Streaming

Batch, Microbatch and Stream Processing

- Batch and Microbatch – when new data arrives it is aggregated into batches and processed at some time in the future.

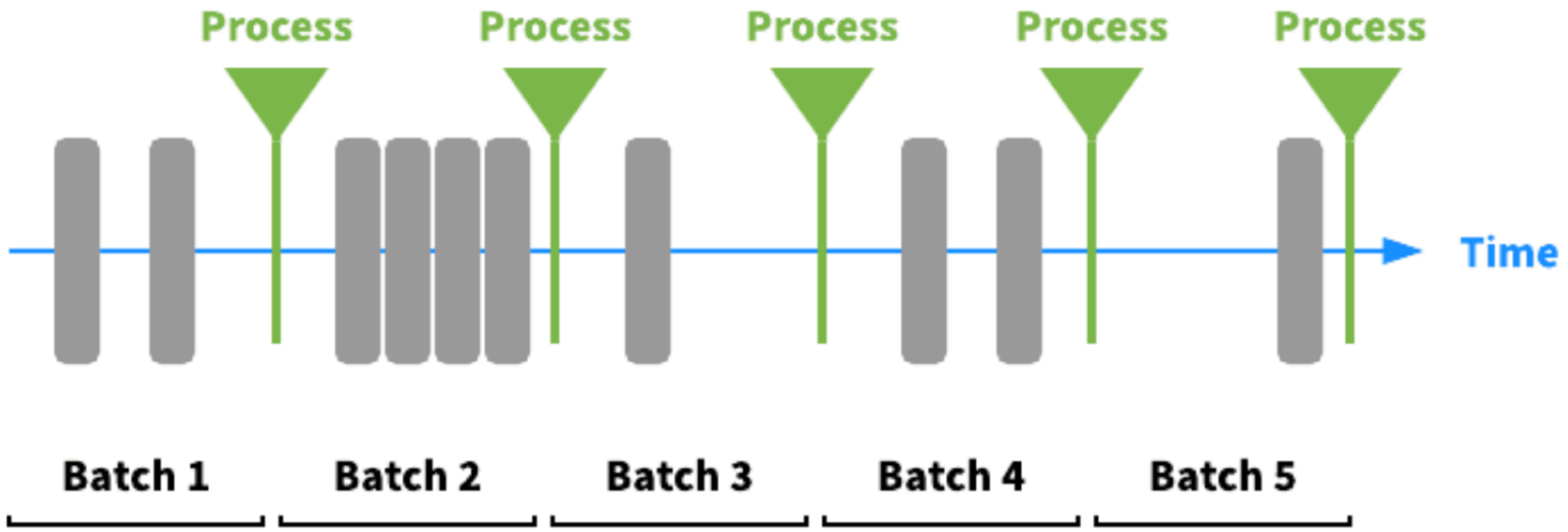
  Apache Spark

- Streaming – data is processed as it arrives

  Apache Kafka - KSQL

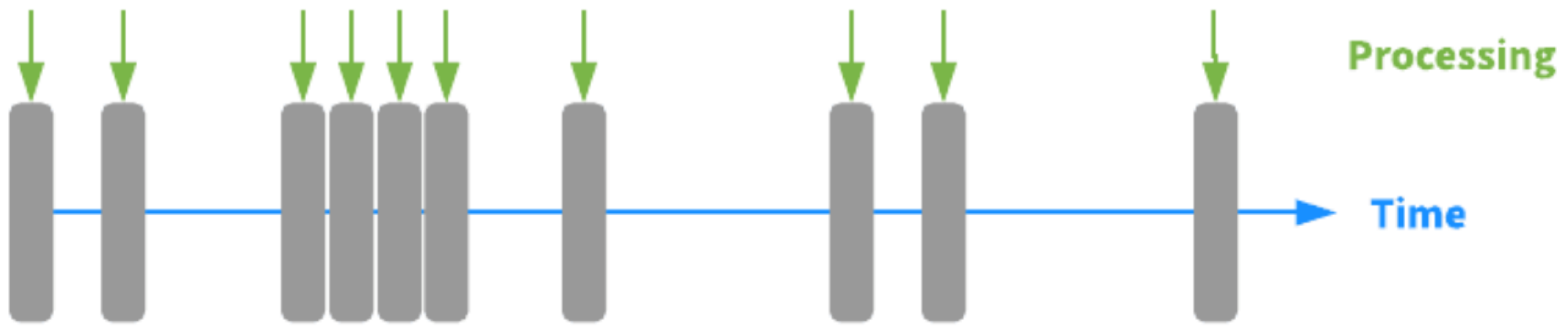
Batch and Microbatch

Data is processed on a time interval or when triggered by some condition. [source]



