Sensors in **vehicles** send data to a streaming application to monitor performance and detect any potential defects in advance
- A financial institution keeps track of the **stock market** in real time, and automatically rebalances portfolios based on stock price trends

Streaming Data Examples

- A real-estate website collects geolocation data from consumers' mobile devices and makes real-time **property recommendations**

Streaming Data Examples

- A real-estate website collects geolocation data from consumers' mobile devices and makes real-time **property recommendations**
- An online gaming company collects streaming data about player-game interactions, and offers **dynamic experiences** to engage its players

Streaming Data Examples

- A real-estate website collects geolocation data from consumers' mobile devices and makes real-time **property recommendations**
- An online gaming company collects streaming data about player-game interactions, and offers **dynamic experiences** to engage its players
- A media publisher streams billions of clickstream records from its online properties to optimize **content placement** on its site

Batch vs. Stream Processing

Batch Processing

Computes results that are derived from **all the data** it encompasses, and enables deep analysis of big data sets using **MapReduce-like** paradigm

Complex analytics

Batch vs. Stream Processing

Stream Processing

Ingests sequences of data, and **incrementally updates** metrics and summary statistics in response to each new incoming data record

Real time monitoring

Batch vs. Stream Processing

Batch Processing

Computes results that are derived from **all the data** it encompasses, and enables deep analysis of big data sets using **MapReduce-like** paradigm

Complex analytics

Stream Processing

Ingests sequences of data, and **incrementally updates** metrics and summary statistics in response to each new incoming data record

Real time monitoring

Batch vs. Stream Processing

	Batch Processing	Stream Processing
Data Scope	Queries or processing over all or most of the data in the dataset	Queries or processing over data within a rolling time window, or on just the most recent data record

Batch vs. Stream Processing

	Batch Processing	Stream Processing
Data Scope	Queries or processing over all or most of the data in the dataset	Queries or processing over data within a rolling time window, or on just the most recent data record
Data Size	Large batches of data	Individual records or micro batches consisting of a few records

Batch vs. Stream Processing

	Batch Processing	Stream Processing
Data Scope	Queries or processing over all or most of the data in the dataset	Queries or processing over data within a rolling time window, or on just the most recent data record
Data Size	Large batches of data	Individual records or micro batches consisting of a few records
Performance	Latencies in minutes to hours	Latency should be in the order of seconds or milliseconds

Batch vs. Stream Processing

	Batch Processing	Stream Processing
Data Scope	Queries or processing over all or most of the data in the dataset	Queries or processing over data within a rolling time window, or on just the most recent data record
Data Size	Large batches of data	Individual records or micro batches consisting of a few records
Performance	Latencies in minutes to hours	Latency should be in the order of seconds or milliseconds
Analyses	Complex analytics	Simple response functions, aggregates, and rolling metrics

Batch vs. Stream Processing

	Batch Processing	Stream Processing
Data Scope	Queries or processing over all or most of the data in the dataset	Queries or processing over data within a rolling time window, or on just the most recent data record
Data Size	Large batches of data	Individual records or micro batches consisting of a few records
Performance	Latencies in minutes to hours	Latency should be in the order of seconds or milliseconds
Analyses	Complex analytics	Simple response functions, aggregates, and rolling metrics

Batch vs. Stream Processing

Many organizations are building a **hybrid** model by combining the two approaches having a **real-time** and a **batch** layer

Batch vs. Stream Processing

Many organizations are building a **hybrid** model by combining the two approaches having a **real-time** and a **batch** layer

Data is first processed by a streaming data platform to extract real-time insights

Batch vs. Stream Processing

Many organizations are building a **hybrid** model by combining the two approaches having a **real-time** and a **batch** layer

Data is first processed by a streaming data platform to extract real-time insights

Then data is persisted into **dedicated storage** systems

Batch vs. Stream Processing

Many organizations are building a **hybrid** model by combining the two approaches having a **real-time** and a **batch** layer

Data is first processed by a streaming data platform to extract real-time insights

Then data is persisted into **dedicated storage** systems

There, it can be transformed and loaded for a variety of batch processing tasks

Working with Streaming Data: Challenges

Streaming Data Processing requires **2 layers**

Working with Streaming Data: Challenges

Streaming Data Processing requires **2 layers**

Storage Layer

Processing Layer

Working with Streaming Data: Challenges

Streaming Data Processing requires **2 layers**

Storage Layer

Must support record ordering and strong consistency to enable fast, inexpensive, and replayable reads and writes of large streams of data

Processing Layer

Working with Streaming Data: Challenges

Streaming Data Processing requires **2 layers**

Storage Layer

Processing Layer

Responsible for consuming data from the storage layer, running computations on that data, and then notifying the storage layer to delete data that is no longer needed

Working with Streaming Data: Challenges

Streaming Data Processing requires **2 layers**

Storage Layer

Must support record ordering and strong consistency to enable fast, inexpensive, and replayable reads and writes of large streams of data

Processing Layer

Responsible for consuming data from the storage layer, running computations on that data, and then notifying the storage layer to delete data that is no longer needed

Scalability, Data Durability, and Fault Tolerance

Streaming Data Processing Platforms

- Many streaming data processing platforms have emerged:
  - Apache Spark Streaming
  - Apache Storm
  - Apache Kafka
  - Apache Flume
  - Amazon Kinesis Streams
  - Amazon Kinesis Firehose

Streaming Data Processing Platforms

- Many streaming data processing platforms have emerged:
  - Apache Spark Streaming
  - Apache Storm
  - Apache Kafka
  - Apache Flume
  - Amazon Kinesis Streams
  - Amazon Kinesis Firehose

Overview of Spark Streaming

Spark Streaming

An extension of the core Spark API that enables **scalable**, **high-throughput**, and **fault-tolerant** stream processing of live data streams

Overview of Spark Streaming

Data Feeding

Data can be ingested from many sources like **Kafka**, **Flume**, **Kinesis**, or **TCP sockets**

Overview of Spark Streaming

Data Processing

Support for complex algorithms using high-level functions like `map`, `reduce`, `join` and `window`

Overview of Spark Streaming

Data Processing

Support for complex algorithms using high-level functions like **`map`**, **`reduce`**, **`join`** and **`window`**

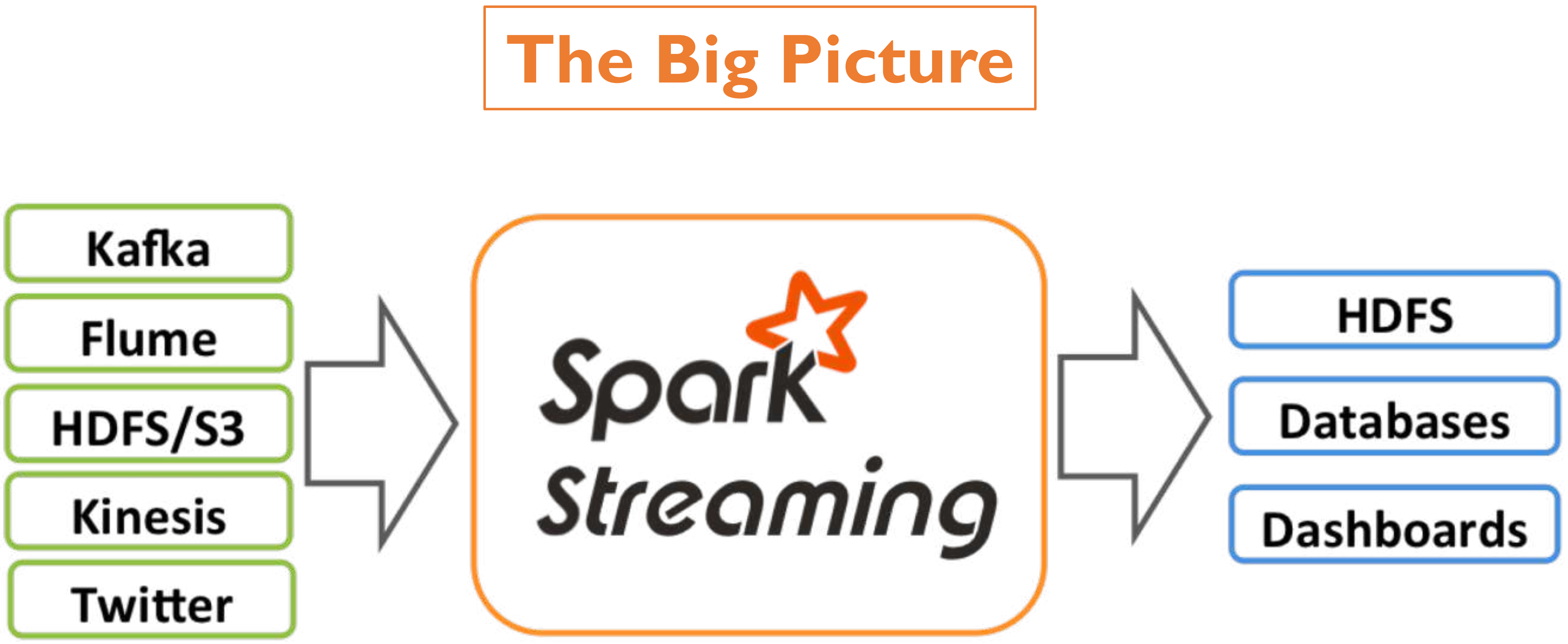
Any Spark's **machine learning** or **graph** algorithms can be applied to data streams

Overview of Spark Streaming

Data Persistence

Processed data can be pushed out to **filesystems**, **databases**, and **live dashboards**

Overview of Spark Streaming



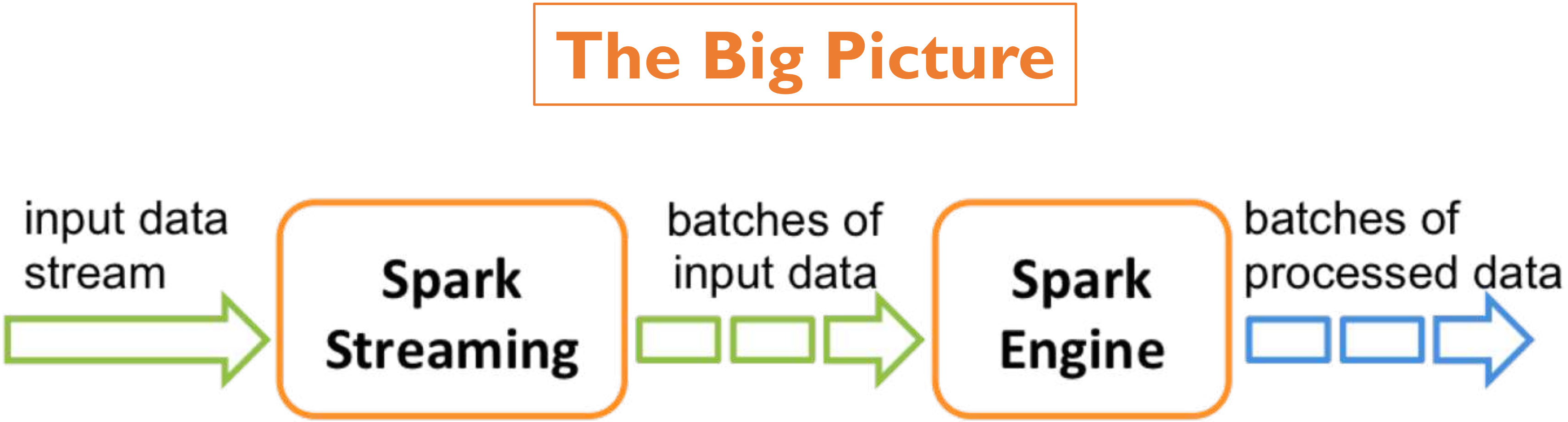
Internals of Spark Streaming

Spark Streaming receives live input data streams and divides them into **batches**

Internals of Spark Streaming

Those batches are then processed by the Spark engine to generate the final stream of batch results

Internals of Spark Streaming



Discretized Stream (DStream)

- The core high-level **abstraction** of Spark

Discretized Stream (DStream)

- The core high-level **abstraction** of Spark
- Discretized Stream (DStream) represents a continuous stream of data

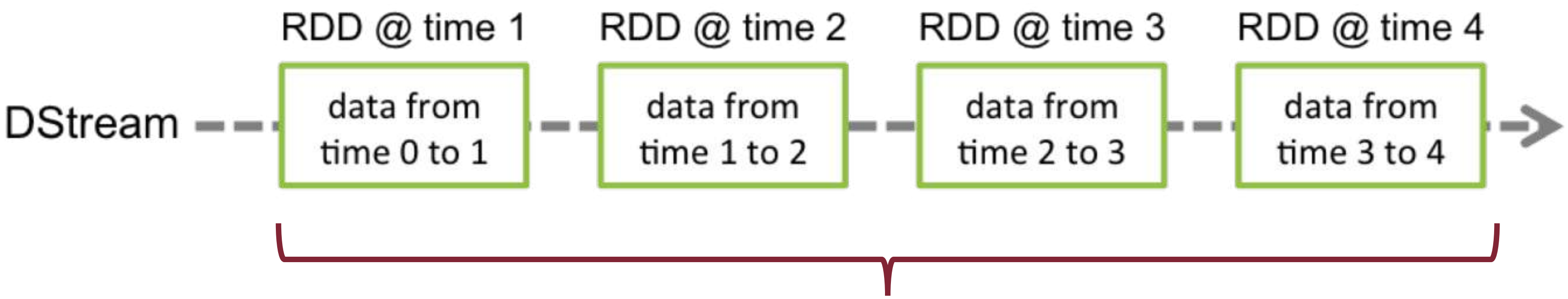
Discretized Stream (DStream)