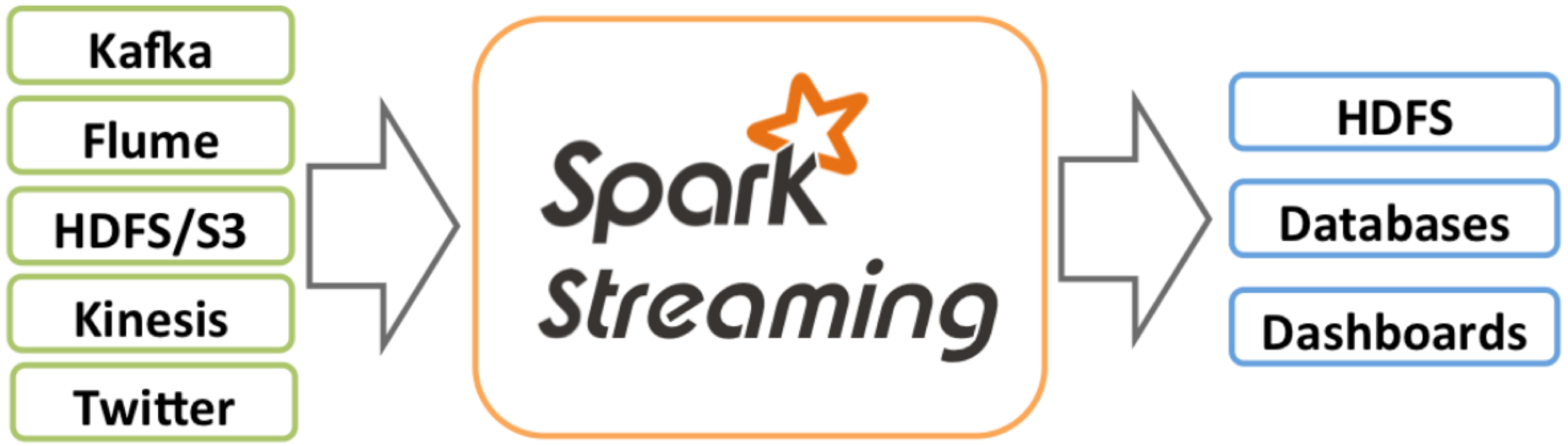
Streaming

- Data is processed as it arrives. [source]



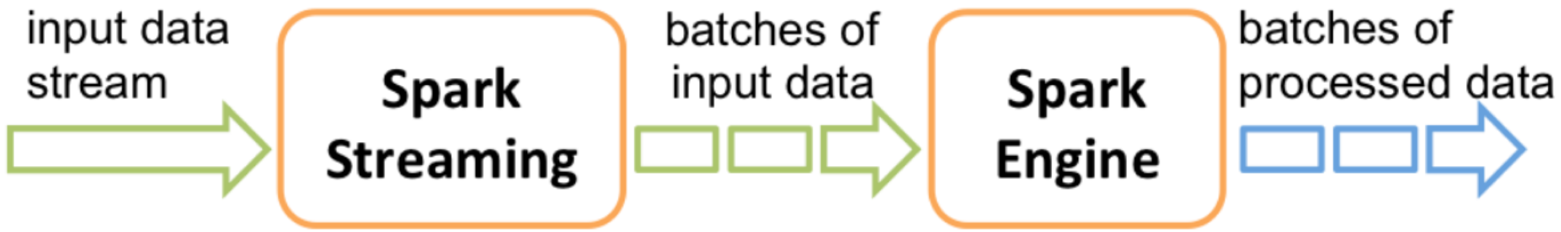
Spark Streaming

Can accept data from many sources and sink to several databases and file systems.