- The core high-level **abstraction** of Spark
- Discretized Stream (DStream) represents a continuous stream of data
- DStreams can be created:
  - from **input data streams** from sources like Kafka, Flume, and Kinesis
  - as the result of the application of **transformations** on other DStreams

Discretized Stream (DStream)

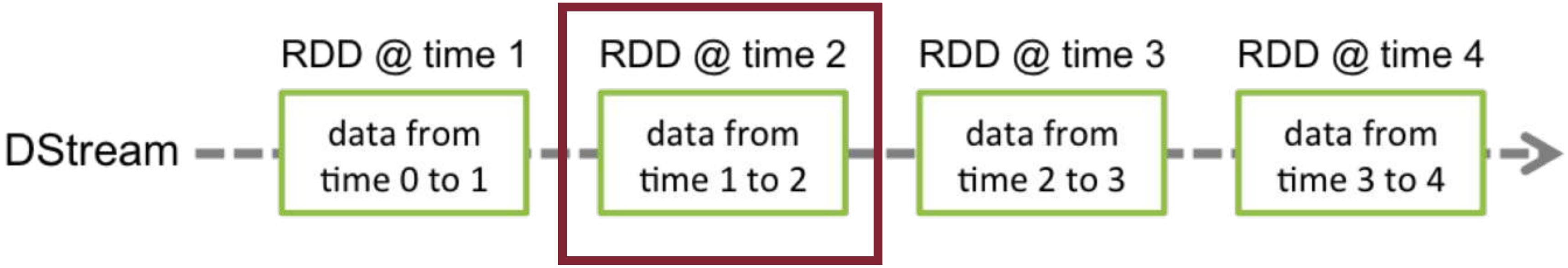
- The core high-level **abstraction** of Spark
- Discretized Stream (DStream) represents a continuous stream of data
- DStreams can be created:
  - from **input data streams** from sources like Kafka, Flume, and Kinesis
  - as the result of the application of **transformations** on other DStreams
- Internally, a DStream is represented as a **sequence of RDDs**

Discretized Stream (DStream)



Sequence of immutable distributed datasets (RDDs)

Discretized Stream (DStream)



Each RDD in a DStream contains data from a certain interval

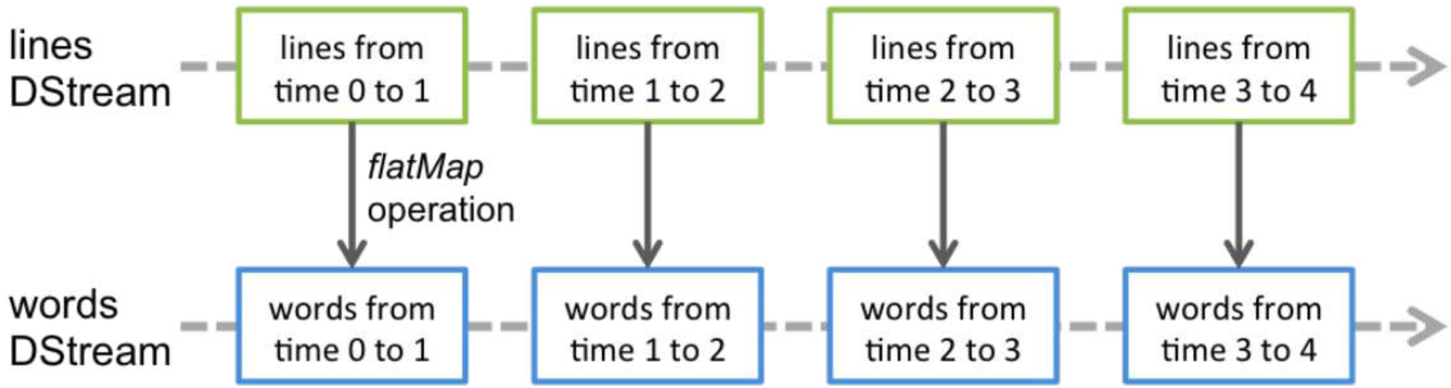
Discretized Stream (DStream)

Any operation applied on a DStream translates to operations on the underlying RDDs

Discretized Stream (DStream)

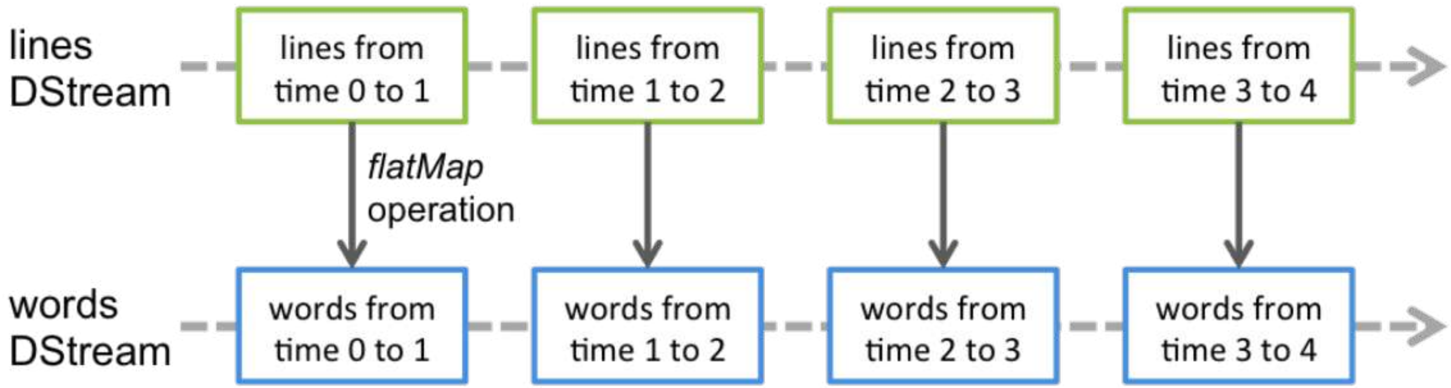
Any operation applied on a DStream translates to operations on the underlying RDDs

Example: Transforming streams of line strings into streams of words



Discretized Stream (DStream)

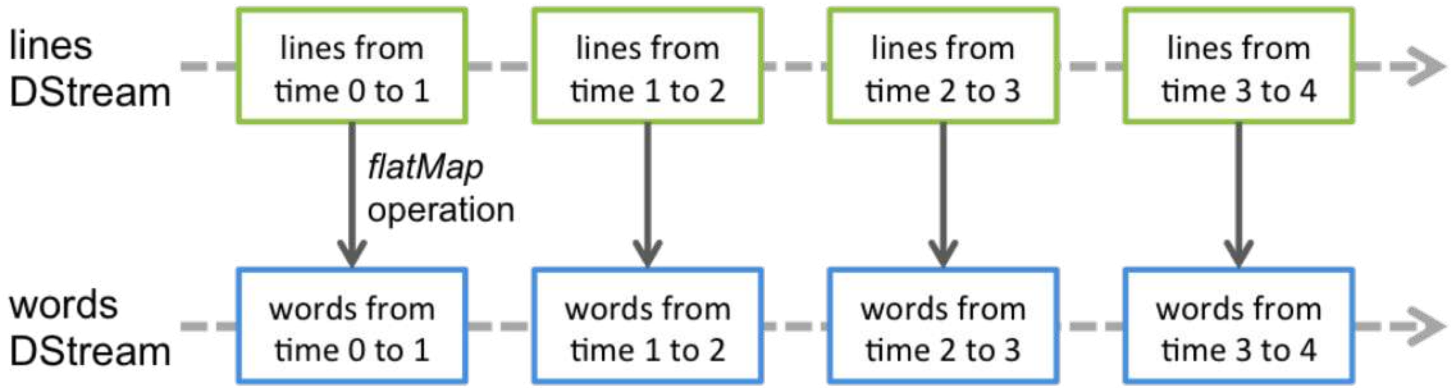
These underlying RDD transformations are computed by the Spark engine



Discretized Stream (DStream)

These underlying RDD transformations are computed by the Spark engine

DStream operations hide most of the details and provide the developer with a higher-level API



Discretize Your Streaming Application

- A key aspect of building a streaming application is to define the **batch interval**

Discretize Your Streaming Application

- A key aspect of building a streaming application is to define the **batch interval**
- This specifies the **unit of time** used to gather data streamed out from the source(s) of interest

Discretize Your Streaming Application

- A key aspect of building a streaming application is to define the **batch interval**
- This specifies the **unit of time** used to gather data streamed out from the source(s) of interest
- If the batch duration is 2 seconds, then the data will be collected every 2 seconds and stored in an RDD accordingly

Discretize Your Streaming Application

- A key aspect of building a streaming application is to define the **batch interval**
- This specifies the **unit of time** used to gather data streamed out from the source(s) of interest
- If the batch duration is 2 seconds, then the data will be collected every 2 seconds and stored in an RDD accordingly
- The resulting chain of continuous series of RDDs is a DStream which is **immutable** and can be used as a distributed dataset by Spark

Input DStreams and Receivers

- Input DStreams are DStreams representing the **stream of input** data received from streaming sources

Input DStreams and Receivers

- Input DStreams are DStreams representing the **stream of input** data received from streaming sources
- Every input DStream (except file stream) is associated with a **Receiver** object

Input DStreams and Receivers

- Input DStreams are DStreams representing the **stream of input** data received from streaming sources
- Every input DStream (except file stream) is associated with a **Receiver** object
- Each Receiver object collects data from a source and stores it in Spark's memory for processing

Input DStreams and Receivers

Spark Streaming provides **2** categories of built-in **streaming sources**

Input DStreams and Receivers

Spark Streaming provides **2** categories of built-in **streaming sources**

Basic

Sources directly available in the `StreamingContext` API

Examples:
file systems and socket connections

Input DStreams and Receivers

Spark Streaming provides **2** categories of built-in **streaming sources**

Basic

Sources directly available in the `StreamingContext` API

Examples:
file systems and socket connections

Advanced

Sources that are available through extra utility classes properly linked

Examples:
Kafka, Flume, Kinesis, etc.

Input DStreams and Receivers

- A streaming application can receive **multiple** streams of data in parallel

Input DStreams and Receivers

- A streaming application can receive **multiple** streams of data in parallel
- In such a case, multiple input DStreams and Receivers are created (i.e., one for each source) to simultaneously receive data from all the streams

Input DStreams and Receivers

- A streaming application can receive **multiple** streams of data in parallel
- In such a case, multiple input DStreams and Receivers are created (i.e., one for each source) to simultaneously receive data from all the streams
- Allocate enough cores (or threads, if running locally) to a Spark Streaming application to process data, as well as to run the receiver(s)

For any further information:

https://spark.apache.org/docs/latest/streaming-programming-guide.html#input-dstreams-and-receivers

Reliability of Data Sources

There can be **2** kinds of data sources based on their **reliability**

Reliability of Data Sources

There can be **2** kinds of data sources based on their **reliability**

Reliable

These sources allow transferred data from them to be acknowledged

Guarantee no data will be lost during the transfer

Example:

Kafka and Flume

Reliability of Data Sources

There can be **2** kinds of data sources based on their **reliability**

Reliable

These sources allow transferred data from them to be acknowledged

Guarantee no data will be lost during the transfer

Example:
Kafka and Flume

Unreliable

No support for acknowledgment by the receiver

No guarantees on the integrity of the data transferred

Example:
Socket

Reliability of Receivers

Consequently, there can be **2** kinds of receivers

Reliability of Receivers

Consequently, there can be **2** kinds of receivers

Reliable

A reliable receiver correctly sends acknowledgment to a **reliable** source when the data has been received and stored in Spark with replication

Reliability of Receivers

Consequently, there can be **2** kinds of receivers

Reliable

A reliable receiver correctly sends acknowledgment to a **reliable** source when the data has been received and stored in Spark with replication

Unreliable

An unreliable receiver does not send acknowledgment to a source

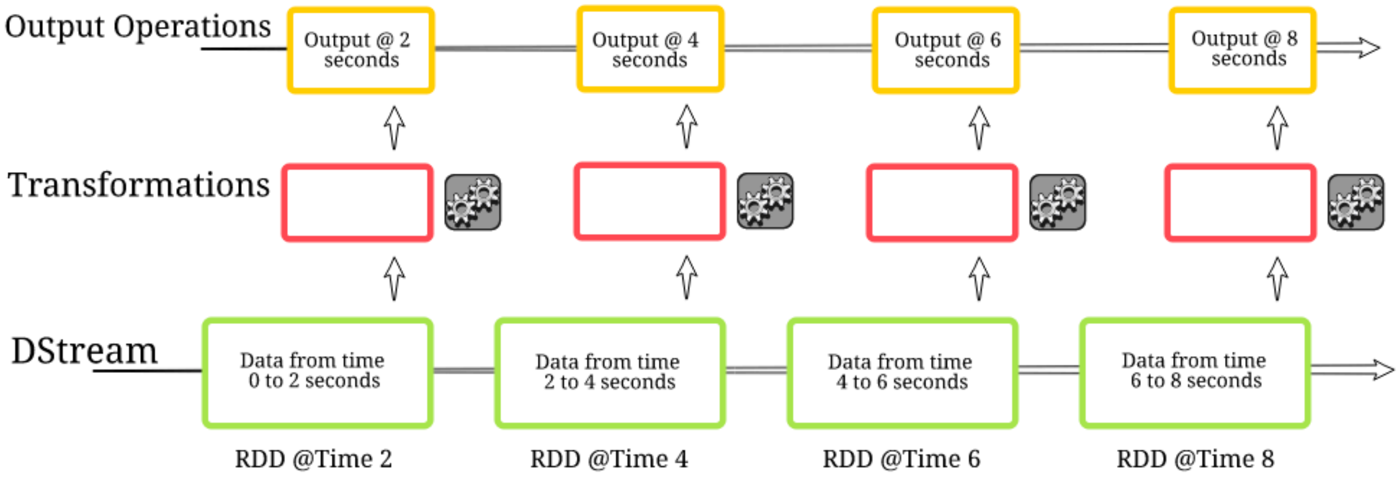
This can be used both in combination with **unreliable** and **reliable** sources