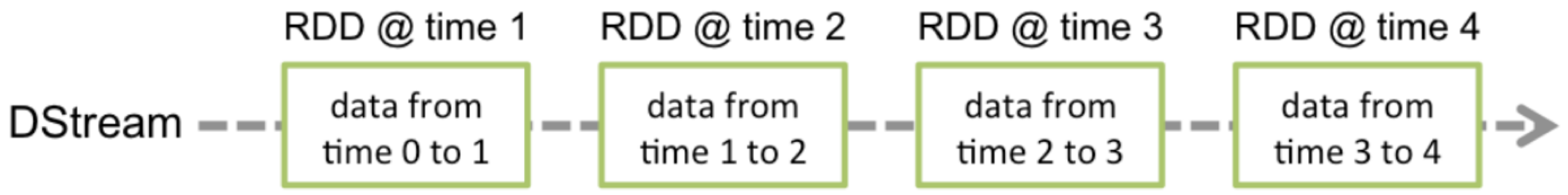
Microbatch approach

- The source is broken up into time divisions an processed.



Dstreams

- The fundamental abstraction is the Dstream a sequence of RDDs



Receivers

- Spark Streaming provides two categories of built-in streaming sources.
- *Basic sources*: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- *Advanced sources*: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

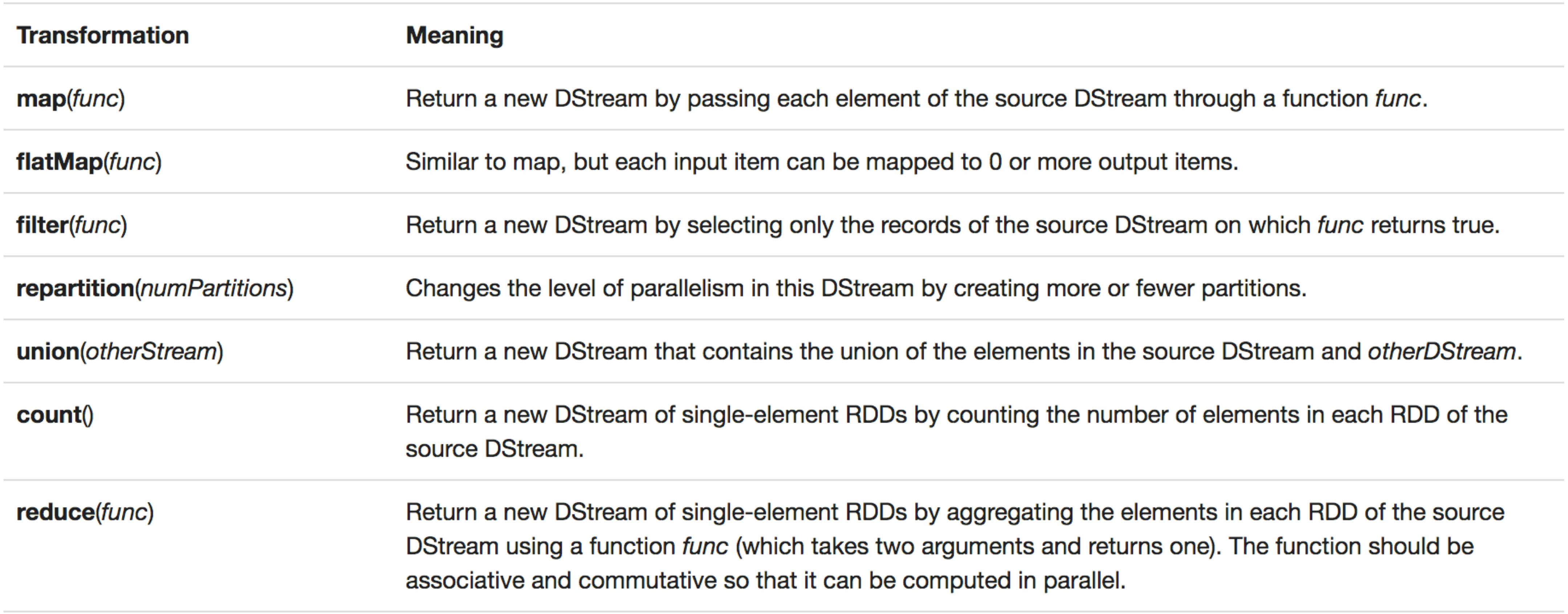
Operations on Dstreams

Operations on Dstreams are:

- Ingestion
- Transformation
- Output

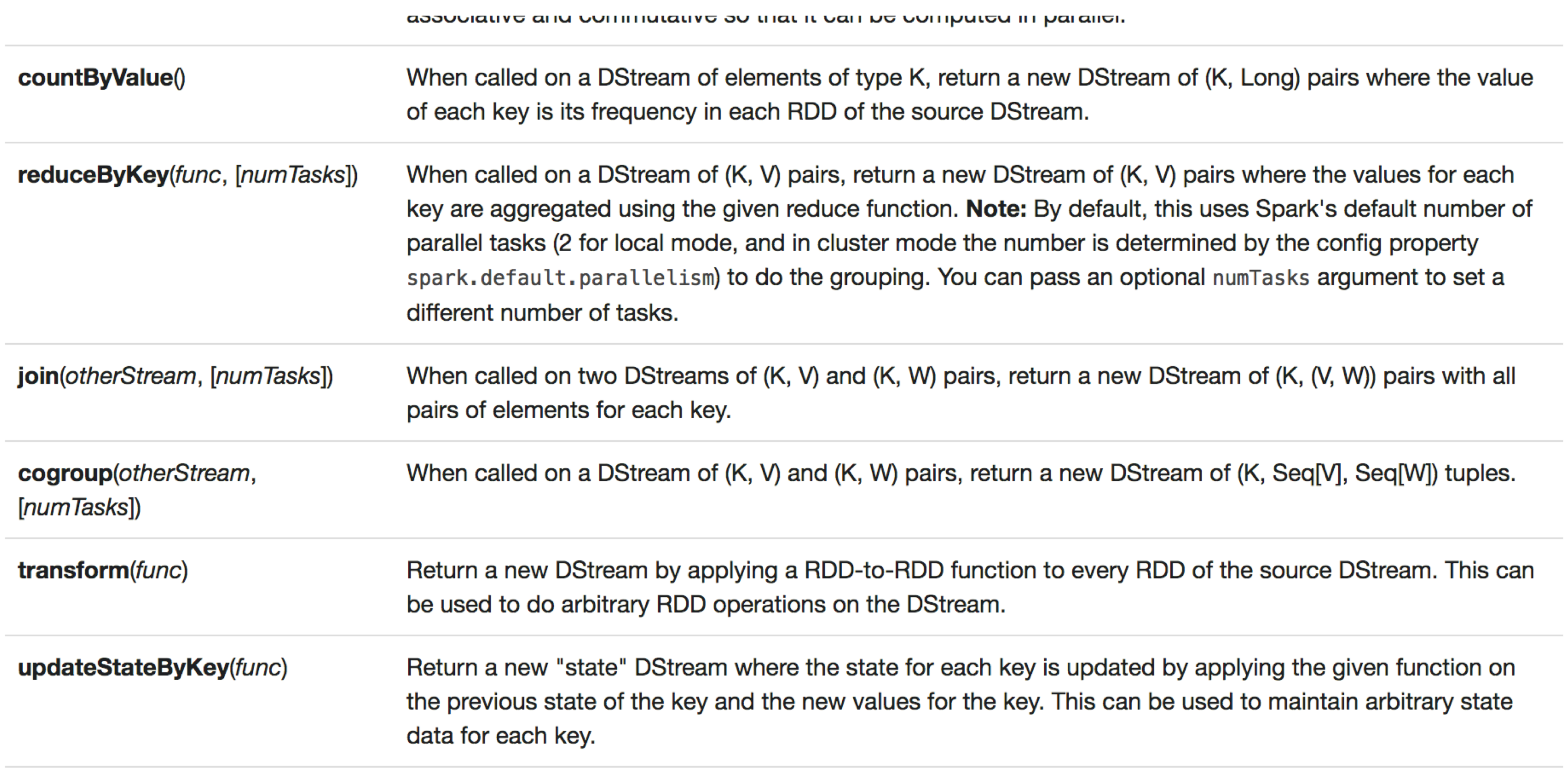
Rather than transformations and actions

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.

Transformations on DStreams



	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.
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.