Transformations on DStreams

- Similar to that of RDDs, transformations allow the data from the input DStream to be modified

Transformations on DStreams

- Similar to that of RDDs, transformations allow the data from the input DStream to be modified
- DStreams support many of the transformations available on normal Spark RDDs



Transformations on DStreams

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.
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 and commutative 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.

Transformations on DStreams

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.
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 and commutative 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.

Transformations on DStreams

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 property `spark.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.
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.

`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
- It can be used to apply any **custom** RDD operation that is not exposed in the DStream API

`transform` Operation

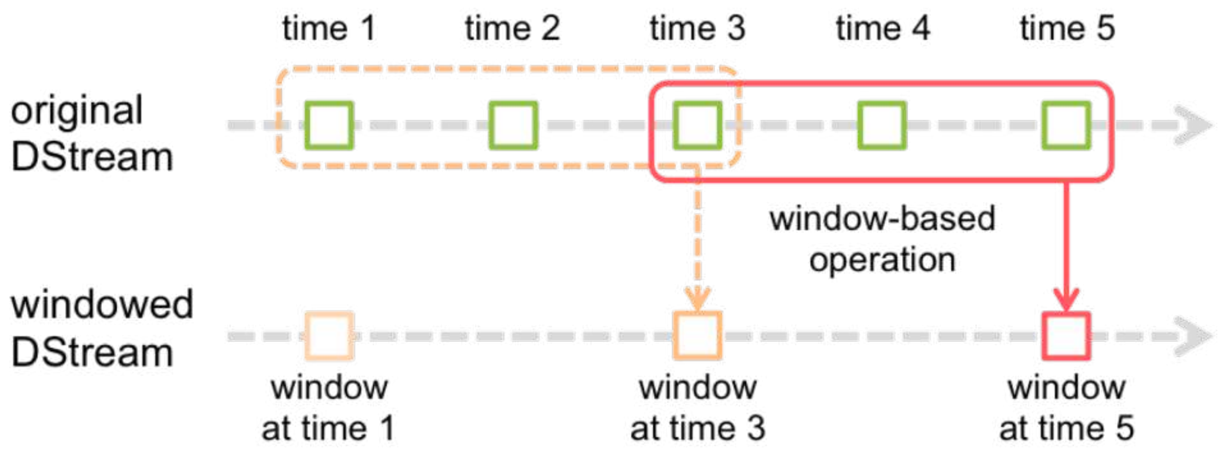
- Allows arbitrary RDD-to-RDD functions to be applied on a DStream
- It can be used to apply any **custom** RDD operation that is not exposed in the DStream API
- For example, the functionality of joining every batch in a data stream with another dataset is not directly exposed in the DStream API
  - Real-time data cleaning by joining the input data stream with precomputed spam information, and then filtering based on it

Window Operations on DStreams

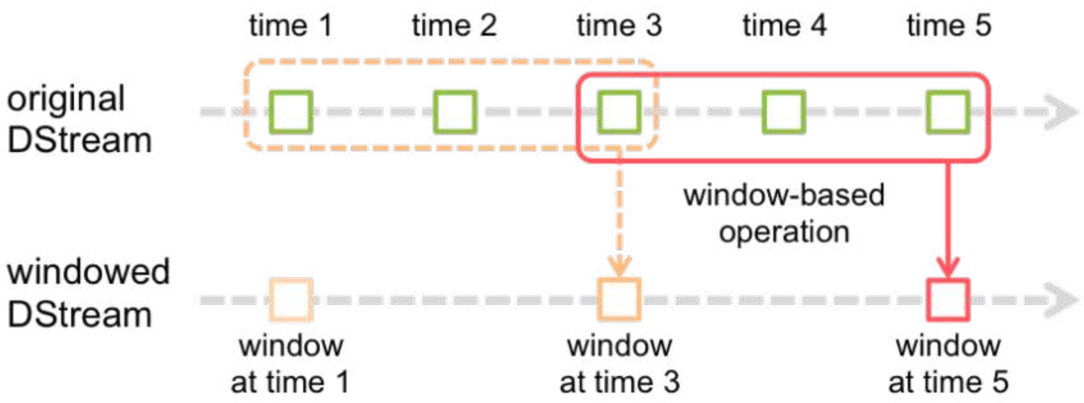
- Spark Streaming also provides **windowed computations**

Window Operations on DStreams

- Spark Streaming also provides **windowed computations**
- Allow you to apply transformations over a **sliding window** of data

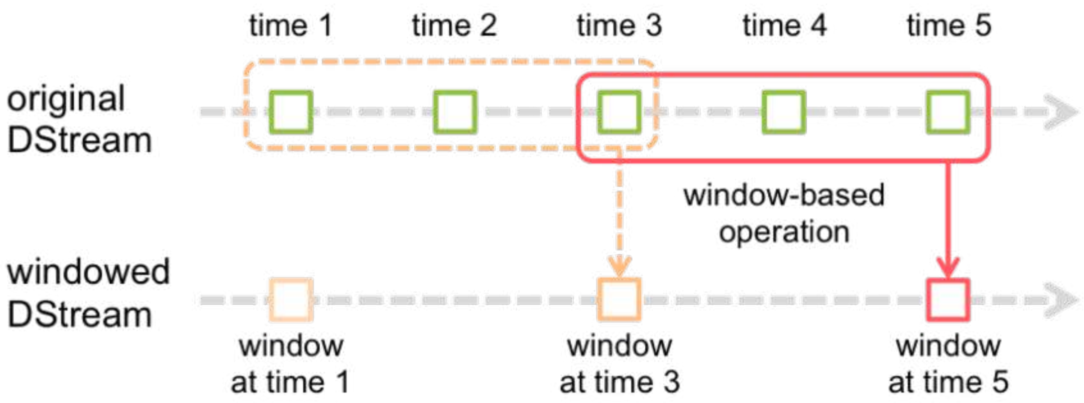


Window Operations on DStreams



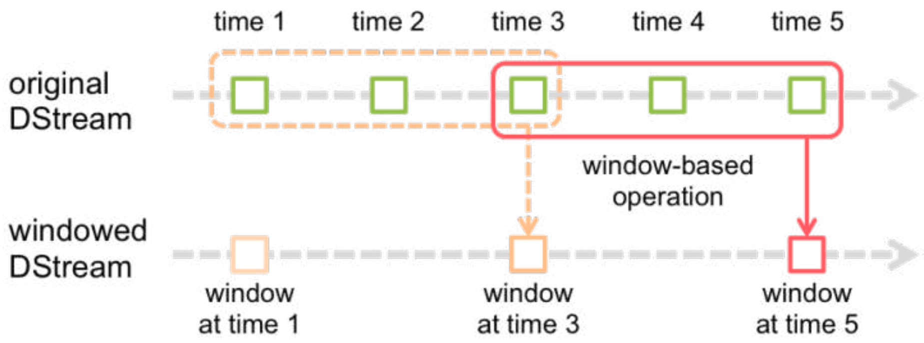
Every time the window slides over a source DStream, the source RDDs that fall within the window are **combined** and **processed** to produce the RDDs of the windowed DStream

Window Operations on DStreams



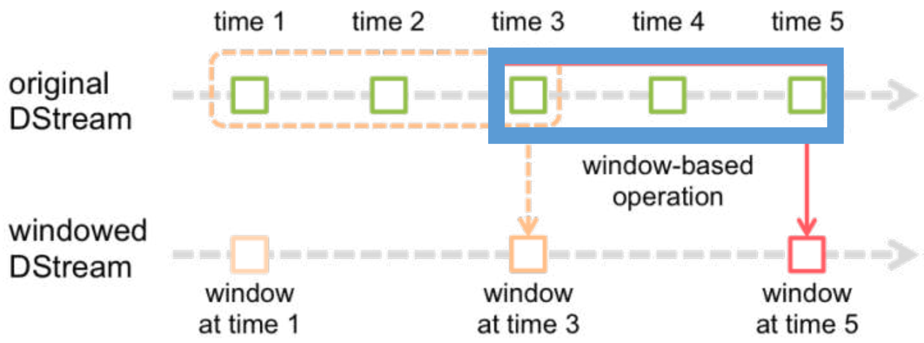
In the example above, the operation is applied over the **last 3 time units** of data, and **slides by 2 time units**

Window Operations on DStreams



Any window operation needs to specify **2 parameters**

Window Operations on DStreams



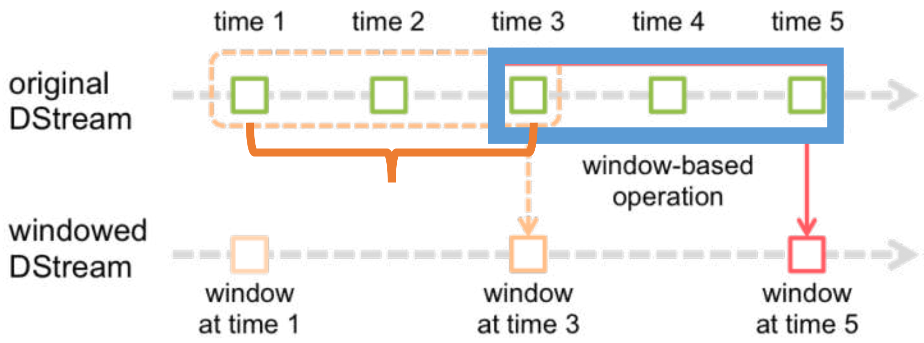
window length = 3

Any window operation needs to specify **2 parameters**

window length

The duration of the window

Window Operations on DStreams



window length = 3

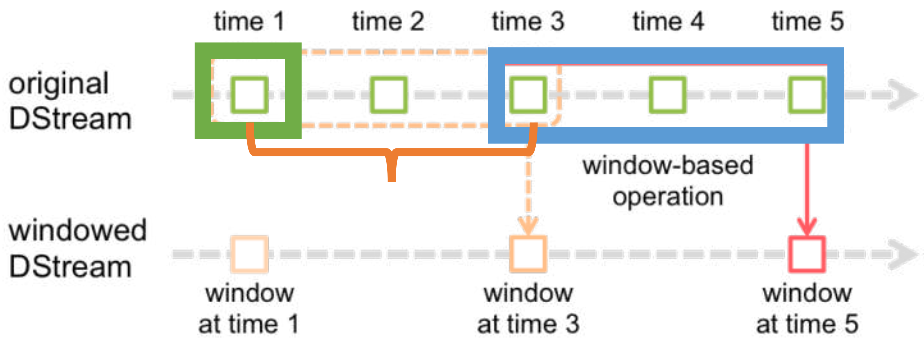
sliding interval = 2

Any window operation needs to specify **2 parameters**

sliding interval

The interval at which the window operation is performed

Window Operations on DStreams



window length = 3

sliding interval = 2

batch interval = 1

Any window operation needs to specify **2 parameters**

window length

The duration of the window

sliding interval

The interval at which the window operation is performed

They must be both multiples of the **batch interval** of the source DStream

Window Operations on DStreams

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel.
reduceByKeyAndWindow(*func*, *windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. **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 property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKeyAndWindow(*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*])	A more efficient version of the above `reduceByKeyAndWindow()` where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable only to "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*). Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument. Note that checkpointing must be enabled for using this operation.
countByValueAndWindow(*windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.

Output Operations on DStreams

Allow DStream's data to be pushed out to external systems (e.g., database or file systems)

Trigger the actual execution of all the DStream transformations (similar to actions for RDDs)

Output Operation	Meaning
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 for development and debugging. Python API This is called **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this 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]*".
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]*". Python API This is not available in the Python API.
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]*". Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from 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 process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

MLlib Operations on DStreams

Integrate streaming data processing capabilities with machine learning algorithms provided by MLlib

MLlib Operations on DStreams

Integrate streaming data processing capabilities with machine learning algorithms provided by MLlib

Streaming ML algorithms (e.g., Streaming Linear Regression, Streaming KMeans, etc.) can simultaneously learn from the streaming data as well as apply the model on the streaming data

MLlib Operations on DStreams

Integrate streaming data processing capabilities with machine learning algorithms provided by MLlib

Streaming ML algorithms (e.g., Streaming Linear Regression, Streaming KMeans, etc.) can simultaneously learn from the streaming data as well as apply the model on the streaming data

For a much larger class of ML algorithms, we can **train** a learning model **offline** and then **apply** it **online** on streaming data

Caching/Persistence

- Similar to RDDs, DStreams allow developers to persist the stream's data in memory using the `persist()` method

Caching/Persistence

- Similar to RDDs, DStreams allow developers to persist the stream's data in memory using the **`persist()`** method
- Useful for multiple operations on the same data in the DStream

Caching/Persistence

- Similar to RDDs, DStreams allow developers to persist the stream's data in memory using the **`persist()`** method
- Useful for multiple operations on the same data in the DStream
- For some window-based operations DStreams are automatically persisted in memory

Caching/Persistence

- Similar to RDDs, DStreams allow developers to persist the stream's data in memory using the `persist()` method
- Useful for multiple operations on the same data in the DStream
- For some window-based operations DStreams are automatically persisted in memory
- Unlike RDDs, the default persistence level of DStreams keeps the data **serialized** in memory (instead of deserialized persistence)

Checkpointing

- A streaming application must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)

Checkpointing

- A streaming application must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)
- Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures

Checkpointing

- A streaming application must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)
- Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures
- There are **2 types** of data that are checkpointed
  - **Metadata checkpointing**
  - **Data checkpointing**

Metadata Checkpointing

- Saving of the information defining the streaming computation to **fault-tolerant storage** like HDFS

Metadata Checkpointing

- Saving of the information defining the streaming computation to **fault-tolerant storage** like HDFS
- Used to recover from failure of the node running the driver of the streaming application

Metadata Checkpointing

- Saving of the information defining the streaming computation to **fault-tolerant storage** like HDFS
- Used to recover from failure of the node running the driver of the streaming application
- **Metadata** includes:
  - Configuration
  - DStream operations
  - Incomplete batches

Data Checkpointing

- Periodic saving of the generated RDDs to reliable storage (e.g., HDFS)

Data Checkpointing

- Periodic saving of the generated RDDs to reliable storage (e.g., HDFS)
- This is necessary in some **stateful** transformations that combine data across multiple batches

Data Checkpointing

- Periodic saving of the generated RDDs to reliable storage (e.g., HDFS)
- This is necessary in some **stateful** transformations that combine data across multiple batches
- Dependency chain between RDDs may indefinitely increase over time

Data Checkpointing

- Periodic saving of the generated RDDs to reliable storage (e.g., HDFS)
- This is necessary in some **stateful** transformations that combine data across multiple batches
- Dependency chain between RDDs may indefinitely increase over time
- Checkpointing avoid unbounded increases in recovery time (proportional to dependency chain)

Shared Variables in Streaming Data

- Sometimes we need to define functions like `map`, `reduce` or `filter` that has to be executed on **multiple clusters**

Shared Variables in Streaming Data

- Sometimes we need to define functions like `map`, `reduce` or `filter` that has to be executed on **multiple clusters**
- The variables used in these functions are copied to each of the machines our application is running (clusters)

Shared Variables in Streaming Data

- Sometimes we need to define functions like `map`, `reduce` or `filter` that has to be executed on **multiple clusters**
- The variables used in these functions are copied to each of the machines our application is running (clusters)
- Here, each cluster has a different executor and we want something that can give us a relation between these variables

Shared Variables in Streaming Data

- Example:
  - A Spark application running on 100 different clusters capturing Instagram images posted by people from different countries
  - We need a count of a particular tag that was mentioned in a post

Shared Variables in Streaming Data

- Example:
  - A Spark application running on 100 different clusters capturing Instagram images posted by people from different countries
  - We need a count of a particular tag that was mentioned in a post
- Each cluster's executor will calculate the results of the data present on that particular cluster

Shared Variables in Streaming Data

- Example:
  - A Spark application running on 100 different clusters capturing Instagram images posted by people from different countries
  - We need a count of a particular tag that was mentioned in a post
- Each cluster's executor will calculate the results of the data present on that particular cluster
- Spark provides **shared variables** to allow aggregating results from different clusters: **accumulator** and **broadcast variables**

Accumulator Variables

Accumulators can be used to keep track of the number of times something happens (e.g., an error or an incoming request)

Accumulator Variables

Accumulators can be used to keep track of the number of times something happens (e.g., an error or an incoming request)

The **executor** on each cluster sends data back to the **driver process** to update the values of the accumulator variables

Accumulator Variables

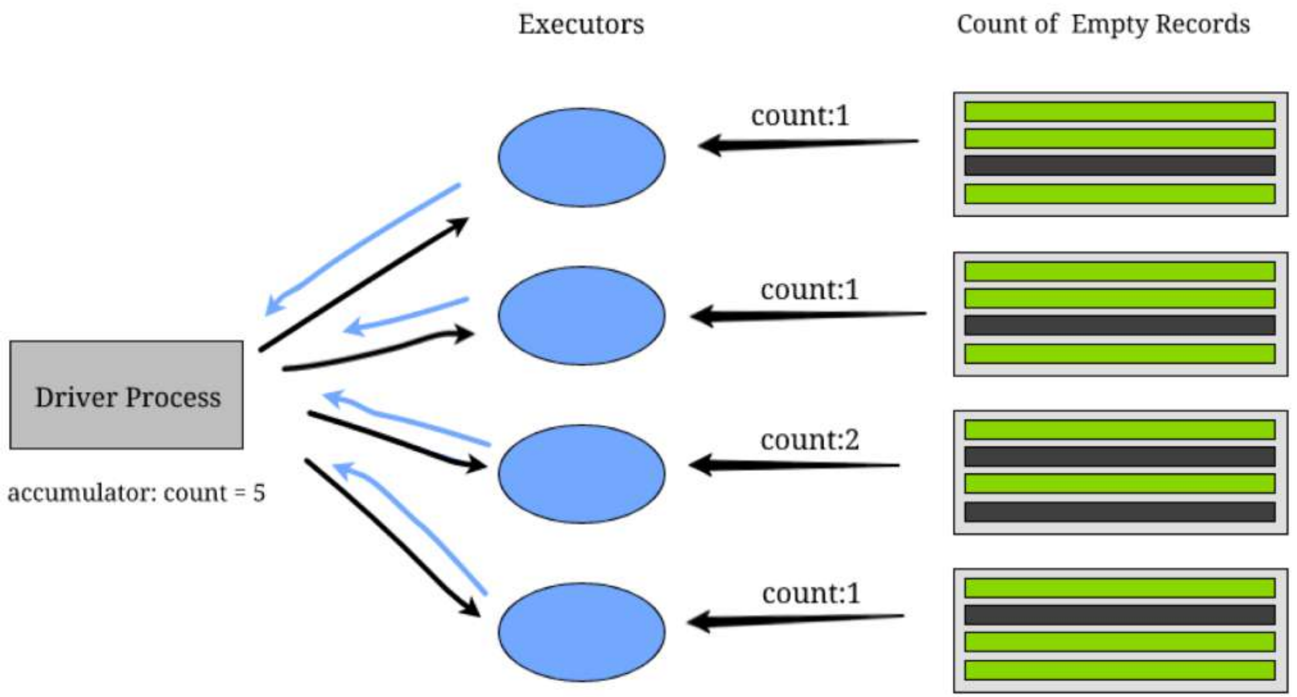
Accumulators can be used to keep track of the number of times something happens (e.g., an error or an incoming request)

The **executor** on each cluster sends data back to the **driver process** to update the values of the accumulator variables

Applicable only to **associative** and **commutative** operations (e.g., sum and maximum but not the mean)

Accumulator Variables

Each cluster sends to the driver process the count of empty records of the data it operates on



Broadcast Variables

Broadcast variables allow the programmer to keep a **read-only** variable cached on each machine rather than shipping a copy of it with tasks

Broadcast Variables

Broadcast variables allow the programmer to keep a **read-only** variable cached on each machine rather than shipping a copy of it with tasks

They can be used, for example, to give every node a copy of a large input dataset in an efficient manner

Broadcast Variables

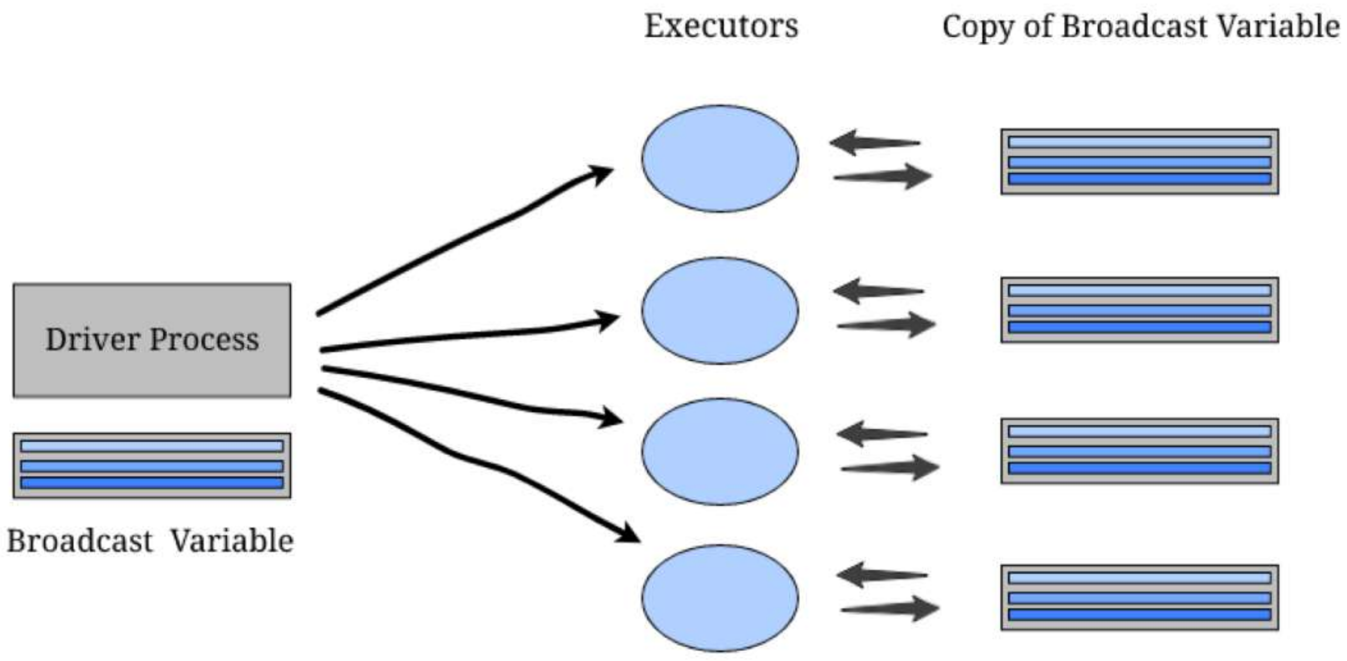
Broadcast variables allow the programmer to keep a **read-only** variable cached on each machine rather than shipping a copy of it with tasks

They can be used, for example, to give every node a copy of a large input dataset in an efficient manner

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Broadcast Variables

Each executor works on its own read-only copy of the broadcast variable



Setting the Right Batch Interval

- A Spark Streaming application should be able to process data as fast as it is being received

Setting the Right Batch Interval

- A Spark Streaming application should be able to process data as fast as it is being received
- In other words, batches of data should be processed as fast as they are being generated

Setting the Right Batch Interval

- A Spark Streaming application should be able to process data as fast as it is being received
- In other words, batches of data should be processed as fast as they are being generated
- The batch processing time should be **less than** the batch interval (double-check this in the streaming web UI monitoring tool)

Setting the Right Batch Interval

- The **batch interval** has an impact on the data rates that can be sustained by the application on a fixed set of cluster resources

Setting the Right Batch Interval

- The **batch interval** has an impact on the data rates that can be sustained by the application on a fixed set of cluster resources
- For a particular data rate, the system may be able to process data every 2 seconds (batch interval), but not every 500 milliseconds

Setting the Right Batch Interval

- The **batch interval** has an impact on the data rates that can be sustained by the application on a fixed set of cluster resources
- For a particular data rate, the system may be able to process data every 2 seconds (batch interval), but not every 500 milliseconds
- So the batch interval needs to be set such that the expected data rate in production can be sustained

Setting the Right Batch Interval

- To find the correct tradeoff, start with a **conservative** batch interval (say, 5-10 seconds) and a **low data rate**

Setting the Right Batch Interval

- To find the correct tradeoff, start with a **conservative** batch interval (say, 5-10 seconds) and a **low data rate**
- Test if the system is able to keep up with the data rate, by checking the end-to-end delay experienced by each processed batch
  - If the delay is comparable to the batch size over time, then system is stable
  - If the delay is continuously increasing, it means that the system is unable to keep up and therefore unstable

Take-Home Message of Today

- Data that is generated **continuously** by many different sources

Take-Home Message of Today

- Data that is generated **continuously** by many different sources
- Streaming data requires specific **data processing** techniques

Take-Home Message of Today

- Data that is generated **continuously** by many different sources
- Streaming data requires specific **data processing** techniques
- Spark provides a complete set of streaming data API (Spark Streaming)

Take-Home Message of Today

- Data that is generated **continuously** by many different sources
- Streaming data requires specific **data processing** techniques
- Spark provides a complete set of streaming data API (Spark Streaming)
- Discretized Streams (**DStreams**) is the core abstraction of data streaming

Take-Home Message of Today

- Data that is generated **continuously** by many different sources
- Streaming data requires specific **data processing** techniques
- Spark provides a complete set of streaming data API (Spark Streaming)
- Discretized Streams (**DStreams**) is the core abstraction of data streaming
- Spark Streaming may work in combination with other Spark libraries for machine learning and graph processing